EtherTrust: Sound Static Analysis of Ethereum bytecode

Ilya Grishchenko
Matteo Maffei
Clara Schneidewind
TU Wien
{ilya.grishchenko,matteo.maffei,clara.schneidewind}@tuwien.ac.at

ABSTRACT

Ethereum has emerged as the most popular smart contract development platform, with hundreds of thousands of contracts stored on the blockchain and covering a variety of application scenarios, such as auctions, trading platforms, and so on. Given their financial nature, the security of these contracts is of paramount importance, as exemplified by recent attacks exploiting programming mistakes to freeze or steal millions of dollars (e.g., the DAO and Parity attacks). An automated security analysis of these contracts is thus of utmost interest, but it is challenging due to the EVM bytecode format in which contracts are uploaded on the blockchain, which exposes very little static information, and due to the specific transaction-oriented programming mechanisms, which feature a subtle semantics.

In this work, we present the first sound and automated static analysis for EVM bytecode, which is practical and scales to large contracts. In particular, our static analysis supports reachability properties, which we show to be sufficient for capturing the most important security properties for smart contracts (e.g., single-entrancy and transaction environment dependency). The soundness of our analysis is proven against a complete and mechanized semantics of EVM bytecode. We implemented our analysis and tested our tool – EtherTrust – on real-world contracts from the Ethereum blockchain, comparing it with Oyente, the state-of-the-art bug finding tool for smart contracts: EtherTrust analyses real-life contracts in a few seconds, outperforming Oyente in efficiency and coverage by one order of magnitude. Furthermore, EtherTrust shows better precision on a benchmark, all of that despite being the first tool in the literature to provide formal security guarantees for EVM bytecode.

1 INTRODUCTION

Smart contracts introduced a paradigm shift in distributed computation, promising security in an adversarial setting for arbitrary distributed programs. Software developers can implement sophisticated distributed, transaction-based computations by leveraging the scripting language offered by the underlying cryptocurrency. While many of these cryptocurrencies have an intentionally limited scripting language (e.g., Bitcoin [27]), Ethereum was designed from the ground up with a quasi Turing-complete language[1]. Ethereum smart contracts have thus found a variety of appealing use cases, such as auctions [19], data management systems [8], financial contracts [13], elections [26], trading platforms [25, 29], permission management [11] and verifiable cloud computing [17], just to mention a few. Given their financial nature, bugs and vulnerabilities in smart contracts may lead to catastrophic consequences. For instance, the infamous DAO vulnerability [1] recently led to a 60M$ financial loss and similar vulnerabilities occur on a regular basis [2, 3]. Furthermore, many smart contracts in the wild are intentionally fraudulent, as highlighted in a recent survey [10].

A rigorous security analysis of smart contracts is thus crucial for the trust of the society in blockchain technologies and their widespread deployment. Unfortunately, this task is a quite challenging for various reasons. First, Ethereum smart contracts are developed in an ad-hoc language, called Solidity, which resembles JavaScript but features non-standard semantic behaviours and transaction-oriented mechanisms, which complicate smart contract development and verification. Second, smart contracts are uploaded on the blockchain in the form of Ethereum Virtual Machine (EVM) bytecode, a stack-based low-level code featuring dynamic code creation and invocation and, in general, very little static information, which makes it extremely difficult to analyze. Finally, some of the contracts interacting with the one under analysis may not be known statically. As a result, despite the increasing attention and progress in smart contract verification, *there exists at present no automated security analysis for EVM bytecode that provides formal security guarantees* (i.e., that is sound with respect to a formal semantics of EVM bytecode), as further detailed below.

1.1 State-of-the-art in Security Analysis of Smart Contracts

We categorize the existing approaches to smart contract static analysis along the following dimensions: target language (bytecode vs source code), provided guarantees (bug finding vs. formal soundness), checked properties (generic properties vs. contract-specific properties), degree of automation (automated verification vs. assisted analysis).

Static analysis tools for automated bug-finding. Oyente [24] is a state-of-the-art static analysis tool for EVM bytecode that relies on symbolic execution. Oyente supports a variety of generic security properties, such as transaction order dependency, timestamp dependency, and reentrancy that can be checked automatically. However, Oyente is not striving for soundness nor completeness. In fact, it has been shown that

[1] While the language itself is Turing complete, computations are associated with a bounded computational budget (called gas), which gets consumed by each instruction thereby enforcing termination.

the underlying semantics is incorrect and the tool has false negatives [18]. On the other hand, the security properties are rather syntactic and are lacking a semantic characterization. Similar reasoning applies to tools extending Oyente [28, 35].

Static analysis tools for automated verification of generic properties. ZEUS [22] analyses smart contracts written in Solidity using symbolic model checking. The analysis proceeds by translating Solidity code first into an abstract intermediate language and then into LLVM bitcode, in order to leverage off-the-shelf symbolic model checking tools for LLVM bitcode. Hence ZEUS can only analyze contracts whose Solidity source code is made available. In addition, while the analysis is claimed to be sound, we identified several problems in the statement and in the proof, which are discussed in detail in Appendix D. Most importantly, the LLVM bitcode obtained by translation does not preserve the semantics of the EVM bytecode obtained by compiling the Solidity source code. In a nutshell, this is due to two fundamental reasons. First, the semantics of the intermediate language does not allow for the revocation of the global system state in the case of a failed call, which however is fundamental feature of Ethereum smart contract execution. Second, the compilation into LLVM bitcode introduces artifacts that, as reported by the authors themselves, require manual adjustments of the code. Finally, the security analysis requires the modification of the Solidity source code, which is not covered by the soundness result. Hence, ZEUS does not provide formal security guarantees for EVM bytecode. Since the tool is not publicly available, we could not experimentally assess the impact of these theoretical flaws on the analysis results in practice. Other static analysis tools are available online (e.g., Securify [14], Mythril [6] and Manticore [30] for EVM bytecode and SmartCheck [32] and Solgraph [7] for Solidity code), but they are not accompanied by any academic paper so that the concrete goals and scope of the analysis stay unspecified.

Frameworks for semi-automated proofs for contract specific properties. A few works [9, 21] have focused on the usage of proof assistants for the formalization and mechanization of security proofs for EVM bytecode. While sound, this approach is not automated and, in fact, requires manual intervention and a significant expertise.

Hildebrandt et al. [20] define the EVM semantics in the $\mathbb{K}$ framework [31] – a language independent verification framework based on reachability logics. The authors leverage the power of the $\mathbb{K}$ framework in order to automatically derive analysis tools for the specified semantics, presenting as an example a gas analysis tool, a semantic debugger, and a program verifier based on reachability logics. The derived program verifier, however, still requires the user to manually specify loop invariants on the bytecode level.

Bhargavan et al. [12] introduce a framework to analyze Ethereum contracts by translation into F*, a functional programming language aimed at program verification and equipped with an interactive proof assistant. The translation supports only a fragment of the EVM bytecode and does not come with a justifying semantic argument.

1.2 Our Contributions

We present the first sound and automated static analysis technique for EVM bytecode, which is practical and scales to large contracts. Specifically, our contributions can be summarized as follows:

- We design a static reachability analysis technique for EVM bytecode, which is based on Horn-clause resolution. Designing a static analysis technique that handles the complexity of EVM bytecode and scales to large contracts is challenging and requires careful abstractions of various EVM components (e.g., the stack-based memory layout, the gas used to bound the smart contract execution, and the data format) as well as the semantic import of contracts unknown at analysis time;
- We prove the soundness of our static analysis technique against the semantics proposed by Grischenko et al. [18], which is complete, formalized in a proof assistant, and tested against the official EVM testsuite;
- We show that a reachability analysis suffices to cover all security properties for smart contracts introduced by Grischenko et al. [18], such as single entrancy and independence of miner-controlled parameters. The former is a safety property ruling out vulnerabilities due to unintended callbacks, like the DAO vulnerability [1]. The latter is a class of hyperproperties imposing that the semantic behaviour of contracts does not depend on data stored on the blockchain and possibly controlled by miners. In particular, we introduce a new reachability property, called call unreachability, that overapproximates single entrancy, and we further show that a simple dependency analysis suffices to overappoximate the other hyperproperties.
- We develop EtherTrust, a static analyzer that internally relies on the Z3 theorem prover for discharging proof obligations. We tested EtherTrust on a benchmark suite collecting code snippets from the literature as well as on real-life contracts stored on the blockchain, comparing its performance against Oyente. EtherTrust analyzes large contracts in a few seconds, outperforming Oyente in coverage and efficiency by one order of magnitude. Furthermore, it also offers better precision in our benchmark, all of that despite being the first tool in the literature to provide formal security guarantees for EVM bytecode.

1.3 Outline

The remainder of this paper is organized as follows. § 2 and § 3 review Ethereum and the semantics of EVM bytecode, respectively. § 4 summarizes the salient security properties for smart contracts. § 5 introduces our static analysis technique for reachability properties. § 6 states the formal results. § 7 describes EtherTrust and presents our experimental evaluation. § 8 concludes by highlighting interesting future research directions.

2 BACKGROUND

Ethereum. Ethereum is a cryptographic currency system built on top of a blockchain. Similar to Bitcoin, network participants publish transactions to the network that are then grouped into blocks by distinct nodes (the so called *miners*) and appended to the blockchain using a proof of work (PoW) consensus mechanism. The state of the system – that we will also refer to as *global state* – consists of the state of the different accounts populating it. An account can either be an external account (belonging to a user of the system) that carries information on its current balance or it can be a contract account that additionally obtains persistent storage and the contract's code. The account's balances are given in the subunit *wei* of the virtual currency *Ether*.[2]

Transactions can alter the state of the system by either creating new contract accounts or by calling an existing account. Calls to external accounts can only transfer Ether to this account, but calls to contract accounts additionally execute the code associated to the contract. The contract execution might alter the storage of the account or might again perform transactions – in this case we talk about *internal transactions*.

The execution model underlying the execution of contract code is described by a virtual state machine, the *Ethereum Virtual Machine* (EVM). This is *quasi Turing complete* as the otherwise Turing complete execution is restricted by the upfront defined resource *gas* that effectively limits the number of execution steps. The originator of the transaction can specify the maximal gas that should be spent for the contract execution and also determines the gas price (the amount of wei to pay for a unit of gas). Upfront, the originator pays for the gas limit according to the gas price and in case of successful contract execution that did not spend the whole amount of gas dedicated to it, the originator gets reimbursed with the gas that is left. The remaining wei paid for the used gas are given as a fee to a beneficiary address specified by the miner.

EVM bytecode. Contracts are delivered and executed in *EVM bytecode* format – an Assembler like bytecode language. As the core of the EVM is a stack-based machine, the set of instructions in EVM bytecode largely consists of standard instructions for stack operations, arithmetics, jumps and local memory access. The classical set of instructions is enriched with an opcode for the SHA3 hash and several opcodes for accessing the environment that the contract was called in. In addition, there are opcodes for accessing and modifying the storage of the account currently running the code and distinct opcodes for performing internal transactions. Another instruction particular to the blockchain setting is the SELFDESTRUCT code that deletes the currently executed contract - but only after the successful execution of the external transaction.

The execution of each instruction consumes a positive amount of *gas*. The sender of the transaction specifies a gas limit and exceeding it results in an exception that reverts the effects of the current transaction on the global state. In the case of nested transactions, the occurrence of an exception only reverts its own effects, but not those of the calling transaction. Instead, the failure of an internal transaction is only indicated by writing zero to the caller's stack.

3 CONCRETE SMALL-STEP SEMANTICS

Our static analysis targets a recently introduced small-step semantics for EVM bytecode [18], which we shortly review below, highlighting the most interesting semantic features of EVM bytecode.

3.1 Execution Configurations

Global State. The global state of the Ethereum blockchain is represented as a (partial) mapping from account addresses to accounts. In the case that an account does not exist, we assume it to map to $\bot$. Accounts are composed of a nonce n that is incremented with every other account that the account creates, a balance b, a persistent unbounded storage *stor*, and the account's code. External accounts carry an empty code which makes their storage inaccessible and hence irrelevant.

Small-step Relation. The semantics is formalized by a small-step relation $\Gamma \vDash S \rightarrow S'$ that specifies how a call stack S representing the state of the execution evolves within one step under the transaction environment Γ. We write $\Gamma \vDash S \rightarrow^* S'$ for the reflexive transitive closure of the relation and call the pair (Γ, S) a *configuration*.

Transaction Environments. The transaction environment represents the static information of the block that the transaction is executed in and the immutable parameters of the transaction, e.g., the gas price or limit. These parameters can be accessed by distinct bytecode instructions and consequently influence the transaction execution.

Call stacks. A call stack S is a stack of execution states which represents the overall state of the initial external transaction. The individual execution states of the stack represent the states of the uncompleted internal transactions performed during the execution. Formally, a call stack is a stack of regular execution states of the form (μ, ι, σ) that can optionally be topped with a halting state $HALT(\sigma, gas, d)$ or an exception state EXC. Semantically, halting states indicate regular halting of an internal transaction, exception states indicate exceptional halting, and regular execution states describe the state of internal transactions in progress. Halting and exception states can only occur as top elements of the call stack as they represent terminated internal transactions. Halting states carry the information affecting the callee state such as the global state σ that the internal execution halted in, the unspent gas *gas* from the internal transaction execution, and the return data d.

The state of a non-terminated internal transaction is described by a regular execution state of the form (μ, ι, σ). The state is determined by the current global state σ of the system as well as the execution environment ι that specifies the parameters of the current internal transaction (including inputs and the code to be executed) and the local state μ of the stack machine.

[2] One Ether is equivalent to 10^{18} wei.

Table 1: Selected Semantic Rules

$$\textsc{Add}\quad \frac{\iota.code\,[\mu.\mathsf{pc}] = \mathsf{ADD} \qquad \mu.\mathsf{s} = a :: b :: s \qquad \mu.\mathsf{gas} \geq 3 \qquad \mu' = \mu[\mathsf{s} \rightarrow (a + b) :: s][\mathsf{pc}\ \mathrel{+}= 1][\mathsf{gas}\ \mathrel{-}= 3]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow (\mu', \iota, \sigma) :: S}$$

$$\textsc{Add-Fail}\quad \frac{\iota.code\,[\mu.\mathsf{pc}] = \mathsf{ADD} \qquad (|\mu.\mathsf{s}| < 2 \vee \mu.\mathsf{gas} < 3)}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow EXC :: S}$$

Execution Environment. The execution environment ι of an internal transaction is a tuple of static parameters (*actor*, *input*, *sender*, *value*, *code*) to the transaction that, among others, determine the code to be executed and the account in whose context the code will be executed. The execution environment incorporates the following components: the active account *actor* that is the account that is currently executing and whose account will be affected when instructions for storage modification or money transfer are executed; the input data *input* given to the transaction; the address *sender* of the account that initiated the transaction; the amount of wei *value* transferred with the transaction; the code *code* that is executed by the transaction. The execution environment is determined upon initialization of an internal transaction execution and it can be accessed, but not altered during the execution.

Machine State. The local machine state μ represents the state of the underlying stack machine used for execution. Formally it is represented by a tuple (gas, pc, m, i, s) holding the amount of gas *gas* available for execution, the program counter *pc*, the local memory m, the number of active words in memory i, and the machine stack s.

The execution of each internal transaction starts in a fresh machine state, with an empty stack, memory initialized to all zeros, and program counter and active words in memory set to zero. Only the gas is instantiated with the gas value available for the execution. We call execution states of this form *initial*.

3.2 Small-step Rules

In the following, we will present a selection of interesting small-step rules (cf. Table 1) in order to illustrate the most important features of EVM bytecode.

Local Instructions. For demonstrating the overall design of the semantics, we start with the example of the arithmetic expression ADD performing addition of two values on the machine stack. We use a dot notation, in order to access components of the different state parameters. We name the components with the variable names introduced for these components in the last section written in sans-serif-style. In addition, we use the usual notation for updating components: $t[\mathsf{c} \rightarrow v]$ denotes that the component c of tuple t is updated with value v. For expressing incremental updates in a simpler way, we additionally use the notation $t[\mathsf{c}\ \mathrel{+}= v]$ to denote that the (numerical) component of c is incremented by v and similarly $t[\mathsf{c}\ \mathrel{-}= v]$ for decrementing a component c of t.

The execution of the arithmetic instruction ADD only performs local changes in the machine state affecting the local stack, the program counter, and the gas budget. For deciding upon the correct instruction to execute, the currently executed code (that is part of the execution environment) is accessed at the position of the current program counter. The cost of an ADD instruction consists always of three units of gas that get subtracted from the gas budget in the machine state. As every other instruction, ADD can fail due to lacking gas or due to underflows on the machine stack. In this case, the exception state is entered and the execution of the current internal transaction is terminated.

Transaction Initiating Instructions. A class of instructions with a more involved semantics are those instructions initiating internal transactions. This class incorporates instructions for calling another contract (CALL, CALLCODE and DELEGATECALL) and for creating a new contract (CREATE). Intuitively, CALL executes the callee's code in its own environment, CALLCODE executes the callee's code in the caller's environment, which might be useful to call libraries implemented in a separate contract, and DELEGATECALL takes a step further by preserving not only the caller's environment but even part of the environment of the previous call (e.g., the sender information), which effectively treats the calle's code as an internal function of the caller's code. Finally, the CREATE instruction initiates an internal transaction that creates a new account. For more detail, we refer to Appendix C.

Instructions from this set are particularly difficult to analyze, since their arguments are dynamically evaluated and the execution environment has to be tracked and properly modified across different calls. Furthermore, it can well be that the code of a called function is not accessible at analysis time, e.g., because the contract transfers money back to the caller (like in the DAO contract).

4 SECURITY PROPERTIES

Grishchenko et al. [18] propose generic security definitions for smart contracts that rule out certain classes of potentially harmful contract behavior. We will show in § 6.2 how they can be over-approximated as pure reachability properties. Due to space constraints, we focus on the properties that characterize the absence of those vulnerabilities that have already been targeted by other analysis approaches such as [22] and [24].

```
1  contract Bob{
2    bool sent = false;
3    function ping( address c){
4      if (!sent) { c.call.value(2)();
5                   sent = true; }}}
```

(a) Smart contract with reentrancy bug

```
1  contract Mallory{
2    function(){
3      Bob(msg.sender).ping(this);}}
```

(b) Smart contract exploiting reentrancy bug

Figure 1: Reentrancy Attack

4.1 Preliminary Notations

Formally, we represent a contract as a tuple of form $(a, code)$ where a and $code$ denote the contract's address and code, respectively.

In order to give concise security definitions, we further introduce, and assume throughout the paper, an annotation to the small step semantics in order to highlight the contract c that is currently executed. In the case of initialization code being executed, we use $\perp$. Finally, for arguing about EVM bytecode executions, we are only interested in those initial configurations that might result from a valid external transaction in a valid block. We call these configurations *reachable* and refer to [18] for a detailed definition.

Next, we introduce the notion of execution trace for smart contract execution. Intuitively, a trace is a sequence of actions. In our setting, the actions to be recorded are composed of an opcode, the address of the executing contract, and a sequence of arguments to the opcode. We denote the set of actions with Act. Accordingly, every small step produces a trace consisting of a single action. Lifting the resulting trace semantics to multiple execution steps consequently results in sequences of actions $\pi \in \mathcal{L}(Act)$. We will write $\pi \downarrow_{\mathsf{calls}_c}$ to denote the projection of π to calls performed by contract c, i.e., actions with opcodes CALL, DELEGATECALL, CALLCODE or CREATE.

4.2 Single-entrancy

For motivating the definition of single-entrancy, we introduce a class of bugs in Ethereum smart contracts called *reentrancy bugs* [10, 24].

The most famous representative of this class is the so called DAO bug that led to a loss of 60 million dollars in June 2016 [1]. In an attack exploiting this bug, the affected contract was drained out of money by subsequently reentering it and performing transactions to the attacker on behalf of the contract. The cause of such bugs mostly roots in the developer's misunderstanding of the semantics of Solidity's call primitives. In general, calling a contract can invoke two kinds of actions: Transferring Ether to the contract's account or executing (parts of) a contract's code. In particular, Solidity's `call` construct (being translated to a CALL instruction in EVM bytecode) invokes the execution of a fraction of the callee's code – specified in the so called *fallback function*. In Solidity, a contract's fallback function is written as a function without names or argument as depicted in the `Mallory` contract in Figure 1b. Consequently, when using the `call` construct the developer may expect an atomic value transfer where potentially another contract's code is executed. For illustrating how to exploit this sort of bug, we consider the contracts in Figure 1. The function `ping` of contract `Bob` sends an amount of 2 *wei* to the address specified in the argument. However, this should only be possible once, which is potentially ensured by the `sent` variable that is set after the successful money transfer. Instead, it turns out that invoking the `call.value` function on a contract's address invokes the contract's fallback function as well.

Given a second contract `Mallory`, it is possible to transfer more money than the intended 2 *wei* to the account of `Mallory`. By invoking `Bob`'s function `ping` with the address of `Mallory`'s account, 2 *wei* are transferred to `Mallory`'s account and additionally the fallback function of `Mallory` is invoked. As the fallback function again calls the `ping` function with `Mallory`'s address another 2 *wei* are transferred before the variable `sent` of contract `Bob` was set. This looping goes on until all gas of the initial call is consumed or the callstack limit is reached. In this case, only the last transfer of *wei* is reverted and the effects of all former calls stay in place. Consequently the intended restriction on contract `Bob`'s `ping` function (namely to transfer 2 *wei* only once) is circumvented.

The security property ruling out these attacks is called *single-entrancy* and is formalized below. Intuitively, a contract is single-entrant if it cannot perform any more calls once it has been reentered.

Definition 4.1 (Single-entrancy [18]). A contract c is single-entrant if for all reachable configurations $(\Gamma, s_c :: S)$, for all s', s'', S'

$$\begin{aligned}&\Gamma \vDash s_c :: S \rightarrow^* {s'}_c :: S' + + s_c :: S\\ &\implies \neg\exists s'' \in \mathcal{S}, c' \in C_\perp.\\ &\qquad \Gamma \vDash {s'}_c :: S' + + s_c :: S \rightarrow^* {s''}_{c'} :: {s'}_c :: S' + + s_c :: S\end{aligned}$$

where ++ denotes concatenation of call stacks.

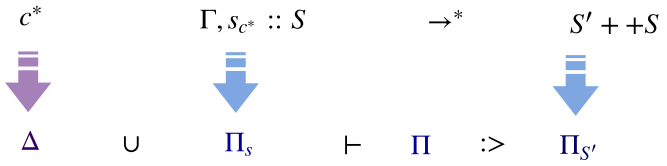

Figure 2: Simplified soundness statement

4.3 Independence of Miner controlled Parameters

A particularity of the distributed blockchain environment is that users while performing transactions cannot make assumptions on large parts of the context their transaction will be executed in. This is due to the inherently asynchronous nature of the system as well as to the fact that miners heavily influence the execution context of transactions. They can decide upon the transaction order in a block (and also sneak their own transactions in first) and in addition they can even control some parameters as the block timestamp within a certain range.

Consequently, contracts whose (outgoing) money flows depend either on miner controlled block information or on state information (as the state of their storage or their balance) that might be changed by other transactions are prone to manipulations by miners. To illustrate, we report below the notion of *independence of the transaction environment*, which is formalized as a hyperproperty, comparing two runs of the smart contract.

To this end, we assume C_Γ to be the set of components of the transaction environment and write $\Gamma =_{/c_\Gamma} \Gamma'$ to denote that the transaction environments Γ, Γ' are equal up to component c_Γ.

Definition 4.2 (Independence of the Transaction Environment [18]). A contract $c \in C$ is independent of a subset $I \subseteq C_\Gamma$ of components of the transaction environment if for all $c_\Gamma \in I$ and all reachable configurations $(\Gamma, s_c :: S)$ it holds for all Γ' that

$$c_\Gamma(\Gamma) \neq c_\Gamma(\Gamma') \ \wedge \ \Gamma =_{/c_\Gamma} \Gamma' \ \wedge \ \Gamma \vDash s_c :: S \xrightarrow{\pi}{}^* s'_c :: S \ \wedge \ \mathit{final}\,(s')$$

$$\wedge \ \Gamma' \vDash s_c :: S \xrightarrow{\pi'}{}^* s''_c :: S \ \wedge \ \mathit{final}\,(s'') \implies \pi \downarrow_{\mathsf{calls}_c} = \pi' \downarrow_{\mathsf{calls}_c}$$

The notion of *independence of the account state* is defined analogously.

5 ABSTRACT SEMANTICS

We developed a static analysis framework for automatically analyzing reachability properties of EVM smart contracts. The analysis relies on an abstract semantics for EVM bytecode soundly over-approximating the semantics presented in Section 3.

Figure 2 gives an overview on the relation between the small-step and the abstract semantics. For the analysis, we will consider a particular contract c^* under analysis as well as a set $C_k \ni c^*$ of known contracts that might be called during the execution of c^*. An over-approximation of the behavior of these known smart contracts is encoded in terms of *Horn clauses* (Δ). These describe how an abstract configuration Π evolves within the execution of the contracts' instructions. Abstract configurations are obtained by translating small-step configurations to a set Π of facts over state predicates that characterize (an over-approximation of) the original configuration. Finally, we will show that no matter how the contract c^* is called (so for every arbitrary reachable configuration $\Gamma, s_{c^*} :: S$), every sequence of execution steps that is performed while executing it can be mimicked by a logical derivation from the abstract configuration Π_s (obtained from translating the execution state s) using the Horn clauses Δ (that model the abstract semantics of the contracts in C_k). More precisely, this means that from the set of facts $\Pi_s \cup \Delta$, a set Π can be derived that is a coarser abstraction (:>) than $\Pi_{S'}$, which is the translation of the execution's intermediate callstack S'.

5.1 Abstract Configurations

Table 2 shows the analysis facts used for describing the abstract semantics. These consist of (instances of) state predicates that represent partial abstract configurations. Accordingly, abstract configurations are sets of closed facts. Finally, abstract contracts are characterized as sets of Horn clauses over the state predicates (facts) that describe the state changes induced by the instructions at the different program positions.

The state predicates model the execution states s_c of contracts $c \in C_k$. For linking these contracts with the corresponding analysis facts, we introduce artificial contract identifiers and (uniquely) label the contracts in C_k with them so that formally elements from C_k are of the form $(id, a, code)$ with identifiers id and addresses a being unique in the set. The state predicates are then parametrized by either a program point pp or a contract identifier id where a program point again is a pair of the form (id, pc) with $id \in \mathbb{N}$ being a contract identifier and $pc \in \mathbb{N}$ being the program counter at which the abstract state holds.[3]

In addition, all state predicates but Code_{id} carry the relative call depth $cd \in \mathbb{N}$ as argument. The relative call depth is the size of the call stack built up on the execution of c^* (cf. callstack S' in Figure 2) and serves as abstraction for the (relative) callstack that contract c^* is currently

[3] Making the program counter a parameter instead of an argument is a design choice made in order to minimize the number of recursive horn clauses and to hence simply automated verification.

Table 2: Analysis Facts. All arguments in the analysis facts marked with a hat ($\hat{\cdot}$) range over $\hat{D} \cup \mathit{Vars}$ where $\hat{D}$ is the abstract domain and *Vars* is the set of variables. Arguments marked with a dot ($\dot{\cdot}$) range over $\mathbb{N} \cup \{\alpha\} \cup \mathit{Vars}$. All other arguments of analysis facts range over $\mathbb{N}$ with exception of *sa* that ranges over $(\mathbb{N} \to \hat{D}) \cup \mathit{Vars}$. Closed facts *cf* are assumed to be facts with arguments not coming from *Vars*.

Facts	f	:=	
Abs. machine state		|	$\mathsf{MState}_{pp}\ ((\mathit{size}, \mathit{sa}), \hat{aw}, \hat{gas}, \mathit{cd})$
Abs. memory		|	$\mathsf{Mem}_{pp}\ (\hat{pos}, \hat{va}, \mathit{cd})$
Abs. execution environment		|	$\mathsf{ExEnv}_{id}\ (\dot{a}, \hat{i}, \hat{va}, \hat{inputsize}, \mathit{cd})$
Abs. input data		|	$\mathsf{Inputdata}_{id}\ (\mathit{pos}, \hat{v}, \mathit{cd})$
Abs. global state		|	$\mathsf{GState}_{pp}\ (\dot{a}, \hat{n}, \hat{b}, \mathit{cd})$
Abs. persistent account storage		|	$\mathsf{Stor}_{pp}\ (\dot{a}, \hat{pos}, \hat{v}, \mathit{cd})$
Abs. successful halting state		|	$\mathsf{Return}_{id}\ (\hat{returndatasize}, \hat{gas}, \mathit{cd})$
Abs. return value		|	$\mathsf{Res}_{id}\ (\mathit{pos}, \hat{v}, \mathit{cd})$
Abs. global state at return		|	$\mathsf{ResGState}_{id}\ (\dot{a}, \hat{n}, \hat{b}, \mathit{cd})$
Abs. persistent storage at return		|	$\mathsf{ResStor}_{id}\ (\dot{a}, \hat{pos}, \hat{v}, \mathit{cd})$
Abs. exception state		|	$\mathsf{Exc}_{id}\ (\mathit{cd})$
Abs. contract code		|	$\mathsf{Code}_{id}\ (\mathit{pc}, \mathit{com})$
Abs. configurations	Π	:=	$\{cf_1, \ldots, cf_n\}$
Horn clauses	H	:=	$\forall x^*.\ \bigwedge_i f_i \implies f$
Abs. contracts	Δ	:=	$\{H_1, \ldots, H_n\}$

executed on. The relative call depth helps to distinguish different recursive executions of contracts $c \in C_k$ and thereby improves the precision of the analysis.

The Code_{id} predicate that represents the code of the contracts in C_k does not need to carry this argument as contract code is immutable during the execution.

Abstract Domain. The predicate's arguments range over an abstract domain $\hat{D}$ or over a subdomain thereof. The abstract domain serves as a means of over-approximating computations performed on concrete values (such as stack values) and also allows for abstracting the semantics of instructions that depend on such values (such as memory accesses or calls). Concretely, we define the abstract domain $\hat{D}$ to be the set $\{\bot, \top, \alpha\} \cup \mathbb{N}$ which constitutes a bounded lattice $(\hat{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ satisfying $\bot \sqsubset \alpha \sqsubset \top$ and $\bot \sqsubset n \sqsubset \top$ for all $n \in \mathbb{N}$. Intuitively, in our analysis $\top$ will represent unknown (symbolic) values and α will represent the unknown (symbolic) address of contract c^*.

Treating the address of the contract under analysis in a symbolic fashion is crucial for obtaining a meaningful analysis, as the address of this account on the blockchain can not easily be assumed to be known upfront. This is as we consider a contract to be analyzed before being deployed on the blockchain, but addresses get only assigned when creating the contract on the blockchain and depend on the creator's account state at the time of creation. Still, a contract can dynamically access its own address during execution and use it as a reference for accessing e.g, its balance or performing a self call. Consequently, we aim at tracking the address of contract c^* as long as it stays unmodified for using it in an accurate manner when it is used for global state accesses or calls, but treat the address as arbitrary value when being used for other purposes (such as arithmetics or memory accesses).

In particular, we will assume versions of the unary, binary and comparison operators on the values of the abstract domain whose semantics follows this intuition. We will mark abstract operators with a hat ($\widehat{\cdot}$) and, e.g., write $\widehat{+}$ for abstract addition.

As we assume the stack values to range over the full abstract domain, most of the other fact's arguments do so too (marked by a hat over the corresponding arguments) as they will in some rules of the abstract semantics get instantiated or accessed by stack values. An exception is given by the arguments of the global state and execution environment predicates that represent addresses. These arguments are not assumed to ever take the value $\top$ as we will always be aiming at keeping a distinction between the account state of contract c^* with symbolic address α and the ones of other contracts.

As, however, all arguments come from subdomains of $\hat{D}$, we can lift the order $\sqsubseteq$ on abstract values to closed analysis facts by writing $\mathit{cf} \sqsubseteq \mathit{cf}'$ whenever *cf* and cf' are instances of the same predicate and for every argument position i it holds for arguments v_i, v_i' that $v_i \sqsubseteq v_i'$. So intuitively, $\mathit{cf} \sqsubseteq \mathit{cf}'$ means that *cf* is a more precise abstraction than cf'. This order again is lifted to abstract configurations by having $\Pi <: \Pi'$ if and only if $\forall \mathit{cf} \in \Pi.\ \exists \mathit{cf}' \in \Pi'.\ \mathit{cf} \sqsubseteq \mathit{cf}'$.

State predicates. In the following, we will discuss the intuitive meaning of the analysis facts. We start by discussing the predicates for abstracting regular execution states of the form (μ, ι, σ).

The local machine state μ is modeled by the predicates MState_{pp} and Mem_{pp}. The fact $\mathsf{MState}_{pp}\ ((\mathit{size}, \mathit{sa}), \hat{aw}, \hat{gas}, \mathit{cd})$ says that, at program point *pp* and relative call depth *cd*, the machine stack is of size *size* and its current configuration is abstracted by the mapping *sa* which maps stack positions to abstract values. In addition the number of active words in memory is over-approximated by $\hat{aw}$ and the gas is over-approximated by $\hat{gas}$. Similarly, the fact $\mathsf{Mem}_{pp}\ (\hat{pos}, \hat{v}, \mathit{cd})$ states that, at program point *pp* and relative call depth *cd*, the value at abstract memory address $\hat{pos}$ is abstracted by $\hat{v}$.

The execution environment ι is modeled by the predicates ExEnv_{id} and $\mathsf{Inputdata}_{id}$. In contrast to the machine state predicates, these predicates are parameterized only by the contract identifier and not by the program counter. The reason is that the execution environment remains unchanged during the execution of a contract call. The predicate $\mathsf{ExEnv}_{id}\ (\dot{a}, \hat{i}, \hat{va}, \hat{inputsize}, \mathit{cd})$ states that when the contract with identifier *id* is executed on relative call level *cd* the address of the executing account is over-approximated by $\dot{a}$, the address of the initiator is

abstracted as $\hat{\imath}$, the value transferred by the calling transaction is over-approximated by $\hat{va}$, and the size of the input given to this transaction is over-approximated by $\hat{inputsize}$. As for the local memory, the abstraction of input data is realized as an own predicate $\mathsf{Inputdata}_{id}$.

The abstraction of the global state σ is given by the predicates $\mathsf{GState}_{(id,\,pc)}$ and $\mathsf{Stor}_{(id,\,pc)}$. The fact $\mathsf{GState}_{(id,\,pc)}\ (\hat{a}, \hat{n}, \hat{b}, cd)$ states that when the contract with identifier id is executed at program counter pc and on relative call level cd, the account with abstract address $\hat{a}$ has nonce over-approximated as $\hat{n}$ and balance over-approximated as $\hat{b}$. For modeling the persistent account storage, as for the local memory, we use an own predicate $\mathsf{Stor}_{(id,\,pc)}$.

Next, we explain the predicates modeling exception and halting states. While exception states can be described by a single predicate, the different components of halting states (namely, the return data and the resulting global state) need to be represented by distinct predicates. The fact $\mathsf{Exc}_{id}\ (cd)$ states that an exception occurred when executing the contract with the identifier id on relative call level cd. Similarly, the fact $\mathsf{Return}_{id}\ (\hat{returndatasize}, \hat{gas}, cd)$ states that the execution of the contract with identifier id halted regularly on relative call level cd with remaining gas over-approximated by $\hat{gas}$ and returned a byte array of size over-approximated by $\hat{returndatasize}$. The array of return data itself is then specified by an own predicate Res_{id} similar to $\mathsf{Inputdata}_{id}$. The new global state that resulted from the successful call is written to predicates $\mathsf{ResGState}_{id}$ and $\mathsf{ResStor}_{id}$ that resemble those of the global state with the only difference that they are parametrized only by the account that finished the execution and not additionally by the program counter.

As the account code is immutable during execution, we do not model it with respect to the program point or the call depth, but simply initialize the static predicate Code_{id} for the contracts $c \in C_k$ with the corresponding integer representations of the account's bytecode. More precisely, the fact $\mathsf{Code}_{id}\ (pc, com)$ states that at program counter pc the contract with identifier id holds the bytecode with integer representation com. The use of such a static predicate allows for dynamically accessing certain code fractions during execution (as it is required for the CODECOPY instruction).

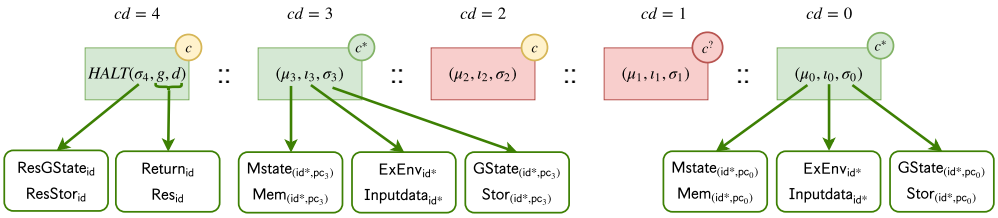

Figure 3: Illustration of the translation of abstract call stacks. Translated execution states are depicted in green, non-translated in red. Accordingly, the contract annotations for c^* are depicted in green, annotations for $c \in C_k/\{c^*\}$ in yellow, and annotations for $c^? \notin C_k$ in red.

Abstraction function. Formally, we establish the relation between a configuration of the small-step semantics and its abstraction by an abstraction function that translates call stacks to a set of analysis facts.[4] Figure 3 gives an overview on how callstacks are mapped into sets of analysis facts. The translation proceeds by translating all execution states on the callstack bottom up: When encountering an execution state of c^*, its components get translated to the corresponding predicates (with identifier id^*) as shown for the right most contract in Figure 3. As long as only contracts $c \in C_k$ are encountered, they are translated in the same fashion, but when encountering a contract $c^? \notin C_k$ the translation of this contract as well as of all following contracts $c \in C_k/\{c^*\}$ is omitted. Only on appearance of an execution state of contract c^*, the translation proceeds. Intuitively, this translation models that we are only over-approximating all executions of contract c^*. The contracts C_k increase the precision of this over-approximation and are faithfully simulated when being part of a 'known' call chain from c^*, but not all of their executions are over-approximated (cf. contract at call depth cd in Figure 3).

The precise definition of the abstraction function is given in Table 3. The function α_S for callstacks proceeds by translating selected execution states to a set of instances of the state predicates as described above. The whole translation of the callstack is performed with respect to the relative call depth cd. This is as for our analysis we will consider the execution of contract c^* on an arbitrary callstack as depicted in Figure 2. The argument cd of function α_S can be therefore thought of as the size of S in Figure 2.

The function α_s for translating executions states gets the identifier id of the contract whose execution state gets translated as additional argument and a mapping $\mathring{\cdot}$ that maps the value of c^*'s address to α. This mapping needs to be applied to all potentially abstract arguments of state predicates during the translation to ensure that values potentially representing c^*'s address are consistently renamed. Otherwise, the translation proceeds in a straight-forward manner: Exception and execution states are directly translated to the corresponding predicates. Regular execution states (μ, ι, σ) are component-wise translated. As machine state and global states are parametrized by the program counter, this information is extracted from the machine state and given as argument to the translation function α_σ for global states. We discuss two particularities of the translations in more detail, that is, the translation of mappings that represent memory or storage and the translation of the machine stack: The local machine stack is translated to an abstract array representation by the function stackToArray. This representation is a pair of the stack's (concrete) size and a mapping from (concrete) stack positions to the stack's abstracted elements. Keeping size and positions

[4] Note that we don't translate the transaction environments Γ as all accesses to it will be directly over-approximated in the abstract execution rules of the corresponding instructions.

of the stack precise is necessary as otherwise it would not be possible to extract the (potentially abstract) arguments to the instructions from the stack in a meaningful way. Memory and storage mappings (such as the memory of the machine state, the storage of the global state, the input data of the execution environment or return data of the regular halting state) instead are for technical reasons[5] not translated to array representations, but to own predicates mapping abstract locations to abstract values.

Table 3: Abstraction function for small-step configurations

$$
\begin{aligned}
\alpha_S\,(S', C_k, id^*, cd) &:= \mathit{let}\ (H, \mathit{flag}) = \alpha_{S-help}\,(S', C_k, id^*, cd)\ \mathit{in}\ H \\
\alpha_{S-help}\,(\epsilon, C_k, id^*, cd) &:= (\emptyset, \top) \\
\alpha_{S-help}\,(s_c :: S', C_k, id^*, cd) &:= \mathit{let}\ (H, \mathit{flag}) = \alpha_{S-help}\,(S', C_k, id^*, cd)\ \mathit{in} \\
&\qquad \mathit{let}\ id = \mathit{getID}\,(C_k, c), \mathring{\cdot} = (\lambda a.\mathit{if}\ (id^*, a, \cdot) \in C_k\ \mathit{then}\ \alpha\ \mathit{else}\ a)\ \mathit{in} \\
&\qquad\quad \mathit{if}\ (id^* = id)\ \mathit{then}\ (H \cup \alpha_s\,(s, id^*, |S'| + cd, \mathring{\cdot}), \top) \\
&\qquad\quad \mathit{else\ if}\ (\mathit{flag} = \top \wedge id \neq \bot)\ \mathit{then}\ (H \cup \alpha_s\,(s, id, |S'| + cd, \mathring{\cdot}), \top)\ \mathit{else}\ (H, \bot) \\
\alpha_s\,((\mu, \iota, \sigma), id, cd, \mathring{\cdot}) &:= \alpha_\mu\,(\mu, id, cd, \mathring{\cdot}) \cup \alpha_\iota\,(\iota, id, cd, \mathring{\cdot}) \cup \alpha_\sigma\,(\sigma, id, \mu.\mathsf{pc}, cd, \mathring{\cdot}) \\
\alpha_s\,(\mathit{EXC}, id, cd, \mathring{\cdot}) &:= \{\mathsf{Exc}_{id}\,(cd)\} \\
\alpha_s\,(\mathit{HALT}(\sigma, gas, data), id, cd, \mathring{\cdot}) &:= \{\mathsf{Return}_{id}\,(|\mathring{data}|, \mathring{gas}, cd) \cup \{\mathsf{Res}_{id}\,(pos, \mathring{v}, cd) \mid data\,[pos] = v \wedge pos \in \mathbb{N}\} \\
&\qquad \cup \{\mathsf{ResGState}_{id}\,(\mathring{a}, \mathring{n}, \mathring{b}, cd) \mid \exists stor, code.\ \sigma(a) = (n, b, stor, code)\} \\
&\qquad \cup \{\mathsf{ResStor}_{id}\,(\mathring{a}, \mathring{pos}, \mathring{v}, cd) \mid \exists n, b, stor, code.\ \sigma(a) = (n, b, stor, code) \wedge stor\,[pos] = v \wedge pos \leq 2^{256}\} \\
\alpha_\mu\,((gas, pc, m, i, s), id, cd, \mathring{\cdot}) &:= \{\mathsf{MState}_{(id,\,pc)}\,(\mathsf{stackToArray}\,(s, \mathring{\cdot}), \mathring{i}, \mathring{gas}, cd)\} \cup \{\mathsf{Mem}_{(id,\,pc)}\,(\mathring{pos}, \mathring{v}, cd) \mid m\,[pos] = v \wedge pos \leq 2^{256}\} \\
\alpha_\iota\,((actor, input, sender, va, code), id, cd, \mathring{\cdot}) &:= \{\mathsf{ExEnv}_{id}\,(\mathring{actor}, \mathring{sender}, \mathring{va}, |\mathring{input}|, cd) \cup \{\mathsf{Inputdata}_{id}\,(pos, \mathring{v}, cd) \mid input\,[pos] = v \wedge pos \in \mathbb{N}\} \\
\alpha_\sigma\,(\sigma, id, pc, cd, \mathring{\cdot}) &:= \{\mathsf{GState}_{(id,\,pc)}\,(\mathring{a}, \mathring{n}, \mathring{b}, cd) \mid \exists stor, code.\ \sigma(a) = (n, b, stor, code)\} \\
&\qquad \cup \{\mathsf{Stor}_{(id,\,pc)}\,(\mathring{a}, \mathring{pos}, \mathring{v}, cd) \mid \exists n, b, stor, code.\ \sigma(a) = (n, b, stor, code) \wedge stor\,[pos] = v \wedge pos \leq 2^{256}\} \\
\mathsf{stackToArray}\,(\epsilon, \mathring{\cdot}) &:= (0, \lambda x.\ 0) \\
\mathsf{stackToArray}\,(x :: s, \mathring{\cdot}) &:= \mathit{let}\ (size, sa) = \mathsf{stackToArray}\,(s, \mathring{\cdot})\ \mathit{in}\ (size + 1, sa^{size}_{\hat{x}})
\end{aligned}
$$

5.2 Abstract Execution Rules

As the state predicates are parametrized by their program points or contract identifiers, the abstract semantics needs to be formulated with respect to program points as well. More precisely, this means that for each program counter of a contract $c \in C_k$ a set of Horn clauses is created that describes the semantics of the instruction of the corresponding contract at this program counter. Formally, a function $(\!|\cdot|\!)^{id^*, C_k}_{(id,pc)}$ is defined that creates the required set of rules given that the instruction *inst* is at position *pc* of contract's *c* (with identifier *id*) code. The translation is parametrized by the contract c^* (more precisely its identifier id^*) under analysis and the set of known contracts C_k. As we will discuss later, this information will be exploited when creating the abstract rules for call instructions.

Table 4 shows the formal translation of the semantics of the contracts in C_k: The abstract semantics for a contract *c* is translated by applying the function $(\!|\cdot|\!)^{id^*, C_k}_{(id,pc)}$ to all its instructions with the corresponding identifier and program counters as argument. In addition the Code_{id} predicates are initialized with (the integer representation of) the code of the contracts.

Table 5 shows the definition of $(\!|\cdot|\!)^{id^*, C_k}_{pp}$ for the ADD instruction. The main functionality of the rule is described by the Horn clause 1 that describes how the machine stack and the gas evolve when executing ADD. First the precondition checks whether or not sufficient amount of gas and stack elements are available. Then the two (abstract) top elements $\hat{x}$ and $\hat{y}$ are extracted from the stack and their sum is written to the top of the stack while reducing the overall stack size by 1. In addition, the local gas value is reduced by 3 in an abstract fashion. In the memory rule

[5] Given that memory is represented as an array, modeling memory usage would require a rich set of array operations that are however not supported by the fixed point engines of modern SMT solvers.

Table 4: Abstraction function for small-step rules

$$
\begin{aligned}
\alpha_C\,(C_k, id^*) := & \bigcup_{(id,\,a,\,code) \in C_k \wedge code\,[pc] = inst \wedge 0 \leq pc < |code|} (\!|inst|\!)^{C_k, id^*}_{(id,pc)} \\
& \cup \{\mathsf{Code}_{id}\,(pc, v) \mid \exists a, code.\ (id, a, code) \in C_k \\
& \qquad \wedge\ code\,[pc] = v \wedge 0 \leq pc < |code|\}
\end{aligned}
$$

Table 5: Abstract execution rules for ADD

$$(\!|\mathsf{ADD}|\!)^{C_k, id^*}_{(id,pc)} =$$

$$\{\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \hat{gas} \mathbin{\widehat{\geq}} 3 \wedge \hat{x} = sa[size-1] \wedge \hat{y} = sa[size-2] \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-1, sa^{size-2}_{\hat{x} \mathbin{\hat{+}} \hat{y}}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 3, cd), \quad (1)$$

$$\mathsf{Mem}_{(id,\ pc)}\ (\hat{pos}, \hat{va}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{gas}, \hat{aw}, cd) \wedge size > 1 \wedge \hat{gas} \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\hat{pos}, \hat{va}, cd), \quad (2)$$

$$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \hat{n}, \hat{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{gas}, \hat{aw}, cd) \wedge size > 1 \wedge \hat{gas} \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \hat{n}, \hat{b}, cd), \quad (3)$$

$$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \hat{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{gas}, \hat{aw}, cd) \wedge size > 1 \wedge \hat{gas} \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \hat{pos}, \hat{v}, cd), \quad (4)$$

$$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{gas}, \hat{aw}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd), \quad (5)$$

$$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{gas}, \hat{aw}, cd) \wedge \hat{gas} \mathbin{\widehat{<}} 3 \Rightarrow \mathsf{Exc}_{id}\ (cd)\} \quad (6)$$

(Horn clause 2), again the preconditions are checked and then (as memory is not affected by the ADD instruction) the memory is propagated. This propagation is needed due to the memory predicate's parametrization with the program counter: for making the memory accessible in the next execution step, its values need to be written into the corresponding predicate for the next program counter. Similar rules are created for propagating the global state and persistent storage (cf. Horn clauses 3 and 4). Finally, Horn clauses 5 and 6 characterize the exception cases: an exception while executing the ADD instruction can occur either because of a stack underflow or as the execution runs out of gas. In both cases the exception state is entered which is indicated by recording the relative call depth of the exception in the predicate $\mathsf{Exc}_{id}\ (cd)$.

By allowing gas values to come from the abstract domain, we enable a symbolic treatment of gas. In particular this means that when starting the analysis with gas value $\top$, all gas calculations will directly result in $\top$ again (and could therefore be omitted) and in particular all checks on the gas will result in true and consequently always both paths (regular execution via Horn clauses 1 and 2 and exception via 6) will be triggered in the analysis. [6]

For over-approximating the semantics of calls, more involved abstractions are needed. We will illustrate these abstractions in the following in an intuitive way and refer to Appendix E for technical details.

When calling another contract with a CALL instruction, the recipient of the internal transaction is specified by an abstract value $\hat{to}$ on the machine stack. In this case two different situations can be faced: Either the recipient $\hat{to}$ corresponds to a known contract ($\hat{to} \notin \{a \mid (id, a, code) \in C_k \wedge id \neq id*\} \cup \{\alpha\}$) or the recipient cannot be matched to a contract in C_k as $\hat{to}$ is either $\top$ or a concrete address that is not known. In the first case, the execution of the known contract can be faithfully mimicked by implying the corresponding contract's predicates at program counter 0. In the second case (which we consider more interesting) potentially every contract on the blockchain could be called. For this reason we perform in this case an abstraction that ensures all potential future executions of contract c^* are correctly over-approximated. For this over-approximation we use the following observations:

(1) The instructions DELEGATECALL and CALLCODE should never be used for calling unknown code.
(2) The persistent storage of an account can only be altered by the executing account.
(3) Contracts have a single entry point.

Due to observation (1), we stop our analysis in case that DELEGATECALL or CALLCODE instructions are used for calling unknown addresses. Assuming that we only deal with CALL instructions, we can conclude from observation (2) that when calling an unknown contract, the persistent storage of the original contract c^* can not be altered till re-entering c^*. As we aim at over-approximating all executions of c^*, we will assume c^* to be re-entered at an arbitrary higher call depth with its persistent storage unchanged at this point.

Figure 4 illustrates the over-approximation performed for a CALL instruction transferring x wei to a potentially unknown contract. The abstract execution states c^* are depicted in blue, while the abstract execution states of unknown contracts (that are only known not to be equal to c^*) which are called in between are colored in red. The picture highlights that an arbitrary number of contracts might be called (and executed) before contract c^* is re-entered again. As those contracts, by observation (2), cannot change the persistent storage of c^* (modeled by $\mathsf{Stor}_{(id^*,\ j)}\ (\alpha, \hat{pos}, \hat{v}, cd)$ with α being the address and id^* being the identifier of c^*), the persistent storage of c^* is unchanged when re-entering. As contracts are always entered at program counter 0 (see observation (3)), the values of the global storage can be copied to the corresponding state predicate at $\mathsf{Stor}_{(id^*,\ 0)}$ at a higher call depth. In contrast, the balance of contract c^* might be changed when re-entering as a contract's balance can be affected by other contracts once the control is handed over.

A more formal description of the rules illustrated in Figure 4 is given in Table 6. This table shows an excerpt of (simplified) rules generated from program points holding a CALL instructions. Equation 7 gives the rule for initializing the global state (nonce and balance) of contract c^* when re-entering. In this case, first the preconditions for the call (sufficient elements on the stack and sufficient balance for performing the call) are checked. For the sake of presentation, we omit here the abstract gas treatment. Finally, the nonce n^* of the abstract address α is propagated to predicate $\mathsf{GState}_{id^*,0}$ (at a higher relative call depth) of the re-entered contract c^* and the balance in this state is initialized to be $\top$. This models that the nonce n^* is assumed not to be affected by executions of other contracts than c^* while the balance at the point of re-entering is unknown and therefore over-approximated as $\top$. Note that the contract c with identifier id that performs the call does not necessarily need to be c^*, but might be another contract from C_k performing a call to the unknown. Still, only the future executions of c^* are over-approximated which means that the view of contract c on the global state of c^* is propagated to the point where c^* is re-entered.

[6] For performance reasons, we omit gas completely in the abstract analysis instead of treating it symbolically. These two options are however equivalent.

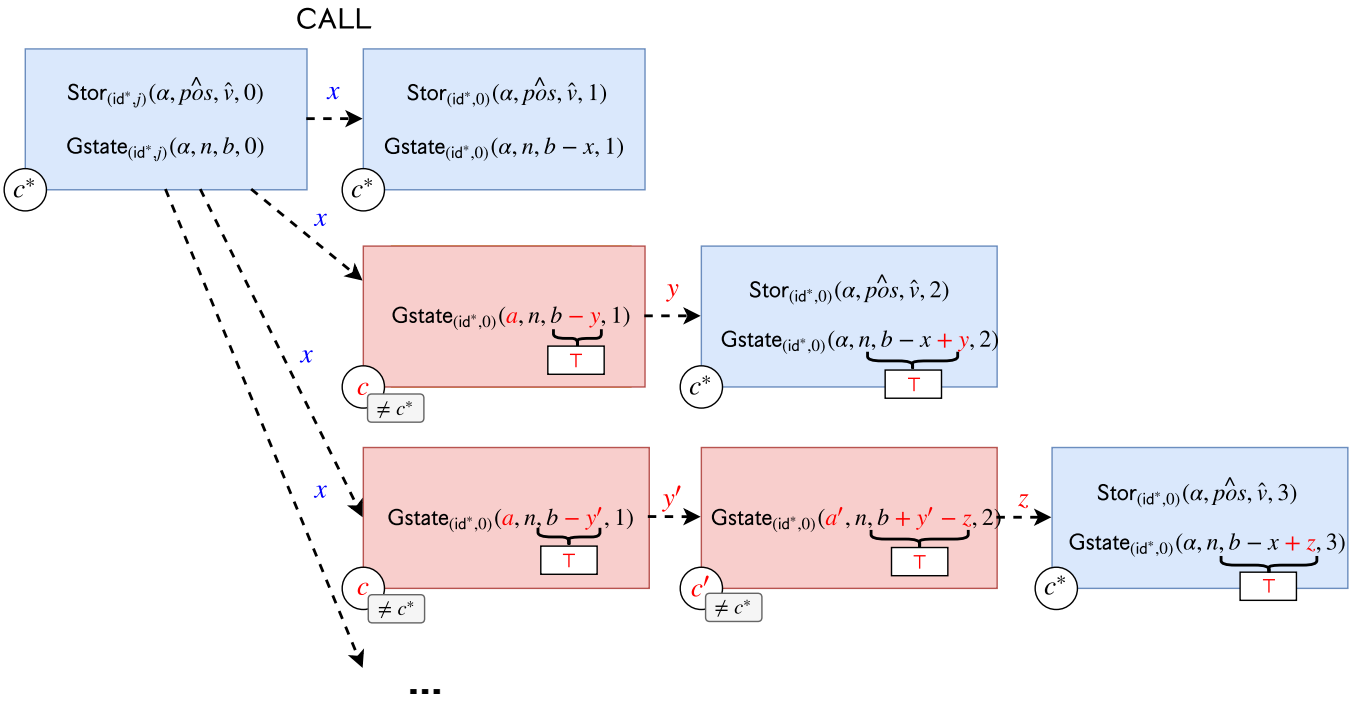

Figure 4: Illustration of the call abstractions

Table 6: Excerpt of the abstract execution rules for CALL

$$
\begin{aligned}
&(\!|\text{CALL}|\!)^{C_k, id^*}_{(id,pc)} = \\
&\quad \{\text{MState}_{(id,\,pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 6 \wedge \hat{to} = sa[size-2] \wedge va = sa[size-3] \wedge \hat{to} \notin \{a \mid (id, a, code) \in C_k \wedge id \neq id^*\} \cup \{\alpha\} && (7)\\
&\quad \wedge\ \text{ExEnv}_{id}\,(\dot{a}, \hat{i}, \hat{v}, \hat{inputsize}, cd) \wedge \text{GState}_{(id,\,pc)}\,(\dot{a}, \hat{n}, \hat{b}, cd) \wedge va \mathrel{\widehat{\leq}} \hat{b} \wedge \ldots \wedge cd' > cd + 1 \wedge \text{GState}_{(id,\,pc)}\,(\alpha, \hat{n}^*, \hat{b}^*, cd) \Rightarrow \text{GState}_{(id^*,\,0)}\,(\alpha, \hat{n}^*, \top, cd'),\\
&\quad \wedge\ \text{MState}_{(id,\,pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 6 \wedge \ldots \wedge \hat{va} \mathrel{\widehat{\leq}} \hat{b} \wedge \dot{a}' \neq \alpha \wedge cd' > cd + 1 \Rightarrow \text{GState}_{(id^*,\,0)}\,(\dot{a}', \top, \top, cd'), && (8)\\
&\quad \wedge\ \text{MState}_{(id,\,pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 6 \wedge \ldots \wedge \hat{va} \mathrel{\widehat{\leq}} \hat{b} \wedge \text{Stor}_{(id^*,\,pc)}\,(\alpha, \hat{pos}, \hat{v}, cd) \wedge cd' > cd + 1 \Rightarrow \text{Stor}_{(id^*,\,0)}\,(\alpha, \hat{pos}, \hat{v}, cd'), \ldots && (9)\\
&\quad \wedge\ \text{MState}_{(id,\,pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 6 \wedge \ldots \wedge \hat{va} \mathrel{\widehat{\leq}} \hat{b} \wedge \dot{a}' \neq \alpha \wedge cd' > cd + 1 \Rightarrow \text{Stor}_{(id^*,\,0)}\,(\dot{a}', \top, \top, cd'), \ldots\} && (10)
\end{aligned}
$$

Equation 8 shows the initialization of the global state for addresses different from α. We do not spell out all checks for the preconditions here as those are the same as in Equation 7. In contrast to the previous rules, also the nonces of the other contracts might be changed in arbitrary fashions and therefore they are over-approximated as $\top$ when re-entering c^*.

Finally, Equation 9 shows how the persistent storage of contract c^* (with address α) is preserved when re-entering at a higher call depth. In contrast, the persistent storage of all other addresses is set to be $\top$ at this point (cf. Equation 10).

We focused here on the over-approximations for the global state when performing a call. In addition, we need to perform several other over-approximations for the other parts of the execution state:

- Similar to the global state, the execution environment for the executions of c^* in the different call depths is considered to be arbitrary and therefore initialized to $\top$ (with the only exception being that it contains α as active account address).
- For returning it is always assumed that potentially the call failed or returned with arbitrary return values.
- After returning the global state is assumed to be altered arbitrarily by the call and therefore its components set to $\top$.

For a complete account of the performed over-approximations we refer to the full specification of the abstract semantics spelled out in Appendix E.

5.3 Limitations

As mentioned before, our analysis will not be able to handle the execution of DELEGATECALL and CALLCODE instructions when they are performed to unknown recipients. We do not consider this limitation a major drawback, because in practice these instructions are mainly used

for calling library functionalities whose bytecode should already be published on the blockchain. In addition, using DELEGATECALL or CALLCODE for calling an unknown contract should always be considered a harmful behavior as handing the control flow over to another contract using one of these instructions implies that this contract is in full power over spending money on the original contract's behalf.

In addition to DELEGATECALL and CALLCODE, we exclude the CREATE instruction from our analysis. The semantics of the CREATE instruction is inherently unsuited to static analysis as when creating new contracts, dynamically generated code (that might depend on the state of blockchain) gets executed. For this reason we omit the treatment of CREATE and leave it as a future challenge.

6 FORMAL RESULTS

We now formally define the scope of our analysis and the formal guarantees that it provides. As we discussed in § 5.3, we exclude contract executions that either perform a CALLCODE or DELEGATECALL to an unknown contract $c^? \notin C_k$ or that execute a CREATE instruction from our analysis. In the following, we will call contract executions not showing this behavior *safety compliant for* C_k and write $\Gamma \vDash S \dot{\rightarrow}^* S'$ for denoting a safety compliant execution.

For the sake of presentation, we will only present the most important definitions and theorems here. For proofs and technical details we refer the reader to Appendix A and Appendix F.

6.1 Expressiveness of the analysis

The previously discussed analysis enables the verification of reachability properties. More precisely, the analysis allows to capture *reachability properties* for the executions of c^* of the form:

$$\begin{aligned} &\mathcal{P}\left(\Psi, c^*, \mathcal{S}_I\right) := \\ &\qquad \forall S, \Gamma, \forall s \in \mathcal{S}_I.\, \neg\exists s', S'. \\ &\qquad \Gamma \vDash s^i{}_{c^*} :: S \dot{\rightarrow}^* s'{}_{c^*} :: S' + +S \wedge \Psi\left(s', |S'|\right) \end{aligned} \tag{11}$$

where $\Psi\left(\cdot,\cdot\right) : \mathcal{S} \times \mathbb{N} \rightarrow \mathbb{B}$ is a predicate on execution states and relative call depths and $\mathcal{S}_I \subseteq \{s_{c^*} \mid \exists \Gamma.(\Gamma, s)$ *is reachable*$\}$ is a set of potential initial states that the execution of contract c^* may start in.

Intuitively, given a contract c^* to be analyzed, a characterization $\mathcal{S}_I$ of the states that the execution of c^* might start in, as well as a property Ψ characterizing undesired states, our analysis allows us to check whether an execution of c^* started in one of the specified initial states can ever reach a state satisfying Ψ. The property Ψ thereby does not range over whole callstacks, but over individual execution states. Still, it additionally considers the relative call depth of the execution which adds expressiveness to the analysis compared to considering execution states exclusively. We demonstrate in § 6.2 how this feature helps us to analyze the single-entrancy property.

Note that we focus on properties of execution states s_{c^*} that execute the contract c^* and that we do not aim at showing properties for execution states s_c with $c \neq c^*$.

Soundness. For the previously discussed properties, our analysis provides *soundness guarantees*. This means that whenever the analysis reports a property of the form specified in Equation 11 to hold, then this property holds true in the concrete execution. Formally, this result is a consequence of the following soundness theorem:

THEOREM 6.1 (SOUNDNESS). *Let C_k be a set of contracts with unique identifiers and addresses and let $c^* \in C_k$ be the contract with identifier id^* and $\mathring{\cdot}$ be a function defined as $\mathring{a} = if\ (id^*, a, \cdot) \in C_k$ then α else a. Additionally, let S' be an annotated callstack such that $|S'| > 0$. Let s be an execution state that is consistent with C_k and s_{c^*} be well-formed. Then the following property holds for all callstacks S:*

$$\begin{aligned} &\Gamma \vDash s_{c^*} :: S \dot{\rightarrow}^* S' + +S \\ &\implies \forall \Delta_I.\, \alpha_s\left(s, id^*, 0, \mathring{\cdot}\right) <: \Delta_I \implies \exists \Delta_S. \\ &\qquad \Delta_I \cup \alpha_C\left(C_k, id^*\right) \vdash \Delta_S \wedge \alpha_S\left(S', C_k, id^*, 0\right) <: \Delta_S \end{aligned}$$

Where the requirement of s_{c^*} being consistent with C_k and well-formed ensures that the code in the execution environment is the code of contract c^* and the contracts in the global state of s with addresses from C_k also carry the codes as specified in C_k.

The soundness theorem establishes a relation between the small-step executions of a contract and the abstract execution of its abstraction. Intuitively it describes that every concrete execution step in the small-step can be mimicked by an abstract one in the abstract semantics. More precisely, it states that from every over-approximation Δ_I of the abstraction of execution state s_{c^*} one can – given the abstract execution rules $\alpha_C\left(C_k, id^*\right)$ obtained from translating the contracts in C_k – derive an over-approximation Δ_S of any abstracted intermediate call stack S' reachable from s_{c^*}.

6.2 Verification of Security Properties

In order to make the security properties from § 4 accessible to our analysis, we over-approximate them as reachability properties.

Over-approximating single-entrancy. We introduce a reachability property called *call unreachability* that implies single-entrancy. Intuitively, a contract c is *call unreachable* if when being executed in a fresh machine state starting, it is not possible to reach a call instruction of the very same contract c on a higher call level.

Definition 6.2 (call unreachability). A contract $c \in C$ is call unreachable if for all regular execution states (μ, ι, σ) such that $(\mu, \iota, \sigma)_c$ is well formed and $\mu = (g, 0, \lambda x.\, 0, 0, \epsilon)$ for some $g \in \mathbb{N}$, it holds that for all transaction environments Γ and all call stacks S

$$\neg \exists s \in \mathcal{S}, S' \in \mathbb{S}_n.\, \Gamma \vDash (\mu, \iota, \sigma)_c :: S \;\rightarrow^* s_c :: S' + + S$$
$$\wedge\, |S'| > 0 \;\wedge\; code\,(c)\,[s.\mu.\mathsf{pc}] \in Inst_{call}$$

Where the set $Inst_{call}$ of call instructions is defined as

$$Inst_{call} = \{\mathsf{CALL}, \mathsf{CALLCODE}, \mathsf{DELEGATECALL}, \mathsf{CREATE}\}$$

Formally, we state the relation between single-entrancy and call unreachability in the following theorem:

THEOREM 6.3. *Call unreachability implies single-entrancy.*

Intuitively, this theorem holds as an internal transaction can only be initiated by the execution of a call instruction. Consequently, for excluding that an internal transaction was initiated after re-entering, it is sufficient to ensure that no call instruction is reachable at this point. In addition, as all contracts start their executions in a fresh machine state (program counter and active words set to 0, empty stack, memory initialized to 0) when being initially called, it is sufficient to check all executions of contract c that started in such a state. Appendix B describes in detail how to express single entrancy in terms of a reachability query.

Over-approximating dependency properties. All other properties presented in [18], and in particular those discussed in § 4.3, constitute simple value independency properties. Inspired by [33], we over-approximate those kinds of properties through a dependency analysis, which we encoded as a reachability property in EtherTrust. In a nutshell, we propagate dependency labels along explicit flows in the expected fashion and capture implicit flows by labeling whole abstract execution states in the case when labeled values affect the control flow. E.g., whenever encountering a conditional jump instruction with a labeled condition, we label all states reachable from this point. In this fashion we can check whether some value influences the call behavior of a contract c^* by checking whether it is ever possible to reach a state of c^* at a program point with a call instruction that is labeled or where labeled values are arguments to the call instruction.

7 EXPERIMENTAL EVALUATION

Based on the abstract semantics discussed in § 5, we developed EtherTrust – a static analyzer for EVM bytecode. EtherTrust verifies reachability and value dependency in a fully automated fashion, even assuming an entirely unknown blockchain environment. The sources of EtherTrust, a cross platform build, as well as all datasets used for evaluation are made available online[7].

7.1 Mode of Operation

EtherTrust proceeds by translating contract code provided in the bytecode format into an internal Horn clause represenation, as specified in § 5. This Horn clause representation, together with a representation facts over-approximating all potential initial configurations that the contract execution might start in, is handed to the SMT solver Z3 [16] via an API. Thanks to our separate internal representation, we can easily extend the tool to interface with other SMT solvers and therefore to benefit from their specific strengths. EtherTrust automatically generates the queries for verifying single-entrancy and independence of the transaction environment and can be easily extended to also support other reachability properties. For showing that the analyzed contract satisfies a reachability property, the unsatisfiability of the corresponding analysis queries needs to be verified using Z3's fixed point engine SPACER [23]. If all analysis queries are deemed unsatisfiable then the contract under analysis is guaranteed to satisfy the original reachability query according to the soundness Theorem 6.1 presented in § 5. Note that we terminate the analysis without result in case that a contract contains an unsupported instruction such as CREATE and thereby ensure that the preconditions of the soundness theorem are met. Additionally, for now the tool only supports the analysis of single contracts and does not allow for specifying an additional set of known contracts. For this reason, the analysis is also directly aborted in case that a contract contains a CALLCODE or a DELEGATECALL instruction.

7.2 Evaluation

For the performance evaluation, we focus on the verification of single-entrancy (SE) and independence of miner controlled state (MI). For the latter, we consider only the information potentially influenced by the miner and that can directly be accessed by a contract, i.e., the transaction environment and the accounts' balances. The reason for considering these two properties is that variations of them are also supported by the state-of-the-art EVM bytecode analysis tool Oyente [24], which allows us to run a comparative performance evaluation. More precisely, Oyente supports re-entrancy (the dual of single-entrancy ($\overline{\text{SE}}$)) and timestamp dependency (a special case of the dual ($\overline{\text{MI}}$) of MI)[8]. We performed experiments on a server with an 8 core Intel Xeon E5-2690 CPU at 2.60GHz, and 8 GB of RAM, running 64-bit Debian GNU/Linux 8 (jessie).

Benchmark. For illustrating how EtherTrust and Oyente generally compare in terms of recall and precision, we run the two tools on a small benchmark that we assembled by crawling all code snippets from the literature that are meant to enforce MI or SE [4, 10, 15, 18, 24, 34], which were designed by the authors to include interesting corner cases. It shows that besides being sound, EtherTrust is more precise than Oyente on the test cases from the benchmark. The results are summarized in Table 7 reporting precision and soundness metrics. While EtherTrust (as

[7] https://sites.google.com/site/ethertrustweb

[8] We could not compare with ZEUS [22], another state-of-the-art smart contract analysis tool, since it supports only Solidity source code and is unfortunately not publicly available.

Table 7: Results of the evaluation of Oyente (O) and EtherTrust (ET) on the benchmark. The table shows the overall numbers of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) for the evaluation of Oyente and EtherTrust for both SE and MI. We consider a result a positive (negative), if it labels a contracts a vulnerable (secure). In addition the table shows precision and recall on the benchmark for the two tools.

	TP	TN	FP	FN	Precision	Recall
O	6	0	3	2	0.67	0.75
ET	8	1	2	0	0.8	1

Table 8: Evaluation of Oyente (O) and EtherTrust(ET) on the top 10k contracts from the blockchain and the full blockchain, respectively. Column $\sum$ holds the overall number of extracted contracts after cleaning. Columns # ter. SE (# ter. MI) denote the number of contracts on which the analysis terminated for SE (MI). Note that Oyente performs checks for both properties in parallel wherefore the numbers for them agree. Columns # $\overline{\text{SE}}$ (# $\overline{\text{MI}}$) hold the number of contracts reported to violate SE (MI). The columns $\varnothing t$ hold the average running time for the analysis of the corresponding property in seconds.

10k	$\sum$	# ter. SE	# $\overline{\text{SE}}$	# ter. MI	# $\overline{\text{MI}}$	$\varnothing t$
O	148	18	12	(18)	3	26,5
ET		100	4	107	2	2,8

being sound) has a recall of 1, Oyente reports two false negatives on the benchmark and therefore ends up with a recall of 0.75. Furthermore, with precision 0.8, EtherTrust is more precise than Oyente which shows a precision of 0.67 on the benchmark.

Blockchain. For comparing the performance of EtherTrust and Oyente on a more representative set of contracts, we ran the tools on contracts from the Ethereum blockchain (status May the 4th, 2018).

We extracted the top 10000 contracts from the blockchain ranked by their account's balance. We eliminated all duplicates (by MD5 hash over the contract code) and additionally removed all contracts that can trivially be deemed safe for SE and MI as they do not contain any CALL instructions. Interestingly, this cleaning step left us with only 148 different contracts. Out of these, EtherTrust terminated on 100 contracts (with a 2-minute-timeout) for SE, while Oyente only terminated on 18 contracts for the same property and timeout. Table 8 reports the results for both SE and MI. Note, however, that the results for MI are not directly comparable, as Oyente only checks for independence from the block timestamp while EtherTrust checks for the independence from the whole transaction environment and the account's balances.

Overall, EtherTrust proves 96 of the contracts that it terminated on for SE to be secure and flags 4 as potentially vulnerable. We could observe that EtherTrust missed 13 of the contracts that Oyente managed to analyze while Oyente failed on 95 contracts of those that EtherTrust could analyze within the same time limit for SE. Out of the 5 contracts that both of them terminated on, for 2 contracts Oyente and EtherTrust report different results (with EtherTrust labeling contracts vulnerable that are considered secure by Oyente). Consequently, we can not easily tell apart, whether this divergence is caused by an imprecision of EtherTrust or the unsoundness of Oyente. Originally, we were striving for exploring the reasons for these different behaviors by manual investigation, but it turned out that, for the contracts in question, no source code is available and manually validating properties on contracts that consist of several hundreds bytecodes is hardly possible. In general, the poor performance of Oyente in terms of coverage together with the lack of ground truth made it difficult to gain deeper insights on how Oyente and EtherTrust compare in terms of precision on real-life contracts and how badly Oyente is affected in practice by unsoundness. Still, we can observe that EtherTrust outperforms Oyente by one order of magniture in efficiency (3s vs 26,5s average execution time) and coverage (100 vs 18 analyzed contracts).

8 CONCLUSION

We presented Ethertrust, the first sound static analyzer for EVM bytecode. The semantics of smart contracts is abstracted into a set of Horn clauses and security properties are expressed as queries, which are solved using Z3. In particular, Ethertrust supports reachability properties, which we show to suffice to capture the most interesting security properties for smart contracts, such as single entrancy and independence of the transaction environment. The analysis of contracts stored on the blockchain typically takes a few seconds and outperforms Oyente in terms of efficiency and coverage by one order of magnitude on real-life contracts and turns out to be more precise on our benchmark too, despite being the first one in the literature to provide formal security guarantees.

This work opens up several interesting research directions. For instance, we plan to extend our analysis to hyperproperties in order to directly verify those introduced by Grischchenko et al. [18]. Furthermore, we intend to leverage Ethertrust for the verification of contract-specific properties, such as the security of cryptographic libraries used in smart contracts. Finally, we would like to investigate how the ideas underlying our static analysis can be generalized in order to support other emerging contract development platforms [4, 5].

REFERENCES

[1] 2016. The DAO Smart Contract. (2016). Available at http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.
[2] 2017. The Parity Wallet Breach. (2017). Available at https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/.

[3] 2017. The Parity Wallet Vulnerability. (2017). Available at https://paritytech.io/blog/security-alert.html.
[4] 2018. *CertiK: Building Fully Trustworthy Smart Contracts and Blockchain Ecosystems*. Technical Report.
[5] 2018. Lisk. https://lisk.io
[6] 2018. Mythril. Available at https://github.com/ConsenSys/mythril.
[7] 2018. Solgraph. Available at https://github.com/raineorshine/solgraph.
[8] Chandra Adhikari. 2017. Secure Framework for Healthcare Data Management Using Ethereum-based Blockchain Technology. (2017).
[9] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. *CPP. ACM. To appear* (2018).
[10] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *International Conference on Principles of Security and Trust*. Springer, 164–186.
[11] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. Medrec: Using Blockchain for Medical Data Access and Permission Management. In *Open and Big Data (OBD), International Conference on*. IEEE, 25–30.
[12] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 91–96.
[13] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure Derivative Contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 453–467.
[14] Florian Buenzli, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Petar Tsankov, and Martin Vechev. 2017. Securify. (2017). Available at http://securify.ch.
[15] Michael Coblenz. 2017. Obsidian: A safer Blockchain programming language. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 97–99.
[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[17] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. 2017. Betrayal, Distrust, and Rationality: Smart Counter-Collusion Contracts for Verifiable Cloud Computing. (2017).
[18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Proceedings of the 7th International Conference on Principles of Security and Trust (POST)*. Springer, 243–269.
[19] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. 2017. Smart Contract-Based Campus Demonstration of Decentralized Transactive Energy Auctions. In *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017 IEEE*. IEEE, 1–5.
[20] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. Technical Report.
[21] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535.
[22] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. NDSS.
[23] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based Model Checking for Recursive Programs. *Form. Methods Syst. Des.* 48, 3 (June 2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4
[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.
[25] Florian Mathieu and Ryno Mathee. 2017. Blocktix: Decentralized Event Hosting and Ticket Distribution Network. (2017). Available at https://blocktix.io/public/doc/blocktix-wp-draft.pdf.
[26] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. 2017. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. *Proceedings of the Financial Cryptography and Data Security Conference* (2017).
[27] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008). Available at http://bitcoin.org/bitcoin.pdf.
[28] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *arXiv preprint arXiv:1802.06038* (2018).
[29] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. 2017. Trading Stocks on Blocks-Engineering Decentralized Markets. In *International Conference on Design Science Research in Information Systems*. Springer, 474–478.
[30] Trail of Bits. 2018. Manticore: Symbolic Execution for Humans. https://github.com/trailofbits/manticore.
[31] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.
[32] SmartDec. 2018. SmartCheck. https://github.com/smartdec/smartcheck.
[33] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. 2014. Checking Probabilistic Noninterference Using JOANA. *it - Information Technology* 56 (Nov. 2014), 280–287. https://doi.org/10.1515/itit-2014-1051
[34] Matt Suiche. 2017. Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode. *DEF CON* 25 (2017).
[35] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security Assurance for Smart Contract. In *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*. IEEE, 1–5.

A OVER-APPROXIMATING SINGLE ENTRANCY

We start introducing some notions and general properties of the small-step semantics that facilitates the understanding of the abstractions that are performed later on.

As the goal analysis is to show properties of specific contracts, we further introduce, and assume throughout the paper, an annotation to the small step semantics in order to highlight the contract c that is currently executed. To this end we assume contracts to be tuples of the form $(a, code)$ where $a \in \mathcal{A}$ denotes the address of the contract and $code \in [\mathbb{B}]$ denotes the contract's code. We denote the set of contracts by C and assume functions $address\,(\cdot)$ and $code\,(\cdot)$ that extract the contract address and code respectively.

The contract annotations allow for arguing easily about the execution of one contract. We can state the property that the execution of EVM bytecode always starts at an initial state. Or more precisely: An execution of a contract c leading to a reachable execution state must have passed an initial state of c before.

LEMMA A.1. *Let* $(\Gamma, s_c :: S)$ *be a reachable configuration. Then there exists an initial execution state* s^i *such that* $(\Gamma, {s^i}_c :: S)$ *is a reachable configuration and*

$$\Gamma \vDash {s^i}_c :: S \;\rightarrow^* s_c :: S$$

In order to approximate the set of reachable execution states, we introduce the notion of well-formation for annotated execution states. An annotated execution state s_c is well-formed if it holds the code of c as active code in the execution environment.

Definition A.2 (Well-formation of annotated execution state). An annotated execution state s_c is well-formed if one of the following holds:

- $s = EXC$
- $s = HALT(\sigma, \mathsf{gas}, d)$
- $s = (\mu, \iota, \sigma)$ and $\iota.\mathsf{code} = code\,(c)$

In addition we define consistency of an execution state with respect to a set of contracts. Intuitively, an annotated execution state s_c is consistent with a set of contracts $C_C \subseteq C$ if the global state of s maps the addresses in C_C to the same codes as C_C does.

Definition A.3 (Consistency of execution state). An execution state $s \in \mathcal{S}$ is consistent with a set of contracts $C_C \subseteq C$ if one of the following holds:

- $s = EXC$
- $s = HALT(\sigma, \mathsf{gas}, d)$
- $s = (\mu, \iota, \sigma)$ and for all $(a, code) \in C_C$ it holds that $\sigma\,(a) = (n, b, stor, code)$ for some $n, b \in \mathbb{N}$ and $stor \in \mathbb{N}_{256} \to \mathbb{N}_{256}$

In order to justify the decision to take well-formed states that are consistent with their annotated contract $\{c\}$ as an over-approximation of reachable states, we observe that each reachable execution state satisfies these properties.

LEMMA A.4 (WELL-FORMATION OF REACHABLE STATES). *Let $(\Gamma, s_c :: S)$ be a reachable configuration. Then s_c is well-formed and s is consistent with $\{c\}$.*

In addition, we can show that well-formation and consistency with the contract annotation are preserved during execution and hence this property serves as a suitable invariant for execution state.

LEMMA A.5 (PRESERVATION OF WELL-FORMATION). *Let s_c be a well-formed execution state such that s is consistent with $\{c\}$. Let $\Gamma \in \Gamma$ be a transaction environment and $S, S' \in \mathbb{S}_n$ such that $\Gamma \vDash s_c :: S \;\to^*\; S' + +S$. Then for all ${s'}_{c'} \in S'$ it holds that ${s'}_{c'}$ is well-formed and s' is consistent with $\{c'\}$.*

In general, we can show that consistency with respect to an arbitrary subset of contracts is preserved during execution as well. This is as the contract code of an account cannot be altered and contracts cannot be deleted during transaction execution. The effects of the SELFDESTRUCT instruction whose execution schedules the executing contract for deletion only apply after the successful completion of the transaction and consequently are not visible during the small-step execution.

LEMMA A.6 (PRESERVATION OF CONSISTENCY). *Let s be a well-formed execution state and $C_C \subseteq C$ be set of contracts such that C_C is consistent with s. Let $\Gamma \in \Gamma$ be a transaction environment and $S, S' \in \mathbb{S}$ such that $\Gamma \vDash s :: S \;\to^*\; S' + +S$. Then for all ${s'}_{c'} \in S'$ it holds that s' is consistent with C_C.*

PROOF. Proof by induction on the number of small steps. □

Another observation is that the influence of the call stack on the contract's execution is limited to the size of the callstack. Depending on the callstack size, an error in the top-level execution might occur due to exceeding the callstack limit.
Formally, we capture this property in the following lemma:

LEMMA A.7 (CALL STACK INDIFFERENCE UP TO SIZE). *Let s be an execution state, c a contract, Γ a transaction environment and let S, S' and U be call stacks such that $|S| = |U|$. Then it holds that*

$$\Gamma \vDash s_c :: S \;\to^*\; S' + +S \;\Leftrightarrow\; \Gamma \vDash s_c :: U \;\to^*\; S' + +U$$

Recall the definition of call unreachability:

Definition A.8 (call unreachability). A contract $c \in C$ is call unreachable if for all regular execution states (μ, ι, σ) such that $(\mu, \iota, \sigma)_c$ is well formed and $\mu = (g, 0, \lambda x.\, 0, 0, \epsilon)$ for some $g \in \mathbb{N}$, it holds that for all transaction environments Γ and all call stacks S

$$\begin{aligned}\neg\exists s \in \mathcal{S}, S' \in \mathbb{S}_n.\, \Gamma \vDash (\mu, \iota, \sigma)_c :: S \;\to^*\; s_c :: S' + +S \\ \wedge\; |S'| > 0 \;\wedge\; code\,(c)\,[s.\mu.\mathsf{pc}] \in Inst_{call}\end{aligned}$$

Where the set $Inst_{call}$ of call instructions is defined as

$$Inst_{call} = \{\mathsf{CALL}, \mathsf{CALLCODE}, \mathsf{DELEGATECALL}, \mathsf{CREATE}\}$$

We need to show that call unreachability is a sufficient criterion for showing single-entrancy. To this end, we first state some properties of smart contract execution.
First, we formalize that every valid contract execution in each call level starts in a fresh machine state.

LEMMA A.9 (CONTRACT EXECUTION FROM FRESH MACHINE STATE). *Let $(\Gamma, s_c :: S)$ be a reachable configuration. Then there exist gas value $g \in \mathbb{N}$, execution environment ι and global state σ such that $(\Gamma, ((g, 0, \lambda x.\, 0, 0, \epsilon), \iota, \sigma)_c :: S)$ is a reachable configuration and*

$$\Gamma \vDash ((g, 0, \lambda x.\, 0, 0, \epsilon), \iota, \sigma)_c :: S \;\to^*\; s_c :: S$$

PROOF. Proof by induction on the number of small steps. □

We state two general properties that relate contract code and the call stacks during execution. First, if an element is added to the callstack, then this happened due to the execution of a call instruction. Second, if an element is added to the call stack during execution and the previous calling state stayed unchanged, then adding the new call stack element was the first execution step of the sequence.

LEMMA A.10 (CALLSTACK GROWTH). *Let* $(\Gamma, s_c :: S)$ *be a configuration and* $\Gamma \vDash s_c :: S \rightarrow S'$ *such that* $|S| + 1 < |S'|$. *Then* $S' = {s'}_{c'} :: s_c :: S$ *for some* $s' \in \mathcal{S}$, $c' \in C$ *and* $s.\iota.\mathsf{code}[s.\mu.\mathsf{pc}] \in Inst_{call}$.

LEMMA A.11 (PRESERVATION OF CALLER STATE). *Let* (Γ, S) *be a configuration and* $\Gamma \vDash S \rightarrow^* S' + +S$ *for some* $S' \in \mathbb{S}_n/\{\epsilon\}$. *Then there is a state* $s \in \mathcal{S}$ *and a contract* $c \in C$ *such that* $\Gamma \vDash S \rightarrow s_c :: S$ *and* $\Gamma \vDash s_c :: S \rightarrow^* S' + +S$.

THEOREM A.12. *If a contract* $c \in C$ *is call unreachable then* c *is also single-entrant.*

PROOF. Assume c not to be single-entrant. Then there exists a reachable configuration $(\Gamma, s_c :: S)$ and states s, s' and a callstack S' such that $\Gamma \vDash s_c :: S \rightarrow^* {s'}_c :: S' + +s_c :: S$ (*) and a state s'' and a contract c'' such that $\Gamma \vDash {s'}_c :: S' + +s_c :: S \rightarrow^* {s''}_{c'} :: {s'}_c :: S' + +s_c :: S$ (**). We show that that there is an initial machine state $\mu = (g, 0, \lambda x.\, 0, 0, \epsilon)$, an execution environment ι and a global state σ such that (μ, ι, σ) is well formed and that $\Gamma \vDash (\mu, \iota, \sigma)_c :: S \rightarrow^* {s'}_c :: (S' + +[s_c]) + +S$. By Lemma A.9 we conclude that there exists fitting μ, ι, σ such that $(\Gamma, (\mu, \iota, \sigma)_c :: S)$ is a reachable configuration and consequently $(\mu, \iota, \sigma)_c$ is well-formed (by Lemma A.4) and $\Gamma \vDash (\mu, \iota, \sigma)_c :: S \rightarrow^* s_c :: S$. By (*) we can conclude that $\Gamma \vDash (\mu, \iota, \sigma)_c :: S \rightarrow^* {s'}_c :: S' + +s_c :: S$. It is left to show that $code\,(c)[s'.\mu.\mathsf{pc}] \in Inst_{call}$. By (**) and Lemma A.11, we can conclude that there is a state s^* and a contract c^* such that $\Gamma \vDash {s'}_c :: S' + +s_c :: S \rightarrow {s^*}_{c^*} :: {s'}_c :: S' + +s_c :: S$. This gives us with Lemma A.10 that $s'.\iota.\mathsf{code}[s'.\mu.\mathsf{pc}] \in Inst_{call}$. As $(\Gamma, {s'}_c :: S' + +s_c :: S)$ is a reachable configuration, ${s'}_c$ is well-formed and consequently $s'.\iota.\mathsf{code} = code\,(c)$ so the claim follows. □

B SINGLE ENTRANCY AS A REACHABILITY QUERY

The call unreachability property falls exactly in the spectrum of properties that can be checked by our analysis (as discussed in Section 6.1). More concretely, we can check this property for a contract c^* with code $code$ and (unknown and therefore symbolic) address α using the following steps:

1) We model the execution state $(\mu, \iota, \sigma)_{c^*}$ by initializing the state predicates at program point $(id^*, 0)$ (where id^* is a freely chosen identifier) at relative call depth 0 with $\top$ for all values that are not further specified and only initializing the stack and local memory precisely as well as setting abstract address α as actor in $\mathsf{ExEnv}_{id^\star}$ at call depth 0. For the resulting abstract configuration it holds that $\Pi :> \alpha_s\,((\mu, \iota, \sigma)_c, id^*, 0, \dot{:})$ for all possible execution states (μ, ι, σ) with a fresh machine state μ.

2) We translate the code of contract c^* and of all other contracts that we want to incorporate in our analysis to Horn clauses. This is done by first picking unique identifiers (different from id^*) for the contract to be analyzed so that formally we can represent them as a set C_k. Note that for all contracts but c^*, which carries symbolic address α, we need to give a concrete address on the blockchain. This is necessary as a contract can only be considered known when it is already deployed on the blockchain. Finally we perform the translation as defined in Table 4 to obtain an abstract contract Δ.

3) We generate reachability queries for checking whether a program point of c^* containing a call instruction can ever be reached at a higher call level. Note that due to CALLCODE and DELEGATECALL this might also happen when c^* executes code of another contract in C_k. More precisely, we will query for the derivability of the elements of the following set of queries (where queries are assumed to be conjunctions of facts):

$$\{\mathsf{MState}_{(id,\,\mathsf{pc})}\,((size, sa), aw, gas, cd) \wedge \mathsf{ExEnv}_{id}\,(\alpha, \hat{\imath}, \hat{va}, \hat{size}, cd) \mid cd > 0 \wedge (id, \cdot, code) \in C_k \wedge code\,[pc] \in Inst_{call}\}$$

For checking pure reachability of a program point it is sufficient to check for the reachability of any fact that is indexed by the program point as it holds that, without putting any constraints on their arguments, all of those facts are reachable if and only if the others are. If none of those facts can be derived then contract c^* is by the soundness property (Theorem 6.1) guaranteed not to be single-entrant.

C SEMANTICS OF CALL AND CREATE FUNCTIONS

We will explain the semantics of those instructions in an intuitive way omitting technical details.

The call instructions initiate a new internal call transaction whose parameters are specified on the machine stack – including the recipient (callee) and the amount of money to be transferred (in the case of CALL and CALLCODE). In addition, the input to the call is specified by providing the corresponding local memory fragment and analogously a memory fragment for the return value.

When executing a call instruction, the specified amount of wei is transferred to the callee and the code of the callee is executed. The different call types differ in the environment that the callee code is executed in. In the case of a CALL instruction, while executing the callee code (only) the account of the callee can be accessed and modified. So intuitively, the control is completely handed to the callee as its code is executed in its own context. In contrast, in the case of CALLCODE, the executed callee code can (only) access and modify the account of the caller. So the callee's code is executed in the caller's context which might be useful for using library functionalities implemented in a separate library contract that e.g., transfers money on behalf on the caller.

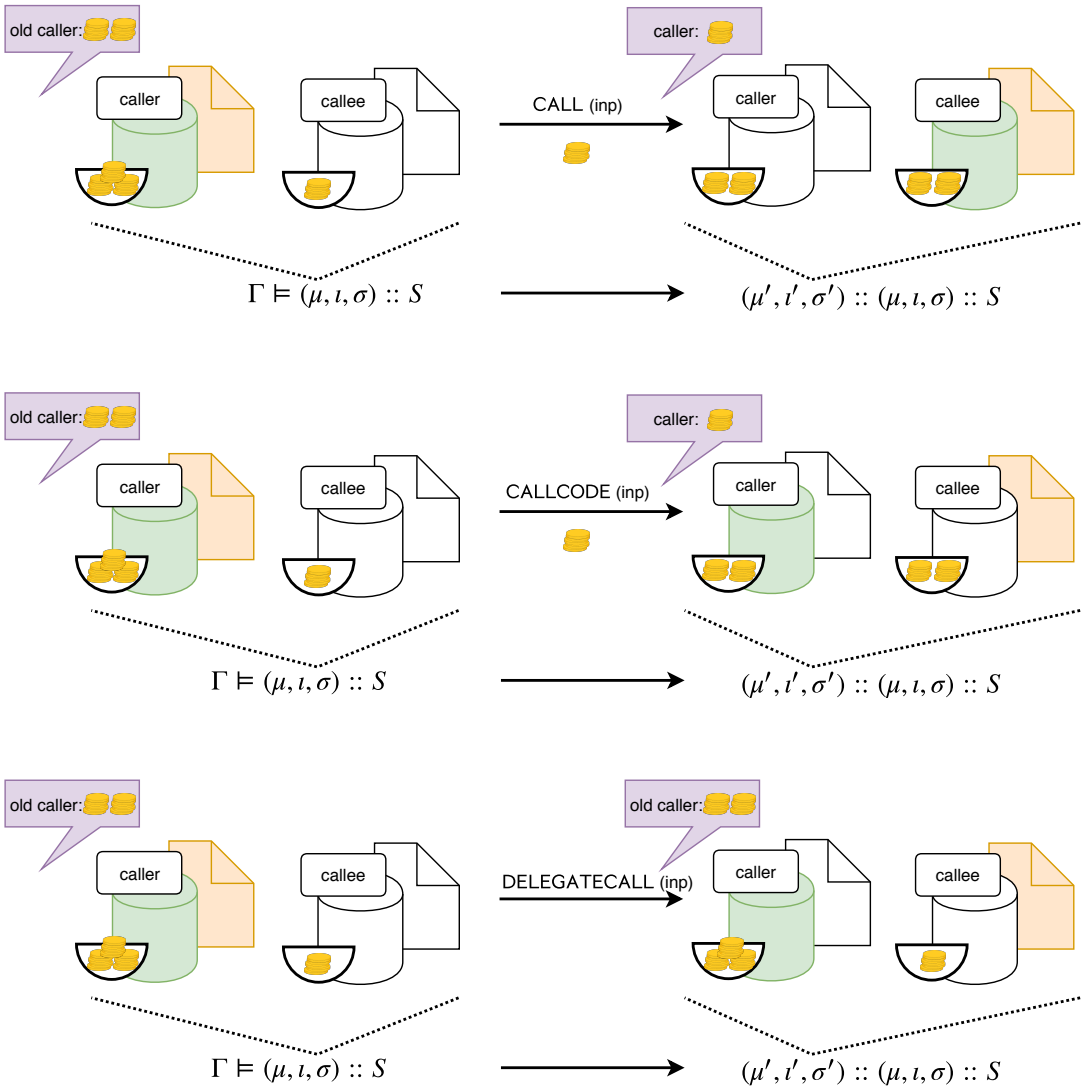

Figure 5: Illustration of of the semantics of different call types

This idea is pushed even further in the DELEGATECALL instruction. This call type does not allow for transferring money and executes the callee's code not only in the caller's context, but even preserves part of the execution environment of the previous call (in particular the call value and the sender information). Intuitively, this instruction resembles adding the callee's code to the caller as an internal function so that calling it does not cause a new internal transaction (even though it formally does).

Figure 5 summarizes the behavior of the different call instructions in EVM bytecode. The executed code of the respective account is highlighted in orange while the accessible account state is depicted in green. The remaining internal transaction information (as specified in the execution environment) on the sender of the internal transaction and the transferred value are marked in violet. In addition, the picture relates the corresponding changes to the small-step semantics: the execution of a call transaction adds a new execution state to the call stack while preserving the old one. The new global state σ' records the changes in the accounts balances, while the new execution environment ι' determines the accessible account (by setting the actor of the internal transaction correspondingly), the code to be executed (by setting code) and further accessible transaction information as the sender, value and input (by setting sender, value and input respectively).

The CREATE instruction initiates an internal transaction that creates a new account. The semantics of this instruction is similar to the one of CALL, with the exception that a fresh account is created, which gets the specified value transferred, and that the input provided to this

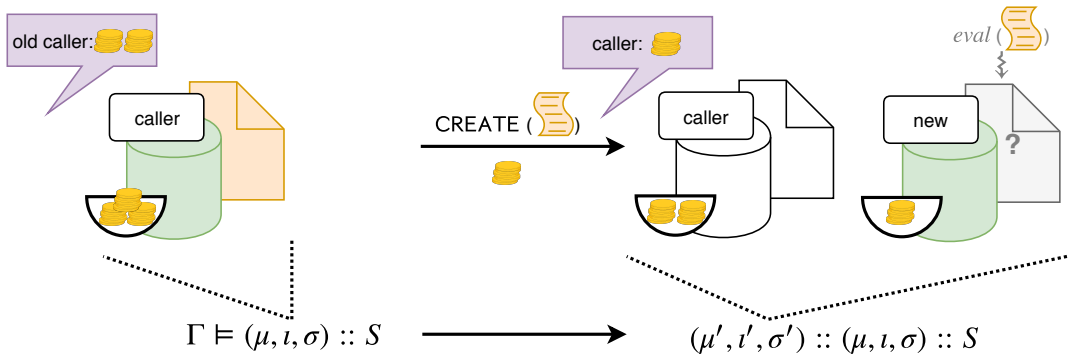

Figure 6: Semantics of the CREATE instruction

internal transaction, which is again specified in the local memory, is interpreted as the initialization code to be executed in order to produce the newly created account's code as output. Figure 6 depicts the semantics of the CREATE instruction in a similar fashion as done for the call instructions before. It is notable that the input to the CREATE instruction is interpreted as code and executed (therefore highlighted in orange) in the context of the newly created contract (highlighted in green). During this execution the newly created contract does not have any contract code itself (therefore depicted in gray), but only after completing the internal transaction the return value of the transaction will be set as the code for the freshly created contract.

D ZEUS

A recently published work is the analysis tool ZEUS [22] that analyses smart contracts written in Solidity using symbolic model checking. The analysis proceeds by translating Solidity code to an abstract intermediate language that again is translated to LLVM bitcode. Finally, existing symbolic model checking tools for LLVM bitcode are leveraged for performing the analysis. The security properties are defined in terms of XACML style policies that are translated to state reachability assertions in the intermediate language (and finally to assertions in LLVM bitcode). The authors evaluate their tool for generic security properties (such as reentrancy) which are however not expressed in terms of policies (which are contract specific), but by an informal description of how to add specific assertions to contracts of interest. In addition, the insertion of assertions is not sufficient for checking, e.g., for reentrancy but additional program modifications are applied to the original contracts. The authors claim their tool to be sound which they support by a proof sketch and empirical results. This claim however has several short comings:

- There is no formal soundness statement made. In particular, there is no formal relation between the policy compliance of Solidity contracts and the analysis results established and also not covered in the proof sketch.
- The proof is more than sketchy and has several holes and at least two flaws: While there is an intuitive argument why given the translation from Solidity to the abstract intermediate language are correct and adding assertions does not influence semantics, there is no proof provided for the statement that the translation from the intermediate language to LLVM bitcode preserves soundness. That this property does not hold is (indirectly) admitted by the authors as they discuss that the compiler optimizations on LLVM bitcode remove relevant contract behavior. Consquently assuming that compiler optimizations on LLVM bitcode are semantics preserving this clearly contradict that the translation from the intermediate language preserves semantics. For one particular optimization, a fix is hard coded, but there is no formal argument given that this particular fix is sufficient for establishing soundness. Also the claim that the the provided translation from Solidity to the intermediate language is faithful can be clearly contradicted. This is due to a clear deviation in the call semantics of the intermediate language from the Solidity semantics. The mechanism underlying Solidity's call functionalities is the one of the CALL instructions in EVM bytecode. In particular, this mechanism determines that the failing of a contract call causes the revocation of the global state to the point of calling. The proposed semantics of the intermediate language however does not allow for such a revocation (even by design). Grishchenko et al. [18] spotted a similar issue in the semantics used in Oyente [24].
- The final results for the predefined properties (such as reentrancy) are not covered by the soundness claim at all as there is no (formal) argument made that the performed program modifications are sound.

E ABSTRACT RULES

E.1 Abstract domain and Abstract operations

Abstract domain. The values of our analysis range over the abstract domain $\hat{D} = \{\top, \bot, \alpha\} \cup \mathbb{N}$. Formally, each element of the abstract domain can be mapped to a subset of $\mathbb{N}$ by the concretisation function $\gamma : \hat{D} \to \mathcal{P}(\mathbb{N})$. More precisely it holds:

$$\begin{aligned} \gamma(\top) &:= \mathbb{N} \\ \gamma(\bot) &:= \emptyset \\ \gamma(\alpha) &:= \mathbb{N} \\ \gamma(n) &:= \{n\} && n \in \mathbb{N} \end{aligned}$$

Accordingly, we can define a corresponding abstraction function $\beta : \mathcal{P}(\mathbb{N}) \to \hat{D}$ mapping subsets of $\mathbb{N}$ to abstract values as follows:

$$\begin{aligned} \beta(\{n\}) &:= n && n \in \mathbb{N} \\ \beta(\emptyset) &:= \bot \\ \beta(\mathbb{N}) &:= \top \\ \beta(N) &:= \top && N \notin \{\mathbb{N}, \emptyset\} \cup \{\{n\} \mid n \in \mathbb{N}\} \end{aligned}$$

Abstract operations. Using the function β, we can easily characterize abstract arithmetic operations as follows:

$$\widehat{op_{bin}}\ (\hat{a}, \hat{b}) := \beta(\{op_{bin}(a, b) \mid a \in \gamma(\hat{a}) \wedge b \in \gamma(\hat{b})\})$$

More precisely this means that only if both operands were unique values, also the abstract operation on those values will result in an abstract value again. This way of defining abstract operations also illustrates well the nature of α: This element only exists in the abstract domain as long as it stays untouched. Otherwise it gets immediately declassified to $\top$.

In the same spirit we can characterize unary operations.:

$$\widehat{op_{un}}\ (\hat{a}) := \beta(\{op_{un}(a) \mid a \in \gamma(\hat{a})\})$$

Using this kind of instructions we can easily show the soundness of abstract operations:

LEMMA E.1. *Let $\hat{a}, \hat{a'}, \hat{b}, \hat{b'} \in \hat{D}$ such that $\hat{a} \sqsubseteq \hat{a'}$ and $\hat{b} \sqsubseteq \hat{b'}$. Then*

$$\widehat{op_{bin}}\ (\hat{a}, \hat{b}) \sqsubseteq \widehat{op_{bin}}\ (\hat{a'}, \hat{b'})$$

Note that in particular this means that

$$op_{bin}(a, b) \sqsubseteq \widehat{op_{bin}}\ (\hat{a}, \hat{b})$$

for $a, b \in \mathbb{N}$ and $\hat{a}, \hat{b} \in \hat{D}$ such that $a \sqsubseteq \hat{a}$ and $b \sqsubseteq \hat{b}$ as the abstract operations agree with with concrete ones given concrete inputs.

A similar property holds for unary operations:

LEMMA E.2. *Let $\hat{a}, \hat{a'} \in \hat{D}$ such that $\hat{a} \sqsubseteq \hat{a'}$. Then*

$$\widehat{op_{un}}\ (\hat{a}) \sqsubseteq \widehat{op_{un}}\ (\hat{a'})$$

Note on the kind of operations transformed. In the small-step semantics presented by Grishchenko et al. [18], arithmetic and logical operations are performed on words of size 256. Here we chose to use the natural number ($\mathbb{N}$) for representing concrete numbers in the abstract domain and hence values on the stack. This however just serves as means of representation and when performing the corresponding arithmetic operations, we make sure to respect the bounds of the corresponding values (by performing all operations modulo 2^{256}) and also to consider whether the corresponding operations are performed on signed or unsigned integers. More precisely, we convert the functions as defined by fun_{bin} in [18] to their corresponding abstract versions. For the sake of representation, we will just use the usual operation symbols (such as $\widehat{+}$) for denoting e.g. abstract addition modulo 2^{256}.

Abstract comparison operators op_{comp} can be seen as logical propositions parametrized by $\hat{D} \times \hat{D}$ and are formally defined as follows:

$$\widehat{op_{comp}}\ (\hat{a}, \hat{b}) :\Leftrightarrow \{(a, b) \mid a \in \gamma(\hat{a}) \wedge b \in \gamma(\hat{b}) \wedge op_{comp}(a, b)\} \neq \emptyset$$

Note that the semantics of the negated versions of the operators differs from the one of negation of the corresponding positive proposition. Accordingly, we define the semantics of an abstract conditional operator as follows:

$$op_{comp}(\hat{a}, \hat{b})\ \widehat{?}\ \hat{s}\ \widehat{:}\ \hat{t} := \beta(\{\hat{s} \mid op_{comp}(\hat{a}, \hat{b})\} \cup \{\hat{t} \mid \overline{op_{comp}}(\hat{a}, \hat{b})\})$$

where $\hat{a}, \hat{b}, \hat{s}$ and $\hat{t}$ are terms over abstract values and operations and $\overline{op_{comp}}$ is the negation of the operator op_{comp}.

E.2 Arithmetic, logical and comparison operations

We define the abstract semantics for binary stack operations. To this end, we recap the definitions from [18]: We define

$$Inst_{bin} := \{\mathsf{ADD}, \mathsf{SUB}, \mathsf{LT}, \mathsf{GT}, \mathsf{EQ}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}, \mathsf{SLT}, \mathsf{SGT}, \mathsf{MUL}, \mathsf{DIV}, \mathsf{SDIV}, \mathsf{MOD}, \mathsf{SMOD}, \mathsf{SIGNEXTEND}, \mathsf{BYTE}\}$$

and

$$\mathrm{cost}_{bin}(i_{bin}) = \begin{cases} 3 & i_{bin} \in \{\mathsf{ADD}, \mathsf{SUB}, \mathsf{LT}, \mathsf{GT}, \mathsf{SLT}, \mathsf{SGT}, \mathsf{EQ}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}, \mathsf{BYTE}\} \\ 5 & i_{bin} \in \{\mathsf{MUL}, \mathsf{DIV}, \mathsf{SDIV}, \mathsf{MOD}, \mathsf{SMOD}, \mathsf{SIGNEXTEND}\} \end{cases}$$

and

$$\mathrm{fun}_{bin}(i_{bin}) = \begin{cases} \lambda(a,b).\ a + b \mod 2^{256} & i_{bin} = \mathsf{ADD} \\ \lambda(a,b).\ a - b \mod 2^{256} & i_{bin} = \mathsf{SUB} \\ \lambda(a,b).\ a < b\ ?\ 1 : 0 & i_{bin} = \mathsf{LT} \\ \lambda(a,b).\ a > b\ ?\ 1 : 0 & i_{bin} = \mathsf{GT} \\ \lambda(a,b).\ a^- < b^-\ ?\ 1 : 0 & i_{bin} = \mathsf{SLT} \\ \lambda(a,b).\ a^- > b^-\ ?\ 1 : 0 & i_{bin} = \mathsf{SGT} \\ \lambda(a,b).\ a = b\ ?\ 1 : 0 & i_{bin} = \mathsf{EQ} \\ \lambda(a,b).\ a \,\&\, b & i_{bin} = \mathsf{AND} \\ \lambda(a,b).\ a \| b & i_{bin} = \mathsf{OR} \\ \lambda(a,b).\ a \oplus b & i_{bin} = \mathsf{XOR} \\ \lambda(a,b).\ a \cdot b \mod 2^{256} & i_{bin} = \mathsf{MUL} \\ \lambda(a,b).\ (b = 0)\ ?\ 0 : \lfloor a \div b \rfloor & i_{bin} = \mathsf{DIV} \\ \lambda(a,b).\ (b = 0)\ ?\ 0 : a \mod b & i_{bin} = \mathsf{MOD} \\ \lambda(a,b).\ (b = 0)?\ 0 : (a = 2^{255} \wedge b^- = -1)?\ 2^{256} : & \\ \quad let\ x = a^- \div b^-\ in\ (sign(x) \cdot \lfloor |x| \rfloor)^+ & i_{bin} = \mathsf{SDIV} \\ \lambda(a,b).\ (b = 0)\ ?\ 0 : (sign(a) \cdot |a| \mod |b|)^+ & i_{bin} = \mathsf{SMOD} \\ \lambda(o,b).\ (o \geq 32)\ ?\ 0 : b[8 \cdot o, 8 \cdot o + 7] \cdot 0^{248} & i_{bin} = \mathsf{BYTE} \\ \lambda(a,b).\ let\ x = 256 - 8(a+1)\ in & \\ \quad let\ s = b\,[x]\ in\ s^x \cdot b[x, 255] & i_{bin} = \mathsf{SIGNEXTEND} \end{cases}$$

where $sign(\cdot) : Int_x \to \{-1, 1\}$ is defined as

$$sign(x) = \begin{cases} 1 & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

and $\&$, $\|$ and $\oplus$ are bitwise and, or and xor, respectively. We let in the following $\widehat{\mathrm{fun}_{bin}}\,(\cdot)$ denote the function that maps binary operations to their corresponding abstract versions as defined in Section E.1.

Then we can compactly define the abstract semantics of binary stack operations $i_{bin} \in Inst_{bin}$ as follows:

$$\begin{aligned}
&(\!|i_{bin}|\!)^{C_k, id^*}_{(id,pc)} = \\
&\{\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 1 \wedge g\hat{a}s \mathrel{\widehat{\geq}} \mathrm{cost}_{bin}(i_{bin}) \wedge \hat{x} = sa[size-1] \wedge \hat{y} = sa[size-2] \\
&\quad \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-1, sa^{\widehat{size-2}}_{\widehat{\mathrm{fun}_{bin}}\ (i_{bin})(\hat{x},\hat{y})}), a\hat{w}, g\hat{a}s \mathrel{\widehat{-}} \mathrm{cost}_{bin}(i_{bin}), cd), \\
&\quad \mathsf{Mem}_{(id,\ pc)}\ (p\hat{o}s, v\hat{a}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size > 1 \wedge g\hat{a}s \mathrel{\widehat{\geq}} \mathrm{cost}_{bin}(i_{bin}) \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (p\hat{o}s, v\hat{a}, cd), \\
&\quad \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \hat{n}, \hat{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size > 1 \wedge g\hat{a}s \mathrel{\widehat{\geq}} \mathrm{cost}_{bin}(i_{bin}) \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \hat{n}, \hat{b}, cd), \\
&\quad \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size > 1 \wedge g\hat{a}s \mathrel{\widehat{\geq}} \mathrm{cost}_{bin}(i_{bin}) \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd), \\
&\quad \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd), \\
&\quad \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge g\hat{a}s \mathrel{\widehat{<}} \mathrm{cost}_{bin}(i_{bin}) \Rightarrow \mathsf{Exc}_{id}\ (cd) \\
&\}
\end{aligned}$$

Next we define the abstract rules for exponentiation EXP. To this end, we first define a macro that summarizes the common precondition of the abstract rules (in case of successful execution):

$$
\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{EXP}} := \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 1 \hat{x} = sa\,[size-1] \hat{y} = sa\,[size-2] \hat{c} = (\hat{y} \mathbin{\widehat{=}} 0)\ \widehat{?}\ 10 \mathbin{\widehat{:}} 10 \mathbin{\widehat{+}} 10 * (1 \mathbin{\widehat{+}} \widehat{\log_{256}}\ (\hat{y})) \wedge g\hat{a}s \mathbin{\widehat{\geq}} \hat{c} \\
&\text{where } \widehat{\log_{256}} = \lambda x.\ \widehat{\lfloor \log_{256}(x) \rfloor}\ .
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{EXP}|\!)^{C_k, id^*}_{(id,pc)} = \\
&\{\mathit{CheckPrems}_{\mathsf{EXP}} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-1, sa^{size-2}_{\widehat{exp}\ (\hat{x},\hat{y})}, a\hat{w}, g\hat{a}s \mathbin{\widehat{-}} \hat{c}, cd), \\
&\mathit{CheckPrems}_{\mathsf{EXP}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (p\hat{o}s, \hat{va}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (p\hat{o}s, \hat{va}, cd), \\
&\mathit{CheckPrems}_{\mathsf{EXP}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \hat{n}, \hat{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \hat{n}, \hat{b}, cd), \\
&\mathit{CheckPrems}_{\mathsf{EXP}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd), \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd), \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 1 \hat{y} = sa\,[size-2] \hat{c} = (\hat{y} \mathbin{\widehat{=}} 0)\ \widehat{?}\ 10 \mathbin{\widehat{:}} 10 \mathbin{\widehat{+}} 10 \mathbin{\widehat{:}} (1 \mathbin{\widehat{+}} \widehat{\log_{256}}\ (\hat{y})) \wedge g\hat{a}s \mathbin{\widehat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd) \\
&\}
\end{aligned}
$$

where $\widehat{exp} = \lambda(x, y).\ \widehat{(x^y) \bmod 2^{256}}$

Another non-standard arithmetic operation is the SHA3 instruction that computes the hash of a provided number. As this is out of scope of our analysis, we directly over-approximate this value as $\top$.

For summarizing the more involved gas check, we again introduce a macro:

$$
\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{SHA3}} := \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 1 \hat{io} = sa\,[size-1] \hat{is} = sa\,[size-2] \wedge a\hat{w}' = \widehat{M}\ (a\hat{w}', \hat{io}, \hat{is}) \\
&\wedge \hat{c} = \widehat{C_{mem}}\ (a\hat{w}, a\hat{w}') \mathbin{\widehat{+}} 30 \mathbin{\widehat{+}} 6 \mathbin{\widehat{\cdot}} \widehat{div_{ceil}}\ (\hat{is}, 32) \wedge g\hat{a}s \mathbin{\widehat{\geq}} \hat{c}
\end{aligned}
$$

where $\widehat{div_{ceil}} = \lambda(\widehat{x, y}).\ \lceil \frac{x}{y} \rceil$

$$
\begin{aligned}
&(\!|\mathsf{SHA3}|\!)^{C_k, id^*}_{(id,pc)} = \\
&\{\mathit{CheckPrems}_{\mathsf{SHA3}} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-1, sa^{size-2}_{\top}, a\hat{w}', g\hat{a}s \mathbin{\widehat{-}} \hat{c}, cd), \\
&\mathit{CheckPrems}_{\mathsf{SHA3}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (p\hat{o}s, \hat{va}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (p\hat{o}s, \hat{va}, cd), \\
&\mathit{CheckPrems}_{\mathsf{SHA3}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \hat{n}, \hat{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \hat{n}, \hat{b}, cd), \\
&\mathit{CheckPrems}_{\mathsf{SHA3}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd), \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd), \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 1 \hat{io} = sa\,[size-1] \hat{is} = sa\,[size-2] \wedge a\hat{w}' = \widehat{M}\ (a\hat{w}', \hat{io}, \hat{is}) \\
&\wedge \hat{c} = \widehat{C_{mem}}\ (a\hat{w}, a\hat{w}') \mathbin{\widehat{+}} 30 \mathbin{\widehat{+}} 6 \mathbin{\widehat{\cdot}} \widehat{div_{ceil}}\ (\hat{is}, 32) \wedge g\hat{a}s \mathbin{\widehat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd) \\
&\}
\end{aligned}
$$

Next, we define the unary stack operations ISZERO and NOT:

$$
\begin{aligned}
&(\!|\mathsf{ISZERO}|\!)^{C_k, id^*}_{(id,pc)} = \\
&\{\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 0 \wedge g\hat{a}s \mathbin{\widehat{\geq}} 3 \wedge \hat{x} = sa[size-1] \wedge \hat{x} \mathbin{\widehat{=}} 0 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_1), a\hat{w}, g\hat{a}s \mathbin{\widehat{-}} 3, cd), \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 0 \wedge g\hat{a}s \mathbin{\widehat{\geq}} 3 \wedge \hat{x} = sa[size-1] \wedge \hat{x} \mathbin{\widehat{\neq}} 0 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_0), a\hat{w}, g\hat{a}s \mathbin{\widehat{-}} 3, cd), \\
&\mathsf{Mem}_{(id,\ pc)}\ (p\hat{o}s, \hat{va}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size > 0 \wedge g\hat{a}s \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (p\hat{o}s, \hat{va}, cd), \\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \hat{n}, \hat{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size > 0 \wedge g\hat{a}s \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \hat{n}, \hat{b}, cd), \\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size > 0 \wedge g\hat{a}s \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, p\hat{o}s, \hat{v}, cd), \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), g\hat{a}s, a\hat{w}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd),
\end{aligned}
$$

$\mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge gâs \mathbin{\widehat{<}} 3 \Rightarrow \mathsf{Exc}_{id}\,(cd)$
}

$$(\!|\mathsf{NOT}|\!)^{C_k, id^*}_{(id,pc)} =$$
$\{\mathsf{MState}_{(id,\,pc)}\,((size, sa), aŵ, gâs, cd) \wedge size > 0 \wedge gâs \mathbin{\widehat{\geq}} 3 \wedge \hat{x} = sa[size-1] \Rightarrow \mathsf{MState}_{(id,\,pc+1)}\,((size, sa^{size-1}_{\widehat{::}\,\hat{x}}), aŵ, gâs \mathbin{\widehat{-}} 3, cd),$
$\mathsf{Mem}_{(id,\,pc)}\,(pô s, vâ, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 0 \wedge gâs \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{Mem}_{(id,\,pc+1)}\,(pô s, vâ, cd),$
$\mathsf{GState}_{(id,\,pc)}\,(\dot{a}, \hat{n}, \hat{b}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 0 \wedge gâs \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{GState}_{(id,\,pc+1)}\,(\dot{a}, \hat{n}, \hat{b}, cd),$
$\mathsf{Stor}_{(id,\,pc)}\,(\dot{a}, pô s, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 0 \wedge gâs \mathbin{\widehat{\geq}} 3 \Rightarrow \mathsf{Stor}_{(id,\,pc+1)}\,(\dot{a}, pô s, \hat{v}, cd),$
$\mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\,(cd),$
$\mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge gâs \mathbin{\widehat{<}} 3 \Rightarrow \mathsf{Exc}_{id}\,(cd)$
$\}$

Finally, we define the abstract semantics for the ternary stack operations ADDMOD and MULMOD:

$$(\!|\mathsf{ADDMOD}|\!)^{C_k, id^*}_{(id,pc)} =$$
$\{\mathsf{MState}_{(id,\,pc)}\,((size, sa), aŵ, gâs, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \wedge \hat{x} = sa[size-1] \wedge \hat{y} = sa[size-2] \wedge \hat{z} = sa[size-3]$
$\Rightarrow \mathsf{MState}_{(id,\,pc+1)}\,((size-2, sa^{size-3}_{\widehat{addmod}\,(\hat{x},\hat{y},\hat{z})}), aŵ, gâs \mathbin{\widehat{-}} 8, cd),$
$\mathsf{Mem}_{(id,\,pc)}\,(pô s, vâ, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \Rightarrow \mathsf{Mem}_{(id,\,pc+1)}\,(pô s, vâ, cd),$
$\mathsf{GState}_{(id,\,pc)}\,(\dot{a}, \hat{n}, \hat{b}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \Rightarrow \mathsf{GState}_{(id,\,pc+1)}\,(\dot{a}, \hat{n}, \hat{b}, cd),$
$\mathsf{Stor}_{(id,\,pc)}\,(\dot{a}, pô s, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \Rightarrow \mathsf{Stor}_{(id,\,pc+1)}\,(\dot{a}, pô s, \hat{v}, cd),$
$\mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\,(cd),$
$\mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge gâs \mathbin{\widehat{<}} 8 \Rightarrow \mathsf{Exc}_{id}\,(cd)\}$

where $\widehat{addmod} = \lambda(x, y, z).\, z = \widehat{0\,?\,0\,:\,}(x + y) \mod z$

$$(\!|\mathsf{MULMOD}|\!)^{C_k, id^*}_{(id,pc)} =$$
$\{\mathsf{MState}_{(id,\,pc)}\,((size, sa), aŵ, gâs, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \wedge \hat{x} = sa[size-1] \wedge \hat{y} = sa[size-2] \wedge \hat{z} = sa[size-3]$
$\Rightarrow \mathsf{MState}_{(id,\,pc+1)}\,((size-2, sa^{size-3}_{\widehat{mulmod}\,(\hat{x},\hat{y},\hat{z})}), aŵ, gâs \mathbin{\widehat{-}} 8, cd),$
$\mathsf{Mem}_{(id,\,pc)}\,(pô s, vâ, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \Rightarrow \mathsf{Mem}_{(id,\,pc+1)}\,(pô s, vâ, cd),$
$\mathsf{GState}_{(id,\,pc)}\,(\dot{a}, \hat{n}, \hat{b}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \Rightarrow \mathsf{GState}_{(id,\,pc+1)}\,(\dot{a}, \hat{n}, \hat{b}, cd),$
$\mathsf{Stor}_{(id,\,pc)}\,(\dot{a}, pô s, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size > 2 \wedge gâs \mathbin{\widehat{\geq}} 8 \Rightarrow \mathsf{Stor}_{(id,\,pc+1)}\,(\dot{a}, pô s, \hat{v}, cd),$
$\mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\,(cd),$
$\mathsf{MState}_{(id,\,pc)}\,((size, sa), gâs, aŵ, cd) \wedge gâs \mathbin{\widehat{<}} 8 \Rightarrow \mathsf{Exc}_{id}\,(cd)\}$

where $\widehat{mulmod} = \lambda(x, y, z).\, z = \widehat{0\,?\,0\,:\,}(x \cdot y) \mod z$

E.3 Accessing the execution environment

$$(\!|\mathsf{ADDRESS}|\!)^{C_k, id^*}_{(id,pc)} = \{$$
$\mathsf{MState}_{(id,\,pc)}\,((size, sa), aŵ, gâs, cd) \wedge size < 1024 \wedge gâs \mathbin{\widehat{\geq}} 2 \wedge \mathsf{ExEnv}_{id}\,(\dot{a}', \hat{\imath}, vâ, \widehat{input size}, cd)$
$\Rightarrow \mathsf{MState}_{(id,\,pc+1)}\,((size+1, sa^{size}_{\dot{a}'}), aŵ, gâs \mathbin{\widehat{-}} 2, cd),$
$\mathsf{Mem}_{(id,\,pc)}\,(\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), aŵ, gâs, cd) \wedge size < 1024 \wedge gâs \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\,pc+1)}\,(\overline{pos}, \hat{v}, cd),$
$\mathsf{GState}_{(id,\,pc)}\,(\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), aŵ, gâs, cd) \wedge size < 1024 \wedge gâs \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\,pc+1)}\,(\dot{a}, \overline{n}, \overline{b}, cd),$
$\mathsf{Stor}_{(id,\,pc)}\,(\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\,pc)}\,((size, sa), aŵ, gâs, cd) \wedge size < 1024 \wedge gâs \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\,pc+1)}\,(\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$
}

$(\!|\mathsf{CALLER}|\!)^{C_k, id^*}_{(id,pc)} = \{$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{\imath}, \hat{va}, \hat{inputsize}, cd)$
$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size + 1, sa^{size}_{\hat{\imath}}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 2, cd),$
$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}, \hat{v}, cd),$
$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$
$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$
}

$(\!|\mathsf{CALLVALUE}|\!)^{C_k, id^*}_{(id,pc)} = \{$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{\imath}, \hat{va}, \hat{inputsize}, cd)$
$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size + 1, sa^{size}_{\hat{va}}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 2, cd),$
$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}, \hat{v}, cd),$
$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$
$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$
}

$(\!|\mathsf{CALLDATASIZE}|\!)^{C_k, id^*}_{(id,pc)} = \{$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{\imath}, \hat{va}, \hat{inputsize}, cd)$
$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size + 1, sa^{size}_{\hat{inputsize}}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 2, cd),$
$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}, \hat{v}, cd),$
$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$
$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$
}

$(\!|\mathsf{CODESIZE}|\!)^{C_k, id^*}_{(id,pc)} = \{$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size + 1, sa^{size}_{|getCode\ (id, C_k)|}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 2, cd),$
$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}, \hat{v}, cd),$
$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$
$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$
$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

$(\!|\mathsf{CALLDATALOAD}|\!)^{C_k, id^*}_{(id,pc)} = \{$

//Case 1: the input data size and position are concrete

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \widehat{\geq} 3 \wedge \overline{pos}' = sa\,[size-1] \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{i}, \hat{va}, \hat{inputsize}, cd)$

$\wedge\ \hat{inputsize}, \overline{pos}' \in \mathbb{N} \wedge \mathsf{Inputdata}_{id}\ (\overline{pos}', \hat{v}_1, cd) \cdots \wedge \mathsf{Inputdata}_{id}\ (\overline{pos}' + k - 1, \hat{v}_k, cd)$

$\wedge\ k = (\hat{inputsize} - \overline{pos}' < 0)\ ?\ 0\ :\ min\ (\hat{inputsize} - \overline{pos}', 32) \wedge \hat{v}'' = (\hat{v}_1\ ||_{k-1}\ (\ldots\ ||_1\ \hat{v}_k))\ ||_{31-k}\ 0^{31-k}$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\hat{v}''}), \hat{aw}, \hat{gas} \widehat{-} 3, cd),$

//Case 2: the input data size is abstract

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \widehat{\geq} 3 \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{i}, \hat{va}, \hat{inputsize}, cd) \wedge \hat{inputsize} \in \{\top, \alpha\}$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), \hat{aw}, \hat{gas} \widehat{-} 3, cd),$

//Case 3: position is abstract

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \widehat{\geq} 3 \wedge \overline{pos}' = sa\,[size-1] \wedge \overline{pos}' \in \{\top, \alpha\}$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), \hat{aw}, \hat{gas} \widehat{-} 3, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \widehat{\geq} 3 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \widehat{\geq} 3 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \widehat{\geq} 3 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \widehat{<} 3 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\mid k \in [1, 32]$

}

E.4 Accessing the transaction environment

As we do not model the transaction environment explicitly, we over-approximate all accesses to it.

$(\!|\mathsf{ORIGIN}|\!)^{C_k, id^*}_{(id,pc)} = \{$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\top}), \hat{aw}, \hat{gas} \widehat{-} 2, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \widehat{<} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

$(\!|\mathsf{GASPRICE}|\!)^{C_k, id^*}_{(id,pc)} = \{$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\top}), \hat{aw}, \hat{gas} \widehat{-} 2, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \widehat{\geq} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \widehat{<} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

$$
\begin{aligned}
&(\!|\mathsf{COINBASE}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\top}), a\hat{w}, g\hat{a}s \mathrel{\widehat{-}} 2, cd),\\
&\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge g\hat{a}s \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{TIMESTAMP}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\top}), a\hat{w}, g\hat{a}s \mathrel{\widehat{-}} 2, cd),\\
&\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge g\hat{a}s \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{NUMBER}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\top}), a\hat{w}, g\hat{a}s \mathrel{\widehat{-}} 2, cd),\\
&\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge g\hat{a}s \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{DIFFICULTY}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\top}), a\hat{w}, g\hat{a}s \mathrel{\widehat{-}} 2, cd),\\
&\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge g\hat{a}s \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{GASLIMIT}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\top}), a\hat{w}, g\hat{a}s \mathrel{\widehat{-}} 2, cd),\\
&\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1024 \wedge g\hat{a}s \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),
\end{aligned}
$$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathrel{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\}$

$(\!|\mathsf{BLOCKHASH}|\!)^{C_k, id^*}_{(id,pc)} = \{$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 20 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 20, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 20 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 20 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 20 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 20 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\}$

E.5 Accessing the global state

$(\!|\mathsf{BALANCE}|\!)^{C_k, id^*}_{(id,pc)} = \{$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \wedge \dot{a}' = sa\,[size-1] \wedge \dot{a}' \in \mathcal{A}_{known} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}', \overline{n}', \overline{b}', cd)$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\overline{b}'}), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 400, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \wedge \dot{a}' = sa\,[size-1] \wedge \dot{a}' \notin \mathcal{A}_{known}$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 400, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 400 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\mid \mathcal{A}_{known} = \{a \mid (id, a, code) \in C_k \wedge id \neq id^*\} \cup \{\alpha\}$

$\}$

The instruction EXTCODESIZE writes the size of code of the specified address to the stack if the address is known (in the set C_k). Otherwise $\top$ is written to the stack.

$(\!|\mathsf{EXTCODESIZE}|\!)^{C_k, id^*}_{(id,pc)} = \{$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \wedge \dot{a}' = sa\,[size-1] \wedge \dot{a}' \in \mathcal{A}_{known}$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{|code|}), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 400, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \wedge \dot{a}' = sa\,[size-1] \wedge \dot{a}' \notin \mathcal{A}_{known}$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 400, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 400 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 400 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\mid code \in \{code \mid (id, a, code) \in C_k \wedge a = \dot{a}' \wedge id \neq id^*\} \cup \{code \mid (id^*, a, code) \in C_k\}$

}

E.6 Stack operations

Instead of defining one Horn clause, we provide a template for the different PUSHx commands that should be used for defining own Horn clauses for each of the PUSH commands. Note that the rules also depend on the size of the accounts code.

Throughout the definition of the abstract rules, we will need to make use of operations that operate on machine words which are technically integers. As our analysis however works on naturals, we define these bit vector specific operations (as extracting a sub bit vector and concatenation two bit vectors) to operate on naturals. *Note.* The positions and length given as arguments to the function refer to the corresponding byte positions (length in bytes). As the rules require the operations only on a byte level, this improves the readability of the rules using the extraction and concatenation function.

$$v[left, right] := \left\lfloor \frac{v}{2^{(31-right)\cdot 8}} \right\rfloor \mod 2^{(right-left)\cdot 8} \qquad left, right \in [0, 31] \wedge left \leq right$$

$$v_1 \,||_{l_2}\, v_2 := v_1 \cdot 2^{l_2 \cdot 8} + v_2$$

Note. In the definition of concatenation, the length of the second vector is needed in order to know how much the values of v_1 need to be lifted.

In the following, we additionally assume k to be defined as follows:

$$k = min\,(pc + x, |code|)$$

where

$$(c, a, code) \in C_k$$

Instead of using the Code. $(\cdot, \cdot)$ predicate one could as well directly extract the needed values from the code (which might be easier) and precalculate the v value.

$$
\begin{aligned}
&(\!|\mathsf{PUSH}x|\!)^{C_k, id^*}_{(id,pc)} = \{ \\
&\mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \,\widehat{\geq}\, 3 \wedge \mathsf{Code}_c\,(pc+1, v_1) \cdots \wedge \mathsf{Code}_{id}\,(pc+x, v_k) \\
&\wedge v = (v_1 \,||_{31-(x-k)}\, (\ldots \,||_1\, v_k)) \,||_{x-k}\, 0^{x-k} \Rightarrow \mathsf{MState}_{(id,\ pc+(1\ +\ x))}\,((size+1, sa^{size}_v), \hat{aw}, \hat{gas} \,\widehat{-}\, 3, cd), \\
&\mathsf{Mem}_{(id,\ pc)}\,(\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \,\widehat{\geq}\, 3 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ (1\ +\ x))}\,(\overline{pos}, \hat{v}, cd), \\
&\mathsf{GState}_{(id,\ pc)}\,(\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \,\widehat{\geq}\, 3 \Rightarrow \mathsf{GState}_{(id,\ pc+\ (1\ +\ x))}\,(\dot{a}, \overline{n}, \overline{b}, cd), \\
&\mathsf{Stor}_{(id,\ pc)}\,(\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \,\widehat{\geq}\, 3 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ (1\ +\ x))}\,(\dot{a}, \overline{pos}, \hat{v}, cd), \\
&\mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \,\widehat{<}\, 3 \Rightarrow \mathsf{Exc}_{id}\,(cd), \\
&\mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\,(cd) \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{POP}|\!)^{C_k, id^*}_{(id,pc)} = \{ \\
&\mathsf{MState}_{(\mathsf{c},\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \,\widehat{\geq}\, 2 \Rightarrow \mathsf{MState}_{(\mathsf{c},\ pc+\ 1)}\,(\hat{gas} \,\widehat{-}\, 2, (size-1, sa), \hat{aw}, cd), \\
&\mathsf{Mem}_{(id,\ pc)}\,(\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \,\widehat{\geq}\, 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\,(\overline{pos}, \hat{v}, cd), \\
&\mathsf{GState}_{(id,\ pc)}\,(\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \,\widehat{\geq}\, 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\,(\dot{a}, \overline{n}, \overline{b}, cd) \\
&\mathsf{Stor}_{(id,\ pc)}\,(\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \,\widehat{\geq}\, 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\,(\dot{a}, \overline{pos}, \hat{v}, cd) \\
&\mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \,\widehat{<}\, 2 \Rightarrow \mathsf{Exc}_{id}\,(cd) \\
&\}
\end{aligned}
$$

We provide templates for abstract execution rules of the DUP and the SWAP instructions. The following definitions hold for all $n \in [1, 16]$:

$$
\begin{aligned}
&(\!|\mathsf{DUP}n|\!)^{C_k, id^*}_{(id,pc)} = \{ \\
&\mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge size > n-1 \wedge \hat{v}' = sa\,[size-n] \wedge \hat{gas} \,\widehat{\geq}\, 3 \\
&\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\,((size+1, sa^{size}_{\hat{v}'}), \hat{aw}, \hat{gas} \,\widehat{-}\, 3, cd), \\
&\mathsf{Mem}_{(id,\ pc)}\,(\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge size > n-1 \wedge \hat{gas} \,\widehat{\geq}\, 3 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\,(\overline{pos}, \hat{v}, cd),
\end{aligned}
$$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge size > n-1 \wedge \hat{gas} \mathrel{\widehat{\geq}} 3 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge size > n-1 \wedge \hat{gas} \mathrel{\widehat{\geq}} 3 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 3 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < n \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

$$
\begin{aligned}
&(\!|\mathsf{SWAP}n|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > n \wedge \hat{v} = sa\,[size-1] \wedge \hat{v}' = sa\,[size-(n+1)] \wedge \hat{gas} \mathrel{\widehat{\geq}} 3\\
&\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, (sa^{size-1}_{\hat{v}'})^{size-(n+1)}_{\hat{v}}), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 3, cd),\\
&\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > n \wedge \hat{gas} \mathrel{\widehat{\geq}} 3 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > n \wedge \hat{gas} \mathrel{\widehat{\geq}} 3 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > n \wedge \hat{gas} \mathrel{\widehat{\geq}} 3 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 3 \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < n+1 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\}
\end{aligned}
$$

E.7 Jumps

Next we introduce rules for JUMPI. For determining validity of a JUMPI instruction we need to decide whether the code jumps to a valid destination. To this end we use the function $D\,(\cdot)$ defined in [18] for determining those program counters with a JUMPDEST instruction. As this information is known statically, the corresponding check for membership can be easily encoded.

We first define again a macro for checking preconditions:

$$
\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{JUMPI}} :=\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \hat{gas} \mathrel{\widehat{\geq}} 10 \wedge \hat{i} = sa[size-1] \wedge \hat{cond} = sa[size-2] \wedge \hat{i} \in D\,(\mathit{getCode}\,(id, C_k)) \cup \{\top, \alpha\}
\end{aligned}
$$

where $\mathit{getCode}\,(\cdot,\cdot)$ extracts the code of the contract with id from C_k

We need to consider that in the case that the jump destination specified on the stack is an abstract value it needs to be considered that this might be potentially both a valid and an invalid jump destination.

$$
\begin{aligned}
&(\!|\mathsf{JUMPI}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{=}} 0 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-2, sa), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 10, cd),\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{\neq}} 0 \wedge j \mathrel{\widehat{=}} \hat{i} \Rightarrow \mathsf{MState}_{(id,\ \mathsf{j})}\ ((size-2, sa), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 10, cd),\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{=}} 0 \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{\neq}} 0 \wedge j \mathrel{\widehat{=}} \hat{i} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ \mathsf{j})}\ (\overline{pos}, \hat{v}, cd),\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{=}} 0 \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{\neq}} 0 \wedge j \mathrel{\widehat{=}} \hat{i} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ \mathsf{j})}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{=}} 0 \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ \mathsf{pc}+1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathit{CheckPrems}_{\mathsf{JUMPI}} \wedge \hat{cond} \mathrel{\widehat{\neq}} 0 \wedge j \mathrel{\widehat{=}} \hat{i} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ \mathsf{j})}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 10 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \hat{i} = sa[size-1] \wedge \hat{i} \notin D\,(\mathit{getCode}\,(id, C_k)) \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\mid j \in D\,(\mathit{getCode}\,(id, C_k))\\
&\}
\end{aligned}
$$

In a similar fashion, unconditional jumps can be defined. First, we define a macro:

$$CheckPrems_{\mathsf{JUMP}} :=$$
$$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{gas} \mathrel{\widehat{\geq}} 8 \wedge \hat{i} = sa[size-1] \wedge \hat{i} \in D\,(getCode\,(id, C_k)) \cup \{\top, \alpha\}$$

Next, we define the abstract execution rules:

$$\begin{aligned}
&(\!|\mathsf{JUMP}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&CheckPrems_{\mathsf{JUMP}} \wedge j \mathrel{\widehat{=}} \hat{i} \Rightarrow \mathsf{MState}_{(id,\ \mathsf{j})}\ ((size-1, sa), \hat{aw}, \hat{gas} \mathrel{\widehat{-}} 8, cd),\\
&CheckPrems_{\mathsf{JUMP}} \wedge j \mathrel{\widehat{=}} \hat{i} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ \mathsf{j})}\ (\overline{pos}, \hat{v}, cd),\\
&CheckPrems_{\mathsf{JUMP}} \wedge j \mathrel{\widehat{=}} \hat{i} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ \mathsf{j})}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&CheckPrems_{\mathsf{JUMP}} \wedge j \mathrel{\widehat{=}} \hat{i} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ \mathsf{j})}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 8 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \hat{i} = sa[size-1] \wedge \hat{i} \notin D\,(getCode\,(id, C_k)) \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\mid j \in D\,(getCode\,(id, C_k))\\
&\}
\end{aligned}$$

Finally, we define the abstract semantics for the JUMPDEST instruction:

$$\begin{aligned}
&(\!|\mathsf{JUMPDEST}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(\mathsf{c},\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{\geq}} 1 \Rightarrow \mathsf{MState}_{(\mathsf{c},\ pc+\ 1)}\ (\hat{gas} \mathrel{\widehat{-}} 1, (size, sa), \hat{aw}, cd),\\
&\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{\geq}} 1 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{\geq}} 1 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd)\\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{\geq}} 1 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd)\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\\
&\}
\end{aligned}$$

E.8 Memory operations

For defining the abstract semantics of MLOAD, we need to lift some of the concrete functions used for calculating memory extension and gas calculation to the abstract setting as described in E.1. More precisely, we have the functions $\widehat{M}\ (\hat{aw}, \hat{o}, \hat{s})$ for abstract memory extension, and the function $\widehat{C_{mem}}\ (\hat{aw}, \hat{aw'})$ for abstractly calculating the cost for memory extension.

In addition, we lift the previously defined extract and concatenation functions to the abstract setting. More specifically, we write $\hat{v}\hat{[}\mathit{left}, \mathit{right}\hat{]}$ for extracting (concrete) positions *left* to *right* of abstract value $\hat{v}$, and $\hat{v}_1 \mathbin{\widehat{||}}_{l_2} \hat{v}_2$ to concatenate the abstract values $\hat{v}_1$ and $\hat{v}_2$ (assuming $\hat{v}_2$ having the concrete size l_2).

Next, we define a macro for the checks that need to be performed for all of the abstract execution rules for MLOAD.

$$CheckPrems_{\mathsf{MLOAD}} :=$$
$$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \overline{pos}' = sa\,[size-1] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \overline{pos}', 32) \wedge \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \mathrel{\widehat{+}} 3 \wedge \hat{gas} \mathrel{\widehat{\geq}} \hat{c}$$

$$\begin{aligned}
&(\!|\mathsf{MLOAD}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&//\mathit{Case1:\ the\ read\ position\ is\ concrete\ and\ all\ read\ values\ are}\\
&CheckPrems_{\mathsf{MLOAD}} \wedge \overline{pos}' \in \mathbb{N} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}', \hat{v}_1, cd) \cdots \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}' + 31, \hat{v}_{32}, cd)\\
&\wedge \hat{v} = (\hat{v}_1 \mathbin{\widehat{||}}_{31} (\ldots \mathbin{\widehat{||}}_2 \hat{v}_{31})) \mathbin{\widehat{||}}_1 \hat{v}_{32} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\hat{v}}), \hat{aw'}, \hat{gas} \mathrel{\widehat{-}} \hat{c}, cd),\\
&//\mathit{Case2:\ there\ is\ something\ written\ at}\ \top\\
&CheckPrems_{\mathsf{MLOAD}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\top, \hat{v}, cd) \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), \hat{aw'}, \hat{gas} \mathrel{\widehat{-}} \hat{c}, cd),\\
&//\mathit{Case3:\ the\ read\ position\ is\ abstract}\\
&CheckPrems_{\mathsf{MLOAD}} \wedge \overline{pos}' \in \{\top, \alpha\} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), \hat{aw'}, \hat{gas} \mathrel{\widehat{-}} \hat{c}, cd),
\end{aligned}$$

$$
\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{MLOAD}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathit{CheckPrems}_{\mathsf{MLOAD}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathit{CheckPrems}_{\mathsf{MLOAD}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \overline{pos} = sa\,[size-1] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \overline{pos}, 32) \wedge \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \mathbin{\widehat{+}} 3 \wedge \hat{gas} \mathbin{\widehat{\geq}} \hat{c}\\
&\Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
\}
\end{aligned}
$$

Note that the MLOAD rules causes in practice quite some performance issues. The main issue is that the $\mathsf{Mem}\ (\cdot,\cdot,\cdot)$ predicate needs to be queried at 32 different positions only for that in the end when one of the derived values is $\top$ the whole compound value collapse to $\top$. For this reason we slightly tweaked the memory representation in the practical implementation not to be byte-wise indexed, but word-wise. This ensures that in the case of MLOAD the memory predicate needs to be queried at most 2 times. This change in the representation introduces quite an overhead of rules as a lot of different corner cases need to be considered and also reduces the precision of the memory analysis (as a whole memory word is declassified to $\top$ once one of its bytes has this abstract value. For the sake of highlighting the general abstractions performed for reading and writing in a more concise fashion, we present the abstract rules in the more intuitive memory model.

As for MLOAD, we first define a macro for checking the preconditions for the MSTORE instruction:

$$
\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{MSTORE}} :=\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \overline{pos}' = sa\,[size-1] \wedge \hat{v}' = sa\,[size-2] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \overline{pos}', 32) \mathbin{\widehat{+}} 3\\
&\wedge \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \wedge \hat{gas} \mathbin{\widehat{\geq}} \hat{c}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{MSTORE}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&//\textit{Case1: the write position is concrete}\\
&\mathit{CheckPrems}_{\mathsf{MSTORE}} \wedge \overline{pos}' \in \mathbb{N} \wedge 0 \leq i < 32 \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}' + i, \hat{v}[\hat{i}, i+1], cd),\\
&//\textit{Case2: the write position is abstract}\\
&\mathit{CheckPrems}_{\mathsf{MSTORE}} \wedge \overline{pos}' \in \{\top, \alpha\} \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\top, \hat{v}', cd),\\
&//\textit{Case3: Propagation}\\
&\mathit{CheckPrems}_{\mathsf{MSTORE}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \overline{pos} \mathbin{\widehat{<}} \overline{pos}' \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathit{CheckPrems}_{\mathsf{MSTORE}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \overline{pos} \mathbin{\widehat{\geq}} (\overline{pos}' \mathbin{\widehat{+}} 32) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathit{CheckPrems}_{\mathsf{MSTORE}} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-2, sa), \hat{aw'}, \hat{gas} \mathbin{\widehat{-}} \hat{c}, cd),\\
&\mathit{CheckPrems}_{\mathsf{MSTORE}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathit{CheckPrems}_{\mathsf{MSTORE}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \overline{pos} = sa\,[size-1] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \overline{pos}, 32) \wedge \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \mathbin{\widehat{+}} 3 \wedge \hat{gas} \mathbin{\widehat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
\}
\end{aligned}
$$

Finally, we define the abstract semantics for the MSTORE8 instruction.

$$
\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{MSTORE8}} :=\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \overline{pos}' = sa\,[size-1] \wedge \hat{v}' = sa\,[size-2] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \overline{pos}', 1) \mathbin{\widehat{+}} 3\\
&\wedge \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \wedge \hat{gas} \mathbin{\widehat{\geq}} \hat{c}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{MSTORE8}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&//\textit{Case1: the write position is concrete}\\
&\mathit{CheckPrems}_{\mathsf{MSTORE8}} \wedge \overline{pos}' \in \mathbb{N} \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\overline{pos}', \hat{v}' \mathbin{\widehat{\mod}} 256, cd),\\
&//\textit{Case2: the write position is abstract}\\
&\mathit{CheckPrems}_{\mathsf{MSTORE8}} \wedge \overline{pos}' \in \{\top, \alpha\} \Rightarrow \mathsf{Mem}_{(id,\ pc+1)}\ (\top, \hat{v}' \mathbin{\widehat{\mod}} 256, cd),
\end{aligned}
$$

$//Case3$: *Propagation*

$CheckPrems_{\mathsf{MSTORE8}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \overline{pos} \mathbin{\widehat{<}} \overline{pos}' \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd)$,

$CheckPrems_{\mathsf{MSTORE8}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \overline{pos} \mathbin{\widehat{\geq}} \overline{pos}' \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd)$,

$CheckPrems_{\mathsf{MSTORE8}} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-2, sa), a\hat{w}', g\hat{a}s \mathbin{\widehat{-}} \hat{c}, cd)$,

$CheckPrems_{\mathsf{MSTORE8}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd)$,

$CheckPrems_{\mathsf{MSTORE8}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd)$,

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge \overline{pos} = sa\,[size-1] \wedge a\hat{w}' = \widehat{M}\ (a\hat{w}', \overline{pos}, 1) \wedge \hat{c} = \widehat{C_{mem}}\ (a\hat{w}, a\hat{w}') \mathbin{\widehat{+}} 3 \wedge g\hat{a}s \mathbin{\widehat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd)$,

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

E.9 Storage operations

First, we define a macro for the checks that need to be performed for all of the abstract execution rules for SLOAD.

$$CheckPrems_{\mathsf{SLOAD}} :=$$
$$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 0 \wedge g\hat{a}s \mathbin{\widehat{\geq}} 200$$

$(\!|\mathsf{SLOAD}|\!)^{C_k, id^*}_{(id,pc)} = \{$

$//Case1$: *the read position is concrete*

$CheckPrems_{\mathsf{SLOAD}} \wedge \overline{pos}' = sa\,[size-1] \wedge \overline{pos}' \in \mathbb{N} \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{i}, \hat{va}, \hat{inputsize}, cd) \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}', \overline{pos}', \hat{v}', cd)$

$\Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\hat{v}'}), a\hat{w}, g\hat{a}s \mathbin{\widehat{-}} 200, cd)$,

$//Case2$: *there is something written at* $\top$

$CheckPrems_{\mathsf{SLOAD}} \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{i}, \hat{va}, \hat{inputsize}, cd) \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}', \top, \hat{v}', cd) \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\hat{v}'}), a\hat{w}, g\hat{a}s \mathbin{\widehat{-}} 200, cd)$,

$//Case3$: *the read position is abstract*

$CheckPrems_{\mathsf{SLOAD}} \wedge \overline{pos}' = sa\,[size-1] \wedge \overline{pos}' \in \{\top, \alpha\} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size, sa^{size-1}_{\top}), a\hat{w}, g\hat{a}s \mathbin{\widehat{-}} 200, cd)$,

$CheckPrems_{\mathsf{SLOAD}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd)$,

$CheckPrems_{\mathsf{SLOAD}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd)$,

$CheckPrems_{\mathsf{SLOAD}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd)$,

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge g\hat{a}s \mathbin{\widehat{<}} 200 \Rightarrow \mathsf{Exc}_{id}\ (cd)$,

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

As for SLOAD, we first define a macro for checking the preconditions for the SSTORE instruction:

$CheckPrems_{\mathsf{SSTORE}} :=$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), a\hat{w}, g\hat{a}s, cd) \wedge size > 1 \wedge \overline{pos}' = stackarray\,[size-1] \wedge \hat{v}' = stackarray\,[size-2] \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{i}, \hat{va}, \hat{inputsize}, cd)$

$\wedge\ \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}', \overline{pos}', \hat{v}'', cd) \wedge \hat{c} = (\hat{v}' \mathbin{\widehat{\neq}} 0 \hat{\wedge} \hat{v}'' \mathbin{\widehat{=}} 0)\ \widehat{?}\ 20000 \mathbin{\widehat{:}} 5000 \wedge g\hat{a}s \mathbin{\widehat{\geq}} \hat{c}$

$(\!|\mathsf{SSTORE}|\!)^{C_k, id^*}_{(id,pc)} = \{$

$//Case1$: *the write position is concrete*

$CheckPrems_{\mathsf{SSTORE}} \wedge \overline{pos}' \in \mathbb{N} \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}', \overline{pos}', \hat{v}', cd)$

$//Case2$: *the write position is abstract*

$CheckPrems_{\mathsf{SSTORE}} \wedge \overline{pos}' \in \{\top, \alpha\} \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}', \top, \hat{v}', cd)$,

$//Propagation$ *of the remaining storage*

$CheckPrems_{\mathsf{SSTORE}} \wedge \overline{pos} \mathbin{\widehat{<}} \overline{pos}' \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}', \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}', \overline{pos}, \hat{v}, cd)$,

$CheckPrems_{\mathsf{SSTORE}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}', \overline{pos}, \hat{v}, cd) \wedge \overline{pos} \mathbin{\widehat{\geq}} \overline{pos}' \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}', \overline{pos}, \hat{v}, cd)$,

$//Storage\ propagation\ for\ all\ but\ the\ active\ account$

$CheckPrems_{\mathsf{SSTORE}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \dot{a} \neq \dot{a}' \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$CheckPrems_{\mathsf{SSTORE}} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size-2, sa), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} \hat{c}, cd),$

$CheckPrems_{\mathsf{SSTORE}} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),$

$CheckPrems_{\mathsf{SSTORE}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \overline{pos}' = stackarray\,[size-1] \wedge \hat{v}' = stackarray\,[size-2] \wedge \mathsf{ExEnv}_{id}\ (\dot{a}', \hat{i}, \hat{va}, \hat{inputsize}, cd)$

$\wedge\ \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}', \overline{pos}', \hat{v}'', cd) \wedge \hat{c} = (\hat{v}' \mathbin{\widehat{\neq}} 0 \hat{\wedge} \hat{v}'' \mathbin{\widehat{=}} 0)\ \hat{?}\ 20000 \hat{:} 5000 \wedge \hat{gas} \mathbin{\widehat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

E.10 Accessing the machine state

$$(\!|\mathsf{PC}|\!)^{C_k, id^*}_{(id,pc)} = \{$$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{pc}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 2, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathbin{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

$$(\!|\mathsf{MSIZE}|\!)^{C_k, id^*}_{(id,pc)} = \{$$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\hat{aw}}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 2, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathbin{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

$$(\!|\mathsf{GAS}|\!)^{C_k, id^*}_{(id,pc)} = \{$$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size+1, sa^{size}_{\hat{gas}}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 2, cd),$

$\mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),$

$\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),$

$\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \wedge \mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1024 \wedge \hat{gas} \mathbin{\widehat{\geq}} 2 \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathbin{\widehat{<}} 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size \geq 1024 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

}

E.11 Logging instructions

We provide a template for the abstract execution rules for LOG instructions. The following definitions hold for all $n \in \{1, 2, 3, 4\}$:

First, we define a macro for the precondition checks:

$$CheckPrems_{\mathsf{LOG}n} :=$$

$$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 + n \wedge \hat{io} = sa\,[size - 1] \wedge \hat{is} = sa\,[size - 2] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \hat{io}, \hat{is})$$
$$\wedge\ \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \mathbin{\hat{+}} 375 \mathbin{\hat{+}} 8 \mathbin{\hat{\cdot}} \hat{is} + n \hat{\cdot} 375 \wedge \hat{gas} \mathbin{\hat{\geq}} \hat{c}$$

Next we define the abstract execution rules for LOG instructions

$$\begin{aligned}
&(\!|\mathsf{LOG}n|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathit{CheckPrems}_{\mathsf{LOG}n} \Rightarrow \mathsf{MState}_{(id,\ pc+1)}\ ((size - (2 + n)), sa), \hat{aw'}, \hat{gas} \mathbin{\hat{\frown}} \hat{c}, cd),\\
&\mathit{CheckPrems}_{\mathsf{LOG}n} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd),\\
&\mathit{CheckPrems}_{\mathsf{LOG}n} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathit{CheckPrems}_{\mathsf{LOG}n} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathbin{\hat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 2 + n \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\}
\end{aligned}$$

Note that the main effects of the LOG instructions are only applied after the transaction execution and therefore are not considered in the analysis. The analysis only tracks the impact on the parts of the state that can affect the execution.

E.12 Halting instructions

First we present the rules for STOP:

$$\begin{aligned}
&(\!|\mathsf{STOP}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \Rightarrow \mathsf{Return}_{id}\ (0, \hat{gas}, cd)\\
&\mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{ResGState}_{id}\ (\dot{a}, \overline{n}, \overline{b}, cd)\\
&\mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{ResStor}_{id}\ (\dot{a}, \overline{pos}, \hat{v}, cd)\\
&\}
\end{aligned}$$

Next we define the abstract semantics for the RETURN instruction. To this end, we first define a macro that contains the checks for common preconditions:

$$\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{RETURN}} :=\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \hat{io} = stackarray\,[size - 1] \wedge \hat{is} = stackarray\,[size - 2] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \hat{io}, \hat{is})\\
&\wedge\ \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \wedge \hat{gas} \mathbin{\hat{\geq}} \hat{c}
\end{aligned}$$

Finally, we can give the abstract semantics for RETURN:

$$\begin{aligned}
&(\!|\mathsf{RETURN}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\mathit{CheckPrems}_{\mathsf{RETURN}} \hat{is}' = \hat{is} = \alpha\ ?\ \top\ :\ \hat{is} \Rightarrow \mathsf{Return}_{id}\ (\hat{is}', \hat{gas} \mathbin{\hat{\frown}} \hat{c}, cd),\\
&//\textit{The return data predicate is only initialized if the size is concrete}\\
&\mathit{CheckPrems}_{\mathsf{RETURN}} \wedge \hat{io}, \hat{is} \in \mathbb{N} \wedge i < \hat{is} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\hat{io} + i, \hat{v'}, cd), \Rightarrow \mathsf{Res}_{id}\ (i, \hat{v'}, cd),\\
&\mathit{CheckPrems}_{\mathsf{RETURN}} \wedge \hat{io} \in \{\top, \alpha\} \wedge \hat{is} \in \mathbb{N} \wedge i < \hat{is} \Rightarrow \mathsf{Res}_{id}\ (i, \top, cd),\\
&\mathit{CheckPrems}_{\mathsf{RETURN}} \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{ResGState}_{id}\ (\dot{a}, \overline{n}, \overline{b}, cd),\\
&\mathit{CheckPrems}_{\mathsf{RETURN}} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{ResStor}_{id}\ (\dot{a}, \overline{pos}, \hat{v}, cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 2 \Rightarrow \mathsf{Exc}_{id}\ (cd),\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 1 \wedge \hat{io} = stackarray\,[size - 1] \wedge \hat{is} = stackarray\,[size - 2] \wedge \hat{aw'} = \widehat{M}\ (\hat{aw'}, \hat{io}, \hat{is})\\
&\wedge\ \hat{c} = \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \wedge \hat{gas} \mathbin{\hat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd)\\
&\}
\end{aligned}$$

Next, we consider the SELFDESTRUCT instruction. As most of the effects of this instruction are only applied after the transaction execution has finished, the abstract semantics of this instructions is fairly simple to describe. The only difficulty is that the two cases of whether the beneficiary is known needs to be handled distinctly as these cases result in different costs for the execution of SELFDESTRUCT.

We define again a macro for the preconditions that we this time parametrize by a flag that indicates which cost should be used for the gas calculation.

$$
\begin{aligned}
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(\mathsf{b}) := \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \dot{a}' = sa\,[size-1] \wedge \hat{c} = \mathsf{b} = 1\ ?\ 5000\ :\ 37000 \wedge \hat{gas} \mathrel{\widehat{\geq}} \hat{c}
\end{aligned}
$$

$$
\begin{aligned}
&(\!|\mathsf{SELFDESTRUCT}|\!)^{C_k, id^*}_{(id,pc)} = \{ \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(1) \wedge \dot{a}' \in \mathcal{A}_{known} \Rightarrow \mathsf{Return}_{id}\ (0, \hat{gas} \mathrel{\widehat{-}} \hat{c}, cd), \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(1) \wedge \dot{a}' \in \mathcal{A}_{known} \wedge \mathsf{ExEnv}_{id}\ (\dot{a}'', \hat{\imath}, \hat{va}, \hat{inputsize}, cd) \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}', \overline{n}', \overline{b}', cd) \\
&\wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}'', \overline{n}'', \overline{b}'', cd) \Rightarrow \mathsf{ResGState}_{id}\ (\dot{a}', \overline{n}, \overline{b}' \mathrel{\widehat{+}} \overline{b}'', cd) \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(1) \wedge \dot{a}' \in \mathcal{A}_{known} \wedge \mathsf{ExEnv}_{id}\ (\dot{a}'', \hat{\imath}, \hat{va}, \hat{inputsize}, cd) \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}'', \overline{n}'', \overline{b}'', cd) \\
&\Rightarrow \mathsf{ResGState}_{id}\ (\dot{a}'', \overline{n}'', 0, cd) \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(1) \wedge \dot{a}' \in \mathcal{A}_{known} \wedge \mathsf{ExEnv}_{id}\ (\dot{a}'', \hat{\imath}, \hat{va}, \hat{inputsize}, cd) \wedge \dot{a} \neq \dot{a}' \wedge \dot{a} \neq \dot{a}'' \\
&\wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{ResGState}_{id}\ (\dot{a}, \overline{n}, \overline{b}, cd) \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(1) \wedge \dot{a}' \in \mathcal{A}_{known} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{ResStor}_{id}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(\mathsf{b}) \wedge \dot{a}' \notin \mathcal{A}_{known} \Rightarrow \mathsf{Return}_{id}\ (0, \hat{gas} \mathrel{\widehat{-}} \hat{c}, cd), \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(\mathsf{b}) \wedge \dot{a}' \notin \mathcal{A}_{known} \wedge \mathsf{ExEnv}_{id}\ (\dot{a}'', \hat{\imath}, \hat{va}, \hat{inputsize}, cd) \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}'', \overline{n}'', \overline{b}'', cd) \\
&\Rightarrow \mathsf{ResGState}_{id}\ (\dot{a}'', \overline{n}'', 0, cd) \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(\mathsf{b}) \wedge \dot{a}' \notin \mathcal{A}_{known} \wedge \mathsf{ExEnv}_{id}\ (\dot{a}'', \hat{\imath}, \hat{va}, \hat{inputsize}, cd) \wedge \dot{a} \neq \dot{a}' \wedge \dot{a} \neq \dot{a}'' \\
&\wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{ResGState}_{id}\ (\dot{a}, \overline{n}, \overline{b}, cd) \\
&\mathit{CheckPrems}_{\mathsf{SELFDESTRUCT}}(\mathsf{b}) \wedge \dot{a}' \notin \mathcal{A}_{known} \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{ResStor}_{id}\ (\dot{a}, \overline{pos}, \hat{v}, cd) \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size < 1 \Rightarrow \mathsf{Exc}_{id}\ (cd) \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas} \mathrel{\widehat{<}} 5000 \Rightarrow \mathsf{Exc}_{id}\ (cd) \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge size > 0 \wedge \dot{a}' = sa\,[size-1] \wedge \dot{a}' \notin \mathcal{A}_{known} \wedge \hat{gas} \mathrel{\widehat{<}} 37000 \Rightarrow \mathsf{Exc}_{id}\ (cd) \\
&\ \mid \mathsf{b} \in \{0,1\}\ \wedge\ \mathcal{A}_{known} = \{a \mid (id, a, code) \in C_k \wedge id \neq id^*\} \cup \{\alpha\} \\
&\}
\end{aligned}
$$

Finally, we specify the abstract semantics of the designated INVALID instruction that directly fails:

$$
\begin{aligned}
&(\!|\mathsf{INVALID}|\!)^{C_k, id^*}_{(id,pc)} = \{ \\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \Rightarrow \mathsf{Exc}_{id}\ (cd) \\
&\}
\end{aligned}
$$

E.13 Contract calls

Next we consider the Horn clauses for calling. We will present rules for calling known contracts (those contracts from the set C_k) as well as for calling unknown contracts. While in the case of known contracts, the execution of those contracts will be over-approximated in a fashion that is similarly precise to the analysis of contract c^*. Instead, in the case when an unknown contract is called, a more coarse abstraction is applied.

(1) *A known contract is called.* The known contracts can be determined from the set C_k that is given as argument to the $(\!|\cdot|\!)^{C_k,\cdot}_{(\cdot,\cdot)}$ function. By the definition of the representation function, the semantics of the codes of all of these contracts is translated to Horn clauses using the state predicates parametrized by the corresponding ids as given in C_k. Consequently the abstract execution of these contracts can be triggered by implying instances of these contract's state predicates. Note that when another contract code is called the address of the contract to be called is specified on the stack. This means that it is not known upfront to the code of which contract the rules need to link. We make use the same trick used for JUMP commands. For the whole set of contracts that can possibly be called, we generate rules and then incorporate a check inside the rules that make sure that only the correct rule can be applied. Consequently the abstract rules for calling a known contract need to be generated for all ids *id_to* of contracts that are callable according to C_k.

(2) *An unknown contract is called.* If the address of the contract called is not a concrete address contained in C_k or equal to the abstract address α, then we do not try to mimick the execution of the called contract faithfully, but just over-approximate the effects that such a contract execution might have on the execution (an potential future executions of) the contract c^*.

Most of the call rules share a common bunch of preconditions that need to be satisfied for calling. For the sake of better readability, we will summarize these preconditions with some macros that will be defined in the following. More precisely, for both of the scenarios of the CALL (calling known and unknown contracts) we distinguish two different cases that have different preconditions.

(1) The conditions for calling are satisfied at call time. These conditions include that the call stack limit is not reached yet and the executing account's balance suffices to transfer the specified amount of money to the callee. In this case the specified account's code is called.
(2) The conditions for the call are not satisfied at call time (as the calling contract's balance is not sufficient or as the call stack limit is reached. In this case the maximal fee for the call is charged and the execution proceeds.

We will define two different macros for more conveniently differentiating these cases. The precondition checks for both cases include the check for the sufficient number of arguments (and extracts them from the stack), calculates the updated number of active words in memory and also performs the gas calculation and checks whether the amount of gas is sufficient to perform the call. If the contract $\hat{to}$ is known (so: $\hat{to} \in \{a \mid (id, a, code) \in C_k \wedge id \neq id^*\} \cup \{\alpha\}$) then the gas can be calculated accordingly. If the contract $\hat{to}$ is not known then the gas needs to be calculated according to both cases. This would result in two potential valid pre-computations. We distinguish these to cases by adding a flag in the pre-computation macro, indicating how the gas was calculated.

For handling abstract gas computation, we need to define some abstract auxiliary functions. These functions are simply lifting those described in [18] to the abstract setting. More precisely we end up with the functions $\widehat{C_{gascap}}\ (\cdot,\cdot,\cdot,\cdot)$, $\widehat{C_{base}}\ (\cdot,\cdot)$.

With all these auxiliary functions in place, we can define the macros for the preconditions of successful calls and calls that fail at call time. The first macro $CheckPrems_{\mathsf{CALL}}()$ that we define just checks for the sufficient number of elements on the stack and for a sufficient amount of gas:

$$
\begin{aligned}
&CheckPrems_{\mathsf{CALL}}(\mathsf{b}) :=\\
&\mathsf{MState}_{(id,\ pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \hat{gas}_{to} = sa[size-1] \wedge \hat{to} = sa[size-2] \wedge \hat{va} = sa[size-3] \wedge \hat{io} = sa[size-4] \wedge \hat{is} = sa[size-5]\\
&\wedge \hat{oo} = sa[size-6] \wedge \hat{os} = sa[size-7] \wedge \hat{aw'} = \widehat{M}\ (\widehat{M}\ (\hat{aw}, \hat{io}, \hat{is}), \hat{oo}, \hat{os}) \wedge size > 6 \wedge \hat{c_{call}} = \widehat{C_{gascap}}\ (\hat{va}, \mathsf{b}, \hat{gas}_{to}, \hat{gas})\\
&\wedge \hat{c} = \hat{c_{call}} \mathbin{\widehat{+}} \widehat{C_{base}}\ (\hat{va}, \mathsf{b}) \mathbin{\widehat{+}} \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'}) \wedge \hat{gas} \mathbin{\widehat{\leq}} \hat{c}
\end{aligned}
$$

The second macro $CheckPrems_{\mathsf{CALL}-succ}()$ additionally checks whether also the call time check that the balance of the executing account is sufficient is passed:

$$
\begin{aligned}
&CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) :=\\
&CheckPrems_{\mathsf{CALL}}(\mathsf{b}) \wedge \mathsf{ExEnv}_{id}\ (\dot{a}, \hat{i}, \hat{va'}, \hat{inputsize}, cd) \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \wedge \hat{va} \mathbin{\widehat{\leq}} \overline{b}
\end{aligned}
$$

As we are not tracking the size of the callstack explicitly, we always need to assume that the call fails at call time given that the other preconditions for calling are satisfied. For this reason, we will formulate the rules for failing at call time assuming that only the preconditions of $CheckPrems_{\mathsf{CALL}}(\mathsf{b})$ are satisfied.

Next we can use the macros to formally specify the abstract execution rules for the CALL instruction:

$$
\begin{aligned}
&(\!|\mathsf{CALL}|\!)^{C_k, id^*}_{(id,pc)} = \{\\
&\\
&//\textit{Case 1: the call does not fail at call time}\\
&//\textit{The execution environment of the callee is initiated}\\
&CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \mathsf{ExEnv}_{id}\ (\dot{a}, \hat{i}, \hat{va}, \hat{inputsize}, cd) \wedge \hat{to} = a_{to} \Rightarrow \mathsf{ExEnv}_{id\_to}\ (\hat{to}, \dot{a}, \hat{va}, \hat{is}, cd+1)\\
&\\
&//\textit{Writing the input to the corresponding input predicate}\\
&//\textit{Case a: position and size values are concrete}\\
&CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \hat{is}, \hat{io} \in \mathbb{N} \wedge 0 \leq i < \hat{is} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\hat{io}+i, \hat{v}, cd')\\
&\Rightarrow \mathsf{Inputdata}_{id\_to}\ (i, \hat{v}, cd')\\
&//\textit{Case b: position value is concrete, but size value is not}\\
&CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \hat{io} \in \{\top, \alpha\} \wedge \hat{is} \in \mathbb{N} \wedge 0 \leq i < \hat{is} \Rightarrow \mathsf{Inputdata}_{id\_to}\ (i, \top, cd')\\
&//\textit{Note that we do not consider the cases where the size is abstract, as in this case this is recorded in the execution environment}\\
&//\textit{and it will never be read from the input}
\end{aligned}
$$

//*Abstract machine state of callee is initialized*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \Rightarrow \mathsf{MState}_{(id\_to,\,0)}\ ((0, sa'), 0, \hat{c_{call}}, cd + 1)$

$\mathsf{checkPrems}^{\mathsf{CALL}}_{(id,pc)}\ (\dot{a}, \hat{to}, \hat{va}, \hat{io}, \hat{is}, \hat{aw'}, cd, \hat{c_{call}}, \hat{c}, \mathsf{b}) \wedge \hat{to} = a_{to} \Rightarrow \mathsf{Mem}_{(id\_to,\,0)}\ (\overline{pos}, 0, cd + 1)$

//*Global state is altered according to the call*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{GState}_{(id,\,pc)}\ (\dot{a}, \overline{n}, \overline{b}, cd) \Rightarrow \mathsf{GState}_{(id\_to,\,0)}\ (\dot{a}, \overline{n}, \overline{b} \mathbin{\widehat{\frown}} \hat{va}, cd + 1)$

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{GState}_{(id,\,pc)}\ (\hat{to}, \overline{n'}, \overline{b'}, cd) \Rightarrow \mathsf{GState}_{(id\_to,\,0)}\ (\hat{to}, \overline{n'}, \overline{b'} \mathbin{\widehat{+}} \hat{va}, cd + 1)$

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{GState}_{(id,\,pc)}\ (\dot{a'}, \overline{n'}, \overline{b'}, cd) \wedge \dot{a} \neq \dot{a'} \Rightarrow \mathsf{GState}_{(id\_to,\,0)}\ (\dot{a'}, \overline{n'}, \overline{b'}, cd + 1)$

//*Rules for returning successfully*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \mathsf{MState}_{(id,\,pc)}\ (sa, \hat{aw}, \hat{gas}, cd) \wedge \hat{gas'} = \hat{gas} \mathbin{\widehat{+}} (\hat{gas_r} \mathbin{\widehat{\frown}} \hat{c}) \wedge \hat{to} = a_{to} \wedge \mathsf{Return}_{id\_to}\ (\hat{returndatasize}, \hat{gas_r}, cd + 1)$

$\Rightarrow \mathsf{MState}_{(id,\,pc+1)}\ ((size - 6, sa_1^{size-7}), \hat{aw'}, \hat{gas'}, cd)$

//*The result value is written to specified fraction in memory.*

//*Case a: offset and size are concrete*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{Return}_{id\_to}\ (\hat{returndatasize}, \hat{gas_r}, cd + 1) \wedge \mathsf{Res}_{id\_to}\ (i, \hat{v}, cd + 1) \wedge \hat{k} = \widehat{min}\ (\hat{os}, \hat{returndatasize})$

$\wedge\ \hat{oo}, \hat{k} \in \mathbb{N} \wedge 0 \leq i < \hat{k} \Rightarrow \mathsf{Mem}_{id,\,pc+1}\ (\hat{oo} + i, \hat{v}, cd)$

//*Case b: size is abstract*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{Return}_{id\_to}\ (\hat{returndatasize}, \hat{gas_r}, cd + 1) \wedge \hat{k} = \widehat{min}\ (\hat{os}, \hat{returndatasize}) \wedge \hat{k} \in \{\top, \alpha\}$

$\Rightarrow \mathsf{Mem}_{id,\,pc+1}\ (\top, \top, cd)$

//*Case c: offset is abstract*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{oo} \in \{\top, \alpha\} \Rightarrow \mathsf{Mem}_{id,\,pc+1}\ (\top, \top, cd)$

//*The remaining memory is propagated.*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{Return}_{id\_to}\ (\hat{returndatasize}, \hat{gas_r}, cd + 1) \wedge \hat{k} = \widehat{min}\ (\hat{os}, \hat{returndatasize}) \wedge \mathsf{Mem}_{id,\,pc}\ (\overline{pos}, \hat{v}, cd)$

$\wedge\ \overline{pos} \mathbin{\widehat{<}} \hat{oo} \Rightarrow \mathsf{Mem}_{id,\,pc+1}\ (\overline{pos}, \hat{v}, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{Return}_{id\_to}\ (\hat{returndatasize}, \hat{gas_r}, cd + 1) \wedge \hat{k} = \widehat{min}\ (\hat{os}, \hat{returndatasize}) \wedge \mathsf{Mem}_{id,\,pc}\ (\overline{pos}, \hat{v}, cd)$

$\wedge\ \overline{pos} \mathbin{\widehat{\geq}} \hat{oo} \mathbin{\widehat{+}} \hat{k} \Rightarrow \mathsf{Mem}_{id,\,pc+1}\ (\overline{pos}, \hat{v}, cd)$

//*The global state at returning is propagated (in case of succesful execution)*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{ResGState}_{id\_to}\ (\dot{a'}, \overline{n'}, \overline{b'}, cd + 1) \Rightarrow \mathsf{GState}_{(id,\,pc+1)}\ (\dot{a'}, \overline{n'}, \overline{b'}, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{ResStor}_{id\_to}\ (\dot{a'}, \overline{pos}, \hat{v}, cd + 1) \Rightarrow \mathsf{Stor}_{(id,\,pc+1)}\ (\dot{a'}, \overline{pos}, \hat{v}, cd)$

//*Exceptional returning*

//*0 is written on the stack and all gas for the call is lost*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{MState}_{(id,\,pc)}\ ((size, sa), \hat{aw}, \hat{gas}, cd) \wedge \mathsf{Exc}_{id\_to}\ (cd + 1)$

$\Rightarrow \mathsf{MState}_{(id,\,pc+1)}\ ((size - 6, sa_0^{size-7}), \hat{aw'}, \hat{gas} \mathbin{\widehat{\frown}} \hat{c}, cd)$

//*Memory and global state are propagated*

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{Exc}_{id\_to}\ (cd + 1) \wedge \mathsf{Mem}_{id,\,pc}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{id,\,pc+1}\ (\overline{pos}, \hat{v}, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{Exc}_{id\_to}\ (cd + 1) \wedge \mathsf{GState}_{(id,\,pc)}\ (\dot{a'}, \overline{n'}, \overline{b'}, cd) \Rightarrow \mathsf{GState}_{(id,\,pc+1)}\ (\dot{a'}, \overline{n'}, \overline{b'}, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(1) \wedge \hat{to} = a_{to} \wedge \mathsf{Exc}_{id\_to}\ (cd + 1) \wedge \mathsf{Stor}_{(id,\,pc)}\ (\dot{a'}, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\,pc+1)}\ (\dot{a'}, \overline{pos}, \hat{v}, cd)$

$| (id\_to, a_{to}, code) \in \{(id, a, code) \mid ((id, a, code) \in C_k\ \wedge\ id \neq id^*) \vee (id = id^* \wedge a = \alpha \wedge \exists a'.(id^*, a', code) \in C_k\}$

}

The rules for failing at call time (calculating the gas assuming that the called contract exists) apply for both cases that the callee is know or unknown:

$\cup\{$

//Case 2: call fails at call time

$CheckPrems_{\mathsf{CALL}}(1) \Rightarrow \mathsf{MState}_{(id,\ pc+\ 1)}\ ((size-6, sa_0^{size-7}), \hat{aw'}, \hat{gas} \mathbin{\hat{-}} \hat{c}, cd)$

$CheckPrems_{\mathsf{CALL}}(1) \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd)$

$CheckPrems_{\mathsf{CALL}}(1) \wedge \mathsf{GState}_{(id,\ pc)}\ (\dot{a}', \overline{n}', \overline{b}', cd) \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}', \overline{n}', \overline{b}', cd)$

$CheckPrems_{\mathsf{CALL}}(1) \wedge \mathsf{Stor}_{(id,\ pc)}\ (\dot{a}', \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}', \overline{pos}, \hat{v}, cd)$

//Case 3: runtime exception

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), aw, gas, cd) \wedge size < 7 \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\mathsf{MState}_{(id,\ pc)}\ ((size, sa), aw, gas, cd) \wedge \hat{gas}_{to} = sa[size-1] \wedge \hat{va} = sa[size-3] \wedge \hat{io} = sa[size-4] \wedge \hat{is} = sa[size-5] \wedge \hat{oo} = sa[size-6]$
$\wedge\ \hat{os} = sa[size-7] \wedge \hat{aw'} = \widehat{M}\ (\widehat{M}\ (\hat{aw}, \hat{io}, \hat{is}), \hat{oo}, \hat{os}) \wedge \hat{c_{call}} = \widehat{C_{gascap}}\ (\hat{va}, 1, \hat{gas}_{to}, \hat{gas}) \wedge \hat{c} = \hat{c_{call}} \mathbin{\widehat{+}} \widehat{C_{base}}\ (\hat{va}, \mathsf{b}) \mathbin{\widehat{+}} \widehat{C_{mem}}\ (\hat{aw}, \hat{aw'})$
$\wedge\ \hat{gas} \mathbin{\hat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\ (cd)$

$\}$

In addition to the rules presented before, we also consider an over-approximated behavior for the case that the callee is not known. As all possible executions of the contract under analysis should be captured, we need to consider two facets:

(1) Calling an unknown contract might change overall state of the contract after returning. It is unclear whether the contract exists at all. It cannot be determined whether the execution failed or was successful and if so which value it returned. In addition, executing unknown code might change the global state in an arbitrary way.
(2) Calling unknown code might cause the contract code to be retriggered in an arbitrary way (on a higher call depth).

$\cup$

$\{$

//Case 1: unknown code is executed and it does not fail at call time

//Both zero or one can be written to the stack

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \Rightarrow \mathsf{MState}_{(id,\ pc+\ 1)}\ ((size-6, sa[size-7 \rightarrow 1], \hat{aw'}, \top, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \Rightarrow \mathsf{MState}_{(id,\ pc+\ 1)}\ ((size-6, sa_0^{size-7}, \hat{aw'}, \top, cd)$

//Arbitrary values can be written to memory in the specified fragment

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge \hat{oo}, \hat{os} \in \mathbb{N} \wedge i \mathbin{\hat{\geq}} oo \wedge i < \hat{oo} + \hat{os} \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (i, \top, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge \hat{oo} \in \{\top, \alpha\} \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\top, \top, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge \hat{os} \in \{\top, \alpha\} \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\top, \top, cd)$

//All values are propagated (as we don't know the returndata size)

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge \mathsf{Mem}_{(id,\ pc)}\ (\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\ pc+\ 1)}\ (\overline{pos}, \hat{v}, cd)$

//After execution the global storage of all accounts can be arbitrary

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \Rightarrow \mathsf{Stor}_{(id,\ pc+\ 1)}\ (\dot{a}', \top, \top, cd)$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \Rightarrow \mathsf{GState}_{(id,\ pc+\ 1)}\ (\dot{a}', \top, \top, cd)$

//The execution of the contract might be reentered at an arbitrary (higer) call level (in an arbitrary execution environment)

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge cd' > cd + 1 \Rightarrow \mathsf{MState}_{(id^*,\ 0)}\ ((0, sa'), 0, \top, cd')$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge cd' > cd + 1 \wedge i \in \mathbb{N} \Rightarrow \mathsf{Mem}_{(id^*,\,0)}\,(i, 0, cd')$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge cd' > cd + 1 \Rightarrow \mathsf{ExEnv}_{id^*}\,(\alpha, \top, \top, \top, cd')$

//The input data does not need to be initialized as its size is anyway unknown

//The global state of all accounts but a might be arbitrary*

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge cd' > cd + 1 \wedge \mathsf{GState}_{(id,\,pc)}\,(\alpha, \overline{n}', \overline{b}', cd) \Rightarrow \mathsf{GState}_{(id^*,\,0)}\,(\alpha, \overline{n}', \top, cd')$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge cd' > cd + 1 \wedge \dot{a}' \in \mathbb{N} \Rightarrow \mathsf{GState}_{(id^*,\,0)}\,(\dot{a}', \top, \top, cd')$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge cd' > cd + 1 \wedge \mathsf{Stor}_{(id,\,pc)}\,(\alpha, \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id^*,\,0)}\,(\alpha, \overline{pos}, \hat{v}, cd')$

$CheckPrems_{\mathsf{CALL}-succ}(\mathsf{b}) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge cd' > cd + 1 \wedge \dot{a}' \in \mathbb{N} \Rightarrow \mathsf{Stor}_{(id^*,\,0)}\,(\dot{a}', \top, \top, cd')$

//Additional failure cases (for the case that a new account was created)

$CheckPrems_{\mathsf{CALL}}(2) \wedge \hat{to} \notin \mathcal{A}_{known} \Rightarrow \mathsf{MState}_{(id,\,pc+\,1)}\,((size - 6, sa_0^{size-7}), a\hat{w}', g\hat{as} \mathbin{\widehat{\frown}} \hat{c}, cd)$

$CheckPrems_{\mathsf{CALL}}(2) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge \mathsf{Mem}_{(id,\,pc)}\,(\overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Mem}_{(id,\,pc+\,1)}\,(\overline{pos}, \hat{v}, cd)$

$CheckPrems_{\mathsf{CALL}}(2) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge \mathsf{GState}_{(id,\,pc)}\,(\dot{a}', \overline{n}', \overline{b}', cd) \Rightarrow \mathsf{GState}_{(id,\,pc+\,1)}\,(\dot{a}', \overline{n}', \overline{b}', cd)$

$CheckPrems_{\mathsf{CALL}}(2) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge \mathsf{Stor}_{(id,\,pc)}\,(\dot{a}', \overline{pos}, \hat{v}, cd) \Rightarrow \mathsf{Stor}_{(id,\,pc+\,1)}\,(\dot{a}', \overline{pos}, \hat{v}, cd)$

$\mathsf{MState}_{(id,\,pc)}\,((size, sa), aw, g\hat{as}, cd) \wedge \hat{to} \notin \mathcal{A}_{known} \wedge g\hat{as}_{to} = sa[size - 1] \wedge \hat{va} = sa[size - 3] \wedge \hat{io} = sa[size - 4]$
$\wedge\ \hat{is} = sa[size - 5] \wedge \hat{oo} = sa[size - 6] \wedge \hat{os} = sa[size - 7] \wedge a\hat{w}' = \widehat{M}\ (\widehat{M}\ (a\hat{w}, \hat{io}, \hat{is}), \hat{oo}, \hat{os}) \wedge c_{\hat{call}} = \widehat{C_{gascap}}\ (\hat{va}, 2, g\hat{as}_{to}, g\hat{as})$
$\wedge\ \hat{c} = c_{\hat{call}} \mathbin{\widehat{+}} \widehat{C_{base}}\ (\hat{va}, \mathsf{b}) \mathbin{\widehat{+}} \widehat{C_{mem}}\ (a\hat{w}, a\hat{w}') \wedge g\hat{as} \mathbin{\widehat{<}} \hat{c} \Rightarrow \mathsf{Exc}_{id}\,(cd)$
$|\ \mathsf{b} \in \{0, 1\}\ \wedge\ \mathcal{A}_{known} = \{a \mid (id, a, code) \in C_k \wedge id \neq id^*\} \cup \{\alpha\}\}$

F SOUNDNESS PROOF

We first state some auxiliary lemmas that will come in handy in the soundness proof. We omit the proofs for these lemmas as those are mostly straight-forward case distinctions and simple inductions.

LEMMA F.1 (CALLSTACK PRESERVATION DURING EXECUTION). *Let (Γ, S) be a configuration such that $\Gamma \vDash U + +S \rightarrow^* U' + +S$. Then the following properties hold:*

- *if $U' = \epsilon$ then $U = \epsilon$*
- *if $U = \epsilon$ and $U' \neq \epsilon$ then there are $s \in \mathcal{S}$, $c \in C$ such that $\Gamma \vDash S \rightarrow s_c :: S$ and $\Gamma \vDash s_c :: S \rightarrow^* U' + +S$.*
- *if $U' \neq \epsilon$ and $\Gamma \vDash U + +S \rightarrow^* S'$ and $\Gamma \vDash S' \rightarrow^* U' + +S$ then exists U'' such that $|U''| > 0$ and $S' = U'' + +S$*

We formally state a useful property for decomposing the results of α_S:

LEMMA F.2. *Let $s_c :: S \in \mathbb{S}_n$, $C_k \subseteq \mathbb{N} \times \mathbb{N}_{160} \times [\mathbb{B}^8]$, $id^* \in \mathbb{N}$ and $cd \in \mathbb{N}$. And let $c^* \in \{(a^*, code^*) \mid (id^*, a^*, code^*) \in C_k\}$. Then we can distinguish the following cases:*

(1) $(c = c^* \vee c \in \{(a, code) \mid (id, a, code) \in C_k \mid\ \wedge id \neq id^*\} \wedge \exists s^*_{c^*}, S', S''.S = S' + +s^*_{c^*} :: S'' \wedge \forall s'_{c'} \in S'.c' \in C_k)$
$\Rightarrow \alpha_S\,(s_c :: S, C_k, id^*, cd) = \alpha_S\,(S, C_k, id^*, cd) \cup \alpha_s\,(s, id^*, |S| + cd, \dot{\cdot})$

(2) $(c \notin \{(a, code) \mid (id, a, code) \in C_k \mid\ \wedge id \neq id^*\} \vee c \in \{(a, code) \mid (id, a, code) \in C_k \mid\ \wedge id \neq id^*\} \wedge \exists s^?_{c^?}, S', S''.S = S' + +s^?_{c^?} :: S'' \wedge \forall s'_{c'} \in S'.c' \neq c^*) \Rightarrow \alpha_S\,(s_c :: S, C_k, id^*, cd) = \alpha_S\,(S, C_k, id^*, cd)$

Intuitively, this lemma allows us to distinguish the cases in which the topmost call stack element is translated and where it is not.
More generally, we can observe, that abstract representation of a callstack is a superset of the abstract representations of its sub-callstacks.

LEMMA F.3. *Let S, S', be annotated callstacks. Furthermore let $id^*, cd \in \mathbb{N}$ and $C_k \subseteq \mathbb{N} \times \mathcal{A} \times [\mathbb{B}^8]$. If there is a callstack S'' such that $S = S'' + +S'$ then*

$$\alpha_S\,(S', C_k, id^*, cd) \subseteq \alpha_S\,(S, C_k, id^*, cd)$$

The representation functions for abstracting the execution environment α_ι and the global state α_σ do not translate the contract codes encoded in there. This is as requiring the initial execution state s being consistent with C_k ensures that the active code in the execution environment and the relevant parts of the global states agree with the information in C_k. This property is preserved during the execution (Lemma A.6).

As contracts cannot be deleted during the execution (the SELFDESTRUCT instruction that allows to delete existing contracts only applies its effects after successful transaction execution and hence does not affect the execution of the transaction itself).

In addition to these Lemmas, we also establish a new notion of well-formation on callstacks. To this end, we first introduce the notion of a *call state*.

Definition F.4 (Call states). A regular execution state (μ, ι, σ) is a call state if the following conditions hold:

- $\omega_{\mu,\iota} = \mathsf{CALL}$
- $\mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s$
- $\sigma(to \mod 2^{160} \neq \bot$
- $|A| + 1 \leq 1024$
- $\sigma(\iota.\mathsf{actor}).\mathsf{b} \geq va$
- $aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os)$
- $c_{call} = C_{gascap}\,(va, 1, g, \mu.\mathsf{gas})$
- $c = C_{base}\,(va, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call}$
- $c \leq \mu.\mathsf{gas}$

So intuitively, a regular execution state is a call state if it satisfies all of the preconditions for executing a CALL instruction. Note that we define here a very restricted version of call states (requiring the executed instruction to be a plain CALL instruction, not considering other variants of calling). For the given setting this is however totally sufficient as we impose the condition for our soundness theorem that the considered execution should be safety compliant and hence should not execute CREATE, CALLCODE and DELEGATECALL instructions.

Definition F.5 (Well-formation of callstacks). An annotated callstack $S \in \mathbb{S}_n$ is well-formed if the following conditions hold:

- $S \neq \epsilon$
- all execution states s_c in S are well-formed
- for all execution states s_c in S it holds that s is consistent with $\{c\}$
- all execution states s_c in $pref(S)$ are consistent with $\{(a, code) \mid last(S).\sigma(a) = (a, n, b, code)\}$ (where $last(\cdot)$ extracts the bottom element from a stack and $pref(\cdot)$ the corresponding prefix)
- all execution states s_c in S, but the top element are call states

LEMMA F.6 (WELL-FORMATION OF REACHABLE CALLSTACKS). *Let (Γ, S) be a configuration that is reachable via a safety-compliant execution. Then S is well-formed.*

LEMMA F.7 (PRESERVATION OF WELL-FORMATION FOR SAFETY COMPLIANT EXECUTIONS). *Let S, S' be callstacks and Γ a transaction environment such that $\Gamma \vDash S \dot{\rightarrow}^* S$. Then if S is well-formed, S' is well-formed.*

The invariant of well-formation that we impose on callstacks is required for proving soundness, as we will need to argue that in lower call-levels already the over-approximations for the re-entering of contract c^* have been introduced.

We give the proof of the soundness theorem.

THEOREM F.8 (SOUNDNESS). *Let C_k be a set of contracts with unique identifiers and addresses and let $c^* \in C_k$ be the contract with identifier id^* and $\mathring{\cdot}$ be a function defined as $\mathring{a} = if\ (id^*, a, \cdot) \in C_k$ then α else a. Additionally, let S' be an annotated callstack such that $|S'| > 0$. Let s be an execution state that is consistent with C_k and s_{c^*} be well-formed. Then the following property holds for all callstacks S:*

$$\begin{aligned}
&\Gamma \vDash s_{c^*} :: S \dot{\rightarrow}^* S' + +S \\
&\implies \forall \Delta_I.\ \alpha_s\,(s, id^*, 0, \mathring{\cdot}) <: \Delta_I \implies \exists \Delta_S. \\
&\qquad \Delta_I \cup \alpha_C\,(C_k, id^*) \vdash \Delta_S \wedge \alpha_S\,(S', C_k, id^*, 0) <: \Delta_S
\end{aligned}$$

PROOF. (sketch) By induction on the number n of small-steps.

- Case $n = 0$. In the case of the empty reduction sequence, we have that $S' = [s_{c^*}]$ and consequently $\alpha_S\,(S', C_k, id^*, 0) = \alpha_s\,(s, id^*, 0, \mathring{\cdot})$ (as $id^* = getID\,(C_k, c^*)$). So the claim follows trivially as each configuration Δ_I that is a coarser abstraction than $\alpha_s\,(s, id^*, 0, \mathring{\cdot})$ is also a coarser abstraction than $\alpha_S\,(S', C_k, id^*, 0)$.
- Case $n > 0$. Let $\Gamma \vDash s_{c^*} :: S \rightarrow^{n-1} S''$ and $\Gamma \vDash S'' \rightarrow S' + +S$. By Lemma F.1, it holds that $S'' = S^* + +S$ for some S^* with $|S^*| > 0$. By the inductive hypothesis we know that for all $\Delta_I :> \alpha_s\,(s, id', 0,)$ there is $\Delta_{S^*} :> \alpha_S\,(S^*, C_k, id^*, 0)^{\cdot}$ such that $\Delta_I \cup \alpha_C\,(C_k, id^*) \vdash \Delta_{S^*}$. Consequently for proving the claim it is sufficient to show that there is some $\Delta_{S'} :> \alpha_S\,(S', C_k, id^*, 0)$ such that $\Delta_{S^*} \cup \alpha_C\,(C_k, id^*) \vdash \Delta_{S'}$ As $|S^*| > 0$, we know that $S^* = s'_{c'} :: S^{**}$ for some $s' \in \mathcal{S}, c' \in C, S^{**} \in \mathbb{S}_n$. The proof is by case analysis on the rule applied in the last reduction step. We show here exemplarily the cases for arithmetic operations as well as the rule for calling.

ADD (non exception case). Then $s' = (\mu, \iota, \sigma)$, $\iota.\mathsf{code}\,[\mu.\mathsf{pc}] = \mathsf{ADD}$ and $S' = (\mu', \iota, \sigma)_{c'} :: S^{**}$. We distinguish the two cases on whether the top stack element $s'_{c'}$ is translated or not:

  - If $s'_{c'}$ is not translated then it holds that $\alpha_S\,(s'_{c'} :: S^{**}, C_k, id^*, cd) = \alpha_S\,(S^{**}, C_k, id^*, cd)$. As the executed contract is not changed, it also holds that $\alpha_S\,(S', C_k, id^*, cd) = \alpha_S\,(S^{**}, C_k, id^*, cd)$. Consequently as $\Delta_{S^*} :> \alpha_S\,(s'_{c'} :: S^{**}, C_k, id^*, cd)$, it also holds that $\Delta_{S^*} :> \alpha_S\,(S', C_k, id^*, cd)$.

* If $s'_{c'}$ is translated, then it holds that $\alpha_S\,(S', C_k, id^*, cd) = \alpha_S\,(S^{**}, C_k, id^*, cd) \cup \alpha_s\,(s', id^*, |S^{**}| + cd, \mathring{:})$. As by F.2, $c' \in C_k$ and s' is consistent with C_k (by Lemma A.6) $\alpha_C\,(C_k, id^*)$ contains $(\!|\mathsf{ADD}|\!)^{C_k, id^*}_{(id', \mu.\mathsf{pc})}$ (where $id' = getID\,(C_k, c')$). The claim then directly follows from the definition of $(\!|\mathsf{ADD}|\!)^{C_k, id^*}_{(id', \cdot)}$ and the soundness of abstract operations E.1. The same argumentation applies to all other arithmetic and logical operations.

CALL (all preconditions satisfied, called account exists). Then $s' = (\mu, \iota, \sigma)$, $\iota.\mathsf{code}\,[\mu.\mathsf{pc}] = \mathsf{CALL}$ and $S' = (\mu', \iota', \sigma')_{\dot{c}} :: S^*$.
Again we distinguish the cases whether the newly pushed callstack element $(\mu', \iota', \sigma')_{\dot{c}}$ is translated or not.

* If $(\mu', \iota', \sigma')_{\dot{c}}$ is not translated then $\alpha_S\,(S^*, C_k, id^*, cd) = \alpha_S\,(S', C_k, id^*, cd)$ and the claim trivially holds.
* If $(\mu', \iota', \sigma')_{\dot{c}}$ is translated then we again make a case distinction on whether $s'_{c'}$ is translated or not
  - If $s'_{c'}$ is translated then it holds that $\alpha_S\,(S', C_k, id^*, cd) = \alpha_S\,(S^{**}, C_k, id^*, cd) \cup \alpha_s\,(s', id^*, |S^{**}| + cd, \mathring{:})$. As by Lemma F.2, $c' \in C_k$ and s' is consistent with C_k (by Lemma A.6), $\alpha_C\,(C_k, id^*)$ contains $(\!|\mathsf{CALL}|\!)^{C_k, id^*}_{(id', \mu.\mathsf{pc})}$ (where $id' = getID\,(C_k, c')$). As $(\mu', \iota', \sigma')_{\dot{c}}$ is translated we know by Lemma F.2 that $\dot{c} \in C_k$ and consequently by the definition of $(\!|\mathsf{CALL}|\!)^{C_k, id^*}_{(id', \cdot)}$ an over-approximation of the state predicates representing $\dot{s}$ (parametrized by $\dot{id} = getID\,(C_k, \dot{c})$) are implied by the Horn clauses in $(\!|\mathsf{CALL}|\!)^{C_k, id^*}_{(id', s'.\mu.\mathsf{pc})}$.
  - If $s'_{c'}$ is not translated then we can conclude that 1) $\dot{c} = c^*$ and 2) there must be execution states ${s^?}_{c^?}, {s^\dagger}_{c^\dagger}$ and callstacks S^+, S^{++} such that $S^* = S^+ + + {s^?}_{c^?} :: {s^\dagger}_{c^\dagger} :: S^{++}$ and $c^? \notin C_k$ and $c^\dagger \in C_k$ and ${s^\dagger}_{c^\dagger}$ is translated. This is as the bottom element of S^* needs to be an execution state annotated with c^* and so at some point in the call chain an unknown contract must have been called in order to break the translation chain (by definition of α_S). We know $\alpha_S\,({s^\dagger}_{c^\dagger} :: S^{++}, C_k, id^*, cd) \subseteq \alpha_S\,(S^*, C_k, id^*, cd)$ (by Lemma F.3), and additionally we know that $s^\dagger = (\mu^\dagger, \iota^\dagger, \sigma^\dagger)$ and $\iota^\dagger.\mathsf{code}\,[\mu^\dagger.\mathsf{pc}] = \mathsf{CALL}$ and all other preconditions for calling are satisfied (by Lemmas F.6 and F.7). As by Lemma F.2, $c^\dagger \in C_k$ and $s^\dagger$ is consistent with C_k (by Lemma A.6), $\alpha_C\,(C_k, id^*)$ contains $(\!|\mathsf{CALL}|\!)^{C_k, id^*}_{(id^\dagger, \mu^\dagger.\mathsf{pc})}$ (where $id^\dagger = getID\,(C_k, c^\dagger)$). Consequently, by the definition of $(\!|\mathsf{CALL}|\!)^{C_k, id^*}_{(id^\dagger, \cdot)}$, Horn clauses for implying an over-approximation of the state predicates representing $\dot{s}$ (which is an execution of c^*) are included in $\alpha_C\,(C_k, id^*)$.

□

Note. Formally, we need to require that all of the contract codes in C_k end with a STOP instruction. This is as in the small-step semantics whenever a program counter runs out of its bounds, a STOP instruction is executed. This problem does not apply to the case of JUMP instructions as in this case the check for valid jump destinations is a precondition for the execution of the instruction. Consequently, the semantics of a smart contract bytecode is preserved when appending a STOP instruction and imposing this restriction does not limit the expressiveness of the analysis technique.