universität wien

DISSERTATION | DOCTORAL THESIS

Titel | Title

Hidden Dangers: Uncovering Security and Privacy Risks Through Large-scale Mobile App Analysis

verfasst von | submitted by

Dipl.-Ing. David Schmidt BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr.techn.)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 786 880
Dissertationsgebiet lt. Studienblatt | Field of study as it appears on the student record sheet:	Informatik
Betreut von | Supervisor:	Ass.-Prof. Mag.art. Karen Azari BSc MSc Ph.D. Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl
Mitbetreut von | Co-Supervisor:	Dipl.-Ing. Dr. Sebastian Schrittwieser

Acknowledgements

I would like to thank everyone who supported me on my journey to the PhD.

First and foremost, I thank my parents, Susanne Schmidt and Rudolf Denk, without whose support this would have never been possible. I am also deeply grateful to my partner, Vanessa Hohenegger, for standing by me throughout this journey. Additionally, I would like to thank all colleagues I met along the way who have become friends, especially Alexander Ponticello, Carlotta Tagliaro, Gabriel Gegenhuber, Florian Holzbauer, and Magdalena Steinböck. Without you, this journey would have been far less enjoyable.

I also thank Katharina Krombholz and Marco Squarcina for their support and guidance during a particularly challenging stage of my PhD. Furthermore, I would like to thank Sebastian Schrittwieser for enabling me to continue my PhD and for being an outstanding advisor with whom it has been a great pleasure to work. I also thank my supervisor, Edgar Weippl, for his guidance and support. In addition, I thank the reviewers, Christopher Krügel and Thijs van Ede, for the time they invested in reading and reviewing this thesis.

Furthermore, I thank the Internet Stiftung for the netidee scholarship, which supported my dissertation.

Abstract

Smartphone apps have become deeply integrated into our daily lives. Approximately 6.9 billion smartphone users worldwide relied on more than 8.9 million apps in 2023. Users employ these apps to manage communication, finances, and health data. Consequently, they must trust that the apps are developed securely and handle their personal data responsibly. However, modern apps rarely operate in isolation. Instead, they interact with the mobile operating system, cloud backends, or Internet of Things (IoT) devices, and each interaction expands the attack surface because the security of the overall system depends on its weakest link. Beyond direct communication partners, the resources used during the development process, e.g., its software supply chain, can be seen as part of the mobile app ecosystem, further extending the attack surface.

In this thesis, we identify weak links in the mobile app ecosystem, develop large-scale analyses to uncover security and privacy issues, and responsibly disclose our findings to improve the security and privacy of the ecosystem. To this end, we analyze four components: First, we study the communication behavior of IoT companion apps. Second, we analyze the iOS local network permission from both a technical and a user perspective. Third, we study secrets embedded in apps, ranging from tokens required for online services to secrets unintentionally included during development. Finally, we analyze the security of iOS dependency management systems to uncover software supply chain threats.

Overall, we uncover various security and privacy issues, for example, Message Queuing Telemetry Transport (MQTT) brokers allowing unauthenticated access, techniques that allow bypassing the iOS local network permission, leaked functional credentials in mobile apps, and supply chain vulnerabilities that affect popular apps from well-known companies.

Kurzfassung

Smartphone Apps sind inzwischen fester Bestandteil des Alltags. Im Jahr 2023 nutzten weltweit rund 6,9 Milliarden Menschen Smartphones und damit insgesamt mehr als 8,9 Millionen Apps. Diese Anwendungen kommen unter anderem für Kommunikation, Finanztransaktionen und den Umgang mit Gesundheitsdaten zum Einsatz. Entsprechend sind Nutzerinnen und Nutzer darauf angewiesen, dass Apps sicher entwickelt werden und personenbezogene Daten verantwortungsvoll verarbeitet werden. Moderne mobile Apps werden jedoch nur selten isoliert betrieben. Stattdessen interagieren sie mit dem mobilen Betriebssystem, mit Clouddiensten oder mit Geräten des Internets der Dinge (IoT). Jede dieser Interaktionen vergrößert die Angriffsfläche, da die Sicherheit des Gesamtsystems stets durch sein schwächstes Glied bestimmt wird. Darüber hinaus können auch die während des Entwicklungsprozesses genutzten Ressourcen, insbesondere die Software-Lieferkette, als Teil des mobilen App Ökosystems gesehen werden und vergrößern somit die mögliche Angriffsfläche.

Ziel dieser Arbeit ist es, Schwachstellen innerhalb des mobilen App-Ökosystems zu identifizieren, großangelegte Analysen zur Aufdeckung von Sicherheits- und Datenschutzproblemen zu entwickeln und die gewonnenen Erkenntnisse zu melden, um Sicherheit und Datenschutz nachhaltig zu verbessern. Zu diesem Zweck untersuchen wir vier Komponenten. Erstens analysieren wir das Kommunikationsverhalten von IoT Apps. Zweitens untersuchen wir die iOS-Berechtigung für lokalen Netzwerk Zugriff sowohl aus technischer als auch aus nutzerzentrierter Perspektive. Drittens extrahieren wir in mobilen Apps eingebettete Geheimnisse, die sowohl für den Zugriff auf Online-Dienste erforderliche Tokens als auch unbeabsichtigt während der Entwicklung integrierte Daten umfassen. Viertens analysieren wir die Sicherheit von iOS-Abhängigkeits-verwaltungssystemen, um Bedrohungen aus der Softwarelieferkette aufzudecken.

Unsere Ergebnisse zeigen eine Vielzahl von Sicherheits- und Datenschutzproblemen. Dazu zählen unter anderem MQTT-Broker ohne Authentifizierung, die von IoT Apps genutzt werden, Methoden zur Umgehung der iOS-Berechtigung für lokalen Netzwerk Zugriff, sowie Zugangsdaten, die in mobilen Apps enthalten sind. Darüber hinaus identifizieren wir Schwachstellen in der iOS App Lieferkette, die weit verbreitete Anwendungen betreffen von bekannten Unternehmen.

List of Publications

[305] *David Schmidt*, Carlotta Tagliaro, Kevin Borgolte, Martina Lindorfer. IoTFlow: Inferring IoT Device Behavior at Scale through Static Mobile Companion App Analysis. In ACM Conference on Computer and Communications Security (CCS), 2023.

[297] *David Schmidt*, Alexander Ponticello, Magdalena Steinböck, Katharina Krombholz, Martina Lindorfer. Analyzing the iOS Local Network Permission from a Technical and User Perspective. IEEE Symposium on Security and Privacy (S&P), 2025.

[299] *David Schmidt*, Sebastian Schrittwieser, Edgar Weippl. Leaky Apps: Large-scale Analysis of Secrets Distributed in Android and iOS Apps. ACM Conference on Computer and Communications Security (CCS), 2025.

[300] *David Schmidt*, Gabriel K. Gegenhuber, Edgar Weippl, Sebastian Schrittwieser. Supply Chain Insecurity: Exposing Vulnerabilities in iOS Dependency Management Systems. *Under Submission.*

---

[298] *David Schmidt*, Sebastian Schrittwieser. Replay Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks, 2025. **Not part of the thesis.**

[194] Caroline König, *David Schmidt*, Philip König, Patrick Felbauer, Sebastian Schrittwieser. SaMBA: Increasing Mixed Boolean-Arithmetic Complexity Through Equality Saturation. ACM ASIA Conference on Computer and Communications Security (ASIACCS), 2026. **Not part of the thesis.**

[129] Viktor E. Garske, Swantje Lange, Gabriel K. Gegenhuber, *David Schmidt*, Andreas Noack, Jiska Classen. Blue Bubbles, Red Flags: Investigating Privacy Leakage in Apple iMessage. ACM Conference on Computer and Communications Security (CCS), 2026. **Not part of the thesis.**

[131] Gabriel K. Gegenhuber, Moritz Grefner, Maximilian Günther, Matthäus Wininger, *David Schmidt*, Aljosha Judmayer. Send and Pretend: Exploiting Transcript Consistency Issues in End-to-End Encrypted Group Chats. USENIX Security, 2026. **Not part of the thesis.**

[169] Florian Holzbauer, *David Schmidt*, Gabriel K. Gegenhuber, Sebastian Schrittwieser, Johanna Ullrich. Context Matters: Repository-Aware Security Analysis of the Agent Skill Ecosystem. Workshop on Agent Skills, 2026. **Not part of the thesis.**

Contents

1 Introduction

In 2007, Steve Jobs introduced the first iPhone with the words *"An iPod, a phone, and an Internet communicator"* [54]. Since then, smartphones and the mobile apps running on them have evolved to support a substantially broader range of tasks and have become deeply integrated into our daily lives.

Along with this integration, smartphone adoption grew rapidly. In 2023, approximately 6.9 billion people worldwide relied on smartphones and more than 8.9 million mobile apps to support everyday activities [157]. The messaging service WhatsApp alone registered more than 3.5 billion users in 2025 [130], despite being banned in several countries, including China. With the increasing integration into our personal and professional environment, apps also manage our communications, finances, social interactions, and personal health data. Consequently, users must trust the apps to handle sensitive information responsibly and securely.

However, modern mobile apps rarely operate in isolation, as their functionality often depends on cloud backends, third-party services, communication with nearby devices, and resources provided by the mobile operating system. Beyond direct communication partners, the resources developers use during development can also be considered part of the mobile app ecosystem. For example, this includes dependency management systems that automatically pull libraries and their transitive dependencies. Each of these components expands the attack surface of the mobile ecosystem. Ensuring security, therefore, requires securing every element on which the app depends. A single vulnerable cloud endpoint, insecure protocol, or embedded credential can compromise an otherwise robust system. This illustrates the security principle of the weakest link [306].

Prior research uncovered weaknesses in various parts of the mobile app ecosystem. For example, Reardon et al. [283] analyzed Android permissions and demonstrated how Android apps could bypass location permissions by leveraging information accessible via local network communication, again highlighting the weakest link principle since the missing security of one component undermined the security of the otherwise secure location permission.

Security and privacy issues not only arise during app execution but also during app development. A case involved Snapchat, which unintentionally embedded parts of its source code in the iOS app bundle [82]. Thus, they revealed intellectual property and potentially enabled repackaging attacks. Other apps have leaked cloud credentials [156], allowing attackers to access user information. Additionally, the software supply chain is a part of the app's security. Supply chain risks have emerged and been demonstrated across ecosystems, such as npm, where researchers have uncovered malicious dependencies infecting numerous other projects through transitive dependencies [368, 360, 250].

One way to improve the security and privacy of the ecosystem is through large-scale analyses. These can help to detect issues early, before attackers exploit them, and,

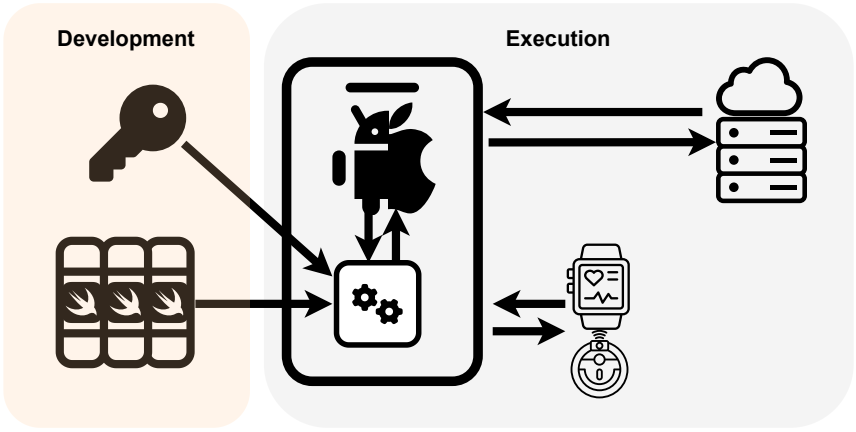


Figure 1.1: The mobile app ecosystem comprises components that apps interact with throughout their lifecycle. During development, this includes, for example, dependency management systems and tokens that developers use as part of the development process. During execution, the ecosystem also includes the mobile operating system, nowadays Android and iOS, as well as devices and remote services with which the app communicates.

by reporting them, help improve the security of the ecosystem. Previous work has already utilized large-scale analysis of mobile apps to identify security and privacy issues. For example, Zuo et al. [370] developed an analysis to detect cloud data leaks through mobile apps. Zhao et al. [364] investigated hidden behaviors in mobile apps, including access keys and master passwords that unlock restricted functionalities. Reardon et al. [283] applied large-scale analysis to uncover methods and apps that bypassed Android permissions.

In this thesis, we fill gaps in existing large-scale analysis by extending it in four directions. First, we identify security and privacy issues of Internet of Things (IoT) devices through their companion apps. Second, we analyze the iOS local network permission. Third, we analyze secrets distributed in mobile apps. Fourth, we uncover supply chain vulnerabilities in iOS apps. To achieve these goals, we leverage information that apps expose about their development process and the embedded information about their communication partners. This information includes credentials, endpoints, protocols, libraries, and system Application Programming Interfaces (APIs).

In summary, the overall goal of this thesis is to identify weak links in the mobile app ecosystem at scale, report them to the responsible parties, and ultimately improve the security and privacy of the ecosystem.

1.1 Research Questions

Due to the complexity and variety of components that constitute the mobile app ecosystem, studying all aspects simultaneously is infeasible. Therefore, this thesis focuses on four concrete parts, as shown in Figure 1.1, each addressing a distinct research question. However, all parts share the common goal: to uncover weak links

in the mobile app ecosystem at scale, report them responsibly, and contribute to improving the security and privacy of the ecosystem.

To achieve this goal, we extract and analyze information about app interactions during execution and during their development. In the first two projects, we focus on components that mobile apps interact with during execution, namely the local network permission and the IoT devices with which apps communicate. In the third project, we transition from execution information to development information by studying secrets leaked in mobile apps, ranging from tokens required for app functionality to unintentionally included development artifacts. In the fourth project, we further analyze development information by extracting dependency information from iOS apps, providing insights into their software supply chains.

1.1.1 IoT Companion Apps

IoT devices often provide mobile apps, so-called companion apps, that allow users to command and control them. For example, smart vacuum cleaners typically do not provide a full physical user interface. Instead, users configure and schedule cleaning tasks through companion apps. To provide these functionalities, companion apps must communicate with the devices, either directly, for example via Bluetooth or the local network, or indirectly through cloud backends. As a result, companion apps embed information about both the devices and the cloud infrastructure.

The communication information contained in mobile apps, therefore, offers an opportunity to study the security and privacy properties of IoT devices at scale, since companion apps exhibit substantially less diversity than the devices themselves. Moreover, acquiring and deploying thousands of devices in laboratory environments for dynamic analysis is practically infeasible.

To avoid purchasing and operating large numbers of IoT devices, we develop a static analysis to extract communication information from companion apps, and answer *RQ1: What security and privacy insights can be derived from the communication of companion apps through static analysis?*

1.1.2 Local Network Permission

In the past, researchers have discovered apps that bypassed the location permission by obtaining the router MAC address via local network communication [283]. Further, researchers identified a malicious app that attempted to reconfigure the router Domain Name System (DNS) server [190]. To mitigate security and privacy threats originating from apps with access to the local network, Apple introduced local network permissions in iOS 14.

To provide effective protection, the permission must be secure from both technical and user perspectives, as users ultimately decide whether to grant or deny it. However, due to the intrinsic technicality of the local network, it is unclear whether users can make an informed decision. On the technical side, this permission differs from others. Typically, permissions guard access to specific APIs. In contrast, the local network permission does not follow this model, as apps can use the same APIs for web requests without requiring the permission. Thus, iOS monitors network request targets and enforces the permission once it detects local network access.

To gain insights into the permission, we first study its implementation and then analyze apps to detect local network accesses. Furthermore, we extract the concepts that constitute the permission prompts, which developers can customize, and then conduct a user study to gain insights into the user understanding. Ultimately, we answer *RQ2: Is the iOS local network permission effective from a technical and user perspective?*

1.1.3 Secrets in Mobile Apps

Apps can include tokens to provide functionality and to access remote services. In other cases, developers accidentally embed secrets and tokens into mobile apps. For example, Snapchat leaked parts of the app's source code via the mobile app. These issues matter because apps are distributed to end-user devices, allowing attackers to reverse-engineer them and extract embedded secrets.

This property also enables us as researchers to download apps, extract secrets, and report them responsibly. To this end, we develop a static analysis capable of handling Android and iOS apps. To identify unintentionally included secrets that are not required for app functionality, we first study the files contained in app bundles and then apply a regular expression-based secret detection approach to answer *RQ3: What secrets do developers distribute in their app bundles?*

1.1.4 iOS Dependency Management Systems

Libraries allow software developers to reuse existing functionality and avoid reimplementing it. Dependency management systems further simplify library integration and often provide public repositories for searching and downloading existing libraries. In practice, including a single library often introduces dozens of additional libraries through transitive dependencies, each of which increases the potential attack surface.

Attacks such as SolarWinds [352], the event-stream compromise [125], and the XZ Utils backdoor [3] increased attention toward software supply chain security. However, the software supply chain of iOS apps remains underexplored. To close this gap, we investigate potential attack vectors in *Carthage* [75], *CocoaPods* [85], and *SwiftPM* [321], three dependency management systems commonly used for iOS app development. We also conduct a large-scale analysis of iOS apps to identify vulnerable apps and report them responsibly. Finally, we analyze five additional dependency management systems to compare the impact of different authentication and distribution strategies on security. Overall, we answer *RQ4: How do dependency management systems in the iOS ecosystem expose apps to supply chain attacks?*

1.2 Methodology and Contribution

In the following, we briefly introduce the methodologies we used to answer each research question and summarize the contributions of each work. While the general methodology, as outlined in Figure 1.2, consistently involved developing an analysis that scales to thousands of apps, applying it to an app dataset, conducting additional evaluations and measurements, and responsibly disclosing our findings, the concrete methodologies differed across projects.

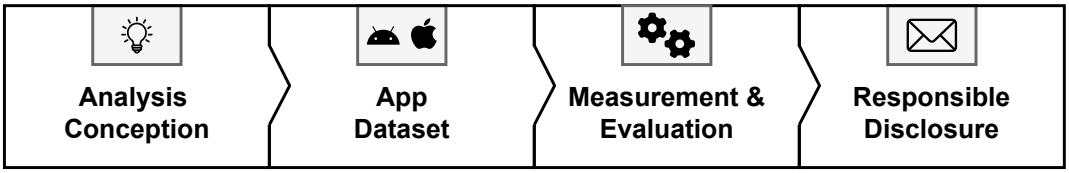


Figure 1.2: High-level thesis methodology. Each work follows these steps, yet the concrete methodologies vary across projects to best fit each goal.

1.2.1 IoT Companion Apps

In Chapter 2, we present a static analysis that combines Value Set Analysis (VSA) with general data-flow analysis to extract information about companion app communication. We used VSA to extract the endpoints that companion apps contact, and we used data-flow analysis to trace data flows from devices to remote endpoints and vice versa.

To answer *RQ1: What security and privacy insights can be derived from the communication of companion apps through static analysis?*, we applied our analysis to 9,889 companion apps, which were manually verified by three related works [240, 235, 185]. Additionally, we compared the communication behavior of these companion apps with that of 947 popular general-purpose apps to highlight the differences between companion apps and other types of apps.

Using the extracted data, we evaluated how companion apps communicate with IoT devices, the protocols employed, the geolocation of endpoints, and the endpoint categories. Moreover, we searched for abandoned domains, analyzed data flows originating from IoT devices and sensitive Android APIs, and analyzed the encryption algorithms used by the apps.

Our analysis uncovered multiple security and privacy issues, including MQTT brokers allowing unauthenticated access, abandoned domains, broken encryption algorithms, and the sharing of Personally Identifiable Information (PII). We responsibly disclosed abandoned domains to app developers via the email addresses listed in the Google Play Store, as attackers could potentially exploit abandoned domains to take control of the corresponding devices by registering them.

In addition, we compared our static analysis results with manual dynamic analysis by recording network traffic from 13 companion apps connected to their associated devices while interacting with them. Although the raw numbers of contacted endpoints and paths differed between static and dynamic analysis, the results showed that our approach provides valuable insights.

1.2.2 Local Network Permission

In Chapter 3, we analyze the iOS local network permission from both technical and user perspectives by combining four studies. We developed an iOS test app that systematically attempts to access local network resources via different networking APIs and endpoints to evaluate when iOS enforces the permission. We executed each test both with and without granting the permission, which allowed us to determine whether iOS enforced it correctly.

To study the prevalence of local network access in apps, we conducted a large-scale dynamic analysis of 10,862 apps available on both iOS and Android. We used dynamic analysis because local network access does not rely on a fixed set of APIs, and this approach aligns with how iOS evaluates local network access at runtime. Thus, the analysis is comparable across both platforms. To identify matching Android and iOS apps, we downloaded a set of popular and random apps from the Apple App Store and the Google Play Store and applied the matching methodology of Steinböck et al. [320], which identifies corresponding apps across two datasets. In addition, we included Android apps that the Android migration API identifies as matching.

We executed the apps in a controlled test network while recording all outgoing network traffic directly on the phones. For each app, we first executed the app without interaction for 30 seconds, then performed 25 automated interactions following a depth-first search strategy over the user interface. This setup allowed us to distinguish between local network accesses at startup and accesses triggered by user interaction. To ensure we can also observe local network access on iOS, we granted all permissions to each app.

We statically extracted local network permission rationales from apps' `Info.plist` files and localized resource files to study the concepts that constitute these rationales. Afterwards, three researchers coded the extracted rationales to identify which concepts users need to make an informed decision about the permission. To capture the user perspective, we also conducted an online survey of 150 iOS users, recruited via Prolific. In the survey, we asked participants about local network security and privacy threats, the concepts required to make an informed decision that we previously identified in the rationales, and misconceptions that we observed in online discussions.

To answer *RQ2: Is the iOS local network permission effective from a technical and user perspective?*, we showed with our systematic test app that two iOS components can interact with the local network without requiring the permission, and that the protected local network address space is insufficient.

We also uncovered that developers frequently employed terminology in permission rationales that is potentially misleading or reflects misconceptions, such as claiming that the permission is necessary for Internet access. Finally, we showed that nearly every participant was aware of at least one threat associated with local network access. However, misconceptions about the permission were even more widespread. We responsibly disclosed our findings to Apple, covering both technical and user perspectives.

1.2.3 Secrets in Mobile Apps

In Chapter 4, we present an analysis that studies secrets in mobile apps. To this end, we developed a static analysis methodology that enables cross-platform analysis of Android and iOS apps. We first analyzed the metadata of files included in the app bundle. We then applied a regular expression-based secret detection approach to identify leaked credentials and validate them against their remote APIs. Finally, we automatically disclosed our findings to affected app developers.

To enable uniform processing across platforms, we extracted the contents of Android APKs and iOS IPAs, both of which are essentially archives. For each file, we stored metadata, including MIME type and file suffix, for further evaluation. Before searching for hardcoded credentials, we extracted a textual representation using `strings` [127] and the DEX decompiler `JADX` [137]. To identify credentials, we adapted TruffleHog [335], which employs a regular expression-based detection approach and was originally designed to find secrets in Git repositories.

We reused the dataset previously collected for studying local network access to answer *RQ3: What secrets do developers distribute in their app bundles?* In addition, we re-downloaded the dataset in 2024, which enabled us to examine how the situation changed over the course of a year. By analyzing file metadata, we discovered a variety of potentially unintended bundled files. For example, we identified scripts that represent development artifacts, such as build scripts or test code. In addition, we found Markdown files containing development documentation, including sensitive content such as API keys, developer names, email addresses, and internal Uniform Resource Locators (URLs), which attackers could abuse, for example, for targeted social engineering attacks.

Overall, we detected 416 functional credentials from 65 services, including 13 Git credentials that granted access to 2,440 private repositories. Our comparison between Android and iOS apps showed that apps from both platforms contained secrets. However, the results revealed a higher prevalence of issues in iOS apps. At the same time, it also highlighted the importance of studying apps from both platforms, as findings were often exclusive to one platform. One possible explanation for this is that Android and iOS versions are often developed by different teams.

We reported the distribution of dependency management files, exposed source code, and functional tokens to app developers via their Play Store email addresses. As the iOS App Store did not provide email addresses for each app at the time of the study, and because our dataset contains a matching Android app for every iOS app, we also used the corresponding Play Store email addresses to contact iOS app developers. In total, we received responses, which ranged from positive reactions, such as developers fixing the issue or planning to do so, to negative responses, such as stating that a fix would be too expensive.

1.2.4 iOS Dependency Management Systems

In Chapter 5, we present a study on supply chain vulnerabilities in iOS apps and answer *RQ4: How do dependency management systems in the iOS ecosystem expose apps to supply chain attacks?* We first analyzed Carthage [75], CocoaPods [85], and SwiftPM [321], three dependency management systems used for iOS development, to identify potential attack vectors.

Based on this analysis, we identified three concrete threats: (1) dependency hijacking via abandoned URLs, (2) dependency hijacking via abandoned GitHub namespaces, and (3) dependency confusion attacks. Dependency hijacking attacks enable attackers to take over existing libraries and inject malicious code. Dependency confusion attacks enable attackers to gain Remote Code Execution (RCE) on developer machines and build servers if they publish an internal dependency name in a public

repository and the dependency management system prioritizes the public dependency over the internal one.

To quantify the number of affected apps, we developed a static analysis that extracts library metadata from iOS apps. We retrieved this metadata from the corresponding `Info.plist` files, which include bundle identifiers and version numbers, revealing the library name, version, and the dependency management system used. We then compared the extracted dependency identifiers against the public CocoaPods specification repository to identify libraries that were not registered publicly and therefore were susceptible to dependency confusion attacks.

To study the prevalence of hijacking attacks, we analyzed the ownership and hosting infrastructure of publicly available CocoaPods libraries. We extracted library ownership information, including maintainer email addresses, via the CocoaPods API and identified abandoned email domains that can enable silent account hijacking. In addition, we extracted all dependency hosting locations referenced by CocoaPod libraries and analyzed whether the corresponding domains or GitHub namespaces were abandoned and were available for registration.

Finally, we extended our analysis to additional dependency management systems, including Cargo, Go modules, npm, pip, and Maven. We compared authentication models, dependency resolution mechanisms, and integrity guarantees to evaluate how different system designs affect resilience against supply-chain attacks.

By applying our analysis methodology to 9,212 iOS apps, an updated version of the app dataset previously collected to study the local network permission, we identified 2,084 apps (22.62%) that included dependencies not registered in the public CocoaPods repository and thus remained potentially susceptible to dependency confusion attacks. Using apps with responsible disclosure programs, we successfully demonstrated and reported dependency confusion attacks against libraries from nine companies.

In addition, we identified 213 CocoaPod libraries whose owners used email addresses associated with abandoned domains, which affected 97 apps in our dataset. Furthermore, we found that 80 apps relied on dependencies hosted on abandoned GitHub namespaces, including popular apps with more than 100 million installations.

Finally, our comparison with Cargo, Go modules, npm, pip, and Maven showed that these risks are not unique to dependency management systems used for iOS apps. However, the comparison also highlighted that enforcing two-factor authentication (2FA), explicitly specifying internal repositories for each dependency, and requiring the dependency management system to host its own dependency repository could help mitigate the identified issues.

2 IoTFlow: Inferring IoT Device Behavior at Scale through Static Mobile Companion App Analysis

Abstract

The number of "smart" devices, that is, devices making up the IoT, is steadily growing. They suffer from vulnerabilities just as other software and hardware. Automated analysis techniques can detect and address weaknesses before attackers can misuse them. Applying existing techniques or developing new approaches that are sufficiently general is challenging though. Contrary to other platforms, the IoT ecosystem features various software and hardware architectures.

We introduce IoTFlow, a new static analysis approach for IoT devices that leverages their mobile companion apps to address the diversity and scalability challenges. IoTFlow combines VSA with more general data-flow analysis to automatically reconstruct and derive how companion apps communicate with IoT devices and remote cloud-based backends, what data they receive or send, and with whom they share it. To foster future work and reproducibility, our IoTFlow implementation is open source.

We analyze 9,889 manually verified companion apps with IoTFlow to understand and characterize the current state of security and privacy in the IoT ecosystem, which also demonstrates the utility of IoTFlow. We compare how these IoT apps differ from 947 popular general-purpose apps in their local network communication, the protocols they use, and who they communicate with. Moreover, we investigate how the results of IoTFlow compare to dynamic analysis, with manual and automated interaction, of 13 IoT devices when paired and used with their companion apps. Overall, utilizing IoTFlow, we discover various IoT security and privacy issues, such as abandoned domains, hard-coded credentials, expired certificates, and sensitive personal information being shared.

Publication The work presented in this Chapter resulted from a collaboration with Carlotta Tagliaro, Kevin Borgolte, and Martina Lindorfer. It has been published at ACM Conference on Computer and Communications Security (CCS), 2023 [305]. I developed the static analysis IoTFlow and evaluated the results. Carlotta Tagliaro performed the comparison with dynamic analysis, carried out the permission analysis, and together we extended IoTFlow to IoT related protocols. Kevin Borgolte and Martina Lindorfer provided feedback and revised the paper.

2.1 Introduction

The number of Internet of Things (IoT) devices, that is, smart devices, is rising rapidly: Forecasts expect the number of IoT devices to grow to 25.4 billion in 2030 [168]. These devices collect data about their users and environment to make smart decisions. For example, to call for help in an emergency, a smartwatch may collect health indicators. This means that users need to trust them to handle their data with care. Unfortunately, smart devices have gained notoriety for their security and privacy issues, leading to the catchphrase *"the S in IoT stands for security."* Notably, employees of Ring had unauthorized access to users' security camera footages uploaded to their cloud backend [204]. Similarly, the European Union (EU) recalled kids' smartwatches because they exposed sensitive information and could be easily compromised by attackers [69].

Prior work extensively analyzed open and closed source desktop and mobile applications (apps) for security and privacy issues, but analyzing smart devices remains an open challenge. Related work in this domain mainly focused on firmware vulnerabilities [93, 77] or on analyzing a handful of selected devices [353, 167, 79, 286, 340, 212]. This does, however, not scale to the wide variety of smart devices with diverse software and hardware architectures. Intuitively, buying thousands of devices to analyze them in a lab setting is financially and practically infeasible.

Therefore, to enable the large-scale discovery and analysis of security and privacy issues in the IoT ecosystem, we propose IoTFlow, a novel static analysis approach for IoT devices via their mobile companion apps. These apps play an important role in controlling IoT devices directly and can serve as intermediaries to their cloud backends. Practically all IoT devices have such apps available for Android and iOS [285, 77, 235, 240]. They allow users to setup and control their devices locally, via the local network or Bluetooth, or remotely, via the Internet. For some devices, their apps are the only gateway to the Internet. Overall, the apps store and process information collected by the IoT devices and about the remote infrastructure. Given the nature of data that the devices collect and use, it may also be highly sensitive. Further, attackers could misuse apps with hard-coded information (e.g., endpoints, credentials) to eavesdrop on others' private information, or distribute malicious content via misconfigured IoT backends or abandoned domains. Using a misconfigured backend, they could exploit vulnerabilities to create a new botnet of hundreds of thousands of devices, even if the devices are not directly reachable on the Internet.

The basic idea of evaluating the security and privacy of IoT devices indirectly by studying their companion apps has been explored by prior work. For example, Wang et al. [345] leveraged it to identify rebranded devices by searching for similar apps. They find vulnerabilities in other devices because of "private labeling" and component re-use. Chen et al. [77] and Redini et al. [285] used companion apps to inform fuzzing IoT devices, while Zuo et al. [371], Sivakumaran et al. [315, 314], and Zhao et al. [363] leveraged companion apps to identify Bluetooth Low Energy (BLE) issues. Wang et al. [344] statically analyzed Samsung SmartThings apps, which are part of the SmartThings smart hub IoT ecosystem.

Existing approaches focus on re-identifying already known issues shared among multiple devices (previously discovered through traditional techniques), still require

physical devices (fuzzing), focus on a subset of companion apps (BLE), or analyze conceptually simple apps that are less widespread than general companion apps [241] (e.g., Samsung SmartThings apps, which are event flow graphs, rather than full apps; similar to "If This Then That" [177]).

In this paper, we introduce a new static analysis approach, IoTFlow, that substantially advances this basic idea. Our new approach enables us to gain new fundamental knowledge about the companion apps and corresponding smart devices at scale without actually requiring the physical device. We focus on addressing two crucial limitations of state-of-the-art techniques: First, we discover new issues automatically instead of re-identifying existing issues, which would require a priori knowledge that they exist. Second, we investigate individual devices instead of assuming that groups of devices share or re-use components. Specifically, our approach enables us to infer and gain new insights into the security and privacy of companion apps and their corresponding smart devices by reconstructing information about the used network protocols, endpoints, and the data they receive. With our approach, we can answer the following important but open questions concerning security and privacy in the IoT ecosystem:

RQ1: *How do companion apps and devices communicate?*

RQ2: *Who are companion apps communicating with?*

RQ3: *Which data are companion apps sharing (and how)?*

Specifically, our approach (1) identifies communication trigger points, (2) uses Value Set Analysis (VSA) to reconstruct network-related information on where data is coming from or transferred to, such as the URLs that are being contacted, (3) utilizes Data-flow Analysis (DFA) to determine what data is being accessed, shared, and with whom, and (4) assesses the corresponding impact.

We evaluate our approach on 9,889 manually verified companion apps [240, 235, 185] to show that we can analyze IoT devices accurately and at scale. Additionally, we study the differences in network behavior between the companion apps and 947 popular general-purpose apps that we collected. Finally, we verify the accuracy of IoTFlow and compare it with dynamic analysis, for which we interacted with 13 IoT devices via their companion apps.

In this paper, we make the following contributions:

- We introduce IoTFlow, a new static program analysis approach utilizing Value Set Analysis (VSA) and Data-flow Analysis (DFA) to analyze the behavior of IoT devices based on their companion apps' interactions with them and their remote backend.
- We show that IoTFlow can accurately infer the network behavior of companion apps at scale by analyzing 9,889 IoT apps.
- We analyze how and with whom companion apps communicate, what data they share locally with devices and remotely, and we highlight their differences to general apps.

- Using IoTFlow, we automatically discover rampant security and privacy issues in the IoT ecosystem, such as abandoned control domains, hard-coded credentials, expired certificates, or shared PII.

Artifacts. To foster reproducibility and future research, we make our open source implementation and analysis artifacts available at https://github.com/SecPriv/iotflow.

2.2 Motivation

Following, we motivate IoTFlow with the need for at-scale IoT device behavior analysis, the interdependence of companion apps and IoT infrastructure, and the unique features of companion apps compared to general-purpose apps.

Large-scale IoT Device Behavior Analysis The plethora of security and privacy issues that supposedly plague smart devices are a well-hypothesized problem in the security community and often anecdotally confirmed when yet another real-world issue is found and the press is reporting on it. Unfortunately, we currently lack techniques to discover such issues and also other vulnerabilities in smart devices automatically and at scale. State-of-the-art approaches focus on analyzing the devices' firmware [93], requiring tedious and substantial manual effort to tailor it to each individual device, possibly even each hardware revision of a device. It also suffers from the many challenges of analyzing firmware, such as having to deduce and infer what sensors and actuators exist, model them, and understand how the firmware is communicating with it. Even if it would be feasible to scale such approaches to the many devices, it is also challenging to automatically gather thousands of firmware images, as devices use different processes to retrieve and update their firmware. At the same time, more and more IoT devices are being manufactured and used. Thus, it remains an open problem how to analyze the increasing number of diverse devices.

For large open source projects, the average lifetime of vulnerabilities is multiple years [206, 5]. Considering the profit-driven nature of the IoT ecosystem, it appears likely that security is indeed an afterthought in the IoT ecosystem and vulnerabilities might remain unpatched similarly long or even longer. Automated large-scale analysis allows us to promptly identify vulnerabilities and mitigate them. Moreover, even when automated analysis cannot replace in-depth analysis, it still helps developers to identify issues and address them. Being able to accurately analyze how IoT devices truly behave also informs privacy policy and behavior of (privacy-conscious) consumers. Practical large-scale automated analysis provides the much-needed foundation and knowledge to better understand IoT devices and improve their security and privacy.

IoT Control Infrastructure The fundamental idea of smart devices is that they coordinate and cooperate with other devices, that is, they do not work in isolation. Typically, the devices communicate with companion apps, smart hubs, or remote cloud-based backends (see Figure 2.1), the latter of which may distributed over different regions world-wide [293]. Users interact with the devices almost exclusively

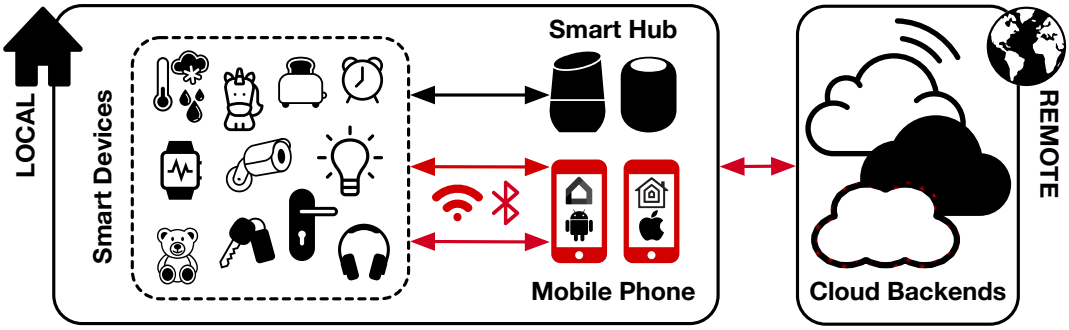


Figure 2.1: Overview of the IoT ecosystem and its command and control scenarios, including apps as intermediaries.

via their companion apps. If a device supports Wi-Fi, then the app may communicate with the device over the local network or the Internet. If a device does not support Wi-Fi but only uses Bluetooth, then all device-to-cloud communication needs to pass through the app or a hub. Moreover, due to missing user interfaces, updating a device's firmware frequently happens via the app [345]. That is, the apps play a central role during the *setup, operation,* and *update* of the devices. In fact, many devices cannot be set up without using a device that can run the app. Thus, apps must contain some information about the devices and their behavior, and they provide a unique analysis opportunity.

General-purpose Apps vs. Companion Apps Compared to general-purpose apps, companion apps face different challenges and introduce new threats. Generally, mobile operating systems restrict access to sensitive data and sensors (e.g., through Android or iOS permissions). However, this does not apply to data collected through smart devices. Users also lack visibility and control over the data the devices collect and share. It is crucial to investigate the threat of collusion between device and app, especially because it circumvents existing defenses and allows to build more accurate user profiles by combining PII and data collected by both [287].

Advertisements (ads) and trackers to collect user data for behavioral targeting appear widely in general-purpose apps [280, 339, 287]. These services are attractive for developers to monetize their apps [161]. For companion apps, one might assume that the business model centers around selling the devices. However, related studies showed that these apps and even devices themselves include ads and tracking [340, 212, 286]. In hindsight, considering the IoT environment and collusion potential, this makes sense: It is additional income. For example, companion apps can interact with the local network to discover and manage devices (a permission often required to set up the device), which is data general-purpose apps have difficulty to collect, and which is also useful for advertisement or tracking [197]. Prior work on network behavior and PII leakage of apps mainly considers traffic sent to remote servers. For IoT devices that use local communication, via Bluetooth or Wi-Fi, app-to-device or device-to-app communication has additional significance [316]. A smart device only using Bluetooth can collude with a companion app to "clean" sensitive data: receive it, encode it in some way, and send it back to the app, which sends it to the tracker.

Existing ways to identify and block such behavior in general-purpose apps cannot address the challenges of the IoT environment, like collusion.

2.3 IoTFlow

We introduce IoTFlow, a new static analysis approach for companion apps. We aim to better understand the behavior of IoT devices without requiring the physical device.

IoTFlow itself has two main phases (see Figure 2.2): Value Set Analysis (VSA) and Data-flow Analysis (DFA). With VSA, we identify *trigger points*, that is, sources and sinks of interesting (network) activities. This appears trivial at first, but it is important to realize that (1) we expect a substantial amount of communication, as smart devices are meant to communicate and coordinate extensively, and (2) we need to be able to determine the communication endpoint. For example, a user might expect and accept that the companion app shares their location to turn on their heating when they are on their way home. But, most users would likely object if it is sent to an advertisement company. Enumerating all potential sources and sinks will lead to inaccurate results and render the analysis impractical. Instead, we need to distinguish where apps send data, to the device or a remote service, to which services, and utilizing which network protocols. We accurately reconstruct this information leveraging VSA (Section 2.3.1) and use it to identify precise sources and sinks for our DFA (Section 2.3.2).

For reconstructed endpoints, which may be third-party services, we then (1) categorize them based on their purpose, (2) analyze their geographic locations, and (3) test for abandoned domains. This allow us to evaluate if communication would be expected and assess their security and privacy impact. For example, a privacy-conscious user within the EU may not expect that their device sends data to a country not bound to the GDPR. Similarly, abandoned domains can lead to devices being taken over by attackers [80, 71, 267].

With DFA, we can then precisely assess which data companion apps share, with whom they communicate, and how. Specifically, we analyze the data-flow for data from the identified and categorized trigger points as well as from sensitive data sources (e.g., GPS location) to relevant sinks.

Motivating Example Considering the examples in Listing 1 and Listing 2, we (1) need to reconstruct the destination of the MQTT broker (Listing 2, line 15), and (2) trace the data flow from the Bluetooth source (Listing 1, line 3) to where the message is published (Listing 2, line 18). An additional challenge is that the data is passed from Listing 1 line 6 to Listing 2 line 7 via Inter Component Communication (ICC). Traditional approaches would miss this example. However, we can reconstruct the keys of the ICC during VSA and then bridge the connection via the reconstructed keys, enabling us to perform more precise DFA across the ICC boundary.

2.3.1 Value Set Analysis

Value Set Analysis (VSA) is a program analysis technique to reconstruct values at specific program points. We utilize it to gain insights about the communication of IoT

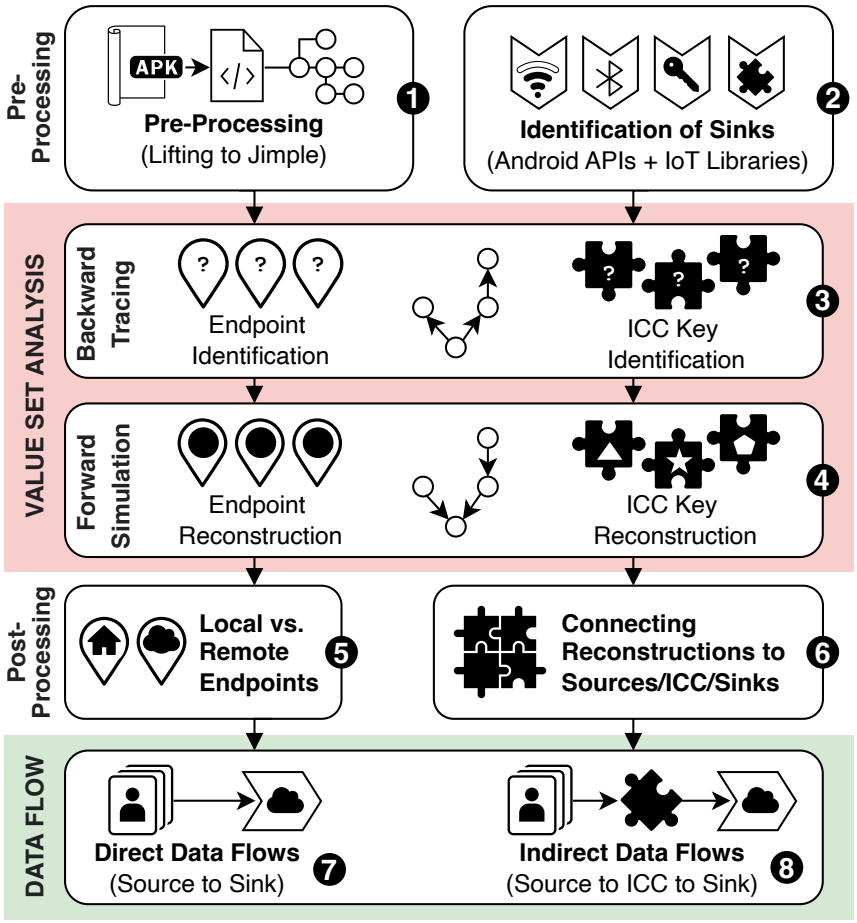


Figure 2.2: Overview of IoTFlow. We use VSA to reconstruct endpoints, cryptographic data, and ICC keys for the flow analysis. We use flow analysis to find data leaks, and connect request/response data with endpoints. With the ICC information of the VSA, we support data flows involving ICC.

apps and to accurately handle ICC for our DFA. VSA has been used by related work before [370, 371], however, with the focus on reconstructing API keys or Universally Unique Identifiers (UUIDs) of BLE to identify vulnerable implementations of the BLE pairing process. That is, related work reconstructed primarily strings using manually derived rules, while IoTFlow supports arbitrary objects (as is required to precisely reconstruct endpoints, like in Listing 2).

Pre-Processing ❶ We implemented our IoTFlow prototype in Java and target Android. We use Soot [338] to parse Dalvik byte code from Android apps. It translates the byte code into the Jimple Intermediate Representation (IR), which simplifies our analysis (e.g., by splitting nested instructions). Notably, both Kotlin and Java Android apps are compiled into Dalvik code, and, in turn, IoTFlow can readily analyze both types of apps. In preparation for the forward computation step, we also translate the Dalvik byte code into Java byte code with `dex2jar` [260] because Java cannot load classes directly from the Dalvik byte code.

```
1  String BLE_DATA = "device";
2  @Override
3  void onCharacteristicRead(BluetoothGattCharacteristic bgc, /*...*/) {
4      Intent intent = new Intent(DeviceActivity.class)
5      byte[] value = bgc.getValue();
6      intent.putExtra(BLE_DATA, parseData(value));
7      this.startActivity(intent);
8  }
```

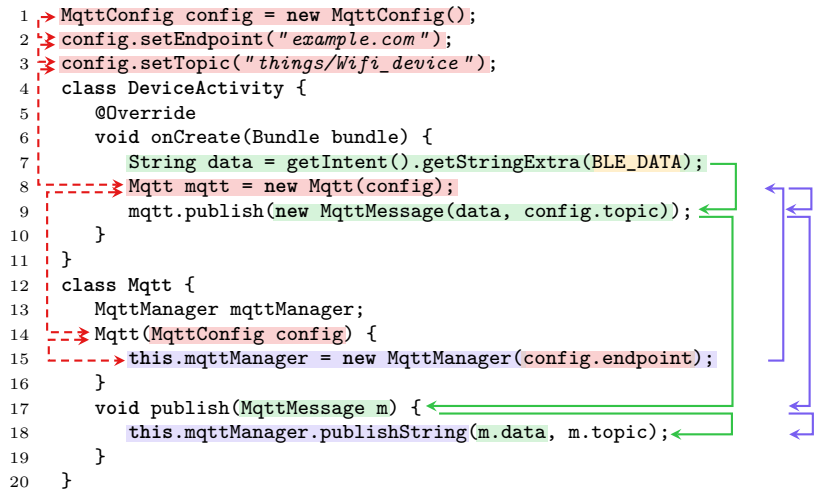
Listing 1: Simplified example code that reads device data via BLE and sends it via an Intent (ICC). Arrows on the left show VSA, and arrows on the right show DFA. We reconstruct the ICC key "device," marked yellow, via VSA ↔ (line 1). The ⟶ arrows show the data flow from source to the ICC sink, via purple statements. The flow continues green in Listing 2.

Identification of Sinks ❷ IoTFlow starts at interesting sinks tracing backward their values. We analyze network-related sinks from Android, Java, and 19 manually selected popular network communication libraries, focusing on IoT application layer messaging protocols (e.g., MQTT, Constrained Application Protocol (CoAP), Advanced Message Queuing Protocol (AMQP), and Extensible Messaging and Presence Protocol (XMPP)) [30, 252]. Additionally, we consider ICC and cryptographic methods as sinks. We later use the reconstructed ICC information to bridge the ICC boundary during DFA. As apps might encrypt data before sending it, we also examine cryptographic methods.

Backward Tracing ❸ We then trace back through the program, starting at the identified sinks to all program points where the app modifies the values we are interested in. Naturally, this yields an over-approximate trace set. For example, if we want to reconstruct the parameter passed to `MqttManager` in Listing 2, then our reconstruction starts at line 15 (following ↔). We trace back the value of `config.endpoint` to line 14, to line 8, to lines 1–3, until we have traced all variables on which `config` depends.

Forward Simulation ❹ In the next step, we reconstruct the actual value set. Here, we must reconstruct arbitrary objects passed to the sinks, or we would miss the value of `config` in our example. That is, only reconstructing string operations is insufficient. Instead, we adapt our value reconstruction to handle arbitrary objects from any classes defined by the app, such as the `MqttConfig` class. We utilize reflection and forward simulate the backward trace, using the classes and methods as the app would do while normally executing it. Using reflection for simulating execution paths has a further advantage: We can handle code where the app itself uses reflection, which prior work cannot. However, reflection also introduces new challenges that we need to address:

(1) **Android Methods.** Some data might not be available statically, like user input. Additionally, we cannot simulate Android methods with reflection because only stub implementations are available and we use placeholders instead (e.g., intents, shared preferences, and database).



Listing 2: Simplified example code of an activity that receives BLE data and publishes it via MQTT. Arrows on the left show identification and reconstruction via VSA, marked ←→. Arrows on the right show DFA. Connecting reconstructions are marked →, via blue statements. The data flow from ICC source to sink, completing the flow from source to ICC sink of Listing 1, is marked →, through green statements. We highlight the reconstructed ICC key "device" yellow again.

(2) **Non-Terminating Methods.** Simulating arbitrary methods with reflection can also lead to non-termination, such as when it waits for an IoT device to connect. We mitigate this issue by terminating it after two seconds. We determined this threshold empirically as a trade-off between precision and time. In practice, most instructions finish within a fraction of a second.

(3) **Partially Reconstructed Values.** Partially reconstructed values can cause us to miss values. We may simulate a substring operation, but the analysis does not reconstruct the whole base string because parts depend on dynamic values that we cannot determine statically. This can then result in an out-of-bounds exception, which would cause us to miss more values. For example, if a URL obtained dynamically would contain a 32 character device serial number, but our placeholder is `from_pref`, then the analysis may cause an out-of-bounds exception if it accesses index 9. We mitigate this issue by preempting the calls that can cause such issues and expand the value on demand. Notably, this is not limited to string operations, but also extends to accessing arbitrary member fields of objects. For missing parameters or base objects, we attempt to create them with their default constructors. For primitive data types (e.g., boolean, int, or float), we assign default values.

Local vs. Remote Endpoints ❺ A unique aspect of companion apps is that their communication can be local, to connect to IoT devices or hubs, or remote, to connect to remote backends. We need to distinguish these classes to answer what data they share, how, with whom, and what the security and privacy impact is. Therefore,

we categorize endpoints as certainly local or possibly remote. That is, we identify local connections by checking whether a reconstructed endpoint points to a local IP address, a broadcast address, a multicast address, or the domain originates from user input (`fromUI.local`). We consider all other endpoints as remote.

2.3.2 Data-Flow Analysis

In the second phase of IoTFlow, we use DFA to trace data flows from IoT devices and sensitive Android methods. IoTFlow builds on FlowDroid [56], which is a data-flow framework for Android. We extended it to address the unique challenges of the IoT ecosystem. We (1) connect reconstructed endpoint information to data sent or received, similar to pointer analysis, and (2) trace flows across ICC.

Considering how modern apps work internally, we must pay particular attention to ICC. It is now the recommended way for app components to communicate with each other and often used, which is why tracing data flow across it is crucial. Theoretically, FlowDroid supports ICC via ICCTA [207]. However, ICCTA cannot generate ICC models for current Android apps [359, 243], which prevents FlowDroid from tracing flows through ICC. IoTFlow addresses this blind spot by treating ICC as sources and sinks, and connecting an ICC sink (writing to a key) to the corresponding ICC sources (reading from the key) by reconstructing the key used in ICC through VSA.

Connecting Reconstructions ❻ After reconstructing network endpoint information with VSA, we must connect them to the points where the app adds data to the request objects or receives a response, as these might be different from where the endpoint is set. For example the endpoint might be set during initialization of a connection object that is later used (repeatedly) to send or receive data. We identify the points where the app receives data and use the receiving statement as communication trigger points. Similarly, we need to connect a request's destination with the request's data when the request is executed. We do so using multiple data-flow analysis runs, which we split by method type for easier parallelization (e.g., MQTT, UDP, or CoAP). Returning to Listing 2, we previously reconstructed the MQTT broker endpoint via VSA (line 15). For our DFA in the next step, we now associate the MQTT broker endpoint (line 15) to the sink `publishString` (line 18) (marked ⟶).

Direct Data Flows (Source to Sink) ❼ We are interested in data flows from sources that are (1) Bluetooth, (2) responses from the local network, or (3) sensitive Android methods. We trace them to (1) ICC sinks and (2) remote sinks, that is, data leaks.[1] Bluetooth data is interesting as it may contain data from smart devices and local network communication is likely data from smart devices.

Crucially, we need to treat flows to and from the same method differently depending on the context and how the app uses the method (e.g., we want to analyze local network responses but ignore responses from remote endpoints). Thus, we extended FlowDroid to support context-sensitive flow analysis. We precisely identified the methods and the context that we need to consider as trigger points with the help of

[1] Full list of sources and sinks: https://github.com/SecPriv/iotflow/tree/main/config

our VSA and by Connecting Reconstructions ❻, which we can utilize to understand potential data leaks.

We focus first on three types of straight-forward immediate flows: (1) Bluetooth to network, (2) local network to network, and (3) sensitive data to network. Additionally, we trace sources to ICC sinks, to analyze flows across ICC, giving us three more flow types: (4) Bluetooth to ICC, (5) local network to ICC, and (6) sensitive data to ICC. Considering our example Listing 1, here, IoTFlow identifies the flow (marked →) from the Bluetooth source `bcg` in line 3 via line 5 to line 6, where the data `value` is passed to the intent using the key `BLE_DATA` (reconstructed via VSA, marked ↔).

Indirect Data Flows (Source to ICC to Sink) ❽ Finally, we need to follow up on the flows we identified that have an ICC sink, to properly bridge the ICC boundary. We trace the additional flow type (7) ICC source to network sink, and then precisely connect the new flows with previously identified flows of types (4)–(6). This allows us to discover and analyze data leaks involving ICC. For our examples Listing 1 and Listing 2, based on Direct Data Flows ❼, we identified a flow from Bluetooth to ICC using the key `BLE_DATA`. In Listing 2, using our indirect flow analysis, we now identify the flow (marked →) from the ICC source `getStringExtra()` in line 7 to line 9 to line 17 to line 18, where the app sends the Bluetooth data to the MQTT broker. Last, we connect the new ICC to network flow to the previously identified Bluetooth to ICC flow leveraging the VSA reconstructed ICC keys, giving us the indirect data flow that crosses the ICC boundary from Bluetooth to ICC to network.

2.4 Insights into the IoT Ecosystem

We evaluate IoTFlow on 10,836 apps on an Ubuntu 20.04.6 machine with 48 physical CPU cores (96 cores with hyper-threading, 2x Intel(R) Xeon(R) Gold 6342 CPU) and 1,024 GiB RAM. We limit the memory for the analysis of each app to 150 GiB (`-Xmx150g`).

2.4.1 Dataset

Verified Companion Apps We analyze IoTFlow on 9,889 unique IoT companion apps that were verified manually by prior work as part of three individual datasets [240, 235, 185]. We refer to our consolidated dataset as IoT-VER. It contains 455 apps collected by Neupane et al. [240] for studying if apps follow best practices, 5,100 apps that Jin et al. [185] used for the training, validation, and testing of IoTSpotter, and 6,208 apps that Nan et al. [235] collected and manually verified for IoTProfiler. Three quarters of the IoTProfiler apps are from the Google Play Store (74.6%), the remaining apps are from third-party stores. We did not augment these datasets with additional apps to not fragment the IoT companion app dataset space, which we deem important for reproducibility. Unfortunately, the public IoTSpotter dataset is incomplete and it misses 128 apps. Neupane et al.'s dataset misses two apps for which only the package name is available. We excluded these apps from our dataset.

All three datasets have 118 apps in common. IoTSpotter and IoTProfiler share 1,430 apps. The dataset of Neupane et al. shares 57 apps with IoTSpotter and 21 apps with IoTProfiler. If multiple datasets contain the same app, we only analyze the most recent version, that is, the app with the highest version code, since it is monotonically increasing [21]. Our consolidated dataset IoT-VER contains 9,889 apps, unique by their package names.

Popular General-purpose Apps We also downloaded 1,000 popular apps and games from the *top selling free* category of the Google Play Store in January 2022, which we use to illustrate the differences between IoT companion apps and other apps. We manually removed companion apps from the dataset and refer to the remaining 947 apps as GP-2022. To do so, two researchers independently classified each app based on its metadata in the Google Play Store. If they disagreed, they studied it in-depth until they reached an agreement.

2.4.2 Performance

We first discuss the performance of IoTFlow on our datasets (see Table 2.1). In addition to the total run time, we investigate the required time separately for VSA and DFA. On average, general apps take almost five times as long to analyze as companion apps (125m31s vs. 26m23s). This difference is even more pronounced when considering the median (129m36s vs. 6m51s): The processing time for companion apps is almost 20x faster than for general apps. Reasons may be the larger code base of general-purpose apps or that they tend to have more sources and sinks that we need to consider. Overall, we consider a median analysis time of less than 7 minutes and an average analysis time of approximately 26 minutes practical.

VSA Performance We allow up to 600 backward traces for each identified statement to prevent long-running analyses. Increasing the number of backward traces typically leads to more combinations of the same data, like request parameters. Each backward trace has up to 300 steps. We determined these thresholds empirically, observing a reasonable trade-off between resources and precision. Additionally, we configure timeouts for backward tracing (15 minutes) and forward computation (20 minutes). Our analysis only triggered the backward timeout when analyzing 11 (0.1%, all from GP-2022) apps and the forward timeout for 304 (2.81%; 155, 1.57% IoT-VER and 149, 15.73% GP-2022) apps, which we consider reasonable. Higher thresholds could lead to more flows being found.

Data-Flow Performance For DFA, we increased the timeout suggestions by the FlowDroid authors [61] by 50%. We set the FlowDroid callback collection timeout to 7m30s and the timeout for flow analysis to 15m. Our analysis triggered the callback timeout for 2,432 apps (22.44%) and the flow analysis timeout for 3,004 apps (27.72%). Separating the two datasets, 1,847 companion apps (18.68%) and 585 general-purpose apps (61.77%) triggered the callback timeout, while 2,484 companion apps (25.12%) and 520 general-purpose apps (54.91%) triggered the flow analysis timeout.

Table 2.1: *Dataset and Performance Overview.* We show for the VSA, Flow Analysis, and the total time (VSA+Flow Analysis), the average time (Avg.), median time (Med.), and standard deviation (Std.) per app in minutes [minutes:seconds].

		VSA			Flow Analysis			Total		
Dataset	**# Apps**	**Med.**	**Avg.**	**Std.**	**Med.**	**Avg.**	**Std.**	**Med.**	**Avg.**	**Std.**
IoT-VER	9,889	1:52	5:40	9:46	3:59	21:19	31:29	6:51	26:23	37:20
GP-2022	947	75:53	70:44	36:40	55:57	54:47	40:29	129:36	125:31	65:56

2.4.3 How Companion Apps Communicate

To answer *RQ1: How do companion apps and devices communicate?*, we identify device-to-app communication and the involved network protocols, and we study certificate pinning.

Direct Device Communication

First, we analyze the reconstructed values for indicators of direct communication with the devices, such as local IP addresses, broadcast, and multicast addresses, user-configurable addresses (i.e., endpoints from user input; marked as fromUI.local), and Bluetooth permissions. The latter indicates that the devices themselves might not have Wi-Fi capabilities, but that they use the companion app as a gateway to access the Internet. Some devices may also spawn their own Wi-Fi network that the phone needs to join for pairing. Within the network, the device has a fixed address known by the companion app. The apps can also use broadcasts to discover devices in local networks, for example apps use Universal Plug and Play (UPnP) to find devices that support screen mirroring. A fourth method is asking the user directly. Table 2.2 summarizes our findings.

IoT-Verified 6,355 (64.26%) apps declare at least one Bluetooth permission. We find a local IP address in 1,483 (14.99%) apps, a broadcast or multicast addresses in 452 (4.57%) apps, and addresses from user input in 123 (1.24%) apps. Among broadcast and multicast addresses, we found the broadcast address 255.255.255.255 (2.44%) most often, followed by the multicast DNS (mDNS) 224.0.0.251 (1.28%), and UPnP's 239.255.255.250 (0.75%). Besides the IPv4 addresses, we found three (0.03%) IPv6 multicast addresses.

General-purpose Apps We observe a significant lower number for all four direct device communication indicators for general-purpose apps in GP-2022. We find local IP addresses in only 2.21% of apps, compared to 14.99% in IoT-VER. Similarly, broadcast and multicast addresses drop from 4.57% in IoT-VER to 0.42% in GP-2022. Only one (0.11%) address depends on user input in GP-2022, compared to 123 (1.24%) addresses in IoT-VER. The number of apps requesting Bluetooth permissions also decreased from 64.26% in IoT-VER to 19.01% in GP-2022. These

Table 2.2: *Number of Apps using Direct Device Communication.* Indicators are hard-coded local network IP addresses (grouped if found in 30 or more apps), user-configurable addresses (fromUI.local), broadcast and multicast, or Bluetooth.

Address	**IoT-VER**	**GP-2022**
10.*.*.*	716 (7.24%)	12 (1.27%)
10.0.0.172	516 (5.22%)	1 (0.11%)
10.0.0.200	438 (4.43%)	
10.10.2.2	48 (0.49%)	7 (0.74%)
other	242 (2.45%)	12 (1.27%)
172.16-31.*	103 (1.04%)	4 (0.42%)
172.17.0.1	49 (0.50%)	1 (0.11%)
other	56 (0.57%)	3 (0.32%)
192.168.*.*	746 (7.54%)	4 (0.42%)
192.168.0.1	115 (1.16%)	
192.168.1.1	180 (1.82%)	2 (0.21%)
192.168.1.3	36 (0.36%)	
192.168.4.1	77 (0.78%)	
other	518 (5.24%)	2 (0.21%)
fe80	3 (0.03%)	
Multicast and Broadcast	452 (4.57%)	4 (0.42%)
224.0.0.251	127 (1.28%)	1 (0.11%)
239.255.255.250	74 (0.75%)	
255.255.255.255	241 (2.44%)	4 (0.42%)
IPv4 other	93 (0.94%)	
IPv6 other	3 (0.03%)	
fromUI.local	123 (1.24%)	1 (0.11%)
Bluetooth	6,355 (64.26%)	180 (19.01%)

findings strengthen our assumption that our direct device communication indicators are indeed meaningful.

Takeaways We identified four strategies apps use to communicate locally with smart devices, and we show by comparing them to general-purpose apps that they are indeed specific to companion apps. Identifying this kind of communication helps security and privacy analyses (see Section 2.4.5). Prompting the user for the device location and using multicast can be dangerous and prone to misconfigurations. Users might make devices unwittingly accessible over the Internet [72]. A Shodan [310] query for open port 554 returns 78,858 results of exposed cameras, suggesting that misconfigured devices accessible remotely are a common issue. Attackers can also sniff broadcast packages or mimic the legitimate device to act as a Monkey-in-the-Middle (MITM) [116]. Finally, we note that any information about local network devices is sensitive and can be abused for advertising and tracking purposes [197].

We recommend to use device discovery and avoid requiring user configurable addresses, to reduce the risk of accidental misconfigurations [99]. Apps should also respect users' privacy and not send local network information to remote servers. In fact, they should prefer local communication over cloud communication whenever possible, as remote requests can reveal usage patterns to others.

URL Protocol Schemes

We identify network protocols based on the values we reconstructed through VSA. First, we analyze the URL schemes of the endpoints that apps communicate. Second, as we reconstruct endpoint information for libraries for *AMQP*, *MQTT* and *XMPP* communication, we can draw conclusions about them, even if they do not use specific schemes. Table 2.3 summarizes our results. The row *IoT-related* summarizes the schemes and protocols that are tailored to IoT devices. We group IPP, IPPs, RMTP, and VNC as *IoT-other* as we found them only in one or two apps, and we group protocols from IANA's list of URI schemes [179] that are less interesting for our use case (e.g., *service, about, info*) as *Other*. Overall, for IoT-VER, we reconstructed schemes in 7,113 unique apps for remote endpoints and in 871 apps for local communication.

HTTP(S) We find that apps still widely use plain HTTP. Our numbers represent an upper bound as we do not know how many actual connections occur over HTTP since we base our results on statically reconstructed endpoints. In practice, HTTP might be upgraded by default, but even if used as a fallback, HTTP can lead to security and privacy issues through protocol downgrade attacks.

Our results show that a high proportion of HTTP traffic, compared to HTTPS traffic, is for local communication. This is not surprising: Local communication might appear safe, and deploying TLS properly for IoT devices remains challenging [261]. Nevertheless, even if communication is local, TLS protects against eavesdroppers, which is important as devices use broadcast media like Wi-Fi.

MQTT Endpoints Smart devices have unique usage scenarios and requirements, such as device-to-device communication and energy efficiency. Traditional communication protocols do not satisfy these requirements. New protocols can fit these demands, but they can also threaten security and privacy, especially if they were designed without considering an adversarial environment or if developers make wrong assumptions about them. One protocol used often by IoT devices is the *Message Queuing Telemetry Transport (MQTT)* protocol. In practice, it often lacks authentication and authorization, allowing attackers to access user data or take over devices [184, 357].

MQTT is the most widespread IoT-specific communication protocol for IoT-VER. We reconstructed 147 MQTT endpoints in 176 apps, of which nine represent local IP addresses. We verify that the remaining 138 remote endpoints are indeed valid by opening a connection to them. To not raise any ethics concerns, we only open and immediately close the connection, and we do *not* perform any action (e.g., subscribing to a topic). We use the Python Paho library [105] for our test and base our results on the return code: If the connection is successfully established (return code 0) or an

Table 2.3: *Number of Apps with Reconstructed URL Protocol Schemes.* Percentages are relative to the numbers of total apps with at least one scheme. For IoT-VER, we identified schemes for 871 local endpoints and 7,113 remote endpoints. For GP-2022, we identified schemes for 14 local endpoints and 898 remote endpoints. Protocols marked with a star (*) are based on IoTFlow identifying the corresponding libraries.

	Local		**Possibly Remote**	
Protocol	**IoT-VER**	**GP-2022**	**IoT-VER**	**GP-2022**
Android			29 (0.41%)	184 (20.49%)
File	4 (0.46%)		2,180 (30.65%)	578 (64.37%)
FTP	1 (0.11%)		8 (0.11%)	
HTTP	788 (90.47%)	13 (92.86%)	4,901 (68.90%)	639 (71.16%)
HTTPS	81 (9.30%)	2 (14.29%)	5,445 (76.55%)	885 (98.55%)
IoT-related	49 (5.63%)		315 (4.43%)	1 (0.11%)
AMQP*			6 (0.08%)	
Cast			4 (0.06%)	
CoAP*	2 (0.23%)		9 (0.13%)	
CoAPs*			2 (0.03%)	
MQTT*	27 (3.10%)		158 (2.22%)	1 (0.11%)
Palm			58 (0.82%)	
RTSP	1 (0.11%)		15 (0.21%)	
RTSPs			2 (0.03%)	
TV			9 (0.13%)	
URN	8 (0.92%)		18 (0.25%)	
XMPP*	11 (1.26%)		29 (0.41%)	1 (0.11%)
IoT-other			4 (0.06%)	
JAR			65 (0.91%)	1 (0.11%)
SMB	4 (0.46%)		62 (0.87%)	2 (0.22%)
WS	14 (1.61%)	7 (50.00%)	130 (1.83%)	10 (1.11%)
WSS	2 (0.23%)		138 (1.94%)	14 (1.56%)
Other	7 (0.80%)		1,604 (22.55%)	830 (92.43%)

error related to connection parameters is returned (return codes 1 to 5), we consider the endpoint as reachable and valid.

We connected successfully (return code 0) to 74 MQTT endpoints (53.62%). To further investigate the remaining 64 endpoints, we probed for other ports typically used for MQTT (1883 and 8883) with `nmap` [211]. Seven endpoints were *closed* and 37 were *filtered*, meaning our connection attempts were prevented at the network level. One reason may be geographical restrictions. The remaining 20 endpoints were unresponsive to ICMP echo requests and we consider them unreachable.

MQTT Credentials IoTFlow can also reconstruct authentication credentials. Hard-coding credentials into the app can lead to attacks on the integrity and confidentiality of data by allowing an attacker to connect and publish or subscribe to topics (e.g., modifying a parameter of a physical actuator). We reconstructed 30 unique user-names and 34 unique passwords in IoT companion apps.

MQTT Topics and Payloads Our analysis can reconstruct the topics (i.e., topics for which the phone or IoT device should receive messages) and message payload formats (i.e., the format of messages shared between phone, IoT device, and the cloud). We found 726 topic names and 330 payload formats. While we may miss dynamic values from communication with the device, the information we gain is valuable to understand the behavior of IoT apps and devices.

Other IoT Protocols We also identified other IoT protocols in IoT-VER apps, namely XMPP, AMQP, and CoAP. Among the 36 XMPP endpoints we identified, we could connect to five, the port was filtered for six, and the remaining 25 endpoints were unresponsive to ICMP echo requests. For the six identified AMQP endpoints, we could connect to two, we received an authentication error for one, and the AMQP-specific port was closed or filtered for the remaining endpoints. For the two CoAP endpoints, one was a local IP address, but we successfully reconstructed 55 unique URL paths used to specify the location of resources on the server.

General-purpose Apps For GP-2022, we reconstructed local addresses only in combination with HTTP, HTTPS, and WS (WebSocket). Like for companion apps, only a minority uses HTTPS locally (14.29%). However, unlike IoT apps, nearly all apps (98.55%) use it for remote communication. Unsurprisingly, general apps do not use IoT protocols. Only a card game app uses MQTT and XMPP.

Takeaways We found widespread adoption of HTTP across IoT-VER apps despite its insecurity. For GP-2022 apps, the situation improves as almost all apps communicate over HTTPS. However, in both datasets, most local communication does not adopt TLS to secure the connection. We also identified how IoT-specific protocols (MQTT, AMQP, XMPP, CoAP) are being used and we reconstructed crucial information, like credentials and topics.

Generally, apps should not use hard-coded credentials but generate them individually during initialization, use limited and narrow authorization scopes, follow best practices (e.g., encrypting Android shared preferences [22]), and encrypt all communication (e.g., via TLS, but preferably end-to-end).

Pinning and Certificates

An additional aspect of how companion apps secure their communication is certificate pinning. It is a contentious topic: While OWASP [342] suggests it when the app wants to verify the host's identity, Google [17] advises not to adopt it because of issues deriving from certificate changes. However, determining whether it is good or bad is out of scope of our work.

We use the approach by Pradeep et al. [269] to identify pinning and the corresponding certificates by analyzing the Network Security Configuration (NSC) specified in the Android Manifest and the certificates included in the app. Table 2.4 summarizes our results.

IoT-Verified More companion apps include certificates (12.21%) than use pinning (3.89%). On average, each app includes 3.21 certificates. More than half of the

Table 2.4: *Certificates and Pinning.* The first rows show the number of apps in which we found pinning, certificates, and apps containing expired or self-signed certificates. The remaining rows show the corresponding certificates. The number of expired certificates at the time of download is a lower-bound for IoT-VER because it is not always known.

		IoT-VER	**GP-2022**
Apps	Pinning	385 (3.89%)	111 (11.72%)
	Certificates	1,207 (12.21%)	119 (12.57%)
	Expired (at download)	474 (4.79%)	49 (5.17%)
	Expired (May 2023)	822 (8.31%)	59 (6.23%)
	Self-Signed	1,042 (10.54%)	91 (9.61%)
Certificates	Total Number	31,285	1,837
	Expired (at download)	3,976 (12.71%)	268 (14.59%)
	Expired (May 2023)	9,129 (29.18%)	321 (17.47%)
	Self-Signed	18,018 (57.59%)	684 (37.23%)
	Avg per App (Std)	3.21 (20.97)	1.94 (14.59)

certificates in IoT-VER were self-signed, possibly to communicate with IoT devices. We also investigate if certificates were expired when the apps were downloaded. If the download date is unknown, we infer it based on the app versions. Our numbers are lower bounds for the apps from IoTProfiler and Neupane et al. because we assume apps were downloaded on the first day of the year when they could have been downloaded. We treat certificates as expired if their expiration date is before 2018 for IoTProfiler and before 2021 for apps by Neupane et al. Apps might be downloaded later, but this does not threaten validity as our numbers are a lower bound. For IoTSpotter apps, the download date is available as they were downloaded via AndroZoo [7]. Overall, 12.71% certificates were expired when the apps were downloaded (in 4.79% apps). In May 2023, 822 (8.31%) apps contain 9,129 (29.18%) expired certificates. Intuitively, expired certificates point to poor security practices and can even prevent communication.

General-purpose Apps Compared to IoT-VER, more apps adopt pinning (11.72% vs. 3.89%), but the same proportion of apps include certificates (12.57% vs. 12.21%). On average, however, they include less certificates (1.94 vs. 3.21). One reason may be the lower number of self-signed certificates. While more than half of all (57.59%) certificates are self-signed for companion apps, only slightly more than one third (37.23%) of certificates are self-signed for general-purpose apps. At the download date, 14.59% certificates were already expired. IoT-VER's older download date could be a reason for the increase in expired certificates in May 2023 (29.18% vs. 17.47%).

Takeaways Comparing included, expired, and self-signed certificates, we can conclude that more certificates do not lead to better security. Companion apps include substantially more certificates than general-purpose apps, and proportionally significantly more of them are expired or self-signed. Interestingly, fewer companion apps adopt the controversial practice of certificate pinning.

Generally, developers should renew certificates well before expiration, as users may not install updates immediately. Further care is needed for self-signed certificates as apps must add code to explicitly trust them, or Android will prevent the communication that attempts to use them. Worse, doing so incorrectly, like instructing TrustManager to trust every certificate, enables MITM attacks [17].

2.4.4 With Whom IoT Apps Communicate

After analyzing how apps communicate, we investigate *RQ2: Who are companion apps communicating with?* We categorize the reconstructed fully-qualified domain names (FQDNs) and effective top-level domains+1 (eTLD+1) to spot potentially problematic endpoints, like trackers, investigate where data is sent geographically, and analyze if endpoints are vulnerable to domain takeovers.

Advertisers and Trackers

We classify the FQDNs to learn who receives data from the app and, via the app, from the devices. We use the domain lists by Ren et al. [287], which they compiled from various ad-blocking lists. Additionally, we use the Exodus tracker list [113]. Table 2.5 summarizes our results.

IoT-Verified Overall, 2,959 (29.92%) apps include 487 unique advertisement FQDNs and 1,647 (16.65%) apps use 114 analytic-related FQDNs. Although they belong to different categories, both domains behave similarly by collecting user information. We also reconstructed 410 FQDNs pointing to Content Distribution Networks (CDNs) in 1,165 (11.78%) apps. Additionally, we identified 84 social network FQDNs shared across 1,046 (10.58%) apps, with the respective standard deviation indicating that if apps use one, they often use more. The remaining 7,248 FQDNs in 4,917 (49.72%) apps do not fall in our categories and we label them *Other*.

General-purpose Apps The average number of advertisement and tracker FQDNs per general-purpose app is 6.33, eight times higher than per companion app (0.76). Additionally, they occur in almost all apps (89.55%), while they only occur in less than one third (29.92%) of companion apps. The situation for analytics FQDNs is similar (71.70% vs. 16.65%). Most analytics and crash reporting FQDNs are shared between the two datasets, while FQDNs from other categories are mainly limited to one dataset.

Takeaways The large number of 7,248 *Other* FQDNs in companion apps combined with the low number of 271 FQDNs shared with general-purpose apps (3.25% of all IoT FQDNs) suggests that many are IoT-specific. Prior work observed a low coverage of existing filter lists for IoT domains [322, 212], highlighting the need for

Table 2.5: Categorized Endpoints by IoTFlow for IoT-VER, GP-2022, and Comparison between IoTFlow (IF) and Dynamic Analysis (DA). We report with the number of unique FQDN per dataset and shared between them, prefixed with # the number of apps with at least one domain per category, the average number of domains per app, and the standard deviation.

	Large-scale IoTFlow Analysis								**IoTFlow vs. Dynamic Analysis**			
	IoT	**GP**	∩	**# IoT**	**# GP**	$\bar{IoT}$ **(SD)**	$\bar{GP}$ **(SD)**	**IF**	**DA**	∩	∩ **TLDs**	**# Apps**
Ads.	487	279	141	2,959 (29.92%)	848 (89.55%)	0.76 (1.73)	6.33 (4.92)	34	56	10 (12.50%)	9 (39.03%)	13 (100%)
Analytics	114	96	78	1,647 (16.65%)	679 (71.70%)	0.38 (1.03)	2.89 (3.68)	12	16	6 (27.27%)	5 (45.45%)	9 (69.23%)
CDNs	410	97	26	1,165 (11.78%)	733 (77.40%)	0.17 (0.64)	1.02 (0.97)	9	16	- (-)	- (-)	9 (69.23%)
Crash Report	4	3	3	195 (1.97%)	192 (20.27%)	0.02 (0.15)	0.23 (0.49)	6	1	1 (16.67%)	1 (50.0%)	3 (23.08%)
Social Net.	84	49	24	1,046 (10.58%)	137 (14.47%)	0.37 (1.45)	0.23 (0.79)	11	6	1 (6.25%)	1 (14.29%)	2 (15.38%)
Other	7,248	1,420	271	4,917 (49.72%)	685 (72.33%)	2.08 (4.57)	3.52 (5.29)	80	84	17 (11.56%)	19 (25.33%)	12 (92.31%)

Table 2.6: *Geographic Location of Network Endpoints.* The numbers show the amount of endpoints from each location and the ratio to the overall number of endpoints.

	US	**CN**	**EU**	**Asia**	**UK**	**RU**	**Other**
IoT-VER	17,283 (46.09%)	10,221 (27.25%)	5,380 (14.35%)	3,166 (8.44%)	438 (1.17%)	113 (0.30%)	901 (2.40%)
GP-2022	10,606 (79.69%)	181 (1.36%)	1,786 (13.42%)	207 (1.56%)	47 (0.35%)	299 (2.25%)	183 (1.38%)

more scrutiny by future work into who receives data by these apps. Prior work showed that users value IoT security and privacy [108] and are willing to pay a premium for devices that respect their security and privacy [109]. Indeed, not using ad services or trackers could be a unique and convincing differentiating value proposition for IoT devices, especially because users already pay for the device.

Geographic Location

Next, we determined the location of the reconstructed FQDNs to study where data is sent and which countries receive IoT data. We first resolved the FQDNs to determine the location of the IPv4 addresses against the allocated blocks [110]. We resolved them from Vienna, Austria, which is in a jurisdiction that has implemented the EU's General Data Protection Regulation (GDPR) [111]. Notably, due to geographic split horizon DNS (GeoDNS), the resolved IP addresses may differ for other vantage points. Table 2.6 shows aggregated geographic regions. We perform our analysis at the FQDN level because the FQDN endpoint receives the data. This granularity is also important because FQDN and eTLD+1 locations can differ. For example, `xiaomi.com` is hosted in China, but `ru.register.xmpush.xiaomi.com` is hosted in Russia.

We can make multiple observations comparing endpoint locations between companion apps and general-purpose apps. First, substantially fewer endpoints for companion apps are in the US (46.09%) than they are for general-purpose apps (79.69%). The difference (33.6 percentage points) stems almost exclusively from more Chinese endpoints (27.25% to 1.36%, 25.89 pp), with the remainder (7.71 pp) being nearly covered by other Asian countries for IoT-VER (8.44% compared to 1.56%, 6.88 pp). Other regions remain mainly stable.

Takeaways The scattered geographic location of endpoints might raise privacy concerns. Countries have implemented various data protection regulations with stricter or more relaxed requirements. For example, the EU's GDPR [111] is considered the world's strongest privacy law. If a European user downloads an app that contacts endpoints outside the EU, their data is subject to GDPR, but the app may transfer it to foreign countries and process it there. This clearly raises privacy concerns and may even be illegal. Moreover, even if no sensitive data is sent directly, metadata can suffice to infer usage patterns, which can be sensitive (e.g., for smart locks).

Abandoned Domains

Domains that could be re-registered but are still used pose severe security and privacy risks for users as attackers could take them over. A similar argument applies to domains that are registered, but for which DNS information is stale and where the corresponding IP address could be taken over [71]. We focus on expired domains as they provide longer-term capabilities to attackers. We extract the eTLD+1 from the reconstructed FQDNs to identify abandoned domains. We then resolve the eTLD+1 to test whether they are in use. For domains we cannot resolve, we use `WHOIS` to check if it is registered or free.

IoT-Verified We identified 136 potentially abandoned domains in companion apps. After manually investigating and removing artifacts, we verified that 67 domains from 73 apps are indeed available for registration. They are in apps for watches, TVs, cars, health equipment, security and baby cameras, lights, and locks. An attacker could take over these devices by registering the domains.

We also investigated if the 73 apps can still be downloaded from the Google Play Store. Unavailable apps remain critical, but differently so. They can still impact users as the devices might not have been replaced and they might still connect to those domains. We found that 27 apps (37.0%) are available. Remarkably, one app has over one million downloads, a second app has over 500,000, and three others have more than 100,000. For ten apps, based on the reconstruction information, it is likely the domains receive IoT data. They use IoT information in URLs, such as `ipcDeviceIdList` as a request parameter, or `petinfoDatas/addpet` as a path. Eight apps use abandoned domains to download files, which may be executed or could be device updates. Sixteen domains are API endpoints and also likely receive sensitive data. We responsibly disclosed our findings to developers and the Google Play Store.

Takeaways Pariwono et al. [262] investigated abandoned domains for general apps, but the dangers can be more serious for IoT devices. Attackers could not only take over the apps and receive PII, but they might also be able to control hundreds of thousands of devices, enabling large-scale distributed denial-of-service attacks and allowing them to create botnets. Our analysis shows (1) that abandoned domains are a real danger in the IoT ecosystem, (2) that they affect a varied range of devices, and (3) what data they receive.

Developers should actively monitor the domains that their apps may contact, including those of third-party libraries. Additionally, old or deprecated domains that may still be contacted should remain registered, as users may depend on outdated app versions, and made inoperable instead of allowing others to register it.

2.4.5 What Data Companion Apps Share

To answer *RQ3: Which data are companion apps sharing (and how)?*, we first report what data apps can access, based on the requested permissions, to understand what data they could share. We then analyze the data flows we extracted to identify leaked data and whether encryption is used to protect data.

Permissions

We extract permissions and *protectionLevel* with Androguard [98]. We focus on permissions with a *protectionLevel* of *dangerous* (permissions protecting sensitive resources) and *privileged* (permissions that third-party apps should not adopt).

On average, GP-2022 apps request more permissions than companion apps (17.54, SD 14.10 vs. 14.26, SD 10.23). For IoT-VER, 8,769 (88.67%) apps request at least one dangerous permission. `WRITE_EXTERNAL_STORAGE` occurs most often (7,209 IoT apps, 72.9% and 559 general-purpose apps, 59.03%). The second and third most frequent permissions are `ACCESS_COARSE_LOCATION` (5,735 IoT apps, 57.99%) and `ACCESS_FINE_LOCATION` (5,529 IoT apps, 55.91%) as in most cases IoT devices also rely on location to perform their functions (e.g., smart watches recording physical activity).

Additionally, 2,604 (26.33%) IoT apps request one or more privileged permissions, while only 73 (7.71%) GP-2022 apps do. Even if system permissions are requested, they will only be granted if the phone is rooted or if the app has a special entitlement (e.g., the phone vendor may grant such an entitlement to their own apps, and they might also produce IoT devices). They can also occur for backward compatibility reasons or be remnants from development that were never removed (e.g., the second most common privileged permission is `READ_LOGS`, which appears in 998 IoT apps).

Finally, 5,660 (57.24%) IoT apps use "non-standard" permissions. The permission occurring the most belongs to Google Cloud Messaging (GCM) (3,656 apps, 36.97%) and is used when receiving a broadcast from GCM. We also find permissions of specific brands, for example, Huawei (60 permissions occur 1,207 times in 424 apps) or Sony (31 permissions 787 times in 424 apps).

Takeaways General-purpose apps request more permissions than companion apps on average. However, IoT apps use more *privileged* and *dangerous* permissions, with two of the most requested *dangerous* permissions being for the user's geographic location.

We recommend to regularly review if permissions are still current, to request the least necessary set of permissions, and to only temporarily acquire them when needed. With new Android updates, permissions might also change, for example, scanning for Bluetooth devices required location permissions only up to Android 12 [20, 163].

Data Flows

To learn more about what data is sent, we analyze the flows we discovered via DFA. Table 2.7 summarize our findings. We distinguish between three flow types based on the destination: Bluetooth, local network, or a sensitive Android API. We determine where the data is sent by connecting the VSA results with the individual flows. Unfortunately, we may not have precise information for all flows for two reasons: (1) VSA might not reconstruct an endpoint precisely, for example, because it depends on dynamic values, (2) we could not connect reconstruction and flow.

We identified data flows from Bluetooth and local network sources only for IoT apps, which is not surprising, as we have shown that such communications are companion app specific (see Section 2.4.3). Overall, we found 579 flows from Bluetooth

Table 2.7: *Flow Analysis.* We separated the flows by their categories. The ICC columns represent the flows involving any ICC, and the endpoint columns (Endp.) the flows with additional endpoint information. The ratio concerns the number of flows from the category. The app columns show the number of apps with the respective flows and the relation to the apps in the dataset.

	Bluetooth				Local Network				Android			
Dataset	ICC	Endp.	Flows	Apps	ICC	Endp.	Flows	Apps	ICC	Endp.	Flows	Apps
IoT-VER	497 (85.84%)	50 (8.64%)	579	90 (0.91%)	4 (5.33%)	49 (65.33%)	75	53 (0.54%)	2,340 (34.89%)	1,952 (29.11%)	6,706	1,682 (17.01%)
GP-2022									420 (30.75%)	619 (45.31%)	1,366	318 (33.58%)

sources in 90 apps. Remarkably, 497 (85.84%) of these flows involve ICC, which highlights the need for DFA that is ICC aware, like our approach. We precisely identified endpoint information (i.e., where the data is sent to) for 50 (8.64%) flows. For local network sources, we discovered 75 flows, of which four (5.33%) involve ICC. IoTFlow reconstructed precise endpoint information for 49 (65.33%) of them. Finally, we identified 6,706 flows from sensitive Android API in 1,682 (17.01%) IoT apps, and 1,366 such flows in 318 (33.58%) GP-2022 apps.

Case Study: Smart Grill Our analysis finds a flow in a companion app for smart grills. The app reads data from the device via BLE, parses it, process it via an intent, and later sends it to an Amazon AWS endpoint via MQTT. We successfully connected to the endpoint without requiring credentials (anonymously) (see Section 2.4.3). This means that we could potentially receive data from others (for ethical reasons, we did not explore this further).

Case Study: Smart Camera In a smart camera companion app, we found a flow from `getDeviceId` to a remote endpoint. The app uses the IMEI together with a username and password for authentication. Worth mentioning is also that the app hashes the password using MD5, which is insecure and cryptographically broken. Afterward, the app encrypts the username and password with 3DES, which is also insecure and cryptographically broken. IoTFlow's VSA reconstructed the key, even though the app developers put one byte of the key into a different class file, potentially trying to obfuscate it and avoid regex-based key recovery.

Sharing the IMEI is problematic because users can only change the IMEI by physically replacing the device as it is a non-resettable hardware identifier. Google strongly discourages developers from using any hardware identifiers, including the IMEI [19], and it is also prohibited by Android's user data policy [151]. With Android 10 (API level 29, released in 2019), Google added additional restrictions to access the IMEI [25, 19], but around 14.4% of users are still using older versions, allowing apps to access these identifiers [67].

Geographic Location We also analyze the geographic location of data flows with endpoint information. For IoT-VER, we find 917 (15.04%) flows sending data to Chinese and 604 (9.90%) to US endpoints. The share of flows with US endpoints

increases for general-purpose apps (75, 27.88%), while the share for Chinese drops (12, 4.46%), aligned with their distribution (see Section 2.4.4). Positively, as we conducted our experiments from the EU, most destinations are within the EU: 73.80% for IoT-VER and 67.66% for GP. Unfortunately, it also means that more than 25% of destinations are outside the EU, potentially violating GDPR. The situation is worse for Bluetooth-based sources than it is for local network flows. For Bluetooth, 27 endpoints (45.76%) are US endpoints, and for local network flows, 37 endpoints (52.86%) are Chinese endpoints. Our artifact provides more details.[2]

Takeaways With the help of IoTFlow's combination of VSA and DFA, we identified real-world security and privacy issues in the IoT ecosystem and we discussed what data companion apps leak and where they send it, which we illustrated with two examples.

Following best practices and for privacy reasons, developers should minimize data they collect and use, and only send data if it is truly necessary. Generally, we recommend to process as much data as possible locally, and to encrypt any data leaving the devices.

Encryption Analysis

Finally, we analyze the encryption algorithms apps use and we investigate the reconstructed data passed to cryptographic methods. We reconstructed the algorithms in 812 (85.74%) GP-2022 apps and in 4,069 (41.15%) IoT-VER apps. Table 2.8 summarizes our results. AES is the most widely used encryption algorithm for IoT apps (92.97%) and GP-2022 apps (99.38%). Algorithms that are considered insecure or cryptographically broken are much more prominent in IoT apps (1,461 apps, 35.80%) using encryption than they are in GP-2022 apps (135, 16.63%).

We also evaluate reconstructed encryption keys. Unfortunately, removing false positive artifacts is extremely challenging because it is difficult to determine whether a key is truly used as is. For example, a 16-byte array with all 0 values could be an insecure key, or it may not have been initialized. Therefore our numbers are an upper bound. Overall, we reconstructed hard-coded keys in 2,321 (57.04%) IoT apps and 408 (50.24%) general-purpose apps.

Takeaways The main differences between IoT apps and GP-2022 apps are in what encryption they use and how they use them. Using hard-coded keys and broken encryption algorithms gives a false sense of security and does not provide security or privacy. Unfortunately, both issues are worse for companion apps.

Beyond using strong encryption algorithms, we also recommend to initialize encryption keys on demand and to store them securely, for example, with the help of Android KeyStore [18].

[2] https://github.com/SecPriv/iotflow/tree/main/scripts/evaluation/dfa

Table 2.8: *Encryption Algorithms.* The number of apps that use the respective encryption algorithm and its relation to the number of apps with encryption algorithms (4,069 IoT-VER, and 812 in GP-2022). Recommended algorithms are marked 🔒. Algorithms considered insecure or broken are marked ⊘.

Algorithm	IoT-VER	GP-2022
🔒 AES	3,794 (92.97%)	807 (99.38%)
🔒 ChaCha	4 (0.10%)	3 (0.37%)
🔒 Diffie-Hellman	14 (0.34%)	44 (5.42%)
🔒 RSA	16 (0.39%)	
Serpent	136 (3.33%)	31 (3.82%)
⊘ Blowfish	79 (1.94%)	10 (1.23%)
⊘ DES	1,366 (33.47%)	120 (14.78%)
⊘ 3DES	351 (8.60%)	66 (8.13%)
⊘ GOST		5 (0.62%)
⊘ RC4	288 (7.06%)	21 (2.59%)

2.5 IoTFlow vs. Dynamic Analysis

Our static analysis approach has some limitations, especially because we do not require access to the device. It is crucial to understand what and how much information we can truly reconstruct from companion apps without the device. We verify the accuracy and completeness of the reconstructed values by analyzing and comparing the results we obtained through IoTFlow with our in-depth manual analysis when interacting with apps and devices. To this end, we recorded traffic when using 13 different devices and their companion apps in our lab environment (see Table 2.9).

Our test environment uses a machine running Ubuntu 20.04 with `frida-tools` [279] and `mitmproxy` [92], and a rooted Google Pixel 4 running `frida-server` [279] on Android 12. The machine hosts a Wi-Fi network to which the phone and the devices connect, providing Internet connectivity through an Ethernet connection. We bypass certificate pinning via Frida's built-in scripting. We test companion apps with two strategies: First, *automatic inputs*, we test apps with the Application Exerciser Monkey (AME) [26] for 10 minutes or until they crash, whichever occurs first. We do not expect to trigger complex behavior of IoT devices (e.g., because it would require us to set up the device), but AME remains a common testing technique [78] and we include it for completeness. Second, *manual inputs*, we manually interact with each app for 30 minutes and trigger all functionalities, including pairing and interacting with IoT devices, changing their settings, etc. Indeed, we observed significantly less traffic with AME than with manual interaction, demonstrating the scalability issues of dynamic analysis.

From the observed traffic, we extract requests' domain names, which correspond to who receive data, and resource paths, which correspond to functionality (e.g., API endpoints). We match them based on exact string equivalence exactly between

Table 2.9: *Tested Devices.* The IoT devices that we tested dynamically together with their device type and package name.

Device	Type	Package Name
Bose QC35	Headphones	com.bose.monet
Divoom Timebox	Alarm Clock	com.divoom.Divoom
Fitbit Inspire 1	Smart Watch	com.fitbit.FitbitMobile
Blaupunkt	Smart Watch	cn.xiaofengkj.fitpro
HHCC FlowerCare	Plant Sensor	com.huahuacaocao.flowercare
Hama WiFi	Light Bulb	com.hama.smart
Philips Hue	Light Bulb	com.signify.hue.blue
Ikea DIRIGERA	Smart Hub	com.ikea.tradfri.lighting
Anti-Lost	Smart Tracker	com.lenzetech.kindelf
LIFX A60	Light Bulb	com.lifx.lifx
Nut Find3	Smart Tracker	com.nut.blehunter
Soundcore Life Q35	Headphones	com.oceanwing.soundcore
Wiz Colour	Light Bulb	com.tao.wiz

IoTFlow and dynamic analysis. Considering the configuration of our dynamic environment, we also manually match domains and paths by (a) identifying over-approximate placeholders, such as the device product code, serial number, etc. and matching them to concrete dynamic information, (b) generalizing the dynamic system configuration, like language and locale, (c) grouping repeated dynamic data (as they also do not provide new information in the dynamic analysis setting, but provide a false sense of accuracy), and (d) resolving network-level redirects (e.g., DNS-based or IP anycast). We remove analysis artifacts that are *clearly not* related: (a) domains and resources that were requested from outside of the IoT app, such as by the Android operating system (e.g., background update checks), (b) domains and resources that were requested by Android WebView components unrelated to device behavior (e.g., opening a vendor's online shop website), and (c) invalid domain names. We retain all data that cannot be clearly attributed to dynamic analysis artifacts, making our results a lower-bound. Last, as we focus on IoT-related behavior, we manually label data as related if it relates to IoT device behavior, security, privacy, or data exchange. Our artifact provides further details on the identified domains and paths, and their matching.[3]

IoTFlow extracted 214 domains from the 13 companion apps, with a minimum of 3 domains, an average of 16.46 domains, and a maximum of 42 domains per app. With dynamic analysis, we observed 218 domains, with a minimum of 1 domain, an average of 16.77 domains, and a maximum of 48 domains per app. Between static analysis and dynamic analysis, 36 domains match exactly and we matched 7 additional domains manually.

We categorize all domains using our previous approach (see Section 2.4.4). Table 2.5 summarizes our results and our artifact provides further details.[3] Notably, we

[3] https://github.com/SecPriv/iotflow/tree/main/dynamic_analysis

find substantially more advertisement domains via dynamic analysis than through IoTFlow. This is expected because of how modern ads are targeted and auctioned, requiring dynamic information. IoTFlow only recovers the entry point for ads, but this is actually sufficient to determine that they are used. It also highlights an important issue: Considering all domains gives a false sense of accuracy toward dynamic analysis, many of which may not provide new insight. For example, while it confirms our findings of extensive tracking in IoT apps, the IKEA companion app contacts 48 domains in total, but it also contacted 20 advertising domains and four social network domains.

Focusing on *certainly* IoT domains, IoTFlow and dynamic analysis share 21 domains across all apps (min 0, avg 1.62, max 5), IoTFlow identified 33 domains that dynamic analysis missed (0/2.54/10), and dynamic analysis found 19 unique domains (0/1.46/4). That is, IoTFlow performs better than or equal to dynamic analysis for 9/13 devices and worse for only 4/13 devices (Fitbit smart watch, Hama light bulb, Soundcore headphones, and Wiz light bulb). For 2/4 of these apps, Fitbit and Wiz, IoTFlow correctly identifies the effective TLD of all domains we observed dynamically, that is, the operator, but it missed some subdomains. For Hama, it misses four IoT endpoints that we saw dynamically, likely because the device is a rebranded IoT device. For Soundcore, it misses one dynamically generated domain pointing to the device's most recent firmware.

Beyond domains, we also compare requests' paths. It allows us to assess which approach is more promising to comprehensively understand IoT device behavior, meaning if one discovers more IoT-related functionality or if they identify distinct (overlapping) sets of behavior. Both approaches identified the same 50 IoT-related paths over all 13 apps (min 0, avg 3.85, max 17). We statically identified an additional 231 IoT-related paths (min 0, avg 17.77, max 45) and 496 general paths (min 2, avg 38.15, max 77). Dynamic analysis found 110 unique IoT-related paths (min 0, avg 8.46, max 32) and 337 general paths (min 1, avg 25.92, max 54). For three apps (Fitbit, Hue, and Wiz), our static analysis performs worse. Fitbit and Wiz use annotations to construct paths, which we cannot analyze, a limitation we share with state of the art (see Section 2.6). For Hue, our approach extracts 2 IoT-related path, while we observe 3 paths dynamically. IoTFlow performs better or equal for 10/13 apps, with a factor of at least 1.14x (Divoom, 41 vs. 36) and up to 31x (Flowercare, 31 vs. 1). For the IKEA app, dynamic analysis did not find any IoT-related paths, while IoTFlow found 45 paths.

Overall, IoTFlow performs better than dynamic analysis and extracts more IoT-related behavior statically from companion apps than dynamic analysis (54 domains and 281 paths vs. 40 domains and 160 paths) for most apps (9/13), it performs comparable for one app, and it performs slightly worse for the remaining apps (3/13).

IoTFlow Findings Taking an in-depth look into IoTFlow's security and privacy findings for the 13 apps, we find that:

- 8/13 apps send information via unencrypted HTTP to third parties, which an attacker could eavesdrop on or modify (e.g., if they are on the network path or the same wireless network). If unencrypted data is used to configure or update the device, then taking over control could be possible [246]. The NUT Find3

item tracker retrieves notifications over unencrypted HTTP, which can allow an attacker to modify a user's notification (e.g., to show that a lost item was found and where or that it has moved away).

- 5/13 apps use hard-coded symmetric encryption keys (e.g., for AES), which allows attackers to eavesdrop on their communication and can allow them to impersonate the remote end (e.g., to push configurations or updates by extracting the keys from the companion app) [245, 247].
- 2/13 apps send the hardware identifiers (IMEI) to countries outside of the EU, that is, outside of the GDPR region (one might send it to Russia and one to China to a remote endpoint that indicates tracking), using an API that is deprecated (see Section 2.4.5).
- 5/13 apps, while less critical, use country-level location information and send this to remote endpoints.
- No apps use hard-coded authentication credentials, but this does not imply that they are secure because they might not use any authentication at all.

2.6 Limitations and Future Work

IoTFlow has limitations inherent to static analyses. Additionally, we utilize the existing frameworks Soot and FlowDroid, and we inherit their limitations. For example, our resilience to obfuscation is limited, which can affect signature-based identification of sources and sinks. We find that they are only a minor share for companion apps (2.66% obfuscated). However, they are more prevalent for general-purpose apps (10.81% obfuscated) based on an APKiD [120] analysis of our datasets. Obfuscation is also an orthogonal problem, and new deobfuscation techniques can readily be adopted. Similarly, we focus our analysis on the Dalvik bytecode of apps, that is, we do not support native code. We currently do not consider code annotations, which the *retrofit* library uses to specify request paths and network methods. Both techniques are infrequently used, and not tackling them is a limitation we share with prior work, as existing frameworks struggle to support them, and the required engineering effort to support them is substantial.

Our DFA supports ICC, but our VSA does not. We plan to extend ICC support to VSA in future work. Only 2.01% of reconstructed values contain ICC data, which does not invalidate our results. Currently, we limit ICC tracking to the same app, but theoretically, ICC can cross app boundaries or come from websites via deep links, providing further avenues for collusion.

For our analysis, we limit the number of backward steps and set a timeout, which trades between precision and resources but could lead to missing values and flows. We empirically determined our thresholds and other limits could yield more precise results.

Motivated by our results on certificate pinning and abandoned domains, we aim to study how companion apps evolve over time. Naturally, identifying network endpoints, protocols, and APIs is only the first step toward truly understanding the security and privacy of device-to-cloud communication in the IoT ecosystem.

2.7 Related Work

Following, we compare IoTFlow to related work in IoT security, IoT companion app analysis, and general-purpose app analysis.

IoT Security Prior work in IoT security largely focused on identifying attacks on a small set of devices. Wood et al. [353] investigated how medical IoT devices communicate and transmit data, and they found them leaking information, like the measurement frequency, despite using encryption. Chu et al. [79] discovered kids' devices sending PII over unencrypted connections. Other work [121, 242, 76, 59, 344] focused on Samsung's SmartThings apps. SmartThings is a smart hub ecosystem unifying the control of compatible devices and allows event flow graphs, which are conceptually simple [241]. In contrast, IoTFlow analyzes arbitrary Android apps, which are more widespread and significantly more complex. Correspondingly, their techniques do not readily transfer to the entire IoT ecosystem. Related work also investigated the ecosystem via crowd-sourced network traffic collection [173] or telemetry data [198] of real-world user devices, which raises ethical and anonymization challenges.

IoT Companion App Analysis Different work investigated IoT companion apps *in combination with* physical devices [77, 216, 286, 285] to find security and privacy issues. For example, Zhou et al. [365] studied the interactions between IoT devices, cloud, and apps using state machines and found issues that can lead to device hijacking. However, they require the IoT devices, which prevents scalability. We overcome this limitation with our new static analysis approach design. Wang et al. [345] analyzed companion apps without the corresponding device. Instead of analyzing and determining *how* and *with whom* the apps communicate, as we do, they focused on identifying re-branding and propagation of known vulnerabilities. That is, they require prior domain knowledge about other devices and existing vulnerabilities. Similarly, Jin et al. [185] aimed to identify companion apps at scale, and then to identify known vulnerabilities in the apps, such as outdated library versions. Nan et al. [235] analyzed IoT apps with machine learning to detect code that handles IoT-related data, and then assessed whether the behavior was communicated to the user. Naturally, their statistical machine learning approach fundamentally differs from our static program analysis approach. Other work [371, 315, 314, 363] aims to find BLE issues in mobile apps. IoTFlow is more general as we investigate communication beyond BLE, thus obtaining a better understanding of a greater part of the IoT ecosystem.

General-purpose App Analysis Understanding general-purpose mobile apps has seen significant work. Some approaches use dynamic analysis to run apps in controlled environments to observe their (network) behavior and endpoints [280, 268, 208, 89, 287]. As we observed, the provided inputs impact the analysis, which is an ongoing research challenge [78, 165]. Moreover, to adopt these approaches, one would need the actual IoT devices, making large-scale analysis infeasible. Several approaches extract information about apps through static analysis techniques, like

VSA or flow analysis, such as network endpoints, API keys, protocol commands, etc. Gadient et al. [128] extract URLs and JSON schemas to study HTTP(S) usage, private APIs, and code injection vulnerabilities. Extractocol [191] reconstructs HTTP requests based on data flow analysis for automated protocol analysis, but does not scale. Stringoid [278] simulates string concatenations, but cannot reconstruct URLs built in other ways, such as *okhttp3.HttpUrl.Builder*. Leakscope [370] reconstructs API keys in mobile apps. Zuo et al. [371] reconstructed BLE UUIDs to identify vulnerable implementations of its pairing process. Wen et al. [351] reconstructed Controller Area Network (CAN) bus commands.

2.8 Conclusion

We introduced IoTFlow, a new technique for the large-scale security and privacy analysis of IoT devices through their companion apps. With Value Set Analysis (VSA), we extract network endpoints and protocols, which enables us to characterize IoT device behavior for local app-to-device communication and remote communication with cloud backends without requiring the physical IoT device. By cleverly combining VSA with Data-flow Analysis (DFA), we trace data flows from IoT devices and sensitive Android methods to understand better what data companion apps share and how. Leveraging IoTFlow, we analyzed 9,889 companion apps and 947 general-purpose apps. We identified striking differences between the two types of apps and discovered various security and privacy problems in the IoT ecosystem, such as abandoned domains, hard-coded credentials, expired certificates, or use of broken encryption algorithms. Our approach shows clear promise for identifying security and privacy issues of IoT devices at scale and it could be used to generate privacy labels or verify claimed behavior automatically.

3 Analyzing the iOS Local Network Permission from a Technical and User Perspective

Abstract

In the past, malicious apps attacked routers or identified locations through local network communication. To mitigate security and privacy risks from local network access, Apple introduced a new permission with iOS 14. To be effective, the permission needs to protect against technical threats, and users must be able to make an informed permission decision. The latter is presumably hindered by the intrinsic technicality of the concept of the local network.

In this paper, we perform the first comprehensive analysis of the local network permission by studying four key aspects. We investigate the security of its implementation by systematically accessing the local network. We explore local network accesses via a large-scale dynamic analysis of 10,862 iOS and Android apps. We analyze the concepts that constitute the permission prompts, as this is all the information users get before making a decision. Based on the identified concepts, we conduct an online survey ($N = 150$) to comprehend users' understanding of the permission, their threat awareness, and common misconceptions.

Our work reveals two methods to bypass the permission from webviews, and that the protected local network addresses are insufficient. We show how and when apps access the local network, and how the situation differs between iOS and Android. Finally, we present the light and shadow of users' understanding of the permission. While nearly every participant is aware of at least one threat (83.11%), misconceptions are even more common (84.46%).

Publication The work presented in this Chapter resulted from a collaboration with Alexander Ponticello, Magdalena Steinböck, Katharina Krombholz, and Martina Lindorfer. The paper has been published at the IEEE Symposium on Security and Privacy (S&P), 2025 [297]. I developed the app for testing the permission, conducted the dynamic analysis, and implemented and performed the data evaluation of the test app, the large-scale analysis, and the user study. Alexander Ponticello designed and evaluated the rationale analysis. Magdalena Steinböck and I supported him in coding the rationales. Alexander Ponticello also designed and conducted the user study. Katharina Krombholz and Martina Lindorfer provided feedback and helped revise the paper.

3.1 Introduction

With the release of iOS 14 in 2020, Apple introduced a new permission to guard access to the local network [32]. Usually, devices connected to a local network cannot be directly accessed from the Internet. However, if apps can access a local network, they can communicate with these devices, opening new attack vectors. Furthermore, collecting information of a person's local network can enable tracking and user profiling [135, 283]. After the permission became widely used, a discussion arose among users as to why certain apps requested such access. Popular apps like AliExpress and Instagram were suspected of scanning the network and collecting data to track users [94, 215].

Prior research already identified threats induced by local network access. Reardon et al. [283] uncovered Android apps that obtained the MAC address of WiFi routers through local network communication to bypass the location permission. Girish et al. [135] demonstrated that household fingerprinting and user tracking are possible via local multicasts. Furthermore, a malicious Android app was found reconfiguring the DNS settings of routers to forward traffic to malicious domains [190]. Finally, Kuchhal and Li [197] explored local network access of websites with desktop browsers and observed legitimate use cases such as fraud and bot detection, but warned of the potential misuse of this technique for tracking and profiling.

Prior research has not yet investigated the situation when a permission protects local network access nor the iOS permission itself. Noteworthy, this permission is distinct from others: (1) It affects not only the device running the app but also its surroundings, i.e., all devices connected to the network. (2) Android, as of version 15 (released in 2024), does not have a corresponding permission that restricts accessing the local network. (3) Compared to other permissions, like location or contact access, it is presumably less obvious to users what the local network comprises and why information about it is sensitive. (4) Developers cannot directly request the permission from the app's code. Instead, the Operating System (OS) detects access and prompts the user to grant the permission [44].

The implementation of the local network permissions is also yet to be studied. Users raised concerns that YouTube was bypassing the local network permission [334]. In this case, the access was caused by Apple's AirPlay functionality, which does not require permission. However, unauthorized access might be possible. Further, it is unclear how prevalent local network access is and if apps behave differently on Android, where no such permission exists.

To fill these gaps, we address this topic with an interdisciplinary approach by looking at both the technical and user perspective. We contribute a comprehensive assessment of the security and privacy of this functionality, how apps access the local network, how they prompt users for it, and the users' understanding of the permission. It is vital to study both perspectives. The technical aspects give us insights into potential misuse of the permission and how widespread local network access is in apps. The user perspective helps us understand whether the permission dialog enables users to make informed decisions. Furthermore, it shows threat awareness related to local network access. Both technical and user aspects need to be effective, as none can achieve sufficient protection without the other. If the permission is not implemented securely, there is a false sense of security and privacy.

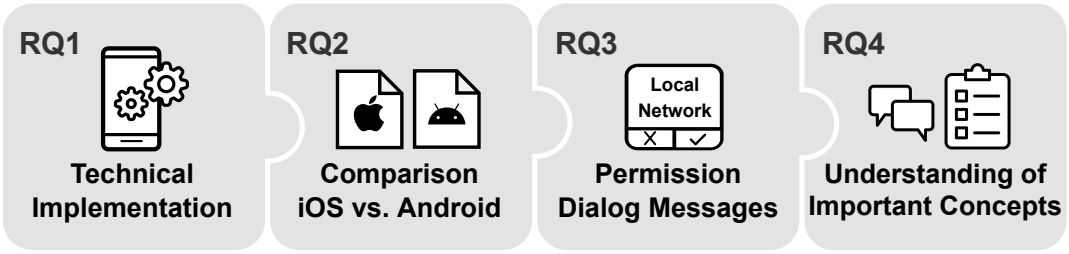


Figure 3.1: Overview of analysis to answer our research questions. (RQ1) We study in Section 3.3 if the local network permission is implemented securely. (RQ2) In Section 3.4, we show how prevalent local network access is. (RQ3) Section 3.5 presents our content analysis of the permission rationales. (RQ4) To investigate the users' perspective, we performed a user study as described in Section 3.6.

Likewise, if users are not able to make an informed decision, malicious apps could trick users into granting the permission. Thus, as shown in the overview in Figure 3.1, we investigate the technical and user perspective in four consecutive analysis steps.

First, we investigate whether the implementation of the permission is secure or if it is possible to bypass it. We systematically access the local network to find potential gaps in its enforcement to answer *RQ1: Is the local network permission implemented securely?* Next, we inspect how widespread local network access is in apps and study how the situation differs between iOS and Android by performing a large-scale analysis of apps available on both platforms. We dynamically analyze 10,862 apps present on both platforms to answer *RQ2: How prevalent is local network access in apps?* We further investigate the information of the permission prompts, where developers can provide a rationale explaining why access is requested. We extract these messages and analyze their contents to answer *RQ3: Which concepts constitute the permission prompts?* Finally, we perform an online user study with 150 participants to shed light on the users' understanding of the local network permission and their awareness of threats in this context to ultimately answer *RQ4: What is the user's understanding of these concepts?*

In summary, we make the following contributions:

- We demonstrate two methods to bypass the permission and show that the protected local IP address range of the permission is insufficient.
- We analyze 10,862 cross-platform apps, out of which 152 iOS and 117 Android apps access the local network, and show differences between both platforms.
- We identify reoccurring concepts in permission prompts and present insights into the developer-specified purposes.
- We show that nearly every participant (83.11%) is aware of at least one threat but also that misconceptions are widespread, as 84.46% hold at least one.

Artifacts. For reproducibility and to enable future work, we publish our code to study the permission, analyze apps, extract and label permission messages, and evaluate the results at: https://github.com/SecPriv/local_network.

3.2 Background and Motivation

In this section, we give an overview of what we consider as a local network, provide threats posed by apps that can access the local network, and discuss the permission on iOS protecting the local network access.

3.2.1 Local Network

We consider the WiFi network to which a mobile phone is connected to as the local network. This is not restricted to users' home network but can also be, e.g., a company network or the WiFi of a café.

Accessible Devices. Connected devices, like the phone itself, have an IP address assigned within the subnet mask of the network. However, depending on the network setup, devices with local IP addresses outside of the subnet mask may also be reachable. Hence, threats that affect devices within the subnet mask also apply to locally reachable devices outside the subnet mask. Therefore, we consider them as part of the local network.

Virtual Private Networks (VPNs). In Section 3.3, we also consider VPNs connected to the phone as similar to a local network. Technically, VPNs are not a local network, but the security and privacy threats we consider also apply to devices within VPNs. Devices within local networks and VPNs are typically not reachable from the Internet. However, they are still accessible from other devices within the network. For VPNs, threats can even have worse consequences, as companies use VPNs for internal services.

3.2.2 Threat Model

Security. Devices connected to a local network are usually not directly accessible from the Internet as a firewall protects them. However, they are still accessible on the local network. Thus, malicious apps running on a user's phone can try to attack other devices on the local network [316, 190]. This is in particular a concern in smart homes, as IoT devices are often configured with default passwords and rarely updated, a fact heavily exploited by IoT malware [4, 9].

Location. Local network access also brings threats to the user's privacy. Apps can use the router's MAC address to look up location information in online databases [233, 291]. In the past, apps used that to bypass the location permission [283, 282]. If an app observes the devices in the user's local network, it can infer location changes and behavior patterns since local networks differ in terms of used IP addresses and connected devices.

Advertisements. Knowledge about connected devices can be relevant for ads agencies. They could decide which ads to show based on the devices on the local network. In online discussions, users raised concerns that AliExpress might use local network information for this purpose [94]. Also, users might not want to share which devices they own with every app, e.g., health-related devices with insurance apps. We have seen that websites use information about users' devices to show different prices, e.g., charging Apple users more, so-called personal pricing [134]. With local

network access, other devices connected to the local network might also influence the prices, making it harder to bypass. Cross-device and cross-user tracking is another privacy threat, as local network information gathered on different devices connected to the same network can be used to link users [135]. Device names of connected devices can be sensitive as well, e.g., if users use their real name [195].

Misconceptions. In a congress hearing, the TikTok CEO was asked if TikTok accesses other devices on the WiFi network. Online users argued that the questioner did not understand how WiFi networks work as the app needs it to access the Internet. This shows that there is a lack of understanding of the local network with its security and privacy implications [166, 312]. If this misconception is widespread, malicious apps could exploit it to trick users into granting permission. For example, apps could tell their users that they require local network access to download large files via WiFi, and if they do not grant it, it could lead to additional costs when downloaded via cellular Internet.

3.2.3 Permission Overview

Terminology. Android restricts apps from accessing specific data or performing actions through permissions. There exist two major types of permissions: (1) Install-time permissions, granted by the system upon app installation if apps declare them [14], and (2) runtime permissions, also called dangerous permissions, which require user confirmation [14].

There are similar concepts on iOS. Apps can declare *entitlements* to gain additional capabilities, which iOS grants upon installation [35]. However, if apps access so-called protected resources, they require user consent [47]. For handling the consent request and keeping track of the user decisions, the Transparency, Consent, and Control (TCC) framework is responsible [159, 57]. In the following, we use the term permissions for actions that require user consent. Otherwise, we state it explicitly.

Local Network Permission on iOS. According to Apple, all outgoing traffic to a local network, multicast, and broadcast addresses requires the local network permission [43]. Technically, iOS does the permission check at the lowest system level, and thus, it should include all network APIs. Apple mentions that they plan to extend the permission to incoming multicast and broadcast traffic [43], but have not yet implemented it as of iOS 18 (released in 2024).

An exception to the permission are Bonjour services provided by the OS for AirPlay and printing. Bonjour is a zero-configuration protocol that allows the discovery of devices and services on the local network. Apple argues that those methods do not reveal any network details to the apps; Thus, they do not require the permission [43]. AirPlay is integrated into different parts of iOS to offer functionalities to stream audio or video to other devices on the local network, e.g., the iOS media player and webviews send such requests when playing a video [38]. However, this does not trigger the permission, as the local network information is not provided to the app.

Compared to other iOS permissions, the process to request the local network access permission differs, which could influence the user's understanding. App developers cannot request permission directly. Instead, the OS asks the user to grant permission if an app tries to access the local network for the first time [48]. All further attempts

to access the local network do not trigger the permission prompt. Users can only change their decision in the settings or have to reinstall the app [42].

To help users understand why apps require the local network permission, app developers can add descriptions to the permission request. If developers do not provide a so-called permission rationale (also named permission purpose string), iOS shows the default message "*This app will be able to discover and connect to devices on the networks you use.*" along a description always provided by the system "*[App name] would like to find and connect to devices on your local network.*"

Lack of Permission on Android. Android, as of version 15 (released in 2024), does not have a comparable local network access permission. While the `CHANGE_WIFI_MULTICAST_STATE` [13] permission exists that allows apps to receive multicast messages, Android ranked its protection level as `"normal"`; thus, it is an install-time permission and does not require user confirmation [14]. Moreover, apps can still send messages to the local network without holding this permission, as it only restricts apps from receiving multicast messages [16].

3.3 Permission Implementation

First, we test when iOS enforces the local network permission to gain further insights into its implementation and answer *RQ1: Is the local network permission implemented securely?* To do so, we systematically try to access different addresses through various methods.

3.3.1 Methodology

To gain insights when iOS enforces the local network permission, we developed a test app. The app tries to contact different broadcast, local network, and multicast addresses. Additionally, we varied the methods to access the network to see if the permission is enforced for all network APIs [43].

We installed the test app on an iPhone 8 running iOS 16 (released in 2022) jailbroken with palera1n [259]. We used tcpdump [329] on the phone to capture its traffic. Moreover, we connected the phone to a VPN to test the protection of the VPN address space for which the same security threats apply. To ensure the jailbreak did not influence our findings and that they still exist in later versions, we retested them manually on an iPhone SE 2020 that was not jailbroken running iOS 17.3 (released in 2024).

We tried to cover as many methods to perform local network requests as possible. Therefore, we consulted Apple's developer documentation [41]. In total, we identified 16 different methods that allowed us to perform ICMP, TCP, UDP, and QUIC requests. In addition to the explicitly mentioned protocols in the Frequently Asked Questions (FAQ) [43] (TCP and UDP), we included ICMP to test a different Internet layer and QUIC for another transport layer protocol. For each method, we attempted to contact different addresses: local addresses within and outside the connected WiFi network mask, IPv4 and IPv6 multicast addresses, IPv4 broadcast addresses, and local IPv6 addresses. In Section 3.10.1, we provide an overview of tested addresses, and in Table 3.1 the methods we used.

Table 3.1: Results of tested methods. ✖ indicates that the app sent the request without the permission, thus bypassing it. ━ indicates that Apple intended that the functionality did not require permission even if it accessed the local network. ✔ indicates that iOS correctly enforced the permission. No symbol means that sending such a request was impossible. *Local* means the local address was within the connected WiFi's subnet, and *local outside* outside of it.

Protocol	Methods	Local	Local outside	Multi-cast	Broad-cast
ICMP	SimplePing [50]	✔	✖		
QUIC	NWConnection	✔	✖	✔	✔
TCP	NSURLConnection	✔	✖		
	NWConnection	✔	✖		
	SendTo	✔	✖		
	SFSafariViewController	✖	✖		
	UIWebview	✔	✖		
	URLSession	✔	✖		
	WKWebView	✖	✖		
UDP	AirPlay			━	
	AirPrint			━	
	NetServiceBrowser			✔	
	NSBonjourServices			✔	
	NWConnection	✔	✖	✔	✔
	SendTo	✔	✖	✔	✔
	ServiceBrowser			✔	

We included local IPv6 addresses in our tests despite the address space of 2^{128}. On the one hand, scanning the whole address space by contacting each address is not practical. On the other hand, if an app learns the addresses of connected devices, they are well suited for user tracking, as different networks likely use different combinations of addresses. IPv6 could be especially suited for tracking if hosts generate their addresses using stateless autoconfiguration based on MAC addresses [332]. On local IPv6 networks, multicasts and neighbor discovery exist to find devices. However, we could not test sending neighbor discovery messages, as iOS restricts app access to raw sockets [46].

To identify the tested method in the traffic dump, we performed a GET request to a remote server under our control, containing a test ID in the URL path before and after each test case. We executed our test app twice, once granting permission to ensure the app sends the request and once denying it to observe if it is enforced.

3.3.2 Results

Based on our systematic tests to trigger the local network permission, we not only found two methods that bypass it (Section 3.3.2), but also that the protected address

space is insufficient (Section 3.3.2), as well as a lack of rationales (Section 3.3.2). Table 3.1 summarizes our results.

Permission Bypass

We discovered that iOS does not correctly enforce the local network permission for two methods that allow including web content in apps (so-called webviews). Compared to Android webviews, for which security and privacy aspects have been well studied [66, 65], iOS offers different types of webview components with different levels of embedding within the host app.

SFSafariViewController. The SFSafariViewController [49] is part of the SafariServices framework and is similar to a Safari window within the app. Apple advises using it to visit external websites [33]. The opened website cannot directly send data back to the app or execute code from the app. However, the website loaded within those components could scan the local network, retrieve privacy-sensitive data by communicating with connected devices or attack connected devices [164, 1]. Thus, the same threat model also applies to websites and their running code [197].

WKWebView. WebViews allow embedding web content directly into an app and can interact with the app [33]. In iOS 8 (released in 2014), Apple added the WKWebView [53], to replace the older UIWebView [52]. The UIWebView has been deprecated since iOS 12 (released in 2016). We found that iOS correctly enforces the permission for the deprecated UIWebView, in contrast to the recommended WKWebView. With WKWebViews, apps can load their own JavaScript code from the app and send the retrieved information back to the app. For example, they could try to load web content from devices connected to the local network; in the case of the Philips Hue bridge, this reveals its device type, and, if the path `/description.xml` is also accessed, its model and serial number. The behavior directly contradicts Apple's developer FAQ, which states that the permission is required for all outgoing traffic, including WKWebView [43].

Case Study: Browsers. Since Apple required third-party web browsers to build upon WebKit [268], they also used WKWebView to render web content. Thus, they can bypass the permission. With Google Chrome [152], Opera [253], Microsoft Edge [222], and Brave [73], it is possible to observe the WKWebView permission bypass. To load the web content in the webview, they send a GET request to `/favicon.ico` to show an icon in the header bar if a user visits a webpage within the local network. iOS loads the page before the user allows it, since the web content loaded with the WKWebView bypasses the permission. The icon request is done outside the WKWebView and triggers the permission dialogue. Other browsers like Safari [39] and Firefox [232] bypass the permission too. However, as they do not perform a request outside the webview, they do not trigger the permission dialogue.

Insufficient Protection

The permission is also insufficient for complex networks and VPNs. We observed that apps can send messages to local addresses outside the connected WiFi's subnet mask without the permission.

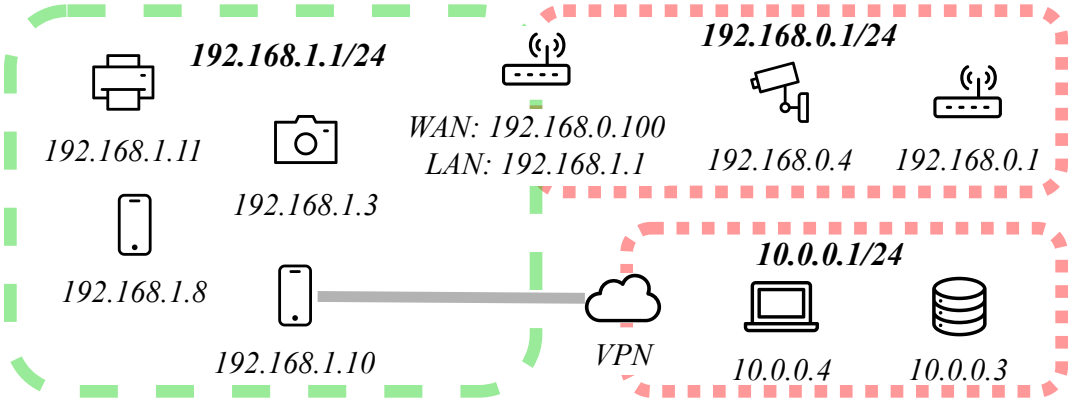


Figure 3.2: Vulnerable local network example. The router of the local network, with the address 192.168.1.1, forwards packets to the 192.168.0.1/24 network. The iPhone with the address 192.168.1.10 is also connected to a VPN. We colored the network protected by the permission green (dashed) and the addresses not protected red (dotted).

Figure 3.2 shows an example of a vulnerable network: The iPhone running the app connects to the WiFi network `192.168.1.1/24`. The router providing access to the iPhone is also part of a different local network, `192.168.0.1/24`, and forwards traffic from `192.168.1.1/24`. Since the permission only protects the `192.168.1.1/24` address space, apps can access devices on `192.168.0.1/24`, for which the same threats apply. Additionally, if the iPhone connects to a VPN, the address space of the VPN is also not protected by the permission. Even if a VPN is technically not a local network, the same security and privacy threat apply: Devices not reachable from the Internet become accessible.

In general, the permission protects access to addresses within the subnet mask of the connected WiFi. That means the protected addresses might not be enough in a complex network setup. In Section 3.4.3, we show that 23 iOS apps tried to access local IP addresses outside the connected network, highlighting the relevance of this finding.

Local Network Rationales

According to Apple's developer documentation, apps must provide a rationale string (also called usage description) to explain why they want to access a protected resource, and if they do not have it, iOS blocks the app's access [47]. We did not observe this behavior regarding the local network, even if it is a protected resource [37]. Apps can acquire local network permission without a rationale. In the documentation of the local network rationales, Apple weakens the requirement by writing that rationales *should* be included when an app accesses the local network. That might be due to the difference in how to acquire the permission [37]. Other permissions restrict apps from calling sensitive methods. In contrast, for the local network permission, there is no sensitive method per se, but the OS detects if an app tries to contact the local network.

Bonjour methods are the most similar local network related methods to other permission-protected methods. We tested these with our test app, as they rely on multicast messages. Apps need to specify Bonjour services in the `Info.plist` with the key `NSBonjourServices` to use those methods [45]. Bonjour service strings can be seen as the name of the service to discover. Surprisingly, iOS also does not enforce a local network rationale string for them.

Responsilbe Disclosure

After our disclosure, Apple ensured us that they are working on fixing the bypass via web components and the insufficient protection in complex networks (reported in June 2022), but we have not been updated about a solution (October 2024). For VPNs (reported in April 2024), Apple did not share our point of view and categorized it as *expected behavior* in August 2024.

Takeaways

To answer RQ1: *Is the local network permission implemented securely?*, we showed:

- The iOS local network permission is not implemented securely. Apps can bypass it using SFSafariViewController and WKWebView.
- The protected address space is insufficient for complex local networks and VPNs.

3.4 Permission Prevalence

After understanding the local network permission, we study the current situation of local network access of apps. Therefore, we dynamically analyze 10,862 cross-platform apps each on iOS and Android to answer RQ2: *How prevalent is local network access in apps?* In addition to iOS apps, we analyze the same apps on Android to compare their local network access behavior, which we suspect to differ due to the lack of comparable permission on Android.

3.4.1 Dataset

To study the prevalence of the local network permission in apps, we analyze iOS and Android apps. For better comparability, we focused on apps that are available on both platforms, so-called cross-platform apps, by using the approach from Steinböck et al. [320], which identifies matching app pairs within iOS and Android datasets by comparing metadata like app names, and app descriptions. Additionally, they compare to the matches from Google's migration API, which we used to find further app pairs.

For iOS, we tried downloading the 10,000 most popular apps based on user ratings and 10,000 random apps, resulting in 19,847 successfully downloaded iOS apps. Our initial dataset for Android consists of 19,333 apps, where we attempted to download the 9,629 most popular apps based on the number of downloads (at least 10 million

installations) and 10,000 randomly selected apps. To find out what apps exist, we used the Androzoo dataset [7]. We downloaded both popular and randomly chosen apps to get a holistic representation of what users are confronted with in their daily lives, as well as potential edge cases. We could not download all apps because of geographical blocking by developers or device constraints [199].

Using the above-mentioned matching approach [320], we identified 11,405 cross-platform app pairs: 3,047 apps were already within our sample datasets, further 8,358 matching Android apps the migration API identified and we successfully downloaded 7,815 of them. Hence, our final dataset consists of 10,862 app pairs.

3.4.2 Methodology

We decided to use dynamic analysis to study the local network access of apps. Compared to other permissions, it is not enough to analyze if specific methods are used, as there is no predefined list of methods that require this permission (see Section 3.2.3). Apps could use various network functionalities to communicate with the local network, which can be found in nearly every app. Also, there is a wide range of possibilities to obtain or use local network addresses, e.g., if apps want to obfuscate it, they can use numerical values. Since we aim to compare the behavior of iOS and Android apps, and the static detection of local network access is challenging, we opted for an approach that works on both platforms. By dynamically executing apps and observing their network traffic, we can detect access for iOS and Android apps using the same methodology. This approach is similar to how iOS triggers the permission prompt, i.e., by checking the destination of outgoing network traffic. For iOS apps, we further statically extracted the local network permission rationales and the Bonjour strings required for Bonjour multicast methods to study their usage. Additionally, we used the extracted rationales in Section 3.5 to analyze the concepts provided by developers.

For the dynamic analysis, we set up a test infrastructure to run experiments and automatically interact with apps while capturing traffic to detect local network accesses.

Test Infrastructure

Our test network consisted of a router (Asus AC750), four IoT devices, mobile phones, and a Mac mini (M1, 2020) running macOS 12.5. We connected four IoT devices (Google Chromecast [145], Amazon Alexa speaker [10], Philips Hue [265], and Ikea Trådfri [178]) to observe if apps try to interact with them. We used the Mac mini to manage the interaction with the apps.

To capture the traffic and detect local network access, we ran tcpdump [329] on the phones. We jailbroke four iPhones running iOS 16 (released in 2022) using palera1n [259] and rooted three Pixel 6a running Android 13 (released in 2022) with Magisk [138]. If we found any local network access, we performed a second test round to ensure it originated from the app and that it was no background traffic from the OS. We only considered apps accessing the local network if they did so consistently.

Automatic App Interaction

Previous work [283, 89] used Android monkey [26] for dynamic testing to randomly interact with an app. However, as discussed extensively by related work [78, 288, 287, 346], this might not be sufficient and may not cover relevant code. Therefore, we use a systematic interaction approach that clicks through the User Interface (UI) in a Depth-first search (DFS) manner.

We first ran each app for 30 seconds without interaction, after which we performed 25 interactions to obtain more context of the local network access. An app accessing the local network after an interaction might need it to provide functionality, e.g., to send a command to an IoT device or find other players in a game. Accessing the local network on startup is more suspicious. According to Android's developer documentation [15], it is a bad privacy practice to immediately collect privacy-sensitive data, even if apps potentially require the information later on.

We implemented our app interaction in Python and used Appium [31] to locate and interact with elements on the phone's screen. We automatically performed 25 interaction steps with each app; a step is one action, e.g., a button pressed. We chose 25 steps as a tradeoff between execution time and coverage. The interactions followed a DFS exploration strategy. The interactor saved information about what it had already visited and the interactions it performed. If an interaction led to a new state, the interactor continued there. If it had tried all available interactions, it attempted to go back to the previous state. A state consisted of the currently available UI elements.

We accepted all iOS permissions the app asked for, not only in the interaction phase but also during the non-interaction phase, as we cannot observe local network access without consenting to the iOS permission. We did not consider the interaction with system dialogues as an interaction step, as it is not a direct interaction with the app. This is in line with our approach on Android, for which we granted all permissions using `adb` [12] before starting an app.

Rationales and Bonjour Strings

By putting the keyword `NSLocalNetworkUsageDescription` to the `Info.plist` and the localization files called `InfoPlist.strings`, developers can add their custom permission rationale messages. Also, apps declare the Bonjour service strings required for Bonjour methods in the property list file (see Section 3.3.2). We developed a Python script using the plistlib library [133] and regular expressions to extract these values from the property list. In Section 3.4.3, we use it to study their occurrences. In Section 3.5, we further analyze the concepts used in the permission rationales.

3.4.3 Results

We evaluate different aspects of local network access. We split them up based on the type of access: (1) broadcast, (2) directly contacting a local address, and (3) multicast. We analyze when the access happens, directly after the start or after user interaction, and show what local network addresses outside the connected WiFi apps try to contact.

Local Network Accesses

We found local network accesses in 152 (1.4%) iOS and 117 (1.08%) Android apps. In Table 3.2, we provide an overview of accesses.

Access Types. The reason why we found more iOS apps accessing the local network is because of multicast messages. We found them in 117 iOS and 73 Android apps. In contrast, we found more Android apps contacting broadcast and local addresses. We encountered 44 iOS and 53 Android apps using broadcasts, while 18 iOS and 24 Android apps directly contacted a local address. A reason why we found more iOS apps using multicast could be the support for Bonjour multicasts. Evaluating the multicast types based on the addresses, we found the multicast DNS (mDNS) address `224.0.0.251` that Bonjour uses in 98 (83.76% of apps using multicasts) iOS apps, while we found it in 50 (68.49%) Android apps. The multicast address we observed the most for Android apps is `239.255.255.250`, which is related to Simple Service Discovery Protocol (SSDP). We found it in 38 (32.48%) iOS and 56 (76.71%) Android apps.

We consider AirPlay as local network access of the OS and not the app. As mentioned in Section 3.2.3, it does not trigger a permission request as no sensitive data is accessible by the app. Further, we found that playing audio or video triggered iOS to send AirPlay mDNS requests in the background. We also observed similar behavior in Android, which sent mDNS requests to find Google cast devices. Therefore, we do not treat this as app traffic but OS related. Consequently, we did not include it in the local network accesses. Overall, we observed this behavior in 401 (3.69%) iOS and 308 (2.84%) Android apps.

On iOS, we found 14 apps that triggered the permission, but we could not observe any local network accesses in the traffic. We manually analyzed them and found that two apps tried to contact their own WiFi IP address. In that case, the OS does not forward it to the WiFi interface but only to the loopback interface. Apps might do this to trigger the permission request, as done by Apple's example code [51]. One app is the dictionary app Linguee, which runs a local server and communicates over the local WiFi IP address, triggering the permission. This could hint at a security issue, as Tang et al. [326] and Wu et al. [354] showed that apps can be vulnerable if they run wrongly configured servers accessible from other devices. For neither of the other 12 apps, we could find local network access in the traffic dump. While we found reports of requests to remote endpoints with port 10161 triggering the permission [203], we could not reproduce it on a current iOS version. We do not include those 14 apps as accesses to the local network as we could not observe any local network traffic, and the threats we analyze in this paper all require access to other devices on the local network.

Access Phases. We further analyzed when apps access the local network. More Android apps (75.21%) than iOS (65.13%) already accessed the local network before any user interaction. The iOS permission could have influenced those differences since it makes it transparent when apps access the local network for the first time. We found 70 apps that access it on both platforms: 13 Android apps (18.57%) access it before any interaction, while the corresponding iOS apps only do so after an interaction. In contrast, five iOS apps (7.14%) access the local network upon opening, while their Android counterparts do so only after an interaction. 42 apps

Table 3.2: Number of apps that accessed the local network. The *Local* column indicates apps that sent a message to another local address, while *Broadcast* and *Multicast* show apps that sent respective messages. The *All* column provides the apps that used any of these methods. The numbers in the *Total* rows are relative to the dataset size. The shows apps that accessed the local network without any interaction, while those in only did so after an interaction. The percentages refer to their total number of accesses.

		All	Broadcast	Local	Multicast
	Total	152 (1.40%)	44 (0.41%)	18 (0.17%)	117 (1.08%)
		99 (65.13%)	22 (50.00%)	7 (38.89%)	84 (71.79%)
		53 (34.87%)	22 (50.00%)	11 (61.11%)	33 (28.21%)
	Total	117 (1.08%)	53 (0.49%)	24 (0.22%)	73 (0.67%)
		88 (75.21%)	34 (64.15%)	16 (66.67%)	63 (86.30%)
		29 (24.79%)	19 (35.85%)	8 (33.33%)	10 (13.70%)

(60%) access it on both platforms without interaction, and 10 apps (14.29%) only after an interaction.

Case Studies. We manually classified the apps that access the local network. Overall, most apps are companion apps for IoT devices (112, 56.28%), followed by video apps (17, 8.54%), and apps for events (16, 8.04%). We provide further details in Section 3.10.2. We found apps from other categories less often, e.g., two shopping apps, a dating app, and a crypto wallet on Android access the local network. As we could not think of any use cases for them, we manually analyzed them. We observed local network access only in the Android version of all four apps and they all have Alibaba libraries in common. The apps perform an ARP scan of the connected WiFi after the app launches the first time. While they scan the network when the user opens them the first time, we did not observe this behavior consistently afterward. The code responsible for scanning is heavily obfuscated with encryption and reflection. We could not find any local network information in the traffic. Online users suspected AliExpress uses it for advertisement [94].

The TanTan dating app's privacy policy is the only one that relates to accessing the local network. They mention collecting "*device network information*," and information about the "*device environment*" [327]. We translated the policy using Google Translate, as the original page is Chinese. The fact that we did not find their counterpart iOS apps accessing the local network could be due to the permission, which makes the first access transparent, as there were also reports of the iOS AliExpress app accessing the local network in the past [94].

We performed another case study on the Among Us game app. It uses UPnP to find other players. The app searches for other players immediately after opening on both platforms. If it finds any UPnP device in the network, it retrieves more information about it. In our test network, that were the Philips Hue bridge and Google Chromecast, which leak sensitive information. For example, the Philips

Hue bridge returns the model name, model number, and serial number. All this information could be used by apps for ads and tracking.

Rationales and Bonjour Services. In our dataset, we found 727 (6.69%) apps with permission rationales, and 586 (5.39%) that declared a Bonjour string. Of those, 69 (11.77%) apps did not declare a rationale. Of all the iOS apps we found triggering the local network permission, 98 (61.25% of triggered permissions) apps declared a permission rationale, and 62 (38.75%) apps did not.

We discovered 27 apps that sent Bonjour mDNS messages but did not declare a Bonjour string, contradicting the documentation. We manually analyzed them and found that 8 (29.63%) apps were published before the string was required or targeted an older iOS version. We assume iOS does not enforce it for them for backward compatibility. 17 (62.96%) apps used a library for Google Cast that did not use any built-in Bonjour method. Instead, they relied on a custom implementation. iOS checks the Bonjour string only for the system methods. Thus, those apps bypassed this check. One app (3.7%) tried to communicate with a local domain name (domain name ending with `.local`). iOS internally uses Bonjour to find the IP addresses of local domains, but since the app did not use the Bonjour method, it also did not require the string. Lastly, the Philips Hue [311] app (3.7%), an IoT companion app, used a system method that requires the string but did not declare it. The app received updates frequently and targeted a current iOS version. We were not able to reproduce this with a test app and reported our findings to Apple in April 2024, who categorized it as *expected behavior* in August 2024.

Local Addresses Outside the Subnet

We found a similar number of iOS and Android apps that trying to contact local network addresses outside the connected local subnet. In total, we found 23 (0.21%) iOS and 22 (0.2%) Android apps exhibiting such a behavior. Nine apps did it across the platforms, 14 apps only on iOS, and 13 apps only on Android.

We found different reasons for this behavior: There were IoT companion apps that tried to contact their corresponding devices at a specific hard-coded address [305]. For example, the Android smart camera apps 4K CAM and Uniden 720 Link tried to reach their devices at `192.168.1.1` if the user did not input any address. Even though both apps were from different developers, they shared the same code for accessing the address, which could be a sign of a rebranded device. In other apps, we consider the contacted addresses to be development artifacts or bugs. For example, we observed that the Android app Door Entry CLASSE300X, belonging to a smart door lock and surveillance system, performed an ARP scan of a `/23` network space, even if connected to a `/24` network. The scan of the wrong network space happened because of a development bug when calculating the network mask. While the Android app scanned the local network after launch, we did not observe the corresponding iOS app accessing the local network as it did not access the local network before a user logged in, which could be due to the permission.

Takeaways

To answer RQ2: *How prevalent is the local network permission?*, we showed:

- 117 iOS and 152 Android apps in our dataset access the local network.
- Over half of the apps access the local network on start. iOS apps do this less often than Android apps, which could be an effect of the permission.
- Apps try to contact local addresses outside the connected subnet, highlighting the relevance of our finding regarding the limited protection of local network addresses in Section 3.3.

3.5 Permission Rationales

After understanding the technical aspects of the permission and the local network access of apps, we investigate users' ability to make informed decisions about the local network permission. When an app requests it, users are presented with a rationale composed of a system-generated description and a message that developers can set to explain why the app needs this permission. A default message is shown if developers do not provide a custom one. These messages are mostly all the information users get before deciding whether to grant the permission or not [70]. Investigating the content of these messages helps us understand what information goes into a user's decision-making process. Thus, we perform a content analysis on rationales to answer *RQ3: which concepts constitute the permission rationales?*

3.5.1 Methodology

As a first step, we automatically extracted rationales from iOS apps in our dataset, as already described in Section 3.4.2. We filtered these strings to only include the first rationale in German and English, as these are the languages in which all involved researchers are proficient. Furthermore, we used the Google Translate library [162] to translate the default language used by the app to English, leaving us with up to three rationales.

Next, three researchers independently coded a sample of 100 rationales. We aimed to identify the concepts rationales contain that we deem crucial for users' understanding to make an informed decision. One researcher coded all 100, while the other two coded 50 rationales each, meaning that two researchers coded each message. After this initial coding round, all researchers compared their codes, merged codebooks, and discussed potential disagreements.

All involved researchers agreed that most rationales follow a simple structure, often using similar compositions. Also, all identified concepts are explicitly contained in the message, meaning that we can use a keyword-based approach to code our dataset [254]. Hence, we developed a keyword list for each code in our codebook. If a rationale contained one keyword from the list, we automatically assigned the corresponding code. Thus, one rationale can have multiple codes assigned. For example, we labeled the rationale "*The app uses the local network to set up and control your connected devices.*" with the codes *local network*, and *device interaction*,

Table 3.3: Our codebook to categorize the rationales. We assigned a code if we found one of the keywords in a rationale. The column *# Apps* shows the number of rationales with the code, related to the total 727 apps with rationales.

Code	**Keywords**	**# Apps**
device interaction	device, devices, camera, cameras, gopro, speaker, speakers, tv, tvs, product, pc, pcs, iphone, ipad, server, servers, hn-grynsite, computer, macos, car, vehicle, smart home, fritzbox, receiver, razor-link, printer, printers	539 (74.14%)
local network	local network, networks you use, lan, local area network	524 (72.08%)
discovery	discover, detecting, detect, discovering, search, find, finding, searching, scan, scanning, identify, identifying	371 (51.03%)
your network	your network, your wifi, your local network, your wi-fi, your local gateway, your gateway	348 (47.87%)
casting	cast-enabled, cast-capable, stream, cast-compatible, cast, chromecast-enabled, chromecast, screen mirroring	294 (40.44%)
WiFi network	wifi, wi-fi, wireless lan	275 (37.83%)
development artifact	debug, debugging, testing, developer, proxyman	32 (4.40%)
gaming	player, multiplayer, multi-player, in-game, game, two-player, players	30 (4.13%)
location dependent network	in-store, around you, nearby	25 (3.44%)
improve user experience	experience, improve, improved, enhance, enhanced, optimize, optimized, optimizing, improving, enhancing, personalized, personalize, stable connection	24 (3.30%)
network knowledge	tcp-network, tcp, udp, dns, voip, bonjour	22 (3.03%)
Internet connection	internet	4 (0.55%)
network quality testing	test the quality	3 (0.41%)

as it contains the keywords *local network*, and *devices*. We refined our keyword lists by looking into rationales for which no matching keyword was found and checking randomly-selected coded rationales. We iterated the processes of coding and refining until we found no new terms to add to the keyword lists, and only rationales without meaningful content remained uncoded. Further, to check that we did not miss prevailing concepts, we computed a count of each word in the rationale messages, which yielded no new concepts. We provide the full codebook and final list of keywords in Table 3.3 along with the number of rationales each code got assigned to.

We used this codebook as the basis for a content analysis, with the goal of deriving the concepts constituting the permission rationales.

3.5.2 Results

In our dataset, 727 (6.69%) apps contain a permission rationale. We report the concepts we identified during the content analysis and their occurrences in rationales based on our keyword-based approach.

Local Network. The most prominent concept we encountered was that of a *local network*. 81.02% of apps with a rationale referred to some form of local network. We found this across all different types of apps and use cases. Rationales primarily used the terminology *local network* or the acronym *LAN*, while several also referred directly to the wireless variant, e.g., *WiFi* or *wireless LAN*. A large portion of apps, 348 (47.87%), used terminology implying some form of ownership of the network in question by including the word *your*, i.e., *your network* or *your WiFi*. A smaller portion, 28 (3.85%), used the phrase *networks you use*, which might point users towards the technical reality, where all networks a device is subsequently connected to are affected by the permission. Finally, 25 rationales (3.44%) employed terminology referring to physical proximity. Expressions such as *around you* and *nearby* suggest that devices close to the user's phone are somehow affected by the permission, even though proximity does not necessarily imply they are part of the same network.

Device Interaction. The second major concept permission rationales contained was *device interaction* with 539 mentions (74.14%). This comprises any kind of communication with another device, e.g., TVs, IoT devices, or phones. The most common use cases for this interaction were *discovering* other devices and *casting* video or audio. Often, these concepts appeared in conjunction. Other use cases were generic interactions of *transferring data* or simply connecting to other devices. However, if discovering other devices was given as the main reason for requesting the permission, we also often found that no specific follow-up interaction was described. That means it was unclear what the app was doing with the gathered information and users are left to infer the specific use case from the app's purpose.

Niche Use Cases. We further identified rationales referring to other use cases without including specific mentions of other devices. None of these use cases was present in more than 5% of rationales. The most frequent one was *gaming* (4.13%), i.e., searching for other players. 3.30% of apps promised an *improved user experience* if the permission was granted. Mostly, this was done without explaining how exactly it was accomplished. Four apps in our dataset stated to require the permission to access the Internet, which is not correct since iOS does normally not restrict Internet access. Finally, three rationales gave *testing the network quality* as their reason for requiring the permission.

Network Knowledge. 22 apps (3.03%) required some *network knowledge* to process, as they included technical abbreviations such as TCP, or VoIP. By looking at the provided text, it became apparent that eleven of these messages, along with 21 others, were artifacts from development. This finding suggests that either the developers failed to remove these messages from the published version of their software, or that the app includes a dedicated debugging mode.

Takeaways

To answer RQ3: *Which concepts constitute the permission prompts?*, we showed:

- The most common concept is *local network*.
- To make an informed decision, users often need to understand what *casting* is, as well as which implications *discovering other devices* (i.e., scanning the network) can have.
- Alarmingly, the terminology in many rationales can be potentially misleading by limiting the scope (*your network*), creating false associations (*devices nearby*), or outright fueling misconceptions (*Internet access*).

3.6 Users' Permission Comprehension

After establishing the information users are presented with, we need to answer *RQ4: What is the user's understanding of these concepts?* We need to understand users' perception of the relevant concepts in order to establish their capability to make an informed decision. We investigate iOS users' knowledge of the permission and draw out their perception of the underlying technology using an online survey. Furthermore, we measure how widespread selected misconceptions are.

3.6.1 Methodology

As a necessary prerequisite to investigate iOS users' understanding of the local network permission, we derive the underlying concepts that are important for a correct understanding. Furthermore, we compose a list of common misconceptions to quantify how widespread they are and investigate users' awareness of common threats resulting from apps having local network accesses, as we discussed in Section 3.2.2.

Concepts Important for Understanding

We determined the fundamental concepts behind the local network permission through an expert brainstorming session. Three researchers met and discussed the results obtained from the content analysis of rationale messages (see Section 3.5.2). Two had a strong background in mobile security, the other one was experienced with usable security research. From this session, we derived the following concepts:

- **Boundaries:** a network's topology, i.e., which devices belong to the same local network, which are outside of it.
- **Transitivity:** the local network of a mobile device can change while it moves from place to place. The permission is granted once and then applies to all local networks.
- **Proximity:** devices physically close to each other are not necessarily part of the same network.

Misconceptions

To identify common misconceptions, three researchers independently conducted an online search on Google for the term *"local network permission."* We limited our search to the most commonly used platforms for tech-related problems: Reddit, StackOverflow, and X (formerly Twitter). The researchers then combined their findings into one coherent list. We include every misconception encountered at least once.

The most frequent beliefs we observed were that apps require the local network permission for using *Bluetooth* [237] or to access the *Internet via WiFi* [289, 182]. Some thought that apps could obtain the *WiFi password* [281] if granted the permission. Finally, people expressed that denying the permission can prevent other devices on the network from seeing their phone [284].

Survey Design

We used Qualtrics [272] to create our survey. We opted for a questionnaire consisting of mostly closed-ended questions. Doing so allows us to cover participants' knowledge of all relevant concepts, and to quantify how widespread misconceptions are. All that while still keeping the survey short to avoid fatigue.

Structure. The final questionnaire can be accessed online.[1] We first asked participants whether they knew what the local network is, and if they confirmed, we asked them to explain it in their own words. Next, we presented all participants with a multiple-choice question about the local network. A second multiple-choice question asked which use cases out of a given list would require an app to request the local network permission.

After this initial assessment of participants' general knowledge of local networks and the corresponding permission, we introduced a brief scenario, which serves as a reference for the remaining questions. For this, we designed a mock-up screenshot of a fictional app displaying the local network permission prompt, while showing the app's login screen in the background. A short text informed participants about the scenario, asking them to imagine that they just installed a new video-based social media app, which prompted them with the given permission request upon first launch.

We chose this setup as streaming to cast-enabled devices was a prominent use case in the permission rationales apps provided (see Section 3.5). We designed the scenario to be as realistic as possible. Hence, our mock-up app mimics a social media platform used by millions of people daily. We went with a fictional app to avoid biases towards existing systems, e.g., whether users trust or like an app. We included the default permission rationale message along with the system-generated description to establish a baseline.

After outlining the scenario as a reference for the remainder of the survey, we presented participants with a series of statements, divided into three blocks. For each statement, participants had to decide whether it is true or false. We also included an "I don't know" option, to discourage guessing. Each block began with

[1] Final survey questionnaire designed with Qualtrics [272] in our artifact: https://github.com/SecPriv/local_network/blob/main/survey.pdf

a short reminder text about the scenario. The survey tool randomized the order of statements within each block for every participant to mitigate learning or unwanted side-effects.

The first block had participants imagine themselves in their own homes, connected to their WiFi. It consisted of nine statements: six covering a different threat model each, two covering misconceptions, and one filler statement. We made sure to include a mix of true and false statements to not give away the correct answer to participants. The second block continued the scenario with the user leaving their home and walking down the street, not being connected to any WiFi network. The three statements in this block covered the concepts of boundaries, transitivity, and proximity, respectively. In the third block we asked participants to imagine themselves sitting in a café or working in an office, with the phone being connected to the WiFi networks of these respective places. Of the six statements in this block, three concerned the concepts of transitivity, and two the concept of boundaries, while one was a filler statement.

Finally, we measured our participants' tech-savyness using the Affinity for Technology Interaction (ATI) scale [126] and collected demographic information, including technology background, and the iOS version they use. To ensure the quality of our dataset, we concluded the survey by asking participants if they had answered all questions carefully, and if they wanted their data to be included in our study. We accompanied this with a brief description of the importance of reliable data for scientific progress and assured participants that they will receive their compensation regardless of their response. In addition, we used the multiple-choice questions and the ATI scale as sanity checks, flagging participants who selected "non of the above" plus other items for the former, or chose 1 or 5 for all items on the latter.

Recruitment

We chose Prolific [270] as a service provider for recruiting participants, as it has shown to deliver high quality results in related work [323]. Before launching our study, we refined the survey through pilot testing. First, we asked friends and colleagues to review our questionnaire and worked out any misunderstandings. In this stage, we approached both tech-savvy individuals, as well as those without a technical background. Then, we advertised the study on prolific for a small sample of ten participants to pilot test in a realistic setting, offering 4£ for compensation. We did not include these responses in the final data set since we adapted one question based on the responses we received. We also used this second pilot test to get a solid estimate of the time it takes to complete. Finally, we opened our study on Prolific for 150 participants. The number is above 121 participants required for a 99.9% confidence interval assuming 50% of the population knows the correct answers, considering a 15% margin of error, which is in line with related work [229] on iOS permissions. We applied the following criteria to participants: (1) above 18 years of age, (2) uses iOS devices, and (3) currently resides in the US. We advertised the survey as a study on iOS permissions, without mentioning security or privacy to minimize selection bias. During our study, two people aborted the study before completing it. Prolific filled up the quota with additional participants. We offered 2£ of compensation for an estimated duration of 12 minutes.

Ethics. Our institution's Ethical Research Board (ERB) approved the study design. We only collect necessary data, and we store and process them in line with the GDPR. Before starting the survey, we presented participants with a consent form detailing all information regarding data collection, their rights w.r.t. consent withdrawal, and the contact information of a responsible researcher. Only after explicitly providing consent, participants are forwarded to the survey.

Data Analysis

We evaluated multiple-choice questions by counting the number of correct answers. For single-choice questions, we counted how many participants answered them correctly, incorrectly, or responded with "I don't know." Some concepts and threats are covered by multiple questions. In those cases, we counted answers as correct if all questions were answered correctly, as incorrect if at least one was wrong, and as not sure otherwise. We evaluated all discrepancies between answers manually.

To draw comparisons, we divided participants into two groups, one with and one without knowledge of what a local network is. For grouping we used the answers to the question "Do you know what a local network is?" We asked participants who responded with "Yes" to briefly describe the concept in their own words. Based on these descriptions we redistributed participants to the group without knowledge. Two researchers independently grouped each open-ended response w.r.t. the participants' understanding: one of people with no or an incorrect understanding of local networks, and the other one with a correct understanding. We used Cohen's Kappa [171] to measure the inter-rater agreement. Afterward, we discussed and resolved disagreements.

To measure the influence of a person's understanding of what a local network is on their ability to correctly answer questions about it, we performed a T-test and calculated the effect size using Cohen's d. We performed χ^2 tests to study the differences between the groups and their answers regarding concepts, threats, and misconceptions.

For all tests, we stated the null hypothesis H_0 as "There is no difference between the two groups" and the alternative hypothesis H_1 as "There is a difference." We rejected H_0 for resulting p-values below the significance level of 0.05.

3.6.2 Results

First, we provide a general overview of our participants, followed by insights into their understanding of the local network and its permissions. We then detail common threats and misunderstandings. We provide an overview of our results and how they split up into participants with and without local network knowledge in Table 3.4.

Overview of Participants

We removed two participants from our study, resulting in a total number of 148 participants. One did not wish for their answers to be included, and one chose the same option for each item on the ATI scale, which indicates careless responding. On average, it took the participants 9 minutes and 58 seconds to fill out our survey. Most participants are between 25 and 34 years old (35.14%). The majority are female

Table 3.4: Results of our user study. The *total* columns reflect results from all 148 participants. The *local network (LN) knowledge* and *no knowledge* columns break down the data into groups based on whether we categorized participants as having an understanding of local networks. We related the numbers to the group sizes. For chi-squared tests marked with *, we focused only on correct and incorrect answers since fewer than five participants were unsure.

		Total ($N = 148$)			**LN Knowledge** ($N = 71$)			**No Knownledge** ($N = 77$)			χ^2 **Test**		
		Correct	**Wrong**	**Unsure**	**Correct**	**Wrong**	**Unsure**	**Correct**	**Wrong**	**Unsure**	χ^2	p	ϕ
Concepts	Boundaries	91 (61.49%)	24 (16.22%)	33 (22.30%)	42 (59.15%)	14 (19.72%)	15 (21.13%)	49 (63.64%)	10 (12.99%)	18 (23.38%)	1.24	0.54	0.09
	Proximity	88 (59.46%)	27 (18.24%)	33 (22.30%)	48 (67.61%)	12 (16.90%)	11 (15.49%)	40 (51.95%)	15 (19.48%)	22 (28.57%)	4.49	0.11	0.17
	Transitivity-1 Base	112 (75.68%)	24 (16.22%)	12 (8.11%)	65 (91.55%)	5 (7.04%)	1 (1.41%)	47 (61.04%)	19 (24.68%)	11 (14.29%)	*9.51	<0.01	0.26
	Transitivity-1	34 (30.09%)	64 (56.64%)	15 (13.27%)	20 (30.77%)	33 (50.77%)	12 (18.46%)	14 (29.17%)	31 (64.58%)	3 (6.25%)	*0.22	0.64	0.05
	Transitivity-2 Base	108 (72.97%)	28 (18.92%)	12 (8.11%)	60 (84.51%)	10 (14.08%)	1 (1.41%)	48 (62.34%)	18 (23.38%)	11 (14.29%)	*19.18	<0.01	0.14
	Transitivity-2	26 (23.85%)	55 (50.46%)	28 (25.69%)	15 (25.00%)	29 (48.33%)	16 (26.67%)	11 (22.45%)	26 (53.06%)	12 (24.49%)	0.24	0.89	0.05
Threats	Cross-user Tracking	75 (50.68%)	39 (26.35%)	34 (22.97%)	35 (49.30%)	19 (26.76%)	17 (23.94%)	40 (51.95%)	20 (25.97%)	17 (22.08%)	0.12	0.94	0.03
	Device Profiling	76 (51.35%)	35 (23.65%)	37 (25.00%)	45 (63.38%)	10 (14.08%)	16 (22.54%)	31 (40.26%)	25 (32.47%)	21 (27.27%)	9.46	0.01	0.25
	Exposing Devices	84 (56.76%)	20 (13.51%)	44 (29.73%)	41 (57.75%)	9 (12.68%)	21 (29.58%)	43 (55.84%)	11 (14.29%)	23 (29.87%)	0.10	0.95	0.03
	Location Profiling	74 (50.00%)	37 (25.00%)	37 (25.00%)	40 (56.34%)	18 (25.35%)	13 (18.31%)	34 (44.16%)	19 (24.68%)	24 (31.17%)	3.55	0.17	0.15
	At Least One	123 (83.11%)	21 (14.19%)	4 (2.70%)	62 (87.32%)	7 (9.86%)	2 (2.82%)	61 (79.22%)	14 (18.18%)	2 (2.60%)	*1.47	0.23	0.10
Misconcept.	Bluetooth	63 (42.57%)	73 (49.32%)	12 (8.11%)	28 (39.44%)	42 (59.15%)	1 (1.41%)	35 (45.45%)	31 (40.26%)	11 (14.29%)	*1.83	0.18	0.12
	Internet Access	48 (32.43%)	88 (59.46%)	12 (8.11%)	29 (40.85%)	41 (57.75%)	1 (1.41%)	19 (24.68%)	47 (61.04%)	11 (14.29%)	*1.86	0.17	0.12
	Visible Phone	32 (21.62%)	85 (57.43%)	31 (20.95%)	18 (25.35%)	42 (59.15%)	11 (15.49%)	14 (18.18%)	43 (55.84%)	20 (25.97%)	2.89	0.24	0.14
	WiFi Password	76 (51.35%)	24 (16.22%)	48 (32.43%)	39 (54.93%)	10 (14.08%)	22 (30.99%)	37 (48.05%)	14 (18.18%)	26 (33.77%)	0.81	0.67	0.07
	No Misconception	21 (14.19%)	125 (84.46%)	2 (1.35%)	9 (12.68%)	61 (85.92%)	1 (1.41%)	12 (15.58%)	64 (83.12%)	1 (1.30%)	*0.07	0.79	0.02

Table 3.5: Demographics of our final 148 participants, all currently reside in the US. This is after we excluded two participants from our dataset who did not want their data to be included or failed attention checks. We recruited a balanced sample in terms of gender, age, and IT background, with the majority running the latest iOS versions.

Demographics	**Participants (%)**
Female	89 (60.14%)
Male	57 (38.51%)
Non-binary	2 (1.35%)
18 - 24 years	35 (23.65%)
25 - 34 years	52 (35.14%)
35 - 44 years	35 (23.65%)
45 - 54 years	15 (10.14%)
55 - 64 years	8 (5.41%)
>= 65 years	3 (2.03%)
IT Background	72 (48.65%)
No Background	76 (51.35%)
iOS 17	96 (64.86%)
iOS 16	23 (15.54%)
iOS 15	9 (6.08%)
iOS 14	3 (2.03%)
iOS 13	5 (3.38%)
<= iOS 12	3 (2.03%)
Not Sure	9 (6.08%)
Permission Seen	116 (78.38%)
Permission Not Seen	16 (10.81%)
Not Sure	16 (10.81%)

(60.14%), and almost half declared to have an IT background (48.65%). The mean ATI score is 3.46 ($sd = 0.87$), which is around the general population's average [126]. The most frequently used iOS version among participants is iOS 17 (64.86%), i.e., the latest version at the time of our study released in 2023, and 78.38% declared that they have seen the local network permission before. In Table 3.5, we provide further details on the demographics of our participants, their background and the iOS version they are using.

Local Network Understanding

Out of 148 participants, 31 (20.94%) stated that they do not know what the local network is, while the remaining 117 provided explanations. The inter-rater agreement on whether an explanation demonstrated a correct understanding of the concept was 0.82, considered almost perfect [171]. After reaching a consensus, we classified

71 responses as correct, leaving 46 incorrect responses, resulting in 77 participants without a correct understanding.

We tested whether our classification impacted the answers about local network knowledge and whether an IT background influenced the classification. With a one-sided T-test, we found a significant difference between the groups we classified and their score on the local network control questions ($p < 0.001$, $effect\ size = 0.88$), which supports the validity of our classification. Of the people with an IT background, we classified 58.33% as having a correct understanding, while for people without a background, the same was true for 38.16%. The χ^2 test shows a significant difference between the groups ($\chi^2 = 5.24$, $p = 0.02$, $\phi = 0.18$).

Concepts

We observed that the concept of *transitivity* is the most challenging for users to understand. First, we asked if a use case is possible at home after granting the permission, e.g., to discover other devices connected to the network. Later, we asked similar questions again in the café and office scenarios. We evaluated if people who correctly answered the question in the home scenario recognized that access would also transfer to other local networks. We split up the evaluation into two parts, as we had two use cases for which we repeated the question in the scenarios. As the results are similar and the second is better suited for χ^2 tests, we only refer to those results, but both are included in Table 3.4 (in the Appendix). 72.97% of the participants answered the base question correctly. Only 23.85% had the follow-up question in other locations correct, and 50.46% answered at least one of them incorrectly. Having a local network understanding did not improve the ability to answer the transitivity concept correctly. The knowledge helped to get the base question right, 84.51% vs. 63.64% ($\chi^2 = 19.18$, $p < 0.01$, $\phi = 0.14$). However, when comparing the follow-up questions among those who answered the baseline correctly, both groups answered similarly. Of the group with knowledge, 25% answered it correctly, 48.33% wrong, and 26.67% were unsure, compared to 22.45% correct, 53.06% wrong, and 24.49% unsure in the other group.

More participants understood the concept of network *boundaries* and the related *proximity* concept. Overall, 61.49% answered the question about network boundaries correctly, while 16.22% answered it wrong. We asked two questions to test the understanding that devices in physically proximity do not have to be part of the same network. Both scenarios took place outside on the street, and we asked if the app could communicate with a smart TV inside a shop or another person's phone passing by. 59.46% correctly answered both questions, 22.30% were unsure what to answer, and 18.24% had at least one of the answers wrong. To improve the understanding, iOS could ask permission again for every new local network the app tries to access. This would make it transparent to the users to which local network the app has access to, allowing them to make individual decisions.

Threats

We asked participants about four threats coming from local network access: (1) *exposing devices*, (2) *inferring the location*, (3) *cross-user tracking*, and (4) *device*

identification. On a positive note, for each threat, at least half of the participants displayed some sort of awareness. Most participants knew that devices protected by a firewall are still reachable from within the network, 56.76% correctly answered the respective questions, while 29.73% were unsure.

Further, we tested the knowledge about device detection within the local network with two questions. One question was general, while the other explicitly asked about sensitive devices, e.g., security cameras or health-related devices. In total, 51.35% could correctly respond. Additionally, 14.19% knew about the general case but were unsure about sensitive devices, and 9.46% were not sure about both questions. Finally, 50.68% correctly identified the possibility to perform cross-user tracking by linking users together based on local network information, 26.35% did not deem that possible while the rest were unsure.

We also asked two questions about inferring a users location: one about the possibility of obtaining the approximate location, e.g., through the router's MAC address, and one about inferring location changes based on differences in the connected network. Half of the participants got both questions right, and 25% had at least one answer wrong. Considering the latter, users seem to be more aware of the potential to estimate a location based on network information, as 13.51% correctly responded, but they were unsure about the other. In contrast, only 3.38% of respondents were unsure about the approximate location but knew of the possibility of tracking changes.

Device detection is the only threat for which we observed that users with local network knowledge did significantly better ($\chi^2 = 9.46$, $p < 0.01$, $\phi = 0.25$). 63.38% of the group with an understanding of the local network were aware of it, compared to 40.26% of those without. Positively, we observed that even without being able to explain the local network correctly, participants were aware of threats coming from apps having access. Even if the consequences might not be that obvious compared to other permissions, 83.11% of participants knew of at least one threat.

Misconceptions

Despite many of our participants having a technical background and an understanding of the local network, we observed that misconceptions are common. Nearly everyone (84.46%) had at least one misconception about the permission. The most common misconception is that apps require local network access to access the Internet (59.46%), closely followed by the belief that phones are not visible within the local network if the permission is denied (57.43%). Nearly half of the participants (49.32%) think that apps require it for Bluetooth communication. The only misconception less widespread is that the permission allows retrieving the WiFi password (16.22%).

Surprisingly, we observed that the group with local network knowledge did not perform significantly better. The misconception about Bluetooth was even more widespread (59.15% vs. 40.26%) among those with local network knowledge, while all others were similarly common: Internet access 57.75% vs. 61.04%, phone visibility 59.15% vs. 55.84%, and WiFi password 14.08% vs. 18.18%.

Rationales. Malicious apps could exploit misconceptions to trick users into accepting the permission by mentioning misconceptions in the rationales. However, exploring the effect requires further user studies. In Section 3.5, we found four

apps mentioning *Internet* in their rationales, e.g., one app states "*This app requires an Internet connection to access your account data and vehicle features. Your local network will be used when cellular data is not available.*", which could trick users into giving permission. However, it could also be that the app accidentally triggers the permission, e.g., a library, and developers themselves suffer from the misconception. To avoid misconceptions, Apple could check if rationales contain keywords hinting towards misconceptions before releasing the app on the App Store.

Takeaways

To answer RQ4 *What is the users understanding of these concepts?*, we showed:

- Even participants with a general understanding of the local network did not know about all the concepts required to make an informed decision.
- Misconceptions about the local network permissions are widespread. Developers could exploit this to trick users into accepting the permission.

We recommend prompting for permission on each distinct WiFi. This behaviour is more in line with users' expectations and would allow them to make better informed decisions.

3.7 Limitation and Future Work

Our work faces limitations and offers opportunities for future work. For the internal workings of the protected resources for network access, we rely on Apple's developer documentation. This might not cover all methods to access the local network and thus miss other bypasses. Future work could reverse engineer the iOS implementation of TCC to study the internal permission handling. It is yet unclear how Apple will handle the local network permission for browsers that are not based on WebKit, after being required to allow other engines in Europe due to the EU's Digital Markets Act [112, 268]. As we have found permission bypasses in webview-related components, this and other permissions' enforcement is worth exploring. Furthermore, Apple announced the extension of the local network permission in macOS 15 (released in 2024) [36], prompting questions about its enforcement on this platform as well.

Our large-scale local network access study has limitations inherent to dynamic analysis. Apps might detect that they are analyzed and behave differently than they normally would [369, 290]. Our dynamic interactor might not trigger functionalities that lead to local network access, e.g., functionality available only after login or verification. Thus, our numbers are a lower bound. We accept all permissions. However, declining permissions might lead to different behavior regarding local network accesses [282]. It also remains unclear if and what data apps share with external parties about the local network and the devices within it. During our manual analysis, we found apps that we suspect of doing so, but we could not confirm it due to obfuscation. Future work could leverage data flow analysis to track data coming from the local network.

Bonjour services, like AirPlay, do not trigger the permission, potentially confusing users as to why some functionality does not require permission [334]. Also, AirPlay

can use Bluetooth to discover nearby devices [34] potentially influencing the Bluetooth misconception. Follow-up work could study the misconceptions further and their influences on users' decisions on granting permission in depth.

Finally, we focused our user study on iOS users. Studying Android users' understanding of the local network could gain further insights into the permission's impact and overall effectiveness, as well as on how to design a similar permission for Android. Finally, our sample of users residing in the US might hinder the generalizability to other user groups with diverse backgrounds, also in light of regional differences in terms of privacy regulations and awareness. One complementary direction for future work is the automated mining of iOS and Android app store reviews, which has shown promise in providing a broad picture of users' perceptions of apps' functionality [117] and [308].

3.8 Related Work

Local Network Access. Kuchhal and Li [197] performed a large-scale empirical investigation of websites' local network accesses. Other work [27, 164, 1] investigated how websites could scan the local network and attack connected devices. We investigate mobile apps instead of websites. We expect different access behavior for mobile apps: (1) there are legitimate use cases for it (e.g., companion apps controlling their IoT devices, or mirroring the screen to a monitor), and (2) in iOS, a permission guards the access.

Sivaraman et al. [316] demonstrated how forged apps could attack devices on the local network. Königs et al. [195] showed privacy implications of zero configuration protocols if they reveal device names by local network communication. In addition to user information, they can reveal sensitive device information, like the OS version, which attackers can use to find known vulnerabilities, as shown by Geneiatakis et al. [132]. Tang et al. [326] identified vulnerable UPnP libraries in 13 apps by analyzing network services in iOS apps and the reuse of vulnerable libraries. In contrast, we focus on the preceding step, which is the local network access of apps and the permission guarding it on iOS.

Girish et al. [135] analyzed how IoT devices and mobile apps communicate via the local network. By executing Android apps they detected app libraries that scanned the local network. Further, they showed that household fingerprinting and cross-device user tracking are possible using multicast protocols, like mDNS or UPnP. On the contrary, we study the permission on iOS, its impact, and the users' understanding of local network accesses.

Permission Analysis. Reardon et al. [283] found Android apps bypassing permissions with side and covert channels. To find those, they executed the apps in an instrumented environment. Yeke et al. [358] and Tileria et al. [333] identified cross-platform privacy leaks between smartwatches and Android apps for which the Android permission model is not sufficiently designed. In comparison, we focused on the local network permission, which does not exist on Android and differs from other permissions.

Previous work on permission rationales investigated mostly Android apps. Liu et al. [209] showed that especially permissions which are difficult to understand

were insufficiently covered by rationales. Elbitar et al. [107] discovered that purpose messages can assist users with making informed decisions but also highlighted that the clarity of a message's content is highly subjective.

Tan et al. [324] studied the effect of developer-provided rationales on iOS in 2014. At the time of their study, most messages (98.3%) informed the users about benefits when granting the permission. Shen et al. [309] investigated to what extent permission rationales can help users understand the scope of a permission. They covered all available permissions on Android and iOS, which at the time did not include the local network permission. Their results suggest that only a small fraction of users can correctly infer an app's capabilities after granting a permission. Most recently, Mohamed et al. [229] studied the app tracking transparency permission rationales. They identified so-called dark patterns that apps use to trick users into granting permissions. In contrast, we study users' local network understanding and misconceptions of its permission, a prerequisite to analyzing dark patterns, which we leave for future work.

3.9 Conclusion

We identified two iOS components that can bypass the permission and that the protection for complex networks and Virtual Private Network (VPN) is insufficient. With our dynamic analysis, we showed that the permission potentially influences when apps access the local network. We found more iOS apps accessing the local network than Android apps, which is likely caused by Bonjour, a multicast protocol by Apple. Further, we found apps using Bonjour methods without declaring a service string, thus bypassing a requirement of their usage. With our content analysis of permission rationales, we show that it is vital to have an understanding of what a local network is to make sense of most messages. Building on that, we studied users' understanding of the permission, threats coming from local network access, and misconceptions. Positively, we found that nearly every participant (83.11%) was aware of at least one threat. However, misconceptions were even more widespread (84.46% had at least one), which could help malicious apps to trick users into granting permission.

3.10 Appendix

3.10.1 iOS Permission Test

We connected our phone to a `192.168.2.1/24` network, and a VPN (`10.1.0.1/24`) for the IPv4 tests. For IPv6 tests, we used a stateful `fd00::1/24` network. For each class from Table 3.1, we tested the addresses in Table 3.6.

Table 3.6: Tested IP addresses of our test app.

	IPv4	**IPv6**
Local	`192.168.2.1` (Router)	`fd00::1fa2` (Laptop)
	`192.168.2.100` (No device)	`fd00::fa` (No device)
Local outside	`10.1.0.1` (VPN)	`fd01::1`
	`10.10.10.10`	
	`192.168.1.1`	
Multicast	`224.0.0.69` (Unassigned)	`ff02::1` (All nodes)
	`224.0.0.251` (mDNS)	`ff02::2` (All routers)
	`239.255.255.250` (SSDP)	`ff02::18d` (Unassigned)
		ff02::fb (mDNS)
		ff05::c (SSDP)
Broadcast	`192.168.2.255`	
	`255.255.255.255`	

3.10.2 App Categories

We manually categorized the apps accessing the local network instead of using the store categories to be more precise, e.g., there exists no IoT category [185, 305]. In Table 3.7, we provide the categories and the number of apps.

Table 3.7: Categories of apps that accessed the local network. We summarized all categories with two or fewer apps in *Other*. and show the number of apps that only accessed it on one respective platform, and ∩ the apps that accessed the local network on both platforms. The numbers are in relation to the 199 apps that access it on at least one platform.

	Total			∩
IoT	112 (56.28%)	25 (12.56%)	28 (14.07%)	59 (29.65%)
Video	17 (8.54%)	11 (5.53%)	1 (0.50%)	5 (2.51%)
Events	16 (8.04%)	16 (8.04%)		
Audio	13 (6.53%)	7 (3.52%)	4 (2.01%)	2 (1.01%)
Games	12 (6.03%)	6 (3.02%)	5 (2.51%)	1 (0.50%)
Network	4 (2.01%)	1 (0.50%)	1 (0.50%)	2 (1.01%)
Fitness	3 (1.51%)	2 (1.01%)	1 (0.50%)	
Organization	3 (1.51%)	3 (1.51%)		
Shopping	3 (1.51%)	1 (0.50%)	2 (1.01%)	
Other	16 (8.04%)	10 (5.03%)	5 (2.51%)	1 (0.50%)

4 Leaky Apps: Large-scale Analysis of Secrets Distributed in Android and iOS Apps

Abstract

Mobile apps store various types of secrets to support their functionalities. These include API keys, and cryptographic material to authenticate users and access back-end services. Once distributed, attackers can reverse-engineer the apps, and these secrets become accessible, posing risks such as data leaks, and service abuse.

In this paper, we conduct a large-scale analysis of 10,331 Android and iOS apps to study how secrets are embedded in mobile apps. Our methodology involves extracting and validating credentials from app bundles and comparing the types and frequency of embedded secrets across Android and iOS to identify systematic differences between the two ecosystems. To assess temporal dynamics, we re-analyze apps released in 2023 after their updates in 2024.

Our findings show that apps not only leak secrets required for functionality but also unintentionally include sensitive information like markdown documentation, and dependency management files.

We discovered 416 functional credentials across 65 services, including 13 Git credentials that grant access to 218 public and 2,440 private repositories. Our analysis reveals that iOS apps are more likely to expose secrets, although information leaks exist in both Android and iOS apps. Finally, we show that even if developers remove embedded credentials in later versions, they frequently forget to revoke them, leaving the credentials exploitable.

Publication The work presented in this Chapter resulted from a collaboration with Sebastian Schrittwieser, and Edgar Weippl. It has been published at ACM Conference on Computer and Communications Security (CCS), 2025 [299] where it has been recogniced with the "Distinguished Paper Award." I developed the static analysis, conducted the evaluation, and wrote the paper. Sebastian Schrittwieser advised the work, gave feedback, and revised the paper. Edgar Weippl also provided feedback.

4.1 Introduction

Smartphone apps have become essential in daily life, with approximately 6.9 billion smartphone users worldwide depending on over 8.9 million apps in 2023 [157]. These apps handle our communications, finances, social interactions, and personal health data, deeply integrating themselves into both our personal and professional

lives. However, this widespread reliance creates significant security risks, particularly when it comes to protecting sensitive information embedded within apps. Security researchers recently uncovered that 13 widely-used mobile apps, some downloaded millions of times, exposed sensitive cloud credentials [156]. These credentials can enable attackers to manipulate or steal user data, potentially resulting in severe privacy breaches.

The issue arises because developers embed sensitive data, such as API tokens and authentication credentials, directly into the app's code. While some of these secrets are intentionally included for necessary functionality, developers also include sensitive information inadvertently due to oversight or rushed development cycles. An illustrative example is Snapchat, which unintentionally leaked portions of its source code through its iOS app [82]. Once an app reaches users' devices, its embedded secrets become vulnerable to extraction through reverse engineering, often referred to as a Man-At-The-End (MATE) threat model [96].

Given the scale of mobile app usage globally, the impact of compromised app secrets is substantial. This threat is underscored by its inclusion in the OWASP Mobile Top 10 of 2024, which ranks *Improper Credential Usage* as the highest security risk [257].

In the past, researchers studied the exposure of secrets in public code repositories. Meli et al. [219] performed a large-scale longitudinal analysis on secrets in GitHub repositories using regular expression-based pattern matching. Jungwirth et al. [188] analyzed secrets in dotfiles, which are often used as configuration files and hidden by default in most OSes. Jin et al. [186] scanned GitHub for IoT cloud policies to identify misconfigured cloud services. Further, several popular rule-based analyses, such as Gitleaks[141] and TruffleHog [335], help to detect potential leaks of secrets in code repositories.

Investigating secrets in mobile apps differs from examining code repositories, primarily because repositories are explicitly intended for sharing code, making developers potentially more cautious about exposing secrets than in apps. Additionally, repositories contain source code, whereas mobile apps distribute app bundles containing compiled code, making secrets less immediately visible.

Still, the situation of secrets in mobile apps is under-explored. Earlier research primarily examined the leakage of PII from mobile apps [89, 236, 288, 287] or targeted specific types of secrets in the app's executed code [367, 370, 220]. Zhou et al. [367] employed static data flow analysis to detect AWS and email credentials in Android apps. Zuo et al. [370] proposed LeakScope, an Android app analysis methodology based on string VSA to identify wrongly configured AWS, Google, and Microsoft cloud credentials. Mendoza et al. [220] extracted app-to-web communication to detect vulnerabilities like web API hijacking. However, these works did not consider the wide range of potentially shared information in mobile apps or information unintentionally included in app bundles.

Our study differs from prior research in three important aspects: (1) We do not limit our analysis to the app's executable code. Snapchat previously leaked source code by accidentally packaging it with the iOS app bundle. Such unintentionally included data might hold secrets undetectable through code analysis methods. (2) Android is not the only widely used mobile OS. Despite iOS having a 58% market share in the US [170], large-scale studies of secret exposure within iOS apps remain

notably absent in the literature. (3) Earlier studies typically provided only a snapshot, lacking an understanding of how secrets evolve over time, for example, whether developers later remove or revoke them.

To get a complete picture, we first study the content of app bundles, addressing *RQ1: What files do mobile apps contain?*. Those can range from binaries and configuration files to unintentionally included items, like build scripts.

After understanding what files developers distribute in their apps, we focus specifically on their contents. We use a regular expression-based detection approach, similar to those successfully used to identify secrets in code repositories. Through this, we answer *RQ2: What secrets do developers distribute in mobile apps?*

Next, we investigate platform-specific differences by performing a large-scale, comparative analysis across 10,331 Android and iOS apps. This allows us to answer *RQ3: How does the situation differ between Android and iOS apps?*

Developer responses significantly affect the security impact when secrets become public. Effective responses include revoking compromised tokens and releasing updates. To understand how developers handle such exposures over time, we updated our original 2023 dataset in 2024 and investigate *RQ4: How did the situation change between 2023 and 2024?*

In summary, we make the following key contributions:

- We performed a comprehensive analysis of files distributed within 10,331 Android and iOS apps, and were able to show that developers share sensitive information, e.g., dependency files, and documentation, by accident in the apps;
- We identified 416 valid credentials across 65 services, including highly critical findings such as 13 valid Git credentials;
- To the best of our knowledge, we conducted the first large-scale analysis, specifically examining secrets in both Android and iOS apps. Our results underscore the importance of studying apps on both platforms, as we frequently identified issues exclusive to one platform;
- We were able to highlight that even after developers remove credentials from apps, they often neglect to revoke them, leaving these credentials exposed to misuse.

Artifacts We publish our code and analysis artifacts to make our study reproducible and to enable future work: `https://github.com/CDL-AsTra/leaky_apps`.

4.2 Threat Model, Secret Definition, and Mitigation

Threat Model Attackers may analyze downloaded apps to extract embedded secrets in a MATE attack scenario. Once an app containing secrets is published, developers must assume that attackers can discover them. Malicious actors might download and analyze a vulnerable version while it is available. Further, older app versions remain accessible through third-party stores [28, 29], the Androzoo dataset [7], or even directly from the Google Play Store [60] or Apple App Store [6].

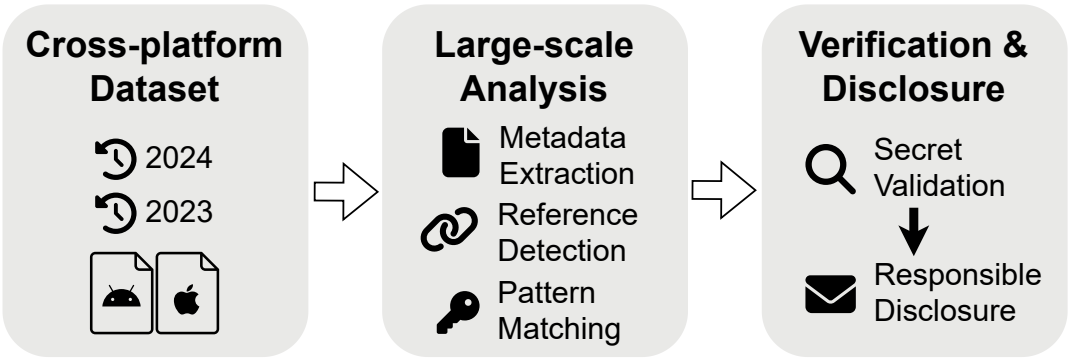


Figure 4.1: Overview of our methodology. We performed a large-scale analysis of 10,331 Android and iOS apps, collected twice, once in 2023 and in 2024. We validated the identified credentials and automatically disclosed them responsibly.

Secret Definition In our paper, we consider following information as secret: (1) Information whose exposure might result in financial loss for app creators, e.g., API tokens with usage-based billing or proprietary source code; (2) Information compromising user privacy, e.g., tokens granting access to online databases; (3) Information exploitable by attackers to target users or developers, like documentation containing internal URLs useful for social engineering.

Mitigation Our research aims to identify and responsibly disclose secrets found in apps. We also aim to provide insights that help developers detect security issues before publishing their apps.

When developers include tokens in apps, they must consider that attackers might extract them. Thus, tokens should have access strictly limited to the required scope. Further, app developers should monitor tokens closely and react immediately to any misuse, such as unintended use of Google Map tokens that could lead to costs due to pay-per-use charges.

If secrets are exposed or misused, developers should revoke them promptly and assess whether more secure alternatives to embedding them in their apps exist. They should review the scope of the credentials and investigate previous activities to detect malicious usage. Developers must also have an update strategy prepared beforehand, as without proper planning, revoking credentials could disrupt older app versions.

Applying code obfuscation might make discovering secrets more difficult, but it does not prevent manual analysis. Therefore, developers should never rely solely on obfuscation to protect sensitive data, especially when exposure could cause significant harm.

4.3 Methodology

We designed a static analysis methodology for large-scale analysis of Android and iOS apps to gain insights into the files and data they distribute. Figure 4.1 provides a high-level overview of our approach.

4.3.1 Large-scale Analysis

Our analysis pipeline includes three components written in Python: (1) file metadata extraction, (2) file reference detection, and (3) secret extraction using regular expressions.

Metadata Extraction To study the contents of Android and iOS apps, we extract data from APK (Android) and IPA (iOS) files, both of which are bundled as archives [273]. This allows for a unified processing approach across both platforms. We also analyze split APK files, which Android uses to package language resources, screen densities, and native libraries [24]. We process them the same way as the main APK.

For each file within an app, we store its size, name, file path, suffix, and MIME type. This metadata supports downstream analyses, such as identifying files that may have been included unintentionally.

Reference Detection We identify filenames referenced within other files to infer their potential use in the app. When a filename appears in other files, e.g., the app's binary, it typically indicates the file is accessed at runtime. This allows us to estimate how different files are used based on these references.

To streamline this analysis step, we exclude media files and general configuration files such as manifests or UI layouts since their usage is implicitly managed by the OS.

Pattern Matching Pattern matching offers two key advantages over other approaches. First, it allows a unified methodology across Android and iOS, facilitating direct comparison across platforms. Second, it enables the detection of secrets in files not actively used by the app, as well as those written in other programming languages such as JavaScript (JS) or code from cross-platform frameworks.

We use TruffleHog [335] to detect secrets, adapting it from its original purpose of scanning code repositories. After reviewing its detection patterns, we extended TruffleHog with seven additional rules sourced from Gitleaks [141] to enhance its detection coverage. We include the complete list of rules as part of our artifact [303]. To improve efficiency, we separated secret verification from the initial detection process. This prevents repeated and unnecessary validation requests when the same secret appears across multiple apps. The decoupled approach allows us to pre-filter results before verification, as detailed in Section 4.3.2.

As a preprocessing step to pattern matching, we run `strings` [127] on all non-text files to extract printable strings. For Android apps, we additionally use JADX [137], a decompiler that converts DEX bytecode into Java source code, which aligns more closely with code found in public repositories.

4.3.2 Verification and Disclosure

Secret Validation

We developed our analysis methodology with attention to ethical standards and responsible disclosure practices. Validating detected secrets is necessary to avoid False

Positive (FP) reports that could cause unnecessary effort for developers. Remote validation introduces the risk of incurring costs if a service charges per request. However, since we issue only one request per token and most services bill by volume, we considered the practical risk to be low. Further, it aligns with validation strategies done by related work of secret detection [106, 370].

We further reduce requests by eliminating FPs before remote validation through rule-based heuristics. We discard results if a single detection rule flags 15 or more potential secrets in one file. This threshold is based on our observation that some rules may match unrelated patterns, such as hash values. We determined this cutoff empirically and discuss the link between detection frequency and actual credentials in Section 4.5.1.

We only perform remote validation after this filtering step. We selected endpoints that return distinct status codes depending on token validity, see [303] for the list of services. All validation requests do not alter data on remote servers. Further, we do not retain any response content. Only status codes and error messages are used to assess whether a credential is valid.

Responsible Disclosure

We responsibly disclosed all findings by contacting developer email addresses listed on the Google Play Store, which we retrieved along with app packages. For reporting of findings in iOS apps, we also used the corresponding Google Play developer addresses as the Apple App Store did not offer explicit developer contact information until February 2025 (we disclosed our findings in January 2025), and our dataset links each iOS app to a matching Android version, see Section 4.3.3.

For each finding category, we used a template message to ensure clarity and consistency. These templates include a description of the issue and proposed mitigation steps (see Section 4.11 for an example). When a finding affected both the Android and iOS versions of the same app, we combined the results into one report. However, we did not merge findings across different apps, even if they were linked to the same developer account. All reports were sent via our university's mail server.

4.3.3 Cross-platform Dataset

To study both Android and iOS apps at scale, we used an updated version of the cross-platform dataset introduced by Schmidt et al. [297]. Their dataset includes both popular and randomly chosen apps. Using the matching method of Steinböck et al. [320], and the Google migration API, they identified 10,862 iOS apps with corresponding Android versions.

Since our analysis requires decrypted iOS apps and the original dataset included encrypted versions, we used frida [8] to decrypt their dataset. We executed frida on an iPhone 8 running iOS 16, which we had jailbroken with palera1n [259]. This step reduced the usable set of iOS apps to 10,331 as decryption failed due to anti-debugging protections [290, 369] and iOS version mismatch. We refer to this set of apps as *2023 dataset*, as the apps were downloaded from the Play Store and App Store in 2023.

To capture how the inclusion of secrets in mobile apps evolves over time, we re-downloaded the latest app versions from the dataset in October 2024 from the official stores. Of the original set, 8,702 Android apps and 9,212 iOS apps were still available. We refer to the updated version as *2024 dataset*.

4.3.4 Data Analysis

Significance Test To measure the significance of differences between Android and iOS versions of the same app, as well as differences between app versions from 2023 and 2024, we perform dependent t-tests and calculate effect sizes using Cohen's d. For t-tests comparing 2023 and 2024, we include only apps that remained available in 2024. For all statistical tests, we define the null hypothesis H_0 as *"There is no difference between the two groups"*, and the alternative hypothesis H_1 as *"There is a difference"*. We reject H_0 when the resulting p-value is below the significance level of 0.05.

Case Studies We select case studies manually based on factors such as the apps relevance, revealing names, e.g., file or directory.

4.4 App Content

All numbers in this section refer to the 2023 dataset, which was chosen for better comparison due to its equal number of Android and iOS apps. We provide insights into the changes between the datasets from 2023 and 2024 in Section 4.7.

To understand what types of files apps contain, we categorized files based on their suffix, see Table 4.1. We derived these categories by merging two existing GitHub projects [104, 100] and manually reviewing the 500 most frequently used suffixes that had not yet been categorized. In total, we mapped 876 suffixes to 17 file categories.

As expected, all successfully analyzed apps contained binary code, configuration files, and system files. The overall percentage is slightly lower than 100% due to analysis failures in 31 Android and five iOS apps caused by corrupted app bundles.

In the following, we report on file types that are typically not bundled with Android or iOS apps but still appeared in our analysis.

4.4.1 Binaries

Windows

We found `.exe` and `.dll` binary suffixes, which are typically associated with Windows, in 321 Android and 228 iOS apps. Manual analysis showed that 266 of these apps were built using Microsoft's cross-platform framework Xamarin [226]. These Windows-related files contain Ahead-of-time (AOT) and Just-in-time (JIT) compiled code designed for execution on mobile devices [224, 225]. Similarly, we discovered `.aspx` files in 1,450 Android apps (14.04%) and one iOS app. All occurrences on Android were linked to the Mono project [230], which is designed for cross-platform development and also used by Xamarin [224].

Table 4.1: File categories found in mobile apps. We categorized files based on their suffix. The **File** columns show the number of files per category, while the **App** columns indicate the number of apps containing at least one file of the category. Note that, although every app requires a binary, the relative number is below 100%, as our analysis failed for 31 Android and 5 iOS apps due to corrupted archives.

	File	**App**	**File**	**App**
ai model	3,293	755 (7.31%)	2,772	491 (4.75%)
archive	17,634	7,977 (77.21%)	20,395	1,767 (17.11%)
audio	196,420	9,710 (93.99%)	219,218	5,309 (51.39%)
backup	9,333	67 (0.65%)	7,294	46 (0.45%)
binary	3,194,424	10,300 (99.70%)	965,111	10,326 (99.96%)
code	171,413	2,989 (28.93%)	238,994	2,574 (24.92%)
config	10,969,802	10,300 (99.70%)	2,905,998	10,326 (99.96%)
cryptography	10,770	1,801 (17.43%)	12,098	2,293 (22.20%)
database	46,120	4,922 (47.64%)	161,343	4,803 (46.50%)
game	220,644	3,484 (33.72%)	292,021	3,456 (33.46%)
image	7,927,423	10,297 (99.67%)	3,138,408	10,319 (99.89%)
split	87,939	1,066 (10.32%)		
spreadsheet	26,208	347 (3.36%)	11,311	372 (3.60%)
system	494,928	10,249 (99.21%)	2,145,176	10,326 (99.96%)
text	258,319	7,631 (73.87%)	178,725	5,683 (55.01%)
video	68,333	2,734 (26.46%)	70,015	1,992 (19.28%)
web	281,587	7,004 (67.80%)	247,478	6,724 (65.09%)
other	751,212	8,628 (83.51%)	690,166	3,333 (32.26%)

In other cases, the binaries came from cross-platform libraries and included Windows executables alongside the mobile versions.

Android and Java on iOS

We found `.apk`, `.dex`, and `.jar` files not only in Android apps but also in 57 iOS apps (0.55%). None of these are directly executable on iOS [88]. Of these apps, 43 used MobiVM [228], a framework that allows Java-based development for iOS via AOT compilation. Among the remaining 14, we found `.dex` and `.jar` files added by libraries in 11 apps. In two apps, the included `.apk` files contained animation assets rather than compiled code. The last app included a `gradle-wrapper.jar` file.

We also searched in Android apps for files with the MIME type `x-mach-binary`, which typically indicates binaries for Apple platforms. We found them in 196 Android apps (1.90%). Libraries included these files to run on multiple OSes.

Case Study: PayPal Business The PayPal business iOS app [263] contained the file `gradle-wrapper.jar` belonging to a project that used gradle to generate Java

code files with default values. Comments hinted that they also used the resulting code files to generate Swift files for iOS.

The project also contained URLs of their internal Git and artifactory. Additionally, it revealed a bug-tracking URL. Attackers could use that information for targeted social engineering attacks.

4.4.2 Source Code and Scripts

App Code

We found source code files in 28.93% of Android and 24.92% of iOS apps. However, not all code files are equally relevant. We distinguished between potential app code, e.g., Java, Kotlin, Swift, or C++, and scripts, e.g., Python or Shell. To focus on developer-authored code rather than libraries, we excluded files appearing in more than two apps. We chose this threshold because our dataset contains both Android and iOS versions. Further, we only considered code from programming languages with at least ten files per app. We empirically determined this threshold after observing that apps with fewer files generally include them as components of libraries or package management systems. For instance, we found apps containing `Package.swift`, which we separately discuss in Section 4.4.2.

We found code files meeting our criteria in 73 Android (0.71%) and 34 iOS apps (0.33%). Java source files were the most common on Android, appearing in 63 Android apps (0.61%). Additionally, one iOS app (0.01%) also contained Java code files. On iOS, Swift files were most prevalent, found in 23 apps (0.22%).

Case Study: Audible In the Audible app [58], we found Swift code labeled `AlexaKit`, used to integrate Amazon Alexa. Notably, it included debugging code that sent error logs to two email addresses. This code was removed in the version we downloaded in 2024.

Case Study: Banking App The iOS banking app of Raiffeisen Romania [277] included 152 Swift files containing the code of app features. This exposed code could aid in reverse engineering and increase the risk of targeted social engineering attacks, as files included developer names in their header comments.

Scripts

We separated *web* related scripts, e.g., JavaScript or TypeScript, as these are commonly used in mobile apps to render content via WebViews or similar components. Beer et al. [65] reported that 80% of Android apps use Custom Tabs, another form of in-app browsing. We found web-related files in 7,004 Android (67.80%) and 6,724 iOS apps (65.09%). The numbers may be lower than expected because apps can load web resources directly from the Internet without bundling them in the app package.

Lua, Python, and Shell Scripts We found scripts in 144 Android (1.39%) and 294 iOS apps (2.85%). The most common scripts were shell scripts, which we discovered in 40 Android (0.39%) and 158 iOS apps (1.53%), followed by Lua scripts in 70

Android (0.68%) and 91 iOS apps (0.88%), and Python scripts in 24 Android (0.23%) and 50 iOS apps (0.48%).

Developers often use shell scripts during development and sometimes forget to remove them before release. For example, we found build scripts in the iOS versions of the iRobot [181] and Microsoft Whiteboard [223] apps. These scripts did not contain credentials, as they loaded secrets via environment variables, a best practice to prevent leaks in code repositories [63]. As these cases show, this approach also helps prevent secret exposure in mobile apps. Unlike public code repositories, where code is shared intentionally, these script leaks likely happened unintentionally.

We found similar cases involving Python scripts. For instance, the Android version of the Expedia app [114] included `github_utils.py`, used for opening pull requests, and to retrieve domain information `fetch_site_configs.py`.

On iOS, the Firefox app [232] bundled a *SyncIntegrationTests* directory containing six Python test scripts. The game JellyBlast [336] included Python scripts used to enable debug features for users.

Unlike Python and shell scripts, Lua scripts are often part of the app's core code, as they are used by mobile development libraries such as MoonSharp for Unity [231] and Corona SDK [91]. In addition to previously reported Lua usage, we found compiled Lua scripts in 174 apps, 80 Android apps (0.77%) and 94 iOS apps (0.91%).

The role of these scripts becomes clearer when comparing their presence across platforms. Of the 181 apps that included shell scripts on at least one platform, only 17 (9.39%) had them on both. For Python, 63 apps included scripts on at least one platform, but only 11 (17.46%) did so on both. In contrast, 51 apps (46.36%) included Lua scripts on both platforms, and 71 apps (68.93%) had compiled Lua scripts on both. This pattern indicates that, typically, Lua scripts are intentionally included as part of the app's functionality, while Python and shell scripts are more likely to have been bundled unintentionally during development.

Dependency Management

We found dependency management files in 372 Android apps (3.6%) and 697 iOS apps (6.75%). An overview of these results is shown in Table 4.2.

These files can reveal the specific versions of libraries used, which helps attackers identify outdated components with known vulnerabilities [101, 274]. They may also reference internal repositories. If these references are misconfigured, they can enable dependency confusion attacks in which an attacker publishes a package with the same name to a public repository, such as NPM [249]. If the dependency manager prioritizes public sources, it may fetch the malicious package instead of the intended internal one. This can lead to remote code execution as the attacker is able to include a pre-installation script in the package [68].

CocoaPods CocoaPods [85] is a popular dependency management system for Swift and Objective-C [295]. In our analysis, we identified three file types that disclose dependency details: *Podfile* files appeared in 92 apps (0.89%), *Podfile.lock* in 24 apps (0.23%), and **.podspec* files in 110 apps (1.06%). The *Podfile* defines which libraries (pods) the app uses, their versions, and the repositories from which they should be fetched. During installation, CocoaPods generates a *Podfile.lock* to store the

Table 4.2: Number of dependency management files in our dataset. We summarized all file types that occurred in fewer than 25 apps and label them as *Other*. The number of findings in 2024 is related to the reduced number of 8,702 available Android and 9,212 iOS apps. We uploaded the full table [302].

	2023		2024	
C/C++/C#	7 (0.07%)	91 (0.88%)	5 (0.06%)	69 (0.75%)
CMakeLists.txt	3 (0.03%)	56 (0.54%)	2 (0.02%)	43 (0.47%)
Other	4 (0.04%)	40 (0.39%)	3 (0.03%)	32 (0.35%)
Dart	12 (0.12%)	9 (0.09%)	16 (0.18%)	18 (0.20%)
Go		3 (0.03%)		3 (0.03%)
Java	233 (2.26%)	7 (0.07%)	172 (1.98%)	6 (0.07%)
build.gradle	4 (0.04%)	6 (0.06%)	2 (0.02%)	6 (0.07%)
pom.xml	229 (2.22%)	1 (0.01%)	169 (1.94%)	
Other	1 (0.01%)		1 (0.01%)	3 (0.03%)
Python	3 (0.03%)	4 (0.04%)	2 (0.02%)	7 (0.08%)
Ruby	6 (0.06%)	13 (0.13%)	5 (0.06%)	12 (0.13%)
Swift	3 (0.03%)	256 (2.48%)	4 (0.05%)	263 (2.85%)
Package.swift	3 (0.03%)	28 (0.27%)	4 (0.05%)	52 (0.56%)
Podfile		92 (0.89%)		78 (0.85%)
*.podspec		110 (1.06%)		100 (1.09%)
Other		48 (0.46%)		28 (0.30%)
Web	121 (1.17%)	336 (3.25%)	117 (1.34%)	234 (2.54%)
bower.json	50 (0.48%)	55 (0.53%)	40 (0.46%)	43 (0.47%)
package.json	107 (1.04%)	320 (3.10%)	104 (1.20%)	222 (2.41%)
package-lock.json	21 (0.20%)	25 (0.24%)	16 (0.18%)	24 (0.26%)
Other	20 (0.19%)	22 (0.21%)	19 (0.22%)	18 (0.20%)
Total	372 (3.60%)	697 (6.75%)	310 (3.56%)	588 (6.38%)

exact versions used, ensuring consistent builds. The *.podspec* files describe individual libraries, specifying their name, version, source, and dependencies.

To assess the risk of dependency confusion attacks, we checked whether pod names used in analyzed apps were unclaimed in the public CocoaPods repository. We found that 81 iOS apps (0.78%) referenced at least one of 97 available pod names. Attackers could register them, enabling the execution of malicious code on developer devices or build servers when installing or updating dependencies.

Carthage and SwiftPM Carthage [75] and SwiftPM [321] are alternative package managers for Swift and Objective-C. Unlike CocoaPods, they do not rely on a central dependency repository which makes them immune to dependency confusion attacks. However, other risks remain, e.g., attackers might use library and version information to identify outdated libraries with known vulnerabilities.

We found *Package.swift* files in 3 Android (0.03%) and 28 iOS apps (0.27%). More apps contained *Package.swift* files than were flagged as including Swift source code. This discrepancy results from our classification method, which requires at least ten unique files in a single programming language to consider an app as containing source code, which not all of these apps had.

Similar to *Podfile.lock*, the *Package.resolved* file records the dependency versions which we found in 7 apps (0.07%). Further, 17 iOS apps (0.16%) contained *Cartfiles* and 4 (0.04%) *Cartfile.resolved*.

Gradle and Maven We also discovered Android-related package management files, including *pom.xml*, *build.gradle*, and *gradle.lockfile*. Specifically, *pom.xml* appeared in 229 Android apps (2.22%) and 1 iOS app (0.01%), *build.gradle* in 4 Android (0.04%) and 6 iOS apps (0.06%), and *gradle.lockfile* in 1 Android app (0.01%).

The Maven *pom.xml* files contained only library descriptions of well-known libraries, while the *build.gradle* files mostly included automation scripts rather than dependency declarations. As a result, their potential to expose sensitive information was limited.

Web Unlike Java and Swift-related package managers, which we mostly found on a single platform, package managers of various web technologies appeared on both platforms. As discussed in Section 4.4.2, apps frequently embed web code to streamline cross-platform development, reducing the need to implement features twice. Still, we observed a higher prevalence of web-related dependency management files in iOS apps, for example, *package.json* was found in 107 Android (1.04%) and 320 iOS apps (3.10%).

To assess the risk of dependency confusion, we analyzed *npm* package names and found 17 that were unregistered in the public repository, spanning six apps. These included three finance apps, two from Disney, and one health-related app.

4.4.3 Misc

Dotfiles and Dotdirectories

Hidden dotfiles are often used for configuration data in development environments but are rarely needed in production apps. Jungwirth et al. [188] found that 73.6% of code repositories leak potentially sensitive data through dotfiles.

Dotfiles In contrast to code repositories, dotfiles in released apps likely result from oversight. Still, we found them in 126 Android apps (1.22%) and 869 iOS apps (8.41%). The most frequent were `.gikeep` (701 files across 21 Android and 202 iOS apps), `.DS_Store` (378 files in 33 Android and 161 iOS apps), and `.gitignore` (300 files in 21 Android and 202 iOS apps).

Dotdirectory We found dotdirectories in 8 Android apps (0.08%), containing a total of 1,401 files, and in 922 iOS apps (8.93%), containing 3,190 files. The most common was `.AppLovinQualityService`, present in 768 iOS apps. AppLovin is a monetization library available for both Android and iOS [55]. This directory

consistently included two files, called: `AppLovinQualityService.json` and `AppLovinServiceRanges.json`. Despite their suffix, these files contain binary data with high entropy (e.g., 7.9), indicating encryption.

Other dotdirectories included `.monotouch` (found in 61 iOS apps), `.vscode` (2 Android and 29 iOS apps), and `.swiftpm` (19 iOS apps).

Case Study: Epic Seven In an iOS game app [317], we found an Apache subversion directory `.svn`, which contained metadata about previous code changes.

Markdown

Markdown is a popular markup language for formatting text [213]. We found Markdown files in 840 Android (8.13%) and 1,004 iOS apps (9.72%). These files appeared for three main reasons: (1) apps used them to store text that they render at runtime; (2) they were included as part of third-party dependencies—e.g., Markdown files in the `node_modules` directory were present in 30 apps; (3) developers used them for internal documentation but forgot to exclude them from production builds. We classified these cases based on file location and whether the files were referenced in binaries or resources.

Most Markdown files originated from third-party dependencies, found in 734 Android apps (7.10%) and 636 iOS apps (6.16%). Files likely included unintentionally were found in 91 Android apps (0.88%) and 398 iOS apps (3.85%).

Case Study: Decathlon Connect In the *Decathlon Connect* app [97], we found Markdown files used for internal documentation, including component descriptions and onboarding materials. These files pose a security risk, as they contained developer names, email addresses, and internal URLs, information that could aid in reverse engineering or targeted social engineering attacks.

Case Study: Scan & Translate+ In the *Scan & Translate+* [2] app, we found general documentation that included a link to an internal repository, as well as a Markdown file named `REMOTE_SERVICES.md`. This file listed various services along with their credentials, including those used for premium features. If abused, these credentials could allow unauthorized access to paid services, resulting in financial costs for the app provider.

AI assets

The rise of AI and machine learning has introduced new security risks, like model stealing, where attackers extract and reuse trained models [148, 176]. While running AI tasks on-device improves privacy by avoiding data transmission to remote servers, extraction of models bundled with an app is straightforward. We found AI-related files in 755 Android apps (7.31%) and 491 iOS apps (4.75%). Not all of these are sensitive, many are publicly-available, pre-trained models offering specific features. For example, a Google model for barcode scanning [149] appeared in 347 apps, and a face detection model [150] in 160 apps. However, we found unique models in 262

Table 4.3: Coded content of responses to the responsible disclosure. Two researchers manually coded the responses. Afterwards, they compared their codebooks, merged them, and discussed disagreements. We received 77 non-automated responses, 37 about the app content, and 40 about secrets.

Responses	**Files**	**Secrets**	**Total**
Aware of issue		2 (5.00%)	2 (2.60%)
Collaboration		1 (2.50%)	1 (1.30%)
Fixed	5 (13.51%)	8 (20.00%)	13 (16.88%)
Fix to expensive	1 (2.70%)		1 (1.30%)
Forward to team	16 (43.24%)	21 (52.50%)	37 (48.05%)
No critical issue	1 (2.70%)	1 (2.50%)	2 (2.60%)
No issue	1 (2.70%)	1 (2.50%)	2 (2.60%)
Old version/legacy		4 (10.00%)	4 (5.19%)
Questions	9 (24.32%)	3 (7.50%)	12 (15.58%)
Resubmit	4 (10.81%)		4 (5.19%)
Security address	6 (16.22%)	1 (2.50%)	7 (9.09%)
Will fix	3 (8.11%)	6 (15.00%)	9 (11.69%)

apps, which may include custom-trained ones. For apps available on both platforms, we counted each model only once.

4.4.4 Responsible Disclosure

We contacted developers when their apps contained source code (in January 2025) or dependency management files (in March 2025). We did not automatically report the presence of markdown files, scripts, and dotfiles, as manual inspection showed that these rarely contained sensitive data. If any of these files included valid credentials, we disclosed them as described in Section 4.5.3. Additionally, we did not report `pom.xml` and `build.gradle` files, as manual review indicated they carried minimal risk (see Section 4.4.2). Nevertheless, even though such files might not be sensitive, their inclusion increases app size and storage use on the user's devices.

In total, we sent 661 disclosure emails on findings in the 2024 dataset: 117 regarding exposed source code, 535 about dependency management files, and 9 covering both. We received mail delivery failures for 69 messages. Within two weeks, 37 developers replied with non-automated responses. In Table 4.3, we provided an overview of the responses. These were evaluated by two researchers who independently analyzed their content. Overall, we had a positive impression of the responses: five (13.51%) stated that they had already fixed the issue, and three (8.11%) mentioned that they plan to fix it. However, one developer responded that implementing a fix would be too expensive (2.7%), another argued that the issue was not critical enough (2.7%), and one stated that it posed no issue in their case (2.7%).

Table 4.4: Number of valid and invalid credentials in context of the number of credential candidates of the same type detected per file. To minimize validation requests, we did not attempt to validate credentials when we found 15 or more credentials of the same type in a single file.

# Credential-type per File	Valid	Invalid	Valid Ratio
1	327	5,374	5.74%
2	55	1,393	3.80%
3	11	611	1.77%
4	16	501	3.09%
5	3	262	1.13%
6	2	284	0.70%
7		162	
8		126	
9	2	181	1.09%
>9 & <15		854	

Takeaways

To answer RQ1: *What files do mobile apps contain?*, we found:

- Mobile apps include numerous file types, including unexpected ones, e.g., Windows binaries and shell scripts;
- Files likely included unintentionally, such as source code, dependency management files, or internal documentation written in Markdown. Those files can expose sensitive security and privacy-related information;
- Even when these files do not pose security risks, they increase app size and storage usage on user's devices.

4.5 Secrets

After examining what types of files mobile apps include, we now address *RQ2: What secrets do developers distribute in mobile apps?*

In this section, all credential counts refer to the combined datasets from 2023 and 2024. We include both years to capture all valid credentials present at the time of analysis. A separate breakdown by year is provided in Section 4.7 to explore temporal trends.

4.5.1 Hardcoded Credentials

Our rule-based detection identified 26,380 potential credentials. We applied a heuristic that discards any file containing 15 or more credentials of the same service type, as we observed that rules producing numerous findings within a file typically represent FPs, e.g., hash values mistakenly matching token patterns. To minimize unnecessary

Table 4.5: Number of *valid* and *invalid* credentials discovered for selected services. We summarized the other 55 services for which we found at least one valid credential as *Other*. We provide the full table online [304].

	Valid	Invalid	Valid Ratio
AWS	58	85	40.56%
Alibaba	10	9	52.63%
Azure	1	52	1.89%
FTP	1		100%
GCP	1	24	4%
GitHub	9	15	37.5%
RazorPay	3	30	9.09%
Squareup	3		100%
Stripe	4	4	50%
Yelp	1	16	5.88%
Other	325	1.546	17.37%

validation requests for ethical reasons, we explored different thresholds by incrementally increasing the cut-off and manually reviewing files containing potentially valid secrets that would have been excluded. This reduced the number of candidates to 10,164. Table 4.4 summarizes the distribution of valid and invalid credentials based on the number of same-type credentials found per file. The majority of valid credentials (327) came from files containing only one instance of a specific credential type. In contrast, only 23 valid credentials were found in files with more than three of the same type, demonstrating our heuristic's effectiveness in filtering out FPs.

The 10,164 potential credentials identified came from 258 different categories. After completing the validation requests, we confirmed 416 as valid (4.09%), spanning 65 services. In Table 4.5, we provide details for 10 services with confirmed valid credentials. The FP rate varied significantly by credential type. This has three reasons: (1) Some credential types are rarely used in mobile apps compared to code repositories (for which TruffleHog was originally developed). For instance, none of the 46 Dockerhub tokens were valid, while all three Squareup tokens were. Squareup provides financial services for mobile payments. Thus, its scope aligns with mobile apps. (2) All Squareup tokens used a consistent prefix (`sq0idp-`), making them easier to detect accurately. (3) Some tokens consist of multiple parts, as seen with YouTube and AWS credentials. For both, one part can be identified easily due to its prefix, while the other lacks a clear pattern, leading to misclassification of unrelated strings.

The valid credentials stem from 200 Android (1.94%) and 292 iOS apps (2.83%). These totals do not match the overall number of valid credentials because 29 credentials appeared in multiple apps ($\bar{x} = 1.12$, $sd = 0.56$, $max = 8$). Android and iOS versions of the same app were counted only once. In addition, we found 106 credentials in 117 apps across both platforms, often due to developers reusing the same credentials. Overall, 67 apps contained more than one valid credential ($\bar{x} = 1.18$, $sd = 0.52$, $max = 5$).

Table 4.6: Number of apps with valid credentials. We categorized the apps by their Android installation count and grouped categories below 1,000 installations into *0-1,000*. Further, we show in ∪ the number of apps where we found a credential in the Android or iOS app.

			∪
1,000,000,000+	1 (0.01%)	1 (0.01%)	2 (0.02%)
500,000,000+	0 (0.00%)	4 (0.04%)	4 (0.04%)
100,000,000+	8 (0.08%)	8 (0.08%)	11 (0.11%)
50,000,000+	7 (0.07%)	12 (0.12%)	14 (0.14%)
10,000,000+	26 (0.25%)	50 (0.48%)	59 (0.57%)
5,000,000+	18 (0.17%)	31 (0.30%)	38 (0.37%)
1,000,000+	29 (0.28%)	48 (0.46%)	60 (0.58%)
500,000+	14 (0.14%)	22 (0.21%)	30 (0.29%)
100,000+	35 (0.34%)	44 (0.43%)	59 (0.57%)
50,000+	11 (0.11%)	10 (0.10%)	15 (0.15%)
10,000+	12 (0.12%)	16 (0.15%)	22 (0.21%)
5,000+	1 (0.01%)	3 (0.03%)	3 (0.03%)
1,000+	14 (0.14%)	17 (0.16%)	21 (0.20%)
0 - 1,000	24 (0.23%)	26 (0.25%)	35 (0.34%)
Total	200 (1.94%)	292 (2.83%)	373 (3.61%)

Popular Apps In Table 4.6, we break down our findings by the number of downloads reported on the Play Store. Each app was counted only once, even if the finding occurred in both its Android and iOS versions. Since the App Store does not provide installation data, we used Play Store downloads as a proxy for iOS apps. The median minimum installation count of apps with secrets was 1 million (MAD 990,500). Notably, two apps had over 1 billion installs, showing that even widely used apps are affected by credential leaks.

Git Credentials

For Git credentials, we additionally performed a GET request to check repository access and assess the potential impact. Valid tokens could allow attackers to push malicious code, regardless of whether the repository is public or private. Private repositories pose an additional risk by exposing internal company data, enabling cloning and republishing of apps.

We identified 13 valid Git credentials: nine for GitHub, three for BitBucket, and one for Gerrit. We also found nine GitLab tokens matching the prefix `glpat-`, but none were valid. This may be due to their use in self-hosted GitLab instances. However, we found no private GitLab URLs in the files containing the identified tokens.

These 13 valid tokens granted access to 218 public and 2,440 private repositories. One token alone had access to 1,140 private and 17 public repositories. Another

Table 4.7: File categories of valid credentials. *Web* indicates that we found credentials in JS files. *Resources* refers to credentials found in `Info.plist`, `Manifest`, or Android resource files. *Cross-platform* denotes files related to cross-platform or game libraries. Note that we found two Android and two iOS credentials in two file categories each. Overall, we found 106 credentials on both platforms.

Binary	133 (61.86%)	184 (59.93%)	277 (66.59%)
Config	26 (12.09%)	15 (4.89%)	33 (7.93%)
Cross-platform	17 (7.91%)	22 (7.17%)	25 (6.01%)
Resources	5 (2.33%)	30 (9.77%)	35 (8.41%)
Unintended		11 (3.58%)	11 (2.64%)
Web	36 (16.74%)	47 (15.31%)	50 (12.02%)
∑ Credentials	215	307	416

appeared particularly sensitive, providing access to six private repositories belonging to a bank.

We manually investigated the causes for including Git credentials in mobile apps. The two main reasons were: (1) leftover code in the app intended to trigger continuous integration workflows, and (2) the inclusion of dependency management files.

Files Containing Credentials

Table 4.7 presents an overview of the file categories in which we identified hardcoded credentials. We used the category *Web* when we found credentials in JS files; *Resources* for those located in `Info.plist`, `AndroidManifest.xml`, or other Android resource files; *Config* for configuration files not tied to the operating system, such as `.json` or `.xml`; *Cross-platform* for files associated with cross-platform or game libraries; *Binary* for common compiled Android or iOS binary files; and *Unintended* for files that are typically not expected in released apps, such as `.gitlab-ci.yml`, `.xcconfig`, or shell scripts.

We found the majority of credentials in binary files, 133 on Android (61.86%) and 184 on iOS (59.93%). However, the second most frequent category is *Web*, with 36 credentials found in Android (16.74%) and 47 in iOS apps (15.31%).

This result highlights limitations in traditional static analysis methods, such as VSA, which would usually overlook credentials embedded in web assets or require substantial customization to detect them. The situation is similar for credentials in cross-platform libraries, as each library may require a tailored analysis approach. Moreover, we found 11 credentials (3.58%) in files likely included unintentionally, making them especially difficult to detect using standard approaches.

We also identified 106 credentials shared across both platforms, with 93 (87.74%) located in the same file category. For the remaining 13 (12.26%), discrepancies in file locations were observed. For example, a Twitter consumer key was found in a `Prototype.xcconfig` on iOS, while on Android, it was stored directly in the app's

bytecode. Similarly, an Infura key appeared in an Android bytecode but in a JS file on iOS. The other differences were limited to binary, resource, and config file categories.

4.5.2 JSON Web Tokens (JWTs) and Private Keys

Unlike hardcoded credentials discussed in the previous section, we did not test the validity of found JSON Web Tokens (JWTs) and private keys. The presence of private keys in mobile apps inherently violates their purpose, which is to remain confidential. The security impact depends on how the key is used in the app. For example, in a payment app [264], the private key was solely used for obfuscation, e.g., encrypting payloads before transmission. While knowing the key simplifies protocol reverse engineering, it does not have further security implications.

We identified 212 private keys across 433 Android (4.19%) and 180 iOS apps (1.74%). Among these, 29 keys appeared in more than one app, and six were found in over 100 apps. These frequently recurring keys originated from default or test values included in libraries such as the Google HTTP Client Library [146]. For this evaluation, we counted findings in the Android and iOS versions of the same app as a single instance.

The situation for JWTs is similar. These tokens are commonly used for user authentication and authorization, and leaking them can enable unauthorized access. In total, we found 1,378 tokens across 1,018 Android (9.85%) and 569 iOS apps (5.51%).

A significant portion of this discrepancy stems from a single token in the 2024 dataset, included by the Unity library [337], which appeared in 645 Android apps. Excluding this token, the number of affected Android apps drops to 385 (3.73%), while the iOS count remains unchanged, resulting in more iOS apps with at least one token. We also identified 75 tokens present in multiple apps, primarily due to reuse by the same developer.

To further analyze the JWTs, we parsed their contents. Of the 1,378 tokens, 859 (62.34%) did not have an expiration date set, 69 (5.01%) were still valid for more than ten years, and 392 (28.45%) were expired. We also searched for the keyword *admin* and found it in 202 tokens (14.66%), only 11 of which were already expired.

In some cases, the credentials appeared unused, suggesting they may have been unintentionally included for debugging or legacy reasons. Nonetheless, their presence can aid manual analysis and help uncover potential security or privacy issues. Automatically determining the purpose of a private key or JWT is challenging, as it requires understanding or executing the surrounding code, which we leave to future work, see Section 4.8.

4.5.3 Responsible Disclosure

We automatically sent 422 emails to developers in January 2025 to report hardcoded credentials. Of these, 12 messages (2.84%) failed to deliver. In total, we received 40 non-automated responses.

We provided insights into the responses in Table 4.3. Positively, eight (20%) responded that they fixed it already, and six (15%) that they will fix it. Interestingly,

four (10%) answered that the issue comes from legacy code or an old app version, and two (5%) mentioned that they are aware of the issue.

Takeaways

To answer RQ2: *What secrets do developers distribute in mobile apps*, we showed:

- We found valid credentials for 65 different services, including unexpected ones such as GitHub;
- Credentials appeared not only in app binaries but also in files likely included unintentionally. We identified 11 such cases;
- Developers embedded private keys and JWTs in their apps. Only 27.16% of the tokens had already expired, and 18.52% contained indications of administrator privileges.

4.6 Platform Differences

To answer *RQ3: How does the situation differ between Android and iOS apps?*, we compare the contents of the apps and the credentials found. We highlight key differences and explore potential factors contributing to discrepancies.

4.6.1 Files

As shown in Table 4.1, Android and iOS apps share similar distributions across most file categories. However, *archives* and *text* categories deviate from this trend, appearing more frequently in Android apps than in iOS apps (77.21% vs. 17.1%; $p < 0.01$, $d = 1.16$ and 73.86% vs. 55%; $p < 0.01$, $d = 0.33$, respectively). The higher prevalence of archive files in Android is primarily due to the OkHttp3 library which includes the file `publicsuffixes.gz` present in 72.75% of Android apps. Similarly, the increase of text files results from the presence of the `androidsupportmultidexversion.txt` in 47.13% of Android apps.

Source Code, Scripts, Markdown As detailed in Section 4.4.2, our analysis identified 73 Android apps (0.71%) and 34 iOS apps (0.33%) containing source code files that may reveal app code ($p < 0.01$, $d = 0.04$). In contrast, a greater share of iOS apps included potentially unintended files: scripts (1.39% vs. 2.85%, $p < 0.01$, $d = -0.09$), markdown files (8.13% vs. 9.72%, $p < 0.01$, $d = -0.05$; non third-party 1.64% vs. 4.09%, $p < 0.01$, $d = -0.11$), dotfiles (1.22% vs. 8.41%, $p < 0.01$, $d = -0.25$), and dotdirectories (0.08% vs. 8.92%, $p < 0.01$, $d = -0.31$). We observed similar trends in the 2024 dataset, as we discuss in Section 4.7.

Factors Contributing to Platform Differences Two main factors explain these differences. First, Android apps are more accessible for analysis, e.g., due to the distribution of app bundles via third-party platforms such as APKPure [29] and APKMirror [28]. As a consequence, previous research primarily focused on Android.

Table 4.8: Comparison of valid credentials found in Android and iOS apps. We display services with large differences separately. ∩ indicates credentials discovered on both platforms. We also report the p-value of a dependent t-test and the effect size (d) calculated using Cohen's d. We provide the complete table online [301].

Services with Major Differences			∩	p	d
AWS	31	47	20	<0.01	-0.02
Flickr	10	5	2	0.29	0.01
Github	4	6	1	0.41	-0.01
Infura	5	12	4	0.02	-0.02
OpenAI	1	13	0	<0.01	-0.02
OpenWeather	5	13	3	<0.01	-0.02
SlackWebhook	17	49	13	<0.01	-0.03
Other	142	162	63	0.01	-0.02

Second, Apple encrypts iOS app binaries [350], which may cause confusion among developers about the encryption status of additional files in the app bundle. One developer responding to our disclosure was surprised that Apple did not take any measures to protect dependency management files that had been unintentionally bundled with their app.

However, even if Apple would encrypt all files, they must be decrypted at runtime. This makes it still possible to extract and analyze the content using jailbroken iOS devices.

4.6.2 Hardcoded Secrets

Overall, fewer Android apps contained valid credentials compared to iOS apps (1.94% vs. 2.83%, $p < 0.01$, $d = -0.04$). This difference is also reflected in the absolute number of valid credentials found: 230 in Android and 352 in iOS apps. We identified 106 credentials shared across platforms, originating from 117 apps that included at least one common credential in both their Android and iOS versions.

We identified fewer credentials than apps that share them across the platforms because developers use the same credentials in multiple apps, see Section 4.5.1.

Services When we look at the services with valid credentials, we learn that a significant difference results from five services. iOS apps contained more valid credentials for AWS, Infura, OpenAI, OpenWeather, and SlackWebhook. In contrast, Android apps included more valid credentials for Flickr. The remaining services yielded 142 valid credentials in Android apps and 162 in iOS apps. We provide an overview in Table 4.8. The table also shows the number of credentials found on both platforms, underscoring the importance of analyzing both Android and iOS apps. For instance, of the nine valid GitHub credentials, five were found exclusively in iOS apps and three

Table 4.9: Comparison of valid credentials found in apps with at least 100 million installations from 2023. ∩ indicates credentials discovered in an app's Android and iOS version.

100,000,000+ in 2023			∩
AWS	3	3	2
Alchemy		2	
BrowserStack		1	
Infura	1	3	1
LaunchDarkly		1	
PubNubSubscriptionKey	1	1	1
SlackWebhook		1	
TwitterConsumerkey	1	3	
URI		1	
Total	6	16	4

only in Android apps. Consequently, we would have missed a significant number if we had only analyzed a single platform.

The most pronounced difference occurred with SlackWebhook credentials. We identified 49 such credentials in iOS apps, 13 of which also appeared in the Android version, while only four were exclusive to Android ($p < 0.01$, $d = -0.03$). SlackWebhooks allow apps to send messages to Slack channels, with the severity of exposed credentials depending on the channel's purpose. In general, we consider this a lower-risk issue. The AWS findings also showed a notable difference across platforms. We found 11 credentials exclusively in Android apps, 27 exclusively in iOS apps, and 20 shared between both ($p < 0.01$, $d = -0.02$).

Files with Credentials A comparison of the file types containing credentials revealed two key differences between the two OSes. On Android, credentials were more frequently found in configuration files (12.09% vs. 4.89%), whereas on iOS, they appeared more often in system resource files (2.33% vs. 9.77%). Additionally, we identified 11 credentials in files likely included unintentionally in 9 iOS apps. These included files such as `.gitlab-ci.yml`, `*.xcconfig`, and shell scripts. We provide a detailed breakdown in Table 4.7.

Popular Apps In 2019, Google launched a bug bounty program for Android apps with over 100 million installations [276], including third-party apps. However, the program was discontinued in August 2024, just before we collected our dataset in October [276].

We report the number of valid credentials found in the 2023 dataset for apps with at least 100 million installations in Table 4.9. With two findings exclusively in Android apps, 12 only in iOS apps, and four in both, the platform difference is significant ($p = 0.02$, $d = -0.1$). Two of the iOS findings resulted from unintentionally included files. In one case, we discovered a SlackWebhook token within a commented-out block of a Swift file in a game app [238]. In another, a BrowserStack token appeared

in a Java test file included with the Wattpad app [348] as part of test automation code.

Discussion At first glance, Android apps seem to exhibit fewer credential exposures. However, we also found several instances where credentials were present only in the Android version. One likely reason is the better accessibility of Android app bundles, which makes them easier to analyze and thus has historically received more research attention. Additionally, Google's bug bounty program likely had a positive impact on popular apps. In Section 4.7, we revisit this topic to examine how findings shifted after the program ended in 2024. Another explanation for platform-specific issues is that different development teams may be responsible for each version, resulting in inconsistencies in secure coding practices. Overall, our research underscores the importance of analyzing both platforms, as many findings were exclusive to a single platform.

Takeaways

To answer RQ3: *How does the situation differ between Android and iOS apps?*, we showed:

- Unintentionally included files were more common in iOS apps. A potential explanation is Apple's closed environment, which limits accessibility and complicates external security analysis;
- The pattern extended to credential exposures, with more valid credentials identified in iOS apps;
- Despite identifying more issues in iOS apps, the situation on Android is only slightly better. We also found cases where findings appeared exclusively in the Android version of the app.

4.7 Changes in 2024

To answer *RQ4: How did the situation change between 2023 and 2024?*, we examine shifts in the inclusion of files such as code and scripts, and highlight differences related to credentials. To ensure comparability of relative values, we normalize the results based on the updated 2024 dataset sizes of 8,702 Android and 9,212 iOS apps.

4.7.1 Files

A comparison of file categories between the 2023 and 2024 datasets revealed no major overall changes. However, we observed a notable increase in iOS apps flagged for potentially including source code: from 62 in 2023 (0.67%) to 156 in 2024 (1.51%). This increase was more pronounced on iOS ($p < 0.01$, $d = -0.04$), while numbers for Android remained relatively stable ($p = 0.62$, $d = 0.01$), with 66 in 2023 (0.76%) and 76 in 2024 (0.74%).

Similarly, the occurrences of dotfiles and dotdirectories increased more notably in the iOS dataset. In 2023, we identified dotfiles in 126 Android (1.22%) and 869

Table 4.10: Comparison of files included in mobile apps across platforms and collection years. The findings in 2024 are relate to the reduced number of 8,702 Android and 9,212 iOS apps.

	2023		2024	
Dotdirectory	8 (0.08%)	922 (8.92%)	11 (0.13%)	979 (10.63%)
Dotfile	126 (1.22%)	869 (8.41%)	122 (1.40%)	1,006 (10.92%)
Code	73 (0.71%)	34 (0.33%)	66 (0.76%)	62 (0.67%)
Markdown	840 (8.13%)	1,004 (9.72%)	918 (8.89%)	1,071 (10.37%)
Scripts	144 (1.39%)	294 (2.85%)	129 (1.48%)	232 (2.52%)

iOS apps (8.41%). In 2024, the relative number on Android slightly rose to 122 (1.40%, $p = 0.54$, $d = -0.01$), while it did clearly on iOS to 1,006 (10.92%, $p < 0.01$, $d = -0.07$). Dotdirectories increased from 8 (0.08%) to 11 in Android apps (0.13%, $p = 0.25$, $d = -0.01$), while on iOS, the number grew from 922 (8.92%) to 979 (10.63%, $p < 0.01$, $d = -0.04$), as detailed in Table 4.10.

For dependency management files, the number of findings stayed mostly constant, as shown in Table 4.2. The most notable change concerned *package.json* files, which dropped on iOS from 330 in 2023 (3.1%) to 222 in 2024 (2.41%, $p < 0.01$, $d = 0.05$). In contrast, the numbers for Android remained relatively constant, with 107 in 2023 (1.04%) and 104 in 2024 (1.2% $p = 0.47$, $d = -0.01$).

4.7.2 Hardcoded Credentials

In Figure 4.2, we present the number of hardcoded credentials that were valid at the time of testing. Notably, we identified 95 credentials in the 2023 dataset that, despite their removal from the app version downloaded in 2024, remained valid. For example, in the 2023 dataset, the Android version of the app `com.viber.voip` included AWS credentials within a native library. Although these were removed in the 2024 version, the credentials themselves remained valid. This finding is particularly concerning given the app's large user base, with over one billion installations. A possible explanation for this practice is legacy support. Developers may remove credentials from newer app versions but refrain from revoking them to prevent breaking functionality in older versions, which could otherwise become non-functional.

Further, we found 99 credentials newly introduced in the 2024 app versions. The total number of valid credentials remained nearly constant, with 317 valid credentials in 2023 and 321 in 2024 ($p = 0.01$, $d = -0.02$). However, relative to the number of apps analyzed, the numbers slightly increased in 2024, as 1,629 Android and 1,119 iOS apps were not available anymore at the time of the dataset update. The number of valid credentials on Android decreased from 175 in 2023 to 161 in 2024, but the relative share increased from 1.69% to 1.85% ($p = 0.01$, $d = -0.03$). We observed a similar trend for iOS: 236 valid credentials in 2023 and 230 in 2024 (2.28% vs. 2.5%, $p = 0.17$, $d = -0.01$). In general, we assumed a slight increase as we expected some

developers to remove or revoke leaked credentials from newer builds. Still, others also introduce new credentials that have not yet been revoked.

Credential Types We observed an increase in OpenAI credentials, rising from three in 2023 to 11 in 2024 ($p = 0.03$, $d = -0.67$). The reason is likely the increased popularity of generative AI [205].

The number of valid Git credentials also increased from 7 in 2023 to 12 in 2024 ($p = 0.01$, $d = -0.2$). The percentage of valid Git credentials rose from 25% to 36.36%. One possible reason could be that developers tend to revoke Git credentials once they become aware of their exposure.

End of Google's Bounty Program To examine the impact of the end of Google's bug bounty program in August 2024, we compared the number of valid credentials found in apps with over 100 million installations across both years.

In the 2023 dataset, we identified valid credentials in 5 Android and 11 iOS apps exceeding 100 million installations. In 2024, the number of affected iOS apps slightly decreased to 8, while the number for Android increased to 7. The difference between platforms in 2023 was statistically significant ($p = 0.02$, $d = -0.1$), but this was no longer the case in 2024 ($p = 0.29$, $d = -0.05$).

Takeaways

To answer RQ4: *How did the situation change between 2023 and 2024?*, we showed:

- The types of files included in mobile apps remained mostly constant across both years;
- Even if developers removed credentials from apps, it does not necessarily mean they also revoked them;
- We observed a slight increase in credential exposure in popular Android apps, possibly linked to the termination of Google's bug bounty program.

4.8 Limitations and Future Work

Our analysis faces limitations inherent to static analysis. For example, we are unable to detect encrypted credentials or files that are dynamically downloaded during runtime. Future work could address such limitations by triggering decryption routines in apps, e.g., with VSA or dynamic forced code execution. Also, pattern-based secret detection faces limitations. Basak et al. [62] reported a high number of FPs and False Negatives (FNs). We eliminate all FPs from our pattern matching results by remotely validating our findings. Thus, our results represent a lower bound, highlighting the severity of our findings.

Developer responses to our findings regarding dependency management and included source code revealed a lack of awareness that such files could be packaged with production apps. One developer expressed surprise that Apple does not obfuscate or

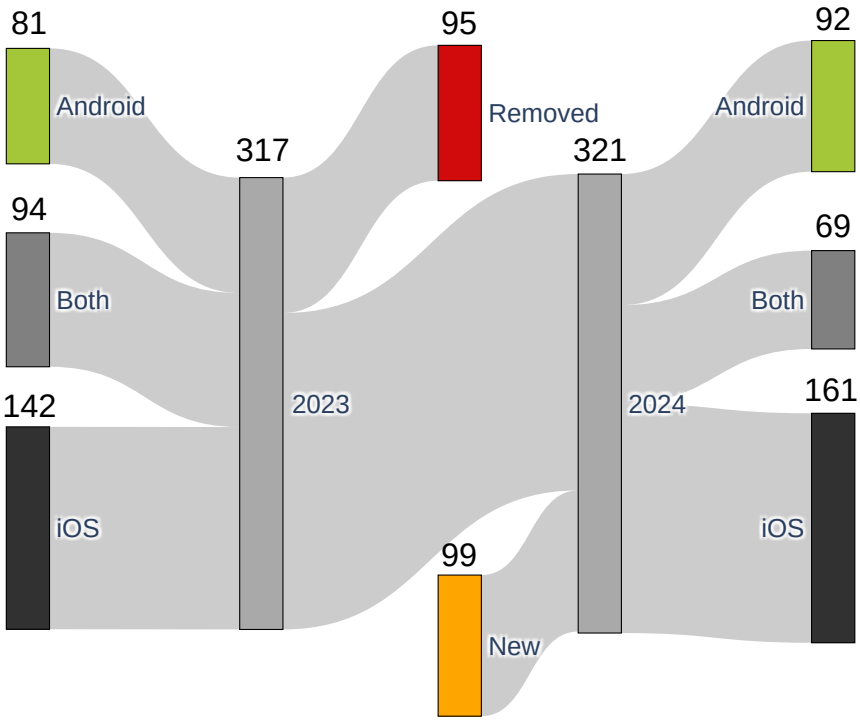


Figure 4.2: Valid credentials discovered in Android and iOS apps. The numbers above the bars indicate the amount of credentials discovered. The *removed* bar represents credentials present exclusively in the 2023 app version. Conversely, the *new* bar highlights credentials detected solely in 2024.

encrypt these files. This suggests that future research should investigate developers' mental models related to file inclusion in mobile apps and explore strategies to improve their awareness.

We did not attempt to detect misconfigurations related to information that developers intentionally include in apps. Prior work has demonstrated the risks of misconfigured services, such as publicly accessible AWS S3 buckets [90]. Detecting such misconfigurations typically requires knowledge of resource identifiers, e.g., bucket names and regions, which mobile apps may reveal. Similarly, Jin et al. [186] showed that improperly configured IoT access policies can lead to data leakage. They identified such issues through analysis of online code repositories. Future work could extend this by inferring misconfigurations from app content.

Future research could target context-dependent secrets, such as JWTs. For these cases, dynamic analysis or AI-based methods could be used to interpret application logic and better understand the context in which values are used.

4.9 Related Work

Secrets in Git Repositories In previous research, pattern-matching approaches have been frequently used to detect secrets in Git repositories [313, 115, 219, 188,

349]. Meli et al. [219] conducted a large-scale longitudinal analysis of public GitHub repositories to identify secrets and private key leaks. Koishybayev et al. [193] highlighted the security risks of CI/CD pipelines executing arbitrary code from untrusted sources and proposed an early warning system to detect security risks, including the exposure of secrets in Git repositories.

Moreover, researchers have explored machine learning techniques to improve secret detection in Git repositories [292, 119, 210, 175, 248]. For instance, Saha et al. [292] used machine learning to reduce false positives and expand the range of detectable secrets.

Others focused on developers' perspectives by analyzing why secrets leak, the challenges developers face in preventing exposure, and mitigation techniques [196, 275, 63].

Analyzing mobile apps for secrets comes with two major differences compared to analyzing code repositories: (1) The analyzed code format differs. While repositories primarily contain source code, mobile apps are distributed in compiled binary form, making extraction and analysis more complex. (2) The purpose of these platforms differs. Repositories facilitate collaboration among developers, whereas apps are built for end-users.

Researchers have also applied regular expression-based detection beyond code repositories. Yadmani et al.[106] employed this method to uncover secrets stored on cloud storage servers. Similar to our work, they validated detected credentials and responsibly disclosed their findings. Dahlmanns et al.[95] applied this approach to Docker container images. However, as with code repositories, the purpose and format of data shared through cloud storage services and container images differ significantly from those of mobile apps.

Mobile Analysis Previous work often focused on privacy aspects of mobile apps, in particular leaks of PII [89, 236, 288, 287] or permission models to protect personal data [283, 297, 358, 209]. In contrast, our study takes a different direction, we focus on secrets embedded in released mobile apps instead of privacy aspects.

Previous work on mobile app analysis researched the threat of hardcoded cryptographic keys and credentials in mobile apps [347, 118, 124, 220, 142]. Schrittwieser et al. [307] identified authentication bypass vulnerabilities in popular messenger apps based on protocol analysis. Zhou et al. [367] used data flow analysis to find email and Amazon AWS credentials in Android apps. Mendoza et al. [220] developed a methodology to identify credential misuse in two iOS SDKs and showed their real-world impact by analyzing 100 apps. Zuo et al. [370] analyzed Android apps to find potential data leaks of cloud APIs due to authentication and authorization issues.

In contrast to existing work, we studied Android and iOS apps to compare the situation on both platforms. Furthermore, we adopted a broader perspective rather than limiting our analysis to specific credentials or those strictly required by the app to function properly. This allowed us to identify sensitive information in files that developers likely included unintentionally, such as source code or documentation, and to examine files originating from cross-platform libraries and embedded web content.

Baskaran et al. [64] and Zhang et al. [362] studied secrets included in so-called super-apps which support third-party mini-programs to extend their functionality.

In contrast, we conducted a large-scale analysis of Android and iOS apps without restricting our scope to the ecosystem of individual super-apps.

4.10 Conclusion

Our analysis revealed that mobile apps often contain unintentionally added files, exposing security- and privacy-sensitive information. These files included markdown documentation, source code, and dependency management files. In particular, dependency management files can pose a critical risk by enabling remote code execution on developer machines or build servers. We identified this threat for 114 dependencies declared in 87 apps (0.84%).

We also uncovered 416 valid credentials spanning 65 different services. Notably, this included 13 Git credentials that provided access to 218 public and 2,440 private repositories. These issues were not limited to niche apps: We identified two apps with over one billion installations, and the median installation count across affected apps was one million.

Overall, our findings showed a higher prevalence of such issues in iOS apps. However, we also documented findings exclusive to either the Android or iOS version of the same app. This underlines the importance of analyzing apps from both OSes.

In some cases, developers removed hardcoded credentials or unintended files from newer app versions. However, this alone does not eliminate the risk. Attackers may have already downloaded earlier versions or still have access to them. Developers should, therefore, carefully audit app bundles before release, and when removing credentials, they must also revoke them. However, this does not always happen. We found 95 credentials that had been removed from the 2024 app version but remained valid.

4.11 Appendix

```
Dear {Developer_Name},

We are security researchers from the University of Vienna studying Android and
↪  iOS apps. During our research, we discovered in your {platform} app ({
↪ app_name}) the following finding:

**Hardcoded Credentials** We found hardcoded AWS credentials (access and
↪ secret key) {CREDENTIALS} (removed parts of the key from the mail) in the
↪ following file {FILE}. The credentials can give attackers unauthorized AWS
↪ cloud resource access and might lead to data breaches on AWS. We recommend
↪ revoking the keys and using Amazon Cognito instead.

Despite best efforts, some findings might be false positives or already fixed
↪ in the latest version.
If you have any further questions or comments, please contact us.
```

5 Supply Chain Insecurity: Exposing Vulnerabilities in iOS Dependency Management Systems

Abstract

Dependency management systems are a critical component in software development, enabling projects to incorporate existing functionality efficiently. However, misconfigurations and malicious actors in these systems pose severe security risks, leading to supply chain attacks. Despite the widespread use of smartphone apps, the security of dependency management systems in the iOS software supply chain has received limited attention.

In this paper, we focus on CocoaPods, one of the most widely used dependency management systems for iOS app development, but also examine the security of Carthage and Swift Package Manager (SwiftPM). We demonstrate that iOS apps expose internal package names and versions. Attackers can exploit this leakage to register previously unclaimed dependencies in CocoaPods, enabling remote code execution (RCE) on developer machines and build servers. Additionally, we show that attackers can compromise dependencies by reclaiming abandoned domains and GitHub URLs.

Analyzing a dataset of 9,212 apps, we quantify how many apps are susceptible to these vulnerabilities. Further, we inspect the use of vulnerable dependencies within public GitHub repositories. Our findings reveal that popular apps disclose internal dependency information, enabling dependency confusion attacks. Furthermore, we show that hijacking a single CocoaPod library through an abandoned domain could compromise 63 iOS apps, affecting millions of users.

Finally, we compare iOS dependency management systems with Cargo, Go modules, Maven, npm, and pip to discuss mitigation strategies for the identified threats.

Publication The work presented in this Chapter resulted from a collaboration with Gabriel Gegenhuber, Edgar Weippl, and Sebastian Schrittwieser. It is currently under submission and a preprint is available online [300]. For this work, I developed the analysis, conducted the evaluation, and wrote the paper. Gabriel Gegenhuber revised the paper and Edgar Weippl provided feedback. Sebastian Schrittwieser advised the work, gave feedback, and revised the paper.

5.1 Introduction

Libraries and dependency management systems form the backbone of software development by allowing developers to integrate existing functionality efficiently. Public

dependency repositories host millions of packages [355]. Due to transitive dependencies, installing a single library can implicitly introduce dozens of additional packages [368]. Each included dependency expands the attack surface, as demonstrated by incidents such as the event-stream compromise [125] and the XZ Utils backdoor [3]. Therefore, a single malicious library can compromise all dependent projects and any device running software that integrates the malicious package.

These attacks exploit the trust developers place in the software supply chain. Supply chain attacks are difficult to detect because they originate from sources that developers inherently trust. For instance, in the SolarWinds compromise, attackers infiltrated the update server and distributed malicious code to over 18,000 customers. The breach remained undetected for at least nine months [352].

Researchers have investigated software supply chain risks through dependency management systems in the past. Zimmermann et al. [368] analyzed the npm ecosystem and demonstrated the cascading impact of vulnerable and malicious libraries through direct and transitive dependencies. Zahan et al. [360] identified weaknesses in npm, such as abandoned domains used for maintainer email addresses, which can facilitate supply chain attacks. Ohm et al. [250] discovered 174 malicious libraries distributed across npm, PyPI, and RubyGems. Birsan [68] showed that package names can leak through GitHub repositories and websites, enabling dependency confusion attacks that result in RCE. Furthermore, Gu et al. [154] identified twelve threats to dependencies hosted on five package repositories and their mirror servers.

Prior work has not analyzed software supply chain risks in the context of iOS apps, despite the deep integration of mobile apps in our daily life [299] and the dominant iOS market share of 58.68% in the United States [319]. Such attacks could impact millions of users worldwide.

Mobile apps are distributed to end-user devices. Thus, attackers can reverse engineer them, revealing insights into an app's supply chain, a scenario known as the MATE threat model [96]. We leverage this property to show that iOS apps expose dependency information, enabling RCE on developer machines and build servers. Moreover, the exposed information creates opportunities for targeted dependency hijacking attacks.

To the best of our knowledge, we are the first to study supply chain attacks on iOS apps. To do so, we evaluate attack surfaces on dependency management systems, focusing on technical issues instead of social engineering techniques, and answer *RQ1: Which supply chain issues do iOS dependency management systems face?* After gaining an understanding of supply chain attacks, ranging from dependency confusion to dependency hijacking attacks, we measure the impact of the discovered vulnerabilities on a dataset of 9,212 iOS apps and open-source projects on GitHub to answer *RQ2: How many iOS apps are vulnerable to supply chain attacks?* Finally, to better understand existing defense mechanisms against the discovered vulnerabilities, we evaluate the security properties of five additional dependency management systems and answer *RQ3: How can dependency management systems protect against the identified vulnerabilities?*

In this paper, we show that iOS apps reveal internal library names, which can lead to RCE on developer machines and build servers. We demonstrate this attack and responsibly disclosed the vulnerability in mobile apps from nine companies.

We further highlight the risks posed by abandoned domains and GitHub namespaces. Attackers can exploit these to hijack iOS dependencies used by at least 162 apps in our dataset, potentially affecting millions of users globally.

By expanding our study to five additional dependency management systems: Cargo, Go modules, Maven, npm, and pip, we discuss potential mitigation strategies and highlight that the identified issues are not confined to the iOS ecosystem.
In summary, we make the following key contributions:

- We show that iOS apps reveal their internal dependency names and versions, which can lead to RCE on developer machines and build servers when dependency management files are misconfigured;
- We demonstrate that dependencies are vulnerable to hijacking attacks, affecting 162 apps (1.76%) in our dataset.
- We analyze five additional dependency management systems to evaluate how they address the identified vulnerabilities. Our findings further show that Go and npm share similar conceptual weaknesses.

5.2 Dependency Management and Attacks

In this section, we describe the recurring properties of dependency management systems and provide an overview of attack scenarios analyzed in this paper.

5.2.1 Dependency Management Systems

Dependency management systems have two aspects that are crucial for the attacks we study in this paper: (1) the location where dependencies are hosted and (2) the mechanisms used to manage library ownership.

Dependency Location

Dependency management systems follow two models for hosting libraries: centralized or decentralized. In the centralized model, dependencies reside in a central repository. For instance, CocoaPods [85] provides a GitHub repository that enables developers to discover and integrate libraries by specifying their names. In contrast, SwiftPM [321] implements a decentralized model that requires developers to define the location of each dependency explicitly. These locations can be given as URLs to Git repositories, such as those hosted on GitHub, or as local file paths.

Since not every dependency should be publicly accessible, dependency management systems with a central repository typically provide mechanisms to host private dependency repositories or specify dependency locations.

Dependency Ownership

The second key aspect is the ownership model of the dependencies.

In centralized systems, the management system is responsible for the ownership, making security dependent on the correctness and robustness of its implementation.

In decentralized systems, ownership is determined by control over the URL that hosts the dependency. When dependencies are hosted on GitHub, for example, ownership depends on GitHub's URL resolution.

5.2.2 Attack Scenarios

In this section, we present supply chain attack scenarios targeting dependency management systems.

Dependency Confusion

Dependency confusion attacks were introduced by Birsan [68], who demonstrated that pip [266], and npm [249] are vulnerable. In this attack scenario, an adversary registers a package in a central dependency repository using the same name as a dependency hosted in a private repository.

If the dependency management system prioritizes public packages over private ones or is misconfigured, it resolves the dependency to the attacker-controlled version instead of the intended internal one. As a result, attackers can achieve RCE on build servers or developer machines that install the dependency, provided the dependency manager supports code execution during dependency installation.

Typo Squatting and Social Engineering

Typo squatting attacks exploit typographical errors that developers make when specifying dependency names [368, 239, 328]. Attackers register packages with names that closely resemble other library names, tricking developers into installing malicious packages instead of intended ones.

Similar strategies include grammatical substitutions and semantic modifications of package names, as shown by Neupane et al. [239].

In contrast to dependency confusion, which exploits technical issues, typo squatting relies on human errors. Thus, it is similar to social engineering, which was also used in the past to hijack dependencies. For example, the XZ Utils backdoor [3] and the event-stream incident [125] succeeded through social engineering. Attackers convinced maintainers to transfer ownership, allowing them to inject malicious code into the packages.

We consider typosquatting and direct social engineering attacks out of scope for this paper, as they exploit human behavior rather than technical vulnerabilities.

Package Hijacking

Another threat is dependency hijacking attacks, in which attackers gain ownership of existing libraries. The feasibility and execution of these attacks depend on the library's hosting location. In centralized repositories, the security depends on the repository's ownership implementation. In decentralized settings, the critical factor is control over the URLs that host the dependencies.

Abandoned GitHub URLs. An attack vector for hijacking dependencies stems from URL ownership [154]. Since dependencies are frequently hosted on GitHub, their URL management plays a critical role.

GitHub allows users to rename or delete their accounts. When a user renames an account, GitHub redirects the old repository URL to the new one. For example, if a user called `conference2025` owns the repository `proceedings` and renames the account to `conference2026`, then `https://github.com/conference2025/proceedings` automatically redirects to `https://github.com/conference2026/proceedings`. GitHub refers to the username as a namespace and the repository name as the image-name. We use this terminology throughout the paper for consistency.

This redirect remains active until another user registers the previously abandoned namespace `conference2025` and creates a repository with the same image-name `proceedings`. At that point, the URL `https://github.com/conference2025/proceedings` resolves to the repository owned by the other user.

If a dependency management system uses this URL to fetch the library, the new user can inject malicious code into the package. From a usability perspective, the redirect mechanism prevents broken links and supports seamless transitions. However, from a security standpoint, it obscures dependency hijacking attacks by preserving apparent functionality even after the namespace has changed.

To mitigate this risk, GitHub recommends updating all URLs after renaming a namespace. Additionally, GitHub permanently retires, and thereby blocks the reuse of specific namespace and image-name combinations. In 2018, GitHub announced it would retire namespaces linked to repositories that had received at least 100 clones within a week [139]. In 2023, this policy was extended to retire any namespace and image-name combination where the associated image has more than 5,000 downloads [140].

Abandoned Domains. Dependencies can be hosted on custom domains. If such a domain expires and becomes available, also referred to as an abandoned domain, an attacker can register it and take control of the dependencies.

Unlike GitHub URL hijacking, where redirects preserve repository availability, dependencies hosted on custom domains become temporarily unavailable during the period between domain expiration and hijacking. This unavailability increases the likelihood that developers or automated systems detect the issue, which could make such attacks easier to identify.

5.3 iOS Dependency Management

In this section, we provide an overview of dependency management systems used by iOS apps, and study attack scenarios which we evaluated on toy examples to answer *RQ1: Which supply chain issues do iOS dependency management systems face?*

5.3.1 CocoaPods

CocoaPods, launched in 2011, uses a central dependency repository and is used in over 3 million apps [85].

```
source 'https://cdn.cocoapods.org/'
source 'https://github.com/Artsy/Spec'
use_frameworks!

target :MyApp do
project 'MyApp.xcodeproj'
    pod 'GoogleAnalytics'
    pod 'OCMock', '~>2.27'
    pod 'Aerodramus', '2.1.4'
    pod 'Artsy+UIFonts', '2.1.4'
end
```

Listing 5.1: A `Podfile` that is vulnerable to dependency confusion attacks due to the inclusion of both public and private dependencies. The example is adapted from the CocoaPods `Podfile` guide, which we identified as vulnerable [84].

Dependency Confusion Attacks

CocoaPods supports the use of both internal and external dependency repositories. Combined with the possibility of executing arbitrary code during installation, it is vulnerable to dependency confusion attacks once developers mix dependency repositories without explicitly specifying dependency locations.

In Listing 5.1, we show a vulnerable `Podfile`, CocoaPods' dependency file, based on the official CocoaPods documentation [84]. Notably, the documentation is itself susceptible to dependency confusion attacks.

Line one specifies the public CocoaPods repository, and line two adds an internal dependency repository. The order of these entries is critical, as it determines the precedence for resolving dependencies. Lines seven and eight declare two dependencies from the public repository, followed by private dependencies retrieved from the internal repository.

If an attacker registers private dependencies in a public repository and includes malicious code that executes during installation, any developer updating dependencies risks compromising the device. To succeed, the attacker must publish matching version numbers corresponding to those specified in the `Podfile`.

`Podfiles` can also use wildcards for versions. In Listing 5.1, no version is specified for the library `GoogleAnalytics`, so it downloads the highest version available. In contrast, the wildcard for `OCMock` '~> 2.27' leaves the third (patch) version undefined, resulting in the highest matching patch version being downloaded. Alternatively, versions can be restricted using >, >=, <, and <=, or by explicitly specifying the version, as done for `Aerodramus`, and `Artsy+UIFonts`.

After the initial installation, CocoaPods generates a `Podfile.lock` file that records the versions and sources of the downloaded dependencies. Subsequent installation commands rely on this file rather than resolving dependencies again. To update this information and potentially confuse CocoaPods into downloading a library from a

public repository instead of the intended one, the `update` command must be used instead of the `install` command.

Dependency confusion attacks enable attackers to gain RCE on developer machines and build servers via the `prepare_command`, which allows the execution of arbitrary shell code. To directly compromise the mobile app's code through dependency confusion, an attacker must publish a malicious version of a functional dependency. This malicious package must implement all expected functions to ensure that the build process completes successfully and the app exhibits identical behavior. Otherwise, bugs and missing functionality could lead to the detection of the attack.

Since dependency confusion typically occurs when internal dependency repositories are used, it is unlikely that the source code of these libraries is publicly available. Consequently, attackers must first obtain the code, for instance, through RCE, or reverse engineer it from the app's binary. However, accurately reverse engineering a complex dependency is inherently difficult and error-prone.

Finding Internal Dependencies. Compared to other ecosystems, attackers targeting iOS apps benefit from the MATE setting. A key information needed for dependency confusion attacks is the internal dependency name and version. Both can be extracted from published apps.

In previous attacks on npm or pip, Birsan [68] identified such names by analyzing leaked dependency management files in public code repositories or by detecting dependency management files embedded in JS. In contrast, iOS apps directly leak the dependency names and versions through their directory structure and configuration files.

If a `Podfile` includes the `use_frameworks` keyword, CocoaPods builds each dependency as a separate framework. These frameworks appear in the app bundle under the `Frameworks` directory, following the naming scheme `framework_-name.framework` [101]. Typically, the framework name matches the library name, although libraries may override it using the `module_name` or `header_dir` fields in their specification file (`.podspec`). As we show in Section 5.4, only a small fraction of libraries use these overrides.

The second critical piece of information for dependency confusion attacks is the version of an internal library, which iOS apps also reveal. Within each `.framework` directory, CocoaPods includes an `Info.plist` file that contains the fields `CFBundle Version` and `CFBundleIdentifier`. By default, unless explicitly set by the library developer, the `CFBundleVersion` shows the version of the dependency. Moreover, the `CFBundleIdentifier` can reveal the use of CocoaPods, as it adds the prefix `or g.cocoapods` to the library name.

This metadata can help attackers identify vulnerable apps and reconstruct the dependency names and versions necessary for dependency confusion attacks.

Mitigation. In Listing 5.2 we provide an updated `Podfile` that is no longer vulnerable to dependency confusion attacks. Developers can address the issue in two ways. First, by changing the order of the sources, which determines the priority CocoaPods uses when resolving dependencies. Second, by explicitly specifying the location of the internal dependency, as done in line 10.

```
source 'https://github.com/Artsy/Spec'
source 'https://cdn.cocoapods.org/'
use_frameworks!

target :MyApp do
project 'MyApp.xcodeproj'
    pod 'GoogleAnalytics'
    pod 'OCMock', '~>2.27'
    pod 'Aerodramus', '2.1.4'
    pod 'Artsy+UIFonts', :git => 'https://github.com/artsy
    ↪ /Artsy-OSSUIFonts.git' :tag => '2.1.4'
end
```

Listing 5.2: A modified version of Listing 5.1 that prevents dependency confusion attacks. By changing the order of sources, CocoaPods resolves dependencies from the internal repository first. An alternative mitigation is to specify the source of internal dependencies explicitly.

Since our example builds on the `Podfile` from the official CocoaPods documentation, we reported their example as vulnerable and recommended updating it accordingly.

Dependency Hijacking

CocoaPods implements its own user management, which requires authentication and authorization to create and update dependencies. No password is required to log in to CocoaPods. Instead, CocoaPods sends a confirmation link to the registered email address. As a result, accounts associated with email addresses from abandoned domains become vulnerable to hijacking attacks.

This threat is further amplified by the fact that publishing new versions can occur silently, even when multiple accounts own a dependency. Typically, all dependency owners receive a notification email when a new library version is published or an old version is deleted. However, neither removing nor adding an account as a library owner triggers any notification. Consequently, attackers could remove other owners from a library, introduce malicious changes, and later re-add the original owners unnoticed.

Moreover, CocoaPods does not directly host libraries. Instead, they forward requests to the URLs that hosts them. As a consequence, attackers can hijack libraries if they manage to hijack their URLs, e.g., through abandoned GitHub namespaces or abandoned domains.

Once a dependency is installed, the `Podfile.lock` stores the library version and the checksum of its specification. Since the checksum is calculated over the specification file [86], and not the entire library, attackers can hijack the URL and silently inject malicious code. CocoaPods caches libraries locally, thus malicious modifications are fetched from the URL only if a developer downloads the dependency for the first time or explicitly clears the cache.

To mitigate this attack scenario, CocoaPods supports specifying the commit hash for Git sources or, in the case of archives, the archive's hash [83]. Thus, in those cases, attackers can only hijack existing library versions by hijacking the owner account of the library, e.g., through an abandoned email domain. An attacker with such access can modify the specification file of an existing library version, for instance, by removing the checksum and pointing to a new source URL. Since this change alters the specification, the checksum in the `Podfile.lock` also changes. However, it is unclear whether developers would recognize this as an attack or ignore the change. Alternatively, attackers can publish new versions of the library. As these versions are only downloaded when developers update their dependencies, a checksum change is expected and therefore less likely to raise suspicion.

A central dependency repository offers the key advantage of providing developers with an overview of existing libraries. We use this in Section 5.4 to measure the number of libraries vulnerable to hijacking attacks.

5.3.2 Carthage and SwiftPM

Carthage [75] and SwiftPM [321] are two decentralized dependency management systems for Swift. Carthage has been available since 2014, whereas Apple introduced SwiftPM in 2017 [295]. Because neither system maintains a central repository, developers must explicitly specify the paths or Git URLs for each dependency.

The decentralized design prevents dependency confusion attacks. However, it shifts dependency ownership entirely to the hosting site. As a result, attackers can hijack dependencies hosted on abandoned domains or abandoned GitHub namespaces.

It is possible to use libraries published to CocoaPods with Carthage and SwiftPM if they are published in a Git repository. Therefore, by analyzing whether any dependencies published in the central CocoaPods repository can be hijacked via an abandoned URL, we also gain insights into Carthage and SwiftPM dependencies (see Section 5.4).

Carthage and SwiftPM maintain a file documenting the versions used after the first installation. For Carthage, this file is `Cartfile.resolved`, which, unlike CocoaPods, only records the used versions and does not include a checksum unless the commit hash is explicitly specified. Consequently, modifications to existing versions can happen unnoticed. In contrast, SwiftPM's `Package.resolved` contains the commit hash of every dependency.

Unlike in CocoaPods, owning the URL is sufficient to provide new updates as the dependency system is decentralized and thus has no separate ownership model. Furthermore, it is unclear how developers would behave once a dependency version becomes unavailable, for example, if they would attempt to update or investigate the issue in depth, potentially uncovering the attack.

Takeaways

To answer RQ1: *Which supply chain issues do iOS dependency management systems face?*, we showed:

- that information about internal dependency names and versions included in iOS apps can enable dependency confusion attacks when dependency management files are misconfigured;
- the CocoaPods authentication is insecure and enables silent hijacking of libraries, if owners are registered with email addresses from abandoned domains;
- abandoned GitHub namespaces can also lead to dependency hijacking attacks.

5.4 Measurement of Vulnerable iOS Apps

After discussing supply chain attacks on iOS apps, we measure the number of vulnerable apps, and answer *RQ2: How many iOS apps are vulnerable to supply chain attacks?*

5.4.1 Dataset

To analyze the usage of vulnerable libraries, we use a dataset of 9,212 iOS apps downloaded in 2024, introduced by Schmidt et al. [299]. We refer to this dataset as `2024`. Their dataset also contains the matching Android version for each iOS app. We leverage this matching information together with the Android installation count as a proxy for the popularity of the iOS version, since the iOS App Store does not provide comparable download metrics [299].

In addition, we manually collected 279 iOS apps that participate in responsible disclosure programs to demonstrate the feasibility of dependency confusion attacks. These apps belong to 105 companies listed on platforms such as HackerOne [158], Bugcrowd [74], and Intigriti [180], as well as well-known companies like Microsoft [221] and Google [147] with their own responsible disclosure programs. We refer to this dataset as `disclosure 2025`.

5.4.2 Dependency Confusion in CocoaPods

Methodology

To identify apps vulnerable to dependency confusion attacks, we implemented an analysis in Python that extracts all framework names from iOS apps and retrieves their version and identifier from the corresponding `Info.plist` files. Then, our analysis compares the extracted information against the publicly available CocoaPods, which we obtained from the CocoaPods `Spec repository` [87] hosted on GitHub.

The specification repository also reveals naming practices of CocoaPod libraries, such as whether developers override default framework directory names using `head`

Table 5.1: Overview of analyzed *candidates*. *Vulnerable* refers to candidates affected by the vulnerability, whereas *targets* denotes the number of vulnerable CocoaPods.

	Candidates	Vulnerable	Targets
Dependency Confusion	24,838	9,866 (39.72%)	
Owner Hijacking	8,243	107 (1.30%)	213
Source Hijacking	1,969	83 (4.22%)	127
GitHub Hijacking	83,632	5,056 (6.05%)	5,137

`er_dir` or `module_name`. Thus, it indicates if our approach to extract dependency names from the framework directory is effective.

To demonstrate the feasibility of dependency confusion attacks and to responsibly disclose them, we used apps that participate in responsible disclosure programs and permitted the demonstration under their test policies. We automatically published the libraries with email addresses of our university. Further, each published library contained a `README` file, explaining the research purpose of our analysis, stating that we will remove the library two weeks after deployment and will delete all collected data within 31 days, as well as including our contact information.

In the event of a successful dependency confusion attack, we collected the hostname, installation directory, dependency name, and external IP address via DNS lookup queries during the installation process. We gathered this data to confirm that the installation originated from a company environment. We relied on DNS lookups to reduce the likelihood of firewalls blocking the requests [258, 68]. This mechanism allowed us to automatically remove the library after receiving callbacks, which helps avoid disrupting build processes. Subsequent updates then downloaded the legitimate internal version of the dependency, ensuring normal functionality. For more details on our ethical considerations, see Section 5.9.

Results

We first present an overview of whether libraries override their default `Framework` names, quantify the usage of public and private CocoaPods libraries, and discuss insights obtained from our Proof of Concept (PoC).

Names of Frameworks. To examine framework naming practices, we analyzed if libraries specified a `header_dir` or `module_name` in at least one version.

Both fields overwrite the default directory name, which otherwise matches the pod name. We found that 95,741 libraries (96.45%) did not specify either, 3,155 libraries (3.18%) used only `module_name`, and 234 libraries (0.24%) specified only `header_dir`. In addition, in 131 libraries (0.13%), both were present.

Even fewer libraries actually changed the default name. In total, only 1,689 libraries (1.70%) specified a name different from their library name. All remaining libraries used the library name as their `module_name` or `header_dir`. These findings underline that, in most cases, the `Framework` names within iOS apps directly reveal the corresponding CocoaPod library names.

Table 5.2: Number of apps using vulnerable dependencies. We use the Android installation count as a proxy for app popularity. The table does not include a row for *abandoned source hijacking* because no app in our dataset depends on a vulnerable version affected by this issue.

		Installations		
	# Apps	**Median**	**MAD**	**Max.**
Dependency Confusion	2,084 (22.62%)	500,000	499,900	1,000,000,000
Owner Hijacking	97 (1.05%)	500,000	499,990	100,000,000
GitHub Hijacking	80 (0.87%)	500,000	499,945	100,000,000

CocoaPod Usage. In 7,981 apps (86.64%), we identified at least one framework directory, while 5,097 apps (55.33%) included frameworks with identifiers prefixed by `org.cocoapods`. This identifier suggests that these frameworks were built using CocoaPods. Therefore, we further analyzed their framework names to detect libraries originating from private dependency repositories.

The comparison of discovered frameworks with CocoaPods identifiers to the public specification repository reveals that 9,866 frameworks from 2,084 apps (22.62%) are not registered in the public repository. Thus, they are potentially vulnerable to dependency confusion attacks.

Proof of Concept (PoC). From the apps with explicit disclosure programs (`disclosure2025` dataset), 116 (41.58%) contained frameworks with CocoaPod identifiers. Those apps belonged to 60 companies (54.04%). 764 frameworks were not registered in the central repository, originating from 67 apps (24.01%) of 34 companies (30.63%).

We selected libraries from 33 companies for our PoC. We excluded one as the discovered library name is a sub-library name of an existing dependency in CocoaPods and, therefore, a FP.

Overall, our PoC succeeded on libraries from nine companies. Of our submitted reports, three companies classified them as critical, three as high, and one as medium. The remaining two companies did not provide a classification. Notably, in all cases, companies requested evidence confirming that the installation originated from within their environment, underscoring the importance of collecting installation data during our tests.

We expect additional apps from our test set also to be vulnerable. However, dependency confusion only triggers when executing a CocoaPods `update` command within an affected project, which likely did not occur during our two weeks testing period. This may explain why our attack succeeded primarily against apps from larger companies, as these companies presumably update their dependencies more frequently.

Discussion. After we launched our PoC at the end of February 2025, we observed someone else deploying CocoaPods targeting dependency confusion by the end of March 2025 [102]. We contacted the user via the email address associated with the published library to clarify their intentions, and identity, as they had used a newly created pseudonym. Additionally, we informed them that we conduct measurements and perform a coordinated disclosure.

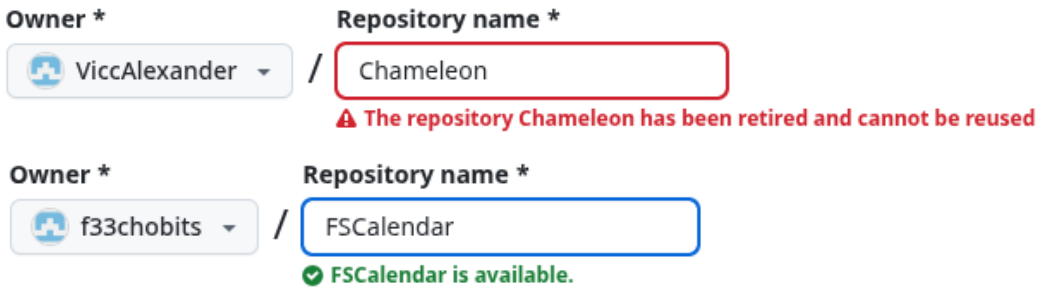


Figure 5.1: GitHub displays whether the repository is available or has been retired before creating it. Since we do not create the repository, the URL forwarding remains functional.

In response, the user stated that their motivation was to earn bug bounties and referenced a video on dependency confusion attacks [244] which explained the attack on npm and pip dependencies demonstrated by Birsan [68]. We are unaware of any other malicious or non-malicious exploitation in the CocoaPods ecosystem.

Positively, this contributed to the CocoaPods developers' decision to reject new libraries containing `prepare_commands` [331] in May 2025, thereby mitigating the threat of RCE during dependency installation via dependency confusion attacks. However, attackers could still infect devices if they have access to the dependency code or can successfully reverse engineer it to provide a functional, but malicious dependency.

5.4.3 Dependency Hijacking

We evaluated the feasibility of hijacking attacks using hand-crafted examples instead of real libraries. We are not aware of any responsible way to conduct such tests on existing libraries. Hijacking real libraries could affect numerous apps and permanently disable the current forwarding mechanism, potentially breaking build processes. We provide more details on ethical considerations in Section 5.9.

Methodology

To assess the risk of dependency hijacking attacks due to abandoned domains and GitHub namespaces, we extracted (1) the list of available libraries and (2) their associated dependency hosting locations, and their integrity verification checks, from the CocoaPods specification repository.

We obtained the email addresses of all library owners to analyze the potential for hijacking CocoaPod libraries through abandoned email addresses. Since this information is not available in the specification repository, we queried the API, accessible via `https://trunk.cocoapods.org/api/v1/pods/{name}`, to obtain the library owners and their email addresses.

We subsequently verified the availability of email domains using the bulk domain search services provided by GoDaddy [144] and Namecheap [234], employing both services to cross-validate results. If a domain appears available on one service only, we manually assessed its availability.

Table 5.3: Number of GitHub repositories using vulnerable dependencies. The column *vulnerable* indicates where the vulnerability resides, that is, in the *owner* domain, the *source* domain, or the *GitHub* URL. For CocoaPods, we searched for the dependency names within dependency files on GitHub, whereas for Carthage and SwiftPM, we searched for vulnerable URLs.

			Stars		
System	**Vulnerable**	**# Repos.**	**Avg.**	**Std.**	**Max.**
CocoaPods	Owner	121	169.06	867.39	7,129
CocoaPods	Source	4	1.25	2.17	5
CocoaPods	GitHub	671	25.69	222.26	3,805
Carthage	GitHub	11	2.82	2.62	7
SwiftPM	GitHub	43	43.26	210.48	1,400
Total		834	42.25	370.71	7,129

To identify abandoned hosting locations, we differentiated between dependencies hosted on GitHub repositories and those hosted on custom domains. Similar to the approach for abandoned email domains, we again checked the availability of each hosting domain. For libraries hosted on GitHub, we initially queried the GitHub API to verify the existence of each namespace. If a repository redirected to a new URL, we also recorded the number of stars for the redirected repository.

In a second step, for abandoned GitHub repositories, we manually registered available GitHub namespaces and entered the matching image-names upon repository creation. Since we did not create the repository, this step did not break the dependency forwarding, see Figure 5.1. However, it allowed us to evaluate whether GitHub has retired the URL.

Further, we extracted the commit hashes and archive hashes listed in the specification repositories. Both could prevent malicious modifications of the library if attackers hijack the associated URLs.

Finally, we utilized our analysis, as described in Section 5.4.2, to identify vulnerable libraries used in iOS apps.

Results

We studied whether owners of libraries registered with abandoned email domains and provide insights into dependencies referencing abandoned URLs.

Abandoned Email Domains. Out of a total of 99,261 CocoaPods, we extracted 8,243 domains of owner email addresses, of which 107 were abandoned (1.30%) owning 213 libraries, see Table 5.1. In our dataset, we identified 97 apps (1.05%) that utilized at least one vulnerable dependency.

The most frequently observed vulnerable library was `DZNEmptyDataSet`, used by 63 apps (0.68%). It was followed by `SlackTextViewController`, found in 15 apps (0.16%), and `DTTJailbreakDetection`, found in eight apps (0.09%). Further, we detected seven other vulnerable libraries, each appearing in at most three apps.

Those usages differ from what we observed in GitHub repositories. We found 121 repositories that relied on vulnerable libraries, as detailed in Table 5.3. The library `SlackTextViewController` was the most prevalent, appearing in 15 repositories, followed by `UIColor-HexString` in 14 repositories, and `DTTJailbreakDetecti on` in 13 repositories. Notably, `DZNEmptyDataSet`, the most common vulnerable dependency in apps, ranked ninth on GitHub with six occurrences. Also, popular open-source repositories used vulnerable dependencies, as the average star count was 169, and one repository even had 7,129 stars.

To mitigate the threat of available domains, we proactively registered the two abandoned email domains associated with the three most frequently encountered dependencies in iOS apps. Since `DZNEmptyDataSet` and `DTTJailbreakDetection` share the same owner. We did not alter library specifications or ownership and registered the domains only to prevent malicious hijacks.

We responsibly disclosed our findings to both CocoaPods developers and affected apps. As of writing, we have not received any update from CocoaPods. However, they plan to transition the public repository to a read-only repository in 2026, which will mitigate the issue of hijacking attacks due to abandoned maintainer email addresses.

Abandoned Source Locations. Unlike the previously discussed issue of abandoned email domains, which only affects CocoaPods, dependencies hosted on abandoned domains or GitHub repositories also impact apps built using Carthage and SwiftPM. Both dependency management systems support direct dependency referencing by specifying an URL pointing to a Git repository.

Of the total 99,261 CocoaPods libraries, we found that 1,354 libraries (1.36%) specify an integrity check in at least one version, either through a commit hash or the hash of the library archive in the specification file. Even fewer, 781 libraries (0.79%), applied these checks consistently across all versions.

We extracted 1,969 unique domains from the CocoaPods specifications, of which 83 were abandoned (4.22%). 127 libraries referenced these abandoned domains. Notably, libraries occasionally changed hosting locations across their versions. For example, some were initially hosted on GitHub but migrated to hosting the library on a domain associated with the project.

25 vulnerable libraries referenced multiple hosting locations. Thirteen libraries used vulnerable domains for their earlier versions, whereas seven libraries transitioned from secure domains in earlier versions to abandoned domains in their recent ones.

Of the 25 vulnerable libraries, only one applied an integrity check. Nevertheless, it does not mitigate the issue, as none of the versions that can be hijacked specify a check.

We identified three libraries that included at least one vulnerable version. However, none of the apps used an actually vulnerable version. On GitHub, we found four repositories that referenced vulnerable CocoaPods versions, while no vulnerable domain references appeared in SwiftPM or Carthage configuration files.

Further, we analyzed the GitHub URLs referenced by CocoaPod libraries. From 83,632 unique repositories, we discovered that 5,056 (6.05%) were abandoned. We manually verified the possibility to register the namespace and image-name of every dependency that forwarded to an image with at least 100 stars (236), and all

remaining URLs used by libraries discovered in an app of our dataset (31). Of the repositories with at least 100 stars, we would have been able to register 186 (78.81%).

Alarmingly, among vulnerable GitHub URLs, we identified 42 repositories with over 1,000 stars. We would have assumed that these repositories had reached the 5,000-download threshold required for retirement. A plausible explanation for this is that they were renamed before reaching the required number of downloads.

With a total of 5,137 CocoaPods libraries, we discovered even more vulnerable libraries than abandoned GitHub URLs. This is because 93 vulnerable URLs were referenced by multiple CocoaPods libraries. We did not observe any malicious intent behind this behavior. Instead, library authors renamed the library or used different library names to reference specific branches and builds, for example, production and debug versions. Of the vulnerable libraries, 49 (0.95%) specified in at least one version a commit hash, while 33 (0.64%) did so across all versions.

In our dataset, 80 apps (0.87%) relied on vulnerable CocoaPod libraries, including three apps with over 100 million installations: two fitness apps, and one antivirus app. In total, we identified the use of 49 distinct vulnerable libraries. With seven occurrences, `JTMaterialSpinner` is the most popular one, followed by `JXPageControl` with six uses. Remarkably, we did not find a single app using an abandoned GitHub URL where the library is protected through specifying its commit hash.

We detected insecure CocoaPods dependencies in 671 GitHub repositories. In two additional cases, the library referenced an abandoned GitHub URL, but the specification of the library version included a commit hash, and thus, the dependency was not vulnerable. Additionally, we identified references to abandoned GitHub URLs in 11 Carthage repositories and 43 SwiftPM repositories, as summarized in Table 5.3. Upon analyzing these repositories, we discovered that three repositories (over 100 stars) are actively maintained, with commits made in the past year. One project used SwiftPM, and two used CocoaPods.

Disclosure. We reported the attack scenario and the vulnerable repositories to GitHub. In their response, they highlighted that the forwarding mechanism is an intentional design decision. Additionally, they mentioned that they mitigate it for popular repositories through the described retirement strategy (see Section 5.2.2). They also mentioned that they show warning messages to users when they delete or rename their namespace.

However, as our results show, this is not sufficient. To mitigate this attack vector, all previously used image-names should be retired for the namespace. Alternatively, removing the forwarding to the new namespace could help. This would cause dependencies to break after renaming, which library owners might notice, for example, through user complaints, and could ultimately lead them to update existing references.

We sent 154 disclosure emails to app developers that use vulnerable dependencies. Following the approach by Schmidt et al. [299], we used the contact information from the Google Play Store to obtain the corresponding email addresses. We merged all reports with a common email address to avoid sending multiple emails to the same developer.

Takeaways

To answer RQ2: *How many iOS apps are vulnerable to supply chain attacks?*, we:

- showed that in our dataset 2,084 apps (22.62%) are potentially vulnerable to dependency confusion attacks. We successfully demonstrated the attack on apps from nine companies with disclosure programs;
- discovered 213 CocoaPod libraries that are vulnerable to hijacking attacks because their owners registered with email addresses belonging to abandoned domains;
- identified 5,056 vulnerable abandoned GitHub URLs whose dependencies were used by 80 apps (0.87%). This highlights that GitHub's retirement policy is insufficient.

5.5 Defense Mechanisms in Other Systems

After analyzing the number of iOS apps vulnerable to dependency confusion and hijacking attacks, we answer *RQ3: How can dependency management systems protect against the identified vulnerabilities?*. To do this, we study cargo, Go modules, Maven, npm, and pip, as these ecosystems are widely used and implement four slightly different approaches to dependency management.

5.5.1 Cargo

The Cargo package manager for Rust [330] has a central repository, namely `crates.io`.

Authentication. To publish packages, `crates.io` uses GitHub for authentication. Even if the GitHub namespace hosting the package is renamed or deleted, hijacking the published package remains impossible because the authentication on `crates.io` is still bound to the original GitHub account. Unlike CocoaPods, `crates.io` stores the dependencies directly on their servers instead of forwarding requests to hosting locations [271]. Consequently, to hijack a library owner account, an attacker needs to hijack the corresponding GitHub account.

Since March 2023, GitHub requires 2FA for all users contributing code [136]. Consequently, attackers cannot hijack GitHub accounts, and therefore their corresponding `crates.io` accounts, solely through abandoned domains. When strong 2FA is enforced, dependency hijacking becomes significantly more challenging, as domain takeover alone is no longer sufficient. In this case, an attacker must also compromise the second-factor device or exploit a vulnerability in the 2FA implementation.

A potential attack scenario arises when projects reference dependencies directly via Git URLs. If the referenced URL is hijackable, an attacker can gain control of the repository and inject malicious code. This threat can be partially mitigated by specifying commit hashes or checksums. However, the risk persists during updates, where changes are expected and can be used to introduce malicious modifications.

```
module example.com/mymodule
go 1.14
require (
    example.com/thismodule v1.2.3
    github.com/myuser/mydep v1.0.0
)
```

Listing 5.3: Example Go dependency file. One dependency is linked to the domain `example.com`, and the other points to the GitHub image `myuser/mydependency`.

Private Dependencies and Code Execution. Cargo requires developers to explicitly specify the dependency registry when using any source other than `crates.io`. This requirement ensures that Cargo does not misinterpret dependencies when a package name exists in both the private and public registries. Furthermore, in contrast to CocoaPods, Cargo does not support arbitrary code execution during library installation.

5.5.2 Go Modules

Listing 5.3 shows a sample `go.mod` dependency file.

Authentication. At first glance, the system appears decentralized because developers specify dependency URLs. However, this impression is misleading. By default, the Go proxy [143, 183] caches retrieved libraries, and once cached, the proxy serves the library directly instead of fetching it from the original URL. Consequently, the system behaves as a centralized one and therefore, it is essential to analyze the requirements for publishing new library versions.

The Go proxy does not enforce authentication directly. Instead, it relies on the authentication mechanisms of the platform hosting the library, most commonly GitHub. Gu et al. [154] demonstrated that attackers can hijack dependencies when the dependency hosting URL on GitHub becomes available. In their study, they identified 11,788 dependency URLs available on GitHub. If attackers register an abandoned namespace and image-name, they can publish new, malicious versions. Once the proxy caches these versions, existing projects may unknowingly update to the compromised version and thus become vulnerable. Since developers can reference other third-party URLs, hijacking libraries also becomes possible once the hosting domain expires.

As discussed in Section 5.2.2, GitHub retires namespace and image-name combinations after they exceed 5,000 downloads to mitigate supply chain attacks since 2023. While this mechanism reduces the likelihood of namespace hijacking, it does not fully eliminate the threat, as demonstrated by our analysis of iOS dependencies. Thus, the question arises whether the caching behavior of Go dependencies further undermines this mitigation. Once a dependency is cached, the proxy serves all subsequent downloads rather than GitHub, potentially preventing the dependency from reaching the required download threshold for URL retirement. Additionally,

the caching mechanism masks the fact that the original account was deleted or the domain abandoned because the dependency continues to function through the proxy.

In Section 5.5.6, we measure the number of dependencies vulnerable to hijacking attacks to understand better how Gos caching mechanism influences the abandonment of GitHub URLs, and to quantify how many dependencies are hosted on abandoned domains. Previous work [154] did not examine either aspect.

Private Dependencies and Code Execution. Developers can configure private proxies for private repositories [143, 183]. However, an attacker would still need to hijack the internally used URL to register a private dependency. In addition, Go modules do not support arbitrary code execution during installation [143]. Consequently, Go is not vulnerable to dependency confusion attacks.

5.5.3 npm

For npm, the popular package manager of Node.js, several security analyses exist [368, 68, 239, 227].

Authentication. When a dependency is retrieved from the central npm registry, protection against hijacking attacks relies on npm's authentication mechanisms. In addition to standard username and password authentication, npm recommends enabling 2FA. However, the second factor is delivered via email by default, which does not provide protection if the associated domain is abandoned. If maintainers do not configure a robust 2FA setup, their packages become vulnerable once the associated domain is abandoned [200, 81]. It is also possible to reference a dependency directly via an URL. In that case, the security responsibility shifts to the platform hosting the dependency, as is the case with Go and Cargo.

Private Dependencies and Code Execution. Developers can mix public and private dependencies. Further, combined with the possibility of executing arbitrary code during dependency installation, it enables the potential for dependency confusion attacks [68, 154]. Npm allows disabling the script execution during dependency installation through a parameter [318]. However, per-default script execution is enabled.

Similar to the leakage of private CocoaPods dependencies, we observed that the directory structure of iOS apps can expose the npm packages in use. Unlike typical browser-based applications, where the `node_modules` directory resides on the server and remains hidden from the client, this directory may be included in distributed mobile apps. Consequently, analyzing the app package can reveal dependency names potentially leading to dependency confusion attacks. In Section 5.5.6, we analyze the private npm packages included in our iOS dataset to gain further insights.

5.5.4 pip

The Python pip package manager uses similar architectural concepts to those of npm.

Authentication. Hijacking attacks through abandoned email addresses associated with library owners have been documented [255]. To mitigate this risk, pip began enforcing 2FA in 2024 for all accounts that publish or modify packages [122]. Moreover, starting in 2025, they implemented daily scans for abandoned domains

associated with account owners. Once a domain is classified as untrustworthy due to expiration, the system no longer sends password reset emails [123]. This measure reduces the likelihood that an attacker can log in and hijack a library after registering an expired domain.

Similar to other dependency management systems, pip supports installing packages directly from Git repositories or by specifying a URL. In such cases, attackers can again hijack the referenced URLs to inject malicious code.

Private Dependencies and Code Execution. It is also possible to install dependencies from a private repository by specifying it through a command-line parameter. However, this configuration remains vulnerable to dependency confusion attacks, as demonstrated by prior work [68].

5.5.5 Maven

The standard dependency management systems for Java and Kotlin are Maven and Gradle, which typically resolve dependencies through Maven Central [217, 153, 252]. Other public repositories also exist, such as JitPack [187].

Authentication. Dependencies follow the format `groupId:artifactId:version`. To publish a package under a specific `groupId`, for example `com.google`, the publisher must prove the ownership of the corresponding domain by adding a DNS record. When publishing packages via source code management platforms like GitHub, the dependency repositories also require proof of ownership [218].

On Maven Central, this verification process is performed automatically or handled by support staff. Consequently, it remains unclear whether previously registered dependencies could be hijacked by acquiring control of an abandoned domain or GitHub namespace, or whether the human verification step prevents such attempts [218, 256]. We refrained from testing the hijacking due to ethical concerns.

Because multiple public dependency repositories coexist, some projects integrate more than one. Attackers can exploit dependencies that do not exist in all repositories with a `groupId` vulnerable to hijacking. They can publish a malicious version of such a dependency to a repository where it was previously absent. Depending on the resolver configuration, the dependency management system may select and download the malicious artifact instead [154, 256].

Private Dependencies and Code Execution. Similar to Go dependencies, even when private and public repositories are combined, attackers must still gain control of the corresponding hosting URL to publish a malicious dependency in a public dependency repository. Additionally, Maven does not offer the ability for code execution during installation.

5.5.6 Go and npm Measurements

We measure the number of vulnerable open-source projects on GitHub to understand how Gos caching mechanism influences the abandonment of GitHub URLs and whether dependencies are hosted on abandoned domains. Prior work [154] did not analyze these aspects. Understanding the effectiveness of GitHubs mitigation strategies in the Go ecosystem and assessing whether abandoned URLs threaten

Table 5.4: Vulnerable Go dependencies. *Dependency* shows the dependency name. *# Proj.* indicates the total number of GitHub projects using the dependency, while *Avg.* and *Max.* refer to the average and maximum number of stars of dependent projects.

Dependency	# Proj.	Avg.	Max.
adigunhammedolalekan/registry-auth	9	839.44	2,582
deniswernert/go-fstab	11	826.36	5,175
deniswernert/udev	3	706.33	2,119
go-spectest/imaging	1	2259.00	2,259
gosidekick/migration	8	548.38	4,373
longbridgeapp/assert	17	111.35	1,729
smartystreets-prototypes/go-disruptor	8	340.62	1,404
tf-controller/terraform-exec	2	725.00	1,450
tmz.dev	209	49.63	3,936
Vernacular-ai/godub	5	340.60	1,698

the ecosystem are essential to evaluate the robustness of Gos design and to develop effective mitigation strategies.

In addition, we study npm dependencies included in mobile apps, a previously unexplored attack surface for dependency confusion and targeted hijacking attacks.

Effectiveness of GitHub Mitigation for Go Dependencies

We present our methodology for finding vulnerable Go dependencies, followed by the results of our measurements, and recommended mitigation strategies.

Methodology. To identify abandoned Go dependencies, we automatically collected all `go.mod` files from public repositories with over 1,000 stars using the GitHub API.

Using regular expressions, we automatically extracted each dependency's URL and applied our analysis from Section 5.4 to identify abandoned domains and GitHub namespaces. To eliminate FPs caused by GitHub's namespace retirement mechanisms, we again manually attempted to register each identified namespace and checked whether the image-name is retired, as illustrated in Figure 5.1. In the second step, we searched all public repositories for usages of vulnerable dependencies via the GitHub API.

Results. In total, we downloaded 5,848 `go.mod` files from 2,811 repositories belonging to 2,076 GitHub namespaces in March 2025. From these dependency files, we extracted dependencies hosted on 4,520 GitHub namespaces and 219 domains. Among them, 30 namespaces (0.66%) and one domain (0.46%) were available for registration.

Our manual verification revealed that seven of the available GitHub namespaces (23.33%), encompassing nine image-names, were vulnerable to hijacking attacks. GitHubs mitigation strategy of retiring URLs effectively prevented hijacking attacks for the remaining ones.

In Table 5.4, we provide an overview of the vulnerable dependencies and the number of projects that would become vulnerable if the dependency were hijacked. The results show that popular projects are also affected. For instance, `cubefs`, a cloud storage software project with 5,175 stars (as of July 2025), is vulnerable. Projects from well-known companies were also impacted, such as `aws/amazon-ecs-agent` in the case of `deniswernert/udev`, or `kubernetes/ingress-gce` and `kubernetes/test-infra` in the case of the abandoned domain `tmz.dev`.

If attackers hijack any of these libraries, they could compromise the affected projects and potentially all devices running them. Our analysis only covers public GitHub projects, yet private projects are also likely to use these vulnerable dependencies. We suspect that the Go module caching mechanism partly undermines GitHubs URL retirement strategy, leading to popular projects being vulnerable.

Mitigation and Disclosure. As an immediate mitigation, we registered the namespaces of all vulnerable image-names and the abandoned domain. This registration does not interfere with any forwarding. Its only purpose is to prevent attackers from performing hijacking attacks.

The mitigation only addresses the attack at a single point in time and only for dependencies used by the most popular Go projects. For a more sustainable solution, we propose two long-term mitigation strategies: (1) Introduce an ownership mechanism within the Go proxy, similar to how Cargo enforces authentication via GitHub. This mechanism would ensure that only legitimate owners can publish new dependency versions, thereby preventing unauthorized hijacks. (2) Strengthen GitHub's namespace retirement policy by permanently retiring all previously used namespace and image-name combinations, rather than relying on popularity metrics. However, this will not mitigate the issue of abandoned domains and relies on the third-party hosting provider.

We responsibly disclosed the issue and the affected dependencies to Google in April 2025. Google acknowledged the report and opened two bug tickets: one concerning abandoned domains and another regarding GitHub namespaces. At the time of writing, Google closed the issue of abandoned domains as *"[...] It is unfortunate, but working as intended. [...]"*, while they left the issue about abandoned GitHub namespaces open.

npm Dependencies in Mobile Apps

In the following, we present our methodology for discovering vulnerable npm dependencies, followed by our measurement results.

Methodology. Similar to our analysis of CocoaPods, we investigated whether package names, leaked through the app directory structure, are available in the npm repository.

We extracted all file paths from each app bundle and filtered for paths containing either `node_modules/` or `www/plugins/`. Then, we extracted the directory names immediately following these path segments, as they correspond to npm package names. If a directory name started with `@`, indicating an organizational namespace, we also extracted the name of the subdirectory within that namespace.

We excluded all identified names containing a dot or uppercase character, as npm does not allow such characters, making these names FPs. We then queried the

availability of each extracted name using a `GET` request to `https://registry.npmjs.org/{npm_name}`. This approach can yield FPs for organizational namespaces, since some may be registered solely to prevent dependency confusion or are used for private packages. To eliminate FPs, we manually attempted to register each available namespace. This process does not interfere with existing name resolution if the corresponding package name remains unregistered. Positively, it proactively prevents attackers from claiming the namespace.

We extracted the associated owner email address for existing packages to determine whether it pointed to an abandoned domain. We then again used the methodology described in Section 5.4 to identify abandoned ones.

Results. Overall, we extracted 671 npm package names from 808 iOS apps (8.77%). Among these, 79 package names were not registered in the public npm registry and vulnerable to dependency confusion attacks if the dependency management file is misconfigured. These vulnerable packages appeared in 27 distinct iOS apps. The median number of installations per app was 500,000 ($mad = 499,500$), with one app exceeding 50 million installations. Positively, we did not identify npm packages linked to accounts with abandoned email addresses.

Responsible Disclosure. We sent responsible disclosure emails using the contact addresses listed on the Google Play Store. Each message described the attack scenario, explained potential consequences, and included mitigation strategies.

5.5.7 Discussion

As shown by Cargo and recent changes in npm and pip, requiring 2FA for authentication is a recommended strategy to mitigate hijacking attacks stemming from abandoned URLs. Active scanning, as performed in our work and also by pip enables the proactive detection of abandoned domains, thereby reducing the likelihood of attacks.

Our measurements of CocoaPods and Go dependencies vulnerable to hijacking on GitHub reveal that GitHub's account retirement policy alone is insufficient. This is particularly evident for Go dependencies, where the caching behavior of the Go proxy may counteract GitHub's mitigation strategies. From a security perspective, it is preferable for dependency management systems to host dependencies directly rather than forwarding requests to external hosting URLs. Direct hosting allows the system to enforce authentication policies independently of third-party platforms. As illustrated by the Go ecosystem, this approach is only effective if the hosting platform implements a reliable authentication and authorization mechanism.

To prevent dependency confusion attacks, dependency management systems should require developers to explicitly specify private dependency sources, as enforced by Cargo. Additionally, they should disable code execution by default during package installation to reduce the attack surface.

Although we recommend requiring developers to directly specify URLs for internal dependencies, applying this practice to public URLs reintroduces reliance on external platforms' security. Therefore, such configurations should only be used if unavoidable. In these cases, developers should specify immutable references, such as commit hashes, and carefully review code changes in updated versions. Otherwise, attackers

may hijack the referenced URL and inject malicious code into newly published versions.

Takeaways

To answer RQ3: *How can dependency management systems protect against the identified vulnerabilities?*, we conclude that dependency management systems should:

- require developers to explicitly specify private dependencies and disable code execution by default to prevent dependency confusion attacks;
- enforce 2FA and host dependencies themselves to mitigate hijacking attacks;
- mandate commit hashes or checksums for dependencies referenced by their URLs.

5.6 Limitations and Future Work

Our analysis of dependency confusion and the use of libraries vulnerable to hijacking provides a lower bound of vulnerable apps. Since we analyzed the contents of the `Frameworks` directory to identify private dependencies and their versions, our measurement misses libraries not included in the `Frameworks` directory when the flag `use_framework!` is not set. In such cases, the library is linked directly into the app's binary. Consequently, vulnerable libraries are likely even more widely used, indicating that our findings are more severe than our measurements show.

Similar to the `Frameworks` directory, `.bundle` directories that typically contain resources [40] could reveal internal dependency names and versions. However, we found this information to be FP prone, as it often did not match the dependency names.

A longitudinal study could reveal whether the overall situation improves or deteriorates over time. However, such data is best collected actively over an extended period rather than retrospectively, as certain information, e.g., the ownership of CocoaPod libraries, is only available at the time of collection. Therefore, we consider this as part of future work.

While we focused on analyzing security aspects of dependency management systems, particularly CocoaPods, we did not examine the use of outdated public libraries or attempt to detect malicious packages within these ecosystems. Because we concentrated on technical flaws in dependency management systems, we consider both aspects out of scope, but recognize them as interesting directions for future work.

Another research direction could investigate the developers' perspective using qualitative research methods to gain insights into the defense mechanisms developers employ and their reactions to attacks.

In November 2024, CocoaPods announced that the central dependency repository will become read-only by the end of 2026 [331]. Additionally, after we demonstrated the feasibility of dependency confusion attacks, CocoaPods announced that they will reject new libraries containing a `prepare_command`, thereby mitigating the risk of

RCE during dependency installation. Once the repository enters read-only mode, the threat of dependency hijacking attacks through abandoned owner domains will disappear. Even if an attacker successfully hijacks the account of a dependency owner, they will no longer be able to publish new malicious versions of the library to CocoaPods.

5.7 Related Work

Detection of Malicious Packages. Related work explored the detection of malicious packages published to dependency repositories [250, 103, 174, 366, 172, 361, 160, 296, 251]. Guo et al. [155] examined the behavior of malicious pip packages, while Vu et al. [341] interviewed developers to understand their needs regarding approaches to detect malicious Python packages. Taylor et al. [328] developed defenses against typosquatting attacks prior to installation. Neupane et al. [239] explored syntactic and semantic variants related to typosquatting designed to trick developers into installing malicious packages. Andreoli et al. [11] analyzed known supply chain attacks, while Martínez and Durán [214] examined the SolarWinds breach.

In contrast, we focus on identifying new attack vectors within the iOS app ecosystem, rather than detecting attacks.

Dependency Confusion and Hijacking Attacks. Zimmermann et al. [368] studied the npm ecosystem and highlighted threats arising from dependency hijacking attacks through transitive dependencies. Birsan [68] demonstrated that leaked internal dependency names can enable RCE when package managers prioritize public repositories. Ladisa et al. [202] analyzed remote code execution vulnerabilities in seven dependency management systems by differentiating between install-time and run-time code execution. Zahan et al. [360] discovered that 2,818 maintainer domains in npm are available and that 2.2% of packages execute install scripts. Wyss et al. [356] proposed Latch, a system that prevents code execution during npm dependency installation. Gu et al. [154] identified twelve threat categories for packages hosted across six repositories and mirrors, including dependency hijacking attacks in Go. We extend this line of research to the iOS ecosystem and demonstrate that widely used apps were vulnerable to dependency confusion and hijacking attacks.

Moreover, we extend the analysis by Gu et al. on Go dependencies to evaluate the effectiveness of GitHubs mitigation strategies in the context of Gos dependency caching, and also assess the extent to which Go dependencies are exposed when hosted on abandoned domains.

Ladisa et al. [201] summarized the risks associated with supply chain attacks and proposed mitigations, including enforcing 2FA, reviewing merge requests, and disabling install-time code execution. Other studies examined the security practices of open-source contributors [192] and investigated software signing procedures [189] to reduce attack surfaces in supply chains. Sammak et al. [294] collected insights from 18 developers on the practical challenges of securing the software supply chain.

In contrast, our study directly addresses the mitigation of concrete attack vectors we identified and demonstrated in iOS dependency management systems.

Alternative Supply Chain Targets. Researchers have also explored alternative supply chain attack surfaces such as backdoored machine learning models [343] and compromised hardware components [325].

These directions, however, lie outside our scope, as we focus on software supply chain attacks enabled through dependency management systems and the dependency information leaked by mobile apps.

5.8 Conclusion

In this paper, we demonstrated that the leakage of library names and versions in iOS app bundles can enable dependency confusion attacks, effectively granting attackers RCE on build servers and developer devices. We further showed that attackers can exploit abandoned domains and GitHub namespaces to hijack iOS libraries.

Our measurement study of 9,212 iOS apps revealed that dependencies of 162 apps (1.76%) could be affected by hijacking attacks, including popular apps with at least 100 million installations. Moreover, we practically demonstrated dependency confusion attacks on apps from nine well-known companies.

Additionally, we analyzed the security of five additional dependency management systems: Cargo, Go modules, Maven, npm, and pip, to highlight countermeasures like 2FA and disabling code execution during installation. However, we also demonstrated that these systems face similar conceptual issues.

5.9 Ethics considerations

5.9.1 Dependency Confusion

We practically demonstrated dependency confusion attacks on dependencies from apps with responsible disclosure programs that explicitly permitted tests through their policies. To verify that the installation originated from the targeted company, we collected the following information: (1) external IP address, (2) hostname, and (3) installation directory. This approach is consistent with previously reported work on dependency confusion attacks [68]. Further, we were asked by disclosure programs for evidence of potential compromise of the company's infrastructure, which we could provide with the collected information. We removed all data once the report was processed or at the latest 31 days after its collection.

We included our contact details, linking to our university email address, and a brief explanation in the `.podspec` file and a `README` for each deployed library. Before conducting our PoC, we assessed the risk of unintentionally targeting unrelated apps and found it minimal for two reasons. First, package names usually follow organization or product-specific naming conventions, which reduces the likelihood of overlap with apps from unrelated companies. Second, the library description explicitly stated that we were performing an experiment, and the library itself exposed only two trivial methods, one returning `true` and one returning `false`, both derived from CocoaPod example code and unrelated to any realistic library functionality.

We strongly believe that it is important to demonstrate such attacks practically to highlight their exploitability and potential impact. Further, it raises awareness of the vulnerability and supports fixing insecure implementations before malicious actors can exploit them. As a result, our findings contributed to the CocoaPods maintainers' decision to reject new libraries containing a `prepare_command` as of May 2025 [331], effectively mitigating the threat of RCE via dependency confusion.

5.9.2 Dependency Hijacking

To demonstrate the feasibility of dependency hijacking, we first validated our approach using handcrafted examples. We did not hijack real libraries or modify other owner accounts of a vulnerable library, as doing so could affect numerous apps and cause lasting damage.

We reported all identified issues to GitHub and the maintainers of vulnerable dependency management systems (CocoaPods and Google in the case of Go). In addition, we disclosed our findings to potentially affected apps to raise their awareness.

To prevent attackers from registering abandoned domains and namespaces, we proactively claimed the most critical ones ourselves without interfering with their functionality. We did not register all of them due to the manual effort required for namespace creation and the cost of domain registration.

6 Conclusion

The overall goal of this thesis was to identify weak links in the mobile app ecosystem at scale to improve its security and privacy through responsible disclosure. To this end, we analyzed four aspects of the ecosystem: (1) IoT devices through companion apps, (2) the iOS local network permission, (3) secrets distributed in mobile apps, and (4) iOS dependency management systems. Therefore, we designed scalable analysis techniques, applied them to thousands of apps, and used the resulting data for further security and privacy relevant evaluations and measurements.

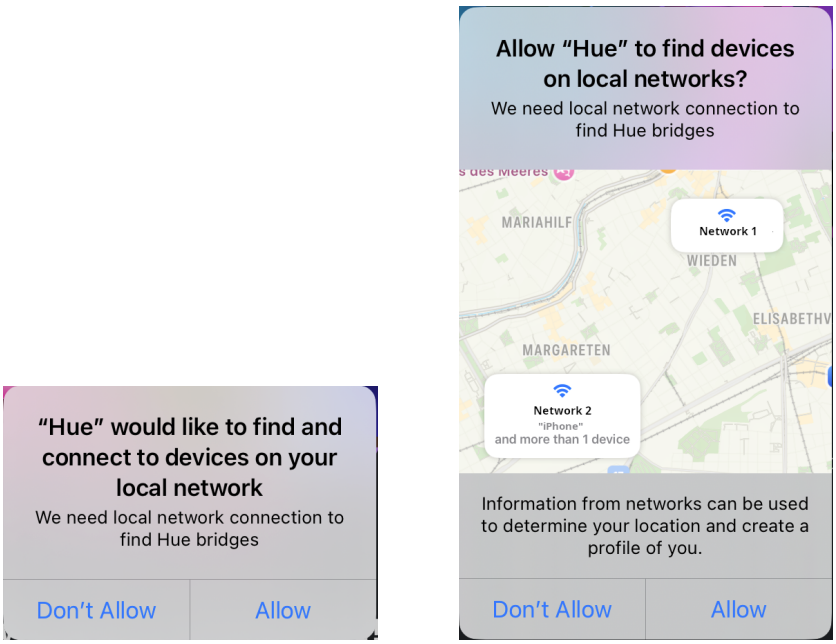
First, we developed a static analysis based on VSA and DFA to study the network communication of IoT companion apps. By analyzing 9,889 companion apps, we uncovered multiple security and privacy issues, including MQTT brokers that allowed unauthenticated access, abandoned domains that could enable attackers to take over IoT devices, the use of broken encryption algorithms, and the exposure of PII.

Second, we analyzed the iOS local network permission from both a technical and a user perspective. We uncovered two iOS components that can bypass the permission and showed that the protected address space was insufficient for complex networks and VPNs. Moreover, we identified local network permission rationales that contained misleading concepts or misconceptions, which could trick users into granting the permission. On the positive side, our user study showed that nearly every participant was aware of at least one threat arising from local network access. However, misconceptions about the permission and the granted access were even more prevalent.

Third, we analyzed unintentionally shared secrets in mobile apps, ranging from internal documentation and source code to embedded credentials. Our results highlighted the importance of studying both Android and iOS apps, as we observed inconsistent credential leaks across Android and iOS versions of the same app. A plausible explanation for this behavior is that different development teams work on apps for different platforms. Overall, we discovered more issues in iOS apps, potentially because Android apps are easier accessible and historically received more research attention.

Finally, we analyzed dependency management systems used by iOS apps and uncovered issues that enable supply chain attacks. We identified dependency hijacking and dependency confusion attacks as concrete threats, and we demonstrated dependency confusion attacks against nine apps from companies with responsible disclosure programs. Such attacks can enable adversaries to gain RCE on developer devices and build servers when dependency management files are misconfigured. In addition, we highlighted the attack potential of dependency hijacking through abandoned email address domains and GitHub namespaces. Moreover, we measured the number of vulnerable libraries and affected apps on a dataset of 9,212 iOS apps.

Looking forward, several opportunities for future work emerge. In this thesis, we focused on improving the security and privacy of the mobile app ecosystem



(a) <= iOS 17.7 dialog. (b) >= iOS 18 dialog.

Figure 6.1: iOS local network permission dialog. The left side shows the dialog design at the time of our study, see Chapter 3. The right side shows the updated dialog, which displays a map of local networks the phone previously connected to.

by identifying and reporting vulnerabilities at scale. Although our analyses can operate on apps before release to detect issues early, we did not study this use case from a developer perspective. Future work should therefore investigate how to integrate app analysis techniques into development workflows. Mobile apps and their ecosystems also continuously evolve, for example, Android recently introduced the local network permission [23]. As our study of iOS local network permissions demonstrates, systematically analyzing newly introduced components is important to identify security issues associated with them.

Another interesting direction concerns the local network permission dialog. Apple updated this dialog and now highlights that once a user grants the permission, the app receives access not only on the currently connected network but also on other local networks, see Figure 6.1. This behavior reflects a concept we called *transitivity*, which participants in our user study found difficult to understand. Future work should therefore examine whether the updated dialog improves users' understanding of this concept.

List of Tables

List of Figures

Bibliography

[1] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster. Web-based Attacks to Discover and Control Local IoT Devices. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*. (2018). doi:10.1145/3229565.3229568.

[2] Aisberg Inc LLC. App Store – Scan & Translate+ Text Grabber. Archived at: https://archive.ph/qHLIP. https://apps.apple.com/us/app/scan-translate-text-grabber/id845139175.

[3] Akamai Security Intelligence Group. XZ Utils Backdoor Everything You Need to Know, and What You Can Do. Archived at https://archive.ph/NVOVO. (2024). https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know.

[4] A. A. Al Alsadi, K. Sameshima, J. Bleier, K. Yoshioka, M. Lindorfer, M. Van Eeten, and C. H. Gañán. No Spring Chicken: Quantifying the Lifespan of Exploits in IoT Malware Using Static and Dynamic Analysis. In *Proceedings of the 17th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2022). doi:10.1145/3488932.3517408.

[5] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. (2022).

[6] M. Alfhaily. Github – majd/ipatool – release v2.2.0. Archived: https://archive.ph/OUJbF. https://github.com/majd/ipatool/releases/tag/v2.2.0.

[7] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. (2016). doi:10.1145/2901739.2903508.

[8] AloneMonkey. GitHub – frida-ios-dump. Commit: 56e99b2. https://github.com/AloneMonkey/frida-ios-dump.

[9] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Z. Snow, F. Monrose, and M. Antonakakis. The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. (2021).

[10] Amazon. Amazon Alexa. Archived at https://archive.ph/2aUEx. https://alexa.amazon.com/.

[11] A. Andreoli, A. Lounis, M. Debbabi, and A. Hanna. On the Prevalence of Software Supply Chain Attacks: Empirical Study and Investigative Framework. *Forensic Science International: Digital Investigation*, 44, (2023). doi:10.1016/j.fsidi.2023.301508.

[12] Android. Android Debug Bridge. Archived at https://archive.ph/jJFBb, Version: 1.0.41. https://developer.android.com/tools/adb.

[13] Android. Manifest.permission. Archived at https://archive.ph/LGldg. https://developer.android.com/reference/android/Manifest.permission#CHANGE_WIFI_MULTICAST_STATE.

[14] Android. Permissions on Android. Archived at https://archive.ph/b21T7. https://developer.android.com/guide/topics/permissions/overview#normal_permissions.

[15] Android. Request runtime permissions. Archived at https://archive.ph/ujApe. https://developer.android.com/training/permissions/requesting.

[16] Android. WifiManager.MulticastLock. Archived at https://archive.ph/SYYh7. https://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock.

[17] Android Developers. Security with network protocols. (2021). Retrieved Oct. 7, 2022 from https://developer.android.com/training/articles/security-ssl.

[18] Android Developers. Android Keystore system. (2022). Retrieved July 25, 2023 from https://developer.android.com/training/articles/keystore.

[19] Android Developers. Best practices for unique identifiers. (2023). Retrieved July 25, 2023 from https://developer.android.com/training/articles/user-data-ids.

[20] Android Developers. Bluetooth permissions. (2023). Retrieved July 28, 2023 from https://developer.android.com/guide/topics/connectivity/bluetooth/permissions.

[21] Android Developers. Version your app. (2023). Retrieved Apr. 27, 2023 from https://developer.android.com/studio/publish/versioning.

[22] Android Developers. Work with data more securely. (2023). Retrieved July 25, 2023 from https://developer.android.com/topic/security/data.

[23] Android Developers. Local network permission. (2025). https://developer.android.com/privacy-and-security/local-network-permission.

[24] Android Developers. Build multiple APKs. Archived at https://archive.ph/s17jN. https://developer.android.com/build/configure-apk-splits.

[25] Android Developers. TelephonyManager. Retrieved July 13, 2022 from https://developer.android.com/reference/android/telephony/TelephonyManager.

[26] Android Developers. UI/Application Exerciser Monkey. Archived at https://archive.ph/EuqsW. https://developer.android.com/studio/test/other-testing-tools/monkey.

[27] S. Antonatos, P. Akritidis, V. T. Lam, and K. G. Anagnostakis. Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. *ACM Transactions on Information and System Security*, 12, 2, (2006). doi:10.1145/1455518.1477941.

[28] APKMirror. Free APK Downloads. Archived at https://archive.ph/h4ABV. https://www.apkmirror.com/.

[29] APKPure. Download APK on Android. Archived at https://archive.ph/oCCd5. https://apkpure.com/.

[30] AppBrain. Android network libraries. Retrieved Sept. 19, 2021 from https://www.appbrain.com/stats/libraries/tag/network/android-network-libraries.

[31] Appium. Appium Documentation. Version: 2.4.1. http://appium.io.

[32] Apple. If an app would like to connect to devices on your local network. Archived at https://archive.ph/ld7Z8. (2020). https://support.apple.com/en-us/HT211870.

[33] Apple. Should I use WKWebView or SFSafariViewController for web views in my app? Archived at https://archive.ph/YLX1c. (2020). https://developer.apple.com/news/?id=trjs0tcd.

[34] Apple. Use AirPlay with Apple devices. Archived at https://archive.is/5R8ov. (2021). https://support.apple.com/guide/deployment/use-airplay-dep9151c4ace.

[35] Apple. Entitlements. Archived at https://archive.is/mep7b. https://developer.apple.com/documentation/bundleresources/entitlements.

[36] Apple. Local Network Privacy FAQ. Archived at https://archive.is/rce8M. (2024). https://forums.developer.apple.com/forums/thread/663858.

[37] Apple. NSLocalNetworkUsageDescription. Archived at https://archive.is/zlwj0. https://developer.apple.com/documentation/bundleresources/information_property_list/nslocalnetworkusagedescription.

[38] Apple. About AirPlay. Archived at https://archive.ph/msqDt. https://developer.apple.com/library/archive/documentation/AudioVideo/Conceptual/AirPlayGuide/Introduction/Introduction.html.

[39] Apple. App Store – Safari. Version: 8617.2.4.10.7. https://apps.apple.com/us/app/safari/id1146562112.

[40] Apple. Apple Developer – Placing content in a bundle. Archived at https://archive.ph/I7oLd. https://developer.apple.com/documentation/bundleresources/placing-content-in-a-bundle.

[41] Apple. Developer Documentation. Archived at https://archive.ph/DKBNW. https://developer.apple.com/documentation.

[42] Apple. Local Network Privacy FAQ-13. Archived at https://archive.ph/CyGqd. https://developer.apple.com/forums/thread/663839.

[43] Apple. Local Network Privacy FAQ-2. Archived at https://archive.ph/Le8lh. https://developer.apple.com/forums/thread/663874.

[44] Apple. Local Network Privacy FAQ-8. Archived at https://archive.ph/2xaE5. https://developer.apple.com/forums/thread/663888.

[45] Apple. Property List Key - NSBonjourServices. Archived at https://archive.ph/FCSrW. https://developer.apple.com/documentation/bundleresources/information_property_list/nsbonjourservices.

[46] Apple. Raw socket on iOS. Archived at https://archive.ph/kYAj7. https://developer.apple.com/forums/thread/653072.

[47] Apple. Requesting access to protected resources. Archived at https://archive.ph/Cin61. https://developer.apple.com/documentation/uikit/protecting_the_user_s_privacy/requesting_access_to_protected_resources/.

[48] Apple. Requesting authorization to use location services. Archived at https://archive.ph/L5MB9. https://developer.apple.com/documentation/corelocation/requesting_authorization_to_use_location_services.

[49] Apple. SFSafariViewController. Archived at https://archive.ph/WVkDI. https://developer.apple.com/documentation/safariservices/sfsafariviewcontroller.

[50] Apple. SimplePing Example code. Version 5.0. https://developer.apple.com/library/archive/samplecode/SimplePing.

[51] Apple. Triggering the Local Network Privacy Alert. Archived at https://archive.ph/oPzYd. https://developer.apple.com/forums/thread/663768.

[52] Apple. UIWebView. Archived at https://archive.ph/tVkyn. https://developer.apple.com/documentation/uikit/uiwebview.

[53] Apple. WKWebView. Archived at https://archive.ph/KM4rb. https://developer.apple.com/documentation/webkit/wkwebview.

[54] Apple Inc. Apple Reinvents the Phone with iPhone. Archived at https://archive.ph/CA0Sf. (2007). https://www.apple.com/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/.

[55] AppLovin. Connect to audiences in-app, on mobile devices, and across CTV. Archived at https://archive.ph/JLpKX. https://www.applovin.com/.

[56] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. (2014). doi:10.1145/2594291.2594299.

[57] S. Ashenbrenner. Full Transparency: Controlling Apple's TCC. Archived at https://archive.is/v1s4s. (2024). https://www.huntress.com/blog/full-transparency-controlling-apples-tcc.

[58] Audible, Inc. App Store – Audible: Audio Entertainment. Archived at https://archive.ph/G5CPz. https://apps.apple.com/us/app/audible-audio-entertainment/id379693831.

[59] L. Babun, Z. Berkay Celik, P. McDaniel, and S. Uluagac. Real-time Analysis of Privacy-(un)aware IoT Applications. In *Proceedings of the 21st Privacy Enhancing Technologies Symposium (PETS)*. (2021). doi:10.2478/popets-2021-0009.

[60] M. Backes, S. Bugiel, and E. Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2016). doi:10.1145/2976749.2978333.

[61] M. Balossini and S. Arzt. GitHub FlowDroid Issue #578 – Flowdroid execution time. (2023). Retrieved Apr. 27, 2023 from https://github.com/secure-software-engineering/FlowDroid/issues/578.

[62] S. K. Basak, J. Cox, B. Reaves, and L. Williams. A Comparative Study of Software Secrets Reporting by Secret Detection Tools. In *Proceedings of the IEEE/ACM International Symposium on Empirical Software Engineering and Measurement (ESEM)*. (2023). doi:10.1109/ESEM56168.2023.10304853.

[63] S. K. Basak, L. Neil, B. Reaves, and L. Williams. What Are the Practices for Secret Management in Software Artifacts? In *Proceedings of the IEEE Cybersecurity Development (SecDev)*. (2022). doi:10.1109/SecDev53368.2022.00026.

[64] S. Baskaran, L. Zhao, M. Mannan, and A. Youssef. Measuring the Leakage and Exploitability of Authentication Secrets in Super-apps: The WeChat Case. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. (2023). doi:10.1145/3607199.3607236.

[65] P. Beer, M. Squarcina, L. Veronese, and M. Lindorfer. Tabbed Out: Subverting the Android Custom Tab Security Model. In *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. (2024). doi:10.1109/SP54263.2024.00105.

[66] P. Beer, L. Veronese, M. Squarcina, and M. Lindorfer. The Bridge between Web Applications and Mobile Platforms is Still Broken. In *3rd IEEE Workshop on Designing Security for the Web (SecWeb)*. (2022).

[67] E. Belinski. Android API Levels. (2023). Retrieved Sept. 8, 2023 from https://apilevels.com/.

[68] A. Birsan. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. Archived at https://archive.ph/455ky. (2021). https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610.

[69] D. Boffey. EU Recalls Children's Smartwatch over Data Fears. (2019). Retrieved Jan. 30, 2022 from https://www.theguardian.com/technology/2019/feb/05/eu-recalls-childrens-smartwatch-over-data-fears.

[70] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft. Exploring decision making with Androids runtime permission dialogs using in-context surveys. In *Proceedings of the 13th Symposium On Usable Privacy and Security (SOUPS)*. (2017).

[71] K. Borgolte, T. Fiebig, S. Hao, C. Kruegel, and G. Vigna. Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (2018). doi:10.14722/ndss.2018.23327.

[72] K. Borgolte, S. Hao, T. Fiebig, and G. Vigna. Enumerating Active IPv6 Hosts for Large-scale Security Scans via DNSSEC-signed Reverse Zones. In *Proceedings of the 39th IEEE Symposium on Security & Privacy (S&P)*. (2018). doi:10.1109/SP.2018.00027.

[73] Brave Software. App Store – Brave Browser: Private VPN. Version: 1.62. https://apps.apple.com/us/app/brave-browser-private-vpn/id1052879175.

[74] Bugcrowd. bugcrowd.com/.

[75] Carthage. GitHub – A simple, decentralized dependency manager for Cocoa. Archived at https://archive.ph/KVMkT. https://github.com/Carthage/Carthage.

[76] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive Information Tracking in Commodity IoT. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. (2018). doi:10.5555/3277203.3277329.

[77] J. Chen et al. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (2018). doi:10.14722/ndss.2018.23159.

[78] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. (2015). doi:10.1109/ASE.2015.89.

[79] G. Chu, N. Apthorpe, and N. Feamster. Security and Privacy Analyses of Internet of Things Children's Toys. *IEEE Internet of Things Journal*, 6, 1, (2019). doi:10.1109/JIOT.2018.2866423.

[80] K. Chung. Taking over a Dead IoT Company. (2023). Retrieved Feb. 5, 2023 from https://blog.kchung.co/taking-over-a-dead-iot-company/.

[81] T. Claburn. The Register – Expert grabs expired domain for NPM package to make a point. Archived at https://archive.ph/zYi1W. (2022). https://www.theregister.com/2022/05/10/security_npm_email/.

[82] G. Cluley. Snapchat's source code leaked out, and was published on GitHub. Archived at https://archive.ph/OgcR6. (2018). https://www.bitdefender.com/en-gb/blog/hotforsecurity/snapchats-source-code-leaked-out-and-was-published-on-github.

[83] CocoaPods. CocoaPods Guides – Podspec Syntax Reference. Archived at: https://archive.ph/yo1tB. https://guides.cocoapods.org/syntax/podspec.html#group_subspecs.

[84] CocoaPods. CocoaPods Guides – The Podfile. Archived at https://archive.ph/bmXAs. https://guides.cocoapods.org/using/the-podfile.html.

[85] CocoaPods. CocoaPods.org. Archived at https://archive.ph/yQ24y. https://cocoapods.org/.

[86] CocoaPods. GitHub – CocoaPods/Core. Archived at: https://archive.ph/oDorz. https://github.com/CocoaPods/Core/blob/a53e235aa4d1eec8a21042e022ba7bcaca14ae56/lib/cocoapods-core/podfile.rb#L259.

[87] CocoaPods. GitHub – CocoaPods/Specs: The CocoaPods Master Repo. Archived at https://archive.ph/ojsNf. https://github.com/CocoaPods/Specs/.

[88] Codename One. How to Build iOS Apps with Java. Archived at https://archive.ph/VdGWX. https://www.codenameone.com/blog/how-to-build-ios-apps-with-java.html.

[89] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*. (2017). doi:10.14722/ndss.2017.23465.

[90] A. Continella, M. Polino, M. Pogliani, and S. Zanero. There's a Hole in that Bucket! A Large-scale Analysis of Misconfigured S3 Buckets. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. (2018). doi:10.1145/3274694.3274736.

[91] Corona Labs Inc. Corona – Free Cross-Platform 2D Game Engine. Archived at: https://archive.ph/FmKN1. https://coronalabs.com/.

[92] A. Cortesi, D. Weinstein, D. Freed, T. Kriechaumer, P. F. Tirenna, M. Hils, and U. Verma. mitmproxy - an interactive HTTPS proxy. Version: 10.1.6. https://mitmproxy.org/.

[93] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A Large-scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. (2014). doi:10.5555/2671225.2671232.

[94] CruZer000. Reddit – Why does AliExpress need access to my local network? Archived at https://archive.ph/Nqw48. (2021). https://www.reddit.com/r/Aliexpress/comments/m1u2en/why_does_aliexpress_need_access_to_my_local/.

[95] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle. Secrets Revealed in Container Images: An Internet-wide Study on Occurrence and Impact. In *Proceedings of the 18th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2023). doi:10.1145/3579856.3590329.

[96] B. De Sutter, S. Schrittwieser, B. Coppens, and P. Kochberger. Evaluation Methodologies in Software Protection Research. *ACM Computing Surveys*, 57, 4, (2024). doi:10.1145/3702314.

[97] Decathlon. App Store – Decathlon Connect. Archived at: https://archive.ph/13znY. https://apps.apple.com/us/app/decathlon-connect/id1288552594.

[98] A. Desnos, G. Gueguen, S. Bachmann, and contributors. GitHub – Androguard - Android Permissions. (2020). Retrieved May 2, 2023 from https://github.com/androguard/androguard/tree/master/androguard/core/api_specific_resources/aosp_permissions.

[99] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig. Investigating Operators' Perspective on Security Misconfigurations. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2018). doi:10.1145/3243734.3243794.

[100] J. Dinneen. GitHub – File Extension Categoriser. Archived at https://archive.ph/DntMe. https://github.com/jddinneen/file-extension-categoriser.

[101] D. Domínguez-Álvarez, A. de la Cruz, A. Gorla, and J. Caballero. LibKit: Detecting Third-Party Libraries in iOS Apps. In *Proceedings of the 10th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. (2015). doi:10.1145/3611643.3616344.

[102] dontdoit8001. GitHub – [Add] bybit_web3_whitebox_encrypt 1.0.0. Archived at https://archive.ph/yOVz2. https://github.com/CocoaPods/Specs/commit/c9a77987c5739f1c915d4f8b7b41f1c0781a9fc7.

[103] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Proceedings of the 28th Network and Distributed System Security Symposium (NDSS)*. (2021). doi:10.14722/ndss.2021.23055.

[104] Dyne.org. GitHub – Organised collection of common file extensions. Archived at https://archive.ph/LS08v. https://github.com/dyne/file-extension-list/.

[105] Eclipse Foundation. Paho - Python Client. v1.5.1. (2022). Retrieved Oct. 8, 2022 from https://www.eclipse.org/paho/index.php?page=clients/python/index.php.

[106] S. El Yadmani, O. Gadyatskaya, and Y. Zhauniarovich. The File That Contained the Keys Has Been Removed: An Empirical Analysis of Secret Leaks in Cloud Buckets and Responsible Disclosure Outcomes. In *Proceedings of the 46th IEEE Symposium on Security & Privacy (S&P)*. (2025). doi:10.1109/SP61157.2025.00009.

[107] Y. Elbitar, M. Schilling, T. T. Nguyen, M. Backes, and S. Bugiel. Explanation Beats Context: The Effect of Timing & Rationales on Users Runtime Permission Decisions. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. (2021).

[108] P. Emami-Naeini, J. Dheenadhayalan, Y. Agarwal, and L. F. Cranor. Which Privacy and Security Attributes Most Impact Consumers' Risk Perception and Willingness to Purchase IoT Devices? In *Proceedings of the 42nd IEEE Symposium on Security & Privacy (S&P)*. (2021). doi:10.1109/SP40001.2021.00112.

[109] P. Emami-Naeini, J. Dheenadhayalan, Y. Agarwal, and L. F. Cranor. Are Consumers Willing to Pay for Security and Privacy of IoT Devices? In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. (2023).

[110] I. Erben. Country CIDR IP Ranges. Retrieved Jan. 25, 2022 from http://www.iwik.org/ipcountry/.

[111] European Parliament and the Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, 59, (2016), L119, (2016). Retrieved July 5, 2021 from http://data.europa.eu/eli/reg/2016/679/oj.

[112] European Parliament and the Council of the European Union. Regulation (EU) 2022/1925 of the European Parliament and of the Council of 14 September 2022 on contestable and fair markets in the digital sector and amending Directives (EU) 2019/1937 and (EU) 2020/1828 (Digital Markets Act). *Official Journal of the European Union*, 65, (2022), L265, (2022). http://data.europa.eu/eli/reg/2022/1925/oj.

[113] Exodus. Trackers. Retrieved Feb. 23, 2022 from https://reports.exodus-privacy.eu.org/en/trackers/.

[114] Expedia. Play Store – Expedia: Hotels, Flights, Cars. Archived at https://archive.ph/aYx88. https://play.google.com/store/apps/details?id=com.expedia.bookings.

[115] C. Farinella, A. Ahmed, and C. Watterson. Git Leaks: Boosting Detection Effectiveness Through Endpoint Visibility. In *Proceedings of the 20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. (2021). doi:10.1109/TrustCom53373.2021.00103.

[116] D. Farrah and M. Dacier. Zero Conf Protocols and their numerous Man in the Middle (MITM) Attacks. In *Proceedings of the 15th IEEE Workshop on Offensive Technologies (WOOT)*. (2021). doi:10.1109/SPW53761.2021.00060.

[117] M. Fassl, S. Anell, S. Houy, M. Lindorfer, and K. Krombholz. Comparing User Perceptions of Anti-Stalkerware Apps with the Technical Reality. In *Proceedings of the 18th Symposium On Usable Privacy and Security (SOUPS)*. (2022).

[118] J. Feichtner. A Comparative Study of Misapplied Crypto in Android and iOS Applications. In *Proceedings of the 16th International Conference on Security and Cryptography (SECRYPT)*. (2019). doi:10.5220/0007915300960108.

[119] R. Feng, Z. Yan, S. Peng, and Y. Zhang. Automated Detection of Password Leakage from Public GitHub Repositories. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. (2022). doi:10.1145/3510003.3510150.

[120] C. Fenton and contributors. GitHub – APKiD. (2020). Retrieved Jan. 29, 2022 from https://github.com/rednaga/APKiD.

[121] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *Proceedings of the 1st IEEE European Symposium on Security & Privacy (EuroS&P)*. (2016). doi:10.1109/SP.2016.44.

[122] M. Fiedler. The Python Package Index Blog – 2FA Requirement for PyPI begins 2024-01-01. Archived at https://archive.ph/Mp08m. (2023). https://blog.pypi.org/posts/2023-12-13-2fa-enforcement/.

[123] M. Fiedler. The Python Package Index Blog – Preventing Domain Resurrection Attacks. Archived at https://archive.ph/JHEX2. (2025). https://blog.pypi.org/posts/2025-08-18-preventing-domain-resurrections/.

[124] A. Forsberg and L. H. Iwaya. Security Analysis of Top-Ranked mHealth Fitness Apps: An Empirical Study. In *Proceedings of the 29th Secure IT Systems - Nordic Conference (NordSec)*. (2024). doi:10.1007/978-3-031-79007-2_19.

[125] Fortinet. Supply Chain Attacks: Examples And Countermeasures. Archived at https://archive.ph/9cJiS. https://www.fortinet.com/resources/cyberglossary/supply-chain-attacks/.

[126] T. Franke, C. Attig, and D. Wessel. A Personal Resource for Technology Interaction: Development and Validation of the Affinity for Technology Interaction (ATI) Scale. *International Journal of Human-Computer Interaction*, 35, (2019), 6, (2019). doi:10.1080/10447318.2018.1456150.

[127] Free Software Foundation, Inc. Coreutils - GNU Core Utilities. Archived at https://archive.ph/9cQ2P. https://www.gnu.org/software/coreutils/.

[128] P. Gadient, M. Ghafari, M.-A. Tarnutzer, and O. Nierstrasz. Web APIs in Android through the Lens of Security. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. (2020). doi:10.1109/SANER48275.2020.9054850.

[129] V. E. Garske, S. Lange, G. K. Gegenhuber, D. Schmidt, A. Noack, and J. Classen. Blue Bubbles, Red Flags: Investigating Privacy Leakage in Apple iMessage. In *Proceedings of the 33rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2026).

[130] G. K. Gegenhuber, P. É. Frenzel, M. Günther, J. Ullrich, and A. Judmayer. Hey there! You are using WhatsApp: Enumerating Three Billion Accounts for Security and Privacy. In *Proceedings of the 33rd Network and Distributed System Security Symposium (NDSS)*. (2026).

[131] G. K. Gegenhuber, M. Grefner, M. Günther, M. Wininger, D. Schmidt, and A. Judmayer. Send and Pretend: Exploiting Transcript Consistency Issues in End-to-End Encrypted Group Chats. In *Proceedings of the 35th USENIX Security Symposium (USENIX Security)*. (2026).

[132] D. Geneiatakis, I. Kounelis, R. Neisse, I. Nai-Fovino, G. Steri, and G. Baldini. Security and privacy issues for an iot based smart home. In *Proceedings of the 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. (2017). doi:10.23919/MIPRO.2017.7973622.

[133] Generate and Parse Apple .plist Files. Version: 3.8. https://docs.python.org/3/library/plistlib.html.

[134] K. George. The Guardian - Prices that change by the second: why shopping around for deals online isnt always worth it. Archived at https://archive.ph/7mw9i. (2022). https://www.theguardian.com/lifeandstyle/2022/dec/12/prices-that-change-by-the-second-why-shopping-around-for-deals-online-isnt-always-worth-it/.

[135] A. Girish et al. In the Room Where It Happens: Characterizing Local Communication and Threats in Smart Homes. In *Proceedings of the 23rd Internet Measurement Conference (IMC)*. (2023). doi:10.1145/3618257.3624830.

[136] GitHub. Configuring two-factor authentication. Archived at https://archive.ph/vAAUQ. https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/configuring-two-factor-authentication.

[137] GitHub – jadx - Dex to Java decompiler. Version: 1.5.1. https://github.com/skylot/jadx.

[138] GitHub – Magisk: The Magic Mask for Android. Version: v26.4. https://github.com/topjohnwu/Magisk.

[139] GitHub, Inc. GitHub Blog – New tools for open source maintainers. Archived at https://archive.ph/AwQcr. (2018). https://github.blog/open-source/maintainers/new-tools-for-open-source-maintainers/.

[140] GitHub, Inc. GitHub – [Already shipped] GHCR namespace retirement [GA] (#34085). Archived at https://archive.ph/RnFEg. (2023). https://github.com/github/docs/commit/fa650c0a9426f90d68243bde31220c9922b07f50.

[141] Gitleaks. GitHub – Gitleaks. Version: 8.21.2. https://github.com/gitleaks/gitleaks.

[142] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini. Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy. In *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2020). doi:10.1145/3320269.3384745.

[143] Go Modules Reference – The Go Programming Language. Archived at https://archive.ph/0gYJ2. https://go.dev/ref/mod.

[144] GoDaddy – Bulk Domain Names – Search and Register Multiple Domains. https://www.godaddy.com/en/domains/bulk-domain-search.

[145] Google. Chromecast with Google TV. Archived at https://archive.ph/Y2VCs. https://store.google.com/us/product/chromecast_google_tv.

[146] Google. GitHub – Google HTTP Client Library for Java. Archived at: https://archive.ph/TQK53. https://github.com/googleapis/google-http-java-client/blob/main/google-http-client/src/main/java/com/google/api/client/testing/json/webtoken/TestCertificates.java.

[147] Google. Google Bug Hunters. https://bughunters.google.com.

[148] Google. ML Kit. Archived at https://archive.ph/oktoP. https://developers.google.com/ml-kit.

[149] Google. ML Kit – Barcode scanning. Archived at https://archive.ph/NmHdk. https://developers.google.com/ml-kit/vision/barcode-scanning.

[150] Google. ML Kit – Face detection. Archived at https://archive.ph/6J3yw. https://developers.google.com/ml-kit/vision/face-detection.

[151] Google. User Data. Retrieved July 25, 2023 from https://support.google.com/googleplay/android-developer/answer/10144311.

[152] Google LLC. App Store – Google Chrome. Version: 122.0.6261.62. https://apps.apple.com/us/app/google-chrome/id535886823.

[153] Gradle – Using Resolution Rules. Archived at https://archive.ph/uV3F3. https://docs.gradle.org/current/userguide/resolution_rules.html.

[154] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan. Investigating Package Related Security Threats in Software Registries. In *Proceedings of the 44th IEEE Symposium on Security & Privacy (S&P)*. (2023). doi:10.1109/SP46215.2023.10179332.

[155] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu. An Empirical Study of Malicious Code In PyPI Ecosystem. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. (2023). doi:10.1109/ASE56229.2023.00135.

[156] Y. Guo and T. Dong. Exposing the Danger Within: Hardcoded Cloud Credentials in Popular Mobile Apps. Archived at https://archive.ph/bC9m3. (2024). https://www.security.com/threat-intelligence/exposing-danger-within-hardcoded-cloud-credentials-popular-mobile-apps.

[157] I. Gupta. Mobile App Industry Statistics 2023: Trends and Insights You Shouldnt Ignore. Archived at: https://archive.ph/0OjDB. https://ripenapps.com/blog/mobile-app-industry-statistics/.

[158] HackerOne. https://hackerone.com.

[159] HackTricks. macOS TCC. Archived at https://archive.is/JITyX. https://book.hacktricks.xyz/macos-hardening/macos-security-and-privilege-escalation/macos-security-protections/macos-tcc/.

[160] S. Halder et al. Malicious Package Detection using Metadata Information. In *Proceedings of the World Wide Web Conference (WWW)*. (2025). doi:10.1145/3589334.3645543.

[161] C. Han, I. Reyes, Á. Feal, J. Reardon, P. Wijesekera, N. Vallina-Rodriguez, A. Elazari, K. A. Bamberger, and S. Egelman. The Price is (Not) Right: Comparing Privacy in Free and Paid Apps. In *Proceedings of the 20th Privacy Enhancing Technologies Symposium (PETS)*. (2020). doi:10.2478/popets-2020-0050.

[162] S. Han. Google Translate API for Python. Version: 4.0.0rc1. https://github.com/ssut/py-googletrans.

[163] haxrob. Twitter - HaxRob - Lightbulb App. (2023). Retrieved July 25, 2023 from https://twitter.com/haxrob/status/1676424071452708864.

[164] M. Hazhirpasand and M. Ghafari. One Leak is Enough to Expose Them All: From a WebRTC IP Leak to Web-based Network Scanning. In *Proceedings of the 10th International Symposium on Engineering Secure Software and Systems (ESSoS)*. (2018). doi:10.1007/978-3-319-94496-8_5.

[165] Y. He et al. TextExerciser: Feedback-driven Text Input Exercising for Android Applications. In *Proceedings of the 41st IEEE Symposium on Security & Privacy (S&P)*. (2020). doi:10.1109/SP40000.2020.00071.

[166] J. Hendrix. Tech Policy – Transcript: TikTok CEO Testifies to Congress. Archived at https://archive.ph/NjL0L. (2023). https://www.techpolicy.press/transcript-tiktok-ceo-testifies-to-congress/.

[167] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner. Smart Locks: Lessons for Securing Commodity Internet of Things Devices. In *Proceedings of the 11th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2016). doi:10.1145/2897845.2897886.

[168] A. Holst. IoT Connected Devices Worldwide 2019-2030. (2021). Retrieved Jan. 15, 2022 from https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[169] F. Holzbauer, D. Schmidt, G. Gegenhuber, S. Schrittwieser, and J. Ullrich. Malicious or Not: Adding Repository Context to Agent Skill Classification. *arXiv preprint arXiv:2603.16572*.

[170] J. Howarth. iPhone vs Android User Stats (2024 Data). Archived at https://archive.ph/L1QOZ. (2024). Retrieved Jan. 13, 2025 from https://explodingtopics.com/blog/iphone-android-users.

[171] L. Hsu and R. Field. Interrater Agreement Measures: Comments on Kappan, Cohen's Kappa, Scott's π, and Aickin's α. *Understanding Statistics*, 2, (2003). doi:10.1207/S15328031US0203_03.

[172] C. Huang et al. DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*. (2024).

[173] D. Huang, N. Apthorpe, G. Acar, F. Li, and N. Feamster. IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale. In *Proceedings of the ACM International Joint Conference on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT/Ubicomp)*. (2020). doi:10.1145/3397333.

[174] Y. Huang, R. Wang, W. Zheng, Z. Zhou, S. Wu, S. Ke, B. Chen, S. Gao, and X. Peng. SpiderScan: Practical Detection of Malicious NPM Packages Based on Graph-Based Behavior Modeling and Matching. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. (2024).

[175] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu. Your Code Secret Belongs to Me: Neural Code Completion Tools Can Memorize Hard-Coded Credentials. *ACM on Software Engineering*, 1, FSE, (2024). doi:10.1145/3660818.

[176] Hugging Face. The AI community building the future. Archived at https://archive.ph/T6678. https://huggingface.co/.

[177] IFTTT. If This Then That – Connect Your Apps. Retrieved July 24, 2023 from https://ifttt.com/.

[178] IKEA. IKEA Home smart app and TRÅDFRI gateway support. Archived at https://archive.ph/N0RmO. https://www.ikea.com/us/en/customer-service/product-support/app-gateway/.

[179] Internet Assigned Numbers Authority (IANA). Uniform Resource Identifier (URI) Schemes. (2022). Retrieved Jan. 30, 2022 from https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml.

[180] Intigriti. Intigriti – Bug Bounty & Agile Pentesting Platform. https://www.intigriti.com.

[181] iRobot Corporation. App Store – iRobot Home. Archived at https://archive.ph/HKgmC. https://apps.apple.com/at/app/irobot-home/id1012014442.

[182] iZarcon. Reddit – Access the Internet. Archived at https://archive.ph/HPZXU. (2021). https://www.reddit.com/r/MicrosoftTeams/comments/n4tkhs/comment/gwxgatv/.

[183] L. Jain. GOPROXY - A Central Module Dependency Management. Archived at https://archive.ph/99FEs. (2021). https://lkumarjain.blogspot.com/2021/08/goproxy-central-module-dependency.html.

[184] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao, and Y. Zhang. Burglars' IoT Paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds. In *Proceedings of the 41st IEEE Symposium on Security & Privacy (S&P)*. (2020). doi:10.1109/SP40000.2020.00051.

[185] X. Jin, S. Manandhar, K. Kafle, Z. Lin, and A. Nadkarni. Understanding IoT Security from a Market-Scale Perspective. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2022). doi:10.1145/3548606.3560640.

[186] Z. Jin, L. Xing, Y. Fang, Y. Jia, B. Yuan, and Q. Liu. P-verifier: understanding and mitigating security risks in cloud-based iot access policies. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2022). doi:10.1145/3548606.3560680.

[187] JitPack – Publish JVM and Android libraries. Retrieved July 17, 2025 from https://jitpack.io/.

[188] G. Jungwirth, A. Saha, M. Schröder, T. Fiebig, M. Lindorfer, and J. Cito. Connecting the .dotfiles: Checked-In Secret Exposure with Extra (Lateral Movement) Steps. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. (2024). doi:10.1109/MSR59073.2023.00051.

[189] K. G. Kalu, T. Singla, C. Okafor, S. Torres-Arias, and J. C. Davis. An Industry Interview Study of Software Signing for Supply Chain Security. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*. (2025).

[190] Kaspersky. Roaming Mantis implements new DNS changer in its malicious mobile app in 2022. Archived at https://archive.ph/3x4cG. (2023). https://securelist.com/roaming-mantis-dns-changer-in-malicious-mobile-app/108464/.

[191] J. Kim, H. Choi, H. Namkung, W. Choi, B. Choi, H. Hong, Y. Kim, J. Lee, and D. Han. Enabling Automatic Protocol Behavior Analysis for Android Applications. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. (2016). doi:10.1145/2999572.2999596.

[192] S. Klivan, S. Höltervennhoff, R. Panskus, K. Marky, and S. Fahl. Everyone for Themselves? A Qualitative Study about Individual Security Setups of Open Source Software Contributors. In *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. (2024). doi:10.1109/SP54263.2024.00214.

[193] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry. Characterizing the Security of Github CI Workflows. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. (2022).

[194] C. König, D. Schmidt, P. König, P. Felbauer, and S. Schrittwieser. SaMBA: Increasing Mixed Boolean-Arithmetic Complexity Through Equality Saturation. In *Proceedings of the 21st ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2026).

[195] B. Könings, C. Bachmaier, F. Schaub, and M. Weber. Device names in the wild: investigating privacy risks of zero configuration networking. In *Proceedings of the 14th International Conference on Mobile Data Management*. (2013). doi:10.1109/MDM.2013.65.

[196] A. Krause, J. H. Klemmer, N. Huaman, D. Wermke, Y. Acar, and S. Fahl. Pushed by Accident: A Mixed-Methods Study on Strategies of Handling Secret Information in Source Code Repositories. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. (2023).

[197] D. Kuchhal and F. Li. Knock and Talk: Investigating Local Network Communications on Websites. In *Proceedings of the 21st Internet Measurement Conference (IMC)*. (2021). doi:10.1145/3487552.3487857.

[198] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric. All Things Considered: An Analysis of IoT Devices on Home Networks. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. (2019). doi:10.5555/3361338.3361419.

[199] R. Kumar, A. Virkud, R. S. Raman, A. Prakash, and R. Ensafi. A Large-scale Investigation into Geodifferences in Mobile Apps. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. (2022).

[200] Laburity Research Team. Research Case Study: Supply Chain Security at Scale Insights into NPM Account Takeovers. Archived at https://archive.ph/R12Lb. (2024). https://laburity.com/research-npm-account-takeovers/.

[201] P. Ladisa, H. Plate, M. Martinez, and O. Barais. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *Proceedings of the 44th IEEE Symposium on Security & Privacy (S&P)*. (2023). doi:10.1109/SP46215.2023.10179304.

[202] P. Ladisa, M. Sahin, S. E. Ponta, M. Rosa, M. Martinez, and O. Barais. The Hitchhiker's Guide to Malicious Third-Party Dependencies. In *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*. (2023). doi:10.1145/3605770.3625212.

[203] lakru. TcpClient.Connect() on iOS 14 triggers local network access permissions required. Archived at https://archive.ph/Sd7u2. (2020). https://forum.unity.com/threads/tcpclient-connect-on-ios-14-triggers-local-network-access-permissions-required.1000701/.

[204] P. Lamkin. Report Claims Ring Employees Had Unfettered Access To Security Camera Footage. (2019). Retrieved July 30, 2021 from https://www.forbes.com/sites/paullamkin/2019/01/11/report-claims-ring-employees-had-unfettered-access-to-security-camera-footage/.

[205] Lareina Yee. McKinsey – The state of AI: How organizations are rewiring to capture value. Archived at https://archive.ph/h26UW. (2025). https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai/.

[206] F. Li and V. Paxson. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2017). doi:10.1145/3133956.3134072.

[207] L. Li et al. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. (2015). doi:10.1109/ICSE.2015.48.

[208] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. In *Proceedings of the 39th International Computers on Software & Applications Conference (COMPSAC)*. (2015). doi:10.1109/COMPSAC.2015.103.

[209] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie. A Large-Scale Empirical Study on Android Runtime-Permission Rationale Messages. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. (2018). doi:10.1109/VLHCC.2018.8506574.

[210] S. Lounici, M. Rosa, C. Negri, S. Trabelsi, and M. Önen. Optimizing Leak Detection in Open-source Platforms with Machine Learning Techniques. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy*. (2021). doi:10.5220/0010238101450159.

[211] G. Lyon. Nmap: the Network Mapper. v7.80. (2019). Retrieved Mar. 30, 2022 from https://nmap.org/.

[212] A. M. Mandalari, D. J. Dubois, R. Kolcun, M. T. Paracha, H. Haddadi, and D. Choffnes. Blocking Without Breaking: Identification and Mitigation of Non-Essential IoT Traffic. In *Proceedings of the 21st Privacy Enhancing Technologies Symposium (PETS)*. (2021). doi:10.2478/popets-2021-0075.

[213] Markdown Guide – Getting Started. Archived at https://archive.ph/s01xz. https://www.markdownguide.org/getting-started/.

[214] J. Martínez and J. M. Durán. Software Supply Chain Attacks, a Threat to Global Cybersecurity: SolarWinds' Case Study. *International Journal of Safety and Security Engineering*, 11, 5, (2021). doi:10.18280/ijsse.110505.

[215] maryjoytattoo. Reddit – Why is Instagram asking for access to devices on a local network? Archived at https://archive.ph/3vE7w. (2020). https://www.reddit.com/r/techsupport/comments/kc67r3/why_is_instagram_asking_for_access_to_devices_on/.

[216] D. Mauro Junior, L. Melo, H. Lu, M. d'Amorim, and A. Prakash. A Study of Vulnerability Analysis of Popular Smart Devices Through Their Companion Apps. In *Proceedings of the IEEE Workshop on the Internet of Safe Things (SafeThings)*. (2019). doi:10.1109/SPW.2019.00042.

[217] Maven – Introduction to the Dependency Mechanism. Archived at https://archive.ph/94Psb. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html/.

[218] maven central repository – Why do I need to verify project ownership? Archived at https://archive.ph/rS1vp. https://central.sonatype.org/faq/verify-ownership/#answer.

[219] M. Meli, M. R. McNiece, and B. Reaves. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. (2019). doi:10.14722/ndss.2019.23418.

[220] A. Mendoza and G. Gu. Mobile Application Web API Reconnaissance: Web-to-Mobile Inconsistencies & Vulnerabilities. In *Proceedings of the 39th IEEE Symposium on Security & Privacy (S&P)*. (2018). doi:10.1109/SP.2018.00039.

[221] Microsoft. Microsoft Bounty Programs. https://www.microsoft.com/en-us/msrc/bounty.

[222] Microsoft Corporation. App Store – Microsoft Edge: KI-Browser. Version: 122.2365.56. https://apps.apple.com/us/app/microsoft-edge-ki-browser/id1288723196.

[223] Microsoft Corporation. App Store – Microsoft Whiteboard. Archived at https://archive.ph/dWhUu. https://apps.apple.com/us/app/microsoft-whiteboard/id1352499399.

[224] Microsoft Corporation. Mono interpreter on iOS and Mac Catalyst. Archived at https://archive.ph/YkA6b. https://learn.microsoft.com/en-us/dotnet/maui/macios/interpreter.

[225] Microsoft Corporation. Native AOT deployment on iOS and Mac Catalyst. Archived at https://archive.ph/MG4HP. https://learn.microsoft.com/en-us/dotnet/maui/deployment/nativeaot.

[226] Microsoft Corporation. Xamarin. Archived at https://archive.ph/29xDC. https://dotnet.microsoft.com/en-us/apps/xamarin.

[227] C. Miller, M. Jahanshahi, A. Mockus, B. Vasilescu, and C. Kastner. Understanding the Response to Open-Source Dependency Abandonment in the npm Ecosystem. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. (2025). doi:10.1109/ICSE55347.2025.00004.

[228] MobiVM. Archived at https://archive.ph/PU5uc. https://mobivm.github.io/.

[229] R. Mohamed, A. Arunasalam, H. Farrukh, J. Tong, A. Bianchi, and Z. B. Celik. ATTention Please! An Investigation of the App Tracking Transparency Permission. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*. (2024).

[230] Mono Project. Mono open source ECMA CLI, C# and .NET implementation. Archived at https://archive.ph/Rtbuv. https://github.com/mono/mono/.

[231] Moonsharp. A Lua interpreter written entirely in C# for the .NET, Mono and Unity platforms. Archived at: https://archive.ph/pp6zg. https://www.moonsharp.org/.

[232] Mozilla. App Store – Firefox: Private, Safe Browser. Archived at https://archive.ph/LEKJo. https://apps.apple.com/us/app/firefox-private-safe-browser/id989804926.

[233] Mozilla Foundation. Mozilla Location Service. Archived at https://archive.ph/53NwD. https://location.services.mozilla.com/.

[234] Namecheap – Bulk Domain Search – Multiple Domain Checker. https://www.namecheap.com/domains/bulk-domain-search/.

[235] Y. Nan, X. Wang, L. Xing, X. Liao, R. Wu, J. Wu, Y. Zhang, and X. Wang. Are You Spying on Me? Large-Scale Analysis on IoT Data Exposure through Companion Apps. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. (2023).

[236] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (2018). doi:10.14722/ndss.2018.23092.

[237] R. Nasoi. Twitter – Bluetooth. Archived at https://archive.ph/c5ZWb. (2021). https://twitter.com/roxananasoi/status/1442205247624159233.

[238] NAVER Z Corporation. App Store – ZEPETO: Avatar, Connect & Live. Archived at https://archive.ph/2fz7N. https://apps.apple.com/us/app/zepeto-avatar-connect-live/id1350301428.

[239] S. Neupane, G. Holmes, E. Wyss, D. Davidson, and L. D. Carli. Beyond typosquatting: an in-depth look at package confusion. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. (2023).

[240] S. Neupane, F. Tazi, U. Paudel, F. V. Baez, M. Adamjee, L. De Carli, S. Das, and I. Ray. On the Data Privacy, Security, and Risk Postures of IoT Mobile Companion Apps. In *Proceedings of the 36th IFIP Annual Conference on Data and Applications Security and Privacy (DBSec)*. (2022). doi:10.1007/978-3-031-10684-2_10.

[241] J. L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, and M. Sridharan. IOTA: a calculus for internet of things automation. In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (2017). doi:10.1145/3133850.3133860.

[242] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel. IotSan: Fortifying the Safety of IoT Systems. In *Proceedings of the 14th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. (2018). doi:10.1145/3281411.3281440.

[243] NicolasFNino and S. Arzt. GitHub FlowDroid Issue #601 – FlowDroid with EPICC. (2023). Retrieved Apr. 27, 2023 from https://github.com/secure-software-engineering/FlowDroid/issues/601.

[244] G. Niedziela. YouTube – $130,000+ Learn New Hacking Technique in 2021 – Dependency Confusion – Bug Bounty Reports Explained. (2021). https://www.youtube.com/watch?v=zFHJwehpBrU.

[245] NIST. CVE-2017-8866. (2017). Retrieved July 26, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2017-8866.

[246] NIST. CVE-2019-16732. (2019). Retrieved July 26, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2019-16732.

[247] NIST. CVE-2022-30271. (2022). Retrieved July 26, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2022-30271.

[248] L. Niu, S. Mirza, Z. Maradni, and C. Pöpper. CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. (2023).

[249] npm – Home. Archived at https://archive.ph/ImYpo. https://www.npmjs.com/.

[250] M. Ohm, H. Plate, A. Sykosch, and M. Meier. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. (2020). doi:10.1007/978-3-030-52683-2_2.

[251] M. Ohm and C. Stuke. SoK: Practical Detection of Software Supply Chain Attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security (ARES)*. (2023). doi:10.1145/3600160.3600162.

[252] F. R. Olivera. Maven Repository. Retrieved Apr. 5, 2023 from https://mvnrepository.com/.

[253] Opera Software AS. App Store – Opera Browser. Version: 4.5.0. https://apps.apple.com/us/app/opera-browser-with-vpn-and-ai/id1411869974.

[254] A.-M. Ortloff, M. Fassl, A. Ponticello, F. Martius, A. Mertens, K. Krombholz, and M. Smith. Different Researchers, Different Results? Analyzing the Influence of Researcher Experience and Data Type During Qualitative Analysis of an Interview and Survey Study on Security Advice. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. (2023). doi:10.1145/3544548.3580766.

[255] C. Osborne. The Daily Swig – Malicious Python library CTX removed from PyPI repo. Archived at https://archive.ph/oLj9G. (2022). https://portswigger.net/daily-swig/malicious-python-library-ctx-removed-from-pypi-repo.

[256] Oversecured. Introducing MavenGate: A Supply Chain Attack Method for Java and Android Applications. Archived at https://archive.ph/Cj0GE. (2024). https://blog.oversecured.com/Introducing-MavenGate-a-supply-chain-attack-method-for-Java-and-Android-applications/.

[257] OWASP Foundation. OWASP Mobile Top 10. Archived at: https://archive.ph/fspDN. https://owasp.org/www-project-mobile-top-10/.

[258] Y. Ozery, A. Nadler, and A. Shabtai. Information Based Heavy Hitters for Real-Time DNS Data Exfiltration Detection. In *Proceedings of the 31st Network and Distributed System Security Symposium (NDSS)*. (2024). doi:10.14722/ndss.2024.24388.

[259] palera1n. Version: v2.0.0-beta.8. https://palera.in/.

[260] B. Pan. GitHub – Dex2jar. v2.1. (2021). Retrieved Jan. 15, 2022 from https://github.com/pxb1988/dex2jar.

[261] M. T. Paracha, D. J. Dubois, N. Vallina-Rodriguez, and D. Choffnes. IoTLS: Understanding TLS Usage in Consumer IoT Devices. In *Proceedings of the 21st Internet Measurement Conference (IMC)*. (2021). doi:10.1145/3487552.3487830.

[262] E. Pariwono, D. Chiba, M. Akiyama, and T. Mori. Don't Throw Me Away: Threats Caused by the Abandoned Internet Resources Used by Android Apps. In *Proceedings of the 13th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2018). doi:10.1145/3196494.3196554.

[263] PayPal, Inc. App Store – PayPal Business. Archived at https://archive.ph/C3sAq. https://apps.apple.com/us/app/paypal-business/id1053148887.

[264] Paytm. Secure UPI Payments. Archived at: https://archive.ph/rPnAX. https://paytm.com/.

[265] Phlips. Philips Hue – Smart lightning. Archived at https://archive.ph/kcSUG. https://www.philips-hue.com.

[266] pip – PyPI. Archived at https://archive.ph/J7eOL. https://pypi.org/project/pip/.

[267] S. Pletinckx, K. Borgolte, and T. Fiebig. Out of Sight, Out of Mind: Detecting Orphaned Web Pages at Internet-Scale. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2021). doi:10.1145/3460120.3485367.

[268] A. Pradeep, Á. Feal, J. Gamba, A. Rao, M. Lindorfer, N. Vallina-Rodriguez, and D. Choffnes. Not Your Average App: A Large-scale Privacy Analysis of Android Browsers. In *Proceedings of the 23rd Privacy Enhancing Technologies Symposium (PETS)*. (2023). doi:10.56553/popets-2023-0003.

[269] A. Pradeep et al. A comparative analysis of certificate pinning in Android & iOS. In *Proceedings of the 22nd Internet Measurement Conference (IMC)*. (2022). doi:10.1145/3517745.3561439.

[270] Prolific. Archived at https://archive.ph/pxBDF. https://www.prolific.com/.

[271] Publishing a Crate to Crates.io – The Rust Programming Language. Archived at https://archive.ph/hVUXD. https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html.

[272] Qualtrics. Archived at https://archive.ph/sixtD. https://www.qualtrics.com/.

[273] L. Quinn. Whats the difference between IPA and APK? Archived at https://archive.ph/llOi9. (2022). https://lovequinn.medium.com/whats-the-difference-between-ipa-and-apk-eff81fb0c61b.

[274] K. Rahkema and D. Pfahl. SwiftDependencyChecker: detecting vulnerable dependencies declared through CocoaPods, carthage and swift PM. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESOFT)*. (2022). doi:10.1145/3524613.3527806.

[275] M. R. Rahman, N. Imtiaz, M.-A. Storey, and L. Williams. Why Secret Detection Tools Are Not Enough: Its Not Just about False Positives - An Industrial Case Study. *Empirical Software Engineering*, 27, 3, (2022). doi:10.1007/s10664-021-10109-y.

[276] M. Rahman. AndroidAuthority – Google Play will no longer pay to discover vulnerabilities in popular Android apps. Archived at https://archive.ph/1RKjo. (2024). https://www.androidauthority.com/google-play-security-reward-program-winding-down-3472376/.

[277] Raiffeisen Bank Romania S.A. App Store – Noul Raiffeisen Smart Mobile. Archived at https://archive.ph/7GjQk. https://apps.apple.com/us/app/noul-raiffeisen-smart-mobile/id1255136212.

[278] M. Rapoport, P. Suter, E. Wittern, O. Lhotak, and J. Dolby. Who You Gonna Call? Analyzing Web Requests in Android Applications. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. (2017). doi:10.1109/MSR.2017.11.

[279] O. A. V. Ravnås. Frida. Version: 16.0.7. https://frida.re/.

[280] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (2018). doi:10.14722/ndss.2018.23009.

[281] ReallyGene. Reddit – WiFi Password. Archived at https://archive.ph/guiDq. (2022). https://www.reddit.com/r/Nest/comments/u0qgzw/comment/i49br82/.

[282] J. Reardon. The Curious Case of Coulus Coelib. Archived at https://archive.ph/atF7n. (2022). https://blog.appcensus.io/2022/04/06/the-curious-case-of-coulus-coelib/.

[283] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman. 50 Ways to Leak Your Data: An Exploration of Apps Circumvention of the Android Permissions System. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. (2019).

[284] Reddit – Other Devices See Phone. Archived at https://archive.ph/mu4zq. (2021). https://www.reddit.com/r/ios/comments/m0vgqu/comment/gqb8v8c/.

[285] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices. In *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*. (2019). doi:10.1109/SP40001.2021.00066.

[286] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *Proceedings of the 19th Internet Measurement Conference (IMC)*. (2019). doi:10.1145/3355369.3355577.

[287] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (2018). doi:10.14722/ndss.2018.23143.

[288] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. (2016). doi:10.1145/2906388.2906392.

[289] S. Rice. Twitter – Local network access discussion. Archived at https://archive.ph/svQvq. (2021). https://twitter.com/DrumLordJr/status/1461590761187655685.

[290] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo. Unmasking the veiled: A comprehensive analysis of Android evasive malware. In *Proceedings of the 19th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2024).

[291] E. Rye and D. Levin. Surveilling the Masses with Wi-Fi-Based Positioning Systems. In *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. (2024).

[292] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera. Secrets in Source Code: Reducing False Positives Using Machine Learning. In *Proceedings of the International Conference on COMmunication Systems & NETworkS (COMSNETS)*. (2020). doi:10.1109/COMSNETS48256.2020.9027350.

[293] S. J. Saidi, S. Matic, O. Gasser, G. Smaragdakis, and A. Feldmann. Deep Dive into the IoT Backend Ecosystem. In *Proceedings of the 22nd Internet Measurement Conference (IMC)*. (2022). doi:10.1145/3517745.3561431.

[294] R. Sammak, A. L. Rotthaler, H. S. Ramulu, D. Wermke, and Y. Acar. Developers' Approaches to Software Supply Chain Security: An Interview Study. In *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*. (2024). doi:10.1145/3689944.3696160.

[295] E. F. D. Santos. Dependency Management in iOS Development: A Developer Survey Perspective. In *Proceedings of the 10th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESOFT)*. (2024). doi:10.1145/3647632.3647992.

[296] S. Scalco, R. Paramitha, D.-L. Vu, and F. Massacci. On the feasibility of detecting injections in malicious npm packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security (ARES)*. (2022). doi:10.1145/3538969.3543815.

[297] D. Schmidt, A. Ponticello, M. Steinböck, K. Krombholz, and M. Lindorfer. Analyzing the iOS Local Network Permission from a Technical and User Perspective. In *Proceedings of the 46th IEEE Symposium on Security & Privacy (S&P)*. (2025). doi:10.1109/SP61157.2025.00045.

[298] D. Schmidt and S. Schrittwieser. Replay Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *Proceedings of the Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks*. (2025).

[299] D. Schmidt, S. Schrittwieser, and E. Weippl. Leaky Apps: Large-scale Analysis of Secrets Distributed in Android and iOS Apps. In *Proceedings of the 32nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2025).

[300] D. Schmidt, S. Schrittwieser, and E. Weippl. Supply Chain Insecurity: Exposing Vulnerabilities in iOS Dependency Management Systems. (2026). arXiv: 2601.20638.

[301] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full Comparison of Valid Credentials Table. https://github.com/CDL-AsTra/leaky_apps/blob/main/tables/comparison.md.

[302] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full Dependency Management Table. https://github.com/CDL-AsTra/leaky_apps/blob/main/tables/dependencies.md.

[303] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full List of Services. https://github.com/CDL-AsTra/leaky_apps/blob/main/analysis/rules.md.

[304] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full Valid Credential Table. https://github.com/CDL-AsTra/leaky_apps/blob/main/tables/service_table.md.

[305] D. Schmidt, C. Tagliaro, K. Borgolte, and M. Lindorfer. IoTFlow: Inferring IoT Device Behavior at Scale through Static Mobile Companion App Analysis. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2023). doi:10.1145/3576915.3623211.

[306] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, Inc., (2000). ISBN: 978-0-471-45380-2.

[307] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. Weippl. Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. (2012).

[308] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi. An Investigation into Android Run-time Permissions from the End Users' Perspective. In *Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESOFT)*. (2018). doi:10.1145/3197231.3197236.

[309] B. Shen, L. Wei, C. Xiang, Y. Wu, M. Shen, Y. Zhou, and X. Jin. Can Systems Explain Permissions Better? Understanding Users' Misperceptions under Smartphone Runtime Permission Model. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. (2021).

[310] Shodan. Shodan Search. Retrieved July 18, 2022 from https://www.shodan.io/search?query=has_screenshot:true+port:554.

[311] Signify Netherlands B.V. App Store – Philips Hue. Version: 4.39.0. https://apps.apple.com/us/app/philips-hue/id1055281310.

[312] C. Silva. Mashable – TikTok users' favorite moments from the TikTok congressional hearing. Archived at https://archive.ph/gfkIV. (2023). https://mashable.com/article/tiktok-congressional-hearing.

[313] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani. Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. In *Proceedings of the 12th International Conference on Mining Software Repositories (MSR)*. (2015). doi:10.1109/MSR.2015.48.

[314] P. Sivakumaran and J. Blasco. A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. (2019). doi:10.5555/3361338.3361340.

[315] P. Sivakumaran, C. Zuo, Z. Lin, and J. Blasco. Uncovering Vulnerabilities of Bluetooth Low Energy IoT from Companion Mobile Apps with Ble-Guuide. In *Proceedings of the 18th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2023). doi:10.1145/3579856.3595806.

[316] V. Sivaraman, D. Chan, D. Earl, and R. Boreli. Smart-Phones Attacking Smart-Homes. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WISEC)*. (2016). doi:10.1145/2939918.2939925.

[317] Smilegate Holdings, Inc. App Store – Epic Seven. Archived at https://archive.ph/ZBZl2. https://apps.apple.com/us/app/epic-seven/id1322399438.

[318] J. Soref, wraithgar, drew4237, and L. Karrys. npm Docs – npm install. Archived at https://archive.ph/5OdJY. https://docs.npmjs.com/cli/v11/commands/npm-install.

[319] Statcounter. Mobile Operating System Market Share United States Of America. Archived at: https://archive.ph/dfNY1. https://gs.statcounter.com/os-market-share/mobile/united-states-of-america.

[320] M. Steinböck, J. Bleier, M. Rainer, T. Urban, C. Utz, and M. Lindorfer. Comparing Apples to Androids: Discovery, Retrieval, and Matching of iOS and Android Apps for Cross-Platform Analyses. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. (2024).

[321] Swift.org. Swift.org – Package Manager. Archived at https://archive.ph/4i8F5. https://www.swift.org/documentation/package-manager/.

[322] C. Tagliaro, F. Hahn, R. Sepe, A. Aceti, and M. Lindorfer. I Still Know What You Watched Last Sunday: Privacy of the HbbTV Protocol in the European Smart TV Landscape. In *Proceedings of the 30th Network and Distributed System Security Symposium (NDSS)*. (2023). doi:10.14722/ndss.2023.24102.

[323] M. Tahaei, R. Abu-Salma, and A. Rashid. Stuck in the Permissions With You: Developer & End-User Perspectives on App Permissions & Their Privacy Ramifications. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. (2023). doi:10.1145/3544548.3581060.

[324] J. Tan, K. Nguyen, M. Theodorides, H. Negrón-Arroyo, C. Thompson, S. Egelman, and D. Wagner. The Effect of Developer-Specified Explanations for Permission Requests on Smartphone User Behavior. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. (2014). doi:10.1145/2556288.2557400.

[325] Z. Tan, S. P. Parambath, C. Anagnostopoulos, J. Singer, and A. K. Marnerides. Advanced Persistent Threats Based on Supply Chain Vulnerabilities: Challenges, Solutions, and Future Directions. *IEEE Internet of Things Journal*, 12, 6, (2025). doi:10.1109/JIOT.2025.3528744.

[326] Z. Tang, K. Tang, M. Xue, Y. Tian, S. Chen, M. Ikram, T. Wang, and H. Zhu. iOS, Your OS, Everybodys OS: Vetting and Analyzing Network Services of iOS Applications. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. (2020).

[327] Tantan – Privacy Policy. Archived at https://archive.ph/iNhVo. http://lp.tantanapp.com/and_play/?id=17.

[328] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi. SpellBound: Defending Against Package Typosquatting. In *Proceedings of the 14th International Conference on Network and System Security (NSS)*. (2020). doi:10.1007/978-3-030-65745-1_7.

[329] Tcpdump Group. TCPDUMP & LIBPCAP. Version: 4.9.3. https://www.tcpdump.org/.

[330] The Cargo Book. Archived at https://archive.ph/9OPy9. https://doc.rust-lang.org/cargo/index.html.

[331] O. Therox. CocoaPods Blog – CocoaPods Trunk Read-only Plan. Archived at https://archive.ph/rs9OM. https://blog.cocoapods.org/CocoaPods-Specs-Repo/.

[332] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862. (2007). https://www.rfc-editor.org/info/rfc4862.

[333] M. Tileria, J. Blasco, and G. Suarez-Tangil. WearFlow: Expanding Information Flow Analysis To Companion Apps in Wear OS. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. (2020).

[334] M. Tran. Reddit – AirPlay. Archived at https://archive.ph/Z9YsS. (2021). https://www.reddit.com/r/ios/comments/m0vgqu/did_google_just_bypass_local_network_permission.

[335] Trufflesecurity. GitHub – TruffleHog. Version: 3.84.0. https://github.com/trufflesecurity/trufflehog/.

[336] United Command International Ltd. App Store – Jelly Blast. Archived at https://archive.ph/ydPsK. https://apps.apple.com/us/app/jelly-blast/id372948897.

[337] Unity. Unity Real-Time Development Platform. Archived at https://archive.ph/qwkgW. https://unity.com/.

[338] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *Proceedings of the 9th Compiler Construction*. (2000). doi:10.1007/3-540-46423-9_2.

[339] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft. Breaking for Commercials: Characterizing Mobile Advertising. In *Proceedings of the 12th Internet Measurement Conference (IMC)*. (2012). doi:10.1145/2398776.2398812.

[340] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq. The TV is Smart and Full of Trackers: Measuring Smart TV Advertising and Tracking. In *Proceedings of the 20th Privacy Enhancing Technologies Symposium (PETS)*. (2020). doi:10.2478/popets-2020-0021.

[341] D.-L. Vu, Z. Newman, and J. S. Meyers. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. (2023). doi:10.1109/ICSE48619.2023.00052.

[342] J. Walton, J. Steven, J. Manico, K. Wall, R. Iramar, and contributors. Certificate and Public Key Pinning. Retrieved Oct. 7, 2022 from https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning.

[343] H. Wang, S. Guo, J. He, H. Liu, T. Zhang, and T. Xiang. Model Supply Chain Poisoning: Backdooring Pre-trained Models via Embedding Indistinguishability. In *Proceedings of the World Wide Web Conference (WWW)*. (2025). doi:10.1145/3696410.3714624.

[344] Q. Wang, W. Ul Hassan, A. Bates, and C. A. Gunter. Fear and Logging in the Internet of Things. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (2018). doi:10.14722/ndss.2018.23282.

[345] X. Wang, Y. Sun, S. Nanda, and X. Wang. Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. (2019). doi:10.5555/3361338.3361418.

[346] Y. Wang, X. Liu, W. Mao, and W. Wang. DCDroid: Automated Detection of SSL/TLS Certificate Verification Vulnerabilities in Android Apps. In *Proceedings of the ACM Turing Celebration Conference (TUR-C)*. (2019). doi:10.1145/3321408.3326665.

[347] T. Watanabe et al. Understanding the Origins of Mobile App Vulnerabilities: A Large-Scale Measurement Study of Free and Paid Apps. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. (2017). doi:10.1109/MSR.2017.23.

[348] Wattpad Corp. App Store – Wattpad - Read & Write Stories. Archived at https://archive.ph/k1ukh. https://apps.apple.com/us/app/wattpad-read-write-stories/id306310789.

[349] E. Wen, J. Wang, and J. Dietrich. SecretHunter: A Large-scale Secret Scanner for Public Git Repositories. In *Proceedings of the 21st IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. (2022). doi:10.1109/TrustCom56396.2022.00028.

[350] H. Wen, J. Li, Y. Zhang, and D. Gu. An Empirical Study of SDK Credential Misuse in iOS Apps. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. (2018). doi:10.1109/APSEC.2018.00040.

[351] H. Wen, Q. Zhao, Q. A. Chen, and Z. Lin. Automated Cross-Platform Reverse Engineering of CAN Bus Commands from Mobile Apps. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*. (2020). doi:10.14722/ndss.2020.24231.

[352] J. Williams. What You Need to Know About the SolarWinds Supply-Chain Attack. Archived at https://archive.ph/Wb3mn. (2020). https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/.

[353] D. Wood, N. Apthorpe, and N. Feamster. Cleartext Data Transmissions in Consumer IoT Medical Devices. In *Proceedings of the Workshop on Internet of Things Security and Privacy (IoT-S&P)*. (2017). doi:10.1145/3139937.3139939.

[354] D. Wu, D. Gao, R. Chang, E. He, E. Cheng, and R. Deng. Understanding Open Ports in Android Applications: Discovery, Diagnosis, and Security Assessment. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. (2019). doi:10.14722/ndss.2019.23171.

[355] E. Wyss, L. De Carli, and D. Davidson. What the Fork? Finding Hidden Code Clones in npm. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. (2022). doi:10.1145/3510003.3510168.

[356] E. Wyss, A. Wittman, D. Davidson, and L. De Carli. Wolf at the Door: Preventing Install-Time Attacks in npm with Latch. In *Proceedings of the 17th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2022). doi:10.1145/3488932.3523262.

[357] H. Xu, M. Yu, Y. Wang, Y. Liu, Q. Hou, Z. Ma, H. Duan, J. Zhuge, and B. Liu. Trampoline Over the Air: Breaking in IoT Devices Through MQTT Brokers. In *Proceedings of the 7th IEEE European Symposium on Security & Privacy (EuroS&P)*. (2022). doi:10.1109/EuroSP53844.2022.00019.

[358] D. Yeke, M. Ibrahim, G. S. Tuncay, H. Farrukh, A. Imran, A. Bianchi, and Z. B. Celik. Wears my Data? Understanding the Cross-Device Runtime Permission Model in Wearables. In *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. (2024). doi:10.1109/SP54263.2024.00077.

[359] yokotayokota, S. Arzt, and J. Samhi. GitHub FlowDroid Issue #386 – Usage of IccTA in Flowdroid. (2021). Retrieved Apr. 27, 2023 from https://github.com/secure-software-engineering/FlowDroid/issues/386.

[360] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams. What are Weak Links in the npm Supply Chain? In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. (2022). doi:10.1145/3510457.3513044.

[361] J. Zhang, K. Huang, Y. Huang, B. Chen, R. Wang, C. Wang, and X. Peng. Killing Two Birds with One Stone: Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence. *ACM Transactions on Software Engineering and Methodology*, 34, 4, (2025). doi:10.1145/3705304.

[362] Y. Zhang, Y. Yang, and Z. Lin. Don't Leak Your Keys: Understanding, Measuring, and Exploiting the AppSecret Leaks in Mini-Programs. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2023). doi:10.1145/3576915.3616591.

[363] Q. Zhao, C. Zuo, J. Blasco, and Z. Lin. PeriScope: Comprehensive Vulnerability Analysis of Mobile App-defined Bluetooth Peripherals. In *Proceedings of the 17th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (2022). doi:10.1145/3488932.3517410.

[364] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin. Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *Proceedings of the 41st IEEE Symposium on Security & Privacy (S&P)*. (2020). doi:10.1109/SP40000.2020.00072.

[365] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang. Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. (2019). doi:10.5555/3361338.3361417.

[366] X. Zhou, Y. Zhang, W. Niu, J. Liu, H. Wang, and Q. Li. An Analysis of Malicious Packages in Open-Source Software in the Wild. In *Proceedings of the 55th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. (2025). doi:10.1109/DSN64029.2025.00045.

[367] Y. Zhou, L. Wu, Z. Wang, and X. Jiang. Harvesting Developer Credentials in Android Apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WISEC)*. (2015). doi:10.1145/2766498.2766499.

[368] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: a study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. (2019).

[369] O. Zungur, A. Bianchi, G. Stringhini, and M. Egele. AppJitsu: Investigating the Resiliency of Android Applications. In *Proceedings of the 6th IEEE European Symposium on Security & Privacy (EuroS&P)*. (2021). doi:10.1109/EuroSP51992.2021.00038.

[370] C. Zuo, Z. Lin, and Y. Zhang. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*. (2019). doi:10.1109/SP.2019.00009.

[371] C. Zuo, H. Wen, Z. Lin, and Y. Zhang. Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (2019). doi:10.1145/3319535.3354240.