

Aikata

Secure and Efficient Implementations of Post-Quantum Cryptographic Algorithms

DOCTORAL THESIS
to achieve the university degree of
Doktor der technischen Wissenschaften
submitted to
Graz University of Technology

Supervisor
Assoc. Prof. Sujoy Sinha Roy (TU Graz, Austria)

Assessors
Assoc. Prof. Sujoy Sinha Roy (TU Graz, Austria)
Prof. Ingrid Verbauwhede (KU Leuven, Belgium)
Prof. Tim Güneysu (Ruhr-Universität Bochum, Germany)

Institute of Information Security (ISEC)
Graz University of Technology
Austria, March 2026

Abstract

"Privacy protects us from abuses by those in power."
In *The Eternal Value of Privacy*, Bruce Schneier

This thesis investigates how to build hardware platforms that remain secure and efficient in a post-quantum era and how to enable practical computation on encrypted data. It emphasizes that "security and privacy are not a luxury, but an absolute necessity". The thesis is structured around two complementary areas within post-quantum cryptography: primitives for secure communication- Key Encapsulation Mechanisms and Digital Signature Algorithms, and techniques for privacy-preserving computation, focusing on Fully Homomorphic Encryption.

On the communication side, the thesis develops unified, compact architectures for NIST-selected lattice-based schemes that jointly support key encapsulation and digital signatures across various parameter settings. These architectures remain adaptable in response to evolving standards and parameter sets. Additionally, it proposes lightweight masking techniques for polynomial arithmetic that leverage small-secret structures, thus retaining the benefits of compact multipliers while ensuring side-channel protection.

On the computational side, the thesis introduces a cost-effective chiplet-based accelerator design methodology that significantly narrows the performance gap between homomorphic and plaintext computation. Building on the Hybrid Homomorphic Encryption paradigm, the work presents end-to-end hardware designs that integrate symmetric and homomorphic cryptography to minimize client latency and communication overhead. It also demonstrates that the resulting protocols are vulnerable to a single-fault differential attack that applies to several emerging primitives.

Finally, the thesis proposes client-side techniques and SIMD-packing strategies to further reduce homomorphic ciphertext communication and KeySwitch overheads in applications such as privacy-preserving machine learning and outsourced matrix multiplication.

Collectively, these contributions connect efficient implementations, potential attacks, and countermeasures, offering a comprehensive hardware-centric perspective on post-quantum communication and privacy-preserving computation.

Acknowledgements

"If I have seen further it is by standing on the shoulders of giants."
Isaac Newton

I started this research journey with nothing but curiosity. Back then, I didn't yet comprehend what research would demand of me- persistence, rigour, and the humility to accept academia with all its flaws. This thesis exists because of the following people who helped me through this learning.

Sujoy, thank you for taking a chance on me early on. When I decided to work on the hardware implementation of FHE, I had the enthusiasm of a beginner and almost none of the knowledge that would have made it sensible. I did not know hardware, and I did not know FHE either ☺. But you simply said, "Don't worry, if you're willing to learn, it's sufficient". You were right! In the beginning, when I often did not know what mattered, what did not, or even where to start, you became the *Manu to my naav*: steady, firm, and always pointing me in the right direction. You didn't just guide me; you also stood firmly behind my works and fought for them. The willingness to learn you encouraged has taken me a long way, giving me the confidence to explore beyond where I began and keep learning. You also put together the awesome 'CryptEng' group, Ahmet, Anisha, David, the Florians, Maciej, and Selim, who made this journey exciting.

A huge thank-you to Ahmet- an excellent postdoc and colleague, who helped me through the early implementation phase when I was still learning how to translate ideas into working hardware code. You introduced me to NTT design techniques, and just as memorably, to peeling tomatoes ☺.

Then came *Anisha- the work-soulmate* ♡, if such a thing even exists. You have a real gift for teaching, and at the same time you brought sharp research instincts, thoughtful questions, and a much-needed balance to the table. I'd be missing something if I didn't mention you, *David*. You brought a strange calmness to research, very delicious caramel custards, sweet wines (often too sweet ☺), and an entertaining take on global politics. Thank you *Florian Krieger* for the home-baked Brötchen, *Maciej* for the waffles, *Selim* for the amazing baklava, and *Florian Hirner* for disliking sweets so we always had more to enjoy ☺. It was truly a pleasure getting to know such exceptionally talented and promising researchers.

I extend gratitude to *Dhiman Saha, whose enthusiasm for cryptographic research is contagious.* You introduced me to the world of cryptographic research, and I hope I can spread your love for cryptography wherever I go.

To Samuel, Malik, Gaetan, Maciej, Florian, Nedzma, Anup, Shibam, Prasanna, Vid, Ratko, Patrik, and everyone I had the luck of working with as colleagues and coauthors- Thank You. You turned long days into shared momentum, celebrated the wins with me, and reminded me that research is rarely a solo effort. I carry forward not just the results of this thesis, but the way you work, the way you think, and the kindness you showed along the way.

This journey was supported by several promoters, who put me forward and opened doors that I didn't know were possible. *Stefan Mangard*, thank you for making my transition from a bachelor's to a PhD possible through a fast-track Master's. It was an unusual path, and you helped make it happen with your time, effort, and support. *Maria Eichlseder*, thank you for putting my name forward for presentations, and nudging me into opportunities that wouldn't have come to me on my own. It helped me build the confidence to show up more and eventually win several competitions. Thank you, *Emad Heydari Beni* and *Lode Hoste*, you offered me a much-needed industrial internship and championed both me and my work at every step. Thank you, *Lejla Batina*, for supporting and promoting me, especially when I was at an all-time low. Thank you, *Ursula Urwanisch*, for your help, support, and for making me feel at home at ISEC.

I was fortunate to receive several awards and recognitions during my studies, and I am deeply grateful to the respective awarding committees and organizations for these honours. These include the *Netidee Stipend* , the *ÖAW Critical Infrastructure Award 2025*, winning the first three *THERMA Challenges* on privacy-preserving machine learning, and the 1st Prize at the TU Graz Forum Technology and Society 2022 for my Masters thesis. I am also thankful for the *IACR student stipends*, which allowed me to attend many interesting conferences.

The journey in this PhD was quite literal, because it took me all the way from India to Austria. It felt impossible to acclimatize until I got to know you, *Michael*. Seeing the city through your eyes made it feel alive, like home. Thank you for the morning chai ♡, and for being my companion on this and so many other journeys around the world.

To Ahaan, thank you for being there. You were dreaming about research long before I even considered it; you encouraged me, and I happened to have reached this chapter first. You've been on the receiving end of all the absurd comments the reviewers threw at me, apologies ☺. On to whatever else the world has to offer.

Finally, my deepest gratitude to my parents- Hemlata and Shashindra, and *my brother- Pratyut, who stood by every decision I made and loved me unconditionally.* You unknowingly carried me through every setback and helped me see how small they are in the larger scheme of things. You believed in my potential, and here we are ♡.

Table of Contents

Abstract **iii**

Acknowledgements **v**

Table of Contents **vii**

List of Tables **xiii**

List of Figures **xvii**

1 Introduction **1**
1.1 Thesis Roadmap 8
1.2 Other Notable Contributions 11
1.3 Open-Source Hardware/Software Artifacts 13

2 Crypto Ground Zero **15**
2.1 Post-Quantum Cryptography 16
2.1.1 Lattice-based public key encryption schemes 16
2.1.2 Dilithium modules 18
2.1.3 Kyber modules 19
2.1.4 Saber modules 19
2.2 Fully Homomorphic Encryption 20
2.3 Hybrid Homomorphic Encryption 24
2.3.1 Pasta Scheme Design Overview 25
2.3.2 Invertible Matrix Generation Method 27
2.3.3 Masta and Rasta design overview 27
2.3.4 Hera design overview 27
2.3.5 Rubato design overview 28
2.4 Polynomial Multiplication Techniques 28
2.4.1 Schoolbook Polynomial Multiplication 28
2.4.2 Karatsuba Polynomial Multiplication 28
2.4.3 NTT-based Polynomial Multiplication 29
2.5 Implementation Attacks Overview 30
2.5.1 Correlation Power Analysis 30
2.5.2 Masking Countermeasure 31
2.5.3 Differential Fault Analysis of SE 31

3 Unified Cryptoprocessor for Post-Quantum Schemes 33
3.1 Introduction . . . 34
3.2 Architecture Design for Saber and Dilithium . . . 35
3.2.1 Polynomial multiplication, pseudo-random number generation, and hash computation . . . 36
3.2.2 Remaining scheme-specific building blocks . . . 38
3.2.3 NTT-based unified polynomial multiplier . . . 40
3.2.4 SHA3-256/512 and SHAKE-128/256 unit . . . 43
3.2.5 Remaining scheme-specific building blocks . . . 45
3.2.6 Parallel processing of instructions . . . 47
3.2.7 Organization of data memory . . . 48
3.3 The KaLi Unified Hardware Architecture . . . 49
3.3.1 The colossal Giant: Polynomial Arithmetic Unit . . . 50
3.3.2 Memory Arrangement . . . 54
3.3.3 Multi-clock domains for ASIC platforms . . . 55
3.3.4 The Giant and the Dwarves: Keccak-based SHA-SHAKE unit and the rejection samplers . . . 56
3.3.5 Optimizations for the Dwarves . . . 56
3.3.6 Instruction set cryptoprocessor . . . 59
3.3.7 Running the Giants and the Dwarves in parallel . . . 59
3.4 Results . . . 59
3.4.1 Results for Saber and Dilithium Cryptoprocessor . . . 59
3.4.2 Comparison with the existing results . . . 61
3.4.3 Results for the KaLi Cryptoprocessor . . . 63
3.4.4 Area and Performance Results . . . 65
3.4.5 Comparison with unified designs in literature . . . 66
3.4.6 Comparison with Dilithium-only designs in literature . . . 66
3.4.7 Comparison with Kyber-only designs in literature . . . 69
3.4.8 Application-Benchmarking and Impact . . . 69
3.5 Discussion and Conclusion . . . 70

4 The Kavach Masking Technique 73
4.1 Introduction . . . 74
4.2 Masking lattice-based PKE and Research gap . . . 76
4.3 Masking NTT in NTT-unfriendly rings . . . 77
4.4 Compact polynomial arithmetic using hybrid masking . . . 79
4.4.1 Hybrid masking . . . 79
4.4.2 Hybrid shares generation . . . 81
4.4.3 Hybrid masking of scalar linear functions . . . 83
4.4.4 Hybrid masking of vector linear functions . . . 85
4.4.5 Polynomial multiplication with hybrid masking . . . 86
4.5 Implementation of masked unit . . . 89
4.5.1 Implementation results . . . 91
4.6 Side-channel evaluation . . . 92
4.7 Conclusion and future work . . . 94

5 REED on the Path to FHE Realization 97
5.1 Introduction . . . 98
5.2 FHE Hardware design goals . . . 101
5.2.1 Monolithic vs Chiplet packaging . . . 101
5.3 FHE-tailored Multi-Chiplet Design . . . 103
5.3.1 REED 2.5D Architecture . . . 105
5.4 Architecture Design of One Chiplet . . . 107
5.4.1 The Hybrid NTT (Frankenstein's Approach) . . . 107
5.4.2 Multiply-Add-Subtract (MAS) and Automorphism (AUT) 110
5.4.3 PRNG-Based Partial Key-Switching Key Generation . . . 112
5.4.4 Programmable Instruction-Set Architecture . . . 112
5.4.5 REED Processing Unit (PU) . . . 113
5.4.6 Throughput Computation for KeySwitch . . . 114
5.4.7 Streamlined Prefetch for On-Chip Storage . . . 114
5.5 Techniques for exploiting Comm-Comp Parallelism . . . 115
5.5.1 Communication cost-analysis . . . 115
5.5.2 Limb-wise vs Coefficient-wise distribution . . . 116
5.5.3 Data Distribution across Multiple Chiplets . . . 116
5.5.4 Efficient Non-Blocking C2C Communication . . . 118
5.5.5 ModDown/Rescaling Task flow . . . 119
5.6 Implementation Results . . . 121
5.6.1 Power and Performance Modelling . . . 123
5.6.2 What to expect from higher-throughput configurations? . 124
5.6.3 Comparison with Related Works . . . 125
5.6.4 Higher Chiplet-Integration Study . . . 127
5.7 Application benchmarks . . . 128
5.7.1 Precision-loss Experimental Study . . . 130
5.8 Future Scope: Journey from 2.5D to 3D . . . 131
5.9 Conclusion . . . 132

6 Secure Cryptoprocessor for Hybrid HE 133
6.1 Introduction . . . 134
6.1.1 Research Gap . . . 134
6.1.2 Our Contribution . . . 137
6.2 Design Methodology . . . 138
6.2.1 XOF and Rejection Sampling Unit . . . 138
6.2.2 Operation Schedule . . . 138
6.2.3 Invertible Matrix Generation & Multiplication Unit Design 139
6.2.4 VectorAddition, Mix, and S-Box Unit Design . . . 140
6.2.5 Overall Architecture Design . . . 141
6.3 Results . . . 141
6.3.1 Evaluation Platforms and Area . . . 141
6.3.2 Performance Benchmarks. . . . 143
6.3.3 Comparisons with Prior Works . . . 144
6.4 Application Benchmark . . . 145
6.5 The Correlation Power Analysis of Hera . . . 146

6.5.1 Practical CPA on Hera: Experimental Setup and Results 146
6.6 Masking Technique . . . 148
6.6.1 Practical Implementation Strategies . . . 151
6.6.2 Test Vector Leakage Assessment . . . 152
6.6.3 First-Order Masking Across Related Cipher Schemes . . . 152
6.6.4 Discussion on Higher-order extension . . . 153
6.7 Conclusion and Future Scope . . . 154

7 SASTA: Nonce Respecting DFA on HHE 155
7.0.1 Research Gap . . . 157
7.0.2 The SASTA Threat Model . . . 157
7.0.3 Contributions . . . 159
7.1 Authenticated Transciphering (AT) . . . 161
7.2 The SASTA-DFA Fault Attack Strategy . . . 162
7.2.1 Attack Methodology . . . 162
7.2.2 How is SASTA-DFA different from standard DFA? . . . 163
7.2.3 Attack on HHE . . . 164
7.2.4 Extending SASTA-DFA to AT . . . 165
7.3 Leveraging SASTA-DFA: Key-Recovery Case Studies . . . 167
7.3.1 Cracking Pasta's Defenses . . . 167
7.3.2 Beyond Pasta: Extending the Attack to Masta . . . 169
7.3.3 Extending the Attack to Hera . . . 170
7.3.4 The Curious Case of Rubato . . . 172
7.3.5 Generalizing SASTA-DFA: SASTA on Rasta . . . 173
7.3.6 Exploiting AES via Known Approaches . . . 174
7.4 The Ambush: Fault Attack Demonstration . . . 174
7.4.1 Fault Injection Technique . . . 175
7.4.2 Attack and Key-recovery . . . 176
7.4.3 Identifying successful fault injection . . . 177
7.4.4 Empirical results for key-recovery . . . 177
7.5 Countermeasure analysis . . . 178
7.5.1 Countermeasure Performance Evaluation . . . 179
7.6 Discussion . . . 180
7.6.1 Impact on Design Decisions . . . 180
7.6.2 Implication on HHE Privacy Guarantees . . . 180
7.7 Future Scope . . . 181
7.8 Conclusion . . . 182

8 Privacy at Your Fingertips: Fast Client-Side FHE 183
8.1 Introduction . . . 184
8.1.1 Our Contributions . . . 186
8.2 Proposed High-level Techniques . . . 187
8.2.1 CoCo Cost Assessment . . . 187
8.2.2 Deflate Technique . . . 189
8.2.3 Boost Technique . . . 191
8.2.4 Boosted-Deflation Technique . . . 193

8.2.5 Security Analysis . . . 195
8.2.6 Comparison with Prior Techniques . . . 196
8.3 Client Enc/Decryption Framework . . . 198
8.3.1 Challenges . . . 198
8.3.2 Framework Evaluation . . . 198
8.4 Comparison with Related Works . . . 200
8.5 Application Benchmarking . . . 203
8.5.1 Logistic Regression Training . . . 203
8.6 Discussion . . . 205

9 Secure Matrix Multiplication with Homomorphic Encryption 207
9.1 Introduction . . . 208
9.1.1 Prior Works. . . . 208
9.1.2 Contributions. . . . 210
9.1.3 Roadmap . . . 210
9.2 Prior Matrix Multiplication Techniques . . . 211
9.3 Proposed Matrix Multiplication Technique . . . 214
9.3.1 Technique for d^2 packing . . . 215
9.3.2 Technique for $2 \cdot d^2$ packing. . . . 216
9.3.3 Generalization to arbitrarily high packing complexity ($> d^2$) 218
9.4 Generalization to Rectangular Matrices . . . 221
9.4.1 Padding based technique . . . 221
9.4.2 Common Divide-and-Conquer Technique . . . 222
9.5 Experimental Evaluation . . . 223
9.6 Discussion Packed Multiplications. . . . 224
9.7 Conclusion . . . 225

10 Conclusion 227
10.1 Contributions at a glance . . . 228
10.2 Open Problems and Future Directions . . . 230
10.2.1 PQ-secure communication beyond KaLi, Kavach . . . 230
10.2.2 From Protocols to Applications: The Last Mile . . . 231

11 Epilogue 233

A Appendix 235
A.1 Task Distribution for KeySwitch . . . 235
A.1.1 Throughput Computation of KeySwitch . . . 238
A.2 The IND-CPA$^{\mathrm{D}}$ Security Overview . . . 240
A.2.1 IND-CPA$^{\mathrm{D}}$ Security for HHE . . . 241
A.2.2 IND-CPA$^{\mathrm{D}}$ Security for HHE under Faults . . . 242
A.2.3 IND-KPA$^{\mathrm{D}}$ Security for AT under Faults . . . 243
A.3 Keccak Hash/PRNG Function . . . 243
A.4 Hera FIPs study . . . 243
A.5 Server-side attack surface . . . 244
A.6 Comparison Table . . . 245

A.7 Client-server data exchange 246
A.8 Baby-step giant-step method 246
A.9 Comparison of Boosted-Deflation with HHE 247
A.10 Algorithmic description of the proposed technique and Security Aspects 248
A.11 Overview of Framework Components 249
A.12 Operation-wise Data Flow 252

B Publications **253**
Main Publications 253
Further Publications 254

Bibliography **257**

C Affidavit **299**

List of Tables

2.1 CKKS Parameters 21
2.2 A simplified algorithmic description of the HHE protocol. Here, i refers to the i-th iteration of the HHE protocol. 25

3.1 Distribution parameters and error probabilities 38
3.2 Instructions where an instruction from the first column can be run in parallel with an instruction from the second column. 48
3.3 Area of the cryptoprocessor on the Zynq Ultrascale+ ZCU102 FPGA platform. All security levels of Dilithium and Saber are supported. 60
3.4 Performance results for Saber-KEM and Dilithium in FPGA . . 61
3.5 Comparison Table for Dilithium-3 63
3.6 Comparison Table for Saber-KEM 64
3.7 Area of KaLi on the Zynq Ultrascale+ ZCU102 FPGA platform. All security levels of Dilithium and Kyber are supported. 65
3.8 Performance results for Dilithium and Kyber-KEM in FPGA . . 66
3.9 Comparison Table for Dilithium-3 FPGA Implementations 67
3.10 Comparison Table for Dilithium-3 ASIC Implementations 67
3.11 Comparison Table for Kyber-1024 FPGA Implementations 68
3.12 Comparison Table for Kyber-1024 ASIC Implementations 68

4.1 Implementation results of the design on Xilinx Artix-7 xc7a100tftg256-3 for the parameters $n = 256, K = 2^4,$ and $q = 2^{13}$. 92

5.1 Comparison of the naive techniques and our technique for the KeySwitch operation, including ModUp, Key Multiplication, and ModDown. The operation count is the maximum reported by any chiplet, which will determine the overall throughput of the design. || Implies Operation is done in parallel with other computations. 105
5.2 Performance micro-benchmarks for 28nm and 7nm. 121
5.3 Total area consumption of 4-chiplet REED 2.5D for different configurations on 28nm and 7nm. 122
5.4 The table compares memory (number of coefficients to store) and multiplier units for different N_1 and N_2 values with and without twiddle factor generation (TFG). 125
5.5 Comparison of REED 2.5D with state-of-the-art 126

5.6 Results for 1024×64 configuration on 28nm technology with parameters ($N/L/dnum = 2^{16}/30/31$). 128
5.7 Application benchmark and the speedup achieved by REED 2.5D. The CPU speed is reported on a 24-core, 2×Intel Xeon CPU X5690 @ 3.47GHz with 192GB DDR3 RAM. 129

6.1 The implementation results of Pasta-3/4 on the Artix-7 board targeting 75MHz clock frequency. 142
6.2 The performance results for Pasta-3/4 encryption of one block, along with the results for CPU reported in Pasta. 143
6.3 Comparisons of Pasta-4 Performance, Area with Prior Works. . 144
6.4 Comparison of not masked and masked Hera implementations in terms of average clock cycles and stack usage. 152

7.1 Comparison of SASTA-DFA with prior works. The average time for key recovery on a 2GHz clock 4GB RAM processor is reported. † stands for results from [AM11b]. 159
7.2 A simplified algorithmic description of the Authenticated Transciphering protocol [Aha+23; BBS21]. 161
7.3 Faulty computation flow during the HHE protocol. For brevity, we omit the Initialization and Function Evaluation parts of the protocol. 165
7.4 Faulty computation flow during Authenticated Transciphering. . 166
7.5 The Security bounds (parameters for 128-bit security) before and after SASTA-DFA considering the linearization attack. Considers the case where matrices are not invertible for Masta, and Rasta, or the key-recovery is incomplete due to insufficient fault diffusion (Hera) or truncation (Rubato). 171

8.1 A description of the data and computation flow after utilizing the proposed techniques. (algorithm in Section A.10) 194
8.2 Comparison on Software (CPU), Embedded (Micro-Controllers), and Hardware (FPGA and ASIC) platforms. 199
8.3 Component-wise FPGA Area Utilization and Performance. Design details are provided in Section A.11. 200

9.1 Comparison with secure d-dimensional matrix multiplication techniques. The division of works is done based on the three types of works discussed in Section 9.1.1. # Key-Switches = # ct-ct Mult + # Rotations. 209
9.2 Complexity of the proposed secure d-dimensional matrix multiplications 218
9.3 Runtime evaluation of privacy-preserving matrix multiplication. . 224

A.1 More comprehensive comparison of SASTA-DFA with prior works. The average time is reported for key-recovery on a 2GHz clock and 4GB RAM processor. † stands for results reported in [AM11b]. . 245
A.2 An algorithmic description of the conventional client and server computations. 246

List of Figures

1.1 TLS 1.3 flow, highlighting the public-key dominated handshake (authentication and key establishment) followed by symmetrically encrypted exchange [Res18]. 2
1.2 Classical vs. quantum adversaries: PQC replaces RSA/ECC, while AES and hash functions mostly scale up. 4
1.3 Conventional versus Privacy-preserving Computation using FHE. 6
1.4 Overview of this thesis in the asymmetric post-quantum setting, grouped into three categories: (i) Client/Server covers acceleration of post-quantum key encapsulation and digital signature schemes, and side channel protections; (ii) Client-only includes embedded implementations enabling homomorphic encryption and decryption, along with associated attacks and countermeasures; and (iii) Server-only comprises server side acceleration of homomorphic encryption and an optimized application. ⓧ denotes the chapter where these works are described. 7

2.1 Depiction of the communication protocol using KEM and DSA. 16
2.2 Depiction of the FHE flow. 21
2.3 A high-level view of the Bootstrapping procedure used to refresh the ciphertext's multiplicative depth. 22
2.4 The client symmetrically encrypts the plaintext (SE ENC) and sends it to the server. The server converts it to FHE ciphertext, evaluates the function, and returns the result. The client then decrypts (FHE DEC) the final ciphertext homomorphically. . . 24
2.5 The Pasta-t permutation (π) takes as input the key (K), nonce (n), and counter (ctr), and generates the truncated result- key stream (KS). 26
2.6 The r-round Masta and a general idea of Hera and Rubato round functions. 27
2.7 Hierarchical (4-step) NTT datapath for $n = 16$. DIT stands for Decimation in Time, and DIF stands for Decimation in Frequency. 30

3.1 Distribution of coefficients after multiplication between secret and public polynomials with coefficients in the range $[-\frac{\mu}{2}, \frac{\mu}{2}]$ and $[0, q-1]$. 39

3.2 The high-level design of the cryptoprocessor with the Saber, Dilithium and common modules are in green, blue and red, respectively. 40
3.3 Internal architecture of the unified butterfly unit. The dotted routing lines for signals a and b are used to represent long interconnects 41
3.4 Unified modular reduction unit 41
3.5 High-level architecture of polynomial arithmetic unit 42
3.6 Coefficients in memory for 3 iterations of NTT for $n = 16$ . . . 42
3.7 Example of bookkeeping with 18 remaining bits 45
3.8 High level architecture of Keccak that supports all the modes of Keccak as well as samplers 45
3.9 Operation scheduling of LightSaber PKE encryption 47
3.10 High-level architecture of `KaLi` 50
3.11 Flexible yet compact integer multiplier. The red/black lines show the control/data path. The DSP units used are highlighted in purple. 51
3.12 Unified modular reduction unit for Dilithium and Kyber primes. 53
3.13 Compact butterfly unit (BFU) with flexibility for both Dilithium and Kyber. The red-and-blue/black lines show the control/data path. 54
3.14 Unified adder and subtractor for the butterfly unit. The red lines show control signals, and the black lines show data movement. . 54
3.15 Butterfly feedback unit for Kyber's polynomial multiplication. . 55
3.16 Timeline showing the unified butterfly unit processing Dilithium and Kyber coefficients. 56
3.17 Storage of coefficients during Kyber's NTT for 16-coefficient polynomial 57
3.18 The data read-and-write handshake between memory and logic unit 57
3.19 Architecture of the Compress/Decompress unit. The red lines show control signals, and the black lines show data movement. . 58
3.20 The figure presents a graphical representation of time-consumption and resource comparisons between our work and existing works. The lower the time-consumption and resource consumption, the better. With both schemes included, our cryptoprocessor still consumes reasonable resources and provides optimal performance. 64

4.1 Architecture diagram illustrating the `AOverflowBit` unit 89
4.2 Polynomial multiplier architecture for the hybrid masking scheme. 90
4.3 Secure multiply-and-accumulate unit for computing $acc[i + j] = acc[i + j] + c[j] \cdot a[i]$ using Boolean shares $c_0[j], c_1[j]$ of $c[j]$. Registers for glitches are not shown. 91

4.4 MAC (multiply and accumulate) unit architecture diagram. It computes multiplication for both Boolean as well as arithmetic shares. 91
4.5 Raw measurement data of `AOverflowBit` and polynomial multiplication of the hybrid shares. **Top:** t-test figures with mask ON for 100,000 traces; **Bottom:** t-test figures with mask OFF for 200 traces. 93

5.1 KeySwitch operation for $l = 2$, where I, F, and K represent INTT, NTT, and key multiplication operations using MAS, respectively. 102
5.2 This depicts the different techniques, data, and task distribution for automorphism, followed by KeySwitch. $l = 2, 1$ for (a), (c) and 4 for (b) 103
5.3 Side and top view of proposed four chiplet-based REED 2.5D. . 106
5.4 Novel routing-friendly Hybrid NTT/INTT design flow for $N = N_1 \times N_2$. 107
5.5 The proposed novel Hybrid NTT/INTT design flow with Memory access for (a) NTT and (b) INTT with $N_1 = 4$, $N_2 = 4$, and $N = 16$. The butterflies represent the Gentleman-Sande butterfly [Sco17a] operation employed in our design. 108
5.6 The set of triadic MAS units. 111
5.7 An example of a `shuffle_tree_`$N_2 \times N_2$ workflow. Every stage has sufficient registers to hold N_2 coefficients. 112
5.8 The REED-PU design. Every data communication (memory to building blocks and off-chip to on-chip) here has a bandwidth of $N_2 w$ bits/clock-cycle. 113
5.9 Timeline of parallel and pipelined operation flow. 115
5.10 Non-blocking ring-based communication for four REED chiplets when $l = 6$. The blocks between executions represent the long communication window to make up for slow inter-chiplet (C2C) communication. 118
5.11 Ring-based communication for four REED chiplets when $l = 15 \rightarrow 16$ (after ModUp) for the (a) ModDown operation and (b) ModDown+Rescale operations. 120
5.12 The complete architecture diagram of 4-chiplet REED 2.5D for 1024×64 configuration. The multiple small black blocks denote I/O interconnects along the edges. 121
5.13 Area increase with rising throughput configurations [GPG23]. . 124
5.14 Relative a) yield of existing monolithic designs versus the proposed 7nm chiplet-based architecture [Ma+22], b) development cost (including Interposer cost) [GPG23; Mer+23; MUSb], and c) cost of SiP development (cost/yield). RD refers to our work REED 2.5D. 127
5.15 The complete architecture diagram of 6-chiplet REED 2.5D for 1024×64 configuration. 128

5.16 Relative metrics comparison for the HELR [Han+19] application with batch sizes 256 and 1024. Under these metrics, the lower the value, the better. 129
5.17 A DNN for MNIST [LC10] with two hidden and one output layers. 130
5.18 Accuracy plot of different word sizes for the DNN. The lines are smoothened and the red dotted zig-zag line resembles the original form. 131
5.19 The side and top view of futuristic RE3D has four REED 3DIC chiplets. 131

6.1 Workflow of HHE schemes is illustrated here. Symmetrically encrypted ciphertexts are sent to the server. The server performs a 'homomorphic HHE decryption' and obtains a homomorphically encrypted ciphertext. 134
6.2 The schedule for one round of the Pasta-π permutation. 139
6.3 The Data Generation and Adder-Tree units utilized in Matrix Generation and Multiplication. The log operation has base 2. . 139
6.4 The Invertible Matrix Generation and Multiplication Units. . . 140
6.5 The overall architecture of the hardware design for Pasta. . . . 141
6.6 Module-wise area utilization. The first pie chart shows the distribution for FPGA and the second for ASIC platforms. Rem. implies Remaining Area. 142
6.7 Frames transferred per second for Maximum (112.5 MBps, left) and Minimum (12.5 MBps, right) 5G available bandwidth. The graphs are on a logarithmic scale. TW refers to this work. . . . 145
6.8 Representative power trace of Hera (top) and correlation for the correct key guess (bottom), highlighting the leakage peak at the point of interest. 146
6.9 Correlation values for all key guesses at the point of interest, with the correct key clearly distinguished by the peak at index 55,444. 147
6.10 CPA results for all 16 key elements. Curves illustrate the correlation across hypotheses, with dots indicating the peak values that reveal the correct elements. 148
6.11 Success rate of the CPA attack as a function of the number of traces for 100 random keys, converging to full recovery after $\approx$40 traces. 148
6.12 Dataflow of first-order arithmetic masking in Hera. Each sensitive intermediate x is decomposed into two shares (x_0, x_1), which are processed independently throughout the cipher. Linear operations (e.g., addition, modular reduction) are applied share-wise, while nonlinear operations (S-box) are adapted to operate on masked inputs. 149

6.13 HERA power traces and corresponding t-test evaluations. The top plot shows a HERA encryption trace for one round, with vertical dashed lines marking major computation phases (AddRoundKey, MatMul, SBox, AddRoundKey). The middle plot presents TVLA results with faulty randomness, where the test statistic exceeds the ś4.5 threshold lines in red, revealing clear leakage. The bottom plot shows the same TVLA evaluation with proper randomness for 10,000 traces, where the values remain within the thresholds, indicating the absence of significant leakage. 153
6.14 Leakage assessment of the Feistel S-box in PASTA. 154

7.1 Workflow of HHE is illustrated here. The homomorphically encrypted key ($K_{\mathtt{FHE}}$) is communicated only once. After this, multiple small ciphertexts ($c_{i_{\mathtt{SE}}}$) encrypted using SE are sent along with nonce. The server performs a 'homomorphic SE decryption circuit evaluation' (HHD) and obtains a homomorphically encrypted ciphertext $c_{i_{\mathtt{FHE}}}$. Here $\mathtt{E}_\pi$ refers to any SE permutation (e.g., AES), and i refers to i-th iteration of the HHE protocol. . 156
7.2 The IND-CPA$^{\mathrm{D}}$ threat model [LM21] and its adaptation to the SASTA-DFA model for HHE and AT, which extends it by incorporating fault injection capability. 158
7.3 Utilizing SASTA-DFA for differential fault analysis of HHE tailored SE schemes. 163
7.4 The diagram illustrates the difference between the standard DFA model (a) and the proposed SASTA-DFA model (c), showcasing how the classical setting relies on nonce-reuse, unlike SASTA-DFA. It also presents a case for standard DFA with server interaction (b), indicating how key recovery is infeasible as per specification, $n_i \neq n_g$. 164
7.5 Exploiting SASTA-DFA for Authenticated Transciphering using AES-GCM. 166
7.6 The attack strategy and FIP for PASTA-4 encryption ($KS = \mathtt{E}_\pi(K, n_i)$). 168
7.7 The fault injection points for inducing a fault in HERA or RUBATO permutation before final add-round-key. 170
7.8 The sample fault injection attack positions targeted in our work. The first position is the beginning of KECCAK permutation, and the second is where the results of permutation are written to the final buffer. 175
7.9 The first plot illustrates fault injection during the last HERA ARK operation. The second plot presents an enlarged view of the normal and faulty computations following the clock glitch. Note that these traces have been extracted from the original plot, and the x-axis is relative. 176

7.10 A comparison of latency and storage for different PASTA-4 implementations (with or without countermeasures). The three highlighted implementations are effective against SASTA-DFA. The graph is on a logarithmic scale 179

8.1 FHE-based Web3 Smart Contracts. The client sends encrypted data to the DApp. It remains secure during computation, and the result is returned to the client for decryption. 184
8.2 Illustration of how this work balances the two extremes explored in prior client-side performance studies. The y-axis is logarithmically scaled. 185
8.3 The simplified encryption process after applying the Deflate technique. The idea is to offload the encoding/decoding step to the server. 189
8.4 A high-level depiction of the key idea behind the proposed Boost technique. The 'OLD' and 'NEW' keywords show the ciphertext and data flow for the conventional and our proposed techniques, respectively. 191
8.5 The number of bootstrapping operations performed on the server side increases by a constant 2 for high-depth computations ($> L_{eff}$) and 1 for lower-depth functions. For the combined Boosted-Deflation technique, the offset is lowered, allowing more computations per bootstrapping. 193
8.6 The operation flow following the proposed Boosted-Deflation technique. The client performs zero-depth encryption without encoding. The server performs a small bootstrapping step, where the initial `SlotToCoeff` operation is omitted, resulting in a cost of only 72% of the original bootstrapping. This helps save computational depth after bootstrapping. The server also performs the homomorphic decode (`SlotToCoeff`), so the client only has to compute decryption. 193
8.7 A comparison with works utilizing HHE to reduce CoCo. BS refers to Bootstrapping, for $L = 30$. Our work offloads at most two bootstrapping depending on the function. 196
8.8 The proposed high-level architecture of the client device performing homomorphic en/decryption. 200
8.9 Comparison with prior works. The bottom plots apply Boost techniques on prior works. 202
8.10 Frames transferred per second for Maximum (112.5 MBps) and Minimum (12.5 MBps) 5G available bandwidth (BW). The graphs are on a logarithmic scale. 204
8.11 Time required for client and server computation, and communication for the Logistic Regression Training 205

9.1 Matrix Column and Row duplication for A and B, respectively. 215

9.2 Matrix duplication and alignment done in one step using rotation. First, Matrix A is rotated and added to the original A. Then, a mask is multiplied to remove the unwanted elements for the upcoming column-wise duplication step. 217

9.3 The adder tree depicting final result accumulation within a ciphertext. Every stage with t elements performs an addition (in SIMD) and results in $\frac{t}{2}$ resultant elements, which are then added in the next stage. This process continues until all the elements are accumulated. 219

9.4 Graph comparing the rotation count reduction for d^3 packing. . 221

9.5 The $A_{2\times4}$ matrix duplication transformation considering padding. The rectangular matrix undergoes two rotated accumulations to result in the middle ciphertext. This ciphertext behaves like a square matrix with zero padding. The grey elements are not needed and are removed via multiplication with the mask. After this step, column-wise duplication follows. 221

9.6 Rectangular matrices division for the divide-and-conquer technique. 222

9.7 Graph comparing the rotation count for $d^3(d = 32)$ packing available peer ciphertext with increasing distinct simultaneous matrix multiplication requirement r. 225

10.1 Kavach masking for higher orders (d=4). 231

A.1 The Data and Task distribution when limbs are distributed digit-wise for KeySwitch computation. The INTT, NTT, and BaseConversion mentioned here refer to computations required per digit. In the first case, Data Exchange is done in parallel with BaseConversion, and in the second case, it is done in parallel with KeyMult. This does not produce a full decoupling effect when C2C communication is slower. 236

A.2 The Data and Task distribution when limbs are distributed alternately for KeySwitch computation. The Data Exchange is done in parallel with INTT, BaseConversion, and NTT+KeyMult of previous digits. 237

A.3 KeySwitch Flow showing computation throughput and communication overhead for ModUp, KeyMult and ModDown for $dnum = 3$. 238

A.4 The IND-CPA$^{\mathrm{D}}$ threat model for FHE as introduced in [LM21]. 240

A.5 The IND-CPA$^{\mathrm{D}}$ threat model [LM21] for HHE. 241

A.6 The adaptation of IND-CPA$^{\mathrm{D}}$ threat model [LM21] for HHE to the SASTA model, which extends it by incorporating fault injection capability. 242

A.7 This figure shows that more KECCAK permutations for random vector generation of HERA make it easier to target a specific vector without affecting others, leading to a higher reduction in the keyspace. 244
A.8 The proposed timeline of the encryption-decryption routines such that maximum residue polynomial storage required at any instance is three. Here, M is the latency of one NTT (or three parallel NTTs). 248
A.9 The NTT architecture which utilizes four butterfly units. 249
A.10 The shift-add based Modular Reduction unit. 250
A.11 The data flow during encryption and decryption routines in the proposed high-level architecture. 252

1

Introduction

Breu

This thesis began five months before NIST announced the first round of schemes selected for post-quantum standardization in July 2022, and four years after the proposal of an approximate arithmetic-supporting Fully Homomorphic Encryption scheme (CKKS). In hindsight, it was a fascinating time to start a PhD, as new directions in public-key cryptography were rapidly emerging- ranging from post-quantum secure communication primitives (KEM and DSA) to post-quantum techniques for privacy-preserving computation, such as FHE. Before exploring the challenges and technical contributions, this chapter introduces the motivations behind these cryptographic building blocks and explains why bridging our real world with the cryptographic domain is essential for security and privacy.

In practice, this bridge is often implemented in *hardware*, where cryptographic algorithms must meet strict constraints on latency, energy, and cost, and where design choices directly shape the attack surface, especially against side-channel and fault adversaries. Hardware, therefore, plays a central role in both themes of this thesis: for PQC, it enables deployable and agile KEM/DSA implementations under tight area and energy budgets while addressing implementation-level threats; for encrypted computation, it helps close the large performance gap of FHE by accelerating its dominant arithmetic kernels and reducing end-to-end latency.

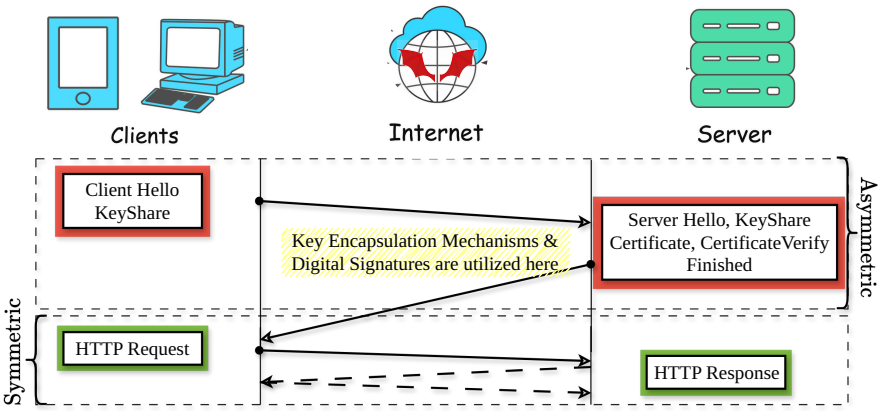


Figure 1.1: TLS 1.3 flow, highlighting the public-key dominated handshake (authentication and key establishment) followed by symmetrically encrypted exchange [Res18].

Cryptography is the science of protecting data by transforming it into a format that appears random, preventing unauthorized access. Over the past few decades, the landscape of data security has undergone significant changes. We have seen a major shift from storing and printing information in physical files to digitizing all forms of data. With this digital transformation came automation and large-scale connectivity, allowing the internet to support communication, commerce, and *critical infrastructure*.

This vast interconnectedness introduced new challenges, including the need to protect sensitive data from unauthorized access, cyber threats, and potential breaches. The growing complexity of digital systems demanded more sophisticated methods of ensuring secure communication and data integrity. As a result, advanced cryptographic techniques have been developed to meet the demands of the modern world. Simultaneously, increasingly powerful adversaries and new attack methods continually raise the standard for what is considered a secure system. Thus, *the need for secure cryptographic techniques remains a constant.*

In practice, security protocols utilize two complementary types of cryptography, as illustrated in Figure 1.1. *Symmetric cryptography* efficiently protects large amounts of data once both parties have a shared secret key. However, establishing and managing these keys on an internet scale is inherently challenging. On the other hand, *public-key (asymmetric) cryptography* solves this issue by allowing parties without a prior relationship to authenticate each other and agree on shared secrets over an untrusted network. Each participant creates a key pair: a *private key*, which must be kept confidential, and a corresponding *public key*, which can be widely distributed.

A central public-key primitive is the *digital signature algorithm* (DSA). To sign a message (or, more commonly, a cryptographic hash of the message), the

sender uses their private key to produce a signature that is computationally impossible to forge. Anyone with access to the corresponding public key can verify the signature's validity, ensuring both integrity (the message has not been altered) and authenticity (the holder of the private key created it). In many cases, this process also supports non-repudiation [Nat23]. In protocols such as TLS [Bür+20; GGF22; Res18] and software update systems, digital signatures authenticate servers, code, and certificates, enabling clients to detect tampering.

Public-key cryptography is also essential for establishing the symmetric keys that ultimately secure application data. A common contemporary abstraction used is the *Key Encapsulation Mechanism* (KEM). During a KEM-based handshake, one party uses the other partys public key to generate (or encapsulate) a fresh shared secret along with a ciphertext. This ciphertext can be transmitted over the network, and only the holder of the corresponding private key can decapsulate it to retrieve the shared secret. This shared secret is then processed through a key-derivation function to produce symmetric session keys, after which the protocol employs efficient symmetric encryption and authentication for the data stream.

In real-world applications, public keys, signatures, and key-establishment mechanisms are typically managed through a *public-key infrastructure*. In this system, certificates issued by trusted certification authorities link a public key to a specific identity. This linkage allows software such as web browsers, operating systems, and update mechanisms to recognize a presented public key as belonging to a particular server or publisher, enabling them to verify signatures and establish session keys with the intended counterparty.

However, the security of this trust framework ultimately relies on the strength of the underlying public-key algorithms and the parameters used to implement them. Historically, widely deployed public-key constructions have been based on either integer factorization (e.g., RSA) or discrete logarithm problems in finite fields and elliptic curves (e.g., ECDH for key establishment and ECDSA for signatures) [Mor+16; Nat23], as shown in Figure 1.2. While these methods remain foundational, real-world security has often been compromised by *legacy parameter choices and deprecated building blocks*.

For instance, export-era TLS configurations allowed "downgraded" Diffie-Hellman handshakes using 512-bit groups (DHE_EXPORT), which have been shown to be practically breakable, enabling man-in-the-middle attacks such as Logjam. Even 1024-bit finite-field groups have been deemed vulnerable to well-resourced adversaries, leading to a push toward substantially larger groups or adopting elliptic-curve variants.

These considerations highlight a practical lesson from deployment: cryptographic security depends not only on a well-defined hardness assumption but also on the careful selection of parameters, meticulous implementation, and, crucially, the ability to update algorithms and parameters. As our understanding improves, hardware speeds increase, or new implementation vulnerabilities are identified, systems must adapt without the need to redesign the entire protocol stack. This adaptability is often referred to as *cryptographic agility*.

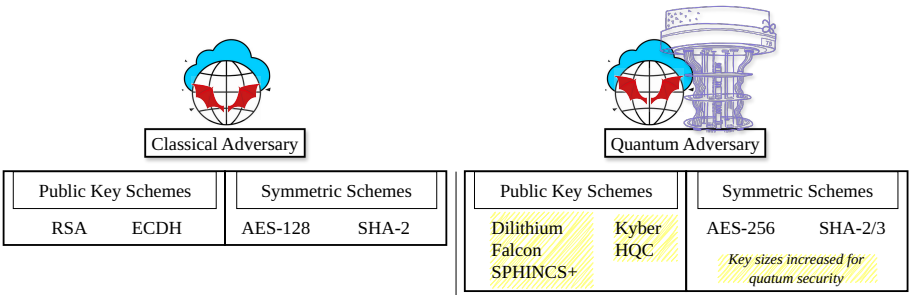


Figure 1.2: Classical vs. quantum adversaries: PQC replaces RSA/ECC, while AES and hash functions mostly scale up.

The need for agility becomes particularly evident when the threat model evolves, rather than simply 'scaling up'. A significant factor driving this change is the rise of large-scale quantum computing. Unlike the gradual, incremental advancements typical of classical computing, quantum algorithms can drastically alter the difficulty of the fundamental mathematical problems on which many current public-key systems are based. For instance, Shor's [Sho97] algorithm provides polynomial-time quantum procedures for integer factorization and discrete logarithms, which could undermine RSA, (EC)DH, and (EC)DSA if executed on a sufficiently powerful quantum computer. This urgency necessitates a shift to **Post-Quantum Cryptography** (PQC), which aims to develop public-key primitives that remain secure against both classical and quantum adversaries, as shown in Figure 1.2.

In response to this shift, the American National Institute of Standards and Technology (NIST) initiated a standardization process for PQC in 2016 [NIS17]. This effort culminated in the selection of several lattice-based schemes, including CRYSTALSKyber [Sch+21], CRYSTALSDilithium [Bai+21], Falcon [Pre+21], and SPHINCS+ [Ber+19] in 2022, followed by HQC [Gab+25] in 2025. Additional schemes are still being evaluated as part of the ongoing standardization process.

At the time of this thesis, the community is undergoing a transition towards post-quantum secure public-key cryptographic techniques, motivated by the anticipated emergence of scalable quantum computers [Aru+19]. While draft standards are now available, migrating the existing cryptographic systems to these new primitives is a long-term effort. This evolving landscape makes a clear case for an agile hardware design methodology for PQC: implementations should be designed to tolerate algorithmic and parameter updates, rather than being tightly coupled to a single fixed instantiation. The goal is to provide a practical, maintainable path to secure, efficient post-quantum deployments. In this context, three aspects emerge as particularly important for future PQC hardware implementations:

① *Unified design.* Most publickey applications require both- KEM for secure key exchange and DSA for authenticity. Hardware architectures that can support

both primitives and multiple parameter sets within a single framework enable resource sharing and reduce area requirements.

② *Compact and agile architectures.* PQC schemes and their recommended parameters are still evolving. Therefore, implementation strategies must remain compact in size and flexible enough to adapt to new security levels or even new schemes without requiring a complete redesign.

③ *Secure implementations.* Side-channel and fault attacks have repeatedly demonstrated that mathematically secure schemes can be compromised through their implementations. Protecting PQC accelerators against such threats is essential for their practical deployment.

When the thesis work began, there were very few efforts in this direction, and these three aspects shaped the **first line of work in this thesis**: *efficient, compact, agile, and secure implementation of PQC schemes* ❶, resulting in several publications [Aik+22; Aik+23a; Aik+23b; Imr+23; Imr+24; Pag+24].

The post-quantum secure KEMs and DSA mentioned earlier are designed to ensure the security of key establishment and authentication, thereby protecting communication channels even against quantum-capable adversaries. However, these methods alone do not offer confidentiality during computation. In typical cloud computing environments, user data is often decrypted for processing on the server, meaning it is available in plaintext at some point during execution. Therefore, privacy relies heavily on effective isolation mechanisms and strong trust in the cloud provider and its software infrastructure.

In practice, these trust assumptions have not always been maintained, as demonstrated by numerous high-profile data breaches in recent years. In addition to the technical risks, such breaches result in significant economic costs. Industry benchmarking studies estimate that the global average total cost of a data breach was USD 4.44 million in 2025, while the average cost in the United States reached USD 10.22 million [IBM25]. Earlier results from the same study series indicated a global average cost of USD 4.88 million in 2024 [IBM24]. Together, these findings underscore that privacy failures not only constitute security incidents but also cause major disruptions to business operations, motivating techniques that reduce the exposure of sensitive data, even in outsourced computation settings.

Fully Homomorphic Encryption (FHE) [RAD78] addresses this gap by enabling arithmetic operations to be performed directly on encrypted data, without revealing the underlying plaintext to the computing party. This is illustrated in Figure 1.3. Gentry introduced the first FHE construction in 2009 [Gen09], establishing the feasibility of general computation over ciphertexts. Since then, several FHE schemes [BGV11; Che+17; Chi+20; FV12] have achieved orders-of-magnitude improvements in performance. Among these, CKKS [Che+17; Che+18a] supports approximate arithmetic, which aligns well with the numerical workloads common in practice, making it widely used in applications such as Privacy-Preserving Machine Learning (PPML).

Despite these substantial advancements and the appealing concept of computing on encrypted data, FHE remains far from practical or efficient. Although

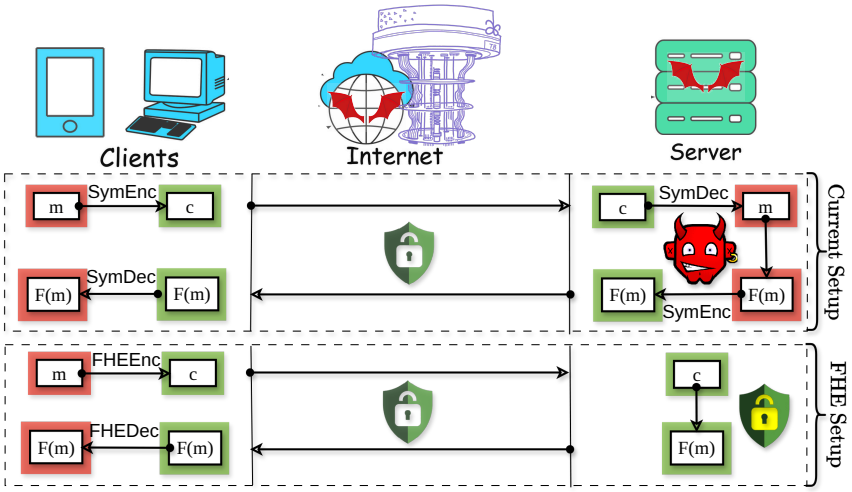


Figure 1.3: Conventional versus Privacy-preserving Computation using FHE.

over a decade of research in algorithmic, mathematical, and implementation optimizations has made modern FHE schemes many times faster than their first-generation counterparts, homomorphic evaluation in software remains typically four to five orders of magnitude slower than comparable plaintext computations. This persistent performance gap is not just an engineering challenge; it directly limits the feasibility of FHE in latency-sensitive and high-throughput environments. Consequently, *hardware acceleration* has emerged as a crucial enabling technology for practical deployments, aiming to reduce the high cost of dominant arithmetic operations and to lower end-to-end latency.

Early efforts in hardware acceleration focused on overcoming the computational bottlenecks of first-generation, integer-based FHE constructions, where performance was hindered by the need for extremely large-integer arithmetic. Notably, the earliest reported FPGA-based [WH13] and ASIC-based [Wan+14] accelerators targeted the 768K-bit modular multiplication required by the first-generation integer-based homomorphic encryption scheme [GH11]. These designs established an important precedent: while algorithmic advancements are essential, the practicality of FHE also relies on dedicated architectures that can efficiently implement its core number-theoretic operations at scale.

Building on early demonstrations, the focus of FHE acceleration shifted as the underlying schemes and their significant bottlenecks evolved. Initial constructions relied heavily on large-integer arithmetic, with massive multi-precision modular multiplication being the primary performance target. However, the widespread adoption of the Residue Number System (RNS) [Gar59] representations fundamentally changed this landscape. By breaking down large modulus into multiple smaller machine-word-sized primes, RNS allows FHE arithmetic to be performed as a series of parallel operations on smaller moduli, with periodic basis conversions.

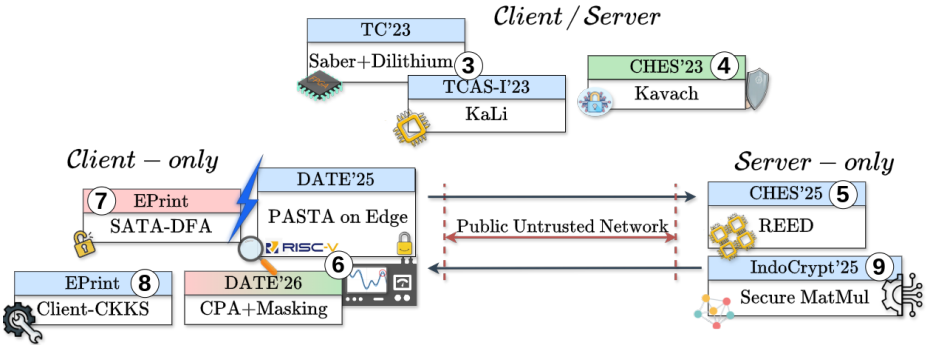


Figure 1.4: Overview of this thesis in the asymmetric post-quantum setting, grouped into three categories: (i) Client/Server covers acceleration of post-quantum key encapsulation and digital signature schemes, and side channel protections; (ii) Client-only includes embedded implementations enabling homomorphic encryption and decryption, along with associated attacks and countermeasures; and (iii) Server-only comprises server side acceleration of homomorphic encryption and an optimized application. ⓧ denotes the chapter where these works are described.

Consequently, contemporary lattice-based FHE schemes are typically dominated by high-level operations such as the Number Theoretic Transform (NTT) [Sco17a] and related modular transforms, rather than large-integer multiplications.

This evolution has spurred the development of a new generation of domain-specific accelerator architectures that target these computational kernels, translating algorithmic advances into significant performance improvements. In recent years, several ASIC-based accelerator architectures have been proposed [Fel+21; Kim+22a; Kim+22b; Sam+22], achieving speedups of three to four orders of magnitude over software for homomorphic evaluation. However, estimates for fabrication costs indicate that producing such ASICs requires multimillion-dollar investments [MUSa]. This makes efficient design methodologies essential, and motivates the **second focus of this thesis**: *cost-effective hardware-accelerator design for FHE* ❷. This line of work has also led to several publications [Aik+25; Aik+26a; Aik+26b; AKR26; AR24; ASR25a; Kim+24a; Mer+23; Muk+24].

Overall, the thesis advances two essential pillars of the digital world: *secure communication* and *privacy-preserving computation*. Firstly, it supports the transition to post-quantum public-key cryptography by developing efficient implementations and analyzing critical design choices regarding security. Secondly, it enhances practical encrypted computation by improving the performance and cost-effectiveness of FHE implementations, particularly through hardware-oriented approaches. The rest of this chapter outlines these contributions and explains how the subsequent chapters will explore these themes through efficient implementations, cryptanalysis, and the design of effective countermeasures.

1.1 Thesis Roadmap

In terms of cryptographic scope, the core primitives studied in this thesis- PQC mechanisms for key establishment and signatures (KEM, DSA) as well as FHE, belong to *asymmetric cryptography*, in contrast to symmetric schemes such as AES [NIS01], which rely on a shared secret key. Nevertheless, several of its most interesting results emerge precisely at the boundary between asymmetric and symmetric techniques: practical systems routinely combine public-key mechanisms for establishing trust or enabling encrypted computation with symmetric primitives that deliver efficiency at scale. The work presented in this thesis is organized around three deployment perspectives in the asymmetric post-quantum setting, as summarized in Figure 1.4. *Client/Server* comprises acceleration of post-quantum key encapsulation and digital signature schemes, as well as side-channel protections. *Client-only* focuses on embedded implementations of homomorphic encryption and decryption, along with the corresponding attacks and countermeasures. Finally, *Server-only* covers server-side acceleration of homomorphic encryption and an optimized application built on top of these capabilities. Throughout the figure, ⓧ indicates the chapter in which each contribution is described, and the chapter-by-chapter summary of these contributions is provided below.

■ **Chapter 2 – Crypto Ground Zero** details the mathematical background of the lattice-based PQC schemes Saber, Kyber, and Dilithium. It then describes the lattice-based FHE scheme CKKS, followed by efficient polynomial multiplication techniques widely used throughout this work, and concludes with an overview of cryptanalytic attacks and countermeasures against side-channel adversaries.

■ **Chapter 3 – Kali** presents hardware implementations of the PQC schemes Saber, CRYSTALSKyber, and CRYSTALSDilithium. As discussed earlier, the motivation is to support a wide variety of applications and therefore to provide unified implementations of both KEM and DSA for these schemes. Thus, in the first work, we propose a unified design for Saber and CRYSTLAS-Dilithium. We leverage their synergies to minimize hardware cost. On a Xilinx UltraScale+ FPGA, it uses just 18,406 LUTs, 9,323 FFs, 4 DSPs, and 24 BRAMs at 200 MHz, completing key-gen/encaps/decaps for LightSaber in 29.6/40.4/58.3 μs and sign/verify for Dilithium-2 in 151.6/75.2 μs. An ASIC implementation in UMC 65nm doubles the clock to cut latencies in half.

• *A Unified Cryptoprocessor for Lattice-Based Signature and Key-Exchange* [Aik+23b] by Aikata, Ahmet Can Mert, David Jacquemin, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy, in IEEE Transactions on Computers (TC 2023).

After NIST declared CRYSTALS-Dilithium and CRYSTALS-Kyber amongst the schemes to be standardized, a second architecture, KaLi, was designed to unify these two schemes. Its flexible polynomial unit runs Kyber operations twice as fast as Dilithiums, while a careful multi-clock-domain memory design minimizes stalls. In 28nm/65nm ASIC KaLi occupies 0.263/1.107 mm^2 at 2GHz/560MHz;

on a Zynq UltraScale+ FPGA it uses 23,277 LUTs, 9,758 FFs, 4 DSPs, 24 BRAMs at 270 MHz and outperforms standalone implementations of each scheme. Thus, we design an accelerator that supports both key exchange and digital signatures across all security variants, ensuring long-term utility. This chapters contents are derived from the following two publications.

• *KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium* [Aik+22] by Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy, published in IEEE Transactions on Circuits and Systems I (TCAS-I 2022).

■ **Chapter 4 – Kavach** addresses protecting the implementations of the PQC schemes against side-channel adversaries. Lattice-based cryptography relies heavily on polynomial arithmetic, and performance is often improved by using compact multipliers because secret coefficient ranges are small. However, these optimizations are difficult to apply in side-channel protected implementations, as masking typically expands the coefficient range. This work proposes two novel techniques- one for NTT-based and one for Schoolbook-based polynomial multiplication, that enable masking while still leveraging small secrets, preserving the efficiency of compact multipliers. Demonstrations include both schoolbook and NTT-based masked multipliers, achieving significant area savings with minimal overhead. The results are validated using Test Vector Leakage Assessment (TVLA). Its contents are derived from the following publication.

• *Kavach: Lightweight masking techniques for polynomial arithmetic in lattice-based cryptography* [Aik+23a] by Aikata, Andrea Basso, Gaetan Cassiers, Ahmet Can Mert, and Sujoy Sinha Roy, appeared in IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES 2023).

■ **Chapter 5 – REED** onwards, we move in a new direction, FHE, and discuss the first work, namely, HW acceleration of FHE for the cloud. Existing ASIC accelerators for FHE are monolithic, which poses challenges such as inflexibility, low yield, and high manufacturing costs. This work introduces REED, the first multi-chiplet-based FHE accelerator, overcoming these limitations through scalable chiplet design, efficient workload distribution, custom inter-chiplet communication, and advanced pipelined Number Theoretic Transform and automorphism units. REED achieves up to $2,991\times$ speedup over a 24-core CPU, improves performance by $1.9\times$ compared to state-of-the-art ASICs, and reduces development costs by 50%. It also demonstrates the first benchmark for encrypted deep neural network training. Its contents are derived from the following publication:

• *REED: Chiplet-Based Accelerator for Fully Homomorphic Encryption* [Aik+25] by Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy, appeared in IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES 2025).

• **Chapter 6 – Secure Cryptoprocessor for Hybrid HE** introduces the Hybrid Homomorphic Encryption (HHE) paradigm, which utilizes symmetric cryptographic techniques in conjunction with FHE to enhance the client's perfor-

mance and reduce communication overhead, particularly for applications such as secure machine learning. Despite various HHE schemes, no comprehensive hardware evaluation existed until this work, which presents PASTA-on-edge, the first implemented HHE scheme over integers [Dob+23] optimized for fast client encryption and homomorphic symmetric decryption on the server. The design was demonstrated on FPGA and ASIC platforms, including a RISC-V SoC in 130nm technology, achieving up to 171× speedup over CPU implementations, and on advanced 7nm and 28nm ASICs, showing up to 97× speedup over prior FHE client accelerators.

• *PASTA on Edge: Cryptoprocessor for Hybrid Homomorphic Encryption* [ASR25a] by Aikata, Daniel Sanz Sobrino, and Sujoy Sinha Roy, appeared in IEEE Design, Automation and Test in Europe (DATE 2025).

Building on these advances, we present the first power side-channel analysis of an HHE-tailored symmetric scheme. Although recent integer-based HHE designs rely on prime-field arithmetic, their side-channel security had not been studied. We demonstrate a practical CPA key-recovery attack on HERA with as few as 40 traces, and introduce the first masking framework for integer-based HHE primitives, combining PINI-secure gadgets with assembly-level protections and validating it via TVLA. We further show its applicability to PASTA, providing leakage models, attack targets, and evaluation benchmarks for side-channel-resilient designs. The design and benchmarks are publicly available to support future research. Its contents are derived from the following work:

• *Attacking and Securing Hybrid Homomorphic Encryption Against Power Analysis* [Aik+26a] by Aikata, Maciej Czuprynko, Nedzma Musovic, Emira Salki, and Sujoy Sinha Roy, appeared in IEEE Design, Automation and Test in Europe (DATE 2026).

■ **Chapter 7** – SASTA-DFA describes an interesting fault-based attack on the HHE paradigm- SASTA-DFA. While HHEs correctness has been studied, its security against fault attacks had not been explored. This work presents SASTA-DFA, the first Differential Fault Analysis (DFA) targeting the HHE protocol as a whole. Unlike prior DFA attacks, Sasta-DFA does not require nonce reuse and achieves full key recovery with a single fault injection, exposing new vulnerabilities inherent to the HHE structure. We validate the attack on emerging HHE-specific symmetric schemes like Rubato [Ha+22], Hera [Cho+21], PASTA [Dob+23], and MASTA [Ha+20], and even show its applicability to schemes like Rasta [Dob+18a] and AES [NIS01]. Experimental validation on an ATXmega128D4-AU microcontroller confirms the attacks practicality. We also extend the analysis to authenticated transciphering protocols. Thus, reaffirming that the need for secure cryptographic protocol designs continues to evolve. This chapter is inspired by the following work:

• *SASTA: Ambushing Hybrid Homomorphic Encryption Schemes with a Single Fault* [Aik+26b] by Aikata, Ahaan Dabholkar, Dhiman Saha, and Sujoy Sinha Roy, in AsiaCCS 2026.

■ **Chapter 8 – CKKS Client acceleration** introduces an efficient framework for FHE Client, which reduces the client's communication as well as computation overhead without relying on the symmetric encryption schemes via the HHE paradigm. This solution lies between the two extremes, namely Vanilla FHE encryption/decryption and HHE. This chapter is derived from the following work:

• *Privacy at your Fingertips: Enabling Rapid Client-Side Operations in Fully Homomorphic Encryption* [AKR26] by Aikata, Florian Krieger, and Sujoy Sinha Roy, in EuroS&P 2026.

■ **Chapter 9 – Privacy-preserving Matrix Multiplication** moves towards high-level application development. This work tackles the critical bottleneck of matrix multiplication in FHE, the backbone of privacy-preserving neural network evaluations, by drastically reducing the expensive KeySwitch operations through smarter SIMD data packing. While prior methods achieve $\mathcal{O}(d)$ complexity at depth two or three, our approach attains depth-2 multiplication with only $\mathcal{O}(\log d)$ KeySwitch operations, matching the best packing efficiency. We further extend the method to arbitrary packing granularities and rectangular matrices, making it universally applicable across all SIMD-capable FHE schemes. This chapter is based on our results as described in:

• *Secure and Efficient Outsourced Matrix Multiplication with Homomorphic Encryption* [AR24] by Aikata and Sujoy Sinha Roy, in INDOCRYPT 2024.

■ **Chapter 10 – Conclusion** completes this thesis and gives interesting insights into future exploration directions for these works. This chapter is a must-read, especially for new researchers who intend to explore these directions.

1.2 Other Notable Contributions

During my PhD thesis, I had the opportunity to contribute to several research efforts initiated by other researchers. A brief summary of these contributions, listed in reverse chronological order of publication, is as follows:

① **ToFa**: This is the first fault attack strategy that leverages impossible differential cryptanalysis to practically break bit-oriented symmetric ciphers like GIFT [Ban+17] and BAKSHEESH [Bak+23]. The attack exploits truncated differential paths caused by fault injections to construct multi-round impossible differentials, enabling efficient key elimination through partial decryption. By analyzing the quotient-remainder structure of the ciphers's linear layers, the distinguisher's filtering capability is significantly enhanced.

I contributed to this work through the practical realization of the fault attack. Clock glitch-based fault attacks were implemented on 8-bit microcontrollers using the ChipWhisperer Lite platform, achieving the expected unique key recovery for GIFT-128 and reducing the key space to 2^{32}, the best known result for GIFT-64.

• *ToFA: Towards Fault Analysis of GIFT and GIFT-like Ciphers Leveraging Truncated Impossible Differentials* [Kun+24] by Anup Kumar Kundu, Shibam

Ghosh, Aikata, and Dhiman Saha, in TCHES 2025.

② **REPQC**: This work exposes a critical weakness in PQC implementations by demonstrating that PQC hardware accelerators can be backdoored by adversaries in the chip supply chain. It introduces REPQC, an automated reverse engineering algorithm capable of accurately locating cryptographic components (e.g., Keccak) within the accelerator, providing anchor points for extracting sensitive information. Using this knowledge, stealthy Hardware Trojan Horses (HTHs) can be inserted without affecting circuit performance and with negligible power overhead. Demonstrated using Dilithium as a case study, the attack shows that even a 0.1% layout density increase is sufficient to embed the HTH undetected.

I contributed to this work by initiating the idea of inserting trojans specifically aimed at compromising secret keys and by designing the trojans to leak information via side-channel mechanisms.

• *REPQC: Reverse Engineering and Backdooring Hardware Accelerators for Post-quantum Cryptography* [Pag+24] by Samuel Pagliarini, Aikata, Malik Imran, and Sujoy Sinha Roy, in AsiaCCS 2024.

③ **FNTT**: This work introduces a multiplier-less number theoretic transform using a Fermat number as an auxiliary modulus to enable fast and resource-efficient polynomial multiplication in homomorphic encryption. To ensure scalability with polynomial degree, a transformation from univariate to multivariate polynomial rings is applied. Building on these algorithmic optimizations, we designed a hardware accelerator architecture for fully homomorphic encryption, achieving up to $1,200\times$ speed-up over software implementations on practical benchmarks.

I contributed to the architecture design of the FHE processor, specifically by integrating the optimized NTT design into the system and handling the increased communication overhead by performing the transformation to multivariate rings on-the-fly.

• *Exploring the Advantages and Challenges of Fermat NTT in FHE Acceleration* [Kim+24a] by Andrey Kim, Ahmet Can Mert, Anisha Mukherjee, Aikata, Maxim Deryabin, Sunmin Kwon, Hyung Chul Kang, and Sujoy Sinha Roy, in CRYPTO 2024.

④ **ModHE**: This work introduces ModHE, a homomorphic encryption scheme based on the Module Learning with Errors (MLWE) problem, aiming to reduce design complexity while supporting deeper encrypted computations. Unlike traditional RLWE-based schemes, which require increasing polynomial sizes for higher circuit depths, ModHE breaks large computations into smaller, more manageable units. As a case study, the CKKS scheme is adapted to the MLWE setting, eliminating polynomial-size blowups and enabling more flexible, reusable, and parallelizable hardware architectures. A hardware prototype demonstrates the practical benefits.

I contributed to this work by designing the hardware architecture, adapting the Medha design [Mer+23] to accommodate the increased computational needs

of ModHE, and developing an efficient data flow model tailored to its structure.

• *ModHE: Modular Homomorphic Encryption Using Module Lattices: Potentials and Limitations* [Muk+24] by Anisha Mukherjee, Aikata, Ahmet Can Mert, Yongwoo Lee, Sunmin Kown, Maxim Deryabin, and Sujoy Sinha Roy, in TCHES 2024.

⑤ **Medha**: FHE allows computations on encrypted data, making it ideal for privacy-preserving cloud computing. Since software implementations of FHE are extremely slow, this work presents Medha, a programmable hardware accelerator designed to speed up cloud-side HE operations. A divide-and-conquer technique enables Medha to flexibly support various encryption parameter sets using a single hardware core. The architecture emphasizes parallelism and efficient data handling, avoiding off-chip memory to improve speed and resource usage. Implemented on a Xilinx Alveo U250 FPGA, Medha significantly outperforms optimized software like Microsoft SEAL.

I began my journey into FHE by contributing to this work, designing the automorphism unit, and assisting in application benchmarking.

• *Medha: Microcoded Hardware Accelerator for computing on Encrypted Data* [Mer+23] by Ahmet Can Mert, Aikata, Sunmin Kown, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy, in TCHES 2022.

1.3 Open-Source Hardware/Software Artifacts

An objective of this thesis is not only to propose algorithms and architectures, but also to bridge the gap between research and real-world applications by creating systems that are operational. To achieve this, every contribution is supported by comprehensive hardware and software artifacts- ranging from FPGA prototypes and ASIC RTL with synthesis processes to microarchitectural simulators, benchmarking tools, and evaluation scripts. A significant portion of these implementations has been made publicly available as open artifacts (where permissible). This includes reference hardware designs, software and firmware components, test and attack infrastructure, and the measurement pipelines used for performance and side-channel evaluations (e.g., TVLA and attack workflows). Below is a list of the artifacts:

- **Kavach** (masked polynomial arithmetic + TVLA tooling) git.
  Contains reference implementations of the lightweight masking techniques for polynomial multiplication, together with test vectors and TVLA scripts to reproduce leakage evaluation and overhead analysis.

- **REED** (software simulation framework) git.
  Provides the REED accelerator artifacts used in the evaluation, including software simulation of the design components and the benchmarking setup for performance results.

- **PASTA-on-edge** (HHE cryptoprocessor, RISC-V SoC).
  Includes the full HW/SW evaluation stack (accelerator RTL and SoC integration), plus build scripts and benchmarks used to reproduce the embedded client-side measurements reported in the thesis.
- **Masking framework for integer-based HHE** (CPA + masking).
  Provides the attack workflow (trace processing and CPA) alongside the proposed masking countermeasures, with measurement scripts and TVLA-based validation to replicate the side-channel evaluation end-to-end.
- **SASTA** (fault-attack proof-of-concept + key-recovery scripts).
  Contains the scripts and tooling to reproduce the single-fault key-recovery attack across multiple HHE-oriented symmetric schemes, including automation for analysis and validation on practical traces/experiments.
- **Boosted-Deflation Client framework** (Full-Stack).
  Implements the client-side optimization framework for CKKS, reducing both client computation and communication overhead. The artifact provides the full-stack software, including hardware implementations (FPGA and ASIC), along with utilities and benchmarks for comparison against baseline CKKS encryption/decryption pipelines.
- **Privacy-preserving matrix multiplication** (software code).
  Implements the proposed SIMD packing strategies that reduce expensive KeySwitch operations, extendable to arbitrary matrix sizes and packing.

Sharing these artifacts improves transparency, makes results easier to reproduce, and lets others inspect design decisions, stress-test masking claims, and see the real costs of cryptographic agility in deployed settings. More importantly, these releases aim to facilitate the start-up process for practitioners and provide researchers with reusable, well-documented reference implementations that can be easily integrated, evaluated, and extended without requiring a complete re-implementation from scratch.

2
Crypto Ground Zero

This chapter provides the foundational background for the two core pillars of this work: PQC and FHE. It begins with an in-depth overview of the PQC schemes employed in this research, followed by a detailed discussion of the selected FHE scheme and its computational routines. Next, we discuss the Hybrid Homomorphic Encryption (HHE) paradigm. This is followed by a description of the widely utilized Number Theoretic Transform (NTT). The chapter concludes with a brief overview of relevant cryptanalytic techniques, particularly those targeting implementations under side-channel adversaries.

Notation

We use $\mathbb{Z}_q$ to represent integers modulo q. We use $\mathbb{R}_q$ to represent polynomial ring $\mathbb{Z}_q/\phi(x)$ where $\phi(x)$ is a reduction polynomial of degree n. We use $\mathcal{B}^l$ to represent a byte array of size l. We use lowercase, bold lowercase, and bold uppercase letters to represent an integer (e.g., $a \in \mathbb{Z}_q$), a polynomial (e.g., $\boldsymbol{a} = \sum_{i=0}^{n-1} \boldsymbol{a}[i] \cdot x^i \in \mathbb{R}_q$) or vector (e.g., $\boldsymbol{a} \in \mathbb{Z}_q^n$), and a vector of polynomials (e.g., $\boldsymbol{A} \in \mathbb{R}_q^{k \times l}$), respectively. We use $\{a\}^l$ to represent a vector of length-l where each element is a. We use $a[i]$, $\boldsymbol{a}[i]$ and $\boldsymbol{A}[i]$ to represent the i-th bit of integer a, i-th of coefficient of polynomial $\boldsymbol{a}$ and i-th polynomial of polynomial vector $\boldsymbol{A}$.

We use $\times$ and $\star$ to represent polynomial multiplication and coefficient-wise multiplication of two polynomials, respectively. We use $\cdot$ to represent integer multiplication (e.g., $a \cdot b$) or multiplication of an integer with coefficients of a polynomial (e.g., $a \cdot \boldsymbol{b}$ or $\boldsymbol{b} \cdot a$). We use $\&$, $|$, $\oplus$, $!$, and $\gg$ / $\ll$ to represent logical AND, OR, XOR, NOT, and right/left shift operations, respectively. We represent the sampling of an integer a from uniform distribution as $a \xleftarrow{\$} \mathbb{Z}_q$. Similarly, we

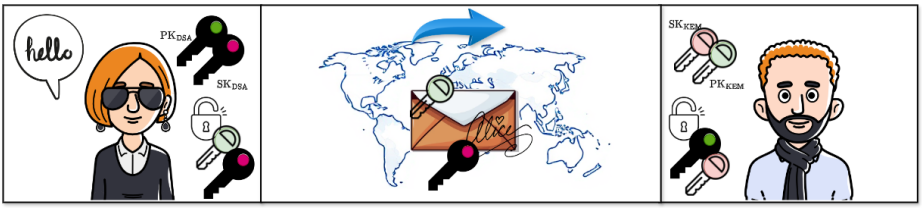


Figure 2.1: Depiction of the communication protocol using KEM and DSA.

use $\boldsymbol{a} \xleftarrow{\$} \mathbb{R}_q$ and $\boldsymbol{a} \leftarrow \chi(\mathbb{R}_q, \sigma)$ to represent the sampling of polynomial $\boldsymbol{a} \in \mathbb{R}_q$ from uniform distribution and χ with standard deviation σ, respectively. A polynomial $\boldsymbol{s} \in \mathbb{R}_q$ is *small* if all its coefficients belong to the range $[-\mu, +\mu]$ mod q where μ is a k-bit positive integer and $K = 2^k$ with $\mu \ll q$.

ining polynomials of degree at most $N-1$ and coefficients in $\mathbb{Z}_Q$. In the Residue Number System (RNS) [Gar59] representation, Q is a composite modulus comprising co-prime moduli, $Q = \prod_{i=0}^{L-1} q_i$. The RNS representation is used to divide a big computation modulo Q into much smaller computations modulo q_i such that the small computations can be carried out in parallel. With the application of RNS, a polynomial $a \in \mathbb{R}_{Q,N}$ becomes a vector, say $\boldsymbol{a}$, of residue polynomials. Let the i-th residue polynomial within $\boldsymbol{a}$ be denoted as $a^i \in \mathbb{R}_{q_i,N}$. We use the 'monotype' font (`c/sk`) to represent ciphertexts/keys. Operators $\cdot$ and $\langle , \rangle$ denote the multiplication and dot-product between two ring elements. Noise (e) is refreshed for every computation. The tilde sign ($\tilde{}$) represents a data in NTT format (e.g., $\texttt{NTT}(a) = \tilde{a}$).

2.1 Post-Quantum Cryptography

PQC aims at developing new cryptographic protocols that will remain secure even after powerful quantum computers are built. These are asymmetric PKC schemes, which will replace the classical PKC schemes. Figure 2.1 depicts a sample protocol, where Alice (left) encapsulates the message (or symmetric encryption key) using Bob's KEM public key, so only Bob can decrypt it, using his private KEM key. To provide proof of authenticity, Alice further signs the Encapsulated message with her Private DSA Key. Thus, Bob can validate that the message could have only come from Alice, using the public DSA key.

2.1.1 Lattice-based public key encryption schemes

Several lattice-based PKE/KEM schemes are based on the Learning With Errors (LWE) problem and its variants [Reg09]. For a given public matrix $\boldsymbol{A} \in \mathbb{Z}_q^{k \times l}$ and vector $\boldsymbol{B} = \boldsymbol{A} \times \boldsymbol{S} + \boldsymbol{E} \in \mathbb{Z}_q^k$, there is no known algorithm that can recover secret vector $\boldsymbol{S} \leftarrow \chi(\mathbb{Z}_q^l, \sigma)$ in polynomial time where $\boldsymbol{E} \leftarrow \chi(\mathbb{Z}_q^k, \sigma)$ is an error vector. Solving the secret $\boldsymbol{S}$ in the presence of error $\boldsymbol{E}$ is known as the LWE problem. A variant of the LWE problem is the Learning With Rounding

Algorithm 1 LPR Encryption Scheme [LPR10]

Procedure PKE.KeyGen()
- $\boldsymbol{a} \xleftarrow{\$} \mathbb{R}_q$
- $\boldsymbol{s}, \boldsymbol{e} \leftarrow \chi(\mathbb{R}_q, \sigma) \in \mathbb{R}_q$
- $\boldsymbol{t} = (\boldsymbol{a} \times \boldsymbol{s} + \boldsymbol{e}) \in \mathbb{R}_q$
- **return** $\mathsf{pk} = (\boldsymbol{a}, \boldsymbol{t}), \mathsf{sk} = (\boldsymbol{s})$

Procedure PKE.Encrypt($\mathsf{pk} = (\boldsymbol{a}, \boldsymbol{t}), \mathsf{m} \in \mathcal{B}^{32}, \mathsf{r} \in \mathcal{B}^{32}$)
- $\boldsymbol{s}', \boldsymbol{e}', \boldsymbol{e}'' \leftarrow \chi(\mathbb{R}_q, \sigma, \mathsf{r})$ `/* r is used as random seed */`
- $\boldsymbol{u} = (\boldsymbol{a} \times \boldsymbol{s}' + \boldsymbol{e}') \in \mathbb{R}_q$
- $\boldsymbol{v}' = (\boldsymbol{t} \times \boldsymbol{s}' + \boldsymbol{e}'') \in \mathbb{R}_q$
- $\tilde{\boldsymbol{m}} = \mathsf{Encode}(\mathsf{m}) \in \mathbb{R}_q$ `/* Encoding msg bits to poly. coefficients */`
- $\boldsymbol{v} = (\boldsymbol{v}' + \tilde{\boldsymbol{m}}) \in \mathbb{R}_q$
- **return** $\mathsf{ct} = (\boldsymbol{u}, \boldsymbol{v})$

Procedure PKE.Decrypt($\mathsf{ct} = (\boldsymbol{u}, \boldsymbol{v}), \mathsf{sk} = (\boldsymbol{s})$)
- $\tilde{\boldsymbol{m}}' = (\boldsymbol{v} - \boldsymbol{u} \times \boldsymbol{s}) \in \mathbb{R}_q$
- $\mathsf{m}' = \mathsf{Decode}(\tilde{\boldsymbol{m}}') \in \mathcal{B}^{32}$ `/* Decoding poly. coefficients to msg bits */`
- **return** m'

(LWR) problem, which introduces deterministic errors by scaling the elements of $\boldsymbol{A} \times \boldsymbol{S}$ by a non-integer constant and rounding them to the closest integer. Ring-LWE/LWR (RLWE/RLWR) and Module-LWE/LWR (MLWE/MLWR) problems [LS15] are implementation-friendly variants of LWE/LWR, and both operate over polynomial rings instead of integers. In the following, we briefly describe the LPR encryption scheme [LPR10], which has served as a framework for constructing modern LWE/LWR-based public-key encryption schemes.

LPR Encryption Scheme: In Algorithm 1, we present the high-level description of key generation, encryption, and decryption procedures of the RLWE-based LPR scheme. Although LPR-based schemes are IND-CPA secure, they are not secure against chosen ciphertext attacks (CCA), where an adversary with access to the decrypted messages can recover the secret key by using carefully selected ciphertexts. To achieve IND-CCA security, LWE/LWR-based schemes employ the Fujisaki-Okamoto (FO) transformation [FO99], which combines encryption and decryption procedures with hash functions to generate CCA-secure encapsulation and decapsulation procedures. With the FO transformation, the decapsulation procedure performs re-encryption after decrypting the message, and then compares the input ciphertext with the generated ciphertext using the re-encryption procedure. If the comparison fails, i.e. the input ciphertext is invalid, the decapsulation mechanism outputs a random byte array.

Next, we give brief descriptions of lattice-based DSA and KEM schemes-CRYSTALS-Dilithium, CRYSTALS-Kyber, and Saber, referred to as Dilithium, Kyber, and Saber from here on. Kyber [Sch+21] and Dilithium [Bai+21] are part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), which were selected for standardization by NIST. Dilithium [Bai+21] is a digital signature scheme, and its security is based on the computational hardness of the Module Learning With Errors (MLWE) and Module Short Integer Solution (MSIS) problems. Kyber is a KEM, and its security relies on the hardness of solving the

MLWE problem. Saber [DAn+21] is also a KEM, and its security relies on the hardness of the Module Learning With Rounding (MLWR) problem.

2.1.2 Dilithium modules

This Digital Signature Algorithm has three main algorithms: key generation, signature generation, and signature verification. The sender generates a public and secret key using the key generation algorithm. Then she uses her private key to sign a message using the signature generation algorithm. The receiver can verify the signature using the sender's public key and signature verification algorithm. The signature generation algorithm continues to generate signatures until a valid one is produced. For a signature to be valid, a set of constraints has to be satisfied to ensure that the signature does not bear any similarity with the message. Depending on the size of the module $\mathbb{R}_q^{k\times\ell}$ with $k, \ell > 1$, Dilithium [Bai+21] also comes with three variants, namely Dilithium-2, 3, and 5 for the NIST-specified security levels 2, 3, and 5, respectively. All the three variants of Dilithium use the polynomial ring $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ with $n = 256$ and $q = 2^{23} - 2^{13} + 1$, a prime modulus. The Dilithium building blocks are briefly explained as follows.

- **Polynomial generation**: SHAKE-128 is used to generate the polynomials of the public matrix $\boldsymbol{A} \in \mathbb{R}_q^{k\times\ell}$ by expanding the seed $\rho \in \{0,1\}^{256}$ along with 16-bit nonce values. The secret polynomial vectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2 \in S_\eta^\ell \times S_\eta^k$ are generated using SHAKE-256. For each polynomial, the seed ς and a 16-bit nonce are fed to SHAKE-256 and passed through rejection sampling in the range $\{-\eta, \eta\}$. The two types of generations are named as `ExpandA`() and `ExpandS`(). `ExpandMask`(), is used to generate a polynomial vector in the range $[0, 2\gamma_1 - 1]$ using a rejection sampler. `SampleInBall`() is used during signature generation and verification, to generate a polynomial with only τ coefficients set to $+1$ or -1 and the remaining coefficients as 0.

- **Polynomial Arithmetic**: Polynomial multiplications are performed using the Number Theoretic Transform (NTT) method. The addition and subtraction operations are coefficient-wise linear operations.

- **Hash functions**: SHAKE-256 is used to make a collision resistant hash function- `CRH`(). It utilizes 384 bits of the output of SHAKE-256.

- **Power2Round** : The function, $\texttt{Power2Round}_q()$, takes an element $r = r_1 \cdot 2^d + r_0$ and returns r_0 and r_1, where $r_0 = r \mod^{\pm} 2^d$ and $r_1 = (r - r_0)/2^d$.

- **Decompose and other related functions**: Let α be a divisor of $q - 1$. The function $\texttt{Decompose}_q()$ is defined in the same way as `Power2Round`() with α replacing 2^d. The $\texttt{HighBits}_q()/\texttt{LoweBits}_q()$ return r_1/r_0 from the output of $\texttt{Decompose}_q()$. `MakeHint` uses $\texttt{HighBits}_q()$ to produce a hint $\boldsymbol{h}$. `UseHint` uses the hint $\boldsymbol{h}$ produced by $\texttt{MakeHint}_q()$ to recover the high-bits.

The signing operation generates a potential signature and checks a set of constraints on the generated signature. If satisfied, a valid signature is produced as the output; otherwise, the loop continues to generate another potential signature.

2.1.3 Kyber modules

Kyber [Sch+21] is an IND-CCA2-secure Key Encapsulation Scheme. It has three principal algorithms: key generation, encapsulation, and decapsulation. The receiver generates a public and secret key using the key generation algorithm and broadcasts the public key. When the sender wishes the send a message, he/she can encapsulate it using the receiver's public key through the encapsulation algorithm. The receiver can then decapsulate it using their secret key through the decapsulation scheme. Three variants of Kyber, Kyber-512, Kyber-768, and Kyber-1024 are provided for NIST Security levels 1, 3, and 5, respectively. All the variants use the polynomial ring $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, with $n = 256$ and $q = 3329 = (2^{12} - 3 \cdot 2^8 + 1)$. These variants differ in module dimensions and coefficient distributions. Kyber has the following internal routines.

- **Pseudorandom functions**: Kyber uses PRF (SHAKE-256) and XOF (SHAKE-128) to generate the pseudo-random numbers for generating the coefficients of polynomials.
- **Hash and other related functions**: Kyber provides functions H and G for SHA3-256 and SHA3-512, respectively, for hashing. The **Key-derivation function** (KDF) is instantiated using SHAKE-256 in Kyber. Uniform sampling (Parse) is used to generate the public polynomials, and Binomial sampling (CBD) is used to generate secret and error polynomials.
- **Polynomial Arithmetic**: Kyber uses an incomplete NTT-based polynomial multiplication unit. Linear operations, such as polynomial addition and subtraction, are also supported.
- **Encode/Decode** and **Compress/Decompress**: The **Encode/Decode** modules are used to serialize/deserialize the polynomials to/from byte arrays. The **Compress/Decompress** modules are used to reduce the size of the ciphertext by discarding low-order bits. They are defined on an element $x \in \mathbb{Z}_q$ as $\lceil (2^d/q) \cdot x \rfloor \pmod{2^d}$ and $\lceil (q/2^d) \cdot x \rfloor$ respectively, where $d < \lceil \log_2(q) \rceil$. The value x' such that $x' = \texttt{Decompress}(\texttt{Compress}(a, d), d)$ is an element close to x.

2.1.4 Saber modules

Saber, like Kyber, is a KEM. It has three variants: LightSaber, Saber, and FireSaber, targeting different security levels. All of these variants use the same polynomial rings $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and $R_p = \mathbb{Z}_p[x]/\langle x^n + 1\rangle$ with polynomial degree $n = 256$ and the power-of-two moduli $q = 2^{13}$ and $p = 2^{10}$. The three variants use different module dimensions and secret distributions. We briefly

describe the internal routines of Saber. For a detailed description of them, readers may follow the original specification of Saber [DAn+21].

- **Pseudorandom functions**: `gen`() expands a uniform seed $\rho \in \{0,1\}^{256}$ using the Keccak-based expandable output function SHAKE-128 and generates the public matrix $\boldsymbol{A} \in \mathbb{R}_q^{l \times l}$. $\beta_\mu()$ samples a secret polynomial vector ($\boldsymbol{s}$) from a binomial distribution with the parameter μ.

- **Hash functions**: Saber uses three hash functions: $\mathcal{F}()$, $\mathcal{H}()$ and $\mathcal{G}()$. The $\mathcal{F}()$ and $\mathcal{H}()$ are implemented using SHA3-256 while $\mathcal{G}()$ is implemented using SHA3-512. All hash functions are Keccak-based.

- **Polynomial arithmetic**: They include polynomial multiplication, polynomial addition/subtraction, coefficient-wise rounding using bit-shifting, equality checking of two polynomials, etc. Unlike Kyber and Dilithium, Saber uses power-of-two moduli, which do not directly enable NTT-based efficient polynomial multiplication.

- **Verify** and **CMOV**: **Verify** is used to perform ciphertext equality check during decapsulation. The result of Verify is stored in a flag register, which is used by **CMOV** (constant-time move) to either copy the decrypted session key or a pseudo-random string to a specified location.

- **AddPack/UnPack** and **AddRound**: **AddPack** performs the operation $(v + \mathbf{h}_1 - 2^{\epsilon_p - 1} m \bmod p).(\epsilon_p - \epsilon_T)$ on a message bits m using precomputed polynomial v and constant $\mathbf{h}_1$. `UnPack` performs the operation $(v + \mathbf{h}_2 - 2^{\epsilon_p - \epsilon_T} c_m \bmod p)(\epsilon_p - 1)$ on a ciphertext c_m using precomputed polynomial v and constant $\mathbf{h}_2$. **AddRound** performs the operation $(((\mathbf{A}^T + \mathbf{s}) \bmod q).(\epsilon_q - \epsilon_p))$ using pubilic matrix vector $\mathbf{A}$ and secret vector $\mathbf{s}$.

2.2 Fully Homomorphic Encryption

Data breaches can put millions of private accounts at risk because data is often stored or processed without encryption, making it vulnerable to attacks [IBM20; LSL20; Mor20]. Fully Homomorphic Encryption (FHE) is a solution that allows secure, private computations, communications, and storage, as shown in Figure 2.2. It enables servers to compute on homomorphically encrypted data and return encrypted outputs. FHE has a wide range of applications, including cloud computing [Kim+23b; Mah+23], data processing [Bia+23], and machine learning [Pap+18]. The concept of FHE was introduced in 1978 by Rivest, Adleman, and Dertouzos [RAD78], and the first FHE scheme was constructed in 2009 by Gentry [Gen09].

Different FHE schemes exist in literature, such as, BFV [FV12], BGV [BGV11], CGGI [Chi+20], and CKKS [Che+17; Che+18b]. These schemes use polynomial arithmetic but differ primarily in the data types they can encrypt. For instance, BGV and BFV encrypt integers, while CKKS encrypts fixed-point numbers. Due to the support for fixed-point arithmetic, CKKS is widely adopted for



Figure 2.2: Depiction of the FHE flow.

Table 2.1: CKKS Parameters

Parameter	Definition
$N, n\ (\leq \frac{N}{2})$	Polynomial size, maximum slots packed
Q, q_i	Coefficient modulus, RNS bases $Q = \prod_{i=0}^{L} q_i$
L, l	Multiplicative depth (#RNS bases - 1) $l < L$
$dnum$	Number of digits in the switching key
P, p_i	Special modulus and its RNS base
$K\ (= \lceil \frac{L+1}{dnum} \rceil)$	Number of RNS bases for $P = \prod_{i=0}^{K-1} p_i$
w	Word size ($\log p_i, \log q_i$)
L_{boot}, L_{eff}	Multiplicative depth of/after bootstrapping

benchmarking machine learning applications [Han+19; Kim+18]. Therefore, this work targets the RNS (Residue Number System) CKKS [Che+18b]. Other FHE schemes, such as BGV and B/FV, are also based on RLWE and require similar operations to those in CKKS. Thus, these schemes can utilize the same design methodology for varying parameters. In the following, we briefly describe the main procedures within the RNS CKKS [Che+18b; HK19; KPP22] for ciphertexts at level l (multiplicative depth is l) where $l < L$, $Q_l = \prod_{i=0}^{l} q_i$, and L is the maximum level. The residue polynomial associated with each modulus q_i in the RNS representation is commonly called the RNS limb. A CKKS ciphertext consists of components, e.g., $\mathtt{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$, where $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$ are vectors of limbs. Table 2.1 describes the CKKS parameters, and algorithmic descriptions are provided for $dnum = L + 1 (K = 1)$.

1. $\mathtt{CKKS.Enc}(m, \mathtt{pk})$: Encrypts message m, and returns $\mathtt{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1) = v \cdot \mathtt{pk} + (\Delta \cdot \mathtt{IDFT}(m) + e_0, e_1) \in \mathbb{R}_Q^2$, where $e_0, e_1/v$ are sampled from distributions-$\chi_{err/enc}$.

2. $\mathtt{CKKS.Dec}(\mathtt{c}, \mathtt{sk})$: Decrypts $\mathtt{c}$ using the secret key $\mathtt{sk}$ to return message $m' = \mathtt{DFT}(\Delta^{-1} \cdot \langle \mathtt{c}, \mathtt{sk} \rangle) = f(m)$.

3. $\mathtt{CKKS.Add}(\mathtt{c}, \mathtt{c}')$: As shown in Algorithm 2, this operation takes two input ciphertexts $\mathtt{c}$ and $\mathtt{c}'$ and computes $\mathtt{c}_{\text{add}} = (\boldsymbol{d}_0, \boldsymbol{d}_1) = (\boldsymbol{c}_0 + \boldsymbol{c}'_0, \boldsymbol{c}_1 + \boldsymbol{c}'_1)$.

4. $\mathtt{CKKS.Mult}(\mathtt{c}, \mathtt{c}')$: It multiplies the two input ciphertexts $\mathtt{c}$ and $\mathtt{c}'$, as shown in Algorithm 4, and computes the non-linear ciphertext $\mathtt{d} = (\boldsymbol{d}_0, \boldsymbol{d}_1, \boldsymbol{d}_2) = (\boldsymbol{c}_0 \cdot \boldsymbol{c}'_0, \boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}_1 \cdot \boldsymbol{c}'_0, \boldsymbol{c}_1 \cdot \boldsymbol{c}'_1)$. Subsequently, $\mathtt{CKKS.KeySwitch}$ transforms $\mathtt{d}$ into a linear ciphertext. The computation is done on data in NTT format.

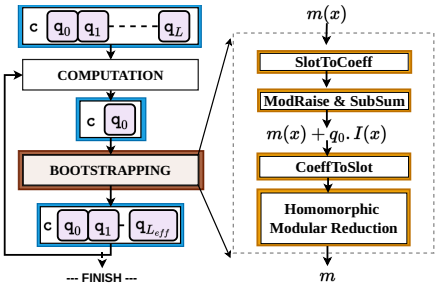


Figure 2.3: A high-level view of the Bootstrapping procedure used to refresh the ciphertext's multiplicative depth.

5. `CKKS.Rotate`$(\mathtt{c}, rot, \mathtt{ksk}_{\mathtt{rot}})$: It rotates the plaintext slots within `c` by *rot*. First, a permutation ρ is applied to the ciphertext polynomial coefficients, as shown in Algorithm 3. This permutation is called an automorphism and is determined by the Galois element $gle = 5^{rot} \bmod 2N$. Finally, the permuted ciphertext is processed by `CKKS.KeySwitch` using the rotation key $\mathtt{ksk}_{\mathtt{rot}}$.

Algorithm 2 `CKKS.Add` [Che+18b]

In: $\mathtt{c} = (\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1)$, $\mathtt{c}' = (\tilde{\mathbf{c}}'_0, \tilde{\mathbf{c}}'_1) \in R^2_{Q_l}$
Out: $\mathtt{c}_{\text{add}} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \in R^2_{Q_l}$
1: $\tilde{\mathbf{d}}_0 \leftarrow \tilde{\mathbf{c}}_0 + \tilde{\mathbf{c}}'_0$, $\tilde{\mathbf{d}}_1 \leftarrow \tilde{\mathbf{c}}_1 + \tilde{\mathbf{c}}'_1$

Algorithm 3 `CKKS.Rot` [Che+18b]

In: $\mathtt{c} = (\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1) \in R^2_{Q_l}$, *rot*
Out: $\mathtt{c}_{\text{rot}} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \in R^2_{Q_l}$
1: $(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \leftarrow \rho_{rot}(\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1)$

Algorithm 4 `CKKS.Mult` [Che+18b]

In: $\mathtt{ct} = (\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1) \in R^2_{Q_l}$
In: $\mathtt{ct}' = (\tilde{\mathbf{c}}'_0, \tilde{\mathbf{c}}'_1) \in R^2_{Q_l}$
Out: $\mathtt{d} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1, \tilde{\mathbf{d}}_2) \in R^3_{Q_l}$
1: $\tilde{\mathbf{d}}_0 \leftarrow \tilde{\mathbf{c}}_0 \cdot \tilde{\mathbf{c}}'_0$
2: $\tilde{\mathbf{d}}_2 \leftarrow \tilde{\mathbf{c}}_1 \cdot \tilde{\mathbf{c}}'_1$
3: $\tilde{\mathbf{d}}_1 \leftarrow \tilde{\mathbf{c}}_0 \cdot \tilde{\mathbf{c}}'_1$
4: $\tilde{\mathbf{d}}_1 \leftarrow \tilde{\mathbf{d}}_1 + \tilde{\mathbf{c}}_1 \cdot \tilde{\mathbf{c}}'_0$

6. `CKKS.KeySwitch`$(\mathtt{d}, \mathtt{ksk})$: It uses evaluation keys `ksk` to homomorphically transform a ciphertext decryptable under one key into a new ciphertext decryptable under another (original) key, as illustrated in Algorithm 6. It computes $\mathtt{c}''$ where $\boldsymbol{c}''_0 = \sum_{i=0}^{l-1} d_2^i \cdot ksk_0^i \in \mathbb{R}_{PQ_l,N}$ and $\boldsymbol{c}''_1 = \sum_{i=0}^{l-1} d_2^i \cdot ksk_1^i \in \mathbb{R}_{PQ_l,N}$. This is followed by $\mathtt{c} = \big((d_0, d_1) + \mathtt{CKKS.ModDown}(\mathtt{c}'')\big) \in \mathbb{R}^2_{Q_l,N}$. `CKKS.ModDown`() scales down the modulus (PQ_l to Q_l), and is described in Algorithm 5 following the works [HK19; KPP22].

7. `CKKS.Bootstrap`: During homomorphic computations on the server, the multiplicative depth l gets successively reduced until it reaches $l = 0$. In this case, no further multiplications can be done, and a bootstrapping procedure (BS) [Bos+21b; CCS19; Che+18c] is required to refresh the computational depth of a ciphertext. However, this process consumes some of the refreshed depth, resulting in a ciphertext with a computational depth $l = L_{eff}$ ($< L$) post-BS. Figure 2.3 provides an overview of the BS process used by CKKS [Bae+24], which consists of four steps.

(a) **Slot to Coefficient Conversion:** After performing several computations, the initial ciphertext `c` encrypting m is left with 1 RNS-base q_0 ($l = 0$). At this stage, the ciphertext contains an encryption of m in slot form. `SlotToCoeff` converts this to a ciphertext encrypting the polynomial form $m(x)$. This involves a homomorphic DFT computation achieved through matrix multiplication with the DFT matrix.

(b) **Modulus Raising:** A `ModRaise` operation is performed to raise the modulus from q_0 to Q. The resulting ciphertext $\mathtt{c}'$ now encrypts $m(x) + q_0 \cdot I(x)$. The extra term $q_0 \cdot I(x)$ has low-magnitude integer coefficients, multiplied by q_0. The next goal is to eliminate $q_0 \cdot I(x)$ to retrieve the original encryption by performing a modular reduction.

(c) **Coefficient to Slot Conversion:** Now, the ciphertext is converted back to encrypt a vector in slot form $m + q_0 \cdot I$, again involving multiplication with the IDFT matrix.

(d) **Homomorphic Modular Reduction:** The data, after `CoeffToSlot` described above, is prepared for `Homomorphic Modular Reduction`. This step usually uses a Chebyshev series-based polynomial approximation of the modular reduction function [Al +22]. Together, these steps consume a computation depth $L_{boot} < L$, resulting in a ciphertext with a refreshed depth $L_{\mathit{eff}} = L - L_{boot}$.

Algorithm 5 `CKKS.ModDown` [Che+18b]

In: $\tilde{\mathbf{d}} \in R_{PQ_l}$
Out: $\tilde{\mathbf{d}}' \in R_{Q_l}$

$t \leftarrow \mathtt{INTT}(\tilde{d}[l+1])$
for $i = 0$ to l **do**
  $\tilde{t} \leftarrow \mathtt{NTT}(\left[t\right]_{q_i}) \in \mathbb{Z}_{q_i}$
  $\tilde{d'}[i] \leftarrow \left[p_0^{-1} \cdot (\tilde{d}[i] - \tilde{t})\right]_{q_i}$
end for

- **Slot to Coefficient Conversion:** Converts the ciphertext from slot form to polynomial form using a homomorphic DFT (Discrete Fourier Transformation). This involves computing a homomorphic matrix-vector multiplication, where the matrix is not encrypted, and the ciphertext is the vector.
- **Modulus Raising:** Raises the modulus from q_0 to Q, introducing an error term that needs removal. This requires a plain ModUp operation,
- **Coefficient to Slot Conversion:** Converts the ciphertext back to slot form after modulus raising via homomorphic IDFT (Inverse DFT) computation. This also involves computing a homomorphic matrix-vector multiplication.

- **Homomorphic Modular Reduction:** This step applies a Chebyshev polynomial approximation to remove the introduced error term after Modulus raising. It results in refreshed computational depth.

Algorithm 6 `CKKS.KeySwitch` [HK19; KPP22] (for $dnum = L + 1$)

In: $\mathtt{d} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1, \tilde{\mathbf{d}}_2) \in R^3_{Q_l}$, $\tilde{\mathtt{ksk}}_0 \in R^l_{PQ_l}$, $\tilde{\mathtt{ksk}}_1 \in R^l_{PQ_l}$

Out: $\mathtt{d}' = (\tilde{\mathbf{d}}'_0, \tilde{\mathbf{d}}'_1) \in R^2_{Q_l}$

1: **for** $j = 0$ to l **do**
2: $\quad d_2[j] \leftarrow \mathtt{INTT}(\tilde{d}_2[j]) \in \mathbb{Z}_{q_j}$
3: **end for**
4: **for** $j = 0$ to $l + 1$ **do**
5: $\quad (\tilde{\mathbf{c}}''_0[j], \tilde{\mathbf{c}}''_1[j]) \leftarrow 0$
6: $\quad$ **for** $i = 0$ to l **do**
7: $\quad\quad \tilde{r} \leftarrow \mathtt{NTT}(\left[d_2[i]\right]_{q_j}) \in \mathbb{Z}_{q_j}$
8: $\quad\quad \tilde{c}''_0[j] \leftarrow \left[\tilde{c}''_0[j] + \tilde{\mathtt{ksk}}_0[i][j] \cdot \tilde{r}\right]_{q_j}$, $\tilde{c}''_1[j] \leftarrow \left[\tilde{c}''_1[j] + \tilde{\mathtt{ksk}}_1[i][j] \cdot \tilde{r}\right]_{q_j}$
9: $\quad$ **end for**
10: **end for**
11: $\tilde{\mathbf{d}}'_0 \leftarrow \tilde{\mathbf{d}}_0 + \mathtt{CKKS.ModDown}(\tilde{\mathbf{c}}''_0)$, $\tilde{\mathbf{d}}'_1 \leftarrow \tilde{\mathbf{d}}_1 + \mathtt{CKKS.ModDown}(\tilde{\mathbf{c}}''_1)$

2.3 Hybrid Homomorphic Encryption

HHE addresses the above issues, and a detailed top-level overview of the HHE protocol is depicted in Figure 2.4, and Table 2.2. It proceeds as follows.

- The client has two types of keys, the symmetric encryption key K and asymmetric private and public keys $\{sK, pK\}$ for FHE. These keys are long-term keys. The client needs to protect both K and sK because knowledge of sK can reveal K, and this, in turn, allows the attacker to gain access to all messages m_i as well as the results sent by the server.

- Next, the client uses the public key pK to encrypt K using $\mathtt{FHE}_{Encrypt}$ and sends this to the server at the beginning itself[1]. After this, whenever clients

[1]Note that the client also computes evaluation keys and sends them to the server in the

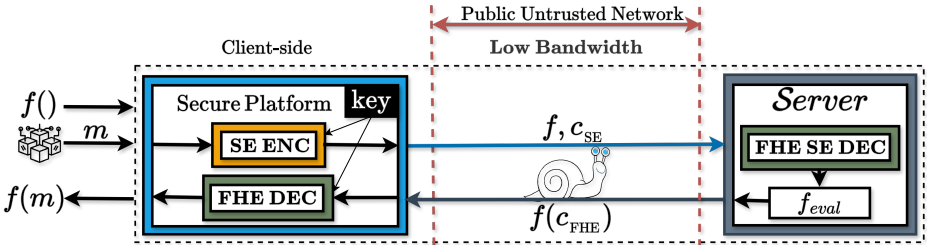


Figure 2.4: The client symmetrically encrypts the plaintext (SE ENC) and sends it to the server. The server converts it to FHE ciphertext, evaluates the function, and returns the result. The client then decrypts (FHE DEC) the final ciphertext homomorphically.

Table 2.2: A simplified algorithmic description of the HHE protocol. Here, i refers to the i-th iteration of the HHE protocol.

Client		**Server**
Initialization		
Generate K and $\{sK, pK\}$ $K_{FHE} = \texttt{FHE}_{Enc}(K)_{pK}$ Send pK and K_{FHE}	$\rightarrow$	$\rightarrow$ Save pK and K_{FHE}
Encryption using SE		
$c_{i_{SE}} = m_i + \texttt{E}_\pi(K, n_i)$ Send $c_{i_{SE}}$, n_i, and $f()$	$\rightarrow$	$\rightarrow$ $c_{i_{SE}}$, n_i, and $f()$
Homomorphic SE *Decryption Circuit Evaluation (*HHD*)*		
		$c_{i_{FHE}} = c_{i_{SE}} - \texttt{FHE}_{E_\pi}(n_i, K_{FHE})$
Function Evaluation		
		Compute $d_{i_{FHE}} = f(c_{i_{FHE}})$
$d_{i_{FHE}}$	$\leftarrow$	$\leftarrow$ Send $d_{i_{FHE}}$
Result Decryption		
$f(m_i) = \texttt{FHE}_{Dec}(d_{i_{FHE}})_{sK}$ ▶*Decryption Oracle*		

need to store or process data on the cloud, they use SE Encryption to encrypt the message blocks m_i using K and send the resultant ciphertext $c_{i_{\texttt{SE}}} = m_i + \texttt{E}_\pi$ to the server along with the public nonce n_i. $\texttt{E}_\pi$ refers to the key-dependent permutation that results in a keystream, which is added to plaintext for encryption.

- The server uses the encrypted key $\texttt{K}_{\texttt{FHE}}$ along with the nonce n_i to evaluate the SE decryption (HHD) circuit homomorphically ($\texttt{FHE}_{\texttt{E}_\pi}$). This results in a ciphertext ($c_{i_{\texttt{FHE}}}$), which is only homomorphically encrypted under key sK. This can now be stored on the server or processed as the client desires.

- The client can retrieve the resultant ciphertext and use sK for decryption.

2.3.1 Pasta Scheme Design Overview

In the context of HHE, the variables can be categorized into two types: public and private. As illustrated in Figure 2.5, the nonce (n) and the counter (ctr) are considered public data, as they are known to both the client and the server. In contrast, the key (K) is private and exclusively known to the client. For HHD, this key is encrypted and securely transmitted to the server (this process is done only once initially). Hence, both K and the client's message (m_i) remain concealed from the server.

With the public and private variables clearly defined, we now describe the Pasta HHE scheme. We have chosen Pasta [Dob+23] for our implementation as other HHE scheme designs can be viewed as adaptations or variations of Pasta.

initialization phase. We do not mention them as they do not play a key role in our investigation.

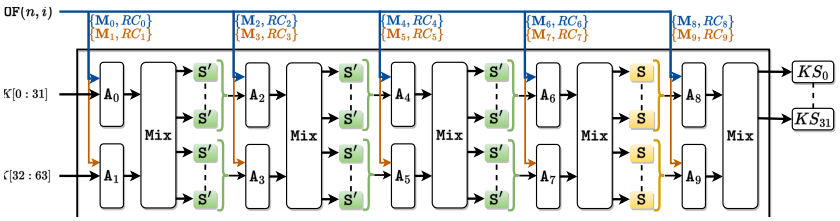


Figure 2.5: The PASTA-t permutation (π) takes as input the key (K), nonce (n), and counter (ctr), and generates the truncated result- key stream (KS).

This scheme operates as a stream cipher and comprises two variants: 3-round PASTA-3 and 4-round PASTA-4. Figure 2.5 demonstrates the PASTA permutation (π). It is important to note that the operations outside the square box, denoted as `XOF` (extendable-output function), are public. SHAKE128 is used for this. Contrarily, the operations within the box are considered private (key-dependent) and involve either addition or multiplication using modular arithmetic in $\mathbb{Z}_p$. Here, p can be any prime between 16 and 60 bits depending on the specific requirements of the underlying FHE scheme.

The state size ($2t$) varies between the PASTA-3 and PASTA-4 variants of the scheme, where t is the size of one block. Specifically, for PASTA-3, $2t = 128$ coefficients, while for PASTA-4, it is 64. These $2t$ coefficients are divided into two halves, X_L and X_R, and then processed via permutation. The resulting KS (KeyStream) is added to the plaintext block (size t) for encryption. Each PASTA round consists of several layers, described as follows:

- $\mathtt{A}_i$ (Affine Layer): For this layer, an *invertible* matrix $\mathbf{M}_i$ and a round constant vector RC_i are generated using the SHAKE128 `XOF` output. Then, the layer performs $\mathbf{M}_i \cdot X_i + RC_i$ operation, where X_i represents the input state comprising t coefficients.

- `Mix` (Mixing Layer): Following $\mathtt{A}_i$, the two halves of the state are mixed using the `Mix` operation. This operation transforms the state into $(2 \cdot X_L + X_R, 2 \cdot X_R + X_L)$. This step is crucial for spreading values evenly across the two-state halves.

- $\mathtt{S}'/\mathtt{S}$ (S-Box Layer): The next layer involves the S-Box operation. For the final round, the cube S-Box (`S`) is applied, while for all previous rounds, a Feistel S-Box ($\mathtt{S}'$) is utilized. Both these S-boxes are *invertible*.

$$\mathtt{S}'(X^j) = \begin{cases} X^j \pmod p & \text{if } j = 0, \\ X^j + (X^{j-1})^2 \pmod p & \text{otherwise,} \end{cases}$$

- Truncation Layer (`Trunc`): It returns the X_L state as the final output. This is applied at the end (pre-final) and truncates the output to prevent round inversion.

2.3.2 Invertible Matrix Generation Method

The affine layer in PASTA requires invertible matrices $\mathbf{M}_i$ to ensure one-to-one mapping between input and output. The steps for generating them are as follows.

- Generate a vector of t elements using SHAKE128 `XOF`. This constitutes the first row of the matrix $\mathbf{M}_i^0 = [\alpha_0, \alpha_1, \cdots, \alpha_{t-1}]$.
- Next, use this row to generate the remaining rows using (2.1) [GPP11; Guo+11]. This ensures that the matrix is invertible

$$\mathbf{M}_i^{j+1} = \mathbf{M}_i^j \cdot \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 0 & 0 & 0 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_{t-1} \end{bmatrix} \quad \forall\ 1 \leq j < t \tag{2.1}$$

2.3.3 MASTA and RASTA design overview

MASTA [Ha+20] follows a slightly different approach as shown in Figure 2.6. It does not split the state into two halves like PASTA and comprises only two primary layers: the affine layer and the S-box layer. The affine layer of MASTA is similar to that of PASTA; however, MASTA employs χ-S-box (Equation 2.2) only. The pre-final step is the most significant distinguishing factor between PASTA and MASTA. While PASTA opts for truncation at this stage, MASTA employs modular addition with the key. Both PASTA and MASTA are versatile and can support operations over $\mathbb{F}_p$, making them suitable for BGV and BFV. RASTA [Dob+18a] is similar to MASTA with the only difference being operation over $\mathbb{Z}_2$ and the choice of S-box.

$$\mathtt{S}_\chi(X^j) = X^j + X^{j+2} + X^{j+1} \cdot X^{j+2} \pmod{p} \tag{2.2}$$

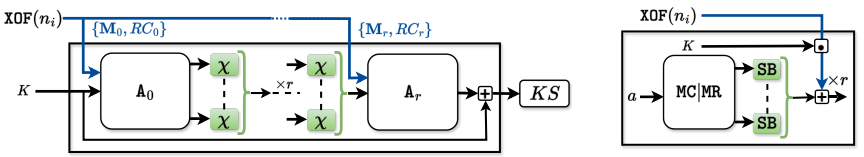


Figure 2.6: The r-round MASTA and a general idea of HERA and RUBATO round functions.

2.3.4 HERA design overview

HERA [Cho+21] adopts a distinct approach compared to PASTA and MASTA (as shown in Figure 2.6). It comprises five rounds, all utilizing cube S-boxes, thus

requiring a modulus such that $\gcd(p-1,3)=1$. The primary distinguishing feature of HERA is its approach to the affine layer. In this layer, matrix multiplication (MC|MR) utilizes constant low-hamming weight matrices, while the addition of round constants is transformed into an add-round key function. For this, the output of the XOF is multiplied by the encryption key, creating a key-schedule-like effect. After the S-box (SB) operation, every round adds this derived key to the state. Importantly, HERA can be applied to FHE schemes over $\mathbb{F}_p$ (BGV, BFV, and CKKS).

2.3.5 RUBATO design overview

RUBATO [Ha+22] was developed after PASTA and HERA, utilizing design principles from both schemes (Figure 2.6). Specifically tailored for CKKS FHE scheme, RUBATO employs the same design philosophy as HERA. However, it deviates from HERA by employing feistel S-boxes instead of cube S-boxes for all rounds. In its pre-final step, RUBATO utilizes truncation like PASTA and adds Gaussian noise, making the ciphertext dependent on the hard problem of LWE. This, in turn, limits its use case to CKKS and introduces an inherent precision loss in the computation.

2.4 Polynomial Multiplication Techniques

Polynomial multiplication is one of the fundamental operations in lattice-based cryptography. There are different approaches to implementing polynomial multiplication. The selection of the proper implementation method depends on the scheme parameters, target performance, and platform. Most implementations use the traditional NTT-based multiplication technique, while others show how methods like schoolbook $\mathcal{O}(n^2)$ or Karatsuba $\mathcal{O}(n^{1.59})$ can be used.

2.4.1 Schoolbook Polynomial Multiplication

For polynomials $\boldsymbol{a} \in \mathbb{R}_q$ and $\boldsymbol{b} \in \mathbb{R}_q$, the polynomial multiplication $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ is defined as $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \boldsymbol{a}[i] \cdot \boldsymbol{b}[j] \cdot x^{i+j}$. When the multiplication is performed in a polynomial ring, a separate reduction operation with reduction polynomial $\phi(x)$ is required. When $\phi(x)$ has a special form, this reduction operation can be merged into the multiplication operation free of cost. For example, when $\phi(x)$ is x^n+1, $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ is defined as $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (-1)^{\lfloor (i+j)/n \rfloor} \cdot \boldsymbol{a}[i] \cdot \boldsymbol{b}[j] \cdot x^{i+j \pmod n}$. using $x^n \equiv -1$ [Liu+19]. Although the Schoolbook method has high computational complexity ($\mathcal{O}(n^2)$), it works with every parameter set and does not impose any constraints on the polynomial ring or ring modulus.

2.4.2 Karatsuba Polynomial Multiplication

Karatsuba method follows a divide-and-conquer approach and has the complexity $\mathcal{O}(n^{\log_2 3})$ [KO62]. Let $\boldsymbol{a} = \boldsymbol{a}_h x^{n/2} + \boldsymbol{a}_l$ and $\boldsymbol{b} = \boldsymbol{b}_h x^{n/2} + \boldsymbol{b}_l$ be polynomials of

Algorithm 7 The Cooley-Tukey NTT Algorithm [Sco17b]

In: A polynomial x with coefficients $\{x_0, \cdots, x_{N-1}\}$ where $x_i \in \mathbb{Z}_q$, n, q
In: Table of $2n^{th}$ roots of unity $\boldsymbol{g}$, in bit reversed order
Out: $\hat{x} \leftarrow \texttt{NTT}(x)$, $\hat{x}_i \in \mathbb{Z}_q$, in bit-reversed order
1: $t, m \leftarrow (n/2), 1$
2: **while** $(m < n)$ **do**
3: $k \leftarrow 0$
4: **for** $(i = 0; i < m; i = i + 1)$ **do**
5: **for** $(j = k; j < (k + l); j = j + 1)$ **do**
6: $V \leftarrow x[j + t] \times \boldsymbol{g}[m + i] \pmod q$ ▷ Butterfly operation starts.
7: $x[j + t] \leftarrow x[j] - V \pmod q$
8: $x[j] \leftarrow x[j] + V \pmod q$ ▷ Butterfly operation ends.
9: **end for**
10: $k \leftarrow k + 2t$
11: **end for**
12: $t, m \leftarrow t/2, 2m$
13: **end while**
14: **return** x

size n where $\boldsymbol{a}_h$, $\boldsymbol{a}_l$, $\boldsymbol{b}_h$ and $\boldsymbol{b}_l$ are $(\frac{n}{2} - 1)$-degree polynomials. Then, $\boldsymbol{a} \times \boldsymbol{b}$ can be written as $(\boldsymbol{a}_h \times \boldsymbol{b}_h)x^n + (\boldsymbol{a}_h \times \boldsymbol{b}_l + \boldsymbol{a}_l \times \boldsymbol{b}_h)x^{n/2} + (\boldsymbol{a}_l \times \boldsymbol{b}_l)$. Karatsuba method reduces the number of $(\frac{n}{2} - 1)$-degree polynomial multiplications from 4 to 3 by using the term $(\boldsymbol{a}_h \times \boldsymbol{b}_l + \boldsymbol{a}_l \times \boldsymbol{b}_h) = (\boldsymbol{a}_h + \boldsymbol{a}_l) \times (\boldsymbol{b}_h + \boldsymbol{b}_l) - \boldsymbol{a}_h \times \boldsymbol{b}_h - \boldsymbol{a}_l \times \boldsymbol{b}_l$. Karatsuba can be applied recursively to reduce the size of the multiplication operations.

2.4.3 NTT-based Polynomial Multiplication

The Fast Fourier Transform (FFT) is an efficient method to compute the DFT. Defined over complex numbers, the FFT is particularly useful in reducing the complexity of DFT from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. This makes it highly relevant in FHE schemes, where efficient polynomial arithmetic is essential. NTT is defined as the Discrete Fourier Transform over finite fields $\mathbb{Z}_{q_i}$. As shown in Algorithm 7, an n-point NTT operation transforms a polynomial a of degree $n - 1$ degree polynomial into another $n - 1$ degree polynomial $\tilde{a}$. The NTT uses the powers of n-th root of unity ω (also referred to as twiddle factors) which satisfies $\omega^n \equiv 1 \pmod q$ and $\omega^i \neq 1 \pmod q$ $\forall\ i < n$, where $q \equiv 1 \pmod n$. Here, q represents an RNS base, q_i. Similarly, inverse NTT (INTT) follows the same method with the modular inverse of ω, and the resulting coefficients should be scaled by $1/n$. The NTT implementation techniques can be broadly classified into three types: Iterative, Pipelined, and Hierarchical. The major difference between works is how they instantiate the butterfly unit highlighted in Algorithm 7.

- **Iterative.** In this type of implementation, the prior works [Mer+23; Sco17a] instantiate multiple butterfly units to process the inner 'for' loop (Algorithm 7) simultaneously, hence faster. Thus, the runtime of the NTT transformation for m butterfly units is $\sim \frac{n \cdot \log n}{2 \cdot m}$.

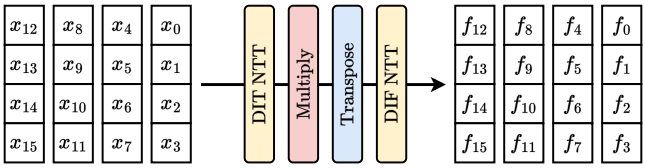


Figure 2.7: Hierarchical (4-step) NTT datapath for $n = 16$. DIT stands for Decimation in Time, and DIF stands for Decimation in Frequency.

- **Pipelined.** This is a bandwidth-efficient implementation where the butterfly units operate in pipelines instead of in parallel. It is also used by prior works [YCH22; Zha+21] for applications such as zero-knowledge proofs, where the polynomial degree is huge. It can be considered a stage unrolled NTT, where the unrolling is done based on the outer 'for' loop instead of the inner loop (in Algorithm 7). While its latency does not decrease, its throughput is much higher and useful for applications requiring multiple polynomial NTT transformations.

- **Hierarchical (4-Step).** This implementation technique adopted by [Fel+21], predominantly utilizes the first technique as shown in Figure 2.7. It splits one polynomial into a matrix of dimensions $N_1 \times N_2 = n$. The NTT units transform N_1 coefficients using multiple butterfly units instantiated in parallel, and there are N_2 such sets to process all $N_1 \times N_2$ coefficients. After this first operation, the resultant data is transposed and multiplied by twiddle factors. Finally, other NTT units transform N_2 coefficients using multiple butterfly units instantiated in parallel, and there are N_1 such sets to process all the coefficients. To save area N_1, N_2 are chosen so that $N_1 = N_2$. This way, the same NTT unit can be utilized for both transformations ($N_1 \times N_2$, $N_2 \times N_1$). In this case, the transpose operation becomes complex and expensive as n increases.

2.5 Implementation Attacks Overview

2.5.1 Correlation Power Analysis

Correlation Power Analysis (CPA) is a side-channel attack that exploits the relation between secret values and power consumption during computations. First introduced as an extension of Differential Power Analysis (DPA) [KJJ99], CPA was studied in [BCO04; CKN00; May00]. A CPA attack targets an intermediate computation of the form $r = f(v, s)$, where v is a known input, s is the secret, and r is the result [MOP07a]. Power consumption is commonly modeled using Hamming weight leakage model, which assumes that the power consumption is proportional to the number of set bits in an intermediate register value r. Over many executions, this leakage reveals information about s. The attacker collects power traces from multiple runs, guesses intermediate values for all

possible secrets, and maps them to predicted power using the leakage model. By correlating the predictions with the measured traces (via Pearson correlation), the secret yielding the highest correlation is identified. In practice, many traces are required for accurate recovery.

2.5.2 Masking Countermeasure

Masking is a key countermeasure against power analysis attacks, including Simple Power Analysis, Differential Power Analysis, and Correlation Power Analysis. Masking a computation means replacing every intermediate value x with a randomized tuple (named a sharing) that represents the value, and whose elements are the shares. Then, all computations are modified to operate on these shares: every computation is replaced by a gadget that performs the same computation on the shares. In this thesis, we are mainly interested in first-order arithmetic masking, where the sharing of a value $x \in \mathbb{Z}_q$ is a tuple $(x_0, x_1) \in \mathbb{Z}_q^2$ such that $x_0 + x_1 = x \mod q$. The sharing (x_0, x_1) of x is *uniform* if its probability distribution is uniform over the set $\left\{(x_0, x_1) \in \mathbb{Z}_q^2 | x_0 + x_1 = x \mod \mathbb{Z}_q\right\}$, and a sharing is *fresh* if its distribution over that set is independent of any other value in the computation (excluding values that are computed from these shares). As a particular case, Boolean masking is arithmetic masking in $\mathbb{Z}_2$. We also extend the notion of sharing to polynomials (coefficient-wise): arithmetic shares of $\boldsymbol{s} \in \mathbb{R}_q$ are the tuple $(\boldsymbol{s}_0, \boldsymbol{s}_1) \in \mathbb{R}_q$ if $\boldsymbol{s}_0 + \boldsymbol{s}_1 = \boldsymbol{s}$.

The fundamental concept involves splitting each sensitive variable x into $d+1$ uniformly distributed random masked variables, referred to as shares, where d represents the masking order. Chari et al. [CRR02] were the first to propose this technique, demonstrating that the measurement complexity of a single-bit DPA increases exponentially with the value of d [CRR02]. The security of masking is most often studied in the t-probing model [ISW03]. A gadget is t-probing secure if any set of t ($t = 1$ for first-order masking) intermediate values in the computations (named probes) is independent of the secret values of the inputs of the gadget, assuming that the input shares are fresh. This abstract leakage model does, however, not cover the leakage caused by glitches (or other intra-cycle data-dependent timing) in hardware implementations, which led to the introduction of the glitch-robust probing model [Fau+18]. In this model, a probe on a wire observes not only the value of the wire but also all the inputs of the combinatorial circuit that computes that wire; therefore, only the synchronization elements (i.e., the registers) stop the propagation of glitches.

2.5.3 Differential Fault Analysis of SE

Differential Fault Analysis is a physical attack that exploits information leaked from faulty computations on a victim's device. Faults are induced by forcing the device to operate in unexpected environmental conditions (such as high voltage surges, clock or EM glitches) and the observed differences between fault-free and faulty outputs are analyzed to reveal information about the internal states. The standard DFA threat model involves a victim device that runs

encryption/decryption protocols with a secret key and an adversary capable of inducing faults in its computations. It also allows the adversary to function in a chosen plaintext or ciphertext setting. The works [Rad+23; RBM21] make an even stronger assumption by allowing repetition of nonces to mount effective attacks.

As pointed out by the authors in [Dob+18b], repeating nonces always incurs a certain loss of security, and the inability to repeat the nonce renders conventional DFA techniques infeasible [SKC14]. The authors in [Dob+16] consider a nonce-respecting threat model for AES-GCM but require multiple encryption queries while necessitating that the client not realize the tag is incorrect and refresh the key. This is a strong assumption for FHE use cases.

The susceptibility of classical SE schemes to fault attacks has been extensively analyzed. Several key-recovery attacks have been demonstrated for RSA-CRT Signatures [BDL97], AES [AM11a; Bar+10; GSR24; Hir+24; Muk09; TMA11]. Prior works [AM11a; SMC09; TMA11] demonstrate that even a single fault during AES encryption can significantly reduce the key space, which can be made unique using known plaintext-ciphertext pairs. We consider DFA over other attack techniques, such as Statistical Ineffective Fault Attack (SIFA) [Dob+18b] or Fault Template Attacks (FTA), because of the HHE setting. The Symmetric Key used in the HHE context is ephemeral and can be refreshed more often than in the traditional Symmetric Encryption setting. This is a problem for techniques like SIFA or FTA that rely on many faulty and fault-free traces under the same key. Moreover, there has been no SIFA or FTA analysis of the new HHE-enabling schemes. Therefore, whether such attacks are even feasible on the new schemes is unknown.

3

Unified Cryptoprocessor for Post-Quantum Schemes

Breu

In this chapter, we first describe our unified architecture for Saber and Dilithium. Then, inspired by those findings and in response to NISTs selection of Kyber and Dilithium, we present a new, unified design methodology for Kyber and Dilithium- KaLi.

Contribution: This chapter is based on two publications for which the thesis author was the lead author. The author contributed to the conceptual development, conducted the experiments (with error analysis contributed by David), implemented a substantial portion of the hardware modules, and wrote large portions of the text.

• *A Unified Cryptoprocessor for Lattice-Based Signature and Key-Exchange* by Aikata, Ahmet Can Mert, David Jacquemin, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy, in IEEE Transactions on Computers (TC 2023).

• *KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium* by Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy, published in IEEE Tran. on Circuits and Systems I (TCAS-I 2022).

3.1 Introduction

This work proposes a design methodology that covers three vital aspects for the future implementations of the PKC algorithms. The first is the need to make a ***unified design***. A majority of PKC applications require both DSA and KEM. Therefore, the design decisions should be adapted to help unify the two algorithms for saving area via resource sharing. Secondly, the design must be ***compact***. The new PKC schemes require much larger memory and logical units to store and process the keys. If we do not attempt to make these designs compact, a lot of resource-constrained CPUs that were designed for classical PKC schemes will be rendered inoperable. The final aspect is ***agility/flexibility***. The architecture design should consider the ever-evolving nature of these algorithms. It will not only help prevent the overconsumption of hardware resources but also enable a smooth transition.

At the time of this work, most of the prior works in literature focused on standalone efficient hardware implementations [Alk+20; BAK21a; BAK21b; BKS19; BNG21; Dan+20; DMG21; FSS20a; Hua+20; LSG21; Ni+22; RB20; Ric+21; Xin+20; XL21a; Zho+21] of PKC schemes. The real-life applications often require both types of schemes. Therefore, these works fell short of providing complete area and timing results for the implementations that make the communication post-quantum secure. The authors in [BUC19b; Zha+22b] presented hardware/-software (HW/SW) co-designs for Dilithium and Kyber. Since they keep some part of the design in software, it was not sufficient to provide a good estimate for hardware-only architectures. There was a need for a unified implementation of these two types of schemes, completely in hardware, to get better performance. We addressed this research gap in this Chapter.

This chapter first describes the unified design for lattice-based KEM Saber [DAn+21] and DSA Dilithium [Bai+21], which were the finalists in NIST's standardization efforts[1]. We extend this with another unified architecture for the NIST-selected KEM Kyber [Sch+21] and Dilithium. We unify the extensive building blocks of these schemes and call the resultant architecture of the latter two schemes- `KaLi`. The design choices aim for reduced area consumption. As a step toward agility, the architecture is modelled as an instruction-set cryptoprocessor. We make the following design decisions towards our goal.

1. Polynomial multiplication is a time- and area-consuming operation in all schemes. To realize a unified polynomial multiplier for Saber and Dilithium, we use the NTT method of polynomial multiplication for both Dilithium and Saber. We show that by using Dilithium 's prime, Saber could use the NTT-based polynomial multiplier of Dilithium readily at the cost of a negligible execution error probability. For `KaLi`, we propose a compact polynomial multiplier architecture that works optimally for the two cryptographic algorithms. Dilithium has a 23-bit prime modulus, whereas Kyber has a 12-bit prime modulus. A unified polynomial arithmetic unit is

[1]At the time of this work, this project was in its second round with Saber and Dilithium as one of the finalists

designed, such that it can process two sets of 12-bit Kyber coefficients. This, in combination with efficient memory management, enables performing arithmetic operations for Kyber twice as fast as Dilithium.

2. All schemes use Keccak-based pseudo-random number generation and hash calculations. However, the pre- and post-prcoessinf performed by these schemes on the input/output data of the Keccak modules differs. To make our cryptoprocessor compact without compromising speed, we implement an optimized wrapper around the Keccak module for performing scheme-specific pre/post-processing of data on the fly. That reduces both the area and the cycle counts significantly.

3. The remaining scheme-specific building block are optimized to reduce their memory access overheads and area. Although these scheme-specific blocks have a linear time complexity, we observe that their optimized implementations (for all the security parameters) in hardware require low-level bit and word manipulations.

4. Starting from the optimized building blocks, we construct a programmable instruction-set architecture (ISA). The ISA computes all the signature and KEM routines and supports all the security levels of these schemes. The designed ISA offers the parallelism to execute several data-independent instructions concurrently so that the KEM and signature operations get faster. The data memory of the ISA is organized to enable concurrent reads/writes by the parallel instructions.

5. `KaLi` is engineered separately for the ASIC platform to reduce area overhead. It uses two clock domains, where the memory unit works at a higher clock frequency than the logic unit. This allowed us to use single-port memory instead of dual-port memory used in the FPGA implementation, thus further reducing the area consumption.

The relevant background for understanding the schemes is provided in Chapter 2. This Chapter is organized as follows. Section 3.2 identifies the synergies between the two schemes- Saber and Dilithium, and presents their hardware architecture. Next, in Section 3.3, we discuss the synergies between Kyber and Dilithium and how it is exploited for designing- `KaLi`. This is followed by results showcased in Section 3.4. Finally, we present a discussion to conclude the Chapter in Section 3.5.

3.2 Architecture Design for Saber and Dilithium

Both Dilithium and Saber are based on module lattices, and therefore, they share several structural similarities. For example, both schemes operate on matrices and vectors of polynomials, where the polynomials are always of 256 coefficients. Hence, the underlying polynomial arithmetic operators are common to Dilithium and Saber. Furthermore, both schemes use Keccak-based hash functions and

pseudo-random number generators. Interestingly, the accumulated time spent on polynomial multiplications, pseudo-random number generations, and hash calculations in the two protocols is around 90% of the overall protocol execution time [Kan+19]. In summary, we observe that the most time-critical primitives in Dilithium and Saber have similarities.

3.2.1 Polynomial multiplication, pseudo-random number generation, and hash computation

We have two options for implementing polynomial multiplications in Dilithium and Saber. The first option is to instantiate an NTT-based multiplier for Dilithium (which uses a prime modulus) and a schoolbook or Toom-Cook or Karatsuba multiplier for Saber (which uses power-of-two moduli), following [RB20], so that both schemes can be executed at their optimal speeds. One significant disadvantage of this approach is that the architecture requires a large area due to the presence of scheme-specific multipliers. Furthermore, a large area requirement could potentially slow down the clock frequency of the implementation due to the increased routing and placement complexities. The other option is to instantiate a common polynomial multiplier for both Dilithium and Saber. In this case, the common polynomial multiplier must be NTT-based as the specification of Dilithium [Zha+22a] makes the use of NTT-based multiplication an integral part of the protocol. Saber [DAn+21] could use any type of polynomial multiplication method, including the NTT-based one.

Both Dilithium and Saber use the Keccak-based hash function SHA3 and pseudo-random number generator SHAKE. Therefore, both schemes could use a common Keccak core along with an appropriate wrapper around the core for realizing different SHA3 and SHAKE functionalities.

Prime selection for Saber To use the NTT-based polynomial multiplication in Saber, a sufficiently large NTT-friendly prime q' is needed to ensure the correctness of the multiplication [Chu+21a]. The polynomial multiplication will be erroneous if a true modular reduction by q' takes place during the internal computation steps. We have two main options for choosing the prime q'. The first option is to choose a sufficiently large q' to ensure no true modular ever happens. The other option is to use a relatively smaller prime that offers some computational advantages over the first option while maintaining a negligible error probability. We describe both options in the following part of this subsection.

Prime selection for error-free multiplication in Saber: To select a prime for Saber which will work for all security levels of Saber without any error, we consider the maximum value that a polynomial coefficient can have after the polynomial multiplication operation. In Saber, one of the input polynomials of a polynomial multiplication operation is always a secret key polynomial which has small coefficients in the range $[-\frac{\mu}{2}, \frac{\mu}{2}]$ where μ is 10, 8, 6 for LightSaber, Saber, FireSaber, respectively [DAn+21]. Similarly, the second polynomial operand of the polynomial multiplication operation in the Saber scheme is always a polynomial with coefficients in the range $[0, q-1]$ where $q = 2^{13}$. If we remap

the coefficients in $[0, q-1]$ to respective coefficients in $[-\frac{q}{2}, \frac{q}{2}-1]$, then the maximum value that a coefficient of the polynomial multiplication result can have is $\frac{\mu}{2} \cdot \frac{q}{2} \cdot n = 5 \cdot 2^{12} \cdot 2^8$. Similarly, the minimum value that a coefficient of the result can have is $-5 \cdot 2^{12} \cdot 2^8$. Therefore, a 24-bit prime q'_{24} can be used in Saber as it will be larger than $2 \cdot 5 \cdot 2^{12} \cdot 2^8$ and therefore no true modular reductions will ever take place during a polynomial multiplication.

In Saber, polynomial matrices and vectors are multiplied, and therefore several polynomial multiplication results are accumulated. These accumulations can be performed directly in the NTT domain to avoid unnecessary inverse NTT transforms. Taking the accumulation into consideration, the maximum and the minimum values that a coefficient of the accumulated result can have will be larger than $\frac{\mu}{2} \cdot \frac{q}{2} \cdot n$ or lower than $-\frac{\mu}{2} \cdot \frac{q}{2} \cdot n$, respectively. The exact maximum value will depend on the dimensions of the matrices and vectors. The public polynomial matrix of Saber has the dimension $l \times l$ where l is 2, 3, and 4 for LightSaber, Saber, and FireSaber, respectively [DAn+21]. The vectors are of size l polynomials.

Hence, the maximum value that a coefficient can take after a matrix-vector multiplication is $l \cdot \frac{\mu}{2} \cdot \frac{q}{2} \cdot n$, which will be at most $3 \cdot 4 \cdot 2^{12} \cdot 2^8$ when Saber or FireSaber is used. Similarly, the minimum value that a coefficient can take after a matrix-vector multiplication is $-3 \cdot 4 \cdot 2^{12} \cdot 2^8$. Therefore, q' is required to be at least 25-bit to ensure the zero error probability for the matrix-vector multiplication in Saber. We selected the 25-bit prime $q'_{25} = 2^{25} - 2^{14} + 1$ for implementing the error-free matrix-vector multiplication of Saber. The sparse structure of the prime enables a fast and lightweight modular reduction circuit.

Prime selection for ensuring a negligible error probability: In the previous subsection, we saw that a 25-bit prime is required when we consider the worst-case scenario. In practice, the probability of having a polynomial coefficient close to the worst-case bound is negligible as the coefficients of the secret polynomials are binomially distributed in the range $[-\frac{\mu}{2}, \frac{\mu}{2}]$ and the coefficients of the public polynomials are uniformly distributed in the range $[0, q-1]$. One attractive choice is the 23-bit prime modulus of Dilithium. With this special prime, Saber can trivially use the NTT-based polynomial multiplier of Dilithium for computing its own polynomial multiplications. At the same time, we must be sure that the smaller prime does not cause a non-negligible error probability.

We now describe a method for estimating the probability of error when a prime smaller than 25 bits is used in Saber's NTT. Note that an error will happen when any resultant coefficient after a matrix-vector multiplication satisfies $|\text{coeff.}| > q'/2$ causing a true modular reduction by q'. To keep the analysis simple, we assume that the secret coefficients are also uniformly distributed in $[-\frac{\mu}{2}, \frac{\mu}{2}]$. When we perform a polynomial multiplication between a secret polynomial and a public polynomial, both of which follow uniform distributions, Irwin-Hall states that the coefficients of the resultant polynomial will follow a normal distribution [JKB95]. Note that a binomially distributed polynomial is significantly sparse compared to a uniformly distributed polynomial in the same range. Hence, the assumption that the secret coefficients follow a uniform

Table 3.1: Distribution parameters and error probabilities

	LightSaber	**Saber**	**FireSaber**
Mean	115.72	48.06	-29.32
Std. dev.	119708.81	97825.69	75672.40
Accumulated mean	231.45	144.18	-117.28
Accumulated std. dev.	169293.83	169439.06	151344.81
$P(\|\text{coeff.}\| > q'_{24}/2)$	2^{-1774}	2^{-1837}	2^{-2219}
$P(\|\text{coeff.}\| > q'_{23}/2)$	2^{-449}	2^{-448}	2^{-558}

distribution enables a more conservative approximation (i.e., an upper bound) of the computation error probability.

We experimentally verified the above statement. We used a Python script to generate pairs of polynomials where the coefficients of the first polynomial are uniformly random in the range $[-\frac{\mu}{2}, \frac{\mu}{2}]$. For the coefficients of the second polynomial, we experimented on both $[-q/2, q/2-1]$ and $[0, q-1]$, and selected $[0, q-1]$ to get the more conservative measurement. Then the two polynomials within a pair are multiplied. The experimentation was performed for millions of independent pairs. The scattered cyan plot in Figure 3.1 shows the distribution of the resultant coefficients from our experimentation. We observe that the experimental distribution closely resembles a normal distribution, and therefore, the Irwin-Hall statement is found to be valid in this scenario.

From the experimental data, the mean and the standard deviation were calculated to plot the theoretical normal distribution (in black color) in Figure 3.1. When such l independent polynomial multiplication results are accumulated, the resultant coefficients will follow a normal distribution with the mean and standard deviation increasing by l and $\sqrt{l}$ times, respectively. In Table 3.1, the distribution parameters are presented for all three security levels of Saber.

Next, we used the cumulative distribution function of the normal distribution to estimate the upper bound for the error probability. Table 3.1 shows the error calculation for a 23-bit prime ($q'_{23} = 2^{23} - 2^{13} + 1$) and a 24-bit prime ($q'_{24} = 2^{24} - 2^{14} + 1$). We can see that the upper bound is extremely low for both the primes. Therefore, the use of Dilithium's 23-bit prime in Saber's NTT-based polynomial multiplication causes negligible error possibilities for all three security levels of Saber.

The authors in [Bas+21a] also provide an error probability estimate when Dilithium's prime is used for Saber. They used combinatorics to count all the possible combinations of a polynomial pair that can lead to an error after polynomial multiplication. Their approach is specific to a *single* polynomial multiplication as they do not take the accumulation of polynomial multiplication results into consideration for calculating the error probabilities.

3.2.2 Remaining scheme-specific building blocks

The remaining building blocks in the two schemes share few similarities, and they mostly perform simpler operations (i.e., addition, packing) with linear

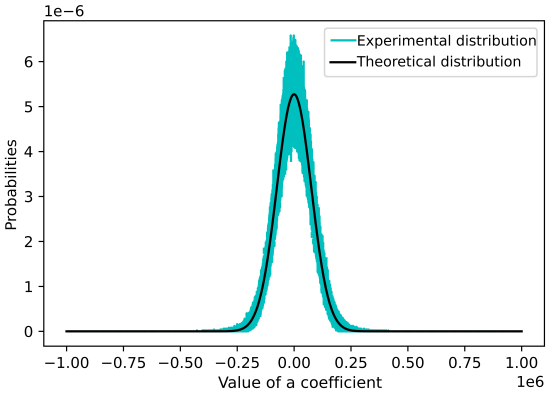


Figure 3.1: Distribution of coefficients after multiplication between secret and public polynomials with coefficients in the range $[-\frac{\mu}{2}, \frac{\mu}{2}]$ and $[0, q-1]$.

time complexity compared to the polynomial multiplication and pseudo-random number generation operations. To further reduce area consumption, an option could be to resource-share a common set of arithmetic circuits (e.g., addition and subtraction) with algorithm-specific finite-state machines for generating the control signals. This design decision might decrease the area, and at the same time, it might make the overall design more complex and serial instead of parallel. Therefore, to make the design simple and easily configurable, we decided to keep the scheme-specific blocks separate in the implementation. However, we do perform optimizations within these blocks. For example, we combine all the different packing and unpacking methods required by Dilithium to create a unified pack/unpack unit that consumes significantly less area compared to separate implementations.

Next, we describe how we design a compact and unified architecture for the lattice-based signature scheme Dilithium and PKE/KEM Saber. Our target is to support all the subroutines and all the security levels of Dilithium and Saber in the same cryptoprocessor. Therefore, we chose the instruction-set architecture (ISA) framework for developing our unified cryptoprocessor. In this framework, the primitive building blocks are 'instructions' that are called in an appropriate sequence during the execution of a cryptographic protocol. Therefore, programmability or flexibility is inherent to the ISA framework, which is a desirable feature for supporting multiple cryptographic schemes.

The high-level block diagram of the proposed unified cryptoprocessor is shown in Figure 3.2. The unified cryptoprocessor for Dilithium and Saber has a common NTT-based polynomial multiplier, a common Keccak-core (with a wrapper around it), and all the scheme-specific building blocks. The first two are the most expensive in terms of both computation time and area requirements, and thus, they must be well optimized to make our unified cryptoprocessor compact and efficient.

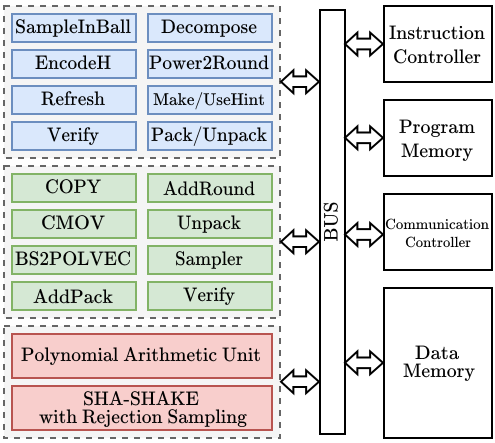


Figure 3.2: The high-level design of the cryptoprocessor with the Saber, Dilithium and common modules are in green, blue and red, respectively.

3.2.3 NTT-based unified polynomial multiplier

This section describes the design decisions we made for implementing the NTT-based polynomial multiplier architecture for Dilithium and Saber. Following the official reference code of Dilithium, we use the Cooley-Tukey (CT) and Gentleman-Sande (GS) butterfly configurations for the NTT and INTT, respectively. Figure 3.3 shows the internal blocks of the unified butterfly core. The proposed unified butterfly core can work with both CT and GS butterfly configurations. Both butterfly configurations perform operations using unsigned arithmetic in a unified butterfly core. The circuits are all pipelined to achieve high clock frequency.

Efficient modular reduction unit: If Dilithium's prime is used for Saber's NTT, then just having an efficient modular reduction circuit for the 23-bit prime will be sufficient. Otherwise, if Saber uses the 24-bit or 25-bit primes (that are different from Dilithium's prime) then a resource-shared modular reduction circuit will be required. As the primes have a similar pseudo-Mersenne structure, designing and implementing a unified modular reduction unit will have a very small area overhead compared to a single prime-specific reduction unit. We followed the add-shift-based modular reduction method [Yam+21a] and employed a similar fast modular reduction technique to create a unified modular reduction unit. Both Dilithium and Saber primes have the form of $q = 2^x - 2^y + 1$, which allows efficient modular reduction operation by using the property $2^x \equiv 2^y - 1 \pmod{q}$.

When the Dilithium prime is used for both schemes, a modular reduction unit for $2^{23} - 2^{13} + 1$ is implemented and used for both schemes. The property $2^{23} \equiv 2^{13} - 1 \pmod{q}$ is used recursively (i.e., $c[45:23] \cdot 2^{23} + c[22:0] = c[45:23] \cdot (2^{13} - 1) + c[22:0]$) to generate six partial results which are added efficiently

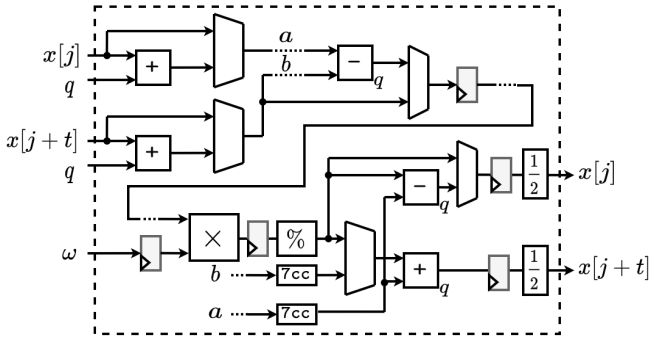


Figure 3.3: Internal architecture of the unified butterfly unit. The dotted routing lines for signals a and b are used to represent long interconnects

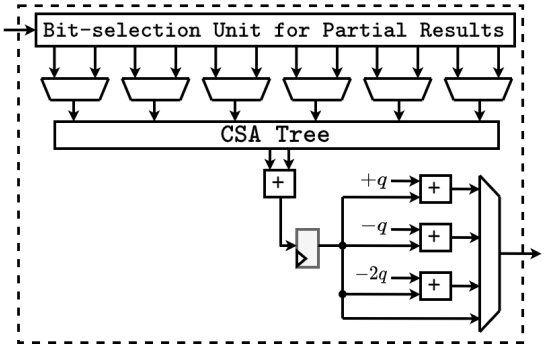


Figure 3.4: Unified modular reduction unit

using a carry save adder tree. Finally, a correction is performed to bring the result to the range $[0, q-1]$.

When Saber uses a 25-bit (or 24-bit) prime, a unified modular reduction unit for $2^{23} - 2^{13} + 1$ and $2^{25} - 2^{14} + 1$ (or $2^{24} - 2^{14} + 1$) is implemented. Since both primes have similar structures, six partial results are generated for each reduction. Then, partial results are selected based on the scheme in use and the same steps are followed as a modular reduction unit for the Dilithium prime. Figure 3.4 shows the high-level diagram of the unified modular reduction unit.

Memory arrangement: As one butterfly core consumes two coefficients and simultaneously produces two coefficients every cycle, we always keep two coefficients in a single memory word following [Roy+14]. This enables reading/writing two coefficients with just one memory read/write. Our NTT unit has two such butterfly cores in parallel to reduce the cycle count of NTT. To feed the two butterfly cores, we spread the coefficients into two BRAM sets. This spreading is necessary because a single BRAM set can only feed one butterfly core due to limitations in the number of read/write ports. In this way, a polynomial of

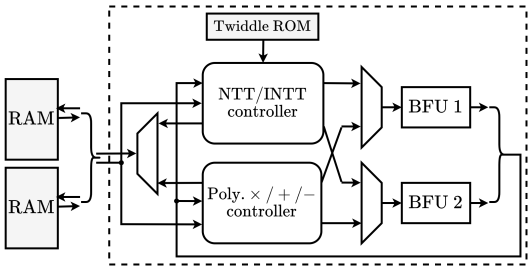


Figure 3.5: High-level architecture of polynomial arithmetic unit

256 coefficients occupies a total of 128 memory words, of which 64 are in the first BRAM set, and the remaining 64 are in the other BRAM set. Note that generally, efficient implementations propose storing the coefficients to be processed together, and this arrangement is maintained throughout the NTT/INTT iterations. However, since the coefficients are generated sequentially, we plan to utilize them in the same way instead of preprocessing them to use the strategies existing in the literature.

Figure 3.6 shows the arrangement of coefficients in memory words during NTT loop iterations using a toy example. During the NTT loops, the newly generated coefficients are written back in the BRAMs such that during the next iteration of the NTT loop, the required coefficients for each butterfly can be read as a pair from the memory. In the first iteration, the coefficients (shown in blue colour in Figure 3.6) zero and eight are input to the first butterfly unit, and the coefficients one and nine are input to the second butterfly unit. After the first iteration, we want the processed coefficients eight and nine to be stored in BRAM1, at the address where the coefficients four and five are currently stored. This will simplify the coefficient read pattern during the next iteration of the loop. Because coefficients four and five have not been processed yet, we cannot write new values to their memory locations. We solve this problem by changing the order in which the coefficients are processed. Hence, we read the coefficients four and five, and similarly twelve and thirteen, immediately after reading the coefficients zero and one, and eight and nine, respectively.

We add pipeline stages in between to avoid conflicts when writing coefficients eight and nine while reading coefficients four and five. Thus simplifying the control logic of the NTT and avoiding any read-write conflict. One NTT or

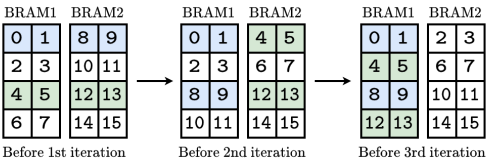


Figure 3.6: Coefficients in memory for 3 iterations of NTT for $n = 16$

INTT operation takes only 512 clock cycles. A high-level view of the complete polynomial arithmetic unit architecture is shown in Figure 3.5. For simplicity, we have designed two separate controllers for NTT/INTT and the remaining operations.

Post-processing elimination after INTT operation: At the end of the INTT operation, the resulting coefficients are scaled by $1/n$, which requires an extra n modular multiplications. In our design, this extra scaling is removed by performing division by two in $\pmod q$ for each butterfly operation output in the Gentleman-Sande configuration during INTT operation. The division by two in $\pmod q$ can be performed using only add and shift operations as shown in Equation 3.1 [Yam+21a]. This way, both the NTT and INTT implementations in our architecture are of the same cost and require no post-processing. This technique reduces the cost of INTT operation by 20% at the expense ofa slight increase in the hardware logic.

$$\frac{x}{2} \pmod q = (x \gg 1) + (x \;\&\; \texttt{0x1}) \times \frac{(q+1)}{2} \tag{3.1}$$

3.2.4 SHA3-256/512 and SHAKE-128/256 unit

Many PQC schemes, such as Dilithium and Saber, utilize Keccak-based operations, including the hash function SHA-3 and the pseudo-random number generator SHAKE. For implementing the Keccak-based hash and expandable output functions, we instantiate a single high-speed Keccak core in the proposed cryptoprocessor architecture. The implementation of the Keccak core is similar to the high-speed Keccak core available on the Keccak-team website [Tea19]. We use a wrapper module around the Keccak core to perform the parsing of input and output data bits. Additionally, the state buffer has been modified to enable the generation of pseudo-random polynomial coefficients in scheme-specific optimal representations, which can then be stored directly in the memory of the cryptoprocessor. This strategy helps reduce the overall cycle counts for both Dilithium and Saber.

The type of parsing varies depending on the operation mode. For example, the public polynomials of Saber are generated by directly picking 13-bit coefficients from the SHAKE-128 output. Whereas, the secret polynomials are generated by passing the SHAKE-128 output to a binomial sampler. Similarly, Dilithium's public and secret polynomials are generated using different types of rejection samplers. To explain the wrapper, we take Saber's public polynomial generation as a case study and later discuss how we use a similar methodology for different samplers.

Generating Saber's public polynomials: Saber's public polynomials, generated using SHAKE-128, have a 13-bit coefficient size. Before these polynomials are multiplied, they are converted into the NTT representation in our unified cryptoprocessor. As described in Section 3.2.3, the NTT unit requires its operand data to be present in 'two coefficients per BRAM word' format, for reading

and writing the coefficients efficiently. One option for processing the public polynomials is to generate a continuous bitstream in 64-bit words (the default output format of Keccak), then store the words in BRAMs, and later parse them into 13-bit coefficients using separate parser hardware. This approach is inherently sequential and results in a bloated cycle count and increased area consumption due to the required parsing buffers. To avoid such a redundant memory read/write step, we modify the output buffer of Keccak to directly produce a pair of 13-bit coefficients during the generation of the public matrix $\boldsymbol{A}$. However, this strategy requires a bookkeeping mechanism as the output length of a SHAKE-128 squeeze operation is 1,344 bits, which is not a multiple of 13. Therefore, after each squeeze of SHAKE-128, there will be leftover bits that must be prepended to the output string generated by the next SHAKE-128 squeeze operation. We observe that during the generation of $\boldsymbol{A}$ in Saber, the number of leftover bits is always an even number in $[0, 24]$. We use this observation to simplify the implementation of the Keccak-output buffer.

The prepending of the leftover bits to a newly generated SHAKE-128 squeeze output requires shifting and filling of the buffer bits. As the size of the Keccak output buffer (when operated as SHAKE-128) is 1,344 bits, which is quite large, we investigated efficient implementation techniques that reduce the area-overhead without affecting the cycle count. The naive method is to implement a simple multiplexer that updates the output buffer with 1,344 bits of the Keccak state and the leftover bits. But since there can be 13 (even numbers in [0, 24]) such possibilities, we will require a 13-to-1 multiplexer for assigning to a buffer of size 1,368 (=1344+24) bits. With this implementation option, there are 13 shift possibilities and as a consequence, the multiplexing overhead is ≈8000 LUTs, which is large. We aim to create a highly efficient and lightweight design for our hardware; therefore, we need a significantly better solution.

We proposed an efficient method for handling the remaining bits using a small 'left-over-bits buffer'. After the Keccak squeeze is done, we write the remaining bits to the leftover bits buffer. To avoid using a multiplexer for deciding on the number of remaining bits we need to pick, we just write the last 24 bits, which is the maximum possible number of remaining bits. To put these remaining bits into the Keccak output buffer, we first need to align the bits in the leftover-bits buffer to the left. In order to reduce the hardware cost, we decided to use only 'shift by two' or 'shift by four' operations for this alignment. Finally, the content of this left-over-bits buffer is concatenated at the beginning of the Keccak output buffer, using only 'shift by two' or 'shift by four' operations. An example for handling 18 leftover bits using the proposed left-over-bits buffer method is illustrated in Figure 3.7. Since we run Keccak in parallel with NTT, the extra cycle count for this bookkeeping does not account for an increase in the total cycle count.

Other sampling operations: Saber uses a binomial sampler for generating secret polynomials, which is implemented as a separate unit. Dilithium requires three different rejection sampling: uniform, η, and γ sampling. For the uniform and η sampling, we need to extract 24 and 4 bits from the Keccak output buffer, and we can utilize the Keccak output fully after every squeeze. On the other

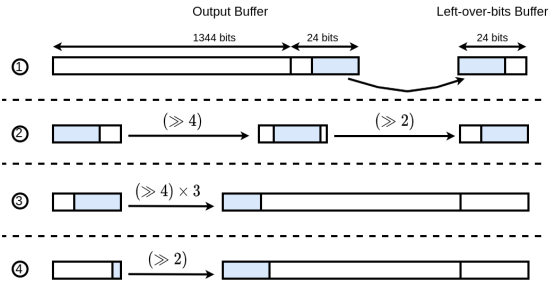


Figure 3.7: Example of bookkeeping with 18 remaining bits

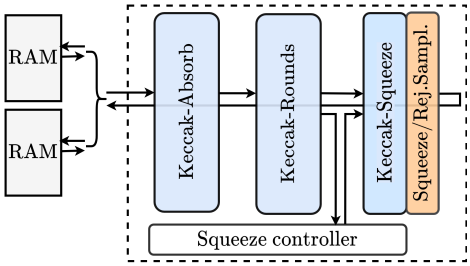


Figure 3.8: High level architecture of Keccak that supports all the modes of Keccak as well as samplers

hand, the γ sampling needs 18 or 20 bits from the Keccak output, and it does not utilize the Keccak output fully after every squeeze, with some leftover bits.

The same approach of shifting the output buffer and leftover bit buffer as described in the previous section can be used for γ sampling as well. However, this leads to a Keccak output buffer generating six different types of outputs: 4, 13, 18, 20, 24, and 64 bits. This can be controlled using a multiplexer, which in hardware is very expensive. To reduce costs, we use an intermediate buffer of size 192 bits (=lcm(4, 24, 64)) and utilize it for compressing the results for 4, 24, and 64 bits. The Keccak output buffer then outputs only four different types of outputs: 13, 18, 20, and 192 bits, thus saving around ≈1200 LUTs in FPGAs. A high-level view of the complete architecture of the SHA-SHAKE unit with samplers is shown in Figure 3.8. The datapaths for Keccak modes and samplers are combined into a single unit. This is then controlled by the squeeze controller.

3.2.5 Remaining scheme-specific building blocks

In the above sections, we discussed how we efficiently implement a common polynomial arithmetic unit for Dilithium and Saber. We also discussed the optimized implementation of other major area-consuming blocks. In this section, we discuss how we implement the remaining building blocks and various instructions that our cryptoprocessor provides to realize a flexible instruction-set cryptoprocessor.

Note that the output given after the polynomial multiplication now has two coefficients per word instead of 4 coefficients per word storage style used in [RB20]. This causes Saber building blocks to consume more clock cycles. However, we can avoid the extra clock cycles required for pre-processing the input for feeding into these modules. Architectures for the Saber-specific modules are as follows.

- Modules 'Unpack (Saber)', 'AddPack', and 'AddRound', provided in Table 3.2 Instruction Set-2, which can run in parallel with Keccak's SHA/SHAKE and all the other instructions provided in Instruction-Set 1.
- 'CMOV', 'Verify', and 'COPY' also have the ability to run in parallel with instructions in Set-1.
- Instruction 'Binomial Sampler' is provided in Set-1. Thus, while one polynomial is being generated, the previously generated polynomial can be transformed to the NTT domain in parallel.
- Instruction 'BS2POLVEC' converts byte-stream to vector form as required by the NTT module for one polynomial at a time. and can be run in parallel with Set-2 instructions.

Architectures for the Dilithium-specific modules are as follows.

- Module 'Pack-Unpack (Dilithium)', in Table 3.2 Instruction-Set 1, allow unpacking/packing of polynomials while performing polynomial arithmetic on already/to-be unpacked/packed polynomials, in parallel.
- Module 'SampleInBall' requires the memory to be refreshed with zeroes. For this 'Refresh Memory' instruction is provided in Set-1. Once the memory is refreshed, the SampleInBall instruction provided in Set-2 can be used.
- Instruction 'Encode_$\mathcal{H}$' provided in Set-2 is used to pack the polynomials, which are then fed to SHAKE-256 for hashing.
- Instructions 'Decompose' and 'Power2Round', in Set-1 and Set-2, are implemented as per the specification, consuming 128 clock cycles for processing one polynomial.
- 'MakeHint' and 'UseHint' modules require the Decompose function. Since this is an instruction-set processor, we utilize the existing Decompose module instead of creating duplicate implementations. We then add equality checkers to the MakeHint and UseHint modules to return the desired output.
- Instruction 'Counter_ref' is provided in Set-2, to ensure that the MakeHint counter for the hamming weight becomes zero if the loop exit conditions are not satisfied.
- Dilithium uses the same seed with different nonce values for generating polynomials. Naively, in hardware, we can make copies of seeds with different nonce values to generate different polynomials. However, this

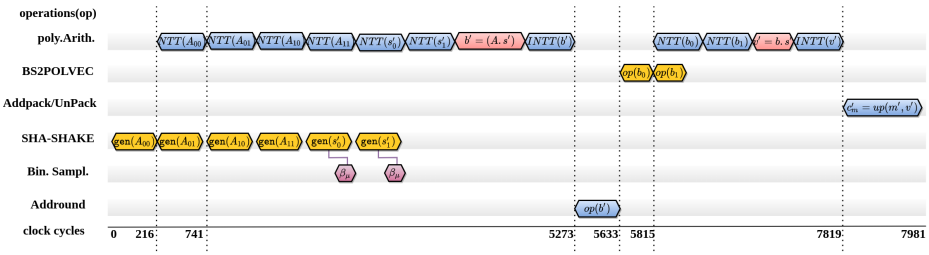


Figure 3.9: Operation scheduling of LightSaber PKE encryption

approach is inefficient and requires additional memory. We provide a 'Write instruction' in Set-1, which can be used to store a nonce of the user's choice after the seed. Thus, not only does it save memory but also allows the user flexibility to generate polynomials in any order.

- Instruction 'Verify (Dilithium)' in Set-1 is used to verify the conditions for a valid signature during the signing procedure. If the verification conditions are not met, it returns the instruction pointer to the instruction that marks the start of the *for* loop.

3.2.6 Parallel processing of instructions

We observe that both Dilithium and Saber protocols have data-independent instructions. Therefore, it is possible to speed up the two protocols by executing data-independent instructions in parallel (when possible). Such parallel processing of low-level operations has been explored in the works [Dan+20; Gaj20]. They design block-pipelined architectures (which differ from instruction-set architectures) and place the building blocks in such a way that several data-independent computation steps are overlapped (i.e., executed in parallel) using the dedicated blocks in the correct order. While an instruction-set-based cryptoprocessor offers significant flexibility over a block-pipelined architecture, overlapping data-independent 'instructions' becomes a relatively challenging task. Implementing the parallel processing of instructions requires a complicated synchronization and control mechanism (e.g., scheduling of instructions, detecting completions of concurrent instructions, initiating new instructions, managing the data memory, etc). In this work, we apply overlapping of data-independent computations in the context of an *instruction-set architecture* and execute data-independent Keccak-based and polynomial arithmetic-based operations in parallel. This strategy reduces the overall cycle count at the cost of a negligible area overhead.

When the instructions are run in parallel, they need to read/write data from memory simultaneously. Since a BRAM can only offer us one set of read/write ports, we needed to devise an efficient solution. To support the parallel execution of Keccak and polynomial arithmetic, we split the memory unit into four BRAM sets. While the NTT unit occupies read and write ports of any two BRAM sets, the Keccak unit works with the remaining two sets. This is discussed

Table 3.2: Instructions where an instruction from the first column can be run in parallel with an instruction from the second column.

	Instruction Set-1	Instruction Set-2
S+D	Reset Keccak	NTT
	SHA-256/512	INTT
	SHAKE-128/256	Coefficient-wise multiply
	SHAKE intermediate reset	Coefficient-wise add
	SHAKE resume	Coefficient-wise subtract
Saber	SHAKE-128 (writing 26 bits)	AddRound/AddPack
	BS2POLVEC	Unpack (Saber)
	Binomial Sampler	Verify/CMOV/COPY
Dilithium	SHAKE-128 rej. $[0, q-1]$	SampleInBall
	SHAKE-256 rej. [-η, η]	Encode_$\mathcal{H}$
	SHAKE-256 rej. [-(γ - 1), γ - 1]	Power2Round
	Pack-Unpack (Dilithium)	MakeHint
	Decompose/Verify (Dilithium)	UseHint
	Write Inst./Refresh Memory	Counter_ref

in detail later in Section 3.2.7. We also add a program controller unit that loads all instructions into an instruction RAM and then sends them one by one to the compute core for processing in parallel or sequence, as specified in the instructions. The two types of instructions are stored together, along with four control bits, in the Instruction RAM. Figure 3.9 shows the instruction flow for LightSaber.`PKE.Enc()`. We can see the instructions in Set-1 working in parallel with instructions in Set-2 given in Table 3.2. We achieve a reduction of 10%, 13%, and 15% in cycle count during decapsulation for LightSaber, Saber, and FireSaber, respectively. For Dilithium signature generation, we reduce the cycle count by 20%, 25%, and 28% for Dilithium 2, 3, and 5, respectively.

3.2.7 Organization of data memory

We need to ensure that our cryptoprocessor has sufficient memory to support all the variants of both Dilithium and Saber. For this, we consider the highest security variants as they require matrices and vectors of the highest dimensions. The signature generation function in Dilithium and the decapsulation function in Saber are the most time-consuming functions and also require the highest amount of data storage. For the Dilithium variant with the NIST security level 5, the public matrix consists of 56 polynomials. During the signing operation, we need to precompute and store the secret key, which consists of 23 polynomials in the NTT domain. FireSaber has much smaller requirements as its public matrix and secret vector comprise only 16 and 4 polynomials, respectively. Thus, Dilithium-5 determines the overall memory requirement of the cryptoprocessor.

Storing the entire public matrix in the memory makes the signing operation faster, as it is used in a loop several times. If we pre-compute and store all the polynomials, we require storage for 79 polynomials before the signing loop starts, along with seeds and hash values. Also, we need to store intermediate results during the signing operation, which further increases the memory requirement.

In this Chapter, we use the ability of our cryptoprocessor to process data-independent instructions in parallel. Instead of generating and storing the public matrix all at once, we generate it on the fly in parallel with the polynomial multiplication operation, thus reducing the memory requirement by more than half without compromising performance.

As we discussed in the previous section, our data memory length is 64 bits, and therefore, we store 2 coefficients per word of data memory. With all the constraints in consideration and flexibility requirements in place, the implementation of Saber requires only four BRAM36K units while Dilithium requires 20 BRAM36K. The proposed cryptoprocessor uses parallel memory organization to ensure efficient load and storage of polynomials. This is especially important for the parallel execution of NTT and Keccak operations. To that end, the memory was split across four major blocks, with each of them having five BRAM36K elements, which enables the parallel execution of NTT and Keccak. The constants for NTT and inverse NTT are kept in a ROM, which is also interpreted using one BRAM in our implementation. Additionally, the program controller, which is used to load all instructions simultaneously and then handle all data-independent executions in parallel, requires three BRAM36K.

3.3 The KaLi Unified Hardware Architecture

The primary goal is to unify the digital signature scheme and the key-encapsulation scheme. While doing this, it is important to ensure that the design is compact and flexible. Unification has a very straightforward three-step approach. First, we must identify the most area and time-consuming building blocks, the Giants. This is because unifying the low area and time-consuming building blocks (the Dwarves) will not significantly reduce area consumption, but instead limit the design's flexibility. The next step is to find the algorithmic synergies between the Giants of the two schemes. The final step is to discern if some of the Dwarves which are dependent on the Giants can be unified with Giants to reduce both area and time consumption.

A high-level view of the proposed architecture KaLi is given in Figure 3.10. The Keccak-based SHA-SHAKE unit and polynomial arithmetic unit are the two Giants in both schemes. The remaining building blocks are deemed as Dwarves since they comprise only 20% of the total area consumption. Unifying the Keccak-based SHA-SHAKE unit is relatively straightforward, as we can utilize a common Keccak core for both schemes. Therefore, in this section, we will discuss how we efficiently unified the polynomial arithmetic unit. We will also discuss how we efficiently manage the memory for the two schemes. Another facet of the work, optimization for ASIC platforms, is also presented. We utilized multiple clock domains and boosted the memory bandwidth budget on ASIC platforms to reduce area consumption.

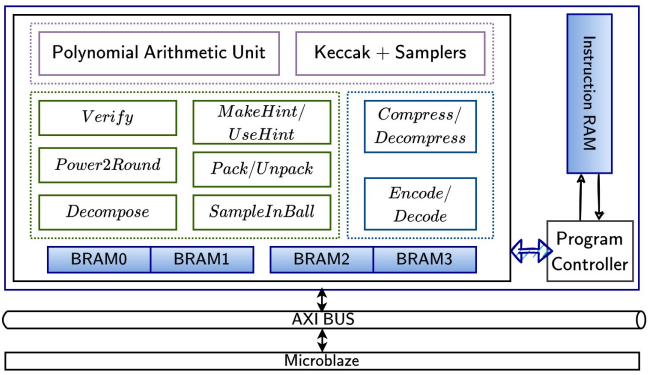


Figure 3.10: High-level architecture of `KaLi`

3.3.1 The colossal Giant: Polynomial Arithmetic Unit

The Polynomial Arithmetic Unit performs polynomial addition, subtraction, and multiplication. Polynomial addition and subtraction are simple coefficient-wise operations; hence, they are inexpensive. Polynomial multiplication is rather complex, and it is what makes the polynomial arithmetic unit a Giant. Both schemes perform this using NTT. Although the two schemes use NTT-based polynomial multiplication units, there are many differences that make their NTT units quite distinct.

A clash of the Giants: The differences between the NTTs

The first distinction between the NTTs used by the two schemes lies in the algorithm itself. The NTT-based polynomial multiplication method used in Dilithium requires the existence of $2n$-th root of unity that mandates $q \equiv 1 \pmod{2n}$. Accordingly, Dilithium uses a *complete-NTT*. After a *complete-NTT* transform of an n-degree polynomial, we get n polynomials of degree 0. In [LS19], Lyubashevsky *et al.* propose a new method for NTT-based polynomial multiplication that requires only $q \equiv 1 \pmod{n}$, without pre-processing and post-processing operations. This technique is adopted by Kyber, and their 12-bit prime modulus does not have a $2n$-th root of unity. Therefore, Kyber has to use an *incomplete-NTT*. An *incomplete-NTT* gives us $n/2$ polynomials of degree 1. These polynomials cannot be multiplied coefficient-wise.

For the *incomplete-INTT*, multiplication operation of two degree-1 polynomials is performed in the ring $\mathbb{Z}_q[x]/(x^2 - \omega^i)$ where ω is the n-th root of unity and i depends on the index of coefficients. For the details, readers may follow original Kyber specifications [Sch+21] or related prior works in the literature [Yam+21b]. Additionally, they have a difference in datapath design. Dilithium has a 23-bit prime modulus, while Kyber has a 12-bit prime modulus. Therefore, while Kyber requires 12-bit adder/subtracter/multiplier units, Dilithium requires them in 23 bits. Designing a datapath for one of them and using it for the other one would

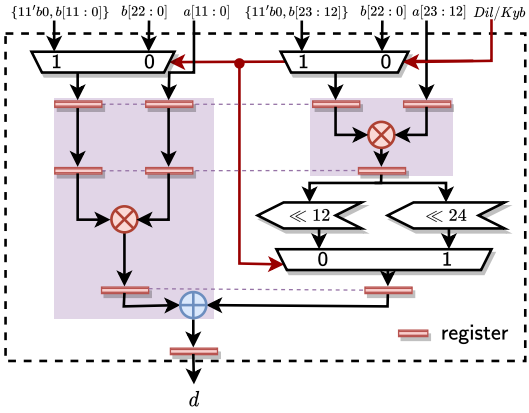


Figure 3.11: Flexible yet compact integer multiplier. The red/black lines show the control/data path. The DSP units used are highlighted in purple.

lead to over- or undersaturation.

Next, we will discuss in detail how we achieved a unified polynomial multiplication unit with full utilization. The unit of interest here is the butterfly unit (BFU). Each BFU performs dyadic addition, subtraction, and multiplication on the two input coefficients. The results are reduced by modulo q. This is shown by steps 7-10 in Algorithm 7. Since modulus multiplication is the most expensive operation, we will discuss how we unify this unit. Then, we will discuss how, with a few more changes, the entire BFU can be consolidated.

Flexible fusion of Modular Multiplier Unit

As discussed above, if we naively use the 23-bit Dilithium polynomial multiplier unit for Kyber, then it will always be undersaturated, as half of it will be unused. Instead, if we aim to use a 12-bit Kyber unit for Dilithium, it will require extra control logic but also slow down Dilithium's NTT. Therefore, we need to find a solution using a 23-bit Dilithium unit that does not lead to undersaturation. The modular multiplier unit has two parts: (i) integer multiplier and (ii) modular reduction unit. We propose an algorithm (Algorithm 8) to make the integer multiplication unit designed for Dilithium flexible for Kyber. It performs two 23-bit$\times$12-bit integer multiplications. The result is added for Dilithium and concatenated for Kyber. This algorithm gives us one multiplied coefficient in the case of Dilithium and two multiplied coefficients in the case of Kyber.

The modular multiplier unit, designed to support modular multiplication using both primes, uses two DSP units of Xilinx FPGAs. The hardware architecture of the re-configurable integer multiplier is shown in Figure 3.11. The datapath depends on the scheme type and is heavily pipelined. We used internal registers of DSP units to synchronize two DSP unit outputs and achieve a high clock frequency. Now we need to design a modular reduction unit accordingly.

Algorithm 8 Integer Multiplication Algorithm

Require: $a, b \in \mathbb{Z}_{8380417}$ or $a[23:12], b[23:12], a[11:0], b[11:0] \in \mathbb{Z}_{3329}$
Require: $sel \in \{0, 1\}$ (0 for Dilithium and 1 for Kyber)
Ensure: $d = a \cdot b$ or $d = \{a[23:12] \cdot b[23:12], a[11:0] \cdot b[11:0]\}$
1: $d_0 = (sel)\ ?\ b[23:12]\ :\ b$
2: $m_0 = d_0 \cdot a[23:12]$
3: $m_1 = (sel)\ ?\ (m_0 \ll 24)\ :\ (m_0 \ll 12)$
4: $d_1 = (sel)\ ?\ b[11:0]\ :\ b$
5: $d = d_1 \cdot a[11:0] + m1$
6: **return** d

Versatile Modular Reduction Unit

The naive solution is to design separate modular reduction units for the two primes. It would require one modular reduction unit for the Dilithium prime and two reduction units for the Kyber prime, resulting in additional hardware costs. To avoid this, we propose a unified modular reduction unit. Both Dilithium ($2^{23}-2^{13}+1$) and Kyber ($2^{12}-2^{9}-2^{8}+1$) primes have pseudo-Mersenne structure. For Dilithium prime, we followed the method described in [Aik+23c] which uses $2^{23} \equiv 2^{13} - 1$ equation recursively. Using this equation, we can reduce a 46-bit integer $d \pmod{2^{23} - 2^{13} + 1}$ to the integer $2^{13}d[45:24] + d[22:0] - d[45:23]$ which consists of addition/subtraction of 36-bit and 23-bit partial results. If we apply this operation recursively, we will obtain $\overline{d}$ $(= d[22:0] + (d[32:23] + d[42:33] + d[45:43])2^{13} - d[45:23] - d[45:33] - d[45:43])$. Similarly, for Kyber, we followed add-shift-based method proposed in [Yam+21b] which generates partial results using equations $2^{12} \equiv 2^9 + 2^8 - 1$ and $2^{11} \equiv -2^{10} - 2^8 - 1$ recursively.

Summing all partial results using carry propagate adders (CPAs) will result in either a very long carry chain or multiple pipeline stages. In order to avoid long carry chain and pipeline delays, we used carry-save adders (CSAs) along with CPA. The proposed unified modular reduction unit is shown in Figure 3.12, where the boxes 'D' and 'K' represent the partial result generation circuits for Dilithium and Kyber primes, respectively. All the subtraction operations are converted into additions by taking the 2's complements of partial results. In Figure 3.12, each number inside a box represents a bit index of the input integer (0 to 45 for Dilithium and 0 to 22 for Kyber). The white and brown (terracotta) boxes represent the normal and negated bits. When the 2's complement operation is applied to the partial results, extra plus ones, along with sign extensions, come into the picture. These are represented with blue circles.

After adding all of these partial results, we also perform a final correction, which brings the resulting integer from the range $(-q, 3q)$ to the range $[0, q)$. The proposed modular reduction unit can either perform one reduction for the Dilithium prime or two reductions for the Kyber prime. The latency of the modular reduction unit is two cycles, and it is fully pipelined.

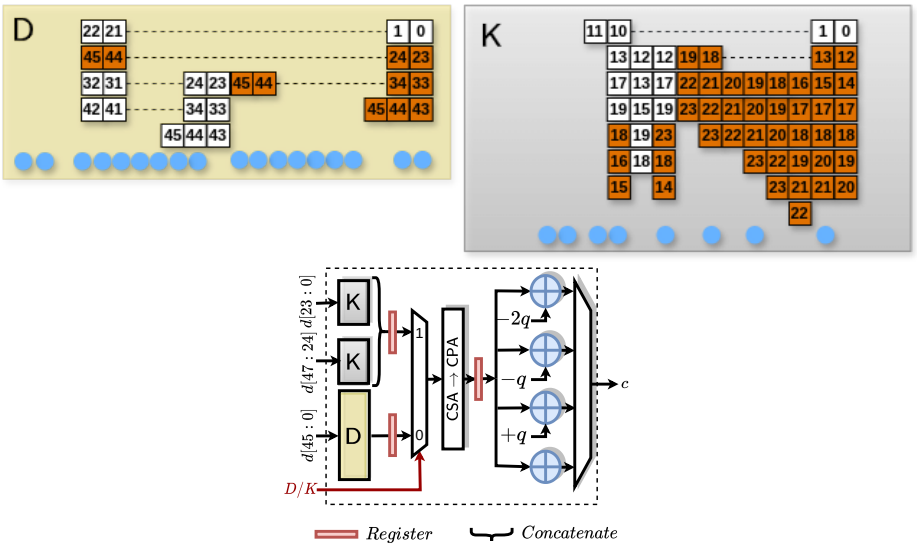


Figure 3.12: Unified modular reduction unit for Dilithium and Kyber primes.

Coalesced datapath for the Butterfly Unit

Now that we have unified the modular multiplication unit, we propose a unified BFU (Figure 3.13). It can perform one butterfly operation for Dilithium and two butterfly operations for Kyber using the same datapath. All the arithmetic units are made re-configurable to work for both schemes. New re-configurable adder and subtractor units are shown in Figure 3.14. The idea is to divide each 24-bit adder/subtractor into two small 12-bit parts and select proper input signals based on the scheme. The complete unified butterfly unit, designed using the re-configurable arithmetic units, is shown in Figure 3.13.

The schoolbook multiplication for Kyber requires five multiplications to multiply two linear (i.e., degree one) polynomials. We have two flexible butterfly units that function as four butterfly units for Kyber, allowing for four multiplications only. One way to perform these five multiplications is to add another DSP multiplier just for the extra multiplication. This unit will not be useful for any other operation. To avoid this extra multiplier, we condense five multiplications into four using Karatsuba-like reduction. Then we use these four independent butterfly units as a pair. The output of the first set is the input to the second set, as shown in Figure 3.15. The inputs and outputs for the BFUs are highlighted in blue. The control flow here is separated from the Dilithium polynomial multiplication control flow for simplicity. The entire flow is pipelined to achieve a high clock frequency.

The coefficient consumption during NTT/INTT is shown in Figure 3.16. Due to the flexible datapath and efficient memory arrangement (discussed in the next subsection), the Kyber NTT coefficients can be consumed more efficiently. This

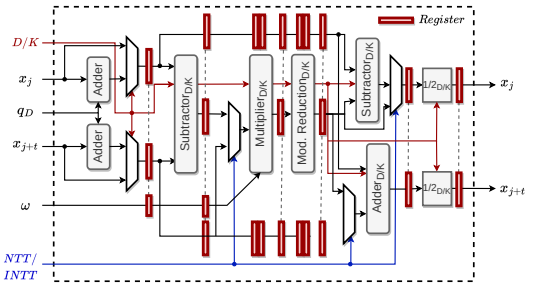


Figure 3.13: Compact butterfly unit (BFU) with flexibility for both Dilithium and Kyber. The red-and-blue/black lines show the control/data path.

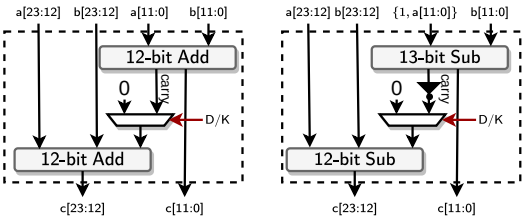


Figure 3.14: Unified adder and subtractor for the butterfly unit. The red lines show control signals, and the black lines show data movement.

enables full utilization of the datapath. The complete polynomial arithmetic unit consumes 3,487 LUTs, 1,918 FFs, 4 DSPs, and 1 BRAM. The BRAM is used to store the powers of roots of unity (twiddle factors) required during NTT/INTT operation. In the next section, we will discuss the efficient memory arrangement designed to optimally feed the polynomial arithmetic unit.

3.3.2 Memory Arrangement

The polynomial arithmetic unit is designed to consume the Kyber coefficients twice as fast as the Dilithium coefficients. It requires that the memory unit feed it at the same rate; otherwise, making these unifications will not help improve the performance. Dilithium coefficients are 23-bit, and we have designed the NTT/INTT unit using two butterfly cores. Each core requires exclusive access to the read/write port of a memory. Therefore, we split the memory into two blocks, each storing two Dilithium coefficients per address. For Kyber, each memory block stores four 12-bit Kyber coefficients. Figure 3.17 shows an example of the coefficients storage during Kyber's incomplete-NTT iterations for a 16-coefficient polynomial. One Dilithium polynomial coefficient will occupy two of these coefficients. It also ensures that the two required coefficients during NTT/INTT are always stored across different BRAMs.

Next, we will discuss how we used multiple clock domains to reduce area consumption in ASIC platforms.

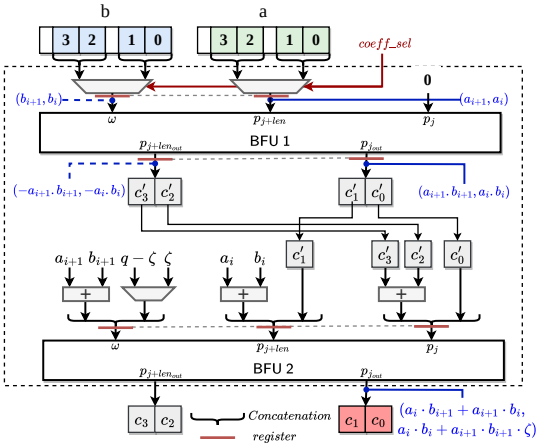


Figure 3.15: Butterfly feedback unit for Kyber's polynomial multiplication.

3.3.3 Multi-clock domains for ASIC platforms

The memory organization discussed above has two sets of BRAMs to feed the two BUFs. These BRAMs are also used by all the remaining building blocks. It is generated using dual-port BRAMs in an FPGA. In ASICs, dual-port RAMs consume more area than single-port RAMs. Therefore, to reduce the area consumption, we decided to replace dual-port RAMs with single-port RAMs, which work at a clock frequency twice as fast as the rest of the design. Using two different sources for the two clocks leads to an asynchronous setting. This creates meta-stability problems due to clock-domain crossing. To avoid these problems, we decided to keep the clocking synchronous and generate the slow clock (clock_logic) using the fast clock (clock_mem).

Figure 3.18 describes the handshake between memory and logic. A wrapper is provided to process the simultaneous reads and writes to the memories. The read operation is given preference over the write operation to ensure data validity when the building blocks fetch it, thereby avoiding any issues due to clock glitches. The read latency is three clock cycles, and all the building blocks are tailored accordingly. This design helps reduce the area for ASIC designs. Note that a similar modification will not change the FPGA area consumption and instead cause timing problems when running the memory at a high clock frequency. Therefore, this adaptation specifically targets ASIC platforms.

To date, we have discussed the major contributions of this work. Next, we will briefly discuss how we efficiently implement the remaining building block. We will start with the rejection samplers used in both schemes. These are the Giant dependent Dwarves that might help reduce the area and time consumption without compromising the flexibility of the design.

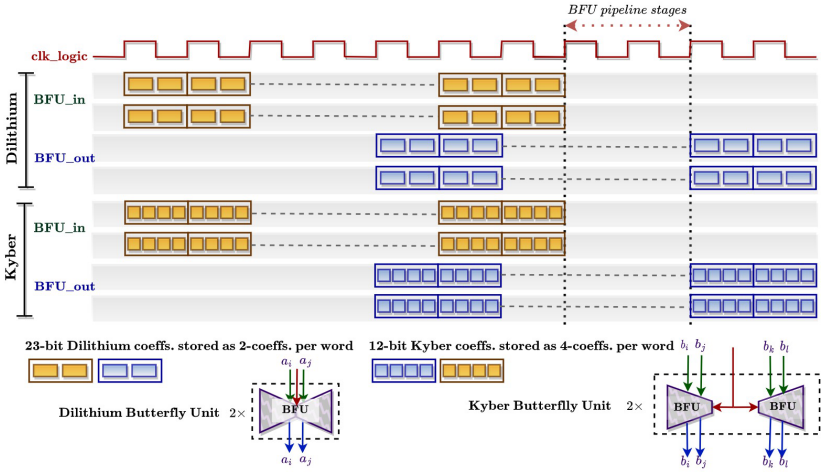


Figure 3.16: Timeline showing the unified butterfly unit processing Dilithium and Kyber coefficients.

3.3.4 The Giant and the Dwarves: Keccak-based SHA-SHAKE unit and the rejection samplers

Dilithium requires SHAKE-128 and SHAKE-256 for pseudo-random number generation and hashing. Kyber requires SHA-3-256 and SHA-3-512 for hashing, and SHAKE-128 and SHAKE-256 for KDF and pseudo-random number generation. These different Keccak-based functions are implemented as modes of the same Keccak output. Therefore, we can use the same Keccak instance for all these modes. Both schemes employ different sampling for the generation of secret and error polynomials. While some of these fully consume the Keccak output, the remaining have to keep track of the leftover bits.

We combine the rejection sampler with the Keccak unit using a book-keeping approach similar to [Aik+23c]. It improves the performance of the sampling operation, as we do not need to store and then read the Keccak output in between. The base implementation of Keccak follows a high-speed and parallel design. The control and datapath are modified to work with rejection samplers, as they depend on coefficients passing the rejection constraints. The complete Keccak unit consumes 12,326 LUTs and 3,560 FFs.

We have unified all the Giants, so now we will discuss the optimized implementation techniques for the Dwarves.

3.3.5 Optimizations for the Dwarves

Making a design compact while keeping it agile increases the life and usability of `KaLi` on the FPGA and ASIC platforms. However, this comes with a series of challenges. We must ensure that, to keep the design agile/flexible, we do not pay

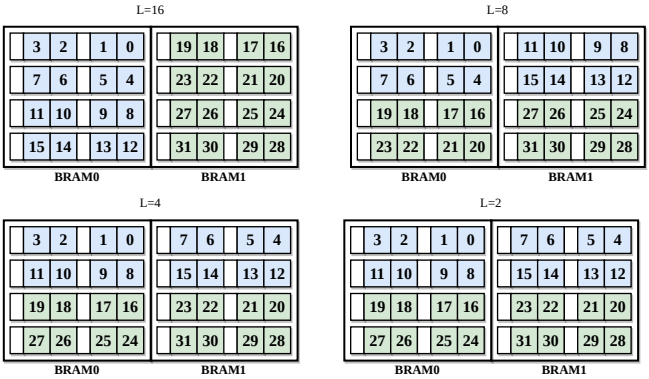


Figure 3.17: Storage of coefficients during Kyber's NTT for 16-coefficient polynomial

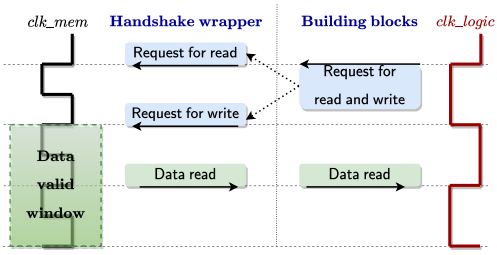


Figure 3.18: The data read-and-write handshake between memory and logic unit

a huge price in terms of area. Similarly, while making the design compact, the performance should not get worse. We now discuss how to make certain building blocks of the two schemes compact, while maintaining flexibility.

Compress/Decompress Unit

The decompress unit performs division by power-of-two and rounding operation. On the other hand, the compress operation requires division by q and rounding. Some works in the literature use Barrett reduction and division algorithms to perform the compress operation. We decide to use sufficient precision and convert the division by q operation into multiplication and shift operations. The proposed multiplication-based compression algorithm is shown in Algorithm 9. The input consists of the Kyber coefficient x and the type of compression required, denoted by d. The compressed coefficient y is returned as the output.

Since the multiplications are by constant values, we implement these operations using the add and shift technique, utilizing the LUTs. Figure 3.19 shows the hardware architecture of this multiplication unit, used for retrieving the t values in Algorithm 9. This is unified and works for both compress and decompress operations. The control flow is dependent on the type of compression or decompression required.

Algorithm 9 The Proposed Compression Algorithm

Require: $x \in \mathbb{Z}_{3329}$, $d \in \{1, 4, 5, 10, 11\}$
Ensure: $y = \lceil (2^d/q) \cdot x \rfloor$
switch (d)
  case 1**:**
    $t = (10079 \cdot x); y = (t \gg 24) + (t[23] \gg 23)$
  case 4**:**
    $t = (315 \cdot x); y = (t \gg 16) + (t[15] \gg 15)$
  case 5**:**
    $t = (630 \cdot x); y = (t \gg 16) + (t[15] \gg 15)$
  case 10**:**
    $t = (5160669 \cdot x); y = (t \gg 24) + (t[23] \gg 23)$
  case 11**:**
    $t = (10321339 \cdot x); y = (t \gg 24) + (t[23] \gg 23)$
end switch
return $y \pmod{2^d}$

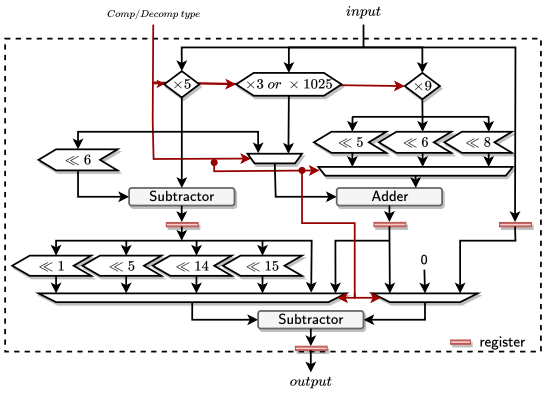


Figure 3.19: Architecture of the Compress/Decompress unit. The red lines show control signals, and the black lines show data movement.

Encode/Decode Unit

Encode and decode units perform coefficient-to-byte and byte-to-coefficient conversions. We employed a similar approach as proposed in [XL21b], which utilizes a 32-bit interface. Our architecture utilizes a 64-bit interface, and therefore, the proposed encode unit employs a 104-bit buffer. It can encode coefficients of 1-bit, 4-bit, 5-bit, 10-bit, and 11-bit lengths. The decode unit can decode 64-bit inputs into 1-bit, 4-bit, 5-bit, 10-bit, and 11-bit long coefficients using a 72-bit buffer.

Pack/Unpack unit

Similar to Kyber, Dilithium requires coefficient-to-byte and byte-to-coefficient conversions for various coefficient sizes. Pack and unpack units perform these conversions for all security levels of Dilithium for coefficient sizes 3, 4, 6, 10, 13, 18, and 20 bits. We again followed the idea proposed in [XL21b] for the pack

and unpack units. The remaining blocks of both schemes differ, and unifying them would not save any area; instead, it would complicate the control logic and reduce the design's flexibility. These building blocks do not require any DSP units and consist of simple bitwise packing, unpacking, or addition/subtraction operations. They are implemented as individual blocks and occupy only 18% of the cryptoprocessor's area.

3.3.6 Instruction set cryptoprocessor

We made the building blocks compact while ensuring flexibility, but this is insufficient. What happens if, in two years, the Keccak pseudo-random number generation and hashing unit are obsolete? Do we then need to redesign the entire cryptoprocessor? To counter this, we design an instruction set architecture (ISA) where each instruction serves as a building block required by the cryptographic schemes. A simple program controller executes the necessary instructions to run the cryptographic protocols, manages the synchronization of parallel instructions, and avoids back-and-forth communication between the CPU and the Cryptoprocessor. Note that the program controller is not a 'control processor', and it does not contain arithmetic circuits to process operand data. It simply decodes an instruction and then activates the corresponding module inside the cryptoprocessor. It consumes only 8% of the cryptoprocessor's area. The instructions and the corresponding hardware modules are listed in Table 3.7.

3.3.7 Running the Giants and the Dwarves in parallel

Our goal was to create a unified design that is both compact and agile. However, does this mean we have to pay an equal price in terms of performance? To some extent, this is correct. However, we should continue to ponder methods that could boost performance without increasing area consumption. One such way is to run the Giant instructions in parallel to each other or to multiple Dwarf instructions, as shown in [Aik+23c]. We make sure that two Giants, the Keccak unit and the polynomial arithmetic unit, can always run in parallel to cancel each other's run-time. It results in a 35% reduction in total runtime. Similar to [Aik+23c], we define two instruction sets (S-1 and S-2), as shown in Table 3.7. Every instruction belonging to the first set can be run in parallel with any instruction that belongs to the second set. The instruction opcodes for each instruction are shown in the INS column of Table 3.7. Following the design methodology, we design the unified cryptoprocessor- `KaLi` as shown in Figure 3.10.

3.4 Results

3.4.1 Results for Saber and Dilithium Cryptoprocessor

We present three different versions of cryptoprocessor implementation: (*i*) the first implementation uses 23-bit Dilithium prime $2^{23} - 2^{13} + 1$ for both Dilithium and Saber's NTTs, (*ii*) the second implementation uses 23-bit Dilithium prime

Table 3.3: Area of the cryptoprocessor on the Zynq Ultrascale+ ZCU102 FPGA platform. All security levels of Dilithium and Saber are supported.

	Unit	LUTs	FFs	DSPs	BRAMs
	ComputeCore	**17,164**	**9,027**	**4**	**21**
Saber	⌊AddPack	398	548	0	0
Saber	⌊AddRound	359	362	0	0
Saber	⌊BS2POLVEC	340	360	0	0
Saber	⌊Unpack (Saber)	393	406	0	0
Saber	⌊Verify (Saber)	103	208	0	0
Saber	⌊CMOV	12	34	0	0
Saber	⌊COPY	28	98	0	0
Saber	⌊Sampler	934	710	0	0
Dilithium	⌊Decompose	548	286	0	0
Dilithium	⌊Power2Round	160	64	0	0
Dilithium	⌊MakeHint	245	119	0	0
Dilithium	⌊UseHint	609	393	0	0
Dilithium	⌊Encode_$\mathcal{H}$	122	231	0	0
Dilithium	⌊Pack (Dilithium)	740	151	0	0
Dilithium	⌊Unpack (Dilithium)	296	156	0	0
Dilithium	⌊SampleInBall	485	260	0	0
Dilithium	⌊Refresh	4	7	0	0
Dilithium	⌊Verify (Dilithium)	28	69	0	0
S+D	⌊Memory	268	12	0	20
S+D	⌊Keccak+Sampler	8,738	3,482	0	0
S+D	⌊ Multiplier^a	2,341	1,048	4	1
S+D	⌊ Multiplier^b	2,347	1,059	4	1
S+D	⌊ Multiplier^c	2,257	1,079	4	1
	ProgramController	**1,371**	**260**	**0**	**3**
	Total	**18,494**	**9,319**	**4**	**24**

^{a}:for 23-bit implementation; ^{b}:for 24-bit implementation (used for total computation); ^{c}:for 25-bit implementation.

and 24-bit prime $2^{24} - 2^{14} + 1$ for Dilithium and Saber's NTTs respectively, and (*iii*) the third implementation uses 23-bit Dilithium prime and 25-bit prime $2^{25} - 2^{14} + 1$ for Dilithium and Saber's NTTs respectively. They provide us with comparative area consumption for the three options. The first, second, and third implementations are referred to as 23-bit, 24-bit, and 25-bit implementations, respectively. The 23-bit and 24-bit implementations ensure negligible-error probability for multiplications in Saber, while the 25-bit implementation performs error-free multiplication for Saber. All three implementations support all the subroutines and all the security levels of Dilithium and Saber using an instruction-set architecture framework.

The proposed unified cryptoprocessor architecture is described entirely in Verilog, and it is implemented for FPGA and ASIC platforms. For FPGA, the 23-bit, 24-bit, and 25-bit implementations are synthesized and implemented using Vivado 2019.1 for the target platform Zynq Ultrascale+ ZCU102 with an area-optimized implementation strategy. The FPGA implementations achieve a clock frequency of 200 MHz. In Table 3.3, we present the resource utilization of the proposed cryptoprocessor and its main building blocks. While the multiplier

Table 3.4: Performance results for Saber-KEM and Dilithium in FPGA

Operation	LightSaber Dilithium-2		Saber Dilithium-3		FireSaber Dilithium-5	
	Cycle	Lat.*	Cycle	Lat.*	Cycle	Lat.*
Sab.Keygen	5,935	29.6	10,980	54.9	17,523	87.6
Sab.Encaps	8,081	40.4	13,941	69.7	21,603	108.0
Sab.Decaps	11,678	58.3	18,991	94.9	27,890	139.4
Dil.Gen	14,183	70.9	22,957	114.7	38,841	194.2
Dil.Sign$_{pre}$	7,554	37.7	9,273	46.3	12,448	62.2
Dil.Sign	21,115	105.5	35,865	179.3	52,955	264.7
Dil.Sign$_{post}$	1,689	8.4	2,280	11.4	3,057	15.2
Dil.Verify	15,044	75.2	25,535	127.6	45,789	228.9

*: Latency for FPGA implementation in μs.

and Keccak (with sampler) blocks are able to achieve a maximum frequency of 260MHz, the remaining building blocks can operate at higher frequency. The implementations with 23-bit/24-bit/25-bit prime use 18,490/18,494/18,406 LUTs (6.9%), 9,308/9,319/9,323 DFFs (1.7%), 4 DSPs (0.1%) and 24 BRAMs (2.6%). The number of BRAMs in our cryptoprocessor is determined by the memory requirement of Dilithium since it is significantly more memory-consuming than Saber. The Keccak and multiplier units together consume more than half of the overall area. For ASIC, the 24-bit implementation is synthesized with UMC 65nm library with low leakage (LL) process device option using Cadence tool. For the ASIC implementation, DSP blocks used for integer multiplications in FPGA are replaced with ASIC multipliers. The rest of the arithmetic logic remains the same. The ASIC implementation synthesis achieves a 400 MHz clock frequency with an area of 0.317 mm^2 (≈220 kGE), excluding on-chip memory.

Table 3.4 presents the cycle count and latency (in μs) for the operations of Dilithium and Saber for different security levels in the FPGA. With a 200 MHz clock frequency in the FPGA, the CCA-secure key generation, encapsulation, and decapsulation operations for Saber take 54.9, 69.7, and 94.9 μs, respectively. We divide Dilithium's signature generation into three parts, pre-sign, sign, and post-sign, and report their performances separately. For a signature generation, the pre-sign and post-sign parts are performed only once, while the sign part is repeated until a valid signature is generated. We report the performance for the *best-case scenario* where the valid signature is generated after the first loop iteration. The key generation, signing and verification operations for Dilithium-3 take 114.7, 237, and 127.6 μs, respectively, in the FPGA. In ASIC, the speed of the cryptoprocessor improves by a factor of 2 as the clock frequency increases by a factor of 2.

3.4.2 Comparison with the existing results

We provide comparisons of our cryptoprocessor with related works in the literature in terms of area, performance, and flexibility for Dilithium-3 and Saber as shown in Table 3.5 and Table 3.6, respectively. A graphical representation of the performance and resource consumption is given in Figure 3.20. For simplicity

and better understanding, we divide the comparisons into three parts: (*i*) comparisons with other designs supporting multiple schemes, (*ii*) comparisons with standalone implementations of Dilithium, and (*iii*) comparisons with standalone implementations of Saber.

Comparisons with unified architectures: We note that only a few works target unified architectures that support multiple PQC schemes [BUC19a; DMG21; Fri+21b; FSS20b]. Their area and performance results, along with our architecture, are presented in Table 3.6. In [BUC19a], the authors present *Sapphire*, a cryptoprocessor coupled with a RISC-V processor implemented in an ASIC for various lattice-based Round 2 schemes in NIST's PQC standardization. It does not support Saber, while the results provided for Dilithium use the outdated Round-2 specifications. In [FSS20b], the authors present a RISC-V architecture tightly coupled with a hardware accelerator for Crystals-Kyber, NewHope, and Saber. Compared to the Saber implementation in [FSS20b], our FPGA and ASIC implementations show up to 304× and 564× better performances, respectively. The work in [Fri+21b] presents a HW/SW co-design of Crystals-Kyber and Saber. Our implementation demonstrates superior performance in terms of both speed and area, as it targets an entirely hardware-based solution. The authors in [DMG21] present high-speed design results for Kyber, NTRU, and Saber. Our unified architecture consumes much less area compared to the Saber results.

Comparisons with Dilithium-only implementations: There are only a few FPGA-based implementations of Dilithium [BNG21; LSG21; Ric+21; Zha+22a; Zho+21] in the literature. Their area and performance results, along with our work, are presented in Table 3.5. Similar to 'Saber only architecture', these architectures have an obvious limitation of providing support for only digital signature and not PKE/KEM. Zhou *et al.* [Zho+21] propose a HW/SW co-design by offloading computationally intensive operations such as SHA3/SHAKE and polynomial multiplication to the hardware while keeping the rest of the operation in the software. Their implementation has a small area as they implement only a few building blocks in hardware. Our pure-hardware solution demonstrates almost two orders of magnitude better performance compared to their HW/SW co-design solution.

In [Ric+21], the authors present three high-performance architectures for Dilithium targeting an FPGA. Their implementations can perform key generation, signature generation, and verification operations in 51.9, 63.1, and 95.1 μs, respectively. Although they show better performance than our implementation, their implementation for the sign operation consumes 3.5×, 9.2×, 241.2×, and 6× more LUTs, DFFs, DSPs, and BRAMs compared to our implementation. Moreover, our work can perform all three operations in a single implementation. The FPGA implementation of Dilithium [LSG21] targets reducing LUT utilization by employing extra DSP units for computations. Their implementation utilizes 1.5× more LUTs and 11.2× more DSP units. Our work shows 1.47×, 1.08×, and 1.36× better performance for the key generation, signature generation, and signature verification operations, respectively. In [BNG21], a high-performance Dilithium implementation is presented. It shows better performance at the expense of

Table 3.5: Comparison Table for Dilithium-3

Ref.	**Plat.**	**Performance** (in μs)	**Freq.** (MHz)	**Area** (mm^2 or LUT/ FF/DSP/BRAM)
[Zho+21]†	Zynq	-/12.6K/9.9K	100	2.6K/-/-/-
[Ric+21]^{a}	US+	51.9/-/-	350	54.1K/25.2K/182/15
[Ric+21]^{b,d}		-/63.1/-	333	68.4K/86.2K/965/145
[Ric+21]^{c}		-/-/95.1	158	61.7K/34.9K/316/18
[LSG21]^{d}	Ar.-7	229/0.3k/0.2k	145	30.9K/11.3K/45/21
[LSG21]^{e}		229/0.85k/0.2k		
[Zha+22a]^{d}	Ar.-7	60/0.12k/63.8	96.9	30K/10.34K/10/11
[Zha+22a]^{e}		60/0.46k/63.8		
[BNG21]^{d}	US+	32/63/39	145	55.9K/28.4K/16/29
[BNG21]^{e}		32/193/39		
Our^{d,f}	US+	114.7/237/127.6	200	18.5K/9.3K/4/24
	65nm	≈57.4/118.5/63.8	400	≈0.317+1.230 mm^2

^{a}: Works for K.Gen.; ^{b}: Works for Sign.; ^{c}: Works for Verify ; ^{d}: Reports best-case scenario ^{e}: Reports average-case scenario; ^{f}: Supports multiple schemes; †: HW/SW co-design.

3× and 4× more LUT/DFF and DSP, respectively. Similarly, in [Zha+22a] the authors achieve better performance at the expense of area. Our implementation is 2× slower but consumes 1.5× less area and provides the flexibility to do the operations in parallel or sequentially.

Comparisons with Saber-only implementations: There are several works in the literature implementing Saber in hardware, e.g., [Abd+; Dan+19; Fri+21b; HLX; Mar+20; RB20] on FPGA and [FSS20b; Gho+22; Imr+21; Zhu+21] on ASIC platforms. Their area and performance results, along with our work, are presented in Table 3.6. Their obvious limitation is that they only provide support for PKE/KEM but not a digital signature scheme. As we discussed in Section 3.2.7, the area of our design is determined by the Dilithium scheme; therefore, it is expected to consume more area than Saber-only implementations. However, when we compare our design to high-performance implementations of Saber, we consume much less area compared to [RB20; Tan+21a], and almost a similar area compared to [Imr+21; Zhu+21]. We consume 2× more area compared to [Gho+22] and deliver 3.5× better performance. Our unified cryptoprocessor outperforms [Abd+; Mar+20] and shows a similar performance compared to the architectures in [Dan+19; HLX].

3.4.3 Results for the `KaLi` Cryptoprocessor

In this section, we present the performance and area results of `KaLi`. The proposed architecture is described in Verilog. It is synthesized and implemented for Zynq Ultrascale+ ZCU102 with a performance-optimized strategy using Vivado 2019.1 tool and achieves 270 MHz clock frequency on the FPGA. The proposed architecture is also implemented with 65nm and 28nm ASIC technologies using the Cadence Genus tool. On 65nm/28nm ASIC technology, it achieves 280

Table 3.6: Comparison Table for Saber-KEM

Ref.	**Plat.**	**Performance** (in μs)	**Freq.** (MHz)	**Area** (mm^2 or LUT/ FF/DSP/BRAM)
[FSS20b][b]	65nm	16K/21K/26K	45.47	0.914 mm^2
[Imr+21]	65nm	7.1/7.1/9.3	1000	0.314 mm^2
[Zhu+21]	40nm	2.7/3.6/4.3	400	0.380 mm^2
[Gho+22]	65nm	89,6/116.9/146.18	40-160	0.158 mm^2
[Fri+21b][†,b]	Ar.-7	3.6K/4.9K/5.5K	62.5	20K/11K/13/36.5
[Mar+20][†]	Ar.-7	3.2K/4.1K/3.8K	125	7.4K/7.3K/28/2
[Abd+]	Ar.-7	/467.1/527.6	100	6.7K/7.3K/32/0
[HLX]	US+	48.9/63.2/78.5	250	10.1K/7.7K/0/3
[Dan+19][†]	US+	-/60/65	322	12.5K/11.6K/256/4
[RB20]	US+	21.8/26.5/32.1	250	23.6K/9.8K/0/2
[Tan+21a]	US+	10.2/12.6/15.6	250	41.5K/22.3K/64/2
[DMG21]	US+	7.6/10.5/14.2	400	21.6K/12.25K/24/10.5
Our[a,b]	US+	54.9/69.7/94.9	200	18.5K/9.3K/4/24
	65nm	≈27.5/34.9/47.5	400	≈0.317+1.230 mm^2

[a]:On-chip memory area is estimated as ≈1.230 mm^2; [b]:Supports multiple schemes; [†]: HW/SW co-design.

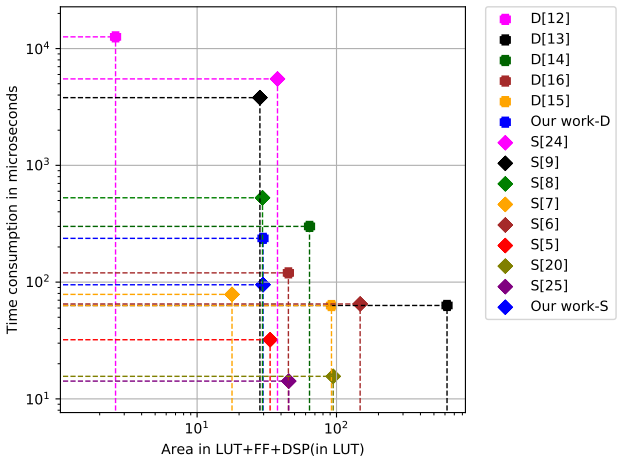


Figure 3.20: The figure presents a graphical representation of time-consumption and resource comparisons between our work and existing works. The lower the time-consumption and resource consumption, the better. With both schemes included, our cryptoprocessor still consumes reasonable resources and provides optimal performance.

Table 3.7: Area of KaLi on the Zynq Ultrascale+ ZCU102 FPGA platform. All security levels of Dilithium and Kyber are supported.

	Unit	S-1	S-2	INS	LUT	DFF	DSP	BRAM
	Comp.Core				**21K**	**9.2K**	**4**	**21**
Dilithium (D)	⌊Decompose	✓	-	1	474	338	0	0
	⌊Pow2Round	-	✓	1	55	84	0	0
	⌊MakeHint	-	✓	2	61	124	0	0
	⌊UseHint	-	✓	3	565	433	0	0
	⌊Encode_$\mathcal{H}$	-	✓	4	202	233	0	0
	⌊Pack	✓	-	2	582	181	0	0
	⌊Unpack	✓	-	3	315	182	0	0
	⌊SampleInBall	-	✓	5	505	285	0	0
	⌊Refresh	✓	-	4	8	7	0	0
	⌊VerifyEq.	-	✓	6	13	76	0	0
Kyber (K)	⌊Encode	-	✓	7	517	190	0	0
	⌊Decode	✓	-	5	237	180	0	0
	⌊Com./Decom.	✓	-	6/7	272	376	0	0
	⌊Verify	-	✓	8	102	216	0	0
	⌊CMOV	-	✓	9	20	120	0	0
	⌊COPY	-	✓	10	15	120	0	0
D+K	⌊Memory	-	✓	11	268	12	0	20
	⌊Keccak	✓	-	8-18	12K	3.5K	0	0
	⌊Multiplier	-	✓	12-16	3.5K	2K	4	1
	Prog.Contr.				2K	296	0	3
	Total				**23K**	**9.7K**	**4**	**24**

MHz/1 GHz for the slow clock (in logic units), and 560 MHz/2 GHz for the fast clock.

3.4.4 Area and Performance Results

Table 3.7 presents the detailed utilization of each building blocks in KaLi for UltraScale+ ZCU102 platform. The proposed cryptoprocessor uses 23,347 LUTs (8.4%), 9,798 DFFs (1.7%), 4 DSPs (0.1%), and 24 BRAMs (2.6%). On ASIC, KaLi consumes 1.107 mm^2 (769.04 KGE) in 65nm technology, and 0.263 mm^2 (747.81 KGE) in 28nm technology.

Table 3.8 presents the cycle count and latency (in μs) for the operations of Dilithium and Kyber. With a 270 MHz clock frequency in the FPGA, the CCA-secure key generation, encapsulation and decapsulation operations for Kyber-768 take 23.2, 29.11, and 41.82 μs, respectively. For the best-case scenario, where a valid signature is generated after the first loop iteration [Aik+23c], the key generation, signature generation, and signature verification operations for Dilithium-3 take 87.48, 179.91, and 96.76 μs, respectively. The ASIC implementation with 65nm/28nm technology (with 560 MHz/2 GHz clock frequency for the memory unit) can perform the operations for Kyber-768 and Dilithium-3 in 22.07/6.18, 27.59/7.73, and 39.62/11.09 μs, and 82.87/23.2, 171.03/47.89, and 91.66/25.67 μs, respectively. Next, we compare these results with prior works.

Table 3.8: Performance results for Dilithium and Kyber-KEM in FPGA

Operation	Dilithium-2 Kyber-512		Dilithium-3 Kyber-768		Dilithium-5 Kyber-1024	
	Cycle	μs	Cycle	μs	Cycle	μs
Dil.Gen	14,594	54.05	23,619	87.48	39,737	147.17
Dil.Sign$_{pre}$	7,883	29.2	9,640	35.7	12,943	46.27
Dil.Sign	21,812	80.79	36,643	135.72	53,965	199.87
Dil.Sign$_{post}$	1,967	7.23	2,463	9.12	3,271	12.12
Dil.Verify	15,423	57.12	26,124	96.76	46,671	172.86
Kyb.Keygen	3,395	12.6	6,291	23.2	9,089	33.7
Kyb.Encaps	4,956	18.4	7,862	29.11	11,351	42.04
Kyb.Decaps	6,807	25.21	11,291	41.82	13,905	51.5

3.4.5 Comparison with unified designs in literature

In [Zha+22b], the authors design a unified architecture for Dilithium and Kyber. They present both hardware/software co-design as well as hardware results for Kyber, while retaining some parts of Dilithium in the software. Their NTT unit occupies 25,674 LUTs, 3,137 DFFs, 64 DSPs, and 6 BRAMs on a Xilinx Artix-7 FPGA. The NTT unit alone occupies more LUT and DSP units than our entire design. On ASIC, it occupies 697 KGE on 28nm technology [Zha+22b], which is very close to our unified cryptoprocessor's 747 KGE area consumption. Their implementation shows similar performance for Kyber even though they target a high-speed design of Kyber in hardware and use 32 butterfly units for NTT, making their NTT unit 8× faster than `KaLi`. For Dilithium, `KaLi` shows 10× better results. The energy consumption of `KaLi` is also approximately half of their design for both Kyber and Dilithium.

To the best of our knowledge, no work exists in the literature that *unifies* Dilithium and Kyber solely in hardware. Therefore, next, we compare our work with standalone implementations of Dilithium or Kyber in hardware.

3.4.6 Comparison with Dilithium-only designs in literature

Comparison with FPGA-based implementations: Different works in the literature use different FPGA platforms. Hence, drawing a one-to-one comparison between works is not always feasible. When we began the hardware implementation of the proposed architecture, we selected Ultrascale+ as the platform and subsequently pipelined the building blocks to achieve a frequency of around 300 MHz on this platform. Several works in the literature optimized their designs for Artix-7 or other FPGAs. While optimizing the critical paths of architecture for meeting the desired clock frequency is heavily dependent on the technology of the platform, area requirements (LUT/DSP/FF/BRAM) do not change significantly across FPGA technologies. In Table 3.9, we present the FPGA implementation results of Dilithium-3 from the literature.

Zhou et al. [Zho+21] present an HW/SW co-design, implementing only the polynomial arithmetic unit in hardware. Thus, they consume less area but report

Table 3.9: Comparison Table for Dilithium-3 FPGA Implementations

Ref.	**Plat.**	**Performance** (in μs)	**Freq.** (MHz)	**Resources** (LUT/ FF/DSP/BRAM)
[Zho+21]†	Zynq	-/8.8K/9.9K	100	2.6K/-/-/-
[Ric+21]^{a}	US+V	51.9/-/-	350	54.1K/25.2K/182/15
[Ric+21]^{b,d}		-/63.1/-	333	68.4K/86.2K/965/145
[Ric+21]^{c}		-/-/95.1	158	61.7K/34.9K/316/18
[LSG21]^{d}	Ar.-7	229/0.3K/0.2K	145	30.9K/11.3K/45/21
[LSG21]^{e}		229/0.85K/0.2K		
[Zha+22a]^{d}	Ar.-7	60/0.12K/63.8	96.9	30K/10.34K/10/11
[Zha+22a]^{e}		60/0.46K/63.8		
[BNG21]^{d}	US+V	32/63/39	145	55.9K/28.4K/16/29
[BNG21]^{e}		32/193/39		
[Aik+23c]^{d,f}	US+Z	114.7/237/127.6	200	18.5K/9.3K/4/24
`KaLi`^{d,f}	**US+Z**	**82.8/171.3/96.7**	**270**	**23K/9.7K/4/24**

^{a}: Implements K. Gen. ^{b}: Implements Sign. ^{c}: Implements Verify. ^{d}: Reports best-case scenario. ^{e}: Reports average-case scenario. ^{f}: Supports multiple schemes. †: HW/SW co-design. US+V/Z refers to Virtex/Zynq US+ platforms.

Table 3.10: Comparison Table for Dilithium-3 ASIC Implementations

Ref.	**Tech.** (nm)	**Perf.*** (μs)	**Freq.** (MHz)	**Area** (KGE)	**SRAM** (KB)	**Energy*** (μJ)
[BUC19b]†	40	18,266	72	106	40.25	88.89
[Zha+22b]$^{\dagger a}$	28	747	540	697	24.75	62.39
[Aik+23c]^{a,b}	65	182.3	400	854	34.82	-
`KaLi`^{a,b}	**65**	**262.7**	**280/560**	**769**	**34.82**	117.9
`KaLi`^{a,b}	**28**	**73.55**	**1K/2K**	**747**	**34.82**	27

*:Performance/Energy is measured as total time/energy for signature generation and verification (key generation can be done offline). †: HW/SW co-design. ^{a}:Supports multiple schemes. ^{b}: Reports best-case scenario.

an inferior performance. Ricci et al. [Ric+21] provide separate designs for each of the Dilithium variants. These designs in total occupy 9× more area compared to our design and still perform as well as our design for signature verification. For a signature generation, their implementation shows only 3× improvement. Thus, our design gives a much better area-time trade-off result.

The authors in [BNG21; LSG21; Zha+22a] present Dilithium implementations, which consume much more area compared to our design. Note that across technologies, the area consumption does not change notably. A lower frequency in [LSG21; Zha+22a] can be justified by the use of Artix-7 FPGA, which is technologically inferior to our Ultrascale+ platform. A limitation of [Zha+22a] is that it employs a segmented pipeline, resulting in an inflexible data path for Dilithium. The implementation in [BNG21] uses a better platform than ours, consumes 3× more area (LUT+FF), and achieves a speed-up of only 2.5× (Sign+Verify). In [Aik+23c], the authors present a unified cryptoprocessor for Dilithium and Saber [DAn+21]. Their area is almost comparable to ours, considering the difference between Kyber and Saber. We achieve a higher clock

Table 3.11: Comparison Table for Kyber-1024 FPGA Implementations

Ref.	**Platform**	**Performance*** (in μs)	**Freq.** (MHz)	**Resources** (LUT/ FF/DSP/BRAM)
[BKS19]‡	Cortex-M4	33,850	100	-/-/-/-
[BUC19b]†	Artix-7	18,560	25	15K/3K/11/14
[FSS20a]†	Zynq	-	-	24K/11K/21/32
[Alk+20]†	Artix-7	85,559	59	2K/2K/5/34
[Hua+20]	Virtex-7	1,260	192	133K/-/548/202
[XL21a]	Artix-7	154	161	7K/5K/2/3
[Dan+20]	Artix-7	63	210	12K/12K/8/15
[BAK21a]	Artix-7	56	185	13K/12K/16/16
[BAK21b]	Artix-7	286	112	16K/6K/12/17
[BAK21b]	Virtex-7	205	156	16K/6K/12/17
[DMG21]	US+Z	23.5	450	11.6K/12K/8/8.5
[Ni+22]	US+Z	3.4 (Encap)	450	18.4K/13.7K/2/0
[Ni+22]	US+Z	4.1 (Decap)	450	15.9K/12.9K/2/0
`KaLi`^{a}	**US+Z**	**93**	**270**	**23K/9.7K/4/24**

*:Performance is measured as the total time for encapsulation and decapsulation (key generation can be done offline). ^{a}:Supports multiple schemes. †: HW/SW co-design. ‡: SW design. US+Z refers to Zynq Ultrascale+ platforms.

Table 3.12: Comparison Table for Kyber-1024 ASIC Implementations

Ref.	**Tech.** (nm)	**Perf.*** (μs)	**Freq.** (MHz)	**Area** (KGE)	**SRAM** (KB)	**Energy*** (μJ)
[BUC19b]†	40	6,444	72	106	40.25	36.06
[FSS20a]†	65	18,444	45	170	465	307.68
[Xin+20]†	28	727	300	979	12	19.57
[BAK21b]	65	160	200	104	190	-
[Zha+22b]^{†,a}	28	206	540	697	24.75	16.24
[Zha+22b]^a	28	22/17.7^b	540	623	36.75	-
`KaLi`^{a}	**65**	**90.2**	**280/560**	**769**	**34.82**	40.48
`KaLi`^{a}	**28**	**25.26**	**1K/2K**	**747**	**34.82**	9.27

*:Performance/Energy is measured as the total time/energy for encapsulation and decapsulation (key generation can be done offline). †: HW/SW co-design. ^{a}:Supports multiple schemes. ^{b}:Depending on the type of schedule.

frequency and report 1.4× better performance.

Comparison with ASIC-based implementations: Table 3.10 gives the comparison of implementation results for Dilihtium-3 on ASIC platforms. Banerjee et al. [BUC19b] present ASIC results for HW/SW co-design of Dilithium with Round 2 parameters. `KaLi` outperforms them significantly in terms of performance. Our hardware only design gives 45× better performance at the cost of only 7.5× more area. `KaLi` consumes almost the same area as reported in [Zha+22b] but gives a 10× and 2.8× better performance with 28nm and 65nm technology, respectively. [Aik+23c] reports a higher number of logic gates than our design. `KaLi` sets new records for energy consumption, in both 28nm and 65nm technologies.

Thus, our FPGA and ASIC models are the most compact compared to all the existing Dilithium implementations.

3.4.7 Comparison with Kyber-only designs in literature

Comparison with FPGA-based implementations: Table 3.11 gives the comparison of implementation results for Kyber-1024 on FPGA platform. Banerjee et al. [BUC19b] present an HW/SW co-design for Kyber. `KaLi` surpasses their performance results on both platforms, at the cost of some areas. Observe that `KaLi` gives better results compared to software only [BKS19] as well as HW/SW co-designs [Alk+20; BUC19b; FSS20a]. The hardware-only designs[BAK21a; BAK21b; Dan+20; Hua+20; XL21a] target Artix-7 or Virtex-7 FPGAs. Our `KaLi` consumes significantly smaller area than [Hua+20]. Authors in [BAK21a; BAK21b; Dan+20; DMG21] target a high-speed Kyber implementation and therefore achieve better performance and frequency. In [Ni+22], the authors present separate results for all the Kyber variants, and present individual encapsulation and decapsulation architectures (unlike `KaLi` combines all operations in a single architecture). Their design goal is also high-speed, and their standalone implementation of Kyber-1024 encapsulation consumes more area than `KaLi`. Note that the area of our design is determined by Dilithium and not by Kyber. Therefore, although the results indicate that we consume a very high area, we only consume the bare minimum and achieve almost the best performance results on the FPGA platform.

Comparison with ASIC-based implementations: Table 3.12 gives the comparison of implementation results for Kyber-1024 on ASIC platform. On the ASIC platform, `KaLi` consumes the same area as reported in [Zha+22b] but gives a 2.3/8.1× better performance under 65nm/28nm technology. In fact, we surpass all existing designs [BAK21b; BUC19b; FSS20a; Xin+20] in terms of performance. However, compared to some of the designs, we use more area, and for this, we must remind you again that Kyber is the recessive scheme among the two, and therefore, this area is higher when compared to Kyber-only implementations. `KaLi` consumes the least energy of all for respective technologies.

We have now established that `KaLi` transcends all the state-of-the-art works that exist in literature. Thus, it is demonstrated that the proposed design methodology yields better results. Next, we discuss the aspect of application benchmarking.

3.4.8 Application-Benchmarking and Impact

Several works, for example, [Bür+20; Geo+21], present application benchmarking using the existing libraries. The authors in [Bür+20] provide results for the TLS protocol using the mbed TLS library and use Kyber for KEM and SPHINCS+ for digital signature. Runtimes are reported for Raspberry Pi 3 Model B+, ESP32-PICO-KIT V4, Fieldbus Option Card, and LPC11U68 LPCXpresso. Compared to these works, `KaLi`'s FPGA implementation shows speedups of 85×, 1349×, 6190×, and 23809×, respectively. Notably, the ASIC models of `KaLi` will further improve the timings.

In [Geo+21], the authors evaluate PQ TLS 1.3, a post-quantum variant

of TLS 1.3. It supports Round 3 parameters for both Kyber and Dilithium, as well as other schemes. They use the ARM Cortex-M4 embedded platform NUCLEO-F439ZI, with and without hardware acceleration. These boards can reach a maximum frequency of 180MHz. Compared to their results for Kyber's decapsulation, we achieve a speedup of 131× if we run our design at 180 MHz. The authors report that replacing RSA+ECDHE with Dilithium3+Kyber5 in TLS handshake increased the runtime by 64%. `KaLi` can help bridge this gap. Thus, replacing these devices with `KaLi` would give a significant speedup. `KaLi` only occupies 8% of the available resources on the Zynq US+ FPGA board, implying the ability to run twelve such unified cores in parallel, further improving the speedup.

There are several data centres and network security appliances where high-performance SIMD processors (e.g., Intel/AMD with AVX) are too expensive to deploy or extremely constrained (or passively powered) devices are too slow to use. There are commercial cryptoprocessors that target such applications. For example, NXPs C29x family of crypto coprocessors [NXP] (which are battery-powered) utilize dedicated hardware acceleration to speed up RSA and elliptic curve-based public-key cryptographic computations. It performs up to 32,000 RSA-2048 public-key operations per second. The more constrained platform, OPTIGA TPM SLB 9672 from Infineon [Inf], features hardware acceleration for RSA-4096. They use the same RSA engine for both public-key signature and key agreement.

Our unified Kyber+Dilithium coprocessor performs faster than the public-key engines of [NXP] and [Inf] and, at the same time, requires only 0.263 mm^2 area in a 28nm node. When a smaller area is required by an application, some of the design parameters (e.g., number of NTT cores, Keccaks data width, etc.) can be tuned accordingly to meet the area budget at the cost of speed. The proposed design techniques and architecture will be useful to replace the classical public-key cryptography used in conventional cryptoprocessors with post-quantum key agreement and signature.

3.5 Discussion and Conclusion

Post-quantum key encapsulation and digital signature algorithms are required for securing communication. In this Chapter, we presented a design methodology for efficient and compact hardware implementation of both post-quantum key encapsulation and digital signature algorithms in a unified cryptoprocessor architecture. Following the proposed methodology, we designed and implemented the first unified cryptoprocessor architecture `KaLi` that can perform all the cryptographic protocol operations of the NIST standardized primitives- Dilithium and Kyber for all the security levels. Architectural optimizations in the data path of the cryptoprocessor were performed to reduce the cycle count and improve the clock frequency. Experimental evaluations on FPGA and ASIC platforms demonstrated that our cryptoprocessor outperforms all existing implementations. Therefore, this implementation is a significant step towards making post-quantum

cryptography compact and agile on hardware platforms. The proposed design methodology can be customized to meet different application-specific constraints and requirements.

A knowledge of compatibility between different schemes is crucial for their effective use in real-life applications. However, the security of such implementations is also a valid concern. The hardware implementations presented in this Chapter are resistant to timing attacks but do not incorporate any side-channel resisting countermeasure, for example, masking against more power side-channel attacks. Side-channel protection of the unified cryptoprocessor architecture will require significant research and is considered future work. There are several works in the literature on masking Saber [Bei+20], Kyber [Bos+21a; Fri+21a], and Dilithium [Mig+19]. How to design an 'optimal and unified side-channel protection mechanism for a unified hardware implementation of Kyber and Dilithium is an interesting topic that is still open and needs to be researched. Furthermore, researching protection techniques against fault injection-based attacks will also be important due to the vast deployment of these cryptographic schemes in various embedded devices.

4

The Kavach Masking Technique

Breu

This Chapter focuses on securing lattice-based PQC implementations against side-channel attacks. It explores the trade-offs between performance optimizations using small secrets and the challenges of applying masking countermeasures for the PQC schemes utilizing power-of-two moduli. The chapter introduces two novel multiplication techniques- one for NTT-based (introduced in the previous Chapter for Saber) and another for Schoolbook-based polynomial multiplication, that support efficient masking without sacrificing compactness.

Contribution: This chapter is based on the following publications, for which the thesis author was the principal author. The author contributed to the conceptual development, performed the experiments, implemented all components, and drafted significant portions of the text, with Gaetan contributing to early discussions on the masked gadget and to the security proofs.

• *Kavach: Lightweight masking techniques for polynomial arithmetic in lattice-based cryptography* by Aikata, Andrea Basso, Gaetan Cassiers, Ahmet Can Mert, and Sujoy Sinha Roy, appeared in IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES 2023).

4.1 Introduction

While the NIST standardization aimed at selecting a portfolio of post-quantum algorithms for *general applications*, there is a need for developing tailored post-quantum schemes that meet application-specific needs, for example, the constraints of the IoT and automotive applications. In recent years, several new post-quantum algorithms [Duc+22; Mer+21] have emerged with better performance or security features (or both) than the candidate algorithms in the NIST standardization. Security against side-channel attacks [MOP07b] has become an essential requirement in applications where an attacker can obtain side-channel information, such as variations in power consumption, electromagnetic emanations, or temperature, of the cryptographic device.

Masking is a widely used countermeasure against differential power analysis-based side-channel attacks [MOP07b]. Several masking schemes have been proposed in the literature [Abd+21a; Bei+21; Fri+22; Kun+22; Ode+18; Rep+15] for lattice-based public-key algorithms. Section 2.5.2 discusses masking works in more detail. Typically, masking schemes for lattice-based algorithms increase the computational overhead by approximately 2.5× to 6× (for first-order protection) compared to unprotected implementations. Therefore, significant research is needed to investigate masking schemes that reduce the performance overhead. One of the computationally-intensive operations in lattice-based algorithms is the multiplication of polynomials [Sin19]. The asymptotically fastest method for this is to use NTT, which reduces polynomial multiplication to coefficient-wise multiplication. The NTT-based polynomial multiplication is, however, limited to polynomial rings where the modulus q is prime (we name such rings "NTT-friendly", while rings with no NTT are "NTT-unfriendly"). It is still possible to use NTT-based polynomial multiplication to implement cryptographic algorithms with NTT-unfriendly rings, but this comes at a performance overhead since it typically involves computing NTTs with a significantly larger modulus.

On the other hand, recent works have shown that when a structured lattice-based scheme uses a power-of-2 modulus, the masking countermeasure incurs the least performance overhead [Bei+21; Fri+22; Kun+22] since, when using binary (masked or not) representations, a modular reduction is essentially free for such moduli. The downside of using a power-of-2 modulus is that the ring is NTT-unfriendly. In learning with errors (LWE) or learning with rounding (LWR)-based public-key schemes, this limitation is mitigated since one operand polynomial (typically a secret or an error polynomial) is always a small-coefficient polynomial, allowing significant optimizations in multiplication algorithms, both "classical" (e.g, schoolbook, Toom-Cook or Karatsuba) and NTT-based (allowing to use a smaller modulus [Aik+23c]). For example, [RB20] presents a very low latency schoolbook polynomial multiplier (unprotected implementation) by using many small-coefficient (hence low-cost) multipliers in parallel. Unfortunately, existing masking schemes for lattice-based cryptography prevent such optimizations since they split a secret or an error polynomial with small coefficients into uniformly random polynomials with large coefficients.

In this chapter, we propose two techniques to extend the benefits of the

small-coefficient property to masked implementations. These techniques are then instantiated in concrete hardware design, implementation, and t-test evaluation to verify real-life security guarantees[1]. Our first method is an optimized technique for NTT-based polynomial multiplication in masked implementations of algorithms with NTT-unfriendly rings. Through a new choice of group for the arithmetic masking, we can reduce the required size of the modulus for the NTT. Among methods that improve over the worst-case "large coefficient" masked NTT-based multiplication for NTT-unfriendly rings, our method is the first general one (i.e., it works with any modulus), and it surpasses all the existing works (in hardware, it uses two to three times less area for the same performance). This method is applicable to any masking order and incurs a cost (in terms of computation time or area) proportional to the number of shares. Another advantage of this method is that it can efficiently utilize hardware that implements *non-masked* NTT-based polynomial multiplication in the same NTT-unfriendly ring, allowing for resource sharing.

Our second method applies to "classical" multiplication algorithms. We propose a new *hybrid masking* that combines arithmetic and Boolean masking, which allows us to split small secret coefficients into small shares, thus preserving the small secret property. We show how to use masking when implementing lattice-based cryptography, including conversions between arithmetic and hybrid masking. Our new hybrid masking gadgets are proven to be secure against first-order leakage in the hardware glitch-robust probing model. Finally, we provide experimental validation by implementing the proposed algorithms in hardware and performing leakage assessment tests [Goo+11]. Regarding performance, the use of hybrid masking in place of the state-of-the-art approach of arithmetic masking with large coefficients allows the use of small multipliers instead of large ones (4.5× area gain), at the cost of performing more additions (since accumulators are already instantiated, this does not increase the area).

With these contributions, we improve the state-of-the-art for masking lattice-based schemes and provide improved insights on implementation costs for future PQC scheme designs.

Organization: We present the research gap in Section 4.2. In Section 4.3, we present the methods to optimize side-channel countermeasures for an NTT-based polynomial arithmetic unit. In the next section, Section 4.4, we introduce the technique of hybrid masking to efficiently mask non-NTT-based polynomial arithmetic. We then implement the proposed techniques and provide implementation details in Section 4.5. Following this, Section 4.6 describes the t-test evaluation setup and results. Finally, in Section 4.7, we discuss the applications of the proposed techniques and conclude the Chapter.

[1] Our implementation is available at `https://extgit.iaik.tugraz.at/sesys/kavach_artifacts`.

4.2 Masking lattice-based PKE and Research gap

In the last several years, especially during the NIST PQC standardization competition, several masking schemes for PQC algorithms have been proposed. In this section, we revisit a subset of existing masking schemes that are relevant to this Chapter.

The first masking of a ring-LWE-based LPR public-key encryption (Algorithm 1) was proposed in [Rep+15], based on arithmetic masking in $\mathbb{R}_q$. Concretely, to mask the decryption, it splits the secret $\boldsymbol{s} \in \mathbb{R}_q$ into two uniformly random arithmetic shares $\boldsymbol{s}_0, \boldsymbol{s}_1 \in \mathbb{R}_q$. Next, $\tilde{\boldsymbol{m}}'$ is computed as two arithmetic shares $\tilde{\boldsymbol{m}}'_0 = (\boldsymbol{v} - \boldsymbol{u} \times \boldsymbol{s}_0)$ and $\tilde{\boldsymbol{m}}_1{}' = -\boldsymbol{u} \times \boldsymbol{s}_1$. Finally, for the comparison-based decoding step, a probabilistic masked decoder is used in [Rep+15]. A hardware implementation of the masked decryption is provided where the masked polynomial multiplication is implemented using the NTT method. Masking the polynomial arithmetic is simple as it incurs only $d\times$ computation overhead while d number of arithmetic shares of the secret. On the other hand, the work shows that masked decoding is a more complex and slow operation. A more efficient masked decoding technique is presented in [Ode+18]. To perform decoding on the input arithmetic shares of a coefficient, it first shifts the distribution of the shares, then applies the arithmetic-to-Boolean (A2B) transformation, and finally computes the sign bit. Besides improving the masked decoding, the work also presents a complete masking framework for ring-LWE-based public-key encryption with CCA security (i.e., chosen ciphertext attack resistance). Hence, the work masks sensitive parts of the re-encryption operation including error and ephemeral secret polynomial sampling, and random number generation. With a first-order masked implementation, the NTT-based polynomial multiplication cost increases to $2\times$ and the bit-wise building blocks for example, error sampling, and random number generation become more than $5\times$ slower. The masked decryption for a CCA-secure variant of NewHope [Alk+16] is around $5.7\times$ slower than its non-masked operation.

Significant reduction in the masking overhead for CCA-secure public-key encryption or KEM is demonstrated in [Bei+21]. The integration of bit-wise building blocks with the polynomial arithmetic blocks inside a masking scheme requires expensive Boolean-to-arithmetic and arithmetic-to-Boolean conversions. The work observes that by choosing the modulus q to be a power-of-2, these conversions can be simplified a lot compared to the case with a prime modulus. For such power-of-2 modulus, the work heavily optimizes masked logical shifting on arithmetic shares and error sampling from a binomial distribution. On the ARM Cortex-M4 platform, masking the Saber KEM (which uses power-of-2 modulus) incurs only $2.5\times$ computation overhead unlike $5.7\times$ overhead with prime modulus in [Ode+18]. Polynomial multiplication is computed using a combination of Toom-Cook, Karatsuba, and low-degree schoolbook methods as the NTT method is not a natural choice with a power-of-2 modulus.

A lightweight masked hardware implementation of Saber KEM is presented in [Abd+21a]. It shows an area overhead of $2.9\times$ and the polynomial multiplication is implemented using the Schoolbook method. In [Kun+22], the authors

propose a higher-order masked Saber implementation on ARM Cortex-M4. They show that the increase in performance overhead for higher-order masking of Saber is smaller compared to Kyber. Their first-order masked implementation shows 2.7× performance overhead. Both works do not take the advantage of secret coefficients of Saber being small.

A masked HW/SW codesign implementation of several NIST PQC finalists (such as Saber, Kyber, and NTRU) is presented in [Fri+22]. It extends the instruction set of a RISC-V processor by incorporating an NTT-based polynomial multiplier. Hence, all PQC schemes irrespective of cohesion with NTT, are forced to use the NTT method for computing polynomial multiplication. Critical non-linear operations for masking are computed in the hardware. When the masking countermeasure is turned off, Kyber (uses NTT-friendly prime) is around 1.2× faster than Saber (uses NTT-unfriendly power-of-2 modulus). However, with first-order masking turned on, Kyber's decapsulation becomes around 1.53× slower than that of Saber. Such results clearly demonstrate that in applications where masking countermeasure becomes essential, using a power-of-2 modulus is more beneficial for the performance even though it may cause a minor slowdown in polynomial multiplication. Therefore, in this Chapter, we primarily consider masking with power-of-2 modulus. We would like to remark that a power-of-2 modulus can be used in standard as well as structured lattices with both LWE and LWR foundations.

Research gap: Taking advantage of small operands for the implementation of polynomial multiplication has been studied in several works. In [RB20], the authors propose an efficient schoolbook multiplier implementation where one of the operands has small coefficients. Their implementation uses an add-shift-based approach to perform the multiplication with small coefficients and eliminates large multiplier units. For NTT-based multiplication implementations in $\mathbb{R}_q$ where q is power-of-2, [Aik+23c; Bas+21b; Chu+21b] show how to take advantage of small coefficients and use a smaller NTT-friendly modulus Q. For example, authors in [Aik+23c] show that a 25-bit NTT-friendly prime Q would be sufficient to perform NTT-based polynomial multiplication for Saber instead of a 34-bit prime [FSS20c].

A natural question is raised in [Abd+21b], can the masked implementations also benefit from one of the multiplicands being small? Existing works targeting masked implementations of schemes with power-of-2 modulus (e.g., Saber, Florete, Espada, and Sable [DAn+21; Mer+21]) fail to utilize the small secret property. The size of the datapath for schoolbook or NTT-based implementations grew in this case as shown in previous works [Abd+21a; Fri+22], which is not needed and is an undersell for efficient MLWR-based schemes. In this Chapter, we show how to utilize small coefficients in masked implementations.

4.3 Masking NTT in NTT-unfriendly rings

In this section, we introduce our first method: an efficient masked NTT-based multiplication for small polynomials in NTT-unfriendly rings. This method is

well-suited to platforms where an NTT accelerator is already available, such as microcontrollers with NTT acceleration or hardware components that implement multiple lattice-based algorithms, including NTT-based ones like Kyber and Dilithium. In general, NTT-based polynomial multiplication requires the coefficient modulus q to be a prime, since otherwise (e.g., when q is a power of 2) the NTT does not exist in $\mathbb{R}_q$. To force NTT-based polynomial multiplication in such rings, one solution is to perform an NTT-based multiplication using a large and prime modulus Q such that the coefficients in all the intermediate values are never larger than the modulus, and therefore no actual modular reduction (mod Q) ever happens. Then, the final result is obtained by reducing modulo q. In general, one must ensure $Q > n \cdot q^2$ to avoid unwanted modular reductions [Aik+23c; Fri+22].

The size of Q can be reduced when one polynomial operand is always of small coefficients, for example, an error or secret polynomial in an LWE/LWR scheme. Concretely, if one of the polynomials is small (i.e. all its coefficients belong to $[-\mu, +\mu] \mod q$), a prime $Q > n \cdot q \cdot \mu$ is sufficient. Optimized but unprotected hardware [Aik+23c] and software [Sin19] implementations of NTT-based polynomial multiplication in NTT-unfriendly rings benefit from this relaxation on Q to improve speed and also reduce the memory requirement and width of the datapath.

However, when the small polynomial is arithmetically masked, the shares of its coefficients are uniform in $\mathbb{Z}_q$ and are therefore not small. The natural solution is therefore to switch back to using a large prime $Q > n \cdot q^2$, resulting in performance degradation [Fri+22]. The key idea of our new technique is the change of perspective: instead of multiplying each of the secret shares in modulo q with the public polynomial (achieved by working with a large $Q > n \cdot q^2$), we consider the non-masked multiplication (hence using a smaller $Q > n \cdot q \cdot \mu$) and mask the operations of this multiplication using arithmetic masking modulo Q. Therefore, we do not concern ourselves with modular reductions occurring on the shares: the final unmasked result is guaranteed to be correct.

Our new method is described in Algorithm 10. Let the unprotected polynomial multiplication be $\boldsymbol{a} \times \boldsymbol{s}$ in $\mathbb{R}_q$ where the public polynomial $\boldsymbol{a}$ is uniformly random in $\mathbb{R}_q$ and the secret polynomial $\boldsymbol{s}$ is a small polynomial with coefficients in $[-\mu, +\mu]$. We take as input an arithmetic sharing $(\boldsymbol{s}_0, \boldsymbol{s}_1)$ in $\mathbb{R}_Q$ for a prime $Q > n \cdot q \cdot \mu$ (converting from $\mathbb{R}_q$ if needed), then compute two polynomial multiplications $\boldsymbol{a} \times \boldsymbol{s}_0$ and $\boldsymbol{a} \times \boldsymbol{s}_1$ independently in $\mathbb{R}_Q$ using NTTs. Finally, we perform masked modular reduction modulo q to bring the result to $\mathbb{R}_q$.

Algorithm 10 NTT-based polynomial multiplication with masking

Require: $\boldsymbol{a} \in \mathbb{R}_q$ (public polynomial), $\boldsymbol{s}_0$, $\boldsymbol{s}_1 \in \mathbb{R}_Q$ (secret shares s.t. $\boldsymbol{s}_0 + \boldsymbol{s}_1 = \boldsymbol{s} \bmod Q$)
Ensure: $\boldsymbol{t}_{0,q}, \boldsymbol{t}_{1,q}$ (s.t. $\boldsymbol{t}_{0,q} + \boldsymbol{t}_{1,q} = \boldsymbol{a} \times \boldsymbol{s} \bmod q$)
1: $\boldsymbol{t}_{0,Q} \leftarrow \texttt{INTT}(\texttt{NTT}(\boldsymbol{a}) \star \texttt{NTT}(\boldsymbol{s}_0)) \in \mathbb{R}_Q$
2: $\boldsymbol{t}_{1,Q} \leftarrow \texttt{INTT}(\texttt{NTT}(\boldsymbol{a}) \star \texttt{NTT}(\boldsymbol{s}_1)) \in \mathbb{R}_Q$
3: $\{\boldsymbol{t}_{0,q}, \boldsymbol{t}_{1,q}\} \leftarrow \texttt{MaskedRed}_{Q \to q}(\boldsymbol{t}_{0,Q}, \boldsymbol{t}_{1,Q})$
4: **return** $\boldsymbol{t}_{0,q}, \boldsymbol{t}_{1,q} \in \mathbb{R}_q$

As a concrete example, let us consider Saber [DAn+21]: $q = 2^{13}$ and $\mathbb{R}_q = \mathbb{Z}_q/\langle x^{256} + 1\rangle$. If NTT is to be used for unprotected $\boldsymbol{a} \times \boldsymbol{s}$, an ephemeral-prime modulus Q (25-bit number [Aik+23c; Chu+21b; DMG23]) is needed since the coefficients of the public $\boldsymbol{a}$ and secret $\boldsymbol{s}$ are respectively 13 and 4-bits long. Our technique (Algorithm 10) can use the same value for Q. Without using our technique, the coefficients of the shares of the secret polynomial are uniformly distributed, and Q must be 34-bit for the sake of correctness, which is commonly done in masking literature [Abd+21b; Fri+22].

For one such NTT-unfriendly parameter set, the authors in [Aik+23c] propose 23, 24, and 25-bit primes in order to support NTT-based polynomial multiplication. The LUT consumption for making a 34×34-bit multiplier is twice that of a 25×25-bit multiplier and three times that of a 23×23-bit multiplier. The 39-bit NTT-based multiplier proposed by the authors in [Fri+22] consumes 2,454 LUTs, 1,917 FFs, 7 DSPs, 4.5 BRAMs, and requires 4,096 clock cycles. The proposed method ensures that no changes are needed to the NTT-based polynomial arithmetic unit, allowing it to complete the NTT in just 512 clock cycles and consume only 2,257 LUTs, 1,079 FFs, 4 DSPs, and 1 BRAM (as reported in [Aik+23c]).

Regarding modulus conversions for arithmetic masking, a generic solution is to use an arithmetic-to-Boolean (A2B) conversion followed by a Boolean-to-arithmetic (B2A) conversion [BC22]. However, some of these conversions can be avoided to reduce costs, e.g. when the input and/or output is generated/used in a Boolean-masked representation.

4.4 Compact polynomial arithmetic using hybrid masking

In this section, we introduce our second method for masking polynomial multiplication. Unlike our NTT-based method, this *hybrid masking* technique produces small arithmetic share coefficients if the unmasked polynomial has small coefficients. Then, the polynomial multiplication can be performed on the shares using any multiplication algorithm, with the ability to optimize for the small coefficients. We introduce the hybrid sharing representation and explain how to generate and use it. Finally, we describe how these techniques can be applied and optimized for polynomial multiplication.

4.4.1 Hybrid masking

In a nutshell, hybrid masking represents a small value s in a large group $\mathbb{Z}_q$ by starting from an arithmetic sharing (s_0, s_1) in a smaller group $\mathbb{Z}_K$ (i.e., $s = s_0 + s_1 \bmod K$). It converts this modular sharing into a non-modular one (i.e., in $\mathbb{Z}$) by explicitly computing the overflow bit c: $s = s_0 + s_1 - Kc$, which is then also a correct (non-uniform) arithmetic sharing in $\mathbb{Z}_q$ for any q. Finally, since that overflow bit c may leak information on the secret, we encode it using

Boolean masking. These two sharings (arithmetic and Boolean) form a hybrid sharing.

Definition 1 (First-order hybrid sharing)**.** Let $s \in \{0, \ldots, K-1\}$[2], $s_0, s_1 \in \{0, \ldots, K-1\}$ and $c_0, c_1 \in \{0, 1\}$. The tuple (s_0, s_1, c_0, c_1) is a first-order hybrid sharing of s if

$$s = s_0 + s_1 - K(c_0 \oplus c_1).$$

The hybrid sharing is *uniform* if (s_0, s_1) is a uniform arithmetic sharing of s in $\mathbb{Z}_K$ and if the Boolean sharing (c_0, c_1) is uniform and independent of (s_0, s_1), conditioned on the overflow bit $c = c_0 \oplus c_1$.

Example. For example, let $K = 8$, and $s = 5$. If the arithmetic part of the sharing is $(s_0, s_1) = (2, 3)$, then there is no overflow and $c = 0$ (hence (c_0, c_1) must be $(0, 0)$ or $(1, 1)$). However if we take $s_0 = 6$, then $s_1 = 7$ (the hybrid sharing definition implies $s = s_0 + s_1 \mod K$), and there is an overflow, giving the carry bit $c = 1$.

Let us now show why it is important to mask c.

Proposition 1. *In a uniform hybrid sharing (s_0, s_1, c_0, c_1) of s modulo K, the distribution of the overflow bit $c = c_0 \oplus c_1$ depends on s.*

Proof. As noted above, $s_0 + s_1 = s \mod K$. Therefore, if $s_0 \leq s$, then $s_1 = s - s_0$ and $c = 0$. Conversely, if $s_0 > s$, $s_1 = K + s - s_0$ and $c = 1$. Moreover, since (s_0, s_1) is a uniform sharing of s in $\mathbb{Z}_K$, s_0 is uniform in $\mathbb{Z}_K$ and independent of s. As a result $\Pr[c = 1] = \Pr[s_0 > s] = (K - 1 - s)/K$. □

Proposition 2. *In a uniform hybrid sharing (s_0, s_1, c_0, c_1) of s modulo K, for any $i, j \in \{0, 1\}$, the hybrid share (s_i, c_j) tuple is uniformly distributed over $\mathbb{Z}_K \times \{0, 1\}$ and independent of s.*

Proof. For all $i, j \in \{0, 1\}$, we marginalize with respect to c_{1-j} the conditional independence assumption for a uniform hybrid sharing to observe that c_j is independent of (s_0, s_1), since c_j is independent of c. This implies that c_j is independent of (s_i, s). Next, since (s_0, s_1) are uniform arithmetic shares of s, s_i is independent of s. We conclude by remarking that c_j and s_i are uniform. □

An interesting property of hybrid masking is that it can be viewed as (non-uniform) three shares arithmetic sharing $(s_0, s_1, -Kc)$ of s in any integer group, with the twist that the bit c is encoded with Boolean sharing. We now discuss how to generate hybrid shares from arithmetic shares modulo K. This will be followed by an introduction to the gadget that exploits this property to compute any linear operation on a hybrid masked value, and outputs arithmetically masked shares.

[2]If s belongs to another integer interval (e.g., $s \in \{-\mu, \ldots, \mu\}$), we can share a shifted version of s (e.g., $s + \mu \in \{0, \ldots, 2\mu\}$) and handle the subtraction of that offset at a later stage.

4.4.2 Hybrid shares generation

Let us first consider the problem of generating uniform hybrid shares from arithmetic shares (s_0, s_1) modulo K. We can directly use s_0 and s_1 as the arithmetic part of the hybrid shares and only have to compute fresh Boolean shares of $c = \lfloor \frac{s_0+s_1}{K} \rfloor$ without leaking any information about $s = s_0 + s_1 \mod K$. In other words, we want Boolean shares of the overflow bit of the summation $s_0 + s_1$.

In this section, we propose two solutions for this problem. The first one is inspired by the table-based Arithmetic-to-Boolean (A2B) technique of [CT03] that can evaluate any function on arithmetic shares in $\mathbb{Z}_K$ and provides its result as an arithmetic sharing in $\mathbb{Z}_{K'}$. The second one is based on the masked modular addition of [BC22] in which the carry is computed in order to perform the modular reduction. Although the second approach is slower than the first, it is more compact.

Table-based technique. We develop a hardware version of the A2B algorithm of [CT03] (using $n = 1$ nibble, hence the fix of [Deb12] is not needed) that generates only the overflow bit. In the following, $K = 2^k$. Our `AOverflowBit` (Algorithm 11) takes two arithmetic shares, $(s_0, s_1) \in \mathbb{Z}_K^2$ and outputs $(c_0, c_1) \in \mathbb{Z}_2^2$ such that $c_0 \oplus c_1 = (s_0 + s_1) \gg k$. The gadget uses three uniform random values, $r \in \mathbb{Z}_K$ and $\gamma, c_1 \in \mathbb{Z}_2$, to hide the sum of the two shares and their carry. Firstly, a table C_A with K entries is generated (implemented as a RAM). For a ranging from zero to $K - 1$, each entry is computed as

$$C_A[a] = \begin{cases} \gamma, & \text{if } a < K - r, \\ \gamma \oplus 1, & \text{otherwise.} \end{cases} \tag{4.1}$$

Finally, the share c_0 is computed from the result of the table lookup and the overflow bit of the index computation.

The idea behind this gadget is that since we cannot compute $s_0 + s_1$ explicitly and compare it to K, we sample randomness r, compute $(s_0 - r) + s_1$ and compare it to $K - r$. Since the result of the comparison must be masked, we take the random bit γ as a Boolean mask for the result of the comparison, which is implemented as a table lookup.

Proposition 3. *`AOverflowBit` (Algorithm 11) is correct.*

Proof. Let us remark that $T_l = (s_0 + s_1 - r) \bmod K$ and that $T_h = \lfloor ((s_0 + s_1 - r) \bmod 2K)/K \rfloor \oplus \gamma \oplus c_1$. Since $-K < s_0 + s_1 - r < 2K$, we identify three cases. In the first case, $-K < s_0 + s_1 - r < 0$, which implies that $s_0 + s_1 < K$, and $T_l = s_0 + s_1 - r + K$, therefore $C_A[T_l] = \gamma \oplus 1$, while $T_h = 1 \oplus \gamma \oplus c_1$, and as a result $c_0 = c_1$, which is correct. In the second case $K \leq s_0 + s_1 - r < 2K$, we have $s_0 + s_1 \geq K$ and $T_l = s_0 + s_1 - r - K$, and the proof is similar to the first case. The third case $0 \leq s_0 + s_1 - r < K$ has to be split in two sub-cases $s_0 + s_1 < K$ and $s_0 + s_1 \geq K$, which are analyzed similarly to the other cases. □

Algorithm 11 `AOverflowBit` ($\texttt{Reg}(\cdot)$ indicates the need for a register in a hardware implementation to ensure security against glitches.)

Require: Shares $s_0, s_1 \in \mathbb{Z}_K$, with $K = 2^k$.
Ensure: Shares $c_0, c_1 \in \mathbb{Z}_2$ such that $c_0 \oplus c_1 = (s_0 + s_1) \gg k$.

1: $\gamma, c_1 \xleftarrow{\$} \mathbb{Z}_2$, $r \xleftarrow{\$} \mathbb{Z}_K$
2: Initialize the table C_A using Equation 4.1
3: $T \leftarrow \texttt{Reg}((s_0 - r - K\gamma) \bmod 2K)$
4: $T_1 \leftarrow \texttt{Reg}((T + Kc_1 + s_1) \bmod 2K)$
5: $T_l \leftarrow T_1 \bmod K$
6: $T_h \leftarrow T_1 \gg k$
7: $c_0 \leftarrow (T_h + C_A[T_l]) \bmod 2$
8: **return** c_0, c_1

Proposition 4. *`AOverflowBit` is first-order glitch-robust probing secure. Its output shares are uniformly distributed and independent of the input shares, conditioned on the unmasked output $c = c_0 \oplus c_1$.*

Proof. Let us assume that (s_0, s_1) are uniform arithmetic shares of s modulo K. s_0 and s_1 are the arithmetic shares of s.

We will show that any glitch-extended probe on an intermediate value is independent of s. First, the pre-computation of the table C_A does not depend on any input share. Then, the computation of T depends on only one input share, hence it is independent of s. Additionally, $r + K \cdot \gamma$ is uniformly distributed in $\{0, \ldots, 2K-1\}$, therefore T is independent of s_0. Next, in the computation of T_1, T_l and T_h, s_0 does not appear (except through T, but it is independent of s_0 since r is uniform[3]), hence all these intermediate values are independent of s. In the table lookup, we assume that the RAM does not glitch (i.e., does not read multiple cells in a single cycle), therefore, this only leaks the address T_l and the read value (γ or $\gamma \oplus 1$). However, T_l is independent of γ (due to the selection of the LSBs in the reduction modulo K), hence leaking both values at once is still independent of s. Finally, in the computation of c_0, T_h is independent of γ thanks to the addition of c_1 in its computation, and therefore it is independent of $C_A[T_l]$, ensuring that the computation is independent of s.

Lastly, (c_0, c_1) are uniform shares due to the independent sampling of c_1. This also ensures the independence of (s_0, s_1), conditioned on c. □

This proof relies crucially on glitch-free memory access, which has been empirically validated by the experiments shown in Section 4.6.

Adder-based technique. We now sketch an adder-based algorithm for the `AOverflowBit` unit. This variant does not use a table, hence it does not require the instantiation of a RAM (it can be fully implemented using registers and combinatorial logic), and moreover its size is $\mathcal{O}(k)$ instead of $\mathcal{O}(2^k)$ of the table-based gadget. However, the adder-based gadget has a minimum latency of $k + 1$

[3]Let us remark that, beyond the probing model, we should avoid horizontal attacks that recover r, that is, re-generate the table C_A often enough that an adversary is not able to recover r.

Algorithm 12 `AOverflowBit-Adder`

Require: Shares $s_0, s_1 \in \mathbb{Z}_K$, with $K = 2^k$.
Ensure: Shares $c_0, c_1 \in \mathbb{Z}_2$ such that $c_0 \oplus c_1 = (s_0 + s_1) \gg k$.
1: $(x_0[0], x_1[0]) \leftarrow (0, 0)$
2: **for** $i = 0, \ldots, k-1$ **do**
3: $(y_0, y_1) \leftarrow (s_0[i], s_1[i])$
4: $(z_0, z_1) \leftarrow (s_0[i] \oplus x_0[i], x_1[i])$
5: $(t_0, t_1) \leftarrow$ `AND-HPC2`$((y_0, y_1), (z_0, z_1))$
6: $(x_0[i+1], x_1[i+1]) \leftarrow (t_0 \oplus s_0[i], t_1)$
7: **end for**
8: $(c_0, c_1) \leftarrow (x_0[k], x_1[k])$
9: **return** c_0, c_1

clock cycles, in contrast with the 3 cycles for Algorithm 11 (assuming one cycle latency for RAM access).

When $K = 2^k$, we remark that $(s_0, 0)$ and $(0, s_1)$ are two k-bit (non-uniform) Boolean shares of s_0 and s_1, respectively. The masked overflow bit can then be obtained with the Boolean-masked k-bit adder described in Algorithm 12. We use the ripple-carry adder of [BC22], and adapt it to the hardware design context by unrolling the loops and using the HPC2 masked hardware AND gadget [Cas+21; CS21], instead of the software PINI1 [CS20].

Algorithm 12 instantiates k carry-generating units (i.e., full-adders that compute only the carry-out) whose inputs are the shares $a = (s_0[i], 0)$, $b = (0, s_1[i])$ and $c = (x_0[i], x_1[i])$. The carry-out shares are computed as $a \oplus ((a \oplus b)\,\&(a \oplus c))$. By using the AND gate in this expression with a `AND-HPC2` gadget, we achieve a latency of one cycle w.r.t. the carry-in bit c (and two cycles with respect to the input a), leading to a total adder latency of $k + 1$ cycles. Since Algorithm 12 is a straight port to the hardware of the masked adder of [BC22], its correctness proof trivially carries over, and our choice of AND gadget ensures that the security proof also remains valid (and covers glitches), since HPC2 satisfies the PINI security property, like PINI1 (see [Cas+21] for details). Finally, `AOverflowBit-Adder` can be generalized to non power-or-two K by using the modular adder of [Cas+21].

4.4.3 Hybrid masking of scalar linear functions

The `HybridLin` gadget (Algorithm 14) computes $L(x)$ where L is a linear function (e.g., multiplication by a constant value) that maps $\{0, \ldots, K-1\}$ to the group $\mathbb{Z}_q$. The gadget takes as input hybrid shares and outputs uniform arithmetic shares in $\mathbb{Z}_q$. The core idea of its construction is a masked Mux gadget (Algorithm 13) that takes as input the Boolean shares $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ of length-l vectors and outputs uniform arithmetic shares $(\boldsymbol{v}_0, \boldsymbol{v}_1)$ of $K \cdot L(1)$ if $\boldsymbol{c} = 1$, or 0 if $\boldsymbol{c} = 0$.[4] Note that operations presented in Algorithm 13 are generic for vectors of length-l. For polynomials,

[4]Here, we select between two public values, but the masked multiplexer can be extended to select from more arithmetic shares by simply duplicating the logic: select v_0 as either a_0 or b_0 depending on (c_0, c_1), and duplicate that circuit to select v_1 as a_1 or b_1.

Algorithm 13 `MskMux` (multiplexer with the masked select bit.)

Require: Boolean sharing $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ with $\boldsymbol{c}_0, \boldsymbol{c}_1 \in \mathbb{Z}_2^l$ representing $\boldsymbol{c} = \boldsymbol{c}_0 \oplus \boldsymbol{c}_1$, $\boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathbb{Z}_q^l$.
Ensure: $\boldsymbol{z}[i] = \boldsymbol{c}[i] \,?\, \boldsymbol{r}_2[i] : \boldsymbol{r}_1[i]$ for $i = 0, \ldots, l-1$.
1: **for** $i = 0, \ldots, l-1$ **do**
2: $\quad \boldsymbol{u}_0[i] \leftarrow \texttt{Reg}\,(\boldsymbol{c}_1[i] \,?\, \boldsymbol{r}_2[i] : \boldsymbol{r}_1[i])$
3: $\quad \boldsymbol{u}_1[i] \leftarrow \texttt{Reg}\,(\boldsymbol{c}_1[i] \,?\, \boldsymbol{r}_1[i] : \boldsymbol{r}_2[i])$
4: $\quad \boldsymbol{z}[i] \leftarrow \texttt{Reg}\,(\boldsymbol{c}_0[i] \,\&\, \boldsymbol{u}_1[i]) \mid \texttt{Reg}\,(\texttt{!}\boldsymbol{c}_0[i] \,\&\, \boldsymbol{u}_0[i])$
5: **end for**
6: **return** $\boldsymbol{z}$

Algorithm 14 `HybridLin` (linear operation on hybrid shares.)

Require: Hybrid sharing (s_0, s_1, c_0, c_1) with $s_0, s_1 \in \mathbb{Z}_K$, $c_0, c_1 \in \mathbb{Z}_2$ representing the secret $s \in \mathbb{Z}_K$.
Require: Linear function $L : \mathbb{Z}_K \to \mathbb{Z}_q$.
Ensure: Arithmetic sharing (y_0, y_1) of $y = L(s)$ with $y_0, y_1 \in \mathbb{Z}_q$.
1: $r_1 \xleftarrow{\$} \mathbb{Z}_q$
2: $r_2 \leftarrow (K \cdot L(1)) + r_1 \mod q$
3: $v_0 \leftarrow -\texttt{MskMux}((c_0, c_1), r_1, r_2)$
4: $v_1 \leftarrow r_1$
5: $y_0 \leftarrow L(s_0) + v_0 \mod q$
6: $y_1 \leftarrow L(s_1) + v_1 \mod q$
7: **return** (y_0, y_1)

$l = n$, and for scalars, $l = 1$. Then, L can be applied to the arithmetic input shares s_0 and s_1, leading to the output sharing $(L(s_0) + v_0, L(s_1) + v_1)$.

Proposition 5. *If $(s_0, s_1, c_0, c_1) \in \mathbb{Z}_K^2 \times \mathbb{Z}_2^2$ is a hybrid sharing of $x \in \mathbb{Z}_K$ and the function $L : \mathbb{Z}_K \to \mathbb{Z}_q$ is linear, then the output (y_0, y_1) of `HybridLin` (Algorithm 14) satisfies $y_0 + y_1 = L(x) \mod q$.*

Proof. We observe that v_0 is equal to $-r_1$ if $c_0 \oplus c_1 = 0$, and to $-r_2$ otherwise. Therefore, since $L(0) = 0$, the output of the masked mux (v_0, v_1) is an arithmetic sharing modulo q of $-K \cdot L(c_0 \oplus c_1)$, (y_0, y_1) is an arithmetic sharing of $L(s_0 + s_1 - K(c_0 \oplus c_1)) = L(x)$. □

Let us remark that Algorithm 13 includes registers (`Reg`()), that are needed to ensure that hardware glitches do not reduce the security order of the gadget, as discussed in the next proof. Additional registers may be used in the implementation (e.g., to allow pipelining). Besides, the output of `HybridLin` (Algorithm 14) is a uniform sharing in the (typically larger) group $\mathbb{Z}_q$. This is because $L(x)$ is typically not small anymore, hence having a hybrid sharing as the output of the gadget would not be efficient. Next, we prove the security of Algorithm 14 in the glitch-robust probing model [Fau+18].

Proposition 6. *`HybridLin` is first-order glitch-robust probing secure.*[5] *Furthermore, its output is a uniform arithmetic sharing in $\mathbb{Z}_q$.*

[5]We actually prove that the gadget is a glitch-robust probe isolating non-interference (PINI) [CS20], which allows for easy composition of this gadget with other gadgets.

Proof. We will prove that any glitch-extended probe is independent of the secret s if the input hybrid sharing is uniform. First, for every variable in the computation of $\boldsymbol{u}_0[i]$ and $\boldsymbol{u}_1[i]$ in Algorithm 13, the only share involved is $\boldsymbol{c}_1[i]$, which is independent of s, as a consequence of Proposition 2. Next, in the computation of v_0 in Algorithm 14, we remark that each $\boldsymbol{u}_0[i]/\boldsymbol{u}_1[i]$ in Algorithm 13 is uniformly distributed in $\mathbb{Z}_K$ and independent of $\boldsymbol{c}_1[i]$. Therefore, $\boldsymbol{c}_0[i] \,\&\, \boldsymbol{u}_1[i]$ and $!\,\boldsymbol{c}_0[i] \,\&\, \boldsymbol{u}_0[i]$ depend on the input share $\boldsymbol{c}_0[i]$ but each of these values is independent of $\boldsymbol{c}_1[i]$. Since only one of $\boldsymbol{c}_0[i]$ and $!\boldsymbol{c}_0[i]$ is non-null, a glitch-extended probe on v_0 depends on $\boldsymbol{c}_0[i]$ and observes either $\boldsymbol{u}_0[i]$ or $\boldsymbol{u}_1[i]$ but not both, therefore it is independent of $\boldsymbol{c}_1[i]$. Finally, a probe on y_0 depends on s_0 and $\boldsymbol{c}_0[i]$ (that is independent of s, by Proposition 2), and similarly, y_1 only depends on s_1, which is independent of s.

To prove the uniformity of the output sharing, it suffices to remark that r_1 is a fresh uniform random value in $\mathbb{Z}_q$, and $(y_0, y_1) = (L(s_0) - K \cdot L(c_0 \oplus c_1) - r_1, L(s_1) + r_1)$. □

The security against transitions [Fau+18] of `HybridLin` depends on its actual implementation, hence cannot be proven at an algorithmic level. Let us simply note that security against transition is trivially guaranteed if the implementation satisfies the pipeline definition of [CS21] (i.e., every logic gate is used once per execution). In more complex implementations, composition techniques of [CS21] may be used. Another solution is the use of formal verification tools such as maskVerif [Bar+19], SILVER [KSM20] or Coco [HB21].

4.4.4 Hybrid masking of vector linear functions

The `HybridLin` gadget operates on a single scalar, hence it is fairly limited. We now generalize it to the `HybrindLinV` (Algorithm 15) gadget that computes any vector linear function $L : \mathbb{Z}_K^l \to \mathbb{Z}_q^{l'}$ and takes as input a length-l vector of hybrid sharings. The core idea of this gadget is similar to `HybridLin`: compute $L(\boldsymbol{s}_0)$, $L(\boldsymbol{s}_1)$ and use `MskMux` to handle the computation of $K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1)$. We exploit the linearity of L to write the latter expression in the canonical form basis: $L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1) = \sum_{i=0}^{l-1} L(\boldsymbol{e}_i) \cdot (\boldsymbol{c}_0[i] \oplus \boldsymbol{c}_1[i])$ (see Algorithm 15 for the definition of $\boldsymbol{e}_i$). This expression can in turn be evaluated using l `MskMux` gadgets, each being l'-element wide.

On top of this base construction, we introduce an optimization to reduce randomness usage and save some computation. Indeed, the shares of output coordinates of `HybridLinV` do not have to be independent if no computation involves these together (this condition is often satisfied in lattice-based cryptography). Therefore, we can use the same randomness for all coordinates. For simplicity, we maintain a vector accumulator $\boldsymbol{V}_1$; however, during implementation, we use a scalar accumulator instead. Finally, since $\boldsymbol{v}_0$ is freshly randomized, we can compute the output sharing as $(\texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0)) + L(\boldsymbol{s}_1), \boldsymbol{v}_1)$, resulting in a common share for all output coordinates, which spares some logic.

Algorithm 15 `HybridLinV` (vector linear operation on hybrid shares.)

Require: Hybrid sharing $(\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{c}_0, \boldsymbol{c}_1)$ with $\boldsymbol{s}_0, \boldsymbol{s}_1 \in \mathbb{Z}_K^l$, $\boldsymbol{c}_0, \boldsymbol{c}_1 \in \mathbb{Z}_2^l$ representing the secret $\boldsymbol{s} \in \mathbb{Z}_K^l$.
Require: Linear function $L : \mathbb{Z}_K^l \to \mathbb{Z}_q^{l'}$.
Ensure: Arithmetic sharing $(\boldsymbol{y}_0, \boldsymbol{y}_1)$ of $\boldsymbol{y} = L(\boldsymbol{s})$ with $\boldsymbol{y}_0, \boldsymbol{y}_1 \in \mathbb{Z}_q^{l'}$.
1: **for** $i = 0, \ldots, l-1$ **do**
2: $\quad r \xleftarrow{\$} \mathbb{Z}_q$
3: $\quad \boldsymbol{r}_1 \leftarrow (K \cdot L(\boldsymbol{e}_i)) + \{r\}^{l'} \mod q$ $\quad \{\boldsymbol{e}_i[i] = 1$ and $\boldsymbol{e}_i[j] = 0$ for all $j \neq i\}$
4: $\quad \boldsymbol{V}_0[i] \leftarrow -\texttt{MskMux}((\{\boldsymbol{c}_0[i]\}^{l'}, \{\boldsymbol{c}_1[i]\}^{l'}), \{r\}^{l'}, \boldsymbol{r}_1)$
5: $\quad \boldsymbol{V}_1[i] \leftarrow \{r\}^{l'}$
6: **end for**
7: $\boldsymbol{v}_0 \leftarrow \sum_{i=0}^{l-1} \boldsymbol{V}_0[i] \mod q$
8: $\boldsymbol{v}_1 \leftarrow \sum_{i=0}^{l-1} \boldsymbol{V}_1[i] \mod q$
9: $\boldsymbol{y}_0 \leftarrow \texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0) \bmod q) + L(\boldsymbol{s}_1) \bmod q$
10: $\boldsymbol{y}_1 \leftarrow \boldsymbol{v}_1$
11: **return** $(\boldsymbol{y}_0, \boldsymbol{y}_1)$

Proposition 7. *If $(\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathbb{Z}_K^{2l} \times \mathbb{Z}_2^{2l}$ is a hybrid sharing of $\boldsymbol{x} \in \mathbb{Z}_K^l$ and the function $L : \mathbb{Z}_K^l \to \mathbb{Z}_q^{l'}$ is linear, then the output $(\boldsymbol{y}_0, \boldsymbol{y}_1)$ of `HybridLinV` satisfies $\boldsymbol{y}_0 + \boldsymbol{y}_1 = L(\boldsymbol{x}) \mod q$.*

Proof. We observe that $\boldsymbol{V}_0[i]$ is equal to $\{-r\}^{l'}$ if $(\boldsymbol{c}_0[i] \oplus \boldsymbol{c}_1[i]) = 0$, and to $-\boldsymbol{r}_1$ otherwise. Therefore, thanks to the linearity of L, $\boldsymbol{v}_0 = -K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1) - \boldsymbol{v}_1 \bmod q$. As a result, $(\boldsymbol{v}_0, \boldsymbol{v}_1)$ is an arithmetic sharing modulo q of $-K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1)$ (taking the vector $\boldsymbol{v}_1$ as the second share of all the elements). The final part of the proof is identical to the proof of Proposition 5. □

Let us now prove the security of `HybridLinV`.

Proposition 8. *`HybridLinV` is first-order glitch-robust probing secure.*[6] *In addition, each of its output coordinates is a uniform arithmetic sharing in $\mathbb{Z}_q$.*

Proof. The proof is almost identical to the proof of Proposition 6. The first difference is that a glitch-extended probe on $\boldsymbol{v}_0$ may reveal a coordinate of multiple $\boldsymbol{V}_0[i]$. This is not a problem since the computation of every $\boldsymbol{V}_0[i]$ uses fresh randomness, hence the arguments in the proof of Proposition 6 apply. The other difference lies in the output computation: every coordinate of $\boldsymbol{v}_0$ is a coordinate of $-K \cdot L(\boldsymbol{c}_0 \oplus \boldsymbol{c}_1)$ masked with the uniform random r. Therefore, each output coordinate is uniform sharing, and each coordinate of $\texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0) \bmod q)$ is uniform, hence $\texttt{Reg}\,(\boldsymbol{v}_0 + L(\boldsymbol{s}_0) \bmod q) + L(\boldsymbol{s}_1) \bmod q$ can be simulated knowing only $\boldsymbol{s}_1$. □

4.4.5 Polynomial multiplication with hybrid masking

In this section, we will use hybrid masking to compute the product $\boldsymbol{a} \times \boldsymbol{s}$ where $\boldsymbol{s}$ is a small secret polynomial and $\boldsymbol{a}$ is a large public polynomial.

[6] As for `Hybridlin`, `HybridLinV` is actually glitch-robust probe isolating PINI.

Algorithm 16 `HybridPolyMul` (hybrid shared polynomial multiplication with a public polynomial.)

Require: Hybrid sharing $(\boldsymbol{s_0}, \boldsymbol{s_1}, \boldsymbol{c_0}, \boldsymbol{c_1})$ with $\boldsymbol{s_0}, \boldsymbol{s_1} \in \mathbb{R}_K$, $\boldsymbol{c_0}, \boldsymbol{c_1} \in \mathbb{R}_2$ representing the secret polynomial $\boldsymbol{s} \in \mathbb{R}_q$.
Require: Public polynomial $\boldsymbol{a} \in \mathbb{R}_q$.
Ensure: Arithmetic sharing $(\boldsymbol{y_0}, \boldsymbol{y_1})$ of $\boldsymbol{y} = \boldsymbol{a} \times \boldsymbol{s}$ with $\boldsymbol{y_0}, \boldsymbol{y_1} \in \mathbb{R}_q$.
1: **for** $i = 0, \ldots, n-1$ **do**
2: $\quad r \xleftarrow{\$} \mathbb{Z}_q$
3: $\quad$ **for** $j = 0, \ldots, n-1$ **do**
4: $\quad\quad \boldsymbol{r}_1[j] \leftarrow (-1)^{\lfloor (i+j)/N \rfloor}(K \cdot \boldsymbol{a}[i]) + r \mod q$
5: $\quad$ **end for**
6: $\quad \boldsymbol{V}_0[i] \leftarrow -\texttt{MskMux}((\{\boldsymbol{c}_0[i]\}^n, \{\boldsymbol{c}_1[i]\}^n), \{r\}^n, \boldsymbol{r}_1)$
7: $\quad \boldsymbol{V}_1[i] \leftarrow \{r\}^n$
8: **end for**
9: $\boldsymbol{v}_0 \leftarrow \sum_{i=0}^{n-1} \boldsymbol{V}_0[i] \mod q$
10: $\boldsymbol{v}_1 \leftarrow \sum_{i=0}^{n-1} \boldsymbol{V}_1[i] \mod q$
11: $\boldsymbol{y}_0 \leftarrow \texttt{Reg}\,(\boldsymbol{v}_0 + (\boldsymbol{a} \times \boldsymbol{s}_0) \bmod q) + (\boldsymbol{a} \times \boldsymbol{s}_1) \bmod q$
12: $\boldsymbol{y}_1 \leftarrow \boldsymbol{v}_1 \bmod q$
13: **return** $(\boldsymbol{y}_0, \boldsymbol{y}_1)$

Let the small coefficients of the secret polynomial $\boldsymbol{s}$ be in $\mathbb{Z}_K$. Starting from arithmetic sharing of $\boldsymbol{s}$ in $\mathbb{Z}_K$, we transform it to a hybrid sharing using `AOverflowBit` (Algorithm 11). We then use the `HybridLinV` gadget (Algorithm 15) with multiplication by $\boldsymbol{a}$ as the linear function L. This algorithm uses two multiplications ($\boldsymbol{a} \times \boldsymbol{s_0}$ and $\boldsymbol{a} \times \boldsymbol{s_1}$) that can be computed using any polynomial multiplication algorithm (these can take advantage of the smallness of the coefficients of $\boldsymbol{s_0}$ and $\boldsymbol{s_1}$ to achieve better performance or area or both). Then, we have to perform the computation on Boolean shares: `MskMux` and additions. In the remaining part of this section, we introduce a small tweak (similar to [RB20]) to `HybridLinV` that exploits specific properties of the polynomial multiplication to ease the hardware implementation and security proof.

Our tweak, implemented in the `HybridPolyMul` gadget (Algorithm 16), is based on the following observation: let $L(\boldsymbol{s}) = \boldsymbol{a} \times \boldsymbol{s}$, `HybridLinV` computes $\sum_{i=0}^{n-1} \boldsymbol{a} \times (x^i \cdot (\boldsymbol{c_0}[i] \oplus \boldsymbol{c_1}[i]))$. Assuming that one term of this sum is computed and accumulated per clock cycle, this requires having the full $\boldsymbol{a}$ available at all clock cycles, as well as one $(\boldsymbol{c_0}[i], \boldsymbol{c_1}[i])$ single-bit sharing. If $\boldsymbol{a}$ is stored in a RAM and $\boldsymbol{c_0}, \boldsymbol{c_1}$ in registers (which is the case in the implementation that we use as the starting point for our implementation [RB20]), then it is better to perform the computation as $\sum_{i=0}^{n-1} \boldsymbol{a}[i] \cdot (x^i \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1}))$. Another advantage of this approach is that it manipulates in parallel all the shares instead of a single one, which typically leads to better side-channel security. In such an implementation, the computation of $x^i \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1})$ can be performed with a cyclic shift register (for negacyclic ring, with a bit to encode positive or negative sign).

For conciseness, we avoid repeating the proofs of correctness and security for `HybridPolyMul`, since they are identical to the ones of `HybridLinV`, except for the functional change explained above, which does not have any impact on the security proof.

Performance and Area Advantage To bring forth the advantage of this method, we will now present a comparison with the naive way of using small coefficient multipliers. We provided a hybrid gadget, but a naive approach to using a small coefficient multiplier is to split one of the coefficients into smaller parts. This technique is used in applications like homomorphic encryption. If we assume there is no hybrid gadget, then naive masking would split the small secret $\boldsymbol{s} \in \mathbb{R}_K$ into two large shares $\boldsymbol{s}_0, \boldsymbol{s}_1 \in \mathbb{R}_q$. Now, if we want to multiply these with the public polynomial $\boldsymbol{a} \in \mathbb{R}_q$ using the compact multipliers, one option would be to split $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$ base K (e.g., $\boldsymbol{s}_0 = \boldsymbol{s}_{0,r} \cdot K^r + \boldsymbol{s}_{0,r-1} \cdot K^{r-1} + \cdots + \boldsymbol{s}_{0,1} \cdot K + s_{0,0}$). Now, this requires r multiplications, where r is the number of words when a coefficient in $\mathbb{Z}_q$ is sliced into smaller words in base K ($r = \lceil(\log_2(q)/\log_2(K))\rceil$). When $K = 2^4$ and $q = 2^{13}$, r will be 4. Thus, we will require four multiplications per share, that is a total of eight multiplications for a first-order masked implementation. By using the proposed hybrid gadget, this is reduced to three multiplications only, where one of the multiplications is just an accumulation and does not require a multiplier. Instead, if a user decides to pay the cost in terms of area and save the runtime, then $\log_2(q) \times \log_2(q)$ multipliers would be required. These consume 4.5× more look-up tables compared to $\log_2(q) \times \log_2(K)$ multiplier and will require two multiplications. Thus, using hybrid shares gives us a significant advantage over state-of-the-art methods.

Higher-order generalization. We remark that the hybrid masking technique can be generalized to higher-order masking, with the following differences: with d shares $s_i \in \mathbb{Z}_K$, the overflow c has to be represented using multiple bits, and each of the d Boolean shares c_i is a $\lceil\log_2(d)\rceil$-bit word (i.e., $c_i \in \{0, \ldots, d-1\}$), such that

$$s = \sum_{i=0}^{d-1} s_i - K \cdot \bigoplus_{i=0}^{d-1} c_i.$$

The definition of uniform shares is unchanged: $(s_i)_{i=0,\ldots,d-1}$ must be a uniform sharing of s in $\mathbb{Z}_K$ and $(c_i)_{i=0,\ldots,d-1}$ must be a uniform Boolean sharing of c, independent of $(s_i)_{i=0,\ldots,d-1}$, conditioned on c. Note that the multiplication with the Boolean shares of the hybrid sharing (Algorithm 13) is expected to have $\mathcal{O}(d^2)$ complexity. Hence, for asymptotically high masking orders, this algorithm will have worse performance than directly operating in $\mathbb{Z}_q$ ($\mathcal{O}(d)$ cost). This also serves as an interesting future scope for the proposed technique.

In this Chapter, we focus on optimizing the first-order masking and leave the implementation of the higher-order as future work (note that `AOverflowBitAdder` already works at arbitrary order). We use the Table-based algorithm `AOverflowBit` since it has the lowest latency. Next, we will introduce an efficient hardware instantiation of `HybridPolyMul`.

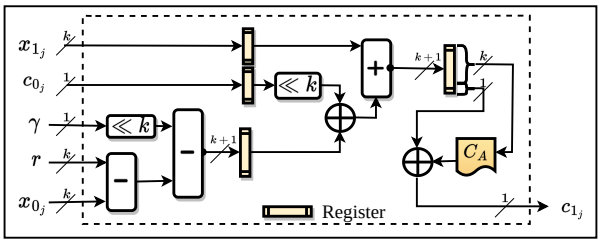


Figure 4.1: Architecture diagram illustrating the `AOverflowBit` unit

4.5 Implementation of masked unit

In Section 4.4, we presented an algorithmic optimization technique for masked and efficient implementation of polynomial multiplication in lattice-based public-key algorithms. An experimental evaluation is necessary to validate the efficiency of the proposed algorithms. In this section, we present a first-order DPA-protected hardware implementation of polynomial multiplication using the hybrid masking scheme.[7] This proof-of-concept architecture works in $\mathbb{R}_q = \mathbb{Z}_{2^{13}}/(x^{256}+1)$ and the unprotected secret polynomial's coefficients are in $[-4, 4]$.

The first step in multiplication is loading the secret polynomial $\boldsymbol{s}$, which has coefficients in the range $[-4, 4]$. We use a shifted representation where $\mu = 4$ is added to each coefficient, resulting in unsigned coefficients in the range $[0, 8]$ (this shift will be compensated later to preserve the correctness of the result). For the sake of simplicity, our implementation receives as input the uniform arithmetic shares $(\boldsymbol{s_0}, \boldsymbol{s_1})$ modulo K of the shifted $\boldsymbol{s}$, where $K = 2^4$ and $n = 256$. The initial sharing of this long-term secret is outside the scope of our implementation. As a second step, we convert the input arithmetic sharing into hybrid sharing by computing the Boolean shares $\boldsymbol{c_0}, \boldsymbol{c_1}$, using a pipelined implementation of Algorithm 11, as illustrated in Figure 4.1. Then, we implement the core part of the multiplication, as described in Algorithm 16.

Our implementation is based on the open-source hardware library provided by [RB20]. The polynomial multiplier in this library uses a heavily-parallel schoolbook algorithm by instantiating 256 small-area multiply-and-accumulate (MAC) units in parallel in the hardware for the polynomial ring $\mathbb{R}_q = \mathbb{Z}_{2^{13}}/(x^{256}+1)$. Moreover, it relies on the property that one operand polynomial is always small (e.g., with coefficients in $[-4, 4]$), and then it optimizes the MAC units to reduce the area consumption. This multiplier is, therefore, a natural starting point for our implementation since our hybrid-shares polynomial multiplication

[7] Note that the masking technique (non-hybrid) in Section 4.3 for NTT-based polynomial multiplication in NTT-unfriendly rings does not require a separate implementation and evaluation, as it does not lead to architectural challenges (we already give the implementation cost in Section 4.3 by adapting a well-known architecture). Regarding security, the NTT routine is evaluated twice in isolation on the two secret-share polynomials without requiring any interaction between the shares; therefore, no order reduction can happen if the state is properly cleared between the two multiplications.

algorithm is based on small-coefficient polynomial multiplications.

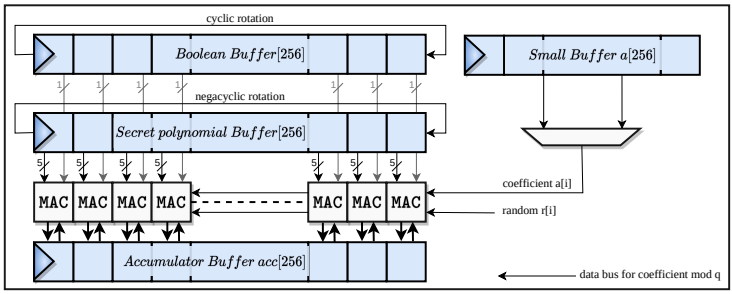


Figure 4.2: Polynomial multiplier architecture for the hybrid masking scheme.

Figure 4.2 shows the high-level architecture diagram of our schoolbook polynomial multiplier. The two main differences with the schoolbook multiplier of [RB20] are the changes in the MAC unit to support the Boolean-masked multiplication $\boldsymbol{a} \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1})$ on top of the normal multiplication (used for $\boldsymbol{a} \times \boldsymbol{s_0}$ and $\boldsymbol{a} \times \boldsymbol{s_1}$), and the addition of the Boolean secret polynomial buffer to store $\boldsymbol{c_1}$ ($\boldsymbol{c_0}$ is stored in the other secret polynomial buffer). The multiplier operates in three steps, following the requirements of Algorithm 16: (1) computation of $\boldsymbol{a} \times (\boldsymbol{c_0} \oplus \boldsymbol{c_1})$ (first share in the accumulator buffer, and the second share $\boldsymbol{v_1}$ in a separate buffer not shown on Figure 4.2), (2) accumulation of $\boldsymbol{a} \times \boldsymbol{s_0}$ in the accumulator buffer, and (3) accumulation of $\boldsymbol{a} \times \boldsymbol{s_1}$ in the accumulator buffer.

We start with the description of steps (2) and (3), which work identically to the unprotected multiplication in [RB20]. Before starting a multiplication, the entire polynomial $\boldsymbol{s_i}$ is loaded into the secret buffer of Figure 4.2. It is a shift register that performs a negacyclic rotation, computing $\boldsymbol{s_i} \leftarrow x \times \boldsymbol{s_i} \bmod (x^{256} + 1)$; the coefficients are stored in the buffer in signed magnitude representation. During the actual polynomial multiplication, in every cycle, one coefficient of $\boldsymbol{a}$ (say $\boldsymbol{a}[j]$) is multiplied by all coefficients of $\boldsymbol{s_i}$, and then the results are accumulated (addition or subtraction depending on the sign of $\boldsymbol{s_i}[j]$) in the accumulator buffer. In the next cycle, the secret polynomial buffer is shifted, and the process is repeated with the new coefficient $\boldsymbol{a}[j+1]$. For a 256-coefficient polynomial multiplication, 256 cycles are required, excluding the overhead of data loading. The accumulator buffer is *not* reset before or after the $\boldsymbol{s_i}$ multiplication, and the '`Boolean buffer`' is not used.

For the first step, we compute a sharing of $-\boldsymbol{a} \times (K \cdot \boldsymbol{c} + \{\mu\})$ where $\{\mu\}$ is a polynomial with all coefficients set to μ. It compensates for the original shift and proceeds as follows. The polynomial $(K \cdot \boldsymbol{c_0} + \{\mu\})$ is loaded in the regular secret polynomial buffer (the multiplication and addition is just a bit concatenation), and the Boolean polynomial $\boldsymbol{c_1}$ is loaded in the 256-bit buffer '`Boolean buffer`' which is also a circular shift register. To process the Boolean coefficients, the normal MAC unit of [RB20] is augmented with the secure Boolean unit of Figure 4.3, implementing the `MskMux` logic of Algorithm 16. This unit is integrated into the original MAC unit, with proper sign handling and registers to

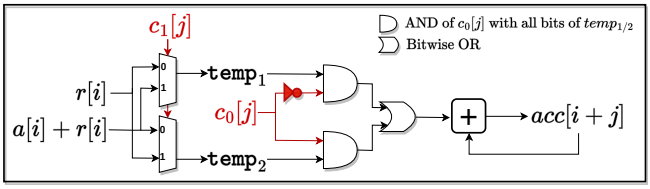


Figure 4.3: Secure multiply-and-accumulate unit for computing $acc[i+j] = acc[i+j] + c[j] \cdot a[i]$ using Boolean shares $c_0[j], c_1[j]$ of $c[j]$. Registers for glitches are not shown.

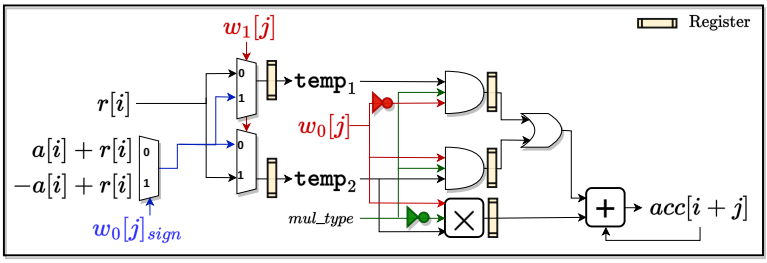


Figure 4.4: MAC (multiply and accumulate) unit architecture diagram. It computes multiplication for both Boolean as well as arithmetic shares.

prevent glitches, as shown in Figure 4.4. In this area-optimized combined unit, a signal `mul_type` selects between unmasked operation (for steps (2) and (3)) and masked operation.

Let us finally discuss the security of our implementation. We rely on the security arguments of Section 4.4 for `HybridPolyMul` and remark that our implementation optimizations do not break the security of this gadget. Indeed, the set of glitch-extended probes for the extended MAC unit is the same as for the basic one, and using the same register for $\boldsymbol{s_0}$ and $\boldsymbol{c_0}$ is not an issue since even jointly, these shares are still independent of the secret. Moreover, this unit is pipelined, and hence, the transitions occur between independently masked variables (except for the adder in front of the accumulator, which only adds old values of the accumulator to the extended probes, a process that is not an issue, as it involves only one of the two shares). Next, we discuss the results of the implementation.

4.5.1 Implementation results

We implemented the proposed architecture design on Xilinx Artix-7 xc7a100tftg256-3 FPGA using Xilinx Vivado 2019.1. The synthesis and implementation strategies were kept to Vivado defaults. The `KEEP_HIERARCHY` flag was set in the security-

Masking	Algorithm	Area (LUT/FF/DSP/BRAM)	Freq. (MHz)	Mul. Latency (clock cycles)
Proposed	AOverflowBit	590/1,014/0/1	130	92
Proposed	HybridPolyMul	25,214/13,280/0/1	130	1020
Naive	PolyMul	62,163/8,801/0/1	125	752
Unmasked	PolyMul	17,429/5,083/0/1	250	336

Table 4.1: Implementation results of the design on Xilinx Artix-7 xc7a100tftg256-3 for the parameters $n = 256, K = 2^4,$ and $q = 2^{13}$.

critical components of the modules to avoid unwanted optimizations. The complete design achieves a clock frequency of ≈ 130 MHz. The implementation of Algorithm 11 (AOverflowBit), illustrated in Figure 4.1, which generates the Boolean shares, consumes 590 LUTs, 1,014 FFs, 0 DSPs, and 1 BRAM. The C_A table (Equation 4.1) is pre-generated and stored before processing each polynomial; however, it can be refreshed more often. Its latency is 92 clock cycles for processing one set of polynomials. The complete core polynomial multiplication unit shown in Figure 4.2 consumes 25,214 LUTs and 13,280 FFs. It can process one polynomial multiplication in 340 clock cycles and consumes 1020 clock cycles to multiply the hybrid shares of a secret polynomial with a public polynomial. From Table 4.1, it is evident that the area-time increase of the proposed technique with respect to the unmasked implementation is much less compared to the naive masking (the secret is split into two shares, and the multiplication requires big multipliers).

Our masked polynomial multiplication implementation consumes 1.7× more area than the unmasked implementation in [RB20] (17,429 LUTs and 5,083 FFs). This is because we have instantiated $n = 256$ MskMux combined with 256 compact multipliers. Therefore, adding the registers to every MAC unit for glitch prevention further contributed to the area consumption. However, this is an implementation choice, and if one MskMux is instantiated for the multiplication with the Boolean shares, there will be no area overhead. Regarding time overhead, one unmasked polynomial multiplication translates to two multiplications and one accumulation in the masked domain. As discussed earlier (Section 4.4.5), the total area-time overhead of this design is less than the naive masked implementations.

4.6 Side-channel evaluation

In this section, we discuss the setup and results of the side-channel evaluation. Our setup is based on a ChipWhisperer NAE-CW305-04-7A100-0.10-X[8] board, which features an Artix-7 FPGA in the FTG256 Package, and an Atmel ATSAM2U32-AU microcontroller that acts as a USB to control the FPGA, as well as a trigger for starting the computations. We generate the bitsream with Vivado 2019.1. The FPGA uses a 10 MHz clock signal generated by the onboard PLL and crystal. The power consumption is measured with a 5 Ω shunt resistor, amplified on-board

[8] https://media.newae.com/datasheets/NAE-CW305_datasheet.pdf

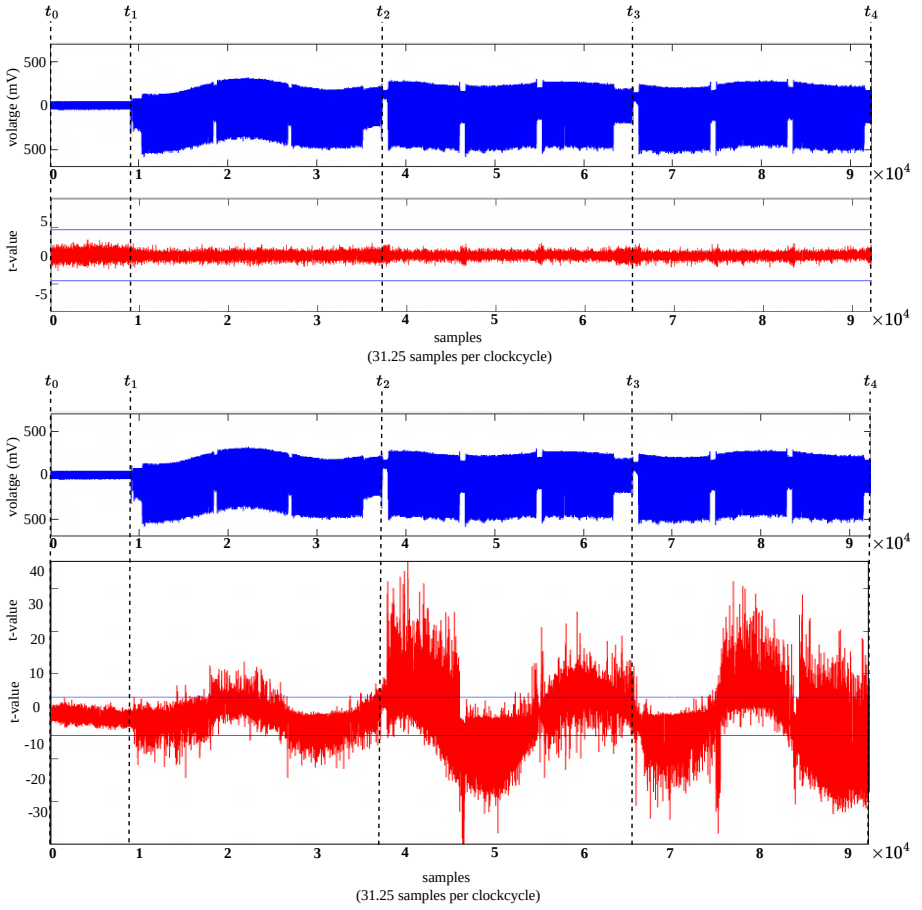


Figure 4.5: Raw measurement data of `AOverflowBit` and polynomial multiplication of the hybrid shares. **Top:** t-test figures with mask ON for 100,000 traces; **Bottom:** t-test figures with mask OFF for 200 traces.

with a 20 dB low-noise amplifier, and measured with the Picoscope 6000. The oscilloscope is synchronized with the FPGA clock and acquires 312.5 MS/s.

As mentioned in the previous section, our implementation is based on the Module-KEM design provided in [RB20]. It implements Saber, and the implementation can multiply several polynomials at once and accumulate them, depending on the polynomial vector dimension. For example, a public polynomial vector $\boldsymbol{A}$, of dimension three will have three polynomials $\boldsymbol{a}[0], \boldsymbol{a}[1], \boldsymbol{a}[2]$. When this is multiplied with the secret polynomials vector $\boldsymbol{S}$ of dimension three, the multiplication result of $\boldsymbol{A} \times \boldsymbol{S}$ is $\boldsymbol{a}[0] \times \boldsymbol{s}[0] + \boldsymbol{a}[1] \times \boldsymbol{s}[1] + \boldsymbol{a}[2] \times \boldsymbol{s}[2]$. Since we have an accumulator, we can keep accumulating the results on the fly for all three multiplications without needing to perform this operation separately, which avoids additional time overhead.

T-test evaluation. We measure first the `AOverflowBit` which generates shares $\boldsymbol{C_0}$ and $\boldsymbol{C_1}$ for polynomial vectors $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$, where both $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$ are secret vectors consisting of three polynomials. Then, this is followed by the multiplication of $\boldsymbol{C_0}, \boldsymbol{C_1}$ with the public polynomial vector $\boldsymbol{A}$ of dimension three. Next, we multiply $\boldsymbol{A}$ with $\boldsymbol{S_0}$ and finally with $\boldsymbol{S_1}$. A hardware implementation of the stream cipher Trivium is used as a source of randomness.

We employ the Test Vector Leakage Assessment (TVLA) method introduced in [Goo+11] to validate the security of our implementation. Namely, we run a first-order fixed-versus-random T-test with 100,000 traces for the secret shared variable $\boldsymbol{S}$ ($\boldsymbol{A}$ is kept fixed). The results are reported in Figure 4.5: no first-order leakage is detected for the commonly-used significance threshold of ± 4.5 for the T-statistic. As a sanity check, we also ran the first-order t-test with 200 traces and randomness masking disabled (Figure 4.5), which shows clear first-order leakage. In Figure 4.5, the timestamps $t_0 - t_4$ show the different operations evaluated by the test. The interval t_0–t_1 is the generation of shares $\boldsymbol{C_0}$ and $\boldsymbol{C_1}$ for secret vectors $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$. Then, the multiplication of $\boldsymbol{A}$ with $\boldsymbol{C_0}, \boldsymbol{C_1}$ is performed during the interval t_1–t_2. This is followed by multiplication of $\boldsymbol{A}$ with $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$, during the t_2–t_3 and t_3–t_4 intervals, respectively.

4.7 Conclusion and future work

In this work, we proposed an efficient masking method for polynomial multiplication of NTT-unfriendly schemes that can benefit from one of the operands being small. Not only does it enhance the masking of current NTT-unfriendly schemes, but it also facilitates the design of new schemes. The proposed hybrid gadget helps retain the small secret property and saves a significant area in masked implementations. Most lattice-based schemes employ small secrets (ternary, binomial, or Gaussian distributed). They can benefit from hybrid masking unless the scheme makes NTT representation an integral part of the scheme. Our contributions are therefore directly applicable to implementations of algorithms such as SABER [DAn+21], Scabbard [Mer+21], NTRU [HPS06] or LAC [Lu+18]. In the long term, our results should aid future lattice-based scheme design choices (in particular, the power-of-two versus NTT-friendly trade-off).

This work raises further research questions. A first direction that would be interesting to explore is the use of RSA/ECC coprocessors for masking lattice-based PQC. Since these are widely deployed, they are an attractive target for implementing PQC (especially when retrofitting PQC into existing systems). Non-masked implementations for Saber and Kyber have been proposed in [GMR21], using the Kronecker substitution method, and the most efficient variants exploit the property that one of the polynomials involved in the product always has small coefficients, both for NTT-friendly and unfriendly rings. Our hybrid gadgets (e.g. `HybridLinV` or `HybridPolMul`) could therefore be used, and they can simply use the algorithms of [GMR21] for the polynomial multiplication. For the `MskMux` (Algorithm 13) part of these gadgets, many possible software implementations depend on the available instructions: the most natural solution is the use of a

conditional move instruction, but arithmetic solutions are also possible (using 0/1 multiplications, then additions), or even using the bitwise logical operations (emulating our hardware implementation).

Secondly, we used a schoolbook multiplier for hybrid masking implementation (Section 4.5). However, we emphasize that this is not the only type of multiplication that can be used with this masking technique: other multiplication types, such as the Karatsuba or Toom-Cook multipliers, can also be used. The integration of these multipliers with the Boolean masking part of the hybrid masking, as we do with our integrated MAC unit, is a future direction for optimization work.

5

REED on the Path to FHE Realization

Breu

Until now, our focus has been on Post-Quantum public-key schemes for securing applications like network communication. However, with the growing reliance on compute-heavy workloads, ranging from cloud services to large language models, the world is rapidly shifting toward cloud-based computation for nearly every task. In this landscape, privacy concerns have reached an all-time high, forcing users to choose between functionality and their right to privacy. This is where FHE enters the picture. While FHE is theoretically elegant and promises privacy-preserving computation, its practical deployment remains hindered by extreme computational overhead.

To bridge this gap, this chapter presents the *first-of-its-kind* multi-chiplet-based FHE accelerator 'REED' for overcoming the limitations of prior monolithic designs. To utilize the advantages of multi-chiplet structures while matching the performance of larger monolithic systems, we propose and implement several novel strategies in the context of FHE.

Contribution: The following paper represents the core contributions of this chapter, for which the thesis author was the principal author, with the key NTT design contributed by Ahmet.

• *REED: Chiplet-Based Accelerator for Fully Homomorphic Encryption* by Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy, appeared in IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES 2025).

5.1 Introduction

Despite significant progress in the mathematical aspects of FHE, state-of-the-art FHE schemes typically introduce 10,000× to 100,000× slowdown [Jun+21c] compared to plaintext calculations. This overhead can be attributed to plaintext expanding into large polynomials when encrypted using an FHE scheme. Subsequently, simple operations, like plaintext multiplication, translate into complex polynomial operations. To bridge this performance gap, researchers have proposed acceleration techniques on various platforms, including GPU, FPGA, and ASIC [Bad+20; Bei+23; Fan+23; Fel+21; Gee+22; Jun+21a; Kim+22a; Kim+22b; Kim+23a; Mer+23; Nab+22; Rea+20; Ria+20; Roy+15; Roy+18; Sam+22; Sin+19; Tak+20; Wan+14; WH13; XZH21]. Software implementations offer flexibility, but often exhibit poor performance. Attempts have been made to provide GPU [Bad+20; Jun+21a] and FPGA-based solutions [Mer+23; Ria+20; Sin+19]. However, the performance gap remains two to three orders of magnitude compared to plain computation.

Currently, the fastest hardware acceleration results have been reported using ASIC modeling [Fel+21; Gee+22; Kim+22a; Kim+22b; Kim+23a; Sam+22]. The works propose utilizing large chip architecture designs with all FHE building blocks onto a single chip to maximize performance, hence monolithic. While simulations of these architectures demonstrate that they can achieve high performance for FHE workloads, the limitations of current manufacturing capabilities, such as inflexibility, low yield, and higher manufacturing costs [Gon21], impact their real-world deployment. For instance, the large architectures [Kim+22a; Kim+22b; Kim+23a] with area-consumption of approximately 400mm^2 result in a manufacturing yield of only 67% [Ma+22], a chip fabrication cost of over 25 million US$ [MUSb], and a long time-to-market (>3 years).

Additionally, several of these proposals overlook the crucial need for parallelizing communication and computation, as the off-chip to on-chip communication is slower than the chip's computation speed. Our analysis reveals that this feature is crucial in an FHE accelerator for achieving optimal performance when executing complex tasks, such as training neural networks. Prior works also utilize higher on-chip bandwidth due to readily available on-chip memory (20TB/s [Kim+22a], 36TB/s [Kim+23a], and 84TB/s [Sam+22]). Replacing this on-chip memory with cheaper HBM3 (1.2 TB/s bandwidth) would require 17 to 70 HBM3 modules to match the necessary bandwidth.

In summary, while the large, monolithic FHE architectures proposed in prior works show promise, they face practical challenges, including high manufacturing costs, low yield rates, and extended time-to-market. Addressing these challenges opens the door to exploring new approaches, such as chiplet-based architecture design. Chiplet-based architecture design utilizes multiple smaller chiplets instead of one large monolithic chip to realize a large system. Chiplets are modular building blocks that are combined to create more complex integrated circuits, such as CPUs, GPUs, Systems-on-Chip (SoCs), or System-in-Package (SiPs).

The transition to chiplet integrated systems represents both the present and future of architectural designs [Gon21; GPG23; Ma+22; Man22; Yin+18;

ZSB21]. In the DATE2024 keynote talk [Rob24], the speaker remarks how chiplet-based designs help 'push the performance boundaries, with maximum efficiency, while managing costs associated with manufacturing and yield'. Chiplet-based architectures also feature the advantage of tiling beyond the reticle limit ($858mm^2$) [GPG23] as multiple chiplets can be integrated for better performance. Although chiplet-based architectures enjoy the aforementioned advantages, they also face a trade-off between performance and yield. Multiple smaller chiplets offer high yields and reduced manufacturing costs, but at the same time, experience performance overhead due to slower chiplet-to-chiplet communication. Taking the advantages and challenges of chiplet-based systems into consideration, we are curious to investigate the following research questions:

How can we design and optimize a multi-chiplet accelerator for FHE that matches the performance of large monolithic FHE accelerators while overcoming the inherent challenges of monolithic designs?

To investigate the question mentioned above, we present REED, a multi-chiplet architecture for FHE acceleration. We propose a holistic design methodology that covers all aspects of FHE acceleration, from low-level building blocks to high-level protocols, and reduces the area to 43.9 mm^2 for one REED-chiplet. This includes the first scalable design methodology for one chiplet and ensures full utilization of chiplets for varying amounts of available off-chip data bandwidths. After finalizing an efficient design of one chiplet, we move to a data and task distribution study for multiple chiplets in the context of CKKS [Che+17] routines. To achieve this, we contribute novel strategies that enable long-term parallelism in computation and communication. Finally, we synthesize the proposed design methodology for ASIC and report application benchmarks. To the extent of our knowledge, this is the *first chiplet-based architecture for accelerating FHE*. Throughout this Chapter, we have followed Occam's razor, seeking the simplest solutions for the best results. We unfold our major contributions as follows:

- **Chiplet-based FHE accelerator:** We present a novel and cost-effective chiplet-based FHE implementation approach, which is inherently scalable[1]. The chiplets are homogeneous (i.e., identical), which reduces testing and integration costs. REED with 2.5D packaging surpasses state-of-the-art work SHARP_{64}[Kim+23a] with 1.9× better performance and 2× less development cost.
- **Workload division strategy:** The first step to realizing a multi-chiplet architecture is to develop an efficient disintegration strategy that helps us divide the workloads among multiple chiplets and reduces memory consumption. Hence, we propose an interleaved data and workload distribution technique for all FHE routines.

[1]Proof-of-concept implementation for multi-chiplet cycle accurate model, NTT/INTT unit, and Automorphism unit is open source and available at `https://github.com/aikata10/REED/`.

- **FHE-tailored efficient C2C communication:** Chiplet-based architectures suffer from slow C2C (chiplet-to-chiplet) communication. We address this by proposing the *first non-blocking ring-based inter-chiplet communication* strategy tailored to FHE. This mitigates data exchange overhead during the KeySwitch macro routine, thereby accelerating Bootstrapping (the most expensive FHE routine).
- **Scalable design:** To attain scalability by design, we propose a configuration-based design methodology such that the memory read/write and computational throughput are the same. Changing the configuration parameters allows the architecture to adapt to the desired area and throughput requirements. This also offers inherent communication-computation parallelism in the design of every chiplet.
- **Novel compute acceleration:** Furthermore, we present new design techniques for the micro-procedures of FHE- the number-theoretic transform (NTT) and automorphism (AUT). Our approach introduces Hybrid NTT, eliminating the need for expensive transpose operation and scratchpad memory. It is easily scalable for higher or lower polynomial degrees. Hence, other applications, such as zero-knowledge proofs, can also benefit from this, where transposition is expensive due to high polynomial degrees.
- **Application benchmark:** Finally, we choose parameters offering high precision and good performance. REED is the *first work to benchmark an encrypted deep neural network training*, showcasing practical and real-world impact. While CPU (24-core, 2×Intel Xeon CPU X5690 @ 3.47GHz) requires 29 days to finish it, REED 2.5D would take only 15.4 minutes, a realistic time for an NN training. We also utilize DNN training to conduct accuracy/precision experiments and validate our parameter choices.

Connection and comparison with chiplet designs for ML

While prior chiplet-based Machine Learning (ML) works address similar problems, our solutions are tailored to meet FHE requirements more effectively. For instance, [Sha+21a] addresses MCM's long tail-latency issue using non-uniform work distribution and communication-aware data placement. In the context of FHE, we resolve this by running parallel computations over extended periods, ensuring uniform task distribution and data placement. Our chiplet interconnections are ring-like and unidirectional in nature. Although we do not propose an automatic tool, our analysis, similar to [Tan+21b], focuses on long-term chiplet utilization based on FHE's computational-depth. Our methodology introduces a new configuration-based design built from scratch with novel building blocks and high-level protocols. In contrast to [Hwa+20], which combines heterogeneous chiplets, we propose homogeneous chiplets that observe the unique data flow of FHE. A common limitation of prior works is that they propose very small chiplet sizes (2 to 6 mm^2), which is too small according to a recent study by the authors in [GPG23]. Thus, we ensure that our chiplet sizes fall within the optimal range.

The necessary background on FHE and the CKKS scheme is reviewed in Chapter 2 (Section 2.2). This chapter is structured as follows. Section 5.2 recalls the design objectives for FHE hardware and summarizes the constraints of prior monolithic accelerators. Section 5.3 introduces the REED multichiplet architecture, including its data and task partitioning strategy and the interchiplet communication mechanism. Section 5.4 describes the internal microarchitecture of an individual REED chiplet, and Section 5.5 details how communication and computation are scheduled in parallel across chiplets. Section 5.6 presents synthesis, performance, and energy results, and Section 5.7 evaluates REED on representative encrypted machinelearning workloads. Section 5.8 sketches an extension towards 3D integrated designs, Section 5.9 concludes the REED work.

5.2 FHE Hardware design goals

A tiered structure exists in the CKKS scheme routines. The high-level or *macro* routines are `CKKS.Add`, `CKKS.Mult`, `CKKS.Rotate`, and `CKKS.KeySwitch`. These macro procedures apply micro procedures, such as forward and inverse Number Theoretic Transforms (NTT/INTT), dyadic Multiplication/Addition/Subtraction (MAS), and Automorphism (AUT). The NTT is used for multiplying two N coefficients long polynomials in $\mathcal{O}(N \log N)$ time complexity, which is the asymptotically fastest one.

The special `CKKS.Bootstrap` procedure uses these macro procedures in a specific sequence to refresh noisy ciphertexts. Note that contrary to schemes like FHEW, TFHE [Chi+20] where bootstrapping is a standalone procedure, CKKS-Bootstrapping is a high-level routine which utilizes KeySwitches, Automorphisms, and MACs. Therefore, while TFHE/FHEW accelerators (e.g., [Bei+23]) focus on optimizing the programmable bootstrapping, acceleration for CKKS relies on optimized KeySwitching, Automorphisms, and MACs. Among these operations, MAC is a straightforward linear operation. Automorphism involves permutation, followed by KeySwitch [HK19; KPP22]. This permutation, if naively implemented, can become complex and expensive as the input polynomial has N=2^{16} coefficients and offers N/2 different permutations. In this Chapter, we demonstrate how to design a permutation unit that is not only compact in terms of area but also has linear time complexity for all permutations. The final operation- KeySwitch, is the most expensive due to the expensive ModUp step (Figure 5.1). Since KeySwitch is the most expensive operation, the task and data distribution approach aims to optimize this particular operation. For simplicity, KeySwitch for $dnum = L + 1$ ($K = 1$) [HK19; KPP22] is utilized throughout the chapter.

5.2.1 Monolithic vs Chiplet packaging

In the context of large Integrated Circuits, authors in [Gon21; Ma+22; Yin+18] discuss the advantages of chiplet-based designs over monolithic designs. The problem with monolithic designs stems from the fact that to keep up with the increasing demand for high performance and functionality, chips need to be scaled

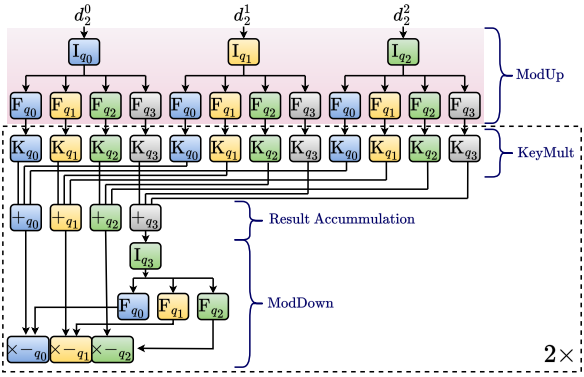


Figure 5.1: KeySwitch operation for $l = 2$, where I, F, and K represent INTT, NTT, and key multiplication operations using MAS, respectively.

up, and advanced technology nodes must be utilized. Manufacturing such large chips reduces wafer yield, as more surface area is exposed to defects per chip, and increases development costs. Such huge designs take a long time-to-market, and it is not straightforward to test and verify them. Factors such as size limitation and sub-optimal die performance due to overload contribute to a shift to SiP [Gon21].

In SiP, multiple heterogeneous smaller chiplets can be manufactured separately and later integrated together using various packaging techniques. This promotes chiplet-reuse, lowering the development costs. The chiplet-packaging techniques can be broadly classified into three main categories: 2D, 2.5D, and 3D [Gon21; Ma+22]. In 2D packaging, different dies are mounted on a substrate, known as a multi-chip module. Due to substrate limitations, this results in slow die-to-die communication and high power consumption.

To address these limitations, silicon interposers are used, a technique known as 2.5D integration [Pau+23; UCI23]. In this approach, an interposer is placed between the die and the substrate, allowing for die-to-die connections directly on the interposer itself. The use of an interposer significantly enhances interconnectivity, leading to improved performance. Taking the integration capabilities a step further, 3D packaging involves stacking different dies on top of each other, akin to a skyscraper. In 3D packaging, the dies are interconnected using through-silicon vias (TSVs). 3DIC is gaining significant popularity and serves as the foundation for advancements like High Bandwidth Memory (HBM/HBM2/HBM3) [Cha+23; Par+23; TGF09; Viv+20]. This approach significantly reduces the critical path, resulting in higher performance, lower power consumption, and increased bandwidth. The slowdown of Moore's law finds hope in 2.5D and 3D IC.

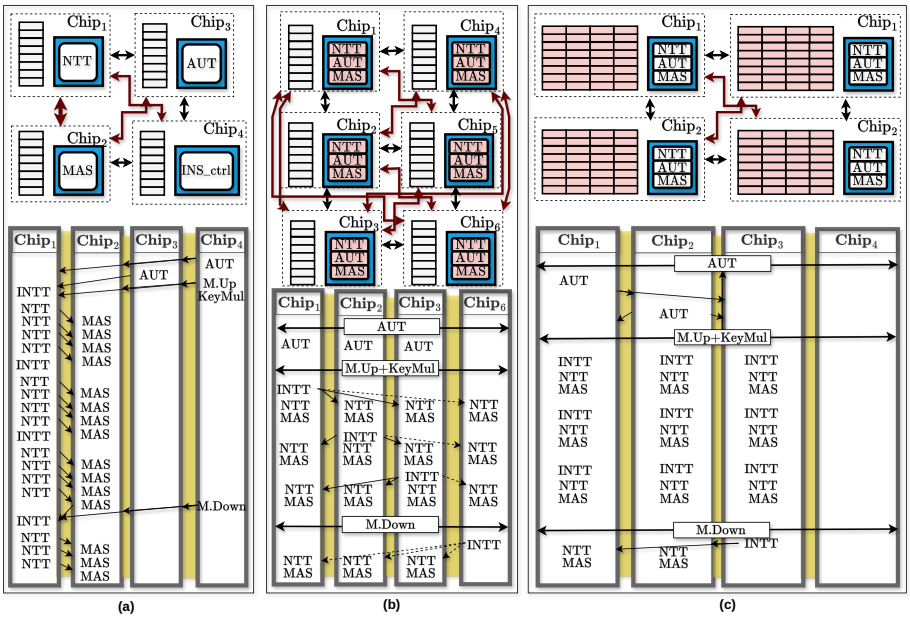


Figure 5.2: This depicts the different techniques, data, and task distribution for automorphism, followed by KeySwitch. $l = 2, 1$ for (a), (c) and 4 for (b)

5.3 FHE-tailored Multi-Chiplet Design

A widely adopted approach for realizing a chiplet-based architecture is to distribute the components of one large monolithic design across multiple chiplets connected in a mesh [Che+11; Kri+21; Res+]. While such a disintegration approach has found utility in applications like Machine Learning [Sha+21a], it falls short in leveraging the inherent algorithmic intricacies of FHE, thereby hindering efficient workload distribution among chiplets. In the following, we investigate the trade-offs of various chiplet design possibilities for FHE and consolidate on an optimal solution. For this, let us consider the most performance-heavy macro routine- KeySwitch. Its data flow is illustrated in Figure 5.1 for depth $l = 2$.

(a) The naive approach for chiplet decomposition would be to closely follow the data flow of Figure 5.1 and allocate one chiplet per square-box in the figure (I=INTT, F=NTT, K=Key Multiplication). This approach, as shown in Figure 5.2 (a), will lead to (*i*) uneven allocation of chiplet resources; for example, the chiplets for NTT or INTT will be much larger than those for MAS, (*ii*) a massive increase in C2C communication overhead due to the continuous data exchange between the components, and (*iii*) increase in required on-chip memory for data duplication. Figure 5.2 (a) further provides a schedule for the KeySwitch operation depicted in Figure 5.1. In addition to illustrating the complexity of chiplet-to-chiplet communication and task dependencies, it demonstrates how

most chiplets remain idle during computation, further emphasizing the imbalance in task distribution.

For a KeySwitch operation, the communication overhead is $(l+1)\cdot(l+2)$ polynomials for ModUp and $2\cdot(l+2)$ polynomials for ModDown. The time required for the operation is equivalent to the time taken to process $(l+1)$ INTT and $(l+1)\cdot(l+2)$ NTT operations for ModUp+KeyMul and 1 INTT and $(l+1)$ NTT for ModDown of each ciphertext component. This is tabulated in Table 5.1 and assumes that transfer can be done fast enough (monolithic) to compute MAS operations in parallel. If the 2× slow communication cost is added, considering chiplet disintegration, the communication would take more time and surpass the computation overhead by 2×, thus becoming the major bottleneck. Hence, this approach significantly inhibits the performance of the design. A deeper disintegration would imply breaking the individual operation into several chiplets, for example, computing one NTT using several chiplets. Although this approach can lead to smaller chiplets, it will result in additional C2C communication overhead.

(b) The second approach, illustrated in Figure 5.2 (b), involves assigning each RNS limb computation, as depicted in Figure 5.1, to a single chiplet encompassing NTT, AUT, and MAS components. Computations with respect to one modulus are performed within the same chiplet. This approach also faces slow and more complex C2C communication overhead due to data dependencies between the processing of RNS limbs in the KeySwitch routine. Furthermore, as the depth of the FHE application decreases over time, reducing the number of RNS limbs (l), their respective chiplets become idle.

Figure 5.2 (b) illustrates the complexity of C2C communication with six chiplets, showcasing the all-to-all dependencies between the chiplets. This figure highlights that before the chiplets can begin evaluating the NTT, they must wait for the INTT operation to complete, resulting in certain chiplets sitting idle during this time. This idle time reduces the overall efficiency of the design and contributes to increased communication overhead due to the frequent data exchanges required between the chiplets. If we assume that the result can be immediately sent to all the PU (monolithic), the runtime can be quantified as $l+1$ INTT and $2\cdot(l+1)$ NTT(+MAS) for ModUP+KeyMul, and 1 INTT and 2 NTT(+MAS) for ModDown of each ciphertext component, as tabulated in Table 5.1. The total data in communication for ModUp is $(l+1)\cdot(l+2)$ polynomials, as all $l+1$ INTT resultant polynomials are broadcast to $l+2$ chiplets. Similarly, for ModDown, the communication cost is $2\cdot(l+1)$ polynomials. Hence, in a chiplet disintegration scenario where communication between chiplets is 2× slower, it becomes the main bottleneck.

(c) A refined third approach, depicted in Figure 5.2 (c), involves enabling each chiplet to support multiple RNS limbs to reduce C2C communication overhead and utilizing data duplication. For instance, the first chiplet can initiate NTT_{q_1} after executing INTT_{q_0} without sending it to the second chiplet, as depicted in Figure 5.1. This strategy minimizes communication overhead during ModUp and increases intermediate storage requirements by storing copies of input RNS

	Polynomials in Communication	Computation			# Chiplets
		INTT	NTT	MAS	
Tech. (a)	$(l+3)\cdot(l+2)$	$l+3$	$(l+1)\cdot(l+4)$	||	4
Tech. (b)	$(l+1)\cdot(l+4)$	2	$l+4$	$l+4$	$L+2$
Tech. (c)	$(l+1)\cdot(l+4)$	$l+3$	$l+4$	$l+4$	$L+2$
Our Tech.†	$4\cdot(l+3)$	$\frac{l+1}{4}+2$	$\frac{(l+1)\cdot(l+4)}{4}$	||	4

Table 5.1: Comparison of the naive techniques and our technique for the KeySwitch operation, including ModUp, Key Multiplication, and ModDown. The operation count is the maximum reported by any chiplet, which will determine the overall throughput of the design. || Implies Operation is done in parallel with other computations.

† This technique is discussed in Section 5.5 for four chiplets ($r = 4$).

limbs d_2^0, d_2^1, d_2^2 in Figure 5.1 on each chiplet. INTT_{q_0}, INTT_{q_1}, NTT_{q_0}, etc., are computed within the same chiplet. As shown in Figure 5.2 (c), this technique does not completely eliminate the complexity of C2C interconnects, as some inter-chiplet communication remains necessary to ensure duplication. Additionally, this approach requires significantly more on-chip memory, as data must be duplicated across chiplets to enable the local processing of multiple tasks, further increasing resource demands. This technique reduces wait time during ModUp but requires an all-to-all broadcast at the end of the computation. Hence, requiring transfer of $(l+1)\cdot(l+4)$ polynomials, including broadcast required for ModDown.

Thus, all available approaches offer certain trade-offs and highlight communication as the major bottleneck; our aim is to determine the best and most practical solution. With this aim, we develop a chiplet-based design approach for REED.

5.3.1 REED 2.5D Architecture

In a chiplet-oriented design process, there are two critical choices: the size of chiplets and the number of chiplets. The manufacturing cost is reduced, and the yield increases when the chiplets are small in area. However, having many small chiplets reduces performance as the complexity of slow C2C communication increases. In this Chapter, we will develop a chiplet design strategy and integration topology, considering the data flow of FHE. This approach provides a balance between yield, manufacturing cost, and C2C communication overhead.

As an example, Figure 5.3 shows a four chiplet-based REED 2.5D architecture, where the chiplets are connected in a ring formation and have exclusive read/write access to HBM in their proximity. Later, we will show that this architecture scales well with an increasing number of chiplets (Section 5.6.4). To overcome the C2C communication overhead and memory storage issues, we propose an RNS polynomial (limb)-oriented task and data distribution strategy, which is built upon the third approach. Specifically, chiplets are assigned certain RNS limbs and all tasks related to these limbs without requiring data duplication

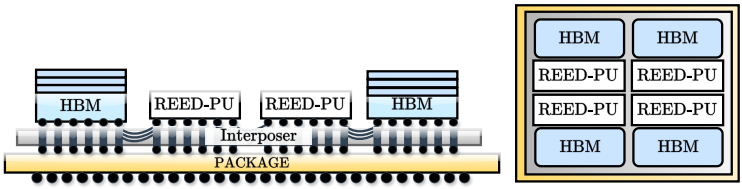


Figure 5.3: Side and top view of proposed four chiplet-based REED 2.5D.

(detailed in Section 5.5). The proposed ring formation (Section 5.5.4) allows us to increase the number of chiplets at the cost of only a linear increase in the number of interconnects. Hence, we can scale it to eight or sixteen chiplets as well. This ring formation for connecting the FHE chiplets is specifically tailored to the data-flow of performance-critical FHE workloads. With this formation, not all dies need to communicate with every other die simultaneously, which is crucial for minimizing C2C communication requirements.

Furthermore, our communication protocol ensures (Section 5.5.4) that no HBM-to-HBM communication is required. Hence, HBMs are positioned on the outer side. We also avoid sharing one HBM among multiple chiplets, ensuring that each HBM is located only in proximity to the one chiplet it serves. Notably, our placement strategy aligns well with [Pau+23; ZSB21], where authors design chiplet-based general-purpose processors with an actual tapeout, demonstrating practical viability. Finally, we also ensure a homogeneous design where all chiplets are identical, simplifying post-silicon realization.

Disintegration Granularity: Chiplet systems face a trade-off between development cost and performance degradation, depending on the disintegration granularity. Existing works on chiplet-based architectures, such as [GPG23; Ma+22; TGF09; Viv+20; Yin+18; ZSB21], show that disintegration improves yield, but it introduces challenges such as floorplanning and post-silicon testing overhead. Hence, the question:

How much disintegration is too much disintegration?

Considering a maximum die area of 800mm^2, dividing it into four chiplets offers an ≈80% yield, while eight chiplets provide a yield of ≈90%. The yield numbers are obtained from [Ma+22].

While the eight-chiplet option shows promise for achieving high yield, it faces the challenge of underutilization over time as the number of RNS limbs decreases after rescaling. Specifically, when l becomes smaller than 8, certain REED chiplets remain idle (Detailed in Section 5.5.3 and Section 5.6.4). On the contrary, the instantiation of four chiplets strikes an optimal balance between manufacturing cost and utilization. We want to remark that the number of REED chiplets is flexible and can be changed as per user requirements, depending on the technology and computation constraints.

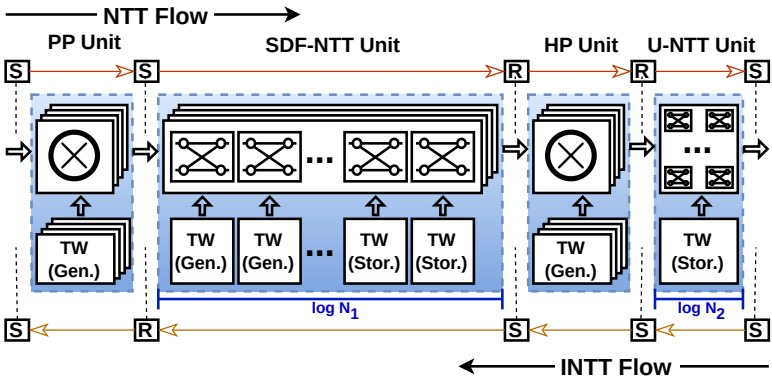


Figure 5.4: Novel routing-friendly Hybrid NTT/INTT design flow for $N = N_1 \times N_2$.

5.4 Architecture Design of One Chiplet

The need for scalability and high throughput drives our design methodology. We introduce the REED design configuration- (N_1, N_2) for polynomial degree N, where $N_1 \cdot N_2 = N$. For the clock frequency f, this configuration provides a throughput of $\frac{f}{N_1}$ operations per second and can process N_2 coefficients in parallel. A configuration-flexible design approach will help obtain computation-communication parallelism within every chiplet by ensuring that the memory read/write throughput is the same as the computational throughput. Now, let us explore how we design the ingredients of REED Processing Unit (PU) to ensure flexibility.

5.4.1 The Hybrid NTT (Frankenstein's Approach)

The NTT/INTT unit plays a crucial role in converting polynomials from coefficient to slot representation and vice versa. It is the most computationally expensive micro building block and occupies over 50% of the architectural area. Therefore, designing an efficient NTT/INTT unit is crucial as it directly impacts the overall throughput and area consumption of REED.

Prior works: There are various approaches in the literature to implement NTT in hardware for large-degree polynomials, such as iterative [Mer+23; Sco17a], pipelined [YCH22; Zha+21] and hierarchical [Fel+21], thoroughly discussed in Section 2.4.3. The implementation complexity of the plain iterative approach increases significantly with the number of processing elements. The pipelined approach (also known as single-path delay feedback (SDF)) provides a bandwidth-efficient solution but at the expense of diminished performance. The hierarchical approach (also referred to as four-step NTT), utilized in [Fel+21], treats a polynomial of size N as an $N = N_1 \times N_2$ matrix and divides a large NTT into smaller parts. It involves performing N_1-point NTTs on the N_2 columns of the matrix, then multiplying each coefficient by $\omega^{i \cdot j}$ (where i and j are matrix row

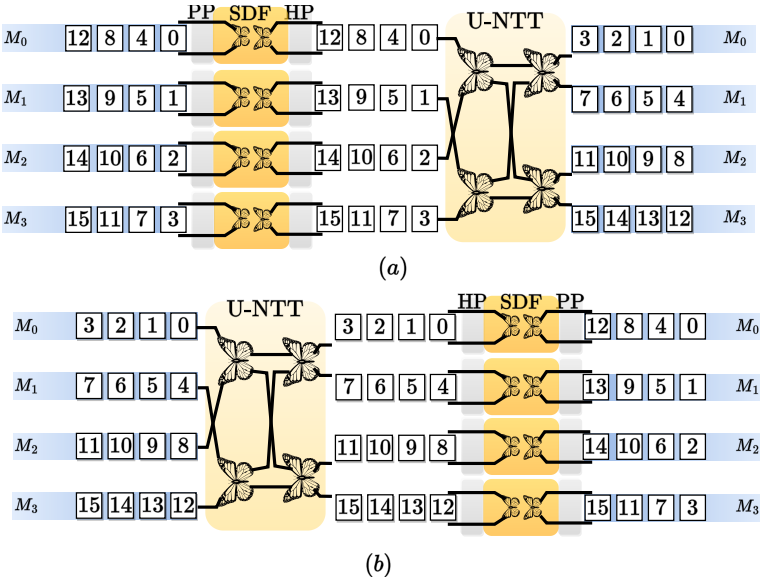


Figure 5.5: The proposed novel Hybrid NTT/INTT design flow with Memory access for (a) NTT and (b) INTT with $N_1 = 4$, $N_2 = 4$, and $N = 16$. The butterflies represent the Gentleman-Sande butterfly [Sco17a] operation employed in our design.

and column indices), transposing the matrix, and finally performing N_2-point NTTs on the N_1 columns.

Transposing a matrix of size $N_1 \times N_2$ requires N_1 separate memories and large data re-ordering units. Hence, in [Fel+21], the transpose unit consumes 14% of the area per compute cluster. Moreover, it also requires additional N_2 cycles for writing data to the transpose memory and N_1 cycles for reading it. Therefore, although the hierarchical approach simplifies the NTT implementation, we observe that it has the following limitations: (*i*) the costly transpose operation, (*ii*) fixed N_1 and N_2 such that $N_1 = N_2$ [Fel+21; Gee+22], and (*iii*) the reliance on scratchpad in some works leads to large memory fan-in and fan-out, causing routing inefficiencies.

Our Technique: We address these problems and propose a novel, scalable, and efficient NTT/INTT architecture. Our proposed design methodology, Hybrid NTT/INTT, divides large NTTs into smaller parts *that may differ in size*. To eliminate the transpose operation and reduce implementation complexity, we employ a different design approach that combines the two approaches mentioned above, hence the name *Hybrid* NTT/INTT. A transpose is required in the hierarchical implementation approach to feed the data from unrolled NTT units back to the same unrolled NTT units. This is also the reason they need to have the same configuration ($N_1 = N_2$). If, instead, we use pipelined NTT to replace

Algorithm 17 Hybrid NTT with NWC

In: a (a matrix of size $N_1 \times N_2$ in row-major order)
In: ω (N-th root of unity), ψ ($2N$-th root of unity)
Out: $\tilde{a} = \texttt{NTT}(a)$ (a matrix of size $N_1 \times N_2$ in column-major order)

1: **for** $(i = 0; i < N_1; i = i + 1)$ **do**
2: **for** $(j = 0; j < N_2; j = j + 1)$ **do**
3: $a[i][j] \leftarrow a[i][j] \cdot \psi^{i \cdot N_2 + j} \pmod q$ ▷ PP:Pre-processing
4: **end for**
5: **end for**
6: Apply N_1-pt NTT to the columns of a ▷ using SDF-NTT
7: **for** $(i = 0; i < N_1; i = i + 1)$ **do**
8: **for** $(j = 0; j < N_2; j = j + 1)$ **do**
9: $a[i][j] \leftarrow a[i][j] \cdot \omega^{i \cdot j} \pmod q$ ▷ HP:Hadamard prod.
10: **end for**
11: **end for**
12: Apply N_2-pt NTT to the rows of a ▷ Unrolled(U)-NTT
13: **return** a

one of the unrolled NTTs, no transpose will be required. This also results in design flexibility regarding N_1 and N_2 values.

We utilize this idea to design an NTT that is routing-friendly, throughput-oriented, and does not necessitate costly transposition. To achieve this, we utilize parts of hierarchical, iterative, pipelined, and plain unrolled NTTs (Frankenstein's approach) and introduce a novel Hybrid NTT. It is fully pipelined, and its flow is shown in Algorithm 17 and Figure 5.4. The input polynomial for NTT operation is stored in N_2 memories of depth N_1, hence forming a matrix of size $N_1 \times N_2$ in row-major order. The NTT unit is fully pipelined and reads N_2 coefficients from memories in each cycle. We illustrated the memory layout and read the pattern of a polynomial during NTT operation for $N = 4 \times 4$ in Figure 5.5. NTT unit first performs pre-processing (Step 3 of Algorithm 17) using N_2 modular multipliers (PP unit), where each coefficient is multiplied with the corresponding twiddle factor.

The resulting N_2 coefficients represent one coefficient from each column of the input matrix, which should be processed via the N_1-pt NTT operation as shown in Step 6 of Algorithm 17. To avoid intermediate memory communication and improve performance, we instantiate N_2 N_1-Point single-delay feedback (SDF)-based NTT units to perform N_2 NTT operations in parallel. Each SDF-based NTT architecture consumes and generates one coefficient per cycle after the pipeline is filled. It utilizes $\log_2 N_1$ cascaded butterfly units where each butterfly is coupled with a FIFO for data re-ordering [Zha+21].

After the SDF-NTT unit, the resulting coefficients are multiplied with powers of ω as shown in Step 9 of Algorithm 17. Similar to the PP unit, we use N_2 modular multipliers to perform this Hadamard product (HP unit). After this

step, the hierarchical approach would require a transpose operation. To eliminate the transpose, we employ a fully unrolled N_2-Point NTT architecture (N-NTT unit) that takes N_2 coefficients as input per cycle and generates N_2 coefficients as output per cycle, performing a cost-free natural transpose operation. It does not incur an extra run-time cost as it is merged with the final step (N_1 N_2-pt NTTs). Thus, the proposed Hybrid design reads N_2 coefficients per cycle and generates N_2 coefficients per cycle after filling the pipeline. It is scalable and enables different area/performance configurations by changing N_1 and N_2 values. This unit features bi-directional flow, and based on the configuration parameters, it provides a throughput of $\frac{f}{N_1}$. Overall, the properties and advantages of the proposed unit are as follows.

- **Transpose elimination:** The Hybrid NTT eliminates transpose by using two orthogonal NTT approaches, a pipelined (SDF) approach for N_1-sized NTTs and an unrolled (U-NTT) approach for N_2-sized NTTs. As shown in Figure 5.5 (a), the output coefficients of SDF-NTT are processed directly by U-NTT, providing a seamless, natural transpose operation.
- **Bi-directional workflow:** The above method of transpose elimination also helps make our NTT unit bi-directional (for INTT), as illustrated in Figure 5.5 (b). The additional routing complexity is balanced with efficient pipelining.
- **Low-level optimizations:** For modular multiplication and reduction unit, we adopted the word-level Montgomery [Mon85; MÖS19] modular reduction algorithm. and optimized it for our special prime form, $2^{w-1} + q_H \cdot 2^m + 1$, where $m = 18$ is Montgomery reduction size, and $\lceil \log_2 q_H \rceil = 10$ is small. A total of $(N_2 + 1)\log_2(N1) + \frac{N_2}{2}(\log_2(\frac{N_2}{2}) + 5) - 7$ modular multipliers are utilized.
- **On-the-fly twiddle generation:** We employ commonly used on-the-fly twiddle generation with a small constant memory that stores a few initial roots of the unity. This helps reduce the on-chip constant storage by up to 98.3%.

One REED-NTT requires $((N_2 + 1) \cdot \log_2(N_1) + (\frac{N_2}{2}) \cdot \log_2(\frac{N_2}{2}) + (5 \cdot \frac{N_2}{2}) - 7)$ multipliers. The proposed transpose-less NTT also finds applications in other cryptographic schemes. For example, in Zero-Knowledge proofs [Lu+23], where higher polynomial degrees make transposing more expensive, our NTT design offers a better area-time tradeoff.

5.4.2 Multiply-Add-Subtract (MAS) and Automorphism (AUT)

MAS is elementarily designed as a *triadic* unit for computing point-wise multiplication, addition, subtraction, or multiply-and-accumulate operations. It utilizes the modular multiplier proposed for the Hybrid NTT unit. A high-level overview

Algorithm 18 Automorphism

In: $a[N_1][N_2], gle$
Out: $\hat{a} = \rho(a)$

1: $index \leftarrow gle$
2: **for** $(l_0 = 0; l_0 < N_1; l_0 = l_0 + 1)$ **do**
3: $\quad l_1 \leftarrow index \pmod{\log(N_1)}$
4: $\quad start \leftarrow index \gg \log(N_1)$
5: $\quad addr[j] \leftarrow (start + j \cdot gle) \pmod{\log(N_2)} \; \forall \; j \in [0, N_2)$
6: $\quad \hat{a}[l_1] \leftarrow$ `shuffle_tree_`$N_2 \times N_2(addr, a[l_0])$
7: $\quad index \leftarrow index + gle$
8: **end for**
9: **return** $\hat{a}[N_1][N_2]$

of the MAS unit is shown in Figure 5.6. On the other hand, designing an efficient AUT unit is challenging [Fel+21; Sam+22]. It permutes ciphertexts using the Galois element (gle) to achieve rotation or conjugation. A polynomial is stored as a matrix $N_1 \times N_2$ in N_2 memories.

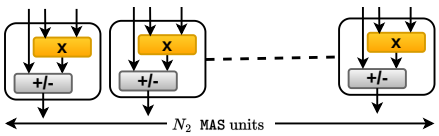


Figure 5.6: The set of triadic MAS units.

For AUT, we make a key observation that when we load N_2 coefficients from memory address l_0 across all N_2 memories, they are shuffled based on the desired rotation offset (ρ_{rot}), and then written to address l_1 across all N_2 memories. Hence, even though the coefficient order is shuffled, they all go to the same address of N_2 distinct memories. We utilize this property to permute all N_2 coefficients in parallel. This out-of-place automorphism is presented in Algorithm 18. The in-place permutation techniques proposed in previous works [Fel+21; Sam+22] increase routing complexity due to memory transposition requirements. We are still left with a quadratically complex and expensive shuffle $\mathcal{O}(N_2^2)$ among the coefficients. However, we analyzed that all the shuffles could be performed pairwise on the coefficient batches, as shown in Figure 5.7. After each stage, two batches of coefficients are merged to form a new batch. We replace the naive and expensive operation with a pipelined binary-tree-like shuffle. Its number of pipeline stages adjusts with N_2, making the pipelining scalable and efficient for higher configurations. Moreover, the unit can handle any arbitrary rotation.

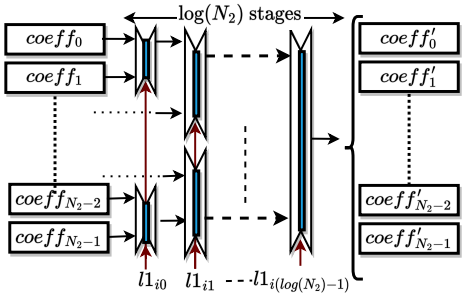


Figure 5.7: An example of a `shuffle_tree`_$N_2 \times N_2$ workflow. Every stage has sufficient registers to hold N_2 coefficients.

5.4.3 PRNG-Based Partial Key-Switching Key Generation

A PRNG is deployed to generate half the key components on the fly. The idea here is that the $ksk_1 = a$ component of the key is generated by expanding a public seed and is assumed to be in NTT form, meaning it does not undergo an NTT transformation. Therefore, instead of storing the complete polynomial, the cloud or server can store the seeds of the key and generate them on the fly when needed. Note that the polynomials are generated in RNS form and have no interdependency. In other words, for ciphertext at depth l, the $(l+1) \cdot (l+2)$ seeds are expanded, and the remaining limbs are ignored.

Our implementation uses the Trivium unit for PRNG similar to [Mer+23], which takes a 64-bit seed as input and initializes over 18 clock cycles, performing 18 rounds of operations. After initialization, the Trivium core generates 64-bit pseudorandom data continuously, producing one word per cycle without stalling. Each PU is equipped with one Trivium unit. The keys are only used for Key Multiplication during Key Switching and are directly fed to the MAS unit, which simultaneously consumes the pseudorandom words generated by the Trivium cores. The on-the-fly generation of ksk_1 substantially reduces the memory footprint for key switching. By generating it during runtime, we cut the required bandwidth (HBM-to-chiplet) and storage space for the key-switching key by 50%. This dynamic approach optimizes memory usage and increases the efficiency of the system for key-switching operations. With this, we conclude the design of micro procedures.

5.4.4 Programmable Instruction-Set Architecture

In this section, we discuss how to program the micro procedures (NTT, AUT, MAS) for high-level FHE routines. This is crucial for determining their placement in the architecture, which will be discussed in the subsequent section.

Prior works define a strict operation flow. This prevents adaptations to future changes in the FHE algorithms or routine flow. Noting this, we utilize an instruction-based architecture design technique [Sin+19], wherein a relatively

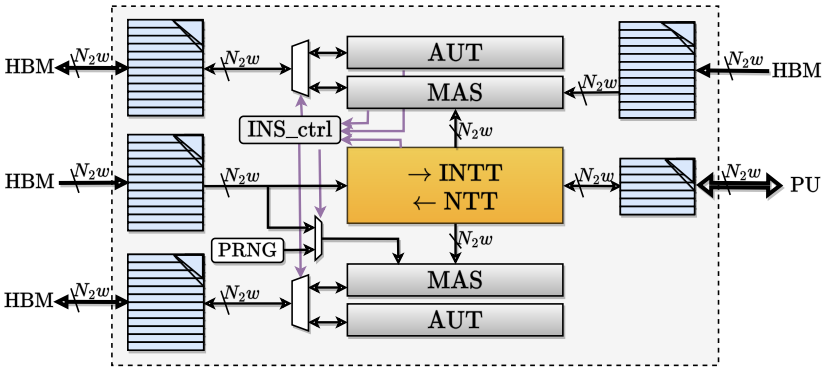


Figure 5.8: The REED-PU design. Every data communication (memory to building blocks and off-chip to on-chip) here has a bandwidth of N_2w bits/clock-cycle.

small instruction controller programs the REED-PU, manages the multiplexers, and collects 'done' signals from these units. The instruction controller includes a small BRAM for storing instructions. These instructions are provided at the start of the computation, along with the input data. Once the user sends an execute command, the instruction controller manages the execution of all stored instructions. This setup gives users the flexibility to specify any desired set of instructions, such as only performing NTT processing, and allows customization of parameters. Two types of instructions handled by the instruction controller are *micro* and *macro*. Micro-instructions are low-level arithmetic procedures, such as NTT, INTT, point-wise modular addition, subtraction, multiplication, multiplication-and-accumulation, and automorphism. They are used to compose microcodes for realizing macro-instructions for homomorphic addition/multiplication, KeySwitch, rotation, and moddown. A bootstrapping is performed using these macro instructions.

We initiate the PU design, as shown in Figure 5.8, and uncover two important design decisions. The first is regarding the placement of NTT/INTT and MAS/AUT units. The second deals with the problems associated with large on-chip memories utilized in prior works [Kim+22a; Kim+23a] for storing keys.

5.4.5 REED Processing Unit (PU)

We initiate the PU design, as shown in Figure 5.8. A PRNG is deployed to generate half the key components on the fly [Mer+23]. We uncover two important design decisions. The first is regarding the placement of NTT/INTT and MAS/AUT units. The second deals with the problems associated with large on-chip memories utilized in prior works [Kim+22a; Kim+23a] for storing keys.

❶ The polynomial processed by the NTT unit is multiplied by two polynomials of the KeySwitch keys and accumulated. Hence, we instantiate a pair of MAS/AUT units capable of simultaneously processing both key components. Thus, the

design enables the concurrent operation of NTT and MAS units (as shown in Figure 5.9), which improves KeySwitch performance by 66.7%, as explained in Section 5.4.6. Moreover, since the AUT and MAS units are relatively cheaper, this design decision also does not add significant area overhead, as presented later in Table 5.3.

❷ In hardware accelerators, on-chip memory causes significant area overhead. Chiplets with large on-chip memory are not power, area, and manufacturing cost efficient [Sie14]. In the context of FHE, each KeySwitch key demands ≈1MB or 91MB storage for $dnum = L + 1$, and $L = 30$ or $dnum = 3$, and $L = 22$ respectively. Given the limited on-chip memory capacity of small chiplets, accommodating even a single KeySwitch key becomes rather challenging. Consequently, reliance on off-chip memory access becomes essential when a KeySwitch operation necessitates a different key. It is also not useful to store just one relinearization key required after ciphertext multiplication, as for most applications, more rotations are required compared to multiplications [AR24]. Therefore, in our architecture, we store all the keys in the large off-chip HBM. To reduce the overhead of off-chip memory access, we develop an efficient prefetch unit that streamlines data movements in parallel to computation, as described next.

5.4.6 Throughput Computation for KeySwitch

The KeySwitch, detailed in Algorithm 6, is the most expensive operation among all the routines. For $dnum = L+1$, it transforms all $L+1$ residue polynomials from slot to coefficient representation (INTT), and then each of these is transformed to $L + 2$ NTTs, multiplied with two key components, and accumulated. This requires $L + 1$ INTTs, $(L + 1)(L + 2)$ NTTs, and $2(L + 1)(L + 2)$ MAS, making the naive throughput of this operation: $\frac{f}{(L+1)(1+3(L+2))\cdot N_1}$. By utilizing REED's parallel processing capability to perform all MAS operations concurrent to the NTT operations (shown in Figure 5.9), we save $2(L + 1)(L + 2)$ clock cycles and increase the throughput to $\frac{f}{(L+1)(L+3)\cdot N_1}$, resulting in a 66.7% improvement.

5.4.7 Streamlined Prefetch for On-Chip Storage

As mentioned in Section 5.4.1, REED's design methodology mitigates the need for scratchpad-like on-chip memory, allowing us to use memory units solely as prefetch units.

Each memory unit in Figure 5.8 exhibits balanced fan-in and fan-out, and among the five memory units depicted, four are fed by off-chip memory. The small memory is responsible for storing and communicating the INTT result to the other PUs (or Chiplets) (elaborated in Section 5.5.4). Only two of the four memories communicating with off-chip memory need to write back the results, as illustrated by bi-directional arrows in Figure 5.8.

Summarily, three memories perform off-chip read/write communication. These memories are physically divided into two parts. When one is utilized for on-chip computation, the other performs off-chip prefetch (similar to ping-pong caching).

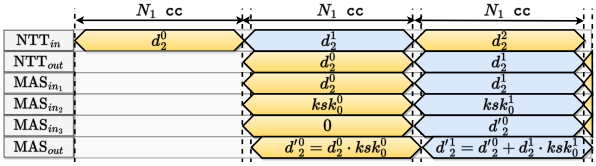


Figure 5.9: Timeline of parallel and pipelined operation flow.

5.5 Techniques for exploiting Comm-Comp Parallelism

The previous section discussed how a configuration-based design methodology enables parallelism in both off-chip and on-chip communication and computation. However, when distributing FHE workloads among multiple chiplets, we must consider the C2C communication overhead. This is important as state-of-the-art HBM3 [Ram] features a bandwidth of 1.2TB/s, while the state-of-the-art C2C interconnect UCIe [Syn23; UCI23] only offers a bandwidth of 0.63 TB/s. Consequently, a chiplet system optimized for HBM bandwidth could face bottlenecks due to slow C2C communication. The routine that necessitates most data exchange is the ModUp portion of KeySwitch, detailed in Algorithm 19. It switches the modulus of each $L+1$ residue polynomial ($\texttt{INTT}(d_{2q_i})_{q_i} \quad \forall\ i \in [0, L]$) to $(L+2)$ residues polynomials ($\texttt{NTT}(d_{2q_i})_{q_j}\ \forall\ j \in [0, L+1]$). The ModUp operation is followed by key multiplication and then the ModDown operation. This is the flow for $dnum = L+1$, and a more detailed discussion of our proposed technique for $dnum < L+1$ is provided in Section A.1.

5.5.1 Communication cost-analysis

In a multi-PU work [Mer+23], the authors briefly discuss limb-based decomposition and propose distributing computation across the RNS polynomials (limbs) by employing one PU per limb. While this approach enables highly parallel computations, as the multiplicative depth decreases, many PUs become idle, causing underutilization.

In [Kim+22a; Kim+22b; Kim+23a], the authors utilize both the limb-based and coefficient-based task distribution during KeySwitch. They [Kim+22a; Kim+23a] propose using limb-wise distribution for INTT and NTT steps and coefficient-wise distribution for the modulus switch (referred to as Base Conversion in the chapter). [Kim+22a] utilizes four PUs, and since this base conversion is needed between the INTT and NTT steps, all-to-all broadcasts are done across PUs to switch from one distribution to another. In the subsequent subsection, we analyse how both techniques attain the same communication overhead.

Additionally, the all-to-all C2C broadcast between the chiplets is slow and increases by $\mathcal{O}(\mathtt{r}^2)$ with the number of chiplets $\mathtt{r}$. In the context of FHE, switching between limb- and coefficient-wise task distribution becomes expensive, as it

requires all-to-all communication between cores (C2C). For example, [Kim+22a] proposes utilizing four PUs in a single monolithic chip. However, when extended to a chiplet setting, where each PU occupies a separate chiplet, the lack of an all-to-all broadcast capability makes it difficult to send data across all chiplets instantly. Using bi-directional C2C communication ability, the polynomial would reach all four chiplets via at least two serial C2C communication interfaces. The on-chip bandwidth used in the prior works is (20TB/s [Kim+22a], 36TB/s [Kim+23a]) is much less than the state-of-the-art C2C communication bandwidth (0.63TB/s [Syn23; UCI23]). As a result, chiplets would have to wait longer for data to arrive before computing, and this C2C communication overhead will significantly inhibit the performance. Hence, there is a need to devise a schedule that can couple most of the communication with computation.

5.5.2 Limb-wise vs Coefficient-wise distribution

[Kim+22a] utilizes four PUs and states that after the ModUp step, the number of polynomials that need to be transferred during limb-based only decomposition is $2 \cdot dnum \cdot (L + K + 1)$, while for coefficient-wise it is $(dnum + 2) \cdot (L + K + 1)$. They present this discussion in the context of a generalized KeySwitch technique for an arbitrary value of $dnum$, presented in Algorithm 29 (in Section A.1).

This assumes the limb-wise distribution is done after multiplication with KeySwitch keys, and all the results are sent to every PU via an all-to-all broadcast. However, we remark that one PU does not need to send all the polynomials and instead only needs to send $\frac{2 \cdot (dnum-1) \cdot (L+K+1)}{dnum}$ polynomials so that every PU holds the results for the supported bases. Therefore, the total cost becomes $2 \cdot (dnum - 1) \cdot (L + K + 1)$, which is less than the cost of coefficient-wise distribution for $dnum = 3$. Furthermore, we would like to note that the polynomial distribution after KeySwitch is expensive as the data doubles in size after multiplication with the two key components. Hence, this distribution should be done immediately after NTT computation, which further reduces the cost to only $(dnum - 1) \cdot (L + K + 1)$. This is much less compared to the coefficient-wise distribution $(dnum + 2) \cdot (L + K + 1)$. Hence, we reassert that limb-based distribution will attain less communication overhead than coefficient-wise without any extra computation overhead.

5.5.3 Data Distribution across Multiple Chiplets

The above analysis establishes that limb-based decomposition is the best task and data distribution technique across multiple chiplets. Within a single chiplet, the computation is coefficient-wise distributed, as N_2 coefficients are processed in parallel (discussed in Section 5.4). However, as discussed in Section 5.3 (b) and (c), the limb-based distribution technique also has limitations. Therefore, we adapt it to offer long-term high performance.

This adaptation stems from two key observations regarding the data flow of the KeySwitch operation, as shown in Figure 5.1. Firstly, the INTT results that need to be shared and duplicated are ephemeral, and thus, they do not

require any long-term storage – subsequent operations immediately consume them. To improve the efficiency, instead of duplicating the INTT data or sharing memory across chiplets, we leverage a *ring-based C2C communication* as shown in Figure 5.10. The chiplets are connected in a ring formation where each chiplet processes one INTT result and then sends it to the next chiplet. The ring-based data movement minimizes the number of C2C interconnects and ensures that each chiplet operates on independent memory, simplifying placement and routing constraints.

The second observation is related to the underutilization of chiplets with reduced levels or depths in the ciphertext after homomorphic rescaling. If we distribute the limbs across the chiplets in an interleaved manner, then as the multiplication depth decreases, the number of limbs per ciphertext in each chiplet also uniformly decreases. To explain the benefits of the interleaved distribution, we will use the analogy of a card game.

The FHE card game: In this game, each player represents a chiplet, while the residue polynomials or limbs act as the cards. The cards dealt out are collected in a LIFO (last-in-first-out) manner [2], imitating the loss of multiplicative depth during computation. All players engage in the game (FHE routine computation) until they exhaust their cards. The start of a new FHE computation game mirrors Bootstrapping, which starts when only one player retains a single card. Until this point, players without cards must wait until the next game to participate. With these rules, the dealer (user or compiler [Via+23]) has two choices: deal out all the cards to one player before moving to the next or do alternate distributions such that every other card goes to different players. Let us say there are $L = 32$ cards and $\mathtt{r} = 4$ players. In the former case, the first player gets the cards drawn at instances $\{0, 1, 2, \cdots, 7\}$, and in the latter case, at instances $\{0, 4, 8, \cdots, 28\}$, and so on for the other players.

In both scenarios, each player receives a total of 8 cards. In the first option, the last player exhausts their cards first, followed by the preceding player, and so on and so forth. Consequently, until the first player runs out of cards, others remain inactive. Conversely, with alternating distribution, each player loses one card in turn. Thus, at any point in the game, players either possess the same number of cards or one less, ensuring active involvement throughout.

The goal of FHE architecture design is to ensure the full utilization of chiplets in the long term and, thus, deliver high performance for the available computation resources. It translates to maximizing the player interaction in the FHE card game. Thus, the latter technique of interleaved alternate distribution offers maximum chiplet participation in the "card game" of computations. Now, if there are too many players, the number of idle players will increase, regardless of the technique used. The latter technique will only minimize the idle time. This is the problem with using more chiplets. Thus, next, we will discuss our final key technique for reducing communication overhead and then derive a good upper bound on the number of chiplets.

[2] It can also be FIFO (first-in-first-out) without any loss of generality.

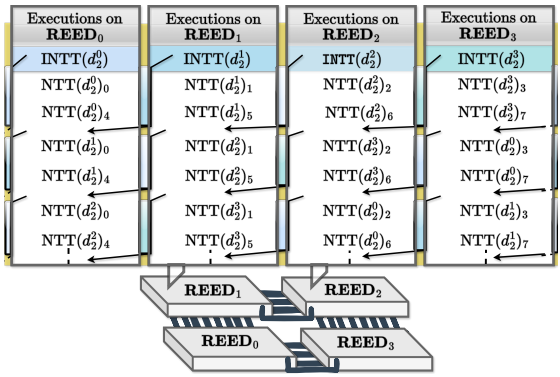


Figure 5.10: Non-blocking ring-based communication for four REED chiplets when $l = 6$. The blocks between executions represent the long communication window to make up for slow inter-chiplet (C2C) communication.

5.5.4 Efficient Non-Blocking C2C Communication

The proposed ring-based communication still incurs overhead, as the chiplets must wait for every chiplet before them in the ring to send the INTT result. Hence, we propose a communication strategy, illustrated in Figure 5.10, to overcome this remaining problem.

The key idea is that the chiplets concurrently operate on different limbs instead of waiting for one limb and then processing it, as shown in Algorithm 19. Each chiplet begins with the assigned limb, computes INTT, and then performs multiple NTT operations on it. While performing NTT, it starts sending/receiving the INTT result. For example, the REED_0 sends its INTT result to REED_3 and receives the INTT result from REED_1. This is a uni-directional *ring-based communication*. Since only one INTT result needs to be sent for $\frac{l+2}{\mathtt{r}}$ parallel NTT computation, we have a larger C2C communication window compared to computation. Consequently, *non-blocking communication* is achieved as data computation can proceed concurrently with relatively slower communication.

This technique necessitates just one read/write port per chiplet, in contrast to the requirement for $(\mathtt{r}-1)$ ports in a star-like (i.e., all-to-all) C2C communication network.

The Adapted Data and Task Distribution Technique.

C2C communication bandwidth plays a major role in the performance versus the number of chiplets trade-off. Let us assume the C2C communication bandwidth is k× slower than the HBM to Chiplet communication bandwidth. Each chiplet operates on $\frac{L+2}{\mathtt{r}}$ polynomials. The total computation time to process all $L+2$ polynomials for the KeySwitch should be close to k. Otherwise, we will not be able to decouple the communication from computation as discussed above. This offers us a loose upper bound $\mathtt{r} < \frac{L+2}{k}$. To ensure u× higher utilization,

Algorithm 19 ModUp_KeyMul

In: $\boldsymbol{d_2}$ (the ciphertext component to be linearized)
Out: BUF = $ModUp_KeyMul(\boldsymbol{d_2})$

1: Following tasks are executed by $\text{REED}_i \ \forall \ i \in [0, r)$
▷ All REED_i operate **in parallel** as shown in Figure 5.10
2: **for** $(j = 0; j < \frac{l+1}{r}; j = j + 1)$ **do**
3: $\mathtt{I}_\mathtt{i}^{\mathtt{rcv}} \leftarrow \mathtt{INTT}(\boldsymbol{d}_2^{j \cdot r + i})$
▷ *Initiate communication with* $\text{REED}_{(i+1) \bmod 4}$, $\text{REED}_{(i-1) \bmod 4}$
4: **for** $(m = 0; m < r; m = m + 1)$ **do**
5: $\mathtt{I}_\mathtt{i}^{\mathtt{proc}} \leftarrow \mathtt{I}_\mathtt{i}^{\mathtt{rcv}}$
▷▷ *Long Communication window opens now* ◁◁
▷ *Receive* $\mathtt{I}^{\mathtt{rcv}}_{(\mathtt{i+1}) \bmod 4}$ *from* $\text{REED}_{(i+1) \bmod 4}$ *and Send* $\mathtt{I}_\mathtt{i}^{\mathtt{rcv}}$ *to* $\text{REED}_{(i-1) \bmod 4}$
6: **for** $(t = 0; t < \frac{l+2}{r}; t = t + 1)$ **do**
7: $\text{BUF}_{t*r+i} \mathrel{+}= (\mathtt{NTT}(\mathtt{I}_\mathtt{i}^{\mathtt{proc}}) \cdot \mathtt{KSK}^{j \cdot r + (i+m) \bmod 4})_{q_{t*r+i}}$
8: **end for**
▷▷ *Ensure* $\mathtt{I}^{\mathtt{j \cdot r + (i+m+1) \bmod 4}}$ *has been received*
▷▷ *Communication window closes* ◁◁
9: $\mathtt{I}_\mathtt{i}^{\mathtt{rcv}} \leftarrow \mathtt{I}^{\mathtt{rcv}}_{\mathtt{(i+1) \bmod 4}}$
10: **end for**
11: **end for**
12: **return** BUF

this bound must be made tighter. u can take any value $\leq \frac{L+2}{k}$ ($u = \frac{L+2}{k}$ for monolithic chip). We take u as 4. The adapted limb-based task/data distribution technique has the following properties:

- The number of chiplets is constrained by $\mathtt{r} \leq \frac{L+2}{4k}$.
- An interleaved data/task distribution approach is utilized such that Chiplet_i gets data and task corresponding to limbs $\mathtt{r}j + i \ \forall \ 0 \leq j < \frac{L+2}{\mathtt{r}}$, instead of sequential allotment ($\text{Chiplet}_i \leftarrow \mathtt{r}i + j$).
- The technique outlined in Algorithm 19 is to be followed by all Chiplets to minimize data exchange overhead and costly C2C interconnects.

5.5.5 ModDown/Rescaling Task flow

So far, we have discussed the data and task flow in the context of ModUp operation, which exhibits the highest complexity in terms of NTT operations. Next, we turn our attention to the ModDown/Rescaling operation. As shown in Figure 5.1, the ModDown operation constitutes the final step of the KeySwitch procedure. This operation involves a single INTT followed by l+1 NTT operations, resulting in significantly lower complexity compared to the ModUp operation. Figure 5.11 (a) illustrates the communication flow for ModDown across multiple chiplets. Specifically, the chiplet that holds the polynomial corresponding to the modulus being dropped performs the INTT and sends the resulting data to the next chiplet. The data is then propagated through the remaining chiplets

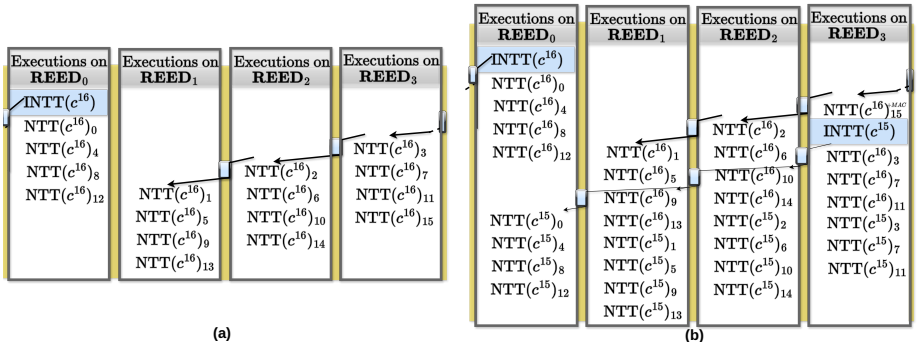


Figure 5.11: Ring-based communication for four REED chiplets when $l = 15 \rightarrow 16$ (after ModUp) for the (a) ModDown operation and (b) ModDown+Rescale operations.

in a feed-forward ring topology, as depicted in the figure. Notably, the MAS operations are done simultaneously with the NTT operations.

While the overall computational complexity of the operation is $\frac{l+1}{r}$ NTTs per chiplet, there is an additional fixed communication overhead of $r-1$ polynomials due to inter-chiplet data transfer. A similar flow is observed in the Rescaling operation, which reduces ciphertext depth following noise growth. However, Rescaling is not strictly needed after the KeySwitch procedure and can be performed independently to mitigate noise after several rotations, accumulations, or plaintext multiplications. ModDown and Rescaling are applied to each ciphertext component, requiring two polynomial broadcasts. Note that the figure shows the processing of one component, as the second component's communication will be done in parallel with the first component's computation. The INTT will be computed by the same chiplet and transferred, while the other chiplets compute the NTT for the previous component. Hence, no communication overhead will be incurred for the second component. Runtime of ModDown and Rescaling is included in the total runtime reported for the Relinearization operation (tabulated in Table 5.2).

When Rescaling is required immediately after a ModDown operation, it leverages the optimized flow from the ModUp operation, as demonstrated in Figure 5.11 (b). Both processes necessitate an INTT broadcast followed by NTT computation, where the chiplet executing the INTT performs one NTT computation at most, effectively reducing the computation with respect to the modulus used for the INTT. Consequently, while the first set of NTT computations is underway, the second INTT result can be broadcast; thus, wait time overhead is incurred only once overall.

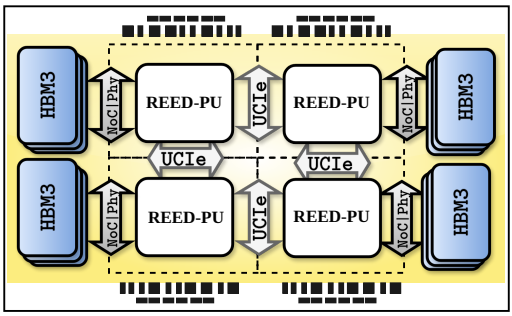


Figure 5.12: The complete architecture diagram of 4-chiplet REED 2.5D for 1024×64 configuration. The multiple small black blocks denote I/O interconnects along the edges.

Table 5.2: Performance micro-benchmarks for 28nm and 7nm.

Micro-Benchmarks ↓	**Level**	**Time (ms)**	
Configuration →	**l**	**1024×64**	**512×128**
AUT/MAS (pt-ct)	30	0.005	0.003
MAS (ct-ct)	30	0.01	0.005
KeySwitch	30→31	0.19	0.08
MULT & Relin.	30→29	0.22	0.11
Bootstrapping	1→30→15	14.2	7.1

5.6 Implementation Results

Based on the precision-loss study done for $dnum = L+1$ (shown in Section 5.7.1), we choose the overall parameters for synthesizing and benchmarking our design as $N = 2^{16}, L = 30, K = 1, L_{boot} = 15, w = 54$. Upon implementation (silicon realization), only the parameters N, w are fixed, and the other parameters (e.g., $dnum$) can be changed as per application requirements. For a higher value of K, the task distribution is discussed in Section A.1. The user can control this via the low-level instruction abstraction provided.

We synthesize the entire REED 2.5D PU for configurations 1024×64 and 512×128 using TSMC 28nm and ASAP7 [Cla+16] 7nm ASIC libraries with Cadence Genus 2019.11, and SRAMs are used for on-chip memories (with total storage capacity of 14 polynomials). The REED-PU and all its building blocks are fully implemented using Verilog Hardware Definition Language. We simulated our design using Vivado 2022.2 for functionality testing. The Instruction Controller is a part of the design and is connected to the master processor via the same master-slave interface as data in prior works [Mer+23]. Our primary objective is to achieve high performance while optimizing area and power consumption. To this end, we set our clock frequency target to 1.5 GHz, use High-vt cells (hvt) configuration for low leakage power, enable clock-gating, and set the optimization efforts to high. We set the input/output delays to 20% of the target clock period

Table 5.3: Total area consumption of 4-chiplet REED 2.5D for different configurations on 28nm and 7nm.

Components	28nm (mm^2)		7nm (mm^2)	
	1024×64	**512×128**	**1024×64**	**512×128**
REED	74.9	115	24	43.9
REED-PU	58.0	81.0	7.01	9.9
⌊ NTT/INTT	38.2	56.8	5.61	7.9
⌊ 2×MAS	3.1	6.6	0.42	0.76
⌊ PRNG	0.15	0.28	0.02	0.04
⌊ 2×AUT	0.14	0.32	0.02	0.04
⌊ Memory	16.1	16.1	1.2	1.2
HBM PHY/NoC	16.9	33.8	16.9	33.8
4×REED	299.6	392.4	96	175.6
C2C	12.32	14.64	0.8	1.6
Total Area	**311.9**	**461.4**	**96.7**	**177**

and leverage incremental synthesis optimization features. Moreover, we take a step further by *prototyping* the building blocks on Xilinx Alveo U250 to verify functional correctness, which has not been done by prior ASIC FHE accelerators.

As off-chip storage, we leverage the state-of-the-art HBM3 [JED22; Lee+15; Par+23; Ram] memory, offering improved performance and reduced power. It is already deployed in commercial GPUs and CPUs [EH22]. HBM3 with 8/12 stacks of 32Gb DRAMs has 32/48 GB storage capacity [JED22; Mur23]. The ciphertexts provided by the client can be transferred to REED using 32 lanes PCIe5, offering a bandwidth of 128 GB/s [Sun+23]. In our work, we present results for HBM3 PHY and HBM3 NoC (Network On Chip), based on [Cha+23; Par+23; Ram] with reported bandwidth of 1.2TB/s [Ram]. For C2C communication, UCIe (Universal Chiplet Interconnect Express) can offer an advanced interconnect with a bandwidth of 0.63 TB/s [Syn23; UCI23] for 2.5D integration. Thus, we can send or receive 64, $54-bit$ coefficients per clock cycle. Overall, we present the results using the following: (i) Verilog description of REED-PU, (ii) Instruction set modeling of the rest, (iii) Synthesis result in two technologies, (iv) Power simulation of a prelayout netlist for single clock, (v) Extrapolation of the power consumption of single clock cycles for the cycle accurate simulation, (vi) Power values estimated from the cycle accurate simulation, and (vii) Area estimations after synthesis.

Table 5.3 presents the area results for the REED 2.5D architecture, featuring a 4-chiplet configuration as illustrated in Figure 5.12. This design conforms to the fabricated chiplets systems [AMD23; Int23]. The inner REED-PU, NoC, and HBM (shown in Figure 5.12) constitute one chiplet (similar to [Pau+23; ZSB21]). In Table 5.2, we present the performance of FHE routines for both configurations (512×128 and 1024×64) with the achieved target clock frequency of 1.5 GHz. Section 5.4.6 explains how the throughput of KeySwitch is obtained. For $dnum = L + 1$, we report high (≈99.9%) utilization which reduces to ≈95% for $dnum = 3$, as detailed in Section A.1.1.

5.6.1 Power and Performance Modelling

Using a cycle-accurate model, we obtain the performance and power consumption estimates for REED. REED's communication and computation are decoupled by design and do not need application-specific schedules to reduce data load/store stalls. REED also has a modular architecture design; all chiplets are identical, and none of the elementary building blocks is distributed across chiplets. The REED-PU was fully described using Verilog Hardware Definition Language. Its complete functionality is tested via simulations in Vivado 2022.2. For the results, one REED-PU was fully synthesized to obtain overall area results using Cadence Genus a, along with data communication and distribution strategies (Section 5.5.3, or software library and hardware implementation. However, in hardware, certain instructions can run simultaneously. Since they have constant execution time, we factored this behaviour into the software model, ensuring the clock cycle count aligned with expectations for hardware execution.

In Section 5.4.4, we elaborated on how our instruction-set architecture handles the micro (NTT, AUT, MAS) and macro (e.g. rotation, KeySwitch) instructions. Thus, a user does not need to handle micro-instructions due to the provided macro-instruction level abstraction. Predefined microcode for static macro-instructions ensures optimized data flow and memory management. Data exchange across chiplets occurs solely during KeySwitch and is incorporated into the microcode and task distribution. The simulator takes into account the bandwidth of C2C and HBM-chiplet communication along with data communication and distribution strategies (Section 5.5.3, Section 5.5.4). The run-time of macro-instructions is obtained using the known static schedule of micro-instructions. Finally, the macro-instructions are scheduled using OpenFHE [Al +22], and runtime is obtained for higher-level operations (bootstrapping, DNN).

We used the Cadence Genus tool to measure total power consumption for the entire REED-PU. For each operation, the duration for which any unit in the chiplet remains active is predetermined and static. The power consumption for each elementary building block is also obtained using the Cadence toolchain. We estimated the average power consumption based on this information, combined with the runtime of each operation, following a methodology commonly employed by prior works, such as [Kim+22a; Kim+22b; Kim+23a; Sam+22], for estimating FHE hardware acceleration on ASIC technology. Similarly, in a multi-chiplet scenario, the operation runtime remains static, and we conservatively estimate communication overhead to be 2× slower than linear operations on the same volume of data. This approach provides a safe estimate of the total runtime. The cycle-accurate model serves as a simulator and effectively imitates the behaviour of the Instruction Set Controller within the chiplets, giving us a reasonable approximation of system performance. We validate this cycle-accurate modelling with Vivado simulation.

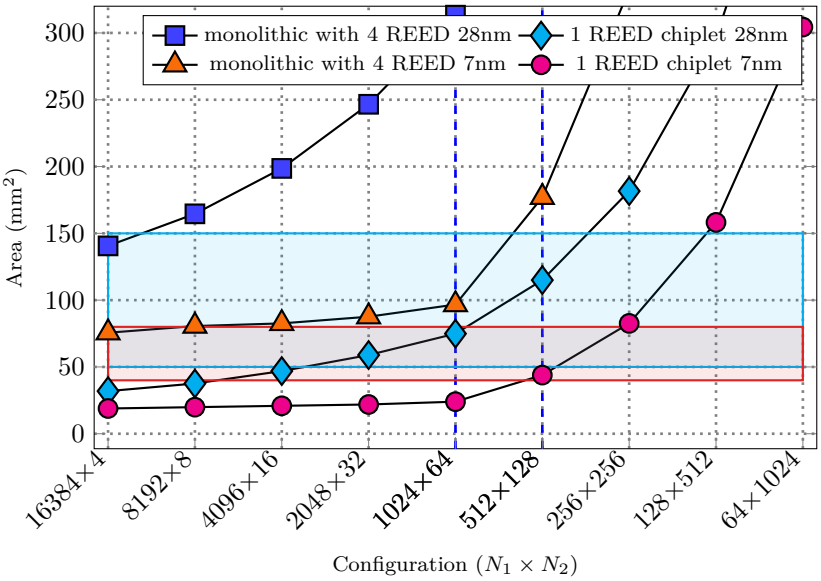


Figure 5.13: Area increase with rising throughput configurations [GPG23].

5.6.2 What to expect from higher-throughput configurations?

Until now, we have examined two configurations (1024×64 and 512×128) that only partially demonstrate the advantages of our proposed scalable design methodology. As we double the throughput (by doubling the value of N_2), the area of PU only increases by approximately 1.5×. This trade-off arises because the chip area comprises two components: (*i*) the computation logic area, which scales linearly with throughput, and (*ii*) on-chip storage that remains fixed to a number of polynomials. When we opt for a higher configuration, the polynomial size remains the same while the number of coefficients to be processed in parallel increases. We also discuss the trade-off between storing and generating twiddle factors on the fly. As shown in Table 5.4, twiddle factor generation (TFG) increases the number of multipliers (Mul.) by only 18% for different configurations (N_1 and N_2). Indeed, reduction in memory (Mem.) decreases (i.e., number of coefficients to store) with larger N_2. Yet, even for large values of N_2 (i.e., $N_1 = 64$ and $N_2 = 1024$), we observe a reduction in the memory by 57% compared to storing the twiddle factors (~~TFG~~).

However, an important question remains: *what configuration strikes the best balance between throughput and manufacturing cost?* To address this, we turn to [GPG23]. The authors report that the best manufacturing size for high yield ranges from 40 to 80 mm^2 for 7nm technology, while for 40nm, it ranges from 50 to 150 mm^2. In Figure 5.13, we present two sets of area consumption results for 28nm and 7nm technologies. The first set corresponds to four REED cores

$N_1 \times N_2 \rightarrow$	2048×32	$\mathbf{1024 \times 64}$	512×128	256×256	128×512	64×1024
Total Mul.	432	**832**	1,600	3,072	5,888	11,264
TFG Mul.	68	**131**	258	513	1,024	2,047
TFG Mem.	222,912	**310,624**	486,400	707,232	1,149,248	1,771,488
~~**TFG** Mul.~~	0	**0**	0	0	0	0
~~**TFG** Mem.~~	4,260,320	**4,228,064**	4,212,704	4,206,560	4,206,560,	4,212,704
Mul. Inc.	16%	**16%**	16%	17%	17%	18%
Mem. Red.	95%	**93%**	88%	83%	73%	58%

Table 5.4: The table compares memory (number of coefficients to store) and multiplier units for different N_1 and N_2 values with and without twiddle factor generation (TFG).

produced as a single monolithic chip, while the second set represents one REED chiplet. The best area ranges are highlighted in black and pink. As we can see, for both 7nm and 28nm, the configuration 512×128 falls within the best development area range and offers high throughput. The configuration 1024×64 is within the optimum range for 28nm and is close to it for 7nm. Monolithic designs, within the best range, offer 4× to 8× less throughput.

5.6.3 Comparison with Related Works

The realization of privacy-preserving computation through FHE holds great potential for the entire community, resulting in various acceleration works. Among these, the ASIC designs [Fel+21; Kim+22a; Kim+22b; Kim+23a; Sam+22] have achieved the most promising acceleration results. However, a direct comparison with these works would be unfair as the benchmarks are provided for different parameters $(dnum, L, w, L_{boot})$. Hence, to ensure fairness, we also provide bootstrapping results for $dnum = 3$ (utilized by [Kim+23a]) in Table 5.5. Next, since we cannot change the word size ($w = 54$) chosen for high precision, we select $L = 23, K = 8,$ and $L_{boot} = 17$ accordingly.

We use the amortized bootstrapping time $\mathrm{T_{A.S.}}$ [Kim+22a; Kim+22b] metric that calculates the bootstrapping time divided by L_{eff} and packing n. This metric overlooks factors such as area, power, and precision. Higher precision necessitates a larger word size, w (or expensive composite scaling). Thus, we use the EDAP (Energy-Delay-Area product) metric [Li+09] and modify it (EDAP_w) to incorporate a linear increase due to word size (discussed in Section 5.7.1).

Table 5.5 compares our design's area consumption, performance, and power consumption for the packed bootstrapping operation with existing monolithic works, F1 [Fel+21], BTS [Kim+22b], ARK [Kim+22a], CraterLake (CLake) [Sam+22], and SHARP (SH) [Kim+23a]. REED achieves 1.9× better performance than the state-of-the-art ($\mathrm{SH_{36}}$) while consuming 1.8× less area. Our area consumption is less as the prior works utilize at least half of the chip area for on-chip memory. In our case, on-chip memory is not significant, as we utilize HBMs for major storage. We achieve better performance results due to the high throughput and 4.8× higher off-chip communication bandwidth offered by four HBM3 blocks,

Table 5.5: Comparison of REED 2.5D with state-of-the-art

Work	**Area** (mm^2)	$\mathbf{T_{A.S.}}$ (ns)	$\mathbf{P_{Avg}}$ (W)	$\mathbf{EDAP}_w$ (/M)	**Parameters** ($N/L/dnum$)	**# B.S.**/Dollar (ops)* [GPG23]
$F1_{32}$	71.02†	470	28.5†	754.5	$2^{14}/23/24$	0.48
BTS_{64}	373.6	45.4	163.2	106.0	$2^{17}/39/2$	0.74
ARK_{64}	418.3	14.3	135	9.74	$2^{16}/23/4$	1.95
$CLake_{28}$	222.7†	17.6	124†	16.5	$2^{16}/60/1{-}3$	3.64
SH_{36}	178.8	12.8	94.7	4.1	$2^{16}/35/3$	6.23
SH_{64}	325.4	11.7	187	7.0	$2^{16}/22/3$	3.4
$\mathbf{REED}^{\ddagger(1)}_{54}$	**98.4****	**6.6**	**48.8**	**0.21**	$2^{16}/23/3$	**25.3**
	96.7	**28.8**	**49.4**	**3.96**	$2^{16}/30/31$	**6.32**
$\mathbf{REED}^{\ddagger(2)}_{54}$	**177**	**14.4**	**83.5**	**3.10**	$2^{16}/30/31$	**6.82**

† Area/power are normalized [Kim+23a; Nar+17] (14nm/12nm to 7nm).
‡ Result for configuration (1) 1024×64 with one HBM per chiplet, and (2) 512×128 with two HBM per chiplet.
* The cost per mm^2 is obtained from [MUSb] ($\$57,500/mm^2$ for 7nm) and yield estimates are taken from [GPG23; Ma+22] including manufacturing, testing, and integration costs.
** For $dnum < L + 1$, the area increases due to BaseConversion MAS units (Section A.1.1).

where each chiplet is exclusively connected to one HBM3.

Prior works utilize up to 1TB/s peak off-chip bandwidth, relying on large on-chip memories for storing keys, which necessitates much higher on-chip communication bandwidths 20TB/s [Kim+22a], 36TB/s [Kim+23a], and 84TB/s [Sam+22]. In contrast, although our proposed REED system uses 4.8× higher off-chip bandwidth (with each HBM offering 1.2TB/s at its peak), it efficiently offloads memory requirements to HBM, resulting in significantly reduced on-chip memory usage and consequently $4.2 - 7.5\times$ lower on-chip memory bandwidth requirements compared to [Kim+22a; Kim+23a; Sam+22]. The off-chip bandwidth requirement of one REED chiplet is comparable to that of a monolithic design. However, it incurs a 2.7× performance loss compared to the state-of-the-art [Kim+23a] while consuming 3.36× less area. This trade-off is a consequence of designing chiplet-based architecture over a monolithic one. The REED 4-chiplet system offers better performance and 35× higher energy efficiency for lower chiplet area, as shown in Table 5.5.

We also assess the yield and manufacturing cost in Figure 5.14 by utilizing the results reported in [GPG23; Ma+22; MUSb], including manufacturing, pre- and post-testing, and integration costs. For a fair comparison, we use the original area, not the word-size scaled area, for prior works. As illustrated, we achieve the highest yield and lowest manufacturing cost on 7nm, resulting in the lowest overall cost (manufacturing cost/yield), which is 50% less than that of the state-of-the-art monolithic design $SHARP_{64}$. The cost is estimated using the yield metric provided in [FM22; Ma+22]. Based on this analysis, we observe that the yield for $SHARP_{64}$ is approximately 73%, whereas the yield for the 4-chiplet REED (512×128) configuration reaches around 96%. This represents a 1.31× improvement in relative yield. Factoring in the manufacturing costs[MUSb] and

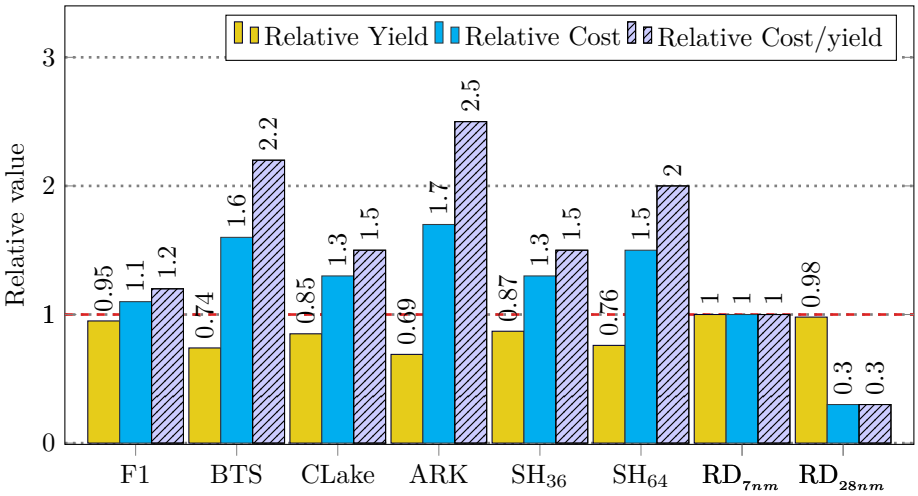


Figure 5.14: Relative a) yield of existing monolithic designs versus the proposed 7nm chiplet-based architecture [Ma+22], b) development cost (including Interposer cost) [GPG23; Mer+23; MUSb], and c) cost of SiP development (cost/yield). RD refers to our work REED 2.5D.

the chiplet packaging costs [GPG23], the design achieves an additional 1.5× cost reduction. Together, these factors contribute to a 50% reduction in the cost-to-yield ratio, offering an estimate of the overall chip manufacturing cost. On 28nm technology, we achieve 85% cheaper design compared to SHARP_{64}. Further, note that the cost per bootstrapping for $dnum = L + 1$ looks similar to prior monolithic works, but at this parameter, we also have the highest ($\approx 2\times$ more) computation depth remaining after bootstrapping for $\omega \geq 54$. Thus, applications utilizing this parameter choice require 2× less frequent bootstrapping operations.

5.6.4 Higher Chiplet-Integration Study

In Section 5.3, we examined multiple chiplet configurations and selected four ($\mathtt{r} = 4$) based on long-term utilization, lower power dissipation, and low integration costs. Our design methodology and data/task distribution approach remain adaptable to any desired chiplet configuration and the number of chiplets. In fully connected chiplet nodes, the interconnect length between chiplets can significantly impact energy consumption and latency. Therefore, our proposed non-blocking ring-based communication technique for KeySwitching shows a better advantage for higher chiplet integration density.

Figure 5.15 illustrates a chiplet-based architecture for six chiplets, which can be expanded to accommodate more chiplets. Additionally, energy-efficient, lower-bandwidth memories, such as DDR, can be integrated with the appropriate (N_1, N_2) configuration based on memory throughput requirements. While increas-

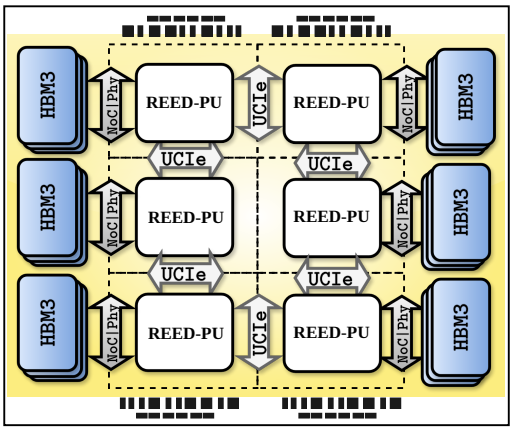


Figure 5.15: The complete architecture diagram of 6-chiplet REED 2.5D for 1024×64 configuration.

Number of Chiplets →	**4**	**8**	**12**
Area (mm^2)	311.9	623.8	935.7 (> reticle limit)
$\mathbf{T_{A.S.}}$ (ns)	14.41	7.37	4.98

Table 5.6: Results for 1024 × 64 configuration on 28nm technology with parameters ($N/L/dnum = 2^{16}/30/31$).

ing the number of chiplets enhances performance, as shown in Table 5.6, it comes at the cost of area and underutilization when the current multiplicative depth (l) falls below the number of chiplets ($l < r$). However, more chiplets can lead to better performance in the long term and allow tiling beyond the reticle limit, albeit with the additional area, power, and integration overhead. Additionally, note that a higher number of chiplet interconnects implies more points of failure, making testability and reliability more complex. A more detailed multi-chiplet study is also presented in concurrent work [Kim+24b], where the authors show that increasing the cores from 4 to 16 results in speedup of only $1.99 - 2.11\times$.

5.7 Application benchmarks

We benchmark three machine learning applications: linear regression, logistic regression, and a Deep Neural Network (DNN). Each application is evaluated for *encrypted* training and inference. In this setting, the server provides computational support without knowledge of the data or model parameters, ensuring complete blind computation. Most applications benchmarked in the previous works [Fel+21] are partially blind; the server does not see the data but knows the model parameters to evaluate it. To the best of our knowledge, none of the previous works benchmark an encrypted neural network training. The speedup results are presented in Table 5.7 using the most area conservative design (1024×64). A

Table 5.7: Application benchmark and the speedup achieved by REED 2.5D. The CPU speed is reported on a 24-core, 2×Intel Xeon CPU X5690 @ 3.47GHz with 192GB DDR3 RAM.

Appl.	Accuracy	Op	Time		Speedup
			CPU	HW	
Lin.Reg.	78.12%	Inf.	0.86 s	0.31 ms	2,873×
		Trn.	13.82 s	4.6 ms	2,991×
Log.Reg.	61.8%	Inf.	1.27 s	0.46 ms	2,785×
		Trn.	11.18 s	3.8 ms	2,865×
DNN	95.2%	Inf.	128.7 s	48.6 ms	2,646×
		Trn.	29 days	920 s	2,725×

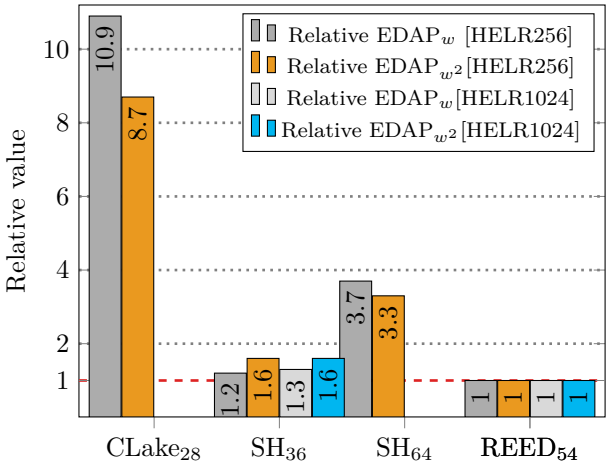


Figure 5.16: Relative metrics comparison for the HELR [Han+19] application with batch sizes 256 and 1024. Under these metrics, the lower the value, the better.

higher configuration (512×128) will improve performance by 2×.

- **Linear Regression**: We employ the Kaggle Insurance dataset [Sco20] to benchmark linear regression. The model uses a batch size of 1204 and 1338 input feature vectors (each containing six features) for training and inference, achieving an accuracy of 78.1% (the same as the plain model [Sco20]).
- **Logistic Regression**: It is a supervised machine learning model that utilizes the log function, evaluated using function approximations in a homomorphic context. Its accuracy depends on the degree of approximation function expansion and precision. Existing works, such as [Kim+23a; Sam+22], utilize the HELR [Han+19] to benchmark encrypted training on MNIST [LC10] data, with batch sizes (256, 1024). In Figure 5.16, we

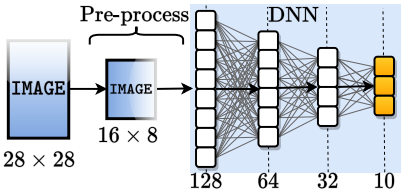


Figure 5.17: A DNN for MNIST [LC10] with two hidden and one output layers.

illustrate the performance advantage of REED 2.5D. To predict cancer probability, we further evaluate logistic regression on the iDASH2017 cancer dataset (similar to [Kim+18]). Here, we achieve a training accuracy of 62% in a single iteration. This dataset comprises 18 features per input, with batch sizes of 1,422 and 1,579 used for training and inference, respectively.

- **Deep Neural Network** : The DNN serves as a powerful tool for Deep Learning. Our study employs a DNN, shown in Figure 5.17, for the MNIST dataset [LC10], with two hidden and one output layer. We pack four pre-processed images per batch to prevent overflow during matrix multiplication. DNN training requires 12,500 batches. Thus, all the existing works [Kim+22a; Kim+22b; Kim+23a; Sam+22] that do not provide computation-communication parallelism will suffer as their on-chip memory is insufficient. The DNN is trained for ≈7000 (≈5.8 Bootstrappings per iteration) iterations and achieves 95.2% accuracy in 29 days using OpenFHE [Al+22]. REED 2.5D could finish this in only 15.4 minutes. This is where our computation-communication parallelism shines, as many ciphertexts are required for such an application. None of the works in literature offers this and is bound to suffer from memory-intensive applications.

5.7.1 Precision-loss Experimental Study

Another facet of privacy-preserving computation is the issue of precision loss. Since the server cannot see the intermediate or final results, the best it can do is to ensure that the parameters it operates on support higher precision. To validate our parameter sets, we ran experiments for the DNN training. In Figure 5.18, we can see how quickly the training accuracy drops as the word size is reduced. Thus, precision plays a vital role in providing privacy-preserving computation on the cloud. Our choice of 54-bit word size strikes the perfect balance between precision and performance. Works offering a smaller word-size [Fel+21; Kim+23a; Sam+22] require in-depth study to mitigate the accuracy loss due to low precision.

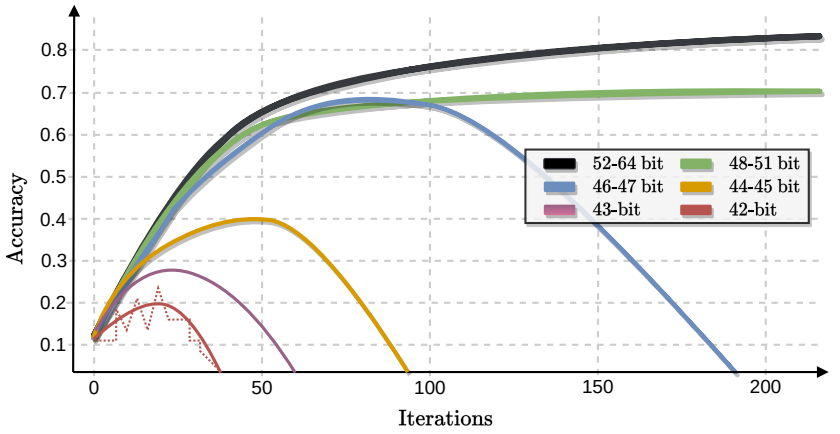


Figure 5.18: Accuracy plot of different word sizes for the DNN. The lines are smoothened and the red dotted zig-zag line resembles the original form.

5.8 Future Scope: Journey from 2.5D to 3D

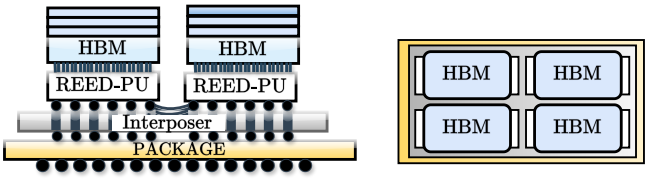


Figure 5.19: The side and top view of futuristic RE3D has four REED 3DIC chiplets.

The extension of REED 2.5D to a complete 3D IC holds immense potential for future computing. To achieve this transition, we have two options: connecting the PU with the HBM controller via TSV (as shown in Figure 5.19) or merging the PU unit with the lower HBM controller die. Since HBM is sold as an IP, the latter approach relies on the IP vendors to integrate the PU. By adopting either of these approaches, we can significantly reduce the reliance on the Network-on-Chip (NoC), leading to a compact chip design with lower power consumption. Each chiplet will be a full 3D IC package (PU and Memory) and will need a C2C link via an interposer for connecting to other chiplets. A reduction in the area is expected due to fewer HBM stacks on the lateral area and the integration of the REED-PU unit with the HBM controller. Additionally, decreased critical paths would further enhance the design's performance. Thus, the REED's 3D IC integration promises a huge reduction in overall chip area and power consumption.

5.9 Conclusion

FHE has garnered considerable interest due to its ability to preserve data and computation privacy, leading to several efforts to accelerate FHE using large, monolithic ASIC designs. However, many of these attempts primarily focus on acceleration at the expense of yield, manufacturing cost, and scalability. We proposed a scalable FHE accelerator design methodology for a multi-chiplet system that can be easily extended to larger configurations while adapting to constrained environments. Chiplet-based designs face inherent challenges, such as increased latency costs due to slow C2C communication. We designed REED to address these issues and demonstrate advantages over monolithic designs in terms of performance, area, and energy consumption. REED achieved this feat by utilizing a non-trivial yet uncomplicated bandwidth-oriented design methodology and a modular design approach.

An efficient workload division strategy optimized for multi-chiplet architectures is proposed to minimize memory usage and enhance efficiency through interleaved data and workload distribution strategy for all FHE routines. Next, to address slow chiplet-to-chiplet communication, a novel non-blocking ring-based inter-chiplet communication strategy tailored to FHE is introduced. Additionally, a scalable bandwidth-oriented design methodology is adopted, offering flexibility to adjust to the varying area and performance needs while supporting communication-computation parallelism within each chiplet. Furthermore, novel design techniques are presented for building blocks (NTT/AS/AUT), enhancing scalability. These techniques accelerate routines such as KeySwitch or Bootstrapping and reduce their latency by ≈67%. Finally, REED benchmarks an encrypted DNN training, demonstrating utility for real-world FHE applications.

All the REED-chiplets are small and identical, allowing us to prototype the building blocks using an FPGA and validate the functionality, which had not been done by prior works. Summarily, the proposed design methodology for REED showcased a robust and scalable approach to FHE acceleration that addressed several key challenges inherent in traditional monolithic designs. This paves the way for interesting future prospects such as formal verification. The advancements presented in this Chapter hold the promise of advancing privacy-preserving computations and promoting the wider adoption of FHE.

6

Secure Cryptoprocessor for Hybrid HE

Breu

In the previous chapter, we addressed FHEs server-side bottlenecks with a multi-chiplet accelerator, but two major challenges remain: costly client-side encryption and high communication overhead. Hybrid Homomorphic Encryption (HHE) mitigates these issues by combining fast symmetric encryption on the client with homomorphic decryption on the server, with recent schemes extending to prime-field settings for improved performance.

This chapter fills the lack of systematic hardware evaluation for HHE by presenting the first hardware implementation of Pasta, an integer-based HHE scheme. We benchmark Pasta on FPGA and ASIC platforms and study its resilience to power analysis.

Contribution: This chapter is based on the following two papers. The thesis author led the conceptual development, completed the implementations, and drafted significant portions of the text. Testing and experimentation were performed by masters thesis students supervised by the thesis author, and the power analysis was conducted by Maciej.

- *PASTA on Edge: Cryptoprocessor for Hybrid Homomorphic Encryption* by Aikata, Daniel Sanz Sobrino, and Sujoy Sinha Roy, appeared in IEEE DATE 2025.

- *Attacking and Securing Hybrid Homomorphic Encryption Against Power Analysis* by Aikata, Maciej Czuprynko, Nedzma Musovic, Emira Salki, and Sujoy Sinha Roy, appeared in IEEE DATE 2026.

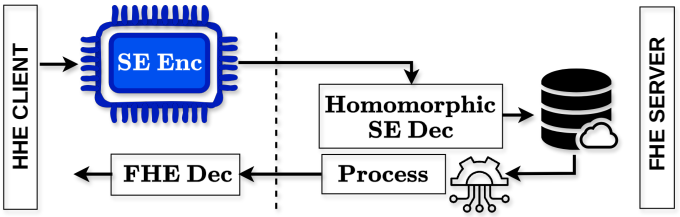


Figure 6.1: Workflow of HHE schemes is illustrated here. Symmetrically encrypted ciphertexts are sent to the server. The server performs a 'homomorphic HHE decryption' and obtains a homomorphically encrypted ciphertext.

6.1 Introduction

The FHE schemes transform plaintext data into much larger polynomials when homomorphically encrypted, resulting in substantial communication and computational overhead, often ranging from $10,000\times$ to $100,000\times$ [Jun+21c]. Generating and transmitting these large ciphertexts is very expensive for the clients due to limited processing capabilities. To address these, HHE was introduced [GHS12], preventing expensive data expansion. It essentially involves using Symmetric key-based Encryption (SE) methods that can be decrypted in a homomorphic manner. The data flow from HHE to FHE is illustrated in Figure 6.1. Initially, HHE schemes were designed to work with binary data ($\mathbb{Z}_2$), such as Rasta [Dob+18a], Flip [Méa+16], Filip [Méa+19], Kreyvium [Can+18], and AgRasta [Dob+18a]. However, in response to the growing need for better performance and application support, especially in integer FHE, they have evolved into schemes like Masta [Ha+20], Pasta [Dob+23], Hera [Cho+21], and Rubato [Ha+22] that can encrypt messages in prime fields $\mathbb{F}_p$, enabling operations on integers.

Although these schemes are lightweight and offload less computation to the server, they are more expensive in terms of computational requirements than schemes over Boolean data. This is because they operate over integers and require integer arithmetic. Furthermore, the round function includes expensive operations, such as matrix multiplication and generating invertible matrices, for matrices of size 32×32 or 128×128. The size of the integers can be up to 64-bit, making these operations expensive in terms of both computation and storage. Therefore, it is worth exploring how optimally we can implement them in constrained hardware or as System-on-Chip (SoC).

6.1.1 Research Gap

■ **Difference with Traditional SE.** Modern SE primitives used within the HHE framework are significantly different from traditional SE primitives such as AES [NIS01] or ASCON [Dob+21]. The primary difference is the data type.

While the traditional SE schemes are defined over boolean values ($\mathbb{Z}_2$), new HHE enabling SE schemes are defined over integers ($\mathbb{F}_p$). Thus, the data path increases significantly based on the size of the modulus p. This also affects the operations/functions computed on the data. Traditional SE schemes use boolean AND, OR, and XOR operations, which are very cheap compared to integer multiplication, addition, and subtraction required by HHE enabling SE.

The SE schemes have two layers, substitution (S-box) and permutation, which provide necessary confusion and diffusion. While traditional schemes can utilize a look-up-table-based S-box, this approach is not feasible for the new schemes due to the sheer size of the data (32-128 state elements, each of size 17-60 bits). Similarly, permutation in traditional schemes involves shifting rows or columns and mixing within columns where state sizes are small (e.g., 128 bits). The permutation in new SE schemes requires the generation and multiplication of expensive invertible matrices of sizes 32×32 to 128×128. Furthermore, while the traditional schemes have standalone descriptions, the new SE schemes utilize another expensive cryptographic standard- SHAKE128 (based on Keccak) as a pseudo-random number generator. Thus, they are bound to consume significantly more area and time than traditional SE schemes.

■ **The Challenge.** While the above subsection brings forth an important discussion regarding the new HHE enabling SE schemes being more expensive than traditional SE schemes, it does not give an idea of the challenge of implementing them and the expected results compared to existing public-key encryption (PKE) client-side routines. In most prior PKE client-side acceleration works, the polynomial degree is $N = 2^{13}$, and for polynomial multiplication (the *most expensive* routine), a Number Theoretic Transform (NTT) based technique is utilized. Each NTT transform requires $\frac{N \cdot \log N}{2}$ multiplications. The number of such transformations required is three per modulus, which is done for three different moduli. Thus, the total number of multiplications required is $\approx 2^{19}$.

For the HHE enabling SE scheme- Pasta, there are two variants- Pasta-3 and Pasta-4. Pasta-3 the state comprises 256 elements, split into two parts of 128 each. The number of rounds is three, and each round involves a 128×128 invertible matrix generation and multiplication. The multiplicative complexity of both operations is the same- $128 \cdot 128$, and eight such operations are required. This brings the total multiplication cost to 2^{18}. It is interesting to note that, while FHE encrypts 2^{12} elements in 2^{13} degree polynomials, Pasta-3 can only encrypt $128 = 2^7$. Therefore, although Pasta-3 encrypts 32 elements twice as fast, it will need 2^6 more encryptions to encrypt 2^{12} elements, resulting in $32\times$ slower computation for data-intensive applications.

Furthermore, HHE, unlike FHE, requires SHAKE128, a giant building block even in Post-Quantum schemes [Aik+22]. Thus, while the new SE schemes offer an advantage in terms of $6\times$ lower communication, any performance or area consumption advantage is not apparent. Given that communication overhead plays a crucial role, HHE requires a thorough design exploration to offer accelerated computation and determine if the communication advantage justifies potential trade-offs.

■ **The Implementation Gap.** To the best of our knowledge, an HHE scheme has not yet been implemented in hardware. While several accelerators exist for the server [Aik+25; Gee+22; Mer+23], there are very few addressing the needs of the FHE client. The prior works on FHE PKE accelerators [Aza+22; Aza+23; Kri+24; LDL23; MGS23] can mitigate computation costs, but communication overhead remains a major bottleneck in practical use. Additionally, the platforms these accelerators target (e.g., AlveoU250) are prohibitively expensive for typical clients. As a result, clients face both *computational* and *communication* overheads. This makes it essential to explore how efficiently HHE schemes, which prevent communication overhead, can be implemented on resource-constrained hardware or as System-on-Chip (SoC).

Furthermore, the practical implementations of HHE remain unexplored in the context of power analysis. To the best of our knowledge, no prior work has investigated power side-channel attacks on HHE-tailored schemes, although several studies have examined their vulnerability to fault attacks [Aik+26b; Rad+23; RBM21]. This absence of exploration leaves a critical gap: while the underlying cryptographic constructions offer strong theoretical security, their physical realizations may still leak exploitable information.

Moreover, in the absence of any prior power analysis attacks, there has likewise been no study of countermeasures designed to protect HHE schemes against such threats. As a result, both the offensive and defensive dimensions of side-channel security for HHE remain open problems. Investigating this gap introduces several unique challenges:

■ **High-Dimensional Leakage and Large Key-space.** Unlike lightweight block ciphers (e.g., AES[NIS01], ASCON [Dob+21]), where leakage often comes from compact operations like S-boxes, integer-based HHE-tailored schemes (e.g., Pasta, Masta, Hera) rely on modular arithmetic. This spreads side-channel leakage across many intermediate values and low-level processor operations, making it harder to locate exploitable points. Keys also consist of many integer coefficients, creating much larger secret and state spaces. As a result, standard side-channel methods that target small subkeys (like 8-bit AES bytes) do not apply naturally. Understanding this leakage is crucial as integer-based designs support integer FHE schemes (e.g., CKKS[Che+17], BGV/FV[BGV11; FV12]) and thus efficient HHE [ASR25b].

■ **Randomness and Transition Leakage.** Securing integer-based HHE-tailored schemes against side-channel analysis requires repeated masking of multiplications with fresh randomness and frequent mask refreshes, which is problematic on constrained devices where randomness generation and management are costly. Moreover, while formal proofs cover the probing model [CS20], software implementations face transition leakage when multiple shares reside in the same stack/register. Balancing robustness against transition leakage with practical efficiency thus remains a key challenge for masking.

6.1.2 Our Contribution

This Chapter addresses the aforementioned research gap and proposes a hardware accelerator for an HHE scheme- Pasta. This is the first hardware realization of any HHE scheme. As a system supporting FHE client, it reduces the communication overhead due to its use of HHE and offers several orders better performance and energy efficiency than software and prior client-side PKE accelerators. The primary design technique includes the optimal realization of the invertible matrix generation and multiplication step. The design further features efficient resource sharing amongst all the routines. The ASIC results show a speedup of 97× compared to prior PKE client accelerators. This design is further realized as an RISC-V SoC on a low-end 130nm ASIC technology and delivers up to 171× speedup compared to a CPU. We report a 857-3,439× reduction in clock cycles compared to [Dob+23] on CPU. Furthermore, we make our design publicly available[1] to support further research.

Next, this thesis addresses the lack of side-channel analysis and masking techniques for integer-based HHE schemes. We present the first comprehensive CPA (Correlation Power Analysis) evaluation of Hera. We also show that our technique is applicable to all the other HHE-tailored SE schemes defined over integers. Prior fault-based attacks on such schemes [RBM21] rely heavily on nonce reuse, while the only exception [Aik+26b] introduced a model that bypasses this restriction but still requires knowledge of the function evaluated. In contrast, our CPA approach does not require any of these assumptions and still achieves full key recovery with 40 traces, making it a far more practical and general threat. We demonstrate the attack using the Chipwhisperer CW308 UFO board with STM32F303 Target, and the capture is done using the CW1173 ChipWhisperer-Lite.

Finally, on the defensive side, we propose the first masked Hera realization, combining glitch-robust Probe Isolating Non-Interference (PINI) secure gadgets with assembly-level countermeasures against transition leakage. Along with providing proofs for security, we also validate them by performing Test Vector Leakage Assessment (TVLA). We further discuss how these masking techniques can be extended to other HHE-tailored schemes by extending it to Pasta. Our work, therefore, establishes leakage models, attack targets, and benchmarks as a foundation for systematic analysis. We further study the trade-offs between performance and leakage resistance, showing that even partial leakage can be neutralized with careful design. To support reproducibility and foster future research, we release the complete artifact package, including implementations and the power-trace dataset of our CPA attack on Hera, masked implementations of Hera and Pasta, as well as their corresponding leakage assessments[2].

The relevant background is described in Section 2.3. The roadmap through the Chapter is as follows. The proposed design methodology is presented in Section 6.2. Detailed evaluation results and comparisons are presented in Section 6.3.

[1] The design and its RISC-V integration are available at `https://github.com/aikata10/Pasta_RISCV`

[2] `https://github.com/aikata10/Masking_HHE`

The power analysis discussion is present in Section 6.5, and masking is described in Section 6.6. Section 6.7 concludes the Chapter.

6.2 Design Methodology

In [Dob+23], the authors have reported the clock cycle count for Pasta encryption (also stated in Table 6.2). They further give a small breakdown showing that the affine generation alone consumes 54-60% of the total clock cycles. Thus, apart from the generation of invertible matrices and matrix multiplication, the generation of pseudo-random numbers plays a significant role in determining the overall performance. With this important observation from the scheme designers of Pasta, we explore techniques for designing a highly parallel, high-performance pipelined design for modulus p in the range of 17-54 bits. We do not perform round-unrolling so that the design can fit in small FPGAs (e.g., Artix-7), which are more suitable for clients.

6.2.1 XOF and Rejection Sampling Unit

A significant challenge arises in the efficient generation of pseudo-random data required by the Pasta-3/-4 cryptographic schemes, which demand 2048/640 coefficients. After consuming the nonce and counter, the XOF unit generates one 64-bit coefficient per clock cycle, followed by a rejection sampling step. This process continues until all 21 pseudo-random coefficients are utilized. At this point, the XOF performs another 24-round Keccak permutation and generates the next set of 21 coefficients. This cycle repeats until the necessary amount of data is produced. A key challenge is that rejection sampling can result in up to half of the generated data being discarded, placing strain on the throughput.

To address this, we implemented a high-performance SHAKE128 design [Aik+22] that tackles this bottleneck by executing Keccak permutations in parallel with the squeeze operation, nearly doubling performance. However, this enhancement requires two 1600-bit state buffers, increasing resource usage. Furthermore, the rejection sampling procedure for generating matrix and round constant vectors is identical, necessitating an efficient integration with the XOF routine. We addressed this by incorporating a DataGen unit, which operates in a ping-pong configuration: while one vector is used to generate the matrix, the other stores XOF results for the subsequent computation. This optimization, as illustrated in Figure 6.3, enables continuous data flow and improves overall throughput.

6.2.2 Operation Schedule

To ensure high performance, the XOF output must be consumed as soon as it is generated. The rate for the rejection sampling utilized can be any value between 1-2×; thus, an XOF data vector can be ready in at least t clock cycles. Each round requires the generation of four random vectors for the affine

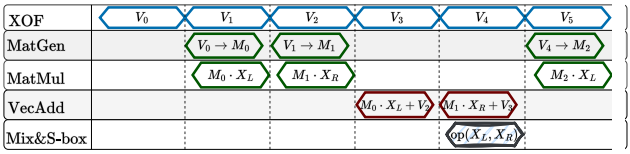


Figure 6.2: The schedule for one round of the PASTA-π permutation.

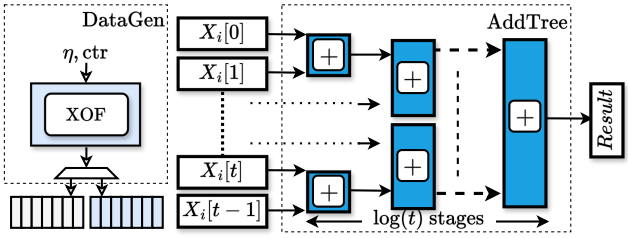


Figure 6.3: The Data Generation and Adder-Tree units utilized in Matrix Generation and Multiplication. The log operation has base 2.

transformation: the first two are used for matrix generation, while the latter two serve as round constants for addition. Matrix generation is followed by matrix multiplication, while Mix and S-box operations follow the round constant addition. The challenge lies in ensuring full parallelizationmatrix multiplications and Mix/S-box operations must be completed in time for the next round of XOF data to become available. Careful coordination is needed to prevent bottlenecks and maintain continuous data flow between rounds.

The overall flow, illustrated in Figure 6.2, shows how the random vectors generated by the XOF (denoted as V_i) are either fed into the MatGen or VecAdd. Matrix multiplications are concurrent with matrix generation, while Mix and S-Box operations follow vector addition. This design ensures the on-time completion of each computation before the next round of data is generated, enabling a balance between parallelism and throughput.

6.2.3 Invertible Matrix Generation & Multiplication Unit Design

Matrix generation and multiplication introduce a computational challenge, as they require $2t^2$ modular multiplications. To finish both tasks within the stipulated time of t clock cycles, we instantiate two multiplication sets, each consisting of t modular multipliers. One set is dedicated to generating invertible matrices. As shown in (2.1), matrix generation also requires addition. Therefore, the unit instantiated for MatGen is a MAC (Multiply-and-Accumulate) unit, as illustrated in Figure 6.4.

The matrix generation unit generates the matrix row by row. Therefore, the multiplication is also done row-wise using MatMul unit, shown in Figure 6.4. The

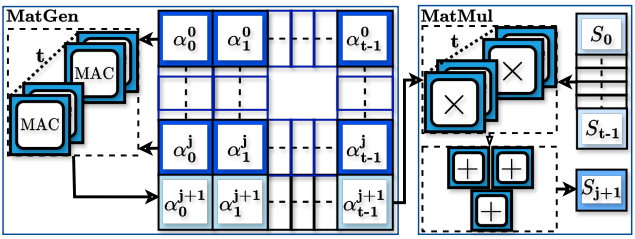


Figure 6.4: The Invertible Matrix Generation and Multiplication Units.

state vector is multiplied by a row for matrix multiplication, and all the resultant elements are accumulated, similar to a vector dot-product. The multiplication is done via the second set of t multipliers. We use a pipelined adder-tree unit for additions, illustrated in Figure 6.3. The time consumed for matrix generation and multiplication is $6 + t + \log_2 t$ clock cycles, accounting for the additional cycles needed to support the pipelined data flow between the MAC and matrix generation/multiplication stages.

This technique not only achieves high performance but also addresses another significant challenge memory efficiency. By eliminating the need to store large 32×32 or 128×128 element matrices, our approach significantly reduces the memory footprint. For matrix generation, we only store the minimal set of two required rows (α^0,α^j) to generate subsequent (α^{j+1}) ones, as highlighted in Figure 6.4. This is due to the fact that the resultant (α^{j+1}) is immediately consumed for matrix multiplication, resulting in substantial area savings in terms of memory without compromising the throughput.

6.2.4 VectorAddition, Mix, and S-Box Unit Design

Similar to multiplication units, we employ t vector addition units. The number of vector additions required per state is t. Therefore, this unit barely consumes three clock cycles, wondering pipelining. The reason for instantiating it in a fast mode becomes evident when considering the fact that Vector Addition (RC Add), Mix, and S-Box operations need to finish in parallel with the generation of V_4 in Figure 6.2. The Mix operation requires the computation of $(2 \cdot X_L + X_R, 2 \cdot X_R + X_L)$. We compute this solely using additions. Thus, three additions are required: (i) $X_R + X_L$, (ii) $X_R + (X_R + X_L)$, and (iii) $X_L + (X_R + X_L)$. The addition unit used for vector addition is also utilized for these additions, thus ensuring resource sharing.

Following this, the S-Box operations present an additional challenge. Pasta employs two S-Boxes: the cube S-Box (`S`), which requires two multiplications, and the Feistel S-Box (`S'`), which involves one multiplication and one addition. As a result, each round must compute four additions and two multiplications (or five additions and one multiplication) in parallel with the generation of t elements. The same multiplication (Mul) and addition (Add) units used in the affine transformation (MatMul, RC Add) are reused for these operations, increasing

resource sharing. Since addition units are relatively low-cost, t-instantiations avoid the overhead of pipelined intermediate storage using additional flip-flops.

All the multiplication and addition units operate modulo a prime p. We observe that the moduli chosen by the authors in [Dob+23] have a Mersenne structure (e.g., 17-bit prime 65,537), enabling the use of an add-shift-based modular reduction unit following each multiplication. This technique is also utilized by prior FHE acceleration works [Mer+23].

6.2.5 Overall Architecture Design

The overall architecture of the design is shown in Figure 6.5. The user provides a nonce (η), counter (ctr), and message block m, with the output being the ciphertext c. The ciphertext c is produced by performing a modular addition between the input message and the keystream (KS), which is the result of the Pasta-π permutation. The DataGen, Modular Multiplier (Mul), and Adder (Add) units are instantiated within a wrapper, enabling their reuse by both the MatMul and RC Add/Mix+S-box units for efficient resource utilization.

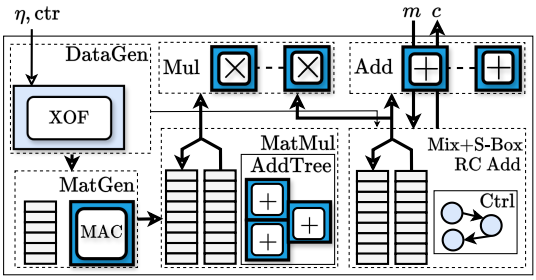


Figure 6.5: The overall architecture of the hardware design for Pasta.

6.3 Results

6.3.1 Evaluation Platforms and Area

We chose three evaluation platforms to analyze the design for varying use cases comprehensively. We first synthesize and implement our design on the Artix-7 FPGA. We synthesize it for 7nm and 28nm ASIC technologies, and provide a RISC-V SoC using the Ibex Core for 130nm technology.

❶ **FPGA.** We set our target as the low-cost Artix-7 AC701 (xc7a200tfbg676-2) FPGA at 75MHz. This FPGA features 134k Look-Up-Tables (LUTs), 269k Flip-Flops (FFs), 740 Digital Signal Processors (DSPs), and 365 Block Random Access Memories (BRAMs) with 36kb capacity. We used Xilinx Vivado 2022.2 to run synthesis, place & route. FPGA Area results are reported in Table 6.1, and the area breakdown is reported in Figure 6.6. The design does not require any BRAM or URAM-based memory storage. Therefore, it is not reported in the table. For Pasta-4, we place the design for a bit-width up to 54-bit to perform

Table 6.1: The implementation results of Pasta-3/4 on the Artix-7 board targeting 75MHz clock frequency.

Scheme	$\omega^{\dagger}$	LUT		FF		DSP	
Pasta-3	17	65,468	49%	36,275	13%	256	35%
Pasta-4	17	23,736	18%	11,132	4%	64	9%
	33	42,330	32%	20,783	8%	256	35%
	54	67,324	50%	32,711	12%	576	78%

† 'ω' denotes the size of the Mersenne moduli.

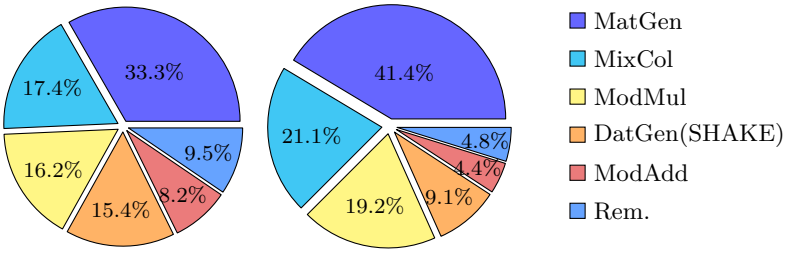


Figure 6.6: Module-wise area utilization. The first pie chart shows the distribution for FPGA and the second for ASIC platforms. Rem. implies Remaining Area.

a comprehensive analysis; however, for Pasta-3, we could not place a bit-width of 33-bit or more due to insufficient area for our design goals.

❷ **ASIC.** We also present ASIC-synthesis results using TSMC 28nm and ASAP7 [Cla+16] 7nm libraries to compare with prior ASIC works [Aza+22; Aza+23]. The synthesis is performed using Cadence Genus 2019.11, and clock gating is employed to help reduce power consumption with a clock frequency target of 1 GHz. It reports an area consumption of 0.24 mm^2 and 0.03 mm^2 on the 28nm and 7nm processes. The maximum power consumed by the design is 1.2W. For a bit-width of 33 and 54, the area increases by $\approx 2.1\times$ and $\approx 4.3\times$.

❸ **RISC-V.** A peripheral has been developed to integrate the Pasta encryption module into the 32-bit RISC-V [Ibe] architecture. It is connected to the central Ibex core data bus using a loosely coupled design. This peripheral is connected to the RAM via an extra bus where the peripheral acts as a master, allowing direct read access to obtain the plaintexts to be processed and subsequently encrypted. The RISC-V SoC targets 100MHz on older 130nm and 65nm technology nodes and requires 1.8 mm^2 (4.6 mm^2 with Ibex core). The peripheral processes data block by block, allowing a flexible number of blocks to be encrypted. This design choice improves area utilization by reducing memory to a 544-bit Pasta state. Another limiting factor in the design is the use of the data bus connecting the core to the peripherals. The latter, as a slave, handles the encryption start signal, the memory address for reading into RAM, reading the key and nonce, and writing the encrypted data, all on a single bus. This prevents simultaneous reading and writing of data blocks. So, the processing of one block must be

Table 6.2: The performance results for Pasta-3/4 encryption of one block, along with the results for CPU reported in Pasta.

Scheme	Elements Processed	clock cycles	FPGA (μs)	ASIC (μs)	RISC-V (μs)
Pasta-3 [Dob+23]	128	17,041,380†	-	-	-
Pasta-3	128	4,955	66.1	4.96	45.5
Pasta-4 [Dob+23]	32	1,363,339†	-	-	-
Pasta-4	32	1,591	21.2	1.59	15.9

† The results are reported on an Intel Xeon E5-2699 v4 CPU [Dob+23].

completed before the next block can be started.

■ **Bitlength Comparison.** The performance stays the same for different bit lengths thanks to our design strategy. The area-time product increases by more than a factor of two when the bit length is doubled. Therefore, we will consider the design with a bit length of 17 bits for comparisons unless specified.

6.3.2 Performance Benchmarks.

In Section 6.2, we described our design strategy tailored to offer high speed, which is limited by the Keccak permutation implementation. SHAKE128 instance of Keccak has a rate of $1,344$ bits and generates 21 words (64-bit) after one permutation. Each permutation takes 24 clock cycles (cc). One round of Pasta-4 requires 32×4 pseudo-random state elements, one each for the two matrix multiplications and two round constant additions. Pasta-4 has five such rounds. Thus, a minimum of 31 Keccak permutation rounds is required.

A rejection sampling follows the pseudo-random number generation. In our case, we have a high rejection rate for the stated prime ($\approx 2\times$) of $65,537$ (0x10001). Therefore, the sampling rate almost doubles, and we require, on average, 60 Keccak permutation rounds. Overall, the Keccak round function alone consumes 1,440 cc (60×24). Sampling 21 elements would add another 21 cc after every round. Therefore, the clock cycle almost doubles for a naive Keccak implementation. However, the Keccak implementation utilised in this work allows us to perform the Keccak permutation in parallel with the squeeze operations, adding only an extra five clock cycles between two squeezes. This helps us reduce the clock cycles by half ($60 \cdot (21 + 5) = 1,560$cc).

The total clock cycle count is appended with $t = 32$ clock cycles required by the last remaining Mix operation after all the data has been generated, bringing the overall cycle count for Pasta-4 encryption to $1,592$, as given in Table 6.2. Similarly, for Pasta-3, the state size increases fourfold while operating for one less round. This gives the average number of Keccak calls as 186 and clock cycles for Keccak round function alone as $4,836$cc ($186 \cdot (21 + 5)$cc). The entire Pasta-3 encryption costs $4,964$cc due to an additional 128cc from the last mix operation. These bounds on the number of clock cycles are based on the rejection sampling rate. Consequently, the number of clock cycles varies slightly upon experimentation, based on the initiating nonce and counter.

Table 6.3: Comparisons of PASTA-4 Performance, Area with Prior Works.

Work	Platform	Area				Encr.‡ (μs)
		kLUT	kFF	DSP	BRAM	
[MGS23]	Zynq US+	-	-	-	-	7.79k (1.91)
[LDL23]	AlveoU250	1,179	1,036	12,288	828.5	16.9k (0.51)
[Kri+24]	Kintex-7	20.7	17.6	100	82.5	1.87k (0.46)
TW	**Artix-7**	**23.7**	**11.1**	**64**	**0**	**21.2 (0.67)**
[Aza+22]	12nm	0.06 mm^2				110k (26.9)
[Aza+23]	12nm†	0.11 mm^2				20k (4.88)
TW	**7/28nm**	**0.03/0.24** mm^2				**1.59 (0.05)**
TW	**65/130nm†**	**0.74/1.80** mm^2				**15.9 (0.50)**

‡ Runtime is reported for one encryption and per element in brackets.
† refers to RISC-V SoC, and 'k' refers to 1,000.

■ **PASTA-3 vs. PASTA-4. Area-Performance Comparisons.**
Table 6.2 lists the performance estimates upon experimentation on various platforms. While PASTA-3 consumes more runtime than PASTA-4 for their respective state size, PASTA-3 reports 22% less processing time than PASTA-4 for the same amount of data. In the previous section, we saw that PASTA-3 consumes approximately 3× more area than PASTA-4. This shows us that PASTA-4 offers a better area-time trade-off than PASTA-3 and should be preferred for client-side devices irrespective of the choice of the implementation platform.

6.3.3 Comparisons with Prior Works

Table 6.2 reports that our design requires 857-3,439× fewer clock cycles than CPU results presented in [Dob+23]. This translates to a speedup of 43-171× as the CPU runs at ≈ 20× higher clock frequency- 2.2GHz. Prior works [Aza+22; Aza+23; Kri+24; LDL23; MGS23] for FHE client-side encryption routine present designs targeting the PKE algorithm involving expensive polynomial arithmetic. No prior work has implemented new and more efficient *arithmetic* symmetric HHE enabling schemes. The parameters supported by the prior works are also insufficient to support bootstrapping in FHE. In contrast, implementing an HHE enabling SE scheme does not prevent bootstrapping.

❶ **FPGA works.** To ensure fair comparisons, the performance is also reported per element encrypted. The number of elements is 32 in our case, 2^{12} for [Kri+24; MGS23], and 2^{15} for [LDL23]. Prior works [Kri+24; LDL23; MGS23] utilize Zynq UltraScale+, AlveoU250, and Kintex-7 FPGAs. These boards have 2-10× more resources than our target Artix-7 board. Table 6.3 shows that our design consumes relatively less area and no BRAMs while delivering similar performance. Furthermore, we deliver similar performance while running our design at almost 2-3× lower clock frequency, thus lowering the overall energy consumption. For ML inference applications encrypting low amounts of data (e.g., 32 coefficients), we deliver much better performance (21.2 μs) as FHE will necessitate the same amount of computations (1,884 μs) for any amount of data upto 2^{12} coefficients.

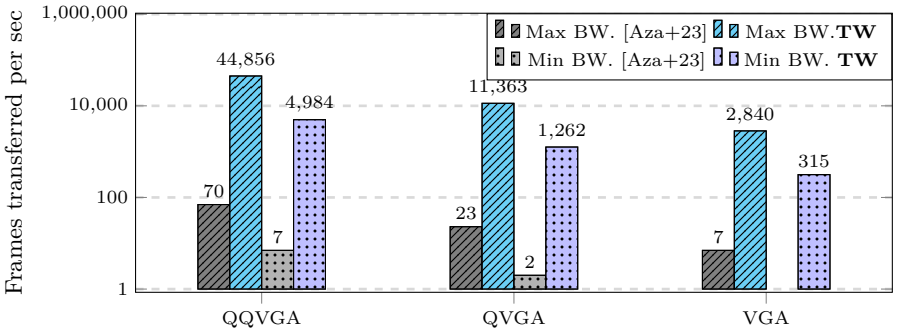


Figure 6.7: Frames transferred per second for Maximum (112.5 MBps, left) and Minimum (12.5 MBps, right) 5G available bandwidth. The graphs are on a logarithmic scale. TW refers to this work.

❷ **RISC-V works.** The existing RISC-V client-side FHE acceleration works [Aza+22; Aza+23] present a hardware-software co-design targeting 1GHz clock frequency on 12nm Technology. Their parameter choice can pack 2^{12} elements per encryption. The comparisons in Table 6.3 reported with these works showcase that the design for HHE enabling SE schemes offers 98-338× better performance as a standalone chip. The proposed RISC-V SoC on the old technology nodes (130nm, 65nm) performs 10-34× better while targeting 10× less clock frequency for low power. Note that the design consumes an area similar to [Aza+23] post-technology normalization.

6.4 Application Benchmark

We evaluate an application benchmark of the video frame encryption [Aza+23; UUS18] application, which sends videos to the cloud processors for surveillance [Lag+21]. The video is split into frames as per the required resolution, which can range from QQVGA (160 × 120 pixels), QVGA (320 × 240 pixels), to VGA (640 × 480 pixels). A grayscale pixel requires 8 bits of storage. The available bandwidth for data transfer ranges between 12.5 and 112.5 MBps (e.g., mid-band 5G network [TAC19]). Our results, compared with those reported in [Aza+23], are presented in Figure 6.7. RISE [Aza+23] can encode one QQVGA frame per ciphertext ($N = 2^{14}, \log Q = 390$) and requires three such ciphertexts for a QVGA frame. One ciphertext size is 1.5MB ($2^{14} \cdot 2 \cdot 390$). Therefore, they can send 70 QQVGA frames per second at the maximum 5G bandwidth. Our ciphertext for ($N = 2^5, \log q_0 = 33$) is only 132 Bytes in size ($2^5 \cdot 17$). Hence, we can send up to 712× more frames per second. [Aza+23] cannot send a VGA frame at the minimum bandwidth.

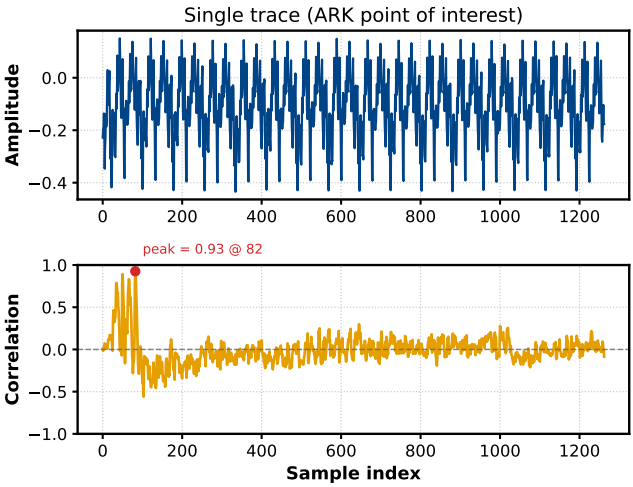


Figure 6.8: Representative power trace of Hera (top) and correlation for the correct key guess (bottom), highlighting the leakage peak at the point of interest.

6.5 The Correlation Power Analysis of Hera

In this work, we focus on Hera for evaluating side-channel resistance of prime-field lightweight ciphers. This is because Hera is specifically optimized for efficient arithmetic and employs low Hamming-weight matrices, making it well-suited for hardware implementations and resource-constrained platforms. The structure of Hera can also be regarded as an adaptation of earlier designs, combining linear diffusion layers with nonlinear S-Box operations in a manner similar to other schemes. As a result, insights gained from the cryptanalysis of Hera naturally extend to other ciphers, such as Pasta, Masta, and Rubato.

Among the different components of Hera, the `ARK` step is particularly vulnerable because it directly combines the secret key K with known intermediate values, making it a prime target for a CPA attack. The `ARK` operation is computed as

$$\texttt{ARK}[K, n, i](X) = X + K \odot RC^i \pmod{p} \tag{6.1}$$

where RC^i is the i-th round constant, and X is the state. Since the round constants can be deterministically regenerated from the public nonce, an attacker can predict the key-dependent intermediate values and correlate them with power traces. This makes it possible to mount a CPA on the modular multiplication (resp. addition).

6.5.1 Practical CPA on Hera: Experimental Setup and Results

To demonstrate the feasibility of the attack, we implemented Hera with parameters Hera `(5, 16,` $p = 2^{16}+1$`)` on the ChipWhisperer CW308 UFO board with

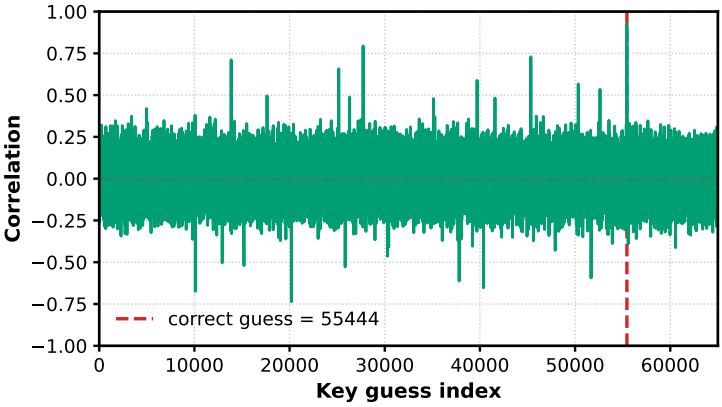


Figure 6.9: Correlation values for all key guesses at the point of interest, with the correct key clearly distinguished by the peak at index 55,444.

an STM32F303 target and captured traces using the CW1173 ChipWhisperer-Lite. The target of the attack is the first `ARK` operation in the first round. To perform the attack, we first collect multiple traces and the outputs of the targeted `ARK` operation in Eq. 6.1 (one such trace is represented in Fig. 6.8) with a fixed key and varying data. Then, we compute the predicted leakage as the Hamming weight of the targeted operation for each of the $2^{16}+1$ possible values of the first element of the secret key. Finally, we compute the Pearson correlation coefficient between the predicted leakage and the actual leakage, selecting the guess with the highest coefficient as the correct one. We illustrate this procedure in the second image of Fig. 6.8 by showing the correlation of the correct guess at each sample point. Fig. 6.9 illustrates the correlation values for all key guesses at the peak, and the correct key guess stands out with a dominant peak.

To recover the entire key, the same procedure is applied iteratively, with the only difference being the positions of the correlation peaks, as illustrated in Fig. 6.10. We perform the attack on 100 random keys to assess the success rate. Fig. 6.11 shows the success rate of the key recovery with respect to the amount of observed traces. The results demonstrate that the correct key can be reliably recovered with fewer than 40 traces, highlighting the vulnerability of the unmasked design.

While this demonstration shows that CPA is highly effective on the chosen parameter set, it is important to emphasize that the straightforward approach requires searching the entire space of an element of the secret key. For a modulus of size $2^{16}+1$, this corresponds to roughly 2^{16} candidates, which remains practical. However, as the bit size of the modulus increases, the complexity of brute-force searching grows exponentially. Advanced techniques, such as those proposed in [Tun+] and applied to a multiplication in [Che+21], address this limitation by incrementally recovering the secret, guessing only a few bits at a time and treating the rest as noise. The trade-off of such approaches is an increase in the

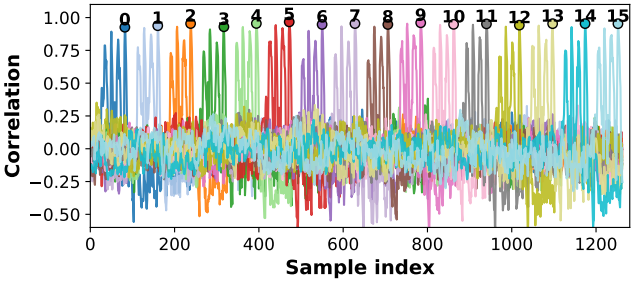


Figure 6.10: CPA results for all 16 key elements. Curves illustrate the correlation across hypotheses, with dots indicating the peak values that reveal the correct elements.

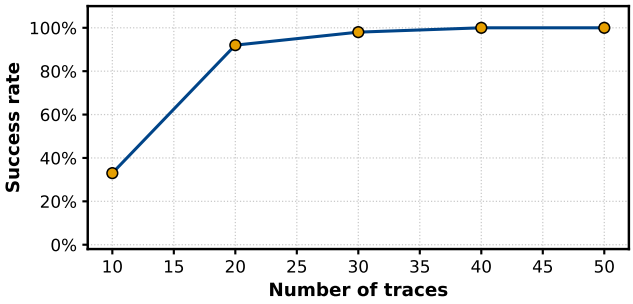


Figure 6.11: Success rate of the CPA attack as a function of the number of traces for 100 random keys, converging to full recovery after ≈40 traces.

number of traces required due to the added statistical noise.

It is important to note that the operation under consideration involves a multiplication with the key, a feature that is common across all recently proposed integer-based HHE schemes. For example, while Rubato employs the same key-schedule-like ARK step as Hera, both Pasta and Masta initialize the state with the key and subsequently apply a matrix multiplication with the state, where the matrix itself is derived from the nonce and hence public. Our attack naturally extends to this operation as well. Thus, all of these schemes are susceptible to the established CPA technique.

6.6 Masking Technique

To prevent DPA, one of the most established protection techniques is *masking*, which randomizes intermediate computations so that single observations reveal no exploitable information about the underlying secrets. In particular, we focus on *first-order arithmetic masking*, where each sensitive variable $x \in \mathbb{Z}_p$ is decomposed into two uniformly distributed random shares, x_0 and x_1, such that:

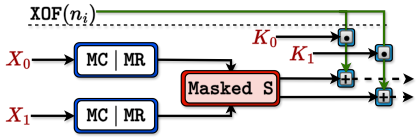


Figure 6.12: Dataflow of first-order arithmetic masking in HERA. Each sensitive intermediate x is decomposed into two shares (x_0, x_1), which are processed independently throughout the cipher. Linear operations (e.g., addition, modular reduction) are applied share-wise, while nonlinear operations (S-box) are adapted to operate on masked inputs.

Algorithm 20 Masked `ARK+MC+MR`

Require: State shares $S_0, S_1 \in \mathbb{Z}_p^n$, round constants $RC^i \in \mathbb{Z}_p^n$
Require: Key shares $K_0, K_1 \in \mathbb{Z}_p^n$, Matrix $\mathbf{M} \in \mathbb{Z}_p^{n \times n}$
Ensure: Output shares $V_0, V_1 \in \mathbb{Z}_p^n$
1: $T_0 \leftarrow S_0 + K_0 \cdot RC^i \pmod p$ ▷ `ARK` on share 0
2: $T_1 \leftarrow S_1 + K_1 \cdot RC^i \pmod p$ ▷ `ARK` on share 1
3: $U_0 \leftarrow \mathbf{M}\,T_0 \pmod p$, $U_1 \leftarrow \mathbf{M}\,T_1 \pmod p$ ▷ `MC` on share 0,1
4: $V_0 \leftarrow U_0\,\mathbf{M}^\top \pmod p$, $V_1 \leftarrow U_1\,\mathbf{M}^\top \pmod p$ ▷ `MR` on share 0,1
5: **Return** V_0, V_1

$x = x_0 + x_1 \bmod p$.

The key $K \in \mathbb{Z}_p^n$ is decomposed into two uniformly random shares, $K_0, K_1 \in \mathbb{Z}_p^n$, with $n = 16$ corresponding to the HERA's state size (see Fig. 6.12). The masked key shares are then processed through the Affine Layer, with `ARK`, `MC`, and `MR` applied independently to each share (Alg. 20), and `MskSB` operating jointly on both shares (Alg. 22). This sequence repeats over five rounds. Throughout the computation, the state is maintained as two shares X_0 and X_1, satisfying $X_0 + X_1 \equiv X \pmod p$. The output of each gadget directly serves as the input to the next, preserving the masked representation.

Proposition 9. *Independently applying linear gadgets `ARK`, `MC`, `MR` is first-order probing secure. Their output shares show uniform distribution and are independent of the input shares.*

Proof. The `ARK`, `MC`, and `MR` operations of HERA are inherently linear and preserve the masking order by construction. Therefore, we can apply these operations independently to each share (see Alg. 20). Each operation acts linearly on a single share, and each input share is uniformly random; therefore, any internal variable or output share is uniformly distributed and independent of the other shares. As a result, any adversarial probe reveals at most one share, making the operations naturally 1-PINI secure [CS20]. □

HERA's S-box introduces the nonlinear confusion by cubing the state. The core challenge in masking this operation lies in realizing a secure masked multiplication. We therefore propose using the gadget `MskMul` [Cas+23] (as presented in Alg. 21) and apply it twice, to square the state and then to cube it, as shown in Alg. 22.

Algorithm 21 `MskMul` (Masked Multiplication)

Require: Shares $x_0, x_1, y_0, y_1 \in \mathbb{Z}_p$, with $x = x_0 + x_1$, $y = y_0 + y_1$, Fresh randomness $r \in \mathbb{Z}_p$
Ensure: Shares $z_0, z_1 \in \mathbb{Z}_p$, s.t. $z_0 + z_1 \equiv x \cdot y \pmod p$

1. $t_{00} \leftarrow x_0 y_0$, $t_{01} \leftarrow x_0 y_1$, $t_{10} \leftarrow x_1 y_0$, $t_{11} \leftarrow x_1 y_1$
2. $v_0 \leftarrow r + t_{10}$, $v_1 \leftarrow v_0 + t_{01}$
3. $z_0 \leftarrow p - r + t_{00}$, $z_1 \leftarrow v_1 + t_{11}$

Return z_0, z_1

Hence, the security of masking this layer primarily relies on the security of the multiplication gadget.

Proposition 10. *The gadget `MskMul` is first-order probing secure. Its output shares show uniform distribution and are independent of the input shares satisfying* $z_0 + z_1 = x \cdot y \pmod p$.

Proof. Assume that x_0, x_1 and y_0, y_1 are uniform arithmetic shares of $x, y \in \mathbb{Z}_p$, such that $x = x_0 + x_1 \pmod p$ and $y = y_0 + y_1 \pmod p$. We show that any probe on an intermediate value is independent of the secrets x and y. The intermediate terms $t_{00} = x_0 \cdot y_0$, $t_{01} = x_0 \cdot y_1$, $t_{10} = x_1 \cdot y_0$, and $t_{11} = x_1 \cdot y_1$ each involve at most one share of each input. Since no partial product simultaneously involves both x_0 and x_1 (or, analogously, y_0 and y_1), and every share is uniform and independent of the secret, these products are independent of x and y. Through the variable v_1, we sum the intermediate multiplications t_{00} and t_{10} to obtain the output share z_1.

To carry out this summation without revealing the secret x (since $t_{00} + t_{10} = y_0 \cdot x$), we must mask this intermediate result. To achieve this, we sample a random mask r uniformly and independently. Hence, $v_0 = r + t_{10}$ is uniform in $\mathbb{Z}_p$, independent of all secrets; consequently $v_1 = v_0 + t_{01}$ is also uniform and independently of the secret state x. The output share $z_0 = (p - r) + t_{00}$ is uniform because it represents an offset of the uniform variable, $(p - r)$ (independent of the secrets). The output share $z_1 = v_1 + t_{11}$ is an offset of the uniform variable v_1; hence, it is also uniformly distributed and independent of the secret, thanks to the randomness in v_1. Therefore, in all steps, no probe leaks any information about the inputs x and y, while ensuring the correct computation of the desired multiplication, since:

$$z_0 + z_1 = p - r + t_{00} + r + t_{10} + t_{01} + t_{11} = x \cdot y \pmod p$$

□

Proposition 11. *The gadget `MskSB` is first-order probing secure. Its output shares show uniform distribution and are independent of the input shares, satisfying* $z = z_0 + z_1 = x^3 \pmod p$.

Proof. Since we have proven the first-order PINI security of `MskMul`, any gadget that uses it while preserving the masking structure is also first-order PINI secure by composition. In particular, this applies to the S-box layer, where we apply the multiplication gadget twice. Notably, we refresh the masks of the shares before and after each multiplication. Mask refreshing is essential to maintain

Algorithm 22 `MskSB` (Masked S-box cube)

Input: Shares $X_0, X_1 \in \mathbb{Z}_p^n$
Requires: $R_1, R_2 \in \mathbb{Z}_p^n \leftarrow$ fresh randomness
Output: Shares $Z_0, Z_1 \in \mathbb{Z}_p^n$, s.t. $Z_0 + Z_1 = X^3 \pmod p$

for $i = 0$ to $n-1$ **do**
  $y_0 \leftarrow X_{0i} + R_{1i} \pmod p$
  $y_1 \leftarrow X_{1i} + p - R_{1i} \pmod p$
  $v_0, v_1 \leftarrow$ `MskMul` $(X_{0i}, y_0, X_{1i}, y_1)$
  $t_0 \leftarrow v_0 + R_{2i} \pmod p$ ▷ Share Refresh
  $t_1 \leftarrow v_1 + p - R_{2i} \pmod p$ ▷ Share Refresh
  $Z_{0i}, Z_{1i} \leftarrow$ `MskMul` (t_0, y_0, t_1, y_1)
end for
Return Z_0, Z_1

the security of masked computations, as multiplications in arithmetic masking are nonlinear when both operands are secret, which may induce correlations between shares and potentially leak secret information. The mask refreshing is first-order PINI secure by design, as it is performed independently on distinct shares - y_0, y_1, t_0 and t_1 each depend on a single share and fresh randomness (R_1 and R_2). Moreover, the output shares of `MskSB` are identical to those of `MskMul`, preserving their uniform distribution. Hence, the S-box layer is first-order PINI secure. □

So far, we have analyzed individual gadgets within one Hera round and established their first-order probing security. We now demonstrate that the repeated application of these gadgets within and across rounds extends this property to full encryption.

Proposition 12. *The full encryption, composed of first-order probing-secure gadgets, is itself first-order probing secure.*

Proof. Sequentially composing 1-PINI secure gadgets over multiple rounds does not introduce any single-probe leakage, since each gadget exposes only masked shares and uses independent randomness. Therefore, the first-order probing security is preserved for the full encryption. □

6.6.1 Practical Implementation Strategies

Importantly, the PINI proofs presented do not account for implementation-specific issues, such as transition leakage. While classical Hamming-distance leakage mostly comes up in the discussion in the context of Boolean sharing [Bal+14], arithmetic shares can also be affected by transition leakage. In particular, if multiple arithmetic shares reside temporarily in the same register, the bitwise transitions caused by updates of arithmetic operations may leak information about the secret through correlations between successive register states.

To mitigate this risk, we enforce strict code-level isolation of operations on individual shares within the affine layer in the C implementation. For the nonlinear layer, we implement all share manipulations in assembly, giving us

Table 6.4: Comparison of not masked and masked HERA implementations in terms of average clock cycles and stack usage.

	Clock cycles		Stack usage	
	Not Masked	Masked	Not Masked	Masked
HERA	38,249	4,853,801	700 bytes	852 bytes
PASTA	4,249,061	15,549,629	1,756 bytes	2,072 bytes

complete control over register allocation and lifetime. Moreover, all functions take only the addresses of shares as input, so the shares never reside in registers in an uncontrolled manner. To further reduce risk, we introduce non-functional memory reads and writes to separate consecutive accesses to the shares. By following these heuristics, we ensure each share resides in a dedicated register, with minimal lifetime and without unintended overlaps.

Specifically, the multiplication gadget `MskMul`, which includes mask refreshing and subsequent share updates, is implemented entirely in assembly for the Cortex-M4 and assembled using GCC 13.2.1. We compile and link with `-fno-lto` flag to avoid link-time transformations (e.g., cross-module inlining, or dead-code elimination) that could alter the call context, register/stack behaviour, or remove security-relevant structure around the gadget. This approach preserves the independence of shares assumed in the formal PINI proofs and yields the results summarized in Tab. 6.4.

6.6.2 Test Vector Leakage Assessment

We validated our implementation via the TVLA, which detects whether any first-order leakage remains in the implementation by evaluating the t-test scores across fixed versus random inputs [GJR+11; SM16]. Passing first-order TVLA tests provides strong evidence that the implementation does not exhibit detectable first-order leakage beyond the scope of the theoretical model. Therefore, we run the t-tests for 10,000 traces and follow the common practice of using a detection threshold of ± 4.5 for the TVLA t-statistic. As depicted in Fig. 6.13, the t-values remain within this interval, indicating no statistically significant first-order leakage.

6.6.3 First-Order Masking Across Related Cipher Schemes

We can apply the masking techniques utilized for HERA across related HHE cipher schemes like MASTA [Ha+20] or PASTA [Dob+23], which also operate over an arithmetic ring. The affine layer of these ciphers allows trivial masking by applying each linear operation independently to each share. These ciphers introduce the nonlinearity through multiplication, in the same manner as in HERA. In particular, PASTA employs a Feistel-like S-box $\left(X^j + [j \neq 0]\,(X^{j-1})^2\right) \pmod p$ in every round except the final one. We can implement the core of this S-box's masking using our `MskMul` multiplication gadget. In its final round, PASTA instead uses the same cubic S-box layer as HERA, which further aligns

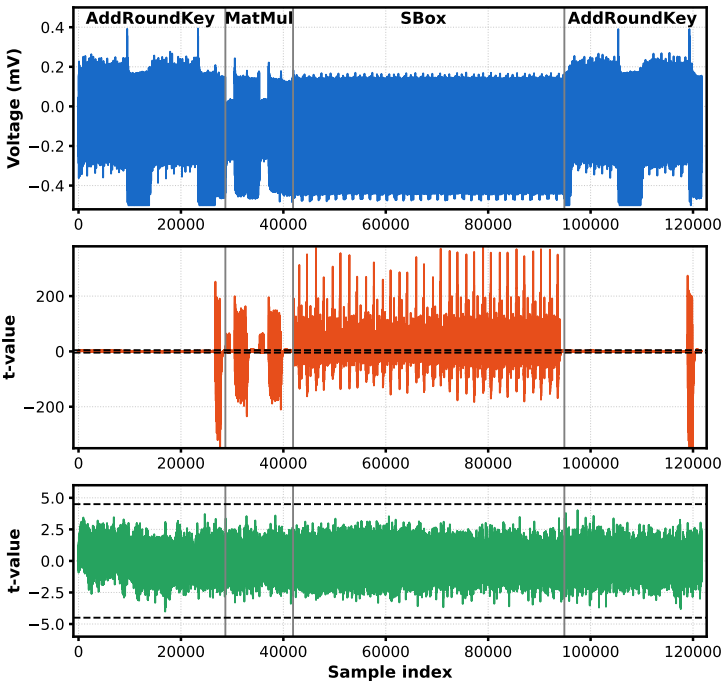


Figure 6.13: Hera power traces and corresponding t-test evaluations. The top plot shows a Hera encryption trace for one round, with vertical dashed lines marking major computation phases (AddRoundKey, MatMul, SBox, AddRoundKey). The middle plot presents TVLA results with faulty randomness, where the test statistic exceeds the ś4.5 threshold lines in red, revealing clear leakage. The bottom plot shows the same TVLA evaluation with proper randomness for 10,000 traces, where the values remain within the thresholds, indicating the absence of significant leakage.

the masking strategies of the two ciphers. Following this, we also masked Pasta (results in Table 6.4) and validated it with TVLA, as shown in Figure 6.14.

6.6.4 Discussion on Higher-order extension

An adversary may exploit higher-order power analysis by correlating information across multiple shares or intermediate values. It is therefore important to emphasize that first-order PINI proofs provide security only against single-probe attacks; achieving resistance against more advanced side-channel adversaries requires higher-order masking or additional countermeasures. The arithmetic masking scheme employed here can be extended to order d security by decomposing each secret into $d + 1$ shares and preserving statistical independence throughout the computations. Linear layers apply directly to each share, while the nonlinear layer - particularly the multiplication gadget - must be adapted

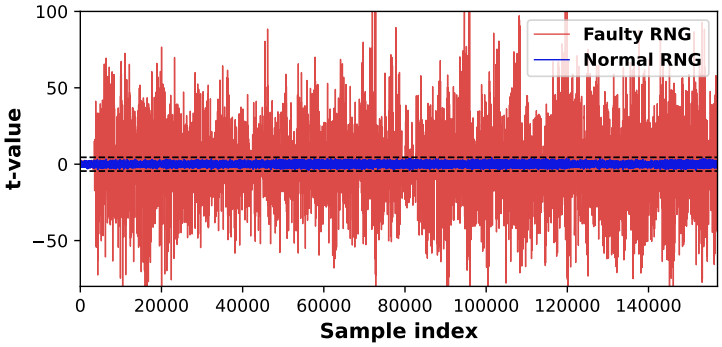


Figure 6.14: Leakage assessment of the Feistel S-box in Pasta.

by combinatorially multiplying the shares of two secrets and redistributing the results into d output shares.

6.7 Conclusion and Future Scope

In this Chapter, we realized an HHE enabling SE scheme, Pasta, in hardware. This is the first such realization of a modern, efficient SE scheme defined over integers, and the results highlight both the potential and the limitations of current HHE designs: for example, we identify a performance bottleneck caused by the heavy reliance on the XOF and quantify its impact on area. This work also provides the first practical demonstration of power analysis attacks on integerbased HHE schemes, using Hera as a case study, and shows that full key recovery is achievable with very few traces, underscoring the importance of sidechannel resistance in real deployments.

To address this, we introduce a masking framework tailored to arithmetic ciphers, validate its effectiveness through leakage assessments, and demonstrate how it generalizes from Hera to other HHE tailored designs such as Pasta. Together, these contributions both promote the hardware realization of HHE schemes and expose open challenges, suggesting future directions such as implementing and comparing additional HHE enabling SE schemes on hardware, as well as systematically analyzing and optimizing the cost of countermeasures against sidechannel and fault attacks [Aik+26b] relative to those required for publickey encryption.

7

SASTA: Nonce Respecting DFA on HHE

Breu

While HHE reduces client-side overhead in FHE systems, its security under active attacks remains underexplored. Prior works have shown that HHE maintains functional correctness, but its resilience to fault attacks has not been fully investigated. In this chapter, we present SASTA-DFA, the first Differential Fault Analysis targeting the HHE protocol as a whole.

Unlike prior DFA attacks on symmetric encryption, SASTA-DFA does not rely on strong assumptions like nonce reuse, making it practical for real-world settings. Our study targets emerging integer-based SE schemes for HHE-Rubato, Hera, Pasta, and Masta and shows that they are vulnerable to our attack. The attack generalizes to Boolean-friendly SEs such as Rasta and even AES. We back our claims with the first experimental demonstration of DFAs for HHE schemes, using an ATxmega128D4-AU microcontroller. We further extend SASTA-DFA to authenticated transciphering protocols under weaker threat models with no functional dependency.

Contribution: This chapter draws on the following paper, led by the thesis author, who contributed to the conceptual development, performed the experiments, completed the implementations, and drafted significant portions of the text.

• *SASTA: Ambushing Hybrid Homomorphic Encryption Schemes with a Single Fault* by Aikata, Ahaan Dabholkar, Dhiman Saha, and Sujoy Sinha Roy, to appear in AsiaCCS 2026.

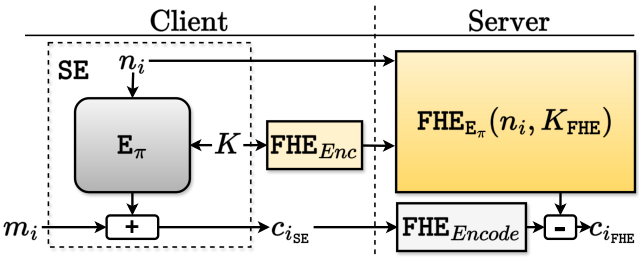


Figure 7.1: Workflow of HHE is illustrated here. The homomorphically encrypted key ($K_{\mathtt{FHE}}$) is communicated only once. After this, multiple small ciphertexts ($c_{i_{\mathtt{SE}}}$) encrypted using SE are sent along with nonce. The server performs a 'homomorphic SE decryption circuit evaluation' (HHD) and obtains a homomorphically encrypted ciphertext $c_{i_{\mathtt{FHE}}}$. Here $\mathtt{E}_\pi$ refers to any SE permutation (e.g., AES), and i refers to i-th iteration of the HHE protocol.

The main idea behind HHE is to replace the expensive homomorphic encryption methods with symmetric key encryption. In the HHE protocol, illustrated in Figure 2.4, clients encrypt their data with a symmetric key scheme (e.g., AES) before sending it to the server. The server then homomorphically evaluates the decryption circuit on the symmetric ciphertext to transform it into a homomorphic ciphertext (asymmetric). The obtained FHE ciphertext can then be used to perform the required computations on the server, for example, Neural Network inference. As a result, this method saves significant communication and computation requirements on the client. While a high-level overview is presented in Figure 2.4, the following Figure 7.1 gives a more detailed overview.

Over the years, researchers have realized that standard symmetric key schemes (e.g., AES) operating on boolean data have a huge performance overhead for server-side homomorphic decryption. As a result new HHE tailored Symmetric Encryption (SE) schemes – Pasta [Dob+23], Masta [Ha+20], Hera [Cho+21], and Rubato [Ha+22], have been proposed. Unlike standard schemes, these operate over integers in prime fields ($\mathbb{F}_p$) and have low multiplicative depth, enabling efficient homomorphic symmetric decryption.

Recent works [Aha+23; BBS21] have proposed *Authenticated Transciphering* (AT) to further enhance the capabilities of HHE by ensuring the integrity and authenticity of encrypted data. The motivation is to ensure that the client receives the result on the correct data and not manipulated data, as the ciphertexts can be maliciously altered in communication. They achieve this by using symmetric key schemes (e.g., AES-GCM) in Authenticated Encryption with Associated Data (AEAD) mode. Its ability to obtain a Tag helps the client to validate the data integrity.

7.0.1 Research Gap

Physical attacks on cryptographic implementations [BDL97; BS97; BU02; SMS23; TBP20; Wan+22] exploit the physical characteristics of the device running the cryptographic primitive to recover secret parameters involved in its computation. These attacks include measuring the output size [BU02], execution time [TBP20], and power consumption [Wan+22] or disturbing a cryptographic computation to recover sensitive data [BDL97; BS97; SMS23]. Physical attacks are often more efficient than mathematical cryptanalytic attacks and are considered a severe threat, especially in applications where attackers can be near the computing device. Despite its promise, the HHE protocol itself and the new SE schemes have not undergone extensive cryptanalysis in the context of physical attacks such as Differential Fault Analysis (DFA) [BS97].

To date, there have been few DFA studies [Rad+23; RBM21] on HHE tailored SE schemes [Can+18; Dob+18a; Méa+16; Méa+19]. These studies focus exclusively on the standalone SE primitives without leveraging the specific characteristics of the HHE protocol. Their attack approaches are similar to classical fault attacks on AES, for example [TMA11], and require strong assumptions like nonce reuse. However, this deviates from the scheme specifications and is prohibited by cryptographic protocols like SSL/TLS [Dob+16; GGF22]. This motivates us to investigate novel DFA methodologies tailored for HHE under realistic constraints. Next, prior DFA works target SE schemes operating on boolean data [Can+18; Dob+18a; Méa+16; Méa+19; MR24]. In contrast, emerging SE algorithms for HHE, such as Pasta [Dob+23], Masta [Ha+20], Hera [Cho+21], and Rubato [Ha+22] operate in prime fields. An analysis of these HHE-tailored SE schemes under DFA is unexplored yet much needed due to the vast potential for deployment of privacy-preserving applications [Niu+25]. Finally, note that both HHE and FHE protocols share the same FHE decryption routine and differ only in the encryption. It remains another open challenge to analyze how a fault attack on HHE encryption routine can be used to recover the secret key.

Thus, while prior works have shown that HHE *introduces minimal loss on the correctness of many applications, we shift the spotlight to a more critical question: Security under Differential Fault Analysis.*

7.0.2 The SASTA Threat Model

Our threat model follows the seminal work of Li and Micciancio [LM21], where the attacker tries to recover the secret key in the setting shown in Figure 7.2, termed as IND-CPA$^{\mathrm{D}}$. They show how this setting, where the attacker has access to the results of the restricted decryption oracle, is practically enabled by FHE. The FHE community widely acknowledged this work and actively added fixes/patches for the attack by changing the error distribution during decryption in various libraries like HElib [HS20] and PALISADE [PAL21]. Following this groundbreaking observation, in this Chapter, we aim to explore the same setting in the presence of a fault, as showcased in Figure 7.2. Note that since our attack

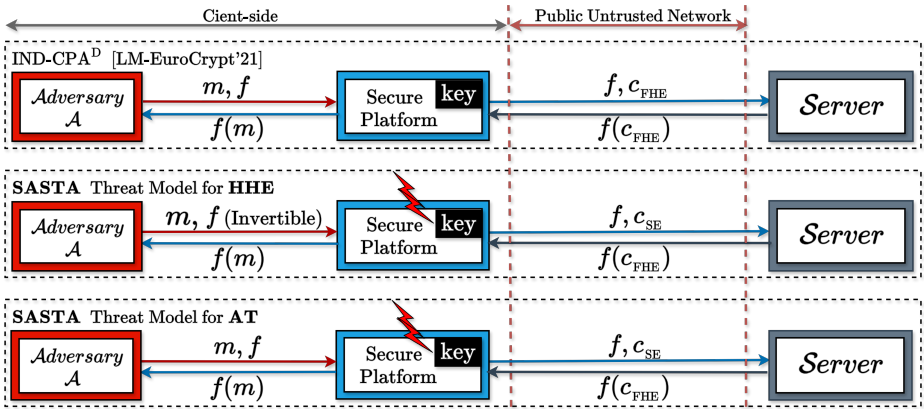


Figure 7.2: The IND-CPA^D threat model [LM21] and its adaptation to the SASTA-DFA model for HHE and AT, which extends it by incorporating fault injection capability.

is not dependent on the error distribution, it cannot be bypassed by the patches used for preventing the attack shown in [LM21]. Thus, our attack remains practical and realistic, and its prevention necessitates additional fault attack countermeasures. We propose a DFA technique that can be applied to both HHE and AT. For HHE, the function to be evaluated can be non-trivial but invertible. However, this assumption is relaxed when we apply SASTA-DFA on AT, and any arbitrary function can be queried, analogous to the original IND-CPA^D setting, as shown in Figure 7.2.

Threat Model in Practical Contexts. In practice, Li and Micciancio [LM21] state that the IND-CPA^D can be realized in systems where decryption results are either publicly accessible or selectively shared with specific parties, for example, in cloud computing systems. In such systems, Keys are securely managed by a hardware security module (HSM) or secure enclave, rendering them inaccessible to an attacker present on the client. A real-life example is MindNetworks' deployment [Min], where the FHE Decryption Network decrypts encrypted consensus results into plaintext for hubs that require decryption. Recent works [Niu+25] have reinforced that HHE and AT are emerging as a practical and efficient approach for FHE-based privacy-preserving applications. In this setting, the adversary can inject faults during local symmetric encryption and observe decrypted outputs of cloud-evaluated ciphertexts (e.g., after FHE processing) but cannot directly access encryption or decryption keys.

For instance, this applies to privacy-preserving data sharing and processing services used in Secure Cloud-based Medical Diagnosis [AI; Goo]. The hospital encrypts patient data or images locally using symmetric encryption and sends them to the cloud for privacy-preserving analysis or disease prediction. A malicious insider who gains access to the compromised device for a short window can see input data, inject faults, and observe decrypted outputs while keys

Table 7.1: Comparison of SASTA-DFA with prior works. The average time for key recovery on a 2GHz clock 4GB RAM processor is reported. † stands for results from [AM11b].

Target Scheme	Security	Field	Nonce Resp.	#Faults	Time	HHE Context
Flip_{530}	80-bit	$\mathbb{Z}_2$	No	1	39hr	✗
Kreyvium	128-bit	$\mathbb{Z}_2$		3	5min	✗
Filip_{430}	80-bit	$\mathbb{Z}_2$		1	751hr	✗
Rasta-6	80-bit	$\mathbb{Z}_2$		1	96hr	✗
AES-GCM	128-bit	$\mathbb{Z}_2$	**Yes**	1	5s	✓
AES-GCM	256-bit	$\mathbb{Z}_2$		1	45min†	✓
Rasta-5	128-bit	$\mathbb{Z}_2$		1	0.11s	✓
Rasta-6	128-bit	$\mathbb{Z}_2$		1	0.12s	✓
Pasta-3	128-bit	$\mathbb{F}_p$		1	0.50s	✓
Pasta-4	128-bit	$\mathbb{F}_p$		1	0.21s	✓
Masta-4	128-bit	$\mathbb{F}_p$		1	0.09s	✓
Masta-5	128-bit	$\mathbb{F}_p$		1	0.07s	✓
Hera-5	128-bit	$\mathbb{F}_p$		1	0.005s	✓
Rubato	128-bit	$\mathbb{F}_p$		2	0.15s	✓

remain protected by HSMs. Our attack shows that a single fault in just one query is sufficient to reveal the encryption key, exposing the entire medical record database of the hospital. Similar attacks also apply to Edge-to-Cloud Secure Video Analytics [Lag+21], where surveillance systems encrypt video frames locally before uploading them for cloud analytics. An attacker with access to the edge device may tamper with encryption and observe final outputs but cannot access keys stored in secure hardware. Our attack could reveal the video stream transmitted by the device to unauthorized parties and result in a complete privacy breach.

7.0.3 Contributions

We present SASTA-DFA, a novel DFA attack on HHE to recover private cryptographic keys of the client device. Our main contributions are three-fold.

① **First protocol-level nonce respecting attack.** We are the first to demonstrate how an attacker can exploit the specific properties of the HHE protocol to achieve complete key recovery with a single fault injection without depending on nonce reuse, thus aligning with the practical constraints in real-world scenarios.

The HHE protocol itself facilitates our efficient attack and is central to it. It helps make the attacks not only realistic but also easily applicable. We further extend our attack and, for the first time, show how Authenticated Transciphering (AT) [Aha+23] becomes vulnerable to DFA by leveraging the SASTA threat model. We present an end-to-end key-recovery firmware experiment to validate this.

② **First DFA of** SE **schemes over** $\mathbb{F}_p$**.** Analyzing the SE schemes defined over $\mathbb{F}_p$ under SASTA-DFA is challenging due to a lack of prior DFA literature on these schemes. Therefore, we develop the first key-recovery attack utilizing SASTA-DFA against emerging HHE tailored SE schemes- Pasta [Dob+23], Masta [Ha+20], Hera [Cho+21], Rubato [Ha+22]. This further extends to Rasta [Dob+18a] defined over $\mathbb{Z}_2$. The results are tabulated in Table 5.5. This also adds new results to the standard DFA literature (Table A.1 in Appendix). Our attack and Key-Recovery technique exploit SASTA-DFA but is not limited to this threat model and can also be utilized in traditional DFA settings.

③ **Attack Proof-of-Concept.** All previous fault analyses conducted on HHE-enabling schemes remain theoretical and lack practical implementations. We bridge this important gap and demonstrate our attack using an 8-bit off-the-shelf micro-controller ATXmega128D4-AU set as a target using the ChipWhispererLite CW1173 board. This attack requires just a *single known fault*, resulting in complete key-recovery. It incorporates an AES firmware. Additionally, we have integrated the firmware for the new HHE tailored SE scheme- Hera to validate SASTA-DFA, further amplifying the scope of our investigation. Our attack implementation and key recovery scripts are open source and available at https://github.com/aikata10/SASTA. We also comprehensively analyze potential protocol as well as implementation strategies to strengthen the security of the HHE protocol against SASTA-DFA.

- Connection between FHE and SASTA-DFA.

Our key observation highlights a significant vulnerability in the HHE protocol, which makes it susceptible to single-fault DFA even in a realistic *nonce-respecting* setting. FHE offers strong privacy guarantees, and since the HHE protocol is built on top of FHE, it should maintain these guarantees. However, SASTA-DFA results in full key recovery, thus leading to a loss of privacy offered by the underlying FHE scheme. This underscores the importance of robust protocol design using FHE and secure implementations to protect against physical attacks.

- Design Considerations.

SASTA-DFA bypasses Pastas truncation, Masta and Rastas final-add-key step, and Heras add-round-key, to recover the secret key. While Rubato combines add-round-key and truncation for added security, its insufficient truncation and reliance on noise make it vulnerable to our attack. Our findings suggest combining truncation with key-whitening and using multiple independent keys can strengthen security against single-fault attacks by design. This discussion is further detailed in Section 7.6.

Chapter Outline.

In Section 2.3, we briefly describe HHE and various schemes enabling HHE. We discuss the prior DFA techniques in Section 2.5.3. In this Chapter, we first

introduce the application AT specific to this chapter in Section 7.1. Next, we describe the threat model and show how a differential can be obtained for HHE and AT protocols in Section 7.2. Section 7.3 shows how the differentials with carefully selected fault injection points can result in key recovery for various schemes. In Section 7.4, we show the experimental evaluation of the attack on a microcontroller and how it results in complete key recovery in real time. The countermeasures for the attack are discussed in Section 7.5. Section 7.6 discusses the implication on design decisions, limitations and future scope of the work, and Section 7.8 concludes the work.

7.1 Authenticated Transciphering (AT)

Table 7.2: A simplified algorithmic description of the Authenticated Transciphering protocol [Aha+23; BBS21].

Client			**Server**
Initialization			
Generate K, $\{sK, pK\}$ $K_{FHE} = \mathtt{FHE}_{Enc}(K)_{pK}$ Send pK and K_{FHE}	$\rightarrow$	$\rightarrow$	Save pK and K_{FHE}
Encryption using SE			
$c_{i_{SE}} = m_i + \mathtt{E}_{\pi}(K, n_i)$ $\mathtt{T}_i = GCM(c_{i_{SE}}, K)$ Send $c_{i_{SE}}$, n_i, and $f()$	$\rightarrow$	$\rightarrow$	$c_{i_{SE}}$, n_i, and $f()$
Homomorphic SE *Decryption Circuit Evaluation*			
			$c_{i_{FHE}} = c_{i_{SE}} - \mathtt{FHE}_{E_{\pi}}(n_i, K_{FHE})$ $\mathtt{T}_{i_{FHE}} = GCM(c_{i_{SE}}, K_{FHE})$
Function Evaluation			
			Compute $d_{i_{FHE}} = f(c_{i_{FHE}})$
$d_{i_{FHE}}, \mathtt{T}_{i_{FHE}}$	$\leftarrow$	$\leftarrow$	Send $d_{i_{FHE}}, \mathtt{T}_{i_{FHE}}$
Result Decryption			
If $\mathtt{T}_i = \mathtt{FHE}_{Dec}(\mathtt{T}_{i_{FHE}})_{sK}$:	▶*Decryption Oracle*		
$f(m_i) = \mathtt{FHE}_{Dec}(d_{i_{FHE}})_{sK}$	▶*Decryption Oracle*		

In the homomorphic setting, AT [Aha+23; BBS21] is used to ensure data integrity; that is, the data sent by the client is the same as that received by the server for computation. We stress on the fact that conventional FHE schemes do not offer data integrity and so the use of AT offers an additional benefit over other HHE related advantages. In a symmetric setting, the client sends encrypted data and a tag to the server. The server generates a tag from the received data and matches it with the received tag to verify data integrity. However, in the homomorphic setting, the server can only generate an *encrypted* tag using encrypted key ($K_{\mathtt{FHE}}$). As a result, it cannot do tag matching to verify integrity. Consequently, the server has to send the encrypted tag back to the client who has to verify whether the two tags match. This is briefly described in Table 7.2.

First, AT was analyzed by the authors in [BBS21]. They utilized Grain128-AEAD [Hel+21], and this was converted to a TFHE [Chi+20] ciphertext on the server side. This is followed by a recent work [Aha+23], where the authors report benchmarks for authenticated transciphering using standard symmetric encryption schemes such as AES-GCM and ASCON for CKKS [Che+17] FHE scheme. The results of this study show that due to the possibility of parallel processing, AES-GCM is much faster than ASCON, which can only process plaintexts sequentially. However, even at its best AES-GCM consumes a total depth of 123, requiring many bootstrapping operations just for HHD. In summary, AT aims to generate an additional authentication tag to ensure data integrity and authenticity.

7.2 The SASTA-DFA Fault Attack Strategy

We adapt the standard DFA framework widely used in prior works [BS97; LYK21; Muk09; RBM21; TMA11] to what we term as SASTA in the context of HHE. While these prior DFA works have made valuable contributions under well-defined models, they assume an adversary capable of replaying the nonce a very strong assumption that, while analytically useful, may not reflect practical deployment settings. Thus, we investigate whether such assumptions can be eliminated by leveraging the structure of the HHE protocol. We show that without relying on nonce reuse, the protocol's natural behaviour can be exploited to mount a successful attack under a much weaker and practical threat model- SASTA (detailed in Section 7.0.2). Finally, we restrict the attacker to only observe a single query-response pair to mount a successful attack, which is not yet feasible using other nonce-respecting fault attacks (e.g., SIFA, FTA). These relaxed assumptions make our threat model significantly more practical and weaker. Importantly, our work is the first to explore fault attack surfaces at the HHE *protocol* level of real-life FHE deployment.

7.2.1 Attack Methodology

The attack proceeds in three broad phases.

❶ **Faulting client's** SE **Encryption.** A single fault is induced by the attacker during the execution of SE encryption routine on the client (Figure 7.3). This fault results in the generation of erroneous ciphertexts, which are sent to the server. Since not every fault can be exploited, the attacker identifies certain Fault Injection Points (FIPs) depending on the SE primitive. These FIPs are targeted, and the resulting faults are subsequently leveraged for key-recovery.

❷ **Generating Differentials.** First, the client sends a faulty message to the server, which evaluates the homomorphic decryption circuit (HHD) on it. The server then sends the function output corresponding to the message back to the decryption oracle, and this output is different from the output corresponding to a fault-free message, thus forming a differential (Table 7.3 ⓑ). This process

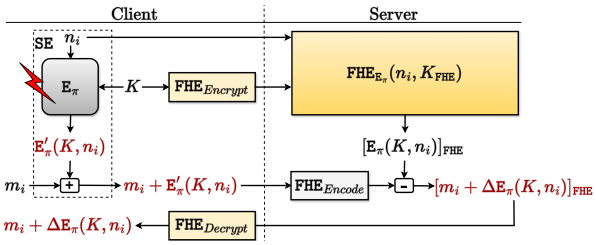


Figure 7.3: Utilizing SASTA-DFA for differential fault analysis of HHE tailored SE schemes.

is further elaborated in Section 7.2.3. The attacker observes this *differential* obtained in the output returned by the decryption oracle.

❸ **Key-Recovery.** The attacker uses the knowledge of the FIP and the generated differentials to extract the private HHE encryption key K that is stored on the client device. The private key can then be used to compromise the privacy of users on the attacked device. We detail the key-recovery methods in Section 7.3.

In the case of AT, the fault is induced during the very first round function ($\mathtt{E}_\pi$) evaluation (Figure 7.5), which is needed only for Tag computation. Thus, the differential is obtained using the valid tag sent by the decryption oracle, which is sent to it by the server in homomorphically encrypted form. This differential is subsequently employed for the purpose of key recovery *without any reliance on the function evaluated*.

7.2.2 How is SASTA-DFA different from standard DFA?

To create an exploitable differential in a standard DFA setting, an attacker, according to prior works, needs to query the client device twice with the same plaintext while injecting faults during one of the executions. However, this requirement alone is insufficient as a nonce is typically used to ensure that each encryption operation is unique. Thus, to utilize standard DFA, attackers in prior works need to rely on strong nonce-reuse assumptions for key recovery, as shown in Figure 7.4 (a). This is illustrated in the following example.

Consider a client-server interaction as shown in Figure 7.4 (b). In the symmetric setting, the client encrypts and sends the ciphertext to the server, and the server decrypts it and performs the evaluation requested by the client. The result is then re-encrypted and sent back to the client. Since the nonce (n_i) used for the client's encryption differs from that used for the server's re-encryption (n_g), a fault in the initial client-side encryption (c'_i) for standard DFA will not be exploitable to obtain a differential post re-encryption (d_g). The same applies to a data fetch operation, where the server does not perform evaluation but still performs re-encryption as per the protocol.

Due to these constraints in practical applications, all prior works [Rad+23;

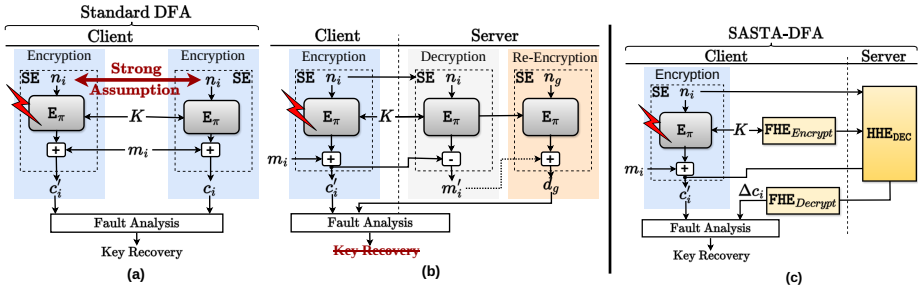


Figure 7.4: The diagram illustrates the difference between the standard DFA model (a) and the proposed SASTA-DFA model (c), showcasing how the classical setting relies on nonce-reuse, unlike SASTA-DFA. It also presents a case for standard DFA with server interaction (b), indicating how key recovery is infeasible as per specification, $n_i \neq n_g$.

RBM21] treat HHE-enabling schemes as standalone symmetric key schemes and assume nonce-reuse to enable DFA. However, nonce-reuse is not permitted in real-world protocols (e.g. SSL/TLS) as it is against the specification (and can lead to replay attacks [GGF22]). Furthermore, as using a nonce is a part of the scheme description, assuming its reuse could also lead to a breakdown of their theoretical security guarantees. Hence, this represents a strong assumption in the standard DFA threat model.

Key Insight. Our key observation is that, by considering the HHE protocol as a whole, we can relax the strong nonce-reuse assumption. As depicted in Figure 7.4 (c), the clients send the encrypted data to the server, and the server performs an SE decryption homomorphically. This ciphertext is evaluated and sent to the decryption oracle as part of the FHE protocol. This specific feature enables our attack and overcomes the limitations of prior works using standard DFA. Even if a re-encryption is done (by adding a new $\mathtt{E}_\pi$), the differential stays preserved and does not change post FHE decryption. The re-encryption will only result in the decryption oracle computing additional symmetric decryption to remove the added new $\mathtt{E}_\pi$. Thus, SASTA enables an attacker to perform key recovery without replaying plaintext or any other public parameter, including the nonce.

7.2.3 Attack on HHE

In HHE, the client device simply performs SE encryption (Table 2.2) to protect the message sent to the server. The encryption involves computing key-dependent permutation ($\mathtt{E}_\pi$) and then adding it to the message, as shown in Figure 7.1. For schemes defined over $\mathbb{Z}_2$, instead of modular addition/subtraction, XOR is used.

The attacker induces a fault in the SE encryption routine (Figure 7.3) and uses the output returned by the decryption oracle to generate a differential for key-recovery, as detailed in Table 7.3. The variable i refers to the i-th iteration of

Table 7.3: Faulty computation flow during the HHE protocol. For brevity, we omit the Initialization and Function Evaluation parts of the protocol.

Client		Server
Initialization		
Encryption using SE		
$c'_{i_{SE}} = m_i + \mathtt{E}'_\pi(K, n_i)$	ⓐ	
Send $c'_{i_{SE}}$, n_i, and $f()$	$\rightarrow$	$\rightarrow$ $c'_{i_{SE}}$, n_i, and $f()$
Homomorphic SE *Decryption Circuit Evaluation*		
		$c'_{i_{FHE}} = c'_{i_{SE}} - \mathtt{FHE}_{E_\pi}(n_i, K_{FHE})$ $= m_{i_{FHE}} + \Delta\mathtt{E}_\pi$ ⓑ where, $\Delta\mathtt{E}_\pi = \mathtt{E}'_\pi(K, n_i) - \mathtt{E}_\pi(K, n_i)$
Function Evaluation		
Result Decryption		
$f(m'_i) = \mathtt{FHE}_{Dec}(d'_{i_{FHE}})_{sK} = f(\Delta\mathtt{E}_\pi + m_i)$ ⓒ		

the HHE protocol, and j is used to index message blocks (for schemes over $\mathbb{F}_p$ or bits for schemes over $\mathbb{Z}_2$). The subscripts *SE* and *FHE* are removed for simplicity. If a fault occurs during the round function computation ($\mathtt{E}_\pi$) in encryption, the resulting ciphertext c'_i with t coefficients can be expressed as follows.

$$c'^j_i = m^j_i + \mathtt{E}'_\pi(K, n_i)^j \pmod p \quad \forall\ 0 \leq j < t$$

The faulty ciphertext c'^j_i is then sent to the server. The server-side HHD results in a faulty message m'_i, calculated as follows.

$$\begin{aligned} m'^j &= m^j + \mathtt{E}'_\pi(K, n_i)^j - \mathtt{E}_\pi(K, n_i)^j \pmod p \ \forall\ 0 \leq j < t \\ \Delta\mathtt{E}_\pi &= \mathtt{E}'_\pi(K, n_i) - \mathtt{E}_\pi(K, n_i) \pmod p \end{aligned}$$

Since the original message (m_i) is known, and the attacker can choose any invertible function $f()$, the differential ($\Delta\mathtt{E}_\pi$) can be obtained by simply subtracting the known message m_i from m'_i, as shown in Table 7.3 ⓒ. Thus, when a fault occurs in the round function computation ($\mathtt{E}_\pi$) during encryption, then due to server-side HHD, the fault-induced differential can be obtained *without* nonce-reuse. It can then be utilized for key-recovery, as detailed in Section 7.3.

7.2.4 Extending SASTA-DFA to AT

As described in Section 7.1, AT is used to ensure data integrity in HHE. This is important to guarantee that the result obtained by the client is for the intended data and not faulty data. In [Aha+23; BBS21], the authors propose AT using schemes on boolean data (e.g., Grain128, AES); however, these can also be replaced by the new schemes over finite fields for efficient computation. Therefore, we keep the discussion in this section generic to any scheme utilizing the GCM-like mode for authenticated encryption and data integrity.

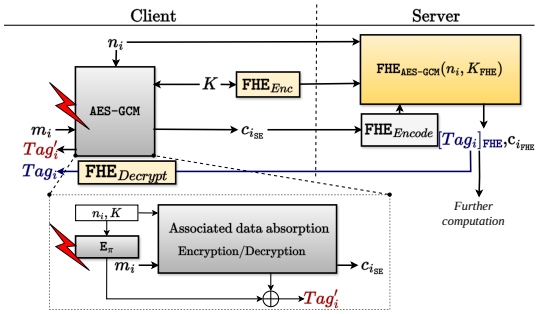


Figure 7.5: Exploiting SASTA-DFA for Authenticated Transciphering using AES-GCM.

Table 7.4: Faulty computation flow during Authenticated Transciphering.

Client	**Server**
Initialization	
Encryption using SE	
$ADE_i = GCM_{partial}(\mathtt{E}_\pi, m_i + \mathtt{E}_\pi(K, n_i), K)$ $\mathtt{Tag}_i' = \mathtt{E}_\pi'(K, n_i) \oplus ADE_i$ ⓘ Send $c_{i_{SE}}$, n_i, and $f()$ $\rightarrow$	$\rightarrow c_{i_{SE}}$, n_i, and $f()$
Homomorphic SE *Decryption Circuit and Function Evaluation*	
Result Decryption	
If $\mathtt{Tag}_i' == \mathtt{FHE}_{Dec}(\mathtt{Tag}_{i_{FHE}})_{sK}$: ⓘⓘ $f(m_i) = \mathtt{FHE}_{Dec}(d_{i_{FHE}})_{sK}$	

We propose mounting SASTA-DFA during the very first round function call ($\mathtt{E}_\pi$) of any new encryption since it is only used for tag generation and does not affect the subsequent ciphertext generation (as shown in Figure 7.5). A stepwise description is provided in Table 7.4. This is useful as a lack of dependence on ciphertexts frees us from relying on the knowledge of $f()$, and the attack can work for any arbitrary function or application, for example, private ML inference/training.

Note that our attack does not alter or influence the associated data absorption and encryption part, shown as a grey box in Figure 7.5. Hence, the output (ADE_i-GHASH result) of this computation stays the same in both the client and server-side computations, and the attacker only needs to know this output.

As mentioned in the previous section, for the scheme over boolean data ($\mathbb{Z}_2$), the modular addition/subtraction shown in the equations is replaced with XOR. The faulty tag computation can be expressed as follows (Table 7.4 ⓘ).

$$\mathtt{Tag}_i' = ADE_i + \mathtt{E}_\pi'(K, n_i) \pmod p$$

This is followed by the encrypted (fault-free) tag computation on the server side,

as shown below.

$$\mathtt{Tag}_i = ADE_i + \mathtt{E}_\pi(K, n_i) \pmod p$$

Since the tag is homomorphically encrypted, the server sends it to the client for verification, and the client obtains the original tag from the decryption oracle. Hence, the attacker obtains a differential ($\Delta\mathtt{Tag}$) without requiring nonce-repetition, thus conforming to our threat model (shown in Table 7.4 (ii)).

$$\begin{aligned}\Delta\mathtt{Tag}_i &= \mathtt{Tag}'_i - \mathtt{Tag}_i \pmod p \\ &= \mathtt{E}'_\pi(K, n_i) - \mathtt{E}_\pi(K, n_i) = \Delta\mathtt{E}_\pi \pmod p\end{aligned}$$

The obtained differential can subsequently be used for key-recovery. Observe that the differential obtained here is the same as that for plain HHE. Therefore, the same key-recovery techniques can be utilized.

7.3 Leveraging SASTA-DFA: Key-Recovery Case Studies

In this section, we present an end-to-end key-recovery attack on a variety of ciphers with unique constructions to show the impact of SASTA-DFA. We provide detailed case studies for each of the targeted ciphers- AES, Pasta, Masta, Hera, Rubato, and Rasta. We identify critical Fault Injection Points (FIPs) which help exploit the differential (Section 7.2.3 or Section 7.2.4) for key-recovery.

7.3.1 Cracking Pasta's Defenses

The Pasta-4 round function is shown in Figure 7.6 (detailed in Section 2.3.1). It consists of four rounds each comprising an Affine Layer $\mathtt{A}()$, a Mixing layer $\mathtt{Mix}()$, and S-box $\mathtt{S}'/\mathtt{S}()$. The state is treated as two concatenated vectors ($X_i||X_j$) and is initialized with the secret key. Affine transform performs matrix-vector (state) multiplication and round constant addition to the state. The Mix layer is used to combine the results of the two partial vectors in the state. All of these transforms are completely reversible and can be traced back to the initial state (secret key). Hence, to prevent trivial round-inversion, at the very end, a truncation is done, and only one of the vectors in the state is given as the final output ($KS = \mathtt{E}_\pi(K, n_i)$).

Several candidate FIPs exist for inducing a fault such that it diffuses widely throughout the Pasta-4 encryption (Figure 2.5). For instance, introducing a fault before the final $\mathtt{S}$ operation will result in its diffusion across the entire state due to the subsequent matrix multiplications ($\mathtt{A}_{8/9}$) and $\mathtt{Mix}$ steps. However, post-experimentation, we concluded that this differential alone would not lead to a unique key-recovery, as multiple keys could fulfil the differential condition. This would necessitate multiple such faults to eliminate potential key guesses. Therefore, we seek alternate fault injection points.

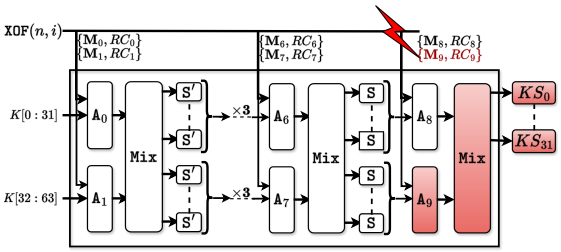


Figure 7.6: The attack strategy and FIP for PASTA-4 encryption ($KS = \mathtt{E}_\pi(K, n_i)$).

We induce a fault in the matrix to be multiplied during step $\mathtt{A}_9$, as depicted in Figure 7.6. If we denote the input state after the last $\mathtt{S}$ operation as $X_8||X_9$, the fault-free result is as follows.

$$\mathtt{E}_\pi(K, n_i) = \mathtt{Trunc}(\mathtt{Mix}(\mathtt{A}_8(X_8), \mathtt{A}_9(X_9))) \tag{7.1}$$

$$\mathtt{E}_\pi(K, n_i) = 2 \cdot (\mathbf{M}_8 \cdot X_8 + RC_8) + (\mathbf{M}_9 \cdot X_9 + RC_9) \tag{7.2}$$

The $\mathtt{Trunc}$ operation prevents easy round inversion for key-recovery. After fault injection, we obtain the following.

$$\mathtt{E}'_\pi(K, n_i) = 2 \cdot (\mathbf{M}_8 \cdot X_8 + RC_8) + (\mathbf{M}'_9 \cdot X_9 + RC_9) \tag{7.3}$$

$$\Delta\mathtt{E}_\pi = \Delta\mathbf{M}_9 \cdot X_9 \tag{7.4}$$

$$X_9 = \Delta\mathbf{M}_9^{-1} \cdot \Delta\mathtt{E}_\pi \tag{7.5}$$

If $\Delta\mathbf{M}_9$ is known and invertible, X_9 and consequently X_8 can be obtained via Equation 7.5 and Equation 7.2, respectively. Uniquely recovering $X_8||X_9$ implies full key-recovery of key K. This is because the permutation consists of matrix multiplications with invertible matrices, and the remaining linear operations are inherently invertible. Therefore, a one-to-one mapping exists between the initial input state K, public variables n, i, and intermediate state $X_8||X_9$. $\Delta\mathbf{M}_9$ is the difference caused by fault injection in the matrix $\mathbf{M}_9$, and its value is known due to known-fault position (the location of the instruction-skipped or bit-flip). For ensuring invertibility with a high probability [KKW08], next we discuss how we choose appropriate fault injection points.

Locating the Achilles Heel (Interesting FIPs). Ensuring $\Delta\mathbf{M}_9$'s invertibility is crucial for determining the appropriate FIP. For this, we look for FIPs that lead to a full matrix diffusion. This is because several studies [DS01; KKW08; Rud06] indicate that the probability of a random n dimensional square matrices being invertible is close to one ($\prod_{k=1}^{n}(1 - p^{k-1-n}) < 1 - \frac{1}{p}$ where p is the modulus). If the fault induced is fairly random at FIP, with high probability, we can ensure $\Delta\mathbf{M}_9$ has no linear dependencies and is thus invertible as needed for SASTA-DFA.

To induce such a fault, we identify the KECCAK (discussed in Section A.3) *permutation* (utilized in $\mathtt{XOF}$) as a viable option. A fault in the $\mathtt{XOF}$ function, just before it generates $\mathbf{M}_9$, leads to a full matrix diffusion due to the diffusion property

of the KECCAK hash function. This exploits KECCAK's diffusion property against HHE. Observe that this does not require a specific fault, for example, skipping a very particular instruction. The sole requirement is knowing which fault is induced so the attacker can know the value of $\Delta\mathbf{M}_9$. A fault in Keccak that causes $\mathbf{M}_9$ to become faulty will also affect RC_9. Since the value of RC_9 is known, we omit it from our explanation for simplicity.

7.3.2 Beyond PASTA: Extending the Attack to MASTA

The MASTA [Ha+20] scheme design was introduced concurrently with PASTA and can be viewed as a direct adaptation of RASTA [Dob+18a], originally defined over $\mathbb{Z}_2$, to $\mathbb{F}_p$. As outlined in Section 2.3.3, MASTA incorporates two primary functions in its round function: the χ S-box and the affine layer $\mathtt{A}_i$ (illustrated in Figure 2.6). The matrix used for multiplication within $\mathtt{A}_i$ is generated in a manner similar to PASTA, ensuring it is invertible by design (Section 2.3.2). In the last round r, if the state with t elements after the last S-box is denoted as X_r, then the final result following the round function is expressed as follows.

$$\mathtt{E}_\pi(K, n_i) = \mathtt{A}_r(X_r) + K \pmod p \tag{7.6}$$

$$\mathtt{E}_\pi(K, n_i) = \mathbf{M}_r \cdot X_r + RC_r + K \pmod p \tag{7.7}$$

The fault is induced in the matrix $\mathbf{M}_r$ (Figure 2.6 Section 2.3.3), and the resulting differential is as presented below.

$$\Delta\mathtt{E}_\pi = \mathbf{M}_r \cdot X_r - \mathbf{M}'_r \cdot X_r \pmod p \tag{7.8}$$

$$\Delta\mathtt{E}_\pi = \Delta\mathbf{M}_r \cdot X_r \pmod p \tag{7.9}$$

Similar to the case of PASTA, the only unknowns in the above equations are X_r and K. Given that the difference matrix is known and invertible, Equation 7.9 can be used to retrieve X_r. No truncation is done here. Instead, the key is added in the end, as shown in Equation 7.7. This gives us an interesting opportunity to directly retrieve the key K using the previously obtained X_r. Hence, no round inversion is required, and because matrix $\mathbf{M}_r$ is invertible, a unique key is obtained.

Regarding constraints on $\Delta\mathbf{M}_r$, similar to the case for PASTA, we can assert that with a high probability (close to 1), $\Delta\mathbf{M}_r$ will be invertible. Hence, MASTA is susceptible to SASTA-DFA, with the key-recovery process significantly simplified due to lack of truncation and the last add-key operation. We also observe that in ciphers with a final add-key operation, the resulting equations are directly expressed in terms of the key. Therefore, in such schemes, the reliance on the invertibility of the matrix $\Delta\mathbf{M}_r$ is significantly reduced, as we discuss next.

Linearization Attack Complexity Analysis. Linearization is one of the most effective attacks on these schemes [Dob+23]. Therefore, it is valuable to examine how, in cases where the matrix $\Delta\mathbf{M}_r$ is not invertible, the available linear constraints contribute to reducing the attack complexity. For a symmetric

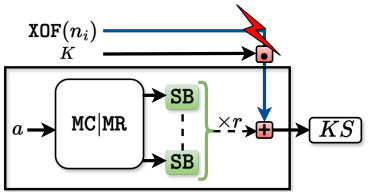


Figure 7.7: The fault injection points for inducing a fault in HERA or RUBATO permutation before final add-round-key.

primitive that admits a polynomial representation of degree d in n_v variables, the total number of possible monomials is given in [Dob+23] as follows:

$$b = \sum_{i=1}^{d} \binom{n_v + i - 1}{i} \tag{7.10}$$

The linearization attack operates with a time complexity of $\mathcal{O}(b^\omega)$, a memory complexity of $\mathcal{O}(b^2)$, and a data complexity of $\mathcal{O}(b)$. Here, ω is the linear algebra constant, and while typically $2 < \omega \leq 3$, the more conservative and realistic range is $2.37 \leq \omega \leq 2.82$ [Gra+23; Liu+21]. In order to attain 128-bit security, the parameters are chosen to ensure that $\log_2 b / \log_2 b^\omega \leq 64/128$ For MASTA-4/5, $n_v = 128/64$ and $d = 16/32$. In case $\Delta\mathbf{M}_r$ is not invertible and instead has a rank rk, then the number of variables in X_r decrease by rk and hence, using Equation 7.7 the total number of unknown key variables also reduce to $n_v - rk$, and the total monomials for linearization reduce as follows.

$$b_{\text{SASTA}} = \sum_{i=1}^{d} \binom{n_v - rk + i - 1}{i} \tag{7.11}$$

We evaluate that for $\Delta\mathbf{M}_r$ of rank $rk = n_v - 1$, the complexity of the linearization attack is at most 2^{11} for MASTA-4 and 2^{14} for MASTA-5 (Table 7.5), which is significantly below the 128-bit security threshold. Notably, the probability of obtaining a non-invertible differential matrix in the attack on MASTA is negligible, making this a worst-case analysis. Thus, in the worst-case scenario, an attacker could successfully perform a linearization attack for full key recovery. Finally, while our study focuses on 128-bit secure parameters, SASTA-DFA reduces the security margin across all security levels without loss of generality.

7.3.3 Extending the Attack to HERA

HERA [Cho+21], elaborated in Section 2.3.4, takes a slightly different design approach and uses a pseudo-key-schedule, where the `XOF` output vector (RV) is multiplied with the key and then added to the state (X). This is followed by the affine layer, which has constant matrices, unlike PASTA and MASTA. A fault in the last `XOF` call utilizing RV_r for state X_r (Figure 7.7) would result in the

Table 7.5: The Security bounds (parameters for 128-bit security) before and after SASTA-DFA considering the linearization attack. Considers the case where matrices are not invertible for MASTA, and RASTA, or the key-recovery is incomplete due to insufficient fault diffusion (HERA) or truncation (RUBATO).

Scheme	n_v	d	$\log_2 b$	$\log_2 b^{\omega}$		$n_v - rk$	$\log_2 b$ SASTA	$\log_2 b^{\omega}_{\text{SASTA}}$	
				$\omega =$ 2.37	$\omega =$ 2.82			$\omega =$ 2.37	$\omega =$ 2.82
MASTA-4	128	16	69.2	164.1	195.2	1	4	**9.5**	11.3
MASTA-5	64	32	84.6	200.5	238.6	1	5	**11.9**	14.1
HERA-5	16	243	83.3	197.5	235.0	1	7.9	**18.8**	22.3
						4	27.2	64.4	76.1
RUBATO-3†	36	8	29.4	69.8	83.0	4	8.9	**21.2**	25.2
						9	14.6	34.5	41.1
RUBATO-5†	16	32	44.4	105.3	125.3	4	15.8	**37.5**	44.7
						6	21.4	50.7	60.3.4
RASTA-5	525	32	172.9	409.8	487.6	1	5.0	**11.9**	14.0
						4	15.8	37.5	44.7
RASTA-6	351	64	253.1	600.1	714.1	1	6.0	**14.2**	16.8
						4	19.6	46.5	55.4

† The results for RUBATO include the αp factor due to the Gaussian noise.

following equations.

$$\mathtt{E}_{\pi}(K, n_i) = X_r + RV_r \cdot K \pmod{p} \tag{7.12}$$

$$\Delta\mathtt{E}_{\pi} = RV'_r \cdot K - RV_r \cdot K \pmod{p} \quad = \Delta RV_r \cdot K \pmod{p} \tag{7.13}$$

In the case of HERA, the known difference is simply a vector ΔRV_r, hence recovering key K, using Equation 7.13, only involves field inversions. Thus, the invertibility constraint is relaxed in this setting, and similar to MASTA, no round inversion is required, further simplifying the key-recovery. Furthermore, in the case of HERA, the `XOF` output vector (RV) is directly utilized instead of being employed to generate a matrix. Consequently, diffusion is solely dependent on the `XOF`. Hence, we rely on attacks that skip the first few values produced by the `XOF` call to populate the vector. This results in a shifted vector that offers adequate diffusion. It can be noted that this FIP is a small subset of possible FIPs for PASTA and MASTA. Section A.4 explains how the larger set of FIPs used in previous cases can also be utilized for HERA.

These FIPs impose additional constraints on the attacker's capabilities. To analyze this, we focus on faults injected into the `XOF` call responsible for generating the final output, RV_r. Our analysis suggests that while a perfect fault diffusion scenario is achievable in the best case, the majority of outcomes result in $rk \in [1, 4]$ values that are not effected, as they were in a previous non-exclusive `XOF` call. In this case, $n_v - r_k$ key values can be derived through simple field diversions. For the remaining values, a linearization attack can be applied. The parameters $n_v = 16$ and $d = 243$ for HERA meet the 128-bit security requirements. However, as shown in Table 7.5, when additional equations are obtained from SASTA-DFA (Equation 7.3.2), the linearization attack complexity reduces to 2^{19}. This reduction indicates that the attacker's capabilities need not be further enhanced

for HERA, as the remaining key space can be recovered using linearization. This attack is particularly effective when the modulus is large, as brute-forcing the remaining keyspace would result in significantly higher complexity.

7.3.4 The Curious Case of RUBATO

The design of RUBATO [Ha+22] (Section 2.3.5) is heavily influenced by HERA and PASTA. Yet, RUBATO cannot support schemes over integers $\mathbb{F}_p$ like BGV and BFV that are supported by PASTA, MASTA, and HERA. It can only support a scheme that does approximate arithmetic- CKKS. The reason being use of additive noise (e) from Gaussian Distribution $D_{\alpha p}$ ($\alpha p = 10.5, 4.1$ for RUBATO-5,3) at the end of the HHE routine, which modifies the Equation 7.12 as follows.

$$\mathtt{E}_\pi(K, n_i) = X_r + RV_r \cdot K + e \pmod p \tag{7.14}$$

After fault injection, the differential is expressed as follows.

$$\Delta\mathtt{E}_\pi = \Delta RV_r \cdot K + e \pmod p \tag{7.15}$$

The e introduced during encryption is not removed during decryption and remains in the data throughout the homomorphic operations. This inherent noise leads to a precision loss, rendering the data unsuitable for precise integer arithmetic and confining its utility to CKKS [Che+17]. On the other hand, the authors claim that this error enhances the security guarantees, which is why they opt for a low multiplicative depth of two per round and fewer rounds in the scheme. The number of monomials for linearization in the case of RUBATO is stated in [Ha+22] as follows.

$$b = \left[\sum_{i=1}^{d} \binom{n_v + i - 1}{i}\right] \cdot \alpha p$$

The authors in [Gra+23] highlight that the linearization attack security of RUBATO is significantly lower than anticipated. This can be attributed to the scheme's heavy reliance on the added Gaussian noise- e to transform the ciphertext into an LWE-like ciphertext, claiming this as a justification for enhanced security. With SASTA-DFA, additional linear equations involving the key K and the noise e are obtained, as illustrated in Equation 7.15. These equations effectively reduce the influence of the added Gaussian noise, thereby decreasing the number of monomials as follows.

$$b_{\text{SASTA}} = \left[\sum_{i=1}^{d} \binom{n_v + i - 1}{i}\right]$$

Thus, the effect of added noise (e) is effectively nullified, and any security assumption solely based on this added noise no longer holds. With $\omega = 2.82$, the linearization attack complexity for RUBATO with 3/5 rounds is $2^{125}/2^{83}$, as shown in Table 7.5. However, after applying SASTA-DFA, this complexity reduces to $2^{115}/2^{77}$ (or $2^{98}/2^{65}$ for $\omega = 2.37$), while eliminating any potential reliance on

the added Gaussian noise for security. While this analysis demonstrates that the security assumptions are significantly weakened in the presence of SASTA-DFA, executing this linearization attack in real-time is challenging.

Therefore, for RUBATO specifically, we investigate whether utilizing two differentials instead of just one could enable full key recovery for a computationally constrained adversary. We do not impose any additional constraints or requirements on the two differentials. We obtain two sets of Equation 7.14 and Equation 7.15, where, as expected for two distinct encryptions, and everything except the key K changes. From the first differential, we derive a set of possible key values corresponding to the error range. Subsequently, we use the second differential to iterate over these sets and determine the intersection. Remarkably, with this second independent faulty query, we successfully recovered the correct key 99.95% of the time, as confirmed through testing over 100,000 cases. The truncation at the end is minimal, concealing only $rk = 4$ key elements. These remaining elements can be retrieved using trivial linearization with an attack complexity of $2^{21}/2^{38}$ (Table 7.5). Hence, we conclude that while RUBATO marginally increases the attack difficulty, it remains vulnerable under the proposed SASTA-DFA attack model.

7.3.5 Generalizing SASTA-DFA: SASTA on RASTA

Until now, we explored state-of-the-art HHE schemes over $\mathbb{F}_p$; however, we note that the SASTA-DFA technique is quite general and can also be applied to schemes over $\mathbb{Z}_2$- RASTA [Dob+18a], that utilize XOF in its construction. The only dependency is that there are much fewer invertible matrices in $\mathbb{Z}_2$ [KKW08] (≈ 0.29). Therefore, it is more probable that the differential might not result in unique key recovery, which would require an exhaustive search over the reduced key space.

In this scenario, for the linearization attack, we observe that the rank of the obtained matrix lies within the range $[n_v - 4, n_v]$, as the probability that the matrix has rank at least $n_v - 4$ is ≈ 0.94 [KKW08]. When the rank is n_v, full key recovery is achievable. However, we estimate the linearization attack complexity for a lower rank ($rk = 4$) as $2^{38}/2^{47}$ for RASTA-5/6 (Table 7.5). It is important to note that the key in this case is boolean, so an exhaustive search would have a complexity of 2^4 for the remaining key variables, making it a more efficient attack strategy for RASTA. We obtain the differential as follows.

$$\mathtt{E}_\pi(K, n_i) = \mathtt{A}_r(X_r) \oplus K \tag{7.16}$$

$$\mathtt{E}_\pi(K, n_i) = \mathbf{M}_r \cdot X_r \oplus RC_r \oplus K \tag{7.17}$$

$$\Delta\mathtt{E}_\pi = \mathbf{M}_r \cdot X_r \oplus \mathbf{M}'_r \cdot X_r \tag{7.18}$$

$$\Delta\mathtt{E}_\pi = \Delta\mathbf{M}_r \cdot X_r \tag{7.19}$$

As discussed in Section 2.3.3, RASTA has the similar design principles as MASTA [Ha+20]. The implementation of RASTA also uses KECCAK for XOF, thus making it the appropriate exploitable FIP. It results in the desired difference

required for SASTA-DFA and leads to a unique key-recovery. Similar to the case of PASTA, fault in Keccak will also affect RC_r. However, its value is known and hence omitted from our explanation.

7.3.6 Exploiting AES via Known Approaches

Since AES has been a standard for over two decades, numerous studies have analyzed it using DFA [AM11a; AM11b; Bar+10; GSR24; Hir+24; Muk09; SMC09; TMA11]. These works also include both single or multiple-fault-based key-recovery techniques [AM11a; TMA11], [GSR24; Hir+24; Muk09]. Most of these techniques do not even rely on known fault conditions. The FIPs employed in these studies can be utilized to realize SASTA-DFA. Consequently, SASTA-DFA can be employed to extend the same attacks in an HHE setting with weaker assumptions due to non-reliance on nonce-reuse. Note that some DFA methods requiring multiple fault injections also depend on both fault-free and faulty ciphertext pairs for the same nonce. However, with SASTA-DFA, the nonce-reuse assumption is relaxed, making such attacks feasible in practical scenarios.

Summary: SASTA-DFA establishes a universal methodology for DFA across diverse HHE-enabling SE schemes. We elaborated on how SASTA-DFA can be utilized to mount DFA, resulting in full key-recovery on the standard (AES-GCM) and new constructions (RASTA, PASTA, MASTA, HERA, and RUBATO). This includes both types of HHE-enabling schemes- defined over boolean fields (RASTA, and AES-GCM) as well as those over prime fields (PASTA, MASTA, HERA, and RUBATO). For all the above case studies, we only considered fault attack surfaces on client-side. However, for the sake of completeness, we also analyzed possible FIPs on the server-side computations detailed in Section A.5. We emphasize that attacking the client's device is a more realistic setting and is, therefore, the primary target of all prior works on HHE enabling SE schemes.

Next, we present a practical fault injection attack resulting in key recovery.

7.4 The Ambush: Fault Attack Demonstration

HHE employs symmetric key schemes for reduced computation and communication overhead, enabling it to be executed on resource-constrained embedded devices. We implement SASTA-DFA on one such constrained embedded device, specifically the ATXmega128D4-AU mounted on the ChipWhisperer-Lite CW1173 evaluation board[1]. It is an 8-bit Harvard architecture RISC single-chip off-the-shelf microcontroller, which runs lightweight schemes like the TINYAES. Similar to prior fault injection works [SMS23], we also utilize the opensource ChipWhisperer toolchain[2]. Given the limitations of this device, it is difficult to perform an entire conventional client-side FHE computation. However, the use of

[1] `https://rtfm.newae.com/Targets/CW303%20XMEGA/`
[2] `https://github.com/newaetech/chipwhisperer`

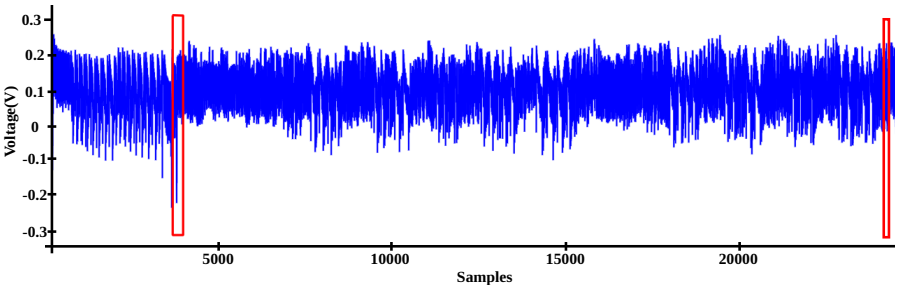


Figure 7.8: The sample fault injection attack positions targeted in our work. The first position is the beginning of KECCAK permutation, and the second is where the results of permutation are written to the final buffer.

HHE makes this easily possible[3]. To illustrate the potential of HHE in enabling FHE utilizable encryption on resource-constrained devices, we have successfully implemented and executed the HERA algorithm on the ATXmega128D4-AU, which, to the best of our knowledge, is the first microcontroller realization. This is essential for the development of secure and efficient encryption solutions for a wide range of applications. Next, we discuss how we induce known faults in KECCAK and attack the implementation of HERA.

7.4.1 Fault Injection Technique

The SASTA-DFA attack model does not require a specific fault, and the attack strategy solely relies on the ability to determine the faulty output. Hence, there are no constraints on the type of fault or the specific operation it affects. Several works in literature [BFP19; BH22; Dut+19; Men+20; SMS23] have validated its feasibility. It can be introduced at any point within the extended execution of the KECCAK permutation. Moreover, we can estimate the position of the fault within a few possible locations (discussed in Section 7.4.2). As a result, we can further relax the requirement for knowledge of the instruction affected by the fault.

To successfully demonstrate fault injection on the KECCAK permutation, we use the FIPS202 standard implementation [Mul24]. Our target platform is CWLITEXMEGA, and we employ clock-glitching to induce faults. The design runs as firmware at the default clock frequency of 7.38 MHz. The fault is mounted on the client's device with the target set to the KECCAK based `XOF`. The KECCAK function call has three sub-calls: absorb, permute, and squeeze. The absorb call places the input into the KECCAK state, the permute call runs the permutation, and the squeeze call returns the output.

Figure 7.8 illustrates the power trace and the two specific locations where we injected our attack. The first location corresponds to the start of the KECCAK Permute operation. Injecting a fault here resulted in skipping this particular

[3]The ChipWhisperer toolchain runs AES firmware and supports AES-GCM mode.

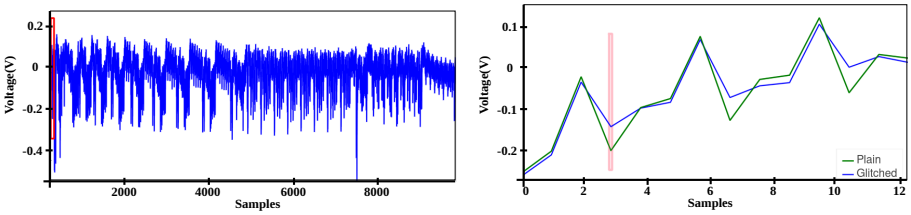


Figure 7.9: The first plot illustrates fault injection during the last HERA ARK operation. The second plot presents an enlarged view of the normal and faulty computations following the clock glitch. Note that these traces have been extracted from the original plot, and the x-axis is relative.

instruction, causing the output from the preceding permutation to be re-fed. The second known fault injection point is towards the latter stages of the KECCAK Permute operation. This led to a shifted version of the non-faulty KECCAK output, providing a known fault. For the latter case, we made minor changes to the FIPS202 implementation to simplify the attack. Since the input to the KECCAK is known and public, after the fault injection, the difference, Δ (in $\mathbf{M}_r$ or RV_r) required for key-recovery also becomes known.

These two known fault injection points serve to demonstrate the feasibility of known fault injections on microcontrollers. Similarly, voltage glitches [SMS23] can also be utilized to mount precise attacks during storage of KECCAK intermediate states. It is essential to note that precise faults, such as bit flips within the KECCAK state, can also be mounted on FPGA implementations.

7.4.2 Attack and Key-recovery

To showcase a fault injection resulting in full key recovery, we analyze the HERA firmware. As the available interesting FIPs for HERA are a subset of those required for RASTA, PASTA, MASTA, and RUBATO (as discussed in Section 7.3.3) an attack on HERA would also showcase practicality of SASTA-DFA on the other schemes. Figure 7.9 displays the power trace for the final Add-round-key operation. As we aim for the entire state to be faulty, we introduce a known fault at the beginning of the computation. This skips the initial few KECCAK squeeze instructions, causing the resulting XOF output to shift and provide the necessary fault diffusion. We then employ the strategy discussed in Section 7.3.3 for recovering the secret key of HERA encryption scheme.

We note that our attack does not necessitate power traces shown in Figure 7.9. These power traces are merely used to demonstrate the fault injection points and their effect on power consumption. Power analysis attacks are more complex and require specialized and precise equipment than fault attacks to remove the computation noise from the traces. As described in the threat model, our attack only requires an attacker capable of inducing faults during SE encryption.

7.4.3 Identifying successful fault injection

During our experiments, we identified multiple instances with a zero difference ($\Delta \mathtt{E}_\pi$), which could be the result of the glitch affecting non-critical portions of the code. This shows that not all fault injections yield faulty results and are thus unusable. When a usable fault is induced, our attack requires knowledge of the specific operation affected. Numerous works in the literature [BFP19; BH22; Dut+19; Men+20; SMS23] have shown this to be possible with high precision. We also demonstrated and thus validated this through our fault attack above.

However, in cases where attackers lack precise fault injection techniques, an alternative approach can be employed. They can induce an unknown fault and rely on the fault injection range within which the fault is triggered. Since the firmware is known, the attacker can simulate all the faults in an XOF emulation within the attack range (template), recovering potential faulty outputs and corresponding key guesses. Finally, they can uniquely recover the key with a single fault-free result via post-encryption equality check and elimination.

In conclusion, while the successful execution of the Hera algorithm on the ATXmega128D4-AU highlights the potential of HHE, the SASTA-DFA attack emphasizes the importance of re-evaluating its security guarantees against realistic attack settings. This is crucial to ensure its robustness and effectiveness in real-world scenarios.

7.4.4 Empirical results for key-recovery

This section details attack specifics and outcomes. First, we examine how the attacker determines the time offset for fault injection. Next, we explore the duration and range of the attack, resulting in unique key-recovery.

The only non-constant portion in the execution of all the HHE schemes is the rejection sampling, which for a known prime has a fixed average rejection rate. It enables the attacker to estimate a good time offset for a given modulus, and we reiterate that an attacker can induce the fault anytime in the entire Keccak permutation. Thus providing a broad attack window with a 92% success probability for faults resulting in an exploitable differential. Once a fault is induced, we build a template for the possible range of instructions that can be skipped and get a key guess for each. We determine the unique key via the known KS. For our experimental parameters, this template has eight possible instructions (hence, eight guesses at most).

Depending on the scheme and prime selection, the attack duration, during which faults can be induced, varies from 4% to 18% of the total execution time of HHE on the client side. An extension to earlier rounds would further increase this range. We show our attack on the last round. This implies countermeasures are only required to protect the last round from our attack. Extending this attack to earlier rounds implies more protection is needed, increasing the cost of countermeasures and, thereby, the severity of the attack. The high success probability of the fault, attributed to its non-specific instruction skip nature, makes it particularly effective when induced during the first Keccak permute

call in the XOF phase. An alternative approach is to target a specific instruction. Experimentally, we observe that in the worst case, one out of eight attempts leads to the right (or expected target) fault. The best case is one and an average of four. The average can be lowered to two using power traces, as the point when the specific XOF computation starts can be approximated more precisely.

Following the attack overview outlined in Section 7.3, our primary target is the Hera scheme, which exhibits unique key-recovery. This characteristic also applies to Rasta, Masta, and Pasta. As detailed in Table 7.1, we evaluate the 128-bit secure variants Pasta-3/4, Masta-4/5, Rasta-5/6, and Hera-5, achieving unique key-recovery (100% success probability once the single known fault has been realized). It only requires the encryption of one message block on the client side. Our key-recovery takes significantly less time than prior works, and our proposed attack model allows a weaker and more practical attacker setting. Furthermore, we have verified that our results remain unaffected by the end-to-end protocol, as our attack specifically utilizes the symmetric-key encryption and FHE decryption part of the HHE protocol.

7.5 Countermeasure analysis

Pre-generation and Storage. One potential countermeasure against SASTA-DFA involves the offline pre-generation and storage of XOF output. This technique works effectively for Rasta, Hera, and Rubato, as the matrices hold boolean data or the XOF output is a vector. However, this is challenging for Pasta and Masta due to the large amount of XOF output and matrix storage needed (the storage demand for one Pasta-4 permutation matrix is approximately 22KB). Given that this data changes with each iteration, generating and storing such substantial volumes of data can lead to significant storage expenses, especially from the client's perspective. Thus, more effective mitigation techniques are essential.

Redundancy and fault detection. In the long run, redundant computations would be a more suitable countermeasure. The same can be achieved by performing a decryption check on the encrypted data, requiring a complete duplicate computation for key-stream generation. Along with redundancy, we propose employing multiple checks at every stage of the last round for fault detection. Note that redundancy is not the ultimate solution as the attacker can bypass the checks, but they need many precise faults. This increases the required strength for mounting an effective fault attack. With a certain degree of redundancy, we can ensure the security of the design in a realistic scenario.

Infective Countermeasure. Since a stronger adversary can bypass fault detection, infective countermeasure [Gho+15] proves more reliable. Towards this, we employ an XOR-based detection mechanism. The last round is duplicated, and the two duplicate computations are XORed at every stage. After the Affine and Mix transform, this XORed result is ANDed with two random vectors, and the result is XORed back to the two duplicated states. Without fault, the XOR result

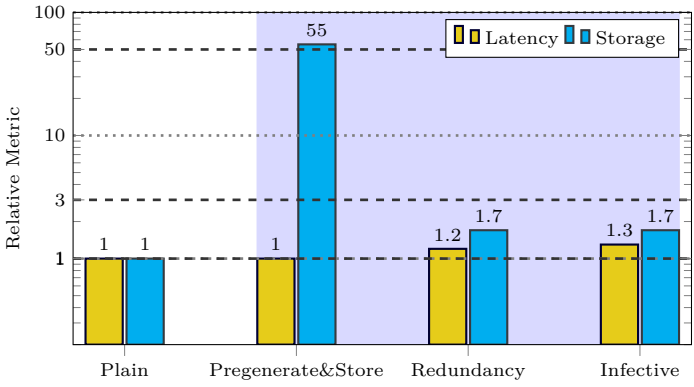


Figure 7.10: A comparison of latency and storage for different PASTA-4 implementations (with or without countermeasures). The three highlighted implementations are effective against SASTA-DFA. The graph is on a logarithmic scale

will always be zero. Hence, `AND` with the random vector will also be zero, and the last `XOR` would not affect the states. On the contrary, if there is a fault, the values `XOR`ed back to the state would introduce random unexploitable garbage.

Algorithmic Countermeasures. To protect against the attack on AT, it is best to ensure that the tag sent by the server cannot be utilized to form the differential required for SASTA-DFA (Section 7.2.4). One performance-hungry way to ensure this is by sending the client's tag also to the server, and it can run an encrypted equality check computation as discussed in [Aha+23]. This incurs a computation overhead on the server side. While the attacker can see the result, it is not exploitable (we need the faulty tag to obtain the differential), thus safeguarding against SASTA-DFA.

7.5.1 Countermeasure Performance Evaluation

We report performance evaluation results based on the reference implementation provided by the authors of PASTA [Dob+23]. The baseline implementation consumes 274,203 clock cycles on average for 100,000 executions of PASTA-$\mathtt{E}_\pi$. The implementation results are reported for the 12th Gen Intel i7 CPU. In Figure 7.10, we estimate the cost of countermeasures regarding latency and required extra storage. For storage requirements, we estimate the cost in terms of maximum state and intermediate variable storage required at any point in time during the PASTA-$\mathtt{E}_\pi$ execution. From Figure 7.10, it is evident that the Redundancy and Infective countermeasures offer the best latency vs memory trade-offs.

Do masked implementations resist SASTA-DFA? Masking is a technique

employed to protect against side-channel attacks, such as differential power analysis. This involves splitting the secret into multiple shares and processing each share separately, ensuring that these shares are never combined directly at any intermediate stage. In our case, the secret is the key (K); hence, all the computations involving the key would have to be masked. This involves only the key-dependent block. In contrast, the `XOF`, which is the attack target, is not key-dependent and hence will not be masked. Therefore, SASTA-DFA would not be affected by masking. Since no masking scheme is proposed for the new HHE enabling scheme, we only analyze the effect of a possible arithmetic masking.

7.6 Discussion

7.6.1 Impact on Design Decisions

Our attack demonstrates that we can bypass the countermeasures employed by various schemes to prevent inversion. For instance, Pasta relies on truncation; however, SASTA-DFA successfully circumvents this, allowing us to recover the original key. Similarly, Masta and Rasta depend on a final Key Add/Xor step for security. Due to this reliance, our attack directly extracts equations in terms of the key, significantly simplifying the attack and drastically reducing the complexity of the linearization attack (if needed)ultimately breaking the security guarantees of these schemes. Likewise, Hera employs a key-scheduling approach that yields a similar vulnerability.

In contrast, Rubato combines Hera's key-scheduling mechanism with truncation, which should, in principle, strengthen its security. However, the truncation is not leveraged to its full potential, as it conceals only the last four values regardless of the state size. Furthermore, the scheme primarily relies on added noise for security, which our attack effectively bypasses, leading to a full key recovery using linearization. However, the linearization attack complexity is still large, and we resort to utilizing two differentials to showcase key recovery in real-time for Rubato. This finding suggests that incorporating sufficient truncation and key-whitening techniques can enhance security, as more than a single fault injection would be required to recover the key. Finally, for schemes employing whitening-like techniques, using at least two independent keys may further hinder full key recovery. An interesting direction for future work is to assess the trade-off between the cost of implementing countermeasures against SASTA-DFA and the impact of modifying the scheme design.

7.6.2 Implication on HHE Privacy Guarantees

The HHE protocol is built on top of FHE and should theoretically maintain the same privacy guarantees. However, in a practical setting, SASTA-DFA demonstrates that full key recovery is possible within the HHE context, resulting in a loss of the privacy guarantees of the underlying FHE schemes as well. Outside the HHE context, specifically with standalone SE schemes, our attack reduces to

a standard DFA attack and provides key insights for improving the design of the SE schemes utilized for HHE. This Chapter is the first to show the vulnerability of HHE to DFA while respecting the nonce usage specified by the scheme and practical applications such as SSL/TLS networking protocols. This does not fundamentally compromise the security of FHE itself but highlights the necessity for robust protocol designs and secure implementations when using FHE to protect against physical attacks.

• **Extension of SASTA-DFA to PASTA$_{v2}$** [Gra+24]. In a more recent work [Gra+24], the authors aim to reduce the randomness required by RASTA-like designs, such as PASTA. To achieve this, they propose generating the affine layer of PASTA only once during the first round and reusing it in subsequent rounds. Since the FIP for PASTA in our attack leveraged the Keccak function used to generate the affine layer in the final round, this raises the question: *Does this modification make* PASTA$_{v2}$ *resilient against* SASTA-DFA*?* It is important to note that the applicability of SASTA-DFA in this context depends on implementation choices.

The MDS matrix within the affine layer occupies considerable space and is therefore often generated dynamically when required [ASR25b]. If this approach is followed in PASTA$_{v2}$, our attack can still be directly applied, where the fault is induced during the matrix generation step. In this case, the attack range is $\approx 6\%$ similar to the attack range for the Keccak function call, as per the results in [ASR25b]. Conversely, if a precomputed matrix is stored and reused, an instruction-skip attack could target the matrix read process (e.g., by skipping the read counter). However, this reduces the available attack range. Thus, while the reduction in randomness [Gra+24] complicates an attacker's efforts and puts the focus on implementation choice, it does not render such attacks impossible and highlights an interesting scope for future research.

7.7 Future Scope

• **Application and/or Extension.** We induce faults in the last round to showcase the utility of our approach. In the peer-reviewed [TCHES2025_2] follow-up works[4], the authors extend this to the last two rounds and also show how our SASTA-DFA model can be applied to the new HHE-enabling schemes (e.g., Elisabeth [Cos+22]) designed for other FHE schemes like TFHE [Chi+20], validating the general applicability of SASTA-DFA to the HHE setting. Thus, analyzing the standard DFA threat model and applying SASTA-DFA could be valuable, as it aids in removing the nonce-reuse assumption. Furthermore, while we have considered all the schemes over $\mathbb{F}_p$, it is worth exploring HHE schemes designed over $\mathbb{Z}_2$ other than RASTA and AES.

• **Threat Model.** Another area for improvement is the IND-CPA$^{\mathrm{D}}$ threat model [LM21], which allows the attacker to choose the evaluation done on her data. We completely remove this reliance for our attack on AT (Section 7.2.4).

[4]Not cited to maintain anonymity

Hence, SASTA-DFA on AT does not face function-related constraints and can be used for any privacy-preserving application of FHE, such as private machine learning inference or training. In this case, the attacker does not require prior knowledge of $f()$. However, it would be interesting to see if the same can be extended to the plain setting.

7.8 Conclusion

This Chapter introduced a novel fault attack technique called SASTA-DFA, designed for mounting DFA-based fault injection attacks on HHE, a protocol based on FHE. Traditional DFA attacks require at least two ciphertext computations with fixed nonce and plaintext assumptions. This is so that fault-free and faulty ciphertexts can be obtained under the same plaintext and public parameters to form an exploitable differential. However, SASTA-DFA eliminates the need for such assumptions in the context of HHE, thus laying the foundation of a realistic and practical attack model for the HHE protocol.

We validated this approach experimentally by executing the first fault injection attack on an off-the-shelf microcontroller running HHE enabling SE firmware for Hera. This proof-of-concept resulted in a unique and full key recovery with a single fault. We further showed how SASTA-DFA can be extended to AT technique built on FHE, eliminating the application dependency. We hope this will inspire the development of robust and secure FHE-based protocol designs in the future for various privacy-preserving applications.

8

Privacy at Your Fingertips: Fast Client-Side FHE

Breu

HHE mitigates slow encryption and significant ciphertext expansion issues but introduces new complexities and enables physical attacks under weak threat models. In this chapter, we propose Boosted-Deflation, a novel client-side optimization technique that offers an in-between solution, balancing the simplicity of classic FHE clients with the performance advantages of HHE.

Our approach leverages bootstrapping to reduce ciphertext expansion and communication overhead, thereby eliminating the need for encoding/decoding on the client and avoiding the floating-point operations commonly used in approximate FHE. The technique retains FHEs security and precision guarantees, while reducing client-side cost by up to 97%. We implement this method in a unified framework compatible with both software and hardware platforms.

Contribution: This chapter is based on the following paper, led by the thesis author, who developed the concept, performed the experiments and benchmarks, completed implementations (with FPGA results contributed by Florian), and drafted significant portions of the text.

• *Privacy at your Fingertips: Enabling Rapid Client-Side Operations in Fully Homomorphic Encryption* by Aikata, Florian Krieger, and Sujoy Sinha Roy, to appear in EuroS&P 2026.

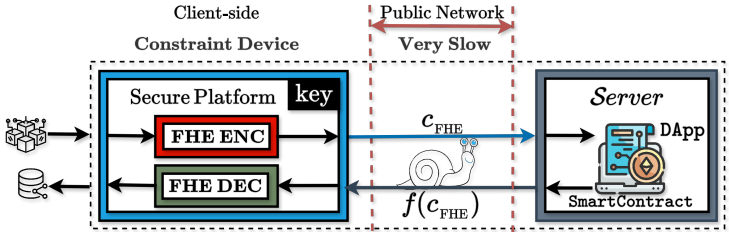


Figure 8.1: FHE-based Web3 Smart Contracts. The client sends encrypted data to the DApp. It remains secure during computation, and the result is returned to the client for decryption.

8.1 Introduction

The practical client devices for FHE are heavily resource-constrained, with examples including embedded systems in wearables (e.g., fitness trackers, medical sensors) and low-power IoT or edge devices (e.g., smart thermostats, home security cameras). Such devices have limited computational power, memory resources, and battery life, making heavy cryptographic computations such as FHE encryption particularly challenging. For example, the client-side FHE software implementation for the video surveillance system [UUS18] that sends data to the cloud for evaluation cannot encrypt or transfer even a single low-resolution video frame per second [Aza+23]. Another example is Satellite Collision Avoidance Systems using FHE [Lag+25], where constrained onboard devices must quickly homomorphically encrypt sensitive trajectory data before sending it to ground stations for secure assessment of collision risk. Minimizing encryption overhead is vital to maintaining the satellite's limited computational resources and communication bandwidth. Addressing these challenges by reducing FHE encryption overhead on constrained client devices thus becomes essential for real-world deployment of FHE's privacy-preserving capabilities across critical application domains.

There are two approaches for enabling FHE enc/decryption on a client device. The first approach follows the standard routines specified in FHE schemes. These routines require evaluating $2^{26}/2^{20}$ multiplications (Section A.9) and floating-point arithmetic support [Che+17], respectively. Additionally, the communication overhead for transmitting encrypted data from the client to the server is approximately 128×. A second approach has been proposed to mitigate this significant overhead, leveraging SE for Hybrid Homomorphic Encryption (HHE) [Cho+21; Dob+23; GHS12].

In this approach, the client encrypts plaintexts using a symmetric cipher in stream mode. The server then transforms the symmetric ciphertexts into homomorphic ciphertexts [Aha+23; BBS21] by homomorphically evaluating the symmetric decryption operation. Client-to-server communication decreases, but this advantage comes at the cost of increased computational overhead on the server, requiring approximately 123 bootstrapping operations [Aha+23], as

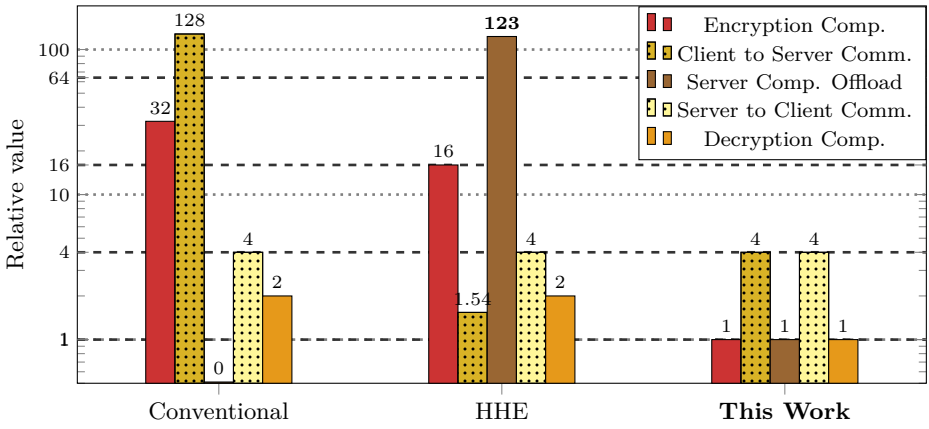


Figure 8.2: Illustration of how this work balances the two extremes explored in prior client-side performance studies. The y-axis is logarithmically scaled.

illustrated in Figure 8.2. Notably, the client must still perform homomorphic decryption. Additionally, the server-to-client communication bandwidth remains unchanged compared to the previous approach.

As shown in Figure 8.2, both techniques incur significant computation overhead on the client. Therefore, a few prior works [Aza+23; Kri+24; LDL23; MGS23; MÖS20] target high performance via compute acceleration. However, they do not consider the communication overhead. Overall, prior works either solely focused on compute acceleration or utilized SE to reduce communication overhead, introducing orders of magnitude higher computation overhead on the server.

Thus, this Chapter aims to address the trade-off between client/server computational cost and network communication overhead. Secure video conferencing is a representative use case motivating this study [Ben+25]. While end-to-end encryption is commonly employed, audio streams are often not fully private, and the computational complexity of audio mixing grows quadratically with the number of participants. To overcome this, Beni et al. [Ben+25] apply FHE to perform audio processing in the encrypted domain, reducing the mixing complexity to linear while preserving privacy. However, their approach incurs substantial communication overhead due to ciphertext expansion, which poses a significant barrier to practical deployment, given the maximum bandwidth limit of 1 Mbps.

HHE has the potential to alleviate the communication burden, but this comes at the cost of increased computational load on the server performing the audio mixing. In latency-sensitive scenarios such as real-time conferencing ($\approx$0.25s), this overhead is a huge problem, as even minor delays can result in disruptions and loss of synchronization. Therefore, a practical solution must minimize both the communication and server-side computational overheads to be viable in real-world settings.

8.1.1 Our Contributions

This Chapter presents Boosted-Deflation – a novel holistic approach for minimizing the practical problems of computation and communication overhead for client-side homomorphic en/decryption, as shown in Figure 8.2. Its emphasis on simplicity, by utilizing existing FHE routines, ensures that the proposed techniques are inherently secure. Our contributions are as follows.

❶ **Eliminated the need for en/decoding:** In a conventional setting, the plaintext is first encoded, then encrypted, and sent to the server. The resultant homomorphic ciphertext returned by the server post-computation is first decrypted and then decoded. A Discrete Fourier Transform (DFT) requiring double-precision floating-point arithmetic is utilized for en/decoding.

Our framework eliminates the need for expensive floating-point encoding and decoding on the client side. As a result, the client is relieved from performing any DFT operations before encryption or after decryption. This approach reduces the computation requirement by approximately half and saves hardware resources and energy that floating-point DFT would otherwise consume for CKKS. Our approach is particularly beneficial for microcontrollers that lack floating-point support on the chip (e.g., Mega32 [ATM24], Cortex-M4 [Kan+19]).

❷ **Reduced Computation Depth:** Ideal lattice-based homomorphic encryption schemes use a large ciphertext modulus, typically ranging from 438 to 1674 bits depending on parameters. Efficient implementations utilize the Residue Number System (RNS) to divide the large modulus into $(L+1)$ smaller prime moduli or shares, where $L > 0$. During the encryption, the client computes the ciphertext for all $(L+1)$ primes from the RNS base. For FHE, L could be 30 or larger [Aik+25; LDL23]. Our optimization technique leverages FHE's bootstrapping to enable encryption with just one small modulus and reduces the computation and ciphertext expansion by $\approx L+1\times$. Additionally, the proposed technique reduces the on-chip memory/area consumption, as well as communication overhead, by a factor of $L+1\times$.

❹ **Framework Implementation Results:** We provide a software implementation in C++ and prototype our technique on a low-cost Kintex-7 FPGA. The complete framework, both in software (C++) and hardware (bitfile), is open source and available at: https://github.com/aikata10/Client_Boost_Deflate. Compared to prior works, it achieves a $4.5\times$ performance increase and up to $49\times$ area reduction for $L = 30$. Additionally, our design synthesized on ASIC demonstrates a $76\times$ speedup compared to prior works while maintaining the same area-to-polynomial-size ratio. We also provide performance estimates using the Embedded SEAL library [ND21], which targets microcontrollers.

Organization.

In Section 2.2, we briefly describe FHE and the routines of an FHE scheme-CKKS. In Section 8.2, we present our proposed high-level technique for reducing

the computation and communication overhead on the client. Section 8.3 shows our detailed framework design across different platforms. In Section 8.4, we compare the experimental evaluation of our framework with prior works. Section 8.6 concludes the Chapter and also presents a discussion on interesting future scopes.

8.2 Proposed High-level Techniques

The client-side FHE routines encounter several challenges, regardless of the implementation platform. To address these challenges, we propose two main techniques: Deflate, and Boost in the context of CKKS as a case study. The Deflate technique specifically targets the computational overhead, while the Boost technique aims to minimize both computation and communication (CoCo) overhead. Before diving into the details of the proposed techniques, we first provide an assessment of the conventional CoCo overhead in the context of the RNS CKKS scheme [Che+17; Che+18b].

Algorithm 23 CKKS Encryption

In: $m \in \mathbb{R}, \mathtt{pk} \in \mathbb{R}_Q^2$
Out: $\mathtt{c} = \text{Encrypt}(m) \in \mathbb{R}_Q^2$

1: $v \xleftarrow{\$} \chi_{enc}$, $e_0, e_1 \xleftarrow{\$} \chi_{err}$
2: $\acute{m} \leftarrow \Delta \cdot \mathtt{IDFT}(m) + e_0$
3: **for** $(i = 0; i \leq L; i++)$ **do**
4: $\quad c_0^i \leftarrow (pk_0^i \cdot v_{\mathtt{NTT}} + \acute{m}_{\mathtt{NTT}})_{q_i}$
5: $\quad c_1^i \leftarrow (pk_1^i \cdot v_{\mathtt{NTT}} + e_{1_{\mathtt{NTT}}})_{q_i}$
6: **end for**
7: **return** $\mathtt{c} = \{\boldsymbol{c}_0, \boldsymbol{c}_1\} \in \mathbb{R}_Q^2$

Algorithm 24 CKKS Dec.

In: $\mathtt{c}, \mathtt{sk} \in \mathbb{R}_{q_0}^2$
Out: $m' = \text{Decrypt}(\mathtt{c})$

1: $c' \leftarrow \mathtt{INTT}(\langle \mathtt{c}, \mathtt{sk} \rangle)_{q_0}$
2: $m' \leftarrow \mathtt{DFT}(\Delta^{-1} \cdot c')$
3: **return** $m' \in \mathbb{R}$

8.2.1 CoCo Cost Assessment

During encryption, as shown in Algorithm 23, the plaintext is first transformed into polynomial form using the inverse DFT and then scaled by a scaling factor (Δ). Next, it is transformed to the NTT form and added to $(v \cdot \boldsymbol{pk_0} + e_0) \in \mathbb{R}_Q$ to generate $\boldsymbol{c_0}$. This operation is broken down into $L+1$ smaller parts in $\mathbb{R}_{q_i}$, utilizing RNS to enhance efficiency. A similar computation is done for $\boldsymbol{c_1}$ but without the message. Note that the encoded plaintext is processed separately for each modulus q_i (base-wise), with no computation dependency across different moduli. The public key is already precomputed and split across all the $L+1$ RNS bases during the key generation procedure.

At the end of the encryption process, the ciphertext $\mathtt{c}$ contains $L+1$ pairs of residue polynomials $\in \mathbb{R}_{q_i}$ (one set each for $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$). The ciphertext is then sent to the server. After performing the necessary computations, the server brings the computational depth of the ciphertext down to zero and sends the results back to the client. Finally, one polynomial multiplication ($\langle \mathtt{c}, \mathtt{sk} \rangle$) modulo q_0 is computed during decryption, as shown in Algorithm 24.

This simplified algorithmic flow is sufficient to analyze the communication overhead. To assess the computation cost, we perform a more detailed analysis and examine the number of NTT or DFT transforms required, as this is the most expensive operation with $\mathcal{O}(N \log N)$ complexity. Since NTT/DFT and INTT/IDFT are equivalent in computational costs, we will refer to both as NTT/DFT for simplicity of cost comparisons.

Communication Overhead Client → Server after Encryption

- The maximum number of elements packed in one ciphertext is $\frac{N}{2}$.
- Encryption of up to $\frac{N}{2}$ elements results in a ciphertext with $2(L+1)$ residue polynomials ($\in \mathbb{R}_{q_i}$) each of size N (total $2(L+1) \cdot N$). Hence, the communication overhead is at least $4(L+1)\times$ compared to symmetric key encryption ($\frac{N}{2}$).

Communication Overhead Server → Client before Decryption

- The maximum number of result elements packed in one ciphertext is $\frac{N}{2}$.
- For the resultant ciphertext with 2 residue polynomials each of size N, the communication overhead is at least $4\times$ compared to symmetric key encryption.

Computation Overhead Encryption

- One DFT transform (IDFT) is required to encode plaintext.
- The encoded message, after being added to a fresh error, needs to be converted to NTT for addition with the $v \cdot \boldsymbol{pk_0}$ result. This is for one ciphertext component (c_0^i) per RNS-base. Additionally, for $\boldsymbol{c}_1$, an error polynomial is converted to NTT $\bmod q_i$ and added to $v \cdot \boldsymbol{pk_1}$. Thus requiring $2(L+1)$ NTT transformations.
- The polynomial multiplications $v \cdot \boldsymbol{pk_0}$ and $v \cdot \boldsymbol{pk_1}$ are also done using NTT for fast computation. Hence, the v polynomial components must also be converted to NTT, and `pk` is precomputed and stored in NTT format. This requires one NTT transform for all $L+1$ residues of the v polynomial. Overall, as shown in Algorithm 23, $3(L+1)$ NTT and one DFT computations are performed during encryption.

Computation Overhead Decryption

- The ciphertext component $\boldsymbol{c}_1 \in \mathbb{R}_{q_0}$ is multiplied with the secret key and added to $\boldsymbol{c}_0 \in \mathbb{R}_{q_0}$. This result falls within the NTT domain; therefore, an INTT operation is performed over one modulus.
- One final DFT transform is required to obtain the decoded plaintext. Thus, overall, one NTT and one DFT are computed.

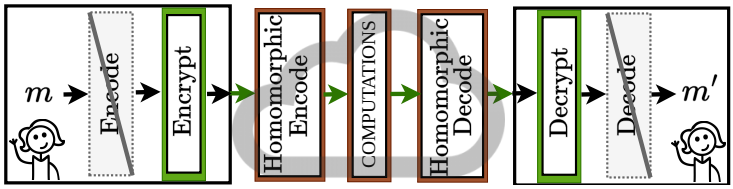


Figure 8.3: The simplified encryption process after applying the Deflate technique. The idea is to offload the encoding/decoding step to the server.

8.2.2 Deflate Technique

The key idea behind this technique is to leverage the fact that the DFT transformation matrix is publicly available and the server performs a similar transformation for bootstrapping (`CoeffToSlot`, `SlotToCoeff`). Therefore, we can offload the DFT transformation required for the encode-decode step to the server, as shown in Figure 8.3. This reduces the client's computational overhead and offers additional advantages, such as increased packing flexibility in implementation. By applying Deflate, the encryption and decryption routines can be rewritten as follows.

- $\texttt{CKKS.Enc}(m, \texttt{pk})$: It encrypts m , and returns $\texttt{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1) = v \cdot \texttt{pk} + (\Delta \cdot \cancel{\texttt{IDFT}}(m) + e_0, e_1) \in \mathbb{R}_Q^2$.
- $\texttt{CKKS.Dec}(\texttt{c}, \texttt{sk})$: It decrypts $\texttt{c}$ using the secret key $\texttt{sk}$ to return message $m' = \cancel{\texttt{DFT}}(\Delta^{-1} \cdot \langle \texttt{c}, \texttt{sk} \rangle)$.

Explanation. The plaintext is initially a vector of elements, also known as in the slot format. After encoding, it is transformed into a polynomial. The ciphertext is then converted back to vector form during the NTT step used for encryption. Therefore, the ciphertext contains the plaintext in slot form, not polynomial form. To address this, a homomorphic decode is performed in `SlotToCoeff` during bootstrapping so that the ciphertext contains the plaintext in polynomial form. After the remaining BS computations, this transformation is reversed using a homomorphic encode (`CoeffToSlot`).

The homomorphic en/decode steps involve homomorphic evaluation of the I/DFT transform. Thus, we propose offloading the pre-encryption and post-decryption en/decode steps to the server. The server will compute this transform homomorphically on the plaintext slots using matrix-vector multiplication, where the ciphertext acts as the vector, and the matrix contains roots of unity in plaintext form for the DFT transformation. The baby-step giant-step technique [HS18] (detailed in Section A.8) reduces the number of operations on ciphertexts and speeds up the computations.

It is important to note that this transformation on the server side consumes a computation depth of three (or four, depending on the parameters) for the matrix-vector multiplication in the DFT transform. Therefore, the ciphertext sent for computation must have sufficient depth for the homomorphic encoding

step when applying the Deflate technique. This is not a problem as the encrypted ciphertext supports the maximum possible depth L.

Impact on CoCo.

- *Decreased En/Decryption Computation overhead*: The computations during decryption are reduced by half, as only one NTT transform is needed. No DFT transformation is required during decryption or encryption.
- The Client $\leftrightarrow$ Server communication overhead is unaffected.

Other Notable Advantages. Apart from the CoCo advantages, this approach offers more benefits than the conventional method. The first is the variable DFT Transform size. The size of the DFT transform varies depending on the number of plaintext elements. While the maximum is $\frac{N}{2}$, the minimum is one. Therefore, a much smaller DFT transform can be used in conventional settings for plaintexts with fewer elements, resulting in faster computation. While this is easier to implement in software, enabling this flexibility is costly in hardware due to the multiple configurations required. Yet, offloading this computation to the server through the Deflate technique allows for arbitrary packing flexibility while reducing computational costs for the client.

Another advantage is the elimination of floating-point arithmetic support required for the DFT transformation. In prior work [Kri+24] supporting full client-side encryption, a dedicated floating-point arithmetic unit contributes to $\approx$30% area overhead on an FPGA platform. By offloading the DFT transform to the server, we eliminate the need for this expensive support on the client side. This is beneficial for edge devices, such as many microcontrollers that do not support floating-point arithmetic (e.g., Mega32 [ATM24], Cortex-M4 [Kan+19], etc.). Additionally, omitting DFT reduces power consumption, as floating-point operations are power-hungry [LN10]. Hence, by leveraging Deflate technique, we significantly reduce the computational burden on the client while maintaining flexibility and efficiency in the encryption and decryption processes.

Impact on Precision. The homomorphic evaluation of the encode and decode steps does not result in additional precision loss. This is because these steps are integral parts of the bootstrapping routine. Therefore, the precision provided remains consistent with the post-bootstrapping precision specified by the scheme parameters. This ensures that the overall precision of the computations is maintained and the limit is set by the bootstrapping procedure.

Computational Overhead on the Server. Deflate offloads all DFT transform computations to the server. The time consumption for these transforms depends on the parameter selection and packing. Still, it is less than the cost of one bootstrapping operation, which involves similar transforms and additional computations. ML applications often require multiple bootstrapping operations (e.g., 96 for Logistic Regression Training [Han+19; Jun+21b] or 7,000 [Aik+25] for DNN training), making the impact of a single additional DFT minimal. Thus, the Deflate technique significantly enhances client-side operations by leveraging the server's computational power for the more complex DFT transforms.

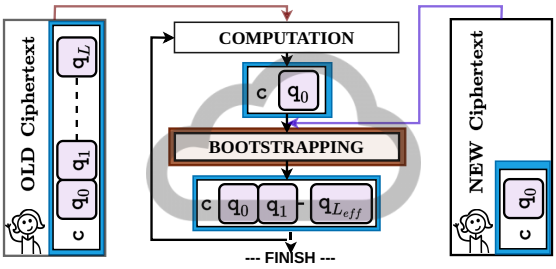


Figure 8.4: A high-level depiction of the key idea behind the proposed Boost technique. The 'OLD' and 'NEW' keywords show the ciphertext and data flow for the conventional and our proposed techniques, respectively.

8.2.3 Boost Technique

This idea is to leverage bootstrapping to reduce CoCo overhead on the client, providing a more efficient solution for secure computations. By following this technique, the decryption routine remains unchanged, and the encryption routine described in Section 2.2 can be rewritten as follows.

- $\texttt{CKKS.Enc}(m, \texttt{pk})$: It encrypts m, and returns $\texttt{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1) = v \cdot \texttt{pk} + (\Delta \cdot \texttt{IDFT}(m) + e_0, e_1) \in \mathbb{R}^2_{q_0}$.

Explanation. The dynamic computational depth l of a ciphertext directly depends on the available RNS bases (l+1) in the ciphertext. Due to this limitation, the initial proposal of schemes was considered levelled (SHE), as they could not support arbitrary computation depth. Gentry's seminal work [Gen09] proposed a method to bootstrap the ciphertext, allowing for unlimited computation depth, which was subsequently applied to all levelled schemes.

While bootstrapping has significantly advanced server-side computing capabilities, it has not been utilized to reduce the client's CoCo overhead. In this Chapter, we bridge this gap, and instead of sending a ciphertext with maximum computation depth, we suggest transmitting a ciphertext at zero computation depth (one RNS base only). The encryption process does not mix different RNS bases, as mentioned in Section 8.2.1. Thus, the CoCo requirements linearly depend on the number of RNS bases in the ciphertext. Encrypting for zero computational depth implies adding the encoded message to the product $v \cdot \texttt{pk}^2 \in \mathbb{R}_{q_0}$ (corresponding to a single RNS base).

Although this ciphertext cannot be used for computations directly, bootstrapping can be employed on the server side to increase its depth, enabling it to be processed as needed. Since the form of the encrypted ciphertext remains unchanged, the decryption routine remains the same. This approach, as illustrated in Figure 8.4, significantly reduces the CoCo overhead on the client side.

Impact on CoCo.

- *Reduced Client → Server Communication Overhead*: The communication

overhead from the client to the server is now equivalent to that from the server to the client, as the ciphertext in transit always has zero computational depth. This results in $(L+1)\times$ reduction in communication overhead.

- *Decreased Encryption Computation overhead*: The computations during encryption are also reduced by a factor of $(L+1)\times$ in terms of NTT operations. This is because only the residue polynomials associated with a single RNS modulus q_0 are processed instead of processing $(L+1)$ moduli.

- The Server $\rightarrow$ Client communication overhead and the decryption computation remain unaffected.

Other Notable Advantages. A noticeable advantage of this technique is that reducing the number of polynomials leads to a decrease in the storage requirements for ciphertexts during and after processing. A less obvious but important benefit is that enforcing bootstrapping as the first step on the server could offer protection against IND-CPA^D attacks, introduced in [LM21]. This attack relies on the fact that the result of CKKS decryption is not m, but instead $\Delta \cdot m + e$. Suppose this error is the same as the error used for encryption, for example, when the server evaluates an identity function. In that case, encryption security $(-a \cdot s + e + \Delta \cdot m)$ does not hold, as a is public, and an attacker can recover secret s. Hence, in the IND-CPA^D threat model, CKKS is not secure. However, the authors list bootstrapping as a possible effective countermeasure due to the noise flooding that results from the process. Thus, this aspect deserves further investigation.

Impact on Precision. There are several works in the literature analyzing bootstrapping for various FHE schemes [Bae+24; BP23; Che+18c]. Our technique imposes no limitations on the type of bootstrapping used, allowing for the utilization of high-precision bootstrapping [BP23]. Since we do not introduce any additional or new operations, the precision of our technique is bounded solely by the bootstrapping precision of the underlying FHE scheme.

Computational Overhead on the Server. This technique reduces the computation overhead for the client, but it requires the server to perform one additional bootstrapping operation. Since the depth after bootstrapping is less than the original depth ($L_{\mathit{eff}} < L$) and $L_{\mathit{eff}} \geq \frac{L}{2}$, applications requiring high-depth computations ($> L_{\mathit{eff}}$) will necessitate one more bootstrapping operation. Consequently, the server's computation overhead will be at least one and at most two bootstrapping operations, as illustrated in Figure 8.5. Many useful applications require numerous bootstrapping operations (e.g., [Aik+25] DNN training benchmark necessitating 7,000 bootstrapping), making the impact of two additional bootstrapping operations minimal.

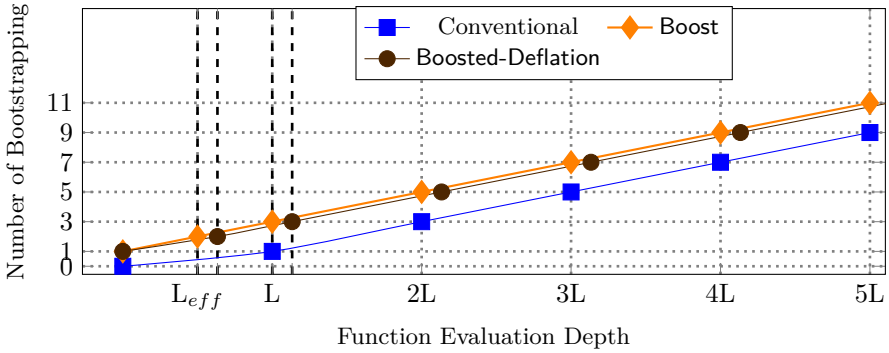


Figure 8.5: The number of bootstrapping operations performed on the server side increases by a constant 2 for high-depth computations ($> L_{eff}$) and 1 for lower-depth functions. For the combined Boosted-Deflation technique, the offset is lowered, allowing more computations per bootstrapping.

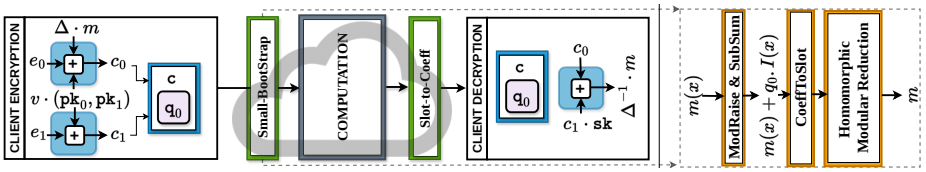


Figure 8.6: The operation flow following the proposed Boosted-Deflation technique. The client performs zero-depth encryption without encoding. The server performs a small bootstrapping step, where the initial `SlotToCoeff` operation is omitted, resulting in a cost of only 72% of the original bootstrapping. This helps save computational depth after bootstrapping. The server also performs the homomorphic decode (`SlotToCoeff`), so the client only has to compute decryption.

8.2.4 Boosted-Deflation Technique

While our two proposed techniques offer several standalone advantages, they have conflicting requirements. The Boost technique proposes encrypting with zero computational depth, while the Deflate technique requires sufficient depth ($l = 3$ or 4 depending on the parameters [Al +22]) post-encryption for the server to compute homomorphic encoding. However, when combined, these techniques complement each other, enhancing their respective benefits as shown in Figure 8.6. After applying both techniques, the final en/decryption procedures are as follows (shown in Table 8.1).

- $\texttt{CKKS.Enc}(m, \texttt{pk})$: It encrypts m, and returns $\texttt{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1) = v \cdot \texttt{pk} + (\Delta \cdot \cancel{\texttt{IDFT}}(m) + e_0, e_1) \in \mathbb{R}^2_{\cancel{Q}\ q_0}$.
- $\texttt{CKKS.Dec}(\texttt{c}, \texttt{sk})$: Returns message $m' = \cancel{\texttt{DFT}}(\Delta^{-1} \cdot \langle \texttt{c}, \texttt{sk} \rangle)$.

Table 8.1: A description of the data and computation flow after utilizing the proposed techniques. (algorithm in Section A.10)

Client		**Server**	
Encryption			
$(e_0, e_1), v$ are refreshed $\boldsymbol{c_0} = v_{\text{NTT}} \cdot \texttt{pk}_0 + (\Delta \cdot m + e_0)_{\text{NTT}} \bmod q_0$ $\boldsymbol{c_1} = v_{\text{NTT}} \cdot \texttt{pk}_1 + (e_1)_{\text{NTT}} \bmod q_0$ Send $\texttt{c} = (\boldsymbol{c_0}, \boldsymbol{c_1})$	$\rightarrow$	$\rightarrow$	Receive $\texttt{c}$
Server performs Function Evaluation			
		▷ Boosted-Deflation $\texttt{c}' = \text{Small-Boot}(\texttt{c})$ $\texttt{c}'' = f(\texttt{c}')$ ▷ Eval. $\acute{c} = \texttt{SlotToCoeff}(\texttt{c}'')$	
Receive $\acute{c} = (\acute{\boldsymbol{c}}_0, \acute{\boldsymbol{c}}_1)$	$\leftarrow$	$\leftarrow$	Send $\acute{c}$
Decryption			
$m' = \Delta^{-1}((\acute{\boldsymbol{c}}_1 \cdot \texttt{sk} + \acute{\boldsymbol{c}}_0)_{\text{INTT}} \bmod q_0)$			

Explanation. Directly combining both techniques would involve initially encrypting with a computational depth of at least three, thereby encrypting with at least four moduli. Thus increasing computations by up to 4× and eliminating only one DFT during encryption. Subsequently, the ciphertext will be sent to the server, where the first homomorphic encode step required by the Deflate technique will be performed. Following this step, the ciphertext will recede to a computational depth of zero, and bootstrapping will be executed to obtain additional computational depth. Finally, a homomorphic decode will be performed after the desired computations are completed.

Overall, the CoCo overhead would be 4× more when the two techniques are put together naively. This overshadows the advantages of using Deflate technique. However, upon more detailed analysis, we find that during the bootstrapping step, the `SlotToCoeff` transform is initially required to convert the ciphertext from plaintext slots to polynomials. This step essentially functions as a homomorphic decode necessary to reverse the encoding performed before encryption. We can omit this step since we are conducting encoding post-encryption as per the Deflate technique. In other words, the homomorphic encode performed for the Deflate technique and the first step of the bootstrapping operation (`SlotToCoeff`) cancel each other out and can thus be omitted. Thus, as depicted in Figure 8.6, a ciphertext can be sent directly for bootstrapping without necessitating an initial encoding or decoding step. The encoding will occur before the desired function evaluations begin, and decoding will take place at the end, before sending the ciphertext for decryption.

Impact on CoCo.

- *Reduced Client → Server Communication Overhead*: The communication overhead from the client to the server stays reduced by $(L+1)\times$, as the ciphertext expansion reduced from $4(L+1)\times$ to only $4\times$.

- *Decreased Encryption Computation Overhead*: The encryption computation

is reduced to three NTTs only, for the same single modulus.

- *Decreased Decryption Computation Overhead*: The computations during decryption are reduced to one NTT instead of one NTT and one DFT.

Impact on Precision. The proposed combination of the two techniques only reduces the extra homomorphic encode/decode transformations. Hence, the proposed technique does not lead to additional loss of precision. Users must be cautious when selecting implementation parameters to ensure overall precision.

Computational Overhead on the Server. By removing the homomorphic encode and decode steps, the computation offloaded to the server is reduced. Therefore, the server only performs computations equivalent to those for the Boost technique. This effectively makes Deflate technique cost-free while significantly reducing the client's computational overhead, as depicted in Figure 8.5.

8.2.5 Security Analysis

We now provide concrete arguments demonstrating that the proposed technique does not introduce any security vulnerabilities. Intuitively, this follows from the fact that Boosted-Deflation strictly adheres to existing CKKS routines.

❶ Deflate **technique analysis**. This technique offloads encoding and decoding to the server. Unlike encryption and decryption, which are private operations, encoding and decoding rely on publicly known DFT and Inverse IDFT operations, where the transformation matrix is fully public. Given a decrypted result, everybody can apply the inverse transform to recover the original message. Similarly, anybody can encode an input and send it to an encryption oracle. These operations do not provide security guarantees but are merely used for efficient data representation and computation.

❷ Boost **technique analysis**. The Boost technique reduces the initial ciphertext modulus, which may raise concerns about parameter safety. The underlying Ring Learning With Errors (RLWE) security assumption [Göt+12] imposes an upper bound on the ciphertext modulus to prevent excessive noise growth, but it does not require a strict lower bound. The hardness of the RLWE problem is determined by the ratio of noise to modulus size, and reducing the initial modulus does not weaken this assumption as long as noise levels remain within secure thresholds. The total ciphertext modulus (upper bound) required to maintain security [Alb+19] is inherently constrained once the ciphertext dimension is fixed during encryption. Our technique ensures this dimension (ring size) remains fixed. The bootstrapping process inherently maintains security under the RLWE assumption by refreshing noise accumulation. Additionally, noise flooding during bootstrapping may offer additional protection against IND-CPA^D attacks [LM21].

❸ Boosted-Deflation **technique analysis**. The combined use of Deflate and Boost results in Deflate becoming computationally free for the server, as no additional modifications are required to support it. This optimization pertains

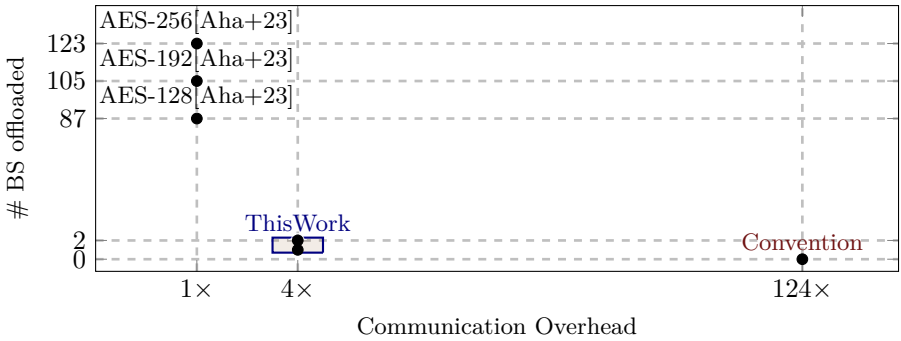


Figure 8.7: A comparison with works utilizing HHE to reduce CoCo. BS refers to Bootstrapping, for $L = 30$. Our work offloads at most two bootstrapping depending on the function.

strictly to computational efficiency rather than security; thus, their combination does not alter the underlying security assumptions of the FHE scheme. For concrete definitions, readers may refer to Section A.10.

❹ **Ciphertext Indistinguishability & IND-CPA Security**. The CKKS encryption scheme derives its security from the hardness of the RLWE problem [Göt+12], ensuring indistinguishability under chosen-plaintext attacks (IND-CPA). It guarantees that an adversary cannot distinguish between the encryptions of two chosen messages, as the ciphertexts are computationally indistinguishable under the RLWE assumption. The Deflate technique does not alter the RLWE structure, the error term, or the statistical distribution of the ciphertexts. As a result, it does not affect the security.

The Boost technique reduces the initial ciphertext modulus. Since the encryption function remains identical across all RNS moduli, the resultant ciphertext component under each modulus is indistinguishable. Consequently, the Boost technique does not violate IND-CPA security assumptions. Thus, our approach preserves the fundamental cryptographic guarantees of CKKS.

8.2.6 Comparison with Prior Techniques

The only other technique in the literature that aims to reduce the CoCo burden on the client and network is HHE [GHS12]. This technique enables the user to encrypt data via SE, and the server must perform decryption using a homomorphically encrypted key to obtain a homomorphic ciphertext (detailed in Section 2.3). For comparison, we first consider the HHE implementation [Aha+23] that utilizes the standard AES in stream modes (e.g., Counter and GCM) to reduce computation depth. Stream mode helps prevent sequential bootstrapping by allowing them to be executed in parallel.

The benchmarks on HHE presented in [Aha+23] demonstrate that 87/105/123 bootstrapping operations are computed by the server for AES-128/192/256

decryption in stream mode using CKKS. Although our technique has effectively reduced communication overhead, it still amounts to 4× the plaintext size. Consequently, while we experience 4× greater communication overhead, we offload significantly less computation to the server $(10 - 14\times)$, as shown in Figure 8.7. In [Aha+23], the authors pack 1 bit per plaintext slot and utilize bit-slicing. This implies that for one AES-256 ciphertext of size 128 bits, 128 FHE ciphertexts are generated. The authors pack 2^{15} AES ciphertexts into 128 FHE ciphertexts. For instance, a plaintext of size 32B and another of 512KB will both result in 128 FHE ciphertexts, with each ciphertext requiring 123 bootstrapping operations.

Let us consider the example of Homomorphic Logistic Regression Training [Han+19; Jun+21b], which requires 96 bootstrapping operations. The existing HHE-based technique will offload 123 bootstrapping per 512KB plaintext. Thus, more time is spent in homomorphic decryption than in actual function evaluation. As the dataset size increases [LC10] ($\approx 47MB$ size testing-set), the number of bootstrapping for HHE-decryption increases to $11,562$ ($\frac{123 \cdot 47MB}{512KB}$). The required amount of bootstrapping operations in the DNN benchmark on [LC10] is $7,000$; with HHE, this increases by 256%. In contrast, our technique offloads up to 98% less (222-444) bootstrapping operations.

A similar conclusion can be drawn from the multiplicative depth consumption reported by newer, more efficient, non-standard HHE schemes [Can+18; Ha+20], for example, Pasta [Dob+23]. Pasta is one of the state-of-the-art efficient schemes that enables HHE. As detailed in Section A.9, Pasta's homomorphic symmetric decryption routine consumes 320× more depth, and Pasta-3/4 require 32/16× higher computation for encrypting the same amount of data (2^{15} data elements). Note that in HHE, only the encryption is symmetric. The final decryption on the client side is homomorphic. Thus, our proposed Boost technique can always be utilized to avoid the homomorphic decoding step. These schemes were initially proposed specifically for the BGV/BFV FHE schemes [BGV11; FV12], which are limited to process integers. In [Cho+21; Ha+22], the authors, for the first time, demonstrated how to obtain a CKKS ciphertext using transciphering (bootstrapping) to convert a BGV ciphertext to a CKKS ciphertext. Therefore, an additional bootstrapping step is offloaded to the server along with homomorphic symmetric decryption to obtain the CKKS ciphertext. The ciphertext ratio for these techniques is also 1.54×, instead of 1 for AES.

Another prior work, [Bae+23], proposes an efficient ring-packing technique using Module-LWE ciphertexts at zero computational depth, which finds applications in transciphering. As the message in the ciphertext after packing is not encoded, the authors also utilize a HalfBoot technique, similar to the RtF framework in HERA [Cho+21], to gain computational depth after packing. The utilized ciphertext form is $\mathbf{c} \leftarrow \{a \cdot s + e, a\}$, where a seed can generate the a component on the server side instead of being sent via the communication channel, reducing the communication overhead. Note that this ciphertext structure differs significantly from the original structure proposed in CKKS/BGV ($a \cdot v + e$, where v and e are both unknown) and prevents on-the-fly computation of $\mathbf{c}_1$ component

by the server. Therefore, while we can not consider this structure, such works can also utilize our design (Section 8.3) to achieve acceleration. Furthermore, our en/decoding elimination analysis from enc/decryption remains useful.

8.3 Client Enc/Decryption Framework

This section details the framework which utilizes the proposed Boosted-Deflation technique across various platforms. To achieve this goal, we perform a detailed design exploration for software and hardware platforms, focusing on resource-constrained devices commonly used in client-side FHE.

8.3.1 Challenges

A common limitation of prior works [Aza+22; Aza+23; Kri+24; LDL23; MGS23] is their choice of parameters, which at most support a polynomial degree of 2^{14}. This degree is insufficient to enable *packed bootstrapping*.

① **Memory Constraints**. For 60-bit first modulus [LDL23], storing one residue polynomial requires $2^{16} \times 60 = 0.5$MB of on-chip memory. Given the constraints of the platforms commonly used for client-side FHE, this is a considerable amount of memory. Therefore, it is crucial to minimize the number of polynomials needing on-chip storage at any instance during computation.

② **Off-Chip Communication.** Due to the constrained on-chip memory, constant data, such as the public keys, is stored off-chip. This results in expensive data transfers between off-chip and on-chip memory, introducing additional communication overhead and negatively impacting performance. Furthermore, data transfers on client-side FHE consume significant energy. Hence, running FHE on resource-constrained or battery-powered devices is challenging, especially for smartphones or IoT devices.

③ **Limited Application Support**. Although a majority of practical applications (e.g., ResNet Inference [EGR23]) require bootstrapping, most of the prior client-side FHE works [Aza+23; Kri+24; MÖS20; ND21] do not support parameters that could employ packed bootstrapping.

④ **Multi-Moduli Support**. To support encryption for all moduli $(q_0 \dots q_L)$, generic Montgomery [Mon85] or Barrett [Bar86] reduction are utilized. These are expensive and involve several multiplications. In contrast, our Boosted-Deflation technique simplifies this by only using one fixed prime q_0 on the client side. Hence, a Mersenne prime can be chosen for a cheap add-shift-based reduction.

8.3.2 Framework Evaluation

❶ **Software Proof-of-Concept.** We first validated our proposed Boosted-Deflation technique via an implementation on the CPU. We utilize the existing HEAAN-library [HC23] and adapt it with minimal changes. HEAAN-library fully supports CKKS routines and uses $L = 28$. We run both the old and

Table 8.2: Comparison on Software (CPU), Embedded (Micro-Controllers), and Hardware (FPGA and ASIC) platforms.

Work	Platform	Freq. (GHz)	$\log_2 \mathbf{N}, q_0$	BS	Area				Latency (ms)		
					kLUT	kFF	DSP	BRAM	NTT	Encr.	Decr.
HEAAN[HC23]	12thGeni7	4.7	16, 60	✓	-				0.95	204.3	41.2
This Work	12thGeni7	**4.7**	**16, 60**	✓	-				**0.95**	**28.2**	**14.2**
Embedded SEAL[ND21]	CortexA7	0.49	16,30‡	✓	-				106	12,354	-*
Embedded SEAL[ND21]	CortexM4	0.06	16,30‡	✓	-				544	58,463	-*
This Work	CortexA7	**0.49**	**16,30**	✓	-				**106**	**753**	-*
This Work	CortexM4	**0.06**	**16,30**	✓	-				**544**	**4,535**	-*
[MÖS20]†	Virtex-7	0.2	10, 32	✗	77	-	952	325.5	4e-4	1e-3	8e-4
[MGS23]	Zynq US+	0.15	13, 32	✗	-	-	-	-	-	7.79	-
ALOHA [Kri+24]	Kintex-7	0.20	13, 54	✗	20.7	17.6	100	82.5	0.27	1.87	0.87
[LDL23]	AlveoU250	0.25	16, 60	✓	1,179	1,036	12,288	828.5	0.13	16.9	3.9
This Work	Kintex-7	**0.19**	**16 , 60**	✓	**31.4**	**34.9**	**204**	**326**	**0.71**	**3.78**	**2.64**
RACE [Aza+22]	ASIC12nm	1	13, 30	✗	0.06 mm^2				-	110.3	19.1
RACE [Aza+22]	ASIC12nm	1	14, 30	✗	0.12 mm^2				-	328.9	51.4
RISE [Aza+23]	ASIC12nm	1	13, 30	✗	0.11 mm^2				-	20.0	19.0
RISE [Aza+23]	ASIC12nm	1	14, 30	✗	0.18 mm^2				-	41.6	37.3
This Work	ASIC7nm	**1**	**16, 60**	✓	**0.22** mm^2				0.13	**0.70**	**0.49**
This Work	ASIC28nm	**1**	**16, 60**	✓	**3.36** mm^2				0.13	**0.70**	**0.49**

‡ Balanced implementation results generalized for parameter choice. ; † Utilizes BFV scheme; * Embedded SEAL does not support decryption.

adapted versions on a 12th Gen Intelő Core i7-1260PŒ16 Processor. The results show that the baseline memory allocated to the program is approximately 233MB, and after encryption, this increases to 261.4MB. Our technique's overall memory consumption is only 233.26MB, thus reducing storage for ≈56 residue polynomials. Table 8.2 shows that the old version takes 204.3/41.2 milliseconds (ms) for en/decryption; it reduces to 28.2/14.2ms with our technique.

❷ **Analysis on Embedded Microcontrollers.** Next, we analyze the benefit of Boosted-Deflation on the Azure Sphere Cortex-A7 (A7) and the Nordic nRF52840 Cortex-M4 (M4) microcontrollers using the Embedded SEAL library [ND21]. The challenge here is that Embedded SEAL only provides benchmarks for ring degree $N = 2^{14}$, which is too low for bootstrapping. Therefore, we first estimate its runtime using existing processing capabilities for a higher ring degree. Since the bit-width of primes considered in Embedded SEAL is 30-bit, we also consider the same bit-width for fair comparisons, implying $L = 56$ for $N = 2^{16}$. Table 8.2 provides the estimated Embedded SEAL encryption runtime for both A7 and M4 as 12.35 and 58.46 seconds. We reduce it by up to 16×.

❸ **Hardware Design on FPGA.** We set our target as the low-cost Kintex-7 FPGA (Genesys2, xc7k325tffg900-2)[1], also utilized in state-of-the-art work [Kri+24]. This FPGA features 204k Look-Up-Tables (LUTs), 408k Flip-Flops (FFs), 840 Digital Signal Processors (DSPs), and 445 Block Random Access Memories (BRAMs) with 36kb capacity. Of the 445 BRAMs, 40 are occupied by the used MicroBlaze CPU. Thus, the available memory can store a maximum of three residue polynomials (each occupying 108 36kBRAMs) at 82.3% utilization. We used Xilinx Vivado 2022.2 to run synthesis, place & route and verified correctness by running the design on the FPGA. The design occupies 31.4k LUTs, 34.9k FFs, 204 DSPs, and 326 36k BRAMs. The runtime reports are present in Table 8.2, and the architecture is shown in Figure 8.8. A detailed design concept is presented

[1]Our implementation is available at https://github.com/aikata10/Client_Boost_Deflate

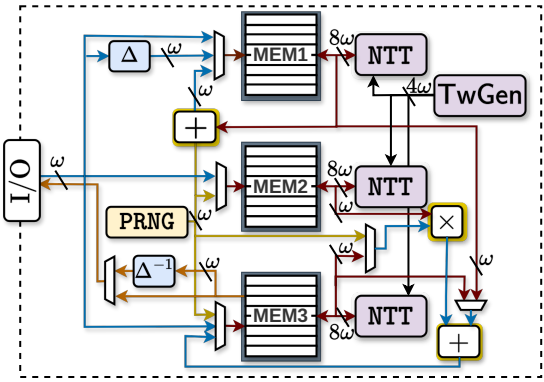


Figure 8.8: The proposed high-level architecture of the client device performing homomorphic en/decryption.

Table 8.3: Component-wise FPGA Area Utilization and Performance. Design details are provided in Section A.11.

Component	Area				Lat. (kcc)
	k LUT	k FF	DSP	BRAM	
MEM (3×)	0.8	1.4	0	108	65.6
NTT (3×)	6.8	7.2	48	0	131
TW Gen.	6.6	8.7	48	2	
M-Ops. (+, ×)	0.9	1	12	0	65.6
PRNG	0.7	0.5	0	0	131
Scaling	0.7	0.3	0	0	65.6
Total	**32/16%**	**36/9%**	**204/24%**	**326/73%**	-

in Section A.11. FPGA utilization of the various building blocks is shown in Table 8.3.

❹ **ASIC Synthesis Assessment.** We also present ASIC-synthesis results using TSMC 28nm and ASAP7 [Cla+16] 7nm libraries to compare with prior ASIC works [Aza+23]. The synthesis is performed using Cadence Genus 2019.11, and clock-gating is employed to help reduce power consumption. The complete design consumes 3.36 mm^2 area on the 28nm process and 0.22 mm^2 area on the 7nm process. The SRAM memories alone consume 83% of the area (2.89 mm^2 in 28nm). The maximum total power consumption of the device is 2.79 watts. Encryption takes 0.70 ms, and decryption operations consume 0.49 ms. Thus, the chip would perform $1,429$ encryptions and $2,041$ decryptions per second.

8.4 Comparison with Related Works

The prior works do not support arbitrary packing. Our work offers this by default due to the proposed Boosted-Deflation technique. Importantly, all the prior works only provide accelerated computation. Our technique additionally helps reduce

communication overhead by up to 31× (97%) for the parameters used in [LDL23]. Table 8.2 compares our area and performance results to state-of-the-art works for client-side operations in FHE across all platforms.

❶ **Software.** Compared to HEAAN [HC23], we achieve up to 7× speedup for the same parameters running on the same platform without any optimization effort. This speedup is attributed solely to our proposed Boosted-Deflation technique. Although the old implementation processes 29 times more data during encryption (for $L = 28$), we only see a 7× speedup. This is due to the concurrent processing of multiple residues on the multithreaded CPU. Since our technique allows for using a single modulus q_0, we do not leverage thread-level parallelism for single encryptions. Instead, the Boosted-Deflation technique allows multiple independent encryptions simultaneously, thus delivering a higher throughput. Moreover, our technique will provide higher speedups on single-core CPUs or microcontrollers common in client-side FHE compared to the multi-threaded case.

❷ **Microcontrollers.** On Cortex-A7, the estimated encryption latency is 12.35 seconds, whereas, with Boosted-Deflation, the latency is only 0.75 seconds. On Cortex-M4, the speedup achieved using the Boosted-Deflation technique compared to Embedded SEAL is 12.9×. Note that the Cortex-M4 only provides a single-precision floating-point unit, although FFT requires double precision. Hence, the compiler emulates double-precision FP arithmetic using integer arithmetic. This leads to a significantly increased runtime for the FFT, as reported using the Embedded SEAL [ND21]. The Cortex-M4 exhibits up to 100× worse FFT performance than the Cortex-A7 with 7.7× lower clock frequency. Furthermore, the double-precision FFT consumes 11.5× more clock cycles than NTT (operating on integers) for the same data size. This highlights the benefit of omitting the floating-point-based encoding proposed by our Deflate technique on microcontrollers without an FPU.

❸ **FPGA.** The authors in [MGS23] propose an accelerator for resource-constrained devices. Only the encryption is performed in hardware, while the software does the encoding. In contrast, [Kri+24] performs the floating-point-based encoding on the FPGA. Compared to our work, both works [Kri+24; MGS23] only support 8× smaller polynomials; yet, we outperform [MGS23] by 2× while being 2× slower than [Kri+24]. This leads to a scaled performance improvement of 16× and 4× compared to [MGS23] and [Kri+24], respectively. Compared to [Kri+24], we require 4× more BRAMs than [Kri+24], although we support 8× larger polynomials. This is an advantage of the Deflate technique, which omits the DFT and avoids the corresponding memory in hardware.

[LDL23] presents an implementation of high-performance encryption and decryption operations. The authors target the Alveo U250 datacenter FPGA and propose a large-scale design. Considering the NTT performance, [LDL23] outperforms our NTT by 5.5 × with the same parameter set. Yet, due to our Boosted-Deflation optimizations, we reduce the overall encryption latency by 4.5×. In addition, we utilize 38×, 30×, 60×, 2.6× fewer resources in terms of LUT, FF, DSP, and BRAM, respectively. This significantly lower resource utilization

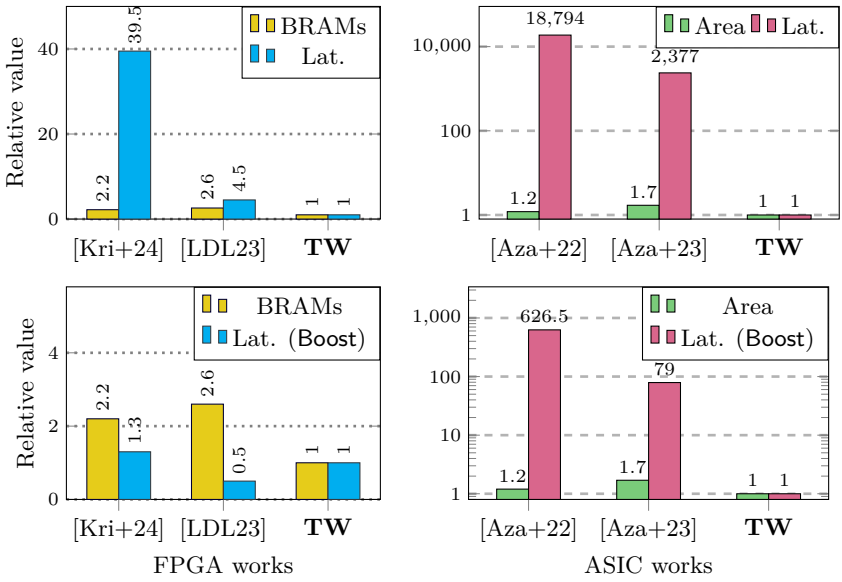


Figure 8.9: Comparison with prior works. The bottom plots apply Boost techniques on prior works.

allows the deployment of our design on resource-constrained FPGAs, whereas [LDL23] solely applies to high-end FPGAs.

The work in [MÖS20] presents a hardware-software co-design for the en/decryption in the BFV [FV12] scheme. They target polynomials that are 64× smaller and primes that are 2× smaller than our work, and report only hardware runtime. We, therefore, scale their latency up to our parameter set. This indicates that we achieve 1.4× higher performance while requiring 2.5×, 4.9×, and 64× fewer LUTs, DSPs, and BRAMs, respectively. In addition, [MÖS20] targets a more powerful and larger FPGA- Virtex 7. This limits their applicability for resource-constrained client devices, highlighting the clear benefit of our methodology.

❹ **ASIC.** There are no standalone client-side FHE acceleration ASIC designs, as prior works [Aza+23] use SW for encoding and decoding steps. Naively comparing the latency shows that we deliver $30 - 157\times$ better performance. Although we achieve better results, the speedup comparisons provided are somewhat unfair to our work and proposed technique, as prior works also process fewer residue polynomials for smaller parameters. Thus, we normalize the area by scaling the memory with the difference in polynomial degrees and ω, since memories consume most of the area in the ASIC design and are the main constraints in FPGA. We present Figure 8.9 to compare our results using normalized area and performance metrics (for FPGA, we use BRAM area). The top two plots illustrate that our approach achieves the lowest area and runtime results, outperforming previous

methods by more than $L + 1 = 31$ times. The bottom two plots depict the scenario where prior works also have reduced computation overhead from the Boost technique, highlighting the advantage of our design exploration.

8.5 Application Benchmarking

We first perform an application benchmarking of video frame encryption [UUS18] to demonstrate that our encryption performance and data overhead are comparable to the available bandwidth in edge devices. This application is used in video surveillance systems to send encrypted video data to the cloud for processing [Lag+21]. It also covers the case of audio transmission required for [Ben+25]. A video is split into multiple frames, adaptable to available bandwidth and desired resolution [Men+24]. The resolution can range from VGA (640×480 pixels), QVGA (320×240 pixels), to QQVGA (160×120 pixels). The authors in [Aza+23] benchmark encrypting QVGA and QQVGA, assuming grayscale pixels (8-bit), for a bandwidth of 12.5 to 112.5 MBps. This bandwidth is for the network connecting the client's edge device to the cloud (e.g., mid-band 5G network [TAC19]). RISE [Aza+23] can encode one QQVGA frame per ciphertext ($N = 2^{14}, \log Q = 390$) and requires 3 such ciphertexts for a QVGA frame. One ciphertext size is 1.5MB ($2^{14} \cdot 2 \cdot 390$). Therefore, they can send 70 QQVGA frames per second when the maximum 5G bandwidth is available (112.5 MBps). It has also been reported that the baseline software implementation is unable to send even a single frame per second.

Our results, compared with those reported in [Aza+23], are presented in Figure 8.10. The ciphertext ($N = 2^{16}, \log q_0 = 60$) is only 0.9 MB in size ($2^{16} \cdot 2 \cdot 60$) due to the use of Boosted-Deflation technique. Consequently, we can send 1.7× more ciphertext for the same bandwidth. Additionally, due to the higher polynomial degree, we can pack 4× more frames, for example, 12 QQVGA and 3 QVGA frames per ciphertext. Hence, we can send $\approx 8 - 20\times$ more frames per second depending on the frame-bits packed per ciphertext, as shown in Figure 8.10. We also provide a benchmark for VGA resolution; for the minimal available 5G bandwidth, we can send six frames per second. The same could not be done using [Aza+23]. At the highest functioning rate of $\frac{1,344}{12} = 112$ encryptions per second, a latency of 8.9 ms per encryption is required. Our readily available FPGA realized implementation (3.78 ms) offers this. Note that the server has to process the video, and due to our bootstrappable parameter support, it can perform deep neural network evaluation [Mar+15; PDC21; Sha+21b].

8.5.1 Logistic Regression Training

The previous example demonstrates that the proposed technique is comparable to sending encrypted data to the server for practical applications, such as video or image analysis. Now, we will show an example of a logistic regression training benchmark, which would also put into perspective the CoCo overhead for the prior techniques versus our proposed Boosted-Deflation, as shown in Figure 8.11.

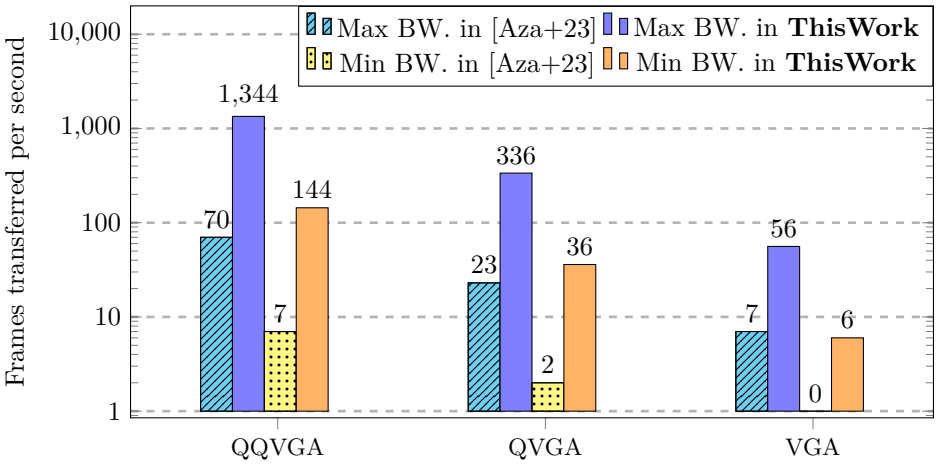


Figure 8.10: Frames transferred per second for Maximum (112.5 MBps) and Minimum (12.5 MBps) 5G available bandwidth (BW). The graphs are on a logarithmic scale.

The application does binary image classification between 3 and 8 on the MNIST dataset [LC10], and its homomorphic evaluation was proposed in [Han+19].

The initial size of each image was 28×28, which the authors preprocessed and compressed to 14×14 using the arithmetic mean for each 2×2 pixels. Out of the total of 13,966 samples in the dataset, 11,982 are used for training, and the remaining 2,984 are used for validation. The authors pack each pixel in one slot and use $N = 2^{16}$. Each ciphertext requires 8.8MB of storage. They use a block size of 1024 and, therefore, require eight such ciphertexts per batch. Thus, overall, they require $12 \times 8 = 96$ ciphertexts for encrypting all the images and sending them to the server, which amounts to 845MB of data. The state-of-the-art client accelerators [Aza+23] require 32s to encrypt this huge amount of data. On the other hand, recent FHE hardware acceleration works for the server [Agr+23] shows that this logistic regression training can be done in a more practical 3.1 seconds.

Thus, we are compelled to ask whether transferring such a large amount of data can be done as quickly as possible. Sending 845MB of data, at the highest 5G available bandwidth, requires ≈8s; at the lowest bandwidth, this increases to ≈68s. Thus, on average, 38 seconds are spent sending data to the server alone. Now, if we use the prior technique of HHE to minimize this, the total ciphertext size will be 19MB ($196 \times 11,982 \times 64$), where each element is again encrypted in one slot of the integer HHE scheme [Cho+21; Dob+23; Ha+22]. This requires ≈1.5s; however, the computation of these many ciphertexts alone requires 56s [Cho+21], and it also offloads 2.1× larger computation overhead to the server, resulting in a total execution time of 6.6s.

With our proposed Boosted-Deflation technique and its framework, the ci-

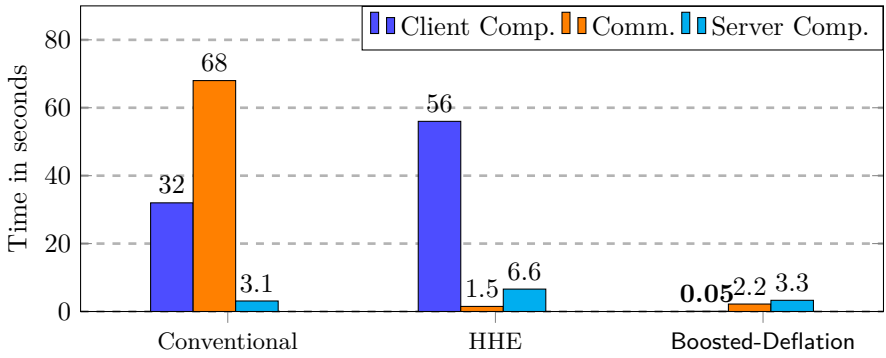


Figure 8.11: Time required for client and server computation, and communication for the Logistic Regression Training

phertext size reduces from 845MB to 76MB, and the server computation time merely increases from 3.1 to 3.8s. Furthermore, we note that the value a pixel can take is in the range (0-7). Hence, 3 bits can be allocated for each pixel, and packing multiple pixels (3-6) in one slot can help reduce the ciphertext size to 13-25MB, which is very close to HHE. Furthermore, this also lowers the computation time on the server to 3.2-3.3s. With our proposed framework, the client-side encryption can also be done in 0.01-0.05s. The amount of time the client requires to encrypt such huge data is reasonable, given that ML training on high-end accelerators takes 3.2 seconds. Thus, the technique presents the best state of the results, minimizing computation for both the client and server, as well as the communication overhead over the network.

8.6 Discussion

The proposed Boosted-Deflation technique is versatile and can be applied beyond plain FHE to other protocols that utilize FHE, such as Multi-Key FHE [Kim+23c] and Multi-Party FHE [Ash+12], to help mitigate the CoCo overhead. These protocols frequently employ the CKKS scheme as well as other FHE schemes.

■ **Applicability to other FHE schemes.** In this study, we used the CKKS FHE scheme as a proof of concept. However, the same concepts can be applied to other FHE schemes such as BGV [BGV11] and BFV [FV12]. These schemes are levelled and support bootstrapping to enable unlimited computation depth, so the proposed Boost technique can be directly applied to their implementation. They operate over an integer plaintext space, and an NTT transform is utilized for en/decoding, which can be offloaded to the server as per the Deflate technique. Hence, the proposed Boosted-Deflation technique can be applied to these schemes and presented as an interesting future benchmarking scope.

Interestingly, the encoding and decoding steps were the major differences

between CKKS and BGV/BFV encryptions. Now that these steps are offloaded to the server, the proposed client-side implementation can be utilized for all these schemes, offering greater flexibility. We further remark that while our technique directly benefits levelled FHE schemes, its applicability to other schemes, such as TFHE [Chi+20], is not straightforward and warrants future exploration, as TFHE has an 8,000× ciphertext expansion [Cas+19].

■ **How can existing accelerators benefit from our technique?** Works utilizing the same polynomial degree [LDL23] can directly implement our Boosted-Deflation technique to reduce computational demands. For other works, adaptations in storage requirements will be necessary. Figure 8.9 illustrates how applying the Boost technique can significantly reduce runtime in existing implementations. Many of these designs follow a strict pipelined flow with the DFT transform integrated as a part of the process. These designs can achieve substantial improvements by modifying the design methodology to fully incorporate the Boosted-Deflation technique. Implementing the Boost technique alone is sufficient to mitigate memory overhead. Therefore, existing works can leverage the proposed technique by adjusting their storage and processing requirements to support bootstrappable parameters and reap its benefits.

9

Secure Matrix Multiplication with Homomorphic Encryption

Breu

Unlike earlier chapters, which focused on the design, acceleration, and implementation security of FHE primitives, this chapter marks an interesting transition to the application layer. Here, we explore how to optimally utilize FHE routines. This work optimizes privacy-preserving Matrix-Multiplication, a backbone of Neural Networks, targeting performance bottlenecks at the algorithmic level.

By reducing KeySwitching complexity from $\mathcal{O}(d)$ to $\mathcal{O}(\log d)$, without increasing multiplicative depth, our method outperforms all known approaches, including those with deeper circuits. This practical (Single Instruction Multiple Data) SIMD-aware optimization unlocks faster privacy-preserving neural network evaluation, representing a decisive step toward real-world FHE applications.

Contribution: This chapter is based on the following paper, led by the thesis author, who contributed to the development of the concept, performed the experiments, completed the implementations, and drafted large portions of the text.

• *Secure and Efficient Outsourced Matrix Multiplication with Homomorphic Encryption* by Aikata and Sujoy Sinha Roy, in INDOCRYPT 2024.

9.1 Introduction

Most of the efficient and high-performing FHE schemes [BGV11; Che+17; FV12] are lattice-based. A key limitation of these schemes is their linear slot-wise ciphertext encoding, which can be conceptualized as a one-dimensional array where each plaintext element occupies a single index. This encoding restricts operations requiring permutation, as elements cannot be easily extracted from the array, unlike in plaintext operations. Consequently, performing permutations on homomorphic ciphertext necessitates costly multiplications with masks and rotations. While this is manageable for operations like approximate function evaluation that operate slot-wise, it poses a significant challenge for matrix multiplication.

Matrix multiplication is fundamental in advanced mathematics and is especially critical for secure data analysis and machine learning (ML), particularly within neural networks (NN) [HZ23; Jan+22; Jia+18; RT22; Zhu+23]. Network components, such as fully connected layers and filters/kernels, depend heavily on efficient matrix multiplication. Although efficient algorithms, such as Strassen's algorithm, exist for plaintext operations, conducting matrix multiplication in the encrypted domain remains an emerging area of research. This field has garnered attention for its potential to facilitate encrypted ML training and inference using FHE schemes like CKKS [Che+18a], which support approximate arithmetic.

9.1.1 Prior Works.

The authors in [HS14] introduced the first technique for multiplying encrypted matrices and vectors, which was later extended to matrix-matrix multiplication by [LKS17] and [WH19]. However, these methods require a substantial number of homomorphic multiplications and rotation operations. The literature [HZ23; Jan+22; Jia+18; LKS17; RT22; WH19; Zhu+23] on encrypted matrix multiplication (referred to as matrix-matrix multiplication from here on) can be broadly categorized into three types, as summarized in Table 9.1.

The first type necessitates a multiplicative depth of two and employs a simple row-wise encoding for the initial input data. The work in [RT22] exemplifies this approach, presenting a technique for d^3 packing availability in the ciphertext. For a square matrix of dimensions $d \times d$, this method requires $2 \cdot d + 3 \cdot \log_2(d) - 2$ rotations and one ciphertext-ciphertext (ct-ct) multiplication. Importantly, these operations are costly, as a KeySwitch operation is required after each to ensure that the ciphertext is decryptable with the same secret key. Thus, the total KeySwitch complexity amounts to $2 \cdot d + 3 \cdot \log_2(d) - 1$. A significant limitation of this approach is its requirement for d^3 slots of packing availability, which limits scalability for large matrices. Additionally, it does not generalize well for lower packing availability.

The second category, as described in [Jia+18], diverges slightly from previous methods by utilizing diagonal-packing for matrix multiplication. In this approach, matrices are packed diagonally rather than row- or column-wise. While this technique is highly complex for a multiplicative depth of two, it allows for some

Table 9.1: Comparison with secure d-dimensional matrix multiplication techniques. The division of works is done based on the three types of works discussed in Section 9.1.1. # Key-Switches = # ct-ct Mult + # Rotations.

Methodology	**Packing**	**# ct-ct Mult**	**# Rotations**	**# Depth†**
Naive	1	$\mathcal{O}(d^3)$	-	2
[LKS17; WH19]	d	$\mathcal{O}(d^2)$	$\mathcal{O}(d^2 \log_2 d)$	2
This Work	$\mathbf{d^2}$	$\mathcal{O}(\mathbf{d})$	$\mathcal{O}(\mathbf{d} \log_2 \mathbf{d})$	**2**
[RT22]	d^3	$\mathcal{O}(1)$	$\mathcal{O}(d)$	2
This Work	$\mathbf{d^3}$	$\mathcal{O}(\mathbf{1})$	$\mathcal{O}(\log_2 \mathbf{d})$	**2**
[Jia+18]	$d^2, d^{3\ddagger}$	$\mathcal{O}(d)$	$\mathcal{O}(d)$	3
[CKY18; HZ23; Jan+22; Zhu+23]	d^2	$\mathcal{O}(d)$	$\mathcal{O}(d \log_2 d)$	2

† This includes (Plaintext-Ciphertext) pt-ct and ct-ct multiplications, which consume the same depth in Libraries like OpenFHE [Al +22].
‡ With $d\times$ more packing, the number of rotations reduce from $3 \cdot d + 5 \cdot \sqrt{d}$ to $d + 2 \cdot \sqrt{d}$.

pre-processing at a higher multiplicative depth of three, reducing the complexity to $3d + 5\sqrt{d}$ rotations and d ct-ct multiplications. If d^3 slot packing is available, it can be further optimized to require $d + 2\sqrt{d}$ rotation computations. However, a major drawback is the necessity for three multiplicative depths. Multiplicative depth is the currency in FHE schemes; the less spent, the better, as more depth remains for remaining computations. Although the algorithms proposed in [Jia+18] can be adjusted to operate within a multiplicative depth of two, this significantly increases the number of rotations and ct-ct multiplications.

The third and final category of works, as discussed in [HZ23; Jan+22; Zhu+23], diverges significantly from the previous two types. These studies leverage a multivariate variant of the CKKS scheme (m-RLWE) [CKY18], which enables the encoding of matrices into a hypercube structure (tensor packing) instead of a linear array-like structure typical of CKKS. This approach enables more efficient rotations, making row-wise or column-wise transformations more cost-effective. Matrix multiplication using this scheme requires only a multiplicative depth of two, with the cost of transformations reduced to $2 \cdot d + 4\sqrt{d}$ rotations and d ct-ct multiplications. However, the multivariate CKKS [CKY18] is incompatible with the original CKKS [Che+18a]. Furthermore, the parameters for multivariate CKKS are not standardized, and its initial proposal [PTP15] was found to be insecure [BCV18], limiting its adaptability for existing implementations.

This category also includes recent works [Ju+23; ZLW23] that propose altering the initial encoding of ciphertexts to facilitate faster multiplication on the server. Such modifications require changing the specifications of the FHE scheme or necessitating client support for different encodings. When client data is already encrypted and stored on the server, these techniques become impractical, as the server would need to adjust the encoding to the desired form. For new computations, the FHE encoding that employs a Discrete Fourier Transform (DFT) is already resource-intensive for the FHE client. Consequently, we opted

not to pursue this direction, as it merely shifts the computational burden from the server to the client, which has even less computational capacity.

9.1.2 Contributions.

In this work, we restrict our solution to a multiplicative depth of two and build on the first type of technique discussed above. We observe that the best-known technique in this direction does not fully utilize the SIMD processing capabilities and leaves significant scope for optimization. We bridge this gap and propose a technique that improves with higher packing availability. For d^3 packing, our technique requires only $5 \cdot \log_2 d$ rotation operations and one ct-ct multiplication. Thus, this work presents an efficient homomorphic matrix multiplication framework for privacy-preserving applications. Its features are as follows.

- The proposed framework fully exploits the SIMD processing capabilities provided by FHE schemes and their routines. It is generalized for various packing availabilities, ranging from d^2 to d^3 for square matrices of dimension d, with benefits increasing with increasing packing availability in the homomorphic ciphertext.
- The KeySwitch operation in FHE is the most resource-intensive low-level routine and serves as a benchmark for assessing the performance overhead of our proposed techniques. We demonstrate that for d^3 packing, our technique achieves the lowest KeySwitch complexity ($\mathcal{O}(\log_2 d)$) compared to all prior works in the literature. This is also illustrated in Table 9.1.
- While our initial proposal focuses on square matrices, we further generalize it to accommodate rectangular matrices. To this end, we introduce two techniques based on padding and divide-and-conquer strategies, thereby enhancing the versatility of our framework for various neural network applications and layers, including filter layers.
- Alongside our proposal, we provide validation artefacts for our technique, which can be accessed at `https://github.com/aikata10/MatMul`. Our approach leverages the open-source FHE library OpenFHE [Al +22], allowing researchers and practitioners to easily integrate our matrix multiplication framework into their own projects.

9.1.3 Roadmap

The chapter is organized as follows: In Section 9.2, we provide an overview of the prior state-of-the-art matrix multiplication techniques. Section 9.3 details our proposed technique, explaining how it achieves improved runtime complexity. This section also discusses the generalized approach for arbitrary packing availability within the range of d^2 to d^3. In Section 9.4, we extend this technique to accommodate arbitrary rectangular matrix multiplication. The experimental

evaluation is presented in Section 9.5, where we assess the performance of our approach. Section 9.6 explores scenarios involving simultaneous matrix multiplications, and Section 9.7 concludes the chapter.

9.2 Prior Matrix Multiplication Techniques

In this section, we will introduce the state-of-the-art technique for matrix multiplication at multiplicative depth two, presented in [RT22]. This technique leverages the available packing capability of d^3. To simplify this, let us take an example of a processing system which only operates on the array. The proposal in the work utilizes arrays which can store d^3 elements, where each matrix to be multiplied is $d \times d$. In this approach, the first step is to decide how to effectively pack a matrix in the array so that it facilitates multiplication. Suppose we have two matrices, A and B, of dimension 4×4 ($d = 4$) for multiplication. This means our array must accommodate 64 elements.

$$\text{A}= \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix} \text{B}= \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix}$$

Packing Strategy. To facilitate efficient multiplication, the authors needed to adopt a packing strategy that organizes the elements of matrices A and B into the array. One common approach is to fill the array in a row-major or column-major order, depending on the operations we intend to perform. They chose a row-major order, where the elements of matrix A would be packed into the first 16 slots of the array. Similarly, matrix B would be packed in the first 16 slots of another array, as shown below.

$$A = [a_0a_1 \quad a_2a_3 \quad a_4a_5 \quad a_6a_7 \quad a_8a_9 \quad a_{10}a_{11} \quad a_{12}a_{13} \quad a_{14}a_{15}] \tag{9.1}$$
$$B = [b_0b_1 \quad b_2b_3 \quad b_4b_5 \quad b_6b_7 \quad b_8b_9 \quad b_{10}b_{11} \quad b_{12}b_{13} \quad b_{14}b_{15}] \tag{9.2}$$

Matrix Multiplication Strategy. The authors in [RT22] employ an efficient multiplication technique that enhances the performance of matrix multiplication in the homomorphic setting. In this approach, the elements of matrix A are duplicated column-wise, while the elements of matrix B are duplicated row-wise. For example, the first column of A and the first row of B are as follows.

$$\text{A}_1= \begin{bmatrix} a_0 \\ a_4 \\ a_8 \\ a_{12} \end{bmatrix} \text{B}_1= \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix}$$

Once the rows and columns are duplicated, the proposed technique gives the multiplied result A·B as follows.

$$\text{A·B}= \begin{bmatrix} a_0 & a_0 & a_0 & a_0 \\ a_4 & a_4 & a_4 & a_4 \\ a_8 & a_8 & a_8 & a_8 \\ a_{12} & a_{12} & a_{12} & a_{12} \end{bmatrix} \odot \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \end{bmatrix} + \begin{bmatrix} a_1 & a_1 & a_1 & a_1 \\ a_5 & a_5 & a_5 & a_5 \\ a_9 & a_9 & a_9 & a_9 \\ a_{13} & a_{13} & a_{13} & a_{13} \end{bmatrix} \odot \begin{bmatrix} b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \end{bmatrix} + \begin{bmatrix} a_2 & a_2 & a_2 & a_2 \\ a_6 & a_6 & a_6 & a_6 \\ a_{10} & a_{10} & a_{10} & a_{10} \\ a_{14} & a_{14} & a_{14} & a_{14} \end{bmatrix} \odot \begin{bmatrix} b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \end{bmatrix} + \begin{bmatrix} a_3 & a_3 & a_3 & a_3 \\ a_7 & a_7 & a_7 & a_7 \\ a_{11} & a_{11} & a_{11} & a_{11} \\ a_{15} & a_{15} & a_{15} & a_{15} \end{bmatrix} \odot \begin{bmatrix} b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix}$$

$$A \cdot B \;=\; \begin{bmatrix} a_0 & a_0 & a_0 & a_0 \\ a_4 & a_4 & a_4 & a_4 \\ a_8 & a_8 & a_8 & a_8 \\ a_{12} & a_{12} & a_{12} & a_{12} \\ \hline a_1 & a_1 & a_1 & a_1 \\ a_5 & a_5 & a_5 & a_5 \\ a_9 & a_9 & a_9 & a_9 \\ a_{13} & a_{13} & a_{13} & a_{13} \\ \hline a_2 & a_2 & a_2 & a_2 \\ a_6 & a_6 & a_6 & a_6 \\ a_{10} & a_{10} & a_{10} & a_{10} \\ a_{14} & a_{14} & a_{14} & a_{14} \\ \hline a_3 & a_3 & a_3 & a_3 \\ a_7 & a_7 & a_7 & a_7 \\ a_{11} & a_{11} & a_{11} & a_{11} \\ a_{15} & a_{15} & a_{15} & a_{15} \end{bmatrix} \odot \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ \hline b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ \hline b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ \hline b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix} \tag{9.3}$$

This technique requires d column-wise and row-wise duplications of the matrices. With $d^3(= 64)$ packing available, the matrices are stored as follows (their matrix form visualization is provided afterwards in Equation 9.3). Notably, if we assume that matrices A and B can be packed into two ciphertexts (one for A and one for B), as illustrated in Equation 9.4, then only one ct-ct multiplication is needed.

$$A \cdot B = [a_0\ a_0\ a_0\ a_0\ a_4\ a_4\ \cdots\ a_{15}\ a_{15}] \odot [b_0\ b_1\ b_2\ b_3\ b_0\ b_1\ \cdots\ b_{14}\ b_{15}] \tag{9.4}$$

There are two challenges associated with the packing format assumption. The first is transforming the input data into the required form. In prior works, all inputs (matrices A and B) are encoded in a row-wise format (row_enc), as shown in Equation 9.1 and Equation 9.2. Therefore, it is necessary to convert this row-wise packing into the desired encoding for our approach. The second challenge is how to accumulate the resulting multiplication results. The methods for addressing these two challenges distinguish the work presented in [RT22] from

Algorithm 25 `Matrix.Mult` [RT22]

In: $A, B \leftarrow \text{row_enc}(\mathtt{A}_{d\times d}, \mathtt{B}_{d\times d})$
Out: $C = \text{row_enc}(\mathtt{A}_{d\times d} \times \mathtt{B}_{d\times d})$

1: *// Preprocess A*
2: **for** $j = 0$ to $d-1$ **do**
3: $\quad \tilde{A}[j] \leftarrow \mathtt{cMult}(A, \pi_j)$ `/* Splitting A cols */`
4: $\quad \tilde{A}[j] \leftarrow \mathtt{Rot}(\tilde{A}[j], -j)$ `/* Right align all the cols */`
5: **end for**
6: **for** $i = 1$ to $d-1$ **do**
7: $\quad \tilde{A}[0] += \mathtt{Rot}(\tilde{A}[j], -i(d^2 - i))$ `/* Add the cols */`
8: **end for**
9: **for** $i = 0$ to $\log_2 d - 1$ **do**
10: $\quad \tilde{A}[0] += \mathtt{Rot}(\tilde{A}[0], -2^i)$ `/* Replicate cols */`
11: **end for**

12: *// Preprocess B*
13: **for** $j = 0$ to $d-1$ **do**
14: $\quad \tilde{B}[j] \leftarrow \mathtt{cMult}(B, \psi_j)$ `/* Splitting B rows */`
15: $\quad \tilde{B}[j] \leftarrow \mathtt{Rot}(\tilde{B}[j], -j \cdot d)$ `/* Top align all rows */`
16: **end for**
17: **for** $i = 1$ to $d-1$ **do**
18: $\quad \tilde{B}[0] += \mathtt{Rot}(\tilde{B}[j], -(d^2 - d)$ `/* Add the rows */`
19: **end for**
20: **for** $i = 1$ to $d-1$ **do**
21: $\quad \tilde{B}[0] += \mathtt{Rot}(\tilde{B}[0], -d \cdot 2^i)$ `/* Replicate the rows */`
22: **end for**

23: *// Compute C*
24: $C = \mathtt{Mult}(\tilde{A}[0], \tilde{B}[0])$
25: **for** $j = 0$ to $\log_2 d$ **do**
26: $\quad C = C + \mathtt{Rot}(C, d^3/2^i)$ `/* Result Accumulation */`
27: **end for**

our proposal, which is discussed in the next section. The technique proposed in [RT22] is detailed in Algorithm 25.

Plaintext Masks. The authors define two plaintext masks π_i and ψ_i, such that multiplication with π_i causes all expected elements of column i to become zero, while multiplication with ψ_i results in all expected elements of row i becoming zero, as illustrated above. This is achieved by placing '1' and '0' in the desired positions within another array of size d^2. Although this multiplication is plaintext-to-ciphertext (pt-ct), it requires an additional multiplicative depth. Consequently, the total multiplicative depth for the overall technique is two.

After obtaining these results, the values are first right-aligned using the rotations specified in Step 3 and Step 10 for matrices A and B. Once aligned, these matrices are appended next to each other, which requires $d-1$ rotations for both A and B (Steps 4-5 and 11-12). A duplication step is then performed to replicate these values and fill in the zeros in the ciphertexts, as shown in Steps 6-7 and 13-14. After multiplication, the results are accumulated again using $\log_2 d$ rotations, detailed in Steps 16-17. Overall, this technique requires $2 \cdot d$ pt-ct multiplications, one ct-ct multiplication, and $2 \cdot d + 3\log_2 d - 2$ rotations, while consuming a multiplicative depth of two. The authors limit this proposal

to square matrices and do not extend it to rectangular matrices or arbitrary packing.

$$
\begin{aligned}
\texttt{cMult}(A, \pi_0) &= [a_0 0 \quad 00 \quad a_4 0 \quad 00 \quad a_8 0 \quad 00 \quad a_{12} 0 \quad 00] \\
\texttt{cMult}(A, \pi_1) &= [0 a_1 \quad 00 \quad 0 a_5 \quad 00 \quad 0 a_9 \quad 00 \quad 0 a_{13} \quad 00] \\
\texttt{cMult}(A, \pi_2) &= [00 \quad a_2 0 \quad 00 \quad a_6 0 \quad 00 \quad a_{10} 0 \quad 00 \quad a_{14} 0] \\
\texttt{cMult}(A, \pi_3) &= [00 \quad 0 a_3 \quad 00 \quad 0 a_7 \quad 00 \quad 0 a_{11} \quad 00 \quad 0 a_{15}]
\end{aligned}
$$

$$
\begin{aligned}
\texttt{cMult}(B, \psi_0) &= [b_0 b_1 \quad b_2 b_3 \quad 00 \quad 00 \quad 00 \quad 00 \quad 00 \quad 00] \\
\texttt{cMult}(B, \psi_1) &= [00 \quad 00 \quad b_4 b_5 \quad b_6 b_7 \quad 00 \quad 00 \quad 00 \quad 00] \\
\texttt{cMult}(B, \psi_2) &= [00 \quad 00 \quad 00 \quad 00 \quad b_8 b_9 \quad b_{10} b_{11} \quad 00 \quad 00] \\
\texttt{cMult}(B, \psi_3) &= [00 \quad 00 \quad 00 \quad 00 \quad 00 \quad 00 \quad b_{12} b_{13} \quad b_{14} b_{15}]
\end{aligned}
$$

In contrast, other techniques in the field [Jia+18] that utilize diagonal-based packing for matrix multiplication incur significantly higher rotation costs due to the required transformations. These techniques perform poorly at a multiplicative depth of two. However, by employing some pre-generation at the expense of additional multiplicative depth, they can reduce the complexity to $\mathcal{O}(d)$ rotations for lower packing availability $\mathcal{O}(d^2)$; unfortunately, this advantage does not translate effectively to higher packing availabilities.

9.3 Proposed Matrix Multiplication Technique

In this work, we optimize the technique outlined in [RT22]. We still perform ciphertext-ciphertext multiplication using Equation 9.4 (as discussed in Section 9.2). However, what sets our approach apart is how we achieve the packing format required for Equation 9.4. Before presenting our technique for d^3 packing, we note a gap in the literature: no existing technique addresses matrix multiplication complexity using Equation 9.4 for d^2 packing. This requirement highlighted in [RT22] poses a limitation for medium- to large-sized matrices.

Furthermore, prior techniques are constrained to specific packing availabilities and do not generalize to accommodate arbitrary slot availability. This limitation restricts their applicability, particularly when the available packing varies with the matrix size across different layers. This flexibility is essential for effectively handling diverse matrix dimensions in practical applications.

For instance, in a CKKS ciphertext, the available packing ranges from $2^{13} - 2^{15}$ depending on the polynomial degree. Consequently, the technique from [RT22] can only be applied to matrices of dimensions up to 2^5. This limitation makes it unsuitable for processing larger datasets, such as high-resolution images of size 64×64 [ang23; Mav17] or 100×100 [Mav17]. In contrast, a technique that utilizes d^2 packing could facilitate fast matrix multiplications for data sizes up to 2^7. Therefore, we begin with a technique designed for d^2 packing and demonstrate how it can be extended to accommodate higher packing availability up to d^3.

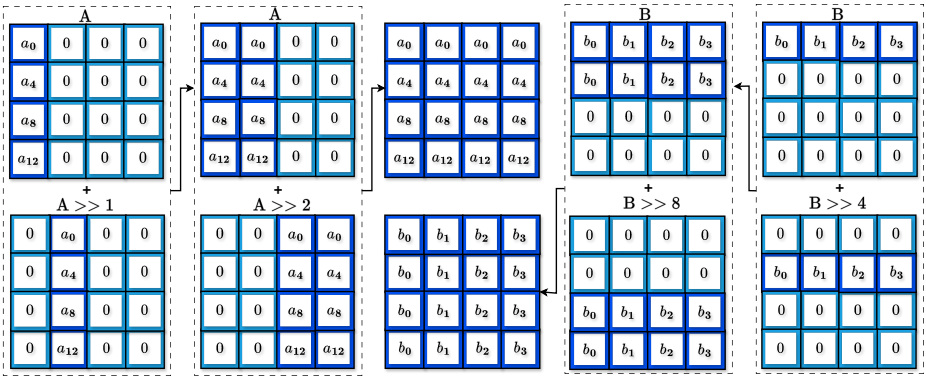


Figure 9.1: Matrix Column and Row duplication for A and B, respectively.

While packing more than d^3 is possible, it does not enhance the complexity of a single matrix multiplication.

9.3.1 Technique for d^2 packing

Each matrix consists of d^2 elements, which means that a ciphertext with d^2 packing capability can only pack a single matrix entirely. To ensure compatibility with previous works, we assume these ciphertexts are initially row-wise encoded and that d is a power of two.

Our proposed algorithm is outlined in Algorithm 26. We initiate the process of multiplication by splitting matrix A column-wise in Step 2, which enables more efficient handling of its elements. This is followed by immediate left-alignment in Step 3 to ensure that the data is properly structured for subsequent operations. Similarly, matrix B is split row-wise in Step 7 and left-aligned in Step 8, maintaining consistency in data organization. The duplication of these matrices necessitates $\log_2 d$ steps, as depicted in Figure 9.1. This duplication process occurs in Steps 4-5 for matrix A and Steps 9-10 for matrix B, allowing us to leverage the SIMD capabilities of the encryption scheme. Since we create a distinct ciphertext for each row/column split, we can perform multiplication and addition separately, as shown in Steps 11 and 12.

Since one ciphertext can only pack data from one row-wise or column-wise split, a post-multiplication transform is unnecessary, streamlining the process. Notably, no left alignment is required in the algorithms when $j = 0$. As a result, this technique demands $2{\cdot}d$ pt-ct multiplications for the necessary transformations, d ct-ct multiplications to combine the results, and $2 \cdot d(1 + \log_2 d) - 2$ rotations to align the data correctly for the final multiplication. This efficient use of resources reduces the overall computational complexity. These details are clearly summarized in Table 9.2.

Algorithm 26 `Matrix.Mult_d²`

In: $A, B \leftarrow \text{row_enc}(\mathtt{A}_{d\times d}, \mathtt{B}_{d\times d})$
Out: $C = \text{row_enc}(\mathtt{A}_{d\times d} \times \mathtt{B}_{d\times d})$

1: *// Preprocess A*
2: **for** $j = 0$ to $d-1$ **do**
3: $\quad \tilde{A}[j] \leftarrow \mathtt{cMult}(A, \pi_j)$ `/* Splitting A Cols */`
4: $\quad \tilde{A}[j] \leftarrow \mathtt{Rot}(\tilde{A}[j], j)$ `/* Right align all Cols */`
5: $\quad$ **for** $i = 0$ to $\log_2(d) - 1$ **do**
6: $\quad\quad \tilde{A}[j] += \mathtt{Rot}(\tilde{A}[j], -2^i)$ `/* Replicate Cols */`
7: $\quad$ **end for**
8: **end for**
9: *// Preprocess B*
10: **for** $j = 0$ to $d-1$ **do**
11: $\quad \tilde{B}[j] \leftarrow \mathtt{cMult}(B, \psi_j)$ `/* Splitting B rows Cols */`
12: $\quad \tilde{B}[j] \leftarrow \mathtt{Rot}(\tilde{B}[j], j \cdot d)$ `/* Top align all rows */`
13: $\quad$ **for** $i = 0$ to $\log_2(d) - 1$ **do**
14: $\quad\quad \tilde{B}[j] += \mathtt{Rot}(\tilde{B}[j], -2^i \cdot d)$ `/* Replicate the rows */`
15: $\quad$ **end for**
16: **end for**
17: *// Compute C*
18: **for** $j = 0$ to $d-1$ **do**
19: $\quad C += \mathtt{cMult}(\tilde{A}[j], \tilde{B}[j])$
20: **end for**

9.3.2 Technique for $2 \cdot d^2$ packing.

In the previous case, we focused on d^2 packing, where one ciphertext could only encode a single matrix. However, for smaller matrices, it is possible that more slots are available– specifically, $2 \cdot d^2$, which allows for packing two matrices within the same ciphertext. To accommodate this scenario, we modify the algorithm outlined in the previous section and describe it in Algorithm 27.

Instead of directly initiating row-wise or column-wise matrix decomposition, we propose packing two copies of the matrices within the same ciphertext. This critical step (Steps 1 and 7) is where our technique diverges from all prior works. The key innovation here is that rather than simply duplicating matrix A, the second ciphertext is shifted by one value to the left, as illustrated in Figure 9.2.

After performing multiplication with the plaintext mask, we achieve duplicated and left-aligned columns as required for the subsequent steps, eliminating the need for an additional rotation for left alignment. Thus, this adjustment allows us to obtain two columns with a single multiplication and rotation. Similarly, the B matrix is duplicated by keeping d values to the left, ensuring a seamless integration of the matrices. Consequently, this technique requires only d pt-ct multiplications, $\frac{d}{2}$ ct-ct multiplications, and $d \cdot (1 + \log_2 d) + 1$ rotations, as outlined in Table 9.2. A final rotation is necessary to accumulate the two partial results packed into the same ciphertext (Step 15).

Algorithm 27 `Optimized.Matrix.Mult_2 · d²`

In: $A, B \leftarrow \text{row_enc}(\mathtt{A}_{d\times d}, \mathtt{B}_{d\times d})$
Out: $C = \text{row_enc}(\mathtt{A}_{d\times d} \times \mathtt{B}_{d\times d})$

1: *// Preprocess A*
2: $A+ = \mathtt{Rot}(A, -d^2+1)$
3: **for** $j = 0$ to $(d/2)-1$ **do**
4: $\quad \tilde{A}[j] \leftarrow \mathtt{cMult}(A, \pi_j)$ `/* Splitting A cols */`
5: $\quad \tilde{A}[j] \leftarrow \mathtt{Rot}(\tilde{A}[j], 2j)$ `/* Right align all Cols Duplication */`
6: $\quad$ **for** $i = 0$ to $\log_2(d)-1$ **do**
7: $\quad\quad \tilde{A}[j]+ = \mathtt{Rot}(\tilde{A}[j], -2^i)$ `/* Replicate Cols */`
8: $\quad$ **end for**
9: **end for**
10: *// Preprocess B*
11: $B+ = \mathtt{Rot}(B, -d^2+d)$
12: **for** $j = 0$ to $(d/2)-1$ **do**
13: $\quad \tilde{B}[j] \leftarrow \mathtt{cMult}(B, \psi_j)$ `/* Splitting B rows */`
14: $\quad \tilde{B}[j] \leftarrow \mathtt{Rot}(\tilde{B}[j], 2\cdot j\cdot d)$ `/* Top align all rows */`
15: $\quad$ **for** $i = 0$ to $\log_2(d)-1$ **do**
16: $\quad\quad \tilde{B}[j]+ = \mathtt{Rot}(\tilde{B}[j], -2^i\cdot d)$ `/* Replicate the rows */`
17: $\quad$ **end for**
18: **end for**
19: *// Compute C*
20: **for** $j = 0$ to $(d/2)-1$ **do**
21: $\quad C+ = \mathtt{cMult}(\tilde{A}[j], \tilde{B}[j])$
22: **end for**
23: $C+ = \mathtt{Rot}(c, d^2)$

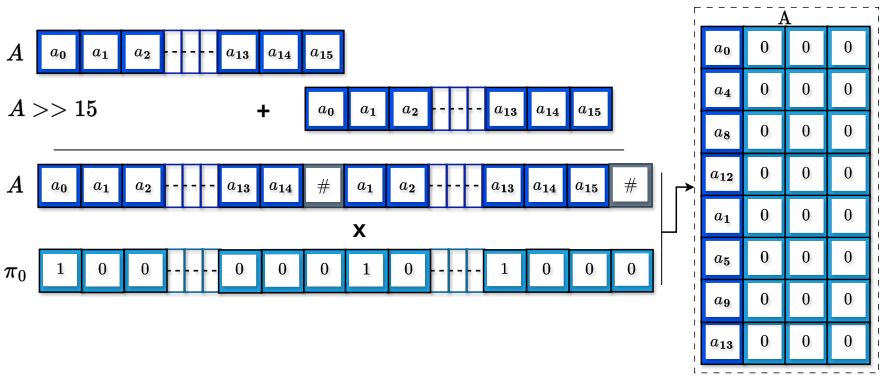


Figure 9.2: Matrix duplication and alignment done in one step using rotation. First, Matrix A is rotated and added to the original A. Then, a mask is multiplied to remove the unwanted elements for the upcoming column-wise duplication step.

Table 9.2: Complexity of the proposed secure d-dimensional matrix multiplications

Packing	# pt-ct Mult	# ct-ct Mult	# Rotations	# Depth†
d^2	$2 \cdot d$	d	$2 \cdot d(1 + \log_2 d) - 2$	2
$2 \cdot d^2$	d	$\frac{d}{2}$	$d(1 + \log_2 d) + 1$	2
$4 \cdot d^2$	$\frac{d}{2}$	$\frac{d}{4}$	$\frac{d}{2}(1 + \log_2 d) + 4$	2
d^3	2	1	$5 \cdot \log_2 d$	2
$d^{3\ddagger}$	1	1	$3 \cdot \log_2 d$	2

† This includes pt-ct and ct-ct multiplications, which consume the same depth in Libraries like OpenFHE [Al +22].
‡ If one of the matrix is unencrypted and can be defined n any form by the server.

9.3.3 Generalization to arbitrarily high packing complexity ($> d^2$)

If we estimate the cost for $4 \cdot d^2$ packing availability, we can initially pack four matrices per ciphertext. Hence, the required pt-ct and ct-ct multiplications, as well as the rotations for duplication and alignment, will decrease by a factor of four. This complexity is also outlined in Table 9.2. For generalization we observe that we can quantify the complexities based on the required steps and packing availability. Let s be the number of matrices that can be packed in a ciphertext. We can break down the complexity into five parts, as follows.

1. Matrix Duplication. This part consists of matrix duplication depending on s. The duplication of this type has logarithmic complexity, as illustrated in Figure 9.1. The cost of each matrix duplication is $\log_2 s$ rotations and additions. For example, when $s = 1$, no duplication is possible, resulting in a cost of 0. When $s = 2$, one duplication is possible per matrix, yielding a cost of 1. Therefore, the total cost for this step is $2 \log_2 s$ rotations and additions for both matrices.

2. RC Extraction. Once the matrices have been duplicated, their row/column-wise (RC) extraction requires plaintext multiplication with the appropriate masks, π_i and ψ_i. This extraction is performed $\frac{d}{s}\times$ per matrix, as duplication within the ciphertext offers s times more rows/columns per multiplication. Thus, the total pt-ct multiplication count amounts to $\frac{2 \cdot d}{s}$. While the first extraction does not require alignment, every subsequent extraction necessitates left alignment via rotations. Therefore, the total number of required rotations for alignment is $2(\frac{d}{s} - 1)$ for both the rows and columns of matrices A and B.

3. RC Duplication. After alignment, the rows and columns require duplication. Each duplication incurs a fixed cost of $\log_2 d$ rotations and additions. However, its frequency changes depending on the number of duplications needed, $\frac{2 \cdot d}{s}$. Thus, the total number of required rotations and additions for this step is $\frac{2 \cdot d}{s}(\log_2 d)$ for both row and column duplications.

4. Multiplication. We proceed to their multiplication once we have the row and column-wise duplicated matrices. This process is directly related to the number of ciphertexts that must be multiplied, resulting in $\frac{d}{s}$ ct-ct multiplications.

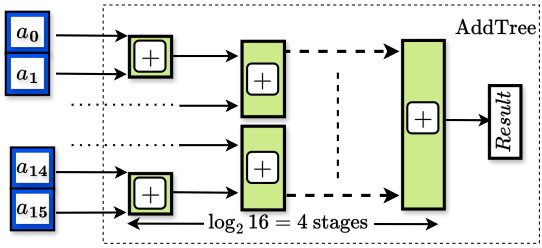


Figure 9.3: The adder tree depicting final result accumulation within a ciphertext. Every stage with t elements performs an addition (in SIMD) and results in $\frac{t}{2}$ resultant elements, which are then added in the next stage. This process continues until all the elements are accumulated.

5. Accumulation. Finally, we must accumulate the results packed within a ciphertext. This accumulation process is not an issue for d^2 packing since no matrix copies can be stored in a single ciphertext. However, accumulation becomes necessary for any higher packing availability. The accumulation is analogous to duplication but in the reverse direction, as depicted in Figure 9.3. It is important to note that elements cannot be accessed prior to rotation in an FHE ciphertext; hence, each $\log_2 s$ stage in the addition tree requires a rotation. Overall, the complexity of this stage is $\log_2 s$ rotations and additions.

Finally, we arrive at the following generalized operation requirements for matrix multiplication at depth two. The algorithm for this process is outlined in Algorithm 28. By substituting the appropriate value of s into the formulas below, we can derive the operation complexities listed in Table 9.2. As previously mentioned, Key Switching is only necessary following the ct-ct multiplications and rotations. Therefore, while pt-ct multiplication reduces the multiplicative depth, it does not significantly impact the overall time consumption.

- **# pt-ct Multiplications:** $\frac{2 \cdot d}{s}$
- **# ct-ct Multiplications:** $\frac{d}{s}$
- **# Rotations:** $\frac{2 \cdot d}{s}(\log_2 d) + 2(\frac{d}{s} - 1) + 3 \cdot \log_2 s$

The d^3 Packing Case.

For d^3 packing, we note that $s = d$, resulting in two required pt-ct multiplications and one ct-ct multiplication, yielding a constant time complexity. Additionally, the number of rotations required is $5 \log_2 d$, which is logarithmic in terms of the matrix dimension d. This performance surpasses all prior works, including those operating at multiplicative depth three [Jia+18].

In specific scenarios where the server has control over the model, and only one matrix is encrypted, this operation count can be further reduced to $3 \log_2 d$ rotations (as shown in Table 9.2). If we assume that the client can format the data as needed during the encoding step before encryption, the rotation operations

Algorithm 28 `Generalized.Matrix.Mult` (for arbitrary s matrix packing)

In: $A, B \leftarrow \text{row_enc}(\mathtt{A}_{d\times d}, \mathtt{B}_{d\times d})$
Out: $C = \text{row_enc}(\mathtt{A}_{d\times d} \times \mathtt{B}_{d\times d})$

1: *// Preprocess A*
2: **for** $i = 0$ to $\log_2 s - 1$ **do**
3: $\quad A+ = \mathtt{Rot}(A, -d^2 \cdot 2^i + 2^i)$ `/* Matrix Duplication */`
4: **end for**
5: **for** $j = 0$ to $\frac{d}{s} - 1$ **do**
6: $\quad \tilde{A}[j] \leftarrow \mathtt{cMult}(A, \pi_j)$ `/* Extracting` A `column-wise */`
7: $\quad \tilde{A}[j] \leftarrow \mathtt{Rot}(\tilde{A}[j], s \cdot j)$ `/* Left align all ciphertexts */`
8: $\quad$ **for** $i = 0$ to $\log_2(d) - 1$ **do**
9: $\quad\quad \tilde{A}[j]+ = \mathtt{Rot}(\tilde{A}[j], -2^i)$ `/* Column Duplication */`
10: $\quad$ **end for**
11: **end for**
12: *// Preprocess B*
13: **for** $i = 0$ to $\log_2 s - 1$ **do**
14: $\quad B+ = \mathtt{Rot}(B, -d^2 \cdot 2^i + d \cdot 2^i)$ `/* Matrix Duplication Duplication */`
15: **end for**
16: **for** $j = 0$ to $\frac{d}{s} - 1$ **do**
17: $\quad \tilde{B}[j] \leftarrow \mathtt{cMult}(B, \psi_j)$ `/* Extracting` B `row-wise */`
18: $\quad \tilde{B}[j] \leftarrow \mathtt{Rot}(\tilde{B}[j], s \cdot j \cdot d)$ `/* Left align all the ciphertexts */`
19: $\quad$ **for** $i = 0$ to $\log_2(d) - 1$ **do**
20: $\quad\quad \tilde{B}[j]+ = \mathtt{Rot}(\tilde{B}[j], -2^i \cdot d)$ `/* Row Duplication */`
21: $\quad$ **end for**
22: **end for**
23: *// Compute C*
24: **for** $j = 0$ to $\frac{d}{s} - 1$ **do**
25: $\quad C+ = \mathtt{cMult}(\tilde{A}[j], \tilde{B}[j])$ `/* Matrix Multiplication */`
26: **end for**
27: **for** $i = 0$ to $\log_2 s - 1$ **do**
28: $\quad C+ = \mathtt{Rot}(C, d^2 \cdot 2^i)$ `/* Accumulation*/`
29: **end for**

can be minimized to $\log_2 d$, and the multiplicative depth requirement becomes one. Therefore, the proposed technique is versatile and can be optimized for various scenarios.

A comparison of actual operation counts for encrypted matrix-matrix multiplication with prior works is illustrated in Figure 9.4, specifically for d^3 packing availability. It underscores the efficiency and effectiveness of our approach in enhancing secure matrix multiplication tasks within untrusted environments.

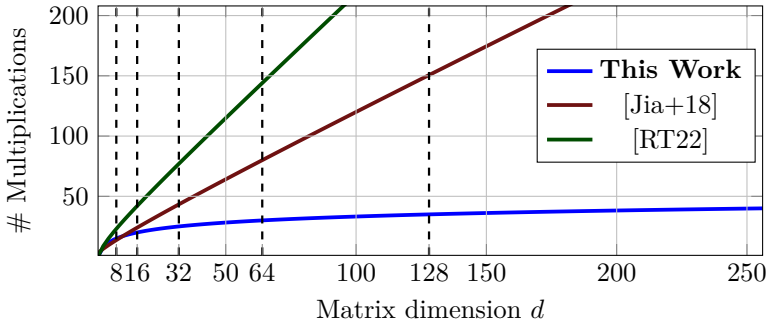


Figure 9.4: Graph comparing the rotation count reduction for d^3 packing.

9.4 Generalization to Rectangular Matrices

9.4.1 Padding based technique

The above technique can also be adapted for rectangular matrices of the form- $A_{l\times d} \cdot B_{d\times t}$. Previous works [RT22] employ zero-padding to the rows and columns of the matrices, transforming them into square matrices. This approach allows them to leverage techniques designed specifically for square matrices. We extend this idea to our method as well. By adding appropriate padding to the matrices A and B, we can ensure they fit the necessary dimensions for our algorithm.

This transformation can be incorporated into the application logic, ensuring that shifts consistently account for the factor d, as shown in Figure 9.5. Notably, most complexities remain unchanged except for row-wise and column duplication steps. Specifically, the columns of A will now need to be duplicated t times, while the rows of B require l duplications. Thus, the complexity for the RC Duplication step is adjusted to $\frac{d}{s}(\log_2 l + \log_2 t)$. This allows us to perform the same efficient operations while accommodating the rectangular structure of the matrices. The zero-padding process does not significantly impact the overall complexity, as it

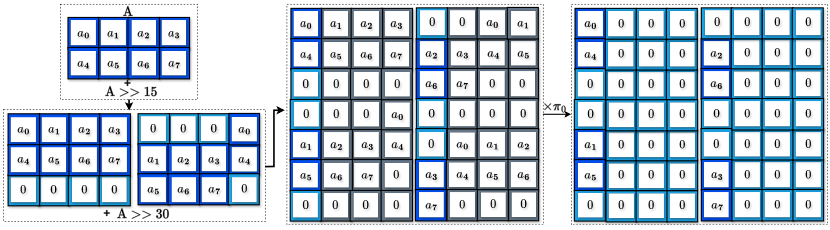


Figure 9.5: The $A_{2\times 4}$ matrix duplication transformation considering padding. The rectangular matrix undergoes two rotated accumulations to result in the middle ciphertext. This ciphertext behaves like a square matrix with zero padding. The grey elements are not needed and are removed via multiplication with the mask. After this step, column-wise duplication follows.

effectively maintains the logarithmic characteristics of our approach.

The technique discussed can also be generalized for rectangular matrices by adopting the approach from [RT22]. Specifically, when $d < t$, $t - d$ zero-padding columns are added to matrix A, and conversely, when $d > t$, zero-padding columns are appended to matrix B. This method is bounded by the naive technique that ensures both matrices become square, with dimensions $k \times k$, where $k = \max(l, d, t)$. As a result, the final rotation complexity for this approach is $5 \cdot \log_2 k$, necessitating k^3 packing.

9.4.2 Common Divide-and-Conquer Technique

The previous technique necessitates high packing availability, a requirement that becomes increasingly challenging when there is a significant difference between d and l or t. To address this, we explore an alternate matrix-splitting and divide-and-conquer approach that can be applied in all cases, irrespective of whether $l \neq d$ or $t \neq d$. This method is illustrated in Figure 9.6, showcasing two scenarios: one where $d < l, t$ and another where $d > l, t$.

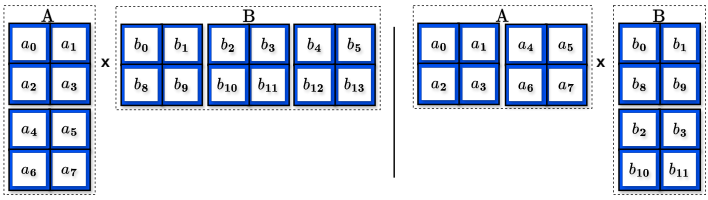


Figure 9.6: Rectangular matrices division for the divide-and-conquer technique.

The essence of this technique lies in dividing the larger matrix into smaller $d \times d$ matrices, which mitigates the need for excessive padding. For instance, in the first scenario (where $l = 4$, $d = 2$, and $t = 6$), instead of requiring 6^3 packings, we only need $6 \cdot 2^2$ slots per ciphertext. This reduces the overall packing requirement significantly. Additionally, the prior rotation complexity of $5 \cdot \log_2 6$ simplifies to $5 \cdot \log_2 2$, as no accumulation is necessary. In the second scenario (where $l, t = 2, d4$), instead of needing 4^3 packing, we again require only $4 \cdot 2^2$ slots per ciphertext. Here, the rotation requirement changes from $5 \cdot \log_2 4$ to $5 \cdot \log_2 2 + \log_2 2$, as one accumulation is required. Consequently, the overall packing requirement can be generalized as $k \cdot p^2$, where $k = \max(l, d, t)$ and $p = \min(l, d, t)$. This alternate approach reduces the need for extensive padding.

Next, the $p \times p$ matrix chunks are appended. These chunks need to be duplicated $\frac{t}{d}\times$ for A and $\frac{l}{p}\times$ for B. After the initial duplication, all the $p \times p$ matrices are packed in one ciphertext. Hence, $3 \cdot (\log_2 p)$ rotations are required for row-wise and column-wise duplication and result accumulation for a $p \times p$ matrix. $\log_2 \frac{d}{p}$ rotations are required for the final accumulation if d is greater than l, t (the second case in Figure 9.6). There is no increase in ct-ct multiplications, and the multiplicative depth stays at two. The rotation complexity for this approach can be expressed as: $(\frac{l}{p} + \frac{t}{p}) \log_2 p + (\log_2 \frac{t}{p} + \log_2 \frac{l}{p}) + 3 \cdot (\log_2 p) + \log_2 \frac{d}{p}$. The

breakdown of each component is as follows.

- *Initial Matrix Duplication.* The term $(\frac{l}{p} + \frac{t}{p}) \log_2 p$ accounts for the number of rotations required to duplicate the initial matrices. This arises from the need to replicate the smaller matrix chunks across the dimensions of l, t.
- *Duplication*$_{p\times p}$. The $p \times p$ matrix chunks need to be duplicated $\frac{t}{d}\times$ for matrix A and $\frac{l}{p}\times$ for matrix B. This contributes to the subsequent rotation complexity of $\log_2 \frac{t}{p} + \log_2 \frac{l}{p}$ operations.
- *Accumulation*$_{p\times p}$. Once the $p\times p$ matrices are packed into a single ciphertext, an additional $3 \cdot (\log_2 p)$ rotations are required for the row-wise and column-wise duplication and the accumulation of results for all the packed $p \times p$ matrix separately.
- *Final Accumulation.* The term $\log_2 \frac{d}{p}$ is included if d exceeds both l and t, indicating the rotations required for the final accumulation of results.

There is no increase in ct-ct multiplications throughout this process, and the multiplicative depth remains at two. This method effectively optimizes both the packing requirements and the rotation complexity.

9.5 Experimental Evaluation

We utilize the CKKS [Che+17] FHE scheme for our experimental evaluation. Prior works also adopt this scheme due to its ability to perform computations over approximate arithmetic, making it suitable for various applications, including Neural Networks. For our benchmarks, we employ the open-source OpenFHE [Al +22] library, ensuring compatibility with the setup provided by the FHERMA matrix multiplication challenge[1]. Our proposed matrix multiplication technique was tested in this challenge, where the available packing was $2 \cdot d^2$, yielding the best results[2]. Our artefacts for the general solution are available at[3].

We use a ring-degree of $N = 2^{16}$, which enables us to fully pack 32×32 matrices, duplicated 32 times. For our benchmarks, we evaluate matrix sizes ranging from 2×2 to 128×128 to demonstrate the scalability and efficiency of the technique. We observe that the proposed method is highly parallelizable. Therefore, we leverage the available parallelization using the '*pragma omp parallel*' routine. The runtimes reported in Table 9.3 are based on benchmarks executed on a 12th Gen Intelő Core i7-1260P processor with 16 threads and 32 GB RAM. We also provide benchmarks for the rectangular matrix case (Case 1), where padding is used. Additionally, the runtime for the divide-and-conquer approach can be extrapolated from the results reported for square matrices.

[1] `https://fherma.io/challenges/652bf669485c878710fd020b/overview`
[2] Winner of the competition.
[3] `https://github.com/aikata10/MatMul`

Table 9.3: Runtime evaluation of privacy-preserving matrix multiplication.

Matrix Dimension		Utilized Packing	Runtime	Slot Usage
A	B		(s)	
2×2	2×2	$d^3 = 2^3$	1.54	0.02%
2×4	4×4	$d^3 = 2^6$	1.79	0.20%
4×4	4×2	$d^3 = 2^6$	1.80	0.20%
4×4	4×4	$d^3 = 2^6$	1.94	0.20%
8×8	8×4	$d^3 = 2^9$	2.21	1.56%
8×8	8×8	$d^3 = 2^9$	2.31	1.56%
16×16	16×4	$d^3 = 2^{12}$	3.12	12.5%
16×16	16×16	$d^3 = 2^{12}$	3.23	12.5%
32×32	32×8	$d^3 = 2^{15}$	7.54	100%
32×32	32×32	$d^3 = 2^{15}$	7.88	100%
64×64	64×64	$8 \cdot d^2 = 2^{15}$	17.42	100%
128×128	128×128	$2 \cdot d^2 = 2^{15}$	157.1	100%

9.6 Discussion Packed Multiplications.

The proposed technique and results may raise the question: for an application requiring several simultaneous matrix multiplications (r), is it more efficient to duplicate a matrix within the same ciphertext or pack r distinct matrices for separate multiplications? In the case of d^2 packing, our technique necessitates $2 \cdot d(1 + \log_2 d - 2)$ rotations. If s packing slots are available, we could pack s distinct matrices into one ciphertext instead of duplicating the same matrix. This approach can be applied $\frac{r}{s}$ times if $r > s$. The ct-ct multiplication count remains d, and the number of rotations required for processing s packed matrix multiplications is $2 \cdot d(1 + \log_2 d) - 2$. However, packing s distinct matrices in one ciphertext requires $s - 1$ additional rotations and additions per matrix. Therefore, the total rotation requirement for $r > s$ becomes- $\frac{r}{s}(2 \cdot d(1 + \log_2 d) + 2 \cdot s - 4)$.

For $r \leq s$, the complexity simplifies to:

$$2 \cdot d(1 + \log_2 d) + 2 \cdot r - 4$$

In the prior work [RT22], similar to our technique for d^3 packing, the approach involves packing (or duplicating) the same matrix s times within a single ciphertext. The total number of rotations required to process $r \leq s$ matrix multiplications using this technique is as follows.

$$2 \cdot r \cdot d(1 + \log_2 d) + 2 \cdot r(\frac{d}{s} - 1) + 3 \cdot r \cdot \log_2 s$$

For the case where $r = s = 1$, both our proposed technique and the technique from [RT22] offer the same computational complexity. However, as r increases, the complexity of the latter technique grows much more rapidly. This makes our proposed technique for d^2 packing more suitable for scenarios where multiple matrix multiplications need to be performed simultaneously, as it lowers computational costs. The choice of technique ultimately depends on the available matrix packing capacity (s) and the required number of matrix multiplications (r). Users can make informed decisions that align with their specific application

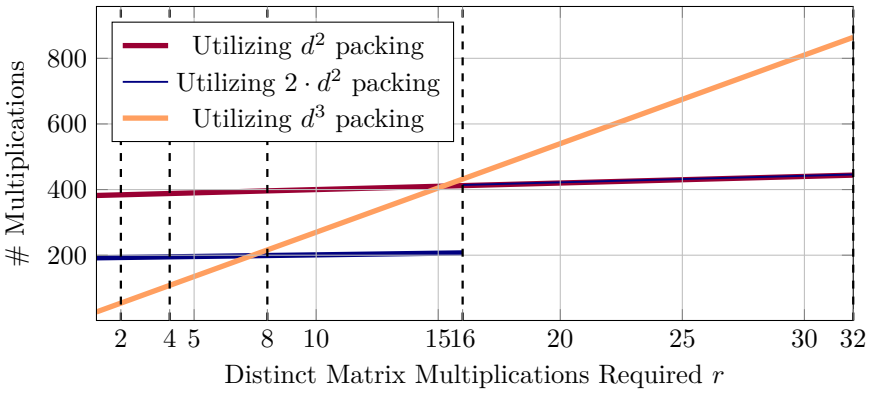


Figure 9.7: Graph comparing the rotation count for $d^3 (d = 32)$ packing available peer ciphertext with increasing distinct simultaneous matrix multiplication requirement r.

requirements by providing a detailed breakdown of complexity regarding these values.

In Figure 9.7, we illustrate the number of required rotations for the above two techniques for matrices of dimension $d = 32$ and packing availability $d^3 = 2^{15}$. When required r is 15 or higher, the former technique, utilizing d^2 packed computation with distinct matrix packing, results in a lower rotation requirement. However, the d^3 packing technique per ciphertext is better for lower simultaneous matrix multiplication requirements. A similar analysis can be done for more combinations of s and r with different packing utilization per matrix multiplication to make it comprehensive.

9.7 Conclusion

In this work, we introduced a novel technique for secure matrix multiplication using homomorphic encryption that significantly outperforms all previous approaches. By optimizing the required key-switch operation complexity from $\mathcal{O}(d)$ to $\mathcal{O}(\log_2 d)$ for square matrices of dimension d, our approach enhances computational efficiency. We also extended this technique to handle rectangular matrices by proposing two methods: one based on padding and another employing a divide-and-conquer strategy. These methods generalize our approach beyond square matrices, making it adaptable to arbitrary packing between d^2 and d^3. This adaptability is particularly beneficial in secure neural network applications, where the matrix dimensions and the available packing per matrix change at each layer. By thoroughly analyzing the complexities and trade-offs of these techniques, we offer insights that can guide future research in privacy-preserving computation. This work lays a strong foundation for future exploration in optimizing secure matrix operations using homomorphic encryption.

10 Conclusion

From Key Encapsulation to KeySwitch and Beyond ...

This thesis set out to answer a straightforward question: *to understand how cryptographic hardware could be made both secure and usable in the post-quantum era, and to help turn computation on encrypted data from a niche research result into a practical, deployable capability.*

At first glance, this appeared to be two separate problems. In practice, they were closely related: both demanded high-performance arithmetic on structured data, both had to run under tight resource constraints, and both faced realistic physical attackers who could observe or perturb implementations.

Two trends motivated this work. The first was the ongoing transition of secure communication toward post-quantum public-key cryptography. As classical public-key schemes became vulnerable to quantum adversaries, lattice-based constructions such as NIST-selected Key Encapsulation Mechanisms and Digital Signature Algorithms increasingly formed the foundation for future security protocols. However, deploying these schemes at scale required more than algorithmic confidence. It required implementations that were both efficient enough for real devices and robust against implementation attacks, such as side-channel leakage.

The second trend was the growing demand for privacy-preserving computation. Many modern services could benefit from performing computations on sensitive data without exposing it to the service provider. Fully Homomorphic Encryption offers an elegant answer in principle: it allows computation directly on encrypted data. In practice, however, the overhead in computation, memory traffic, and communication remained a major barrier. This overhead was not merely an inconvenience; it could determine whether an approach was usable at all.

Rather than treating post-quantum communication and encrypted computation as unrelated domains, this thesis viewed them as two ends of the same

hardware design spectrum: *post-quantum communication* on one side and *post-plaintext computation* on the other. This perspective emphasized that the core challenge was not only to implement cryptographic primitives, but to engineer *systems* around them. A fast design that leaked secrets, a secure design that was too slow to deploy, or an accelerated kernel that did not improve end-to-end performance could all fail in different but equally practical ways.

Consequently, the goal of this thesis was broader than acceleration alone. It was to develop architectural and implementation approaches that (i) delivered high performance for emerging post-quantum primitives, (ii) incorporated protection against practical attackers, and (iii) scaled from constrained platforms to high-performance settings. In parallel, it aimed to reduce the cost of encrypted computation through hardware methodologies and protocols, as well as application-level optimizations, so that privacy-preserving computation became a more realistic option in real-world workloads.

In summary, this thesis adopted a hardware-centric yet system-aware approach, examining how cryptographic designs behaved when implemented, how implementations behaved when attacked, and how both could be improved to ensure that post-quantum security and encrypted computation were not only theoretically sound but also practically usable.

10.1 Contributions at a glance

The contributions of this thesis began with efficient implementations of cryptographic primitives, progressed to implementation security, and then scaled up to system- and application-level acceleration.

1. Unified and compact PQC cryptoprocessor architectures.

For post-quantum secure communication, the thesis developed a unified methodology for implementing NIST-selected lattice-based primitives in hardware, including both KEMs and digital signatures, without treating each scheme as a separate design. The key outcome was a compact, reusable cryptoprocessor architecture that supported multiple standardized schemes and parameter sets. The design was optimized to reduce cycle count while keeping performance competitive across different platforms. In doing so, it supported the broader goal of *hardware agility* for PQC: a single architecture that could be updated to accommodate changes in schemes or parameters, rather than being tied to a single fixed design point.

2. Lightweight masking that keeps compact multipliers compact.

In hardware, security against implementation attacks is not a 'nice to have'-*it is a requirement.* At the same time, protecting a design could easily undo the very optimizations that made it practical. A recurring bottleneck in lattice-based implementations was that many efficient techniques (for example, exploiting small secret values) could not be extended to straightforward side-channel countermeasures. Naïve masking approaches often increased datapath width, added

substantial overhead, and ultimately made the compactness advantages of small-operand arithmetic obsolete.

This contribution addressed this gap by proposing lightweight masking techniques for polynomial arithmetic that preserved the benefits of compact computation. The proposed methods were designed to integrate naturally into polynomial multiplication and to remain compatible with NTT-based implementations using NTT-friendly moduli. Overall, the result was a practical, efficient technique for side-channel resistance, an important step if post-quantum cryptography were to be deployed on resource-constrained embedded platforms.

3. A chiplet-based design for FHE acceleration.

On the privacy-preserving computation side, accelerating FHE was not solely about increasing arithmetic throughput. End-to-end performance was often constrained by memory traffic, data layout, and available bandwidth, rather than by computation alone. Consequently, scaling an accelerator without addressing these system-level constraints could yield limited gains.

This contribution presented a chiplet-based methodology for FHE acceleration that treated scalability as a balanced design problem across compute, memory, and interconnect. The central idea was to utilize chiplet integration to explore and realize architectures, rather than designing one large chip, in order to reduce cost and time-to-market. The resulting framework provided a structured approach to evaluating trade-offs and mapping FHE workloads to feasible hardware configurations, aiming to significantly reduce the performance gap between homomorphic and plaintext execution.

4. HHE in hardware and the security reality check.

The thesis next investigated the Hybrid Homomorphic Encryption paradigm, which combined efficient symmetric encryption on the client with homomorphic decryption on the server, thereby reducing two major costs that commonly limited FHE-based systems: the client-side encryption workload and the communication overhead associated with large ciphertexts. This contribution provided a systematic hardware study of HHE-oriented designs, including concrete implementations and an evaluation of the resulting performance trade-offs. It further addressed implementation security by analyzing power side-channel leakage in these integer-based primitives and developing masking techniques tailored to the HHE setting.

5. SASTA-DFA: fault attacks on the HHE paradigm.

With this contribution, the security narrative deepened by moving beyond passive leakage to active manipulation. In the HHE setting, this thesis presented the SASTA-DFA approach and showed that protocol robustness remains a critical consideration when deploying hybrid protocols. The core result was that, by injecting a single fault during symmetric encryption, an adversary could achieve *full key recovery*. Notably, the attack did not rely on assumptions that are often central to classic DFAs on symmetric primitives, such as nonce reuse; instead, it exploited the structure introduced by the surrounding protocol and the way its

operations are composed. By treating the *protocol* itself as the attack surface, rather than viewing symmetric encryption in isolation, this work demonstrated, *for the first time in the context of HHE*, how vulnerabilities can arise even when the individual building blocks are standard and well understood.

6. Pushing the limits of Client-side acceleration with vanilla CKKS. This thesis also pushed the limits of client-side acceleration using *vanilla* CKKS, that is, without relying on symmetric-encryption-based HHE protocols. In this setting, the proposed Deflate/Boost techniques reduced both client computation and communication overhead by changing *when* and *how* CKKS ciphertext components were formed and transmitted. The central idea was to *deflate* the client workload and ciphertext footprint by transmitting the bare minimal compact ciphertext, and then *boost* it back into a standard CKKS ciphertext form on the server side using inexpensive reconstruction steps (Half-Bootstrapping). This contribution highlighted that client-side costs are often driven by architectural and algorithmic choices, and that substantial gains can be achieved through careful representation, more efficient routines, and hardware/software co-design, while preserving the standard CKKS workflow and security model.

7. Application-layer optimization: SIMD-aware matrix multiplication. Finally, this thesis moved to the application layer by optimizing privacy-preserving matrix multiplication, a central performance bottleneck in encrypted machine learning. The key idea was to redesign SIMD packing so that the computation made more effective use of the ciphertext slots and the linear operations they support. With this packing strategy, the number of expensive KeySwitch operations was substantially reduced, shifting from linear complexity in the matrix dimension to logarithmic complexity in the same dimension, without increasing the multiplicative depth. The approach was also extended to support broader packing granularities, and rectangular matrices. Overall, this contribution demonstrated that significant performance gains in FHE do not solely result from faster primitives, but also from utilizing those primitives in a more workload-aware manner.

10.2 Open Problems and Future Directions

Having established the contributions, it is natural to ask what comes next. The directions below are organized around two pillars: post-quantum communication and post-quantum computation on encrypted data.

10.2.1 PQ-secure communication beyond KaLi, Kavach

A natural extension of the KaLi post-quantum hardware suite is to expand its scheme coverage by incorporating hardware implementations of additional NIST-standardized candidates, such as Falcon [Por19; Smo+26] (for digital signatures) and HQC [Gab+25] (for key encapsulation mechanisms). The next important step

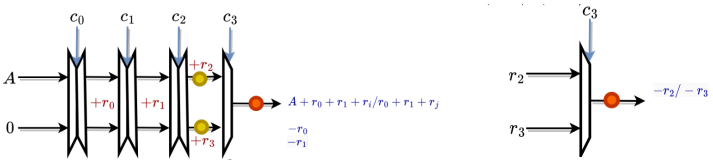


Figure 10.1: Kavach masking for higher orders (d=4).

would be to examine masking and implementation hardening at the suite level, rather than treating each scheme in isolation. A unified evaluation framework would allow for consistent design choices, such as randomness generation, masking order, leakage and fault models, and measurement methodology, to be reused across different schemes. This approach would facilitate an apples-to-apples comparison of costs and help identify how a shared countermeasure strategy can lead to reduced area consumption.

In parallel, our Kavach work indicates a clear, practical next step: ensuring that masked gadget construction scales efficiently with the masking order. A straightforward extension would involve replicating the boolean selector gadget as the masking order increases [Aik+23a], which could result in quadratically complex growth- $\mathcal{O}(d^2)$, quickly becoming expensive in hardware. However, by using a more thoughtful order extension, reusing intermediate structures, introducing randomness only when necessary for security, and adapting the selection mechanism as the masking order grows, the same security goal can be achieved with significantly better scaling compared to the naive approach. Figure 10.1 illustrates one such extension strategy. An important direction for future work is to formalize these design principles, and implement and test higher-order variants of Kavach that remain both secure and efficient in implementation.

10.2.2 From Protocols to Applications: The Last Mile

For computation on encrypted data, several directions remain both natural and urgent. One important avenue is *HHE protocol hardening*, including authenticated variants. While HHE remains interesting because it shifts computation cost away from the client, it can also introduce protocol structures that are particularly sensitive to fault attacks. This leaves room for protocol-level defenses and practical mitigation techniques that preserve the benefits of the hybrid paradigm.

A second avenue is *bridging the last mile from kernels to applications*. The matrix multiplication work reinforces that end-to-end performance can shift significantly when data layouts and packing are optimized. A major opportunity is therefore to move these insights into compilers, runtimes, and libraries, enabling hardware accelerators and application-level scheduling to cooperate rather than operate in isolation. Further generalization toward compiler-automated packing [Gar+25] and scheduling would make these optimizations broadly usable, ensuring that applications benefit without requiring users to become FHE experts.

11
Epilogue

"The end is where we start from." T. S. Eliot

If there is one thread running through this thesis, it is that secure systems succeed or fail at the boundaries, where computation meets communication, and where theory meets implementation. Communication and computation play complementary roles and must be designed together: optimizing one while ignoring the other only relocates the bottleneck, and can just as easily relocate the vulnerability. Efficient algorithms and efficient hardware are therefore inherently end-to-end concerns- "in theory, theory and practice are the same; in practice, they are not".

The post-quantum era raises the stakes for secure communication, while computation on encrypted data expands what privacy can mean in deployed services. Hardware sits at the center of both: it is where performance becomes real, where constraints become design, and where attackers often find their most practical leverage. By combining new architectures and techniques with careful attacks and defenses, this thesis takes a step towards making post-quantum communication and encrypted computation not only possible, but practical. As a final note, the author shares with her dear readers a ghazal she has carried with her throughout her PhD journey:

"Sitaron Se Agay Jahan Aur Bhi Hain,
Tu Shaheen Hai, Parwaz Hai Kaam Tera
Tere Samne Asman Aur Bhi Hain,
Issi Roz-o-Shab Mein Ulajh Kar Na Reh Ja
Ke Tere Zaman-o-Makan Aur Bhi Hain"
Iqbal

(Meaning: Do not limit yourself and always strive for higher goals and deeper horizons.)

A Appendix

A.1 Task Distribution for KeySwitch

Algorithm 29 `CKKS.KeySwitch` [HK19; KPP22] (for arbitrary $dnum$)

In: $\mathtt{d} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1, \tilde{\mathbf{d}}_2) \in R^3_{Q_l}$, $\tilde{\mathtt{ksk}}_0 \in R^{dnum}_{PQ_l}, \tilde{\mathtt{ksk}}_1 \in R^{dnum}_{PQ_l}$
Out: $\mathtt{d}' = (\tilde{\mathbf{d}}'_0, \tilde{\mathbf{d}}'_1) \in R^2_{Q_l}$

for $j = 0$ to $dnum - 1$ **do**
  $\tilde{\mathbf{y}}[j] \leftarrow \tilde{d}_2[j \cdot K : (j+1) \cdot K - 1] \cup \mathtt{BConvRout}_{Q_K \to PQ_{l/K}}(\tilde{d}_2[j \cdot K : (j+1) \cdot K - 1])$
  $(\tilde{\mathbf{c}}''_0[j], \tilde{\mathbf{c}}''_1[j]) \leftarrow 0$
  for $i = 0$ to $l + K$ **do**
    $\tilde{c}''_0[j][i] \leftarrow \left[\tilde{c}''_0[j][i] + \tilde{\mathtt{ksk}}_0[j][i] \cdot \tilde{y}[j][i]\right]_{q_j}$
    $\tilde{c}''_1[j][i] \leftarrow \left[\tilde{c}''_1[j][i] + \tilde{\mathtt{ksk}}_1[j][i] \cdot \tilde{y}[j][i]\right]_{q_j}$
  end for
end for
for $j = 1$ to $dnum - 1$ **do**
  $\tilde{\mathbf{c}}''_0[0] \leftarrow \left[\tilde{\mathbf{c}}''_0[0] + \tilde{\mathbf{c}}''_0[j]\right]$
  $\tilde{\mathbf{c}}''_1[0] \leftarrow \left[\tilde{\mathbf{c}}''_1[0] + \tilde{\mathbf{c}}''_0[j]\right]$
end for
$\tilde{\mathbf{d}}'_0 \leftarrow \tilde{\mathbf{d}}_0 + (\tilde{\mathbf{c}}''_0[0]_{Q_l} - \mathtt{BConvRout}_{P \to Q_l}(\tilde{\mathbf{c}}''_0[0]))$
$\tilde{\mathbf{d}}'_1 \leftarrow \tilde{\mathbf{d}}_1 + (\tilde{\mathbf{c}}''_1[0]_{Q_l} - \mathtt{BConvRout}_{P \to Q_l}(\tilde{\mathbf{c}}''_1[0]))$

The generic version of the KeySwitch routine [HK19; KPP22] is presented in Algorithm 29. The $L + 1$ limbs of the ciphertext are split into $dnum$ digits of K limbs. Every digit consisting of K limbs is used to obtain $l + K + 1$ limbs after the ModUp operation (via BConvRout Algorithm 30) for ciphertext at depth l.

Algorithm 30 $\texttt{CKKS.BconvRout}_{P \to Q_l}$

In: $\tilde{\mathbf{y}}_2 \in R_P$
Out: $\tilde{\mathbf{y}} \in R_{Q_l}$

1: **for** $j = 0$ to $K - 1$ **do**
2: $\quad y_2[j]_{p_j} \leftarrow \texttt{INTT}(\tilde{y}_2[j])$
3: **end for**
4: **for** $j = 0$ to l **do**
5: $\quad y[j]_{q_j} \leftarrow \sum_{i=0}^{K-1} (y_2[i] \cdot \hat{p}_i^{-1} \bmod p_i) \cdot \hat{p}_i \bmod q_j$ ▷ BaseConversion
6: $\quad \tilde{y}[j]_{q_j} \leftarrow \texttt{NTT}(y[j+K])_{q_j}$
7: **end for**

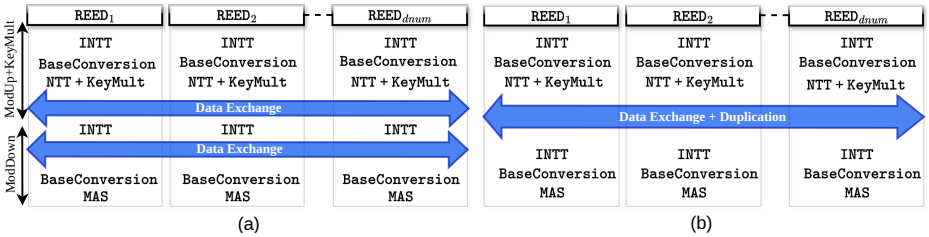


Figure A.1: The Data and Task distribution when limbs are distributed digit-wise for KeySwitch computation. The INTT, NTT, and BaseConversion mentioned here refer to computations required per digit. In the first case, Data Exchange is done in parallel with BaseConversion, and in the second case, it is done in parallel with KeyMult. This does not produce a full decoupling effect when C2C communication is slower.

This results in $dnum \cdot (l + K + 1)$ limbs, which are then multiplied with the keys and accumulated to return $(l + K + 1)$ limbs per ciphertext component. Finally, a ModDown is done to reduce the limbs to $(l + 1)$ limbs and the operation flow is very similar to that of ModUp. As discussed in Section 5.5.2, maintaining the limb-based decomposition proves advantageous compared to switching between limb-based and coefficient-based methods. We now explore how this limb-based decomposition applies when $dnum < L + 1$. It is important to note that for $dnum < L + 1$, NTT computation cannot start until the INTT results are multiplied with base hats $(\hat{p}_i, \hat{p}_i^{-1})$ for BaseConversion.

The first technique involves digit-wise distributing data and tasks across limbs, where the number of chiplets equals $dnum$. This allows each chiplet to independently execute the outer loop for ModUp in Algorithm 29 without requiring inter-chiplet data exchange. Data is only shared after ModUp (before Key Multiplication) and during ModDown, as illustrated in Figure A.1 first figure. The authors in [Kim+22a], note that this has an overhead of $2 \cdot dnum \cdot (K + l + 1)$ polynomials per chiplet, which can be reduced to $\frac{2 \cdot (dnum-1) \cdot (l+K+1)}{dnum}$ for ModUp as explained in Section 5.5.2. During ModDown, $2 \cdot K$ polynomials would be communicated for BaseConversion. ModDown can also be handled within each chiplet, eliminating the need for cross-chiplet data exchange if the result of Key Multiplication is duplicated across all chiplets, as depicted in the

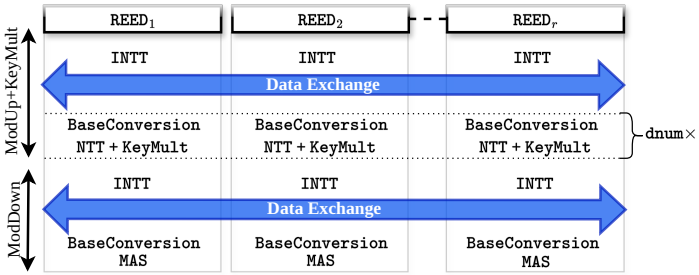


Figure A.2: The Data and Task distribution when limbs are distributed alternately for KeySwitch computation. The Data Exchange is done in parallel with INTT, BaseConversion, and NTT+KeyMult of previous digits.

second figure of Figure A.1. This results in a one-time communication cost of $\frac{2\cdot(dnum-1)\cdot(l+1)}{dnum} + 2 \cdot K$ polynomials per chiplet and requires additional storage for $2 \cdot (K-1)$ polynomials in each chiplet.

Each chiplet would have to perform duplicate K INTT operations and BaseConversion steps during ModDown while the $(l+1)$ NTT operations would be distributed across the chiplets. A key limitation of this technique is that not much computation can be done in parallel with the communication after ModUp. While the polynomial transfers can overlap with the NTT and Key Multiplication computations, the high throughput of NTT+KeyMult operations means this overlap does not achieve the desired decoupling effect. As a result, the potential for parallelism is limited, leading to delays. Additionally, this approach demands many chiplets for higher values of *dnum*. As the digits for the key switch decrease, many chiplets become idle, further reducing efficiency. This inefficiency in fully utilizing chiplets highlights a bottleneck in achieving optimal performance.

In contrast, the alternate limb distribution technique distributes the task for each digit across a fixed number of chiplets, as illustrated in Figure A.2. This means communication is only required during ModUp/ModDown, after which all the data needed for Key Multiplication and accumulation remains confined within a single chiplet. During ModUp, the chiplets broadcast the INTT results (similar to the case of $dnum = L+1$), which is efficient as only $l+1$ polynomials undergo INTT conversion. After ModUp, the number of polynomials increases to $dnum \cdot (l+K+1)$. Although communication is necessary for both ModUp and ModDown, our technique minimizes communication overhead since the number of polynomials broadcasted during ModUp $(l+1)$ and ModDown $(2 \cdot K)$ is significantly lower compared to the previous method's $\frac{2\cdot(dnum-1)\cdot(l+1)}{dnum} + 2 \cdot K$ for $dnum > 2$.

We investigated the possibility of using data duplication to decouple INTT data transfer from NTT computation. However, we found that the overhead caused by duplicating data after each operation outweighed any potential communication benefits during ModUp and ModDown. Thus, we converge to an approach where the chiplets do not compute only K INTT required for each digit

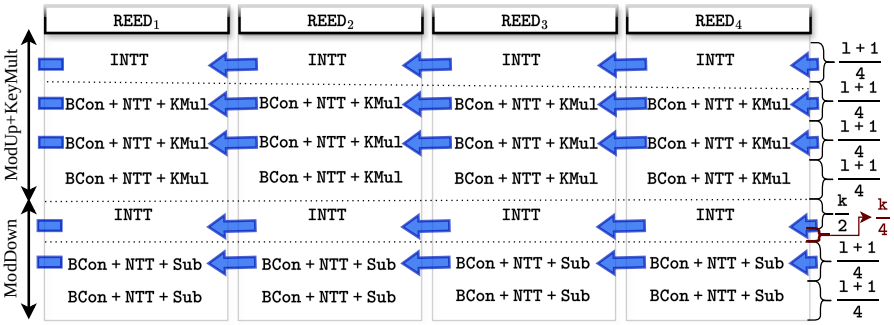


Figure A.3: KeySwitch Flow showing computation throughput and communication overhead for ModUp, KeyMult and ModDown for $dnum = 3$.

computation during ModUp but all $l + 1$ INTT operations, which they would require for computation corresponding to all the digits.

A.1.1 Throughput Computation of KeySwitch

In this section, we discuss how the throughput is obtained when $dnum = 3$, in line with Section 5.4.6, in the four $(= r)$ chiplet setting of REED. As discussed above we, perform all $l+1$ INTT at once, to bridge the computation communication gap. Thus, each chiplet performs at most $\frac{l+1}{r}$ INTT. For the first set of BaseConversion followed by Key Multiplication, only $\frac{l+1}{dnum}$ INTT results are needed. Since, these limbs are distributed across chiplets a communication of $\frac{(r-1)\cdot(l+1)}{dnum\cdot r}$ is required to every chiplet. This communication also works in ring manner, similar to the strategy proposed in Section 5.5.4.

Note that we cannot start NTT before we have accumulated all the $\frac{l+1}{dnum}$ results as required for Algorithm 30. When $dnum = L + 1$, this value is 1, hence we can start the NTT immediately on the INTT result, however, the same cannot be done for lower values of $dnum$. Thus, in the prior case, while a long communication window was available, here the communication window is limited. The first set of $\frac{(r-1)\cdot(l+1)}{dnum\cdot r}$ INTT results have to be communicated while the chiplet is computing $\frac{l+1}{r}$ INTTs. Each BaseConversion computes $l + 1$ NTTs ($\frac{l+1}{r}$ NTTs), which gives $\frac{dnum}{r-1}\times$ larger communication window for the INTT results required for subsequent BaseConversions. Note that, for $dnum = 3$ and $r = 4$, its value is one, hence we only face computation overhead for the last polynomial as the C2C communication offers the same throughput as the configuration 1024×64. However, when the C2C communication becomes slower, than this directly impacts the communication and hence the delay before the BaseConversions.

Now, the goal of computation is to prevent computation from becoming the bottleneck in this case. Thus, instead of sending plain INTT results, the results are multiplied with $\hat{p}_i^{-1}$ for Step 5 [Kim+22a], in Algorithm 30. After

this the chiplets only need to perform a MAS operation on all the INTT results for each new base during BaseConversion. We are only restricted by off-chip communication bandwidth, and therefore we wait until all the polynomials have been shared. The polynomials in INTT form are in on-chip memory. The memory for each polynomial is split across various SRAMs. Therefore, one polynomial can be loaded at a higher throughput. The operations on these can be handled by an extra MAS unit for BaseConversion using $K\times$ ($K = 8$ for $dnum = 3$) more multipliers to offer the same throughput as NTT. Thus, the BaseConversion, followed by NTT, and Key Multiplication and accumulation works in a pipeline. The key multiplication with the pre-existing K bases is done in parallel to their INTT conversion, as it does not require any pre-computation. Hence, the overhead of the ModUp+KeyMul is directly correlated to the computation time of the INTT polynomials $\frac{(l+1)}{r}$ and the $dnum$ BaseConversions $\frac{dnum\cdot(l+1)}{r}$, as shown in Figure A.3.

While, the communication could be completely decoupled for the ModUp+KeyMul step, the decoupling reduces for ModDown. Each chiplet computes the $\frac{K}{r}$ INTT results and requires communication of $\frac{(r-1)\cdot K}{r}$ INTT results before it can perform the BaseConversion requiring $\frac{(l+1)}{r}$ NTT computations per chiplet and the subtraction in pipeline. Thus, the overall overhead of this step is determined by communication of $\frac{(r-1)\cdot K}{r}$ polynomials and followed by $\frac{(l+1)}{r}$ NTT computation. This is incurred twice, once for each ciphertext component. Instead of doing the ModDown sequentially for each ciphertext component, we follow the technique proposed earlier and perform all the INTT $\frac{2\cdot K}{r}$ INTT computations at once. Thus the communication overhead for the first ModDown is reduced to $\frac{(r-3)\cdot K}{r}$. While the first ModDown is being computed the limbs for the next ModDown are communicated. Thus, the delay due to communication overhead of $\frac{(r-3)\cdot K}{r}$ is incurred only once, and the computation overhead is $\frac{2\cdot(l+1+K)}{r}$ NTT/INTT computations.

Overall, this results in runtime complexity of $\frac{(2\cdot(l+1+K)+(dnum+1)\cdot(l+1)+(r-3)\cdot K)}{r}$. With $dnum = 3$ and $r = 4$, the communication overhead is $\approx$5%, while INTT/NTT is being computed in parallel and MAS operations in pipeline. Substituting the values of $dnum = L + 1$, shows how this reduces to bare minimum. Thus, NTT unit stays almost fully utilized ($\approx$95%) during the hybrid key-switch for $dnum = 3$. We have kept our analysis generic, so that similar results can be derived for varying values of $dnum$ and r. This also shows how $r = 4$, gives a sweet spot, and a higher value would result in higher communication delay, not only during ModDown but also during ModUp, for lower values of $dnum$. Note that, this 5% communication delay can be completely bridged by interleaving the last BaseConversion+NTT+KeyMul step with the INTT required for ModDown, as no communication happens during this step. However, we leave this dataflow utilization for future works.

A.2 The IND-CPA^{D} Security Overview

The IND-CPA^{D} (Indistinguishability under Chosen Plaintext Attacks with Decryption oracle) security model was introduced by Li and Micciancio [LM21]. In an FHE scheme $\mathsf{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$, if an adversary knows m, f, and $c_{\text{FHE}} = \mathsf{Enc}(m)$, then access to $\mathsf{Dec}(\mathsf{Eval}(f, c_{\text{FHE}}))$ seems harmless, as they can compute $f(m)$ directly. Since FHE ensures $\mathsf{Dec}(\mathsf{Eval}(f, Enc(m))) = f(m)$ This suggests INS-CPA^{D} security follows from IND-CPA security, as shown in Figure A.4. However, Li and Micciancio [LM21] disproved this for approximate FHE schemes. Their attack exploits the fact that given a ciphertext $(c_{\text{FHE}} = (-a \cdot s + e + m, a))$ after homomorphic encryption of a message m for secret polynomial s and public polynomial a, the decrypted result becomes $m + e$. If an attacker, posing as the client, obtains the result of encryption c_{FHE} and decryption $(m + e)$, she can easily recover the secret s since polynomial a is public. This attack notion was further extended to exact FHE schemes by several follow-up works [Che+24a; Che+24b].

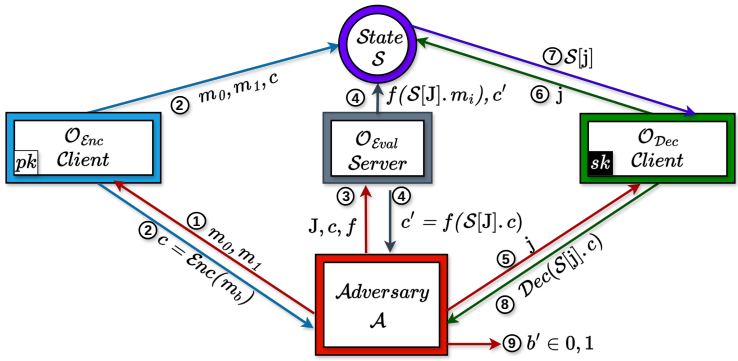


Figure A.4: The IND-CPA^{D} threat model for FHE as introduced in [LM21].

Definition (IND-CPA^{D} Security Game). Let $\prod = (\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec}, \mathrm{Eval})$ be an FHE scheme with plaintext space $\mathcal{M}$ and ciphertext space $\mathcal{C}$. For an experiment parameterized by a bit $b \in \{0, 1\}$ and an adversary $\mathcal{A}$ that is given access to the following oracles $\mathcal{O}_{Enc}$ (Encryption Oracle), $\mathcal{O}_{Eval}$ (Evaluation Oracle), and $\mathcal{O}_{Dec}$ (Decryption Oracle) sharing a common state $S \in (\mathcal{M} \times \mathcal{M} \times \mathcal{C}^*)$ consisting of message-message-ciphertext triplet sequence:

- Enc: $\mathcal{A}$ submits a pair of plaintext messages (m_0, m_1), to $\mathcal{O}_{Enc}$ which randomly chooses a secret bit b (0 or 1), and computes $c_{\text{FHE}} \leftarrow \mathrm{Enc}_{\mathrm{pk}}(m_b)$ and extends the state $S = [S; (m_0, m_1, c_{\text{FHE}})]$; and returns c_{FHE} to the adversary.
- Eval: $\mathcal{A}$ gives a function $f : \mathcal{M}^k \to \mathcal{M}$ and a sequence of indices $J = (j_1, \cdots, j_k) \in \{1, \cdots, |S|\}^k$ to $\mathcal{O}_{Eval}$ which computes the ciphertext $c_{\text{FHE}} \leftarrow \mathrm{Eval}_{\mathrm{pk}}(f, S[j_1] \cdot c_{\text{FHE}}, \cdots, S[j_k] \cdot c_{\text{FHE}})$, where $S[l_i] \cdot v$ is the element $v \in$

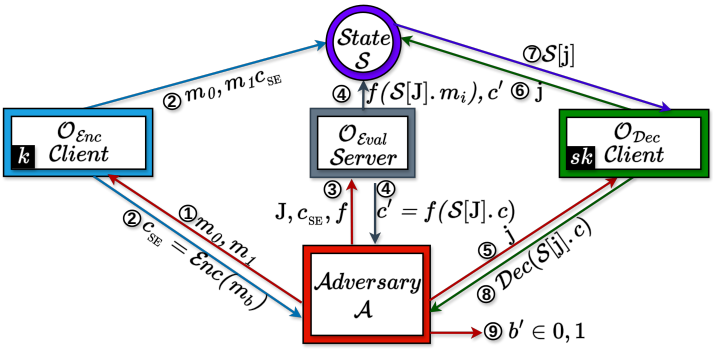


Figure A.5: The IND-CPA^{D} threat model [LM21] for HHE.

$\{m_0, m_1, c_{\text{FHE}}\}$ of the state S for the l_i-th request. Then, it extends the state with one more triplet $S := [S; (f(S[j_1]{\cdot}m_0, \cdots, S[j_k]{\cdot}m_0), f(S[j_1]{\cdot}m_1, \cdots, S[j_k] \cdot m_1), c_{\text{FHE}})]$. Ciphertext c_{FHE} is returned to the adversary.

- Dec: $\mathcal{A}$ gives an index $j \leq |S|$ to $\mathcal{O}_{Dec}$ who checks whether $S[j] \cdot m_0 = S[j] \cdot m_1$, and, if so, returns $\mathsf{Dec}_{\text{sk}}(S[j] \cdot c_{\text{FHE}})$, otherwise $\perp$ to the adversary.
- Guessing Stage: After multiple such queries, $\mathcal{A}$ tries to guess b. If correct, $\mathcal{A}$ wins; otherwise, she loses.

It should be noted –

- FHE schemes lack IND-CCA security because a decryption query with a malformed homomorphic ciphertext can reveal the secret key. To avoid this trivial case, assumptions are imposed on the queries. Specifically, the decryption oracle only accepts ciphertexts generated by the encryption or evaluation oracles. In other words, ciphertexts must either result directly from an encryption operation or be obtained through homomorphic evaluation of previously encrypted values. Consequently, only properly formed ciphertexts produced by valid homomorphic operations are permitted. This limitation prevents the adversary from submitting arbitrary ciphertexts for decryption, ensuring that only those created within the game can be queried.
- In this game, the adversary has control over the homomorphic computations performed, as they can specify the function f in the evaluation request.

A.2.1 IND-CPA^{D} Security for HHE

In the context of HHE, $\mathcal{O}_{Enc}$ performs symmetric encryption, and $\mathcal{O}_{Eval}$ first computes homomorphic symmetric decryption and then evaluates f. The use of nonce for the new and efficient stream ciphers (or CBC/Counter mode for AES) ensures every encryption result is distinct, even for the same ciphertext. Thus, the following changes are applied to the above IND-CPA^{D} security game, as shown in Fig. A.5.

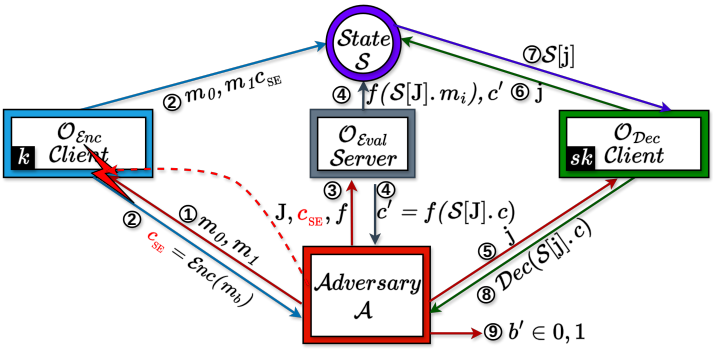


Figure A.6: The adaptation of IND-CPA$^{\mathrm{D}}$ threat model [LM21] for HHE to the SASTA model, which extends it by incorporating fault injection capability.

- Enc: $\mathcal{A}$ submits a pair of plaintext messages (m_0, m_1), to $\mathcal{O}_{Enc}$ which randomly chooses a secret bit b (0 or 1), and computes $c_{\mathrm{SE}} \leftarrow m_b + \mathtt{E}_\pi(K, n_i)$ and extends the state $S = [S; (m_0, m_1, c_{\mathrm{SE}})]$; and returns c_{SE} to the adversary.
- Eval: $\mathcal{A}$ gives a function $f : \mathcal{M}^k \rightarrow \mathcal{M}$ and a sequence of indices $J = (j_1, \cdots, j_k) \in \{1, \cdots, |S|\}^k$ to $\mathcal{O}_{Eval}$ which first computes homomorphic symmetric decryption on c_{SE} to obtain c_{FHE}, and the rest of the steps follow.

A.2.2 IND-CPA$^{\mathrm{D}}$ Security for HHE under Faults

While prior works [Che+24a; Che+24b; LM21] could attack FHE schemes under the IND-CPA$^{\mathrm{D}}$ security notion, these attacks can be mitigated using noise flooding or rounding [LM21]. Thus, in the setting where the IND-CPA$^{\mathrm{D}}$ attacks are mitigated (or for the FHE schemes [Chi+20] unexploitable under this model), we explore if we can mount a key-recovery attack by inducing fault in the encryption routine in the context of HHE. Notably, we restrict this to a single faulty query. Thus, the changes to the existing definition (Appendix A.2.1) are as follows (Fig. A.6).

- Enc: $\mathcal{A}$ submits a pair of plaintext messages (m_0, m_1), to $\mathcal{O}_{Enc}$ which randomly chooses a secret bit b (0 or 1), and computes $c_{\mathrm{SE}} \leftarrow m_b + \mathtt{E}_\pi(K, n_i)$ during which an adversary induces a fault during computation of $\mathtt{E}_\pi(K, n_i)$, resulting in faulty encryption $c'_{\mathrm{SE}} \leftarrow m_b + \mathtt{E}'_\pi(K, n_i)$.

Note that due to the use of nonce, every $\mathtt{E}_\pi(K, n_i)$ is distinct, and the adversary does not know the current value of $\mathtt{E}_\pi(K, n_i)$. Hence, it can also not predict the faulty value and consequently cannot guess b. The decryption requirement- $S[j] \cdot m_0 = S[j] \cdot m_1$ eliminates trivial attacks where the adversary can distinguish between two computations that lead to different results when

computed on exact values. Thus ensuring that the fault induced is non-trivial; for instance, no rounds are skipped, and the addition of $\mathtt{E}_\pi(K, n_i)$ to the message is not bypassed. To further substantiate this in the practical context of the new HHE enabling stream ciphers, we explicitly state that the fault can only be induced in its non-Key-dependent phase.

A.2.3 IND-KPA$^{\mathrm{D}}$ Security for AT under Faults

The KPA attack allows the adversary to access (plaintext, ciphertext) pairs passively through eavesdropping. Our formal definition of IND-KPA$^{\mathrm{D}}$ (Indistinguishability under Known Plaintext Attacks with Decryption oracle) follows from the IND-CPA$^{\mathrm{D}}$ definition described above. The only difference is that the IND-KPA$^{\mathrm{D}}$ adversary does not have query access to an encryption oracle and cannot request an evaluation on the function of her choice. Thus, this definition augments the IND-KPA security by allowing the adversary to access the encryption and decryption result. Similar to the original setting, the decryption queries are restricted to ciphertexts produced by encryption or evaluation oracles to ensure ciphertext validity and avoid trivial attacks.

A.3 Keccak Hash/PRNG Function

The Keccak [PA11] permutation function is renowned for its role as the underlying component of the secure hashing algorithm standard SHA-3. For the SE schemes, Keccak, in its SHAKE128/SHAKE256 modes, is utilized as XOF, and its inputs are the nonce and counter values. The data generated undergoes a process known as rejection sampling, where it is filtered to ensure that it is smaller than the prime modulus. This output is then used to generate matrices and round constants in schemes like Pasta and Masta and obtain the key schedule for Hera and Rubato.

A.4 Hera FIPs study

As stated in Sec. 7.3.3, the diffusion in RV_r completely depends on the diffusion obtained from XOF. To understand why the FIP of inducing a fault in Keccak permutation, used for Pasta and Masta, may not always lead to full diffusion, let us look at how the XOF calls work. It retrieves values from the Keccak output, which are then rejection sampled. If the Keccak output state is fully consumed, then another permutation is done, and a new output is generated. Note that primes that lead to a high rate of rejection sampling would require more permutations. Similarly, large moduli would also require more permutations. Hence, the random vector obtained would be distributed across many permutations. On the other hand, primes that have a low rejection rate will require fewer permutations. This is shown in Fig. A.7.

When we induce a fault, we have to ensure that we do not make the XOF results required in the previous rounds faulty. Hence, we must induce the fault in the

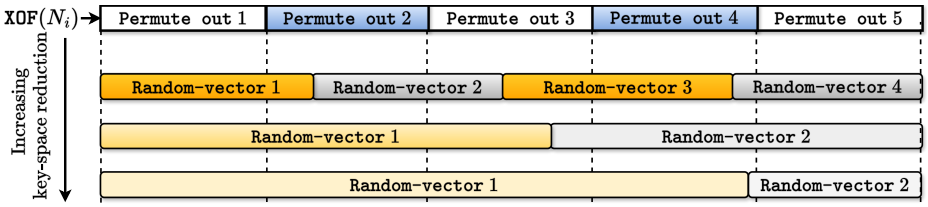


Figure A.7: This figure shows that more KECCAK permutations for random vector generation of HERA make it easier to target a specific vector without affecting others, leading to a higher reduction in the keyspace.

first permutation, which is exclusive to the XOF call for RV_r. This would diffuse the fault in all the remaining permutations as well. If the coefficients in RV_r are more distributed, more values would be faulty. Consequently, most key values can be recovered, and the reduced keyspace becomes susceptible to brute-force attacks. To summarize, our analysis reveals that in the case of HERA, prime moduli leading to a high rate of rejection sampling are more vulnerable under SASTA-DFA than primes with a lower rate of rejection sampling. Note that this analysis is specific to fault type- KECCAK instruction skip. Another fault type or location could have a similar effect, such as a matrix counter skip or skipping the matrix multiplication instruction. This will have a success probability of 1 without any reliance on prime size or rejection rate.

A.5 Server-side attack surface

Although attacking the client's device is more realistic and is generally the target of all prior works, we cannot completely rule out the possibility of the fault being induced on the server side by a malicious insider. Hence, informing the community about the potential of such an attack is imperative, acknowledging that its feasibility remains an open debate.

Theorem 1 (**The mirror effect**). *The* SASTA-DFA *fault model can be used on the server side during* HHD *to obtain the differential:*

$$\Delta E_\pi = E_\pi(K, n_i) - E'_\pi(K, n_i) \pmod{p}$$

Let us examine the ideal scenario where a fault (non-zero by definition) fully diffuses only to the diagonal of the last matrix/vector utilized in all the constructions discussed above. The steps for generating an invertible matrix are explained in Appendix 2.3.2. The server-side computation offers a remarkable opportunity for attack surface, thanks to the use of the baby-step giant-step based matrix-vector multiplication method. This method encodes the diagonals of matrix $\mathbf{M}$, or the vector in NTT form (discussed in Appendix A.3). These encoded values are then utilized for multiplication.

When the fault is introduced on data in NTT form, it diffuses throughout the entire plaintext after the Inverse NTT operation. This is because of the matrix multiplication used to reverse the transform. Since our plaintext is a diagonal/vector, the fault spreads to the whole diagonal/vector. For practically inducing this known fault, an instruction skip fault during the NTT transform is sufficient, as the values are public. Such a fault can be induced with high precision as shown in [BFP19; BH22; Dut+19; Men+20]. However, attacking the server-side computation can be challenging due to limited opportunities and constraints.

Although the fault can be introduced on both the client and server side (Corollary 1), it is more realistic that the attacker gains momentarily access to one device and not both. Hence, although we show interesting attack surfaces on both the client and server sides, we only analyze the schemes under the client-side fault injection points.

A.6 Comparison Table

Table A.1 presents the detailed comparisons with prior works. It shows that while our technique is utilized for HHE, it can also be utilized in standalone symmetric setting with standard assumptions for a key-recovery.

Table A.1: More comprehensive comparison of SASTA-DFA with prior works. The average time is reported for key-recovery on a 2GHz clock and 4GB RAM processor. † stands for results reported in [AM11b].

Target Scheme	Security	Field	Nonce Resp.	#Faults	Time	HHE Context	Reference
Flip_{530}	80-bit	$\mathbb{Z}_2$	No	1	39hr	✗	[RBM21]
Kreyvium	128-bit	$\mathbb{Z}_2$		3	5min	✗	[RBM21]
Filip_{430}	80-bit	$\mathbb{Z}_2$		1	751hr	✗	[Rad+23]
Rasta-6	80-bit	$\mathbb{Z}_2$		1	96hr	✗	[Rad+23]
Rasta-5	128-bit	$\mathbb{Z}_2$	**No**	1	0.11s	✗	Section 7.3.5
Rasta-6	128-bit	$\mathbb{Z}_2$		1	0.12s	✗	Section 7.3.5
Pasta-3	128-bit	$\mathbb{F}_p$		1	0.50s	✗	Section 7.3.1
Pasta-4	128-bit	$\mathbb{F}_p$		1	0.21s	✗	Section 7.3.1
Masta-4	128-bit	$\mathbb{F}_p$		1	0.09s	✗	Section 7.3.2
Masta-5	128-bit	$\mathbb{F}_p$		1	0.07s	✗	Section 7.3.2
Hera-5	128-bit	$\mathbb{F}_p$		1	0.005s	✗	Section 7.3.3
Rubato	128-bit	$\mathbb{F}_p$		2	0.15s	✗	Section 7.3.4
AES-GCM	128-bit	$\mathbb{Z}_2$	**Yes**	1	5s	✓	Section 7.3.6
AES-GCM	256-bit	$\mathbb{Z}_2$		1	45min†	✓	Section 7.3.6
Rasta-5	128-bit	$\mathbb{Z}_2$		1	0.11s	✓	Section 7.3.5
Rasta-6	128-bit	$\mathbb{Z}_2$		1	0.12s	✓	Section 7.3.5
Pasta-3	128-bit	$\mathbb{F}_p$		1	0.50s	✓	Section 7.3.1
Pasta-4	128-bit	$\mathbb{F}_p$		1	0.21s	✓	Section 7.3.1
Masta-4	128-bit	$\mathbb{F}_p$		1	0.09s	✓	Section 7.3.2
Masta-5	128-bit	$\mathbb{F}_p$		1	0.07s	✓	Section 7.3.2
Hera-5	128-bit	$\mathbb{F}_p$		1	0.05s	✓	Section 7.3.3
Rubato	128-bit	$\mathbb{F}_p$		2	0.15s	✓	Section 7.3.4

A.7 Client-server data exchange

Table A.2 gives the algorithmic description of the conventional client and server data exchange. The client-side computations are highlighted in detail. Algorithm 7 shows the iterative Cooley-Tukey algorithm for the NTT transformation.

Table A.2: An algorithmic description of the conventional client and server computations.

Client	**Server**
Encryption	
$(e_0, e_1)^*, v$ are refreshed every time	
$m' = \Delta \cdot \texttt{IDFT}(m) + e_0$	
$\boldsymbol{c_0} = \texttt{NTT}(v) \cdot \texttt{pk}_0 + \texttt{NTT}(m') \mod Q$	
$\boldsymbol{c_1} = \texttt{NTT}(v) \cdot \texttt{pk}_1 + \texttt{NTT}(e_1) \mod Q$	
Send $\texttt{c} = (\boldsymbol{c_0}, \boldsymbol{c_1})$ $\rightarrow$	$\rightarrow$ Receive $\texttt{c}$
Function Evaluation	
	$\acute{\texttt{c}} = f(\texttt{c})$
Receive $\acute{\texttt{c}}$ $\leftarrow$	$\leftarrow$ Send $\acute{\texttt{c}}$
Decryption	
$m' = \texttt{DFT}(\Delta^{-1}(\texttt{INTT}(\acute{\boldsymbol{c}}_1 \cdot \texttt{sk} + \acute{\boldsymbol{c}}_0)))$	

* Although we had mentioned that the error is refreshed every time, for clarity, we have given it two variable names e_0, e_1.

A.8 Baby-step giant-step method

The matrix-vector multiplication required for processing the DFT transformation on the server side is expensive. This is not a problem on the client side, as data can be fetched and stored as desired. Contrarily, on the server side data vector is encoded/encrypted into polynomials, which do not offer much flexibility and consume significant multiplicative depth. To reduce this, the baby-step giant-step method [HS18] is used for expediting matrix-vector multiplication on encoded/encrypted data, as shown in the equation below.

$$\texttt{c}' = \sum_{k=0}^{N_2-1} \texttt{rot}_{kN_2} \left(\sum_{j=0}^{N_1-1} \texttt{diag}_{kN_1+j}(\mathbf{M}) \odot \texttt{rot}_j(\texttt{c}) \right)$$

where matrix ($\mathbf{M}$) represents encoded plaintext data for DFT transformation in this context, and the vector corresponds to encrypted ciphertext ($\texttt{c}$). N is the ciphertext size, and $N = N_1 \cdot N_2$. The $\texttt{rot}_j$ function rotates the input vector by j, and the $\texttt{diag}$ function gives rotated diagonals of $\mathbf{M}$.

The method involves three main steps: first, the diagonals of the matrix are homomorphically encoded; second, these encoded diagonals are rotated; and

third, the rotated diagonals are multiplied with the rotated ciphertext, followed by additional rotations and accumulations.

A.9 Comparison of Boosted-Deflation with HHE

We present the comparison with respect to one of the state-of-the-art schemes over integers- Pasta [Dob+23]. Pasta-4 requires up to 5 multiplications per encryption and can handle up to 32 data elements. However, the duplicated packing necessary for the utilized matrix multiplication technique hinders the efficient packed multiplication of multiple ciphertexts. Consequently, for 2^{15} data elements, the depth consumption would reach $5,120$ (with 1024 parallel executions). In contrast, our technique, which involves only one bootstrapping operation, would at most consume a depth of 16 for 2^{15} data elements.

In Pasta-3, the state comprises 256 elements, split into two parts of 128 each. The number of rounds is three, and each round involves a 128×128 invertible matrix generation and multiplication. The multiplicative complexity of both operations is the same- $128 \cdot 128$, and eight such operations are required. This brings the total multiplication cost to 2^{18}. It is interesting to note that, while FHE encrypts 2^{15} elements in 2^{16} degree polynomials, Pasta-3 can only encrypt $128 = 2^7$. Therefore, although Pasta-3 can encrypt 32 elements fast, it will need 2^8 more encryptions to encrypt 2^{15} elements, thus requiring total 2^{26} modular multiplications. For Pasta-4, the required computations for encrypting the same amount of data are halved due to half state size, which has a quadratic impact on multiplication count and a linear on the amount of data packed.

The polynomial degree in our work is $N = 2^{16}$, and for polynomial multiplication (the *most expensive* routine), a Number Theoretic Transform (NTT) based technique is utilized. Each NTT transform requires $\frac{N \cdot \log N}{2}$ multiplications. Conventionally, this process would have repeated $31\times$, bringing the cost to $\approx 2^{26}$ modular multiplications, similar to that required for Pasta-3 and half of what is required for Pasta-4. With our proposed approach, the total number of multiplications required is $\approx 2^{21}$, which is $32\times$ less than those required for Pasta-3. A similar analysis for Pasta-4 shows that our proposed technique delivers $16\times$ less modular multiplication computation.

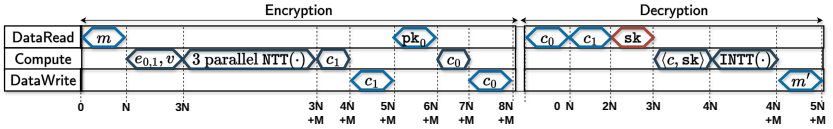


Figure A.8: The proposed timeline of the encryption-decryption routines such that maximum residue polynomial storage required at any instance is three. Here, M is the latency of one NTT (or three parallel NTTs).

A.10 Algorithmic description of the proposed technique and Security Aspects

Algorithm 31 and Algorithm 32 provide algorithmic descriptions of FHE encryption and decryption routines after applying the proposed Boosted-Deflation technique.

Algorithm 31 New CKKS Enc.

In: $m \in \mathbb{R}$, $\mathtt{pk} = (\boldsymbol{pk}_0, \boldsymbol{pk}_1) \in \mathbb{R}^2_{q_0}$
Out: $\mathtt{c} = \text{Encrypt}(m) \in \mathbb{R}^2_{q_0}$

1: $v \xleftarrow{\$} \chi_{enc}$, $e_0, e_1 \xleftarrow{\$} \chi_{err}$
2: $\acute{m} \leftarrow \Delta \cdot m + e_0$
3: $v' \leftarrow \mathtt{NTT}(v) \bmod q_i$
4: $c_0^0 \leftarrow (pk_0^0 \cdot v' + \acute{m}_{\mathtt{NTT}})_{q_0}$
5: $c_1^0 \leftarrow (pk_1^0 \cdot v' + e_{1_{\mathtt{NTT}}})_{q_0}$
6: **return** $\mathtt{c} = \{\boldsymbol{c}_0, \boldsymbol{c}_1\} \in \mathbb{R}^2_{q_0}$

Algorithm 32 New CKKS Dec.

In: $\mathtt{c}, \mathtt{sk} \in \mathbb{R}^2_{q_0}$
Out: $m' = \text{Decrypt}(\mathtt{c})$

1: $c' \leftarrow \mathtt{INTT}(\langle \mathtt{c}, \mathtt{sk} \rangle)_{q_0}$
2: $m' \leftarrow \Delta^{-1} \cdot c'$
3: **return** $m' \in \mathbb{R}$

The security definitions for the proposed Boosted-Deflation technique are as follows.

1. **Confidentiality**: Encoding and decoding do not involve secret key routines. Thus, offloading this operation on encrypted data does not impact confidentiality guarantees.
2. **Integrity**: Once encrypted, data retains its security level regardless of whether it was originally encoded using DFT or any other transformation.
3. **Attack Resilience**: Since encoding and decoding are a publicly known transformation, an adversary gains no advantage by observing encoded messages before encryption.
4. **RLWE Security**: The difficulty of solving RLWE instances remains unchanged as long as noise levels do not exceed decryption failure thresholds. Our approach follows standard parameter recommendations to ensure this condition holds.

5. **Noise Management**: Reducing the initial ciphertext modulus does not accelerate noise growth beyond what is already accounted for in standard FHE parameter selection.

6. **Bootstrapping Resilience**: The bootstrapping process inherently maintains security under the RLWE assumption by resetting noise accumulation. Additionally, noise flooding during bootstrapping may offer additional protection against IND-CPA^D attacks [LM21].

A.11 Overview of Framework Components

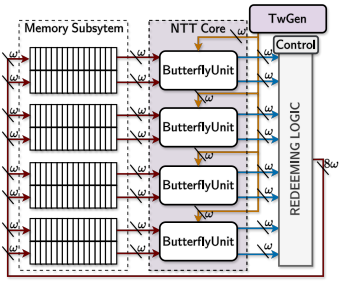


Figure A.9: The NTT architecture which utilizes four butterfly units.

We present this overview in the context of hardware design, as for the software we adapt the exiting open source library HEAAN [HC23]. A key component of client-side FHE acceleration architecture is the NTT unit, which utilizes modular arithmetic. For converting the plaintext from floating-point representation to integer representation and vice versa during en/decode, a scaling unit is also integrated which performs simple power-of-two multiplications of Δ/Δ^{-1}. The design details of the NTT and modular reduction unit are as follows.

NTT Unit.

In our design, we use three parallel NTT cores. Algorithm 7 in the Section 2.4.3 details the NTT transformation. Each core performs one $N = 2^{16}$-point NTT on independent polynomials. This is used to compute $\texttt{NTT}(m + e_0)$, $\texttt{NTT}(e_1)$, and $\texttt{NTT}(v)$ simultaneously during the encryption procedure. The implementation of NTT (shown in Figure A.9) utilizes a divide-and-conquer approach by recursively calling the Butterfly Unit (BFU). Since we need both NTT and INTT transforms, our BFU is unified and performs both operations [Aik+22]. We further reduce the latency of NTT 4× by instantiating four $BFUs$ per NTT core. This improved NTT latency matches the latency of other hardware components, such as PRNG. Hence, the overall encryption latency is balanced across all suboperations.

During each NTT transformation, a total of N twiddle factors (generated using $2N$-th primitive root of unity) are required. However, storing this amount

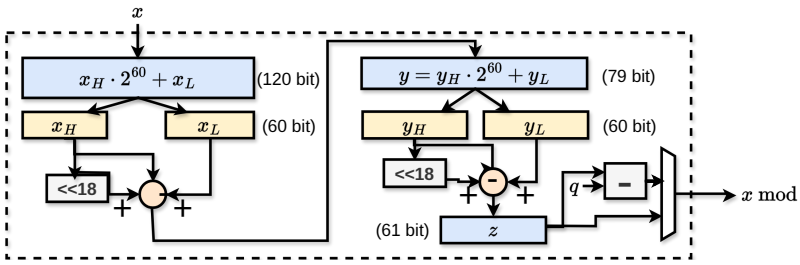


Figure A.10: The shift-add based Modular Reduction unit.

of twiddle factors on-chip would require 108 BRAMs, which exceeds the available resources for the target FPGA. Therefore, we use on-the-fly generated twiddle factors to lower the BRAM consumption to just 2 BRAMs for storing initial twiddle factors. Since all three NTT cores perform the same operation, they require the same twiddle factors. Hence, we only instantiate one such twiddle factor generation module. This measure effectively reduces the resource and energy consumption.

Butterfly Unit Utilized in NTT/INTT.

Each *BFU* performs dyadic addition, subtraction, and multiplication on the two input coefficients, with results reduced modulo q_0. The Gentleman-Sande (GS) [Sco17a] butterfly is utilized in Decimation-in-Frequency (DIF) transformations, while the Cooley-Tukey (CT) [Sco17a] butterfly is applied in Decimation-in-Time (DIT) transformations. Each butterfly operation takes two coefficients as input and produces two outputs. Additionally, the twiddle factor, ψ, is employed during butterfly computations [Sco17a]. Consequently, CT and GS butterfly configurations are used for the NTT and INTT. To facilitate both NTT and INTT, a unified butterfly core is designed to accommodate both CT and GS butterfly configurations.

Modular Reduction Unit and Data Flow.

The modular reduction in prime field multiplication is a central hardware design aspect. As discussed earlier, we only need to support one modulus. To leverage this fact, we choose the 60-bit prime $q_0 = 2^{60} - 2^{18} + 1$ and design an optimized add-shift-based reduction unit shown in Figure A.10. Compared to classical word-level Montgomery reduction, our approach uses 32% less LUTs and no DSPs. In addition, our reduction unit's power consumption is 3.5× lower.

High-level architecture.

The algorithm described in Table 8.1 serves as a reference for discussing architectural design techniques. Our proposed architecture is shown in Figure 8.8. The central components are the three memories MEM1 to MEM3, wherein each

memory stores one $N = 2^{16}$-degree residue polynomial. Attached to the memories are three unified NTT cores capable of performing forward NTT during encryption on up to three polynomials simultaneously and an inverse NTT during decryption. In addition, modular arithmetic modules $(\times, +)$ are integrated. These modules perform coefficient-wise arithmetic between two polynomials in the NTT domain. During encryption, a lightweight Pseudo-Random Number Generator (PRNG)- Trivium [Bou20] is followed by sampling to generate v, e_0, and e_1 directly in hardware. In addition, one part of the public key, $pk_1 \in \mathbb{R}_{q_0}$, is also sampled on-demand based on a given seed. These aspects address reducing off-chip communication effort to minimize the overall latency. The I/O interface of our hardware accelerator facilitates control and data exchange with the DDR memory. To stream a large amount of data to and from the DDR, a Direct Memory Access (DMA) controller is utilized. This DMA transfers one coefficient size (ω) per clock cycle in our design.

Data Flow.

The data flow for the complete encryption and decryption procedure is illustrated with the timeline in Figure A.8. An architecture-based flow for encryption decryption routines is depicted in Section A.12.

During encryption, the client only needs to send the message m and public key $pk_0 \in \mathbb{R}_{q_0}$. This data is sent once, and all subsequent data is generated dynamically (on-the-fly). First, the message is received, followed by the generation of the error (e_0, e_1) and v polynomials. These fully occupy the three available memory blocks, preventing additional data from being loaded and at this stage, only the c_1 component of the ciphertext can be generated. Hence, initially, the data in all three memories ($m + e_0$, v, and e_1) is processed via three parallel NTT cores. Following this, the data in the second and third memories ($\texttt{NTT}(v)$, $\texttt{NTT}(e_1)$) is processed along with on-the-fly generated pk_1 to compute c_1. Finally, c_1 is offloaded to the client device, freeing one memory block. This allows pk_0 to be loaded into this memory block to compute c_0.

As depicted in the figure, the latency of error sampling is $2N$, whereas all other linear operations take N clock cycles. The latency of the NTT is $M = \frac{N \log_2 N}{2 \cdot \#BFU}$, where $\#BFU$ stands for the number of butterfly units [Sco17a] utilized per NTT core. As the maximum latency among the non-NTT steps is $2N$, we have chosen $\#BFU = 4$ so that for $\log_2 N = 16$, $M = 2N$ and the NTT unit's latency is at par with the remaining units. This flow can be pipelined for larger, non-constrained environments. Consequently, the encryption and decryption routines will offer the same latency but much higher throughput. The complete data flow for encryption with the architecture is shown in Figure A.11 (Section A.12).

For decryption, the two residue polynomials in the ciphertext (c_0, c_1) and the secret key $\texttt{sk}$ are initially loaded into the three available memory blocks. After a coefficient-wise multiplication and addition ($\langle c_1, \texttt{sk} \rangle + c_0$), one of the NTT units is utilized to perform the INTT operation (shown in Figure A.11, Section A.12). Following this, the resultant plaintext is sent back to the DDR memory.

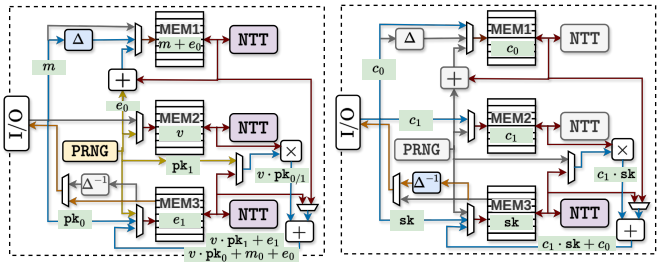


Figure A.11: The data flow during encryption and decryption routines in the proposed high-level architecture.

A.12 Operation-wise Data Flow

The flow diagram in Figure A.11 illustrates the flow of plaintext, public key and ciphertext during the CKKS encryption procedure. Similarly, the second flow diagram in Figure A.11 shows the ciphertext, secret key, and resultant plaintext flow during the CKKS decryption procedure.

B
Publications

Main Publications

[Aik+22] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. "KaLi: A Crystal for Post-Quantum Security." In: *IEEE Transactions on Circuits and Systems I (TCAS-1) 2022* (2022). https://eprint.iacr.org/2022/1086.

[Aik+23a] Aikata Aikata, Andrea Basso, Gaëtan Cassiers, Ahmet Can Mert, and Sujoy Sinha Roy. "Kavach: Lightweight masking techniques for polynomial arithmetic in lattice-based cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.3 (2023), pp. 366–390. DOI: 10.46586/TCHES.V2023.I3.366-390. URL: https://doi.org/10.46586/tches.v2023.i3.366-390.

[Aik+23b] Aikata Aikata, Ahmet Can Mert, David Jacquemin, Amitabh Das, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy. "A Unified Cryptoprocessor for Lattice-Based Signature and Key-Exchange." In: *IEEE Trans. Comput.* 72.6 (June 2023), pp. 1568–1580. ISSN: 0018-9340. DOI: 10.1109/TC.2022.3215064. URL: https://doi.org/10.1109/TC.2022.3215064.

[Aik+25] Aikata Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. "REED: Chiplet-Based Scalable Hardware Accelerator for Fully Homomorphic Encryption." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2025), p. 1190. URL: https://eprint.iacr.org/2023/1190.

[Aik+26a] Aikata Aikata, Maciej Czuprynko, Nedma Musovic, Emira Salki, and Sujoy Sinha Roy. *Attacking and Securing Hybrid Homomorphic Encryption Against Power Analysis*. IEEE DATE. 2026. URL: https://eprint.iacr.org/2025/2302.

[Aik+26b] Aikata Aikata, Ahaan Dabholkar, Dhiman Saha, and Sujoy Sinha Roy. "SASTA: Ambushing Hybrid Homomorphic Encryption Schemes with a Single Fault." In: *ASIA CCS '26: The 21st ACM Asia Conference on Computer and Communications Security, 2026*. 2026, p. 41. URL: https://dblp.org/rec/journals/iacr/AikataSR24.bib.

[AKR26] Aikata Aikata, Florian Krieger, and Sujoy Sinha Roy. "Privacy at your Fingertips: Enabling Rapid Client-Side Operations in Fully Homomorphic Encryption." In: *11th IEEE European Symposium on Security and Privacy (Euro S&P)*. 2026.

[AR24] Aikata Aikata and Sujoy Sinha Roy. "Secure and Efficient Outsourced Matrix Multiplication with Homomorphic Encryption." In: *Progress in Cryptology - INDOCRYPT 2024 - 25th International Conference on Cryptology in India, Chennai, India, December 18-21, 2024, Proceedings, Part I*. Ed. by Sourav Mukhopadhyay and Pantelimon Stanica. Vol. 15495. Lecture Notes in Computer Science. Springer, 2024, pp. 51–74. DOI: 10.1007/978-3-031-80308-6_3. URL: https://doi.org/10.1007/978-3-031-80308-6%5C_3.

[ASR25] Aikata Aikata, Daniel Sanz Sobrino, and Sujoy Sinha Roy. "Pasta on Edge: Cryptoprocessor for Hybrid Homomorphic Encryption." In: *Design, Automation & Test in Europe Conference, DATE 2025, Lyon, France, March 31 - April 2, 2025*. IEEE DATE, 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992820. URL: https://doi.org/10.23919/DATE64628.2025.10992820.

Further Publications

[Ada+25] Janis Adamek et al. "FHERMA Cookbook: FHE Components for Privacy-Preserving Applications." In: *IACR Cryptol. ePrint Arch.* (2025), p. 1302. URL: https://eprint.iacr.org/2025/1302.

[Dob+26] Patrik Dobias, Aikata Aikata, Azade Rezaeezade, Lejla Batina, Lukas Malina, and Tomas Gerlich. "Two birds, one mask: Side-channel-resistant unified hardware for ML-DSA and ML-KEM." In: *Preprint* (2026).

[Imr+23] Malik Imran, Aikata Aikata, Sujoy Sinha Roy, and Samuel N. Pagliarini. "Towards High-speed ASIC Implementations of Post-Quantum Cryptography." In: *IACR Cryptol. ePrint Arch.* (2023), p. 716. URL: https://eprint.iacr.org/2023/716.

[Imr+24] Malik Imran, Aikata Aikata, Sujoy Sinha Roy, and Samuel Pagliarini. "High-Speed Design of Post Quantum Cryptography With Optimized Hashing and Multiplication." In: *IEEE Trans. Circuits Syst. II Express Briefs* 71.2 (2024), pp. 847–851. DOI: 10.1109/TCSII.2023.3273821. URL: https://doi.org/10.1109/TCSII.2023.3273821.

[Kim+24] Andrey Kim, Ahmet Can Mert, Anisha Mukherjee, Aikata Aikata, Maxim Deryabin, Sunmin Kwon, HyungChul Kang, and Sujoy Sinha Roy. "Exploring the Advantages and Challenges of Fermat NTT in FHE Acceleration." In: *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part III*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14922. Lecture Notes in Computer Science. Springer, 2024, pp. 76–106. DOI: 10.1007/978-3-031-68382-4_3. URL: https://doi.org/10.1007/978-3-031-68382-4%5C_3.

[Kun+24] Anup Kumar Kundu, Shibam Ghosh, Aikata Aikata, and Dhiman Saha. "ToFA: Towards Fault Analysis of GIFT and GIFT-like Ciphers Leveraging Truncated Impossible Differentials." In: *IACR Cryptol. ePrint Arch.* (2024), p. 1927. URL: https://eprint.iacr.org/2024/1927.

[Mer+23] Ahmet Can Mert, Aikata Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. "Medha: Microcoded Hardware Accelerator for computing on Encrypted Data." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.1 (2023), pp. 463–500. DOI: 10.46586/tches.v2023.i1.463-500. URL: https://doi.org/10.46586/tches.v2023.i1.463-500.

[Muk+24] Anisha Mukherjee, Aikata Aikata, Ahmet Can Mert, Yongwoo Lee, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. "ModHE: Modular Homomorphic Encryption Using Module Lattices Potentials and Limitations." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 527–562. DOI: 10.46586/TCHES.V2024.I1.527-562. URL: https://doi.org/10.46586/tches.v2024.i1.527-562.

[Pag+24] Samuel Pagliarini, Aikata Aikata, Malik Imran, and Sujoy Sinha Roy. "REPQC: Reverse Engineering and Backdooring Hardware Accelerators for Post-quantum Cryptography." In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. Ed. by Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro A. Cárdenas. ACM, 2024. DOI: 10.1145/3634737.3657016. URL: https://doi.org/10.1145/3634737.3657016.

[Rav+24] Prasanna Ravi, Shivam Bhasin, Anupam Chattopadhyay, Aikata Aikata, and Sujoy Sinha Roy. "Backdooring Post-Quantum Cryptography: Kleptographic Attacks on Lattice-based KEMs." In: *Proceedings of the Great Lakes Symposium on VLSI 2024*. GLSVLSI '24. Clearwater, FL, USA: Association for Computing Machinery, 2024,

pp. 216–221. ISBN: 9798400706059. DOI: 10.1145/3649476.3660373. URL: https://doi.org/10.1145/3649476.3660373.

[Smo+26] Vid Smole, Aikata Aikata, Pilipovic Ratko, and Sujoy Sinha Roy. *Giving Falcon Its Wings: Fast Hardware Acceleration for Signing and Verification using FN-DSA*. PrePrint. 2026.

Bibliography

[Abd+] Abubakr Abdulgadir, Kamyar Mohajerani, Viet Ba Dang, Jens-Peter Kaps, and Kris Gaj. "Lightweight Implementation of Saber Resistant Against Side-Channel Attacks." In: ().

[Abd+21a] Abubakr Abdulgadir, Kamyar Mohajerani, Viet Ba Dang, Jens-Peter Kaps, and Kris Gaj. "A Lightweight Implementation ofăSaber Resistant Against Side-Channel Attacks." In: *Progress in Cryptology – INDOCRYPT 2021*. Ed. by Avishek Adhikari, Ralf Küsters, and Bart Preneel. Cham: Springer International Publishing, 2021, pp. 224–245. ISBN: 978-3-030-92518-5.

[Abd+21b] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. "Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (Nov. 2021), pp. 127–151. DOI: 10.46586/tches.v2022.i1.127-151. URL: https://tches.iacr.org/index.php/TCHES/article/view/9292.

[Agr+23] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. "FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption." In: *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 2023, pp. 882–895. DOI: 10.1109/HPCA56546.2023.10070953. URL: https://doi.org/10.1109/HPCA56546.2023.10070953.

[Aha+23] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. "Poster: Efficient AES-GCM Decryption Under Homomorphic Encryption." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023, pp. 3567–3569. DOI: 10.1145/3576915.3624377. URL: https://doi.org/10.1145/3576915.3624377.

[AI] Verrtex AI. *Vertex AI*. https://cloud.google.com/vertex-ai.

[Aik+22] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. "KaLi: A Crystal for Post-Quantum Security." In: *IEEE Transactions on Circuits and Systems I (TCAS-1) 2022* (2022). https://eprint.iacr.org/2022/1086.

[Aik+23a] Aikata Aikata, Andrea Basso, Gaëtan Cassiers, Ahmet Can Mert, and Sujoy Sinha Roy. "Kavach: Lightweight masking techniques for polynomial arithmetic in lattice-based cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.3 (2023), pp. 366–390. DOI: 10.46586/TCHES.V2023.I3.366-390. URL: https://doi.org/10.46586/tches.v2023.i3.366-390.

[Aik+23b] Aikata Aikata, Ahmet Can Mert, David Jacquemin, Amitabh Das, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy. "A Unified Cryptoprocessor for Lattice-Based Signature and Key-Exchange." In: *IEEE Trans. Comput.* 72.6 (June 2023), pp. 1568–1580. ISSN: 0018-9340. DOI: 10.1109/TC.2022.3215064. URL: https://doi.org/10.1109/TC.2022.3215064.

[Aik+23c] Aikata Aikata, Ahmet Can Mert, David Jacquemin, Amitabh Das, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy. "A Unified Cryptoprocessor for Lattice-Based Signature and Key-Exchange." In: *IEEE Trans. Comput.* 72.6 (June 2023), pp. 1568–1580. ISSN: 0018-9340. DOI: 10.1109/TC.2022.3215064. URL: https://doi.org/10.1109/TC.2022.3215064.

[Aik+25] Aikata Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. "REED: Chiplet-Based Scalable Hardware Accelerator for Fully Homomorphic Encryption." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2025), p. 1190. URL: https://eprint.iacr.org/2023/1190.

[Aik+26a] Aikata Aikata, Maciej Czuprynko, Nedma Musovic, Emira Salki, and Sujoy Sinha Roy. *Attacking and Securing Hybrid Homomorphic Encryption Against Power Analysis*. IEEE DATE. 2026. URL: https://eprint.iacr.org/2025/2302.

[Aik+26b] Aikata Aikata, Ahaan Dabholkar, Dhiman Saha, and Sujoy Sinha Roy. "SASTA: Ambushing Hybrid Homomorphic Encryption Schemes with a Single Fault." In: *ASIA CCS '26: The 21st ACM Asia Conference on Computer and Communications Security, 2026*. 2026, p. 41. URL: https://dblp.org/rec/journals/iacr/AikataSR24.bib.

[AKR26] Aikata Aikata, Florian Krieger, and Sujoy Sinha Roy. "Privacy at your Fingertips: Enabling Rapid Client-Side Operations in Fully Homomorphic Encryption." In: *11th IEEE European Symposium on Security and Privacy (Euro S&P)*. 2026.

[Al +22] Ahmad Al Badawi et al. "OpenFHE: Open-Source Fully Homomorphic Encryption Library." In: *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 53–63. DOI: 10.1145/3560827.3563379. URL: https://doi.org/10.1145/3560827.3563379.

[Alb+19] Martin R. Albrecht et al. "Homomorphic Encryption Standard." In: *IACR Cryptol. ePrint Arch.* (2019), p. 939. URL: https://eprint.iacr.org/2019/939.

[Alk+16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum Key Exchange - A New Hope." In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 327–343. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

[Alk+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. "ISA Extensions for Finite Field Arithmetic Accelerating Kyber and NewHope on RISC-V." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 219–242. DOI: 10.13154/tches.v2020.i3.219-242.

[AM11a] Subidh Ali and Debdeep Mukhopadhyay. "A Differential Fault Analysis on AES Key Schedule Using Single Fault." In: *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*. Ed. by Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi. IEEE Computer Society, 2011, pp. 35–42. DOI: 10.1109/FDTC.2011.10. URL: https://doi.org/10.1109/FDTC.2011.10.

[AM11b] Subidh Ali and Debdeep Mukhopadhyay. "An Improved Differential Fault Analysis on AES-256." In: *Progress in Cryptology - AFRICACRYPT 2011 - 4th International Conference on Cryptology in Africa, Dakar, Senegal, July 5-7, 2011. Proceedings*. Ed. by Abderrahmane Nitaj and David Pointcheval. Vol. 6737. Lecture Notes in Computer Science. Springer, 2011, pp. 332–347. DOI: 10.1007/978-3-642-21969-6_21. URL: https://doi.org/10.1007/978-3-642-21969-6%5C_21.

[AMD23] AMD. *AMD Instinct MI300 Series Accelerators*. Tech. rep. AMD, 2023. URL: https://www.amd.com/en/products/accelerators/instinct/mi300.html.

[ang23] angelolmg. *Textile Texture Database (TILDA) for defect detection*. 2023. URL: https://www.kaggle.com/datasets/angelolmg/tilda-400-64x64-patches.

[AR24] Aikata Aikata and Sujoy Sinha Roy. "Secure and Efficient Outsourced Matrix Multiplication with Homomorphic Encryption." In: *Progress in Cryptology - INDOCRYPT 2024 - 25th International Conference on Cryptology in India, Chennai, India, December 18-21, 2024, Proceedings, Part I*. Ed. by Sourav Mukhopadhyay and Pantelimon Stanica. Vol. 15495. Lecture Notes in Computer Science. Springer, 2024, pp. 51–74. DOI: 10.1007/978-3-031-80308-6_3. URL: https://doi.org/10.1007/978-3-031-80308-6%5C_3.

[Aru+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, and many more. *Quantum supremacy using a programmable superconducting processor*. Nature. https://doi.org/10.1038/s41586-019-1666-5. 2019.

[Ash+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. "Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE." In: *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 483–501. DOI: 10.1007/978-3-642-29011-4_29. URL: https://doi.org/10.1007/978-3-642-29011-4%5C_29.

[ASR25a] Aikata Aikata, Daniel Sanz Sobrino, and Sujoy Sinha Roy. "Pasta on Edge: Cryptoprocessor for Hybrid Homomorphic Encryption." In: *2025 Design, Automation & Test in Europe Conference (DATE)*. 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992820.

[ASR25b] Aikata Aikata, Daniel Sanz Sobrino, and Sujoy Sinha Roy. "Pasta on Edge: Cryptoprocessor for Hybrid Homomorphic Encryption." In: *Design, Automation & Test in Europe Conference, DATE 2025, Lyon, France, March 31 - April 2, 2025*. IEEE DATE, 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992820. URL: https://doi.org/10.23919/DATE64628.2025.10992820.

[ATM24] ATMEL. *8-bit Microcontroller with 32KBytes In-System Programmable Flash*. Tech. rep. ATMEL, 24. URL: https://www.microchip.com/en-us/product/atmega32.

[Aza+22] Zahra Azad, Guowei Yang, Rashmi Agrawal, Daniel Petrisko, Michael Bedford Taylor, and Ajay Joshi. "RACE: RISC-V SoC for En/decryption Acceleration on the Edge for Homomorphic Computation." In: *ISLPED '22: ACM/IEEE International Symposium on Low Power Electronics and Design, Boston, MA, USA, August 1 - 3, 2022*. ACM, 2022. DOI: 10.1145/3531437.3539725. URL: https://doi.org/10.1145/3531437.3539725.

[Aza+23] Zahra Azad, Guowei Yang, Rashmi Agrawal, Daniel Petrisko, Michael Bedford Taylor, and Ajay Joshi. "RISE: RISC-V SoC for En/Decryption Acceleration on the Edge for Homomorphic Encryption." In: *IEEE Trans. Very Large Scale Integr. Syst.* 31.10 (2023), pp. 1523–1536. DOI: 10.1109/TVLSI.2023.3288754. URL: https://doi.org/10.1109/TVLSI.2023.3288754.

[Bad+20] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. "PrivFT: Private and Fast Text Classification With Homomorphic Encryption." In: *IEEE Access* 8 (2020), pp. 226544–226556. ISSN: 2169-3536. DOI: 10.1109/access.2020.3045465. URL: http://dx.doi.org/10.1109/ACCESS.2020.3045465.

[Bae+23] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. "HERMES: Efficient Ring Packing Using MLWE Ciphertexts and Application to Transciphering." In: *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14084. Lecture Notes in Computer Science. Springer, 2023, pp. 37–69. DOI: 10.1007/978-3-031-38551-3_2. URL: https://doi.org/10.1007/978-3-031-38551-3%5C_2.

[Bae+24] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. "Bootstrapping Bits with CKKS." In: *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part II*. Ed. by Marc Joye and Gregor Leander. Vol. 14652. Lecture Notes in Computer Science. Springer, 2024, pp. 94–123. DOI: 10.1007/978-3-031-58723-8_4. URL: https://doi.org/10.1007/978-3-031-58723-8%5C_4.

[Bai+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Dilithium*. Proposal to NIST PQC Standardization, Round3. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions. 2021.

[Bak+23] Anubhab Baksi et al. "BAKSHEESH: Similar Yet Different From GIFT." In: *IACR Cryptol. ePrint Arch.* (2023), p. 750. URL: https://eprint.iacr.org/2023/750.

[BAK21a] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. "High-Speed NTT-based Polynomial Multiplication Accelerator for CRYSTALS-Kyber Post-Quantum Cryptography." In: *IACR Cryptol. ePrint Arch.* (2021), p. 563.

[BAK21b] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. "Instruction-Set Accelerated Implementation of CRYSTALS-Kyber." In: *IEEE Trans. Circuits Syst. I Regul. Pap.* 68.11 (2021), pp. 4648–4659. DOI: 10.1109/TCSI.2021.3106639.

[Bal+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations." In: *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*. Ed. by Marc Joye and Amir Moradi. Vol. 8968. Lecture Notes in Computer Science. Springer, 2014, pp. 64–81. DOI: 10.1007/978-3-319-16763-3_5. URL: https://doi.org/10.1007/978-3-319-16763-3%5C_5.

[Ban+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. "GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption." In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 321–345. DOI: 10.1007/978-3-319-66787-4_16. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_16.

[Bar+10] Alessandro Barenghi, Guido Marco Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. "Fault attack on AES with single-bit induced faults." In: *Sixth International Conference on Information Assurance and Security, IAS 2010, Atlanta, GA, USA, August 23-25, 2010*. IEEE, 2010, pp. 167–172. DOI: 10.1109/ISIAS.2010.5604061. URL: https://doi.org/10.1109/ISIAS.2010.5604061.

[Bar+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults." In: *ESORICS (1)*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318.

[Bar86] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor." In: *Annual International Cryptology Conference*. 1986. URL: https://api.semanticscholar.org/CorpusID:19251441.

[Bas+21a] Andrea Basso, Furkan Aydin, Daniel Dinu, Joseph Friel, Avinash Varna, Manoj Sastry, and Santosh Ghosh. *Where Star Wars Meets Star Trek: SABER and Dilithium on the Same Polynomial Multiplier*. Crypto. ePrint Arch., Report 2021/1697. 2021.

[Bas+21b] Andrea Basso, Furkan Aydin, Daniel Dinu, Joseph Friel, Avinash Varna, Manoj Sastry, and Santosh Ghosh. *Where Star Wars Meets Star Trek: SABER and Dilithium on the Same Polynomial Multiplier*. Cryptology ePrint Archive, Paper 2021/1697. https://eprint.iacr.org/2021/1697. 2021. URL: https://eprint.iacr.org/2021/1697.

[BBS21] Adda-Akram Bendoukha, Aymen Boudguiga, and Renaud Sirdey. "Revisiting Stream-Cipher-Based Homomorphic Transciphering in the TFHE Era." In: *Foundations and Practice of Security - 14th International Symposium, FPS 2021, Paris, France, December 7-10, 2021, Revised Selected Papers*. Ed. by Esma Aimeur, Maryline Laurent, Reda Yaich, Benoit Dupont, and Joaquin Garcia-Alfaro. Vol. 13291. Lecture Notes in Computer Science. Springer, 2021, pp. 19–33. DOI: 10.1007/978-3-031-08147-7_2. URL: https://doi.org/10.1007/978-3-031-08147-7%5C_2.

[BC22] Olivier Bronchain and Gaëtan Cassiers. "Bitslicing Arithmetic/-Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 553–588.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model." In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2. URL: https://doi.org/10.1007/978-3-540-28632-5%5C_2.

[BCV18] Carl Bootland, Wouter Castryck, and Frederik Vercauteren. "On the Security of the Multivariate Ring Learning with Errors Problem." In: *IACR Cryptol. ePrint Arch.* (2018), p. 966. URL: https://eprint.iacr.org/2018/966.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)." In: *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4. URL: https://doi.org/10.1007/3-540-69053-0%5C_4.

[Bei+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. *A Side-Channel Resistant Implementation of SABER*. Crypto. ePrint Arch., Report 2020/733. 2020.

[Bei+21] Michiel Van Beirendonck, Jan-Pieter Danvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. "A Side-Channel-Resistant Implementation of SABER." In: *J. Emerg. Technol. Comput. Syst.* 17.2 (Apr. 2021). ISSN: 1550-4832. DOI: 10.1145/3429983. URL: https://doi.org/10.1145/3429983.

[Bei+23] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. "FPT: A Fixed-Point Accelerator for Torus Fully Homomorphic Encryption." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023, pp. 741–755. DOI: 10.1145/3576915.3623159. URL: https://doi.org/10.1145/3576915.3623159.

[Ben+25] Emad Heydari Beni, Lode Hoste, Geert Heyman, Paschalis Tsiaflakis, Barry van Leeuwen, Robin Geelen, Marc Rivinius, and Claudia Bartoli. "QRYPT: End-to-End Encrypted Audio Calls via Blind Audio Mixing." In: *Real World Crypto* (2025). URL: https://www.youtube.com/live/6FQX2otSbuE?si=PMnUlIJT6m2Cuubp.

[Ber+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. *The SPHINCS+ Signature Framework*. London, United Kingdom, 2019. DOI: 10.1145/3319535.3363229. URL: https://doi.org/10.1145/3319535.3363229.

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 199–224. DOI: 10.13154/tches.v2019.i2.199-224. URL: https://doi.org/10.13154/tches.v2019.i2.199-224.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "Fully Homomorphic Encryption without Bootstrapping." In: *Electron. Colloquium Comput. Complex.* (2011), p. 111. URL: https://eccc.weizmann.ac.il/report/2011/111.

[BH22] Jakub Breier and Xiaolu Hou. "How Practical Are Fault Injection Attacks, Really?" In: *IEEE Access* 10 (2022), pp. 113122–113130. DOI: 10.1109/ACCESS.2022.3217212. URL: https://doi.org/10.1109/ACCESS.2022.3217212.

[Bia+23] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. "HE3DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023, pp. 2930–2944. DOI: 10.1145/3576915.3616608. URL: https://doi.org/10.1145/3576915.3616608.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. "Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4." In: *Progress in Cryptology - AFRICACRYPT 2019*. Vol. 11627. Springer, 2019, pp. 209–228. DOI: 10.1007/978-3-030-23696-0_11.

[BNG21] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. *High-Performance Hardware Implementation of CRYSTALS-Dilithium.* Crypto. ePrint Arch., Report 2021/1451. 2021.

[Bos+21a] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. "Masking Kyber: First- and Higher-Order Implementations." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021 (2021), pp. 173–214.

[Bos+21b] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. "Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys." In: *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I.* Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. Lecture Notes in Computer Science. Springer, 2021, pp. 587–617. DOI: 10.1007/978-3-030-77870-5_21. URL: https://doi.org/10.1007/978-3-030-77870-5%5C_21.

[Bou20] Charles Bouillaguet. *Trivium.* https://github.com/cbouilla/trivium. Dec. 2020.

[BP23] Ahmad Al Badawi and Yuriy Polyakov. "Demystifying Bootstrapping in Fully Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* (2023), p. 149. URL: https://eprint.iacr.org/2023/149.

[BS97] Eli Biham and Adi Shamir. "Differential Fault Analysis of Secret Key Cryptosystems." In: *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings.* Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 513–525. DOI: 10.1007/BFb0052259. URL: https://doi.org/10.1007/BFb0052259.

[BU02] John Black and Hector Urtubia. "Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption." In: *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002.* Ed. by Dan Boneh. USENIX, 2002, pp. 327–338. URL: http://www.usenix.org/publications/library/proceedings/sec02/black.html.

[BUC19a] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.4 (2019), pp. 17–61. DOI: 10.13154/tches.v2019.i4.17-61. URL: https://doi.org/10.13154/tches.v2019.i4.17-61.

[BUC19b] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols (Extended Version)." In: *IACR Cryptol. ePrint Arch.* (2019), p. 1140.

[Bür+20] Kevin Bürstinghaus-Steinbach, Christoph KrauSS, Ruben Niederhagen, and Michael Schneider. "Post-Quantum TLS on Embedded Systems: Integrating and Evaluating Kyber and SPHINCS+ with mbed TLS." In: *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, 2020*. ACM, 2020, pp. 841–852. DOI: 10.1145/3320269.3384725.

[Can+18] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. "Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression." In: *J. Cryptol.* 31.3 (2018), pp. 885–916. DOI: 10.1007/s00145-017-9273-9. URL: https://doi.org/10.1007/s00145-017-9273-9.

[Cas+19] Benjamin M. Case, Shuhong Gao, Gengran Hu, and Qiuxia Xu. "Fully Homomorphic Encryption with k-bit Arithmetic Operations." In: *IACR Cryptol. ePrint Arch.* (2019), p. 521. URL: https://eprint.iacr.org/2019/521.

[Cas+21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. "Hardware Private Circuits: From Trivial Composition to Full Verification." In: *IEEE Trans. Computers* 70.10 (2021), pp. 1677–1690.

[Cas+23] Gaetan Cassiers, Loic Masure, Charles Momin, Thorben Moos, and François-Xavier Standaert. "Prime-Field Masking in Hardware and its Soundness against Low-Noise SCA Attacks." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.2 (2023), pp. 482–518. DOI: 10.46586/TCHES.V2023.I2.482-518. URL: https://doi.org/10.46586/tches.v2023.i2.482-518.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Improved Bootstrapping for Approximate Homomorphic Encryption." In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Springer, 2019, pp. 34–54. DOI: 10.1007/978-3-030-17656-3_2. URL: https://doi.org/10.1007/978-3-030-17656-3%5C_2.

[Cha+23] Kwanyeob Chae et al. "A 4nm 1.15TB/s HBM3 Interface with Resistor-Tuned Offset-Calibration and In-Situ Margin-Detection." In: *IEEE International Solid- State Circuits Conference, ISSCC 2023, San Francisco, CA, USA, February 19-23, 2023*. IEEE, 2023,

pp. 406–407. DOI: 10.1109/ISSCC42615.2023.10067736. URL: https://doi.org/10.1109/ISSCC42615.2023.10067736.

[Che+11] Kun-Chih Chen, Shu-Yen Lin, Wen-Chung Shen, and An-Yeu Wu. "A scalable built-in self-recovery (BISR) VLSI architecture and design methodology for 2D-mesh based on-chip networks." In: *Des. Autom. Embed. Syst.* 15.2 (2011), pp. 111–132. DOI: 10.1007/S10617-011-9074-6. URL: https://doi.org/10.1007/s10617-011-9074-6.

[Che+17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers." In: *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 409–437. DOI: 10.1007/978-3-319-70694-8_15. URL: https://doi.org/10.1007/978-3-319-70694-8%5C_15.

[Che+18a] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "A Full RNS Variant of Approximate Homomorphic Encryption." In: *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*. Ed. by Carlos Cid and Michael J. Jacobson Jr. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 347–368. DOI: 10.1007/978-3-030-10970-7_16. URL: https://doi.org/10.1007/978-3-030-10970-7%5C_16.

[Che+18b] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "A Full RNS Variant of Approximate Homomorphic Encryption." In: *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*. Ed. by Carlos Cid and Michael J. Jacobson Jr. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 347–368. DOI: 10.1007/978-3-030-10970-7_16. URL: https://doi.org/10.1007/978-3-030-10970-7%5C_16.

[Che+18c] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "Bootstrapping for Approximate Homomorphic Encryption." In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. Lecture Notes in Computer Science. Springer, 2018, pp. 360–384. DOI: 10.1007/978-3-319-78381-9_14. URL: https://doi.org/10.1007/978-3-319-78381-9%5C_14.

[Che+21] Zhaohui Chen, Emre Karabulut, Aydin Aysu, Yuan Ma, and Jiwu Jing. "An Efficient Non-Profiled Side-Channel Attack on the CRYSTALS-Dilithium Post-Quantum Signature." In: *2021 IEEE*

39th International Conference on Computer Design (ICCD). 2021, pp. 583–590. DOI: 10.1109/ICCD53106.2021.00094.

[Che+24a] Marina Checri, Renaud Sirdey, Aymen Boudguiga, and Jean-Paul Bultel. "On the Practical CPA^{D} Security of "exact" and Threshold FHE Schemes and Libraries." In: *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part III*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14922. Lecture Notes in Computer Science. Springer, 2024, pp. 3–33. DOI: 10.1007/978-3-031-68382-4_1. URL: https://doi.org/10.1007/978-3-031-68382-4%5C_1.

[Che+24b] Jung Hee Cheon, Hyeongmin Choe, Alain Passelègue, Damien Stehlé, and Elias Suvanto. "Attacks Against the IND-CPA^{D} Security of Exact FHE Schemes." In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. Ed. by Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie. ACM, 2024, pp. 2505–2519. DOI: 10.1145/3658644.3690341. URL: https://doi.org/10.1145/3658644.3690341.

[Chi+20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "TFHE: fast fully homomorphic encryption over the torus." In: *Journal of Cryptology* 33.1 (2020), pp. 34–91.

[Cho+21] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. "Transciphering Framework for Approximate Homomorphic Encryption." In: *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part III*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13092. Lecture Notes in Computer Science. Springer, 2021, pp. 640–669. DOI: 10.1007/978-3-030-92078-4_22. URL: https://doi.org/10.1007/978-3-030-92078-4%5C_22.

[Chu+21a] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. "NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 159–188. DOI: 10.46586/tches.v2021.i2.159-188. URL: https://doi.org/10.46586/tches.v2021.i2.159-188.

[Chu+21b] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. "NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2." In: *IACR Trans.*

Cryptogr. Hardw. Embed. Syst. 2021.2 (2021), pp. 159–188. DOI: 10.46586/tches.v2021.i2.159-188. URL: https://doi.org/10.46586/tches.v2021.i2.159-188.

[CKN00] Jean-Sebastien Coron, Paul C. Kocher, and David Naccache. "Statistics and Secret Leakage." In: *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*. Ed. by Yair Frankel. Vol. 1962. Lecture Notes in Computer Science. Springer, 2000, pp. 157–173. DOI: 10.1007/3-540-45472-1_12. URL: https://doi.org/10.1007/3-540-45472-1%5C_12.

[CKY18] Jung Hee Cheon, Andrey Kim, and Donggeon Yhee. "Multidimensional Packing for HEAAN for Approximate Matrix Arithmetics." In: *IACR Cryptol. ePrint Arch.* (2018), p. 1245. URL: https://eprint.iacr.org/2018/1245.

[Cla+16] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. "ASAP7: A 7-nm finFET predictive process design kit." In: *Microelectronics Journal* 53 (2016), pp. 105–115. ISSN: 0026-2692. DOI: https://doi.org/10.1016/j.mejo.2016.04.006. URL: https://www.sciencedirect.com/science/article/pii/S002626921630026X.

[Cos+22] Orel Cosseron, Clément Hoffmann, Pierrick Méaux, and François-Xavier Standaert. "Towards Globally Optimized Hybrid Homomorphic Encryption - Featuring the Elisabeth Stream Cipher." In: *IACR Cryptol. ePrint Arch.* (2022), p. 180. URL: https://eprint.iacr.org/2022/180.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks." In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28. DOI: 10.1007/3-540-36400-5_3. URL: https://doi.org/10.1007/3-540-36400-5%5C_3.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference." In: *IEEE Trans. Inf. Forensics Secur.* 15 (2020), pp. 2542–2555.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. "Provably Secure Hardware Masking in the Transition- and Glitch-Robust Probing Model: Better Safe than Sorry." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 136–158.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. "A New Algorithm for Switching from Arithmetic to Boolean Masking." In: *CHES*. Vol. 2779. Lecture Notes in Computer Science. Springer, 2003, pp. 89–97.

[Dan+19] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. "Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign." In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 206–214. DOI: 10.1109/ICFPT47387.2019.00032.

[Dan+20] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. "Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches." In: *IACR Cryptol. ePrint Arch.* (2020), p. 795. URL: https://eprint.iacr.org/2020/795.

[DAn+21] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. *SABER*. Proposal to NIST PQC Standardization, Round3. 2021.

[Deb12] Blandine Debraize. "Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking." In: *CHES*. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 107–121.

[DMG21] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. "High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber." In: *IACR Cryptol. ePrint Arch.* (2021), p. 1508.

[DMG23] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. "High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber." In: *IEEE Trans. Computers* 72.2 (2023), pp. 306–320. DOI: 10.1109/TC.2022.3222954. URL: https://doi.org/10.1109/TC.2022.3222954.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. "Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes." In: *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 369–395. DOI: 10.1007/978-3-662-53887-6_14. URL: https://doi.org/10.1007/978-3-662-53887-6%5C_14.

[Dob+18a] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. "Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit." In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I.* Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. Lecture Notes in Computer Science. Springer, 2018, pp. 662–692. DOI: 10.1007/978-3-319-96884-1_22. URL: https://doi.org/10.1007/978-3-319-96884-1%5C_22.

[Dob+18b] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. "SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 547–572. DOI: 10.13154/tches.v2018.i3.547-572. URL: https://doi.org/10.13154/tches.v2018.i3.547-572.

[Dob+21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. "Ascon v1.2: Lightweight Authenticated Encryption and Hashing." In: *J. Cryptol.* 34.3 (2021), p. 33. DOI: 10.1007/S00145-021-09398-9. URL: https://doi.org/10.1007/s00145-021-09398-9.

[Dob+23] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. "Pasta: A Case for Hybrid Homomorphic Encryption." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.3 (2023), pp. 30–73. DOI: 10.46586/tches.v2023.i3.30-73. URL: https://doi.org/10.46586/tches.v2023.i3.30-73.

[DS01] K. Davidson and S. J. Szarek. "Local operator theory, random matrices and Banach spaces." In: *Handbook of the geometry of Banach spaces, Vol. I,* 2001, pp. 317–366.

[Duc+22] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel P. J. van Woerden. "Hawk: Module LIP makes Lattice Signatures Fast, Compact and Simple." In: *IACR Cryptol. ePrint Arch.* (2022), p. 1155. URL: https://eprint.iacr.org/2022/1155.

[Dut+19] Jean-Max Dutertre, Timothée Riom, Olivier Potin, and Jean-Baptiste Rigaud. "Experimental Analysis of the Laser-Induced Instruction Skip Fault Model." In: *Secure IT Systems - 24th Nordic Conference, NordSec 2019, Aalborg, Denmark, November 18-20, 2019, Proceedings.* Ed. by Aslan Askarov, René Rydhof Hansen, and Willard Rafnsson. Vol. 11875. Lecture Notes in Computer Science. Springer, 2019, pp. 221–237. DOI: 10.1007/978-3-030-35055-0_14. URL: https://doi.org/10.1007/978-3-030-35055-0%5C_14.

[EGR23] Austin Ebel, Karthik Garimella, and Brandon Reagen. "Orion: A Fully Homomorphic Encryption Compiler for Private Deep Neural Network Inference." In: *CoRR* abs/2311.03470 (2023). DOI: 10.

48550/ARXIV.2311.03470. arXiv: 2311.03470. URL: https://doi.org/10.48550/arXiv.2311.03470.

[EH22] Anne C. Elster and Tor A. Haugdahl. "Nvidia Hopper GPU and Grace CPU Highlights." In: *Computing in Science & Engineering* 24.2 (2022), pp. 95–100. DOI: 10.1109/MCSE.2022.3163817.

[Fan+23] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. "TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU." In: *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 2023, pp. 922–934. DOI: 10.1109/HPCA56546.2023.10071017. URL: https://doi.org/10.1109/HPCA56546.2023.10071017.

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 89–120.

[Fel+21] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Christopher Peikert, and Daniel Sanchez. *F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption (Extended Version)*. 2021. arXiv: 2109.05371 [cs.CR].

[FM22] Yinxiao Feng and Kaisheng Ma. "Chiplet actuary: a quantitative cost model and multi-chiplet architecture exploration." In: *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*. Ed. by Rob Oshana. ACM, 2022, pp. 121–126. DOI: 10.1145/3489517.3530428. URL: https://doi.org/10.1145/3489517.3530428.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure integration of asymmetric and symmetric encryption schemes." In: *Annual international cryptology conference*. Springer. 1999, pp. 537–554.

[Fri+21a] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (Nov. 2021), pp. 414–460. DOI: 10.46586/tches.v2022.i1.414-460.

[Fri+21b] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography." In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 479.

[Fri+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 414–460. DOI: 10.46586/tches.v2022.i1.414-460. URL: https://doi.org/10.46586/tches.v2022.i1.414-460.

[FSS20a] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.4 (2020), pp. 239–280. DOI: 10.13154/tches.v2020.i4.239-280.

[FSS20b] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.4 (Aug. 2020), pp. 239–280. DOI: 10.13154/tches.v2020.i4.239-280. URL: https://tches.iacr.org/index.php/TCHES/article/view/8683.

[FSS20c] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.4 (2020), pp. 239–280. DOI: 10.13154/tches.v2020.i4.239-280. URL: https://doi.org/10.13154/tches.v2020.i4.239-280.

[FV12] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* (2012), p. 144. URL: http://eprint.iacr.org/2012/144.

[Gab+25] Philippe Gaborit et al. *Hamming Quasi-Cyclic (HQC)*. Selected for NIST PQC Standardization. https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf. 2025.

[Gaj20] Kris Gaj. "Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs." In: *NIST PQC Round 3 Seminars* (Oct. 2020).

[Gar+25] Karthik Garimella, Negar Neda, Austin Ebel, Nandan Kumar Jha, and Brandon Reagen. "Network and Compiler Optimizations for Efficient Linear Algebra Kernels in Private Transformer Inference (Invited Paper)." In: *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2025, Munich, Germany, October 26-30, 2025*. IEEE, 2025, pp. 1–10. DOI: 10.1109/ICCAD66269.2025.11240733. URL: https://doi.org/10.1109/ICCAD66269.2025.11240733.

[Gar59] Harvey L. Garner. "The Residue Number System." In: *IRE Trans. Electron. Comput.* 8.2 (1959), pp. 140–147. DOI: 10.1109/TEC.1959.5219515. URL: https://doi.org/10.1109/TEC.1959.5219515.

[Gee+22] Robin Geelen et al. *BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption*. 2022. DOI: 10.48550/ARXIV.2205.14017. URL: https://arxiv.org/abs/2205.14017.

[Gen09] Craig Gentry. "A fully homomorphic encryption scheme." PhD thesis. Stanford University, USA, 2009. URL: https://searchworks.stanford.edu/view/8493082.

[Geo+21] Tasopoulos George, Jinhui Li, Apostolos P. Fournaris, Raymond K. Zhao, Amin Sakzad, and Ron Steinfeld. "Performance Evaluation of Post-Quantum TLS 1.3 on Embedded Systems." In: *IACR Cryptol. ePrint Arch.* (2021), p. 1553.

[GGF22] Paul A. Grassi, Michael E. Garcia, and James L. Fenton. "Digital Identity Guidelines." In: *NIST Special Publication 800-63-3* (2022), p. 75. DOI: https://doi.org/10.6028/NIST.SP.800-63-3.

[GH11] Craig Gentry and Shai Halevi. "Implementing Gentry's Fully-Homomorphic Encryption Scheme." In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 129–148. ISBN: 978-3-642-20465-4.

[Gho+15] Shamit Ghosh, Dhiman Saha, Abhrajit Sengupta, and Dipanwita Roy Chowdhury. "Preventing Fault Attacks Using Fault Randomization with a Case Study on AES." In: *Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings*. Ed. by Ernest Foo and Douglas Stebila. Vol. 9144. Lecture Notes in Computer Science. Springer, 2015, pp. 343–355. DOI: 10.1007/978-3-319-19962-7_20. URL: https://doi.org/10.1007/978-3-319-19962-7%5C_20.

[Gho+22] Archisman Ghosh, Jose Maria Bermudo Mera, Angshuman Karmakar, Debayan Das, Santosh Ghosh, Ingrid Verbauwhede, and Shreyas Sen. "A 334uW 0.158mm2 Saber Learning with Rounding based Post-Quantum Crypto Accelerator." In: *IEEE Custom Integrated Circuits Conference, CICC 2022, Newport Beach, CA, USA, April 24-27, 2022*. IEEE, 2022, pp. 1–2. DOI: 10.1109/CICC53496.2022.9772859.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. "Homomorphic Evaluation of the AES Circuit." In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 850–867. DOI: 10.1007/978-3-642-32009-5_49. URL: https://doi.org/10.1007/978-3-642-32009-5%5C_49.

[GJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. "A testing methodology for side-channel resistance validation." In: *NIST non-invasive attack testing workshop*. Vol. 7. 2011, pp. 115–136.

[GMR21] Aurélien Greuet, Simon Montoya, and Guénaël Renault. "On Using RSA/ECC Coprocessor for Ideal Lattice-Based Key Exchange." In: *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*. Ed. by Shivam Bhasin and Fabrizio De Santis. Vol. 12910. Lecture Notes in Computer Science. Springer, 2021, pp. 205–227. DOI: 10.1007/978-3-030-89915-8_10. URL: https://doi.org/10.1007/978-3-030-89915-8%5C_10.

[Gon21] Joe L. Gonzalez. "Heterogeneous Integration of Chiplets Using Socketed Platforms, Off-Chip Flexible Interconnects, and Self-Alignment Technologies." PhD thesis. Georgia Institute of Technology, Atlanta, GA, USA, 2021. URL: https://hdl.handle.net/1853/64749.

[Goo] Google. *Medical Imaging Suite*. https://cloud.google.com/medical-imaging.

[Goo+11] Gilbert Goodwill, Benjamin Jun, Joshua Jaffe, and Pankaj Rohatgi. "A testing methodology for side channel resistance." In: 2011.

[Göt+12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss. "On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes." In: *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*. CHES'12. Leuven, Belgium: Springer-Verlag, 2012, pp. 512–529. ISBN: 9783642330261. DOI: 10.1007/978-3-642-33027-8_30. URL: https://doi.org/10.1007/978-3-642-33027-8_30.

[GPG23] Alexander Graening, Saptadeep Pal, and Puneet Gupta. "Chiplets: How Small is too Small?" In: *ACM/IEEE Design Automation Conference (DAC),* (2023).

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. "The PHOTON Family of Lightweight Hash Functions." In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Springer, 2011, pp. 222–239. DOI: 10.1007/978-3-642-22792-9_13. URL: https://doi.org/10.1007/978-3-642-22792-9%5C_13.

[Gra+23] Lorenzo Grassi, Irati Manterola Ayala, Martha Norberg Hovd, Morten Øygarden, Håvard Raddum, and Qingju Wang. "Cryptanalysis of Symmetric Primitives over Rings and a Key Recovery Attack on Rubato." In: *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part*

III. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14083. Lecture Notes in Computer Science. Springer, 2023, pp. 305–339. DOI: 10.1007/978-3-031-38548-3_11. URL: https://doi.org/10.1007/978-3-031-38548-3%5C_11.

[Gra+24] Lorenzo Grassi, Fukang Liu, Christian Rechberger, Fabian Schmid, Roman Walch, and Qingju Wang. "Minimize the Randomness in Rasta-Like Designs: How Far Can We Go?" In: *IACR Cryptol. ePrint Arch.* (2024), p. 791. URL: https://eprint.iacr.org/2024/791.

[GSR24] Anit Kumar Ghosal, Amit Sardar, and Dipanwita Roychowdhury. "Differential fault analysis attack-tolerant hardware implementation of AES." In: *J. Supercomput.* 80.4 (2024), pp. 4648–4681. DOI: 10.1007/S11227-023-05632-2. URL: https://doi.org/10.1007/s11227-023-05632-2.

[Guo+11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. "The LED Block Cipher." In: *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 326–341. DOI: 10.1007/978-3-642-23951-9_22. URL: https://doi.org/10.1007/978-3-642-23951-9%5C_22.

[Ha+20] Jincheol Ha, Seongkwang Kim, Wonseok Choi, Jooyoung Lee, Dukjae Moon, Hyojin Yoon, and Jihoon Cho. "Masta: An HE-Friendly Cipher Using Modular Arithmetic." In: *IEEE Access* 8 (2020), pp. 194741–194751. DOI: 10.1109/ACCESS.2020.3033564. URL: https://doi.org/10.1109/ACCESS.2020.3033564.

[Ha+22] Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Jooyoung Lee, and Mincheol Son. "Rubato: Noisy Ciphers for Approximate Homomorphic Encryption." In: *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13275. Lecture Notes in Computer Science. Springer, 2022, pp. 581–610. DOI: 10.1007/978-3-031-06944-4_20. URL: https://doi.org/10.1007/978-3-031-06944-4%5C_20.

[Han+19] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. "Logistic Regression on Homomorphic Encrypted Data at Scale." In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI

Press, 2019, pp. 9466–9471. DOI: 10.1609/AAAI.V33I01.33019466. URL: https://doi.org/10.1609/aaai.v33i01.33019466.

[HB21] Vedad Hadzic and Roderick Bloem. "COCOALMA: A Versatile Masking Verifier." In: *FMCAD*. IEEE, 2021, pp. 1–10.

[HC23] HEAAN and Cryptolab. *heaan, C++ Interface to 'HEaaN'*. Tech. rep. HEaaN.stat, CryptoLab, 2023. URL: https://github.com/snucrypto/HEAAN?tab=readme-ov-file.

[Hel+21] Martin Hell, Thomas Johansson, Alexander Maximov, Willi Meier, and Hirotaka Yoshida. "Grain-128AEADv2: Strengthening the Initialization Against Key Reconstruction." In: *Cryptology and Network Security - 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings*. Ed. by Mauro Conti, Marc Stevens, and Stephan Krenn. Vol. 13099. Lecture Notes in Computer Science. Springer, 2021, pp. 24–41. DOI: 10.1007/978-3-030-92548-2_2. URL: https://doi.org/10.1007/978-3-030-92548-2%5C_2.

[Hir+24] Haruka Hirata, Daiki Miyahara, Victor Arribas, Yang Li, Noriyuki Miura, Svetla Nikova, and Kazuo Sakiyama. "All You Need Is Fault: Zero-Value Attacks on AES and a New λ-Detection M&M." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 133–156. DOI: 10.46586/TCHES.V2024.I1.133-156. URL: https://doi.org/10.46586/tches.v2024.i1.133-156.

[HK19] Kyoohyung Han and Dohyeong Ki. *Better Bootstrapping for Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Report 2019/688. https://ia.cr/2019/688. 2019.

[HLX] Pengzhou He, Chiou-Yng Lee, and Jiafeng Xie. "Compact Coprocessor for KEM Saber: Novel Scalable Matrix Originated Processing." In: ().

[HPS06] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. "NTRU: A ring-based public key cryptosystem." In: *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*. Springer, 2006, pp. 267–288.

[HS14] Shai Halevi and Victor Shoup. "Algorithms in HElib." In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. Lecture Notes in Computer Science. Springer, 2014, pp. 554–571. DOI: 10.1007/978-3-662-44371-2_31. URL: https://doi.org/10.1007/978-3-662-44371-2%5C_31.

[HS18] Shai Halevi and Victor Shoup. "Faster Homomorphic Linear Transformations in HElib." In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. Lecture Notes in

Computer Science. Springer, 2018, pp. 93–120. DOI: 10.1007/978-3-319-96884-1_4. URL: https://doi.org/10.1007/978-3-319-96884-1%5C_4.

[HS20] Shai Halevi and Victor Shoup. *Design and implementation of HElib: a homomorphic encryption library*. Cryptology ePrint Archive, Paper 2020/1481. 2020. URL: https://eprint.iacr.org/2020/1481.

[Hua+20] Yiming Huang, Miaoqing Huang, Zhongkui Lei, and Jiaxuan Wu. "A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse." In: *IEICE Electron. Express* 17.17 (2020), p. 20200234. DOI: 10.1587/elex.17.20200234.

[Hwa+20] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. "Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations." In: *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 968–981. DOI: 10.1109/ISCA45697.2020.00083. URL: https://doi.org/10.1109/ISCA45697.2020.00083.

[HZ23] Hai Huang and Haoran Zong. "Secure matrix multiplication based on fully homomorphic encryption." In: *J. Supercomput.* 79.5 (2023), pp. 5064–5085. DOI: 10.1007/S11227-022-04850-4. URL: https://doi.org/10.1007/s11227-022-04850-4.

[Ibe] Ibex. *Ibex RISC-V Core*. URL: www.github.com/lowRISC/ibex.

[IBM20] IBM. "IBM Cost of a Data Breach 2022 Highlights for Cloud Security Professionals." In: *Technical Report* (2020).

[IBM24] IBM Security. *Cost of a Data Breach Report 2024*. Report (research data compiled by Ponemon Institute). Accessed: 2026-01-13. 2024. URL: https://cdn.table.media/assets/wp-content/uploads/2024/07/30132828/Cost-of-a-Data-Breach-Report-2024.pdf.

[IBM25] IBM Security. *Cost of a Data Breach Report 2025*. Report (research data compiled by Ponemon Institute). Accessed: 2026-01-13. 2025. URL: https://webobjects2.cdw.com/is/content/CDW/cdw/on-domain-cdw/brands/ibm/cost-of-a-data-breach-2025-full-report.pdf.

[Imr+21] Malik Imran, Felipe Almeida, Jaan Raik, Andrea Basso, Sujoy Sinha Roy, and Samuel Pagliarini. *Design Space Exploration of SABER in 65nm ASIC*. 2021. arXiv: 2109.07824 [cs.CR].

[Imr+23] Malik Imran, Aikata Aikata, Sujoy Sinha Roy, and Samuel N. Pagliarini. "Towards High-speed ASIC Implementations of Post-Quantum Cryptography." In: *IACR Cryptol. ePrint Arch.* (2023), p. 716. URL: https://eprint.iacr.org/2023/716.

[Imr+24] Malik Imran, Aikata Aikata, Sujoy Sinha Roy, and Samuel Pagliarini. "High-Speed Design of Post Quantum Cryptography With Optimized Hashing and Multiplication." In: *IEEE Trans. Circuits Syst. II Express Briefs* 71.2 (2024), pp. 847–851. DOI: 10.1109/TCSII.2023.3273821. URL: https://doi.org/10.1109/TCSII.2023.3273821.

[Inf] Infineon. "OPTIGA TPM SLB 9672 from Infenion." In: (). URL: https://www.infineon.com/cms/en/about-infineon/press/market-news/2022/INFCSS202202-051.html.

[Int23] Intel. *Intel Xeon CPU Max 9400*. Tech. rep. Intel, 2023. URL: https://geizhals.at/intel-xeon-cpu-max-9400-sockel-4677-v125548.html.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks." In: *CRYPTO*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[Jan+22] Jaehee Jang, Younho Lee, Andrey Kim, Byunggook Na, Donggeon Yhee, Byounghan Lee, Jung Hee Cheon, and Sungroh Yoon. "Privacy-Preserving Deep Sequential Model with Matrix Homomorphic Encryption." In: *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*. Ed. by Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako. ACM, 2022, pp. 377–391. DOI: 10.1145/3488932.3523253. URL: https://doi.org/10.1145/3488932.3523253.

[JED22] JEDEC. "High Bandwidth Memory DRAM (HBM3)." In: *Tech. Rep. JESD238* (2022).

[Jia+18] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. "Secure Outsourced Matrix Computation and Application to Neural Networks." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1209–1222. DOI: 10.1145/3243734.3243837. URL: https://doi.org/10.1145/3243734.3243837.

[JKB95] Norman L Johnson, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous univariate distributions, volume 2*. Vol. 289. John wiley & sons, 1995.

[Ju+23] Jae Hyung Ju, Jaiyoung Park, Jongmin Kim, Donghwan Kim, and Jung Ho Ahn. "NeuJeans: Private Neural Network Inference with Joint Optimization of Convolution and Bootstrapping." In: *CoRR* abs/2312.04356 (2023). DOI: 10.48550/ARXIV.2312.04356. arXiv: 2312.04356. URL: https://doi.org/10.48550/arXiv.2312.04356.

[Jun+21a] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (Aug. 2021), pp. 114–148. DOI: 10.46586/tches.v2021.i4.114-148. URL: https://tches.iacr.org/index.php/TCHES/article/view/9062.

[Jun+21b] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 114–148. DOI: 10.46586/TCHES.V2021.I4.114-148. URL: https://doi.org/10.46586/tches.v2021.i4.114-148.

[Jun+21c] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. "Accelerating Fully Homomorphic Encryption Through Architecture-Centric Analysis and Optimization." In: *IEEE Access* 9 (2021), pp. 98772–98789. DOI: 10.1109/ACCESS.2021.3096189. URL: https://doi.org/10.1109/ACCESS.2021.3096189.

[Kan+19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*. Crypto. ePrint Arch., Report 2019/844. 2019.

[Kim+18] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. "Logistic Regression Model Training based on the Approximate Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* (2018), p. 254. URL: http://eprint.iacr.org/2018/254.

[Kim+22a] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. *ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse*. 2022. DOI: 10.48550/ARXIV.2205.00922. URL: https://arxiv.org/abs/2205.00922.

[Kim+22b] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. "BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption." In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 711–725. ISBN: 9781450386104. DOI: 10.1145/3470496.3527415. URL: https://doi.org/10.1145/3470496.3527415.

[Kim+23a] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. "SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption." In: *Proceedings of the 50th Annual International Symposium*

on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023. Ed. by Yan Solihin and Mark A. Heinrich. ACM, 2023, 18:1–18:15. DOI: 10.1145/3579371.3589053. URL: https://doi.org/10.1145/3579371.3589053.

[Kim+23b] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. "Asymptotically Faster Multi-Key Homomorphic Encryption from Homomorphic Gadget Decomposition." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023, pp. 726–740. DOI: 10.1145/3576915.3623176. URL: https://doi.org/10.1145/3576915.3623176.

[Kim+23c] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. "Asymptotically Faster Multi-Key Homomorphic Encryption from Homomorphic Gadget Decomposition." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023, pp. 726–740. DOI: 10.1145/3576915.3623176. URL: https://doi.org/10.1145/3576915.3623176.

[Kim+24a] Andrey Kim, Ahmet Can Mert, Anisha Mukherjee, Aikata Aikata, Maxim Deryabin, Sunmin Kwon, HyungChul Kang, and Sujoy Sinha Roy. "Exploring the Advantages and Challenges of Fermat NTT in FHE Acceleration." In: *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part III*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14922. Lecture Notes in Computer Science. Springer, 2024, pp. 76–106. DOI: 10.1007/978-3-031-68382-4_3. URL: https://doi.org/10.1007/978-3-031-68382-4%5C_3.

[Kim+24b] Sangpyo Kim, Jongmin Kim, Jaeyoung Choi, and Jung Ho Ahn. "CiFHER: A Chiplet-Based FHE Accelerator with a Resizable Structure." In: *International Symposium on Secure and Private Execution Environment Design, SEED 2024, Orlando, FL, USA, May 16-17, 2024*. IEEE, 2024, pp. 119–130. DOI: 10.1109/SEED61283.2024.00022. URL: https://doi.org/10.1109/SEED61283.2024.00022.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. URL: https://doi.org/10.1007/3-540-48405-1%5C_25.

[KKW08] Caroline Kim, John Y. Kim, and Dr. Guofang Wei. "Invertibility Probability of Binary Matrices Team." In: *Team*. 2008. URL: https://api.semanticscholar.org/CorpusID:15037510.

[KO62] A Karatsuba and Yu Ofman. "Multiplication of many-digital numbers by automatic computers." In: *Dokl. Akad. Nauk SSSR* 145.2 (1962), pp. 293–294. URL: http://mi.mathnet.ru/dan26729.

[KPP22] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. "Approximate Homomorphic Encryption with Reduced Approximation Error." In: *Topics in Cryptology - CT-RSA 2022 - Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings*. Ed. by Steven D. Galbraith. Vol. 13161. Lecture Notes in Computer Science. Springer, 2022, pp. 120–144. DOI: 10.1007/978-3-030-95312-6_6. URL: https://doi.org/10.1007/978-3-030-95312-6%5C_6.

[Kri+21] Gokul Krishnan, Sumit K. Mandal, Manvitha Pannala, Chaitali Chakrabarti, Jae-Sun Seo, Ümit Y. Ogras, and Yu Cao. "SIAM: Chiplet-based Scalable In-Memory Acceleration with Mesh for Deep Neural Networks." In: *ACM Trans. Embed. Comput. Syst.* 20.5s (2021), 68:1–68:24. DOI: 10.1145/3476999. URL: https://doi.org/10.1145/3476999.

[Kri+24] Florian Krieger, Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. "Aloha-HE: A Low-Area Hardware Accelerator for Client-Side Operations in Homomorphic Encryption." In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2024, pp. 1–6.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. "SILVER - Statistical Independence and Leakage Verification." In: *ASIACRYPT (1)*. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[Kun+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. "Higher-Order Masked Saber." In: *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*. Ed. by Clemente Galdi and Stanislaw Jarecki. Vol. 13409. Lecture Notes in Computer Science. Springer, 2022, pp. 93–116. DOI: 10.1007/978-3-031-14791-3_5. URL: https://doi.org/10.1007/978-3-031-14791-3%5C_5.

[Kun+24] Anup Kumar Kundu, Shibam Ghosh, Aikata Aikata, and Dhiman Saha. "ToFA: Towards Fault Analysis of GIFT and GIFT-like Ciphers Leveraging Truncated Impossible Differentials." In: *IACR Cryptol. ePrint Arch.* (2024), p. 1927. URL: https://eprint.iacr.org/2024/1927.

[Lag+21] Brent Lagesse, Gabriel Nguyen, Utsav Goswami, and Kevin Wu. "You Had to Be There: Private Video Sharing for Mobile Phones using Fully Homomorphic Encryption." In: *19th IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events, PerCom Workshops 2021, Kassel, Germany, March 22-26, 2021*. IEEE, 2021, pp. 730–735. DOI: 10.1109/PERCOMWORKSHOPS51409.2021.9431029. URL: https://doi.org/10.1109/PerComWorkshops51409.2021.9431029.

[Lag+25] Svenja Lage, Felicitas Hörmann, Felix Hanke, and Michael Karl. "Collision Risk Analysis for LEO Satellites with Confidential Orbital Data." In: *CoRR* abs/2501.09397 (2025). DOI: 10.48550/ARXIV.2501.09397. arXiv: 2501.09397. URL: https://doi.org/10.48550/arXiv.2501.09397.

[LC10] Yann LeCun and Corinna Cortes. *MNIST handwritten digit database*. http://yann.lecun.com/exdb/mnist/. 2010. URL: http://yann.lecun.com/exdb/mnist/.

[LDL23] Jaehyeok Lee, Phap Duong-Ngoc, and Hanho Lee. "Configurable Encryption and Decryption Architectures for CKKS-Based Homomorphic Encryption." In: *Sensors* 23.17 (2023), p. 7389. DOI: 10.3390/S23177389. URL: https://doi.org/10.3390/s23177389.

[Lee+15] Dong-Uk Lee, Kang Seol Lee, Yongwoo Lee, Kyung Whan Kim, Jong-Ho Kang, Jaejin Lee, and Jun Hyun Chun. "Design considerations of HBM stacked DRAM and the memory architecture extension." In: *2015 IEEE Custom Integrated Circuits Conference, CICC 2015, San Jose, CA, USA, September 28-30, 2015*. IEEE, 2015, pp. 1–8. DOI: 10.1109/CICC.2015.7338357. URL: https://doi.org/10.1109/CICC.2015.7338357.

[Li+09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures." In: *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*. Ed. by David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez. ACM, 2009, pp. 469–480. DOI: 10.1145/1669112.1669172. URL: https://doi.org/10.1145/1669112.1669172.

[Liu+19] Weiqiang Liu, Sailong Fan, Ayesha Khalid, Ciara Rafferty, and Máire ONeill. "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.10 (2019), pp. 2459–2463.

[Liu+21] Fukang Liu, Santanu Sarkar, Willi Meier, and Takanori Isobe. "Algebraic Attacks on Rasta and Dasta Using Low-Degree Equations." In: *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part I*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13090. Lecture Notes in Computer Science. Springer, 2021, pp. 214–240. DOI: 10.1007/978-3-030-92062-3_8. URL: https://doi.org/10.1007/978-3-030-92062-3%5C_8.

[LKS17] Wenjie Lu, Shohei Kawasaki, and Jun Sakuma. "Using Fully Homomorphic Encryption for Statistical Analysis of Categorical, Ordinal and Numerical Data." In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[LM21] Baiyu Li and Daniele Micciancio. "On the Security of Homomorphic Encryption on Approximate Numbers." In: *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. Lecture Notes in Computer Science. Springer, 2021, pp. 648–677. DOI: 10.1007/978-3-030-77870-5_23. URL: https://doi.org/10.1007/978-3-030-77870-5%5C_23.

[LN10] Wei Liu and Alberto Nannarelli. "Power dissipation challenges in multicore floating-point units." In: *21st IEEE International Conference on Application-specific Systems Architectures and Processors, ASAP 2010, Rennes, France, 7-9 July 2010*. Ed. by François Charot, Frank Hannig, Jürgen Teich, and Christophe Wolinski. IEEE Computer Society, 2010, pp. 257–264. DOI: 10.1109/ASAP.2010.5540986. URL: https://doi.org/10.1109/ASAP.2010.5540986.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings." In: *Advances in Cryptology EUROCRYPT 2010*. Vol. 6110. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 1–23. ISBN: 978-3-642-13189-9.

[LS15] Adeline Langlois and Damien Stehlé. "Worst-case to average-case reductions for module lattices." In: *Designs, Codes and Cryptography* 75.3 (2015), pp. 565–599.

[LS19] Vadim Lyubashevsky and Gregor Seiler. "NTTRU: Truly Fast NTRU Using NTT." In: *IACR Trans. on CHES* 2019.3 (May 2019), pp. 180–201. DOI: 10.13154/tches.v2019.i3.180-201.

[LSG21] Georg Land, Pascal Sasdrich, and Tim Guneysu. "A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware." In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 355.

[LSL20] James Andrew Lewis, Zhanna L. Malekos Smith, and Eugenia Lostri. "The Hidden Costs of Cybercrime." In: *Technical Report* (2020).

[Lu+18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. "LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus." In: *IACR Cryptol. ePrint Arch.* (2018), p. 1009.

[Lu+23] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. "cuZK: Accelerating Zero-Knowledge Proof with A Faster Parallel Multi-Scalar Multiplication Algorithm on GPUs." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.3 (2023), pp. 194–220. DOI: 10.46586/TCHES.V2023.I3.194-220. URL: https://doi.org/10.46586/tches.v2023.i3.194-220.

[LYK21] Duc-Phong Le, Sze Ling Yeo, and Khoongming Khoo. "Algebraic Differential Fault Analysis on SIMON block cipher." In: *IACR Cryptol. ePrint Arch.* (2021), p. 436. URL: https://eprint.iacr.org/2021/436.

[Ma+22] Xiaohan Ma, Ying Wang, Yujie Wang, Xuyi Cai, and Yinhe Han. "Survey on chiplets: interface, interconnect and integration methodology." In: *CCF Trans. High Perform. Comput.* 4.1 (2022), pp. 43–52. DOI: 10.1007/s42514-022-00093-0. URL: https://doi.org/10.1007/s42514-022-00093-0.

[Mah+23] Rasoul Akhavan Mahdavi, Haoyan Ni, Dimitry Linkov, and Florian Kerschbaum. "Level Up: Private Non-Interactive Decision Tree Evaluation using Levelled Homomorphic Encryption." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023, pp. 2945–2958. DOI: 10.1145/3576915.3623095. URL: https://doi.org/10.1145/3576915.3623095.

[Man22] Tobias Mann. *AMD was right about chiplets, Intel's Gelsinger all but says*. Tech. rep. AMD, 2022. URL: https://www.theregister.com/2022/09/28/intel_chiplets_advanced_packaging/.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems (Video Classification)*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/tutorials/video/video%5C_classification.

[Mar+20] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. "Compact domain-specific co-processor for accelerating module lattice-based KEM." In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218727.

[Mav17] Arda Mavi. *Sign Language Digits Dataset*. 2017. DOI: 10.34740/KAGGLE/DSV/11071. URL: https://www.kaggle.com/dsv/11071.

[May00] Rita Mayer-Sommer. "Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards." In: *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1965. Lecture Notes in Computer Science. Springer, 2000, pp. 78–92. DOI: 10.1007/3-540-44499-8_6. URL: https://doi.org/10.1007/3-540-44499-8%5C_6.

[Méa+16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. "Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts." In: *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. Lecture Notes in Computer Science. Springer, 2016, pp. 311–343. DOI: 10.1007/978-3-662-49890-3_13. URL: https://doi.org/10.1007/978-3-662-49890-3%5C_13.

[Méa+19] Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. "Improved Filter Permutators for Efficient FHE: Better Instances and Implementations." In: *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*. Ed. by Feng Hao, Sushmita Ruj, and Sourav Sen Gupta. Vol. 11898. Lecture Notes in Computer Science. Springer, 2019, pp. 68–91. DOI: 10.1007/978-3-030-35423-7_4. URL: https://doi.org/10.1007/978-3-030-35423-7%5C_4.

[Men+20] Alexandre Menu, Jean-Max Dutertre, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Luc Danger. "Experimental Analysis of the Electromagnetic Instruction Skip Fault Model." In: *15th Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2020, Marrakech, Morocco, April 1-3, 2020*. IEEE, 2020, pp. 1–7. DOI: 10.1109/DTIS48698.2020.9081261. URL: https://doi.org/10.1109/DTIS48698.2020.9081261.

[Men+24] Vignesh V. Menon, Amritha Premkumar, Prajit T. Rajendran, Adam Wieckowski, Benjamin Bross, Christian Timmerer, and Detlev Marpe. "Energy-efficient Adaptive Video Streaming with Latency-Aware Dynamic Resolution Encoding." In: *Proceedings of the 3rd Mile-High Video Conference, MHV 2024, Denver, CO, USA, February 11-14, 2024*. ACM, 2024, pp. 21–27. DOI: 10.1145/3638036.3640801. URL: https://doi.org/10.1145/3638036.3640801.

[Mer+21] Jose Maria Bermudo Mera, Angshuman Karmakar, Suparna Kundu, and Ingrid Verbauwhede. "Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 474–509. DOI: 10.46586/tches.v2021.i4.474-509. URL: https://doi.org/10.46586/tches.v2021.i4.474-509.

[Mer+23] Ahmet Can Mert, Aikata Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. "Medha: Microcoded Hardware Accelerator for computing on Encrypted Data." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.1 (2023), pp. 463–500. DOI: 10.46586/tches.v2023.i1.463-500. URL: https://doi.org/10.46586/tches.v2023.i1.463-500.

[MGS23] Stefano Di Matteo, Matteo Lo Gerfo, and Sergio Saponara. "VLSI Design and FPGA Implementation of an NTT Hardware Accelerator for Homomorphic SEAL-Embedded Library." In: *IEEE Access* 11 (2023), pp. 72498–72508. DOI: 10.1109/ACCESS.2023.3295245. URL: https://doi.org/10.1109/ACCESS.2023.3295245.

[Mig+19] Vincent Migliore, Benoit Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. *Masking Dilithium: Efficient Implementation and Side-Channel Evaluation.* Crypto. ePrint Arch., Report 2019/394. 2019.

[Min] MindNetwork. *FHE for Web3.* URL: https://agent.mindnetwork.xyz/agenticworld/2.

[Mon85] Peter L Montgomery. "Modular multiplication without trial division." In: *Mathematics of computation* 44.170 (1985), pp. 519–521.

[MOP07a] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007. ISBN: 978-0-387-30857-9.

[MOP07b] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007. ISBN: 978-0-387-30857-9.

[Mor+16] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. *RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2.* RFC Editor. Accessed: 2026-01-13. 2016. URL: https://www.rfc-editor.org/rfc/rfc8017.

[Mor20] Steve Morgan. "McAfee Vastly Underestimates The Cost Of Cybercrime." In: *Cybersecurity Report* (2020).

[MÖS19] Ahmet Can Mert, Erdinç Öztürk, and Erkay Sava. "Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture." In: *2019 22nd Euromicro Conference on Digital System Design (DSD).* IEEE. 2019, pp. 253–260.

[MÖS20] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savas. "Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme." In: *IEEE Trans. Very Large Scale Integr. Syst.* 28.2 (2020), pp. 353–362. DOI: 10.1109/TVLSI.2019.2943127. URL: https://doi.org/10.1109/TVLSI.2019.2943127.

[MR24] Pierrick Méaux and Dibyendu Roy. "Theoretical differential fault attacks on FLIP and FiLIP." In: *Cryptogr. Commun.* 16.4 (2024), pp. 721–744. DOI: 10.1007/S12095-024-00698-Y. URL: https://doi.org/10.1007/s12095-024-00698-y.

[Muk+24] Anisha Mukherjee, Aikata Aikata, Ahmet Can Mert, Yongwoo Lee, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. "ModHE: Modular Homomorphic Encryption Using Module Lattices Potentials and Limitations." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 527–562. DOI: 10.46586/TCHES.V2024.I1.527-562. URL: https://doi.org/10.46586/tches.v2024.i1.527-562.

[Muk09] Debdeep Mukhopadhyay. "An Improved Fault Based Attack of the Advanced Encryption Standard." In: *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings.* Ed. by Bart Preneel. Vol. 5580. Lecture Notes in Computer Science. Springer, 2009, pp. 421–434. DOI: 10.1007/978-3-642-02384-2_26. URL: https://doi.org/10.1007/978-3-642-02384-2%5C_26.

[Mul24] Multiple. Based on the public domain implementation in crypto_hash/keccakc512/simple/ from http://bench.cr.yp.to/supercop.html by Ronny Van Keer and the public domain "TweetFips202" implementation from https://twitter.com/tweetfips202 by Gilles Van Assche, Daniel J. Bernstein, and Peter Schwabe. 2024.

[Mur23] Brett Murdock. "What Designers Need to Know About HBM3." In: *Synopsys* (2023). URL: https://www.synopsys.com/designware-ip/technical-bulletin/hbm3-ip-dwtb.html.

[MUSa] MUSE Semiconductor. *TSMC MPW Full Block Tapeouts.* https://www.musesemi.com/full-block-tapeout-pricing. Accessed July 5th 2022. (Visited on 07/11/2022).

[MUSb] MUSE Semiconductor. *TSMC UNIVERSITY FINFET PROGRAM.* https://www.musesemi.com/university-finfet-program. Accessed July 27th 2023. (Visited on 07/26/2023).

[Nab+22] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, Mizan Abraha Gebremichael, Homer Gamil, Eduardo Chielle, Ramesh Karri, Mihai Sanduleanu, and Michail Maniatakos. *CoFHEE: A Co-processor for Fully Homomorphic Encryption Execution.* 2022. DOI: 10.48550/ARXIV.2204.08742. URL: https://arxiv.org/abs/2204.08742.

[Nar+17] S. Narasimha et al. “A 7nm CMOS technology platform for mobile and high performance compute application.” In: *2017 IEEE International Electron Devices Meeting (IEDM)*. 2017, pp. 29.5.1–29.5.4. DOI: 10.1109/IEDM.2017.8268476.

[Nat23] National Institute of Standards and Technology (NIST). *FIPS 186-5: Digital Signature Standard (DSS)*. NIST CSRC. Accessed: 2026-01-13. 2023. URL: https://csrc.nist.gov/pubs/fips/186-5/final.

[ND21] Deepika Natarajan and Wei Dai. “SEAL-Embedded: A Homomorphic Encryption Library for the Internet of Things.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 756–779. DOI: 10.46586/TCHES.V2021.I3.756-779. URL: https://doi.org/10.46586/tches.v2021.i3.756-779.

[Ni+22] Ziying Ni, Ayesha Khalid, Dur-e-Shahwar Kundi, Máire O’Neill, and Weiqiang Liu. “Efficient Pipelining Exploration for A High-performance CRYSTALS-Kyber Accelerator.” In: *IACR Cryptol. ePrint Arch.* (2022), p. 1093.

[NIS01] NIST. “FIPS pub 197: Advanced encryption standard (AES).” In: *p. 311* (2001). URL: https://doi.org/10.6028/NIST.FIPS.197.

[NIS17] NIST. “Post-Quantum Cryptography- Call for proposals.” In: (2017). URL: https://csrc.nist.gov/projects/post-quantum-cryptography.

[Niu+25] Chao Niu, Benqiang Wei, Zhicong Huang, Zhaomin Yang, Cheng Hong, Meiqin Wang, and Tao Wei. *SoK: FHE-Friendly Symmetric Ciphers and Transciphering*. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2025. URL: https://eprint.iacr.org/2025/669.

[NXP] NXP. “NXPs C29x family of crypto coprocessors.” In: (). URL: https://www.nxp.com/docs/en/fact-sheet/C29XFAMFS.pdf.

[Ode+18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. “Practical CCA2-Secure and Masked Ring-LWE Implementation.” In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.1 (Feb. 2018), pp. 142–174. DOI: 10.13154/tches.v2018.i1.142-174. URL: https://tches.iacr.org/index.php/TCHES/article/view/836.

[PA11] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. *The Keccak reference*. Round 3 submission to NIST SHA-3. 2011. URL: http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[Pag+24] Samuel Pagliarini, Aikata Aikata, Malik Imran, and Sujoy Sinha Roy. “REPQC: Reverse Engineering and Backdooring Hardware Accelerators for Post-quantum Cryptography.” In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. Ed. by

Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro A. Cárdenas. ACM, 2024. DOI: 10.1145/3634737.3657016. URL: https://doi.org/10.1145/3634737.3657016.

[PAL21] PALISADE. *PALISADE Lattice Cryptography Library (release 1.11.2)*. https://palisade-crypto.org/. May 2021.

[Pap+18] Nicolas Papernot, Patrick D. McDaniel, Arunesh Sinha, and Michael P. Wellman. "SoK: Security and Privacy in Machine Learning." In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 399–414. DOI: 10.1109/EUROSP.2018.00035. URL: https://doi.org/10.1109/EuroSP.2018.00035.

[Par+23] Myeong-Jae Park et al. "A 192-Gb 12-High 896-GB/s HBM3 DRAM With a TSV Auto-Calibration Scheme and Machine-Learning-Based Layout Optimization." In: *IEEE J. Solid State Circuits* 58.1 (2023), pp. 256–269. DOI: 10.1109/JSSC.2022.3193354. URL: https://doi.org/10.1109/JSSC.2022.3193354.

[Pau+23] Gianna Paulin et al. " Occamy: A 432-core, Multi-TFLOPs RISC-V-Based 2.5D Chiplet System for Ultra-Efficient (Mini-)Floating-Point Computation." In: *PULP Platform, ETH Zurich* (2023). URL: https://pulp-platform.org/occamy/.

[PDC21] Sikha Pentyala, Rafael Dowsley, and Martine De Cock. "Privacy-Preserving Video Classification with Convolutional Neural Networks." In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 8487–8499. URL: http://proceedings.mlr.press/v139/pentyala21a.html.

[Por19] Thomas Pornin. "New Efficient, Constant-Time Implementations of Falcon." In: *IACR Cryptol. ePrint Arch.* (2019), p. 893.

[Pre+21] Thomas Prest et al. *FALCON*. Proposal to NIST PQC Standardization, Round3. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions. 2021.

[PTP15] Alberto Pedrouzo-Ulloa, Juan Ramón Troncoso-Pastoriza, and Fernando Pérez-González. "Multivariate lattices for encrypted image processing." In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*. IEEE, 2015, pp. 1707–1711. DOI: 10.1109/ICASSP.2015.7178262. URL: https://doi.org/10.1109/ICASSP.2015.7178262.

[Rad+23] R. Radheshwar, Meenakshi Kansal, Pierrick Méaux, and Dibyendu Roy. "Differential Fault Attack on Rasta and FiLIP_{DSM}." In: *IEEE Trans. Computers* 72.8 (2023), pp. 2418–2425. DOI: 10.1109/TC.2023.3244629. URL: https://doi.org/10.1109/TC.2023.3244629.

[RAD78] R L Rivest, L Adleman, and M L Dertouzos. “On Data Banks and Privacy Homomorphisms.” In: *Foundations of Secure Computation, Academia Press* (1978), pp. 169–179.

[Ram] Rambus. *HBM3 Memory: Break Through to Greater Bandwidth.* URL: https://go.rambus.com/hbm3-memory-break-through-to-greater-bandwidth.

[RB20] Sujoy Sinha Roy and Andrea Basso. “High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware.” In: *IACR Trans. Crypt. Hardw. Embed. Syst.* 2020.4 (2020), pp. 443–466. DOI: 10.13154/tches.v2020.i4.443-466.

[RBM21] Dibyendu Roy, Bhagwan N. Bathe, and Subhamoy Maitra. “Differential Fault Attack on Kreyvium & FLIP.” In: *IEEE Trans. Computers* 70.12 (2021), pp. 2161–2167. DOI: 10.1109/TC.2020.3038236. URL: https://doi.org/10.1109/TC.2020.3038236.

[Rea+20] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S. Lee, and David Brooks. *Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference.* 2020. DOI: 10.48550/ARXIV.2006.00505. URL: https://arxiv.org/abs/2006.00505.

[Reg09] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography.” In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.

[Rep+15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. “A Masked Ring-LWE Implementation.” In: *CHES*. Springer, 2015, pp. 683–702. DOI: 10.1007/978-3-662-48324-4_34. URL: https://www.iacr.org/archive/ches2015/92930660/92930660.pdf.

[Res+] Midia Reshadi, Ahmad Khademzadeh, Akram Reza, and Maryam Bahmani. *A Novel Mesh Architecture for On-Chip Networks.* URL: https://www.design-reuse.com/articles/23347/on-chip-network.html.

[Res18] Eric Rescorla. *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3.* RFC Editor. 2018. URL: https://www.rfc-editor.org/rfc/rfc8446.

[Ria+20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. “HEAX: An Architecture for Computing on Encrypted Data.” In: *ASPLOS ’20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020.* Ed. by James R. Larus, Luis Ceze, and Karin Strauss. ACM, 2020, pp. 1295–1309. DOI: 10.1145/3373376.3378523. URL: https://doi.org/10.1145/3373376.3378523.

[Ric+21] Sara Ricci, Lukas Malina, Petr Jedlicka, David Smékal, Jan Hajny, Peter Cibik, Petr Dzurenda, and Patrik Dobias. "Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs." In: *The 16th International Conference on Availability, Reliability and Security*. ARES 2021. Vienna, Austria: Association for Computing Machinery, 2021. ISBN: 9781450390514. DOI: 10.1145/3465481.3465756.

[Rob24] United Kingdom Robert Dimond System Architect ARM. *Keynote: CHIPLET STANDARDS: A NEW ROUTE TO ARM-BASED CUSTOM SILICON*. Tech. rep. ARM, 2024. URL: https://www.date-conference.com/node/1750.

[Roy+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. "Compact Ring-LWE Cryptoprocessor." In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391. ISBN: 978-3-662-44709-3.

[Roy+15] S. Sinha Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede. "Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation." In: *Cryptographic Hardware and Embedded Systems - CHES*. 2015.

[Roy+18] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede. "HEPCloud: An FPGA-based Multicore Processor for FV Somewhat Homomorphic Function Evaluation." In: *IEEE Transactions on Computers* (2018).

[RT22] Panagiotis Rizomiliotis and Aikaterini Triakosia. "On Matrix Multiplication with Homomorphic Encryption." In: *Proceedings of the 2022 on Cloud Computing Security Workshop, CCSW 2022, Los Angeles, CA, USA, 7 November 2022*. Ed. by Francesco Regazzoni and Marten van Dijk. ACM, 2022, pp. 53–61. DOI: 10.1145/3560810.3564267. URL: https://doi.org/10.1145/3560810.3564267.

[Rud06] Mark Rudelson. "Norm of the inverse of a random matrix." In: *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*. IEEE Computer Society, 2006, pp. 487–496. DOI: 10.1109/FOCS.2006.52. URL: https://doi.org/10.1109/FOCS.2006.52.

[Sam+22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. "CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data." In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 173–187. ISBN: 9781450386104. DOI: 10.1145/3470496.3527393. URL: https://doi.org/10.1145/3470496.3527393.

[Sch+21] Peter Schwabe et al. *CRYSTALS-KYBER*. Selected for NIST PQC Standardization. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions. 2021.

[Sco17a] Michael Scott. "A Note on the Implementation of the Number Theoretic Transform." In: *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*. Springer, 2017, pp. 247–258.

[Sco17b] Michael Scott. "A Note on the Implementation of the Number Theoretic Transform." In: *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*. Springer, 2017, pp. 247–258.

[Sco20] Michael Scott. *Linear regression - Insurance dataset*. 2020. URL: https://www.kaggle.com/code/kianwee/linear-regression-insurance-dataset.

[Sha+21a] Yakun Sophia Shao et al. "Simba: scaling deep-learning inference with chiplet-based architecture." In: *Commun. ACM* 64.6 (2021), pp. 107–116. DOI: 10.1145/3460227. URL: https://doi.org/10.1145/3460227.

[Sha+21b] Vijeta Sharma, Manjari Gupta, Ajai Kumar, and Deepti Mishra. "Video Processing Using Deep Learning Techniques: A Systematic Literature Review." In: *IEEE Access* 9 (2021), pp. 139489–139507. DOI: 10.1109/ACCESS.2021.3118541. URL: https://doi.org/10.1109/ACCESS.2021.3118541.

[Sho97] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In: *SIAM J. Comput.* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 0097-5397. DOI: 10.1137/S0097539795293172.

[Sie14] Mark Siegesmund. *Chip Memory*. Tech. rep. ScienceDirect, 2014. URL: https://www.sciencedirect.com/topics/engineering/chip-memory.

[Sin+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data." In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 387–398. DOI: 10.1109/HPCA.2019.00052.

[Sin19] Sujoy Sinha Roy. "SaberX4: High-Throughput Software Implementation of Saber Key Encapsulation Mechanism." In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 2019, pp. 321–324. DOI: 10.1109/ICCD46524.2019.00050.

[SKC14] Dhiman Saha, Sukhendu Kuila, and Dipanwita Roy Chowdhury. "EscApe: Diagonal Fault Analysis of APE." In: *Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*. Ed. by Willi Meier and Debdeep Mukhopadhyay. Vol. 8885. Lecture Notes in Computer Science. Springer, 2014, pp. 197–216. DOI: 10.1007/978-3-319-13039-2_12. URL: https://doi.org/10.1007/978-3-319-13039-2%5C_12.

[SM16] Tobias Schneider and Amir Moradi. "Leakage assessment methodology - Extended version." In: *J. Cryptogr. Eng.* 6.2 (2016), pp. 85–99. DOI: 10.1007/S13389-016-0120-Y. URL: https://doi.org/10.1007/s13389-016-0120-y.

[SMC09] Dhiman Saha, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. "A Diagonal Fault Attack on the Advanced Encryption Standard." In: *IACR Cryptol. ePrint Arch.* (2009), p. 581. URL: http://eprint.iacr.org/2009/581.

[Smo+26] Vid Smole, Aikata Aikata, Pilipovic Ratko, and Sujoy Sinha Roy. *Giving Falcon Its Wings: Fast Hardware Acceleration for Signing and Verification using FN-DSA*. PrePrint. 2026.

[SMS23] Xhani Marvin SaSS, Richard Mitev, and Ahmad-Reza Sadeghi. "Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M." In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 6239–6256. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/sass.

[Sun+23] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices." In: *CoRR* abs/2303.15375 (2023). DOI: 10.48550/arXiv.2303.15375. arXiv: 2303.15375. URL: https://doi.org/10.48550/arXiv.2303.15375.

[Syn23] Synopsys. *How Universal Chiplet Interconnect Express Changes SoC Design*. 2023.

[TAC19] Sreenath Reddy Thummaluru, Mohammad Ameen, and Raghvendra Kumar Chaudhary. "Four-Port MIMO Cognitive Radio System for Midband 5G Applications." In: *IEEE Transactions on Antennas and Propagation* 67.8 (2019), pp. 5634–5645. DOI: 10.1109/TAP.2019.2918476.

[Tak+20] Jonathan Takeshita, Dayane Reis, Ting Gong, Michael Niemier, X. Sharon Hu, and Taeho Jung. *Algorithmic Acceleration of B/FV-like Somewhat Homomorphic Encryption for Compute-Enabled RAM*. Cryptology ePrint Archive, Report 2020/1223. https://ia.cr/2020/1223. 2020.

[Tan+21a] Weihang Tan, Antian Wang, Yingjie Lao, Xinmiao Zhang, and Keshab K. Parhi. "Low-Latency VLSI Architectures for Modular Polynomial Multiplication via Fast Filtering and Applications to Lattice-Based Cryptography." In: *CoRR* abs/2110.12127 (2021). arXiv: 2110.12127.

[Tan+21b] Zhanhong Tan, Hongyu Cai, Runpei Dong, and Kaisheng Ma. "NN-Baton: DNN Workload Orchestration and Chiplet Granularity Exploration for Multichip Accelerators." In: *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 1013–1026. DOI: 10.1109/ISCA52012.2021.00083. URL: https://doi.org/10.1109/ISCA52012.2021.00083.

[TBP20] Florian Tramèr, Dan Boneh, and Kenny Paterson. "Remote Side-Channel Attacks on Anonymous Transactions." In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 2739–2756. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/tramer.

[Tea19] Keccak Team. *Keccak in VHDL: High-speed Core*. https://keccak.team/hardware.html. 2019.

[TGF09] Thorlindur Thorolfsson, Kiran Gonsalves, and Paul D. Franzon. "Design automation for a 3DIC FFT processor for synthetic aperture radar: a case study." In: *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*. ACM, 2009, pp. 51–56. DOI: 10.1145/1629911.1629928. URL: https://doi.org/10.1145/1629911.1629928.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault." In: *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings*. Ed. by Claudio A. Ardagna and Jianying Zhou. Vol. 6633. Lecture Notes in Computer Science. Springer, 2011, pp. 224–233. DOI: 10.1007/978-3-642-21040-2_15. URL: https://doi.org/10.1007/978-3-642-21040-2%5C_15.

[Tun+] Michael Tunstall, Neil Hanley, Robert McEvoy, Claire Whelan, Colin Murphy, and William Marnane. "Correlation power analysis of large word sizes." In: (), pp. 13–14.

[UCI23] UCIe. *For the First Time, UCIe Shares Bandwidth Speeds Between Chiplets*. 2023. URL: https://www.hpcwire.com/2023/06/07/for-the-first-time-ucie-shares-bandwidth-speeds-between-chiplets/.

[UUS18] Muhammad Arslan Usman, Muhammad Rehan Usman, and Soo Young Shin. "An intrusion oriented heuristic for efficient resource management in end-to-end wireless video surveillance systems." In: *15th IEEE Annual Consumer Communications & Networking Conference, CCNC 2018, Las Vegas, NV, USA, January 12-15, 2018*. IEEE, 2018, pp. 1–6. DOI: 10.1109/CCNC.2018.8319177. URL: https://doi.org/10.1109/CCNC.2018.8319177.

[Via+23] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. "HECO: Fully Homomorphic Encryption Compiler." In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 4715–4732. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/viand.

[Viv+20] Pascal Vivet et al. "2.3 A 220GOPS 96-Core Processor with 6 Chiplets 3D-Stacked on an Active Interposer Offering 0.6ns/mm Latency, 3Tb/s/mm^2 Inter-Chiplet Interconnects and 156mW/mm^2@ 82%-Peak-Efficiency DC-DC Converters." In: *2020 IEEE International Solid- State Circuits Conference, ISSCC 2020, San Francisco, CA, USA, February 16-20, 2020*. IEEE, 2020, pp. 46–48. DOI: 10.1109/ISSCC19947.2020.9062927. URL: https://doi.org/10.1109/ISSCC19947.2020.9062927.

[Wan+14] Wei Wang, Xinming Huang, Niall Emmart, and Charles Weems. "VLSI Design of a Large-Number Multiplier for Fully Homomorphic Encryption." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.9 (2014), pp. 1879–1887. DOI: 10.1109/TVLSI.2013.2281786.

[Wan+22] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86." In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 679–697. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen.

[WH13] Wei Wang and Xinming Huang. "FPGA implementation of a large-number multiplier for fully homomorphic encryption." In: *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2013, pp. 2589–2592. DOI: 10.1109/ISCAS.2013.6572408.

[WH19] Shufang Wang and Hai Huang. "Secure Outsourced Computation of Multiple Matrix Multiplication Based on Fully Homomorphic Encryption." In: *KSII Trans. Internet Inf. Syst.* 13.11 (2019), pp. 5616–5630. DOI: 10.3837/TIIS.2019.11.019. URL: https://doi.org/10.3837/tiis.2019.11.019.

[Xin+20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. "VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture." In: *IEEE Trans. Circuits Syst. I Regul. Pap.* 67-I.8 (2020), pp. 2672–2684. DOI: 10.1109/TCSI.2020.2983185.

[XL21a] Yufei Xing and Shuguo Li. "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 328–356. DOI: 10.46586/tches.v2021.i2.328-356.

[XL21b] Yufei Xing and Shuguo Li. "A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 328–356.

[XZH21] Guozhu Xin, Yifan Zhao, and Jun Han. "A Multi-Layer Parallel Hardware Architecture for Homomorphic Computation in Machine Learning." In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401623.

[Yam+21a] Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savas. "A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme." In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*. IEEE, 2021, pp. 1020–1025. DOI: 10.23919/DATE51398.2021.9474139. URL: https://doi.org/10.23919/DATE51398.2021.9474139.

[Yam+21b] Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Sava. "A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme." In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1020–1025.

[YCH22] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. "PipeNTT: A Pipelined Number Theoretic Transform Architecture." In: *IEEE Trans. Circuits Syst. II Express Briefs* 69.10 (2022), pp. 4068–4072. DOI: 10.1109/TCSII.2022.3184703. URL: https://doi.org/10.1109/TCSII.2022.3184703.

[Yin+18] Jieming Yin, Zhifeng Lin, Onur Kayiran, Matthew Poremba, Muhammad Shoaib Bin Altaf, Natalie D. Enright Jerger, and Gabriel H. Loh. "Modular Routing Design for Chiplet-Based Systems." In: *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. Ed. by Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi. IEEE Computer Society, 2018, pp. 726–738. DOI: 10.1109/ISCA.2018.00066. URL: https://doi.org/10.1109/ISCA.2018.00066.

[Zha+21] Ye Zhang et al. “PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture.” In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 416–428.

[Zha+22a] Cankun Zhao et al. “A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 270–295. DOI: 10.46586/tches.v2022.i1.270-295.

[Zha+22b] Yifan Zhao, Ruiqi Xie, Guozhu Xin, and Jun Han. “A High-Performance Domain-Specific Processor With Matrix Extension of RISC-V for Module-LWE Applications.” In: *IEEE Trans. Circuits Syst. I Regul. Pap.* 69.7 (2022), pp. 2871–2884. DOI: 10.1109/TCSI.2022.3162593.

[Zho+21] Zhen Zhou, Debiao He, Zhe Liu, Min Luo, and Kim-Kwang Raymond Choo. “A SoftwareHardware Co-Design of Crystals-Dilithium Signature Scheme.” In: *ACM Trans. Reconfigurable Technol. Syst.* 14.2 (June 2021). ISSN: 1936-7406. DOI: 10.1145/3447812.

[Zhu+21] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. “LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR.” In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.3 (2021), pp. 1146–1159. DOI: 10.1109/TCSI.2020.3048395.

[Zhu+23] Lin Zhu, Qiang-Sheng Hua, Yi Chen, and Hai Jin. “Secure Outsourced Matrix Multiplication with Fully Homomorphic Encryption.” In: *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part I*. Ed. by Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis. Vol. 14344. Lecture Notes in Computer Science. Springer, 2023, pp. 249–269. DOI: 10.1007/978-3-031-50594-2_13. URL: https://doi.org/10.1007/978-3-031-50594-2%5C_13.

[ZLW23] Xiaopeng Zheng, Hongbo Li, and Dingkang Wang. “A New Framework for Fast Homomorphic Matrix Multiplication.” In: *IACR Cryptol. ePrint Arch.* (2023), p. 1649. URL: https://eprint.iacr.org/2023/1649.

[ZSB21] Florian Zaruba, Fabian Schuiki, and Luca Benini. “Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing.” In: *IEEE Micro* 41.2 (2021), pp. 36–42. DOI: 10.1109/MM.2020.3045564.

C
Affidavit

I declare that I have authored this thesis independently, that I have not used sources other than those declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. Generative AI has been used for refining the content. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.