Engineering & Physical Sciences
School of Computer Science
University of Birmingham
United Kingdom

—Breaking Boundaries—

Security Analysis of the Interfaces Between Applications, Systems and Enclaves

Abdulla Aldoseri
[redacted]

Supervisors
Prof. David Oswald, Prof. Flavio Garcia
[redacted]

A thesis presented for the degree of
Doctor of Philosophy in Computer Science

April 24, 2023

Abstract

Application interfaces allow apps to communicate with each other or use resources. Several platforms, namely: browser, mobile and computer, offer various instances of these interfaces at different architecture levels. The interfaces range from simply sending and receiving data to accessing hardware resources. Due to the increase in introducing services across several platforms, there has been limited research on the impact of interference between services and interfaces. Additionally, platforms provide permissions and policies that serve as an authorisation layer to counter the rising security issues of these interfaces. In this thesis, we aim to tackle this issue and contribute to this research area by analysing a subset of these interfaces, addressing their common weaknesses in their respective platform, and assessing their attack surface.

In the first part of the thesis, we study and evaluate interfaces for the browser platform: local schemes in mobile browsers and hardware application programming interfaces (APIs) in desktop browsers. Our study demonstrates several security issues within these interfaces, ranging from spoofing to privilege escalation. As a result, introducing components like new input methods, output methods, internal processes, and different contexts is crucial in affecting interface security.

In the second part, we move to the mobile platform. We analyse the security of the mobile app interfaces. We consider new services like background restriction policy and multi-user profile features that interfere with mobile interfaces. Our study demonstrates threats that bypass the proposed security models of these services. We find that evaluating new services and understanding their correlation with existing interfaces is essential to introduce them to a platform.

Finally, in the third part of the thesis, we focus on analysing the trusted execution environment (TEE) platform. Previous studies show substantial efforts to ensure secure, trusted shielding runtime. However, its attack surface is not generally understood. Therefore, we evaluate the security of enclave interfaces and their TEE applications, namely remote attestation. We present a side-channel attack in the intel SGX enclave that leaks confidential data and demonstrate weaknesses in the design of hardware-based remote attestation protocols: Samsung Knox V2 and Key attestation.

We conclude that the area of interface security is vast. Platforms regularly introduce components like input methods, output methods, internal services and different contexts. Introducing these components to the platforms increases its attack surface. Furthermore, these components shape a complex factor in evaluating these interfaces. Platform developers should be aware of such an issue, and new methods need to be proposed to assess the attack surfaces of these interfaces.

Acknowledgement

بسم الله والصلاة والسلام على رسول الله. والحمدلله لله كثيرا.

Thank god

Every PhD has a unique story, and mine was wonderful. It was a journey filled with challenges, new experiences, and enjoyable moments. I am grateful to everyone who assisted me in my studies. The School of Computer Science offered a variety of social activities and events, such as the cookie break on Tuesday (I miss the British cookies :P). Likewise, the Cyber security academic group organised excellent weekly seminars to debate ideas and present research (of course, in a PhD, no one should ask about anyone's progress until he talks about it). Following the sessions, there was a social group lunch.

I am grateful to the school's academic staff for teaching me valuable skills. I enjoyed every minute of working and collaborating with them, especially with my supervisor, David Oswald. I learned very much from him. He was very supportive and encouraging person. His guidance was invaluable, and he constantly pushed me to continue with my studies. He taught me the importance of comprehending the problem domain before assessing or analysing it. He taught me how to work independently on research. Now I feel ready to begin my own research. Flavio Garcia, my co-supervisor, mentioned to me at the beginning of my PhD that I might not be able to find answers online while conducting PhD research. Surprisingly, I found this after just about two weeks of working with him. Flavio was a one-of-a-kind co-supervisor who was supportive in proposing new scientific ideas and turning them into real-world research. Tom Chothia, a co-author of one of my publications, is constantly open to new ideas and enjoys working on challenging problems to make novel contributions. He is very welcome and supportive. I learned how to examine and express a problem in several aspects from him. Then, how to identify and select the appropriate approach for presenting the problem's answer. Mark Ryan and René Mayrhofer were my PhD viva examiners. We had extensive talks throughout the viva, examined numerous ideas, and discussed future work. I can honestly say that I enjoyed the viva. The examiners were asking pertinent and intriguing questions. Even though the viva lasted four hours, it seemed like a minute. I learned a lot from them, so thanks to both of them. Also, I would like thank all the staff of the school computer science, especially Peter Hancox, Kashif Rajpoot, Sarah Brookes, and Kara Crozier.

I am thankful to my friends: Ali Darwish, Ahmed Almohanadi, Abdulla Alharbi, Akram Alofi, Andreea-Ina Radu, Chris McMahon Stone, Chris Hicks, Daniel Clark, Eduard Marin, Fuad Hassan, Giorgos Vasilakis, Hassan Labni, Kit Murdock, Jose Moreira-Sanchez, Maria Verghelet, Michel, Mihai Ordean, Mo Zhang, Owen Pemberton, Salem Al Qassimi, Rajiv, Zhuang Xu, Zitai Chen, and many others. We had a great time discussing ideas, working together, participating in hackathons, having fun, and visiting places. They were very kind and supportive, so thank you all.

Nobody has been more important to me in my academic pursuits than my family members. I would like to thank my family for their encouragement and support while I pursue my education. Thanks to my wonderful family: Khalifa Aldoseri, Amina Alsubaie, Latifa Aldoseri, Mariam Aldoseri, Noora Aldoseri, Bashayer Aldoseri, Ahmed Aldoseri and Rashid Al Naebi.

Finally, I recall a day when my friends congratulated me on passing my PhD. "We did it," I simply replied to them. I chose "we" since my PhD would not have been possible without the help of my friends and family, so thank god and thank you everyone.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

BYOD Bring Your Own Device.

CORS CORS Cross-Origin Resource Sharing.

CSRF CSRF Cross-Site Request Forgery.

CVE Common Vulnerabilities and Exposures.

GUID GUID Globally Unique Identifiers.

IEE Isolated Execution Environment.

IME Input Method Editor.

OS Operating System.

PKI Public Key Infrastructure.

SDK Software Development Kit.

SDP Sensitive Data Protection.

SOP SOP Same-Origin Policy.

TEE Trusted Execution Environment.

UID Unique Identifier.

URI Uniform Resource Identifier.

XSS Cross-Site Scripting.

Chapter 1

Introduction

واني لما نظرت فيما يحتاج اليه الناس من الحساب وجدت جميع ذلك عددا.

"When I considered what people generally want in calculating, I found that it always is a number."

— *Mohammed Ibn Musa Al-Khwarizmi (THE ALGEBRA,831~833)*
Translated by Frederic Rosen, 1831 [152]

It starts with a line of code. It becomes a script. Then, a program. Finally, it occasionally becomes an application. This is the life cycle of application development. Applications ship on different platforms: mobile, desktop and over the Web. Currently, they have access to powerful device resources to perform numerous services through application programming interfaces (API). Furthermore, applications can communicate through channels to share data and serve end users. These channels are operated by so-called application interfaces (or entry points). Although these interfaces enable several beneficial features, they enlarge the attack surface of the applications and their platform. In this thesis, we study and evaluate these interfaces across different platforms from a security perspective considering their security model and attack surface.

1.1 Hello, Interfaces World

The applications' interfaces are entry points that allow applications to communicate with each other. They offer methods to send and receive data while ensuring a number of security properties like confidentiality and integrity. These security properties are applied to several authentication and authorisation mechanisms available at different levels: application, system, network, and hardware. Permissions and policies are primarily used to apply these mechanisms. Policies define rules that the operation should undertake, and permissions permit the use of a resource for a particular application. Such an abstraction is implemented differently in today's platforms but shares the same concepts. It can be seen in browsers, mobiles and computers as follows:

Firstly, Web browsers run Web applications. Each application is considered an origin. An origin is a protection domain grouped from Uniform Resource Identifiers (URIs) by browsers. Technically, each origin consists of a protocol, a hostname and a port number [178, 45]. It uses different browser schemes and protocols like HTTP and HTTPS. For protection, the browsers enforce policies to secure access to web applications. Among them is the Same-Origin policy (SOP), which ensures that origins cannot access each other's resources. It distinguishes origins by their scheme, host and port (e.g. http://example.com and https://example.com are considered two different origins) [178, 45]. This protection provides isolation and protects private session data like cookies and sessions. Browsers use permission-based APIs to secure access to computer resources like cameras and microphones. APIs are granted per origin to prevent unauthorised origins from accessing the computer resources.

Secondly, Android mobile OS uses a similar concept to Web browsers. Android use Linux user-based protection to isolate and protect apps from malicious apps. It assigns a unique ID (UID) for each app to set up a kernel-level application sandbox [166]. Generally, Android OS enforces policies and permissions to prevent applications from accessing each other's data. However, applications can communicate remotely via an online channel, locally via a shared file within the device storage and using app components. App components are entry points that provide access for applications to send and receive data from each other. There are four app components: activity, service, broadcast receiver and content provider. While they operate differently, they share the same purpose of sending and receiving data between apps [13]. Applications can declare and set custom permissions to restrict access to these components. Furthermore, Android OS provides permission-based APIs similar to browsers to access the device's resources [29].

Thirdly, computer platforms have recently taken an advanced step by introducing trusted computing. Trusted computing enables applications to operate in a trusted execution environment (TEE) protected by secure hardware. Technically, the computer's processor provide a mechanism to secure the runtime memory of these applications at the hardware level. Based on the computer architecture, TEE can be implemented differently and offers several permissions and policies. For instance, the processor Intel SGX architecture ensures the protection of memory regions where the trusted computation occurs at the hardware level. Trusted applications (named enclaves in the Intel SGX platform) require having ecall/ocall methods that function as interfaces to allow them to communicate with normal world applications [66, 102].

These platforms share a similar design concept where permission and policies contribute to safeguarding their interfaces. Bypassing these protections in an unauthorised manner refers to a privileges escalation attack where an unprivileged process under the adversary's control can abuse other applications with higher privileges [53]. In this work, we aim to study the attack surface of these interfaces considering their security model and their interaction with other services. However, first, we briefly introduce the technical design of these interfaces per their platform.

1.1.1 Browser platform: Schemes and Web APIs

Browsers utilise URI schemes to operate their procedures. These schemes act as interfaces for mobile browsers. They can be classified into remote schemes and local schemes. Remote schemes are used for Internet communications (e.g. HTTPS and FTP) and are generally known as Internet protocols. In contrast, local schemes operate locally and perform specific client-based tasks like accessing local files or rendering data (e.g. file:// and data://). Overall these schemes have a specific format and accept formatted input to function. Also, several policies and permissions have been applied to ensure users' confidentiality and privacy. Among them is the SOP. SOP provides per-origin isolation based on origins' information. Because local schemes have different properties, browsers treat them differently. For instance, some browsers assign each accessed document by File URI an implemented defined value as on origin. Then, SOP can distinguish between them [178, 45].

Remote schemes have received much attention from academics [72, 6] because of their important role in Internet communication. As they must comply to safeguard transmitted data, further protocols have been proposed that enhance their protection like HTTPS that accesses the Internet over a secure channel [151]. Conversely, local schemes have received limited attention. Some studies have demonstrated security issues in these schemes that include cross-site scripting (XSS) via the Javascript URI scheme [180], origin spoofing via the Data URI scheme [38, 43] and unauthorised access via the File URI in [192, 180].

In addition to schemes as browser interfaces, browsers introduce Web APIs (or so-called HTML5 APIs). These APIs allow access to devices' resources like cameras, GPS and microphones [137]. They are a safe replacement for deprecated browsers' plugins like Adobe Flash and Microsoft Silverlight. Similar to permissions and policies of mobile platforms, these APIs are permission-based and granted per app based on its origin. The few studies which evaluate the security model of these APIs demonstrated permission and leakage issues in Web APIs like GPS API [110] and screen sharing API [181]. Additionally, other studies have assessed the impact of abusing permitted Web API permissions for harvesting user data [126, 2].

1.1.2 Mobile platform: Interfaces across apps and resources

Mobile applications are isolated and sandboxed from each other. Apps can access device resources (e.g. camera and microphone) using APIs. The APIs are restricted by permission and policies enforced by the Android OS to ensure users' privacy and prevent unauthorised access to their resources. Similarly, applications use different interfaces known as app components to communicate with each other. There are four types of app components, namely: Activity, Service, Broadcast receiver and Content provider. The messages received by these interfaces are encapsulated as Android Intent for Activity, Service and Broadcast receiver types, and query for the Content provider [13]. Regarding access control over these components, app developers can define custom app permissions to restrict access to these interfaces. These permissions are declared in the Android manifest and can be assigned a protection level that defines the access restriction. Protection levels are normal, dangerous, signature and signatureOrSystem. For instance, a custom permission with a signature protection level allows only apps signed by the same developer key to obtain that permission [27].

The most common issue is confused deputy, where malicious apps abuse unrestricted app components of vulnerable apps to access their confidential data or use devices resources on their behalf (e.g. access to the camera and microphone). In this case, vulnerable apps miss configuring their app components by exposing them publicly (technically, setting their exported flag in their apps' manifest). This allows full access to these interfaces.

Security research of app components has taken two main directions. The first direction focuses on exploring and assessing the impact of misconfiguration issues that target these components. The second direction aims to build security tools that detect and identify these issues. In the case of the first direction, several studies have discussed instances of these issues [79, 150, 183, 69]. Because developed apps are error-prone and such an issue commonly occurs, in the second direction researchers tend to propose tools to detect such an issue either statically via taint techniques [124, 198, 74] or dynamically via monitoring runtime system services [195, 52]. These methods are not practical in real life as tainting techniques require a high level of resources to run [195], and runtime monitoring solutions require high privileges, which void some system functionality (e.g. Samsung Knox). Both directions aim to safeguard applications. The first aims to inspect and assess these issues, and the second aims to propose detection tools and evaluate their performance.

1.1.3 TEE: Enclave's interfaces and remote attestation protocol

In trusted computing, applications can run in a TEE, commonly known as a secure world. These applications (enclaves) are secured from an adversary-controlled host OS. They are known as enclaves in a single address-space TEE design (e.g., Intel SGX [66] and Sancus [143]) or two-worlds design like ARM Trustzone (e.g., ARM TrustZone [147] and Keystone [116]). In the first design model, an enclave runs in the address space of an unprivileged host application. The enclave can access unprotected memory locations outside the enclave, and the processor enforces a policy to prevent accessing the enclave memory from outside the enclave. Conversely, the two-world design model logically divides the processor into the "normal world" and the "secure world". Communications between these worlds are performed through a security monitor software layer via calling predefined entry points. Similarly to the first design, the processor enforces a policy that prevents normal world code from accessing the secure world memory and resources.

For communication, different mechanisms are deployed to ease the enclave's interaction with the normal world. For instance, Intel SGX offers an SDK that provides ecall/ocall mechanisms to switch between the normal and secure worlds [66]. They represent interfaces at the high-level abstraction of hardware-level isolation primitive to ease the development of enclaves.

However, over the last decade, several studies have demonstrated design issues that leak data from secure world memory. Exploiting these issues can lead to several attacks that include blind memory corruption attack in SGX [117] and hardware side-channel attacks by abusing the bit-flip issue in the design of the Intel processor [57]. Nevertheless, such contributions have served to sustain and improve the security of TEEs' design, leading to emerging applications that adopt TEE like remote attestation.

Remote attestation is a protocol that aims to provide an authentic, timely report for a third-party entity about the validity of the attested platform (device and/or app) [65]. It is based on the challenge-response protocol. Google's SafetyNet [91] is an Android implementation for remote attestation. Other implementations of the protocol have been proposed that utilise TEE, like Intel SGX remote attestation [101], Samsung Knox remote attestation [163, 155] and Android hardware key attestation [36].

Intel SGX remote attestation has been evaluated for security in [101, 42]. However, we noticed the mobile versions of these protocols receive less attention, with the exception of the early SafetyNet software-based version, which is easily compromised as described in [114, 55].

1.2 Dissertation Scope

Based on this brief summary of the literature, we see a pattern in interface research. First, communities or vendors propose an interface design for a platform. Then, security researchers establish studies to evaluate the security of such a design. After that, interfaces' vendors propose other services and policies that could interfere with the design of the proposed interfaces or their policies and permissions (e.g. interactions of app components interfaces across multi-user features in Android). The later stage has been studied less than when the interfaces were introduced. As addressed previously, numerous recently introduced services that have access to interfaces directly or indirectly have not been evaluated for their security (e.g. hardware APIs in browsers and remote attestation in mobile devices). These can be considered as multiple small research gaps. Considering the policies and permissions enforced by each platform, evaluating these services against the existing interfaces is a crucial problem that is difficult to solve automatically (e.g. via unit testings or fuzzing). Therefore, we established research to identify potential correlation issues in these services, considering their impact on existing interfaces. Our research aims to answer the following questions:

- To what extent do the security models of proposed interfaces hold with existing and newly introduced services, permissions and policies in each platform?
- What are the key components that should be considered when designing and integrating new interfaces in a platform?

The first question seeks to evaluate the security of interfaces, determine their attack surface, examine their design and propose mitigation to enhance the interfaces in the long term. The second question aims to identify the essential requirements for such an evaluation and determine the components (e.g. input screen, background process, etc.) that must be considered when designing a new interface.

1.3 Contributions

Vulnerability analysis of URI scheme handling in Android mobile Our first contribution is discussed in Chapter 2. We analyse local URI schemes handling in mobile. We found that local URI schemes (Javascript, Data and File) have received less attention from academics in the mobile platform. Therefore, we established a security analysis to both understand their implementation and differences compared to desktop browsers and determine if platform differences raise new security threats. As a result, we found three security issues in notable mobile browsers: Samsung Internet, Google Chrome, Microsoft Edge and Opera. First, we found a self-XSS attack in Chromium browsers that originated from improper sanitiszation of the JavaScript URI of the scheme via input method editor (IME) keyboards (*i.e.* software-based virtual keyboard). Secondly, we found an origin spoofing issue in Samsung Internet that affect the Data URI by abusing improper rendering of Data URI. Finally, we found a privilege escalation issue on Samsung's Android OS while analysing the File URI scheme in the Samsung Internet browser. The issue allows an arbitrary mobile application to access the device's internal storage without permissions, fully bypassing the dedicated Android storage permission in Android OS. The issue originated from a vulnerable Samsung Knox SDK called Sensitive Data Protection (SDP), which allows such access without prior authentication or authorisation.

Our work demonstrates that platform differences raise new security issues, as seen in the IME keyboard and Samsung Knox cases. Because these issues affect multiple input methods, output methods and specific SDKs, traditional security testing might not be sufficient to detect them.

We reported the issues to the affected vendors, yielding several Common Vulnerabilities and Exposures (CVE) described in Table 1.1. Additionally, this work was published as a workshop paper in MadWeb 2022, co-hosted with Network and Distributed System Security Symposium 2022 (NDSS) 2022 on Mars 2022 and awarded the best paper award.

> Abdulla Aldoseri and David Oswald. "insecure://Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers." In Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb). Part of The Network and Distributed System Security Symposium 2022 (NDSS). Mar. 2022. (`https://dx.doi.org/10.14722/madweb.2022.23003`) [3].

Issue	**Affected software**	**Reported at/to**	**References & CVE**
Self-cross-site scripting	Opera Mobile Browser	2020-11-25 (Opera)	CVE-2020-6159
Self-cross-site scripting	Chromium browsers	2020-12-01 (Google)	i-1154353
Improper storage permission handling in File URI	Samsung Internet Browser	2020-10-30 (Samsung)	CVE-2021-25348
Improper authorisation in SDP SDK	Samsung's Android OS	2021-02-09 (Samsung)	CVE-2021-25417
Origin Spoofing in Data URI	Samsung Internet Browser	2021-03-19 (Samsung)	CVE-2021-25419

Table 1.1: Summary of the reported issues discussed in Chapter 2

Vulnerability analysis of Hardware APIs As a second contribution, in Chapter 3, we performed the first security analysis of hardware APIs recently added as Web APIs. As a result, we identified two main security issues. The first issue is origins spoofing, which occurs due to improper implementation of the permission dialogue screen of hardware APIs. The second issue allows sharing a granted permission via File URI to any other served document within the same browser. The latter issue is caused because of improper design of the SOP of the File URI scheme in Chromium browsers. The issues affect all the Chromium-based browsers that support hardware APIs.

As a result, we conclude that integrating new interfaces requires testing them against existing policies. As discussed in Chapter 2, SOP has been applied in several browser implementations like schemes, web protocols and WebAPIs. However, we noticed that such a policy had yet to be thoroughly tested for hardware API, which results in such a threat.

We reported these issues to Google, yielding a CVE and an as yet unsolved issue described in Table 1.2. We kindly ask the readers to keep the second issue confidential until a patch is released from Google. This work has yet to be published and is still ongoing.

Issue	**Affected software**	**Reported at/to**	**References & CVE**
Origin spoofing in WebUSB	Chromium browsers	2021-12-15 (Google)	CVE-2022-0803
Shared hardware API permissions across File URI	Chromium browsers	2021-12-09 (Google)	Confidential - Not fixed yet *

Table 1.2: Summary of the reported issues discussed in Chapter 3

Analysis of app components in the mobile platform Thirdly in Chapter 4, after studying the interfaces of the browser platform, we address the mobile platform. We performed a systematic analysis of app components which represent interfaces for mobile applications. We examined the interaction between apps across multiple user profiles and studied the background restriction policy of Android OS. As a result, we found two significant issues. Firstly, we demonstrated an issue in the design of the multi-user profile feature that allows leaking data from and, in some cases, taking control of other Android userspaces, bypassing their dedicated lock screen by abusing shell user permissions through app components. Such an issue appeared in several adaptations of the multi-user profile feature, including Samsung's secure folder, Huawei's private space, and Xiaomi's second space. Secondly, we found that spyware can abuse app components to access sensors like the camera and the microphone in the background up to Android 10, bypassing the background restriction policy. As the issue's patch is only available in Android 11, we designed an open-source static analysis tool to detect such issues for numerous apps with complex codebases systematically. Our tool identified exposed components in 34 out of 5,783 apps with an average analysis runtime of 4.3 s per app (faster than other tools: AppSealer [198] and FIRMSCOPE [73]) and detected both known malware samples and unknown samples downloaded from the F-Droid repository.

Recalling previous contributions, we determined that introducing new interfaces that adopt existing policies and permissions could raise new threats, especially if the target platform has a different input method, output method, or a custom internal process. Here, we can see a reverse situation. The security of newly introduced services like the multi-user feature and background restriction policy has been violated due to existing interfaces and permissions. Even though those app components as interfaces were available long before these new services and policies were introduced, their interactions with services are generally not well understood.

We report these issues to the affected vendors, which yields the CVEs described in Table 1.3. Xiaomi's issue was reported by a third party prior to our report. This work was published as a conference paper at the European Symposium on Research in Computer Security 2022 (ESORICS) in September 2022.

> Aldoseri, Abdulla, David Oswald, and Robert Chiper. "A Tale of Four Gates." In European Symposium on Research in Computer Security 2022 (ESORICS), pp. 233-251. Springer, Cham, Sep. 2022. (`https://link.springer.com/chapter/10.1007/978-3-031-17146-8_12`) [4].

Issue	**Affected software**	**Reported at/to**	**References & CVE**
Privileges escalation across user profiles	Huawei's private space	2020-08-28 (Huawei)	CVE-2020-9119
Privileges escalation across user profiles	Samsung's Secure folder	2020-08-18 (Samsung)	CVE-2020-26606
Using camera and mic in the background	Android OS 10	2020-12-10 (Google)	Fixed in Android 11

Table 1.3: Summary of the reported issues discussed in Chapter 4

Security Analysis of remote attestation protocols in Mobile Fourthly, looking towards analysing interfaces of trusted execution environments, in Chapter 5, We address remote attestation protocols for mobile. These protocols use trusted hardware to ensure the integrity of mobile devices and, in some implementations, applications integrity. Interestingly, some of these protocols are not available publicly for testing. Therefore, we formally modelled these protocols using Tamarin Prover and verified their security properties in the symbolic model of cryptography. Our analysis identified two vulnerabilities. First, we found a relay attack against Samsung Knox V2 that allows a malicious app to masquerade as a simple app. Second, we determined an error in the recommended use case of Android key attestation that replays old —possibly out-of-date—attestation. Furthermore, as a case study, we employed these findings to tackle one of the most challenging problems in Android security, code protection. We proposed and formally modelled a code-protection protocol that uses a hardware root of trust to ensure source code protection for mobile apps.

Here, we determined that integrity-based remote attestation protocols require attesting the integrity of the device and the application they run, and attesting one of them is not sufficient. Additionally, our proposed case study shows the potential of these protocols to overcome complicated problems like safeguarding source code that runs in the normal world.

We reported the discovered issues to the affected vendors. Samsung confirmed the issue theoretically, but they asked for a proof of concept to issue a CVE, which we could not deploy due to the Samsung Knox licence. Google confirmed they would update their documentation in future releases. Further information about the issue is available in Table 1.4. This work was accepted as a conference paper at the 18th ACM ASIA Conference on Computer and Communications Security (2023) ASIA CCS in Janaury 2023.

> Abdulla Aldoseri, Tom Chothia, Jose Moreira-Sanchez and David Oswald. "Symbolic modelling of remote attestation protocols for device and app integrity on Android." In 18th ACM ASIA Conference on Computer and Communications Security (2023) ASIA CCS, . Association for Computing Machinery (ACM), Jan. 2023. DOI:10.1145/3579856.3582812 (`https://doi.org/10.1145/3579856.3582812`) [5].

Issue	**Affected software**	**Reported at/to**	**References & CVE**
Missing challenge phase in the documentation of the remote attestation protocol	Hardware key attestation	2021-11-09 (Google)	Confirmed to be fixed
Relay attack in the protocol	Knox remote attestation	2020-08-13 (Samsung)	Not fixed yet

Table 1.4: Summary of the reported issues discussed in Chapter 5

Assessing the Vulnerability of Enclave Shielding Runtimes Finally, as a fifth contribution in Chapter 6, we analysed the vulnerability space of TEE. We discovered a sanitisation vulnerability at the application level that leads to exploitable memory safety and side-channel vulnerability in a compiled Intel SGX enclave. We demonstrated that the current mitigation techniques, such as Intel's `Edger8r` failed to eliminate such an issue. Based on that, we deployed a proof of concept demonstrating an attack scenario that abuses the vulnerability to leak secret keys from an enclave.

Based on the literature, considerable research and industry efforts have been dedicated to developing TEE runtime libraries that offer a transparent shielding enclave application code from an adversarial environment. However, based on our result, we found that the shielding requirement is generally not well-understood in real-world TEE runtime implementations such as Intel SGX. This has been seen in our work as Intel's `Edger8r` failed to satisfy its purpose, making an enclave vulnerable. This work was published as a conference paper at the ACM SIGSAC Conference on Computer and Communications Security (CCS) in November 2019.

> Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes." In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS), Nov. 2019, pp. 1741-1758. 2019.(`https://dl.acm.org/doi/abs/10.1145/3319535.3363206`)[188].

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 to 6 presents our work as summarised previously in the contribution section. These chapters are based on accepted conference papers with the exception of Chapter 3. Each chapter is introduced by a preamble describing the chapter's scope, results and contributions.

The order of the chapters is based on the three discussed platforms: browser, mobile and TEE. In Chapter 2, we analyse local schemes in mobile browsers. Then, Chapter 3 discusses an analysis of hardware API in browsers. Subsequently, we move to the mobile platform in Chapter 4, analysing app components with consideration to services and policies that interact with them. Toward studying TEE, Chapter 5 presents formal modelling of hardware-based remote attestation protocols for mobile. Later, in Chapter 6, we shift to TEE. We study the interfaces of trusted applications, namely enclaves, in Intel SGX architecture. Finally, we conclude in Chapter 7, summarising our outcome, identifying limitations and proposing opportunities for future work.

Chapter 2

insecure://

Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers

The contents of this chapter were published as:

Abdulla Aldoseri and David Oswald. "insecure://Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers." In Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb). Part of The Network and Distributed System Security Symposium 2022 (NDSS). Mar. 2022. (`https://dx.doi.org/10.14722/madweb.2022.23003`) [3].

Preamble

This chapter discusses our initial study of browser interfaces. Because browsers operate on multiple platforms and expose a number of interfaces, we focused on specific categories of browsers: mobile browsers. We selected mobile browsers as our case study because they are the most privileged apps and have similar functionality that simplifies the analysis process (e.g. most browsers use Android storage permission to access the device storage). One of the primary input sources of mobile browsers is the address bar. It accepts URI schemes. These schemes instruct browsers to conduct specific actions depending on the requested scheme.

Many previous research focus on analysing Web URI schemes [72, 6] (e.g. HTTP and HTTPS) for desktop and mobile browsers. Less attention has been paid to local schemes (e.g. data: and file:), specifically for mobile browsers. In this work, we examined the implementation of such schemes in Android OS browsers, analysing the 15 popular mobile browsers. As a result, we discovered three vulnerability types that affect several major browsers (including Google Chrome, Opera and Samsung Internet).

First, we demonstrated a URI sanitisation issue that leads to an XSS attack via the JavaScript scheme. The issue is caused by missing sanitising input from an Android-specific keyboard (IME Keyboard). It affects Chromium browsers, including Chrome, Opera, Edge, and Vivaldi. Second, we found a display issue in Samsung Internet that allows abusing data URIs to impersonate origins and protocols, posing a threat in the context of phishing attacks. Finally, we discovered a privilege escalation issue in Samsung's Android OS, leading to thorough read-and-write access to the internal storage without user consent and bypassing the Android storage permission. While this issue was originally discovered in the file scheme of the Samsung browser, utilising a combination of static and dynamic analysis, we traced the problem back to an authorisation issue in Knox SDP SDK. We then showed that any app can abuse this SDK to obtain full access to the internal storage without appropriate permission on Samsung devices running Android 10. We responsibly disclosed the vulnerabilities presented in this chapter to the affected vendors, leading to four CVEs and security patches in Chrome, Opera and Samsung Internet browsers.

2.1 Introduction

Mobile browsers are among the most widely used apps on mobile devices. They feature various methods, known as URI schemes, to access the Internet (e.g. HTTPS) and device resources (e.g. file:) [178]. These schemes access sensitive user data like online accounts or users' local files. Therefore, it is essential to ensure their safety and security.

As addressed in Chapter 1, URI schemes can be classified into Web schemes and local schemes. Web schemes have been extensively scrutinised in the literature [72, 6], while local schemes received less attention. Nevertheless, some notable issues in local schemes have been reported for desktop browsers (e.g. phishing with top-level navigation via data URIs [109, 59] and Cross-Site Scripting (XSS) via encoding JavaScript URIs [112]).

In this chapter, we set out to analyse local URI schemes on mobile devices. Our initial assumption was that the specific usage context and the different OS characteristics compared to desktop browsers give rise to new vulnerability types specific to the mobile context.

2.1.1 Our contribution

In this chapter, we systematically analyse local browser schemes on mobile devices, focusing on JavaScript, data:, and file: schemes. We concentrated on these three schemes. However, we also analysed other schemes (e.g. about:) but did not find significant issues. We then describe several vulnerabilities affecting mobile browsers that we found. In summary, our main contributions are:

First, we discovered a self-XSS issue [77, 54, 107] in the JavaScript scheme. The issue allows executing JavaScript in the context of a loaded Web page, e.g. leading to session hijacking. The issue is caused by a lack of sanitisation of the clipboard of Android's IME keyboards. Most Chromium browsers are affected by this issue, including Google Chrome, Microsoft Edge, Opera, and Brave.

Second, we demonstrated how data URIs in the Samsung Internet browser can be used to impersonate websites, *i.e.* render content that seems hosted on a genuine origin. Samsung Internet's behaviour causes this problem by displaying the last URI characters in the address bar, making the data: scheme prefix invisible.

Third, by examining the file scheme, we uncovered a privilege escalation issue in Samsung's Android variant that allows an arbitrary app to access the device's internal storage without user consent, bypassing the Android storage permission. We traced the issue to SDP, a Samsung Knox module. As these issues affect Chromium-based browsers, we found that Firefox browsers (e.g. Firefox and Firefox Focus) are not vulnerable to these issues when we examined their local schemes URIs.

2.1.2 Responsible disclosure

The vulnerabilities described in this chapter have been responsibly disclosed through the respective channels. Their details are available in Table 1.1. The JavaScript scheme issue described in Section 2.4.1 was reported to Google and Opera on December 1, 2020, and November 25, 2020, respectively. They can be tracked via CVE-2020-6159 for Opera and #1154353 in the Chromium bug tracker. The data scheme issue in the Samsung Internet browser described in Section 2.4.2 was reported on March 19, 2021, and can be tracked via CVE-2021-25419. Finally, the file scheme vulnerability described in Section 2.4.3 was reported on October 30, 2020. It resulted in two CVEs: CVE-2021-25348 for bypassing the permission security check in the browser and CVE-2021-25417 for the underlying privilege escalation issue in Samsung's Android OS variant.

2.1.3 Outline

The remainder of this chapter is structured as follows: In section 2.2, we provide a background on browser schemes and related work regarding their security. Based on this, we defined the scope of our work and chose specific schemes for our analysis. section 2.3 discusses the selected research methodology. We discuss our analysis in section 2.4 and present the issues discovered in widely-used browsers. Subsequently, we assess the impact of the discovered issues and propose mitigations in section 2.5. Finally, we conclude in section 2.6, discussing limitations and opportunities for future work.

2.2 Background and Related Work

This section provides an overview of browser schemes and reviews their security aspects. We then discuss related work in terms of security issues and common weaknesses.

2.2.1 URIs

User agents (browsers) accept URIs from users and execute an action based on the URI. RFC3986 defines a URI to have the following structure [50]:

```
URI         = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
hier-part   =  "//" authority [:port] /path  |  "/" path
```

From the browser perspective, the scheme refers to the used protocol in a request (e.g. HTTPS and local schemes like file and JavaScript). The "authority" includes the host for requests to remote servers.

Browsers adopt the SOP as a security model. SOP provides isolation between origins (authorities), *i.e.* prevents different origins from accessing each other unless they have the same origin or explicitly authorised each other, for example via Cross-Origin Resource Sharing (CORS) [131]. SOP prevents network adversaries from compromising retrieved responses from one origin or carrying out actions on behalf of the user from compromised origins (e.g. via JavaScript) [45].

2.2.2 Browser schemes

Browsers support different schemes for remote communication (e.g. HTTP, HTTPS, FTP) and local schemes (e.g. JavaScript, data, file). Unlike remote schemes, these local schemes solely operate on the client machine and do not require internet access. As they operate locally, they may expose local computer privileges, which can cause security issues. Therefore, this chapter focuses on understanding and analysing local schemes from a security perspective.

The JavaScript scheme is implemented in web browser applications to execute custom JavaScript from a URI [99]. The scheme uses the following URI format:

```
javascript:<script>
```

where `<script>` is JavaScript code. The scheme has two sequential operations: source text retrieval and in-context evaluation. The first operation retrieves the source text included in the `<sc ript>` part of the URI and applies necessary decoding and characters replacements operations to it. Then, the in-context evaluation operation evaluates the generated text. A typical JavaScript scheme example for embedding a script as a hyperlink in an HTML document is as follows [99]:

```
<a href="javascript:doSomething()">click</a>
```

In this example, when the user clicks on the hyperlink, the browser executes the `doSometing()` function in the context of the currently loaded origin. The embedded JavaScript inherit the current origin [135]. The JavaScript scheme can also be invoked (like any URI) from the browser address bar.

The data scheme renders binary data as-is and allows the inclusion of external data [127]. An according URI has the following form:

```
data:[<mediatype>][;base64],<data>
```

In this case, `<mediatype>` is a media type specification for the represented data (e.g. txt or png). `base64` indicates if the data is Base64-encoded, otherwise, ASCII encoding is assumed and `<data>` is the payload data itself. The following is an example of rendering a PNG image using the data scheme [127]:

```
<img src="data:image/png;base64,aGVsbGl..." />
```

The maximum length of URIs in browsers limits the size of the `<data>` part in data URIs. Similarly to JavaScript URIs, data URIs can be used from the browser's address bar.

The file scheme allows access to the files and directories of the local host machine (*i.e.* access of local files of a client machine only). Due to the SOP, remote hosts cannot query the scheme to access local files [178]. A file URI is structured as follows:

```
file://<host>/<path>
```

Where `<host>` is the mount point on the host machine (e.g. drive name or `/sdcard` on mobile devices), and `<path>` is the path to the requested file. Each file accessed by a file URI is assigned an implementation-defined value as on origin. (In some user agents, it uses a unique use Globally Unique IDentifiers (GUID) for each URI.) This mechanism prevents files from accessing each other's contents through the SOP policy [178, 45].

2.2.3 Related work

This section gives an overview of related work regarding reported attacks on browsers using URI schemes.

Vulnerabilities in the JavaScript scheme: Improper handling of JavaScript URIs enables attacks, including Cross-Site Request Forgery (CSRF) and XSS. CSRF refers to an attack where the adversary causes the browser to initiate a request to a particular server without user consent, while XSS is an attack that allows an adversary to inject malicious scripts into the visited (legitimate) website [136]. In both attacks, a victim can be affected by the attacks unintentionally by visiting a vulnerable website or using vulnerable services (e.g. plugins, browsers) [96]. XSS is classified into several subtypes: reflected XSS, stored XSS, DOM-based XSS, and self-XSS [145, 7, 54, 96]. Among them, the one related to the JavaScript scheme is self-XSS: a social engineering attack in which victims are tricked into executing scripts that compromise their Web accounts or leak their data [107, 77, 54]. Figure 2.1 illustrates an example of a self-XSS attack. The attack aims to obtain a victim's session cookie. It allows an adversary to use the victim's session to access the victim's information or perform actions on the victim's behalf on the target website. First, the adversary sets up a server to accept requests and instructs the victim to use a malicious JavaScript URI ①. The victim accesses a benign target website ②. Then, they copy-paste the malicious URI into the browser address bar ③. Based on the script, the adversary can forward session data and cookies to its server ④. Adversaries rely on obfuscation, minification and encoding of JavaScript code to increase the willingness of the victim to execute the scripts [54]. Cao et al. measured the effectiveness of various obfuscation techniques for JavaScript URIs. They found that obfuscated JavaScript URIs achieve 10% more execution by participants (38.4%) compared to regular JavaScript URIs (29.4%) [54].

In Android, Terada demonstrated an issue with the JavaScript scheme in Google Chrome and Android browsers. Because the browsers did not sanitise the JavaScript scheme URIs in incoming Android intents, an adversary can perform XSS attacks by sending multiple intents to the browsers to first request a target domain and then to obtain personal data (e.g. session cookies) via the JavaScript scheme [180].

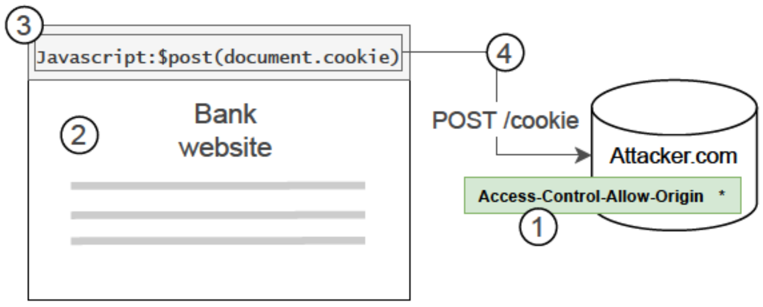

③
Javascript:$post(document.cookie)
④
②
Bank website
POST /cookie
Attacker.com
Access-Control-Allow-Origin *
①

Figure 2.1: Scenario showing a self-XSS attack to steal a victim's session cookie. The adversary sets up a server to accept requests from any origin ①. The victim logs into his bank account ② and copy-pastes a malicious self-XSS URI into the address bar ③. The session cookie is sent to the adversary server via an XML HTTP request ④.

Phishing with the data scheme: In contrast to the JavaScript scheme, the data scheme focuses on data representation. A common issue relates to the user clicking e.g. a hyperlink and not realising that this leads to a data URI instead of a Web URI [132]. The adversary can then render a phishing page or provide a malicious executable from that data URI. Therefore, most browsers, including Firefox and Chromium, block top-level navigation to data URIs [109, 59]. Regarding sanitisation, Chromium did not sanitise SVG contents in the pasted text. This issue allows embedding a JavaScript payload in SVG content in a data URI, leading to JavaScript execution in the target document [58].

Address bar spoofing using data URIs refers to displaying data: content that appears to be hosted on a legitimate origin, e.g. showing an attacker-controlled form that seems hosted on `example.com` and thus enabling phishing attacks [67]. Nishimura reported a related bug in Firefox on Android [142]: data URI were persistently shown in the address bar regardless of Web navigation. This issue occurred when a data URI link opened from a stored shortcut or a bookmark intent sent from an Android application. It hides the true origin of the current content. Baloch showed a different attack against Opera, Safari and UC, where data URI content is loaded while the address bar shows a different origin. The issue is due to the browser preserving a target URI in the address bar when requested over an arbitrary port repeatedly [38, 43].

Privilege escalation with file URIs: Mobile browsers expose an interface to accepting browsing requests by other apps or Web pages. Wu and Chang illustrated a weakness in this feature because of not appropriately sanitising "intents": if an intent requests a file URI while auto-download is enabled, browsers might download their private data (e.g. session cookies) to the SD card, allowing malicious apps on the device with storage permission to read this information [192]. Terada reported an issue in Opera with similar consequences: Web pages on Opera can access any private activities in the Android Opera browser because of improper filtering. Utilising this issue allows file URI to access the browser's cookies and render them as an HTML document. A malicious website can set a malicious JavaScript code in a cookie to execute it when rendered as an HTML allowing it to steal the rest of the cookies [180].

Improper implementation of SOP in browsers can equally introduce security issues: as demonstrated by Wu and Chang, 63 Android browsers improperly implemented SOP, resulting in leaking sensitive data from local files (e.g. HTML) [192]. In Chrome, Barth, Jackson, Reis, et al. showed that remotely hosted XML files could retrieve contents of local files due to the same root issue [46].

Browsers analysis and detection tools: Conventional automated unit tests are often insufficient to detect the issues mentioned above. As a result, the research community has developed according to testing and debugging tools. Google developed a remote debugger for troubleshooting Chromium-based mobile browsers and Android Webviews. The debugger provides a developer console for inspecting components and a log monitor [90]. Wu and Chang implemented an automated test tool to check for their file URI issues. The tool interacts with the browser-under-test and the adversary's app using Android Debug Bridge (ADB) and can find certain issues automatically [192].

2.3 Research methodology

The work in this chapter required a high-level and low-level understanding of browser policies and the respective Android infrastructure from a security perspective. We thus used two methods to aid us in structuring and organising our work:

Documentation review: We systematically reviewed official documentation on Android browser infrastructure to comprehend relevant security policies and threat models.

Reverse engineering: Because the source code for Android mobile browsers is often not accessible, we statically reverse-engineered relevant components if required. We combined this technique with dynamic analysis to test and confirm discovered vulnerabilities.

We focused our analysis on three browser schemes (JavaScript, data, file), because these have been the most common sources of vulnerabilities in the past and expose the largest attack surface compared to other limited schemes (e.g. about).

Browser	JavaScript	Data	File
Google Chrome *C	Clip-trim	✓	
Samsung Internet *C	Query	✓	✓
Opera *C	Clip-trim	✓	✓
Brave *C	Clip-trim	✓	✓
Microsoft Edge *C	Clip-trim	✓	✓
Vivaldi *C	Clip-trim	✓	✓
FireFox	Query		✓
FireFox focus	Query		✓
DuckDuckGo	Query	✓	Crash
Mint	Not supported	✓	✓*2
Mi Browser	Null origin *1	✓	✓*2
MX6	Not supported		✓*2
US browser	Query	✓	✓*2
Phoenix browser	Query	✓	✓*2
Dolphin	Null origin	✓	✓*2

Table 2.1: Summary of how browsers handle JavaScript, file and data schemes. (*C) indicates Chromium browsers. (*1) indicates that JavaScript is executed only in the null origin where no website is loaded. (*2) indicates support of the file scheme but without implementing an index page that lists the internal storage files.

2.4 Case studies

We focused our analysis on three browser schemes (JavaScript, data, file) because these have been the most common sources of vulnerabilities in the past and because they expose the largest attack surface compared to functionally limited schemes (e.g. about:). We analysed the implementation of each considered scheme in the 15 popular mobile browsers (based on Google Play downloads). As an initial step, we reviewed how each browser handles JavaScript, data, and file URIs. The results are shown in Table 2.1 and are further elaborated in the following sections.

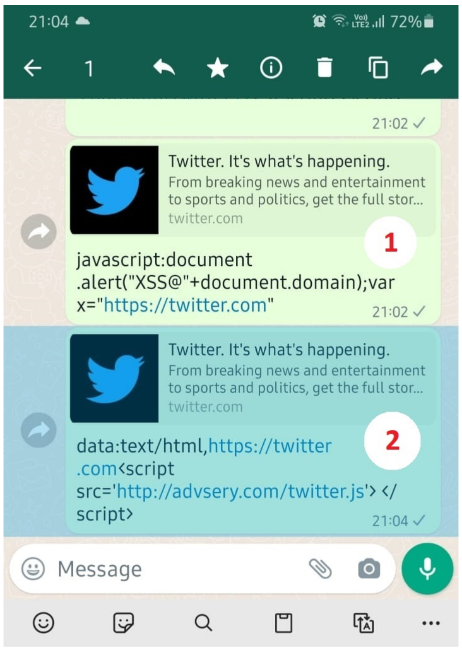
2.4.1 Self-XSS using the JavaScript scheme

The JavaScript scheme, as described in section 2.2, allows JavaScript code to be executed in the context of the currently loaded origin. This functionality enables numerous JavaScript-based threats, particularly self-XSS, which we investigated in the following.

Threat model: We considered the threat model of Facebook, Cao et al., in which an adversary tricks a victim into performing a "self-XSS" by sending them a malicious URI, e.g. through a messenger app or social network app. The attack does not require the victim to install apps and proceeds as follows:

1. The adversary sends a URI containing the JavaScript scheme with malicious code to the victim and tricks them into copy-pasting the URI.
2. The victim copies the URI using the clipboard and pastes it into a vulnerable web browser.
3. The script is executed, potentially in the context of the current origin.

Self-XSS requires social engineering: copy-pasting URIs is common, especially in emails where it is often requested if clicking the URI does not work. The adversary might include the target website's original URI to display its original logo in a social media post or direct message. As can be seen in Figure 2.2, WhatsApp v2.21.23 (latest version at the time of writing) mis-recognises the URI of Twitter within a malicious JavaScript scheme and displays the website information, potentially misguiding users. Cao et al. have shown that 30% of participants in a self-XSS experiment were deceived by such an attack [54].


21:04
72%
1
21:02
Twitter. It's what's happening.
From breaking news and entertainment to sports and politics, get the full stor...
twitter.com
1

```
javascript:document
.alert("XSS@"+document.domain);var
x="https://twitter.com"
```

21:02
Twitter. It's what's happening.
From breaking news and entertainment to sports and politics, get the full stor...
twitter.com
2

```
data:text/html,https://twitter
.com<script
src='http://advsery.com/twitter.js'></
script>
```

Figure 2.2: An example of sharing a JavaScript URI ①, and data URI ② on WhatsApp. The app recognises only the fake HTTPS URI at the end and shows website info, tricking users into believing the URI to be genuine.

JavaScript scheme implementations in mobile browsers: To understand how our selected browsers handle and sanitise JavaScript URIs, we analysed the behaviour by supplying several test URIs to each browser. The results are summarised in the JavaScript column in Table 2.1. Concretely, we found that browsers exhibited one (or a combination) of the following handling practices:

Search query: A JavaScript URI is treated as a search query for the browser's default search engine but is never executed.

Clipboard trimming: The browser intercepts URI paste events from the clipboard and strips the scheme from the pasted text. This technique prevents attacks where a victim is lured into copying a URI into the address bar. However, manually typing a JavaScript URI using the keyboard is still allowed.

Only on null origin: JavaScript URI is executed only on a null origin, *i.e.* when no website loaded in the browser.

Not supported: JavaScript URIs are not supported at all. This was only the case for the Mint and MX6 browsers.

As can be seen in Table 2.1, Most Chromium-based browsers adopt the clipboard trimming approach. The browser detects paste events and trims the scheme if required. Nevertheless, the Android Clipboard Manager does not provide an event listener for intercepting paste events [18]. Therefore, it appears that Chromium employs a custom solution based on the address bar to sanitise pasted URIs, leading us to a potential security problem:

Potential issue #1: The Android Clipboard Manager does not provide a mechanism to listen to paste events. Chromium-based browsers use a custom solution to sanitise pasted URI, which might contain security issues in the implementation.

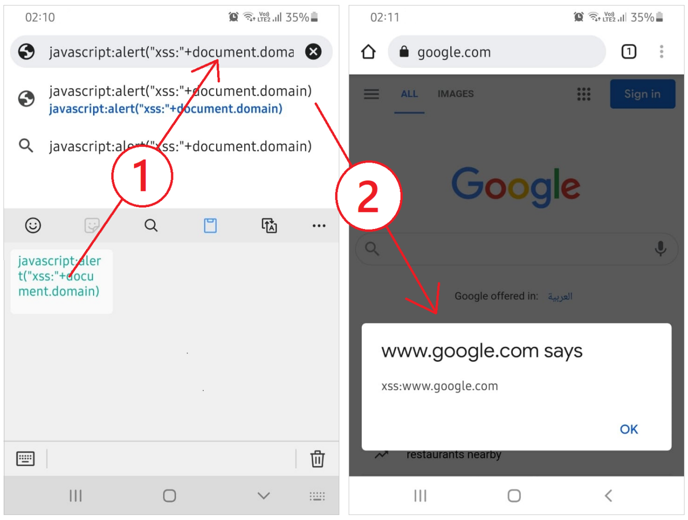
Intercepting clipboard events: We focused on Chromium browsers because they are the only browsers in our set to implement clipboard trimming (apart from Samsung Internet, cf. Table 2.1). Rather than analysing each Chromium browser individually, we examined the Chromium source code as the likely base [61]. From this, we found that the JavaScript scheme is handled as follows:

1. The URI address bar is a custom field that inherits the functionality of `AutoCompleteEditView`.
2. The `onTextContextMenuItem` method is overridden to listen for context menu events (e.g. cut, copy, and paste).
3. The so-called Omnibox API sanitises the pasted text, trimming the JavaScript scheme.

We reviewed the Omnibox unit test cases and found that the sanitisation correctly handles even obfuscated scheme expressions, where special characters or spaces are used, e.g. `java\x0d\x0ascript:alert(0)` and `java script: alert(0)`. However, we noticed that paste interception only works for paste events from the context menu. Therefore, trimming will *not* be applied if the paste events originate from a different source.

Bypass clipboard trimming: Since Android 7, Android supports IME keyboards [24]. IME keyboards are custom keyboards that extend the functionality of the default keyboard e.g. with emojis and pictures. Several IME keyboards are shipped with a custom clipboard implementation that is not context menu-based. Respective keyboards are pre-installed as the primary keyboard by major manufacturers, e.g. Gboard for Google devices and the Samsung keyboard for Samsung devices.

As suspected, we found that using the paste operation of IME keyboards does not trigger the JavaScript sanitisation of Chromium-based browsers. This issue can thus be exploited to enable self-XSS attacks against Chrome, Opera, Brave, Edge and Vivaldi if the user pastes from the IME keyboard clipboard as shown in Figure 2.3.


02:10

javascript:alert("xss:"+document.doma

javascript:alert("xss:"+document.domain)
javascript:alert("xss:"+document.domain)

javascript:alert("xss:"+document.domain)

javascript:alert("xss:"+document.domain)

02:11

google.com

www.google.com says

xss:www.google.com

OK

Figure 2.3: An example of self-XSS against Google Chrome with an IME keyboard: pasting a JavaScript URI from the keyboard bypasses sanitisation (1). Thus, navigating to the URI causes the JavaScript code to run (2).

Issue #1: Self-XSS attacks against Chromium browsers are possible if URIs are pasted from IME keyboards.

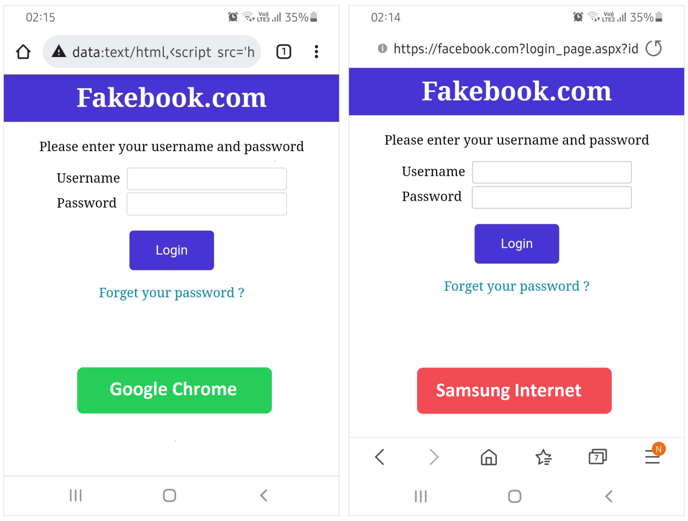
IME keyboards are widely pre-installed by major manufacturers, including Google and Samsung: the Gboard IME keyboard has over 1 billion installs, according to Google Play [88]. The issue can be utilised for session hijacking (e.g. stealing session cookies) or general JavaScript code execution in the browser (e.g. to submit forms on behalf of the user without consent and knowledge). In the affected browsers, JavaScript code can access the current session data, including the document components, because the scheme inherits the currently loaded origin. The browser's SOP can be circumvented by configuring the adversary's back-end servers to accept requests from any origin using CORS.

Disclosure: We reported this issue to Opera and, because it affects all Chromium browsers, subsequently also to the Chromium team. Both vendors confirmed the issue. The Opera issue can be tracked via CVE-2020-6159, while the Chromium issue in the patching process. While certain practices from Table 2.1 could be used for mitigation (e.g. "Search query" or "Null origin"), the Chromium team intends to repair the issue while maintaining support for typing JavaScript URIs. Applying this solution is complicated because IME keyboards do not provide APIs for listening to clipboard events. Thus, we propose a solution that meets the above criteria in section 2.5.

2.4.2 Impersonation of website origins using the data scheme

Data URIs allow rendering data included in the URI string in the Web browser. Consequently, data URIs have been used to display "fake" origins in browsers as described in section 2.2. Thus, we opted to study this problem in the context of our mobile browsers. As evident from Table 2.1, only Firefox and Firefox Focus lack support for data URIs.

While all browsers restrict top-level navigation for data URIs, we noticed an issue in the behaviour of the address bar in the Samsung browser: it displays the last characters of the data URI, which truncates the data scheme and most of the actual data from lengthy URIs. Consequently, we found it possible to craft a data URI that renders a Web page that seems to be hosted on a legitimate origin, e.g. a fake login phishing page. As can be seen in Figure 2.4 (right), such a custom data URI renders a form that appears to be hosted on a legitimate domain (`fakebook.com` in our example) and served via HTTPS. We tested the crafted data URI on two different Samsung


02:15
data:text/html,<script src='h
Fakebook.com
Please enter your username and password
Username
Password
Login
Forget your password ?
Google Chrome

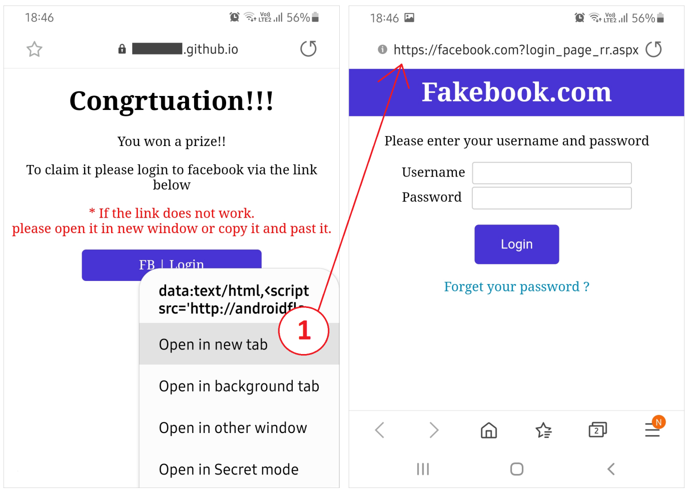
02:14
https://facebook.com?login_page.aspx?id
Fakebook.com
Please enter your username and password
Username
Password
Login
Forget your password ?
Samsung Internet

Figure 2.4: These screenshots show loading a crafted data URI in Google Chrome (left) and Samsung browser (right). The data scheme is visible on Google Chrome. In contrast, the Samsung browser displays the last characters of the URI, leading the user to think that the content is hosted on a legitimate origin and served via HTTPS.

devices with different display resolutions, Samsung A75 and Samsung S20 Ultra, and confirmed that the URI produces the same result regardless of the screen size. In contrast, all other browsers in our set correctly displayed the beginning of the URI, including the data scheme, as can be seen for Chrome in Figure 2.4 (left).

Issue #2: Displaying only the end of a data URI in the Samsung browser address bar allows an adversary to show a fake origin of the rendered data.

Threat model: Two threat models can be considered for this issue. First, since Chromium browsers support opening a data URI via the new tab option, an adversary can set up a phishing data URI link asking the user to open it with a new tab if it is not working, resulting in origin spoofing as illustrated in Figure 2.5. Alternatively, the same threat model as the JavaScript scheme (Section 2.4.1) can be considered. Technically, a victim is tricked into copying a data URI and pasting it into their browser. Our considered browsers do not apply any trimming for data URI, so pasting via the context menu or an IME keyboard leads to identical results. Like the JavaScript scheme, the adversary can prepend the URI of a legitimate origin to the crafted data URI to display the target website's logo, as seen in Figure 2.2.



18:46

[redacted].github.io

Congrtuation!!!

You won a prize!!

To claim it please login to facebook via the link below

* If the link does not work.
please open it in new window or copy it and past it.

FB | Login

data:text/html,<script src='http://androidf

Open in new tab

Open in background tab

Open in other window

Open in Secret mode

18:46

https://facebook.com?login_page_rr.aspx

Fakebook.com

Please enter your username and password

Username

Password

Login

Forget your password ?

Figure 2.5: An example of opening a data URI via new tab option (1). Samsung browser displays only the last characters leading to origin spoofing.

Conventional phishing defences, such as showing the origin in the address bar or black-listing phishing sites [46] are ineffective against this issue because the rendered data is embedded in the URI rather than being fetched from a remote server.

We reported the issue to Samsung, and it can be tracked via CVE-2021-25419. We discuss mitigations in section 2.5.

2.4.3 Permission system bypass on Samsung Android OS through the file scheme

The file scheme in mobile browsers gives access to the content of the internal storage. Thus browser apps should require Android storage permission. Similar to the previous case studies, we evaluated the implementation of the file scheme in the selected browsers, cf. Table 2.1. Most browsers support the scheme, with the notable exception of Chrome, and give access to mobile internal storage. Many popular browsers, including Samsung browser, Opera, Brave, Edge, Vivaldi and Firefox (Focus), also display an HTML index page that lists internal storage files on the device. Because of SOP, accessing local content through the file scheme from a local or an external HTML file is not permitted in all supported browsers.

Starting from a fresh browser install with no permissions, we tested the handling of file URIs and the respective permission requests in the considered browsers. Our testing revealed an unexpected behaviour in the Samsung browser that was not present in the other browsers: navigating to `file:///sdcard` results in a prompt to grant or deny storage permissions. However, if the user selects "deny and don't ask again", *i.e.* permanently declines storage permission, and then repeats the request to the same URI, the internal file list appears and access to the internal storage is given.

Potential issue #3: The Samsung browser can access the device's internal storage without storage permission. Apparently, the app uses a proprietary non-standard mechanism that bypasses the Android permission system.

Conditions	Case I	Case II	Case III (Exploit)
App can request storage permissions	T	F	F
App does not have storage permissions	T	F	T
Request does not access Samsung browser storage	T	T	T
Access to internal storage	Denied	Granted	Granted

Table 2.2: The table demonstrates cases of file URI restrictions in the Samsung browser. Case I: user denies storage permission; case II: user grants storage permission; case III: user denies storage permission and disables further prompts.

Handling of file URIs in the Samsung browser: To understand this surprising behaviour and apparent bypass of the Android permission system, we reverse-engineered the handling of file URIs in the Samsung browser. We found that a request to access the file scheme is only *denied* if the following three conditions are all satisfied:

- The app can request storage permissions.
- The app does not have storage permissions.
- The request does not access the Samsung browser's private files located at `/data/data/com.sec.android.app.sbrowser/sbrowser`.

At first glance, this logic (cf. Table 2.2) appears to prohibit requests for local file access without storage permission. The first and second conditions seem identical, *i.e.* the first condition seems to imply the second and vice versa. More specifically, if storage permission is not granted, it is possible to request it (case I), and once storage permission is granted, it is no longer possible to request it (case II).

However, the relatively new "deny and don't ask again" option in the Android permission system allows for the circumvention of these rules: it does not grant the app storage permission, nor does it allow subsequent requests. Thus, when the Samsung browser does not have storage permission and can no longer request storage permission, access to the internal storage will bypass the check and proceed with the access (case III).

Root cause analysis: Still, it remains unclear how the Samsung browser -even though it theoretically allows access- can gain access to the internal storage without holding the actual permission. Finding the root cause is challenging because many Android components are involved in the Android OS. It was, however, clear that the issue was caused by the interplay of Samsung-specific modifications, as the problem did not occur with other browsers (and because the Samsung browser only runs on Samsung devices). We, therefore, examined the involved components step-by-step.

System-level analysis: At the Android system level, we identified two possible mechanisms that could cause the observed permission system bypass:

Privileged permissions: Android can give read/write access to specific apps via so-called allowlists [31].

Signature-based permissions: Apps signed with the vendor key are granted system signature permissions.

We analysed Samsung's allowlist and found no rule granting Samsung browser storage permission, eliminating the first option. Further, we found that the Samsung browser is not a system app and installed on the data partition. We repacked and self-signed the app with a debug key, invalidating any signature permission that Android might grant to the app. We found that the issue still exists in the self-signed browser, thus excluding the possibility of signature-based permissions as the root cause.

App-level analysis: As we excluded a system-level root cause, we considered the possibility that specific app characteristics caused the behaviour. First, we determined whether the issue was related to Chromium: we patched other Chromium browsers and eliminated the storage permission request. We, however, found that (as expected) access is denied.

We thus statically analysed the Samsung browser app to find any components that access the internal storage, both within the app and through external components, e.g. third-party apps, services or SDKs. Because the code base of the Samsung browser is large (278 MB after decompression), we concentrated on the file scheme handling and any involved app components (activities, services, content providers and broadcast receivers). From this analysis, we found that file URIs are handled as follows:

1. The requested URI is retrieved from the address bar.
2. It is sanitised against the allowed schemes.
3. The URI is passed to the Terrace browser engine to process and issue the request.

Inside the Terrace browser engine: Terrace is the browser engine used by Samsung. It is a native C library (compiled for ARM) with a large code base of 70 MB for 64-bit and 50 MB for 32-bit processors. Terrace is based on Blink, a rendering engine part of the Chromium project [62]. Unfortunately, the size of the Terrace code base and the lack of debug symbols makes a comprehensive static analysis infeasible. Nevertheless, we noticed several interesting strings related to the file scheme. At this stage, we decided to switch to dynamic analysis. We used Frida [81] to intercept and override potentially interesting methods within the browser engine. Frida's interception abilities allowed us to inspect parameters and return values of relevant function calls. Figure 2.6 illustrates

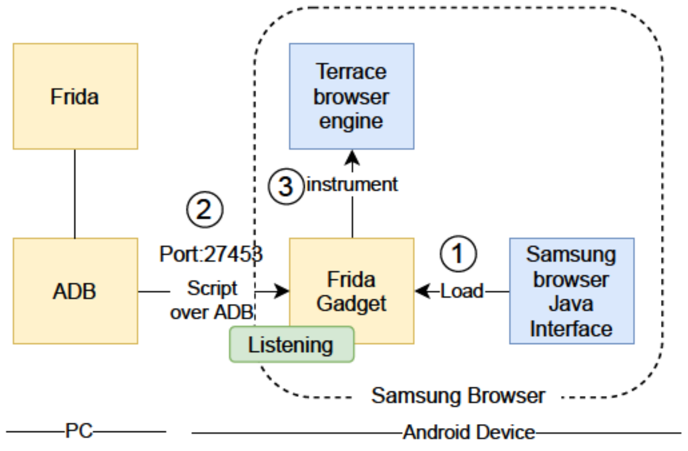


Figure 2.6: Our dynamic analysis setup for the Samsung browser to inspect the Terrace browser engine. The original browser components are blue, and our test components are yellow. The analysis starts by loading a Frida gadget to open a communication port ①. The Frida module communicates with the gadget through `adb` to transmit inspection scripts ②. Finally, the Frida gadget uses the supplied scripts to perform the analysis ③.

our dynamic analysis setup using Frida. We inject a Frida gadget into the Samsung browser and instruct it to start when the app starts ①. The browser is self-signed with our own debug key. We used `adb` to connect the smartphone to a PC. We use the Frida module from the PC to communicate with the Frida gadget in the app over `adb` ②. The Frida gadget starts in listen mode, ready to accept scripts to inspect components in the target application ③.

The setup does not require rooting the device, as we noticed that the Samsung browser relies on Samsung Knox. Rooting the device would remove the Knox modules, possibly causing misbehaviour or crashes of the browser [160].

We identified and intercepted the Java method responsible for loading URIs and witnessed how file URIs are forwarded to the Terrace engine. To overcome Terrace's lack of debug symbols, rather than inspecting potentially relevant Terrace functions, we hooked native file access operations like `r ead()` and `open()` syscalls. When accessing the internal storage using the file scheme, we indeed observed Terrace invoking `open()` and `opendir()` to open files and directories, respectively. At this stage, we confirmed that the Samsung browser, as a single process, has access to the internal storage and is not reliant on app components or external applications.

Finding the root cause: By determining that syscalls access of the internal storage always succeed in the Samsung browser (independent of Android permissions), we injected a compiled module into the app to print its effective permission. From this, we found that the Samsung browser is in the group `sdcard_rw`, finally explaining why access to the internal storage is granted. However, it remained unclear *how* the app obtains membership of the `sdcard_rw` group.

To understand that, we repacked the app numerous times, removing one component and checking if group membership was removed each time. We removed native C libraries, app components, and classes and libraries in the app's source code. We found that neither native C libraries, including the Terrace engine nor app components, affect access to internal storage.

However, we identified a specific metadata entry called "SDP" in the Android manifest of the application that grants the `sdcard_rw` permission to the Samsung browser.

```
<meta-data android:name="sdp"  android:value="enabled"/>
```

SDP is a Samsung Knox SDK component that can protect sensitive data within an app [162, 161].

Exploiting SDP to bypass storage permissions: Any app can request the inclusion of SDP and its SDK through a simple metadata setting, and both are used in the Samsung browser. To verify that, as a proof-of-concept, we cloned the code base of the Samsung browser app but changed its package name and self-signed the binary. This process is complex for an app with a large code base like the Samsung browser because it requires changes in several places (classes, directories and metadata) and because the authority names of app components must be altered as well. Considering this, we successfully created an independent app with a different name that uses SDP and has access to the internal storage. The app does not interfere with and can be installed alongside the original Samsung browser. Because it has a different package name, it can be published to Google Play.

Alternatively, one can also include the SDP configuration in any other app. Thus, an adversary who can trick the user into installing a malicious app (e.g. disguised as a game or similar) can exploit this issue to exfiltrate the internal storage even though the user did not grant the Android storage permission. We reported these issues to Samsung on October 30, 2020, and February 9, 2021. They assigned two CVEs: CVE-2021-25348 for bypassing the security check in the browser and CVE-2021-25417 for the underlying privilege escalation issue in Samsung's Android variant. At the time of reporting, both issues affected Android 10. Samsung patched the issues on December 1, 2020, and June 8, 2021, respectively, with the Samsung browser version v13.0.1.64.

Issue #3: Regardless of Android permission, the Samsung Knox SDP gives any app access to the internal storage without user approval.

2.5 Discussion and Mitigations

Our case studies show that, although the general topic has been extensively studied, handling local URI schemes in mobile browsers still suffers from substantial oversights. Thus, we conclude that currently used test methods are insufficient to detect complex issues, especially those caused by interaction with external components, e.g. the Knox SDK or IME keyboards. Similarly, more principal mitigations are required for the issues pointed out in this chapter, especially those related to JavaScript and data URIs.

Mitigating self-XSS via the JavaScript scheme: As shown in Section 2.4.1, hooking paste events is not sufficient to capture all sources of input into the address bar (other than the user typing): IME keyboards are third-party components and do not expose APIs for event listening (e.g. paste events), nor does Android provide a generic interface. The issue cannot be resolved through changes to the IME keyboard code, as the Android guidelines state that IME keyboards are not responsible for sanitising their output [19]. Furthermore, new input methods might be introduced in the future that could also bypass the existing address bar sanitisation logic. Google thus sought a solution that fundamentally prevents future similar issues but still allows manual typing of URIs with local schemes (like JavaScript).

We thus propose a generic multi-character handler for Android that solves the issue. Our solution is based on text insertion behaviour. User-typed text is inserted character-by-character, whereas pasted text (from context menu or IME keyboard clipboard) is inserted as a block. Android keyboard uses the `commitText()` [1] method to send a text to a designated input field. If the text is a single character, it is sent as it is. Otherwise, the keyboard sends a special key event along with the text in a block [2]. Therefore, to intercept paste events, we can override the address bar's `onTextChange()` method and inspect the number of inserted characters[3]. If a multi-character insertion is detected, this indicates a paste event (from any source) that can be blocked.

We implemented this approach as a proof-of-concept and confirmed that it intercepts paste events from the context menu and the IME keyboards. Alternately, instead of overriding the address bar's `onTextChange()` method, it is possible to attach a `TextWatcher` to the address bar and override its `onTextChange()` method to achieve the same results. Google's security team considered our proposed solution, adopted it and deployed a fix using `TextWatcher` [60].

Standard approach to avoid phishing with data URIs: While most browsers adopt the correct approach to display the beginning of a URI rather than its end, we found that Samsung browsers did not follow this approach. Its prefix data URI scheme for long URIs is hidden as shown in Figure 2.4, enabling an adversary to create phishing URIs that appear to be hosted on legitimate origins. To resolve this issue for data URIs (and other schemes), we propose that the community defines a standard approach to correctly and securely display URIs in the browser. For example, such a standard could mandate showing the start of the data URI as implemented in most browsers (and also how Samsung patched the issue after our report).

Preventing permission system bypass on Samsung Android: The issue presented in section 2.4.3 cannot be entirely prevented at the browser level, as the underlying reason is rooted in Samsung's modifications to Android. Because SDP does not authenticate apps that utilise it, any app may publicly subscribe to it and obtain access to internal storage. Therefore, an OS update or an update to the Knox SDK is required to patch the vulnerability. While Samsung informed us that the issue had been resolved for Android 10, we did not receive information on their mitigation strategy. However, recently we noticed that SDP is deprecated in the latest Knox SDK patch v3.7.

Chromium browsers and non-Chromium browsers: We found that the Firefox browser was not vulnerable to any of the discovered issues. It offers better protection than Chromium-based browsers for the local URI scheme. This improvement in protection could be because Firefox uses a different browser engine named Gecko [133].

Limitations: Our work is based on manually inspecting the most common local schemes in the most popular mobile browsers. Automating (parts of) our analysis and extending it to other browsers and schemes is an interesting research area we leave for future work. Such automation is challenging because the founded issues relate to complex interactions between UI components (e.g. text fields) and data entry methods (e.g. IME keyboards) or OS-specific configuration options.

2.6 Conclusion

This chapter demonstrated several security issues in local URI schemes, affecting major mobile browsers, including Google Chrome, Edge, Opera and the Samsung browser. We have shown that a lack of proper sanitisation of JavaScript URIs can lead to self-XSS attacks, while data URIs can be

[1] `https://android.googlesource.com/platform/frameworks/base/+/56a2301/core/java/android/view/inputmethod/BaseInputConnection.java#194`

[2] `https://android.googlesource.com/platform/frameworks/base/+/56a2301/core/java/android/view/inputmethod/BaseInputConnection.java#529`

[3] `https://source.chromium.org/chromium/chromium/src/+/main:chrome/browser/ui/android/omnibox/java/src/org/chromium/chrome/browser/omnibox/AutocompleteEditText.java;l=189;drc=385121b715b153153f7626baf728cbd588250ff7`

abused for spoofing origins in phishing attacks. Finally, an issue in file URIs led us to discover a deeper design flaw in Samsung's Android, giving an arbitrary app access to the internal storage without user consent and bypassing the dedicated Android storage permission. Our results highlight that, even though the overall attack surface is well-understood, testing methods and tools that (semi)automatically detect URI handling issues in mobile browsers are still lacking and motivate future work in this direction.

Chapter 3

insecure:// Vol.2

Vulnerability Analysis of Hardware API in Internet Browsers

This chapter represents a short piece of research conducted at the end of my PhD study. It is still ongoing research, which I aim to complete.

Preamble

This chapter takes a more profound step into analysing browser interfaces. While the previous chapter focused on the schemes' implementation in mobile browsers, here, we focus on Web APIs (or so-called HTML 5 APIs). Most are Web interfaces which access several computer resources like cameras and microphones. Interestingly, some are extended to access external devices like USB and Bluetooth. The latter subset of APIs is known as Hardware APIs. They are recently introduced, and nearly no academic studies have examined them from a security perceptive. Therefore, based on our understanding of the browser's threat model, we chose to target this research area to investigate and evaluate the security model of these APIs considering their permissions and policies.

This work represents the first security analysis for hardware APIs. We examined their permissions, supported schemes and policies. As a result, we discovered two security vulnerabilities that affect Chromium browsers (including the most popular browser Google Chrome). Additionally, we demonstrated possible real-world scenarios of abusing both issues. We responsibly disclosed the vulnerabilities presented in this chapter to Google leading to a fix for the first issue. The second issue is still in the patching phase.

3.1 Introduction

Web APIs enable browsers to access various computer resources (e.g. camera and microphone). Such APIs are used for various Web applications (e.g. video conferencing). Recently, Hardware APIs have been added as a subset of Web APIs. These APIs extend the browsers' accessibility to external computer resources like USB and Bluetooth devices. While academic research mainly focuses on analysing the design of Web APIs and assessing the impact of their misuse, no studies have been conducted to study hardware APIs from a security perspective. Therefore, in this work, we aim to target this research area. Our work presents the first security analysis of hardware APIs aiming to evaluate the security of their permissions and policies.

3.1.1 Our contribution

In this chapter, we analyse the security of hardware APIs, focusing on their permissions and policies. Our analysis finds two vulnerabilities affecting hardware APIs. In summary, our main contributions are:

Firstly, we discovered an origin spoofing issue in the hardware APIs permission request. The issue allows adversaries to display target origins (not owned by adversaries) to request access to hardware devices. The issue is caused by a mis-implementation of the displayed origin name in the permission request that displays the last committed origin.

Secondly, we demonstrate an implementation issue in the SOP for Hardware APIs. The issue allows documents loaded over the file URI scheme to obtain any hardware APIs permission granted to already loaded documents over the same scheme. The issue is caused by treating all documents loaded over file URI as having the same origin.

Finally, we illustrated examples of abusing both issues in real-world scenarios. Among them is a demonstration of the first issue that affects Vysor.io, one of the most popular Android mirror websites, with 1M downloads according to Google play [189]. Additionally, Both issues affect the most popular Chromium browsers (including Google Chrome, Opera and Edge).

3.1.2 Responsible disclosure

The vulnerabilities described in this chapter have been responsibly disclosed to Google. The origin spoofing issue described in section 3.5.1 was reported to Google and Opera on December 15, 2021. It can be tracked via CVE-2022-0803 and `crbug.com/1280233` through the Chromium bug tracker. The file URI scheme issue described in section 3.5.2 was reported to Google on December 9, 2021. Currently, it is considered embargoed until Google release a patch. Therefore, we asked the readers not to disclose this issue until a patch is released.

3.1.3 Outline

The remainder of this chapter is structured as follows: In section 3.2, we provide a background on Web APIs and their subset hardware APIs to have a better understanding of their permissions and policies. Then, in section 3.3, we discuss related work considering the security model of Web APIs defined in the background section and identify the research gap. In section 3.4, we define the considered threat model for our analysis. Then, we describe our analysis in section 3.5. Both section 3.5.1 and section 3.5.2 discuss the reported issues, their causes and demonstration examples. Finally, we conclude in section 3.7.

3.2 Background

In this section, we illustrate the infrastructure of Chromium hardware APIs.

3.2.1 Web browsers and Web APIs

Through the years, Web browsers have become feature-rich software to access the Internet. They have access to several computer resources (e.g. cameras and storage) through browser-implemented Web APIs to ease online operations on users [137]. Some Web APIs have permission-based restrictions as they access dangerous resources (e.g. cameras and microphones). Abusing these resources could lead to compromising users' privacy. In terms of design, browsers manage access to computer resources via their subsystems. Among them is the browser engine. The browser engine shapes the backbone of Web browsers. It is a high-level interface for the rendering engine and operates various browser actions such as load origins, reloads, forward and back [94]. Notable browser engines are Blink for Chromium browsers and Geeko for Mozilla. Web APIs (e.g. access USB) are interfaces implemented based on the browser engine. For instance, hardware APIs (e.g. WebUSB, WebHID and WebBluetooth) are built as independent blink modules.

3.2.2 Permissions and polices

In browses, access to computer resources is safeguarded by permissions and policies. The Web APIs permissions system is designed similarly to the Android permissions system [28]. However, instead of granting permissions per app in Android, permissions are granted per origin (where an origin is a composite of a protocol (e.g. HTTPS, file://), a hostname (e.g. domain) and a port number). Upon requesting permission, users are required to either accept or reject the request interactivity. User interactions are mandatory because most permissions are considered dangerous (*i.e.* may expose sensitive data). Then, the granted permission will be limited to only the requested origin (*i.e.* SOP) [190].

SOP is one of the notable policies to secure Internet browsing and resource access in browsers. It ensures isolation between origins. It disallows loaded origin from accessing other origins unless they are the same origin or have explicitly authorised each other via CORS [131, 45]. For instance, granting a USB permission to an origin should not permit other origins to use the USB unless they are permitted [191]. Notably, policies are implemented differently based on the used scheme. Remote origins are distinguished by their domain, scheme and port number(e.g. http://example.com and https://example.com are considered two different origins). While local origins like the File scheme are assigned an implementation-defined value (e.g. GUID in some user agents) per document (e.g. file://myfilename.txt) [178, 45].

3.2.3 Hardware API

Recently, Web browsers (specifically Chromium-based) have integrated several Hardware APIs as a part of Web APIs to access generic hardware resources (e.g. USB and Bluetooth). These APIs, namely WebUSB, WebBluetooth and WebHID allow origins to access hardware resources (USB, Bluetooth and HID, respectively) to ease hardware interactions. WebHID allows access to input devices like game controllers. It has properties to access devices' information, control their connection, and allow communication using reports[134].

They are built as blink modules in Chromium browsers [63]. These blink modules implement the renderer process details and bindings for their designated hardware to interact with it. Figure 3.1 illustrates the interaction between the WebUSB blink module and a physical USB device. First, the WebUSB module communicates with USB service through two main interfaces: the WebUSB Services Mojo interface and the public Mojo interface. The first interface is a parallel interface to UsbDeviceManager aimed to handle USB operations like permission checking and device chooser showing in the browser before passing the control to the public interface. Then, these interfaces can communicate with the USB service and the device service to communicate with an actual physical device. Currently, these APIs are supported only in Chromium browsers (e.g. Chrome, Edge, Opera).

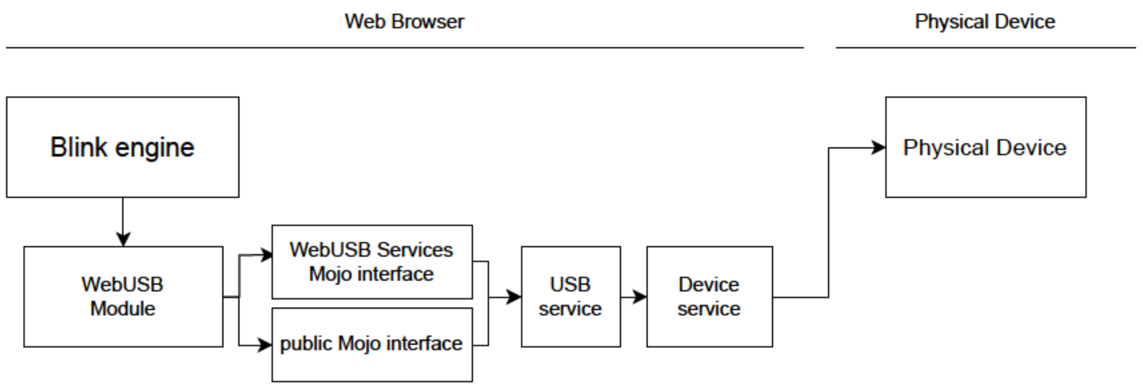


Figure 3.1: The architecture of interaction of WebUSB module with physical device

For simplicity, Hardware APIs offer a hardware filter (*i.e.* permission descriptor). It allows developers to supply metadata (e.g. vendor ID and device ID) to filter requested devices when integrated (e.g. a Web page can choose only to allow specific hardware devices to connect to it). Additionally, hardware APIs only function on top-level frames, except WebUSB. WebUSB works in embedded frames (e.g. iframe) but requires specifying the USB policy feature in its embedded frame tag. As some hardware devices have several communication interfaces at the hardware level, WebUSB API implements interface claiming operation for such a case. Each interface can be claimed only by a single origin at a time. This mechanism provides isolation at the browser process level.

3.3 Related work

Based on section 3.2, we determined that the security design of Web APIs' is based on its permission and policies. Therefore, we reviewed these aspects from the literature to understand the testing techniques and identify potential security issues.

Permission issues in Web APIs: Starting with permissions, Kim et al. demonstrated privacy issues in geolocation API in Android browsers. 14 out of 60 browsers provided GPS location for websites without requesting permission from the end users. Additionally, they highlight several privacy issues in the design of GPS API. For instance, the GPS API does not present a specific purpose for using the GPS when requesting it from end users. Because the GPS request shows a common generic message, it is difficult for end users to determine if it means getting end users' locations once or tracking them continuously [110]. Tian et al. analysed the security of screen sharing API. They found that screen sharing API allows the creation of a cross-origin feedback loop. This issue allows websites from different origins to monitor all the victim's screen content without their consent. As in impact, the vulnerability can be exploited to initiate CSRF-iframe attacks and leak user data (assuming the target website does not enable the X-frame option) [181].

SOP and Web APIs: Secondly, considering the Web APIs' policy, namely SOP, it is noticeable that it has received less attention than other browser components. For instance, Zalweski discussed SOP for cookies, local storage, Flash, XMLHttpRequest, Java, Silverlight, and Gears [197]. Zheng et al. have analysed the SOP for HTTP cookies, and Schwenk et al. analysed SOP on a subset of Web objects, specifically DOM objects. It is unclear why SOP is not well-studied for Web APIs. It seems an interesting area to investigate, but it is important to assess the impact of exploiting Web APIs before establishing research about them.

Assessing the impact of exploiting/abusing Web APIs: To assess the impact of exploiting Web APIs, we focus on reviewing literature that measures the usage of Web APIs and estimates potential threats caused by them. Marcantoni et al. conducted a large-scale study on mobile sensor-based Web API. They analysed their attacks and identified their risks on 183,571 domains [126]. Their study shows that 2.89% (5,313) websites use at least a sensor API like geolocation and device orientation. Considering this ratio, the author highlights that sensor data can be abused for various attacks, including user tracking and physical activity inference. Full-screen API that displays the page in full screen can be abused for phishing attacks. Adversaries can render fake UI sent over hyperlinks to deceive users [2]. Finally, Aldoseri et al. demonstrate an issue in the multi-user feature in Android OS to access users' profiles in the same device bypassing their dedicated lock screen. The issue allows accessing victims' internal storage and installing apps. (We further discuss it in chapter 4). Utilising the WebUSB API allows a remote domain to connect to a device to access all the device's userspaces without the main user awareness and authorisation[4]. Web APIs are permission-restricted for how dangerous they are. As explained, numerous attacks can be initiated by either abusing permitted permissions or exploiting a design flaw in the API.

What about Hardware APIs? As hardware APIs were a recent addition to Web APIs, we noticed a lack of attention to them. Most previous studies mainly discussed the integration of hardware APIs for software engineering purposes. The APIs were used as off-the-shelf and proposed for several projects: designing a multifactor authentication system with WebUSB [82], WebUSB as an IoT extension for smartphone [122] and integrating web-Bluetooth in a perishable shipment tracker system to reduce hardware cost [83]. Surprisingly, we could not find reported security threats that target hardware API while reviewing reported issues in the Chromium bug website [64]. Therefore, we aimed to fill this research gap by analysing hardware APIs considering their permissions and policies. We based our research on the addressed testing techniques from previous studies to explore the security model of hardware APIs, identify their potential threats and assess their issues.

3.4 Methodology and adversary model

We mainly focus on source code review of the hardware APIs source code (as it is a module of the Chromium project) and manual testing with different hardware devices. Therefore, we consider an adversary model that requires a Chromium browser. No additional software, rooting kit or admin privileges are required.

Hardware APIs	HTTP/HTTPS	Data://	File://	About://	iFrame DOM
WebUSB	✓	✗	✓	✗	✓*
WebHID/WebSerial WebBluetooth	✓	✗	✓	✗	✗

Table 3.1: Summary of Hardware API support for browser schemes and protocols in Chromium browsers. (*) Require enabling the USB policy feature tag in the embedded frame.

3.5 Hardware API analysis

We started our analysis by exploring and testing hardware APIs against browsers supported schemes and embedded elements. Our analysis is highlighted in Table 3.1. As can be seen, only the file URI scheme supports hardware APIs among local schemes. Furthermore, all hardware APIs do not work in iframes except for WebUSB. Here, we limited the scope of testing and explored the supported schemes to understand their applied policies and permissions better. Our analysis highlighted two reported issues, explained as follows.

3.5.1 Origin Spoofing in secure permission UI delegation

Following the support of iframe in WebUSB, we noticed that WebUSB permission request UI shows the origin of the site that requests access to the device (*i.e.* The origin of the embedded frame is displayed if a request is initiated from the frame). Surprisingly, upon selecting 'connect' option to permit the embedded site the permission, the embedded site within the frame does not obtain that permission. Effectively, the host of the embedded frame obtains the permission regardless of its origin (*i.e.* whether it matches the embedded frame origin or not). This design flaw creates a spoofing issue, allowing a malicious adversary to embed target websites and claim WebUSB permissions by displaying the origins of target websites in the permission request for end users. Even worse, Since iFrame is an HTML element, it is feasible to shrink its frame size and, with some CSS, to display only the request permission button of a target site.

Figure 3.2 shows a demonstration of the spoofing issue. We demonstrated this issue on Vysor, one of the most popular Android mirror websites. As can be seen, on the left image, the adversary site (buhadod.github.io) has an embedded iframe for a target website (app.Vysor.io). The permission request UI will be displayed when the connect button is clicked. It will show the name of the target origin (child) instead of the adversary origin (the host and the parent). Upon granting the permission, the host website will only receive the permission effectively.

After reporting the issue to Google, they concluded that it resulted from retrieving the last committed origin regardless of the frame (*i.e.* iframe or main frame).

3.5.2 Unauthorised shared access of hardware devices via hardware APIs over File: URI schemes

Secondly, we focused on analysing SOP for the supported scheme. For HTTP/HTTPS, the Hardware APIs can distinguish between domains and subdomains; each is considered a different origin. Since the communication between the browser and the devices occurs locally (via Front-end JavaScript), network adversaries should not shape a threat to remote protocols like HTTP unless the messages are sent over the network.

For the file URI scheme, first, we tested the support of Web APIs. The scheme fully supports hardware APIs functionality. Interestingly, such support enables hardware manufacturers to use HTML documents to run hardware devices instead of old-fashioned local applications shipped in a CD or USB. Additionally, the hardware APIs work regardless of the file type. For instance, JavaScript-supported files (e.g. HTML, XHTML and SVG) can fully use hardware APIs.

As addressed in section 3.2, file URI origins are defined differently. It is assigned an implementation-defined value as an origin to enforce SOP [178, 45]. Based on this, we evaluated how Chromium enforces SOP for hardware API. Surprisingly, files opened with file URI were treated as having the same origin. Technically, granting permission to a document loaded over file scheme access to hardware APIs means that all new and running documents loaded over file scheme will have access to the granted permission. More importantly, JavaScript-supported downloadable documents can

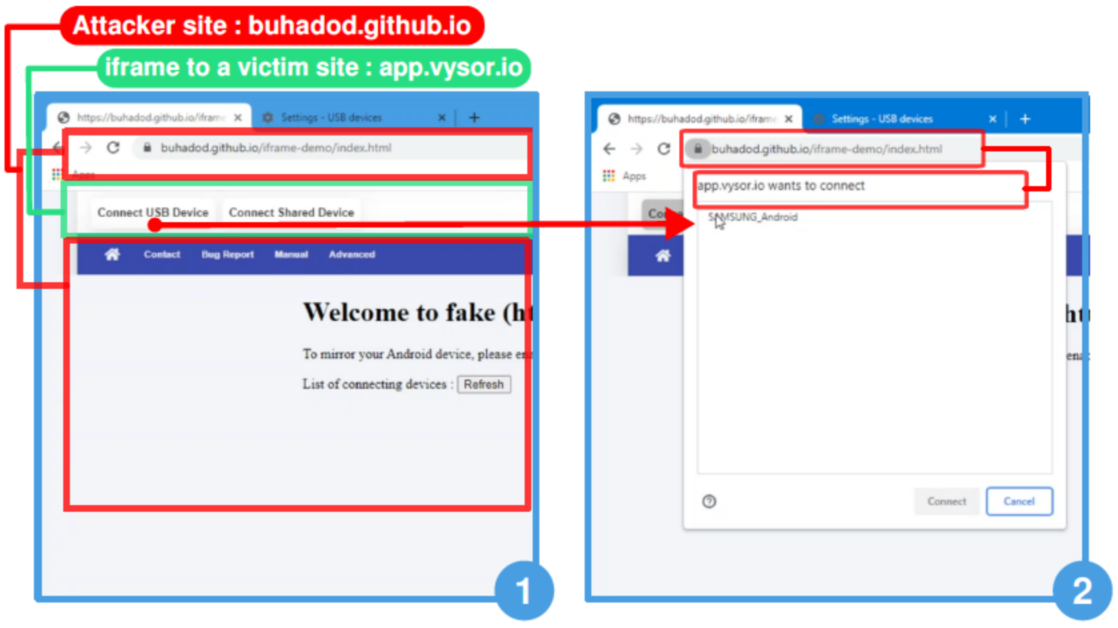

Attacker site : buhadod.github.io
iframe to a victim site : app.vysor.io
buhadod.github.io/iframe-demo/index.html
Connect USB Device
Connect Shared Device
Welcome to fake (ht
To mirror your Android device, please en
List of connecting devices : Refresh
app.vysor.io wants to connect
SAMSUNG_Android
Connect
Cancel
1
2

Figure 3.2: A demonstration of the spoofing issue in Web USB using iFrame. The adversary site (buhadod.github.io highlighted in red) embedded a victim site in an Iframe (app.vysor.io highlighted in green). Clicking on the 'Connect USB device' button will display the permission request screen. The screen shows the domain of the target site. Only the adversary site (The parent site) will obtain the permission upon granting the permission.

also have the same access. For instance, an adversary can abuse the popular SVG files (*i.e.* vector images) that are commonly used by shipping them with a malicious JavaScript code to access victims' hardware devices.

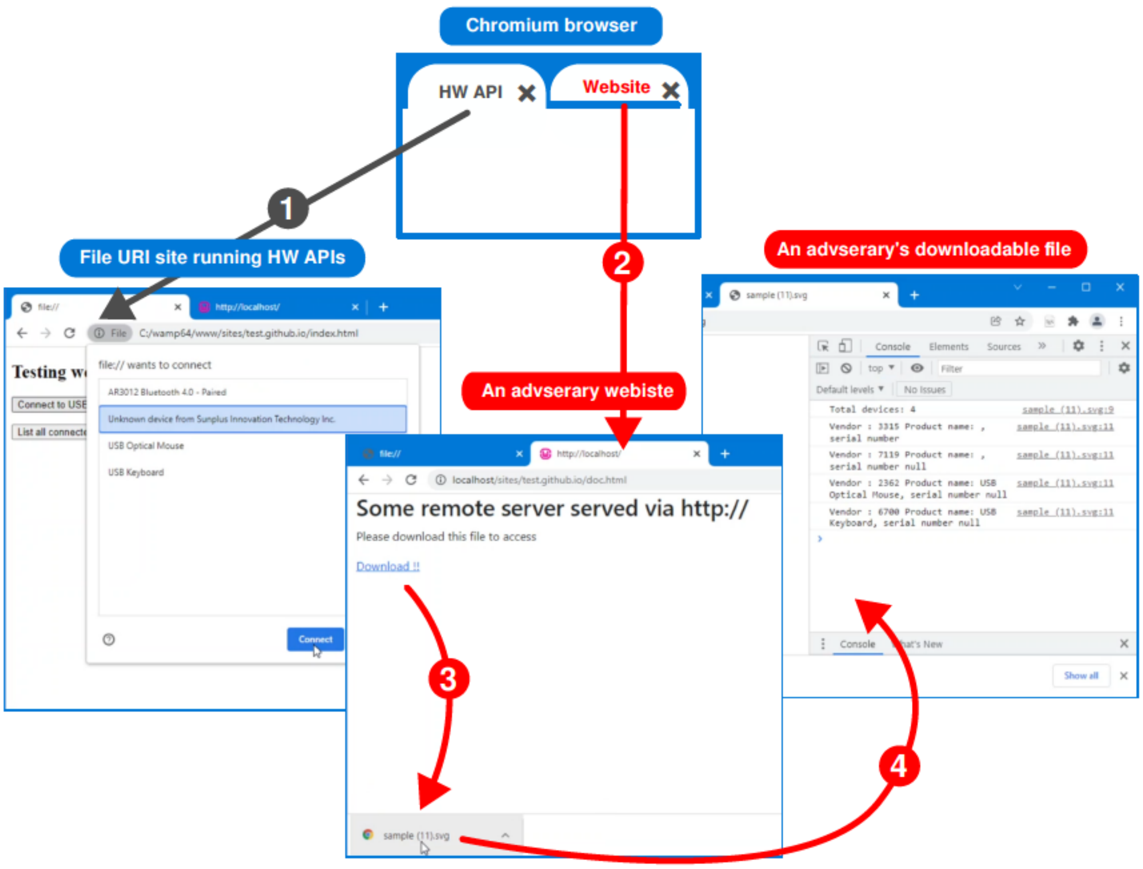
Figure 3.3 shows a potential scenario that abuses this issue. A victim possibly runs a document using file Scheme URI that already has permission to access some devices by hardware API (1). Then, in a different tab, the victim visits a site that asks to download a JavaScript-supported file (possibly a picture or an icon) (2). The victim opens the file after it finishes downloading using the same browser (3). Opening the file will execute the hidden JavaScript code within the downloaded file to gain access to the devices using previously granted permissions (4).

Even worse, Chromium browsers mis-implement listing functionality for file URI schemes to determine granted permissions and revoke them. Victims may be unaware of previously granted permissions and unable to revoke them easily without closing the browser. Also, instructing users always to close browsers if hardware APIs are used via file URI documents is not a proper solution, especially for less technically able users.

3.6 Discussion and limitations

The reported issues in this chapter abuse the mis-implementation of the design of permission UI of Hardware APIs and SOP for the file URI scheme. Crucially, the impact of these issues varies and depends on the target hardware device. Some hardware devices might be limited and do not shape a threat like a mouse or a keyboard, as in most cases they do not expose input interfaces. Alternatively, other hardware devices like Android phones can result in nearly full access to the device functionality, including access to the internal storage, installing apps and granting Android permission to apps silently. For instance, exploiting our previously reported issues that will be discussed in Chapter 4 results in accessing the secure private profiles (or userspaces depending on the implementation) remotely over WebUSB, bypassing their dedicated lock screen protection [4] (See Chapter 4).

We reported both of these issues to Google as they affect Chromium browsers. Firstly, for the origin spoofing issue addressed in section 3.5.1, Google assigned it a moderate severity and built a patch that displays the main frame (*i.e.* the host) origin instead of the target origin. The


Chromium browser
HW API
Website
1
File URI site running HW APIs
2
An advsery's downloadable file
An advsery webiste
Testing w
Connect to USB
List all connect
file:// wants to connect
AR3012 Bluetooth 4.0 - Paired
Unknown device from Sunplus Innovation Technology Inc.
USB Optical Mouse
USB Keyboard
Connect
Some remote server served via http://
Please download this file to access
Download !!
3
4
sample (11).svg
Console
Elements
Sources

```
Total devices: 4                                  sample (11).svg:9
Vendor : 3315 Product name: ,                     sample (11).svg:11
serial number
Vendor : 7119 Product name: ,                     sample (11).svg:11
serial number null
Vendor : 2362 Product name: USB                   sample (11).svg:11
Optical Mouse, serial number null
Vendor : 6700 Product name: USB                   sample (11).svg:11
Keyboard, serial number null
```

Figure 3.3: A scenario demonstrates the abuse of miss-implementation of SOP for hardware APIs in Chromium browsers. The victim runs a document via file URI and permits it to access a device via hardware APIs (1). Then, The victim downloads an adversary's JavaScript-supported file (e.g. HTML, XHTML and SVG)(2-3). Finally, upon opening the document, the document will be opened via file URI and will have access to previously granted permission for the selected devices (4).

currently implemented Webapps via WebUSB do not need a patch as the permission UI patch will explicitly state the origin of the requester and specify its granted permissions. Web developers and hardware manufacturers can use X-Frame-Options in their websites to prevent third-party sites from embedding their websites [138] similar to Google's Android flash tool over WebUSB [11]. Specifying a vendor-id of the requested device is essential to prevent the impact on end-user devices when using hardware APIs. It will limit the accessible devices for adversaries and avoid connecting to the wrong device.

Secondly, the unauthorised access issue by abusing the miss-implementation of SOP for the file URI scheme may require substantial changes on chrome to resolve it. Upon reporting the issue, the Google team referred to a similar issue that affects cookies and local storage and results in the same consequences when using the file URI scheme [129]. We noted in our analysis that such an issue does not affect the Firefox browser as it potentially implements the SOP for File scheme correctly. We believe such an issue will remain open and unfixed for a long time since the cookie and storage issue has remained open since 2013. Therefore, we advise end-users to close browsers after using hardware APIs over the file URI scheme to revoke granted permissions. Web developers and hardware manufacturers that aim to use Web APIs should publish their front-end pages that use hardware APIs online over HTTPS and avoid relying on the file URI scheme as much as possible.

Finally, we noticed that hardware APIs lack a data inspection tool to monitor traffic data between a device and a remote host. To analyse these APIs and inspect their traffic, testers and developers need to use tools like Wireshark for USB traffic and a physical Bluetooth sniffer for Bluetooth traffic. Additionally, macOS requires disabling OS security features for USB debugging at a system level. Overall these tools are limited. They only show the traffic "on the wire" (not per-origin as Chromium implemented), they may, for example, require additional configuration to decrypt transmitted data and, importantly, they cannot show the origin URI for each request. Therefore, we proposed to the Google DevTools team a feature that helps inspect hardware APIs data. Our idea is summarised by simply adding a debug tab (part of the F12 developer tools) to log hardware APIs traffic into the developer console (e.g. showing packets, their host, and to which devices they are sent). Information to be displayed could include: device name, device ID, claimed interface, connected hosts authorised to use the device(s), and messages sent from/to devices and hosts. Regarding privacy, adopting the same policy as the current network logging tab is necessary by only logging data when such a feature is enabled and activated. Keyloggers should not cause a problem because hardware APIs cannot access input devices (e.g. keyboard and mouse) [93]. The proposal can be viewed from here [1].

3.7 Conclusion

In this chapter, we demonstrated two hardware APIs vulnerabilities caused by the miss-implementation of their permission display and policy. We showed that the WebUSB API permission request interface could be spoofed to display other origin names. In terms of policy, we found that the SOP is not properly implemented for the file URI scheme for hardware APIs, and this causes loaded documents via the file URI to inherit all granted hardware APIs permissions for any previously loaded document within the same browser session. We conclude that hardware APIs require further research to evaluate their permissions and policies.

[1] `https://groups.google.com/g/google-chrome-developer-tools/c/FhaYMeyckv0`

Chapter 4

A Tale of Four Gates

Privilege Escalation and Permission Bypasses on Android through App Components

The contents of this chapter were published as:

> Aldoseri, Abdulla, David Oswald, and Robert Chiper. "A Tale of Four Gates." In European Symposium on Research in Computer Security 2022 (ESORICS), pp. 233-251. Springer, Cham, Sep. 2022. (`https://link.springer.com/chapter/10.1007/978-3-031-17146-8_12`) [4].

Preamble.

The purpose of this chapter is to aid better understanding of mobile apps' capabilities and functionality. Therefore, this chapter moves from browsers' interfaces to android app components: Activity, Services, Broadcast receivers and Content providers. Android apps use app components to interact and exchange data with each other.

Previous research has shown that app components can cause application-level vulnerabilities, such as data leakage across apps. Alternatively, apps can (intentionally or accidentally) expose their permissions (e.g. for camera and microphone) to other apps that lack these privileges. This situation creates a confused deputy issue, where an unprivileged app accesses unauthorised resources through a highly privileged app. Technically, a high-privilege app exposes its app components, which use these permissions, to the less-privileged app. While previous research mainly focused on these issues, less attention has been paid to how app components can affect the security and privacy guarantees of Android OS.

In this chapter, we demonstrate two vulnerabilities affecting recent Android versions. First, we show how app components can be abused to leak data and, in some cases, take complete control of other Android users' data, bypassing the dedicated lock screen. We demonstrate the impact of this vulnerability on major Android vendors (Samsung, Huawei, Google and Xiaomi). Secondly, we found that spyware can abuse app components to access sensors like the camera and the microphone in the background up to Android 10, bypassing mitigations specifically designed to prevent this behaviour. Using a two-app setup, we found that app components can be invoked stealthily to, for example, periodically take pictures and audio recordings in the background.

Finally, to prevent the rising confused deputy issues, we present Four Gates Inspector, our open-source static analysis tool, to systematically detect these issues for many apps with complex codebases. Our tool not only detects permission-based issues but can detect usage of classes that do not rely on permissions (e.g. scoped storage in Android 11). Our tool successfully identified exposed components in 34 out of 5,783 apps with an average analysis runtime of 4.3 s per app and detected both known malware samples and unknown samples downloaded from the F-Droid repository. We responsibly disclosed all vulnerabilities presented in this chapter to the affected vendors, leading to several CVE records and a currently unresolved high-severity issue in Android 10 and earlier.

4.1 Introduction

App sandboxing and isolation on widely deployed mobile operating systems like Android have brought various security benefits. However, due to the need to exchange data across apps and to access system resources like sensors (e.g. camera and microphone), Android provides several externally accessible entry points into apps beyond the user interface. The four main entry points (Activities, Services, Broadcast receivers and Content providers) are called app components. Over the last few years, app components have received considerable attention from security researchers [98, 73, 200]. Numerous vulnerabilities have been found that led to, among others, privileges escalation and side-channel leakage of user data [201, 73]. However, less attention has been paid to how these components interact with Android-wide restrictions (e.g. the separation of userspaces and the permission system). In this chapter, we address this issue and show how exposed app components can be used in unintended ways to break Android Multi-user feature, misuse permissions held by other apps, and conceive a construction that allows accessing the camera and microphone in the background on recent Android versions. Subsequently, we introduce Four Gates Inspector, an open-source static analysis tool that can aid developers in detecting and preventing specific issues discovered in this chapter.

4.1.1 Contributions

In this chapter, we systematically analyse exposed app components across system restrictions and multiple userspaces on the same device. Based on our analysis, we discover several vulnerabilities impacting user privacy and security. We first show that users with the `INTERACT_ACROSS_USERS` or `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permissions can invoke app components belonging to other userspaces. As the `adb shell` user has this permission by default, an attacker with either physical or remote access via WebUSB can misuse those interfaces to install apps into another userspace, grant arbitrary permissions and then exfiltrate the data through app components. Notably, the `shell` user does not have read/write access to other userspaces through standard means (e.g. using filesystem commands like `ls`, `cd`). The attack bypasses the dedicated lock screen of the target userspace.

Secondly, we show an issue that allows adversaries to construct stealthy spyware that can access sensors like the camera and microphone in the background via app components. This issue is performed by installing two apps: one exposes access to the sensors through app components, and a second (unprivileged) app repeatedly invokes this functionality in the background. The issue bypasses countermeasures against background spyware for recent Android versions up to Android 10, thus affecting all Android devices in use at the time of reporting (Sep 2020) [175].

To mitigate the issues presented in this chapter, we propose Four Gates Inspector, a static analysis tool to detect the usage of a given class or API (e.g. camera or microphone) by tracing the invocation of methods of each app component using graph-based analysis. Additionally, the tool aids in detecting confused deputy issues in exposed app components [97]. Our main contributions are:

- We performed a systematic and empirical analysis of app components, including communication across userspaces, visibility of the user, and restrictions.
- We analysed the main four implementations of the Android Multi-user feature (Samsung secure folder, Huawei private space, Xiaomi second space, Google Multi-user) and show how to bypass Android lock screen protection in them, giving full access to an adversary through app components.
- We showed how spyware can use app components to stealthily access the camera and microphone in the background, breaking the OS countermeasures against such techniques up to Android 10.
- We presented Four Gates Inspector, our open-source static analysis tool, to detect the use of specific APIs (e.g. sensor access) in app component handlers. We showed that Four Gates Inspector can not only detect known samples of background spyware but also identify exposed components in 34 (benign) apps (out of a sample of 5,783) downloaded from F-Droid [75] with an average runtime of 4.3 s per app.

4.1.2 Responsible disclosure

The vulnerabilities described in this chapter have been responsibly disclosed through the proper channels. Samsung secure folder's issue was reported on August 18, 2020, assigned moderate severity, and can be tracked via CVE-2020-26606. The vulnerability of Huawei's private space was reported on August 18, 2020, assigned moderate severity, and can be tracked via CVE-2020-9119. We reported Xiaomi's second space issue on August 28, 2020. However, Xiaomi informed us that a third party had reported this issue in parallel and it was fixed on August 31, 2020. Finally, we reported Google's Multi-user feature issue on September 17, 2020. In contrast to the other vendors, Google considered this as intended behaviour so it does not intend to deploy a fix.

The stealthy background access to the camera and microphone issue was reported to Google on September 18, 2020, and assigned a high severity. Google then confirmed that the issue had been fixed in Android 11. Google did not assign a CVE to this issue, as our report coincided with the release of Android 11 and thus did not affect the latest Android OS. The issue can be tracked via id 175232797[1] on the Google issue tracker. We further explored the respective changes in Android 11 and found that Google re-designed the permission system to mitigate this issue [30] specifically.

To ensure the reproducibility of our work and to provide the community with a relevant sample of vulnerable apps for evaluating future attacks and defences, we provide our code, demo videos, including Four Gates Inspector at `https://akaldoseri.github.io/a-tale-of-four-gates/` and `a-tale-of-four-gates` repository at `github.com`.

4.1.3 Outline of this chapter

The remainder of this chapter is structured as follows: In section 4.2, we provide an overview about app components and Android's Multi-user feature. Then we discussed the related work concerning their security to define the scope of our work in section 4.3. Research methodologies discussed in section 4.4. Based on this, in section 4.5, we present our analysis and the vulnerabilities discovered in various implementations of Android Multi-user feature. Subsequently, section 4.6 discusses how stealthy spyware can abuse app components to access the camera and the microphone in the background. To assess the practical relevance of these issues, in section 4.7, we propose Four Gates Inspector, a static analysis tool for app components to evaluate real-world mobile apps. We discuss the discovered vulnerabilities in section 4.8 and propose specific mitigations. Finally, we conclude in section 4.9, discussing limitations and proposing opportunities for future work.

4.2 Background

We first give an overview of app components and their security aspects.

App components: Mobile apps are isolated using kernel-level application sandboxing by Android OS [166]. For app communication, Android offers mechanisms to share data between apps. Among them are app components which serve as entry points allowing communication between apps. They are widely used in practice. Thus, we focus on this aspect. There are four types of app components: *Activity*, *Service*, *Broadcast receiver* and *Content provider*. An *activity* consists of a user interface and an executable code section. *Services* run in the background without user interface/interaction. *Broadcast receivers* respond, once registered, to broadcasted Android Intents. *Content providers* abstract the interaction with app data (e.g. files or SQLite databases) [13].

App components are accessible to other apps if their respective `exported` attribute in the manifest is set. They can be restricted by assigning them app-specific (custom) permissions which provide a protection level from low (*normal*) to higher levels (*dangerous*, *signature*, and *signatureOrSystem*) [27]

Multi-user feature: In addition to app isolation, Android also provides a Multi-user feature that allows one to set up several isolated users on a single device. Each user has a workplace to store data and install apps [148]. To prevent users (including the `shell` user) from accessing other's data, users' data is stored in a separate virtual area in the internal storage [33]. Users can also lock their space via Android Gatekeeper (e.g. through their PIN or fingerprint) [23]. Use cases include a device shared by family members or an on-call team.

[1]`https://issuetracker.google.com/issues/175232797`

4.3 Related work

Privileges escalation on custom permissions: Custom permissions have a unique name and protection level to ensure access protection for their allocated resources. Several issues have been reported in custom permissions related to OS and app updates: Xing et al. showed that one can claim ownership of resources (e.g. permissions and package names) by defining them in an app in an old Android OS before they are introduced in an OS update [193]. (e.g. a malicious app can obtain the system permission `ADD_VOICEMAIL`). Tuncay et al. showed that an adversary can obtain signature permissions from third-party apps without signature matching. The adversary first installs two apps for this: one defines the target permission, and the other grants it. By uninstalling the first app and installing the victim app, the second app gains access to the victim app using its target permissions, regardless of signature mismatches. Android OS does not revoke granted permissions by default [183]. Considering the properties of Android that caused these issues, Rui Li et al. proposed a fuzzer to detect such problems by evaluating custom permissions in randomly created apps installed on devices across different test cases, including system update and app update. Their work demonstrates several issues that allow adversaries to obtain sensitive permissions without user consent [119].

Confused deputy attacks and exposed app components: Confused deputy attacks are vulnerabilities in which an (unprivileged) attacker misuses permissions granted to a higher privileged component due to missing checks in communication interfaces [97]. In the context of mobile devices, it has been repeatedly shown that this problem can manifest in apps [79, 150, 183, 69]. In most cases, the underlying reason was the exposure of app components to all apps on the device by setting their `exported` flag. For instance, as a consequence, a Content provider that manages private app data may expose complete read/write access to this data if no restrictions are imposed [201]. Therefore, researchers have focused on developing tools and methods to detect such issues. Zhou and Jiang systemically analysed 62,519 apps w.r.t. two impacts: data leakage and denial of (some) Android services. Their analysis showed that 1,279 (2.0%) and 871 (1.4%) apps were vulnerable to those issues, respectively [201]. Heuser et al. propose DroidAuditor, an Android module that observes apps behaviour at runtime and generates a graph-based representation of access to sensitive resources (e.g. camera, SMS, etc.) to inspect collusion attacks and confused deputy attacks [98]. Similarly, Yang Xu et al. developed a framework to detect permission-based issues by collecting runtime app states and applying policies and capability-based access control to mitigate these issues at runtime [195]. Bugiel et al. implemented a detection tool that monitors IPC communication at runtime and uses heuristics and detection rules [52]. Reardon et al. proposed a detection tool for covert channels and side channels in apps by monitoring an app's runtime behaviour and network traffic, whereas the interaction with the apps was automated with a user interface fuzzer [150]. Felt et al. developed an IPC inspector tool that temporarily revokes permissions when apps communicate. The revocation process keeps only the commonly granted permissions between the communicating apps [79]. As a limitation, these solutions require changes at the OS level and access to high-privilege services. Also, their dynamic analysis nature may leave some code paths unexplored if only triggered by specific inputs (e.g. on a login screen) [150].

Alternatively, other researchers have proposed approaches based on static analysis. CHEX is a taint-based method to detect leakage in exposed components [124]. AppSealer follows CHEX's approach, which is based on TaintDroid [198, 74]. AppSealer additionally introduced a patch code generator to patch the detected issues. Zhong et al. utilised tainting to detect these issues between selected pairs of applications by comparing their permission and performing inter-application control flow analysis [200]. Finally, Elsabagh et al. proposed FIRMSCOPE, a static analysis tool based on practical context-sensitive, flow-sensitive, field-sensitive, and partially object-sensitive taint analysis [73]. It detects several privilege escalation vulnerabilities in system apps, including code execution using exposed components. FIRMSCOPE achieved a better performance than its predecessors FlowDROID [40], AmandDroid, and Droidsafe. Overall, methods based on taint analysis are limited by their high-performance costs, which make them less practical [195], and over-tainting problems that may lead to false positives.

To overcome these limitations, in section 4.7, we propose a static analysis tool that detects (i) confused deputy issues and the potential bypass of system restrictions and constraints (e.g. using the camera in the background) and (ii) exposure of specific libraries (e.g. tracking and scoped storage). For (i), we noticed that [79] does not consider the scenario of two apps that expose the use of the same permission (e.g. access to the camera) and communicate with each other. However, in section 4.6, we show that this can bypass Android OS restrictions to access the camera in the background. For (ii), most previously proposed solutions rely on permission settings. Recently, noticeable operations like scoped storage [20] introduced in Android allow access to the internal storage for read/write without storage permission, which might not be detectable by existing tools. Finally, given that taint-based approaches are computationally costly, we decided to investigate the alternative approach of statically analysing the execution trace of app components in a graph-based representation and detecting issues on the level of Smali code.

Compromising user privacy: Starting from Android 9, Android prevents background services from accessing the camera and the microphone, even if they have the necessary permissions. Apps with foreground services can still use the camera and the microphone without being them in focus, but only if they display a persistent notification to users [15]. Sutter and Tellenbach discovered a race condition in this functionality: quickly hiding the foreground service's notifications allowed them to create spyware that uses the camera and microphone without showing any visible notification [177]. In Android 11, the camera and microphone can be used only while the app is active and in use (*i.e.* in the foreground) [22]. In our work, we bypassed the restriction that prevents access to the camera and the microphone while the app is in the background. We achieved this by accessing these sensors through apps via exposed components from another app, which by activity manager implementation are considered foreground visibility apps (*i.e.* not background).

Containers and Multi-user feature: The Android Multi-user feature has received considerable attention from the research community: Ratazzi et al. performed a systematic security evaluation, considering a threat model where multiple users share the same device. They discovered several vulnerabilities, e.g. the secondary users can access the owner's services like VPN, network, backup, and reset settings because they are exported publicly [148]. OEM-specific functionality derived from the Multi-user feature has suffered from several issues in the past: the security space service that manages Xiaomi's second space allowed an adversary to switch to the second space without user authentication [76]. This issue occurred because a related service was publicly accessible, and an adversary could start it by crafting an intent to bypass the authentication process. Kanonov and Wool reported several issues in Samsung Knox containers, including a man-in-the-middle vulnerability in the VPN service and data leakage from the Knox clipboard [108]. We demonstrate an issue that allows extracting data from Multi-user and OEM-specific spaces, bypassing their dedicated lock through the shell user permissions.

4.4 Research methodology

In this paper, we used two main methodologies for our analyses.

Android source code and documentation review: We reviewed Android documentation to understand the functionality and limitations of possible app entry points. Also, we reviewed changes in the documentation for different Android OS versions to discover issues that might affect older but still widely installed OS versions. In addition, a manual source code review of relevant code parts was conducted to investigate the technical details of different Android components to understand the root causes of the discovered vulnerabilities.

Black box testing and reverse engineering: In addition to documentation and code review, we also applied black box testing in certain services, *i.e.* without knowledge of the source code [141]. We used this approach for two reasons. Firstly, the Android architecture contains numerous complex components and services, which makes full source code review prohibitively time-consuming. Secondly, some services' source code is not publicly available (e.g. Samsung Secure folder). Finally, our work also required reverse engineering of mobile applications to determine their behaviour and app components. This approach was automated with our Four Gates Inspector tool described in Section 4.7.

4.5 Analysis of app components across userspaces

Starting from Android 5, Android supports a Multi-user feature that allows adding additional users to the device so that a device can be shared by multiple people [33]. Regarding the security guarantees provided by the Multi-user functionality, the relevant documentation states that "each user gets a workspace to install and place apps" and "No user has access to the app data of another user" [33]. Technically, as discussed in section 4.2, each user's data is stored in a separate virtual area to prevent accessing the data of other users. Users can lock their workspace via Android Gatekeeper [23]. However, apps can interact with each other using app components. Thus, the following questions arises: (i) if it is possible to invoke app components of an app in one userspace from another userspace; and (ii) if this can be used to bypass the user-specific lock screen.

Accessing app components with different users: We experimentally investigated interactions between apps in two userspaces. We started by creating two users, "adversary" and "victim", and two apps in each space: a callee app with four exposed app components (Activity, Service, Broadcast receiver and Content provider) and a caller app to invoke the victim app components. Our initial analysis showed that apps can only interact with each other if they are in the same space. Debugging the caller to send messages via `adb` using the activity manager `am` and `content` tools showed that messages target apps in the same userspace by default. Yet, we found that both tools accept a `--user` flag to specify the user ID of the target user, unlike their equivalent Java API. Specifying a user ID (obtained via the package manager `pm`) in the messages results in a permission denied exception from the `UserController` class. We thus confirmed that apps cannot communicate across userspaces by default.

Underlying permission: After analysing the exception, we found that `UserController` class manages Multi-user functionality for `am` and, a user can interact with app components from other userspaces if it has `INTERACT_ACROSS_USERS_FULL`, `INTERACT_ACROSS_USERS` permissions for Android intents, and `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permission for Content providers.

These permissions are granted only for system apps with signature protection level (*i.e.* apps signed with the system image certificate) [16]. An accessible user is the `shell` user (implemented by `com.android.shell`). We analysed its respective APK file and found that the shell package has all these permissions. Utilising `dumpsys`, we confirmed that the `shell` user is effectively granted these permissions.

By default, the `shell` user does not have read/write access to other userspaces. Using commands like `ls` or `cd` to access them leads to insufficient permission errors. Therefore, we focused on exploiting these permissions. We sent both an intent and a content query using the `shell` user, specifying the victim's user id, and noticed that sending an intent to an activity does trigger the lock screen protection (Gatekeeper). However, this is not the case for Services, Broadcast receivers and Content providers, as they do not have a visible user interface. We thus could successfully bypass the Gatekeeper protection and interact with apps from different userspaces. (e.g. a secondary user can extract information from an owner user). Yet, practically exploiting this requires a vulnerable app in the target userspace or the ability to install specific apps (e.g. an app that exposes user information through an app component) into the victim's userspace.

Issue #1: A user with `INTERACT_ACROSS_USERS` and `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permissions (e.g. the `adb` shell user) can access all userspaces and perform actions in their context by interacting with app components even if the user does not have read/write access to these spaces.

Threat model: Ratazzi et al. considered a threat model based on physical access to the device where multiple users share a single device to show data leakage issues in the Multi-user feature [148]. Such a model applies to our issue. However, we also considered a more generic threat model, where a victim uses a service that requires access to ADB (e.g. Vysor, GenyMobile, MirrorGo, ApowerMirror). These services are widely used for screen mirroring, operate on desktops or browsers via WebUSB, and have over 12M downloads in total. Because ADB gives the service access to potentially sensitive data in the userspace, we further assumed that the user has created a second userspace to limit the information accessible via ADB. Clearly, this threat model may be seen as specific and may require additional social engineering or similar circumstances. It does not require knowledge of PIN or fingerprint of the target user or rooting the device and is thus applicable to Android devices where rooting is detectable, e.g. through a warranty bit [164] or where PIN/fingerprint are protected by secure hardware.

Data extraction across user userspaces: To demonstrate a proof-of-concept of data extraction from another userspace (e.g. private profile or secure space), we first observed that the package manager system binary `pm` also accepts a `--user` flag, similar to `am` and `content`. `pm` exposes various app management operations, including app installation and granting arbitrary permissions. Thus, a combination of `am`, `pm`, and `content` allows for a complete bypass of the isolation between userspaces for a `shell` adversary. Concretely, the attack proceeds as follows:

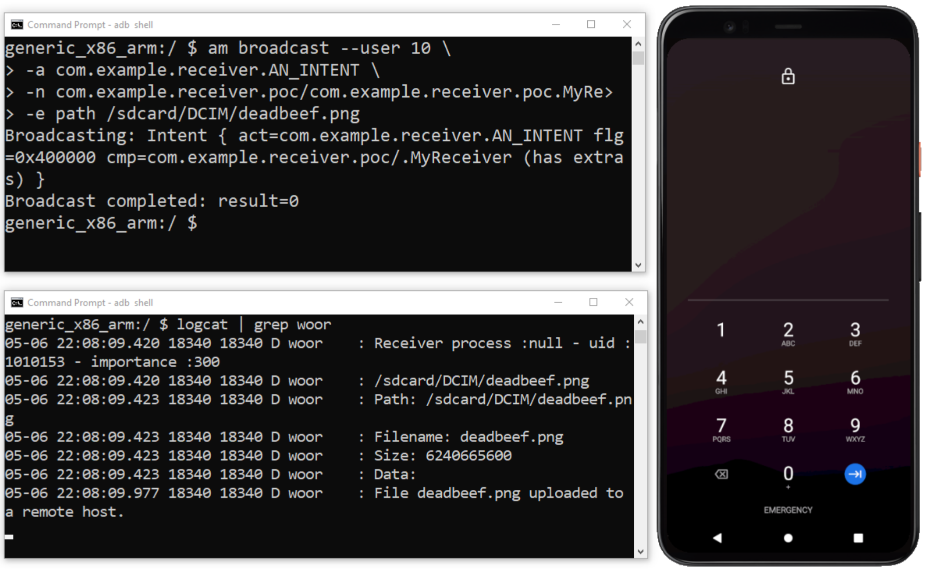
- The victim visits an adversary-controlled website with ADB support (e.g. disguised as a screen mirroring service) and connects their device to it via WebUSB.
- The adversary's website silently installs a malicious app into the victim's other userspace (e.g. one created to separate sensitive data from a userspace used for screen mirroring) through `pm`.

  ```
  $ pm install com.app.malicious.apk --user 2
  ```
- Using `pm`, the adversary grants the malicious app read access to the victim userspace's internal storage (e.g. `READ_EXTERNAL_STORAGE`) (or other permissions).

  ```
  $ pm grant com.app.malicious READ_EXTERNAL_STORAGE --user 2
  ```
- Finally, the adversary use the `shell` user to either send an intent or a content query to the malicious app to communicate with its app's components (Service, Broadcast receiver, or Content provider) and instruct it to upload data (e.g. pictures and content of the internal storage) to a server under the attacker control.

  ```
  $ content query --uri content://com.app.malicious/cp --user 2
  ```
- To hide the attack's traces, the malicious app can finally be removed via `pm`.

Alternatively, the above attack can also be carried out by an adversary with physical access to the device and a userspace (e.g. an abusive partner or victim's family member). Figure 4.1 shows a screenshot of sending a broadcast intent to a malicious app's Broadcast receiver in the victim space, instructing its app to upload an image to a remote host.



```
generic_x86_arm:/ $ am broadcast --user 10 \
> -a com.example.receiver.AN_INTENT \
> -n com.example.receiver.poc/com.example.receiver.poc.MyRe>
> -e path /sdcard/DCIM/deadbeef.png
Broadcasting: Intent { act=com.example.receiver.AN_INTENT flg
=0x400000 cmp=com.example.receiver.poc/.MyReceiver (has extra
s) }
Broadcast completed: result=0
generic_x86_arm:/ $
```

```
generic_x86_arm:/ $ logcat | grep woor
05-06 22:08:09.420 18340 18340 D woor    : Receiver process :null - uid :
1010153 - importance :300
05-06 22:08:09.420 18340 18340 D woor    : /sdcard/DCIM/deadbeef.png
05-06 22:08:09.423 18340 18340 D woor    : Path: /sdcard/DCIM/deadbeef.pn
g
05-06 22:08:09.423 18340 18340 D woor    : Filename: deadbeef.png
05-06 22:08:09.423 18340 18340 D woor    : Size: 6240665600
05-06 22:08:09.423 18340 18340 D woor    : Data:
05-06 22:08:09.977 18340 18340 D woor    : File deadbeef.png uploaded to
a remote host.
```

Figure 4.1: Utilising the shell user to send a broadcast intent to an app's receiver to upload an image to a remote host.

An adversary can also perform other actions using the system's Media Content Provider, which abstracts the interactions with a user's gallery. This includes: (i) reading all images from the victim's userspace, (ii) placing own images into the victim's userspace, possibly with incriminating content, and (iii) Denial-of-service (or ransomware-like) attack by deleting data (or threatening to do so) inside the victim's userspace.

Vendor-specific Multi-user implementations: The above issue was demonstrated for the Multi-user feature of Google devices. Other vendors modified this functionality and referred to it by their names: Samsung's secure folder, Huawei's private space, and Xiaomi's second space. They have similar features to the default Android Multi-user mode. However, some of these variants use Android profile like Samsung Secure folder [21]. Additionally, they have been enhanced with protections to prevent access in case of rooting the device (e.g. through Samsung Knox) [165, 164] or while the device is in debug mode for Huawei.

We thus analysed to what extent these implementations suffer from the same issues as the default Multi-user feature. For all tested variants, an `adb` shell provided a user with the required `I NTERACT_ACROSS_USERS` or `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permission. We further found that, depending on the vendor, some restrictions were imposed on the `pm`, `am`, and `content` binaries. Section 4.5 summarises our findings.

Apart from the Samsung secure folder, all implementations allowed installation of apps in the victims' userspace or profile and were thus vulnerable to the above attacks and exposed full access to the gallery through the Media Content Provider. For Samsung, we compared two versions of the secure folder (1.2.32 for older devices below Android 9 and 1.4.06 for new devices). In both cases, installing apps was prevented. However, the Media Content Provider exposed full image data (for 1.2.32) or at least metadata (for 1.4.06), e.g. timestamps and geolocation.

Samsung and Huawei assessed the respective vulnerability as "moderate". The issues can be tracked for those vendors via CVE-2020-26606and CVE-2020-9119, respectively. Xiaomi was already aware of this issue at the time of our report and rolled out a fix shortly after that. Google regarded the issue as intended behaviour and thus did not plan to release mitigations.

Issue #2: Adopted Android Multi-user feature variants are accessible by users with INTERACT_ACROSS_USERS permission.

Vendors		Google/ Android One Multi-user	Samsung Secure Folder 1.2	Samsung Secure Folder 1.4	Huawei /Xiaomi Spaces
Activity Manager	#1 Access broadcast receiver/service	●	●	●	◐
Content Provider	#2 Access Content provider	●	●	●	●
Media Content Provider	#3 List images / meta data	●	●	●	●
	#4 Insert (empty) images	●	●	●	●
	#5 Read images	●	●	○	●
	#6 Write images	●	○	○	●
	#7 Delete images	●	●	●	●
Package Manager (PM)	#8 List packages	●	●	●	●
	#9 Pull applications	●	●	●	●
	#10 Install applications	●	○	○	●
	#11 Uninstall applications	●	●	●	●
	#12 Grant/Revoke permissions	●	●	●	●

Table 4.1: A Summary of operations that can be executed on behalf of a target user across userspaces for different vendors. Symbols indicate whether a vulnerability was successfully exploited (●), not found to apply (○), or not tested (◐). Note that Android One devices share the same vulnerabilities as Google devices.

4.6 Analysis of sensor background access

Based on section 4.2, we determined that any other app can start app components with appropriate permissions. Also, app components like Services, Broadcast receivers and Content providers are invisible to the user (no notifications) but can nevertheless access certain Android APIs, including sensors like camera and microphone (if the app has the required permissions). Since Android 9, the OS prevents apps from accessing those sensors in the background, even if the app has the required sensor access permissions [15] to ensure user privacy. This restriction applies unless a foreground service displays a persistent notification to the user indicating its activity [22]. This restriction raises the question of how Android distinguishes between “foreground” and “background” at a technical level, especially how invocations of app components like Content providers and Services are handled with reference to this aspect.

Importance of app components: To answer this question, we developed a test app with four app components (Activity, service, Broadcast receiver, and Content provider). Each component then dumps the current “importance” of the app. This importance refers to a numerical value that precisely specifies the visibility and foreground/background state of the app [9]. Generally, the foreground is given an importance of 100, while larger values indicate states increasingly in the background. Section 4.6 summarises the observed importances.

Status/AppComponents	Activity	Content provider	Broadcast receiver	Service	Foreground Service
Foreground	100	100	100	100	100
Background (minimised)	300	200	300	300	125
Stopped	100	200	300	N/A	125

Table 4.2: Importances of an app when accessed by the four app components

We noticed that the app has an importance of 100 (foreground) when visible to the user, regardless of any app components used. When the app is minimised and stopped, the observed importances range from 125 (foreground service) over 200 (visible) to 300 (service). Using the activity manager’s `start-foreground-service` option, background services can be run as foreground services and be assigned the respective importance, even if they are not developed as foreground services [22]. This observation highlights that “background” in Android OS does not refer to a single state but a range of importances. As Android prevents access to the camera and the microphone in the “background”, it is essential to determine for which importance this block is active.

Potential issue #3: Android prevents access to the camera and the microphone for apps running in the “background”. This translates to a block based on specific background conditions and/or apps’ importances. If there are app components that are not affected by these restrictions, it might be possible to obtain stealthy access to sensors by invoking an app component to “leak” the sensor data from another (background) app.

To determine this, we developed two apps. The first app had three app components (Service, Broadcast receiver and Content provider). We excluded the Activity because it always runs in the foreground, and thus it is visible to the user. We granted access to the app’s microphone and camera and intentionally exposed these functions through all three app components. The second app then invokes the app components of the first app. We configured the intent for accessing the Broadcast receiver and the Service with `FLAG_INCLUDE_STOPPED_PACKAGES`, allowing it to invoke stopped apps. Crucially, we found that *all three app components were capable of accessing the camera with the first app in the background*, while the microphone was accessible from the Service and Content provider. No notification is shown to the user, making exploitation of this issue stealthy. Yet, in the initial proof-of-concept, the second (invoking) app was in the foreground, making practical exploitation obvious. To avoid this, we next considered invoking the respective app components from a background app.

4.6.1 Stealthy background spyware

Android provides several services (e.g. Timer, JobSchedule, AlarmManage and Runnable) for an app to persist in the background after it has been closed. These services are described in section 4.6.1. While direct access to the camera and microphone is blocked in the context of these services by Android to prevent background spyware, app components can be invoked using these methods. As the timer class provides a minimum invocation interval of 1 s, it represents the best choice for an adversary to frequently capture camera images and audio through a second app's components.

Service	Importance	Package	Interval
Timer	400	Java.util	1 s
JobSchedule	230	Android.app.job	15 min
AlarmManager	300	Android.app	1 min
Runnable	300	Java.lang	Stopped in few sec

Table 4.3: Methods for applications to persist in the background after being closed (importance 230: perceptible; 300: background service; 400: cache)

Combining the caller app running in the background with a callee app that exposes sensor access through app components, stealthy, persistent spyware can be devised. Concretely, the attack proceeds as follows:

- The adversary installs (or tricks the user into installing) a *sensor app* that exposes suitable app components (Service, Broadcast receiver, or Content provider). This app must be granted sufficient permission to access the respective device sensors (*i.e.* camera and microphone).
- The adversary then installs (or tricks the user into installing) the second *spyware service app* ①. This app has a single background service that starts a timer. This timer periodically communicates with the app component of the sensor app to record images, and audio ②, and e.g. then upload them to a remote server under adversary control. The spyware app requires no permissions, and its service can run in the background ③.

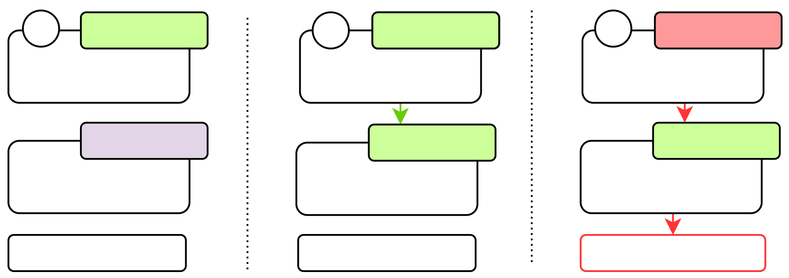


Figure 4.2: Example scenario showing how spyware can use the camera and microphone in the background by being split into a background spyware app and a sensor app with an app component that exposes camera or microphone. The visible state represents the visible visibility where an app is not visible to the user and is considered to be in the foreground.

The spyware app periodically communicates with the app component of the sensor app in the background. This will bring the sensor app temporarily to importances of 125, 200, and 300 (foreground service, visible and service), enabling access to all device sensors. Figure 4.2 illustrates the process of a background spyware that abuses an exposed sensor app to use the camera and the microphone in the background. First, the spyware app starts in the foreground ①. Then, it communicates with the sensor app through an exposed app component via a background job service ②. At this point, the sensor app enters the visible state. The sensor app is not visible on the screen in this state. However, it can access the camera and the microphone like an app in the foreground. Finally, the spyware service can turn to the background and communicate with the sensor app, instructing it to take photos using the camera or record audio via the microphone ③. While the pair of our attacker apps run, there are no user-visible indications or notifications. Because the service app does not consume substantial device resources, after 3–5 s of running, Android will white-list it and consider it a cache process, which will not be terminated. The cached process can remain active for a long time—in our tests, it ran continuously for more than two hours until we force stopped it. The attack works regardless of using other apps, cameras, or phone locking.

Issue #3: Background services cannot use the camera and microphone in the background. However, they can continuously instruct other apps to use the camera and the microphone via their app components.

Reviewing the source code [89] of Android 10 shows that the camera and microphone rely on the activity manager to determine if the caller app is active (not idle or in the background)[2] by checking its Proc state[3] [4]. The Proc state[5] value is a numeric value convertible to an importance. The activity manager implementation translates the observed importance of 125, 200, and 300 into the Proc states of foreground service (PROCESS_STATE_FOREGROUND_SERVICE), foreground (PROCESS_STATE_IMPORTANT_FOREGROUND), and service (PROCESS_STATE_SERVICE), which are considered not in the background and thus allow bypassing sensor access restrictions.

Threat model: We considered the "two-app setup" threat model widely used in the literature [79, 150, 183, 69]. We assumed an adversary that can install two apps on the target device (instead of one for "classical" spyware). This adversary model is realistic for spyware, where the attacker (e.g. an abusive partner) might have temporary access to the device or be able to social-engineer a victim into installing apps disguised as harmless software (e.g. games or other apps by the same developer). Additionally, we considered the threat model of [201], where a benign app exposes access to sensors via exposed components: a malicious app can scan for such vulnerable app components and thus "inherit" their permissions. According to our experiments, the issue affects all devices running Android 10 and earlier. It does not replicate on Android 11 due to changes in the permission system as discussed in section 4.8. Our attack does not require root or `adb shell` access and hence also applies to devices where no root exploit exists or rooting leaves traces. As we detail further in section 4.7, we found instances of this issue in some of the tested apps.

4.6.2 Confused deputy issues in benign apps

Apart from the case where an adversary installs two spyware apps (one with sensor permissions exposed through app components), spyware might also exploit missing restrictions in benign apps as explained in section 4.3. For example, suppose a widely-installed, legitimate camera helper app accidentally exports its access to the camera through an app component (e.g. to facilitate some sharing functionality). In that case, a malicious app might scan for such vulnerable app components and thus "inherit" their permissions. As we further detail in section 4.7, we found this issue in some of the tested apps.

4.7 Evaluation

We evaluate the discovered issues to assess their impact and provide a detection mechanism for security researchers and mobile developers. First, the Multi-user issues only affect OEM devices and are somewhat limited. Therefore, debugging the device using the *pm* and *am* binaries as discussed in Section 4.5 is sufficient for detection. A limitation of this approach might be that some devices do not have developer options or the shell user is inaccessible.

Conversely, the background spyware issue affects mobile apps and requires analysing the apps to detect them. Therefore, we designed Four Gates Inspector to systematically discover corresponding issues and resolve limitations of existing solutions. Security researchers and mobile developers can use our tool to detect possible malicious apps. This is especially relevant when vendors pre-install third-party apps as part of system images. The following discusses the design, implementation and practical results obtained with Four Gates Inspector.

4.7.1 Four Gates Inspector

As discussed in section 4.3, runtime analysis tools previously proposed in the literature might not reach code paths that require specific input, e.g. login screens [150] and require changes to the OS, which makes them less practical. Additionally, taint-based solutions suffer from non-negligible

[2] `https://android.googlesource.com/platform/frameworks/native/+/refs/heads/android10-c2f2-release/libs/binder/IActivityManager.cpp#82`

[3] `https://android.googlesource.com/platform/frameworks/av/+/refs/heads/android10-c2f2-release/services/camera/libcameraservice/CameraService.cpp#982`

[4] `https://android.googlesource.com/platform/frameworks/av/+/refs/heads/android10-c2f2-release/services/audiopolicy/service/AudioPolicyService.cpp#782`

[5] `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android10-c2f2-release/core/java/android/app/ActivityManager.java#2956`

computational cost and over-tainting, possibly leading to false positives [195]. To overcome this limitation, we investigated the potential of developing a static analysis tool that aids in detecting these issues. We designed Four Gates Inspector to statically analyse apps and detect confused deputy issues based on given classes and methods. Our tool is based on analysing the execution trace of the decompiled Smali code of app components. Four Gates Inspector offers a filter list that limits the scope of the analysis (e.g. camera usage, storage leakage, GPS). It provides fine-grained control over the analysis scope on the class level, compared to the permission level used by Bugiel et al. 's framework [52]. This allowed us to trace issues related to non-permission classes (e.g. scoped storage in Android 11 [20]). Security testers and Android marketplaces can use the tool to investigate these issues and perform automated security testing for published apps. Because the tool achieved high performance (It can analyse apps in a few seconds), it is suitable for Android marketplaces to analyse millions of apps quickly.

Design and Implementation: Four Gates Inspector focuses on the execution trace of method invocation of app components, in contrast to taint-based solutions that trace the input and may introduce over-tainting [195]. Thus, we avoid false positives (e.g. sensors usage not within app components) and can detect attempts to hide sensor use (e.g. through nested calls). Our tool is implemented in Python and requires `apktool` [37] to decompile mobile apps and `untangle` to parse XML files. Four Gates Inspector consists of four main modules: *Smali Handler*, *Manifest handler*, *APK handler* and *Component inspector*.

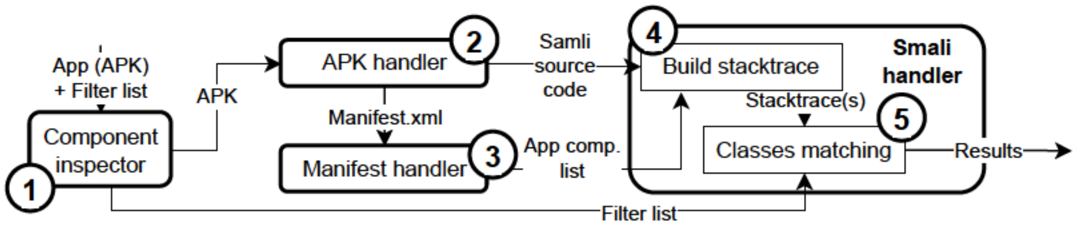


Figure 4.3: Flow of Four Gates Inspector for analysis of app components.

Overall, Four Gates Inspector implements the basic flow shown in Figure 4.3 to analyse an app. First, the *Component inspector*, which controls the flow of the tool, starts by loading the application file (APK) and the filter list (1). The *APK handler* unpacks the app to extract the Smali code and Android manifest (2). The *Manifest handler* parses the app's Android manifest to detect app components and the use of specific classes and APIs (e.g. camera, microphone or storage) (3). To this end, the *Smali handler* initiates static analysis. It receives the component name from the Android manifest, builds a stack trace of all invoked methods and their classes for each component by parsing their Smali source code, and explores the stack trace for the usage of filter list (4).

The stack trace (Figure 4.4) is a tree-structured graph `G(N,V)`: each app component has its `G(N,V)` tree, where `N` is a set of nodes that represent the invoked classes/methods in the execution trace, and `V` is a set of vertices that represent the invocation sequence between nodes. Figure 4.4 shows an example of the generated stack trace tree of an app that uses the camera through an `CameraService` component via a custom `Helper` class. The graph starts with a single root node `n0` for the app package class. Node `n1` at level 1 represents an app component class. The *Smali handler* starts by identifying method and class invocation within `n1`. The top-level invoked methods in `n1` (e.g. constructor) are added as children to `n1`. Then, each method at level 2 is analysed to find further invocations to be added to the tree at level 3. The level 3 nodes might have an invocation to external classes and methods. These external invocations are added to the root node `n0` as child nodes. The process is repeated, starting with the Smali handler for the new level-1 nodes until all code has been traced. Once the stack trace is complete, the Smali handler performs tree traversal for each stack trace to detect the use of certain classes/methods. The detection is determined by checking if a class/method is within the filter list. This process is repeated for each component. Finally, Four Gates Inspector produces a summary report containing detailed information about the inspected issues, class usage, stack traces, (custom) permissions and exported flags for each usage of the filtered APIs (5). Each stack trace of an app component is generated independently and reported to allow partial processing of apps, which avoids exhausting the memory for mass scans. Additionally, the graph approach includes measures to prevent infinite loops or recursions by

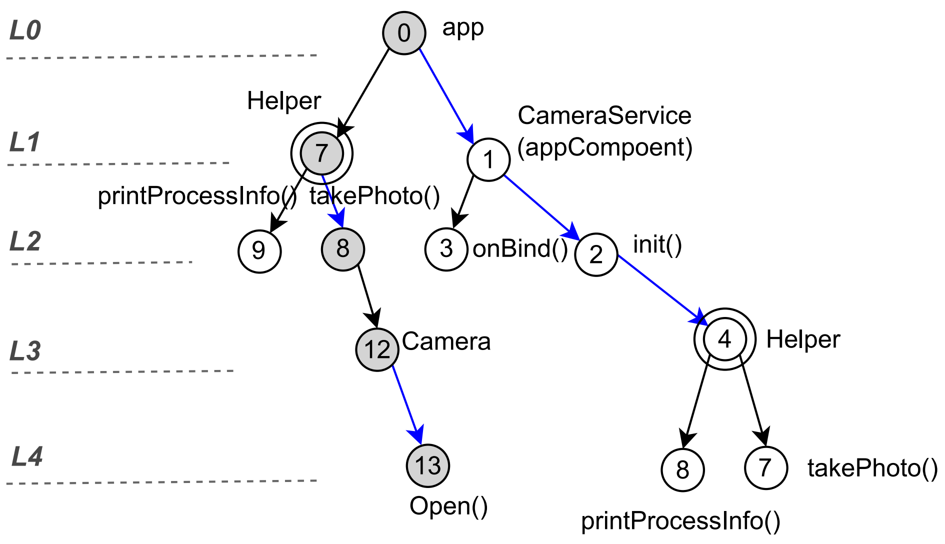


Figure 4.4: Stack trace tree generated by Four Gates Inspector.

ensuring that nodes at level 1 have unique values to avoid exploring the same classes unless new methods are invoked.

4.7.2 Real-world tests

We evaluated the correctness and effectiveness of the tool using two test samples:
Known samples test: We analysed two known samples for using the camera in the background. The first sample is the proof-of-concept for CVE-2019-2219, utilising a race condition in a foreground service to use the camera in the background [177]. The second sample is our proof-of-concept from section 4.6.

Unknown samples test: To detect unknown instances, we downloaded 6,687 apps from F-Droid, a repository of free and open-source apps [75]. The apps are from multiple categories, including navigation, development, graphics and system apps. We also considered multiple versions of an app to detect exposed components that were fixed (or introduced) with new versions. After eliminating duplicates, this yielded 5,783 apps. We analysed these apps with Four Gates Inspector, filtering for the camera, microphone, hardware sensors (accelerometer/gyro) and GPS use. The testing was followed by a manual analysis of the detected components to measure the correctness of the tool. We did not use commercial apps (e.g. from Google Play) to avoid legal issues with terms and conditions that prohibit reverse engineering—this decision was taken because manual screening of the terms of thousands of apps is infeasible.

4.7.3 Evaluation and results

In the known sample test, as expected, Four Gates Inspector successfully detected both spyware samples and indicated the exact app components where the camera and the microphone are used. In the unknown sample test, the tool successfully generated reports for all 5,783 apps. Of these, 151 apps used sensors. Filtering for exported components apps yielded 34 apps with 43 components using sensors. We evaluated the correctness of the tool by manually analysing the Smali code of all those 34 cases of sensor use. We found that all 43 were correctly identified with sensor usages by Four Gates Inspector

Regarding performance overhead, we measured the execution time for analysis of each app under Ubuntu 16 on an i7 with 16 GB RAM. We calculated the average runtime of over 5,783 apps to be 4.3 s, which is reasonable for mass analysis, especially considering that our tool can be run in parallel. Comparing our tool to existing solutions, Four Gates Inspector is faster than other tools: AppSealer and FIRMSCOPE report 1–3 min and 7 min on average, respectively, to process an app. It should be noted that the authors of AppSealer used a machine similar to ours (Intel i7 with

8 GB RAM), while FIRMSCOPE ran on a high-performance server (Intel Xeon 40-core processor with 150 GB RAM).

Remarkably, Four Gates Inspector found two instances of confused deputy issues in the tested apps, highlighting the practical relevance of our approach: `com.commonslab.commonslab` and `org.wikimedia.commons.wikimedia` , both used to access Wikimedia Commons, and expose the audio recording permission to any unprivileged app on the same device.

We did not find clearly malicious spyware in our sample collection, indicating that this type of malware is not common on open platforms like F-Droid. Comparing our results to Zhou and Jiang's study from 2012 [201], we observed a clear improvement in adopting proper security practices: most apps disabled the `exported` flag for their app components. Additionally, our tool can correctly distinguish between vulnerable and secure versions of apps. For instance, only version 9.1 of `net.majorkernelpanic.spydroid`had a component the use a sensor, but not the earlier and later versions.

4.7.4 Limitations

Four Gates Inspector has certain limitations. It handles inner classes, interfaces, extended classes, and case-sensitive naming (tested by considering different versions of the apps). However, the tool cannot detect overridden methods unless their invocation can be traced (e.g. within the app component). This is because Smali code does not label them, which makes detection difficult, especially for event-based (e.g. `onStopped`, `onResume`, etc.) and native classes not shipped as part of the app package. Furthermore, our tool does not detect native (non-JVM) code nor runtime-evaluated code, which includes dynamically-registered Broadcast receivers, reflection and `eval()`. To overcome this limitation, the Smali handler needs to be changed to semantically analyse such instructions and then process them using the usual flow. We leave this for future work and welcome improvements from the community.

4.8 Discussion and mitigation

Multi-user feature: The vulnerabilities in Android Multi-user functionalities (section 4.5) affect both Android Multi-user feature and Android profile. They allow an adversary with *shell* access (remotely via WebUSB or physically) to the device to bypass the dedicated Gatekeeper protection and extract data from other userspaces. We conclude that this issue is caused by granting the `INTERACT_ACROSS_USERS` and `ACCESS_CONTENT_PROVIDER_EXTERNALLY` permissions to the `shell` user. In addition, Services, Broadcast receivers and Content providers do not have a user interface, and hence Gatekeeper protection is not invoked when such app components are accessed across userspaces. We suggest the following mitigations:
Removing the permission from the shell user: Not granting these permissions to the `shell` user would resolve the issue partially for both the default Android implementation and vendor-specific variants. However, a side effect of this mitigation is that developers will not be able to install applications for other users within the same device using the debug mode. Therefore, removing the permission from the shell user is not a practical solution.

Sanitise the user flag of system binaries: Alternatively, system binaries that allow access across userspaces (`am`, `pm` and `content`) could blocklist user IDs of other spaces. This prevention technique can be applied to vendor-specific implementations, as they provide a single secure space with a fixed user ID. In contrast, Android Multi-user feature allows the creation of several users with different user IDs. Instead of blocklisting certain IDs, an allowlisting approach might be applicable.
Samsung seems to have applied the second approach, blocking the user ID of their secure spaces for `am`, `pm`, and `content`. While Huawei's patch includes verifying the password of private space when the developer mode is enabled and informing the users about the developer mode's risks when the developed mode is enabled and entered. The mitigations taken by Xiaomi are unclear, as a parallel disclosure led them to not engage further during the patching process. As mentioned, Google's assessment regards this issue as not being a security vulnerability but an intended behaviour. Google does not plan to roll out a fix. The reported issue can be tracked via `https://issuetracker.google.com/issues/168724195`.

We note that users of affected devices (e.g. Google and Android One like Xiaomi and Nokia) might not be aware of the threat model assumed by Google internally and use Multi-user with wrong assumptions about its security guarantees.

Background access to restricted sensors: The root cause of this issue is the definition of "background" in Android. We suggest the following mitigations:

Restrict sensor access from app components: Background services, Broadcast receivers, and Content providers should generally not have access to restricted sensors when the app is invisible. However, applying such a change to an old Android version might break app functionality.

Restrict sensors access when an app is not in use: This would require redesigning the camera and microphone permissions to ensure that sensors are only accessible when the app is visible in the foreground and in-use like in Android 11 [30]. Android 11 relies on the "AppOpsService" [12] to mitigate this issue. The service tracks the app's proc state (importance) using mainly two modes, `MODE_ALLOWED` to allow and `MODE_IGNORED` to suppress access. The service receives the current proc state of an app and capability updates from ActivityManagerService, indicating "while in use" permissions. This mitigation restricts sensor access: If the app is in the foreground, the mode is `MODE_ALLOWED` and behaves normally. When the app goes into the background, this changes to `MODE_IGNORED`, preventing sensor access. Note that `MODE_IGNORED` does not revoke the granted permission.

Google assigned a high severity to our issue and considered it fixed in Android 11 only since our report of the issue coincided with the release of Android 11 (Sept 2020). However, in practice, the privacy impact on users of older versions might be substantial: Many - especially non-tech-savvy - users might not choose or be able to upgrade their devices to Android 11. As of Jan 2022 (two years after the Android 11 release), the majority of Android users (64.63%) still use Android 10 and below [175], without any easily available mitigations.

4.9 Conclusion

In this paper, we demonstrated how app components on Android can be abused to break some of the system's security and privacy guarantees. We show they can be accessed across userspaces boundaries, leading to data exfiltration for various vendors. We demonstrate how spyware can bypass Android's mitigations against background access to the camera and microphone by splitting the process into two apps that communicate over certain app components. Finally, we present our tool Four Gates Inspector that can automatically detect such issues through static analysis. Using the tool, we found, for example, issues in apps that expose the microphone to any unprivileged app on the same device.

Acknowledgement

We thank Ali Darwish, ElMuthana Mohamed, Hamad Salmeen, Abdulla Subah and Zhuang Xu for participating in mobile testing. This research was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1 and EP/V000454/1. Abdulla Aldoseri is supported by a stipend from the University of Bahrain.

Chapter 5

Symbolic Modelling for Android Remote Attestation Protocols

Symbolic Modelling of Remote Attestation Protocols for Device and App Integrity on Android

The contents of this chapter were published as:

> Abdulla Aldoseri, Tom Chothia, Jose Moreira-Sanchez and David Oswald. ”Symbolic modelling of remote attestation protocols for device and app integrity on Android.” In 18th ACM ASIA Conference on Computer and Communications Security (2023) ASIA CCS, . Association for Computing Machinery (ACM), Jan. 2023. DOI:10.1145/3579856.3582812 (`https://doi.org/10.1145/3579856.3582812`) [5].

Preamble.

Based on the research results from Chapter 4, we concluded that Android app-level interfaces' issues can bypass system-level protections and extend to hardware-level protections (e.g. Knox's Samsung Secure folder). In this chapter, we further investigate this area. We noted that several vendors implement attestation services that ensure the integrity of software and hardware of Android devices. In this chapter, we aim to determine if app-level interfaces can bypass hardware-level assurance.

Ensuring the integrity of a mobile app or device is one of the most challenging concerns for the Android ecosystem. Previous software-based solutions provide limited protection and can usually be circumvented by repacking the mobile app or rooting the device. Newer protocols use trusted hardware to provide stronger remote attestation guarantees (e.g. Google SafetyNet, Samsung Knox (V2 and V3 attestation), and Android key attestation). So far, the protocols used by these systems have received relatively little attention.

However, not all the attestation protocols are publicly available, therefore, we formally modelled them using the Tamarin Prover and verified their security properties in the symbolic model of cryptography, revealing several vulnerabilities. We found a relay attack against Samsung Knox V2 that allows a malicious app to masquerade as an honest app and an error in the recommended use case for Android key attestation that means that old – possibly out-of-date – attestation can be replayed.

Finally, we employed our findings and the modelled platforms to tackle one of the most challenging problems in Android security, namely code protection, proposing and formally modelling a code protection scheme that ensures source code protection for mobile apps using a hardware root of trust.

Remarks regarding my work

While I am the main author of this work, several co-authors have also contributed. Following my initial writing of the paper, it has been through many changes and improvements. The co-authors contributed to re-wording, proofreading and writing a small number of the sections. For symbolic modelling, the co-authors guided and helped me complete the models, especially in terms of resolving some technical issues.

5.1 Introduction

One of the major issues confronting the mobile development industry today is ensuring the integrity of a mobile app and its device. Failing to achieve these security goals might introduce issues for the app's user and the developer. For instance, sensitive apps (e.g. banking and payment) often check the integrity of their code and the mobile operating device using various techniques to avoid information leakage [140] (e.g. on a rooted device). Similarly, in mobile gaming, evading integrity checks may enable cheating. Nguyen Vu et al. analysed over 28,000 apps, including mobile banking apps. They concluded that most root detection techniques are bypassable [140], even though Android provides a variety of dedicated mechanisms to safeguard contents and data, such as Digital Right Management (DRM) and the Android Keystore [87]. Android's architecture implies that mobile apps only run in "normal" userspace (and not e.g. ARM TrustZone), thus easily allowing static and dynamic attacks (e.g. reverse engineering, repacking, and debugging) to circumvent integrity checks [34].

Several researchers have proposed software solutions to preserve the integrity of apps and their host devices [49, 176, 174, 121]. However, these software defences can usually be bypassed [140], creating a cat-and-mouse game. Hence, these measures increase the difficulty of app tampering rather than stopping it. More recently, Samsung Knox [156], and Google SafetyNet [91] have introduced a hardware root of trust for attestation, promising much stronger app integrity guarantees. However, their security claims remain largely untested.

In this paper, we examine Samsung Knox, versions 2 and 3, Google SafetyNet, and Android Key Attestation, systematise the problem space of app attestation, and evaluate their corresponding methods symbolically. In order to prove (or disprove) the correctness of app/device integrity checks with hardware-backed remote attestation on Android, we formally modelled and verified them using the Tamarin Prover [47, 128] and its front-end SAPiC [115]. Not all the considered protocols are fully open, so we performed symbolic verification based on publicly available documentation or source code samples.

The novelty of our framework is represented in our way of modelling devices and apps. Each device has a secure world that models the cryptographic operations of the platform, app fingerprinting, and measuring its integrity. Only apps installed on the device can access their secure world. Our framework includes two types of apps: "honest" and "arbitrary". Honest apps are explicitly modelled as a Tamarin process and have a fixed, known fingerprint. In contrast, arbitrary apps may be controlled by attackers. These apps can still access the secure world but cannot control their fingerprint. Therefore, they cannot trick the secure world into attesting them as an honest app.

The framework presented in this work verifies the overall security of attestation platforms. Additionally, it can check general security properties of the attestation platforms (e.g. if there is a way for any app to attest using Samsung Knox V2 incorrectly) and check the security properties of particular apps that use an attestation platform (e.g. does the design of an app, which uses Google SafetyNet, keep a particular value secure). Therefore, developers can use this framework to evaluate the security of their apps.

The security assumptions made during the design of remote attestation platforms are often subtle and sometimes not stated. For example, all frameworks implicitly assume that the devices cannot be rooted at runtime via, for instance, a bug in the Android kernel. Another important assumption is that unlocking the bootloader of an Android phone will wipe the apps installed on it [26]. We investigated and stated these assumptions and included them in our attacker model.

Our analysis of the platforms reveals that Samsung's Knox V2 attestation fails to satisfy the device integrity security property, allowing an arbitrary app on a tampered device to relay an attestation statement from a second unrooted device and pass this off as its own. While studying the security of Android Key Attestation, we found that the challenge phase is missing in the official recommended practice of the protocol. This issue does not guarantee the freshness of the attestation. This means that an app can return an arbitrarily old attestation statement to any challenge, which may not accurately reflect the device's current state. While the only default fresh values in the attestation are system time and the public key, both cannot guarantee the freshness of the statement. This is because the system time is an attacker-controlled value and not part of the secure world, while the key is not a known challenge value by the developer. In summary, our main contributions are:

- We analysed remote attestation protocols and developed an attacker model that captures their assumptions.
- We performed symbolic verification of common remote attestation protocols, namely Samsung Knox attestation V2 and V3, Google Android Key Attestation, and Google SafetyNet, using the Tamarin Prover, leading to a framework that others can use to check the security properties of apps that use attestation platforms. We showed this by modelling attested key exchange and code protection protocols.
- Based on these models, we showed that the device integrity check in Knox V2 remote attestation is flawed. We also demonstrated a freshness issue in the recommended use of Android Key Attestation. Both issues were proved by symbolic verification and not practically.
- We employed the models to present a real-world case study on code protection for Android apps, which stops attackers that cannot root the device at runtime without rebooting.

We provide all our open source artefacts, symbolic models at `https://akaldoseri.github.io/modelling_android_ra/`

Responsible disclosure: We reported the issue of Samsung attestation V2 to Samsung in August 2020. Samsung confirmed the issue theoretically and stated that it had been fixed in Samsung attestation V3, which we can confirm with our models. Samsung requested a proof-of-concept as part of their reporting process, and we could not create this because the Knox SDK is only available under a non-disclosure agreement. In a follow-up discussion, Samsung informed us they were deprecating V2 in the latest OS and that only V3 can be used with Android 13 (released in August 2022).

We reported the issue in Android Key Attestation to Google in November 2021, who have accepted the issues and responded that they have fixed the Key Attestation documentation, which will be available in a future release. The issue can be tracked on the Android public tracker via `https://issuetracker.google.com/205589624`. As at March 2023, the documentation had not yet been updated.

5.2 Background

Android device architecture: Android devices consist of two worlds: the *normal world* (untrusted) and the *secure world* (trusted). The Android OS and mobile apps operate in the normal world. They have access to the majority of the device's resources (e.g. display, storage and sensors). The secure world offers a Trusted Execution Environment (TEE) for Android devices [34]. It provides secure cryptographic operations offered to trusted apps (Trustlets). This enables data-sensitive services to operate securely in Android devices, such as NFC payments and fingerprint authentication.

Tampering with Android applications: Because mobile apps are considered untrusted entities, they are installed in the normal world (Android OS). Android enforces app signature verification for each app prior to installing it. This policy ensures that apps must be signed by a key to be installable. The app signature is generated by signing the app code base, its resources and the public key certificate of the signing key. The signature and the certificate are attached to the app for verification at installation time [14]. The Android package installer performs signature verification and assures the app's validity before installing it. It is advantageous to sign apps with the same certificate for data sharing and permissions access (e.g. system apps utilise this feature to access high-level permissions and restrict them from third-party apps).

The signature verification process does not authenticate the apps (*i.e.* verifying the developer's ownership of the app). Thus it allows stripping the apps' signatures and certificates to change their content, re-sign them, and re-install them on devices. We refer to this practice as "application repacking" or "application tampering". Furthermore, Android stores mobile apps in a public directory on the data partition, making the repacking process a core challenge to application integrity checks and anti-repacking solutions. Because of this, adversaries not only can extract the apps' code, but they can also perform dynamic analysis and debug the apps. In contrast, trusted apps have tamper-resistance protection. They can be extracted on a compromised OS, but only signed versions by OEM keys are eligible to be installed.

Verified boot and device tampering: The bootloader is locked on the majority of OEM Android device types (e.g. Samsung and Google's Pixel). This prevents users from flashing arbitrary custom bootloader software or executables into device partitions. Nevertheless, OEM vendors can flash verified executables, such as Android OS, signed with OEM keys. The verified boot validates the authenticity and integrity of the flashed components, and it operates when the device boots up. It establishes a chain of trust starting with a hardware-protection root of trust and progressing through the bootloader to verify device partitions, including boot, system, vendor, and optionally OEM partitions [35].

The latest Android devices support unlocking the bootloader through the `UnlockOEM` system option [26]. This option allows flashing executables into them, regardless of their signature, which allows full control over their normal world (Android OS), their data, apps, and resources. We refer to this process as "device tampering" as it voids the integrity of OEM devices. This issue poses a crucial challenge to identify the status of these devices because tampered devices can behave as non-tampered devices. Therefore, to ensure detecting such a tampering, OEM devices keep track of the status of these devices and their bootloader. For instance, most Google-based devices keep track of the status of their bootloader and verified boot by storing them in tamper-resistant storage in the secure world [36]. Additionally, unlocking/locking devices' bootloader completely factory reset them [26]. This mechanism prevents adversaries from accessing the devices' data after unlocking/locking their bootloader. Samsung additionally uses a special one-time programmable warranty bit that is set when their devices have been tampered with. This disables data-sensitive apps and services [157].

Remote attestation: Remote attestation is a mechanism that provides an authentic, timely report for a third-party entity (known as a *challenger*) about the validity of the attested platform. [65]. It is based on a typical challenge-response protocol. The protocol begins with a mobile app that requests a measurement report from a trusted entity (e.g. a trustlet). The report includes measurements and information about the device's status and the requester app info, if applicable. In certain implementations, device information is saved in tamper-resistant storage in secure hardware (e.g. Android Key Attestation). The report is signed with a per-device private key and sent to a remote server to be verified for its authenticity and integrity via the requester app. Finally, the server decides to either trust the device and continue communicating with it or not [163, 155, 91].

On Android, there are several generic and vendor-specific implementations for remote attestation (e.g. Samsung Knox remote attestation [163, 155], Google SafetyNet [91] and Android Key Attestation [36]). Older versions of SafetyNet were software-based. Therefore, they were bypassable in a compromised OS [114, 55]. However, the recent versions rely on a hardware root of trust similar to Samsung Knox and Android Key Attestation.

Modelling with Tamarin and SAPiC: We used the Tamarin Prover [47, 128] and its front-end SAPiC [115] for modelling. Tamarin is a state-of-the-art tool used to verify the security properties of protocols in the symbolic modelling of cryptography (the Dolev-Yao model [71]). SAPiC allows modelling protocols in (a dialect of) the applied pi-calculus [1] and converts the processes specification into multiset rewrite rules (MSRs) that Tamarin can process. Everything that SAPiC does can be expressed in MSRs in Tamarin. However, SAPiC provides a convenient encoding (e.g., for locks, reliable channels, and state handling). Security properties are expressed as first-order logic formulas. Then, Tamarin determines if the protocol expressed as MSRs satisfies such properties.

Dolev-Yao adversaries have absolute control over the public channel: they are free to intercept, forward, delay, drop, rearrange the order or modify (via public function symbols) any message in that channel. The public channel represents communication over insecure networks, and it is expressed by the SAPiC constructs `Out(`m`)` and `In(`m`)` for sending and receiving messages to/from the public channel. There are several alternatives to model communications over private channels. We have opted to use the SAPiC construct that allows giving access to Tamarin MSRs directly:

$[\ ] \mathrel{-\![}\ ]\!\!\rightarrow [\mathrm{SEND}(secret)]$ for sending a message and $[\mathrm{SEND}(secret)] \mathrel{-\![}\ ]\!\!\rightarrow [\ ]$ for receiving it. For further details about Tamarin and the SAPiC syntax, please see appendix A or refer to [47, 115].

5.3 Related work

Several solutions have been proposed to ensure the integrity of applications and Android devices. We categorised them into software-based and hardware-based techniques.

Software-based device and app integrity techniques: Berlato and Ceccato performed a survey for anti-debugging and anti-tampering techniques for mobile apps. The techniques presented are mainly software-based, which included emulator detection, dynamic analysis framework detection, and debugger detection [49]. Additional device tampering techniques are introduced in [176]. Other solutions rely on app hardening to hinder tampering. For instance, Android Studio uses ProGuard and R8 (in recent Gradle versions) to obfuscate mobile apps by renaming parts of the app's source code. Such an operation complicates apps modification, making reverse engineering them a difficult task [32]. DexGuard is based on ProGuard with some enhancements, which include code encryption [95]. Yuxue Piao et al. found that DexGuard's code encryption is hard-coded within the app [196]. AppIS [174] aims to protect apps against modification by inserting "guards" at random locations throughout the code and checking the hash of certain regions of the apps. Removing the guards individually and repacking the mobile app is difficult since the guards' locations are dynamically changed in each run. Finally, OWASP has suggested using a combination of different (imperfect) software-based approaches to frustrate adversary attempts to tamper with apps [146].

The major drawbacks of these software-based methods are related to the core design issues of Android (app tampering and device tampering), which are discussed in section 5.2. The proposed solutions are based on querying a fundamentally untrusted host (the Android OS) about its status and assessing the result within the app. However, an untrusted host might be compromised and provide an inaccurate status (e.g. by overriding relevant methods with frameworks such as Xposed [153], Cydia Substrate [167] or Frida [149]). Furthermore, as mentioned in section 5.2, in-app checks may be removed from the app prior to their execution.

Hardware-based device and app integrity techniques: Hardware-based techniques offer more assurance than software-based techniques. However, because they rely on attestation, they can only inform developers about the status of devices at the time of the request, which may be altered if the devices are compromised later. Definitionally, attestation only guarantees that the device was in a trustworthy state at some point between the attestation request and the verification of the attestation report. In such a circumstance, continuous detection is required. As discussed in section 5.2, these solutions (*i.e.* Google SafetyNet, Samsung Knox remote attestation and Android Key Attestation) rely on trusted components in the secure world (trusted apps) to assess the integrity of the devices and apps that they run. Kozyrakis and Census lab analysed the security of SafetyNet (software-based) versions and showed how to bypass its protections [114, 55]. SafetyNet has introduced hardware measurements to overcome these issues. Ibrahim et al. [100] evaluate the misuse of SafetyNet on mobile apps. Nevertheless, the study did not find any weaknesses in SafetyNet's core implementation. Samsung Knox remote attestation V2 and V3 are similar systems that ensure device integrity of Samsung devices and, for V3 only, the apps. Finally, Android Key Attestation is an Android technique that uses the Android Keymaster to attest a hardware key, in addition to providing application and device integrity.

Modelling remote attestation: Fotiadis et al. presented a symbolic abstraction for TPM-based remote attestation protocols to verify the integrity of network-attached devices (e.g. routers) [80]. Their work only looked at the attestation of devices and did not consider individual or attacker apps running on the same device as honest apps.

De Oliveira Nunes et al. discussed a design and verification for a hybrid (hardware/software) remote attestation protocol for embedded devices named VRASED [144]. They concluded that software-based approaches (e.g. Viper [120], and Pioneer [171]) are not suitable for such a case. Based on that work, de Oliveira Nunes et al. extended VRASED to include additional functionality (e.g. software updates). Similarly, for embedded devices, they focused on a more specific problem in remote attestation, which was time-of-check to time-of-use issues [70].

Jacomme et al. [106] presented a reporting capability for SAPiC to create and verify a cryptographic report in an isolated execution environment IEE (e.g. Intel SGX and ARM TrustZone). Processes modelling IEEs are given a unique identifier (called a *location*). These processes can produce reports bound to the named location. Then, an external party can verify that a given report was produced inside a given location. They demonstrated the feasibility of their approach through several case studies, including attested computation and the OTP protocol.

Our work uses this reporting mechanism but includes a framework to allow the attestation of apps outside the trusted computing base, the hardware measurements and rooted devices that allow the attacker to give false information to the trusted hardware. Our framework models the installation of multiple applications on the same device, including both honest apps and arbitrary/attacker apps. These extensions make it possible for us to model leading attestation platforms and analyse their security.

Unlike our work, Jacomme et al.'s [106] framework was aimed at modelling IEEs where the code runs inside the trusted hardware. Such hardware-based solutions (e.g. Intel SGX and ARM TrustZone) are promising. However, neither Intel SGX nor ARM TrustZone is fully accessible to third-party Android applications. VRASED [144] considered the architecture of low-level IoT devices (*i.e.* the MSP430 microcontroller). In contrast, we considered Android architecture without modifying its system, hardware level services and features.

Code protection: In section 5.7, we demonstrate the effectiveness of the modelled protocols' framework by solving a real-world problem, namely the app's code protection. The problem is challenging because Android OS lacks protection for apps' source code. Over the years, several studies have proposed source code protection solutions, and this section discusses them while pointing out their drawbacks.

First, Faruki et al. [78] conducted a survey illustrating different code protection techniques used by malwares. The techniques are varied like obfuscation, encryption, stenography and packaging using either custom tools or off-the-shelf tools like Progaurd, DexGuard and APKProtect packer. Among the obfuscation techniques is DexPro, which is a byte-level obfuscator for Android apps that obfuscates the program control flow by inserting opaque predicates before the return instruction of function calls [199]. Such complexity makes it harder for an attacker to trace protected calls. Kim et al. [111] enhanced Android packers tools Bangcle, Ijiami and Liapp, to protect multidexing apps. They do this by decrypting dex files of the app at runtime, performing a kind of dynamic code loading, which is widely used in packer tools.

The dynamic code loading approach relies on excluding partial parts of the application's source code from the application and safeguarding them in a particular place (e.g. remote server or encrypted locally). Then, these parts are retrieved at application runtime. Tanner et al. proposed a repacking protection architecture that verifies the application integrity at runtime and decrypts encrypted bytecode sections of an app using a derived key at runtime [179]. Utilising remote features, Google dynamic delivery is based on dynamic code loading. It allows certain app functionality to be downloaded conditionally or on-demand by splitting a mobile app into a base module and feature modules [8]. The base is the application's core, and it may conditionally request feature modules from Google servers.

None of these solutions change the Android OS architecture guarantees. Android OS only permits mobile applications to operate in the untrusted section of the device, allowing static and dynamic analysis techniques (e.g. reverse engineering, applications repacking and debugging) to uncover their source code [139, 78]. Overall these proposed solutions are software-based and similar to the techniques addressed at the beginning of section 5.3. They also share similar limitations as they increase the difficulty of apps' reverse-engineering but do not prevent it. Therefore, in section 5.7, we propose a hardware-based code protection protocol that overcomes these limitations and provides trust level protection using trusted hardware.

5.4 An attacker model for remote attestation platforms

The security assumptions made during the design of remote attestation platforms are often subtle and sometimes not stated. In this section, we analyse the design of the attestation systems and identify these assumptions. Then, we state our attacker model, which we use in the formal modelling section.

Design assumptions: We considered the following assumptions to model remote attestation protocols. These assumptions are based on the behaviour of Android OS and the remote attestation protocols' public documentation. First, in terms of tampering with devices (e.g. rooting), it is assumed that tampering can only occur by unlocking the bootloader when the device is powered off [26]. We are not considering tampering to occur at run time as this could result in leaking the data of a previously locked device. Additionally, changing the bootloader's status (e.g. unlock/lock) will factory reset the device [26]. Compromising a Samsung-based device will set a special Knox warranty bit and trigger appropriate reactions. It will block access to any keys stored in TrustZone and functions that rely on Knox security [157, 158].

Second, we assumed that application tampering is feasible, which includes supplying malicious apps and repacking apps. Third, we assumed that the hardware root of trust that manages the attestation is secured from all software-based attacks and cannot be broken by hardware-based attacks. Finally, regarding network communication involving attested apps, the official documentation states that the protocols should run over a secure channel like HTTPS [156, 91]. However, this will not prevent a local adversary from obtaining messages before sending/receiving them through the network.

These are strong assumptions, which may not always hold. For instance, a vulnerability in the Android kernel which would allow rooting a device without restarting, and implementation attacks (e.g. fault injection [57]) may be able to extract keys from the trusted hardware. However, these are the assumptions on which the attestation platforms are built, so we included them in our model.

The effect of extracting keys from the secure world would completely remove any protection an attestation platform offers. Being able to root a device while it is running would let an attacker attest a non-tampered device and then root it. So, while less powerful, this could still lead to a complete compromise of a single instance of an app. Unlocking the device without a factory reset would allow an attacker to learn long-term secrets stored on the device; however, a well-designed app that always attested the device and did not store secrets in long-term memory might still be secure. In terms of Samsung devices, Samsung tamper detection relies on Real-time Kernel Protection (RKP), which monitors the integrity of the kernel [159]. However, vulnerabilities that trick the kernel into changing its memory are beyond RKP protection [41]. In such a case, tampering will not be detected and may cause a similar impact to compromising the Android kernel addressed above.

Attacker model: Since these protocols ensure the integrity of devices and applications, we considered a threat model consisting of two adversaries: physical and network adversaries. The physical adversary has access to devices and can tamper with them (i.e., unlock their bootloader and root them), tamper with apps (*i.e.* has access to mobile apps) including installing honest apps or their own repacked apps as addressed in section 5.2, and communicate with the attestation endpoints (e.g. developers servers). For the network adversary, we consider the Delov-Yao adversary that can intercept messages in the network [71]. Honest apps can be retrieved from secure channels (*i.e.* representing a trust source like an honest app' developer or an app marketplace). On the other hand, an adversary can intercept network traffic in a tampered device or tampered app before sending them to developers' servers. The adversary aims to bypass either the app integrity check, device integrity, or both to continue communicating with the developer server to retrieve sensitive information that should only be obtained by verified, attested devices and apps.

5.5 Modelling remote attestation platforms

We considered all Android device-specific remote attestation protocols that rely on hardware-level protection namely: Google SafetyNet (Software and hardware), Samsung Knox remote attestation (V2 and V3), and Android Key Attestation. Google's software-based SafetyNet was included to ensure our models can detect known software-based issues.

We chose the Tamarin Prover and its process calculus SAPiC to model the protocols. Its syntax allowed us to model protocols as sets of rules and processes. It supports falsification and unbounded verification of security properties expressed as lemmas [115].

Most of the attestation protocols take a similar flow. Five parties are involved in the attestation process, an app, an attestation's client, an attestation's trusted app, a developer server and an attestation server. Here, we describe these protocols while highlighting their differences. Their technical details are addressed in their formal models.

5.5.1 Modelling SafetyNet remote attestation protocol

The SafetyNet remote attestation protocol flow is illustrated in Figure 5.1. The setup phase, highlighted in the boxes with dotted outlines in Figure 5.1, represents the keys setup and the creation of an app that utilises SafetyNet remote attestation protocol.

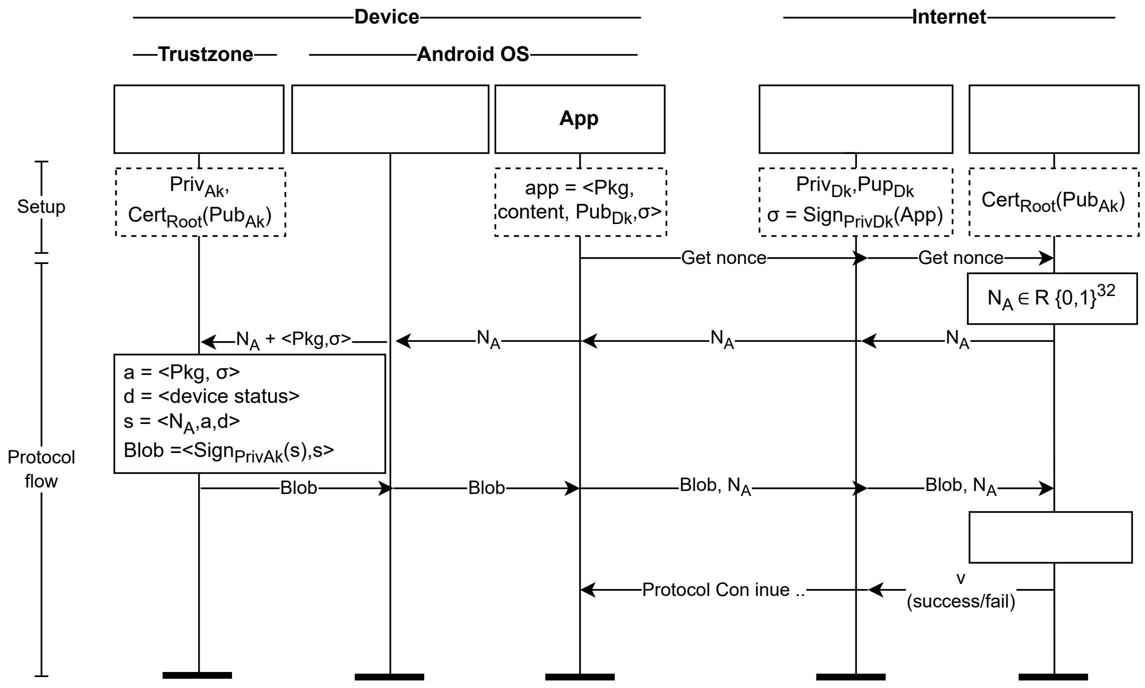


Figure 5.1: Flow of SafetyNet remote attestation protocol based on [91]

Setup phase: Samsung's and Google's documentation does not detail how the setup is performed. Therefore, we abstracted the process based on our best understanding of the available source code and documentation [157, 91, 163] and generalised it for all the protocols. It goes as follows:

Each device contains an attestation key based on a key pair certificate (`PrivAk` and `CertRoot(PubAk)`). The certificate is signed with a root certificate available on the attestation server. Hence, the attestation server can verify data signed by the attestation's trusted app.

The developer creates an `app`, which has source code (as binary code) and data resources, which are denoted by `content`, and package name `Pkg`. The developer signs the application (including its source code and resources) using its developer key `PrivDk`. The signing process attaches the public certificate of the developer key `PubDk` to the app to create the application signature σ. The Android package installer can verify the application signature when installing it on the device.

Protocol flow: The protocol runs as follows:

- Mobile apps request a nonce `Na` from the developer server.
- `Na` can either be generated by the developer or attestation server. Its purpose is to ensure the freshness of the attestation reports' content (*i.e.* device integrity and application integrity).
- `Na` is transferred to the app, which passes it to the attestation client on the device (Google Play service in SafetyNet and Attestation agent in Knox). The attestation client requests from the attestation trusted app in the TrustZone to initiate the attestation.
- The attestation-trusted app creates a report `Blob` containing the `Na`, device integrity information `deviceStatus d` and, app integrity information `a` (*i.e.* package name `Pkg` and the application's signature σ). Then, it signs them with an attestation key `PrivAk`.

- The signed `Blob` is sent to the attestation server through the app and the developer server, which verifies its integrity.
- The attestation server verifies the `Blob`. Then, it replies to the developer server with a verdict `V` indicating the verification result, app status and device status.
- Finally, the developer server can verify the app's signature σ and the `Na` using the received verdict.

5.5.2 Modelling Samsung Knox attestation V2 remote attestation

Figure 5.2 illustrates the flow of Samsung Knox attestation V2 protocol. The protocol takes a similar flow to SafetyNet with an exception to the `Blob` content. The `Blob` in Knox V2 lacks app information.

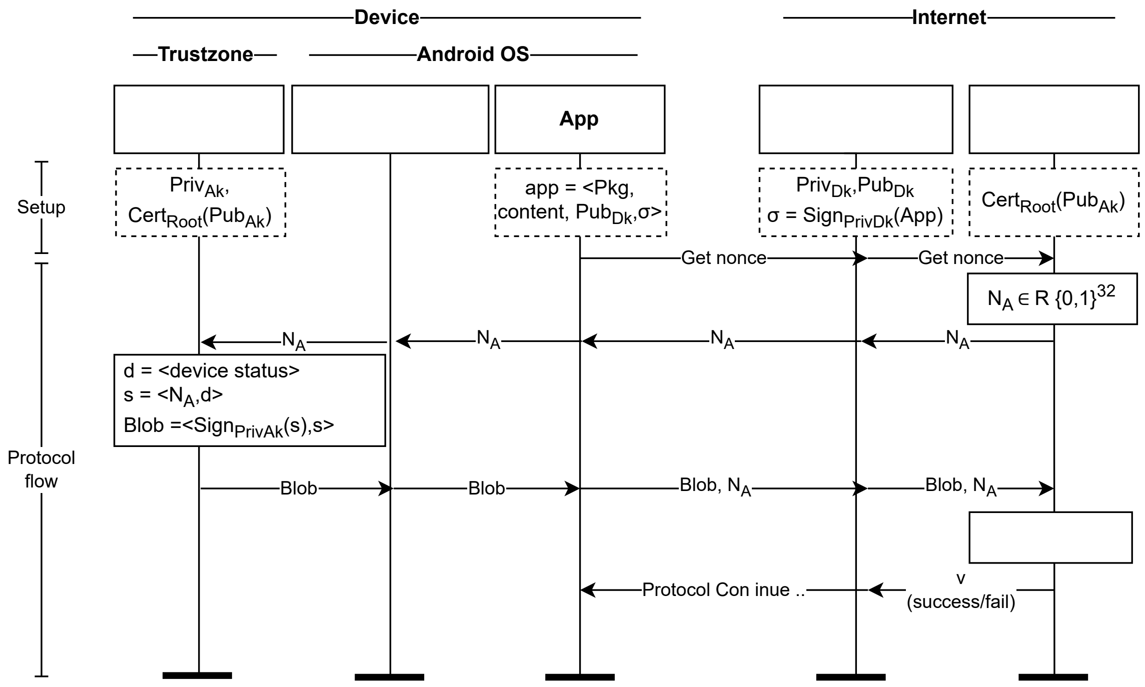


Figure 5.2: Flow of Samsung Knox remote attestation protocol V2 based on [163]

5.5.3 Modelling Samsung Knox attestation V3 remote attestation

The flow of Samsung Knox attestation V3 (illustrated in Figure 5.3) is also very similar to SafetyNet. The only difference is that Knox V3 includes an additional step after creating the `Blob`. In Knox V3, the signed `Blob` is sent directly from the attestation client to the attestation server in response to a Unique Id (UID). The `UID` instead is forwarded to the attestation server. Then, the application forwards the `UID` to the developer server. After that, The developer server communicates with the attestation server to verify the `Blob` by supplying the `UID`. After the verification, the attestation server replies to the developer server with a verdict `V` indicating the verification result, app status and device status.

5.5.4 Modelling Android Key Attestation

Key Attestation attests key pairs in addition to device and app integrity. It relies on certificate chain verification rather than an attestation server for verification. Therefore, the developer undertakes the verification process, and it is crucial to validate the entire certificate chain.

The Key Attestation protocol flow is depicted in Figure 5.4, based on Google's documentation [36]. It starts with the key setup phase and app installation. Application installation and its notation are identical to the previous model.

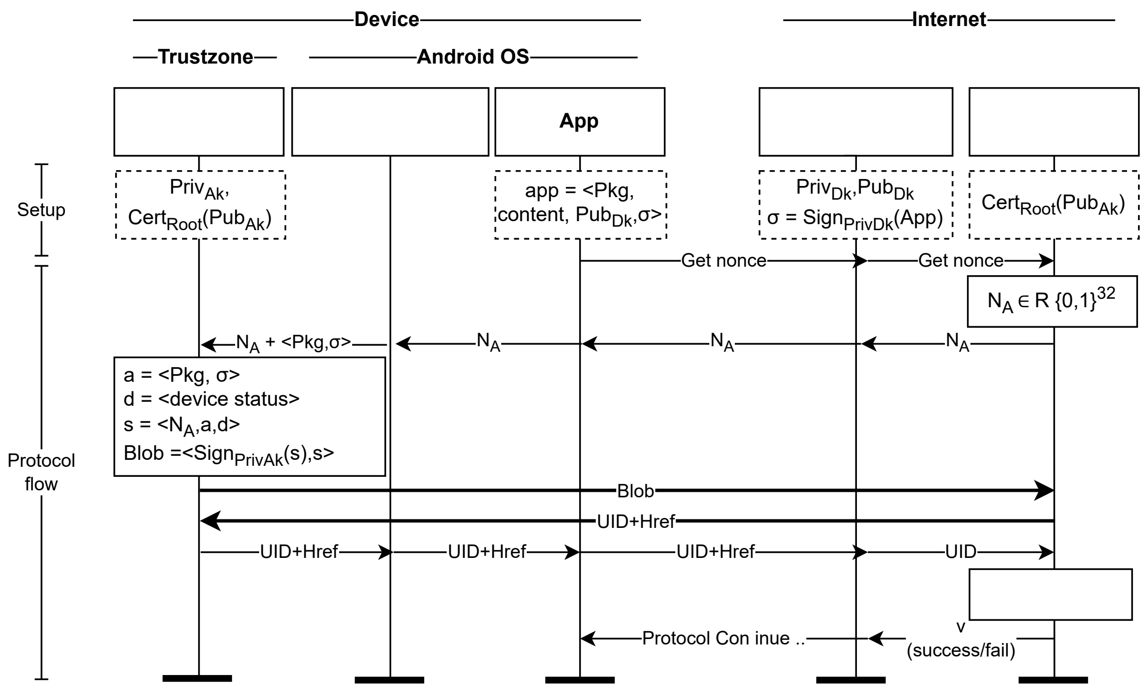


Figure 5.3: Flow of Samsung Knox remote attestation protocol V3 based on [155]

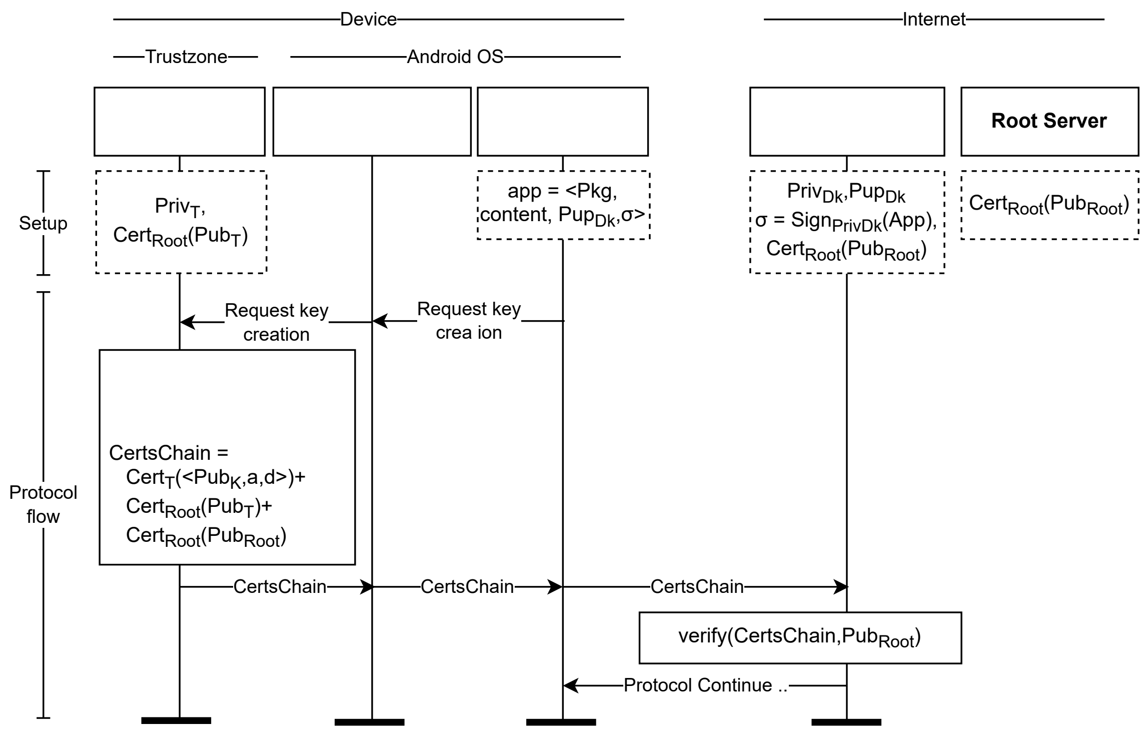

Device
Internet
Trustzone
Android OS
Root Server
Setup
$Priv_T$, $Cert_{Root}(Pub_T)$
app = <Pkg, content, Pup_{Dk},σ>
$Priv_{Dk}$,Pup_{Dk} σ = $Sign_{PrivDk}$(App), $Cert_{Root}(Pub_{Root})$
$Cert_{Root}(Pub_{Root})$
Request key creation
Request key crea ion
CertsChain = $Cert_T$(<Pub_K,a,d>)+ $Cert_{Root}(Pub_T)$+ $Cert_{Root}(Pub_{Root})$
Protocol flow
CertsChain
CertsChain
CertsChain
verify(CertsChain,Pub_{Root})
Protocol Continue ..

Figure 5.4: Android Key Attestation protocol based on [36]

Setup phase: Based on the available documentation and analysis of exported certificates from Android devices [25, 36], we found that the Keymaster is a trusted app available in the secure world that generates key certificates. It has a unique per-device certificate key `PrivT` and `CertRoot(PubT)`, which is signed with Google root certificate `CertRoot(PubRoot)` to form a certificate chain. Any key generated by `PrivT` will have the full certificate chain of the Keymaster and Google root certificate, thus ensuring that the key created in the trusted hardware can be publicly verified using the Google root public certificate `CertRoot(PubRoot)`.

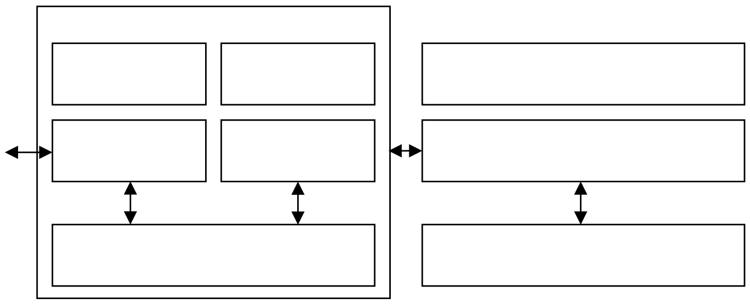


Figure 5.5: An overview of the modelling framework: each box represents a process in our SAPiC model

Protocol flow: The protocol's flow proceeds as follows:

- The mobile app requests a key pair certificate from the Android Keystore `PrivK,CertT(<PubK,a, d>)`.
- The Android Keystore requests that the Android Keymaster generates a key pair certificate. The certificate is signed by a TEE's key (or optionally a Strong box's secure element key Google Pixel phones only) `PrivT`. The certificate is generated in the form of a certificate chain `CertsChain` that includes the certificate itself `CertT(<PubK,a,d>)`, TEE certificate `CertRoot(PrivT)` and Google root certificate `CertRoot(PubRoot)`. Device `d` and app status information `a` is added to the attribute section of the generated certificate. Unlike previous protocols, the app signature is not included in the app status information, but the app's public certificate is.
- The mobile app can retrieve the chain certificate `CertT(<PubK,a,d>)` without the private key. The private key never leaves the secure hardware but can be used for cryptographic operations via the Keymaster and the Android Keystore. Then, the app sends the certificate chain `CertsChain` to the developer server for verification.
- Finally, the developer server verifies the certificate chain `CertsChain`. The verification process includes verifying `CertT(<PubK,a,d>)` and `CertRoot(PrivT)` and then the root certificate `CertRoot (PubRoot)` using a public version to ensure they are identical. The information about the device `d` and the app `a` can then be extracted from the certificate's attribute section to verify the app certificate and identify the device status to either continue communicating with the app or not.

5.5.5 Modelling Android OS and the adversary

We modelled a minimalist Android OS functionality that includes the app life cycle, and the required cryptographic operations. An overview of our modelling framework is given in Figure 5.5. Each box in this figure represents a process in our SAPiC model. The "App code" and "Dev Server: Protocol for App" boxes model a protocol that uses the "Attestation Server" process. The "SecureWorld" process models the trusted hardware on the device and works with the app installation and "Dev Server Set Up" processes, which ensure the secure world can correctly measure the device and apps. The "Attacker App Installer" and stub allow for arbitrary attacker apps to run on the same device as the honest app. In this section, we describe the app processes that use this framework.

Modelling Android Applications: An Android application consists of a package name, source code, resources and a public key certificate of the app's developer. A package name is an identifier to distinguish an app within a device. The source code and resources shape the content and logic of the app, and we model these in Tamarin as constants (e.g., "App1PackageName" and "App1Content" in the example below). The user of our framework must ensure that the same values are used in the Dev Server and app installation process. The app code functionality is modelled as a SAPiC process.

The Dev Server Set Up process signs the package name, contents string and developer certificate with its signing key `advk` to obtain the application signature `appSignature`. This is packed together with the signed components to make the `app`. Then, the app is outputted on a public channel, and we use a SAPiC private fact "App_Published" to allow authentic installation(e.g. via Android marketplace or an honest developer).

```
let DevServer =
  //creating and publishing the  application
  new ~advk; // Developer server signing key
  new ~devId;// Developer Id

  // Signing the app
  let appSignature = sign(~advk,
    ⟨'App1PackageName','App1Content',pk(~advk)⟩) in
  //packing the app
  let app = ⟨'App1PackageName','App1Content',pk(~advk),
    appSignature⟩ in
  event App_Created('App1PackageName',appSignature);

  // Send the app publicly
  out(app);
  // Send the app privately (e.g., to Android marketplace)
  [ ] −[ ]→ [!App_Published(~devId,app)];
```

For installing apps onto a device, we considered two forms: honest installation and arbitrary installation. Honest App installation refers to installing apps that are explicitly modelled in the Tamarin code. This could be, for instance, an app from a known developer that we want to analyse. The installation starts with the "Device" process . It creates a new device ID, sets up a secure world process for this device ID, and starts the app installation process. The device ID is a model-only reference number to distinguish multiple devices within each run. The "Honest App Installation" process uses the `App_published` channel to receive the app, checks its signature (using pattern matching) and starts the app running.

```
let HonestAppInstalltion =
  [!App_Published(devId,⟨'App1PackageName','App1Content',
  pk(advk), sign(advk, ⟨'App1PackageName','App1Content',
  pk(advk)⟩)⟩)] −[ ]→ [ ];
  !AppCode
```

The "app code" process now has the device ID, package name and contents string of the app, and these are then used when the process calls to the secure world.

Arbitrary installation (Attacker apps): Following our attacker model, we wanted to allow arbitrary attacker apps to run on the device and make calls to the secure world. The attacker can craft its app using any package name and key, sign the app and start the attestation using the "AttackProcess" process. The attacker process receives any input, forwards it to the secure world, and then broadcasts the reply. This allows the attacker to use the secure world however they wish.

```
let ArbitraryAppInstallation =
 in(⟨packagename, devKey⟩);
 new ~content;  //content is different from any honest app
 let appSignature = sign(devKey, ⟨packagename,~content,
   pk(devKey)⟩) in
 let app = ⟨packagename,~content,pk(devKey),appSignature⟩ in
 out(app);
 !AttackerProcess
```

Attackers app's content (Fresh content or attacker-controlled content): The content of the attacker app can be modelled as fresh because the attacker might use an honest app code and include its own code. In this case, the secure world on a non-tampered device should still be able to correctly measure these devices (i.e., an attacker app cannot run its custom functionality and be measured with the same app contents string as an honest app). Therefore, the content needs to be fresh to include the attacker functionality in a non-tampered device. If the attacker chooses not to include its functionality, then the installation will be like an honest installation. In a tampered device, the attacker can use any content bypassing these system-level restrictions, as in section 5.5.6. So, the attacker app installation process lets the attacker pick any package name and signing key but uses a fresh name to represent the app contents.

Alternatively, it is possible to assume that the content needs to be controlled by the attacker in the non-tampered device. This assumption excludes the attacker's functionality that changes the content. In this case, the process will allow the attacker to choose the content from the public channel (e.g. it could be from an honest app) as follow:

```
let ArbitraryAppInstallation =
 in(⟨packagename, content, devKey⟩);
 let appSignature = sign(devKey, ⟨packagename,content,
   pk(devKey)⟩) in
 let app = ⟨packagename,content,pk(devKey),appSignature⟩ in
 out(app);
 !AttackerProcess
```

These two approaches might result in different outcomes. Therefore, We verified both and found they showed the same results addressed in section 5.6. The result's justification lies in the source of freshness used to create the "appSignature" of the measurement report. In the first approach, because of the attacker's functionality, the content is always fresh. Therefore, The attacker app cannot have the same signature as an honest app. In the second approach, the attacker could not obtain the developer keys of the honest developers, and, therefore, it cannot forge an app with "appSignature" that matches the honest apps.

Modelling network: The protocols' documentation recommends using a secure channel (e.g. TLS) when communicating with developer servers for attestation. MSR facts can be used to model sending messages through a secure channel as follows: $[\ ] \dashv [\ ]\rightarrow [\text{Blob\_TLS\_Ch}(blob)]$ for sending the attestation report `blob`, and $[\text{Blob\_TLS\_Ch}(blob)] \dashv [\ ]\rightarrow [\ ]$ for receiving the attestation report `blob`. However, the network adversary cannot send/receive messages to/from the developer servers through the secure channel with such an approach. Therefore, we added two processes to allow the adversary to communicate with the developer servers via a secure channel as follows:

```
let Attacker1 = [Nonce_TLS_Ch(devId,nonce)]-[ ]->[Out(nonce)]
let Attacker2 = [In(blob)]-[ ]->[Blob_TLS_Ch(blob)];
```

Here, the adversary can receive a nonce from the developer server over the secure channel and output it to the public channel. Similarly, it can send a blob to the developer server over the secure channel. The attacker utilises `In(blob)` and `Out(nonce)` to receive and send messages to the public channel. These allow the adversary to craft its own nonce and attestation report `blob` or use a generated one from a tampered device or app.

5.5.6 Security measurement

The secure world measures the device's status and creates an attestation report `blob`; in Knox, Samsung relies on a warranty bit (a one-time not programmable bit), which indicates the status of the device. The bit is set when tampering is detected by the Runtime Kernel Protection (RKP), and the TrustZone-based Integrity Management Architecture (TIMA), which periodically monitors the kernel and certain device components [41, 159]. SafetyNet uses Compatibility Test Suite (CTS) profile match measurements to detect tampered devices. CTS is a set of unit tests that test the compatibility of various Android classes and components of an Android device, including signature checking for public Android APIs [17]. Key Attestation relies on the bootloader status (e.g. locked or unlocked) and the verified boot status that are stored in secure, tamper-resistant, hardware storage[36]. As stated in our attacker model, we assume these approaches work correctly to measure the device and app.

As seen in the following code, we generalise these measured properties and abstract them into two Tamarin terms: `hardwareMeasurement` and `softwareMeasurement`. On the one hand, `hardwareMeasurement` represents a hardware-based measurement that is obtained from a trusted source and stored in tamper-resistant storage (e.g. the Knox warranty bit and Bootloader status). On the other hand, `softwareMeasurement` refers to a measurement performed in the normal world within the operating system (e.g. software-based SafetyNet), and therefore it can be controlled by the attacker on a tampered device only. Hence, when it is used, we model this as public input in Tamarin.

The attacker's device rooting attempt is modelled by a public input (`in(status)`) at the start of the secure world process. After this, the secure world will run in either unlocked or locked mode. We do not model the unlocking of a locked device because unlocking, according to the specification, factory resets the device. Therefore, this is modelled as creating a newly rooted device. In software-based attestation systems, `softwareMeasurement` will be used for measurement. This value is supplied by an adversary-controlled input `customSoftwareMeasurement` when the device has been tampered with.

To create an attestation report `blob` for Samsung Knox and SafetyNet, we use the extended `report` function for creating cryptographic measurement reports in an IEE [106].

```
let SecureWorldTA =
( //Did the attacker unlock the bootloader?
 in(status);
 !(
    // Apps call to the secworld
    [SecureWorld_Ch_In(sessionID, deviceId,~swId,nonce
    ,packagename,content,appSignature)]-[ ]→ [ ];
  new ~atId; // attestation id
  event Attestating_App(~atId,packagename,appSignature,deviceId);

  if(status = 'unlockBootloader') then
   let hardwareMeasurement = 'invalid'  in
   //Software measurement is not used in HW attestation
   //in(customSoftwareMeasurement);
   //let softwareMeasurement = customSoftwareMeasurement in

   //On rooted devices nonce, packagename and appSig can be forged:
   in(⟨fnonce,fpackagename,fappSignature⟩)
      //Creating a measurement report
   let report =  report(⟨~atId,hardwareMeasurement,fnonce
        ,fpackagename,fappSignature⟩) in
      //Form a blob of measurement report and its signature
      // via the attestation key
      let blob = ⟨~atId, fnonce, hardwareMeasurement,fpackagename
        ,fappSignature, report,sign(report,~skAT)⟩ in
   event DeviceStatus(~atId,deviceId,hardwareMeasurement);
   //return attestation report to the app
   [ ]--[ ]->[ SecureWorld_Ch_Out(sessionID,blob) ]
  else
   let hardwareMeasurement = 'valid'  in
   let report = report(⟨~atId,nonce,hardwareMeasurement,
        packagename,appSignature⟩) in
   let blob = ⟨~atId,nonce,hardwareMeasurement,packagename,
        appSignature,report,sign(report,~skAT)⟩ in
   event DeviceStatus(~atId,deviceId,hardwareMeasurement);
   //return attestation report to the app
   [ ]--[ ]->[ SecureWorld_Ch_Out(sessionID,blob) ]
 )
 )@⟨'loc',pk(~skAT)⟩ //Indicating a trusted location.
```

This process creates a signed attestation report at a location (TEE) identified by the tuple including the secure world's public key ⟨'loc',`pk(~skAT)`⟩ using the `report()` function. The report contains `hardwareMeasurement`, `packageName` and `appSignature`. This report can be verified later by the attestation server using this particular secure world's public key. Note that the technical details on how the TEE cryptographically protects the report are abstracted away in SAPiC. We use the appropriate Tamarin predicates [47], allowing the adversary to create reports in any other untrusted location.

Key Attestation certificates: Key Attestation relies on certificates as attestation reports instead of signed Blobs. Therefore, we modelled an abstraction of the Internet X.509 Public Key Infrastructure (PKI) Certificate , based on RFC5280 [68]. We modelled `create_certificate/3` to create certificates given their information, their subject's public key and their issuer's private key. While, `verify_certificate/2` is used to verify certificates for a given subject certificate and an issuer certificate. We modelled the functions `get_public_key_certificate/1`, `get_signature_certificate/1`, `get_tbsInfo_certificate/1` to extract information from certificates, specifically the attribute section to retrieve bootloader status. We symbolically verified the PKI operations separately, considering different real-world scenarios, including self-signed certificates, forged certificate detection, chain certification verification, verification of certificates with extension data, and extraction of certificates' public keys.

5.5.7 Security properties

Below, we describe the security properties we require for the protocols, expressed as first-order logic formulas. The syntax `Event(...)`$@i$ denotes that `Event(...)` was executed at timepoint i. We assume a scenario where a developer verifies the integrity of an attested app running on a device. We aimed to ensure the security properties below:

Verification of device and app integrity for honest apps: This property ensures that an honest app's validation of an attestation report implies that the attestation was done correctly. The property states that for all "valid" attestation verdicts (`Verdict_app` event), they have a valid report generated at a TEE, valid device integrity and valid application integrity. An honest developer must have created the app (via `App_Created` event), it must be installed in a device (via `Application_Installed` event) and have been attested by the device, with a valid state (via `DeviceStatus` event).

$$\begin{aligned}
&\forall atId, blob, i.\ \texttt{Verdict\_app}(atId, blob, \text{`valid'}, \text{`valid'}, \text{`valid'})@i \Rightarrow \\
&\quad \exists deviceId, packageName, appSignature, a, b, c. \\
&\texttt{App\_Created}(packageName, appSignature)@a \wedge \\
&\texttt{Application\_Installed}(deviceId, packageName, appSignature)@b \\
&\wedge \texttt{DeviceStatus}(atId, deviceId, \text{`valid'})@c \\
&\wedge a < b \wedge b < c \wedge c < i.
\end{aligned}$$

Satisfying this property ensures that honest apps can perform a successful attestation that yields a valid verdict. However, if this property fails, the attestation report does not prove that the device is valid.

Attestation report secrecy: This property ensures that the adversary cannot learn/craft an attestation report blob that results in a valid verdict by any method, including tampering with apps and devices. It states that for all "valid" attestation verdicts (`Verdict_app` event), the adversary KU cannot learn their attestation report blob before it is validated.

$$\begin{aligned}
&\forall atId, blob, i.\ \texttt{Verdict\_app}(atId, blob, \text{`valid'}, \text{`valid'}, \text{`valid'})@i \Rightarrow \\
&\quad \neg(\exists k.\ KU(blob)@k \wedge k < i).
\end{aligned}$$

If this property fails and the attacker learns the attestation report before the verdict, then the attacker can use the attestation report to interact with the developer server and, for instance, obtain secret messages that should only be available for valid attested apps.

Attestation report (blob) uniqueness: This property ensures that each accepted genuine attestation report (blob/certificate) by the developer server is unique.

$$\forall \texttt{n}, i, j.\ \texttt{BlobAccepted}(\texttt{n})@i \wedge \texttt{BlobAccepted}(\texttt{n})@j \Rightarrow i = j.$$

If this property fails, an attacker could reuse an old attestation report, successfully completing attestation to the development server when a device is absent or has changed its state.

Attestation report (blob) recentness: This property checks that the attestation report (blob/certificate) was generated in response to the development server's request. The development server's request for, and acceptance of, the attestation report are tagged with a `requestID` to be matched. The property states that if the developer server accepts an attestation report, it must have been requested by the developer server before it was created on the device.

$$\begin{aligned}
&\forall requestID, blob, i.\ \texttt{RequestedBlobAccepted}(requestID, blob)@i \Rightarrow \\
&\quad \exists j, k.\ \texttt{BlobRequested}(requestID)@j \wedge \\
&\qquad \texttt{BlobCreated}(blob)@k \wedge (j < k) \wedge (k < i)).
\end{aligned}$$

Protocol	**Device and App integrity**	**Attestation Report Secrecy**	**Attestation Report Uniqueness**	**Attestation Report Recentness**
SafetyNet (Software-based)	✗	✗	✓	✓
SafetyNet (Hardware-based)	✓	✓	✓	✓
Knox Remote Attestation V2	✗	✗	✓	✓
Knox Remote Attestation V3	✓	✓	✓	✓
Key Attestation	✓	✓	✓	✗

Table 5.1: Satisfied security properties by existing remote attestation protocols.

Failure of this property would mean that attestation reports were not linked to the request for them, meaning that the developer might accept old, expired or compromised reports.

Reachability properties: To ensure that the model and all the processes finish, we add reachability properties identified with the "`Correctness`" prefix. These properties ensure all model's branches are executable. For example, the property that ensures that an honest app's process finishes is encoded as:

$$\exists i.\ \texttt{checked}(\text{`honestAppFinished'})@i.$$

Failure of these lemmas would indicate an error in the model or the protocol's design.

5.6 Discussion and results

Our symbolic verification results are illustrated in Table 5.1 and demonstrate that software-based SafetyNet failed to satisfy the device and app integrity properties due to an adversary being able to tamper with the device's software measurement to obtain the attestation report as addressed in [114, 55]. This results in falsifying the attestation report secrecy property as the adversary can obtain the report and attest themselves to obtain secret messages from the developer. Surprisingly, Knox V2 failed to meet the same properties. The model shows traces of an adversary using an arbitrary app to make an attestation and obtain a valid verdict. Because Knox V2 does not include the app contents in the attestation blob, an attacker (through the public network or on a rooted device) can relay a valid attestation `blob` from another app running on a non-tampered device and have the attestation server validate this. Technically, this convinces a developer server that the phone is not rooted when it is. Because Knox V2 attestation does not include app integrity, it cannot distinguish between honest and arbitrary apps. As described above, we disclosed this to Samsung, who confirmed the issue and we were the first to report this.

SafetyNet (hardware-based) and Knox V3 satisfied all the security properties. Both of these ensured device integrity and app integrity. Ensuring only device integrity (as Knox V2 does) or only app integrity (as in SafetyNet (software-based)) will result in compromising integrity.

Our model of standard use case of Android Key Attestation satisfied all security properties except the certificate/blob recentness property. Unlike the other protocols, the developer server does not generate a nonce for the attestation report, but the uniqueness of the report is still guaranteed due to the freshness of the subject key within the report.

Key Attestation uses the untrusted OS's device clock for certificate creation. We modelled this by allowing the secure world to receive the time on a public channel, and as this is the only source of freshness for the recommended use case, the attacker can send a time in the future to the secure world and so get a blob to attest a key it can use much later when it is no longer fresh, and when the key or device may have been compromised.

Therefore, a developer following the standard use case will be unaware of when the key was created and when the attested certificate with its integrity information was created. We reported this issue to Google (as described below). They have stated that they will update their recommended use case for Key Attestation.

Performance overhead: In terms of performance overhead, the Tamarin prover manages to prove all the lemmas in no more than 28 steps for SafetyNet and Knox for each lemma. Key Attestation took longer to prove (a maximum of 36 steps). The modelling of X.509 Public Key Certificate may have been the reason for the extra steps. Knox V2 achieves the shortest runtime of four minutes to verify all the lemmas. While Key Attestation was the longest with seven minutes. The runtime was measured on a laptop with 8 cores Intel i7 processor and 16 GB RAM running Ubuntu 22.04.1 LTS.

Mitigation: As a mitigation, Knox V2 requires binding the attested app to the attestation process similar to Knox V3 by including the app package name and its signature in the measurement report. We concluded that device-only remote attestation is not sufficient to ensure integrity, as an adversary could always tamper with the app and relay the attestation report from a non-tampered device. Samsung confirmed the issue and stated that it was fixed in Knox V3. However, as part of their reporting policy, they asked for a proof of concept to prove the attack was practical. Because Knox licence is not publicly available and requires a specific partner licence agreement, which Samsung would not provide to us, we could not provide them with a proof of concept. Therefore it remains an open issue.

For Key Attestation, when we modelled the protocol including a nonce from the developer server as an attribute in the generated certificate, we found that the certificate freshness security property holds. With a nonce in the attestation, the developer can verify when it was created and only accept fresh attestations.

This is not an issue with the Android Key Attestation protocol. Rather it is an issue in the guidance given to developers on how to use the `KeyGenParameterSpec.Builder` class for key creation. I.e., unlike the Knox V2 vulnerability, which corresponds to breached security lemma for the Knox V2 system interacting with an arbitrary attacker process. The Key Attestation vulnerability is found after reading Android recommended practice documentation of Android Key Attestation [36].

To overcome the issue, the nonce needs to be set via `setAttestationChallenge` method in Key Attestation as advised by [154]. Google confirmed this issue and said they would address it in the future release of their documentation.

5.7 Case study: Modelling a code protection protocol

Above, we used basic use cases to evaluate the security properties of remote attestation protocols. In this section, we show how our framework can be used to verify a more complex design that uses attestation.

Our case study discusses how to use attestation to provide code protection for apps. This study can benefit private organisation apps that run using a bring your own device (BYOD) model . The difficulty of the problem is due to the design of the application installation life-cycle in the Android OS. Android does not protect applications' source code. The Android platform security model does not consider the source code confidential as Android provides the APK Analyzer tool to decompile android apps [10]. Therefore, leaking the source code is feasible via static and dynamic analysis techniques. Previous studies discussed in section 5.3 proposed several solutions that provide limited protections that may be circumvented using conventional techniques (e.g. repack a mobile application or compromise a device). No code protection solution for mobile applications has been proven to properly prevent code leakage yet, leaving the feasibility of such a goal to be questioned.

Threat model For this issue, we considered a more aggressive threat model. In this model, we drop the use of secure channels (e.g. TLS) in the proposed protocol. Unlike the previous threat model, messages sent by honest apps and developers are sent through the public channel. This allows a Delov-Yao adversary to intercept, drop, modify and relay messages in the network [71] without relying on relay processes as before. Additionally, the physical adversary has access to devices and apps, and can tamper with them, including installing their own repacked apps.

Candidate selection To overcome the limitation of previously proposed code protection solutions, we relied on hardware remote attestation. Therefore, the suitable candidates based on our findings from section 5.6 were Knox V3, SafetyNet hardware-based and Key Attestation.

Key Attestation has potential since the attested key can only be used in the generated device. Moreover, having a per-device key can identify and authenticate messages between an app and its developer server. This creates a form of continuous integrity check (*i.e.* as long as the key is valid and in use, the integrity of their apps and devices are preserved). Such an option is not available by default in Knox V3 and SafetyNet hardware-based . Secondly, encrypting a secret with an attested key, binds the secret to the hardware root of trust. For all these reasons, we selected the Key Attestation protocol as a candidate for the code protection model.

Modelling code protection It is difficult to have full protection for the app source code because applications, in general, need to be installed in a device to operate. Therefore, following previous studies, we choose to perform partial code protection in which an application is divided into base code and protected code. The purpose of the base code is to fetch and load protected code at runtime conditionally. This approach has been addressed before as dynamic code loading and used in packer apps [78] locally and remotely by Google Dynamic delivery [8]. Our model differed from the previous studies that were based on software-based techniques, as we relied on the hardware root of trust (namely hardware-based attestation) to ensure the integrity of applications and their devices (1). We were considering a more aggressive adversary that has full control of devices, including installing apps and monitoring the network (2). Finally, we were not considering any custom changes to the Android OS. Our model considered Android OS policy with no changes to any system and hardware level services to ensure that it is applicable for practical implementation (3).

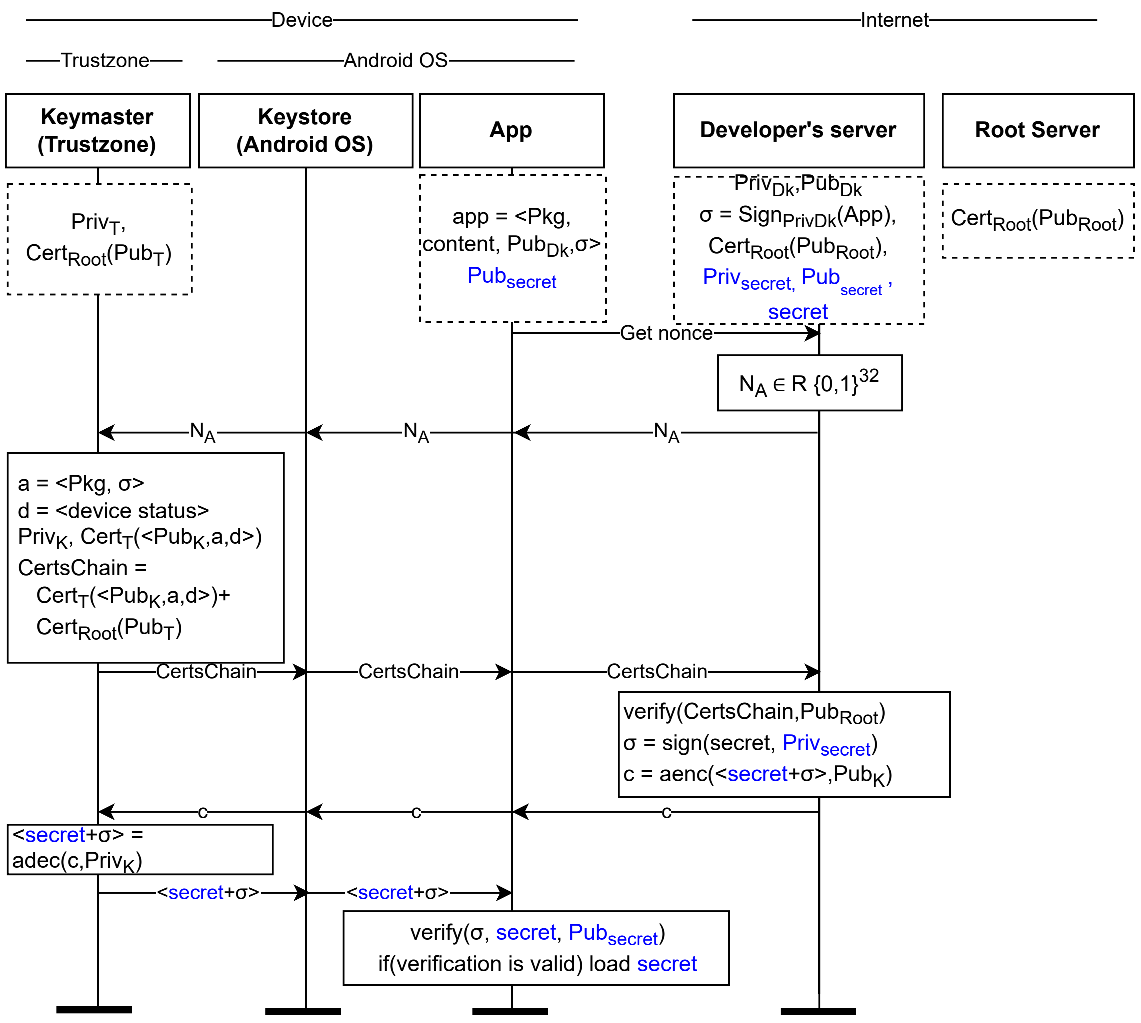

Device
Internet
Trustzone
Android OS
Keymaster (Trustzone)
Keystore (Android OS)
App
Developer's server
Root Server
$Priv_T$, $Cert_{Root}(Pub_T)$
app = <Pkg, content, Pub_{Dk},σ>
Pub_{secret}
$Priv_{Dk}$,Pub_{Dk}
σ = $Sign_{PrivDk}(App)$,
$Cert_{Root}(Pub_{Root})$,
$Priv_{secret}$, Pub_{secret} ,
secret
$Cert_{Root}(Pub_{Root})$
Get nonce
$N_A \in R \{0,1\}^{32}$
N_A
N_A
N_A
a = <Pkg, σ>
d = <device status>
$Priv_K$, $Cert_T(<Pub_K,a,d>)$
CertsChain =
$Cert_T(<Pub_K,a,d>)$+
$Cert_{Root}(Pub_T)$
CertsChain
CertsChain
CertsChain
verify(CertsChain,Pub_{Root})
σ = sign(secret, $Priv_{secret}$)
c = aenc(<secret+σ>,Pub_K)
c
c
c
<secret+σ> = adec(c,$Priv_K$)
<secret+σ>
<secret+σ>
verify(σ, secret, Pub_{secret})
if(verification is valid) load secret

Figure 5.6: Code protection protocol based on Android Key Attestation

Protocol Flow Figure 5.6 illustrates the flow of the code protection protocol. In the setup phase, first, the developers create secret source code. We modelled it as a fresh value (line 02). Then, the developers need to set up their apps to have a public verification key (`pk(secretSk)/ PubSecret)` as part of the application's content (line 07). This key is used in later steps to verify the authenticity and integrity of the retrieved secret source code.

```
01|let DevServerKA =
02| new ~secret; //generate secret
03| event SecretGenerated(~secret);
//generate secret 's verification key
04| new ~secretSk; out(pk(~secretSk));
//creating and publishing the application
05| new ~advk; // Dev server signing
06| let packagename = 'App1PackageName' in
07| let appContent = ⟨'App1Content', pk(~secretSk) ⟩ in
08| let appSignature = sign(
  |           ⟨packagename,appContent,pk(~advk)⟩,~advk)
09| let app = ⟨packagename,appContent,pk(~advk),appSignature⟩ in
10| out(app);
```

- The protocol starts with a request from the app to the developer server to get a nonce, which is passed from the developer server to the app to the Android Keymaster to initiate the attestation.
- The Keymaster creates a public key certificate `PubK` containing the nonce, app's information (*i.e.* package name and signature), and device status signed with the Keymaster certificate to form a chain certificate.
- The certificate chain is passed publicly to the developer server that verifies its content. The validation of the app's information is crucial, as it not only attests the application for tampering, but also ensures that the verification key `PubSecret` within the app's content never changes. Also, it ensures the authenticity of the secure world generated key.
- After verifying the certificate and its content, the developer server signs the secret source code with `PrivSecret` to make the signature σ. Together, σ and the `secret` are encrypted with the attested key `PubK` to make the cipher `c` which is sent out publicly.
- Once an app receives the cipher `c`, it is passed to the Keymaster for decryption. Only the device that owns `PrivK` will be able to decrypt the cipher to obtain both the `secret` and its signature σ.
- Finally, the app verifies the integrity and the authenticity of the `secret` using `PubSecret` and σ prior to loading it at runtime.

The cipher `c` can be stored locally in app-storage. While the `secret` can be decrypted, loaded at run time and cleared whenever it is not being used. The decryption process can operate locally afterwards. No further attestation is required because as long as the key is in use, the integrity of the device and the app should be preserved. Rooting the device by unlocking the bootloader should factory reset the device, deleting the generated keys and preventing access to the protected source code.

Security properties. For the security properties, we focused on code protection specific lemmas. We need to ensure that the following three lemmas are satisfied.

- Secret Validity / Correctness: The purpose of this property is to ensure that the loaded secret code must have been generated by an honest developer previously. It states that for all the received secrets to a valid device (*non-tampered device*) and an honest application, then this secret must have been generated by an honest developer at some time before.

$$\begin{aligned}&\forall secret, sessionID, j, k.\\&\texttt{SecretReceivedatDevice}(sessionID, secret, \text{'valid'})@j \wedge\\&\texttt{SecretReceivedAppStatus}(sessionID, secret, \text{'valid'})@k \Rightarrow\\&\quad \exists i.\ \texttt{SecretGenerated}(secret)@i \wedge (i < j) \wedge (j < k).\end{aligned}$$

- Secret secrecy: This property ensures the confidentiality of the secret code against tampered devices, arbitrary attacker apps, and network adversaries. It states that, for all secrets generated by an honest developer, the secrets must not be known or obtained by these adversaries.

$$\begin{aligned}
&\forall secret, i.\ \texttt{SecretGenerated}(secret)@i \Rightarrow\\
&\neg\big((\exists k.\ KU(secret)@k)\\
&\quad \vee (\exists sessionID, k.\\
&\quad\ \texttt{SecretReceivedAppStatus}(sessionID, secret, \text{`invalid'})@k)\\
&\quad \vee (\exists sessionID, k.\\
&\quad\ \texttt{SecretReceivedatDevice}(sessionID, secret, \text{`invalid'})@k)\big).
\end{aligned}$$

- Code injection: This property ensures that the code generated by an adversary cannot be loaded by an honest app running on a valid (non-tampered) device.

$$\begin{aligned}
&\forall secret, sessionID, i, k.\ \texttt{SecretReceivedatDevice}(sessionID,\\
&\quad secret, \text{`valid'})@i \wedge KU(secret)@k \Rightarrow\\
&\neg(\exists j.\ \texttt{SecretReceivedAppStatus}(sessionID, secret, \text{`valid'})@j).
\end{aligned}$$

Discussion, results and limitations The modelled protocol managed to verify all the security properties. Starting with the secret validity and secrecy properties, which ensure that no adversary can learn the honest developer's generated secret code. The verification's source code key prevents adversaries-generated code from injecting apps in a non-tampered device. The properties are true even without using a secure channel between the app and the developer server due to the use of Key Attestation. Considering these findings led us to model a variant of this protocol where instead of sending a secret code, the developer can send a secret decryption key to decrypt an encrypted, protected code within the app. The variant achieved the same protection level. We implemented this protocol and ran it on a locked Samsung device (A73), loading code that ran 10 sorting algorithms from [92]. Running the app 100 times, we found that it takes, on average 2.74 seconds to complete the whole protocol. In terms of limitations, while the protocol provides protection against strong adversaries that have not been considered by software-based techniques addressed in section 5.3 and the threat model of remote attestation protocols in section 5.4, it cannot hold when device tampering happens at runtime (e.g. via exploits in the kernel or memory vulnerabilities). Developers can choose to attest only keys of recent Android OS versions or specific device brands. However, this makes the solution less practical. Ultimately, a continuous runtime integrity check is required to overcome this issue.

5.8 Case study II: Beyond the framework

The previous sections demonstrate several uses for our proposed modelling framework. Here, we show that our framework is general by modelling an existing protocol, namely attested key exchange that is described in [106, 44]. The protocol uses remote attestation to establish a shared key between a user and an IEE. The user sends their public key to the IEE and expects a fresh symmetric key in return, encrypted with the user's public key. We modelled the protocol between a developer server and the trusted app. We considered the same threat model as the code protection protocol.

The protocol is illustrated in figure 5.7. It starts by creating an honest app that is shipped with its developer's public key. Then, the app starts the Key Attestation protocol, requests a nonce from the developer server, and requests creating a certificate chain from the Android Keystore. Because the Android Keystore does not export symmetric keys, the app will create the symmetric key and request to sign it with the generated certificate chain. Then, the Android Keystore encrypts the certificate chain, the symmetric key and its signature using the developer's public key. Finally, the encrypted content is sent to the developer server over a public channel for decryption and verification.

We modelled several security properties, including 'verification of device and app integrity' and 'shared key secrecy.' The first property ensures that all the attested, shared keys must be created by honest developers and attested by a valid device. On the other hand, the second property ensures that an adversary cannot learn/supply a key to the developer that yields a valid verdict. All the security properties were proven successful, with our framework showing that this protocol would be secure when implemented as an Android app.

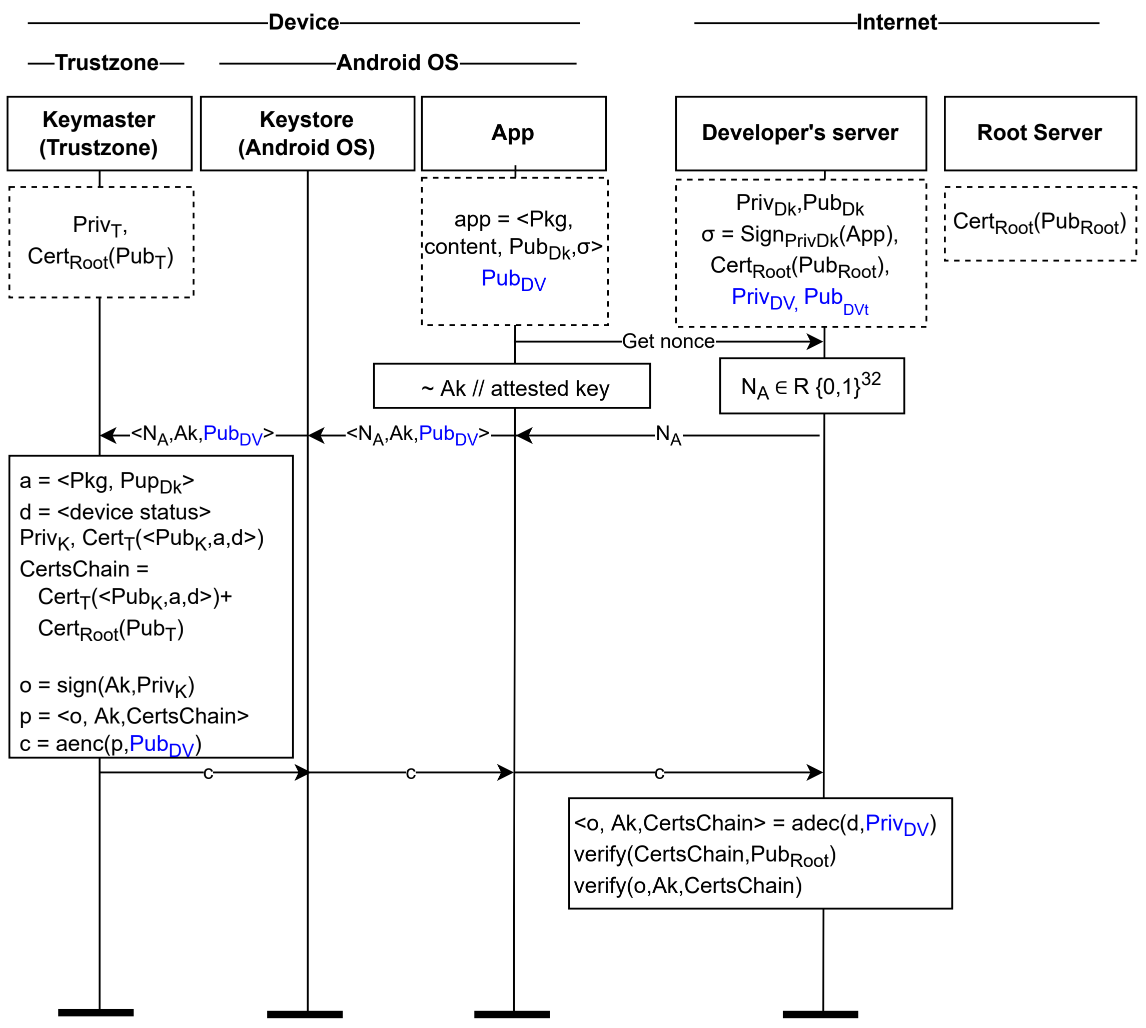


Figure 5.7: Modelling attested key exchange protocol from [106, 44] using the proposed remote attestation framework

5.9 Conclusion

In this paper, we performed symbolic verification of several Android remote attestation protocols that ensure app and device integrity. Our modelling framework allows us to check the general security properties of the attestation frameworks and also the properties of apps that use these frameworks. We have shown that Samsung Knox V2 attestation fails to ensure the integrity of devices because it fails to ensure app integrity. We conclude that ensuring device integrity alone is not enough; app integrity is required as well. Also, we have shown that the recommended practice of Android Key Attestation misses a challenge phase, which allows an adversary to attest old keys that might be compromised without the developer's awareness. We have also discussed possible mitigation for both issues. Samsung Knox V2 requires a patch. While Key Attestation requires updating the recommended practice documentation to include a challenge phase. Finally, we presented two case studies to generalise the framework and solve a code protection problem in Android by utilising our findings and models.

Acknowledgement

This research is partially funded by the NCSC through the TimeTrust project and by EPSRC under grants EP/R012598/1 and EP/V000454/1. The results feed into DsbDtech. Abdulla Aldoseri is supported by a stipend from the University of Bahrain.

Chapter 6

A Tale of Two Worlds

Assessing the Vulnerability of Enclave Shielding Runtimes

This chapter was previously published as:

> Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. ”A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes.” In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS), Nov. 2019, pp. 1741-1758. 2019.(`https://dl.acm.org/doi/abs/10.1145/3319535.3363206`)[188].

Preamble.

Finally, in this chapter, we moved to TEE platform. This chapter discusses our analysis of enclaves of trusted execution environment (TEE). It analyses the vulnerability space arising in TEEs when interfacing a trusted enclave application with untrusted, potentially malicious code. Considerable research and industry effort have gone into developing TEE runtime libraries with the purpose of transparently *shielding* enclave application code from an adversarial environment. However, our analysis reveals that shielding requirements are generally not well-understood in real-world TEE runtime implementations. We expose a sanitization vulnerability at the application level that can lead to exploitable memory safety and side-channel vulnerability in the compiled enclave in Intel SGX. Mitigation of the discovered vulnerability is not as simple as ensuring that pointers are outside enclave memory. We demonstrate that state-of-the-art mitigation techniques such as Intel’s `edger8r` fail to fully eliminate this attack surface. We practically exploit this vulnerability and demonstrate an attack scenario to leak secret keys from the enclave. We have responsibly disclosed our findings, leading to a designated CVE record.

Remark regarding my work

I participated in this work in the first year of my PhD. This work is a collaboration between many researchers from the University of Birmingham and KU Leuven University and discovered 35 enclave interface sanitisation vulnerabilities in eight major open-source shielding frameworks for Intel SGX, RISC-V, and Sancus TEEs. This chapter only highlights the part of the original published work on which I collaborated, which focuses on Intel SGX. My contribution to this work include the implementation of the proof-of-concept attack scenario that was discussed in section 6.5. I developed a PoC application with an enclave that performs AES encryption. Another co-author contributed to the PoC by improving it and implementing the side-channel attack part. The credit for this work goes to my colleagues: Jo Van Bulck, David Oswald, Eduard Marin, Flavio D. Garcia and Frank Piessens.

6.1 Introduction

Minimization of the Trusted Computing Base (TCB) has always been one of the key principles underlying the field of computer security. With an ongoing stream of vulnerabilities in the mainstream of the operating system and privileged hypervisor software layers, Trusted Execution Environments (TEEs) [125] have been developed as a promising new security paradigm to establish strong hardware-backed security guarantees. TEEs such as Intel SGX [66] and ARM TrustZone [147]realize isolation and attestation of secure application compartments, called *enclaves*. Essentially, TEEs enforce a dual-world view, where even compromised or malicious system software in the normal world cannot gain access to the memory space of enclaves running in an isolated secure world on the same processor. This property allows for drastic TCB reduction: only the code running in the secure world needs to be trusted for enclaved computation results. Nevertheless, TEEs merely offer a relatively coarse-grained memory isolation primitive at the hardware level, leaving it up to the enclave developer to maintain useful security properties at the software level. This can become particularly complex when dealing with interactions between the untrusted host OS and the secure enclave, e.g. sending or receiving data to or from the enclave. For this reason, recent research and industry efforts have developed several TEE runtime libraries like Intel's SGX-SDK [104] that transparently shield enclave applications by maintaining a secure interface between the normal and secure worlds.

Intel's SGX-SDK offer a form of `ecall`/`ocall` mechanisms to switch from the normal to the secure word (and vice versa). Building on this hardware-level isolation primitive, TEE runtimes aim to ease enclave development by offering a higher level abstraction to enclave programmers. Also, it offers a secure function call abstraction, where untrusted code is allowed to only call explicitly annotated `ecall` entry points within the enclave. Furthermore, at this level of abstraction, the enclave application code can call back to the untrusted world by means of specially crafted `ocall` functions. It is the TEE runtime's responsibility to safeguard the secure function call abstraction by sanitizing low-level ABI state and marshalling input and output buffers when switching to and from enclave mode. However, the SDK-based approach still leaves it up to the developer to manually partition secure application logic and design the enclave interface. As an alternative to such specifically written enclave code, one line of research [48, 182, 39, 173] has developed dedicated enclave library OSs that seamlessly enforce the `ecall`/`ocall` abstraction at the system call level. Ultimately, this approach holds the promise to securely running unmodified executables inside an enclave and fully transparently applying TEE security guarantees.

Over the last years, security analysis of enclaved execution has received considerable attention from a microarchitectural side-channel [184, 118, 123, 185, 130] and more recently also transient execution perspective [186, 56, 113]. However, in the era where our community is focusing on patching enclave software against very advanced Spectre-type attacks, comparably little effort has gone into exploring how resilient commonly used trusted runtimes are against plain *architectural* memory-safety style attacks. Previous research [117, 51] has mainly focused on developing techniques to efficiently exploit traditional memory safety vulnerabilities in an enclave setting, but has not addressed the question of how prevalent such vulnerabilities are across TEE runtimes. More importantly, it remains largely unexplored whether there are *new* types of vulnerabilities or attack surfaces that are specific to the unique enclave protection model. Clearly, the enclave interface represents an important attack surface that so far has not received the necessary attention and thus is the focus of this chapter.

Our contribution In this chapter, we study the question of how a TEE trusted runtime can securely "bootstrap" from an initial attacker-controlled machine state to a point where execution can be safely handed over to the actual application written by the enclave developer. As case study, we select Intel SGX (a TEE runtime), examines it, and observe its shielding process for enclave application to preserve its program semantics all times.

Then, we review the arguments of its enclave (*i.e.* pointers and size arguments) according to the SGX design as it shape a borderline because the enclave's address space is shared with untrusted adversary-controlled code [66]. Hence, the enclaved binary may assume that untrusted pointer arguments are properly sanitized to point outside the trusted memory, or that `ocall` return values have been scrutinized.

Our main contributions are:

- We analyse entry points of a TEE runtime namely Intel SGX as case study and identify a memory vulnerability in it.

- We practically demonstrate an attack scenario in which the discovered vulnerability can be abused as software-based side-channel attack to leak protected enclave's memory.
- Finally, We show that state-of-the-art automated enclave interface sanitization approaches such as `edger8r` fail to fully prevent our attack, highlighting the need for more principled mitigation strategies.

Responsible disclosure. The security vulnerability of Intel SGX described in this work has been responsibly disclosed through the proper channel for affected TEE runtime. It can be tracked via CVE-2018-3626. To ensure the reproducibility of our work, and to provide the community with a relevant sample of vulnerable enclave programs for evaluating future attacks and defences, we published the attack code at `https://github.com/jovanbulck/0xbadc0de`.

6.2 Background

This section reviews enclave operation and TEE design. It introduces the trusted runtime libraries we analysed in this work. Finally, it summarises related work on TEE memory corruption attacks.

6.2.1 Enclave entry and exit

TEE design The mechanisms to access enclaves vary depending on the underlying TEE being used. We are considering two types of TEE designs: those that rely on a single-address-space model (e.g., Intel SGX [66] and Sancus [143]) vs. the ones that follow a two-world view (e.g., ARM TrustZone [147] and Keystone [116]). In the former case, enclaves are embedded in the address space of an unprivileged host application. The processor orchestrates enclave entry/exit events and enforces that enclave memory can never be accessed from outside the enclave. Since the trusted code inside the enclave is allowed to freely access unprotected memory locations outside the enclave, bulk input/output data transfers are supported by simply passing pointers in the shared address space.

In the case of a two-world design, on the other hand, the CPU is logically divided into a "normal world" and a "secure world". A privileged security monitor software layer acts as a bridge between both worlds. The processor enforces that normal world code cannot access secure world memory and resources, and may only call a predefined entry point in the security monitor. Since the security monitor has unrestricted access to the memory of both worlds, an explicit "world-shared memory" region can typically be set up to pass data from the untrusted OS into the enclave (and vice versa).

Enclave entry/exit Given that the runtimes we studied focus mainly on Intel SGX, we now describe `ecall`/`ocall` and exception handling following SGX terminology [66]. Note that other TEEs feature similar mechanisms, the key difference for a two-world design being that some of the enclave entry/exit functionality may be implemented in the privileged security monitor software layer instead of in the processor.

In order to enter the enclave, the untrusted runtime executes the `eenter` instruction, which switches the processor into enclave mode and transfers execution to a predefined entry point in the enclave's Trusted Runtime System (TRTS). Any metadata information, including the, requested `ecall` interface function to be invoked, can be passed as untrusted parameters in CPU registers. TRTS first sanitizes CPU state and untrusted parameters before passing control to the `ecall` function to be executed. Subsequently, TRTS issues an `eexit` instruction to perform a synchronous enclave exit back to the untrusted runtime, again passing any parameters through CPU registers. The process for `ocall`s takes place in reverse order. When the enclave application calls into TRTS to perform an `ocall`, the trusted CPU context is first stored before switching to the untrusted world, and restored on subsequent enclave re-entry.

When encountering interrupts or exceptions during enclaved execution, the processor executes an Asynchronous Enclave eXit (AEX) procedure. AEX first saves CPU state to a secure Save State Area (SSA) memory location inside the enclave, before scrubbing registers and handing control to the untrusted OS. The enclave can subsequently be resumed through the `eresume` instruction. Alternatively, the untrusted runtime may optionally first call a special `ecall` which allows the enclave's TRTS to internally handle the exception by inspecting and/or modifying the saved SSA state.

6.2.2 Intel SGX as a TEE shielding runtime

With the release of Intel SGX's open-source SGX-SDK, Intel [104] supports a secure function call abstraction to enable production enclave development in C/C++. Apart from pre-built trusted runtime libraries, a key component of the SDK is the `edger8r` tool, which parses a developer-provided Enclave Description Language (EDL) file in order to automatically generate trusted and untrusted proxy functions to be executed when crossing enclave boundaries.

6.3 Related work

During the last decade, significant research efforts have been made to discover and mitigate memory corruption attacks in Intel SGX. Lee et al. [117] were the first to execute a completely blind memory corruption attack against SGX by augmenting code reuse attack techniques [172] with several side-channel oracles. To successfully mount this attack, adversaries require kernel privileges and a static enclave memory layout. Recently, these techniques were improved by Biondo et al. [51] to allow even non-privileged adversaries to hijack vulnerable enclaves in the presence of fine-grained address space randomization [170]. Their approach is furthermore made application-agnostic by leveraging gadgets found in the trusted runtime library of the official Intel SGX-SDK. In a perpendicular line of research, Schwarz et al. [168] criticized SGX's design choice of providing enclaves with unlimited access to untrusted memory outside the enclave. They demonstrated that malware code executing inside an SGX enclave can mount stealthy code reuse attacks to hijack control flow in the untrusted host application.

Importantly, all previous SGX memory safety research focused on contributing novel exploitation techniques while assuming the prior presence of a vulnerability in the enclave code itself. Hence, those results are *complementary* to the vulnerabilities described in this work. We have indeed demonstrated control flow hijacking for some of the pointer sanitization issues below, and these may further benefit from exploitation techniques developed in prior work.

6.4 Methodology and adversary model

This section defines the research objective and outlines the research methodology to determine the scope of the research. Based on that, we construct the attacker model and describe it afterwards.

6.4.1 Research methodology

Our objective is to pinpoint enclave shielding responsibilities, and to find vulnerabilities where a real-world TEE runtime fails to safeguard implicit interface assumptions made by the enclaved binary. Our methodology mainly focuses on code review of Intel SGX considering its architecture design aspects.

We review how its runtime validates different kinds of arguments passed in through an `ecall` or as the return value of an `ocall` considering authenticity and confidentiality. First, for authenticity, We focus in particular on the handling of pointers and strings, where it is the TEE runtime's responsibility to ensure that variable-sized buffers lie entirely outside the enclave before copying them inside and transferring execution to the enclaved binary. For confidentiality, we check again that all memory copied outside the TEE only contains explicit return values, and that no avoidable side-channel leakage is introduced.

6.4.2 Attacker model

We consider systems with hardware support for a TEE and where a trusted runtime supports the secure, shielded execution of an enclaved binary produced by the application developer. With *enclaved binary*, we specifically mean that the binary is the output of a standard compiler, which is not aware of the TEE. It is the responsibility of the shielding runtime to preserve intended program semantics in a hostile environment. We focus exclusively on vulnerabilities in the TEE runtime and assume that there are no application-level memory safety vulnerabilities in the enclaved binary.

We assume the standard TEE attacker model [125], where adversaries have full control over *all* software executing *outside* the hardware-protected memory region. This is a powerful attacker model, allowing the adversary to, for instance, modify page table entries [194, 187], or precisely execute the victim enclave one instruction at a time [185]; yet, this is the attacker that TEEs are designed to defend against.

6.5 Validating string arguments in enclaves

Intel SGX enclave can be written in a low-level language like C. In such a case, String arguments do not carry an explicit length and may not even have been properly null-terminated. Thus, shielding runtimes need to first determine the expected length and always include a null terminator when copying the string inside the enclave.

Attack vector (strings): Runtimes should avoid computing untrusted string sizes, and always include a null byte at the expected end. ▷ At least one related instance repeated across two production SDKs.

TEE design We show below how computing on unchecked string pointers may leak enclave secrets through side-channels, even if the `ecall` is eventually rejected. While side-channels are generally a known issue across TEE technologies [66, 147, 184, 116] and may even be observed by non-privileged adversaries, for example by measuring overall execution time [130] or attacker-induced cache evictions [169, 123], we show that TEE-specific design decisions can still largely affect the overall exploitability of subtle side-channel vulnerabilities. Particularly, we develop a highly practical attack that abuses several privileged adversary capabilities that have previously been proven notorious in the Intel SGX design, e.g. untrusted page tables [194, 187], interrupts [118, 185, 184], and storing interrupted CPU register contents in SSA memory frames [186, 56].

Intel SGX-SDK We discovered that `edger8r`-generated code may be tricked into operating on unchecked in-enclave pointers when computing the size of a variable-length input buffer. While such illegal `ecall` attempts will always be properly rejected, we found that adversaries can exploit the unintended size computation as a deterministic oracle that reveals side-channel information about arbitrary in-enclave memory locations. This vulnerability is tracked via CVE-2018-3626 (Intel SA-00117), leading to enclave TCB recovery and changes in the EDL specification [105]. Prior to our disclosure, EDL allowed programmers to specify a custom `[sizefunc]` attribute that takes as an argument an *unchecked* pointer to an application-specific structure, and returns its size. Likewise, there is a dedicated `[string]` EDL attribute to specify null-terminated string arguments. Essentially, this special case comes down to `[sizefunc=strlen]`.

Consider the code skeleton generated by `edger8r` in listing 6.1 for an `ecall` that expects a single-string pointer argument. In order to verify that the complete string is outside the enclave, the trusted edge routine *first* computes the size of the argument buffer (through either `strlen()` or a dedicated `sizefunc` in general), and only *thereafter* checks whether the entire buffer falls outside of the enclave. It is intended that the edge code first determines the length in untrusted memory, but we made the crucial observation that the `strlen()` invocation at line 7 operates on an arbitrary unchecked pointer, potentially pointing into enclave memory. Any pointer poisoning attempts will subsequently be rejected at line 10, but the unintended computation may have already leaked information through various side-channels [118, 185]. In general, leakage occurs whenever there is secret-dependent control or data flow in the specified `sizefunc`. This is most obviously the case for the common `[string]` EDL attribute, since the amount of loop operations performed by `strlen()` reveals the number of non-zero bytes following the specified in-enclave pointer.

Our attack builds on top of the open-source SGX-Step [185] enclave interrupt framework to turn the subtle `strlen()` side-channel leakage into a fully *deterministic oracle* that reveals the exact position of all `0x00` bytes in enclave private memory (thereby for instance fully breaking the confidentiality of booleans or providing valuable information for cryptanalysis). Particularly, we use SGX-Step to reliably step the `strlen()` execution, one instruction at a time, leveraging the "accessed" bit in the page table entry of the targeted in-enclave memory location as a noise-free oracle that is deterministically set by the processor for every `strlen()` loop iteration [187]. We confirmed that our single-stepping oracle continues to work reliably even when the victim enclave was compiled to a single, extremely compact `rep movsb` instruction (x86 string operations can indeed be interrupted in between every loop iteration [103]).

```
1  static sgx_status_t SGX_CDECL sgx_my_ecall(void* pms)
2  {
3    CHECK_REF_POINTER(pms, sizeof(ms_my_ecall_t));
4    ms_my_ecall_t* ms = SGX_CAST(ms_my_ecall_t*, pms);
5    char* _tmp_s = ms->ms_s;
6
7 ★  size_t _len_s = _tmp_s ? strlen(_tmp_s) + 1 : 0;
8    char* _in_s = NULL;
9
10   CHECK_UNIQUE_POINTER(_tmp_s, _len_s);
11   __builtin_ia32_lfence(); // fence after pointer checks
12   ...
```

Listing 6.1: Proxy function generated by `edger8r` for the EDL specification: `public void my_ecall([in,string] char *s)`.

We developed a practical end-to-end AES-NI key extraction PoC in an application enclave built with a vulnerable version of `edger8r`. Our victim enclave provides a single, multi-threaded `ecall` entry point that encrypts the first 16 bytes of a given string using side-channel resistant AES-NI instructions with a secret in-enclave key. Since AES-NI operates exclusively on CPU registers (e.g. `xmm0`) and due to the limited nature of the `strlen()` side-channel, we cannot perform key extraction by directly targeting the AES state or key in memory. Instead, our attack uses repeated encryption `ecall`s, assuming varying (but not necessarily known) plaintext and known ciphertext. We further abuse that the Intel SGX architecture enables a privileged adversary to precisely interrupt a victim enclave at a chosen instruction-level granularity [185], thereby forcing the processor to write the register state to a fixed SSA location in enclave memory (this includes the `xmm` registers that are part of the `XSAVE` region of the SSA frame). Figure 6.1 depicts the high-level phases of the attack flow, using two threads A and B:

(a) Invoke the encryption `ecall` from thread A ① and interrupt the enclave ② before the final round of the AES (*i.e.* before the `aesenclast` instruction). To keep the PoC simple, we achieve this requirement by inserting an access to a dummy page at the appropriate point, and catching accesses to this page in a signal handler on the untrusted side. Note that in a real-world attack, the single-stepping feature of SGX-Step could be used to execute the victim enclave exactly up to this point, without relying on a more coarse-grained page fault for interruption.

(b) While the `ecall` in thread A is interrupted, prepare the timer used by SGX-Step ③ and launch a second thread B ④ to probe the position of the first zero byte (if any) in the intermediate AES state. Concretely, this involves a second `ecall` to the same entry point, but this time supplying an illegal in-enclave target address pointing to the fixed memory location containing the `xmm0` register in the SSA frame of the interrupted thread A. Each time when a timer interrupt arrives ⑤, we monitor and clear ⑥ the "accessed" bit of the targeted SSA page table entry.

(c) After the `strlen()` probing has finished, the obtained leakage is stored alongside the corresponding ciphertext, and thread A is resumed by restoring read/write access to the dummy page.

(d) Repeat from step (a) with a different plaintext until the full key has been recovered (see algorithm 1).

Experimentally, we determined that this attack succeeds with 881 AES invocations on average (over 1000 runs with random keys, minimum: 306, maximum: 3346), given a deterministic, noise-free `strlen()` oracle. Note that this attack could also be adapted to work with noisy measurements, using the so-called zero-value model known from hardware side-channel attacks [86]. Besides, the attack would also be applicable when targeting the first round of the AES in a known-plaintext scenario.

Properly closing this side channel requires profound changes in the way `edger8r` works. Notably, the bridge code includes an `lfence` instruction at line 11 to rule out advanced Spectre-v1 misspeculation attacks that might still speculatively compute on unchecked pointers before they are architecturally rejected. However, our attack is immune to such countermeasures because we directly observe the side effects of normal, non-speculative execution. Further, early rejecting the `ecall` when detecting that the start pointer falls inside the enclave does not suffice in general. In such a case, adversaries might still pass pointers below the enclave base address, and observe secret-dependent behavior based on the first bytes of the enclave. Intel implemented our recommended mitigation

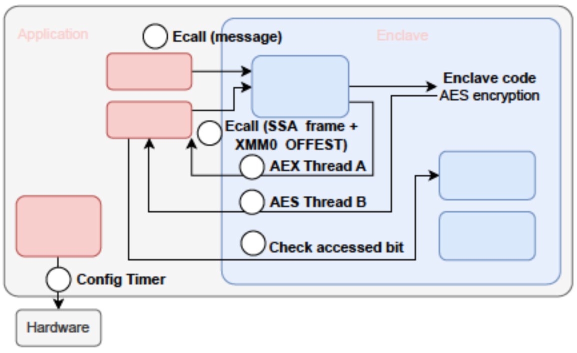


Figure 6.1: Overview of the key extraction attack exploiting `strlen()` side-channel leakage in Intel SGX-SDK based on [188]

Algorithm 1 `strlen()` oracle AES key recovery where $S(\cdot)$ denotes the AES SBox and $SR(p)$ the position of byte p after AES ShiftRows.

```
while not full key $K$ recovered do
    $(P, C, L) \leftarrow$ random plaintext, associated ciphertext, strlen oracle
    if $L < 16$ then
        $K[SR(L)] \leftarrow C[SR(L)] \oplus S(0)$
    end if
end while
```

strategy by dropping support for the superfluous `[sizefunc]` EDL attribute entirely, and further abstaining from computing untrusted buffer sizes inside the enclave. Instead, alleged buffer sizes are computed outside the enclave, and passed as an *untrusted* argument, such that the `CHECK_UNIQUE_POINTER` test can take place immediately. For the `strlen()` case, the untrusted memory can simply be copied inside, and an extra null byte inserted at the alleged end. This solution conveniently moves all secret-dependent control flow from the enclave into the untrusted application context.

6.6 Discussion and mitigation

The most intuitive solution to defend against our attack is to incorporate additional checks in the enclave code to properly sanitize API arguments' values. However, leaving the decision of whether (and how) to correctly implement numerous interface validation checks to enclave developers, who are likely unaware of this class of vulnerabilities, may be problematic. This highlights the need for more principled approaches to rule out this class of vulnerabilities at large, as well as defense-in-depth code hardening measures that may raise the bar for successful exploitation.

Code hardening The presented attack are free from non-static address dependencies, and hence remain inherently immune to software randomization schemes. The SGX-SDK `strlen()` oracle in fig. 6.1 depends solely on the *fixed* address of the victim's SSA frame, which is deterministically dictated by the SGX hardware and immutable from software.

As a perpendicular code hardening avenue, we recommend implementing more aggressive responses when detecting pointer violations in the trusted runtime. That is, most of the runtimes we studied merely reject the `ecall` attempt when detecting pointer poisoning. In the SGX-SDK `strlen()` oracle attack of section 6.5, we for example abused this to repeatedly call a victim enclave, each time passing an illegal pointer and making side-channel observations before the `ecall` is eventually rejected. To rule out such repeated attacks, and reflect that in-enclave pointers represent clear adversarial or buggy behavior, we recommend immediately destroying secrets and/or initiating an infinite loop upon detecting the *first* pointer poisoning attempt in the trusted runtime.

Safe programming languages The combination of TEEs and safe programming languages has been proposed as a promising research direction to safeguard enclave program semantics, but still requires additional interface sanitizations [85]. However, it is important to note that safe languages by themselves are not a silver bullet solution as trusted runtime code remains responsible for bootstrap memory safety guarantees by providing a correct implementation of sanitization in the untrusted pointer type. For instance, as an alternative to Intel's `edger8r` tool, the use of separation logic has been proposed to automatically generate secure wrappers for SGX enclaves [84] aims to provide the advantages of safe languages, and even formal verification guarantees, but still relies on explicit developer annotations.

6.7 Conclusions and future work

Our work highlights that the shielding responsibilities in today's TEE runtimes are not sufficiently understood, and that security issues exist in the respective trusted computing bases. As a case study, We identified an interface sanitization vulnerability in intel SGX that could lead to leak the enclave's runtime memory. In the defensive landscape, our work emphasizes the need to research more principled interface sanitization strategies to safeguard the unique TEE shielding responsibilities. We particularly encourage the development of static analysis tools, and fuzzing-based vulnerability discovery and exploitation techniques to further explore this attack surface.

Acknowledgments

We thank Jethro Beekman (Fortanix), Job Noorman (KU Leuven), and Johannes Götzfried for insightful discussions, and the anonymous reviewers for constructive feedback that helped to improve the paper. We further would like to thank the maintainers of the open-source projects we studied for their contributions to the community and for promptly responding and working on mitigations.

This research is partially funded by the Research Fund KU Leuven, and by the Agency for Innovation and Entrepreneurship (Flanders). Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/R008000/1 and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM). Abdulla Aldoseri is supported by a stipend from the University of Bahrain.

Chapter 7

Conclusion

Here, we conclude the thesis by summarising our results, addressing limitations and purposing opportunities for future work.

7.1 Summary

In this thesis, we studied the attack surface of interfaces across three main platforms namely browser, mobile and computer. We established our study by briefly defining and explaining the interfaces similarities across the platforms. Through the literature, we set a scope for the project to asses interfaces across these platforms by identifying the less researched interactions between the target interfaces and their corresponding services, permissions and polices. Starting with Chapter 2 and Chapter 3, we analysed the interfaces of Web browsers that included local schemes and hardware APIs. Our study demonstrated several security issues that affect Web browsers. Also, it shows that different components like new input methods, output methods, internal processes or different context that interact with interfaces can lead to security threats. In Chapter 4, we studied mobile interfaces namely app components and their interaction across multi-user service and background restriction policy in Android. This study resulted in the discovery of security vulnerabilities that lead to the bypassing of security protection enforced by this service and policy. As a takeaway, this study has shown that it is important to evaluate new services with existing interfaces. Later, in Chapter 5 and Chapter 6, we moved to TEE and evaluated enclaves' interfaces and their application, namely remote attestation. These studies revealed several security threats that allow leaking data from protected compiled intel SGX enclave and relay attack in Samsung Knox remote attestation protocol. Based on this, we conclude that while there are efforts specifically devoted to use and develop trusted shielding runtime, the attack surfaces are not generally well-understood.

7.2 Limitation and future work

The limitations of the work detailed in each chapter have been specifically addressed specifically within the individual chapters. However, we generalise these limitations into two main points as follows. Firstly, most of the analyses were conducted through manual inspection of the targets' source code or their public manual beside the Four Gates Inspector that was discussed in Chapter 4. Secondly, there are still numerous interfaces have not been studied yet and have not ben covered in our study. Therefore, as future work, we will consider tackling these two limitations to further advance our research. We will consider involving different detection techniques like fuzzing and machine learning for interface analysis. Also, we will further study APIs and their interaction with different services. For instance, at the time of writing, we are studying the effect of Chrome extension and WebAPIs to present Chapter 3 as a scientific paper. Further details about this topic's recent reported issue that has not been discussed in this thesis is available at Chromium bugs website (`https://bugs.chromium.org/p/chromium/issues/detail?id=1371867`).

7.3 Concluding thoughts

The area of interface security is huge. Previous studies focused on improving the protection of interfaces, while the attack surfaces of these interfaces are generally not well understood. In a matter of fact, introducing new components that interact/interfere with interfaces can increase the attack surface of these interfaces exponentially, voiding their defined security properties. As addressed, these components - new input methods, output methods, internal processes or porting an interface to a different platform - affect the security of the interfaces directly or indirectly. Therefore, further testing mechanisms are needed to evaluate the attack surfaces of these interfaces.

Bibliography

[1] Martín Abadi and Cédric Fournet. “Mobile values, new names, and secure communication”. In: *ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL)*. London (UK): ACM, Jan. 2001, pp. 104–115.

[2] Feross Aboukhadijeh. *Using the HTML5 fullscreen api for phishing attacks*. 2012.

[3] Abdulla Aldoseri and David Oswald. “insecure://Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers”. In: *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. 2022.

[4] Abdulla Aldoseri, David Oswald, and Robert Chiper. “A Tale of Four Gates: Privilege Escalation and Permission Bypasses on Android Through App Components”. In: *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part II*. Springer. 2022, pp. 233–251.

[5] Abdulla Aldoseri et al. “Symbolic Modelling of Remote Attestation Protocols for Device and App Integrity on Android”. In: *18th ACM ASIA Conference on Computer and Communications Security*. Association for Computing Machinery (ACM). 2023.

[6] Johanna Amann et al. “Mission accomplished? HTTPS security after DigiNotar”. In: *Proceedings of the 2017 Internet Measurement Conference*. 2017, pp. 325–340.

[7] Mohit Dayal Ambedkar, Nanhay Singh Ambedkar, and Ram Shringar Raw. “A comprehensive inspection of cross site scripting attack”. In: *2016 international conference on computing, communication and automation (ICCCA)*. IEEE. 2016, pp. 497–502.

[8] Android. *About Dynamic Delivery — Android Developers*. `https://developer.android.com/studio/projects/dynamic-delivery`. (Accessed on 12/01/2019). 2019.

[9] Android. *ActivityManager.RunningAppProcessInfo*. `https://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo`. Accessed on 09/17/2020. 2020.

[10] Android. *Analyze your build with the APK Analyzer — Android Developers*. `https://developer.android.com/studio/debug/apk-analyzer`. (Accessed on 03/17/2023).

[11] Android. *Android Flash Tool*. `https://flash.android.com/welcome`. (Accessed on 10/18/2022). Oct. 2022.

[12] Android. *App-ops*. `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android11-d1-b-release/core/java/android/app/AppOps.md#foreground`. (Accessed on 04/27/2022). Apr. 2022.

[13] Android. *Application Fundamentals — Android Developers*. `https://developer.android.com/guide/components/fundamentals`. (Accessed on 05/05/2021). May 2021.

[14] Android. *Application Signing — Android Open Source Project*. `https://source.android.com/security/apksigning`. (Accessed on 11/24/2021). Nov. 2021.

[15] Android. *Behavior changes: all apps*. `https://developer.android.com/about/versions/pie/android-9.0-changes-all`. Accessed on 09/13/2020. 2020.

[16] Android. *Building Multiuser-Aware Apps*. `https://source.android.com/devices/tech/admin/multiuser-apps`. (Accessed on 05/05/2021). May 2021.

[17] Android. *Compatibility Test Suite — Android Open Source Project*. `https://source.android.com/compatibility/cts`. (Accessed on 05/28/2022).

[18] Android. *Copy and Paste — Android Developers*. https://developer.android.com/guide/topics/text/copy-paste. (Accessed on 09/29/2021). Sept. 2021.

[19] Android. *Create an input method — Android Developers*. https://developer.android.com/guide/topics/text/creating-input-method. (Accessed on 09/29/2021).

[20] Android. *Data and file storage overview — Android Developers*. https://developer.android.com/training/data-storage. (Accessed on 06/25/2021). June 2021.

[21] Android. *Employing Work Profiles — Android Open Source Project*. https://source.android.com/docs/devices/admin/managed-profiles. (Accessed on 04/02/2023). Apr. 2023.

[22] Android. *Foreground services — Android Developers*. https://developer.android.com/guide/components/foreground-services. (Accessed on 04/14/2022). Apr. 2022.

[23] Android. *Gatekeeper*. https://source.android.com/security/authentication/gatekeeper. (Accessed on 11/01/2020). 2020.

[24] Android. *Image keyboard support — Android Developers*. https://developer.android.com/guide/topics/text/image-keyboard. (Accessed on 09/29/2021). Sept. 2021.

[25] Android. *Key and ID Attestation*. https://source.android.com/docs/security/keystore/attestation. (Accessed on 09/01/2022). Sept. 2022.

[26] Android. *Locking/Unlocking the Bootloader — Android Open Source Project*. https://source.android.com/devices/bootloader/locking_unlocking. (Accessed on 11/25/2021). Nov. 2021.

[27] Android. *Permission — Android Developers*. https://developer.android.com/guide/topics/manifest/permission-element. (Accessed on 05/05/2021). May 2021.

[28] Android. *Permissions on Android — Android Developers*. https://developer.android.com/guide/topics/permissions/overview. (Accessed on 10/18/2022). Oct. 2022.

[29] Android. *Permissions on Android — Android Developers*. https://developer.android.com/guide/topics/permissions/overview. (Accessed on 03/18/2023).

[30] Android. *Permissions updates in Android 11 — Android Developers*. https://developer.android.com/about/versions/11/privacy/permissions. (Accessed on 09/19/2021). Sept. 2021.

[31] Android. *Privileged Permission Allowlisting — Android Open Source Project*. https://source.android.com/devices/tech/config/perms-allowlist. (Accessed on 10/08/2021). Aug. 2021.

[32] Android. *Shrink, obfuscate, and optimize your app — Android Developers*. https://developer.android.com/studio/build/shrink-code. (Accessed on 11/26/2021). Nov. 2021.

[33] Android. *Supporting Multiple Users*. https://source.android.com/devices/tech/admin/multi-user/. (Accessed on 11/01/2020). 2020.

[34] Android. *Trusty TEE — Android Open Source Project*. https://source.android.com/security/trusty. (Accessed on 11/24/2021). Nov. 2021.

[35] Android. *Verified Boot — Android Open Source Project*. https://source.android.com/security/verifiedboot. (Accessed on 11/25/2021). Nov. 2021.

[36] Android. *Verifying hardware-backed key pairs with Key Attestation*. https://developer.android.com/training/articles/security-key-attestation. (Accessed on 11/23/2021). Nov. 2021.

[37] Apktool. *Apktool—A tool for reverse engineering 3rd party, closed, binary Android apps*. https://ibotpeaches.github.io/Apktool/. (Accessed on 05/05/2021). May 2021.

[38] Apple Support. *About the security content of Safari 14.0*. https://support.apple.com/en-bh/HT211845. (Accessed on 10/17/2021). Sept. 2020.

[39] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association. 2016, pp. 689–703.

[40] Steven Arzt et al. “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps”. In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269.

[41] Ahmed M Azab et al. “Hypervision across worlds: Real-time kernel protection from the ARM Trustzone secure world”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* Scottsdale, Arizona, USA: ACM, 2014, pp. 90–102.

[42] Freddy Eduardo Veloz Baez. “Evaluating SGX’s Remote Attestation Security Through the Analysis of Copland Phrases”. PhD thesis. WORCESTER POLYTECHNIC INSTITUTE, 2022.

[43] Rafay Baloch. *Multiple Address Bar Spoofing Vulnerabilities In Mobile Browsers—Miscellaneous Ramblings of An Ethical Hacker.* `https://www.rafaybaloch.com/2020/10/multiple-address-bar-spoofing-vulnerabilities.html`. (Accessed on 10/05/2021). May 2021.

[44] Manuel Barbosa et al. “Foundations of hardware-based attested computation and application to SGX”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE. 2016, pp. 245–260.

[45] A. Barth. *RFC 6454 — The Web Origin Concept.* `https://tools.ietf.org/html/rfc6454`. (Accessed on 01/26/2021). Dec. 2011.

[46] Adam Barth, Collin Jackson, Charles Reis, et al. “The security architecture of the Chromium browser”. In: *Technical report.* Stanford University, 2008.

[47] David Basin et al. *Tamarin Prover (v. 1.6.1).* `https://tamarin-prover.github.io`. Aug. 2021.

[48] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding applications from an untrusted cloud with Haven”. In: *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation.* USENIX Association. 2014, pp. 267–283.

[49] Stefano Berlato and Mariano Ceccato. “A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps”. In: *Journal of Information Security and Applications* 52 (2020), p. 102463.

[50] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986. Jan. 2005. DOI: `10.17487/RFC3986`. URL: `https://rfc-editor.org/rfc/rfc3986.txt`.

[51] Andrea Biondo et al. “The Guard’s Dilemma: Efficient Code-Reuse Attacks Against Intel SGX”. In: *Proceedings of the 27th USENIX Security Symposium.* 2018, pp. 1213–1227.

[52] Sven Bugiel et al. “Towards Taming Privilege-Escalation Attacks on Android”. In: *NDSS.* Vol. 17. San Diego, California, USA: NDSS, 2012, p. 19.

[53] Sven Bugiel et al. “Xmandroid: A new android evolution to mitigate privilege escalation attacks”. In: *Technische Universität Darmstadt, Technical Report TR-2011-04* (2011).

[54] Yinzhi Cao et al. “Abusing browser address bar for fun and profit—an empirical investigation of add-on cross site scripting attacks”. In: *International Conference on Security and Privacy in Communication Networks.* Springer. 2014, pp. 582–601.

[55] CENSUS. *Examining the value of SafetyNet Attestation as an Application Integrity Security Control.* `https://census-labs.com/news/2017/11/17/examining-the-value-of-safetynet-attestation-as-an-application-integrity-security-control/`. (Accessed on 08/15/2019). 2017.

[56] Guoxing Chen et al. “SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution”. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE. 2019, pp. 142–157.

[57] Zitai Chen et al. “{VoltPillager}: Hardware-based fault injection attacks against Intel {SGX} Enclaves using the {SVID} voltage scaling interface”. In: *30th USENIX Security Symposium (USENIX Security 21).* 2021, pp. 699–716.

[58] Chromium. *1040755 - Security: Another “universal” XSS via copy&paste.* `https://bugs.chromium.org/p/chromium/issues/detail?id=1040755`. (Accessed on 10/17/2021). Jan. 2020.

[59] Chromium. *684011 - Remove: Top frame navigations to data URLs - chromium.* https://bugs.chromium.org/p/chromium/issues/detail?id=684011. (Accessed on 10/17/2021). Jan. 2017.

[60] Chromium. *81e2250b60d4a6d863864249cc2f1da97669f03d - chromium/src - Git at Google.* https://chromium.googlesource.com/chromium/src/+/81e2250b60d4a6d863864249cc2f1da97669f03d. (Accessed on 03/23/2022). Mar. 2022.

[61] Chromium. *AutocompleteEditText.java — Chromium Code Search.* https://source.chromium.org/chromium/chromium/src/+/main:chrome/browser/ui/android/omnibox/java/src/org/chromium/chrome/browser/omnibox/AutocompleteEditText.java;l=189. (Accessed on 12/01/2021).

[62] Chromium. *Chromium Blog: Blink: A rendering engine for the Chromium project.* https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html. (Accessed on 10/09/2021). Sept. 2021.

[63] Chromium. *modules - Chromium Code Search.* https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/modules/. (Accessed on 11/25/2022).

[64] Bugs - Chromium.org. *Issues - chromium.* https://bugs.chromium.org/p/chromium/issues/list. (Accessed on 04/14/2023).

[65] George Coker et al. "Principles of remote attestation". In: *International Journal of Information Security* 10.2 (2011), pp. 63–81.

[66] V. Costan and S. Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016.086 (2016), pp. 1–118.

[67] CWE. *CWE-451: User Interface (UI) Misrepresentation of Critical Information (4.5).* https://cwe.mitre.org/data/definitions/451.html. (Accessed on 10/17/2021). Oct. 2021.

[68] R. Housley W. Polk D. Cooper S. Santesson S. Farrell S. Boeyen. *rfc5280.* https://datatracker.ietf.org/doc/html/rfc5280. (Accessed on 11/28/2021). May 2008.

[69] Lucas Davi et al. "Privilege escalation attacks on Android". In: *International Conference on Information Security.* Springer. Boca Raton, Florida, USA: Springer, 2010, pp. 346–360.

[70] Ivan De Oliveira Nunes et al. "On the TOCTOU problem in remote attestation". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* 2021, pp. 2921–2936.

[71] Danny Dolev and Andrew Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[72] Zakir Durumeric et al. "Analysis of the HTTPS certificate ecosystem". In: *Proceedings of the 2013 conference on Internet measurement conference.* 2013, pp. 291–304.

[73] Mohamed Elsabagh et al. "{FIRMSCOPE}: Automatic Uncovering of {Privilege-Escalation} Vulnerabilities in {Pre-Installed} Apps in Android Firmware". In: *29th USENIX Security Symposium (USENIX Security 20).* 2020, pp. 2379–2396.

[74] William Enck et al. "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), pp. 1–29.

[75] F-Droid. *F-Droid - Free and Open Source Android App Repository.* https://f-droid.org/. (Accessed on 05/11/2022).

[76] F-secure Labs. *Xiaomi Redmi 5 Plus Second Space Password Bypass.* https://labs.f-secure.com/advisories/xiaomi-second-space/. (Accessed on 05/05/2021). May 2021.

[77] Facebook. *What is a self-XSS scam on Facebook? — Facebook Help Centre.* https://www.facebook.com/help/246962205475854. (Accessed on 10/05/2021). May 2021.

[78] Parvez Faruki et al. "Android code protection via obfuscation techniques: past, present and future directions". In: *arXiv preprint arXiv:1611.10231* (2016).

[79] Adrienne Porter Felt et al. "Permission Re-Delegation: Attacks and Defenses". In: *USENIX security symposium.* Vol. 30. San Francisco, CA, USA: USENIX, 2011, p. 88.

[80] Georgios Fotiadis et al. “Root-of-Trust Abstractions for Symbolic Analysis: Application to Attestation Protocols”. In: *International Workshop on Security and Trust Management.* Springer. 2021, pp. 163–184.

[81] Frida. *Frida — A world-class dynamic instrumentation framework.* `https://frida.re/docs/examples/android/`. (Accessed on 10/09/2021). Sept. 2021.

[82] Yuki Fujita, Atsuo Inomata, and Hiroki Kashiwazaki. “Design and Implementation of a multi-factor web authentication system with MyNumberCard and WebUSB”. In: (2019).

[83] Roland Galibert. “Perishable Shipment Tracker: Using IoT, Web Bluetooth and Blockchain to Raise Accountability and Lower Costs in the Perishable Shipment Process”. PhD thesis. 2020.

[84] N. van Ginkel, R. Strackx, and F. Piessens. “Automatically generating secure wrappers for SGX enclaves from separation logic specifications”. In: *Asian Symposium on Programming Languages and Systems.* 2017, pp. 105–123.

[85] N. van Ginkel et al. “Towards safe enclaves”. In: *Hot Issues in Security Principles and Trust (HotSpot).* 2016, pp. 1–16.

[86] J. D. Golić and C. Tymen. “Multiplicative Masking and Power Analysis of AES”. In: *Cryptographic Hardware and Embedded Systems (CHES).* 2003, pp. 198–212.

[87] Google. *Android keystore system.* `https://developer.android.com/training/articles/keystore`. (Accessed on 09/16/2019). 2019.

[88] Google. *Gboard — the Google Keyboard.* `https://play.google.com/store/apps/details?id=com.google.android.inputmethod.latin&hl=en&gl=US`. (Accessed on 10/06/2021). Oct. 2021.

[89] Google. *IActivityManager source code.* `https://android.googlesource.com/platform/frameworks/native/+/refs/heads/android10-c2f2-release/libs/binder/IActivityManager.cpp#82`. (Accessed on 16/05/2022). May 2021.

[90] Google. *Remote debug Android devices.* `https://developer.chrome.com/docs/devtools/remote-debugging/`. (Accessed on 10/14/2021). Apr. 2015.

[91] Google. *SafetyNet Attestation API.* `https://developer.android.com/training/safetynet/attestation`. (Accessed on 07/23/2019). 2019.

[92] Diptangsu Goswami. *GitHub - diptangsu/Sorting-Algorithms: Sorting algorithms in multiple languages.* `https://github.com/diptangsu/Sorting-Algorithms`. (Accessed on 03/25/2023).

[93] Reilly Grant. *Intent to Implement and Ship: WebUSB Interface Class Filtering.* `https://groups.google.com/a/chromium.org/g/blink-dev/c/LZXocaeCwDw/m/GLfAffGLAAAJ`. (Accessed on 11/25/2022).

[94] Alan Grosskurth and Michael W Godfrey. “Architecture and evolution of the modern web browser”. In: *Preprint submitted to Elsevier Science* 12.26 (2006), pp. 235–246.

[95] Guardsquare nv. *DexGuard: Android obfuscation and runtime-self protection (RASP).* `https://www.guardsquare.com/en/products/dexguard`. (Accessed on 09/16/2019). 2019.

[96] Shashank Gupta and Brij Bhooshan Gupta. “Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art”. In: *International Journal of System Assurance Engineering and Management* 8.1 (2017), pp. 512–530.

[97] N. Hardy. “The Confused Deputy (or why capabilities might have been invented)”. In: *ACM SIGOPS Operating Systems Review* 22.4 (1988), pp. 36–38.

[98] Stephan Heuser et al. “DroidAuditor: forensic analysis of application-layer privilege escalation attacks on android (Short paper)”. In: *International Conference on Financial Cryptography and Data Security.* Springer. 2016, pp. 260–268.

[99] Bjoern Hoehrmann. *The ’javascript’ resource identifier scheme.* Tech. rep. Work in Progress. Internet Engineering Task Force, Sept. 2010. 6 pp. URL: `https://datatracker.ietf.org/doc/html/draft-hoehrmann-javascript-scheme-03`.

[100] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. “SafetyNOT: on the usage of the SafetyNet attestation API in Android”. In: *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services.* 2021, pp. 150–162.

[101] Intel. *Attestation Services for Intel® Software Guard Extensions*. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html. (Accessed on 11/15/2022). Nov. 2022.

[102] Intel. *input-types-and-boundary-checking-edl-737361.pdf*. https://www.intel.com/content/dam/develop/external/us/en/documents/input-types-and-boundary-checking-edl-737361.pdf. (Accessed on 03/18/2023).

[103] Intel. "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide". In: 325384 (2016).

[104] Intel. *Intel Software Guard Extensions – Get Started with the SDK*. online, accessed 2019-05-10: https://software.intel.com/en-us/sgx/sdk. 2019.

[105] Intel. *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits*. Revision 1.0. Mar. 2018.

[106] Charlie Jacomme, Steve Kremer, and Guillaume Scerri. "Symbolic models for isolated execution environments". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 530–545.

[107] CK Joe-Uzuegbu, UC Iwuchukwu, and LC Ezema. "Application virtualization techniques for malware forensics in social engineering". In: *2015 International Conference on Cyberspace (CYBER-Abuja)*. IEEE. 2015, pp. 45–56.

[108] Uri Kanonov and Avishai Wool. "Secure containers in Android: the Samsung KNOX case study". In: *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. Vienna, Austria: ACM, 2016, pp. 3–12.

[109] Christoph Kerschbaumer. *Blocking Top-Level Navigations to data URLs for Firefox 59 - Mozilla Security Blog*. https://blog.mozilla.org/security/2017/11/27/blocking-top-level-navigations-data-urls-firefox-59/. (Accessed on 10/17/2021). Nov. 2017.

[110] Hyungsub Kim, Sangho Lee, and Jong Kim. "Exploring and mitigating privacy threats of HTML5 geolocation API". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014, pp. 306–315.

[111] Nak Young Kim et al. "Android Application Protection against Static Reverse Engineering based on Multidexing." In: *J. Internet Serv. Inf. Secur.* 6.4 (2016), pp. 54–64.

[112] KirstenS. *Cross Site Scripting (XSS) Software Attack — OWASP Foundation*. https://owasp.org/www-community/attacks/xss/. (Accessed on 12/03/2021).

[113] E. Mohammadian Koruyeh et al. "Spectre returns! speculation attacks using the return stack buffer". In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2018.

[114] John Kozyrakis. *SafetyNet: Google's tamper detection for Android*. https://koz.io/inside-safetynet/. (Accessed on 08/05/2019). Sept. 2015.

[115] Steve Kremer and Robert Künnemann. "Automated analysis of security protocols with global state". In: *Journal of Computer Security* 24.5 (2016), pp. 583–616.

[116] D. Lee et al. "Keystone: A Framework for Architecting TEEs". In: *arXiv preprint arXiv:1907.10119* (2019).

[117] J. Lee et al. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *Proceedings of the 26th USENIX Security Symposium*. 2017, pp. 523–539.

[118] Sangho Lee et al. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *Proceedings of the 26th USENIX Security Symposium*. 2017, pp. 557–574.

[119] Rui Li et al. "Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 70–86.

[120] Yanlin Li, Jonathan M McCune, and Adrian Perrig. "VIPER: Verifying the Integrity of PERipherals' Firmware". In: (2011).

[121] Kyeonghwan Lim et al. "An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks". In: *JoWUA* 7.3 (2016), pp. 40–52.

[122] Ruben Lindström. *Enabling Smartphones to act as IoT Edge Devices via the Browser-based'WebUSB API': The future of the browser and the smartphone in home electronics IoT systems.* 2021.

[123] M. Lipp et al. "Armageddon: Cache attacks on mobile devices". In: *Proceedings of the 25th USENIX Security Symposium.* 2016, pp. 549–564.

[124] Long Lu et al. "Chex: statically vetting android apps for component hijacking vulnerabilities". In: *Proceedings of the 2012 ACM conference on Computer and communications security.* 2012, pp. 229–240.

[125] Pieter Maene et al. "Hardware-Based Trusted Computing Architectures for Isolation and Attestation". In: *IEEE Transactions on Computers* PP.99 (2017).

[126] Francesco Marcantoni et al. "A large-scale study on the risks of the html5 webapi for mobile sensor-based attacks". In: *The World Wide Web Conference.* 2019, pp. 3063–3071.

[127] Larry M Masinter. *The "data" URL scheme.* RFC 2397. Aug. 1998. DOI: `10.17487/RFC2397`. URL: `https://rfc-editor.org/rfc/rfc2397.txt`.

[128] Simon Meier et al. "The TAMARIN prover for the symbolic analysis of security protocols". In: *International Conference on Computer Aided Verification (CAV).* Vol. 8044. LNCS. Saint Petersburg, Russia: Springer, July 2013, pp. 696–701.

[129] mkwst@chromium.org. *271996 - SOP not observed for local storage for file: URLs - chromium.* `https://bugs.chromium.org/p/chromium/issues/detail?id=271996`. (Accessed on 10/18/2022). Oct. 2022.

[130] A. Moghimi et al. "Memjam: A false dependency attack against constant-time crypto implementations". In: *International Journal of Parallel Programming* 47.4 (2019), pp. 538–570.

[131] Mozilla. *Access-Control-Allow-Origin — HTTP — MDN.* `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin`. (Accessed on 01/26/2021). Jan. 2021.

[132] Mozilla. *Data URLs — HTTP — MDN.* `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs`. (Accessed on 10/05/2021). May 2021.

[133] Mozilla. *Gecko - MDN Web Docs Glossary: Definitions of Web-related terms — MDN.* `https://developer.mozilla.org/en-US/docs/Glossary/Gecko`. (Accessed on 03/17/2023).

[134] Mozilla. *HIDDevice - Web APIs — MDN.* `https://developer.mozilla.org/en-US/docs/Web/API/HIDDevice`. (Accessed on 03/17/2023).

[135] Mozilla. *Same-origin policy — Web security — MDN.* `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`. (Accessed on 01/26/2021). Jan. 2021.

[136] Mozilla. *Types of attacks — Web security — MDN.* `https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks`. (Accessed on 10/05/2021). May 2021.

[137] Mozilla. *Web APIs — MDN.* `https://developer.mozilla.org/en-US/docs/Web/API`. (Accessed on 10/18/2022). Oct. 2022.

[138] Mozilla. *X-Frame-Options - HTTP — MDN.* `https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/X-Frame-Options`. (Accessed on 10/18/2022). Oct. 2022.

[139] Gleb Naumovich and Nasir Memon. "Preventing piracy, reverse engineering, and tampering". In: *computer* 36.7 (2003), pp. 64–71.

[140] Long Nguyen Vu et al. "Android rooting: An arms race between evasion and detection". In: *Security and Communication Networks* 2017 (2017).

[141] Srinivas Nidhra and Jagruthi Dondeti. "Black box and white box testing techniques—a literature review". In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[142] Muneaki Nishimura. *Addressbar spoofing through stored data URL shortcuts on Firefox for Android — Mozilla.* `https://www.mozilla.org/en-US/security/advisories/mfsa2016-05/`. (Accessed on 10/05/2021). Jan. 2015.

[143] J. Noorman et al. "Sancus 2.0: A low-cost security architecture for IoT devices". In: *ACM Transactions on Privacy and Security (TOPS)* 20.3 (2017), 7:1–7:33.

[144] Ivan De Oliveira Nunes et al. “VRASED: A Verified Hardware/Software Co-Design for Remote Attestation”. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1429–1446.

[145] OWASP. *A7:2017-Cross-Site Scripting (XSS) — OWASP*. https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS). (Accessed on 12/01/2021).

[146] OWASP. *Mobile Top 10 2016: M8: Code Tampering.* (Accessed on 08/15/2019). Feb. 2017.

[147] S. Pinto and N. Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Computing Surveys (CSUR)* 51.6 (2019), p. 130.

[148] Paul Ratazzi et al. “A systematic security evaluation of Android’s multi-user framework”. In: *arXiv preprint arXiv:1410.7752* 1.1 (2014), pp. 1–10.

[149] Ole André V. Ravnås. *Frida — A world-class dynamic instrumentation framework*. https://www.frida.re/docs/android/. (Accessed on 08/15/2019). 2019.

[150] Joel Reardon et al. “50 ways to leak your data: An exploration of apps’ circumvention of the android permissions system”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, USA: USENIX, 2019, pp. 603–620.

[151] E. Rescorla. *RFC 2818: HTTP Over TLS*. https://www.rfc-editor.org/rfc/rfc2818. (Accessed on 03/18/2023).

[152] Frederic August Rosen et al. *The Algebra of Mohammed Ben Musa Edited and Translated by Frederic Rosen*. Oriental translation fund, 1831.

[153] rovo89. *Xposed Module Repository*. https://repo.xposed.info/module/de.robv.android.xposed.installer. (Accessed on 07/22/2019). 2019.

[154] Shiv Sahni. *Android Key Attestation. What the heck is Android Key... — by Shiv Sahni — InfoSec Write-ups*. https://infosecwriteups.com/android-key-attestation-581da703ac16. (Accessed on 02/13/2022). July 2019.

[155] Samsung. *Enhanced Attestation (v3)*. https://docs.samsungknox.com/dev/knox-attestation/about-attestation.htm. (Accessed on 11/23/2021). Nov. 2021.

[156] Samsung. *Knox Attestation*. https://seap.samsung.com/html-docs/android/Content/knox-attestation.htm. (Accessed on 08/05/2019). 2019.

[157] Samsung. *Knox: Device Health Attestation*. https://docs.samsungknox.com/whitepapers/knox-platform/attestation.htm. (Accessed on 07/31/2019). 2019.

[158] Samsung. *Knox: Root of Trust*. https://docs.samsungknox.com/whitepapers/knox-platform/hardware-backed-root-of-trust.htm. (Accessed on 07/31/2019). 2019.

[159] Samsung. *Real-time Kernel Protection — Knox Platform for Enterprise White Paper*. https://docs.samsungknox.com/admin/whitepaper/kpe/real-time-kernel-protection.htm. (Accessed on 11/28/2021). Nov. 2021.

[160] Samsung. *Root of Trust — Knox Platform for Enterprise White Paper*. https://docs.samsungknox.com/admin/whitepaper/kpe/hardware-backed-root-of-trust.htm. (Accessed on 10/09/2021). Sept. 2021.

[161] Samsung. *Sensitive Data Protection — Knox Platform for Enterprise White Paper*. https://docs.samsungknox.com/admin/whitepaper/kpe/sensitive-data-protection.htm. (Accessed on 10/09/2021). Sept. 2021.

[162] Samsung. *Sensitive Data Protection (SDP)*. https://docs.samsungknox.com/dev/knox-sdk/sensitive-data-protection.htm. (Accessed on 10/09/2021). Sept. 2021.

[163] Samsung. *Tutorial: Attestation (v2)*. https://docs.samsungknox.com/dev/knox-attestation/tutorial-v2.htm. (Accessed on 11/23/2021). Nov. 2021.

[164] Samsung Knox. *Root of Trust*. https://docs.samsungknox.com/admin/whitepaper/kpe/hardware-backed-root-of-trust.htm. (Accessed on 05/06/2021). May 2021.

[165] Samsung Knox. *Secure Folder — Samsung Knox*. https://www.samsungknox.com/en/solutions/personal-apps/secure-folder. (Accessed on 05/06/2021). May 2021.

[166] Application Sandbox. *Application Sandbox — Android Open Source Project*. https://source.android.com/docs/security/app-sandbox. (Accessed on 03/18/2023).

[167] SaurikIT, LLC. *Cydia Substrate*. `http://www.cydiasubstrate.com/`. (Accessed on 07/22/2019). 2012.

[168] M. Schwarz, Samuel Weiser, and Daniel Gruss. "Practical enclave malware with Intel SGX". In: *DIMVA*. 2019, pp. 177–196.

[169] M. Schwarz et al. "Malware guard extension: using SGX to conceal cache attacks". In: *DIMVA*. 2017, pp. 3–24.

[170] J. Seo et al. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs." In: *NDSS 2017*. 2017.

[171] Arvind Seshadri et al. "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems". In: (2005).

[172] H. Shacham et al. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *ACM CCS 2007*. 2007, pp. 552–561.

[173] S. Shinde et al. "Panoply: Low-TCB Linux Applications With SGX Enclaves". In: *NDSS 2017*. 2017.

[174] Lina Song et al. "AppIS: protect Android apps against runtime repackaging attacks". In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. Shenzhen, China: IEEE, 2017, pp. 25–32.

[175] StatCounter Global Stats. *Mobile & Tablet Android Version Market Share Worldwide — StatCounter Global Stats*. `https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/#monthly-202006-202009`. (Accessed on 08/20/2021).

[176] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. "Android Rooting: Methods, Detection, and Evasion". In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '15. Denver, Colorado, USA: ACM, 2015, pp. 3–14. ISBN: 978-1-4503-3819-6. DOI: `10.1145/2808117.2808126`. URL: `http://doi.acm.org/10.1145/2808117.2808126`.

[177] Thomas Sutter and Bernhard Tellenbach. "Simple spyware: Androids invisible foreground services and how to (ab)use them". In: *Black Hat Europe, London, 2.-5. Dezemeber 2019*. London, UK: Black Hat Europe, 2019, p. 27.

[178] T. Berners-Lee et al. *IETF RFC 1738 - Uniform Resource Locators (URL)*. `https://www.w3.org/Addressing/rfc1738.txt`. (Accessed on 01/27/2021). Jan. 2021.

[179] Simon Tanner, Ilian Vogels, and Roger Wattenhofer. "Protecting android apps from repackaging using native code". In: *International Symposium on Foundations and Practice of Security*. Springer. 2019, pp. 189–204.

[180] Akeshi Terada. *Attacking Android browsers via intent scheme URLs*. `https://www.mbsd.jp/Whitepaper/IntentScheme.pdf`. (Accessed on 10/16/2021). Mar. 2014.

[181] Yuan Tian et al. "All your screens are belong to us: attacks exploiting the html5 screen sharing api". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 34–48.

[182] C.C Tsai, D. E Porter, and M. Vij. "Graphene-SGX: A practical library OS for unmodified applications on SGX". In: *USENIX ATC*. 2017.

[183] Güliz Seray Tuncay et al. "Resolving the predicament of Android custom permissions". In: *Network and Distributed System Security Symposium* 1.1 (2018), pp. 1–15.

[184] J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying microarchitectural timing Leaks in rudimentary CPU interrupt logic". In: *ACM CCS 2018*. 2018.

[185] J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A practical attack framework for precise enclave execution control". In: *SysTEX*. 2017, 4:1–4:6.

[186] J. Van Bulck et al. "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution". In: *Proceedings of the 27th USENIX Security Symposium*. 2018.

[187] J. Van Bulck et al. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution". In: *Proceedings of the 26th USENIX Security Symposium*. 2017, pp. 1041–1056.

[188] Jo Van Bulck et al. “A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1741–1758.

[189] Vysor. *Vysor - Android control on PC - Apps on Google Play*. `https://play.google.com/store/apps/details?id=com.koushikdutta.vysor`. (Accessed on 10/19/2022). Oct. 2022.

[190] W3C. *Permissions*. `https://www.w3.org/TR/permissions/`. (Accessed on 10/18/2022). Oct. 2022.

[191] W3C. *WebUSB API*. `https://wicg.github.io/webusb/`. (Accessed on 10/18/2022). Oct. 2022.

[192] Daoyuan Wu and Rocky KC Chang. “Analyzing android browser apps for file:// vulnerabilities”. In: *International Conference on Information Security*. Springer. 2014, pp. 345–363.

[193] Luyi Xing et al. “Upgrading your android, elevating my malware: Privilege escalation through mobile os updating”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. Berkeley, CA, USA: IEEE, 2014, pp. 393–408.

[194] Y. Xu, W. Cui, and M. Peinado. “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”. In: *IEEE Symposium on Security and Privacy*. 2015, pp. 640–656.

[195] Yang Xu et al. “An adaptive and configurable protection framework against android privilege escalation threats”. In: *Future Generation Computer Systems* 92 (2019), pp. 210–224.

[196] Jin-Hyuk Jung Yuxue Piao and Jeong Hyun Yi. “Server-based code obfuscation scheme for APK tamper detection”. In: *Security Comm. Networks*. (Accessed on 11/26/2021). 2016.

[197] Michal Zalewski. *Browser Security Handbook*. `https://code.google.com/archive/p/browsersec/`. (Accessed on 10/18/2022). Oct. 2010.

[198] Mu Zhang and Heng Yin. “AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications.” In: *NDSS*. 2014.

[199] Beibei Zhao et al. “Dexpro: A bytecode level code protection system for android applications”. In: *International Symposium on Cyberspace Safety and Security*. Springer. 2017, pp. 367–382.

[200] Xingqiu Zhong et al. “Privilege escalation detecting in android applications”. In: *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE. 2017, pp. 39–44.

[201] Yajin Zhou and Xuxian Jiang. “Detecting passive content leaks and pollution in Android applications”. In: *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*. Bangalore India: Association for Computing Machinery New York NY United States, 2013, pp. 1–16.

Appendix A

Appendix A — SAPiC Syntax

SAPiC Syntax

Fig. A.1 describes the SAPiC syntax. The syntax allows to define a protocol as a process. It is then translated into a set of Tamarin MSRs that adhere to the semantics of the calculus, which is a dialect of the applied pi-calculus [1]. The calculus comprises an *order-sorted term algebra* with infinite sets of publicly known names PN, freshly generated names FN, and variables V. It also comprises a signature Σ, i.e., a set of function symbols, each with an arity. Messages are elements from a set of terms T over PN, FN, and V, built by applying the function symbols in Σ. Events in SAPiC are similar to Tamarin action facts, and they annotate specific parts of the process to be used to define security properties. We denote F the set of Tamarin action and state facts. As opposed to the applied pi-calculus [1], SAPiC's input construct $\texttt{in}(M, N);\ P$ performs pattern matching instead of variable binding. See [47, 115] for the complete details.

$\langle M, N \rangle ::= x, y, z \in V$	variables
$\mid p \in PN$	public names
$\mid n \in FN$	fresh names
$\mid f(M_1, \ldots, M_k)$ s.t. $f \in \Sigma$ of arity k	function application
$\langle P, Q \rangle ::=$	processes
$\mid \texttt{0}$	terminal (null) process
$\mid P \mid Q$	parallel execution of processes P and Q
$\mid !P$	replication of process P
$\mid \texttt{new}\ \sim n; P$	binds n to a new fresh value in process P
$\mid \texttt{out}(M, N); P$	outputs message N to channel M
$\mid \texttt{in}(M, N); P$	inputs message N to channel M
$\mid \texttt{out}(M); P$	outputs message N to the public channel
$\mid \texttt{in}(M); P$	inputs message N to the public channel
$\mid \texttt{if}\ Pred\ \texttt{then}\ P\ \texttt{[else}\ Q\texttt{]}$	P if predicate $Pred$ holds; [else Q]
$\mid \texttt{event}\ F; P$	executes event (action fact) F
$\mid P + Q$	non-deterministic choice
$\mid \texttt{insert}\ M, N; P$	inserts N at memory cell M
$\mid \texttt{delete}\ M; P$	deletes content of memory cell M
$\mid \texttt{lookup}\ M\ \texttt{as}\ x\ \texttt{in}\ P\ \texttt{[else}\ Q\texttt{]}$	if M exists, bind it to x in P; else Q
$\mid \texttt{lock}\ M; P$	gain exclusive access to cell M
$\mid \texttt{unlock}\ M; P$	waive exclusive access to cell M
$\mid [L] \text{-}[A]\!\!\rightarrow [R]; P \quad (L, R, A \in F^*)$	provides access to Tamarin MSRs

Figure A.1: SAPiC syntax. Notation: $n \in FN, x \in V, M, N \in T, F \in F$.