THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

A Multidimensional Analysis of The Android Security Ecosystem

Thèse présentée et soutenue à Biot, le 14/09/2021, par

Andrea Possemato

Rapporteurs	Prof. Patrick Gerard Traynor	University of Florida
	Prof. René Mayrhofer	Johannes Kepler University
Examinateurs	Prof. Yousra Aafer	University of Waterloo
	Prof. Antonio Bianchi	Purdue University
	Prof. Davide Balzarotti	EURECOM
Directeur de thèse	Prof. Aurélien Francillon	EURECOM
Co-Encadrant	Dr. Yanick Fratantonio	Cisco Systems Inc.

Abstract

With more than 2.5 billion active devices based on Android, Google's mobile operating system is now one of the most widely used in the world. This success is also due to the fact that numerous vendors contribute to its diffusion, producing numerous devices that are based on Android, but that offer several other modifications, both in terms of functionality and usability. Although this introduces diversity into the ecosystem and contributes to its expansion, from a security point of view, this also introduces challenges. In fact, having diverse entities contributing and modifying the code means that the final security of the device is no longer in the hands of a single entity, but becomes a multi-party effort. Despite all the efforts made by Google to constantly improve the security of the entire Android ecosystem, there are still several problems that remain unresolved. In this thesis, we analyse in detail some of the open problems that affect different components and players that are part of and contribute to the Android ecosystem. We start with the security analysis of the network communication of Android applications, showing how, even if Android provides several techniques to secure network communications, developers sometimes are still forced to use cleartext protocols. Our study continues with the analysis of another issue that puts the security and privacy of the user at risk. We analyze the vulnerabilities exploited by malicious applications to perform phishing attacks and how there is still no system in place to allow applications to protect themselves against these attacks. Last, we analyze what we think may be the perfect representation of how difficult it is to ensure security in a domain as extensive as Android analyzing how customizations, even though beneficial to vendors, can lead to security problems that are lowering down the overall security of the Android system. In this thesis, for each of the problems, we analyze the issue in detail, we measure how widespread it is, and we propose an alternative solution with the aim of solving the problem, making a step towards a more secure Android ecosystem.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

With its first appearance in September 2007, Android is today one of the most widely used operating systems in the world [osm21]. Among the many strengths that have allowed the success of this system, two are perhaps the most important ones. The first one is that its open source nature allows different vendors to produce devices based on Android. The second one, instead, is the wide availability of applications that the user can use. This success can also be seen in how products that were previously thought and designed only for use in Personal Computers, such as email clients or office suites, are now also present for Android. Moreover, services that were previously available only through web platforms (e.g., banking applications), are now also finding their place within the Android system. All this gives the user the ability to perform more and more operations from the comfort of their smartphones.

This migration of users from Personal Computers to mobile devices, has made Android a well explored and researched topic since the beginning, both by academic and industrial research, especially regarding security issues that might affect this new platform. Unfortunately, several problems identified over the years remain, to date, unresolved. Furthermore, the great spreading of the system among users has also interested cybercriminals who have started to hit this new mobile operating system [Zer21]. In fact, over the years, attacks that were previously used only to compromise PC and web services, have been slowly adapted to the mobile ecosystem, arriving even to create new types and categories of attacks that are specific to Android smartphones [bK21].

The appearance of these new attacks and problems have pushed researchers to study new mitigations aimed at improving the security and the privacy of users and devices, and presenting, over the years, numerous research efforts that have made more and more difficult the exploitation of Android devices. However, this increase in security measures has not shifted the interests of malicious actors from this platform: on the contrary, it has pushed attackers to research and employ increasingly innovative and complex techniques to compromise the security of the system.

Nonetheless, as much as the developers of the Android system try to create a hardened system, the evolution and complexity of the Android ecosystem has made it very challenging for Google alone to manage the security of the entire system.

Indeed, despite efforts to introduce mitigations at the level of the Android Open Source Project (AOSP), which is used by default by all Android-based devices, it was realized that it was not enough to guarantee the security of the overall ecosystem [Kra17]. In fact, the great number of actors

involved in the development of a single device, whether contributing with software or hardware, very often have little to do with the security proposed by AOSP. At the same time, each of them contributes to the final security of the device. It is thus necessary that the developers of each component understand the security issues their component is subjected to, and correctly implement all the necessary security measures to ensure that the whole system cannot be compromised.

In the Android ecosystem, therefore, the final security of the device and the user, no longer becomes an effort made only by the main AOSP system developers, but it becomes a collective effort that includes all the actors that play a role in the entire supply-chain of the device: we thus argue that Android system security must be approached from a perspective of a complex and multi-player ecosystem [wis19].

This, unfortunately, has also given rise to several challenges: these different organizations, in fact, despite collaborating in the development of the same device, might approach security in different ways, might have different threat models, and problems that forces them to make decisions that may not see security as a key and fundamental element [ZLZ+14]. There are cases in which the security takes a back seat, as developers prefer usability, functionality, and beating the market. Many times, introducing new security features or fixing vulnerabilities can require investing significant time in refactoring code. Moreover, introducing these new security measures can make the application not compatible with older versions of Android. Investing in the security of the application therefore can be seen as an investment that slows down the application's release time, and potentially causes a loss of users: this is when security might become an obstacle, not a feature.

Ensuring that security is enforced and respected at each step of the entire supply-chain is becoming an increasingly widespread problem. Even though these issues have been extensively analyzed over the years, and numerous papers have been published, this problem still remains unresolved, and still hides many challenges and problems that have never been previously studied.

To solve these problems, we first need to fully understand the issues that prevent the developers to approach security in the right way, and how these problems can be solved by considering the issues from developers' perspective. This is the only way to create security measures that will allow developers and all actors involved in the Android ecosystem to sense security as an integral part of the product, and not as a barrier.

1.1 Problem statement

To date, the Android Operating System is one of the most used mobile systems on the market, and it has reached, over the years, more than 2.5 billion devices built by more than 1,300 different vendors [osm21, wis19]. The great success that has made Android one of the most popular systems is also due to the large number of applications that can be found in the official and unofficial stores. Nowadays smartphones allow users to perform operations that were previously available only on desktop computers.

Unfortunately, the shift of platform by users has pushed attackers to adapt and create attacks that were previously performed on PCs to mobile platforms. The interest of attackers in mobile platforms makes it clear how important it is to study the security of these systems [Zer21, bK21].

As the first goal of this thesis, we set out to understand and analyze how some of these attacks have evolved and adapted from the web and PCs to mobile devices. We decided to analyze, specifically, two major issues that still affect several components of the Android platform: networking attacks and vulnerabilities that allow attackers to mount phishing. We investigate how the Android system identifies and mitigates these attacks, whether applications have mechanisms in place to block them, and what are the security and privacy risks for the end user.

As mentioned above, the Android ecosystem involves numerous entities that contribute in different ways to the development of the device and its operating system, providing components ranging from software to hardware. Because of this complexity, the security of the system is not in the hands of a single entity, but requires an effort from all parties involved. However, each of these players that contribute to the ecosystem have different needs and they approach security differently. There may be situations where security cannot be considered unless the developers spend considerable time on it. Very often this clashes with the objective to introduce a new product on the market, with the shortest time-to-market possible. Thus, application developers are sometimes put in a position to choose between usability and security, and unfortunately, in certain scenarios, the two cannot be taken simultaneously, forcing the developers to prefer the former at the expense of the latter.

Therefore, as a second objective of this thesis, we analyze how the approach to security changes depending on the component and layer that is taken into account, highlighting how, even today, there is a lack of defense mechanism that allow the various actors to consider security not as a feature to be enabled when needed, but as an integral part of the whole system.

One of the additional strengths of the Android system is its openness. This allowed its rapid diffusion, allowing it to reach significant portions of the mobile market. In particular, this gave the possibility to other vendors to customize the system entirely, but this also introduced significant security issues. It is in fact not uncommon to find vulnerabilities that affect proprietary components introduced by vendors, or that affect only certain devices for which the vendor has not correctly applied the security fixes published by Google in its monthly security bulletin: this problem is also known as "fragmentation," and has been afflicting the Android ecosystem for years.

As the last goal of this thesis, we analyze in detail the evolution of this issue over the years. We uncover how the entire supply-chain of an Android-based device is much more complex than we previously believed, and we show how the actors contributing to the ecosystem might have a negative impact on the security posture of the final device.

Each of the analysis we performed has highlighted numerous challenges and brought to light issues that were not known nor addressed by previous work. This thesis presents the path we took to address these issues, showing how we tackled them and how we tried to solve the challenges that, at reporting time, were still open.

1.2 Contributions

The research we present in this thesis has been guided by several key questions. After having identified a specific interesting problem, one aspect we wondered about was how widespread the problem was in the real world. Determining the extent and spread of the problem also helps us understand, potentially, how many devices and users are affected by a given issue; The more widespread the problem, the more important it is to try to solve it. We therefore focus on the problems that affect the Android ecosystem and that potentially put millions of users at risk.

Understanding the extent of the problems is just the first step. As a second step, our research has looked for problems for which AOSP and Google have potentially provided solutions and mitigations, but that nevertheless continue to affect the ecosystem. Understanding why developers did not adopt available defense mechanisms to mitigate a specific threat poses interesting research questions. Furthermore, for each of these defenses, we tried to identify how much developer's effort was required. Indeed, leaving the developer the possibility to configure these components can lead to numerous problems, such as adopting of configurations that are too permis-

sive, or in the worst case, opening up devices to critical security problems. These issues were also the focus of our research, and we attempted to quantify how widespread the problem of incorrect security configurations within Android systems was. This thesis, however, is not limited to just conducting large-scale analyses to measure the extent of these issues: the last direction we have set to explore was to make the system safer, thus raising the bar for attackers that want to compromise the system. Alongside the measurements, this thesis documents a significant engineering effort aimed at making at developing new security measures and at making the current security measures more security, effective, and practical.

In the following paragraphs, we illustrate the three principal contributions of this thesis. Each contribution is presented in the form of an analysis of a security problem that affects the system, and aims to study and propose new approaches to mitigate the problem.

The first problem we address relates to the security of Android application, with a focus on their network communications, and how they can be exploited to compromise the integrity of an application or the entire system. The security of Android applications has been the subject of numerous works, and is a topic that has been studied extensively over the years. Networking issues are still of fundamental importance since, nowadays, virtually all mobile applications rely on network communication to exchange sensitive data with their network backend. Thus, the number of applications, and consequently users, that might be impacted by this issue is potentially significant. Unfortunately, despite the efforts and improvements made to the operating system to mitigate these issues, some problems are still unresolved: to date, developers still adopt weak and potentially vulnerable network security configurations and settings.

As a first contribution, we present the first comprehensive and large-scale study on the recently introduced Android network security mechanism, called Network Security Policy. Our study identified several strengths but also numerous common pitfalls that might occur, and shows that the root causes leading to weak policies can be attributed to the adoption, by the application developer, of external components—like third-party advertisement libraries—that encourage and require unsafe practices. Since the developer cannot control or change the security of these components, they must use these external libraries as they are, even if they are potentially vulnerable. This reflects the problem discussed above: current incentives push developers to choose functionality over security. This study identified as one of the key problems the coarse-grain nature of the security policy, which does not allow developers to set different policies for different components. To this

end, we propose a drop-in extension to the current Network Security Policy format that allows developers to comply with the needs of third-party libraries without weakening the security of the entire application.

As a second contribution, we study how the Android framework protects applications from attacks that have evolved and adapted from the web and moved on to affect mobile systems, focusing our analysis on "phishing." The effectiveness of phishing attacks on the web have led attackers to adapt and improve it to make it effective on mobile platforms as well. The problem of malicious applications mounting this attack has been deeply studied over the years, both in the realm of web security as well as in the realm of mobile security. Phishing attacks are particularly problematic for mobile platforms because it is not possible, nowadays, to provide enough information for a user to reliably distinguish a legitimate application from a malicious app spoofing its user interface. Despite all the efforts, detecting and preventing phishing attacks has not been completely solved yet, thus making this a good research problem. We first studied in detail the various techniques used by the attackers, and we noticed how a common pattern is to abuse vulnerable APIs that "leak" the device's state (e.g., which app the user is currently interacting with). To this end, we aimed at identifying all vulnerabilities exploited by malware to mount phishing attacks. As a result, we were able to identify 18 new vulnerabilities that can be abused by malicious actors, leading to 6 CVEs. Despite being an important first step towards eradicating this issue, automatically identifying these vulnerabilities is not possible: the complexity of the system and its continuous evolution make it difficult to catch all bugs in the general case. Hence, since the Android system does not provide a method for applications to protect themselves from the phishing threat and it does not have a mechanism that detects these attacks at runtime, we envisioned a new detection system that identifies and blocks certain categories of phishing attacks at the moment they occur.

As a third contribution, we bring our attention to the security posture of the entire Android ecosystem. Because of the way the ecosystem has expanded and evolved over the years, security now has to be analyzed by considering the perspectives of all its components and players. We believe that the Android ecosystem has evolved so much that it is important to analyze how individual entities, such as vendors, implement the various security mechanisms that are available and introduced on AOSP. In fact, the mere fact that AOSP introduces or implements certain security features does not guarantee that vendors will use them in their final products, or that they will configure them correctly. All the efforts made over the years to ensure a secure system become meaningless if they are not correctly

implemented and enabled by all the nodes of the chain that contributes to the final device.

This perspective pushed us to study how the security of the ecosystem depends on each of its components, and we show how the remaining components that contribute to the whole supply-chain of a device approach security. We perform this by analyzing in detail the largest and most significant component, the one identified by the various Android vendors. In this study, we find how the fragmentation of the ecosystem has repercussions in terms of adoption and configuration of various security mechanisms. This thesis presents the results of our study, and how this problem may affect the entire security posture of the vendor system. We performed the first longitudinal and large-scale analysis of non-Google devices aimed at understanding the security repercussions of system and framework customizations. Our analysis takes an in-depth look at two key aspects. The first one is the "compliance," which checks whether a certified system actually follows the rules dictated by Google, while the second one relates to how customizations may affect the security of the overall device. We uncover how vendor-specific components significantly lag behind with respect to the security posture of the main Android Open Source Project.

This research allowed us to explore different issues related to security in the Android ecosystem. The continuous shift in viewpoints made it possible for us to analyze how different actors that contribute to this system sense and approach security differently. Our research showed that there is still a gap, in terms of security measures and practices, for all the players contributing to the development and growth of this ecosystem to approach and sense security as an integral part for the components they develop, and not as an obstacle. Although there is still a lot of work to be done to make the Android system completely secure, we believe that this thesis has taken a step towards this direction, bringing to light new problems, solving old ones, and proposing new security systems and models that can allow the various entities of the ecosystem to increase their level of security.

1.3 Thesis outline

The thesis consists of a total of 6 chapters.

Chapter 2 presents the background and how we envisioned the partition of the ecosystem into layers. For each layer, we describe its role within the ecosystem, what security issues may affect it, and how security is approached.

In Chapter 3, we start by discussing a specific security issue which affects Android applications. Indeed, to better understand how the security is approached amongst the application developers, we use as example the security of network communications. In this chapter, we describe in detail the recent introduction of the Network Security Policy (NSP). NSP is an XML-based configuration file that allows applications to define their network security settings without defining any code that implements these configurations. Furthermore, we perform the first comprehensive study and large-scale analysis on this new network defense mechanism, identifying its strengths and common pitfalls. Last, we propose a new drop-in replacement of the Network Security Policy that solves the problems and limitations we identified.
This chapter is based on the publication *"Towards HTTPS Everywhere on Android: We Are Not There Yet"* [PF20].

In Chapter 4, we present a comprehensive study on the issue of phishing attacks on Android. This is a problem that the entire Android system and framework has been facing for years but remains unresolved. To properly understand the problem and identify its root causes, we start with a systematic study of the vulnerabilities exploited by malware to mount this attack during the years. We identify one of the core and active issues: vulnerable system APIs that "leak" the device's state. Those APIs provide the attackers with numerous privileged information that allow them to mount more effective phishing attacks. To address this class of vulnerabilities, we devised an automatic system that can assist system developers in identifying these APIs automatically. However, we faced numerous issues and challenges in this process, and we describe them in the remainder of the chapter, showing how some of these challenges are still unresolved. We conclude the chapter by presenting how the experience gained in trying to automatically identify these vulnerabilities, allowed us to devise a new on-device detection system to identify phishing attacks at the moment they occur.
This chapter is based on the publication *"Preventing and Detecting State Inference Attacks on Android"* [ADY21].

Moving on to Chapter 5, we present the last main contribution of this thesis. In this chapter we present the analysis on security implications and repercussions of "fragmentation" of the Android ecosystem. We illustrate the security issues that can be introduced by the changes made by vendors, and how they can lower the security of the system, either by reintroducing old issues already solved, or introducing new ones. Furthermore, this chapter presents the first longitudinal study on Android OEM customizations and shows how numerous Android-based devices do not meet the security

prerequisites defined by Google and significantly lag behind with respect to the security posture of the main Android Open Source Project.
This chapter is based on the publication *"Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization"* [ASDY21].

Chapter 6 concludes the journey of our thesis. In this chapter, we present future work and research directions that we hope will be pursued, thus making the entire Android ecosystem more secure. Moreover, we describe ideas for improvement on the research we presented in this thesis, which could solve limitations or challenges that still remain open despite our efforts. We conclude this thesis with a summary of the challenges we undertook, how we solved them, and how we made a step towards a more secure Android ecosystem.

Chapter 2

The Layers of Android Security

When we think of an Android smartphone and operating system, the big and known brands come to mind, such as Google, Samsung, Sony, and LG. However, the Android ecosystem is a lot more complex than one may think: in fact, it is made up of numerous layers and actors that cooperate with each other to produce the final smartphone. Although these entities have important and different roles within the smartphone supply chain (a term with which we indicate the entire process leading to a device, from manufacturing to software customizations), they are not often analyzed in detail and are not, to date, the main subjects of research activities. Each of these layers, contributing differently with hardware and software components to the final product, offers numerous analysis insights, especially regarding their impact and implications on security. In fact, from the security perspective, each of the layer we identified introduces a potential additional attack surface that attackers can try to exploit to compromise the device. Since these layers have to face very different security challenges, it is important to analyze them independently: as we will present next, they vary profoundly in the way the developers contributing to a given layer approach and view security, in how they can mitigate a security issue, and how they can improve the security of the overall layer.

We now introduce the various layers we identified, and we describe how they are involved in the supply chain of an Android device. There are many ways this discussion can be organized: for this thesis, we organize the ecosystem by "software layers." We identified three major layers: the *Application Layer*, which consists of third-party apps normally used by the user, the *Android Operating System Layer*, which is the base Android Open Source Project used as main codebase by vendors to build their system, and, to conclude, the *Vendor Layer*, which represents the vendors that package and sell the smartphone. For each of them, we illustrate their functionality, the components they manage, how they approach security, what issues they might introduce, and their impact in terms of devices and users in case of compromise.

2.1 The Application Layer

Among the various strengths that have allowed the expansion and success of Android, an important role is played by third-party applications. If one considers that in the fourth quarter of 2020, only in the Google PlayStore, Android users were able to choose between 3,14 million different applications, it is easy to understand the importance of guaranteeing the security for this layer. To prevent malicious applications from compromising the sta-

bility and integrity of the device, applications are executed in a protected and sandboxed environment, which limits and controls their capabilities. In addition to protecting the underlying system, this sandbox also introduces mechanisms that limit the ability of malicious applications to affect the security of other apps.

Nowadays, some of the applications are pre-installed on devices, and normally, these cannot be uninstalled, but (in some cases) only disabled. Others, instead, like the very popular WhatsApp, Instagram, Facebook, and TikTok can be freely installed by the user. To understand, in terms of numbers, the potential amount of users that may be affected in case a vulnerability impacts one of the applications mentioned above, consider that these applications alone have more than one billion active unique downloads and millions of daily users. The great success and evolution of Android has also meant that many companies that previously developed programs for personal computers, started to offer the same application on mobile systems. This allowed users to perform tasks on smartphones, even those that were previously available only on PCs, such as email and online banking.

The success and popularity of these applications have also attracted the attention of attackers: over the years, it has been often possible to find vulnerabilities in these applications, that, once exploited, have allowed attackers to compromise the integrity of the application itself, and in some cases, to use them as entry point to compromise then the entire device.

Furthermore, it is interesting to analyze how the complexity of these applications and their interaction within the system has introduced new security issues. The applications, being constituted by numerous components that interact with each other, with the system, but also with other installed applications, require numerous efforts to be configured correctly and to avoid potentially dangerous interactions. This problem applies both to interactions that are initiated by the application in question, but also to those that are initiated by external apps, thus needing to manage security in both directions. This complex system of interactions among applications and system was found to be an important source of security problems that attackers started to study. Attackers, in fact, started to target these specific components of Android applications, creating new attack scenarios that were previously unknown and for which numerous security mechanisms had to be studied and implemented to mitigate them. Among the most known Android-specific attacks, we find for example the exploitation of components that should not have been exported and reached by other applications, such as Content Providers or Broadcast Receivers, or vulnerabilities that afflict the configurations and implementations of WebView and JavascriptInter-

face.

But the problems do not stop there; on the contrary, in addition to these new attack surface(s), it is often possible to find known vulnerabilities that are simply "ported" to work on smartphones. In fact, since applications can contain components of native code, usually written in memory-unsafe languages such as C and C++, all the problems that come with these languages are also present on the application layer, allowing the attacker to use old techniques to exploit, for example, memory corruption bugs. Moreover, web attacks can also be exploited to compromise the security of an Android application. Among the most popular examples, we can find problems related to Cross Site Scripting (XSS). This attack, usually exploited on websites, has been adapted to work also mobile, attacking WebView components of an application: simple XSS have allowed, over the years, to compromise the security of entire applications, allowing attackers to mount remote code execution attacks. This was, for example, the case for CVE-2013-7201 and CVE-2013-7202, which, when combined, allowed attackers to remotely execute code within the PayPal application for Android [Lab14]. This attack also exploited additional vulnerabilities, such as an incorrect management of the validation of SSL connections. In this specific instance, the developers wrongly configured the component handling SSL errors for the WebView: upon identifying an error in the SSL validation procedure, the logic of the application would let the HTTP flow continue, instead of stopping it and raising a warning about a potential network hijack. The bypass of the SSL validation check allowed the network attacker to inject malicious Javascript in the application, and achieve remote code execution by exploiting an exposed JavascriptInterface within the vulnerable WebView.

Unfortunately, identifying, testing, and fixing these vulnerabilities takes significant time, and very often this is at odds with time-to-market goals: For application developers, it is often more important to publish new versions of the application with new features than ensuring that they do not have introduced new vulnerabilities. Going back to our HTTPS-related example, there is often the wrong conception that, if the application communicates with a remote server in HTTPS, it is safe and cannot be manipulated. And for this reason, these issues are often not even considered as security problems: it is in fact interesting that, at the time of the bug being report, PayPal was not even considering networking and SSL attacks as security problems.

By undermining the importance of these attack scenarios, which in this case proved to be very dangerous, does not help aligning developers' incentives to the security-related incentives, especially when considering that

testing for these vulnerabilities can be quite time consuming. In fact, in this case, a developer would need a proper test environment to simulate a remote attacker, hijack the SSL connection (in many assumed to be safe) and inject malicious code. All these testing procedures, unfortunately, take time, and the identification of eventual vulnerabilities risks prolonging the time of publication of the application; a risk that is very often not taken. Moreover, from Paypal's technical report, it is also possible to evince that the vulnerable JavascriptInterface component that were exploited to achieve code execution was not even essential for the app's functionality, and it had been already flagged as a component to be removed. Very likely, that functionality was replaced with a new implementation, but developers forgot to remove the legacy code, thus unnecessarily opening the app to vulnerabilities.

These issues affect in-house components that developers have access to, but they are also posted by applications containing proprietary components for which source code is not available (i.e., external libraries) and for which identifying and mitigating security issues is very challenging. There are a number of problematic scenarios: for example, a developer may face situations in which the external library contains vulnerable or outdated code and non-secure default configurations, possibly because the library itself is not maintained anymore. To make things worse, if such a library plays a key role in the application and is integrated with all other components, the developer may be "forced" to use it, despite the fact that it may introduce security problems and open the app to attacks. These problems and challenges often bring application developers in front of a choice on what to prioritize, whether the app's security or functionality of the application, and, oftentimes, the current incentives (e.g., time-to-market) unfortunately do not allow to treat both aspects with the proper importance.

2.2 The Android Operating System Layer

Virtually all Android-based devices use the source code provided by the Android Open Source Project (AOSP) as a foundation on which to base their operating systems and customizations. The lead and core contributor for AOSP is Google: its role is to oversee the engineering process for the core framework and platform, as well as managing the overall direction of the project. However, there is nothing that prevents external developers to contribute to the code of AOSP, by, for example, sending patches to fix usability or security problems. From a security standpoint, being the base reference for potentially all Android-running devices, AOSP is of critical im-

portance for the health of the ecosystem, and there are numerous challenges to overcome.

To begin with, the impact of a vulnerability affecting a core component developed in AOSP may potentially affect *all* devices using such vulnerable versions. Thus, depending on when the vulnerability was introduced, identified, and fixed, the number of compromised devices may vary. Naturally, the opposite is also true: a patch, or a security measure, introduced in AOSP can potentially mitigate security issues and improve the overall security posture of a large number of devices. These are just some of the reasons why the AOSP layer plays a key role in managing the security of Android devices. Given its importance, it is clear that security cannot be seen, as in the application layer, as something optional. On the contrary, one of the strengths of the AOSP project is to have been developed throughout the years with security as a core design principle. However, developing a system that is secure *and* functional often present many technical challenges. In fact, AOSP must guarantee, for example, the security of the applications and the integrity of their data, but at the same time it must allow applications to interact with each other, offering the possibility, when necessary, to modify data that would normally be protected. As long as we consider the interaction between benign applications, security issues normally do not arise. Unfortunately, the execution of malicious applications that aim to affect the other applications (to steal sensitive data, for example) or the security of the underlying system is a possible scenario that needs to be considered. Therefore, the security of AOSP becomes crucial at multiple levels: its security must guarantee, and protect, interactions that could be harmful, without affecting the usability of the entire system.

The approach to security changes significantly when compared with that of the application layer. Since securing Android is essential, mitigating and fixing security vulnerabilities is considered a top priority, and developers have strong incentives to properly deal with the security aspects and prioritize it over time-to-market of new features. This "security push" is reflected by the constant evolution of AOSP security over the years. The Android system has in fact seen a continuous evolution, with new defense mechanisms and features being introduced with each new version of Android. In many situations, these changes have (successfully) forced application developers to update their codebases to be considered "compatible" with the versions of AOSP.

These changes over the years have been very frequent, and continue to be so today. Among these, one of the most important changes that required the refactoring of the code, was the new model of permissions management.

This new approach saw the introduction of runtime permissions, migrating from a model that assigned all permissions at installation time, to a model in which the app must explicitly prompt the user to ask for each "dangerous" permission. This new model allows users to take more informed decisions and to be more aware of potential risks of granting a given permission to the application.

Other Android versions introduced substantial changes regarding the Application Programming Interface (API). While changing APIs and breaking backward compatibility is often seen as a last-resort when comes to updates, in some cases they have proven to be necessary. In fact, over the years, several APIs have been found to have design vulnerabilities or default configurations that were too permissive, and many of these updates have required backward-incompatible changes, potentially risking to introduce unexpected behavior in the applications that use them. For instance, this was the case for several APIs defined in WebSettings, an important class that manages WebView configurations and security. One example is `setAllowFileAccessFromFileURLs`: this API specifies whether Javascript code, executed within a `file://` or `content://` scheme should be able to access other resources with the same scheme. The default value for the API was `true` for API level 15 and below, but, however, this default configuration turned out to be too permissive and prone to problems like file-based XSS or data stealing. Thus, starting from API level 16 and above, the default value was changed to `false`: this change forced apps using this API to properly update their code; And, starting from API level 30, this API has been deprecated.

These two examples of "breaking" changes discussed above are just two of the many security improvements that have required developers to actively update their code base. Among the latest features introduced in AOSP, for instance, we find a radical change in the management of file permissions on the external storage, but also a new network policy that, by default, prevents applications from communicating via HTTP, as it is an insecure protocol susceptible to MITM attacks. These protection mechanisms not only work at the application level, but are also made available to other system components, which operate at a lower, more privileged level.

These many changes witness that AOSP gives security aspects much higher importance with respect to generic apps. In turn, AOSP provides developers with a number of defense mechanisms to protect their applications: it is then up to the developers to decide whether to make good use of them or not, or whether to implement "quick solutions" that bypass them altogether.

2.3 The Vendor Layer

This last layer, the vendor layer, is perhaps the most complex one, as it consists of numerous contributors to its development, both in terms of software and hardware. This is the layer that incorporates many components of the entire smartphone supply chain, which are often not taken into account, but are very important for the final security of the device. We now present, in order, the actors that work from the lowest to the highest level, starting from the Chipset Manufacturers, then the Original Design Manufacturer, and finally, the Original Equipment Manufacturer, that is the final node of the chain, i.e., the company that puts the brand and sells the final product.

2.3.1 Chipset Manufacturer

Starting from the lowest level, the hardware, a mobile device is usually equipped with a Central Processing Unit (CPU). Between the most known brands that produce CPU, in the Android world, we find companies like MediaTek and Qualcomm. These companies can be considered "generic," as they provide processors for multiple vendors. Other processors, however, may be designed and built directly by the companies that sell the final device. Among these, for example, we can find the Exynos processors, created directly by Samsung just for specific categories of devices, or the Kirin series processors produced by HiSilicon, a company that produces processors for Huawei smartphones. In addition to hardware, these companies also provide software components that usually run at a privileged level, such as firmware or kernel drivers. Although the amount of code introduced by the various chipset manufacturers constitutes a very small portion (especially when compared to the entire Android system), it is not free from vulnerabilities that, if exploited, can be used to compromise the security of the entire device. Moreover, these vulnerabilities are particularly important because they potentially allow to compromise devices from different vendors that are using the same chipset. To make things worse, it is difficult for the devices vendors to mitigate issues on their own, since they would need to wait for the chipset manufacturer to fix any vulnerabilities. These issues, unfortunately, are more common than one might think. Just as an example, in 2019 it was discovered an issue that affected most of the chipsets produced by the company MediaTek [Rah20]. As mentioned earlier, MediaTek provides processors for numerous devices, including, for example, some Samsung smartphones. The vulnerability, an arbitrary read and write of physical memory addresses, once exploited, allowed a local unprivileged attacker to mount a privilege escalation and gain root access (temporary —

i.e., not surviving reboot) on the device [Bel20]. This problem (identified as CVE-2020-0069) has been exploited for many months [LLC20a] before it was fixed in 2020.

2.3.2 Original Design Manufacturers

The hardware component, however, is not only the responsibility of the chipset manufacturer: another important layer deals with low-level hardware components such as the design of the entire board and the sensors (e.g., GPS, NFC), and it is defined by the various Original Design Manufacturers (ODM). These companies usually design smartphone models and hardware for numerous brands. A 2020 analysis showed that five Chinese companies (Huaqin, Wingtech, LongCheer, CNCE (Chino-E), TINNO) are responsible, alone, for more than 85% of the market, providing their services to the most popular brands that make up the Android landscape [nok21]. For example, Wingtech alone has been, between 2019 and 2020, the ODM of reference for the brands of Samsung, Huawei, Xiaomi, Oppo, Lenovo, LG, and Nokia. Also in this case, as for chipset manufacturers, the ODM can introduce different code that normally run with elevated privileges. Unlike chipset manufacturers though, ODMs can also introduce SELinux policies, native libraries, and applications [LLC20b]. The importance of ODMs, and the code they introduce, is reflected in the new structure of Android: starting from Android 10, the code added by ODMs can be optionally placed in a dedicated directory `/odm`. As with chipset manufacturers, this code can contain vulnerabilities that, if exploited, can have serious effects and compromise the security of the entire device. Considering that one ODM alone is used by seven different brands, the percentage of devices that would be exposed to a given vulnerability could be very high. For example the vulnerability known as CVE-2019-15340 affects several devices manufactured by Huaqin. As another example, which shows that these vulnerabilities are not as rare as one may thing, this ODM released, together with the device, an application vulnerable to confused-deputy attack named `com.huaqin.factory`, which allowed a local attacker to bypass the restrictions imposed for the arbitrary management of low-level components such as WiFi, Bluetooth, and GPS, allowing an attacker to manage these peripherals without the corresponding permissions. To make things worse, the application, being pre-installed and executed as *system*, could not be uninstalled by the user [oST19]. For this specific vulnerability however, it is interesting to note that, despite Huaqin being an ODM that collaborates with many brands (e.g., Samsung, Huawei, Xiaomi, Oppo, LG), the vulnerable application was found only on Xiaomi Redmi series. This shows that,

within a single layer, it is important to track and analyze the specifics of various customization efforts to accurately assess the security of the entire ecosystem.

2.3.3 Original Equipment Manufacturers

Moving to a higher level, and shifting from hardware to software, the situation becomes even more complex and the number of potential players increases. Devices produced by ODMs are normally branded with the names and labels of the Original Equipment Manufacturer (OEM). As of today, the Android ecosystem counts more than 1,300 brands producing Android-based devices. Among these, we have Samsung, LG, Lenovo, HTC, and Huawei. In particular, HTC was the OEM chosen by Google to launch Android: In September 2008, the HTC Dream was the first device commercialized running the Android Operating System. An OEM can make many changes, touching practically all the abstraction levels of the system, starting from code changes of the bootloader up to graphic changes of the entire UI of the system. An OEM can introduce changes in several areas, which we can grouped under three categories: kernel, system, and applications.

Kernel: Changes to the kernel can involve many aspects. A vendor can introduce changes to introduce new features, can modify components that are already present (for Android, these changes can be performed, for example, on Android-specific components like Binder or the "Paranoid Network"), or can introduce new security mechanisms. Among the OEMs that have made the most significant changes to the kernel, we find Samsung, which added a series of security patches that take the name of *Real-Time Kernel Protection* (RKP) [GB17]. These modifications have changed significantly the internal workings of the kernel, adding protection mechanisms from exploitation techniques such as Jump and Return Oriented Programming (JOP and ROP), but also protecting the kernel against Data-Only attacks. Moreover, Samsung was among the first vendors to adopt a custom solution for implementing Control Flow Integrity (CFI) [Ada21]. These changes have definitely and significantly raised the bar for attackers, making it harder to compromise the Samsung kernel or made it almost impossible to reuse public exploits out of the box. Unfortunately, over the years, all these mitigations have been bypassed, always allowing the attacker to reach root privileges and compromise the kernel [She17].

At the same time, however, it is fair to point out that changing signifi-

cantly a complex codebase such as the Linux Kernel (used in Android) can increase the time needed to patch a vulnerability in the original codebase. This, in turn, can significantly increase the prevalence of an issue known as "patch gaping,", which indicates that there is a delay between "a security patch is merged into the source tree" and "the security patch is deployed on devices". Patch gaping is a problem because an attacker could monitor projects' source trees for security-related patches, and could attempt to write an exploit before the patch reaches the end-user devices.

System: The second area where OEMs make changes is the system and framework component. This component is very large and can include many layers of the Android system, starting from the native libraries up to the framework itself or even the entire system graphics and UI. The changes to the system are perhaps the most visible to the end user. However, some vendors often make so many changes to the internals and graphics of the original AOSP code that the final product hardly resembles the original one, thus resulting in very different custom forks of Android-based systems. Among the most important forks are ColorOS [Opp13] from Oppo, MIUI [Xia10] from Xiaomi, and EMUI [Hua12] from Huawei.

From a security point of view, the issues that arise in these situations are multiple, and very often difficult to be managed correctly. As a matter of fact these OEMs have to manage the security of two very different codebases. This introduces many problems, the biggest of which is the proper management of security updates. The same issue that affects the Kernel also affects this component: the more the code base of Android is modified, the more difficult it is to apply automatically and in a timely manner the security patches provided monthly by Google. Furthermore, properly managing patches on such a large and modified component opens up other issues: in fact,Nohl et al. [KN18] showed how, in 2016, only 17% of Android-based devices were fully updated with the latest set of patches. Moreover, Dai et al. [DZJ+20] showed how vendors fail to properly patch, sometimes forgetting to fix all vulnerable components and then applying all proposed and available patches.

Application: OEMs also make numerous changes at the software application level, introducing numerous third-party applications or services to implement specific services and features. Those applications are very often deeply integrated in the system and executed in privileged contexts. Unfortunately, the same problem that affects the previous modified components also affects the applications. One of the best known and problematic

examples at application level, which includes many problems at security management level, is the one related to "Custom Browsers," based on forks of the open-source Chromium project, provided by vendors in their devices as default browser application.

Vulnerabilities on browsers are particularly dangerous because of the threat model to which they are exposed, opening the device to remote attacks. In recent years, these components have been subject to numerous attacks, some of which have exploited the patch gap issue mentioned above, making known vulnerabilities (so called *n-day*) effective. Moreover, in 2020 Google detected a sophisticated Android hacking operation that used a combination of both unknown (so called *0-day*) known vulnerabilities in Chrome and Android to remotely compromise a device [Zer21].

Instead, vulnerabilities that exploited new components introduced by vendors, are for example CVE-2018-20523 [Det18] and CVE-2019-10875 [Det19]. Both these vulnerabilities affect the default custom browser of Xiaomi devices. The first bug allowed an attacker to exfiltrate the entire search history of the browser, exploiting a vulnerability in a content provider (an Android-specific component), while the second bug allowed URL spoofing due to incorrect parsing. These are just some of the most recent examples of how these components can be attacked to affect the confidentially and integrity aspects of modern smartphones.

When it comes to third-party services, however, these can vary greatly depending on the OEM and the various partnerships that are created. In fact, vendors can decide to rely on external companies to introduce specific functionalities within their devices, reusing existing technologies. While this practice potentially shortens the time-to-market for the vendor, it also potentially introduces numerous problems. In terms of security, depending on how the external component is delivered (e.g., as source code or binary form), there are two main potential scenarios for the vendor to address a vulnerability. If the vendor is in possession of the source code, it will be able to prepare a patch, in relatively short time. On the other hand, if the vendor has to wait for the patch to be provided by other companies, this would likely lengthen and delay the release time of a security update.

This is not just a theoretical problem. For example, Samsung had to face such a problem. In fact, to introduce system-wide support for a particular image format (Qmage), Samsung relied on the software company Quramsoft [Qur20] for everything related to this format. Quramsoft code was used for all the coding and decoding of this type of image. This code has been integrated into Samsung's system by changing all the components that manage the most common image formats, to integrate

support for Qmage. Unfortunately, the code produced by Quramsoft had many vulnerabilities in the image decoding process. These problems allowed Google Project Zero researchers to remotely compromise several Samsung smartphones, using Multimedia Messaging Service (MMS) as a remote attack vector [MJ20]. In fact, the changes made by Samsung to the components that handled multimedia formats widened the attack surface, adding and exposing remotely the (vulnerable) parsing of this new format. The several issues found in these proprietary libraries were collectively assigned CVE-2020-8899.

As we can see, the Android ecosystem is much more complex than one may initially think, with vulnerabilities potentially affecting a number of different layers. Having multiple players that contribute and manage important parts of code introduces many challenges, which, to date, continue to affect the security posture of the entire ecosystem. It is therefore important to analyze in detail these components, especially those that are often underinvestigated. This allows us to understand how the actors approach and view security, and makes it possible to develop new methodologies to increase the security of those components.

Chapter 3

Securing the Application Layer: the Networking Problem

3.1 Introduction

Nowadays, users rely on smartphones for a variety of security-sensitive tasks, ranging from mobile payments to private communications. Virtually all non-trivial mobile applications rely on communication with a network backend. These applications adopt different networking protocols to communicate with the remote servers. Given the sensitive nature of the data exchanged between the application and the backend, developers strive to protect the network communication by using encryption, so that network attackers cannot eavesdrop (or modify) the communication content. However, several works have shown how properly securing network connections is still a daunting challenge for application developers. The OWASP Mobile Top 10, a standard awareness document for application developers, shows how *Insecure communication* is still one of the most serious security issue for applications and does not only affect cleartext protocols but also encrypted connection performed over SSL/TLS.

As these problems can harm the safety of several million users in recent years, within the context of Android, Google has introduced several new network security features to tackle these problems at the core of the system.

For example, starting from Android 4.x, Android started to display alert information to the user if a "custom" certificate was added to the set of trusted CAs.

Later versions of Android, instead, started supporting two different repositories for CAs: the *System KeyStore*, which contains the "default" set of trusted CAs; and the *User KeyStore*, which contains custom CAs "manually" added by the user. This separation allows Google to make applications trust only the system CAs by default while performing secure connections.

From Android 6.0, Google started to push towards "HTTPS everywhere" even further, by providing the developer a mechanism to completely block any connection attempt with a cleartext protocol. It first introduced a new application attribute—that could be specified in the `AndroidManifest`—to specify whether cleartext (e.g., HTTP) connections should be allowed or blocked. It then extended these settings by introducing the *Network Security Policy* (NSP): this mechanism allows a developer, without modifying the application code and logic, to specify complex policies (with an XML configuration file) affecting the network security of her application.

Motivated by the significant security efforts that Google has made to protect users from network attacks, by these recent changes, and by their

potential security impact on the ecosystem, in this chapter we present the first comprehensive study on these new defense mechanisms. In particular, we first discuss in detail these new features, the attacks that are mitigated by the Network Security Policy, and the relevant threat models. We then highlight several security pitfalls that might occur when a developer is configuring the Network Security Policy: in fact, since the policy is defined by the developer and it is neither generated automatically nor verified, we identify some inconsistencies that the developer might introduce while defining the even simple policies, leading to unwanted behavior. We identified several patterns for which policies may provide a *false sense of security*, while, in fact, they are not useful. These policies might reassure a developer but in fact, when the policy is enforced, the developer does not get any security benefit.

Guided by these insights, we then present the first analysis of the adoption of the Network Security Policy on the Android ecosystem. This analysis, performed over 125,419 Android applications crawled during June and July 2019, aims at characterizing how developers are using these new features and whether they are affected by misconfigurations.

The results are concerning. We found that only 16,332 applications are defining a Network Security Policy and that more than 97% of them define a policy to allow cleartext protocols. Since starting from November 2019 Google changed some important default values related to Network Security Policy (and especially related to cleartext), we repeated the experiments over a fresh crawl of the same dataset (this time performed from April to June 2020): Our results show that, while more applications do adopt this new security mechanism, a significant portion of them still do not take fully advantage of it (e.g., by allowing usage of insecure protocols).

Guided by this result, we then set out to explore *why* applications adopt such permissive policies. Surprisingly, we found that many of these policies are simply copy-pasted from popular developer websites (e.g., StackOverflow), and we noticed that the practice of copying a policy is much more widespread than we expected. Upon closer inspection, we also found how many of the weak policies could be "caused" by embedding advertisement libraries. In particular, we found that the documentation of several prominent ad libraries *requires* application developers to adapt their policy and make it very permissive, for example by allowing the usage of cleartext within the entire application. While the Network Security Policy format provides a mechanism to indicate a domain name-specific policy, we found that the complex ad ecosystem and the many actors that are part of it make it currently impossible to adopt safer security policies.

Thus, as another contribution, we designed and implemented an extension of the current Network Security Policy, which allows developers to specify policies at the "application package" granularity level, and which solves a key conceptual problem of the Network Security Policy. We then show how this proposal enables application developers to embed ad libraries without the need of weakening the policy of the core application, how it is fully backward compatible, and how it can thus act as a drop-in replacement of the current version.

In summary, this study has advanced research in the area of application network security through the following contributions:

- We perform the first comprehensive study on the newly introduced Android network security mechanisms, identifying strengths and common pitfalls.
- We perform the first large-scale analysis on the adoption of the Network Security Policy on the Android ecosystem, using a dataset of 125,419 apps. Our study found that a significant portion of applications using the NSP are still allowing cleartext.
- We investigate the root causes leading to weak policies, and we found that several popular ad libraries and the complex advertisement ecosystem encourage unsafe practices. We systematically analyzed the compatibility of the Network Security Policy with the advertisement ecosystem, identifying key conceptual problems on the actual design of this defense mechanism.
- To mitigate this conceptual problem and limitation, we propose a drop-in extension to the current Network Security Policy format that allows developers to comply with the needs of third-party libraries without weakening the security of the entire application.

3.2 Network Communication Insecurity

This chapter explores the different threats that an application might be exposed to due to insecure network communications. We first present the problems related to the adoption of cleartext protocols, such as HTTP. Then, we discuss a number of threats that are relevant in the context of encrypted communication, as well as the ones introduced by wrong implementations of certificate pinning. We conclude this section by discussing

the possible security repercussions when trusting additional Certificates Authorities (CAs) different than the ones pre-installed on an Android device. For each ofthe issues,we also discuss the relevant threat models.

3.2.1 HTTP

An application using a cleartext protocol to exchange data with a remote server allows an attacker to mount so-called Man-In-The-Middle (MITM) attack. A MITM attack consists of an attacker monitoring the network communications between a client and a server: the data transmitted from both parties can then be, potentially, eavesdrop or even modified by the attacker. This, in turn, can lead to the compromisation of the user's private information or of the application itself [Lab13, AGoAR17].

The actual severity of this threat changes depending on the nature of the data exchanged by the application and the network backend. Several works showed how applications can put at risk the privacy of their users when sending, in cleartext, personally identifiable information (PII). However, sending data using an unencrypted channel is not the only cause of threats. A serious issue is also posed by an application which *retrieves* data from an endpoint in cleartext. Other works showed that, depending on the type of data exchanged, these MITM attacks lead to a number of other attacks, ranging from phishing attacks to remote code execution [Wel15, PFB+14].

An attacker can exploit the use of unencrypted and unsecured connections in the following context:

Threat Model 1. An attacker on the same WiFi network (or on the network path) of the victim can eavesdrop and arbitrarily modify applications' unencrypted connections and data at will.

To avoid these threats, the developer needs to ensure that *at least* all the sensitive network operations are performed over a secure channel.

3.2.2 HTTPS and Certificate Pinning

By adopting the "secure" version of HTTP, *HTTPS*, it is possible to perform network operations over a secure and encrypted channel. Exchanging data using HTTPS (SSL/TLS) ensures integrity, confidentiality, and authenticity over the connection between the application and the remote server. This mechanism works as follows. First, when an application tries to contact a remote server using SSL/TLS, a "handshake" is performed. During this phase, the server first sends its certificate to the client. This certificate contains multiple pieces of information including its domain name and a

cryptographic signature by a so-called Certificate Authority (CA). To determine whether the client should trust this CA, the system consults a set of hardcoded public keys of the most important (and trusted) CAs: If the certificate is signed (directly or indirectly) by one of these CAs, the certificate is then considered trusted and the (now secure) connection can proceed; otherwise, the connection is interrupted [52808].

While SSL/TLS is a powerful mechanism, it does not address all possible problems. In fact, HTTPS connections can be compromised by an attacker within the following threat model: While SSL/TLS is a powerful mechanism, it can be compromised by an attacker within the following threat model:

Threat Model 2. An attacker that can obtain a rogue certificate can perform MITM over HTTPS connections. We consider a certificate to be "rogue" when it is correctly signed by a (compromised) trusted CA without an attacker owning the target domain name. An attacker can obtain a rogue certificates using a compromised CAs [Ley11, Adk11].

Attacks within this threat model can be mitigated by implementing *Certificate Pinning*. Certificate pinning consists in "hardcoding" (or, pinning) which is the expected certificate(s) when performing a TLS handshake with a given server. From the technical standpoint, this "expectation" is hardcoded within the application itself, and the application can thus verify, during the handshake, that the certificate sent from the server matches with the expected one. With certificate pinning, the application is not relying anymore on the on the Certificate Authorities to verify the certificate, thus mitigating the threat posed by compromised CA. By using Certificate Pinning, even rogue certificates would not be enough to trick an application into performing an insecure connection.

Even though pinning is a powerful security mechanism, previous works have shown how it is very challenging to properly implement it. In fact, to implement pinning, developers are tasked to rely on a wide variety of libraries, each of which exposes a distinct set of APIs. Handling diverse implementations of pinning may push developers to take some shortcuts: It was shown how it is not uncommon for developers to rely on "ready-to-use," but *broken*, implementations of certificate pinning copied from websites like StackOverflow [FHM+12]. To make it even worse, these broken implementations often have a net negative impact, in some cases leading to accepting arbitrary certificates without even verifying which CA signed them, or whether the certificate was issued for the given domain. Moreover, it has also been shown how even popular network libraries themselves may fail to

properly implement pinning [Koz16].

3.2.3 User Certificates

The Android system comes with a set of pre-installed CAs to trust and uses them to determine whether a given certificate should be trusted. These CAs reside in a component named *KeyStore*. The system also allows the user to specify a *User KeyStore* and to install custom CAs. There might be situations where the custom CAs allow to perform a MITM over SSL/TLS connections, as we describe in Chapter 3.4. However, performing MITM over a secure connection should not always be considered a malicious activity. For example, proxies used to debug network issues rely on the same technique. Self-signed certificates generated by these tools do not have a valid trust chain and thus cannot be verified, and the application would terminate the connection. By adding a custom CA, applications can successfully establish a network connection.

Unfortunately, *User KeyStore* and self-signed certificates can also be abused by malware. Of particular importance is the emerging threat of "stalkware" (also known as "spouseware") [Gru19, Cim19]. Attacks on the device KeyStore can be exploited by an attacker within the following threat model:

Threat Model 3. An attacker that has physical access to the device can silently install a new custom certificate to the *User KeyStore*, and mount MITM (including on HTTPS connections) to spy the user's activities.

3.3 Network Security Policy

In the previous chapter we discussed how attacks against network communications may affect the security and privacy of both users and applications, and how it is not always straightforward to protect an application from these threats. To make the adoption and implementation of "secure connections" easier for a developer, Google recently introduced several modifications and improvements, which we discuss next in this chapter.

The first problem that Google tried to address relates to the installation of *self-signed certificates*. In very early versions of Android, it was possible to silently install one of these certificates, thus allowing anyone who controls it to perform stealthy MITM on SSL/TLS connections. In Android 4.4, however, Google introduced the following change: if a self-signed certificate is added to the device, the system would display a warning message informing the user that the *network may be monitored* and about the risks

and consequences of MITM on SSL traffic [Tea14]. The threats posed by self-signed CAs are still causing a serious problem for the security of the Android users: starting from 2014, in fact, Google monitored and identified "several hundred instances each day where users have installed a local certificate to MITM network connections." [Tea14]. However, since there might be scenarios where trusting a (benign) self-signed certificate is necessary (e.g., to perform network debugging), Google decided to split the KeyStore into two entities. The first one, named *System KeyStore*, is populated with pre-installed CAs, while the second one, named *User KeyStore*, allows the user to install self-signed certificates without altering the System KeyStore.

The second problem Google tried to mitigate is the adoption of *cleartext protocols* [Ale16]. Starting from Android 6.0, Google introduced a new security mechanism to help applications preventing cleartext communication, named *Network Security Policy* [AD]. With this new policy, an application can specify the `usesCleartextTraffic` boolean attribute in its manifest file and, by setting it to `false`, the application can completely opt-out from using cleartext protocols, such as HTTP. What is important to highlight is that this defense mechanism is not applied only to HTTP: all the cleartext protocols without TLS or STARTTLS — like FTP, IMAP, SMTP, WebSockets or XMPP are covered by this new policy [AD16].

Moreover, from Android 7.0, the new default is that applications do not trust CAs added to the User KeyStore [Bru16]. However, it is possible to override this default, but the developer needs to explicitly specify the intention of using the *User CAs* within the policy.

Note that, from an implementation point of view, the policy is *not* enforced by the operating system (as it would be impractical), but it is up to the various network libraries to actually honor it (e.g., by interrupting an outbound HTTP connection if cleartext traffic should not be allowed). Note also that, to address backward compatibility concerns, for an application targeting an API level from 23 to 27 (i.e., from Android 6.0 to Android 8.1), the default value of the `usesCleartextTraffic` attribute is `true`, which means that the policy would not enforce any constraint, thus allowing cleartext connections by default. However, if an application targets API level 28 or higher (i.e., Android 9.0+), then the default for that attribute is `false`, forcing developers to explicitly opt-out from this new policy in case their applications require HTTP traffic.

While this policy is a significant improvement, for some applications it may currently be impractical to completely opt-out from cleartext communications. In fact, this policy follows an "all-or-nothing" approach, which

might be too coarse-grained. This is especially true when a developer is not in complete control of its codebase, such as when embedding closed-source third-party libraries. In fact, these third-party libraries may reach out to remote servers using cleartext protocols or to some domain names that are not even supporting HTTPS. To allow for a more granular specification, with the release of Android 7.0, Google introduced an extended version of the Network Security Policy, which we discuss next.

3.3.1 Policy Specification

The new version of the Network Security Policy, introduced by Google in Android 7.0, has undergone a complete redesign [AD19]. The policy now resides on an external XML file and it is not mixed anymore with the AndroidManifest. The most interesting feature introduced in this new version is the possibility to specify additional network security settings other than allowing or blocking cleartext protocols. Moreover, to overcome the lack of granularity of the previous version, the policy now allows for more customizations through the introduction of the new `base-config` and `domain-config` XML nodes.

The semantics of these two nodes is the following: all the security settings defined within the `base-config` node are applied to the entire application (i.e., it acts as a sort of default). The `domain-config` node, instead, allows a developer to explicitly specify a list of domains for which she can specify a different policy.

The remainder of this chapter presents additional technical details and what the main benefits of this new version of the Network Security Policy are.

Cleartext. Allowing or blocking cleartext protocols can now be easily achieved with the `cleartextTrafficPermitted` attribute—as it was already possible to configure with the first version of the Network Security Policy. However, the developer can decide "where" to apply this security configuration. This attribute can be defined both within a "base" and "domain" config node. To enforce this setting at runtime, networking libraries can rely on the `NetworkSecurityPolicy.isCleartextTrafficPermitted()` API, which returns whether cleartext traffic should be allowed for the entire application. Instead, to check if cleartext traffic is allowed for a given host, a library can use the `isCleartextTrafficPermitted(String host)` API.

Certificate Pinning. It is now possible, for a developer, to define also Certificate Pinning. Its configuration is now much simpler and straightforward than it was in the past. First, since Certificate Pinning is used to

verify the identity of a specific domain, all the configurations need to be defined in a `domain-config`. Second, the developer needs to define a `pin-set` node (with an optional `expiration` attribute to specify an expiration date for this entry).

The `pin-set` node works as a wrapper for one or multiple `pin` nodes, each of which can contain a base64-encoded SHA-256 of a specific server's certificate. Multiple `pin`s can be used as a form of *backup*, to avoid issues while performing *key rotations*, or to *pin* additional entities like the *Root CA* that emitted the certificate for the domain. Certificate Pinning validations kicks-in only when the application tries to reach one of the listed domains. The connection is allowed *if and only if* the hash of the certificate provided by the server matches with at least one hash in the `pin-set` node.

KeyStore and CAs. The new version of the policy allows a developer to specify which KeyStore to consider as trusted when performing secure connections. The developer has first to define a `trust-anchors` node, which acts as a container for one or more `certificate` nodes. Each `certificate` node *must* have a `src` attribute, which indicates *which* certificate(s) to trust. The values for `src` can be one of the following: `system`, which indicates that the System KeyStore, the default one; `user`, which indicates the *user-installed certificates* within the *User KeyStore*; or a path to an *X.509 certificate* within the application package. When multiple `certificate` nodes are defined, the system will trust their union.

Besides, the developer can also specify an `overridePins` boolean attribute within a `certificate` node. This attribute specifies whether the CAs within this certificate node should bypass certificate pinning. For example, if the attribute's value is `true` for the *system CAs*, then pinning is not performed on certificate chains signed by one of these CAs.

Debug. Applications protected by the Network Security Policy are more difficult to debug. To address these concerns, the policy can contain a `debug-overrides` node to indicate which policy should be enforced when the application is compiled in *debug mode*, by setting the `android:debuggable` manifest attribute accordingly—setting its value to `true` to enable the *debug mode*, or `false` to enable the *release mode*. However, application *must* be compiled in *release* mode to be accepted on the Play Store. If the developer leaves a `debug-override` node in the policy of a release build, the content of the node is simply ignored.

The *debug-overrides* nodes overrides the "trust-anchors" with a custom configuration. The developer can then specify which CAs trust while performing secure connections. If this node is defined, then certificate pinning

is not enforced at runtime. It is important to specify that this oversight does not introduce any security issue to the policy specified by the developer.

Following, a concrete example of a (complex) policy that touches on the various points previously discussed.

Listing 3.1: Network Security Policy

```
<network-security-config>
    <domain-config
      cleartextTrafficPermitted="false">
      <domain includeSubdomains="false">
      android.com</domain>
      <pin-set expiration="2020-12-12">
        <pin digest="SHA-256">YZPgTZ+woNCCCIW3LH2CxQeLzB/1
            m42QcCTBSdgayjs=
        </pin>
      </pin-set>
    </domain-config>
    <debug-overrides>
      <trust-anchors>
        <certificates src="system"/>
        <certificates src="@raw/custom_cert"/>
      </trust-anchors>
    </debug-overrides>
</network-security-config>
```

The policy defines that the application should reach the *android.com* domain—but not for its subdomains—only via HTTPS and only with a specific certificate (i.e., it implements certificate pinning). The policy also defines an expiration date for this certificate. Moreover, when the application is compiled in *debug mode*, network connections can be trusted if they are signed with CAs defined within the system KeyStore or with a custom, hardcoded CA *"custom_cert"*. Also, no certificate pinning is enforced.

3.3.2 Towards HTTPS Everywhere

Starting from Android 7.0, during the installation of a given application, the system checks whether the developer did define a policy: if yes, it loads the policy; otherwise, it applies a default one. Note also that if a policy is defined but it does not specify a node or an attribute, the system *fills* the missing values by inheriting them from a similar node, or, when none are available, from the default configuration.

The default values applied by the system do change over time depending on the target API level and are becoming stricter—and by forcing application developers to target high API levels to be admitted on the official Play

Store, Google is leading a push towards *HTTPS everywhere*. We now discuss how these default values change depending on the target API level.

API 23 (Android 6.0) and Lower. An application targeting an API level *lower or equal than 23* cannot specify a policy since this mechanism was introduced from API level 24. In this case, the system will then enforce the following default policy:

Listing 3.2: Network Security Policy API 23 and Lower

```
<base-config cleartextTrafficPermitted="true">
  <trust-anchors>
    <certificates src="system" />
    <certificates src="user" />
  </trust-anchors>
</base-config>
```

This configuration allows an application to use cleartext protocols and to trust the union of CAs from both *System* and *User KeyStore*.

From API 24 (Android 7.0) to 27 (Android 8.1). The default policy for applications targeting API levels *from 24 to 27* changes as follows:

Listing 3.3: Network Security Policy API 24 to 27 Lower

```
<base-config cleartextTrafficPermitted="true">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

That is, cleartext traffic is still allowed, however, only CAs in the *System KeyStore* are trusted by the application. Thus, *User KeyStore* is not trusted anymore by default thanks to the given default policy configuration.

API Level 28 (Android 9) and Higher. For applications targeting an API level *greater or equal of 28*, the policy is even stricter:

Listing 3.4: Network Security Policy API 28 and Higher

```
<base-config cleartextTrafficPermitted="false">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

This change enforces that all cleartext protocols are blocked [AD20].

While developers were free to target any API level, starting from November 1st, 2019, all applications (and updates as well) published on the official

Google Play Store *must target at least API level 28*, corresponding to Android 9.0 [GN19]. This has the effect that the latest policy will be enforced by default to all applications not defining a custom one.

3.3.3 TrustKit

One library that is particularly relevant for our discussion is *TrustKit* [Dat16]. This library allows the definition of a Network Security Policy for applications targeting versions of Android earlier than 7.0 (which, as we mentioned before, do not support this new defense mechanism). From a technical standpoint, this library reimplements the logic behind the Network Security Policy, allowing an application to import it as an external library. Thus, by using this library, an application can define a custom policy that will be enforced by the library itself. Note that TrustKit only supports a subset of features: the developer cannot specify a `trust-anchors` within a `domain-config` node, and it is not possible to trust CAs in the User KeyStore. However, the library implements a mechanism to send *failure reports* when pinning failures occur on specific domains, allowing a developer to constantly monitor for pinning violations. Interestingly, *this feature is not available by the system-implemented Network Security Policy.*

Following, a concrete example of a policy configured using TrustKit.

Listing 3.5: Network Security Policy - TrustKit

```
<network-security-config>
  <domain-config>
    <domain>www.datatheorem.com</domain>
    <pin-set>
      <pin digest="SHA-256">k3XnEYQCK79AtL9GYnT/nyhsabaso3V+
          bhRQYHQbpXU=</pin>
      <pin digest="SHA-256">2kOi4HdYYsvTR1sTIR7RHwlf2SescTrpza9ZrWy7poQ
          =</pin>
    </pin-set>
    <trustkit-config enforcePinning="false">
      <report-uri>http://report.datatheorem.com/log_report</report-uri>
    </trustkit-config>
  </domain-config>
  <debug-overrides>
    <trust-anchors>
      <certificates overridePins="true" src="@raw/debugca" />
    </trust-anchors>
  </debug-overrides>
</network-security-config>
```

What it is interesting to see in this configuration, is the `report-uri` node. Everytime the library detects a violation of the Certificate Pinning—the

domain exposes an unexpected certificate to the application—such violation is reported and logged remotely to the http://report.datatheorem.com/log_report endpoint.

3.4 Policy Weaknesses

As previously discussed, Network Security Policy is undoubtedly making the specification of a fine-grained network policy more practical for application developers. However, despite the Network Security Policy allows a developer to define all the security settings in an unique configuration file by using an easy and declarative syntax, each of the features introduced by the Network Security Policy may be *inadvertently disabled or weakened* by an inexperienced developer during the definition of the policy. Unfortunately, to date, there are no tools that help developers to verify the correctness of the defined policy and to check that the settings she wanted to implement are effectively the ones enforced by the system.

We now discusses several potential pitfalls that may occur when an inexperienced developer configures a Network Security Policy.

3.4.1 Allow Cleartext

As described in the previous chapter, a developer has multiple ways to define the usage of cleartext protocols. For example, the developer can define a list of domains and limit the adoption of cleartext only to them. Otherwise, if the application contacts all the endpoints securely, the developer can completely opt-out from cleartext communications and be sure to identify potential regression issues. However, a developer may configure the application with the following policy:

Listing 3.6: Network Security Policy - Allow Cleartext

```
1 <base-config cleartextTrafficPermitted="true">
2     ...
3 </base-config>
```

This configuration allows the application to use cleartext protocols, *potentially* exposing the user and the application to threats described in Chapter 3.2. To make things worse, as we will discuss throughout the rest of the chapter, several online resources suggest implementing this very coarse-grained policy, with the goal of disabling the safer defaults: the main concern is whether the inexperienced developer is fully aware of the security repercussions of such policy. Note that, as described in Chapter 3.3.2, this is

the default configuration used by applications targeting an API level lower than 27.

For the sake of clarity, it is important to mention how *this specific configuration does not impact an application where all the endpoints are already reached securely*—this policy is useful only when acting as a safety net. In other words, this configuration does not lower nor weaken the security of an application performing all the network operations using, for example, HTTPS. However, this configuration is not able to identify *regression* issues: if an endpoint is inadvertently moved from HTTPS to HTTP, the insecure connection is allowed due to this "too open" policy (while the default policy could have blocked that). A similar scenario also affects complex applications, which are either developed by different teams within the same organization or that are developed by embedding a high number of third-party dependencies: in these cases, it is extremely challenging, if not outright impossible, to make sure that no connection would rely on cleartext protocols.

Unfortunately, as we previously discussed, even one single endpoint (or resource) reached through HTTP might be enough to compromise the security of the entire application.

3.4.2 Certificate Pinning Override

The Network Security Policy makes the adoption and configuration of certificate pinning straightforward. The developer now only needs to declare a valid certificate for each of the domains she wants to protect: then, the system takes care of all the logic to handle the verification of the certificates at connection time. On the other hand, we identified pitfalls that an inexperienced developer may not be aware of. For example, consider the following policy (which we took from a real application):

Listing 3.7: Network Security Policy - Certificate Pinning Override

```
<domain-config>
  <domain>DOMAIN</domain>
    <pin-set>
      <pin digest="SHA-256">VALID_HASH</pin>
    </pin-set>
  </domain-config>
<trust-anchors>
  <certificates src="system" overridePins="true"/>
</trust-anchors>
```

We argue that this policy is misconfigured and that it is very likely that the developer is not aware of it. Given the specification of the `pin-set` en-

tries, it is clear that the intent of the developer was to actually implement certificate pinning. However, the `overridePins` attribute of the system certificate entry is set to `true`: this indicates that certificate pinning should *not* be enforced for *any* CAs belonging to the *System KeyStore*, thus making the previous `pin-set` specifications useless. Not implementing Certificate Pinning mechanism leaves the application unprotected from network attackers and expose it to threats discussed in Chapter 3.2

We believe that this kind of policy offers a "false sense" of security for a developer, especially since no warnings are raised at compilation time nor at runtime. This situation is potentially exposing the developer into thinking she has correctly implemented the Certificate Pinning mechanism, even if, at runtime, the verification is not performed.

A more correct configuration of the Network Security Policy, for what concerns the implementation of the Certificate Pinning, should have *the "pin" node and the "overridePins" attribute mutually exclusive.* Instead, the current configuration of the Network Security Policy, allows a developer to wrongly configure the policy by mixing these two features, resulting in the complete override of Certificate Pinning.

3.4.3 Silent Man-In-The-Middle

Switching from HTTP to HTTPS does not always guarantee that the communication cannot be eavesdropped. As described in Chapter 3.2, the only adoption of SSL/TLS might be not enough to ensure the confidentiality, integrity and authenticity of the network communication. Under certain specific circumstances, it is possible to perform MITM over SSL/TLS encrypted connection, through which an attacker can eavesdrop and even modify the data sent between a client and a server.

Consider the following policy taken from a real application:

Listing 3.8: Network Security Policy - Silent MITM

```
1 <trust-anchors>
2   <certificates src="system"/>
3   <certificates src="user"/>
4 </trust-anchors>
```

This policy may expose an application to MITM, even if it is potentially not allowing any cleartext communication and all the endpoints are correctly reached via HTTPS. In fact, this policy trusts the union of the Certificate Authorities in the *System and User KeyStore*: hence, the traffic of the application can be eavesdropped by anyone who controls a custom CA in one of the KeyStores.

This policy overrides the default configuration introduced on Android 7.0, which prevents applications from trusting CAs stored in the *User KeyStore* when performing secure connections. Even though trusting "user" certificates may be the norm at the development phase, we believe that a "release application" that actually trusts user certificate is often a symptom of misconfiguration since it is very rare that an appliaction would actually need to trust User defined CAs.

For example, even network-related applications such as VPN apps do not need to trust User CAs, even when trusting custom certificates is required: in fact, VPN solutions can hardcode the custom CA within the application, and add a `trust-anchors` node pointing to it. This has the net effect of trusting *only* this specific certificate, and nothing else.

One scenario where trusting User CAs seems required relates to Mobile Device Management applications (MDM), which need to install different CAs coming from different sources and that cannot be pre-packaged within the released application. However, these MDM solutions constitute a rare exception, rather than the norm. For the all the other cases (which represent the majority) this *should* be considered a misconfigration, because of the aforementioned reasons.

3.5 Policy Adoption

As one of the contributions of this study, we set out to explore how the Network Security Policy has been adopted by the Android ecosystem. This chapter illustrates how the Network Security Policy is used in the Android ecosystem and it is organized as follows: First, in Chapter 3.5.1, we show and describe the dataset we used for our large-scale analysis. Second, in Chapter 3.5.2, we discuss how Android applications use this new security mechanism. We provide statistics on how frequently each feature of the policy is used, and we present insights related to applications adopting policies that are inherently "weak" and that likely constitute inadvertent misconfigurations. Last, in Chapter 3.6, we present a security evaluation and analysis of *network libraries*, which, from a technical standpoint, is where the "enforcing" of the policies actually lies, and we show the automatic testing framework we developed to determine whether a given network library correctly honors the various elements of network policies.

3.5.1 Dataset

To perform our analysis, we first built a comprehensive and representative dataset of Android applications. To determine the applications to download, we obtained the list of package names from AndroidRank [And11], a service that provides "history data of applications on Google Play." We opted to select those applications belonging to the "Top Apps" category for what concerns the *installation distribution*, with applications whose unique installation count ranges from 10K to more than a billion. This category contains the "most-installed applications" on the Google Play Store according to AndroidRank. In total, we identified and downloaded 125,419 applications, period of time ranging from June to July 2019.

3.5.2 Dataset Exploration & Weaknesses

Methodology. We now present the methodology used to explore our dataset. After extracting the policies from the applications, we first perform clustering to highlight common patterns and whether two or more applications share the same exact policy (or specific portions of it). In particular, we group two policies in the same cluster if they contain the same nodes, attributes, and values, in any order. The order in which nodes and attributes are defined is not important, as the system does not take them into account when processing the policy. This approach also helps us to determine whether application developers "copied" policies from known developer websites, such as StackOverflow.

We then analyze the clusters to identify peculiar configurations or weaknesses. Once an interesting configuration has been identified, we then proceed by performing queries on the entire dataset (that is, inter-cluster) to measure how common this specific aspect of the configuration is and whether it affects many applications.

Last, we performed an additional analysis step, which is based on similar clustering techniques, but performed over a *normalized dataset*. We refer to a policy as "normalized" after we remove artifacts that are clearly specific to an application. More specifically, we normalize a policy by replacing all the concrete values of domains with the symbolic value `URL`, all certificate hashes with `HASH`, and all the `expiration` dates with `DATE`. The rationale behind this normalization step is to be able to group policies "by semantics," which is not affected when some specific concrete values differ.

Overview. After the analysis, one of the first insights is that, even though the Network Security Policy was firstly introduced in Android 6.0 in 2015,

we note how 109,087 of the applications do *not* implement any policy (in either of the two forms). Of the remaining 16,332 applications that do implement a policy, only 7,605 of them (6% of the total) adopt the original version of the policy (available in Android 6.0), while 8,727 (6.95%) adopt the new, more expressive policy format (available in Android 7.0).

From an API level perspective, our dataset is distributed as follows:

- $\sim$ 0.5% of the applications (83) target API level 29,
- 75% (12,261) API level 28,
- 11% (1,803) API level 27,
- 12% (2,077) API level 26,
- $\sim$ 0.6% (108) target API level 25 or lower.

The first clustering process creates in total 271 clusters (where a cluster is formed by *at least* two apps): these clusters group 7,184 apps out of the 8,727 apps defining the policy—the remaining 1,543 policies were unique and did not fit any cluster. The clustering process on the normalized dataset, instead, generates 170 clusters, this time with only 311 applications not belonging to any group.

We now proceed to illustrate and discuss interesting insights and common patterns extracted form the systematic analysis amongst these clusters.

3.5.3 Cleartext

Among the generated clusters, we immediately noticed the presence of a few very big groups, one of the biggest one representing apps that define a trivial policy to allow cleartext globally. This cluster is formed by 1,595 applications, all of which share the trivial policy of "allowing cleartext globally." The exact same configuration, in terms of nodes and attributes, is embedded and used also by other 2,016 applications belonging to 60 different clusters. Among the applications not belonging to any cluster, the configuration that allows the application to use cleartext protocols globally within the app, appeared other 199 times. This first analysis on the clusters allowed us to identify a total of 4,174 applications that allow cleartext protocols to be used for all the network communications.

We then investigated, instead, the opposite configuration, namely how many apps opted out from "global cleartext." Surprisingly, we identified only 156 applications that entirely block cleartext, and thus forbidding the usage

of unsecure protocols like HTTP. This result confirms how the problem of using insecure protocols is still not eradicated in the Android ecosystem, and how there is still a long way to go for developers to force the application to use only encrypted communication protocols.

Then, we considered also applications using the first version of the policy since it also allows a developer to fully opt-in, or opt-out, from cleartext—even if with less granularity. Among the 7,605 applications using the first version of the policy, 97.5% (7,416) of them allow cleartext protocols, while only the 2.48% (189) opted out from them.

As previously discussed in Chapter 3.3, the cleartext attribute can also be enabled by default if an application is targeting an API level lower or equal to 27 and it does not override it. By considering also the default settings, the numbers are even more worrisome. For instance, we noticed that among the 16,332 applications with a Network Security Policy,

- 84.8% (13,847) allow the usage of cleartext protocols,
- 12.3% (1,837) enable cleartext due to the default configuration not being overridden
- only 1.2% (170) of the applications opt-out from cleartext just for a specific subset of domains.

3.5.4 Domains Definition

The second aspect we investigated relates to the adoption and usage of the `domain` node. Amongst the applications defining a policy to define a specific behavior for one or multiple domains, we identified only 2,891 applications that allow cleartext for a subset of domains while only 219 applications which force the domain in the list to be reached only securely.

Figure 3.1 shows the cumulative distribution function of the number of domains defined within policies.

In general, most of the applications ($\sim 95\%$) specify custom policies for at most three domain names. Note how 62.5% of the applications, instead, do not define a custom policy for any domain. 21% of the applications in our dataset define exactly one domain, while 8.5% specifies up to 2 domains within their policy. As it is possible to see from Figure 3.1, the cumulative distribution function has a long tail, with several applications defining more than 30 domains within the same policy, and two applications specifying 368 and 426 domains.

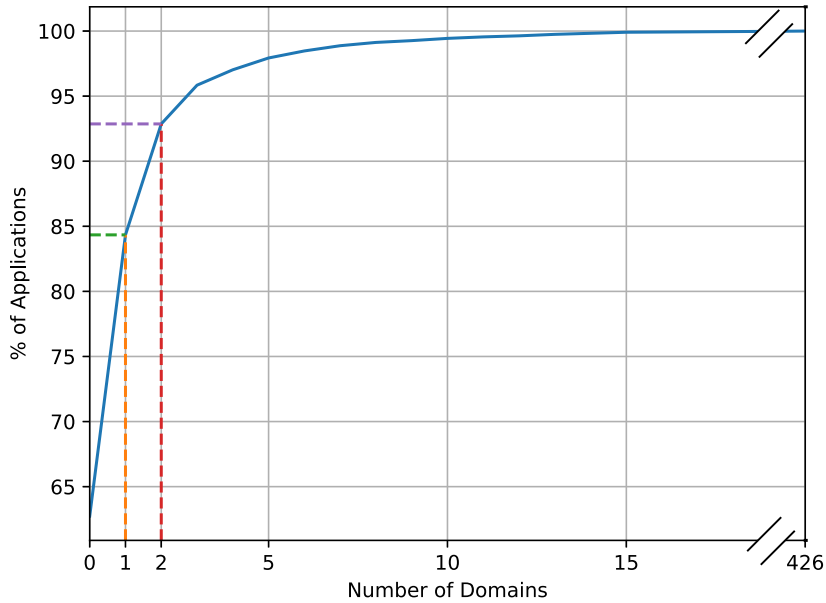


Figure 3.1: Cumulative Distribution Function of defined domains.

3.5.5 Policy for 127.0.0.1

Moving on with the analysis, this time by focusing on more complex clusters in terms of nodes and attributes, we noticed some interesting and recurring patterns. We identify how 492 applications configure a very specific `domain-config` node for the IP address 127.0.0.1, *localhost*.

Even if *this policy does not introduce any security vulnerability* and should be considered as a safe policy, we found it interesting for an aspect that this time is not related to security: while it may be common practice to spawn a local server, it is very uncommon that all the 492 applications define the same policy for localhost. This configuration, however, is very common among other applications not belonging to that cluster: in total, we identify other 512 applications belonging to 43 different clusters having the same `domain-config` setup, and 109 applications defining the same configuration for localhost, but that do not belonging to any cluster. In total, this specific domain configuration is used by 1,113 applications.

We then set out to pinpoint the underlying source of this policy, and we eventually determined that this policy is defined by the *Audience Network*

Android SDK, the Facebook advertisement framework. In particular, we noticed how a developer who wants to use this library *must modify* the policy to include this specific configuration to avoid unintended behavior. The official library's documentation makes clear that this modification is *mandatory* due to the internals of the library itself.

This finding opens a scenario that is different than the simple "developers copy policies": in this case, an advertisement library explicitly requested the developer to modify her policy to make the library work. We suspected that this pattern could be common to many other advertisement libraries. Unfortunately, our suspicion proved to be correct since we identified several advertisement libraries that explicitly request developers to copy-paste a given policy. Moreover, we found how the ad libraries' documentations often attempt to convince developers by including misleading and/or inaccurate arguments, and how many of such policies' modifications actually negatively affect the overall security of the entire application. We postpone an in-depth discussion of these findings and we discuss them in Chapter 3.7.

3.5.6 Trusted Certificates.

As we continue with the exploration and analysis of the other clusters, we identified another interesting cluster formed by 427 applications, which use a `trust-anchors` node for the entire application to trust the union of *System and User CAs*. As previously discussed, this configuration might allow, under specific circumstances, to perform a MITM over SSL/TLS connections.

Nonetheless, notwithstanding the potentially dangerous consequences to which an application may be subject as a result of the use of this policy, we notice how this specific configuration is shared among other 1,083 applications, 600 of which belong to 24 different clusters.

Going into more detail, we investigated and measured how many applications that instead use the same configuration for a subset of domains ending up identifying 73 of them. Thus, in total, we identified 1,159 applications adopting this specific configuration, among which 1,038 of them allow their SSL/TLS traffic to be potentially intercepted silently and stealthy.

3.5.7 Domain `example.com` and Invalid Digests

While we were analyzing the various domains defined in the policies of the various applications, we realized that some of these domains, repeated among several policies, were "interesting." This peculiar domain is *example.com*, and it is used in several configurations of different applications.

What it is interesting to observe is that this specific configuration is used amongst several applications, belonging to different clusters.

The biggest cluster is formed by 41 applications: all of them force the adoption of secure connection while contacting the domain *example.com*. The remaining 58 applications defining a specific policy for that domain are distributed as follows: 48 applications come from 7 different clusters, while the remaining 10 do not belong to any cluster and have a very unique policy configuration.

Since some of these configurations combined both the domain name (example.com) and an *unique* digests, we tried to track down the original policies. We then found that these policies are copied verbatim from the Android Developer website and from StackOverflow. These policies define certificate pinning on *example.com* with the correct hash or with invalid digests formed by "B" repeated 44 times.

These policies represent two real cases found on our dataset.

Listing 3.9: Network Security Policy - (a) Certificate Pinning on example.com

```
<domain-config>
  <domain>example.com</domain>
    <pin-set>
      <pin digest="SHA-256">HASH</pin>
    </pin-set>
</domain-config>
```

Listing 3.10: Network Security Policy - (b) Certificate Pinning on example.com

```
<domain-config>
  <domain>valid_domain</domain>
    <pin-set>
      <pin digest="SHA-256">BBBBB..BBBBBB</pin>
    </pin-set>
</domain-config>
```

On the policy (a), it is possible to see how the developer enforced the certificate pinning on the "example.com" domain, while in the policy (b) the developer enforced certificate pinning with a wrong digest certificate formed of only "B."

We believe that there are two possible explanations to justify the adoption of these (useless) policies. In the first one, the developer wants to define one specific feature of the policy: she then copies an existing policy that contains both the requested feature and the unique configuration of certificate pinning. In the second one, this policy might have been used by a

developer who was looking for a certificate pinning implementation and she copied the first available policy. While copying security policies that contain "dummy" domain names such as *example.com* is not a security problem per se, we believe that these policies may create a false sense of security in the developer's mind: the developer may wrongly believe that certificate pinning is correctly implemented in her application, while, in fact, it is not.

3.5.8 Certificate Pinning

The adoption of the Certificate Pinning increases the security of the network communication, ensuring integrity, confidentiality, and authenticity. Thankfully, implementing this additional defense mechanism via Network Security Policy is now much simpler than it was in the past.

However, despite all the efforts made to make this defense easier to use and reducing the number of potential mistakes a developer can make, we found that only 102 applications enforce it through the policy. Out of these 102 applications enforcing Certificate Pinning, an interesting cluster is constituted by those that implement Pinning *but then mistakenly override it*. We identified 9 applications that specify one or more `pin-set`, but also set the `overridePins` attribute to `true`, making the various `pin-set` useless. We argue that it is very likely that the developer is not aware of it, otherwise she would not have specified any `pin-set` entry. We believe Android Studio (or other IDEs) should flag this kind of policy as potentially misconfigured.

3.5.9 Invalid Attributes

All the values that can be used in nodes and attributes are well listed in the official documentation. Although the grammar to be followed is well defined, we identified a group of applications defining attributes that are not specified within the official documentation [Dev16]. For example, we pinpointed two applications defining the `usesCleartextTraffic` attribute in the policy (even if this is only valid in the *old* version of the Network Security Policy), and two applications defining the `cleartextTrafficPermitted` attribute within a wrong node. We also found one application declared the `hstsEnforced` attribute, which is *not* mentioned in the official documentation. However, by looking at the source code of the policy parser, we notice how this attribute is actually recognized as valid. This attribute allows a developer to define HSTS for the WebView component of her application (which would "force" the WebView to always contact via HTTPS websites sending the HSTS header [con20]). We note how the concept of HSTS significantly overlaps with the cleartext aspect of the Network Security Policy. We investigated

the reason why this attribute is still available within the policy and we found out that it may exist because older versions of the WebView were not enforcing the `cleartextTrafficPermitted` attribute [Dev16]—but were enforcing HSTS instead.

3.5.10 TrustKit

As we discussed earlier, applications can use the TrustKit library to implement the Network Security Policy. So, as a further analysis among the policies defined by the applications in our dataset, we analyzed those that use this library. The cluster of policies defined using TrustKit is formed by 53 applications. Among these apps, 10 define a `reporting-endpoint` to use when a pinning failure is identified, while 16 applications explicitly disabled this feature. To conclude, we identified 46 applications defining and implementing Certificate Pinning.

3.5.11 Remaining Applications

Our methodology based on clustering and targeted queries allowed us to systematically group a vast portion of our dataset. However, as we mentioned earlier, some applications did not fit in any cluster due to their peculiar policy configuration: in total, we identified 311 of these applications. We then manually inspected them all, to look for additional interesting patterns and configurations.

Among these, we identified 98 applications that define a very unique policy in terms of `domain` nodes used with the policy. For instance, we identify, several applications defining a substantial list of domains, up to identifying two applications that in total have configured a Network Security Policy for 368 and 462 unique domains, as shown in Figure 3.1.

The other 46 applications shared a specific policy that did not take advantage of the "wrapper nodes" like `pin-set` or `domain-config`: for each of the domains, these applications opened a new `domain-config` node each time instead of defining all the domains within one node.

We also found 44 applications that specify more than one custom certificate.

Another interesting configuration comes from apps whose policy appears very verbose and that could have been reduced. We noticed how 32 applications specify a default "allow cleartext" for the entire app and, on top of that, configured a very detailed list of domains and subdomains with the same exact policy.

Another interesting peculiarity we found in 21 applications is related to the definition of additional text—like comments or left-over in between nodes—that is then removed by the system during the parsing process.

To conclude, the remaining applications defined very unique and complex policies that do not belong to any of the aforementioned groups, but that, from the security perspective, do not represent anything special.

3.5.12 Dataset Evolution

Starting from November 1st, 2019, all applications *must target at least API level 28* [GN19]. This means, from a Network Security Policy perspective, that all the new apps, by default, will forbid cleartext—if they do not override the new default.

Since our dataset was crawled before November, as presented in Chapter 3.5.1, we decided to repeat some of the measurements, this time on a dataset downloaded after this new mandatory requirement. Our goal is to investigate how the applications evolved after the introduction of the new default value that forbids the usage of any cleartext protocol.

We started a re-crawl of the same initial dataset, starting from the 125,419 package names. These applications were re-crawled from April to June 2020. We were able to download 86.5% of the initial dataset, for a total of 108,542 applications. Of the remaining apps that we could not re-download, 15,749 of them were removed from the Google Play Store while 1,128 were moved from a free to "paid" download or introduced in-app purchases not available in our geographical region. The applications that we were able to re-crawl are now distributed as follows:

- 14.3% (15,531) target an API level 29,
- 46.2% (50,191) target API level 28,
- 9.5% (10,351) the API level 27,
- 12.7% (13,795) API level 26,
- the remaining 17.2% (18,674) target an API level 25 or lower.

Unsurprisingly, the number of apps defining a Network Security Policy increased: 33.3% of the applications (36,165) now specify one of the two types of policy. Among them, 65.5% (23,718) still adopts the first version of the Network Security Policy through the `AndroidManifest`, while the remaining applications (15,492) opted for the new and more recent version. Interestingly, 8.4% of the applications (3,045) use both versions of the policies.

We then looked for how many apps effectively adopted *the new default of forbidding cleartext protocols for the entire application*. Surprisingly, approximately the 33% of the entire dataset (35,789 out of 108,542) enforced a default configuration that does not permit cleartext protocols. Out of these applications, 419 configured this behavior using the first version of the policy. The remaining 67% of the apps still configure a Network Security Policy that permits cleartext traffic. From this 67%, the 32% (23,229) still adopt the first version of the policy. However, what it is interesting to notice is that 58% (42,353) of applications allow cleartext due to default configuration, dictated by the API level.

To conclude, we note that only a small portion of applications, 0.4% (349), allow cleartext as base configuration and also define a set of domains for which they allow only encrypted connections.

These results somehow highlight an ecosystem-wide problem that affects Android applications: even if Google provides a simple and easy way to configure the SSL/TLS for an application (the Network Security Policy), and even though it explicitly changed the defaults to force the usage of cleartext protocols, a significant portion of applications still opt to stick, for one reason or another, to plain and unencrypted networking protocols: while the community is making progress, we are not there yet for a full adoption of HTTPS by Android applications.

3.6 Android Networking Libraries Adoption

So far, we focused on the exploration of how applications adopt Network Security Policy. However, we did not tackle the aspect of *enforcing* these policies. The Network Security Policy is simply an XML configuration file, and it is then up to the various networking libraries to properly honor (and enforce) what is specified by such a configuration file.

To this end, we set out to explore how Android Network Libraries do enforce these policies. First, we checked the official Android documentation, which states that "third-party libraries are strongly *encouraged* to honor the *cleartext setting*" [Dev16]. However, we found the documentation concerning, for two reasons. First, the wording of the documentation only mentions that honoring the policy is "strongly encouraged." However, we believe that since the policy relates to security-relevant aspects, network libraries should be forced to honor the policy—and in case they do not, that should be considered as a vulnerability. In fact, a network library not honoring the policy would have the negative side-effect of silently making the policy useless. Moreover, having a library in charge of network communication that is not

enforcing the policy might create of a "false sense of security" for the developer since she might have configured a very strict policy which is then not enforced by the underline network component embedded into the application. Second, the documentation only mentions the "cleartext settings." However, as we discussed in Chapter 3.3, the new version of the policy touches on many more aspects: Unfortunately, the documentation does not even mention the other features (e.g., which *KeyStore* to trust, pinning).

Next, we checked the official API, implemented by the `NetworkSecurityPolicy` class. This is the API that, in theory, network libraries should rely on to obtain the content of the policy (and honor it). However, this API appears very limited: the only available API is `isCleartextTrafficPermitted()`, which returns whether cleartext traffic should be allowed. There is no other API to query the remaining fields of the policy, and it is thus not clear how network libraries are supposed to enforce them.

For these reasons, we set out to explore how and whether popular network libraries honor the policy. We now discuss how we built a dataset of network libraries, we present an automatic analysis framework to test whether a given library honors the various aspects of a policy, and we show the results of this analysis.

Libraries Dataset. To perform this investigation, we first built a comprehensive dataset of the most used networking libraries. We identified these libraries from AppBrain [App11b], a service that provides multiple statistics on the Android application's ecosystem such as "Android libraries adoption" by different applications. Our dataset consists of all the network libraries mentioned by AppBrain: *URLConnection, Robospice, HttpClientAndroid, AndroidAsync, Retrofit, BasicHttpClient, OkHttp, AndroidAsyncHTTP, Volley,* and *FastAndroidNetworking*. Except for *URLConnection*, which is the default HTTP library on Android, all the libraries are "external," which means that application developers need to manually specify them as external dependencies. Note that these external libraries, even though they are not the default, are used by almost 30% of all the applications published on the Google Play Store ($\sim 250K$ unique applications) and are used by applications with more than 500M of unique installations. Table 3.1 provides more detailed statistics about the adoption of such libraries and summarizes the distribution of our dataset. For each library, it first presents statistics about the number of applications using the given library and how many downloads has the top downloaded application. Then, for each of the tested feature, we mark whether or not the library passes all the testcases. The last column represents whether an application

is honoring the entire policy, only a subset of features or none of them. For URLConnection, statistics are not available on AppBrain.

Analysis Framework. Determining whether a given library is implementing the Network Security Policy is not a straightforward process. In fact, the source code of these libraries is often not available, and manual reverse engineering may be challenging and error-prone. Moreover, the library might come might implement the security checks necessary to honor the Policy in different ways. Thus, we opted for an automatic approach based on dynamic analysis. We built an automatic framework to check whether a given networking library honors the policy defined in an application. Note that while for this analysis we tested the ten popular network libraries in our dataset, our framework is completely generic and can be easily used to vet an arbitrary network library.

Our framework analyzes each network library individually. For each of them, it performs the following steps. First, we generate *all* the possible combinations of a policy, by combining all possible nodes, attributes, and representative values. In particular, the framework considers the following nodes: `base-config`, `domain-config`, `pin-set`, and `trust-anchors`. For each node, our framework considers all the relevant child nodes, such as `domain`, `pin`, and `certificate`. Each node is then configured with all the possible attributes that might be used within a given node, like `overridePins` for what concerns `trust-anchors`, or `src` for the `certificate` node. The entire list of the nodes, please refer to Chapter 3.3. For what concerns the values, we generate "representative values." For the value field representing a certificate hash, we generate various policies with the following values: a valid hash matching the hash of the certificate actually used during the tests, a valid hash that is different than the expected one, and a non-valid hash (e.g., the character "A" repeated several times). The combinations of all nodes, attributes, and representative values, generates 72 unique policies. As mentioned above, some of the policies will be "properly configured" while others will "misconfigured," as described in Chapter 3.4.

Then, the framework creates an application that attempts to connect to an endpoint via HTTP and via HTTPS by using the library under test. The application is then built multiple times, each time with a different policy. Each of these applications is then tested in three different "testing environments," each of which simulates the different threat models discussed in Chapter 3.2:

- the application is tested without attempting to perform MITM,
- we simulate an attacker performing MITM (by using a proxy),

Table 3.1: Compliance of Networking Libraries.

Networking Library	# of Apps	Top App Downloads	Cleartext	Certificate Pinning	Trust Anchors	Compliant
Retrofit	> 104k	1 B	✓	✓	✓	●
Volley	> 66k	5 B	✓	✓	✓	●
OkHttp	> 39k	5 B	✓	✓	✓	●
AndroidAsyncHTTP	> 22k	100 M	✗	✓	✓	◐
AndroidAsync	> 7k	100 M	✗	✗	✗	○
FastAndroidNetworking	> 3k	100 M	✓	✓	✓	●
HttpClientAndroid	∼ 1,000	100 M	✗	✓	✓	◐
BasicHttpClient	∼ 1,000	100 M	✓	✓	✓	●
Robospice	∼ 1,000	10 M	✓	✓	✓	●
URLConnection	N/A	N/A	✓	✓	✓	●

✓ when the library passes all the testcases.
✗ when the library fails at least one testcase.
● when the library honors the entire policy.
◐ when the library honors only a subset of the features of the policy.
○ when the library does not honor the policy.

- we simulate an attacker performing MITM with the attacker's custom CA added to the User KeyStore.

At each execution, the framework logs whether a given connection with a given policy in a given testing environment was successful or not. These logs are compared with a ground truth, which is generated by a Python-based implementation that takes into account the various aspects of the policy and the various testing environments. We flag a library as *compliant* with the Network Security Policy specification if and only if the runtime logs match with the expectations of the ground truth, otherwise, we mark it as *not-compliant*.

Analysis Framework: Simulation. Following an example of how the framework works: first, it generates one of the possible configuration of the Network Security Policy.

Listing 3.11: Network Security Policy - Automatic Evaluation Framework

```
<domain-config>
  <domain>DOMAIN</domain>
   <pin-set>
    <pin digest="SHA-256">VALID_HASH</pin>
    <pin digest="SHA-256">INVALID_HASH</pin>
   </pin-set>
   <trust-anchors>
    <certificates overridePins="false"
                  src="system"/>
    <certificates overridePins="true"
                  src="user"/>
   </trust-anchors>
</domain-config>
```

Then, it proceeds by creating an Android application to test the security of the a given networking library. The application embeds the network library and defines in its source code two methods to contact *DOMAIN* both via HTTP and HTTPS. Once the application is ready, the framework embeds the generated policy, compiles the application with the given networking library and runs the application. At runtime, it then monitors for the "success" or "failure" of both the network requests. Then, the same execution is repeated with both the MITM scenarios described above. Since from the policy definition it is possible to infer the expected result, the framework is then cross-verifying if the expected result matches with the one analyzed at runtime. For this specific example, the following results are expected:

- With no MITM, the HTTP connection should fail while the HTTPS

connection should succeed, since at least one certificate digest is matched,

- For the first MITM scenario, where the attacker has no certificate on the device, both the HTTP and HTTPS connection should fail. The fist one should fail because of the forbidden of cleartext communication while the second one should fail because of certificate's signature mismatch,
- For the second MITM scenario however, the HTTP should still fail while the HTTPS connection should succeed. In fact, the permission removes the signature check on the secure connection due to the *overridePins* set to True for the "user" certificates.

In case of success, we mark the "testcase" as passed, otherwise we raise a warning, which means that the library is behaving differently than expected. If the library passes all the testcases, we mark it as "compliant," "not-compliant" otherwise.

Compliance Results. We used our framework to automatically evaluate the compliance to the Network Security Policy of the 10 networking libraries extracted from AppBrain. Surprisingly, we noticed how not all the libraries are completely compliant with the standards defined by the Network Security Policy: we identified that *HttpClientAndroid, AndroidAsync and AndroidAsyncHTTP* are not honoring the cleartext attribute. These libraries allow HTTP even though the policy would prohibit it. We note how these libraries are used by more than tens of thousands of popular apps with hundreds of millions of unique installations.

Instead, for what concerns *certificate pinning* and *trusted anchors*, we noticed that nine of the ten libraries do correctly honor the policy. Given the difficulty and missing documentation, we were positively surprised by this high adoption rate. We thus decided to investigate why libraries are enforcing such a difficult part of the policy and not the easier-to-enforce cleartext settings. For these libraries, we performed manual analysis—including source code analysis, when available—to determine how the policy is actually enforced. We found that none of these libraries is implementing SSL/TLS-related operations from scratch nor defining a custom handler for CAs. Instead, they are all relying on core Android framework methods to perform SSL operations, which includes handshake and management of the KeyStores. All these operations are handled by the *Conscrypt* [Dev20b] package, which provides a *Java Security Provider* (JSP) that implements parts of the *Java Cryptography Extension* (JCE) and *Java Secure Socket*

Extension (JSSE). While this is clearly a positive news, we find it surprising that these popular network libraries do not adhere to arguably more critical cleartext settings.

We also found that *AndroidAsync*, used by thousands of apps, does not support the Network Security Policy at all. In fact, we found that the mere presence of a `domain-config` node is enough to break the network library, leading to an exception, and thus making it essentially incompatible with the Network Security Policy. Table 3.1 summarizes our findings.

3.6.1 Disclosure

We disclosed our findings to Google, with an emphasis on the misconfiguration of the SSL Pinning (which may give a false sense of security to inexperienced app developers). We also proposed to extend the Android-Studio IDE with a linter for the Network Security Policy that checks for these misconfigurations and informs the developer about the potential risks. Google acknowledged that this is, in fact, a rather odd configuration. For what concerns the networking libraries not compliant with the actual Network Security Policy (see Table 3.1), we have disclosed our findings to the developers. We are still working towards full bugs fixes.

3.7 Impact of Advertisement Libraries

Advertisements (ads, in short) play a key role in mobile applications. To better understand how this ecosystem works, we first provide an overview of how advertisement libraries (ad libraries) operate and their complexity, and we then explore the implications for the adoption of the Network Security Policy.

Ads are the most important source of income for many application developers, especially when applications can be freely downloaded from the Google Play Store. An application can simultaneously embed one or multiple ad libraries. While the app is running, the ad library retrieves ads content from a remote server and it displays it to the user. Every time an ad is shown to the user, the developer earns a revenue. If the user clicks on the ad, the developer then gets a more substantial revenue. Depending on the type of agreement that the application developer has with the advertiser, the amount of money earned by the application developer may vary depending on the user's interaction with the banner ad [ads21].

Even though this mechanism is conceptually simple, the actual implementation details and the underlying process are far from trivial. We now

quickly discuss the main steps, which are also depicted in Figure 3.2. The diagram shows what are the different stages, and players, necessary to make an application showing a given advertisement to the user.

First, the developer embeds a given ad library (or multiple libraries) in her application, and implements a set of callbacks and handlers to use when an ad is ready to be shown. Then, when the application is running, the ad library contacts its backend server and asks for an ad to be displayed.

Depending on the ad library's implementation, this first request can reach one or multiple servers. In case of an *individual ad network*, the library contacts a single server, while in case of an *ads aggregator* the request is sent to multiple servers. The server then forwards the request to its *ad network*, which might be more or less complex. Within the ad network, the *bidding auction* starts. Bidding consists of advertisers (brand) declaring the maximum amount of money they are willing to pay for each impression (or click) of their ad. The winner sends the content of the ad back to the library, and the ad is then displayed in the application, normally within a WebView. By displaying the content of the ad, the application acts as the content publisher for the brand that won the auction.

The interaction of the user who is using the application with the advertisement just shown can trigger a series of additional operations and network connections. For instance, if the user clicks on the ad, then the *full enriched content* might be retrieved from the server of the auction's winner (which is related to the specific ad, and *not* to the ad library itself). The full content can then be displayed within the application, or the user can be redirected to a different component, like a browser, where it is displayed.

The complexity of the ad ecosystem and the interconnection of multiple players—each of which only controls a *portion* of the ecosystem—opens interesting questions related to the Network Security Policy. Since the winner of the auction is usually not under the control of the ad library, the *enriched content* downloaded upon a user's click may be served via HTTP: this aspect makes it interesting to determine how different ad libraries deal with this "uncertainty" on the protocol used by the advertiser. Motivated by these observations, we set out to perform the first systematic analysis of the Network Security Policies defined by ad libraries.

The remainder of the chapter is organized as follows: First, we present the dataset of ad libraries that we built for the analysis. Then, we analyze and characterize the different Network Security Policies defined by ad libraries, and we show how several of these libraries push the application developers to severely weaken their original policies, oftentimes justifying these requests with misleading arguments. We conclude our discussion with

an in-depth case study. We note that, ideally, it would be interesting to perform large-scale and automated analysis over many ad libraries and applications embedding them. However, we refrain from performing such a study due to ethical concerns: in fact, automatically visiting applications with the mere goal of generating ad impressions that would *not* be seen by real users (or, even worse, automatically clicking on these ads) would generate illegitimate revenues for the application developer (who could be framed as fraudster), and it would damage all the ads ecosystem's parties involved.

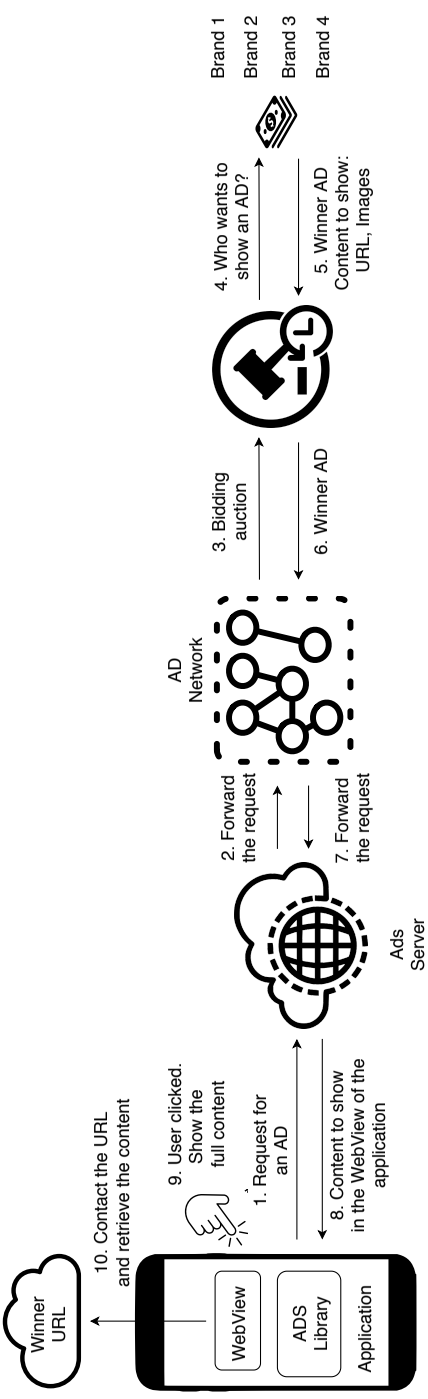


Figure 3.2: Ecosystem of *individual ad network*.

3.7.1 Dataset

To perform this investigation, we first built a comprehensive and representative dataset of the most used ad libraries. We choose the Top 29 ad libraries from AppBrain [App11a] based on the ranking "number of applications." With these libraries, our dataset covers all different types of advertisements, from libraries offering "Video Ads," to the ones offering a "Mediation service," and "Native Ads." Table 3.2 summarizes the statistics about the ad libraries and synthesizes our dataset.

3.7.2 Policy Characterization

We investigated whether a given ad library requires a Network Security Policy modification and of which kind. To identify if a library requires a policy, we start by looking at its official documentation. In case we do not find any reference to the Network Security Policy, we then proceed by analyzing the source code of the "reference example application," which is always provided by the ad library developers to show how such a library can be integrated. Since these reference applications use a specific ads library and should come with a predefined policy, thus becoming valid substitute in case of lack of official documentation.

Among the 29 libraries that we analyzed, we found that *12* of them do require the developer to modify the policy. The remaining 17 do not require any modification, which suggests that their backend infrastructure is fully compliant with the latest standards and defaults.

We then proceed by inspecting which configuration of the policy is pushed by the advertisement library. One of these is the Facebook Audience Network ad library, which only requires the developer to specify a configuration for a single domain, as discussed in Chapter 3.5.2. The other libraries require more invasive modifications, which we discuss next.

Cleartext. Our first finding is concerning: All the 11 libraries require the developer to allow cleartext on her application. Amongst the 11 ad libraries that force the developer to allow cleartext, *we found that MoPub, HyprMx, HeyZap, Pollfish, AppMediation, and Appodeal do force the developer to completely allow cleartext protocols for all domains.* Moreover, we also found that *AdColony, VerizonMedia, Smaato, AerServ, and DuApps push the developer to adopt the first version of the policy*, with similarly negative consequences. These configurations make ineffective any safety net that a Network Security Policy may provide. However, we note that these ad libraries may be *required* to ask for this modification since it could

Table 3.2: Analysis of Advertisement Libraries.

Library	# of Apps	Top Apps Downloads	Requires Network Security Policy
AdMob	> 464k	1B	
Facebook Audience Network	> 96k	500M	✓
Unity	> 67k	50M	
AppLovin	> 34k	100M	
Chartboost	> 30k	1B	
Startapp	> 29k	100M	
AppsFlyer	> 29k	500M	
AdColony	> 24k	100M	✓
Vungle	> 20k	100M	
MoPub	> 19k	1B	✓
Ironsource	> 19k	50M	
Amazon Mobile Ads	> 13k	500M	
Tapjoy	> 11k	100M	
InMobi	> 11k	100M	
Pollfish	> 9k	10M	✓
AppNext	> 8k	100M	
Adjust	> 8k	1B	
HeyZap	> 7k	100M	✓
Smaato	> 4k	100M	✓
Fyber	> 4k	100M	
Millennial Media	> 3k	500M	✓
MyTarget	> 3k	100M	
Appodeal	> 3k	50M	✓
Kochava	> 2k	100M	
AerServ	> 2k	100M	✓
Tenjin	~ 1,000	100M	
HyprMX	~ 1,000	100M	✓
DU Ad	~ 500	100M	✓
AppMediation	N/A	N/A	✓

62

be that a given ad framework does not have enough control over the type of URLs (HTTP vs. HTTPS) that are served as part of the ads.

As illustrated in Chapter 3.5, please note that the policy defined by the "Facebook Audience Network" should not be considered as violating the least privilege principle even if it is allowing cleartext communications since the policy declares the adoption of cleartext only for a specific "domain" that is supposed to be used only locally.

Trusted Anchors. The analysis of the advertisement ecosystem had led us to imagine that libraries could only force the developer to use cleartext. To our surprise, we noticed that some libraries also push the developer to define other parameters for her Network Security Policy. Indeed, we have identified several ad libraries defining a `trust-anchors` node. Even in this case, the findings are concerning: *Appodeal* [Dev19a] and *HeyZap* [Dev19b] suggest the developer to add *User KeyStore* as trusted, thus providing a venue to perform MITM attacks. Moreover, none of these libraries provide any custom CA, nor ask the developer (or the user) to do so, making this risk completely unnecessary. We believe it is important to emphasize how, even in the case where the library would have needed a custom Certificate, it would have configured the policy to trust a local certificate within the application, and not the entire User KeyStore. We can conclude that the definition of this policy is not strictly necessary and dangerous.

Misleading Documentation. We argue that the security repercussions of Network Security Policy modifications should be properly explained and justified to developers so that they can take informed decisions on whether to include a given ad library. However, we found how this "transparency" is not a common practice amongst advertisement libraries. After closely inspecting the documentation of the 11 ad libraries mentioned above, we found that *none* of them inform developers of the possible consequences of allowing cleartext protocols or trusting User KeyStores. Some of these libraries simply inform the developers that they *need* to apply their modifications of the Network Security Policy in the name of "usability" and to avoid any faulty behavior or to avoid ads with HTTP asset URLs from being served incorrectly. Moreover, we identified how *Millenial Media, Smaato, HyprMX, and AerServr* simply ask the developer to copy-paste the provided sample AndroidManifest, without explicitly mentioning the fact that such a sample manifest silently specifies a "usesCleartextTraffic" policy. Even worse, we found how *Du Apps* misleadingly justifies the need to allow cleartext traffic because it is *"required for target SDK28."* We believe that the underlying reason for these problems is that most of these ad libraries found themselves in difficulty due to their infrastructure not being ready to deal

with Google's HTTPS everywhere push.

3.7.3 Ad Libraries in Applications

As previously discussed, we identified some ad libraries that ask developers to weaken their security policy and to allow cleartext. We performed additional experiments that aim at determining how frequently these ad libraries are used within our dataset and whether these applications allow cleartext as part of their Network Security Policy.

To detect a third-party library within a given application, we use `LibScout` [BBD16a], the state of the art static analysis tool for this kind of task. According to [BBD16a], `LibScout` can detect the inclusion of external libraries within applications even when common bytecode obfuscation techniques are used. `LibScout` supports two types of detection: the first one is based on a simple matching with the package name, while the second one relies on code similarity. By default, it reports only matches that have a similarity of at least 70%. For our experiment, we used the same threshold. At the time of writing, `LibScout` supports only the *Facebook Audience* ad library. We extended it by creating profiles, necessary for the detection, for all the ad libraries that require the developer to modify the Network Security Policy to allow cleartext. Then, for each of the applications in our dataset, we run `LibScout` for a maximum time of one hour.

We run `LibScout` on the first dataset of 16,324 applications (which specify a Network Security Policy), and also on the second "fresher" dataset of 108,542 apps. For the first dataset, `LibScout` was not able to conclude the analysis in time for only 8 applications, while it analyzed correctly the entire second dataset. In total, the matching engine identified that 19.7% of the applications belonging to the first dataset (3,230) do have one of the ad libraries that requires cleartext. For the second dataset, instead, it determined that 8.8% of the applications (9,645) contain at least one of the libraries of our dataset.

Table 3.3 summarizes the results. Unfortunately, we suspect that `LibScout` might have missed several matches (that is, it does not find libraries even if they are included). In fact, Table 3.3 shows how the matching results are dominated by the "package name" heuristic, and how only 41 matches for the first dataset, and 341 for the second, were solely due to the similarity analysis engine (i.e., all other matches were already covered by the package name heuristic, hinting that the applications were not obfuscated). We thus remind the reader that, for the numbers reported next, the accuracy of these numbers is based on the accuracy of the underlying libraries matching engine, `LibScout`.

Table 3.3: LibScout external libraries identification.

# Apps with Ad library matched by	Dataset 1	Dataset 2
Package Name (PN)	3,189	9,304
Code Similarity (CS)	2,072	5,918
$PN \wedge \neg CS$	1,158	3,727
$CS \wedge \neg PN$	41	341
$PN \wedge CS$	2,031	5,577
$PN \vee CS$	3,230	9,645

Dataset 1 represents the analysis over 16,324 applications.
Dataset 2 represents the analysis over 108,542 applications.

We then proceeded by checking how many of the applications identified by `LibScout` effectively have a Network Security Policy that allows global cleartext, as defined by the ad libraries. Table 3.4 summarizes our findings and presents the distribution of the dataset in terms of inclusion of ad libraries that ask developers to weaken their policy and whether the applications' Network Security Policy allows cleartext. We note how for the first dataset, 89% of the applications (2,891) embedding an ad library do have a Network Security Policy that allows cleartext. However, 11% (339) do not allow it: for these applications, the ads served over HTTP will not be displayed and an exception is thrown. We also note that, even if applications do not use ad libraries, a large portion of them (83%) still use HTTP. Thus, while ad libraries asking developers to weaken their security policy certainly does not help, it does not seem to be the only reason application developers stick to insecure HTTP connections. For the second dataset, we found that, among apps that include an ad library, 75.6% of them (7,298) define a Network Security Policy that permits cleartext. The percentage of applications that allow cleartext decreases to 66.1% when considering apps that do not include one of the ad libraries we have checked for.

3.7.4 Case Study: MoPub

We now present an in-depth analysis of one of the most prominent ad libraries, MoPub [Dev20a]. This library is an *individual ad serving platform* used by over 19k applications, some of which have more than 50M unique installations. MoPub is one of those libraries that *requires* a developer to allow cleartext for her entire application. For this case study, we set out to determine whether this library really had no other choice but to require

cleartext on the entire application to properly work. To shed some light, we aimed at monitoring the network requests performed by this ad library at run time. We note that a simple network monitor on the traffic generated by the entire application is not enough: by just observing network traces, it would be very challenging to determine which traffic has been generated by the ad library and which by unrelated components of the application.

Thus, we developed an instrumentation framework that records all network activities and, moreover, hooks the network `Socket.connect` API (by using Frida [Rav20]). This API is the lowest-level API used for any HTTP or HTTPS connection and it provides the target domain name and the port. We we have not considered native implementation of the same API since we did not find any ad libraries using native components. Every time the API is invoked, we perform a stack trace inspection to determine which package has originated the call: this setup allows us to match which component (i.e., library) of the application initiated the network request.

Due to the ethical concerns mentioned earlier, we limited ourselves to a very small-scale experiment: we opted to select and analyze only one representative application, *Hunter Assassin* [Stu20], an action game with more than 50M installations. This app embeds MoPub and specifies a Network Security Policy that reflects MoPub's documentation. For the experiments, we executed the application 10 times, with each execution lasting 10 minutes. Due to ethical concerns, we opted to not use automatic UI stimulation techniques, but we performed this analysis step manually, by just simulating the interaction of a "real" user. This approach allows us to avoid generating excessive traffic which would have potentially damaged the reputation of the application developer towards the advertising platform.

During the analysis, our instrumentation framework detected that the MoPub library initiated connections to 83 unique domains—for this measurement, we discarded the domain names reached by other components of

Table 3.4: Distribution of the dataset in terms of inclusion of ad libraries and cleartext configuration.

Network Security Policy	Dataset 1		Dataset 2	
	Ads	No Ads	Ads	No Ads
Cleartext	2,891	10,956	7,298	65,455
No Cleartext	339	2,138	2,347	33,442

Dataset 1 represents the analysis over 16,324 applications.
Dataset 2 represents the analysis over 108,542 applications.

the application. Surprisingly, for 82 domains (out of 83) the connection was actually established using HTTPS, the only exception loaded over HTTP being an image, retrieved from a MoPub server. Even though this HTTP connection would be blocked by a non-permissive cleartext policy, we do not believe this is *the* core reason why MoPub requires the policy to allow cleartext for the entire application.

According to the MoPub documentation, it requires HTTP because it *may* need to serve ads via HTTP—and to do so, it asks the developer to weaken the policy for the entire application. Indeed, the official documentation reports "Android 9.0 (API 28) blocks cleartext (non-HTTPS) traffic by default, which can prevent ads from serving correctly. To mitigate that, publishers whose apps run on Android 9.0 or above should ensure to add a network security config file. Doing so allows cleartext traffic and allows non-HTTPS ads to serve." [Dev20a]

We believe this to be a clear violation of the principle of least privilege, as the ad library should allow cleartext for its own connections, without interfering with the rest of the application. However, we note that this current situation is not solely fault of the ad library: with the current policy format, it would be impossible to enumerate all possible domain names that the ad library should be able to reach since this list is not known in advance (and since the Network Security Policy cannot be changed at run-time). We identified a conceptual limitation: the current policy format allows developers to specify policies *per domain*, but we believe a better abstraction for policy specification to be *per package*. In an ideal world, the ad library should be able to express that *only the connections that are initiated by the MoPub library itself* should be subject to use cleartext, without the need of weakening the rest of the application. Guided by these insights, we designed and implemented a drop-in extension to the Network Security Policy that would address this concern and allows a developer to adhere to the principle of least privilege.

3.8 Network Security Policy Extension

As previously discussed, third-party libraries can significantly weaken the Network Security Policy of an application, and as we have demonstrated through extensive analysis, ad libraries actually often do so. In some scenarios, however, it is very challenging for ad libraries to "do better." In fact, the complexity of the ad ecosystem may make it impossible, for example, to know in advance which domain names require HTTP connections, thus leaving the ad library developers to ask to allow cleartext for the entire

application. However, this problem is not caused only by advertisement libraries: conceptually, any third-party library that needs to perform a network operation can potentially force the developer to modify the Network Security Policy. Please note that not all the modifications made by third-party libraries are weakening the Network Security Policy. For instance, if a library defines all its configuration inside a "domain-config" tag (like the Facebook ADS), it is *not* going to impact the original definition of the policy since the configuration will be enforced only on the domains specified inside the tag. However, as discussed in Chapter 3.7, we highlighted how almost all the advertisement libraries that are forcing the developer to modify, or adopt, a Network Security Policy are tampering the base configuration, affecting the entire application.

We believe the current format of the policy is fundamentally limited. The current policy allows developers to specify different policies at the granularity level of domain names: however, we argue that, in some scenarios (e.g., ad libraries), this is the wrong abstraction level.

We now discuss and present our proposal for an extension of the Network Security Policy format to allow for the specification of policies at a different granularity: we propose a new level of granularity, targeting this time specific application components, identified by their package names.

Our New Extension. The core idea behind the extension is to allow a developer to bind a specific policy to a specific package name(s). To this end, we introduce a new XML node, `package-config`, which allows developers to specify custom policies for specific external libraries, without the need to modify (and negatively affect) the policy of the main application, thus creating a "sandbox" for a third-party library that needs a specific Network Security Policy configuration. The `package-config` node contains multiple `package` childs. Each of these `package` nodes contain a `name` attribute, pointing to the package name the developer wants to sandbox, as well as a `cleartextTrafficPermitted` attribute. To ease the explanation, consider the following concrete example:

Listing 3.12: Network Security Policy - Extension

```
<base-config cleartextTrafficPermitted="false" />
<!--introduced by our extension-->
<package-config>
  <package name="com.adlib.unsafe"
    cleartextTrafficPermitted="true"/>
</package-config>
```

This policy specifies that, by default, all HTTP traffic should be blocked. However, it would allow HTTP connections if they are initiated by the

`com.adlib.unsafe` ad library. Note how the ad library can now support occasional HTTP connections even without knowing the list of domain names a priori and, more importantly, without affecting the policy of the application. The same process can also be used by the developer to make sure that none of the connections originated by `com.adlib.unsafe` are executed in cleartext, using insecure communication protocols. The developer will only need to change the value of the `cleartextTrafficPermitted` attribute to false. We believe that our proposal allows a developer to adhere to the *least privilege principle.*

Implementation. We implemented this new extension by modifying the `isCleartextTrafficPermitted` API to make it aware of the XML policy node. Our modification performs stack trace inspection to determine which package name has initiated the call. For each package name appearing in the stack trace, we then check whether the Network Security Policy contains a custom policy for a specific package name: if yes, we use that policy. Otherwise, we apply the default.

If the connection is started from a method belonging to one of the package names in the "package-config" whitelist, we allow the connection, otherwise, we raise a `RuntimeError`, indicating a policy violation—since the connection should *not* be allowed by the policy definition. It is important to highlight how a similar check is already adopted by the libraries "compliant" with the Network Security Policy: every time a new cleartext connection is made, both the system and libraries should invoke the `isCleartextTrafficPermitted` API to verify if they need to allow or deny the connection.

Adoption & Backward Compatibility. Our extension can be trivially adopted by application developers and network libraries. In fact, since we modify an API that all these libraries already invoke—and that was a key design choice—they can enjoy the benefits of our policy without the need to make any modification. We also note that our extension is fully backward compatible and it can act as a drop-in replacement of the old version. In fact, apps and policies that are not "aware" about our extension are supported exactly the same as before. For instance, since our implementation extends a mechanism that is already used by networking libraries, our defense mechanism works on the already compliant libraries (see Table 3.1). This is a key design choice that will make the adoption of our proposal flawless and fully backward compatible.

Performance Considerations. We implemented our extension on a Pixel 3A running Android Pie (pie-qpr3-b-release). Our patch consists of less

than 30 lines of code and modifies only two components of the Android framework (the policy parser and the `isCleartextTrafficPermitted` API). Once the patch is applied to the system, all the networking libraries that were honoring the Network Security Policy (see Table 3.1) can take advantage of our extension. We measured the overhead of our extension with a microbenchmark. This benchmark consists in making multiple HTTP connections and measuring the time needed by the `isCleartextTrafficPermitted` API to perform all the required checks to identify the origin the requests and accept or deny the connections based on the configuration of the policy. More in detail, for our benchmark we wrote an app that performs 1,000 HTTP requests using the *OkHttp3* library. We then run the app 100 times, with and without our modifications, and we compute the difference. The average execution time of the `isCleartextTrafficPermitted` API, without our modification, is *4 μs* with a standard deviation of *6 μs*. The average execution time of the same API with our modification is instead *300 μs*, with a standard deviation of *94 μs*. We believe that the overhead of our defense mechanism is negligible, especially when compared to the overhead incurred by network I/O operations.

3.9 Limitations

Even though our implementation raises the security bar of the current Network Security Policy, we acknowledge that it currently suffers from some limitations. First, it is important to mention that, since we operate with the same threat model of the actual Network Security Policy, we do not protect the application against malicious third-party libraries that want to evade the policy defined by the developer. We note that this affects the standard Network Security Policy as well: in fact, a malicious library can bypass even the strictest security policy by performing network connections with its "custom" API or by using native code.

A second limitation relates to the fact that we rely on the stack trace to identify which component initiated the network connection. We acknowledge that there may be benign situations where the stack trace cannot be fully trusted and there might be the risk of losing the real "caller," for example, when using dynamic code loading or threading with worker threads. A very detailed analysis of the potential problems of using the stack trace to perform "library compartmentalization" has been studied in FlexDroid [SKC+16]. Even if the current threat model of FlexDroid is considering malicious libraries, we believe that their proposal of a secure inter-process stack trace inspection combined to our defense mechanism

might create a full-fledged implementation to tackle the compartmentalization problem.

To conclude, we currently support only the `cleartextTrafficPermitted` attribute for the `package-config` tag. However, note that some features already provide a sufficient granularity and do not need to be sandboxed on a "per-package" basis. For example, the Certificate Pinning feature already creates a sort of "per-site sandbox," and it thus does not need to be restricted to a single component of the application. For the debug tag, instead, we believe that the current Network Security Policy works well since its content is not taken into account once the application is installed on the device.

3.10 Related Work

There are several areas of works that are relevant to this research: network security, the dangerousness of "code reuse," and advertisements.

For each of these categories, we now present the state of the art, illustrating how related and complementary work to ours has been addressed in the past.

3.10.1 Network Security

Network security on Android has been the subject of several studies who have analyzed several of its issues. A concept similar to the Network security has been first introduced by Fahl et al. [FHP+13]: this work proposed a completely new approach to handle SSL security, allowing developers to easily define different SSL configurations and options, like certificate pinning, just by using a XML policy. Thus, [FHP+13] completely prevents the developer to write any code responsible of handling the validation and verification of a given certificate, addressing multiple problems at their roots. Damjan et al. [BHMW16] performed an analysis over 50k Android applications aimed at identifying and studying their implementation of SSL/TLS. Their work, focusing on the custom implementation of the *TrustManager* component, showed how more than 20% of the applications embedded a broken and vulnerable implementation of the component, thus exposing the applications to network attacks, even on secure connections. To solve the problem, [BHMW16] propose a new defense mechanism to overcome the issue of broken SSL/TLS implementations named *dynamic certificate pinning*, leveraging dynamic instrumentation to patch, at runtime, broken implementation of the *TrustManager* component. Another category of work, again focused on SSL and Secure Connections, instead tried to enumerate

all the possible issues that can lead to having an incorrect configuration of SSL, and to measure how extensive and common these are among applications, adopting both static and dynamic analysis approaches. One such example is by Fahl et al. [FHM+12] who presented "MalloDroid:" in this work, they applied static code analysis to identify applications with SSL/TLS code that is potentially vulnerable to MITM attacks. Another major contribution that is brought by this work is the exhaustive categorization of all possible misconfigurations that can lead to this type of attack. Hubbard et al. [HWC14] and Onwuzurike et al. [OC15], instead, applied a combination of static and dynamic analysis to identify SSL vulnerabilities in popular Android applications. [HWC14] analyzed issues related to the absence of a standard way of alerting a user of an SSL error, and showed the possibility of attacks on applications running on both Android and iOS platform. [OC15] instead, dynamically confirmed the problems described in [FHM+12], showing how a network attacker can access sensitive information, including credentials, files, personal details, and even credit card numbers. Razaghpanah et al. [RNV+18] instead, measured the adoption of different libraries performing SSL/TLS operations by fingerprinting their handshake while Oltrogge et al. [OAD+15] analyzed and measured the adoption of certificate pinning amongst Android application. The analysis performed in [OAD+15], thanks to a survey that involved the developers, have discovered that the implementation of pinning is considered complex and hard to correctly implement. This confirms the importance of having more and more abstract and high level solutions, like the Network Security Policy, that allow the developer to better configure these protections, without the risk of introducing bugs at the code level.

To conclude the work on network issues, numerous studies have also measured how many applications use unencrypted communication channels to exchange data. Vanrykel et al. [VAHD17] study how applications send unique identifiers and sensitive information over unencrypted connections exposing the user to privacy threats. But the adoption of HTTP does not just lead to privacy issues, it also leads to severe security issues that can completely compromise a vulnerable application or the entire device. In fact, the works by Poeplau et al. [PFB+14] and Choi et al. [CK18] showed how several applications are vulnerable to remote code injection due to code updating procedures over HTTP.

3.10.2 Code Reuse

Our research confirmed the practice of copying potentially vulnerable code from the Internet. Several works highlighted how developers rely on on-

line platforms like *StackOverflow* for their development process. Linares-Vásquez et al. [VBP+14] analyzed more than 213k questions on StackOverflow (related to Android) and built a system to pair a given snippet of code of StackOverflow with a given snippet of code within the Android framework. With this study, they also highlighted how this practice—ask questions and change the code—is increasingly common when the behavior of a given API changes. Fischer et al. [FBX+17], instead, measured the proliferation of security-related code snippets from StackOverflow in Android application available on Google Play. [FBX+17] showed how more than 200k applications contain copy-pasted security-related code snippets from StackOverflow. This issue though, does not just afflict the Android ecosystem and its developers. In fact, a similar work, not focused on Android, is from Verdi et al. [VSA+19] in which they investigated security vulnerabilities in C++ code snippets shared on StackOverflow. They showed how 2,859 GitHub projects are still affected by vulnerabilities introduced by vulnerable C++ code snippet copied from StackOverflow.

3.10.3 Advertisements

Another aspect that we have analyzed in the course of our research is related to the advertisements used by developers, and how these libraries, due to the complex ecosystem in which they are organized, can introduce security issues. This problem has also been analyzed over the years, both in terms of privacy and security, and numerous solutions have been proposed. The first category of works studies ad libraries to identify the privacy implications for the user. Book et al. [BPW13] tracked the increase in the use of ad libraries among applications and highlight how the permissions used by these libraries may pose particular risks to user privacy. Son et al. [SKS16] instead, demonstrate how malicious ads can leak the PII of the user. Last, Stevens et al. [SGC+], showed how users can be tracked across ad providers due to the amount of personal information sent from the ads libraries and expose how these libraries checked for permissions beyond the required ones to obtain more PII.

The second group of works, instead, focuses mostly on the security impact of ad libraries and proposes different solutions to achieve privilege separation for applications and ads. AdDroid [PFNW12] proposes a new advertisement API to separate privileged advertising functionality from the host application, thus allowing the application using AdDroid to show advertisements without requesting privacy-sensitive permissions. AFrame [ZAD13] and AdSplit [SDW12], instead, propose a different "isolation" approach to let ad libraries run in a process separate from that of the application.

[ZAD13] proposed a new XML tag `<aframe>` to embed in the application Manifest. Thus, at installation time, the system will parse the new element and will create a new isolated "user" that will host the advertisements for the given application. The approach used by [SDW12] instead, implements this isolation by rewriting the application. In fact, AdSplit automatically recompiles the application to extract its ad services and run them in a separeted process, owned by a different user.

Chapter 4

Securing the System Layer: the Phishing Problem

4.1 Introduction

One of the key security features of Android is the application sandbox. This mechanism aims at enforcing a strong security boundary between different applications and protects sensitive information. One of such sensitive information is the "state" a given application is currently in. With "state," we refer to, for example, whether an application is currently in the *background*, in the *foreground*, or is *transitioning between these states*. Attacks aiming at determining the state of another application are called *state inference attacks*, which are particularly relevant in the context of phishing attacks.

Phishing attacks consist of luring an unsuspecting user into revealing her sensitive information (e.g., credentials) to a malicious application that mimics the UI of the legitimate one, a technique we refer to as *UI Spoofing*. A recent research conducted by Kaspersky Lab has highlighted how the threats posed by phishing application, such as "Banking Trojans," are increasing, counting only in 2020 a number of installations for these malware equal to 156,710, almost double the number recorded in 2018 [bK21]. The peculiar problem of mobile platforms is that the user cannot understand whether she is inserting her credentials into a legitimate or into a malicious application spoofing its UI. State inference attacks play a key role in this context since, if the malicious application can infer, for example, that *the user is about to use a specific application*, it can show the spoofed UI at the *right time*, and hijack the legitimate app's flow.

In the context of Android security, malicious applications are able to leak this state-related information by exploiting vulnerable APIs or resources (e.g., `/proc` file system). For example, a vulnerable API, when invoked with specific arguments, may return data that can be used to infer whether another application was just started, thus allowing the attacker to infer the state of the target app.

These attacks have been known for several years, and previous works have shown that several APIs and resources do leak sensitive information [CQM14, BCI+15]. Given the security implications of these vulnerabilities, Google has restricted access to the `/proc` file-system—eradicating potential bugs at its root—and fixed all APIs known to be vulnerable [Kra17]. However, as for many forms of bugs, this is an arms race and there can potentially be many more vulnerable APIs left to discover.

Therefore, moved by the challenge created by this arms race, we design, implement, and evaluate a new analysis framework to automatically pinpoint Android APIs that may disclose state-related information about other applications or the operating system itself. The main idea is to first

systematically enumerate the attack surface in terms of which APIs could be potentially abused by a malicious application, to then repeatedly invoke each API—with appropriate arguments—while changing the surrounding context (e.g., another application is started), and finally, to monitor how the returned values change, if they do, depending on such context. We note that we are not the first ones to propose this research direction. A recent work that tackles a similar problem is SCAnDroid [SPM18], which attempts to employ a technique similar to ours. We shows that while SCAnDroid's direction is indeed promising, there are several conceptual and technical challenges that were overlooked, leading to undetected vulnerabilities.

One of the main problems we uncover is that previous works have mischaracterized the attack surface available to a malicious app, leading to many APIs to not be even selected as candidates for analysis in the first place: our analysis shows that it considered only ∼44% of the attack surface. One other open challenge is *how* each of these APIs should be analyzed to uncover potential problems, and previous works oversimplified this step as well, missing out vulnerable APIs. As the last example, we found that even the task of determining whether the return value of an API contains sensitive information can be challenging, and we find that this is another venue for mistakes.

Thus, we systematize these challenges and discuss how we solve them: we show that *each of these overlooked challenges is the direct cause of false negatives of the closest related work, SCAnDroid.* A detailed comparison with this work is performed in Chapter 4.6.7.

We tested the effectiveness of our framework on Android 8.1, 9.0, and 10, unveiling 18 previously unknown bugs. All the vulnerabilities were reported to Google and several of these have been already acknowledged and fixed, leading us to get assigned 6 CVE.

While we believe that our framework is a good first step to automatically detect this category of bugs, we also acknowledge that identifying and removing *all* vulnerable APIs is not always possible. This issue is caused by the fact that our and existing analyses are based on static and dynamic analysis techniques, which can be *conceptually affected by false negatives.*

To try to overcome this issue and to reduce the number of false negatives to a minimum, we decided to look for a complementary solution. We design and implement an on-device monitoring system to detect state inference attacks when they occur. This system builds on two observations: the first is that *all* existing state inference attacks implement a *polling behavior*, thus making it per-se a good candidate for detection. The second observation, instead, which to the best of our knowledge has not been explored before,

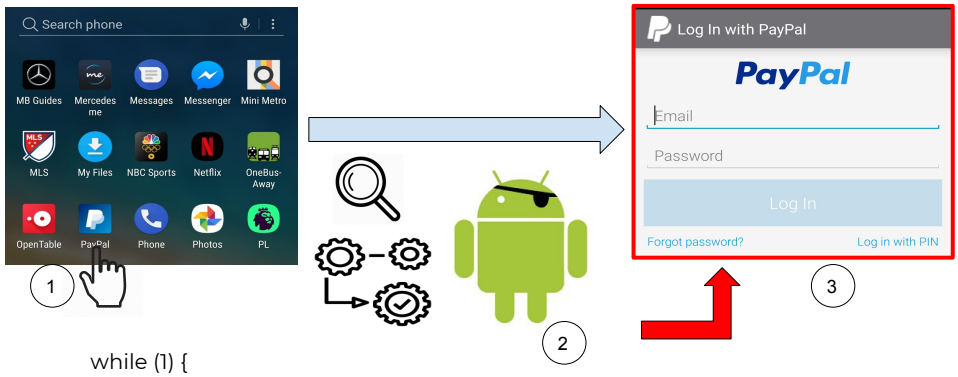
is based on the following key hypothesis: *benign applications rarely rely on polling and, when they do, the nature of their behaviors is different than those of malicious applications.* In other words, *if benign applications do not commonly employ polling*, the mere detection of these behaviors could be then used as a strong signal for flagging an application as suspicious.

To verify the validity of our hypothesis, we performed an empirical study over *all* known families of malware exploiting vulnerabilities to perform phishing attacks, as well as on a set of more than 10,000 popular benign applications. The results of this experiment show that, as expected, all malicious samples implement some form of polling when mounting state inference attacks. For what concerns the benign applications, our study unveils a surprising insight: there are several benign apps that also perform polling; However, more in-depth experiments show that these behaviors are of different natures, and it is easy to distinguish between them and their malicious counterparts. We thus show that polling itself can be leveraged as a strong signal to detect state inference attacks. We implemented this system as a modification to the Android framework, and our experiments show that this system would incur a negligible overhead.

We note that using "polling detection" as a mean to identify malicious applications is not novel per-se: a previous work, Leave Me Alone [ZYN+15], has explored this aspect. However, we show how this related work is not suitable when tasked to detect phishing attacks on modern versions of Android. We offer a detailed comparison in Chapter 4.9.7. We thus believe that our work discusses a new interesting point in the design space of detection approaches.

In summary, the study presented in this chapter makes the following contributions:

- We systematize and pinpoint open challenges to tackle the automatic detection of APIs vulnerable to state inference attacks. Among these, we show that the attack surface is bigger than what previously thought.
- We implement an automatic framework to unveil vulnerable APIs leading to state inference attacks. We tested its efficacy on Android 8.1, 9.0, and 10, identifying 18 new vulnerable APIs, 6 of which obtained a CVE identifier.
- We hypothesize that the mere polling can be used as a strong signal to identify in-progress state inference attacks. To validate our hypothesis, we performed an empirical study on both malware and benign applications, and we show it is indeed possible to reliably and




```
while (1) {
      ActivityManager am = (ActivityManager)
      Context.getSystemService(ACTIVITY_SERVICE);
      List<ActivityManager.RunningTaskInfo> apps = am.getRunningTasks(1);
      String app = apps.get(0).topActivity.getPackageName();
      if (app.equalsIgnoreCase(TARGET_APP)) {
        // Perform the attack
      }
}
```

Figure 4.1: Anatomy of a Phishing Attack.

efficiently pinpoint attacks. This can be the basis for an on-device detection system that does not have the limitations affecting previous works.

4.2 Phishing Attacks on Android

We now provide the technical background about phishing attack and how a malicious application can successfully mount it on an Android system. Then, we present a systematic survey on all known classes of state inference vulnerabilities, their role in the context of phishing attacks, and which of these classes are still problematic on recent versions of Android.

4.2.1 Phishing

One common task of Android malware is "phishing." With this term, we refer to malicious applications trying to steal user's sensitive information (e.g., credentials). Phishing attacks are particularly problematic for mobile platforms because they do not provide enough information for a user to reliably distinguish a legitimate application from a malicious app spoofing its UI. To make the attack even more effective, malware relies on the ability to mount state inference attacks, useful to monitor when the user is about

to interact with a target application. Inferring the right time is important, as it allows a malicious application to ask for user's credentials exactly when the user expects to insert them.

We note that these techniques are not only known and studied in the academic world [FQCL17, CQM14, RZX+15, SKGM18, SPM18], but they are used by real-world malware [Jag18, Thr19, WPN17, PNC17, Kev17, Nic18, Thr18, Kas16, Vit19].

4.2.2 Anatomy of a Phishing Attack

Figure 4.1 depicts the various phases of a phishing attack. We start from a scenario where the user (①) wants to interact with a sensitive application (e.g., PayPal). Meanwhile, in the background, the attacker (②) repeatedly invokes the once-vulnerable API `getRunningTasks` API to determine which application is in foreground. Please note that, to mount this specific configuration of phishing attack, the attacker needs to have malicious code—like an application—installed on the device. Before the victim clicks on the PayPal icon, and thus starting the PayPal application, the attacker could determine that the foreground application that occupies the screen is the "Home Launcher." However, by repeatedly invoking this API and checking its return value, the attacker could mount a state inference attack and infer the exact moment the user clicks on the PayPal icon: the attacker would in fact notice the transition from the Home Launcher to the PayPal application. At this point, the attacker knows this is the best time to hijack the PayPal activity with a spoofed one which looks the same as the original (③). A successful attack will leave the user completely unsuspecting since the victim initiated the interaction with the target application herself, she would not find an authentication request from that particular target application unexpected.

4.2.3 Characterizing State Inference Attacks

Previous research identified many venues to mount state inference attacks [FQCL17, CQM14, RZX+15, SKGM18, SPM18]. With the goal of better characterizing this threat and to better understand the state-of-the-art of Android state inference attacks, we analyzed all the different vulnerabilities exploited by malware and discovered during the years [FQCL17, CQM14, RZX+15, SKGM18, SPM18, Jag18, Thr19, WPN17, PNC17, Kev17, Nic18, Thr18, Kas16, Vit19]. All existing vulnerabilities can be grouped into two conceptual categories:

Filesystem layer. The first category relates to the *filesystem* layer. The root cause of these vulnerabilities resides in the presence of sensitive information obtainable by reading files accessible by any unprivileged application. From the technical standpoint, all known vulnerabilities are caused by unrestricted access to `procfs`, via the `/proc` directory. For example, one of the first vulnerabilities relied on accessing `/proc/$PID/cmdline`, which contains the name of the program run by a process with a given `$PID`. By continuously monitoring the content of this directory, the attacker could identify the creation of new processes (by monitoring sub-directories of `/proc`), and infer the application started by the user (by reading the `cmdline` file).

Many similar vulnerabilities were discovered, but they all had the same root cause: unprivileged applications had access to `procfs`. Thus, to patch these vulnerabilities, from Android 7.0 the access to almost the entire `/proc` directory is forbidden thanks to a more strict SELinux policy. We believe this solution eradicates this category of vulnerabilities at its root.

Android System Services layer. The second category of vulnerabilities relates to *Android System Services*. Services are a fundamental sub-system in the Android Framework. They allow applications to interact with "lower" operating system and hardware components, such as GPS, network, etc. Since this operation normally requires interaction with privileged components, services are offered by a process called `system_server`, which runs as the privileged `system` user. This process is in charge of handling almost all the core services and provides a bridge between the functionality requested by the application and the service implementing it.

We note that *all* API-related vulnerabilities identified by previous works relate to APIs exposed by services. Some of the root-causes identified over the years are *weak protection mechanisms* and *wrong sanitization of the return value*. For instance, the once vulnerable API `getRunningTasks`, implemented in `ActivityManagerService`, was affected by both problems. The API was not protected by any permission and the output contained detailed information about which application was used by the user as well as all the other applications running on the device.

Even though Google has fixed all known vulnerabilities, the complexity of the services infrastructure makes it significantly more challenging to identify the root causes that led to all existing vulnerabilities.

It is important to emphasize that the *filesystem category* is easier to identify and fix: applications are not supposed to interact with these files and there are no legitimate use cases for an application to directly interact with files inside `/proc/$PID/` directory not belonging to its own `$PID`.

Thus, Google forbids the access to almost the entire `/proc` directory. However, fixing *Android System Services layer* is not that simple and far from trivial: benign applications are supposed to use all the APIs exposed by the framework, and identifying whether a given API is leaking sensitive information or not is not an easy task. For example, the information can be exposed only when a certain event happens or can be "hidden" in a nested field of the return value. This chapter focuses on identifying this category of state-inference attacks and highlights all the technical challenges addressed to identify vulnerable APIs and prevent their abuse. We show also how there are several previously overlooked challenges and subtleties that make the automatic vulnerability discovery process more difficult than what previously thought, and that this is the direct cause for false negatives in recent related works [SPM18].

4.3 Threat model

We consider a threat model in which an attacker controls a malicious application on the victim's phone. We also assume that such app can ask (and obtain) those permissions that are usually available to non-system third-party applications. Some of these permissions are automatically granted, while others require user interaction. An example of a permission automatically granted is the `INTERNET` permission: at installation time, the system grants this permission to the application and no user interaction is required. Instead, examples of permissions that require user interactions to be granted are `ACCESS_COARSE_LOCATION` or `PACKAGE_USAGE_STATS`. Note that, in Android, this *interaction* may be implemented in two ways. The first type of interaction relies on *runtime prompt* and it is used to grant the permissions labeled as *dangerous*, like the `ACCESS_COARSE_LOCATION` permission. By interacting with this prompt, the user can decide whether to grant or deny the permission to the application. The second type of interaction, which does not rely on prompts, is reserved for privileged permissions. These permissions might be labeled as *signature, system, signatureOrSystem, privileged, development, appop, or retailDemo*. An example of this category of permission are the `PACKAGE_USAGE_STATS`, `SYSTEM_ALERT_WINDOW`, and `BIND_NOTIFICATION_LISTENER_SERVICE` permissions. For example, the `PACKAGE_USAGE_STATS` permission is used to mainly protect the *UsageStatsManager* service [Goo].

With that being said, Android offers a mechanism for third-party applications to obtain sensitive information accessible only via these permissions,

even without technically being granted such permissions. From the technical standpoint, the way it works is that a third-party application can ask the user of the device to grant the permission through the System Settings, which updates some internal settings. The sensitive system services that do have the signature permissions then check such settings to determine whether a requesting application is entitled to have access to such sensitive permission-protected information. One may erroneously think that a third-party application cannot get these permissions. However, we also note that not only is it possible to access information protected by these signature-level permissions, but that many real-world applications (both benign and malicious) currently use them [Nic18, YLC+19, Bro16]. Thus, since third-party applications may require some of these permissions, we believe it is appropriate to consider them within our threat model.

We also assume the malicious application *cannot* obtain the `BIND_ACCESSIBILITY_SERVICE` permission (`a11y`): this permission alone allows an attacker to fully monitor all UI events [FQCL17], making mounting phishing attacks trivial. Finally, we do *not* consider the scenario where a malicious application can gain root privileges nor perform any privilege escalation: once again, these powerful attackers can easily steal sensitive information without mounting phishing attacks.

4.4 Exploring the Attack Surface: System Services

This study aims at developing an automated approach to identify vulnerable APIs that could be used to mount state inference attacks. For the aforementioned reasons, we focus on considering the attack surface exposed by *System Services*. We now present the inner workings of system services and the known security-related pitfalls that affect this component.

4.4.1 Android System Services

System Services are a fundamental sub-system in the Android Framework, and they are the key mechanisms for applications to interact with low-level, security-sensitive operating system and hardware components. The technical details of these mechanisms, and how third-party applications can rely on them (by means of invoking Android APIs) are not trivial, and it involves several sub-components, discussed next.

Figure 4.2 gives an overview of how system services work. In the example, the goal of the application is to interact with the ActivityManagerSer-

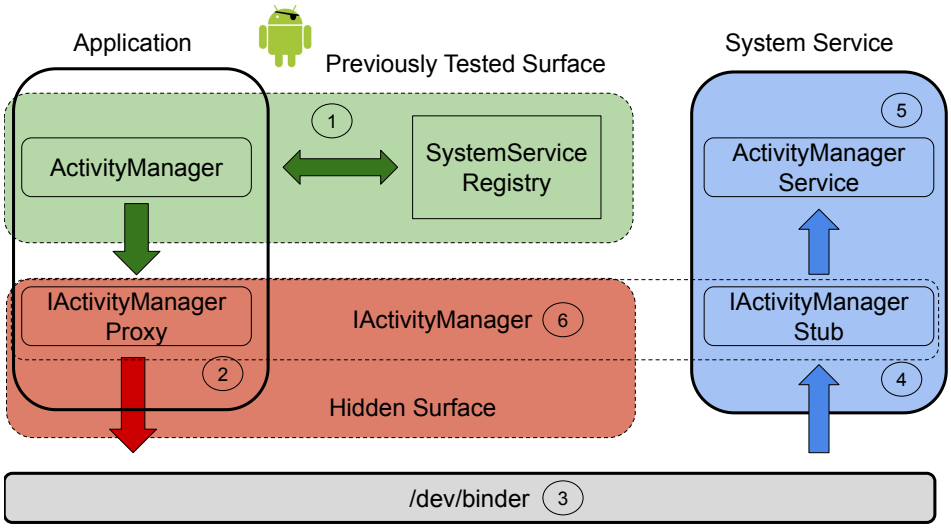


Figure 4.2: Interaction with ActivityManager System Service.

vice: to do so, the app needs:

- to first request a "client" — named `Manager` (1) — to the `SystemServiceRegistry` class to interact with the service.
- Once obtained, the application can start invoking the methods exposed by the Manager.
- Each method invocation is then wrapped and forwarded to another component, named `Proxy` (2), in charge of sending the data from the application to the `Binder` component (3).
- This component "forwards" the request to its associated `Stub` (4), which can be seen as the counterpart of the Proxy, residing in the Service.
- Finally, Stub forwards the request to the actual implementation of the `Service` (5).

The response follows the same, but reversed, flow.

Another important technical aspect is represented by the Interface (6), written in AIDL (Android Interface Definition Language). AIDL is an Android-specific language used to define the methods exposed in the `Stub`

that can be reached from the `Proxy`. The actual code can then be automatically generated by the AIDL compiler for both components, according to the specification defined by the Interface.

4.4.2 Known Potential Pitfalls

The complexity of system services opens to many potential vulnerabilities. One specific aspect that has been explored by previous works relates to inconsistencies in the placement of security checks like permission enforcing or identity control [ZYH+18, SCM+16, GCY+16]. The common root cause is that the checks were performed only in the Manager and not also in the Service counterpart. Thus, a malicious application could use a lower-level Proxy to communicate directly with the Service, bypassing the security checks. Another bypass that can be achieved by invoking directly a Proxy to communicate with the remote component relates to the *argument creation*. Indeed, the Manager is also in charge of instantiating some of the arguments for the method call it wraps. Previous works show how the arguments filled by the Manager component were not correctly validated at the Service side. Following the same flow described above, an application using a Proxy can craft an arbitrary argument and send it directly to the Service, bypassing the validity checks on arguments. The same logical issue affects not only the creation of arguments, but also how the return value is handled. Sanitizing the return value of a method invocation is an important process which avoids the leak of sensitive information. All the information that are not belonging to the caller of the method should be removed from the return value. However, "where" the sanitization is performed is crucial: if it is performed at the Manager layer, an attacker would still be able to obtain all the "raw data" just by interacting with a Proxy.

All these existing vulnerabilities have been fixed by Google and do not pose a threat in recent versions of Android. However, we show that this "layered" architecture still leads to new challenges and that they play a key role when looking for APIs vulnerable to state inference attacks. While the layered architecture is known to create problems in terms of placement of security checks, we believe we are the first ones to show how this complex architecture affects other security aspects as well.

4.5 Technical Challenges

One key contribution of this study consists in the design and implementation of an automatic framework to identify vulnerable APIs leading to

state inference attacks. We now discuss an overview of the several technical challenges we faced while designing this system, most of which have been overlooked by previous works and were a direct cause of false negatives.

The closest related work to ours is SCAnDroid [SPM18]: while we agree that its direction is indeed promising, we uncover numerous issues that limited its results, leading it to fail to identify numerous vulnerabilities. Later, in Chapter 4.6.7 and 4.7.4, we present a complete analysis that compares our system and the one proposed in [SPM18] both in terms of its functioning and results obtained.

Enumerating the attack surface. The first key challenge is to determine the effective attack surface available to a potential attacker. Past works analyzed client- and server-side APIs and they highlighted security-relevant differences [ZYH+18, SCM+16, GCY+16]. However, we show that there are server-side APIs (available to an attacker) that *do not have their associated client-side API.* There is thus a "hidden" layer of APIs that has not been considered by previous works. This makes previous approaches that enumerate the attack surface by only checking the client-side API significantly incomplete. In fact, in an attempt to quantify how much attack surface is "missed" we performed static code analysis on the Android framework itself and found that, *in the best case*, only about 44% of the attack surface is considered. A detailed analysis of this measurement will be presented later in this Chapter 4.7.4.

Argument creation, validation, and System stimulation When directly invoking server-side APIs, one has to determine how to create "valid" arguments, otherwise the API may just return an error. We also note that, by interacting with the server-side API, one has even more flexibility in terms of argument creation since the client-side-only sanitization routines (if any) are bypassed. However, creating a successful object automatically is not so immediate and hides many challenges. For example, even a single field of a complex object, if not initialized correctly, can lead to the generation of exceptions with the risk of completely blocking the automatic analysis process. Another important challenge consists in properly stimulating the system to induce the information leak. It is important to give, or create, the chance to the vulnerable APIs to actually leak sensitive data.

Systematic inspection of return values. One last overlooked challenge relates to how properly inspect values returned by an API. Previous works have relied on invoking every public (and private) method of the returned object, hoping to access fields that could be interesting for an attacker. However, this approach has several problems. First, the proper order of the

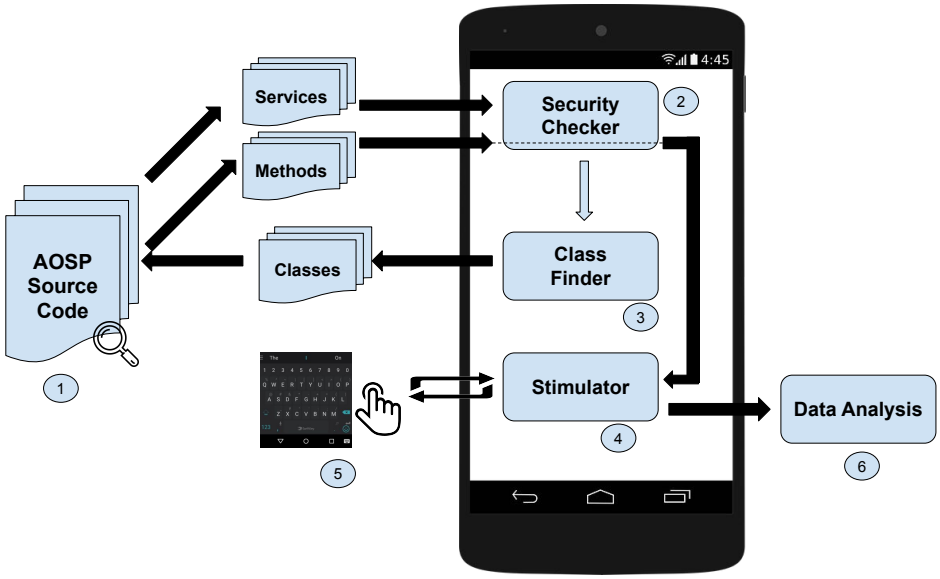


Figure 4.3: State-Inference Vulnerabilities Finder Framework.

invocations is unknown and may make a difference: for example, invoking a setter method before a getter method may cause the field value to be overwritten and permanently lost. Second, a client-side API may have access to some security-sensitive information, but it may "sanitize" the information before returning it to the caller. Even if the sanitization is not present, there can be private fields that are not accessible via the object's methods — not even the private ones. We found that, if not handled properly, this is yet another direct cause for false negatives.

4.6 Analysis Framework

We now introduce our new analysis framework. We start by presenting an general overview of how the framework works, we then discuss the various analysis steps and how we addressed various challenges. We conclude with a direct comparison with the most recent related work, SCAnDroid [SPM18].

4.6.1 Overview

Our analysis framework is constituted of several steps, each of which tries to solve one of the challenges listed above. The first step enumerates the attack surface and its APIs, as described in Chapter 4.6.3. Then, we analyze each

API to determine if it leaks sensitive information about other applications. The framework starts invoking it several times while keeping the system "at rest" (i.e., without performing any other operations). Then, it starts a victim application (the actual app used for this part of the experiment is not relevant), while it keeps repeatedly invoking the API under analysis — and logging every invocation and every returned object. To conclude, we post-process these logs to identify potential correlations between the returned value of a given API and changes in the surrounding environment (e.g., the moment in which the victim application has been started or retrieved from the background execution). The output of the analysis system is the list of APIs that could be potentially used to mount state inference attacks.

4.6.2 Analysis framework organization.

The framework is composed of six different modules, and is divided between an *on-device* and *off-device* analysis. Figure 4.3 provides a detailed overview. First, it *enumerates the attack surface*: this process involves three modules: the `Extractor` (①), which extracts Android services; the `SecurityChecker` (②), which removes "candidates" (i.e., Services or Methods) that are causing any kind of Security Violation when invoked; and the `ClassFinder` (③), which, for each service, extracts from the device the name of the classes implementing it (both Managers and Interface). Once the services and classes have been enumerated, the `Extractor` and `SecurityChecker` modules extract and analyze all the methods implemented by these classes. All methods whose invocations do not cause `SecurityException` are then automatically invoked by the `Stimulator` module (④), while, in the background, the `UI-Interaction Automator` module (⑤) injects different types of UI events to simulate a user starting a potentially sensitive application and her interaction. All the previous modules run on the device itself. Finally, the collected results are processed by the `Data Analysis` module (⑥). Note that this last analysis step is performed off-device.

4.6.3 Enumerating the Attack Surface

The enumeration of the attack surface is perhaps the fundamental component of our system. Its correct identification, however, is not as easy as one might think and many challenges lie behind this complex process. To begin with, Android offers multiple ways to register and expose a service to applications. Moreover, there is not a single central location to locate all the services inside the source code tree. However, we note that all An-

droid services should be exposed to the system by using one of the following methods:

- `addService`, in `SystemService` class, or
- `publishBinderService`, in `SystemService` class, or
- `registerService`, in `SystemServiceRegistry` class.

The parsing process is handled by the `Extractor` module and it is built on top of *JavaParser*, a parser for Java source code which generates the Abstract Syntax Trees [Tom15]. We parse the source code of the Android Open Source Project (AOSP) and extract all the services that are statically included in the system by looking at the methods listed above. To avoid missing any reference to a service, we extract the services running on our test device using the "`service`" command-line utility. The two lists are then merged together. Other works used the same approach to enumerate and list the services available in the Android OS [SCM+16, ZYH+18, GCY+16].

Note that a non-system application cannot interact with all services. Our threat model, as described in Chapter 4.3, assumes the attacker has control over a non-system application that can request any non-system permissions. However, some privileged services are protected by strict SELinux policies or by some permissions that only system applications can request and any attempts to access them cause a `SecurityException` to be thrown.

To enumerate the services that are accessible by an attacker, we perform a dynamic analysis step: first, we grant all non-system permissions to our test application, then we communicate with a given service while in the background we monitor for security exceptions and violations like SELinux runtime violation or security exceptions raised by missing permissions.

For those services that we can interact with, we enumerate the methods accessible to an attacker. To this end, previous works [SPM18] relied on the Android API reference documentation [LLC08]. However, this documentation only exposes public client-side methods: this approach entirely miss the "hidden layer" of server-side methods that do not have their respective client-side one. In our work, we do consider client-side methods, but we extend this enumeration by considering server-side methods as well, independently from whether they have a client-side counterpart.

Server-side methods are implemented starting from AIDL specifications. AIDL is an extension of Java and introduces some meaningful keywords that are adding information about the behavior of a given method. Since *JavaParser* is not handling AIDL as language, we wrote our own parser.

We note that, for certain aspects, AIDL is more expressive than Java. In fact, in AIDL, each method and arguments can be prefixed by so-called *keywords*. Among the many AIDL keywords, three of them are particularly important for our work. The first one is the `out` keyword, which specifies that an input argument "can be modified by the callee." All the primitive values are defined, on the opposite, with the keyword `in`, meaning that they cannot be changed. This helps us recognize this argument as a potential output value. The last relevant keyword is `oneway`. Like `out` and `in`, `oneway` gives us the information that the method is not going to "return" a meaningful result, but it instead returns immediately after having sent the data. It is normally used to register callbacks: since callbacks are normally triggered after an event occurs, a method in charge of register them can return immediately. From the documentation, we know that is not possible to have a situation where we have an argument marked as `out` and a method with `oneway` in its signature. Arguments that are not explicitly defined as `in` or `out`, have a default value of `in`.

Thus, our analysis proceeds in discarding a method if:

- its return value is `void` and none of its arguments are marked with the keyword `out`, or
- its return value is `void` and it does not require any argument, or,
- its signature shows that the method is defined with the keyword `oneway`.

We note that previous works did not consider these possibilities, leading to yet another venue for false negatives.

At a first glance, one can be tempted to always discard a method returning `void`, since it is not returning a value. However, this does not necessarily mean that no state inference is possible. In fact, an object passed as parameter can be modified during the method invocation by the callee, and the modification can then be potentially used to mount the attack.

As an additional filtering step, we also discard methods that have at least one argument of type `IBinder` since it is not possible, to the best of our knowledge, to obtain a reference to a valid `IBinder` token without relying on a permission granted only to system applications.

Previous works have addressed the API protection mapping problem [AZHL12, BBD+16b, ATH+18], however, we opted for a different and dynamic approach. For each of the potential candidate methods, we repeated the dynamic analysis monitoring for security exceptions, and keeping for further analysis only the ones not throwing any security violation since they are indeed accessible by a malicious application.

4.6.4 Stimulation Strategies

Once we collect the candidate methods, we then proceed to invoke them and analyze their returned values. The idea behind this enumeration is to perform a "guided testing" on these methods and to show how it can be helpful to unveil previously-unknown vulnerable APIs that are leaking sensitive data. The module in charge of the automatic testing is the `Stimulator` component.

Semantics-aware arguments generation. Understanding the "semantics" of an argument can be very helpful to improve the effectiveness of this step. This analysis step considers information taken from source code type information and the argument names. Our analysis extracts the following arguments' semantics:

- *Application identifiers*: this category contains all the arguments identifying a specific application installed on the device, such as *uid* or *packageName*;
- *Process identifiers*: arguments identifying a specific process running on the device, such as *pid*;
- *Filesystem locations*: every storage volume in the filesystem can be identified by a specific UUID such as *storageUuid* or *volumeUuid*;
- *Time values*: arguments related to time and time-ranges, such as *beginTime, startTime* and *endTime*.

Identifying and properly supporting these values allows us to maximize the likelihood that the target method will return something relevant since we passed an expected value. During our analysis, these arguments are initialized with specific values defined both statically or dynamically. For example, for arguments like *uid* or *packageName* we can statically define a value — such as the *uid* of the application we want to target. Instead, for arguments like *pid*, we need to retrieve them at runtime. Moreover, for time-related values, we enforce a "logical-constraint" such that *beginTime* will always be lower than *endTime*. For the rest of the arguments for which we do not have a semantic, we automatically instantiate objects with random content, as discussed next.

Generic argument creation. Even knowing the types of objects, it is not always trivial to create valid instances. In fact, the objects in the Android Framework can be very complex, and can contain many references to other

objects, each of which must be correctly solved in order to correctly create the final object. To invoke a given method, all the objects necessary to perform its invocation must be properly created and instantiated: this is, of course, a process that, if done manually, would be time-consuming. We decide to adopt an automatic approach and instantiate all the objects using a *recursive algorithm* that tries to instantiate an object by iterating through all the available constructors and recursively tries to create a valid sequence of object to match at least one of them. We repeat the process for each nested object and for all objects belonging to the method's signature. All the *primitive types* and their corresponding *wrapper classes* are filled with random values. Values for arguments marked as *semantically interesting* during the source code analysis are taken from a different bucket of pre-filled values.

Since a recursive approach may incur in *circular dependencies* and crashes due to the increasing size of the call-stack, we configure our system with a maximum threshold of five recursive calls. In cases of failure, we resorted to custom handlers. This was needed for 105 objects (2.4% of a total of 4,390 Objects that are defined in the Android Framework). We note that this is a one-off effort (that does not need to be repeated for each version of Android).

Argument generation strategies. The analysis runs in two configurations. In the first one, a method is invoked multiple times without changing the arguments. This means that, for every method, the arguments are created only once. In the second one, instead, arguments mutate at each invocation of the method. Having multiple configurations is important since this allows us to analyze different behavior and explore different execution paths. We identified situations where APIs were leaking sensitive information over time only with some particular arguments. For example, we found a vulnerable API leaking information only when one of its arguments (related to "time") was changing from the previous invocation to the new one. Testing this API without mutating the arguments would end up in wrongly marking it as "safe."

User-Interface interaction. Interesting APIs for an attacker are the ones leaking the current "state" of a target application. An application can be in one of the following main states: *background state*, when its UI is not shown on the screen, and therefore its content is not on top of the system's Activity Stack, and *foreground state* otherwise. When the user taps on the application icon, it initiates a "state transition" from background to foreground state. Viceversa, when the user taps on the "Home" button, the state shifts from foreground to background. In this work, we analyze and

trigger the following subsequent states:

- the application is not started yet: *background state*,
- the user opens it and interacts with it: *transition from background to foreground*,
- the user interacts with the application: *foreground state*,
- the user stops the app: *transition from foreground to background*,
- the application is not used: *foreground state*,
- last, the user resumes the application: *transition from background to foreground*.

The goal of this module is to inject multiple "events" while, in the background, the Stimulator repeatedly invokes the API currently under test. The automatic interaction with the UI tries to mimic the behavior of a real user. This module is built on top of `AndroidViewClient` [dtm15], a library which helps the creation of "Android test automation" scripts [dtm15].

4.6.5 Data Serialization

If the automatic creation of complex objects posed a challenge to solve, so it is creating a generic serialization method that can be applied to all objects returned the invoked APIs. To solve this problem, we implemented a custom serialization algorithm to store both *arguments* and *return value* collected by the Stimulator. The serialization algorithm follows the process we used to instantiate the different objects, but in a reverse order, following a *depth-first exploration strategy*. For each object, we start by defining a "child node" — represented by the actual object we want to serialize — we recursively dump all its fields and store both their name and value in a `key-value` format. For fields with "primitive" types, we store their textual representation, otherwise, we recursively apply the same algorithm to all its fields until we reach the "root" class, `java.lang.Object`. This allows us to unfold complex object in a flattened format — a JSON-like structure. If we detect circularity, we only store the reference of the object without recursively analyzing it a second time — none of the known JSON serialization libraries support it. This technique allows us to have a very detailed representation of a given object, including all (possibly private) objects that it encapsulates, no matter its complexity.

Available libraries which are normally used for this task suffer from some key limitations: for example, `GSon`, Google JSON Library [Goo12], does not automatically handle circular dependencies. Handling circular dependencies with `Gown` requires a manual effort and this means writing a custom serializer for each object having one of these dependencies which makes the approach not scalable.

4.6.6 Data Analysis

The last component of our framework is in charge of results analysis. Its main goal is to find which APIs can be used by an attacker to successfully mount state inference attacks. More in general, we want to automatically identify APIs whose return value is somehow influenced by the surrounding context and leak a meaningful value when the target application is going to be used by the victim.

Automatically identifying which APIs can potentially be used by an attacker hides many challenges. In fact, applying a too conservative approach may result in having a large number of false positives to analyze manually. The opposite problem is the case in which one adopts an overly restrictive approach, as there is the risk of eliminating a valid API and thus incurring in false negatives.

Our analysis, divided into two stages, represents a tradeoff. As part of the first stage, we start by considering all the collected API's return values. We consider the keys of these return values and we discard all keys whose value is constant across all the API invocations. It is safe to discard these keys because the attacker would not have any chance to infer any state-change just by observing a constant value.

Then, we identify those keys whose value is particularly noisy, i.e., the value has almost always a different value. These values are likely not providing a strong signal for the attacker, but we opted to err on the safe side and we proceed to further inspection before discarding them. In particular, we empirically found that the vast majority of these noisy values belong to one of the following categories: timestamps, incremental values (relative timestamps and auto-incremented sequential numbers), and pointers (i.e., memory addresses). We developed a simple, *entirely conservative* heuristics to identify whether a noisy value belongs to one of these categories, in which case, given their non-security-relevant semantics, we can safely discard them. We consider a key to be of a certain category if these conditions apply:

- *Timestamp* if, when all the values are converted in a "datetime" object,

the dates are always compatible with when we run the experiments;

- *Incremental values* if, when we calculate the difference between each consecutive value, these differences are always a small positive number;

- *Pointers* if all the values, when interpreted as memory addresses, would point to memory locations within valid, mapped memory pages.

We stress that, if we cannot recognize the semantics of a noisy key, we do *not* discard it and we consider it as a potentially interesting. At the end of this stage we obtain a set of candidate APIs for which at least one key has *not* been discarded; that is, these APIs have a chance to be useful for an attacker. We note that the APIs detected as part of this stage could already be interesting for an attacker, in the sense that these APIs are potentially returning a (changing) value that may be correlated with the outside environment. We then proceed to identify those APIs that can be used to determine state transitions of other applications.

The second stage is conceptually straightforward: we focus on identifying APIs that return the same value before the application has started, and that suddenly start returning a different value just after the user (in our case, the `Stimulator` module) has started the victim application or moved it from foreground to background. The resulting APIs are the final output of the analysis pipeline.

4.6.7 Comparison with SCAnDroid

As we mentioned throughout the chapter, we are not the first ones to propose an analysis framework to pinpoint Android APIs vulnerable to state inference attacks. We now offer a direct comparison with a recent work with a similar goal, SCAnDroid [SPM18], and we show how it overlooked several of the challenges discussed in Section 4.5. Most of these shortcomings are not just implementation issues, but they are about important aspects that were not considered.

The first group of shortcomings relate to how SCAnDroid determines the set of potential APIs to test. First, they only consider client-side APIs — the ones implemented in the Manager, by relying on the Android API documentation [LLC08]. Our analysis, instead, considers a wider attack surface — the full list of methods exposed by client- *and* server-side components. We determined the list of APIs to test by relying on the source code of the AOSP project. Second, to limit the number of APIs to analyze, SCAnDroid performs a filtering step by only considering APIs whose name

starts with a prefix such as *get*, *query*, *has* or *is*, assuming that only similarly named methods could constitute vulnerabilities. Our filtering process instead is based on the internal functioning of the Android system.

Our evaluation shows that these strategies allow SCAnDroid to potentially reach only ∼44% of the available attack surface. The detailed comparison on the final results is presented next, in Chapter 4.7.4. Last, we note how SCAnDroid cannot be easily extended to identify and support the test of server-side APIs, the ones reachable only via AIDL. These APIs are not accessible neither via *Reflection*, nor are described in the official documentation available to the developers. These two techniques are the ones used by SCAnDroid to enumerate the attack surface. Thus, it is conceptually and technically not possible for SCAnDroid to cover and analyze this important portion of APIs.

One other conceptual limitation relates to the limited ability to invoke APIs with proper arguments (e.g., pass a valid *process id* when needed) and, more importantly, how it inspects the return values. In fact, SCAnDroid recursively invokes all methods implemented by the returned object through *Reflection*, leading to two conceptual problems. First, the *order* these methods are invoked with may permanently modify the return value and some data may be lost. For example, invoking a *setter* method before the *getter* method of a specific field overwrites the field's value, potentially losing information. Second, and more importantly, *there is no guarantee that all information stored in an object is accessible via its public or private methods*. Our approach, instead, relies on a custom serialization that can recursively dump *every* field that is directly or indirectly stored within a given object, do not perform any operation on the object that can tamper the content of its fields—preserving all the valuable data— thus solving the problems of the previous approach.

Moreover, inspecting the return value though can hide another issue. As previously explained, obtaining the return value using the Manager can mean accessing a return value that is sanitized—by the Manager—and in which potential values that could lead to a state inference have been eliminated.

Our analysis found that these conceptual limitations are the direct cause of false negatives for SCAnDroid. In fact, our approach identified vulnerable APIs that were either not analyzed or for which the analysis wrongly marked them as "not vulnerable."

4.7 Evaluation

4.7.1 Experimental setup

We evaluate our framework's efficacy on three versions of the Android OS: Android 8.1, running on a Nexus 5X with the latest available security patch (with security patch, December 2018), Android 9, running on a Xiaomi MI A2 (August 2019). Finally, we also tested our system on the latest version available at the time of writing, Android 10. However, we noticed that our system was not able to identify any new vulnerability on this latest version, despite the fact that the attack surface had been correctly identified and several APIs had been tested. Moreover, we have also manually verified and confirmed that all bugs we identified affecting Android 8.1 and 9 have been correctly fixed on Android 10. Thus, since no additional vulnerabilities were found on Android 10, we will focus the discussion and the analysis of the results obtained on Android 8.1 and 9 versions.

4.7.2 Attack Surface Enumeration

Attacker-reachable services. Our system extracted a total of 160 services for Android 8.1. After having applied the filtering steps described in Chapter 4.6.3, it identified how a non-system application can interact and reach 100 of them ($\sim 62\%$).

For what concerns Android 9 instead, we identified 180 services, but only 95 reachable ($\sim 52\%$) from an unprivileged application.

As it is possible to see, for both versions of Android, the majority of the services not reachable by a third-party application is due to security violation. By monitoring these denials, in fact, our system identified how more than the 23% of the services for Android 8.1, and 25% for Android 9.0, were not reachable by a third-party application due to missing permissions or SELinux violations. This first filtering procedure applied to services has allowed our system to extract only those services that can actually be used by an attacker.

Table 4.1 shows how many services were not reachable and for what reason.

API enumeration. Starting from the extracted services, we then proceed by identifying and extract first the Manager and the server-side services implementation, and then the candidate APIs. On Android 8.1, the 100 services define a total of 157 classes. These classes are divided in 71 Client classes ($\sim 45\%$) and 86 Server ($\sim 55\%$). From these 157 classes, we identified a total of 6,219 invocable methods. These are all the methods that

Table 4.1: Extraction of attacker-reachable services.

Description	Android 8.1.0	Android 9
Available Services	160	180
Proprietary Services	2	16
`SELinux` Denials	35	44
`Runtime Permission` error	2	1
Unreachable Services	9	14
Native Services	12	10
Attacker-Reachable Services	100	95

Table 4.2: Distribution of the dataset in terms of inclusion of ad libraries and cleartext configuration.

Methods	Client-side		Server-side	
	Ver. 8.1.0	Ver. 9	Ver. 8.1.0	Ver. 9
Total	3,536	4,092	2,683	2,887
After static analysis	1,080	1,324	1,384	1,472

Table 4.3: Method Filtering Process.

# APIs	Fixed Arguments		Mutated Arguments	
	Ver. 8.1.0	Ver. 9	Ver. 8.1.0	Ver. 9
Total	2,464	2,796	2,464	2,796
Accessible by an attacker	1,616	1,931	1,614	1,929
By removing constant APIs	813	1,127	816	1,141
By removing noisy APIs	48	35	51	52
Unique Methods	66			
Potentially vulnerable	24			

can be potentially used by an attacker to mount state inference attacks. We then proceed by applying the filtering rules, as described in Chapter 4.6.3. This process allowed us to obtain, from the initial bucket of 6,219 methods, 2,464 candidates to test on Android 8.1. Out of these 2,464 methods, 1,080 are exposed through the Client while the remaining 1,384 are available from the Server. We then dynamically tested all these methods looking for security violations. These methods have to be discarded since a third-party application cannot invoke them. This stage identified how only 1,616 of them is effectively reachable by a third-party application. Thus, the combination of both static and dynamic analysis reduced the candidates from 6,219 to 1,616.

We then applied the same identification and filtering process to Android 9. For what concerns this version, from the 95 initial services, we extracted a total of 157 classes: 76 acting as Client ($\sim$ 48%) and the remaining 81 as Servers ($\sim$ 52%). From these classes, we then extracted a total of 6,979 invocable methods. The first static filtering allowed our system to extract, from the 6,979 methods, 2,796 candidates (1,324 methods declared in the Client, while 1,472 defined in the Server). Instead, by removing the methods raising a security violation at runtime when invoked, our system pinpointed 1,931 methods effectively reachable by a potential malicious application. Thus, the combination of these pruning strategies allowed us to lower the number of methods to test from 6,979 to 2,796.

Table 4.2 and Table 4.3 summarize the results obtained during these pruning stages.

We analyzed each method for an average of 70 seconds (60 seconds plus time used for booting with both the configurations of the Stimulator). The overall execution time to run all the experiments is of *63 hours* for Android 8.1, while it took *68 hours* for Android 9.

4.7.3 Analysis Results

We then proceed to analyze the data collected during the tests. We start by discarding APIs not leaking any sensitive information due to their values remaining constant, as well as very noisy APIs, as described in Chapter 4.6.6. This process drastically reduces the number of APIs to analyze in the second stage. For Android 8.1, we reduced the number of APIs from 1,616 to 51 — *discarding $\sim$96.6%*: for Android 9, we started from 1,931 APIs and we ended up with 52 candidates — *discarding $\sim$97.5% of APIs.* In total, we obtained 66 unique APIs whose return value change appears to be conditioned by the surrounding context. Out of the 66 APIs, the second stage of the algorithm identified 24 potentially leaking APIs that can be used

Table 4.4: Systematization of the vulnerable APIs.

Classname	Method	Permission	Hidden	Affected versions	Fixed?
IActivityManager	isAppForeground	None	**Yes**	8.1 and 9	**CVE-2019-9292**
IActivityManager	getProcessPss	None	**Yes**	8.1 and 9	**CVE-2020-0087**
ActivityManager	getProcessMemoryInfo	None	No	8.1 and 9	**CVE-2020-0372**
IUsageStatsManager	isAppInactive	None	No	8.1 and 9	**CVE-2020-0317**
INetworkStatsService	getUidStats	ACCESS_NETWORK_STATS	**Yes**	Only 9	**CVE-2020-0327**
INetworkStatsService	getDataLayerSnapshotForUid	ACCESS_NETWORK_STATS	**Yes**	8.1 and 9	**CVE-2020-0343**
StorageStatsManager	getFreeBytes	None	No	8.1 and 9	**Duplicate**
StorageManager	getAllocatableBytes	None	No	8.1 and 9	**Duplicate**
IActivityManager	isUidActive	PACKAGE_USAGE_STATS	**Yes**	Only 9	Won't fix
NetworkStatsManager	querySummary	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix
NetworkStatsManager	queryDetailsForUidTagState	PACKAGE_USAGE_STATS	No	Only 9	Won't fix
IActivityManager	getUidProcessState	PACKAGE_USAGE_STATS	**Yes**	8.1 and 9	Won't fix
IActivityManager	getPackageProcessState	PACKAGE_USAGE_STATS	**Yes**	8.1 and 9	Won't fix
NetworkStatsManager	queryDetailsForUidTag	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix
UsageStatsManager	queryEvents	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix
UsageStatsManager	queryUsageStats	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix
UsageStatsManager	queryAndAggregateUsageStats	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix
IStorageStatsManager	queryStatsForUid	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix
IStorageStatsManager	queryStatsForPackage	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix
NetworkStatsManager	queryDetailsForUid	PACKAGE_USAGE_STATS	No	8.1 and 9	Won't fix

to determine whether a target application went to "foreground." Table 4.3 summarizes all the intermediate results of these filtering stages.

Out of these 24 APIs, 18 are indeed vulnerable: 4 APIs require no permission at all, *2 require a permission marked as Normal*—and so automatically granted to third-party applications at installation time, while the remaining APIs are protected with the `PACKAGE_USAGE_STATS` permission, which allows an application to collect the usage statistics of other apps, *including the application in foreground*. This information, as discussed in Chapter 4.2.2, is of essence when mounting phishing attacks. Table 4.4, reports more detailed information about these APIs. For each API, we report the vulnerable service, the type of permission protecting it, if the API was present in the Manager or only in the Proxy component, which version contains the vulnerable API, and if the bug has been fixed.

We now discuss the 6 false positives. Interestingly, two APIs actually leak *some* information about the surrounding system: `getInputMethodWindowVisibleHeight`, which returns the size of the keyboard on the screen, and `getPendingAppTransition`, which tells the attacker that an application "is going to be moved on foreground." The attacker can reliably infer that *an* application is about to change its state, but she cannot determine *which* one. However, since this scenario could lead to a more generic phishing attack, we conservatively consider these as false positives. For example, with the `getPendingAppTransition` API the attacker can evince that the user is about to interact with an application: Thus, she can simply display a pop-up a message informing the user that an update is available—without the need of specifying the name of the application. Since the timing is perfect, the user might be lured into clicking it. The same attack can be mounted with the `getInputMethodWindowVisibleHeight` API. In fact, the attacker can infer when the user is going to use the keyboard, giving her the chance to show a popup informing the user that a keyboard update is available.

Two other APIs (*createAppSpecificSmsToken* and *DownloadManager.Query*) return very noisy values, which change at every invocation. We note how the filtering step described in Chapter 4.6.6 does not discard these APIs because the noisy values are not belonging to one of the categories known to *not* leak information (e.g., timestamps). In fact, a deeper analysis of these two APIs allowed us to confirm that their return value is either a *pseudo-random token* (for the first API) or an *object identifiers* (for the latter). Neither of the API, thus, return a value correlated with the current state of the system. This is another indication that our "filtering' is indeed conservative.

The last two APIs that are misinterpreted as potentially interesting are *launchLegacyAssist* and *getAllCellInfo*. Their values changed after the start of the target application but it does not appear to be correlated to the target application's state transition.

For completeness, we manually inspected the remaining 42 APIs out of the 66 that have been filtered out by the second stage. We identified how 7 APIs leak "system state" information, such as the total amount of bytes written by apps, or aggregate statistics about the disk usage. 15 APIs, instead, leak sensitive network information, like the overall network usage. We found that the remaining APIs do not seem to leak any relevant information.

An interesting observation comes from the vulnerable 3 APIs affecting only Android 9. In fact, they are all new features introduced in existing services, which were also available in Android 8.1. *This continuous evolution underlines the importance of having an automatic analysis tool to flag these potential problems.*

Disclosure. We disclosed our findings to the Android security team. *Six* APIs have been acknowledged and fixed by Google and a CVE was assigned. Table 4.4 provides a detailed list of the APIs fixed and the assigned CVE. We believe this confirms how seriously Google is considering this class of vulnerabilities. For what concerns the remaining APIs, the Android security team considered them as "won't fix" due to the type of permission protecting the API. However, it is important to highlight how these APIs are exposing to the attacker sensitive information about the state of the applications running on the phone. Moreover, we note how real-world malware already abuse similar APIs that require the same permission, as documented by recent findings by security companies [Jag18, Nic18]. We believe it would be possible to secure these APIs by adjusting the granularity of the information returned.

4.7.4 Results Comparison with SCAnDroid

To further illustrate the performance of our system, and to show how our contributions play a key role on the automatic identification of state inference vulnerabilities, we compare our results against those obtained by SCAnDroid on the same Android version — Android 8.1.

Overall, our system was able to correctly detect *all* the vulnerable APIs identified by SCAnDroid. However, we note that most of the vulnerable APIs identified by SCAnDroid belong to bugs that we categorized as leaking information related to the *system* (like the total amount of bytes written by

applications, or aggregate statistics about the disk usage) and *network* states (like the overall network usage), as previously described in Chapter 4.7.3. While these bugs are interesting and they can be exploited with *template attacks*, as showed in [SPM18], it is not trivial to weaponize them. In fact, creating a template for each of the application the malware wants to attack might be impracticable to adopt in a real scenario since it is highly dependent and influenceable from the system load, which can vary in a non-deterministic way, making the template inaccurate.

Our approach focuses on finding vulnerable APIs—and this is one first difference with SCAnDroid—such as those ones that allow an attacker to pinpoint which application the user is currently interacting with, or that at least do not require building "templates" for each target victim application. As presented in Table 4.4, only two of the bugs we found were marked as *Duplicate*, while all the other ones were previously unknown. All the APIs identified by our system are generic and are not related to a specific feature or configuration of a specific application, making our findings more generic and scalable.

Extending the attack surface allows us to examine components and methods that were not even taken into account by SCAnDroid. To determine how many methods SCAnDroid missed, we identified the server-side methods that are *not* reachable from the Managers. To collect this number, we first extracted all the server-side methods defined in the Android OS, version 8.1, obtaining 5,216 methods. Then, we extracted "interesting candidates," as previously discussed in Chapter 4.6.3: we identified that only *1,384* of them are actually potentially reachable by an attacker and thus represent the attack surface analyzed by SCAnDroid.

Since SCAnDroid uses as entrypoints only a subset of "client-side" methods, we then determined how many of the 1,384 methods are effectively reachable from the Managers. To this end, we computed a forward call-graph for each of the methods defined in the Managers. If one considers only client-side methods, only *835* methods, out of the 1,384, are potentially reachable, representing the ∼60% of the attack surface. However, SCAnDroid does *not* take into account *all* client-side methods, but it applies a filtering process based on the method's name. We applied the same filtering process on the client-side methods and found that SCAnDroid would be able to reach only *616* server-side methods, which is *only about the 44% of the attack surface.*

We also note how, for what concerns Android 8.1, the 33% of the bugs we identified (5 out of 15) resides in the server-side component. This shows, once again, that the server-side attack surface should not be overlooked.

Even more noteworthy is the fact that our analysis correctly detected 10 vulnerable APIs that satisfied SCAnDroid's filtering. Thus, these APIs have been tested, but were not marked as vulnerable. All 10 APIs are present in Android 8.1, are exposed in a Manager and match the prefixes constraints that would pass SCAnDroid's filter (e.g., *getProcessMemoryInfo*, *queryUsageStats*, or *queryAndAggregateUsageStats*). We believe that a possible explanation relates to how SCAnDroid stimulates the APIs or how it processes the return value. An emblematic case is *getProcessMemoryInfo(int[] pid)*. This API leaks statistics about the memory usage of running applications. However, to detect this leak, the API needs to be invoked with a list of valid "process id," otherwise a set of `NULL` is returned. We believe SCAnDroid might have misclassified this API due to not passing proper arguments. Since our system identified "pid" as a *meaningful argument*, our analysis handles this case and spots the vulnerable API.

This is another important result that shows how the argument generation we applied, described in Chapter 4.6, improves the effectiveness of the identification of vulnerable APIs.

4.8 Case Studies

We now present three case studies to demonstrate how the vulnerable APIs we identified can be used to mount phishing attacks. We opted to discuss specific instances of vulnerabilities highlighting three different categories of problematic APIs. Each one of the case studies presents a vulnerable API protected by a different class of permissions, and we report a concrete proof-of-concepts on how the APIs can be exploited in a real attack scenario. We start with the class of APIs that does not require any permission, we continue with one of those APIs that requires permission automatically granted at install time, and we conclude with an API that requires a *Privileged* permission that an attacker can ask the user to grant.

Note that to prove the feasibility of exploitation of all the vulnerable APIs identified by our system and listed in Table 4.4, we provided to Google, during the disclosure process, a Proof-Of-Concept for each API to show how it can be used to infer which application is going to be used by the victim.

4.8.1 CVE-2019-9292

The *isAppForeground* API is implemented by the *ActivityManager* system service: it takes as argument a Linux user id (UID) and it returns a boolean indicating if the application run, by this user, is in foreground. Since in

Android each installed application is assigned a different UID, and since the UID → mapping can be easily obtained, an attacker can invoke multiple times the API to check when the target application goes to foreground, which is the proper time to spoof its UI. This API thus represents the "ideal" case for an attacker, as she can monitor the state of any application installed on the device. This API does *not* require any permission, and it is only accessible via AIDL interface: that is, no client-side equivalent exists for this API, making this vulnerability impossible to be found via the client-side-only analysis. This API is referenced from the Common Vulnerabilities and Exposures system (CVE) with id CVE-2019-9292.

Listing 4.1: CVE-2019-9292: isAppForeground

```
1  void attack(int uid) {
2    final Handler handler = new Handler();
3    handler.postDelayed(new Runnable() {
4    /* Executed every second */
5    public void run() {
6      try {
7        /* Obtain a reference to IActivityManageService */
8        Method getServiceMethod = Class.forName("android.os.ServiceManager").
             getDeclaredMethod("getService", new Class[]{String.class});
9        IBinder binder = (IBinder) getServiceMethod.invoke(null, new Object[]{"
             activity"});
10       IActivityManageService iams = IActivityManagerService.Stub.asInterface(binder
             );
11       boolean res = iams.isAppForeground(uid);
12       if (res) {
13         /* Hijack the original activity */
14         startSpoofedUI();
15       }
16       handler.postDelayed(this, 1000);
17     } catch (Exception e) {
18         /* Handle the exception */
19     }
20   }, 1000);
21 }
```

4.8.2 CVE-2020-0343

The *getDataLayerSnapshotForUid* API is implemented by the *NetworkStats* system service, and it is only available through the AIDL interface. This API takes the *UID* of a target application and it returns a *NetworkStats* object encapsulating network statistics for said app. Our framework identified multiple fields leaking sensitive information; two of them — namely *set* and *txPackets* — can be used in combination to successfully mount a state infer-

ence attack. The *txPackets* field indicates how many packets the application transmitted since the boot, while the *set* field indicates whether the packets are sent while in foreground. When the malware notices an increment of *txPackets*, in conjunction with a change in the *set* field, it can infer that the target application is performing, for example, a login, and can react accordingly. This API requires the ACCESS_NETWORK_STATE permission: since this permission is "normal," it is silently granted at installation time and the user will not be notified. This API is referenced from the Common Vulnerabilities and Exposures system (CVE) with id CVE-2020-0343.

Listing 4.2: CVE-2020-0343: getDataLayerSnapshotForUid

```
/* First measure of txPackets */
public long prevTxPackets;
void attack(int uid) {
  final Handler handler = new Handler();
  handler.postDelayed(new Runnable() {
  /* Executed every second */
  public void run() {
    try {
      /* Obtain a reference to INetworkStatsManager */
      Method getServiceMethod = Class.forName("android.os.ServiceManager").
          getDeclaredMethod("getService", new Class[]{String.class});
      IBinder binder = (IBinder) getServiceMethod.invoke(null, new Object[]{"
          netstats"});
      INetworkStatsService inss = INetworkStatsService.Stub.asInterface(binder);
      NetworkStats ns = inss.getDataLayerSnapshotForUid(uid);
      /*
       * 1 is for foreground data
       * Check if the application is sending data and if is trasmitting in foreground
       */
      if (ns.set == 1 && ns.txPackets > prevTxPackets) {
        /* Hijack the original activity */
        startSpoofedUI();
      }
      prevTxPackets = ns.txPackets;
      handler.postDelayed(this, 1000);
    } catch (Exception e) {
        /* Handle the exception */
    }
  }, 1000);
}
```

4.8.3 Won't Fix

Amongst the 12 APIs marked as "Won't fix" by Google, we now present the *queryEvents* API. This API is implemented as part of the *UsageStats*

system service. It takes as input a range of time and returns a *UsageEvents* object, which embeds information about all the events triggered by the applications running during that time span. Our framework identified a number fields leaking information about the state of an application, which, if combined together, represent a valuable signal to mount a state inference attack. In particular, an attacker can combine *mPackage*, that indicates the package name of the application performing the "event," and *mEventType*, that specifies the type of the event. Note that other combinations are effective as well. In this case the attacker is interested in monitoring for a MOVE_TO_FOREGROUND event, which indicates that the application moved to foreground, the ideal moment to show the spoofed UI. This API requires the PACKAGE_USAGE_STATS permission, which the user needs to manually approve. Nonetheless, *real-world malware has been found in the wild that had the same exact requirements, showing that this request is legitimate* [Nic18, Thr19, Jag18].

Listing 4.3: Won't Fix: queryEvents

```
/*
 * Define start timer, and target app
 */
public long prevTime = System.currentTimeMillis();
public String TARGET_APP_PACKAGE_NAME = "com.target.app"
void attack() {
  final Handler handler = new Handler();
  handler.postDelayed(new Runnable() {
  /* Executed every second */
  public void run() {
    try {
        UsageStatsManager usm = (UsageStatsManager)getSystemService(Context.
            USAGE_STATS_SERVICE);
        UsageEvents ue = usm.queryEvents(prevTime, System.currentTimeMillis());
        prevTime = System.currentTimeMillis();
        while (ue.hasNextEvent()) {
            UsageEvents.Event e = new UsageEvents.Event();
            ue.getNextEvent(e);
            if (e.getPackageName().equalsIgnoreCase(
                TARGET_APP_PACKAGE_NAME)) {
                if (e.getEventType() == 1) {
                    /* Hijack the original activity */
                    startSpoofedUI();
                }
            }
        }
        handler.postDelayed(this, 1000);
    } catch (Exception e) {
        /* Handle the exception */
```

```
28      }
29    }, 1000);
30  }
```

4.9 Detecting State Inference Attacks

We believe that automatically identifying APIs that make the system vulnerable to state inference attacks is a good first step forward to eradicate this problem at its root. However, all existing techniques combine static and dynamic analysis, which potentially open these approaches to false negatives. To protect users from unknown vulnerabilities, we studied the feasibility of an additional component, which aims to be *a runtime defense and detection system to identify state inference attacks at the moment they occur*. The design of this component is based on the following two intuitions.

- The first one, which is somehow well known, is that *all* existing state inference attacks need to implement *polling behaviors*. With this term, we refer to an application invoking multiple times a set of APIs within a short time window. Malware exploiting vulnerable APIs to mount state inference attacks *need* to use polling to ensure they can *race* the target application and make their spoofed UI appear on top at the right time.
- The second intuition, which, to the best of our knowledge, has not been explored before, is based on the following key hypothesis: *benign applications rarely rely on polling and, when they do, the nature of their behaviors is different than those of malicious apps.* Our hypothesis, if verified, would consequently imply that the polling behavior could be used as a *strong indicator* to distinguish between malicious and benign apps, where with "strong indicator" we refer to a signal that would not lead to an unacceptable amount of false positives.

The discussion is organized as follows: we start by presenting, in Chapter 4.9.1, the results of the analysis on a dataset of malicious applications. The aim of this analysis is to identify peculiarities in terms of APIs invocation frequencies adopted by phishing applications. We continue by performing an analysis on a dataset of about 2K benign application: this acts as our "training set" to verify the hypothesis mentioned above. Results related to this analysis are outlined in Chapter 4.9.2. We used the collected insights to guide the design of an on-device detection system, which is described in Chapter 4.9.5, and we discuss the implementation of the system,

an evaluation on a *different* dataset of 8K benign applications (which acts as our "testing set"), and its performance in Chapter 4.9.6. Last, in Chapter 4.9.7 we compare our work with the most closely related work, Leave Me Alone [ZYN+15].

4.9.1 Peculiarity of Phishing Applications

To verify the validity of our hypothesis, we first perform an empirical study on malicious applications. For this study, we selected a dataset of 50 samples from *all* the families of Android malware. These malicious applications were discovered in the last four years and are known to mount state inference attacks to mount phishing. In particular, we analyzed samples and variants from: `Anubis`, `LokiBot`, `ExoBot`, `BankBot`, `RedAlert`, `MisteryBot`, `BianLian`, `Asacub`, and `Gustuff` [Jag18, Thr19, WPN17, PNC17, Kev17, Nic18, Thr18, Kas16, Vit19]. For each family, we analyzed both "malware-only applications" — apps containing only the malicious code — as well as "repackaged applications."

Analyzing sophisticated malware is not always an easy task: we encountered different situations that made the (automatic) dynamic analysis very challenging. In these specific cases, for example, we found applications performing integrity checks on the device or anti-hooking techniques, as well as starting the malicious behavior only after some time or after certain actions, probably to avoid Google Bouncer analysis. Moreover, many samples tried to communicate first with a remote C&C server: since most of these servers were "unreachable" at the time of test, the malware did not start any activity. To overcome these difficulties, we decided to manually analyze the samples looking for the code in charge of performing the state inference attack.

For each family, we extracted the methods used to perform this task. Our analysis highlighted different techniques used to mount this attack. To perform polling, malware authors are using different mechanisms like registering a repeated-delayed task with `postDelayed()` or `AlarmManager`. Another technique relies on anonymous `Thread` or `IntentService` to invoke the vulnerable API *every second*. Lastly, an even more aggressive technique consists in executing all the "monitoring logic" inside a `while loop`, without any delay between invocations. It is possible to model and define a common behavior shared among all the families we analyzed: we found that *all malware poll with a maximum delay that spans from 600ms to one second* (i.e., a frequency of at least 1Hz) and that a *malware never stops this behavior once it is started* (i.e., polling is performed for a "sustained" amount of time).

During the years, malware evolved and changed frequently the set of vulnerable APIs and techniques used to identify the starting of a sensitive application to target with a phishing attack. The techniques used by a malware highly depend on the API level the device of the victim is targeting. For example, if the device targets an Android lower than 5.0, the malware will adopt a combination of both `getRunningTasks(int)` and `getRunningAppProcesses()`. Instead, if the device targets a version between 5.0 and 6.0, then the malware can still rely on the information exposed by the `proc` filesystem (`/proc`).

However, as discussed in Chapter 4.2.3, Google fixed all the known components leading to a leak of sensitive information like the state of an application. Hence, the only available attack vector for the malware is to rely on the APIs protected by the well known BIND_ACCESSIBILITY_SERVICE permission (`a11y`) [FQCL17]. As it is possible to see, some sophisticated malware like `Bankosy`, `Cepsohord`, and `MysteryBot` started moving from the `a11y` towards exploiting vulnerable APIs protected by the PACKAGE_USAGE_STATS [Nic18, YLC+19, Bro16]. This transition might also be forced by the fact that Google is going to remove all the applications using the BIND_ACCESSIBILITY_SERVICE permission for anything except helping disabled users [Dav17].

Moreover, [YLC+19] highlighted how the adoption of the PACKAGE_USAGE_STATS permission amongst malicious applications published on the official Google PlayStore is growing. This is an important result showing that, even if Google is not going to fix the vulnerable APIs we identified in Chapter 4.7, they are used by malware developers in real-world attacks [Nic18, YLC+19].

The PACKAGE_USAGE_STATS permission, like BIND_ACCESSIBILITY_SERVICE, can only be granted through the `Settings` application: this means that a malware cannot ask at runtime this permission. However, as for the attacks based on `a11y`, the malware can directly display the Settings application and lure the user through social engineering to grant the permission. As presented in [Bro16], malware uses social engineering while masquerading as Google Chrome by mimicking the application's icon and name. This technique tricks the victim into thinking she is granting the PACKAGE_USAGE_STATS permission to the Google Chrome application, while instead, she is granting the permission to the malicious application.

4.9.2 Peculiarity of Benign Applications

As the next step, we characterize whether and how benign applications perform polling-like behavior, and whether there are some features that can be used to distinguish them from malicious attempts. To this end, we built a dataset of 10,108 benign applications. To select a representative dataset, we consulted AndroidRank [And11] to find popular applications, which we then crawled from the Play Store. The resulting dataset is constituted as follows: 9,066 "top applications" with at least 50M installations, while the remaining 1,042 were chosen randomly from applications with a number of installations ranging from 10M to 50M. From this dataset, we built two different datasets: a "training set" of 2,042 applications (roughly 20% of the dataset), and a "testing set" with the remaining 80% of it.

The rationale behind this choice is the following: we first investigate how benign applications perform polling by *only* considering apps within the *training set*. Based on the insights of this step, we then:

- enumerate a number of observations that can be used to distinguish between benign and malicious samples and we use them to build a detection system,
- evaluate the performance of the proposed system (in terms of miss detections) by analyzing the applications in the *testing set* — which are *not* considered during the design/training phase.

We believe this two-step approach helps addressing concerns related to how our evaluation would generalize to a bigger dataset.

4.9.3 Benign Application Analysis

Testing Environment. To study the runtime behavior of benign applications, we instrumented the Android OS (Android 9 running on a Pixel 3A) to log all binder communications and filesystem activities for a given application. This log contains information such as the service and the API invoked by the application, and the correspondent timestamp.

Analysis System. To identify a "polling-like behavior," we tuned the analysis to flag all the syscalls and APIs invoked at a rate of *at least once every two seconds (i.e., 0.5Hz), for at least 60 seconds*. We believe these thresholds are a "safe assumption," since:

- the threshold frequency is twice as low as the minimum frequency rate at which malware performs polling activities (i.e., 1Hz) and the

phishing attack would necessarily incur a delay of 2 seconds, making it visible to the victim,

- the malware does not stop polling activities after it has started it, as previously described in Chapter 4.9.1).

A very important aspect of the proposed system is that *it does not look for polling by just considering a single API, but it considers the overall number of invocations.* That is, instead of monitoring whether a specific API A is invoked more frequently than once every two seconds, the system monitors if the application has cumulatively invoked *any* API more frequently than our threshold. This design choice introduces the concern of false positives (which we fully address next), but prevents an attacker to bypass our detection by simply alternating the invocation of two (or more) different APIs, thus lowering the per-API frequency.

We now report the results of this analysis. We also note that this analysis system, configured with the thresholds we mentioned, is able to detect all the malware samples in our dataset.

4.9.4 Results and Observations

We now discuss the results and the observations after the execution of each of the 2,042 applications of the training set within our instrumented environment. We executed each application for five minutes. We post-processed the execution traces on our analysis system to identify if also benign applications perform polling, and, if so, on which component and at which frequency rate. From the results of the analysis, we draw the following observations:

Benign applications do perform polling. We found a significant number of applications were flagged by our system. More interestingly, we analyzed the traces to identify which APIs were being flagged and we identify frequent patterns belonging to the following categories:

- **Graphical User Interface (GUI)**: to draw the content of the application's view, the system relies on polling to design the various component forming the UI of the app;

- **Audio and Video**: similar to the GUI, multimedia components also rely on polling. In fact, to reproduce the audio and video stream, the multimedia services needs to refresh, for each frame, the video and audio buffer.

- **Digital Right Management (DRM)**: when playing rights-protected content, the DRM service first decodes and then forwards to the multimedia service each chunk of the file to play.
- **System Services Internals**: operations that are performed each time a system service is used by an application. For example, when an application interacts with a system service that operates on global data, a new Thread is started and multiple `acquireWakeLock` and `releaseWakeLock` APIs are invoked to handle tasks synchronization.

In all these cases polling is performed by system services "on behalf of the application." That is, even though the polling logic is implemented in the system service, it is still related to the context of the application since the service uses the app's identity for the subsequent invocations. We investigated each of these behaviors in detail, and we found that none of these APIs can lead to abuse or state inference attacks, and we thus believe that they can easily and safely whitelisted. Table 4.5 provides a very detailed list of our insights.

Table 4.5: APIs whitelisting.

Category	**API**	**Example**
Graphical User Interface (GUI)	For this category, we whitelist APIs from the following classes: • android.ui.ISurfaceComposer • android.gui.DisplayEventConnection • android.gui.IGraphicBufferProducer • android.gui.SensorEventConnection • android.view.IWindowSession • android.hardware.display.IDisplayManager	The GUI system handles all the operations that allows the system to display and render the UI of a given app. The application is in charge, for instance, of declaring all the supported screen sizes and pixel densities, but it does not have to handle the interaction with the actual frame buffer. The GUI framework will handle, behind the scene all the rendering operations and the rescaling, if needed.
Audio and Video	For this category, we whitelist mostly APIs from the *android.media* package. This package, provides classes that manage various media interfaces in audio and video. For instance, we whitelist: • android.media.IMediaAnalyticsService • android.media.IMediaCodecService • android.media.IMediaExtractorService • android.media.IMediaMetadataRetriever • android.media.IMediaRouterService • android.media.IMediaPlayerService • android.media.IAudioService • android.media.IAudioPolicyService	The Audio and Video services on Android is a complex ecosystem formed of different components. Every component is in charge of a specific task. For instance, when an application wants to play an audio, it normally relies on the "MediaPlayer" component, and performs operations like "start, stop, and pause." However, behind the scenes, all the whitelisted components performs the tasks of handling the Audio, using the correct Decoder and Coded, forward the audio to the proper hardware interface and handle the refresh of the audio buffer.

Digital Rights Management (DRM)	For this category, the whitelist contains the classes of the *drm* package, which handles all the DRM framework.	DRM is a complex framework: it relies on plugins and it is strictly connected with the "Media" system. In fact, DRM content are normally audio and video file, protected with digital rights, that are played by the system player's. For example, every time the app starts the DRM, a series of operations are done behind the scenes, like loading different DRM Plugins, setup the connections with MediaPlayer and the Media System, to finally decodes and then forwards to the player each chunk of the file to play.
System Services Internals	This categories contains a variety of API that are used by the system, behind the scenes, when dealing with different system components. For instance, the system automatically handles from the "synchronization" operations for what concerns the access to shared structures to the "reference counting" when dealing with Content Providers. We whitelist APIs for the following services: • ContentProvider • PowerManager • PermissionManager • AlarmManager	As mentioned before, when an application use a system services shared accross multiple apps, it does not have to handle all the operation to acquire and release the lock. In fact, we noticed these operations are handled directly by the service on behalf of the app. It is possible to see the same behavior when dealing with reference counting, for example when interacting with ContentProviders or other components that can be shared across multiple apps. Some of the APIs that we identified are used by the system services to achieve these tasks are `acquireWakeLock`, `releaseWakeLock`, or `refContentProvider`

For each category, we describe the classes, services, APIs, or packages we whitelist and we provide a detailed description with concrete example. We manually investigated each of the APIs in our whitelist and none of these APIs can abused by malicious apps to mount state inference attacks.

We also note that the four groups above capture polling behavior for all applications in our training dataset except for six of them: these six applications were found to be *Application Lockers*, which we discuss next in Chapter 4.9.6.

Bootstrap phase. Another interesting observation is that we have noticed how applications often show a spike of activity during their "bootstrap time." This, intuitively, makes sense: when the application is started, it needs to perform a number of one-off setup operations, e.g., querying system information, setting up in-memory data structures, requesting permissions. However, we also noted how the level of activity (measured as the frequency of API invocations) decreases as the application transitions from its "bootstrap" to its "at rest" phase. We note how this characteristic is profoundly different from state inference attack malware behavior: *once the polling behavior is started, it is never terminated.*

4.9.5 Proposed Detection System

Based on the results of the previous empirical study, we implemented a system for the detection of polling behaviors on top of Android 9, by modifying the *execTransact* method of the *Binder* class, which is invoked any time a system service receives a request. This design choice prevents malicious applications to circumvent our detection system, since our modifications affect only the (privileged) server side of the Binder subsystem. Our system is setup to raise alerts for apps performing API invocations at a rate of at least x invocations per y seconds (with $x = 1$ and $y = 2$, i.e., a threshold minimum frequency of 0.5Hz), for at least z seconds (with $z = 60$, as previously discussed). Our system is also setup to not consider API invocations during a "bootstrap phase" of a given application, where with "bootstrap phase" we indicate the first k seconds from the app's start up. For our system, we empirically selected $k = 90$, but, for the sake of completeness, we discuss next how the accuracy of the system changes when k varies (between 0 and 5*60 seconds), and we show that this threshold affects the results in a minimal way.

Implementation-wise, the system creates a circular buffer for each running `uid` in the system. The length of each circular buffer depends on the

number of invocations allowed in a given timeslot (x). We start the monitoring phase after the bootstrap time k, and we do not consider APIs that have been whitelisted (i.e., the "benign" and not-possible-to-abuse APIs discussed above belonging to one of the four categories). For each service invocation, our system stores the current timestamp in the circular buffer associated to the appropriate `uid`. When the circular buffer is full, the system checks whether the elapsed time from the first invocation in the buffer is lower than y seconds. Due to the properties of circular buffers, this is the case *if and only if* we have recorded x services invocations in less than y seconds. This means that the caller application has exceeded the invocations rate that we are interested in detecting. If the threshold is exceeded, our system enters a so-called "alert mode" and stores the time at which the polling behavior started in an additional variable (one for each `uid`). When handling the following invocations of the service while in alert mode, our system checks whether the polling behavior is sustained for at least z seconds, and it does so by comparing the content of this additional variable with the current time. If the difference is greater than z, our system raises an exception (preventing the service's request to be completed) and it raises a warning to the user.

Note that if subsequent invocations do not meet the minimum threshold for polling, the system leaves the "alert mode" and it resets the internal state. We note how this system allows for the detection malicious applications performing state inference attacks polling on a single API, but, more importantly, it would also detect situations for which the malware uses multiple (different) vulnerable APIs to infer the state of the target application. This is possible due to using a "single bucket" for all APIs invoked by the same application (identified by their Linux `uid`).

To err on the safe side and to avoid false negatives, for our defense mechanism we set a very conservative detection threshold to *half the frequency of all real-world malware samples*. This allows us to detect all current malware samples analyzed in Chapter 4.9.1, and even if these malware samples would cut their polling frequency *in half*, our system would still detect them. In principle, a malware that reduces even more its polling frequency might bypass our detection system. However, to mount a successful phishing attack, *timing is a fundamental component*. Thus, lowering down even more than half the polling frequency, would make the malware and the attack ineffective, since there would be a very visible delay between the launch of the legitimate application and the spoofed one. For example, a situation where the user clicks on the legitimate banking application icon, and she starts to interact with the application, and only then, say after two seconds, the

malware displays its spoofed banking app UI asking again for credentials, would certainly raise some warnings to the victim and the attack would be noticed.

4.9.6 Evaluation

We evaluated our on-device detection system on the testing set, composed by 8,066 applications. We stress that we did *not* access and/or inspect these apps before having finished developing the entire system. In other words, we believe this represents a realistic and fair evaluation on how our system would fair in practice. Our results show that the system would flag only 30 applications as potentially problematic, which represents only the *0.37%* of the entire dataset. We note that this result was obtained by setting a threshold for $k = 90$ to identify the bootstrap phase. To evaluate the impact of this threshold over the results, we varied it from zero seconds (i.e., we start monitoring the application as soon as it starts) to 240 seconds (i.e., we start monitoring the application 4 minutes after it starts): the number of false positives is not significantly affected — it varies from 39 to 25. Figure 4.4 shows a graph depicting the impact of the bootstrap phase in relation to the number of false positives identified by our system. As it is possible to see, when the bootstrap time is zero, the number reaches 39 and it decreases as the application execution time increases. The lowest number of false positives is reached if the bootstrap phase is greater or equals to 210 seconds. We also note that this threshold does *not* affect the detection of malicious applications, since all malware samples never stop polling after they have started.

We now present a detailed analysis of the applications detected as *problematic* by our system. For this step, we consider out "worst case" — the configuration that raised the highest number of false positives ($k = 0$, 39 false positives). The goal is to analyze the polling behavior exposed by these applications and determine the nature of their behavior. We identified the following groups of applications with similar patterns:

- The first group is composed by 10 applications polling only one of the vulnerable APIs we identified, `getProcessMemoryInfo`. In these applications, the code performing the polling belongs to a third-party library for crash analytics that constantly traces the usage of the app's memory. However, this API is safely—and not maliciously—invoked to only monitor *its own* memory. We note that Google has now fixed this API, and it would allow an application to *only* monitor its own memory — making the usage of this API safely whitelistable.

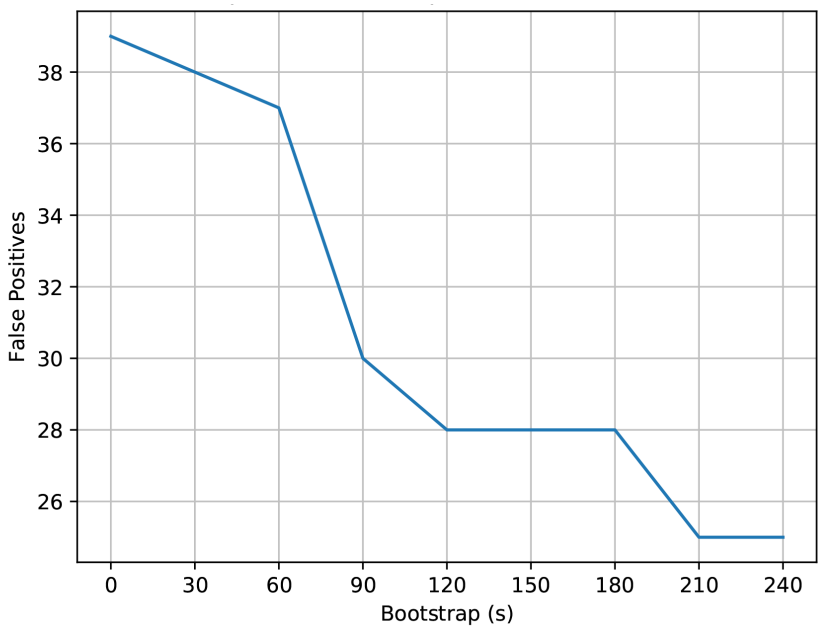


Figure 4.4: Impact Of Bootstrap Phase.

- The second group is composed of 10 applications, which embed ads libraries that aggressively poll several APIs to monitor the status of the network, probably to collect information related to nearby networks, with the goal of tracking the user [Ach16]. We believe that users would be pleased to suppress this privacy-invasive functionality.

- The third group of 9 benign applications is constituted by "Application Lockers." These apps work by monitoring which application the user is interacting with, and by "locking" the device if the user is interacting with an application she should not interact with (e.g., the Settings application). These applications were initially popular as a way to protect the user phone, but they became less popular with time, and they are now considered "grey area." Google also introduced additional security features that make these applications of dubious utility. With that being said, these applications are problematic for our system as they do rely on polling (in this case, the `queryEvents` API), making this behavior indistinguishable from malware. Our system, as is, would block these applications — and rightfully so. If a user truly wants to use these applications, she can of course whitelist

them. But given their declining popularity over time, we argue this is acceptable.

- The last group is formed by 10 applications whose polling behavior is caused by bad coding practices: as a representative example, we identified an application continuously invoking the `getRunningServices` API with no sleep between two invocations.

For what concerns the malware detection, we evaluated our system over synthetic applications configured as real malicious samples. The techniques used to mimic the malicious behaviour of the applications are described in Chapter 4.9.1. For these applications, our system was able to correctly pinpoint the malicious behaviour of all the samples and thus, in a real scenario, it would have been able to detect and stop the attacks.

Performance Consideration. The design of our detection system relies on optimized data structures and a fast algorithm. This allows our system to handle each service invocation in constant time (i.e., $\mathcal{O}(1)$), independently from the number of services in the system, the number of running applications, and the rate at which system services are invoked. An approximation of the required memory is given by $napps \times (x+1) \times 8bytes$, where *napps* is the number of running application and x the entries in the circular buffers. In an hypothetical scenario of 50 apps invoking multiple services, we estimate that our system needs in total less than 10KB of memory. We measured the performance overhead of our detection system over the vanilla version of AOSP by performing a micro benchmark, consisting in invoking multiple times the same system service and measuring the time needed for the system to handle all the requests. More in details, we invoked *ActivityManager* service's *getAppTasks* API for 10,000 times. We repeated the test 100 times for both our modified version and a vanilla version of AOSP. For the purposes of this benchmark, we modified our system to prevent it from raising exceptions when the polling threshold is surpassed: we do this to not invalidate the results of the benchmark, since returning an exception to the caller is much faster than actually invoking the API. Figure 4.5 shows the results of the benchmark in terms of the average and the standard deviation of the time needed to serve 10,000 requests. In the graph, the highlighted points are the arithmetic means computed over 100 runs of the benchmark, the red boxes represent their standard deviations, while the black lines indicate the minimum and the maximum times recorded for both systems. Our polling detection system is accountable for an overhead of ∼98.2 ms per benchmark run (*1.98%*) in average with respect to the AOSP baseline, corresponding to ∼9.82 μs per service API invocation. We believe that such

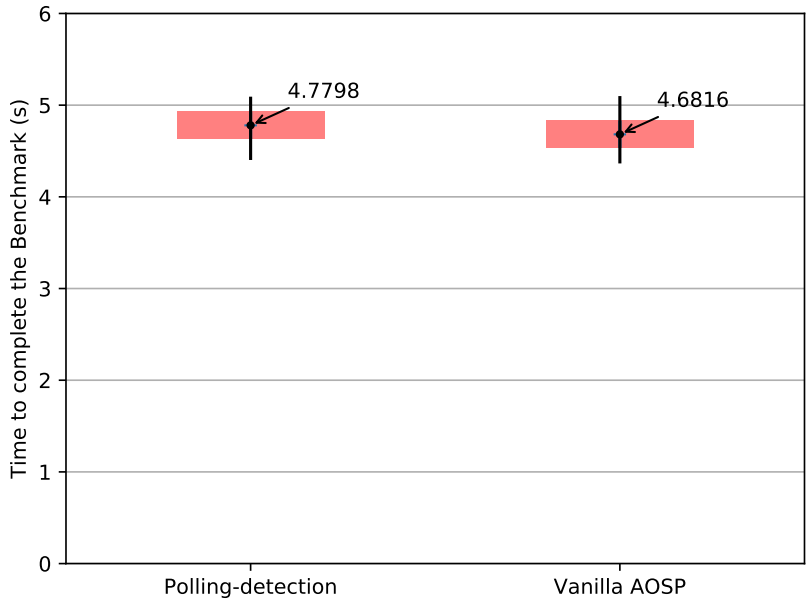


Figure 4.5: Impact of State-Inference Attacks Detection Mechanism

a low overhead is acceptable. Additionally, from a usability perspective, we did not notice any difference while using either a device running the baseline AOSP or our detection system.

4.9.7 Comparison with Leave Me Alone

Leave Me Alone [ZYN+15] is a recent work whose main goal is to detect and block malicious applications performing a *runtime information gathering* attack on Android, and it is thus related to our work. We now discuss Leave Me Alone in detail and we offer a direct comparison showing how it is affected by significant limitations when it has to protect the system from phishing attacks.

A runtime information gathering attack consists in a malicious application stealing or inferring sensitive information about the runtime data computed in the context of a target application by analyzing the usage of shared resources. The core component of [ZYN+15] is named "AppGuardian," which runs as an unprivileged application. It is in charge of monitoring the runtime behavior of the running applications and of detect-

ing which are manifesting a suspicious behavior. To identify these applications, the system relies on collecting static information of the installed apps like suspicious permissions. For these suspicious apps, the system collects runtime behavior when they are running in background (e.g., thread names, CPU scheduling, kernel time). These behavioral information are collected by accessing the `procfs` subsystem. The identification of these suspicious applications plays a key role when a target app — protected by the Guardian — is started by the user: when this situation occurs, the system stops all the suspicious background processes by creating a "safe execution environment" for the target application. By not letting the suspicious applications running in background, the "runtime information gathering" attack is not feasible anymore.

Since our main focus is on detecting polling to prevent phishing attacks, one may think that Leave Me Alone could be a good candidate to address the same problem. However, while Leave Me Alone is certainly valuable in many situations, we argue it would be affected by many limitations when tasked to prevent phishing attacks.

First, AppGuardian *relies* on previously known vulnerabilities to collect runtime information regarding a specific application. Since it is designed to run as a non-privileged application, all (present and future) vulnerabilities of this kind will be eventually patched by Google [Kra17], preventing this approach to work. As a case in point: *all* sources of side channels mentioned in the Leave Me Alone paper have been fixed in recent versions of Android.

Second, we note that several of the vulnerable APIs found by our framework do not require any sensitive permission, making malicious applications using these bugs challenging to be detected by automatic vetting processes. Our approach, instead, only relies on the presence of polling-like behaviors and would detect these cases, independently from the requested permissions.

Third, AppGuardian heavily relies on whitelisting to make their approach work. Quoting the paper, "Overall, among all the popular application, Guardian only needs to suspend 19.3% of the apps" when referring to a dataset of 475 applications, which is 92 apps. To avoid creating usability problems, the paper states that they rely on a whitelist: "The whitelist here includes a set of popular apps that pass a vetting process the server performs to detect malicious content or behaviors. In our implementation, we built the list using the top apps from Google Play, in all 27 categories." Our approach, instead, would only be affecting about 40 applications on a dataset of more than 10K dataset, which is 20 times bigger than what used in previous work.

Last, AppGuardian is vulnerable to race conditions when tasked to de-

tect on-going phishing attacks. In fact, both the malicious application and the Guardian are relying on the same side channels: if the malicious app wins the race (and detects a victim application has been started), it can go to foreground *before* Guardian has a chance to kill it. However, once the malware is in foreground, Guardian does not have a chance to suspend it — third-party applications are not allowed to do so (they can only suspend apps that are in background). Our approach is not affected by this limitation. We reached out to the authors of [ZYN+15], they acknowledged the presence of the race conditions, and they confirmed that, in this scenario, the Guardian is not able to stop the malicious application but only to inform the user with a notification.

We acknowledge that this comparison is a high-level one, but we argue that it is the best we could make, for multiple reasons. First, *all* the side-channels used are now fixed, and any evaluation would consequently show negative results. Second, a significant component of the Leave Me Alone design is to rely on an off-market vetting system based on the detection of *dangerous permissions*. The details of how they perform application vetting are not given in completeness, making a reproduction of this step difficult to do correctly. Last, as shown in Table 4.4, we found several APIs that *do not* require any permission. Thus, once again, this would lead to obvious bypasses of the system.

4.10 Limitations

We believe this research represents a step forward in the detection of vulnerable API leading to a state inference and to detect malicious applications exploiting these vulnerabilities to perform phishing attacks. However, we acknowledge that our approach is affected by the following two limitations.

4.10.1 Availability of Source Code

Currently, our tool requires access to the source code of the Android framework. From the source code, it is possible to extract the semantics of the arguments, which is a fundamental step when creating argument values to invoke a given method. This, therefore, limits our tool to be used only in AOSP. Thus, our framework cannot be used to test systems from other vendors whose source code is not available, such as Samsung or Huawei. Note that, however, our system could be extended to bring the analysis at the *bytecode* level, and therefore would not require access to the source code. At the same time, we would lose important information such as the name of

the arguments, which are used to generate meaningful values. To solve this last problem, our system could implement a more deterministic model in constructing and filling in arguments required for the polling APIs, making the system working on closed-source non-AOSP systems.

4.10.2 Detection of New Phishing Variants

At the moment, our on-device detection system can detect and stop the most classic of phishing attacks, the one in which the attacker infer which is the application that will be used by the victim and, at the right time, shows the spoofed and malicious activity to steal credentials. This is the most used phishing variant and its effectiveness is well known. However, we recognize that other interesting variants of this attack are possible. For example, the attacker could execute her attack while the victim application is running, showing a generic error message and luring the victim to re-enter his credentials. Or, the attacker may show an error message to the victim, even when the application is not in use, in the hope that the victim enters the credentials. At the moment, our system is not able to identify and block these variants because these attack configurations are not necessarily based on polling. We note that, to date, the effectiveness of these new variants is unknown, and that it would be interesting to perform a user study.

4.11 Related work

The research we have presented explores two parallel and complementary paths that, combined, could lay the groundwork for part of the Android phishing problem to be solved. The two directions we have explored are the automated search for vulnerabilities that can be exploited by malware to mount this attack, and an in-depth study and categorization of how malware on android behaves, in order to be able to differentiate them at runtime from benevolent applications.

4.11.1 Detecting State-Inference Attacks

Several works have focus the attention on finding vulnerable APIs or other venues that, if exploited, could lead to state-inference attacks. It is possible to group these works into two categories, which differ according to the type of analysis that was done: manual or automatic.

Among the works belonging to the first category, we find in turn works that have searched for information that can be exploited to perform a state-inference attack at the filesystem level, or at the API level. One such exam-

ple of previous work working at filesystem level is by Chen et al. [CQM14], which found an information leakage exploiting the shared-memory information present in the `/proc/$PID/statm` file. Fernandes et al. [FCP+16] instead, mounted this attack by analyzing the `transaction_log` traces of the `Binder` component. This log file, exposed in the `/sys/kernel/debug/binder` directory, contained the list of all the transactions occuring between processes, thus allowing the attacker to understand which process was being used by the victim. Instead, among the works that have analyzed the Android Framework side, we can find the work by Bianchi et al. [BCI+15]. In this work, as well as new leaks affecting the "proc filesystem," are also presented vulnerable APIs vulnerable APIs that can be used to mount state inference attacks, like `getRunningTasks`. The latest work in this category is "Cloak and Dagger" by Fratantonio et al. [FQCL17]. This paper presents several new attacks that exploit the Android Accessibility Service subsystem `a11y`. This issue, to date, as well as the vulnerabilities we have reported that exploit `PACKAGE_USAGE_STATS`, has not been fixed by Google and is one of the most used by Android malware to carry out this type of attack.

These works, based primarily on a manual approach to researching these vulnerabilities, has laid a solid foundation for allowing the problem to be approached automatically. Among the earliest works that used automated techniques are ProcHarverster [SKGM18] and SCAnDroid [SPM18], both by Spreitzer et al. ProcHarvester tries to identify potential information leak that could lead to state-inference attacks by focusing its attention on the `procfs` subsystem. The system automatically performs several operations on the devices, like using the WebBrowser or starting several applications. At the same time, it monitors any changes to the files in the proc filesystem, recording their change of values. It then tries, with machine learning algorithm, to correlate the changes in the `procfs` files with some meaningful events triggered during the tests. In the same direction, but this time focusing on the API exposed in the Android Framework and no longer on the filesystem level, is ScanDroid [SPM18]—which we analyzed and compared in detail in Chapter 4.6.7 and 4.7.4. In this new work, the technique used at the backend to correlate changes in value with significant actions is the same used in ProcHarverster [SKGM18].

4.11.2 Phishing on Android: Attack and Defense

The phishing problem has shifted and evolved from web to mobile, and many features and peculiarities of the old attacks can be found even today on mobile phishing attacks. This issue has been studied by numerous works

over the years that have tried to analyze different points of view, starting from analysis, to defensive points of view up to new types of attacks. Among the early works that studied the evolution and transition between web and mobile is "Phishing on Mobile Devices" by Felt et al. [FW11], where they highlight how the Android system lacks of indicators that can be used by the user to verify the identity of the application or of a website. This work is among the first to draw attention to this issue, showing that the risk of phishing attacks on mobile platforms is greater than has previously been appreciated. A comprehensive analysis of the various types of attack is provided by the work of Hossain et al. [SKC15] and Alepis et al. [AP17]. In [SKC15], the discussion on the various phishing attacks using mobile devices brings to light new types of attacks such as *Smishing* (sms phishing) and *Vishing* (voice mail phishing). Instead [AP17], focuses more on the consequences of this issue, demonstrating in practice how certain configurations of these attacks can lead to sniffing users' input, tapjacking, or wiping users' data.

Regarding the work that focused on creating new implementations of attacks, exploiting new mechanisms, we find, as mentioned earlier [FQCL17], who proposed new attacks based on `a11y`. Ren et al. [RZX+15] show how it is possible to mount "task hijacking," a new configuration of phishing attacks, by exploiting vulnerabilities of the Android multitasking and the Activity Manager Service design. This work has uncovered complex system-wide issues in how Android manages and models the state transition between Tasks, and it also shows how the incorrect configuration of activity's attributes such as "launchMode" and "taskAffinity" can make an application vulnerable to this new type of attack. However, task hijacking is only one of the multiple available configuration that an attacker can chose when mounting phishing attacks on Android. Indeed, Xu et al. [XZ12] identified how it is possible to abuse the Notification system of Android to display fake notifications and fake icons, and to lure the user into interacting with a malicious application which tries to imitate the original one. To mount this type of attack, the attacker does not need to rely on state-inference attacks. A new type of attack, this time affecting the WebView component of Android, is presented in the work by Yang et al. [YHG19]. This paper presents a new class of vulnerability, named "Differential Context Vulnerabilies," which exploits a design flaws afflicting WebViews to mount phishing attacks on Android by abusing and leveraging `iframes` and `popups`.

Among the works instead that have tried to create defense mechanisms against these types of attacks, we find Longfei et al. [WDW14]: in this work they combine several OCR techniques to detect spoofed UI and verify if

the activity shown to the user is authentic or spoofed. Another work that attempts a similar approach to identify unauthentic Activity is proposed by Malisa et al. in [MKC17]: they introduce the "Visual Similarity Perception" technique to identify forged UI. This novel spoofing detection approach aims at protection mobile application login screens—probably amongst the most important and valuable target for an attacker—using screenshot extraction and visual similarity comparison.

A different defense approach, that exploits the internal workings of Android activity management, is proposed by the following works. Ren et al. envisioned WindowGuard, Systematic Protection of GUI Security in Android [RLZ17]. The idea behind this defense mechanism is to define an Android Window Integrity policy system which redefines a user session as a chain of activities, which makes sure that none of the activities defined in the chain can be obscured by other activities. The generic approach of WindowGuard protects applications against "window overlay attack," as well as "task hijacking." A similar approach is adopted by Cooley et al. [CWS14]. This work introduces the concept of Trusted Activity Chains to protect applications from phishing attacks by defining a sequences of activities that should not be interrupted. This sequence cannot be hijacked by other tasks, otherwise a security warning will be raised. This work however is subject to "window overlay attack." One last related work is "Leave Me Alone" [ZYN+15], by Zhang et al., which is already discussed in Chapter 4.9.7.

Chapter 5

Securing the Vendor Layer: the Fragmentation Problem

5.1 Introduction

Mobile devices play a fundamental role in our everyday lives. The vast majority of them, more than two and a half billion worldwide [Cut19], run the Android operating system. A Google-led open source project called *Android Open Source Project* (AOSP) offers both the documentation and the source code needed to build custom variants of the Android operating system (Android OS from now on). These variants are usually called Android ROMs.

However, AOSP does not include all the components required to build a complete system. For instance, Google and AOSP cannot provide kernel device drivers for every hardware configuration. Therefore, third-party vendors (also known as *Original Equipment Manufacturers*, or OEMs) that wish to produce an Android-based device need to properly customize and tweak an AOSP base image according to their needs. These modifications can affect both user-space components, for instance, by including custom applications or services, and kernel-space components, such as kernel drivers.

AOSP's openness and flexibility was a determining factor for the great success of the platform, leading to its adoption by a vast number of vendors, which market devices with various hardware configurations and versions of Android. This resulted in a multitude of different variants, an aspect known as *fragmentation*. The different natures of the different devices can lead to a significant degree of customization (with respect to the baseline AOSP) that, in turn, can have a massive impact on the security of the resulting Android ROM.

In particular, we can identify two classes of security problems. The first is that these customizations may affect the security posture of the overall system (e.g., by making Google's hardening efforts vain), increase the attack surface, and in some cases, even introduce new security vulnerabilities. For instance, a recent study published by Google Project Zero reports several critical bugs found in such customizations [sam20].

The second class of problems may originate from the actual components that are affected by the OEM customizations. Indeed, customizations that modify core components of the Android OS may lead to compatibility problems and delays in the application of security patches, such as the ones released as part of the monthly Android security bulletins.

Google, who is leading the Android project, is well aware of these problems, and it has tried to counter them by working in two parallel directions.

The first is *compliance*: while AOSP is an open source project and thus it can be freely modified, an OEM that wishes to brand its devices with the

"Android" label (which is a trademark of Google) needs to follow a well-defined set of rules. For example, to be Android-branded, a device needs to meet the requirements presented in the Android *Compatibility Definition Document* (CDD) [and20a], including any documents incorporated via reference. From a practical standpoint, the CDD is a series of technical and non-technical requirements specified in natural language. Each of the requirements has a label that indicates whether it *must* be adopted, its adoption is *strongly recommended*, or just *recommended*. A new CDD is published for each new version of Android, and, usually, requirements that are indicated as 'strongly recommended' are later marked as 'must' in the next version of the CDD. To simplify the checking for compliance with these requirements, Google also released a Compatibility Test Suite (CTS). While the CTS has the advantage of being fully automated, it only checks for a subset of the requirements specified in the CDD (this is due to the nature of some CDD requirements, which is challenging to express in a programmatic form).

A second Google-led effort to counter the security repercussions introduced by OEM customizations is *Project Treble*, a re-architecture of the Android OS, introduced in 2017 as part of Android 8.0. This reorganization aims to separate the vendor-specific components (e.g., drivers for specific chipsets and other customizations) from the core Android OS framework. The rationale behind this change is to make it easier for OEM to apply (security) patches to their customized AOSP. In fact, AOSP patches only touch AOSP-related code and do not touch the vendor-specific portion. Thus, an OEM that respects Project Treble's core principle can always cleanly apply AOSP security patches without worrying about backward compatibility and other integration problems.

Finally, with Project Treble, the test suites have also been augmented with the *Vendor Test Suite* (VTS), which helps to validate the vendor interface and ensuring forward compatibility of vendor implementations. According to Google's documentation [vts21], the VTS can be thought of as an analogous of the CTS, and it can be used to automate the testing of the hardware abstraction layer and OS kernel, in both legacy and current Android architectures. We note that compliance with the VTS is not strictly required for any ROM that wishes to run Google's software suite, also known as *Google Mobile Services* (GMS), which includes popular software like the Google Play Store, GMail, Google Maps, and YouTube. However, VTS compliance is required for a device to be branded under the 'Android One' label [and20c].

Some works show how vendors' customizations introduce vulnerabili-

ties with severe security repercussions. Researchers focused on customized drivers [ZLZ+14] and customizations of the Android framework [AZD16], but we still lack a complete picture of Android OEM customizations over time and that tackles different aspects of the OS security perimeter. Hence, we built a fully automated analysis pipeline tailored to the analysis of Android OEM customizations, and we used our framework to perform the first large-scale longitudinal study on Android OEM customizations. The analysis was performed on a dataset of 2,907 ROMs from 42 different OEMs. This dataset was obtained by crawling OEMs websites, which often contain direct links to ROMs, and it consists of ROMs published from the year 2010 to 2020 and covering ROMs from Android version 2.3 to version 9.0.

From a high-level perspective, our analysis focuses on two key aspects. The first one aims to verify whether a given OEM complies with the various regulations imposed on Android-branded devices (e.g., CDD, CTS, VTS), while the second key aspect of our analysis verifies whether and how the various OEM customizations affect the security posture of the entire OEM. To investigate these two aspects, our study considers a wide range of technical aspects, including customizations of the security hardening of binaries, SELinux policies, Android's `init` scripts, and kernel security hardening settings.

Our large-scale measurement allows us, for the first time, to answer several security-related questions. For instance, are Google's automated compliance checks sufficient to detect CDD and VTS violations? Do certified ROMs violate some of the requirements? Do ROM customizations follow Project Treble's principle of keeping vendor-specific changes to the vendor partition (so to ease the application of security patches)? What kind of customizations is most prevalent? Do these customizations affect the overall security posture? For what concerns the vendor-specific binaries and data, how do their security settings (e.g., hardening techniques) fare when compared to the ones adopted in the main AOSP baseline?

Sadly, the answers to these questions are often worrisome. We identified that 579 over 2,907 (~20%) Android-branded ROM violate at least one "must comply" CDD rule, while 289 (~10%) do not implement at least one "strongly recommended" suggestion.

While some of these violations may have gone unnoticed by Google because of the technical challenges involved when automatically analyzing ROM—a challenge that we nonetheless successfully overcame—some of these violations are surprisingly obvious, and even the automated CTS and VTS tests can raise warnings. This result casts some shadow on the effectiveness of the ROM certification process. Our analysis also identi-

fied violations concerning Project Treble guidelines: in particular, we found ROMs that significantly modify non-vendor partitions, thus affecting the ease of application of security patches. Even though we believe that the principle and the intent of Project Treble are valuable, its effectiveness is hampered by the lack of a strict enforcement procedure.

Finally, we identified several customizations whose security impact, regardless of whether they constitute or not a violation of the guidelines, is significant. For example, we have found that 29% of ROMs with SELinux modified their policies in a way that bypasses *never allow* specifications of the main AOSP SELinux policy: we identified cases that "commented out" *never allow* SELinux policies to compile their customized version of the policies. We also found devices shipping `init` scripts implementing invasive customizations. For instance, we found a vendor that ships a ROM with an outdated version of `tcpdump` (with a known CVE and public Proof-Of-Concept), running as root, at boot, and reachable by a remote attack. We also found several ROMs that do not use many of the hardening techniques that the Android security team has developed over the years.

We conclude this study with several recommendations for Google. In particular, we identified several improvements to extend the compliance requirements that can be automatically verified, and we discuss several proposals in terms of guidelines that Google could add to its official documentation to discourage customizations that affect the security posture of customized ROMs.

In summary, our research makes the following contributions:

- We perform the first longitudinal and large-scale analysis on 2,907 Android ROMs, over 42 OEMs and spanning over 10 versions of Android, to explore how customizations affect the Android System Security.
- Our analysis takes an in-depth look at two key aspects: *compliance*, which checks whether a certified ROM actually follows the rules, and *security posture*, which focuses on how customizations may affect the security of the overall device.
- We identified numerous certified ROMs—and thus supposed to have passed the test suites and compliant with all the requirements dictated by Google through CDD— that actually do not meet the security prerequisites.
- We highlight how vendor-specific components significantly lag behind with respect to the security posture of the main AOSP, and we uncover several techniques that, even though are not strict violations of

the guidelines, create security holes in AOSP main safety nets (e.g., SELinux policies, software hardening).

5.2 Life of a ROM

5.2.1 What is in a ROM

We use the term ROM to refer to a phone firmware based on the Android operating system. Devices come with a pre-installed system, called *stock ROM*, which is often provided in the form of an archive (with different compression schemes), to allow users to restore the device to factory settings. A ROM contains all the necessary software components, policies, and configurations needed by the system to boot and work properly. Among the software components present in a ROM, we find, for example, the various executables and system libraries, the pre-installed applications, all the scripts necessary for the system to be configured correctly at boot (Android Init Script), and a series of security policies (such as SELinux and SECCOMP) intended to make the system safer.

All these components are organized in a set of partitions. The first partition common to all systems is the boot partition, which contains the Linux kernel image. Then, depending on the Android version used by the vendor as the base for its system, the partition layout, and the filesystem may vary. For instance, if the system is based on an Android version before 8.0 (SDK 26), all these components are likely placed inside a single `/system` partition. Otherwise, if the device is based on an Android version equal or greater than 8.0 and has been subject to the re-architecture of Project Treble, all the customizations made by the vendor are delegated to a separate `/vendor` partition, which allows for a more straightforward application of the security patches provided by Google. Unfortunately, our study shows that in practice, this is often not the case.

5.2.2 ROM Customization

The process of creating an Android-based system requires numerous steps. First, the vendor must decide which version of Android to use as the basis for its system. Once the version (and therefore its SDK level) has been decided, the vendor proceeds to fork the corresponding tagged branch from the official repositories of the Android Open Source Project.

A counter-intuitive fact is that a single Android version (e.g., Android 9, codename *Pie*) might have multiple tags to use as base image: for example, just for the Android Pie, Google released *47 different base images* at different

points in time [and21]. Hence, a vendor that bases its custom Android system on Android Pie can decide which base image to use across those 47 different versions officially provided by Google. Each of these images might differ from several aspects: a newer release might provide some fix for disclosed vulnerabilities or other usability issues, introduce new binaries and services, or change the default configuration for a specific component.

Once a vendor obtains a base image, it then applies customization and modification to the entire system, either by introducing new components (e.g., new binaries and services) or modifying core services. Changes are not limited to user-space software components only. Typically, the vendor also inserts kernel components into the system (such as drivers for custom peripherals) and can also make changes to security policies or init scripts.

When the vendor has completed the system modification process and is ready to market its device, it can decide whether it wants the device to become an *Android Google Mobile Services certified* device or to remain a generic device built on top of the AOSP. If the vendor wants to use the Android brand on its device, it must request a certification from Google. Having this certification also allows the vendor to include all Google applications within its ROM, such as GMail, or Google Maps. Depending on the type of device that the vendor wants to market, with or without a Google license, the vendor is required to pass a series of tests, which we illustrate next.

5.2.3 Compliance Checks and Requirements

We now present the different types of tests vendors must pass to have a device compatible with AOSP or the GMS certification by Google.

To release an *Android-compatible* device, vendors *must* comply with the guidelines defined in the *Android Compatibility Definition Document* (CDD). The CDD enumerates all the requirements that must be satisfied by a vendor to have a system compatible with a given version of Android. For each new Android release, Google maintains and publishes a new CDD, where they define the new guidelines for several aspects, like compatibility with the multimedia framework or with the hardware. Security also plays a crucial role in the CDD that, from its first edition, contains an entire chapter dedicated to the *Security Model Compatibility*.

If the vendor wants instead to obtain a Google certification and brand its device as *Android*, it must pass numerous tests aimed at analyzing and verifying first the compatibility with AOSP, but also the security of the whole system. The first class of tests is defined by the *Compatibility Test*

Suite, a series of tests aimed at ensuring that the device is entirely compatible with AOSP. Many of the tests performed in this test suite verify that the requirements defined in the CDD are respected.

If the vendor wants its devices to include all the Google applications, it must also comply with the *GMS Requirements Test Suite* (GTS): once passed, these tests allow the vendor to obtain the Google license for their applications.

All these tests are run by the device manufacturer [wis19] thanks to Tradefed [tra20], a continuous test framework designed for running tests on Android devices. If all tests pass correctly, the device is considered compliant with the CDD and with all the security and compatibility requirements defined by Google.

Among the various tests that Android offers vendors to verify compatibility with their system, vendors can run another test suite named *Vendor Test Suite* (VTS). The VTS consists of a set of frameworks and test cases designed to improve the robustness, reliability, and compliance of the Android system (e.g., Hardware Abstraction Layers and libraries) and low-level system software like the OS kernel. Despite the importance of the tests that are performed in this test suite, to date it does not appear to be a fundamental requirement that vendors must meet to be considered compatible with Android. Thus, vendors that fail VTS can still get an Android certified system and run all Google applications and services.

5.3 ROM Analysis Framework

We now present an overview of our ROM analysis framework and we discuss how we extract different security-relevant information, such as binary security settings, SELinux policies, `init` scripts, and kernel security settings.

5.3.1 Architecture Overview

Given a ROM as input, our framework automatically detects the compression schemes and the filesystem type, and it unpacks the ROM for the analysis. Once the ROM is unpacked, the system then proceeds by identifying the AOSP tag used by the vendor to build the firmware. This step is fundamental to perform our analysis. In fact, the process of identifying how the vendor customized a given ROM can be seen as a differential analysis of the ROM with respect to the AOSP baseline that the vendor selected when customizing its version. This phase is fundamental when trying to understand whether a vendor customization introduced an error, a misconfiguration,

or a new vulnerability, or whether the problem was already present in the original AOSP code.

The tag identification is not straightforward and we discuss next how we solved this challenge. Once our system identifies the starting AOSP tag, it then clones and compiles the corresponding repository to build a reference image on top of which it can perform the differential analysis. This process is repeated for each ROM.

Finally, for each ROM, the system extracts information related to binaries and libraries (ELF), SELinux policies, init scripts, and Linux kernel configurations. Each of these components is handled by an ad-hoc plugin, each of which we discuss in detail next. The entire procedure takes approximately 20 minutes for each ROM.

Implementation Details. The analysis steps above discuss what the process needed to analyze a single ROM, but our framework allows for parallel analysis of multiple ROMs at once. For this work, we parallelized the analysis of the ROMs across 13 virtual machines (VMs), sending the results to a centralized database. Each VM is equipped with *Ubuntu 20.04*, running on *6 CPU @ 2.4Mhz* with *6GB of RAM*.

5.3.2 Tag Identification

Finding the right base image (identified by a git tag) used by a vendor as a starting point for its customization is crucial for our work. Unfortunately, identifying the base image used by a given ROM is not always a straightforward process as there are often many different base images for each "main" version of Android. We now discuss the various techniques and heuristics we developed to pinpoint the base image used by a given ROM.

During the building process of a system image, the build system adds a large amount of information that may help recover the exact git tag forked by the vendor. However, since the vendor controls the entire building system, this information might—and, in fact, often is—removed entirely.

In case the vendor did not modify the build system, the ROM usually includes a `build identifier` that uniquely identifies the starting base image. The format of this identifier may change across different Android versions, and it resides in the `ro.build.id` property of the `build.prop` file. Since the "build.prop" file is present in the system image, we can build a mapping between build identifiers and base images, starting from the official Android documentation [and21]. This mapping shows that, for example, a "build id" equal to `NOF27B` corresponds to Android Nougat release 25 (`android-7.1.1_r25`).

However, if the vendor modified the build system and stripped this information, the identification becomes more challenging. In these cases, we adopt different strategies. First, we look at different values including the `ro.build.description` property (that may still contain the original Android build identifier) and the `ro.com.google.gmsversion` property (which, when combined with the `ro.build.version.sdk` value, can be used to pinpoint the base image). It is important to note that this value *should* always be present when the vendor obtained the *GMS* certification. However, there is no guarantee that the vendor obtained this certification, and we also found ROMs that contain GMS applications but that however did not include a correspondent `gmsversion` property.

If none of these pieces of information is available, we rely on the combination of two properties that are *always* included: `ro.build.version.sdk` (i.e., the Android version) and `ro.build.date.utc` (i.e., the build date). By combining these two values, we can determine the "best" candidate to be considered the base image. In particular, we first list all the AOSP `tags` associated with the target Android SDK version, and we then identify the tag with the nearest creation timestamp.

We note that a vendor cannot easily modify these two final values, because that would introduce usability problems: changing the `sdk` value might introduce undefined and unexpected behavior both from the system and the applications, while changing the `build date` might introduce issues when dealing with system updates (e.g., anti-rollback protections might use this information to avoid booting older firmware [ant20]).

As we explain next in Chapter 5.4, this process worked well in practice. Moreover, even if some errors might have occurred, our analysis is not particularly sensitive to small imprecisions.

5.3.3 Analysis of Binary Customization

We start our analysis by looking at the binaries contained in the ROM. ELF binaries are not just present as standard executables or libraries, but they might also part of APKs' code. Vendors can modify or even add binaries, libraries, or APKs to the system, thus potentially increasing the attack surface. Since those components usually run with high privileges, they are an obvious target for the attackers. This attack surface is particularly important because most of the critical bugs are found within these binaries components, as they are created by using memory unsafe languages (e.g., C, C++).

Moreover, customizations and proprietary code have been the root cause of several recent critical 0-click bugs, e.g., those recently reported by Google

Project Zero's in the (custom) Skia component of Samsung devices [sam20]. The Android OS internally uses the Skia library to handle the processing of pictures for many applications, both installed by the user and the system applications. However, since these pictures might be provided by an untrusted and malicious sources, they can be used as attack vector—both local and remote. Samsung customized this subsystem to add the support for a few other custom and proprietary formats, introducing new functions and logic to Skia. Identifying the code written by Samsung was straightforward, thanks to the symbols being shipped with the library. Several of those functions were vulnerable to memory overwrite, and the exploitation of this vulnerability allowed an unauthenticated, remote attacker to execute arbitrary code on the device.

In this phase, we check how customizations affect three main aspects related to binaries. First, we focus on security hardening techniques: we check whether vendors introduced customizations that lower the security posture of existing AOSP binaries. Second, we check whether the vendor introduced new functionality by adding new binaries or by modifying existing ones (we check for modifications of these binaries by inspecting ELF metadata such as the symbol table). Third, we check how the security posture of new binaries compares to exiting AOSP binaries and settings.

5.3.4 Analysis of SELinux Policies

Security-Enhanced Linux (*SELinux*) is a Mandatory Access Control (MAC) system developed by the NSA and Red-Hat and publicly released in December 2000.

To implement its access control mechanism, SELinux enforces a system-wide security policy. SELinux policies are used to define rules that a process should follow. Each rule of the policy defines the operations a process can perform over a specific labeled resource. All the system resources (e.g., files, sockets, etc.) are labeled; thus, all the interaction between processes (running on a given *context*) and resources can be modeled via the policy. For instance, a rule can indicate that *a process in context X is allowed to open a network connection*. SELinux follows the principle of least privilege: if no rule grants a capability C to a context X, then X does not have access to that capability.

With the introduction of SELinux, the concept of *root* no longer exists, since processes (even the ones running as root) are confined within a given *context*.

In 2012, starting from Android KitKat (4.3), Google officially introduced the support for SELinux on Android (SEAndroid) and partially en-

forced it on the system [SC13]. Then, starting from Android Marshmallow (5.0), the system was fully protected by SELinux. Before this date, the only available sandbox on Android was based on the Linux user-based Discretionary Access Control (DAC).

However, during the years, SELinux has proved to be at the same time a powerful exploit mitigation [sel14], but also the direct cause of several critical security issues due to vendor customizations [cve18]. Indeed, vendors *must* customize SELinux policies as every new file (including those introduced by the vendor) need to be appropriately labeled, and new rules need to be introduced to give proper access to the right contexts. Therefore a custom rule needs to be added to the base policy, inherited from the base image used as a foundation for building the customized system. However, these customizations may also have security repercussions. For example, since base AOSP SELinux can be quite restrictive, vendors may be tempted to relax the policies and somehow circumvent the safety nets implemented by AOSP. To give an extreme example: AOSP defines several "never allow" SELinux rules, which are rules that tell the SELinux compiler "refuse to compile if a different rule is violating it." We found several ROMs with customized SELinux policies that violate base AOSP rules: *this implies that the vendor must have commented out the problematic "never allow" rule instead of redesigning their customization more safely.*

Thus, the analysis of SELinux policies plays a crucial role in understanding the impact of third-party customization over the Android hardening features. Our analysis framework first extracts all customizations to the base SELinux policy and then inspects them to identify several problematic patterns, like the one discussed above.

5.3.5 Analysis of Init Scripts

Unlike other Linux systems, Android uses its initialization process. Android init scripts are textual files with the `.rc` extension and they are written in a dedicated language, namely the *Android Init Language* [and20b]. During the years, this component has been subject to several attacks [pae14]. Most of the vulnerabilities were introduced by third-party customizations, and, most of the time, their exploitation allowed a local attacker to escalate privileges to `root`.

Init scripts are loaded at boot, just after the kernel initialization, and play a crucial role in the Android system setup and bootstrap. Default AOSP init scripts are located in the `/system/` directory, while vendors can add custom scripts in the `/vendor/` or `/odm/` folders.

If a script is loaded from the `/system/` directory, it usually means that it contains the definition for an Android core system component or service, like *ActivityManager* or *installd*. Instead, if a script is loaded from the `/vendor/` or `/odm/` directory, it probably contains instructions to start and setup custom and proprietary services and components. These components might be daemons needed for core System-On-Chip functionality, motion sensor, or other peripheral functionality [and20b].

Init scripts can specify the user/group the binary should be run with, the Linux capabilities that should be granted, and the SELinux context the program should be run with (by default, all init scripts run within the `init` context).

Given the potential security consequences of improper customizations, our framework includes support for the analysis of Android init script to study whether vendors customize default AOSP init scripts or add new ones and to verify if the new services are executed with appropriate user/group and Linux capabilities, and as part of a "safe" SELinux context.

Unfortunately, our experiments show that vendors often customize these scripts, and in some cases, significantly increase the attack surface and leave the device vulnerable to remotely exploitable bugs (with known CVEs).

5.3.6 Kernel Security Analysis

Kernel security has grown in importance in recent years as the number of kernel security bugs reported for Android increased almost ten times in only three years [Kra17]. Moreover, a more detailed analysis of the type of vulnerabilities afflicting the Android kernel showed that vendor customizations introduce 100% of vulnerabilities hitting the *Linux Performance Counter subsystem on such subsystem.*

As a consequence, many kernel hardening techniques were recently introduced. These are so important that the CDD itself introduced a number of "must" requirements in this area that a vendor needs to satisfy to brand its devices as Android.

In Android ROMs, the kernel is usually provided in a binary form within the `boot.img` file. This file contains, in addition to the kernel image, also the *ramdisk*. The Vendor Test Suite implements checks for some of the CDD requirements, but unfortunately, they are quite limited due to the binary-only format of the kernel.

To study the kernel's security, our framework includes various analyses that can extract several security-sensitive information and test for additional CDD requirements. For each kernel, the system first extracts its version, and all the information generally provided within the *Linux Kernel*

banner [ker92]. It then attempts to extract the kernel build configuration options.

However, extracting the kernel configurations from a binary kernel is not straightforward. In fact, the configurations are available if and only if the kernel is compiled with CONFIG_IKCONFIG configuration flag set to Y. This option embeds the entire .config file within the kernel binary. In this case, it is possible to retrieve the configuration by using the tools available in the official Linux Kernel project [lin91, ext10]. However, vendors can also disable this feature, and thus, not embedding the configuration file within the final kernel binary. When kernel configurations are not available, we proceed by extracting the symbols exported by the kernel [vml19]. From the symbols, it is then possible to infer a subset of configurations used while compiling the final kernel: for instance, the symbol `___stack_chk_fail` can be paired with the CONFIG_CC_STACKPROTECTOR_REGULAR and CONFIG_CC_STACKPROTECTOR_STRONG configurations. However, symbols are only exported in the `___ksymtab Section` if and only if the kernel is compiled with the CONFIG_MODULES option.

It is worrisome to note that, even though this approach does not support *all* kernel config options, it was sufficient to identify ROMs that violated several CDD "must" requirements.

5.4 Dataset Characterization

To perform our longitudinal analysis, we set out to build a comprehensive dataset of Android ROMs that would help us to completely visualize the evolution of the entire ecosystem from a vendor perspective. For what concerns the official Google ROMs, we downloaded them from their official website [goo21]. We downloaded the other vendors' ROMs from firmwarefile.com [fir15] and stockrom.net [sto21]. In total, our dataset consists of 2,907 Android ROMs, which span across 42 different vendors and cover 1,403 different device models. For what concerns the SDK distribution, our dataset covers the Android system's evolution from version 2.3.3 to version 9 (i.e., from SDK 10 to 28). The oldest image dates back to 2010, while the newest is from 2020. Figure 5.1 presents the distribution of our dataset in terms of SDK distribution.

According to public statistics [ven21], our dataset is also heterogeneous in terms of coverage of different vendors: half of our dataset is constituted by "big players" (e.g., Samsung, Huawei, LG, and Xiaomi), while the remaining ROMs belong to vendors with a market share less than 4% (e.g., Google,

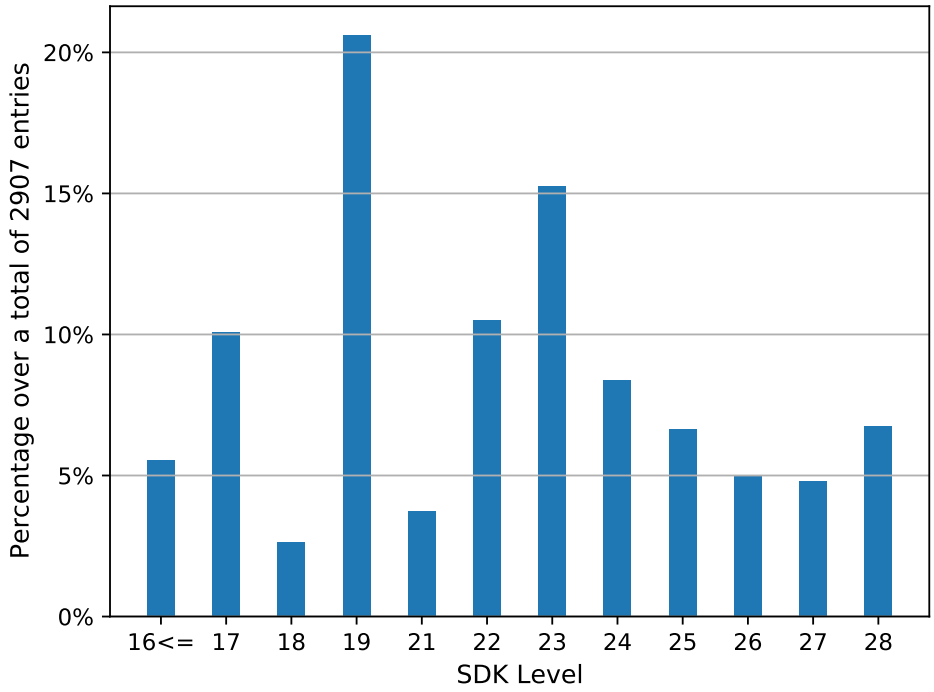


Figure 5.1: SDK Level Distribution.

Lenovo, Mobicel, Motorola, Oppo, Realme, and Vivo).

According to public statistics from 2010 to 2020 [ven21], our dataset is also heterogeneous in terms of coverage of different vendors according their market share [ven21]. Half of our dataset is constituted by "big players" (e.g., Samsung, Huawei, LG, and Xiaomi), while the remaining ROMs belong to vendors with a market share less than 4% (e.g., Google, Lenovo, Mobicel, Motorola, Oppo, Realme, and Vivo).

Moreover, as discussed in the previous section, our customization analysis needs to compare a given ROM against its associated "base image." To this end, we also created a set of 326 ROMs by compiling *all* AOSP versions (i.e., tags) that are the base image of a least one ROM in our dataset, as presented in Chapter 5.3.2. Obviously, these last 326 ROMs are not counted in our statistics.

To identify the tag of the AOSP base image, we relied on the following heuristics:

- `ro.build.id` value for the 88% (2,566) of our dataset,

- Combination of build date (`ro.build.date.utc`) and `ro.build.version.sdk` for 9% of the ROM (261),
- `ro.com.google.gmsversion` property for 2% (59) of the system image,
- for the remaining 23, the value of the `ro.build.description`

Note that all the 2,907 ROMs in our dataset contain applications from the Google Suite; thus, we can assume they all obtained the GMS certification from Google.

5.5 Compliance

We now present the results of our *compliance analysis*. We start by describing the *knowledge base*, a set of rules and requirements, extracted from the CDD defined by Google, that vendors should follow to obtain the GMS Certification and be compliant and compatible with a given Android version. Then, we present the analysis we performed over our dataset to verify if vendors are effectively enforcing these requirements. Our analysis is divided into three main components: *Kernel Configurations, SELinux, and User-space binary hardening*. For each component, we analyze if it is compliant with the requirements defined in the CDD.

Given the fact that we report some cases in which there is a violation of the Compatibility Definition Document—and the test suites should have detected it—we double-checked if a given ROM has effectively obtained the GMS certification; we check for the presence of the Google apps and the field GMS in the `build.prop` file.

All the ROMs in our dataset are branded Android and contain applications from the Google Suite (and thus obtained the GMS certifications). Therefore, one would expect them to be compliant with the mandatory requirements of the CDD. This is important because system security aspects always played a crucial role in the CDD, which contained an entire chapter dedicated to the *Security Model Compatibility* since its first edition in 2009. Mandatory requirements are clearly marked as "*must*," and a failure to implement them is a clear *violation* of the CDD. Alternatively, a feature can be defined as *strongly recommended*: in this case, not implementing such a feature is not a strict violation of the CDD.

We manually extracted all these requirements from the CDD of Android 1.6 to Android 9, as listed in Table 5.1. In order of appearance, the first system hardening requirement was introduced in the CDD of Android 4.3, where Google announced the support of *SELinux* for Android devices.

Therefore all ROMs based on Android 4.3+ *must support and implement the SELinux Mandatory Access Control*. Then, starting from Android 7, the Security Model Compatibility section has focused mainly on kernel configuration options. Surprisingly, the CDD security requirements do not mention *user-space hardening* until Android 9, and the only user-space hardening requirement is defined only as *strongly recommended*. Since the introduction of Project Treble made updates faster and easier for OEMs to roll out to devices and introduced several tests to verify and test the OS kernel, we present our results divided before and after its introduction. This distinction can help us understand to which extent the introduction of Project Treble was able to mitigate the problem of Android customizations.

5.5.1 Kernel Configurations Compliance

As previously discussed in Chapter 5.3.6, for each of the 2,907 ROMs, we analyzed their Linux-based kernel binary to identify potential misconfigurations in contrast with the strict requirements defined in the CDD. However, we identified that 262 ROMs in our dataset did not contain the kernel binary, and therefore we excluded them from our analysis.

For 249 of the remaining 2,645 kernels, our system was unable to extract neither their kernel configuration nor the symbols from the kernel binary. This is because those kernels were compiled without the CONFIG_IKCONFIG and CONFIG_MODULES [lkm20] configurations. However, as described in [cor20], both configurations *must* be enabled for kernels targeting Android 8.0 and higher. Besides, by reading the *Core Kernel Requirements* [cor20] defined in the *Vendor Test Suite (VTS)* for Android 8.0 and higher, we noticed how the configuration of these 249 kernels should violate and fail the tests. Out of the 249 kernels missing these configurations, 162 corresponded to the Android version $\geq$ 8.0. Thus, as the first result of this analysis, we highlight how these 162 kernels *are not compliant with Android*.

This casts a shadow on the strictness of these requirements' enforcement, especially since some of these could have been automatically checked.

For the other 2,396 kernels, we retrieved the textual configuration from 561 kernels and the symbol table for the remaining. Identifying violations on kernels having their configuration is straightforward as the Compatibility Definition Document precisely indicates which configuration options must be used. On the other hand, verifying violations with the only support of the kernel binary symbols is not immediate. However, we noticed how almost every kernel configuration defined in the CDD introduces a set of specific symbols, and therefore it is possible to infer a specific compilation flag based on

Table 5.1: System Hardening Requirements defined in the Compatibility Definition Document.

SDK	Version	Compatibility Definition Document	
		MUST	**STRONGLY RECOMMENDED**
4 - 17	1.6 - 4.2	**None**	**None**
18	4.3	**SELinux:** support Permissive Mode	**None**
19	4.4	**SELinux:** contexts installd, netd, and vold in Enforcing Mode	**SELinux:** other domains remain in Permissive Mode
20 - 23	5.0 - 6.1	**SELinux:** global Enforcing Mode **SELinux:** all domains in Enforcing Mode **SELinux:** not modify, omit, or replace the neverallow rules present within the SELinux AOSP folder all domains in Enforcing Mode	**None**
24 - 25	7.0 - 7.1	**Kernel:** support for seccomp-BPF support (TSYNC)	**None**
26 - 27	8.0 - 8.1	**Kernel:** support for CONFIG_CC_STACKPROTECTOR_REGULAR or CONFIG_CC_STACKPROTECTOR_STRONG **Kernel:** support for CONFIG_DEBUG_RODATA or CONFIG_STRICT_KERNEL_RWX	**Kernel:** support for data read-only after initialization (ro_after_init) **Kernel:** support for CONFIG_HARDENED_USERCOPY **Kernel:** support for CONFIG_CPU_SW_DOMAIN_PAN or CONFIG_ARM64_SW_TTBR0_PAN **Kernel:** support for CONFIG_RANDOMIZE_BASE
28	9	**Kernel:** support for CONFIG_PAGE_TABLE_ISOLATION or CONFIG_UNMAP_KERNEL_AT_EL0 **Kernel:** support for CONFIG_HARDENED_USERCOPY	**Userspace:** do not disable CFI/IntSan on components that have it enabled

the symbols included within the binary. The mapping we use to connect kernel configurations and symbols is summarized in Table 5.2. We noticed how, however, some kernel configurations might map to the same symbol, while other kernel configurations, depending on the version of the kernel, might change the symbol used. For these configurations, we rely on regular expressions to identify valid symbols. It is important to note that since these flags are interchangeable, we conservatively mark a kernel to be *not compliant* if and only if it does not implement *any* of the available options. For example, if a kernel adopts CONFIG_CC_STACKPROTECTOR_STRONG rather than CONFIG_CC_STACKPROTECTOR_REGULAR, we *do not* mark it as not compliant since the CDD requires the vendor to implement at least one of the two. For 3 configurations, we were not able to identify any symbol for the mapping.

Our analysis identified that 7.9% (190 out of 2,396) of the kernels (from 10 different vendors) violate the CDD for their specific Android version since they do not implement one or more *mandatory* security requirement.

Amongst these 162 are used in ROMs re-architectured with Project Treble, thus targeting an Android version greater or equal than 8.0. The most common violation, found on 150 kernels, relates to the absence of kernel memory protections aimed at marking sensitive memory regions and sections read-only or non-executable (which can be enabled with CONFIG_DEBUG_RODATA or CONFIG_STRICT_KERNEL_RWX).

We also identified 10% (241 out of 2,396) of the kernels (from 10 vendors) do not implement one or more strongly recommended features. This time, we noticed how 160 vendors did not enable CONFIG_RANDOMIZE_BASE (no Kernel Address Space Layout Randomization); hence, these kernels do not implement any randomization of their base address once loaded. Although these features are not mandatory, the Vendor Test Suites inform the vendor if any strongly recommended features are missing. Thus, even though these vendors were warned about the lack of these features, they ignored the advice and did not include them in their final product.

Table 5.3 shows the evolution of violations across different SDK levels. The table shows that the re-architecture introduced with Project Treble and the testing performed with the VTS, even though not mandatory, are not enough to counter the problem of customization on Android from the Kernel Security perspective.

On the contrary, it can be observed that many kernels still do not comply with the directives imposed by Google and continue to release on the

Table 5.2: Mapping from Kernel Configuration To ELF Symbols.

Kernel Configuration	**Kernel Symbol**
CONFIG_SECURITY_SELINUX	Symbol contains `selinux`
CONFIG_SECCOMP	Symbol contains `seccomp`
CONFIG_CC_STACKPROTECTOR_REGULAR	`__stack_chk_fail`
CONFIG_CC_STACKPROTECTOR_STRONG	`__stack_chk_guard`
CONFIG_DEBUG_RODATA	`rodata_enabled`, `set_debug_rodata`, `___setup_set_debug_rodata`
CONFIG_STRICT_KERNEL_RWX	`mark_readonly`
CONFIG_HARDENED_USERCOPY	`__check_heap_object`, `___check_object_size`
CONFIG_ARM64_SW_TTBR0_PAN	`reserved_ttbr0`
CONFIG_RANDOMIZE_BASE	Symbol contains `kaslr`
CONFIG_PAGE_TABLE_ISOLATION	`tlb_flush_mmu_tlbonly`
CONFIG_UNMAP_KERNEL_AT_EL0	`__initcall_map_entry_trampoline1`
CONFIG_HARDEN_BRANCH_PREDICTOR	`__nospectre_v2`
CONFIG_SHADOW_CALL_STACK	`init_shadow_call_stack`
CONFIG_SECURITY_DMESG_RESTRICT	`dmesg_restrict`
CONFIG_SECURITY_KPTR_RESTRICT	`kptr_restrict`
CONFIG_ARM64_PAN	`cpu_enable_pan`
CONFIG_CFI_CLANG	Symbol contains `___cfi_*`
CONFIG_DEFAULT_MMAP_MIN_ADDR	No symbol mapping found, variable
CONFIG_CPU_SW_DOMAIN_PAN	No symbol mapping, inline assembly
CONFIG_LTO_CLANG	No symbol mapping found

Table 5.3: Violations regarding the kernel configuration.

SDK	Version	# Kernel	# Violations CDD	# Strongly Recommended
18	4.3	77	26 (33.8%)	–
19	4.4	599	3 (0.5%)	–
26	8.0	145	50 (34.5%)	70 (48.3%)
27	8.1	140	33 (23.6%)	66 (47.1%)
28	9.0	196	78 (39.8%)	101 (51.5%)
		2396	**190 (7.9%)**	**237 (9.9%)**

market devices equipped with kernels that do not meet the mandatory security specifications. The numerous tests should have identified (and likely actually did identify) all these violations, which would be enough to mark the final ROMs as *non-compliant*.

5.5.2 SELinux Compliance

For each Android version that supports SELinux, AOSP provides a standard policy that vendors can use as a base to build and customize their SELinux configuration. As previously discussed in Chapter 5.3.4, starting from Android 4.3, Google introduced as a strong requirement that all third-party vendors *must* adopt this new Mandatory Access Control system. The Compatibility Definition Document mandates that third-party vendors *must support SELinux in Permissive Mode*, which means that every violation is logged, but not enforced, so to provide vendors enough information for an adequate fix to the component causing the error. Instead, from Android 4.4, Google started to protect few critical services with SELinux and forced the vendors to do the same: hence, vendors were required to set up SELinux in *Enforcing Mode* at least for the three domains `installd`, `netd`, and `vold`. Starting from Android 5.0, instead, vendors were required to set up SELinux in Enforcing Mode for all the domains. Moreover, from this version, vendors *must not* modify, omit, or replace some AOSP specific rules, which act as a safety net for misconfigurations. These rules are the so-called `neverallow` rules: if a custom SELinux policy directly or indirectly violates any of these rules, the SELinux toolchain would throw a compilation error, thus preventing the adoption of unsafe configurations from the beginning. With these rules, it is possible to *avoid and mitigate potential known security issues and harmful behaviors*, such as forbidding any third-

party application to write to files in the `/sys` directory or preventing them from receiving and sending `uevent messages`. We note that modifying (or removing) any of these `neverallow` rules is a strict violation of the CDD.

To determine whether a ROM is compliant with the SELinux requirements, we proceed in two steps. First, we look at violations related to *Permissive Mode* by inspecting the SELinux policy available in the ROM (since it is possible to retrieve all the permissive domains directly from the compiled policy). Second, we look for vendors that manipulated the base policy provided in AOSP to overcome the restrictions imposed by the `neverallow` rules. For this step, we retrieve the tag of used as a base image by the vendor, as described in Chapter 5.3.2), and we compare the two sets of policies.

Out of the 2,907 ROMs, we identified 1,090 of them not containing a SELinux policy. Of these 1,090, 452 are targeting an Android version lower than 4.3, and it is thus expected that they do not have any policy.

Since SELinux must have kernel support to work, we decided to intersect the remaining ROMs with the results extracted from the previous kernel analysis, presented in Chapter 5.5.1) and we identified how 29 lack CONFIG_SECURITY_SELINUX: for those, it is expected that we do not find SELinux configurations.

The remaining 609 ROMs are divided as follows: for 167 we were not able to obtain the `boot.img`, and for 91 of them we were not able to extract neither the kernel configuration nor the symbol table; thus, we cannot perform any measurement on these ROMs. For 351 ROMs, we identified that they correctly support SELinux at kernel level, but no policy has been found: we suppose these might be incremental updates, not containing the policy.

We now focus our discussion on the remaining 1,817 ROMs that define a SELinux policy. Out of them, 7% (108 ROMs) violate the CDD specification for their corresponding Android version as they still define one or more `permissive` domains. We found this violation spread across 16 different vendors. We also analyzed the distribution of these violations with respect to their SDK level to determine whether this problem only affects older versions of Android. Surprisingly, we noticed that even if Google forbids permissive domains starting from Android 5.0 (and thus from SDK 20), several ROMs are still not complaint even after *four* major releases, and after an almost complete redesign of SELinux on Android 8 [pro17].

Table 5.4 summarizes the results of this analysis. We divided the results before and after the introduction of Project Treble to show once more how the problem persisted even after the introduction of the new system design.

Table 5.4: Violations of permissive domains in the SELinux policy.

SDK	Version	# ROM CDD Violations	Permissive Domains			
			Max	Min	Avg	σ
21	5.0	1/58 (1.7%)	5	5	5.0	0
22	5.1	26/251 (10.3%)	7	1	3.1	2.2
23	6.0	21/359 (5.8%)	5	1	2.0	1.3
24	7.0	11/226 (4.8%)	2	1	1.0	0.3
25	7.1	2/163 (1.2%)	1	1	1.0	0
26	8.0	21/141 (14.8%)	4	1	2.0	1.3
27	8.1	18/139 (12.9%)	1	1	1.0	0
28	9.0	8/196 (4.0%)	1	1	1.0	0
108/1533 (7.0%)						

s We then performed the second analysis to identify whether a vendor tampered with any of the predefined `neverallow` rules, which is a strict violation of the CDD, starting from Android 5.0. However, detecting this type of violation is not straightforward. Each Android version contains a preset of SELinux rules: the `neverallow` rules are part of this base policy. At compilation time, the SELinux policy compiler *verifies if any rules defined in the policy are in contrast with what is defined by the neverallow rules*: if a violation is identified, the compiler throws a compilation error. However, these checks are performed *only* at compilation time and are not enforced at runtime. Therefore, potentially, a third-party vendor facing a violation of a `neverallow` rule introduced by one of its customizations may be tempted to "solve" the issue by just changing or removing the `neverallow` rule that prevents the compilation of the final policy. Thus, by analyzing only the vendor policy of the final ROM is not possible to conclude whether it violates the CDD requirement.

To detect these violations, we proceed as follows: for each ROM, we first retrieved the tag used by the vendor as a base system, and we save both the textual and the compiled version of the policy. Then, we identify all the differences between the compiled policies, and we collect the customizations introduced by the vendor. For each of the additional vendor-only rules, we then try to recompile the original AOSP policy with the addition of the new rule, and we check for compilation errors. In case of compilation errors, we finally check whether it is due to a *neverallow rule violation*, and if so, we

Table 5.5: AOSP SELinux Violations.

SDK	Version	# ROM CDD Violations	Neverallow Rules Violations			
			Max	Min	Avg	σ
21	5.0	1/58 (1.7%)	8	8	8.0	0
22	5.1	20/251 (7.9%)	39	1	4.6	10.4
23	6.0	58/359 (16.1%)	121	1	3.6	15.7
24	7.0	8/226 (3.5%)	10	1	7.2	3.9
25	7.1	3/163 (1.8%)	158	1	56.3	88.1
26	8.0	121/141 (85.8%)	27	1	7.2	7.7
27	8.1	110/139 (79.1%)	25	2	7.2	7.5
28	9.0	122/196 (62.2%)	37	1	4.0	8.8
		443/1533 (28.9%)				

mark the vendor policy as not compliant.

Out of 1,533 ROMs with a SELinux policy (and that target Android ≥ 5), we identified that 29% of them (443) violated the CDD by defining one or more rules violating one of them default neverallow rules. For all these images, from 21 unique vendors, it was possible to identify SELinux policies allowing operations that were not supposed to be available. Table 5.5 summarizes the results of this second analysis. Also in this case, the introduction of Project Treble failed to mitigate the vendors' problems related to SELinux customizations. As can be seen from Table 5.5, if we consider the results for SDK level 26, 27, and 28, we see how this problem has increased dramatically, reaching peaks of 85% of the ROMs having at least one violation.

The results we have collected show that *vendors actually do violate the Compatibility Definition Document often*, making them not compliant with the defined rules, and potentially introducing security issues. We would like to note that these results do *not* imply maliciousness bad faith on the vendors' part: we believe most of the vendors use this practice to fix compatibility issues introduced by their customizations quickly. Indeed, modifying or commenting out a `neverallow` rule is much easier than potentially re-architecting a customization to fit the requirements.

5.5.3 Binary Compliance

The last category of system hardening defined by Google is related to user-space binaries. As previously discussed, the requirements for binaries were introduced only in Android 9, and so far, they only cover two aspects: *Control Flow Integrity* (CFI) and *Integer Overflow Sanitization* (IntSan). Control Flow Integrity is a security mechanism that tries to prevent changes to the control flow of a compiled binary, making exploitations that require hijacking the "expected" control flow much harder. Integer Overflow Sanitization, instead, provides compile-time instrumentation to detect signed and unsigned arithmetic integer overflow: when an overflow is detected, the process safely aborts.

Both protection systems have been gradually introduced by Google to harden the Android Media Stack component [med16, med20], which has been subject to numerous attacks over the years, including Stagefright [sta15], which could have been prevented with these two hardening techniques.

To take advantage of these new protections, the developer must use a compiler that supports them. Officially, Google uses and supports Clang, but both features are also available on the GCC compiler.

As presented in Table 5.1, the only requirement for the user-space binaries is defined as *strongly recommended*, and it asks vendors to not remove CFI or IntSan compiler mitigations from components that have them enabled. Thus, to identify if vendors adhere to this recommendation, we proceed as follows: for each ROM, we first identify its AOSP base image and the we extract all binaries shared between the vendor ROM and the corresponding AOSP base image. Finally, for each of these binaries, we tested their security features: if the original binary (present in the AOSP base image) has CFI or IntSan enabled and the corresponding binary in the third-party ROM does not, we mark the ROM as not respecting the recommendation suggested in the CDD.

Since both defense mechanisms were introduced in the CDD from Android 9, we only considered the 196 ROMs with SDK ≥ 28. Among them, 85 (4.34×10^1 %) contained at least one binary that disabled CFI and 104 (5.31×10^1 %) contained at least one that disabled IntSan. In these cases, six unique vendors *lowered the security of a binary*, with respect to AOSP, thus violating the CDD recommendation.

However, these vendors did not entirely disable CFI or IntSan for all the binaries: on average, among the ROMs that have violated the recommendations, the vendors disabled CFI for 38.7% ($\sigma = 36.5$) of the binaries, while they disabled IntSan for 35.8% ($\sigma = 34.9$) of them.

5.6 Additional Customizations

We now discuss our analysis of OEM customizations that, even though may not constitute a strict violation of the requirements, do negatively impact the security posture of the overall ROM. We start by presenting the analysis on ELF binaries, showing the impact that vendors have, with their customizations, on binaries and system libraries. We then proceed with the analysis of the Android Init Script customizations, illustrating which are the most common patterns of modification by vendors and which may be the issues they introduce. We conclude by presenting the results of the SELinux policy study, showing its evolution over the years, and emphasizing how fragmentation has had and continues to have an important influence and impact on this component.

5.6.1 New Functions in System Libraries

The vast majority of Android's core system components are still written in unsafe memory languages like C and C++ and shipped as ELF libraries. Vendors can add functionalities to such libraries, which can result in an increased attack surface. A recent (fixed in May 2020) impactful example is a bug found in Samsung's customizations of Google's Skia library [sam20]. The library is used to process pictures for many applications, and Samsung customized it to add support for new proprietary formats. Unfortunately, several of these functions were vulnerable to memory corruption bugs, and the exploitation of this vulnerability allowed an unauthenticated remote attacker (i.e., 0-click) to execute arbitrary code on the device. Therefore, we quantitatively measured the new functions introduced by the vendors modifying the AOSP libraries to depict this phenomenon.

To assess the prevalence of vendor customizations that add functionalities, given a ROM as input, we first inspect all binaries that are also found in the original AOSP (we refer to this subset of binaries with the term *Shared Libraries*), and we extract the list of exported functions that were not present in the original version. For example, let us suppose we have a ROM containing the library `/system/lib/example.so` that exports the functions $[f1, f2, f3]$, and the same library (i.e., same absolute path and same name) in the base AOSP is exporting just $[f1, f2]$; it can be safe to suppose that the vendor introduced $f3$. Since a vendor can use different library versions that might have been taken from another AOSP branch, we opted for the following conservative approach: we consider a ROM to have added functionality to a given binary if it contains symbols that do not appear in *any* (i.e., older or newer) AOSP releases. Moreover, we also con-

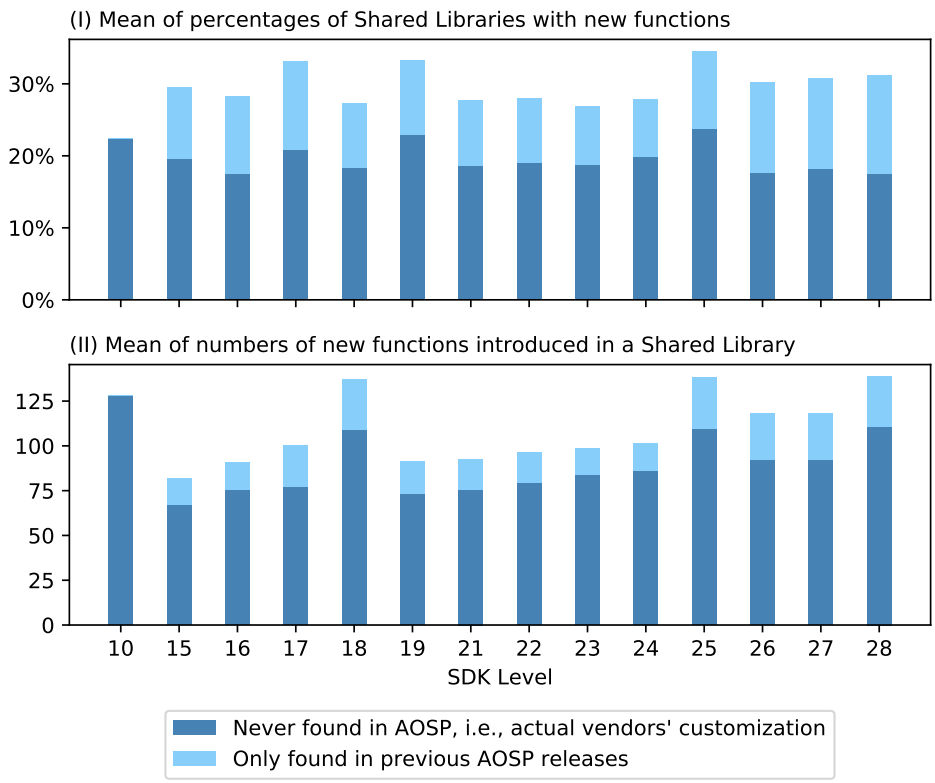


Figure 5.2: Analysis of new exported functions introduced in AOSP libraries.

sidered the scenario in which a vendor is still using a function from an old version of AOSP that is no longer present in the subsequent AOSP tagged releases. So, using the example above, if we find a new exported function $f3$, we also check if in the entire future history of the AOSP base images, the library `/system/lib/example.so` does not export $f3$. At the end of this analysis, we obtained all the new functions introduced by the vendors in the Shared Libraries. Figure 5.2 summarizes our findings.

The two bar plots share the same x-axis (the SDK level) and report respectively the mean of the percentages of Shared Libraries in a ROM in which a vendor has introduced new functions and the numbers of new functions (when a vendor introduced at least one). The dark bar highlights the *actual* vendors' customizations, while the soft bar displays a modification found in a previous version of AOSP. There is no soft bar in the correspondence of SDK 10 because there are no ROMs older than that in our

dataset.

The results show an almost constant trend of roughly 80 new functions added to 20% of the system libraries, thus vanishing Project Treble's efforts. Nevertheless, we also note that vendors are still using old AOSP functions, maybe because their code still depends on them. However, using a function that is no longer maintained in AOSP can be dangerous because it does not receive security patches.

We suppose that these results might be due to the high percentage of custom code being added throughout the years; thus, it might be difficult for vendors to remove all the legacy code from such libraries and restore the functionalities on separated components.

5.6.2 Compile-time Hardening

In addition to the Compatibility Definition Document, Google also maintains a Security Enhancements (SE) webpage [sec20], in which it presents the security and privacy enhancements for each Android version. While the CDD only started to discuss binary hardening in Android 9 (2018), the Security Enhancements discusses this topic since Android 3 (2009). This webpage is not directly linked from the CDD, so it is not mandatory for the OEMs to implement such enhancements. However, since these aspects are security-relevant, we analyzed customizations related to these aspects as well.

Thus, we first went over all the security features reported in the SE and collected all mitigation techniques related to binary hardening. We now present a detailed description of the several mitigation techniques with which it is possible to protect a binary or library. For each technique we provide a short description and we explain which artifacts we considered to detect if and ELF file implements it or not.

I) Stack Canaries. Stack canaries, introduced in Android 1.5, work by placing a random integer (canary) in memory just before the stack return pointer. In order to overwrite the return pointer (and thus take control of the execution flow), stack-based buffer overflows attacks must also overwrite the canary value. Before a function returns, the stack canary integrity is checked using the function `__stack_chk_fail` (or `__intel_security_cookie` in an alternative implementation), and if it appears to be modified due to an overwrite, the program exits immediately. Thus, we checked for the presence of the aforementioned function among the binary's symbols. We highlight how the stack protector works in two configurations. The first one protects the buffer only if it is greater than a

certain size (depending on the architecture), while the second one (named "strong") protects buffers even if they are one byte size. We assumed, as safe assumption, that real world binaries have at least one buffer that can be protected by this compiler defense mechanism. Even though this assumption might sometimes fail, we believe that the numbers of binaries without a buffer to protect is negligible and it is not going to affect the overall results of our measurement.

II) No eXecute (NX). NX marks certain areas of the program as not executable. NX can be implemented both via software or hardware (almost all modern processors uses it). In our analysis, we checked if the `GNU_STACK` segment of the binaries, which tells the system whether the stack should be executable or not.

III) Position Independent Executables (PIE). The code of a PIE binary can be placed into random locations in memory, and it executes properly regardless of its absolute address. PIE works in tandem with Address Space Layout Randomization (ASLR). ASLR randomly arranges the address space positions of key data areas of a process (e.g., the base of the executable, stack, heap, and libraries). If the executable is position independent, the location of the executable code within the process is also randomized, making it more difficult for an attacker to predict target addresses. As of Android 4.0 (SDK 15), the kernel gained support for ASLR, but Android still lacked userspace support. The Android 4.1 (SDK 16) release introduced support for full ASLR by enabling heap randomization and adding linker support for PIE. Android 5 (SDK 21) is the latest step forwards, as non-PIE executable support was dropped, and all processes now have full ASLR. PIE is the security enhancement with the greater adoption because, after Android 5, the linker does not load non-PIE executables. A PIE ELF file is of the type `ET_DYN`, and its `.dynamic` section contains the `DT_DEBUG` tag.

IV) Full Relocation Read-Only (RELRO). A dynamically linked ELF binary uses a look-up table called Global Offset Table (GOT) to dynamically resolve functions located in shared libraries. The dynamic linker defers function-call resolution to the point when the function is called rather than at load time. This technique is known as lazy binding, and it needs that the GOT lives in a predefined place and is writable. Hence, if an attacker finds a bug allowing them to write a few bytes (as many as the length of a valid address), they can overwrite a GOT entry. If a GOT entry is properly overwritten, the attacker can hijack a library call to their malicious code. However, the immediate binding is a valid countermeasure: the linker can

resolve all the dynamically linked functions at the beginning of execution and make the GOT read-only. This mitigation is known as *Full RELRO*, and it appears in the SE of Android 4.1. If an ELF implements the Full RELRO, it has the GNU_RELRO segment and its .dynamic section contains the DT_BIND_NOW tag. The GNU_RELRO segment indicates the memory region which should be made read-only after relocation is done, while the .dynamic section contains an array of tags. The DT_BIND_NOW tag indicates the linker that all relocations must be processed before returning control to the program, i.e., using immediate binding.

V) FORTIFY_SOURCE. This is a macro (available in both GCC and Clang) that provides lightweight checks for detecting buffer overflows in various dangerous functions, like memcpy. Some of the checks can be performed at compile time while other checks take place at run-time and result in a run-time error if the check fails. FORTIFY_SOURCE works in two phases: first, it tries to computes the number of bytes of the destination buffer used in a dangerous function. If it succeed, it replaces the dangerous functions with their secure __chk counterpart (e.g., memcpy → __memcpy_chk) adding as new argument the size of the buffer. If an attacker tries to copy more bytes, the __chk function detects the overflow and the program's execution is stopped. If the first step fails, the compiler cannot harden a function (e.g., it might fail with dynamic memory allocated buffers). For dynamically linked executables, the libc contains the implementation of the __chk functions. Therefore, we first checked whether the libc supports FORTIFY_SOURCE, that is, the libc contains at least one __chk function among its exported symbols. If yes, for each binary, we check if it contains at least one __chk function among its imported symbols.

VI) setuid/setgid. These are a special type of file permissions that permit users to run specific executables with temporarily elevated privileges, to perform a specific task.

We do not mention Control Flow Integrity and Integer Overflow Sanitizer because they were already discussed. We then compiled a list of artifacts whose presence or absence can be used to infer whether an ELF binary implements or not each mitigation technique. This list is summarized in Table 5.6.

This information allowed us to compare the security-related compiler options used by vendors for their binaries with respect to the one used by the corresponding AOSP base image. First, we found a positive indication. The vendor's binaries in common with the AOSP base image have the same

Table 5.6: Userspace Mitigation Techniques.

SDK	Version	Enhancement	Artifact
3	1.5	Stack Canaries	`___stack_chk_fail` function symbol, or `___intel_security_cookie` function symbol
9	2.3	No eXecute (NX)	`GNU_STACK` segment `RW-`
16	4.1	Position Independent Executables (PIE)	ELF type `ET_DYN`, and `.dynamic` section with `DT_DEBUG` tag
		Full Read-only Relocations (RELRO)	`GNU_RELRO` segment, and `.dynamic` section with `DT_BIND_NOW` tag
17	4.2	FORTIFY_SOURCE	`*_chk` function symbols, and `*_chk` exported function in libc
18	4.3	No setuid/setgid programs	setuid/setguid bit in file's permission

mitigation techniques, that is, third-party vendors do not (usually) modify the AOSP settings.

However, the results are different when we compare the binaries that are only present in the vendor's ROM (i.e., those binaries that are not present in any version of vanilla AOSP). Figure 5.3 presents the result of this analysis as the mean computed over the ROMs aggregated by SDK version. The vertical red line shows the point in time when the security feature was mentioned in the Security Enhancement for the first time (Stack Canaries and NX have no vertical line because they were introduced even before SDK 10). The dash-dotted line represents the means of AOSP binaries, while the continuous line (supplied with standard deviation) the vendors' binaries. All graphs clearly show that the new binaries added by the vendors consistently use fewer mitigation techniques than the binaries in AOSP. At a closer look, we can also observe other interesting trends. For instance, even if stack canaries are amongst the oldest security feature, it took several years for vendors to adopt them—and still today, around 40% of vendors binary lack this basic feature—probably because it slightly penalizes the performance [DMW15]. NX adoption and Full RELRO have instead always been very common in AOSP binaries, while the gap with the vendors' binaries is still substantial.

Moreover, we found an inconsistency concerning NX adoption: the CDD never mentions NX, while the CTS contains a test to verify at run-time if NX is enabled [cts11]. The Security Enhancement webpage presents NX in Android 2.3, released in December 2010, and the test was committed in March 2011. This fact is odd because the test is checking (and thus enforcing) for a feature that is not explicitly requested.

Finally, we measured the prevalence of `setuid`/`setgid` files. Since Android 4.3 (SDK 18), the AOSP removed all `setuid` executables. Among the vendor binaries, we found that 319/447 (71%) of the ROMs with SDK < 18 and 371/2453 (15%) of the ROMs with SDK ≥ 18 contain at least one `setuid` executable. In particular, ROMs with SDK ≥ 18 should never contain any `setuid` executables (and in fact AOSP contains none). At a closer analysis, the binaries that appear more frequently in those ROMs are `su` (18%), `procmem` (17%), `netcfg` (16%), `procrank` (12%), and `tcpdump` (11%). Developers often use these executables for debugging purposes, but they should be removed from the final released ROM since the presence of `setuid` executables can severely affect the overall security posture of the device.

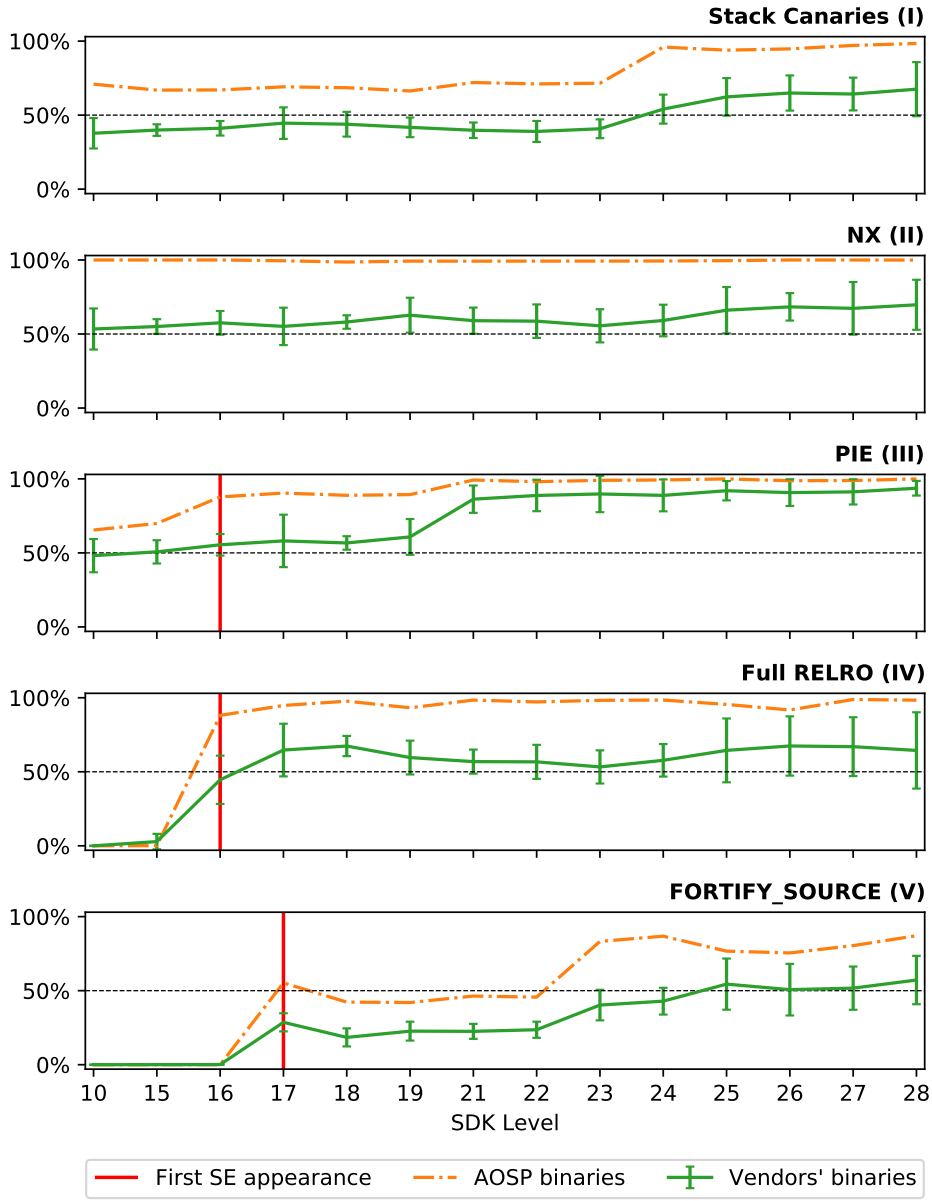


Figure 5.3: Mean of percentages of binaries using a security feature.

5.6.3 Android Init Script Customizations

Among the various binaries added by vendors, some of them are used to perform critical functionalities for the entire system operation, acting as

daemons. Android OS relies on a custom init script system to start binaries at boot time. By default, all the native daemons are started at boot, as root user. Unfortunately, over the years, this component has been subject to numerous problems, which have very often led to the introduction of logical bugs that have resulted in the complete compromise of the device. Most of the time, it has been noted how these vulnerabilities were introduced by vendors [pae14].

To study this aspect, our system extracts from each ROM how many *new* services it defines with respect to its corresponding AOSP base image. However, not all services are started every time. Therefore, among them, we mainly focus on those that start at system boot and that run with root user privileges.

Figure 5.4 summarizes our findings.

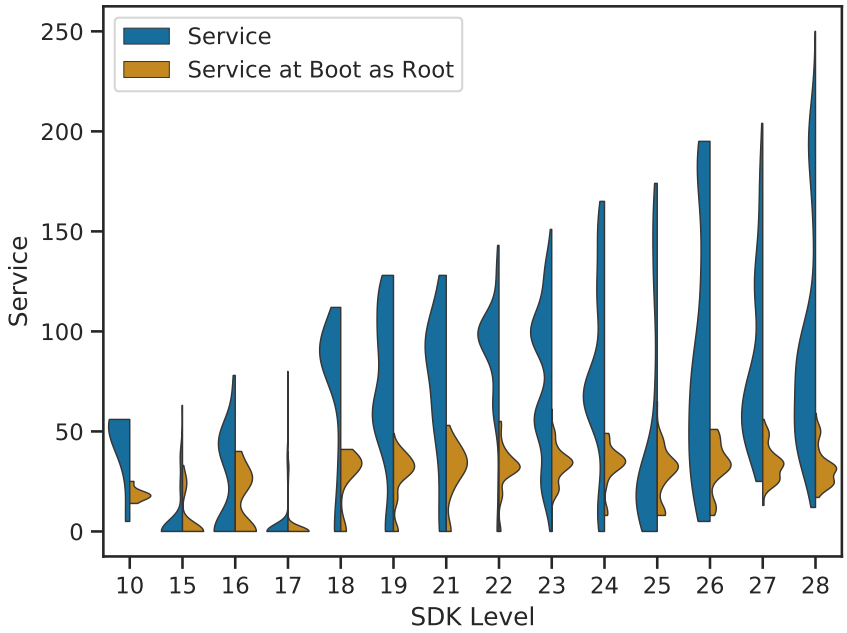


Figure 5.4: Evolution of Android Init Scripts.

For each SDK level, the figure displays the distribution of the number of new services: the left side of the violin plot covers *all* new services, while the right side only covers those that start at boot *and* with root permissions. The plot shows how, over the years, vendors have made considerable

changes to init scripts and, in particular, how the total number of newly defined services is constantly growing—with some ROMs defining almost 250 additional services compared with AOSP.

To put this in terms of absolute numbers, for instance, an AOSP 8.0 (SDK 26) had, on average, 59 services defined in the init script, while an average ROM had 90, with a peak of vendors defining 195 additional services. The astonishing number of services that start as root is worrisome, and vendors are likely violating the least privilege principle. It is, in fact, much more straightforward for vendors to run a binary as root with respect to running it with less privilege, granting it only the capabilities that are strictly needed, and properly configuring SELinux policies.

Another interesting trend we observed while analyzing the init script ecosystem is how vendors customize AOSP specific services by changing or adding a *root* user as the owner of the service. Even though the numbers are not very high, we noticed how, over the years, starting from Android 4.0.3, on average, vendors customize at least one service. This customization is very hard to explain since these core services are supposed to run on the system without modification. Extending the permissions of these services might result in having an unnecessary over-privileged service. One possible cause of this change might be due to aggressive and dangerous customization that requires root privileges to work correctly.

We also cross-referenced the list of new services with the results obtained in the previous subsection, where we found binaries usually used for debugging purposes in commercial ROMs. Surprisingly, we found that some of those binaries are also automatically started at boot. For instance, we identified 18 ROMs (of 2 different vendors) configured to start `tcpdump` at boot and with root privileges. In this case, a manual investigation showed that the `tcpdump` process was configured to monitor incoming packets on all the interfaces and save the first 134 bytes of data from each packet into a log file. To make it worse, some of these ROMs use `tcpdump` version 4.9.2, which is outdated (it was released in 2017) and is affected by several CVEs [tcp20b, tcp20a] some of which with public proof-of-concept exploits. In another case, we found another service executing `tcpdump` – however, not started at boot. This service is named *sniffer*, and it logs the entire network packets that pass on the *wlano* interface in a log file in the */sdcard/* partition—once readable by any application that has granted the `READ_EXTERNAL_STORAGE` permission. Surprisingly, we identified these problems even in a ROM branded as Android One, and built in 2019. Listening and processing packets from untrusted sources expose the device to potential remote and local attackers, thus severely negatively affecting

the entire device's security.

5.6.4 SELinux Customization

As previously presented in Chapter 5.5.2, vendors often customize the SELinux configuration. We now present an in-depth analysis of the most frequent vendor SELinux-related changes and their impact on the overall system's security (independently from whether these changes are compliant with the CDD and other requirements).

SELinux plays a crucial role in the entire security of Android, and this component can also be used to introduce temporary patches to mitigate a vulnerability introduced at the software level. For example, the *vold* privilege escalation bug was properly mitigated first by a SELinux rule, before the vulnerable daemon `vold` was patched [MTC+18, vol14a]. SELinux can be at the same time a solid defense barrier if properly configured, but it can also be the direct cause for a complete compromise of the device, if not correctly configured. Unfortunately, there might be cases in which vendors modify these policies without verifying whether the change can introduce new vulnerabilities (or make existing vulnerabilities exploitable). This is the case for Motorola that, by just introducing *one single* policy change for some of its devices, has introduced a logical bug that reverted the patch introduced to mitigate the problem on vold, allowing attackers to get root [vol14b]. In other cases, the vulnerability can also be part of the base AOSP policy defined by Google. This is the case for *CVE-2018-9488* in which one of the default SELinux domain of AOSP was wrongly configured and allowed a local attacker to perform a privilege escalation [cra18]. Given this component's importance, any modification should be tested to ensure that it does not introduce logical issues and vulnerabilities and does not enlarge the attachment surface.

These examples show that defining SELinux policies is a delicate and error-prone process. However, SELinux changes are necessary for the vendors. Every new process, file, and resource must be correctly labeled, and each new change introduced by a vendor must be configured accordingly. This means that each new component introduced by the vendor potentially requires introducing new domains, types, classes, and rules.

In our analysis, we extracted and analyzed all vendor rules that were not present in the corresponding basic AOSP policy. We identify three different cases:

- rules that modify a pre-existing rule to extend the permissions and

operations allowed on a given resource;

- rules that modify an exiting core policy domain but just by extending it to support new resources introduced by the vendor;
- rules that are completely new and that operate on domains and resources that are not present in the original AOSP policy.

Since our dataset consists of numerous ROMs distributed over the years, this analysis also allows us to understand how, over time, vendors have modified their SELinux policies.

We now present the results of our analysis. Figure 5.5 shows the changes to the *allowrule* defined by a vendor for its system, while Figure 5.6 shows the changes affecting the definition of domains, types, and classes.

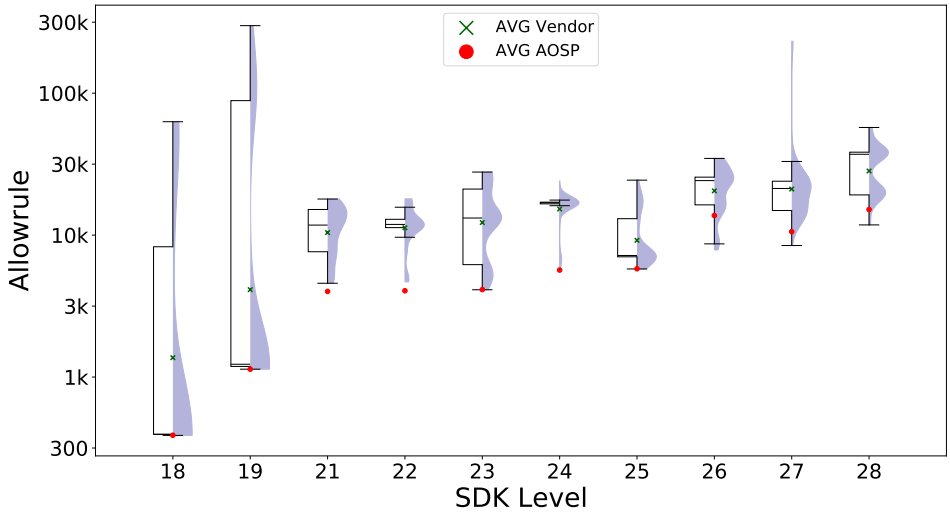


Figure 5.5: Distribution of SELinux rules in the policies.

For each SDK, the first figure shows the distribution of the number of SELinux rules present in the policy. The graph combines a traditional boxplot on the left, showing the first and third quartiles, with a violin plot on the right side to show the distribution of the number of ROMs that define a given number of rules. The plot also includes two dots to indicate the average number of rules present in the correspondent AOSP policy compared to the average number of rules found in the different vendor policies. Also note that to accommodate outliers better, the Y-axis is plotted on a logarithmic scale.

We noticed how these outliers aggressively modified the default policy defined in AOSP to add a significant number of rules. For example, for SDK 27, an AOSP policy contained in average 10,000 rules, but some vendors defined a policy containing more than 232,000 rules (i.e., an increment of over 20 times).

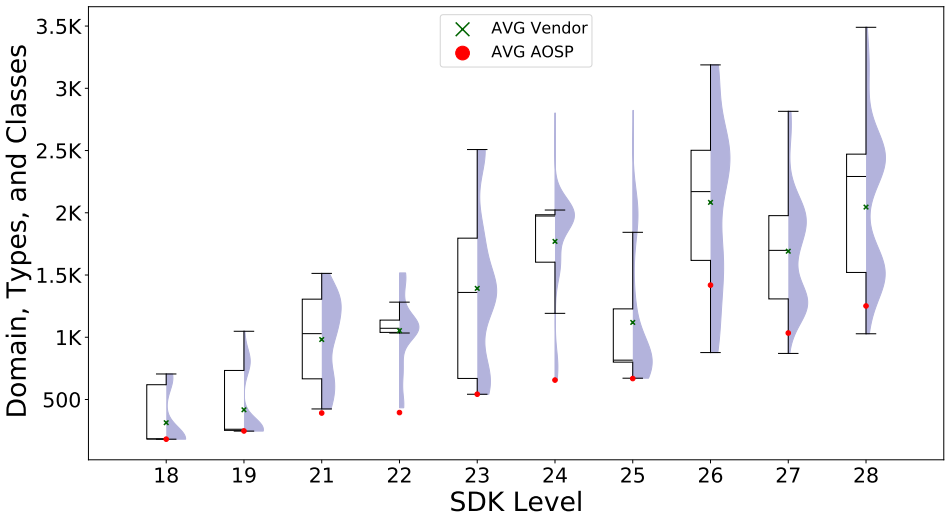


Figure 5.6: Distribution of SELinux domains, types, and classes present in the policies.

A similar trend also appears in the changes to the definition of domains, types, and classes, as presented in Figure 5.6. Although the number of additions for these new components seems to have stabilized in the latest versions of Android, we still observe, in almost all versions, the problem of aggressive vendors introducing an impressive number of new domains, classes, and types. Analyzing the data for SDK level 26, it is possible to observe an interesting peak. It is possible to see how, on average, the basic AOSP policy defines 1418 elements divided between domains, classes, and types. At the same time, some vendors extend this policy more than twice as much by defining 3188 of these elements in their policy.

These results highlight how the problem of customization has significantly affected SELinux policies and how vendors often behave very differently from one another. If we consider that even very restrictive policies with a very limited number of rules, such as those provided by AOSP, have been found to contain problems, it is difficult to foresee vendors' policies that introduce a number of rules 20 times greater than the average can be free

from logical misconfiguration or even from real vulnerabilities introduced by a completely insecure rule.

Figure 5.7 presents a more fine-grained breakdown of the changes. In this case, for each modified rule, we checked if it applies to new domains added by the vendor, if it interests AOSP domains, if it is adding permissions to a previous rule, or if it is a modification that interests sensitive domains. The graph shows that most of the vendors' changes consist of rules that involve new domains that are not present in the original AOSP policy. These new rules are usually introduced by the vendors to configure custom components properly. We also observe a substantial number of changes to rules that affect domains shared with AOSP, but which see the introduction of classes and types that are not present in the basic policy. These numbers exemplify the problem of customization by showing how many changes the vendor makes to the initial policy configuration so that new third-party components can interact correctly with the entire system. But this also shows how intrusive vendor changes are, and how, as noted above, this dangerous trend is continuing to grow.

Figure 5.7: Evolution and Classification of SELinux Modifications.

Although these changes can lead to severe system security issues, as is the case with vold, as it is possible to see, vendors continue to make these dangerous changes. For example, If we analyze the data relating to the SDK level 27, it is possible to see how this type of changes almost reaches the levels of the other customizations on the policy made by the vendors. If we consider that *even a simple change* to the permissions of a rule was enough to reintroduce the bug on *vold*, it is very difficult to think that all these changes are free from possible security problems.

A more important finding emphasized by Figure 5.7 is the number of changes vendors made to the base policy, by extending the permissions and privileges for default AOSP domains. In principle, these rules only affect AOSP components that the vendor should not modify. However, if a vendor applies some customizations to services running on these domains, some of these modifications might raise a runtime SELinux violation since one new feature introduced by the vendor violates a rule defined in the original policy. Furthermore, this trend has seen a significant increase in versions from SDK 25 to 27. For instance, in SDK 27, we found vendors introducing more than 130,000 permission changes to the corresponding AOSP core policy.

Depending on the attack model considered, some SELinux policy changes can be more problematic than others. In particular, two domains, *isolated_app*, and *untrusted_app*, play a particular role in the security of the system, and therefore their basic policy included in AOSP is very restrictive. The *isolated_app* domain is mainly used as an additional sandbox for the Chrome renderer process, or to sandbox processes that should not have permissions of their own. Adding rules to this context could widen the attack surface for remote attacks, potentially allowing an easier sandbox escape. The *untrusted_app* domain is used instead for all third-party applications, and therefore also for potential malware that might be inadvertently installed by the user. Any change to this predefined policy could widen the attack surface for local privilege escalation attacks.

Figure 5.7 shows that vendors modify these domains less often, but the numbers are still not negligible. In fact, we noticed how vendors have drastically reduced the changes to these domains starting from SDK level 24 up to 26. However, this significant decline was followed by a steep rise from vendors in the next release. By looking at the averages of the changes, in fact, we moved from an overall average of 6 customizations for the SDK level 26, followed by an average of 95 changes to `isolated` and `untrusted_app` for the next release of Android.

For each domain, we now present significant dangerous changes made by vendors and discuss their impact on the overall security.

Customization *isolated_app* Domain. We identified a total of 58,776 changes to this context, 1,375 of which are unique. By manually inspecting these modifications, we identified several severe and dangerous cases. For instance, in SDK 19, we found 44 different ROMs that allowed an *isolated_app* process to perform read, write, and ioctl operations directly on kernel drivers. More recently, some devices targeting SDK 23 and 25 changed a rule that allows a process running on this domain to perform open, read, and write operations on application data files. We noticed how this rule is also violating a *neverallow* rule for this process since isolated applications and processes should never directly open application data files themselves.

Customization *untrusted_app* Domain. Across the years, vendors have made 95,577 changes to this context, 4,228 of which are unique. Among these customizations, we found old systems (based on Android 4.4) that allowed an *untrusted_app* process to perform read and ioctl operations directly on kernel drivers. On newer policies, the risks have been reduced by removing the *ioctl* capability, while still allowing the domain to read from kernel components. Another interesting finding, affecting newer devices targeting SDK 27, is related to a rule that allows a process running on this domain to read files containing the device's MAC address. Google is restricting access to this information in many ways, including a `neverallow` rule that prevents this operation [mac15]. Despite this effort, we still find vendors that nullify these defensive measures by allowing, unintentionally, other applications to access this information.

5.7 Related Work

The research we conducted touched on numerous related to the problem of fragmentation on Android. Similar problems have been analyzed by previous works, which however, unlike ours, have focused on a particular aspect of fragmentation. These works can be divided into two groups: the first, related to issues related to code customization by vendors, highlight the perils of Android customizations. The second group of works instead, is related to the analysis and problems found in the years concerning SELinux. We now proceed to present these relevant works in detail.

5.7.1 The Perils of Android Customizations

Among the first works that have analyzed the potential vulnerabilities introduced by vendor code changes are the works of Aafer et al. [AZD16]

and Zhou et al. [ZLZ+14]. In [AZD16] the focus is on the Android security features—permissions, UID/GID, components and their protection—that can cause potential vulnerabilities if wrongly altered by vendor. Their analysis mainly focused on the framework component, analyzing the various modifications between the XML security configurations of the various ROMs, looking for inconsistencies. The results have demonstrated how vendors have introduced severe security issues within their systems by modifying security components within the Android framework with disastrous consequences for user privacy and safety. In fact, among the identified attack types, it was possible to find from "Unauthorized factory reset" to "Stealing of emails." In [ZLZ+14] instead, the authors envisioned "ADDICTED," a dynamic analysis framework with the focus of identifying security issues related to customized kernel device driver introduced by vendors. The framework collects system-call level information to link the device operations protected by Android permissions to their related Linux files. The mapping is collected on both Google phones and custom devices, and differential analysis is then applied to identify potential security misconfiguration. The results of this research are worrisome in that they highlight how the customized drivers are often sources of security problems and how this problem is not so widespread and common in the drivers offered by the official Android platform. Among the various potential attacks identified, it was possible to bypass the Android permissions system and access the camera without the relevant permission, exploiting a misconfiguration of the `/dev/videoo` device driver. Continuing with the works that have analyzed the issue of customizations, we find the work of Tian et al. [THC+18]. The focus of this work touches on another very important component for the Android ecosystem, modems. This paper presents the first comprehensive vulnerability analysis of AT Commands within over 2,000 Android smartphone firmwares across 11 vendors. Among the various attacks that it has been possible to carry out thanks to this research, some turn out to be particularly dangerous for the security and privacy of the end user. In fact, it has been shown that custom and hidden AT commands allowed an attacker to completely rewrite device firmware or perform screen unlocks, thus bypassing all the Android security mechanisms in place trying to prevent such actions. We believe this research has pointed the lights on an aspect not so much analyzed to date, the security of modems and how they interface with the system. Another component of the Android ecosystem strongly affected by customizations are the applications, with particular attention to those that are pre-installed on the device, since normally these cannot be uninstalled. Wu et al. [WGZ+13] and Gamba et al. [GRR+20] illustrate and

analyze this issue. Their analysis show how vendors' customizations and third-party services—introduced by one of the players in the entire supply chain—introduce several issues for what concerns the security of the system as well as the user privacy. Due to the large number of involved parties that, on a single device, introduce and modify code, and due to the lack of transparency of this entire process, several potentially harmful behaviors have been facilitated. This research confirms, as we have shown, that the problem of fragmentation is still open and, to date, far from being solved. However, these issues are not only caused by code introduced by vendors, but also by code that is *not* introduced. In fact, a problem that has plagued vendors a lot is to bring security patches on a system that many times, differs a lot from the original one. Dai et al. [DZJ+20] indeed, illustrate how the Android customizations on the framework might be a direct cause of the *patch gapping*, showing how vendors fail to roll out all of the security patches published by Google in a reasonable time. On top of [DZJ+20], Daniel et al. [TBR15] suggest that another reason updates are not provided in a reasonable time may be due to the large number of entities involved in the supply chain that have to cooperate for the patch to reach the device.

5.7.2 SELinux Policy Analysis

As well as the perils introduced by code-level customizations, also SELinux has been analyzed over the years by several works. Reshetova et al. envisioned two analysis tools that helps OEMs overcome common challenges and avoid mistakes when writing SELinux policies. The first one, SELint [RBA17]: the idea and the goal behind this tool is to make policy writing accessible to non-expert developers, thus allowing the developer to create a policy easy to configure and extend. To allow the developer to write these rules, SELint starts with a solid base configuration provided by an experienced developer, and then, with a series of plugins, allows the less experienced developer to extend it. To avoid introducing policies and rules that can introduce security issues, SELint models risky patterns (e.g., rules working on `untrusted_app` or `tee` domains) that are verified before the final policy is put into production The second work instead, SEAL [RBN+16], can be used by policy writers to improve the quality and the security of the final policies. This framework performs a differential analysis between numerous policies from different vendors, identifying problematic patterns. These patterns and issues can be used as a knowledge base to avoid their introduction into future policies. Among the various problems identified, it is possible to find how vendors overuse Default Types, Predefined Domains, and how vendors forget useless rules in the final policy. Another relevant

work in the field is EASEAndroid [WER+15], by Wang et al.. EASEAndroid is envisioned as an analytic platform for automatic SELinux policy analysis and automated refinement. This refinement process is automated using semi-supervised learning. The system analyzes SELinux audit logs—the logs produced when a rule is violated—from devices, in order to identify correct and proper "denials" (and therefore a potential attack correctly blocked by the policy), as well as incorrect ones (i.e. benign accesses that instead are blocked because of a too strict policy). The idea proposed in this paper is effective and proved applicable to real problems. In fact, the evaluation was performed over more than 1.3 million audit logs from real-world devices, successfully identified 2,518 benign and malicious access patterns, and generated 331 policy rules as a refinement. The point of view of SELinux analysis is changed by Im et al. [ICW18]: in their work in fact, they do not focus on the evolution or issues related to SELinux vendor policies, but rather they focus on the evolution of SELinux within AOSP, proposing a new metric to measure the complexity of a given policy. The policy evolution is performed by analyzing the various git commits made to the official AOSP repository, identifying those related to SELinux and extracting the point changes that were added in a given commit. This analysis takes into account 16,100 commits, between January 2012 and August 2018, and shows how the growing complexity of Android's SELinux will lead to policies that are increasingly difficult to understand and analyze To conclude, the last work we present is BigMAC, by Hernandez et al. [HTY+20]. In [HTY+20], the authors designed and implemented a new SELinux policy analysis framework that works on firmware images. This system, unlike many other proposed approaches, recreates the security state of a running system starting from the static firmware and the Android domain knowledge, thus eliminating the need for a physical device and thus allowing this framework to be executed at scale. The representation of the system security state—and thus the combination of DAC, MAC, etc.— is then embedded in a dataflow graph, which can be queried by the user to identify potential attack paths between processes. The evaluation of BigMAC on firmware images allowed the system to identify known problems like CVE-2018-9488, but also numerous new problems such as untrusted applications able to interact with a kernel monitoring service and several processes configured with `CAP_SYS_ADMIN` capability.

Our work significantly differs and complements all these previous ones, for both the type of analysis performed and the components considered, as it is the first to discuss a longitudinal analysis on OEM customizations, their *compliance* aspect, and details about system binaries, libraries, SELinux

policies, Android init scripts, and user- and kernel-space hardening techniques.

Chapter 6

Conclusion and Future Work

6.1 Future work

As we have seen in this thesis, the Android system has positioned itself, over the years, as a leader in the market of mobile devices, with significant numbers both in terms of devices sold and active users. The number of Android-based devices continues to grow yearly, and has now reached 2,5 billion devices produced by over 1,300 brands. The same trend goes for the evolution of the Google PlayStore, which nowadays allows the 2 billion Android users to choose from more than 3 million applications.

Just by looking at these numbers, it is easy to understand how important and extremely difficult it is to ensure security in such extended ecosystem. Indeed, guaranteeing and ensuring security at every node of the supply-chain is a problem that is far from being solved.

This thesis did not attempt to solve all the security problems that affect the ecosystem, but rather set out to analyze security issues from different perspectives, to provide insights, and to build foundations for further research. We believe that the complexity of the Android ecosystem has led to the insight that issues affecting individual layers cannot be dealt with in a generic way: Each of the players in the supply-chain has its own incentives and views on security, which are influenced by several factors that should be taken into account when studying and researching a specific issue.

This work has shown that the direction that the Android ecosystem is taking in approaching its security is very encouraging, but that, as one may expect, it cannot be considered a problem solved. In fact, there is still a lot of work to be done to allow all the components involved in the supply-chain to be able to use the latest security measures without repercussions, and to allow them to view security as an integral part of their final product.

The research we presented followed an analysis divided into layers. For each of these, we analyzed a specific issue that characterizes and affects it, and we studied the limitations and issues that allow this problem to be, still today, significant for the entire ecosystem.

Starting from the first layer we analyzed, the Android Applications, we believe Network Security Policy is a great step forward to ensure safety in the network communications. However, as we discussed in Chapter 3, as long as advertisements remain a major source of revenue for developers and as long as their ecosystem will not allow for easy adoption of HTTPS, we believe that research should focus on an even more aggressive solution.

We believe that a further approach to this problem, which would improve the current state of security even more, would be to study the feasibility of isolating and executing, within a sandboxed process (such as

`isolated_process`), all code retrieved unsafely using HTTP. Taking advantage of the Network Security Policy configuration, which already allows to specify the endpoints that should be reachable in cleartext, and by combining our solution to specify also the component of the application that should be able to open unsecured connections, it would be possible to identify the code that needs to be isolated and executed in a restricted environment. This approach can, of course, be extended to all those components that force the developer to use insecure network configurations. As highlighted in Chapter 3, several previous works already envisioned an isolated execution of advertisement libraries from the main application. We therefore believe that research should aim at this direction to have even more usable solutions for the developer so that she no longer has to perceive security as a compromise.

Network Security Policy, as we have shown, also suffers from a significant problem related to misconfiguration. Thus, another complementary direction that might help addressing this issue involves the study of new verification methods that can help assuring that a policy is indeed correct, that it reflects the developer's intent, and that it is aligned with the application code.

We believe that research should move in the direction of creating a verification system that, given a policy and an application, can provide the developer with information regarding the correctness of the policy, whether it should be extended or not, or whether it honors the principle of least privilege.

However, while we believe this approach would be useful, it is very challenging to make it successful in the general case. For example, if the application uses external libraries that dynamically loads content from unknown endpoints, the system would not know the resources that would be loaded at runtime and thus could not provide a completely accurate analysis.

Bringing the research towards the development of techniques and methodologies of automatic testing leads to numerous advantages, among which certainly the possibility of reducing the time in which problems are identified, and the possibility of scaling up the testing phase. Indeed, this is what we attempted to accomplish with the automatic identification of vulnerable APIs, presented in Chapter 4, and which allowed us to find as many as 18 new vulnerable APIs that could be used by attackers to mount state-inference attacks. The large amount of vulnerable APIs we found raises an important question, which is whether AOSP has automatic tests aimed at detecting these type of issues. Even though our work was performed primarily on the Android Open Source Project codebase, we do realize how it can

be improved and extended. We believe that the addition of code coverage, and a more detailed and principled approach to the automatic argument generation, could stimulate new execution paths and, potentially, help our system to identify even more vulnerabilities.

As we discussed in this thesis, limiting ourselves to analyzing only the code present on AOSP would give us a distorted and limited view of the reality of the entire ecosystem. Since only few customizations of the Android system are released publicly by vendors, this makes it necessary to have more approaches based on the analysis of compiled code. Therefore, we believe that to make our system usable in large scale and to allow problems to be found even in vendor customizations, there is a need to shift it from source code to bytecode. However, moving this analysis framework from an environment where we can collect a lot of information just from source code, to a closed source environment, hides many challenges and risks making the analysis less accurate and effective.

We believe, however, that the information that is lost by analyzing the bytecode can be recovered or at least inferred. In fact, attempting an approach in which Natural Language Processing techniques are applied to the AOSP source code and its documentation may help to understand, given the name of an API never seen within the codebase, the task and functionalities of the given API, and also infer what, if any, arguments it expects, and its return value. The NLP approach was already heavily used in the realm of Android security, to infer and automatically extract from different sources of text, like applications reviews, or source code, like information used to perform security analysis. Improving our model with the addition of these new techniques and features could make our research more generic and applicable to more Android implementations.

To conclude, we believe that the differential analysis between AOSP and vendor implementations of the system can be applied to numerous other components and can open several new research directions. Indeed, the wide diffusion of the Android system even on devices other than smartphones, such as Android-based televisions, automotives, and watches, has made it possible that the Compatibility Definition Document started to provide security requirements also for such devices. The research we presented in Chapter 5 covers only one of the many layers the ecosystem is constituted of, but it can be easily extended to analyze also several other aspects. In fact, by considering the expansion of these new devices, we believe that research should also move towards this direction by analyzing the security and privacy issues of these new ecosystems.

During our analysis of vendor-modified binaries and libraries, we noticed

how we were unable to identify and quantify the security repercussions of the changes made to the code, or changes to configurations at build time. We believe that differential analysis can help in this situation as well. For example, we envision a new research direction that could help to measure and analyze code introduced by vendors to assess its security "automatically," through software testing techniques like fuzzing and differential testing.

For example, starting from two different implementations of a given library, from two different vendors, we can analyze several aspects related to the customizations: for instance, we can analyze if the same input behaves differently on the libraries, and if so, the reason why this happens. The same idea and approach can be applied to an input that results in a crash only on one library but not on another: the analysis can identify whether the different behavior is due to a modification made by the vendor, a lack of a patch, or other reasons.

In a similar vein, the same analysis could highlight and identify whether a change to the compiler configuration flags introduces issues that would have been identified and stopped at runtime. In this case, we would therefore be interested in testing two implementations of the same library, where the only changes made are at compile time. By using automated testing techniques, we could identify scenarios where removing a certain flag from the compiler, like `FORTIFY_SOURCE`, reintroduces exploitable bug that were prevented by the compiler or the runtime environment. We therefore believe that research should focus more deeply on understanding and testing the possible repercussions of code modified or added by vendors. Moreover, this effort must see the development of automated or semi-automated solutions for this problem: the large number of devices to be tested, and the continuous evolution of the ecosystem, in fact, makes a potential manual approach not very scalable and effective.

6.2 Conclusion

The various projects that we tackled and discussed in this thesis allowed us to analyze numerous issues related to the security of the Android ecosystem. This research allowed us to understand, by constantly changing the point of view for the analysis, how security is not always considered as an integral piece of a system. Ensuring the security of the entire ecosystem becomes a very challenging task: different actors at play, with different requirements and security constraints, make a generic approach to security ineffective. This, in turn, pushed us to change perspective when analyzing the problems of a given player of the ecosystem.

This study allowed us to analyze and measure several security issues that have affected the Android system for years.

In Chapter 3, we start with the analysis of the first software level we identified in the ecosystem, the Application layer. We discussed how the problem related to network communication security still affects this layer, potentially having repercussions on millions of users. In this chapter we presented the first large-scale analysis of Network Security Policies, and we systematically explored the adoption of this new defense mechanism amongst Android applications. Our analysis showed that despite Google's effort to bring HTTPS everywhere, we are not quite there yet. We highlighted, in fact, how developers are still allowing full cleartext on their application, despite the fact that the Network Security Policy could fix and eradicate this issue once and for all. An in-depth study of this layer, and the analysis of some key components in the application developer ecosystem such as external libraries, allowed us to identify some conceptual limitations to this defense mechanism: thus, we designed and implemented a drop-in extension on the actual Network Security Policy, which allows developers to address these limitations.

Then, in Chapter 4, we continued our journey with the analysis of the System layer, and we saw how a more aggressive approach to security, during the years, made it increasingly difficult for malicious applications to mount some specific attacks. Our study focuses on one specific threat, posed by phishing. Despite the general advancement of Android security, nevertheless, this issue still creates numerous problems and endangers the security of many users. Therefore, we decided to analyze this interesting problem with the goal of raising the bar for attackers.

Given the magnitude and the complexity of this problem, we decided to approach it by using two complementary research directions. The continuous evolution of the Android system, and its constant modification of the

codebase to add functionality, do not allow a manual approach when dealing with the identification of vulnerable APIs, and so, we envisioned and developed an automatic approach to test the security of the system APIs.

Thus, we systematically studied and uncovered the attack surface that is actually available to attackers, and we designed an automatic framework that discovered 18 new vulnerable APIs, affecting both Android 8.1 and 9, and that allow a malicious application to mount timely phishing attacks. Nevertheless, the identification of these APIs is only the first step: in fact, no automatic vulnerability system can always guarantee to find all possible vulnerabilities (except in some very specific scenarios), and even missing one vulnerable API may be problematic. Therefore, we implemented a new on-device detection mechanism based that blocks state inference attacks at their root at the moment they occur, even when exploiting unknown vulnerable APIs.

We concluded this thesis by analyzing the security issues from the perspective of the last layer we identified, the Vendors layer. This layer is the one that most of all represents and encloses the problem of fragmentation, and describes the diversity and complexity of the Android ecosystem. The analysis we presented in Chapter 5 is the first longitudinal and large-scale study that tries to analyze security issues due to customizations in important system components such as SELinux, system binaries, init scripts, and the Android Linux kernel.

We observed that, over the years, numerous vendors have failed to comply with Google's defined security requirements, releasing systems that lacked basic protection systems. We conclude by highlighting how there are several areas of customizations that, even if they do not violate any strict security requirement, are the root cause for severe security problems and have serious repercussions on the overall security posture of the final system. These results demonstrate that the current set of regulations and checks, although they represent the first step and barrier to mitigate and limit these issues, are not enough, and that more aggressive controls should be in place to ensure end-user security.

I believe this thesis has advanced the security of the Android ecosystem and that it provided useful insights to accelerate the adoption of HTTPS everywhere on Android applications, that has made it possible to move forward with Google's effort to eradicate the phishing problem, and I hope that it will inspire future works and analyses in the research areas that this thesis focused on.

Appendices

Appendix A

French Summary

A.1 Introduction

L'ère dans laquelle nous vivons aujourd'hui voit la mobilité et la possibilité d'être toujours connecté comme des éléments clés de la vie quotidienne. Des études récentes ont montré que, d'ici 2020, près de 3 milliards de personnes posséderont un smartphone, et comment, au cours des cinq dernières années, le trafic créé par les appareils mobiles a connu une croissance incroyable de 700%. Malgré le grand nombre de smartphones et d'utilisateurs, si nous examinons le choix des systèmes d'exploitation pour ces appareils, nous constatons que le marché est principalement divisé entre deux systèmes seulement : Les appareils Android couvrent près de 70% des parts de marché, et les appareils iOS les 30% restants. Compte tenu du pourcentage plus important du marché couvert par les appareils basés sur Android, qui ne se limitent pas aux smartphones mais incluent également les téléviseurs intelligents, les smartwatches et les appareils automobiles, nous étudions dans cette thèse la sécurité de l'écosystème Android.

Le succès et la diffusion d'Android sont dus à de nombreux facteurs. Tout d'abord, on observe que la nature ouverte du Android Open Source Projet a permis à de nombreux fournisseurs, tels que Samsung, Xiaomi, etc., de développer leurs propres produits et de les baser sur ce système. Cela a également conduit à un plus grand choix pour l'utilisateur en termes d'appareils à acheter. Deuxièmement, il existe un large éventail d'applications dont les utilisateurs peuvent bénéficier, ce qui leur permet d'effectuer des tâches qui étaient auparavant réalisées sur des ordinateurs personnels, mais qui le sont désormais aussi sur des smartphones. La grande diversité en termes de smartphones, de systèmes basés sur Android et d'applications fait que la gestion de la sécurité de cet écosystème est aussi délicate que difficile, et nécessite avant tout une analyse minutieuse prenant en compte tous les points de vue des acteurs qui y contribuent. Une autre raison pour laquelle une analyse approfondie de la sécurité de cet écosystème est essentielle est que la migration de la base d'utilisateurs des ordinateurs personnels vers les smartphones a également attiré l'attention des attaquants et des cybercriminels, qui ont commencé à étudier soigneusement de nouvelles stratégies pour compromettre ces systèmes.

Au fil des ans, presque chaque nouvelle version d'Android a introduit de nombreuses nouvelles fonctionnalités, dont beaucoup concernent la sécurité de la plateforme. Cette amélioration continue a mis la barre plus haut pour les attaquants, mais cela ne les a pas empêchés d'étudier de nouvelles techniques plus avancées pour compromettre un appareil. En outre, outre les efforts déployés par le projet AOSP, de nombreux autres projets de

recherche universitaires et industriels ont contribué à améliorer la sécurité des appareils Android. Cependant, en dépit de ces efforts collectifs, certains problèmes importants restent non résolus. C'est la complexité et le défi de ces questions qui m'ont poussé à axer ma thèse sur certains de ces problèmes qui, bien qu'ayant été étudiés pendant de nombreuses années, sont toujours ouverts aujourd'hui.

Dans cette thèse, nous présentons une étude de sécurité selon trois "points de vue" différents, et nous analysons les trois principales couches qui contribuent au développement de cet écosystème. Nous avons identifié ces trois couches comme étant l'Application, le Système, et la couche Fabricant.

Chacune de ces couches aborde et voit la sécurité d'une manière unique, qui est très souvent différente de celle adoptée par les autres composants. En fait, chacune de ces couches peut avoir des objectifs, des modèles de menace, des exigences et des contraintes différents et uniques en matière de sécurité. Il est donc essentiel de comprendre les problèmes rencontrés par chacun de ces niveaux afin de concevoir et de proposer des mesures de sécurité qui répondent à leurs exigences, et la seule façon de le faire est d'examiner le problème à travers leur perspective.

Cependant, tenter de résoudre tous les problèmes affectant les différentes couches, aussi souhaitable que cela puisse être, est très difficile à réaliser. C'est pourquoi, pour chacune de ces couches, nous avons décidé d'examiner de manière détaillée et fondée sur des principes une question spécifique et importante. Déterminer les questions à aborder n'a pas été facile, nous avons donc essayé de classer par ordre de priorité celles qui nous semblaient les plus importantes. Pour nous aider dans ce choix, nous avons essayé de répondre à certaines questions qui ont guidé notre décision finale. Par exemple, nous avons essayé de comprendre et de quantifier le nombre de smartphones, d'applications ou d'utilisateurs qui seraient affectés et exposés à une menace spécifique. Dans ce cas, nous avons toujours essayé de donner la priorité aux problèmes affectant le plus grand nombre d'utilisateurs ou de smartphones. Ou bien, nous nous sommes demandé s'il existait déjà des mesures pour prévenir ce problème, et si oui, pourquoi elles n'étaient pas utilisées. Une fois encore, nous avons mis l'accent sur les problèmes pour lesquels, au fil des ans, des mesures de sécurité ont été envisagées—soit pour prévenir un problème particulier, soit pour l'identifier—mais qui, pour une raison ou une autre, ne sont pas totalement efficaces. En fait, pour concevoir une mesure de sécurité efficace et efficiente, il est d'abord important de comprendre pourquoi, malgré la disponibilité d'outils qui aideraient les développeurs à prévenir les erreurs et les problèmes, ceux-ci ne sont pas adoptés.

Cette étude préliminaire nous a conduit à investiguer et à concentrer notre attention sur les problématiques suivantes : les menaces derrière l'utilisation de connexions non sécurisées pour la couche Application, celle concernant le phishing pour la couche Système, et enfin celle liée aux répercussions de la fragmentation sur la sécurité pour la couche Fabricant. Les recherches que nous avons menées dans cette thèse ne se limitent cependant pas à mesurer simplement l'ampleur de ces problèmes, mais s'appuient sur ces résultats pour proposer de nouvelles mesures de renforcement de la sécurité qui pourraient aider à résoudre les problèmes que nous avons explorés.

Nous espérons que les recherches présentées dans cette thèse et ses contributions pourront servir d'étape vers un écosystème mobile plus sûr, et qu'elles pourront servir de point de départ et de base pour d'autres recherches dans ce domaine important.

Cette thèse est structurée comme suit. Le chapitre 2 sert de contexte et présente notre vision de l'écosystème Android en couches. Chacune des couches est expliquée en détail, soulignant des aspects importants tels que le rôle d'une couche donnée au sein de l'écosystème, les questions qui l'ont affectée au fil des ans, et quels sont les problèmes en matière de sécurité. Ce chapitre pose les bases et présente les différentes approches de la sécurité, en expliquant quelles peuvent être leurs limites.

La première couche que nous analysons concerne les applications : dans le chapitre 3, nous présentons la première étude sur la Network Security Policy, un nouveau mécanisme de protection qui permet aux développeurs de configurer la sécurité des connexions réseau de leurs applications sans avoir à introduire de nouveau code. Cette étude nous a permis d'identifier non seulement les erreurs potentielles que les développeurs peuvent commettre et qui rendraient l'utilisation de cette politique inefficace et inutile, mais aussi les limitations de conception qui empêchent son utilisation correcte. Ce chapitre se termine par notre version de la Network Security Policy qui tient compte de ces limitations et permettrait aux développeurs d'adopter plus largement ce mécanisme de défense.

Dans le chapitre 4, nous étudions le problème du phishing, qui touche le système Android dès les premières versions. Pour aider à résoudre cette menace, nous avons décidé de procéder dans deux directions complémentaires. La première consiste à identifier automatiquement les vulnérabilités dans le code source du système d'exploitation qui sont normalement exploitées par les logiciels malveillants, afin de les identifier et de les corriger avant que le code ne soit utilisé par l'utilisateur final. Ce système nous a permis d'identifier de nombreuses vulnérabilités au sein de différentes ver-

sions d'Android. La seconde, compte tenu du temps nécessaire pour patcher un éventuel bug utilisé par les applications malveillantes pour monter le phishing, est de développer un mécanisme de détection pour identifier ces attaques au moment où elles se produisent, pour les bloquer au bon moment et ne pas les laisser compromettre la sécurité d'une application.

Le chapitre 5 introduit la dernière problématique examinée dans cette thèse. Le problème analysé est celui lié aux répercussions, en termes de sécurité, des personnalisations apportées par les fabricants au système AOSP. Ce phénomène, connu sous le nom de "fragmentation," risque de rendre inutiles tous les efforts déployés au fil des ans pour assurer un système plus sûr. Les contributions que nous apportons dans ce chapitre montrent que depuis les premières versions d'Android, les modifications apportées par les fabricants ont très souvent un impact négatif sur la sécurité globale du système, et comment les composants spécifiques aux fabricants sont significativement en retard par rapport à la sécurité de la version Open Source de Android.

Enfin, le parcours de cette thèse, qui visait à montrer comment, dans l'écosystème Android, la façon dont la sécurité est perçue et abordée change en fonction de la couche analysée, s'achève avec le chapitre 6, où nous résumons brièvement les défis auxquels nous avons été confrontés, la façon dont nous les avons résolus, et où nous fournissons enfin quelques éléments qui, nous l'espérons, inspireront les recherches futures.

A.2 Sécuriser la couche d'Application

Ce chapitre est basé sur la publication *"Towards HTTPS Everywhere on Android: We Are Not There Yet"* [PF20].

Dans le domaine de l'écosystème Android, les attaques réseau sont un problème qui, encore aujourd'hui, compromet la sécurité des applications. Ce problème touche à la fois les applications qui utilisent des protocoles réseau en clair, tels que HTTP, mais aussi celles qui configurent mal les connexions sécurisées et cryptées qui utilisent SSL/TLS. La résolution de ces problèmes devient critique car, de nos jours, pratiquement toutes les applications mobiles reposent sur la communication avec un backend réseau. Ces dernières années, Google a développé et introduit dans le projet Open Source Android plusieurs mécanismes de sécurité pour protéger la communication réseau des applications Android, allant de plusieurs KeyStores à la récente introduction de la nouvelle Network Security Policy, un fichier de configuration basé sur XML qui permet aux applications de définir leurs configurations de sécurité réseau. Cette nouvelle politique n'a cependant jamais

été analysée en détail et nous pensons qu'elle n'a pas reçu une attention suffisante. Pour bien comprendre les choix qui ont conduit au développement de ce nouveau mécanisme de défense, il est nécessaire d'étudier comment, au fil des ans, les attaques réseau ont permis aux attaquants de compromettre la sécurité des applications et la confidentialité des données des utilisateurs. Ainsi, dans ce travail, dans un premier temps, nous avons mené une étude systématique et complète sur les mécanismes de défense réseau adoptés dans Android au cours des années, nous discutons des attaques contre lesquelles ils se défendent, des pièges potentiels et des modèles de menaces pertinents. Cette étude nous permet d'introduire la Network Security Policy et de l'analyser en connaissant les problèmes qui, au fil des ans, ont conduit les développeurs à utiliser des configurations réseau incorrectes ou non sécurisées. L'analyse de cette nouvelle politique nous a permis d'identifier ses points forts, mais aussi de découvrir certaines configurations qui, bien que considérées comme valides pour la politique, exposent toujours l'application à certaines attaques, risquant ainsi d'introduire un faux sentiment de sécurité pour le développeur. Pour vérifier que ces problèmes ne sont pas seulement théoriques, mais qu'ils sont aussi présents dans des applications réelles, nous avons mesuré ce phénomène par une analyse à grande échelle sur plus de 125K applications, montrant que ces problèmes sont malheureusement très fréquents. Comprendre pourquoi les développeurs utilisent ces configurations est d'une importance énorme pour apporter des améliorations à cette nouvelle politique de défense efficace. Grâce à l'analyse de ces applications, nous avons pu identifier plusieurs problèmes qui limitent le potentiel de cette politique, et nous avons proposé une nouvelle extension de la Network Security Policy qui est entièrement compatible, mais qui cherche à éliminer ces limitations.

A.2.1 Network Security Policy: Les Faiblesses

L'analyse commence par une étude approfondie de toutes les mesures de sécurité qui ont tenté d'atténuer certaines catégories d'attaques réseau au fil des ans. La conception de la politique de sécurité du réseau a bénéficié de toutes ces recherches passées, et a permis la création d'un système efficace permettant au développeur de configurer une application de manière sécurisée sans introduire de code dans l'application. Avec cette nouvelle politique, il est possible de spécifier les configurations suivantes, mais aussi d'introduire des problèmes qui, à notre connaissance, n'avaient jamais été considérés et analysés avant ce travail.

Texte clair et Man-In-The-Middle: La politique permet au développeur

de spécifier les domaines qui seront contactés par des protocoles clairs (HTTP, SMTP, etc.), et ceux qui *ne le seront pas.* Comme il est parfois difficile de dresser une liste exhaustive des domaines qui doivent adhérer à une règle donnée, la politique autorise des caractères de remplacement qui permettent au développeur de spécifier des configurations tout ou rien, ou des domaines qui respectent une expression régulière donnée. Cependant, un développeur peut configurer son application avec une politique qui permet à tous les domaines d'être contactés par des connexions non sécurisées, exposant ainsi l'ensemble de l'application à des attaques de type Man-In-The-Middle. Pour aggraver les choses, comme nous le verrons plus loin, plusieurs ressources en ligne suggèrent de mettre en œuvre cette politique très grossière et dangereuse.

Autorités de certification et Man-In-The-Middle sur HTTPS: Avec la Network Security Policy, il est possible de spécifier les autorités de certification (AC) auxquelles il faut faire confiance lors des connexions sécurisées. Il est possible de faire confiance à un ensemble de clés publiques codées en dur des AC les plus importantes (*system*), ou à des AC personnalisées qui sont installées dans le KeyStore (*user*), ou intégrées dans l'application. Cependant, le protocole HTTPS ne garantit pas toujours que la communication ne peut pas être écoutée. En fait, il est possible de configurer la politique pour faire confiance à l'union des autorités de certification dans le KeyStore du système et de l'utilisateur : ainsi, le trafic de l'application peut être écouté par quiconque contrôle une autorité de certification personnalisée dans l'un des KeyStores. Nous pensons qu'une "application de production" qui fait effectivement confiance au certificat de l'utilisateur est souvent le symptôme d'une mauvaise configuration.

L'épinglage des Certificats et sa Neutralisation: L'épinglage des certificats consiste à "coder en dur" (ou épingler) quel est le ou les certificats attendus lors d'une poignée de main TLS avec un serveur donné. Avec cet attribut, le développeur peut définir tous les paramètres de vérification sans avoir à se soucier de l'implémentation sous-jacente. Cependant, nous avons réalisé qu'il est possible de configurer la politique de manière à définir l'épinglage des certificats pour certains domaines, tout en informant le système que l'épinglage ne doit pas être appliqué. En pratique, nous avons constaté qu'il est possible d'activer et de désactiver ce mécanisme en même temps, et que les deux configurations ne s'excluent pas mutuellement. Nous pensons que ce type de politique offre un "faux sentiment" de sécurité pour un développeur, d'autant plus qu'aucun avertissement n'est émis à la compilation ou à l'exécution.

Malgré certaines limites et certains pièges potentiels, la Network Security Policy rend incontestablement plus pratique la spécification d'une politique de réseau à grain fin. Malheureusement, il n'existe à ce jour aucun outil permettant aux développeurs de vérifier l'exactitude de la politique définie et de s'assurer que les paramètres qu'ils souhaitaient mettre en œuvre sont effectivement ceux appliqués par le système.

A.2.2 Network Security Policy: L'adoption

L'identification de ces erreurs potentielles qui peuvent être introduites lors de la configuration d'une politique, nous a conduit à vérifier combien d'entre elles étaient effectivement présentes dans les applications du Google Play Store officiel. L'analyse a été effectuée sur un jeu de données de 125,419 applications. La même analyse a été répétée en retéléchargeant le jeu de données après un an, afin de mesurer et de quantifier une éventuelle évolution. Nous avons constaté que, bien que la Network Security Policy permette d'améliorer considérablement la sécurité des configurations du réseau, elle a été adoptée par moins de 7% (8,727) des applications lors de la première mesure. Un an plus tard, nous avons remarqué que le nombre d'applications utilisant la politique a presque doublé (15,492). C'est un signe important qui montre que les applications sont en train d'adopter la Network Security Policy, mais en même temps, cela nous montre que sa diffusion n'est pas aussi étendue que l'on pouvait l'espérer. En ce qui concerne le type de configurations utilisées et les problèmes rencontrés, nous avons remarqué que plus de 70% des applications définissant une politique permettent encore l'utilisation de protocoles en clair. Ce qui est surprenant, cependant, c'est le très petit nombre d'applications qui utilisent la politique pour mettre en œuvre l'épinglage des certificats. Sur les 102 applications appliquant l'épinglage des certificats, nous en avons identifié 9 qui mettent en œuvre l'épinglage mais l'annulent par erreur.

A.2.3 Network Security Policy: Les Limites

Certaines des politiques analysées ci-dessus ont attiré notre attention en raison du type de configuration particulière utilisée, mais aussi en raison du nombre de politiques identiques partagées entre plusieurs applications dans l'ensemble de données. Une analyse approfondie nous a permis d'identifier deux problèmes principaux. Le premier, déjà connu depuis longtemps, est celui qui pousse les développeurs à copier du code depuis l'internet sans vraiment comprendre le résultat final. C'est une des raisons qui peut justifier les mêmes politiques partagées entre différentes applications. La seconde, qui à

notre connaissance n'a jamais été analysée, est liée aux composants tiers (par exemple, les bibliothèques de publicité) qui "forcent" les développeurs qui veulent les utiliser à utiliser certains paramètres dans leur propre politique. Cette idée nous a conduit à mesurer l'impact des bibliothèques de publicité sur les politiques de réseau des applications. L'analyse, menée sur les 29 bibliothèques les plus utilisées, nous a permis d'identifier comment 12 d'entre elles poussent le développeur à modifier la politique, et comment 11 d'entre elles le font de manière non sécurisée. Sur ces 12 bibliothèques, 11 exigent du développeur qu'il autorise globalement les connections non sécurisées par l'application, tandis que 2 bibliothèques obligent également le développeur à faire confiance aux AC des utilisateurs, mais aucune d'entre elles ne fournit d'AC, ce qui rend ce risque complètement inutile. Ces résultats nous ont fait réaliser que la granularité offerte par la Network Security Policy n'est pas suffisante pour couvrir ces cas. Guidés par ces résultats, nous avons conçu et implémenté une extension qui permet aux développeurs de spécifier une politique "par paquet." Le développeur peut ainsi continuer à utiliser ces bibliothèques, mais peut les confiner dans un bac à sable, sans risquer de compromettre la stabilité de l'ensemble de la politique d'application.

A.3 Sécuriser la couche Système

Ce chapitre est basé sur la publication *"Preventing and Detecting State Inference Attacks on Android"* [ADY21].

Une menace importante qui affecte le système d'exploitation Android depuis les premières versions est le phishing. L'importance de cette question est également soulignée par une récente recherche menée par Kaspersky Lab, qui a mis en évidence l'augmentation du nombre d'applications malveillantes qui mettent en œuvre cette attaque, telles que les "chevaux de Troie bancaires," en comptant seulement en 2020 un nombre d'installations de ces logiciels malveillants égal à 156,710, soit presque le double du nombre enregistré en 2018. Les attaques par phishing sont particulièrement problématiques pour les plateformes mobiles car elles ne fournissent pas suffisamment d'informations pour qu'un utilisateur puisse distinguer de manière fiable une application légitime d'une application malveillante usurpant l'interface utilisateur de l'application légitime. Un facteur clé qui détermine le taux de réussite d'une attaque de phishing est le bon timing : l'utilisateur est plus enclin à fournir des données sensibles (telles que des mots de passe) si l'interface utilisateur malveillante apparaît au moment où la victime s'attend à interagir avec l'application cible. Sur Android, les

logiciels malveillants déterminent le bon moment en montant des attaques par inférence d'état, qui peuvent être utilisées, par exemple, pour déduire le moment exact où l'utilisateur a lancé une application cible et s'attend donc à interagir avec elle. Même si le bac à sable Android est conçu pour empêcher ces attaques, elles sont toujours possibles en abusant des API vulnérables qui laissent échapper de telles informations sensibles : le scénario habituel est celui d'une application malveillante qui "pollue" ces API vulnérables, déduit quand une application cible est sur le point d'être utilisée par l'utilisateur et fait apparaître l'interface utilisateur usurpée en haut de l'écran au bon moment.

Compte tenu de la dangerosité de ces attaques, nous nous sommes fixés comme objectif de tenter de résoudre ce problème. Dans ces travaux de recherche nous explorons et analysons donc deux directions complémentaires, qui visent à prévenir ces attaques en identifiant les vulnérabilités à l'avance, mais aussi à explorer la possibilité de détecter les attaques au moment où elles sont montées sur le smartphone.

A.3.1 Prévention des attaques par inférence d'état

La première voie que nous avons explorée dans cette recherche est liée à l'identification des vulnérabilités normalement utilisées par les applications malveillantes pour réaliser cette attaque. Compte tenu du temps qu'il faut pour qu'une vulnérabilité soit corrigée, et compte tenu également du grand problème du retard dans la fourniture des correctifs par les fabricants et de la lenteur des utilisateurs à les installer, il est nécessaire d'agir de manière préventive et d'essayer d'identifier ces problèmes avant qu'ils ne soient introduits dans le système final. Ainsi, nous avons imaginé et mis en œuvre un nouveau système de détection des vulnérabilités qui, en combinant des techniques d'analyse statique et dynamique, vise spécifiquement à identifier les nouvelles vulnérabilités qui peuvent être utilisées pour monter des attaques par inférence d'état.

Pour bien comprendre le problème et identifier ses causes profondes, nous avons commencé par une étude systématique des vulnérabilités exploitées par les logiciels malveillants pour monter cette attaque au fil des ans, en identifiant l'un des problèmes fondamentaux et actifs dans la mauvaise mise en œuvre des API du système. Pour tenter de prévenir d'autres vulnérabilités, il est donc nécessaire d'identifier, le cas échéant, d'autres API susceptibles d'être exploitées par les attaquants. Cependant, en raison de la taille du système et du nombre toujours croissant d'API disponibles, une approche manuelle n'est pas recommandée car elle ne serait pas évolutive et prendrait beaucoup de temps : nous avons donc opté pour un système

automatique. Le développement de ce système cache cependant de nombreux défis majeurs. La première concerne la détermination de la surface d'attaque effective disponible pour un attaquant potentiel : en résolvant ce premier problème, nous avons remarqué qu'il existe une couche "cachée" d'API (disponible pour un attaquant) qui n'a jamais été prise en compte et analysée par les travaux précédents sur le phishing. Une fois que la surface d'attaque correcte et les API à analyser ont été identifiées, il est nécessaire d'automatiser ce processus. Cependant, la création automatique d'un objet valide n'est pas si immédiate et cache de nombreux problèmes. Par exemple, même un seul champ d'un objet complexe, s'il n'est pas initialisé correctement, peut conduire à la génération d'exceptions avec le risque de bloquer complètement le processus d'analyse : pour résoudre ce problème, nous avons utilisé, lorsqu'elles étaient disponibles, les informations sémantiques du code source pour guider la génération des arguments.

Dans la continuité, il est nécessaire de simuler autant que possible un système réel qui se rapproche le plus de celui attaqué par les applications malveillantes. Pour cette raison, nous avons automatisé et simulé le comportement naturel de l'utilisateur, en interagissant automatiquement avec le smartphone et en effectuant certaines tâches qui sont normalement réalisées par l'utilisateur. Ce système, après avoir identifié la surface d'attaque, extrait les API à analyser et collecté les informations sémantiques liées aux arguments, procède à un test automatique de chacune des API pendant que le smartphone est stimulé, et collecte et analyse la valeur de retour de chaque invocation à la recherche de tout signal qui pourrait permettre de déduire l'état d'une application.

Nous avons testé notre outil sur 3 versions d'Android (8.1, 9 et 10) et nous avons pu identifier 18 nouvelles vulnérabilités qui pourraient être utilisées par des logiciels malveillants pour monter des attaques de phishing sur ces versions du système. Une analyse plus approfondie de ces vulnérabilités a confirmé que la couche cachée des API que nous avons identifiée en définissant la surface d'attaque correcte à analyser, s'est avérée être la source de nombreuses vulnérabilités. En effet, près de 40% (7) de ces vulnérabilités se situaient précisément dans cette couche, jamais analysée. La dangerosité des API vulnérables que nous avons identifiées réside dans le fait que, pour 6 d'entre elles, le malware n'a pas besoin de demander la moindre permission à l'utilisateur, ce qui permet de monter cette attaque de manière totalement inaperçue.

A.3.2 Détection des attaques par inférence d'état

La deuxième direction que nous avons explorée est la création d'un système de détection à utiliser dans le smartphone pour identifier ces attaques lorsqu'elles se produisent. Nous pensons que l'identification automatique des API qui rendent le système vulnérable aux attaques par inférence d'état est un bon premier pas vers l'éradication de ce problème.

Nous devons cependant reconnaître que le cadre présenté précédemment, basé sur une combinaison de techniques d'analyse statique et dynamique, est malheureusement soumis aux limites intrinsèques de ces techniques. Le risque de ne pas identifier une API comme vulnérable est donc malheureusement présent, et donc être capable de bloquer les attaques devient d'une importance fondamentale pour garantir la sécurité du système et des applications.

Pour protéger les utilisateurs des vulnérabilités inconnues, nous avons étudié la faisabilité d'un composant supplémentaire, qui se veut être un système de défense et de détection à l'exécution pour identifier les attaques par inférence d'état au moment où elles sont exploitées. Le système que nous envisageons est basé sur l'hypothèse clé que de simples "comportements de sondage" peuvent être utilisés comme un signal fort d'une attaque potentielle, indépendamment d'autres facteurs. En fait, tous les logiciels malveillants existants exploitant les attaques par inférence d'état doivent mettre en œuvre des comportements de sondage. Par ce terme, nous faisons référence à une application invoquant plusieurs fois un ensemble d'API dans une courte fenêtre de temps. Les logiciels malveillants qui montent le phishing doivent utiliser le polling pour s'assurer qu'ils peuvent courir après l'application cible et faire apparaître leur interface utilisateur usurpée au bon moment.

Notre étude, réalisée sur un ensemble de données contenant des échantillons de toutes les familles de logiciels malveillants Android découverts au cours des quatre dernières années et connus pour monter des attaques par inférence d'état, a confirmé que tous les logiciels malveillants interrogent avec un délai allant de 600 ms à une seconde, et qu'un logiciel malveillant ne cesse jamais ce comportement une fois qu'il est lancé. Malheureusement, l'analyse du comportement des logiciels malveillants ne suffit pas à garantir que l'interrogation est un indicateur suffisant. Afin de disposer d'un système utilisable, il est nécessaire de vérifier que les applications bénignes ont rarement recours à l'interrogation et que, lorsqu'elles le font, la nature de leurs comportements est différente de celle des logiciels malveillants. Ainsi, nous caractérisons si et comment les applications bénignes ont un comportement de type polling, et s'il existe des caractéristiques qui peuvent être utilisées pour les distinguer des tentatives malveillantes : cette étude

est réalisée sur un ensemble de données de 10,108 applications bénignes. L'ensemble de données a été divisé en un ensemble d'entraînement (20% des applications) et un ensemble de test (les 80% restants). Nous avons exécuté chaque application de l'ensemble d'entraînement pendant cinq minutes en recherchant des comportements d'interrogation à une fréquence deux fois plus faible que le taux de fréquence minimum auquel les logiciels malveillants effectuent des activités d'interrogation. Les résultats montrent, à première vue, qu'un nombre important d'applications qui ont exposé un comportement d'interrogation similaire à celui du logiciel malveillant. Cependant, nous avons remarqué que dans tous ces cas, l'interrogation est effectuée par un autre composant "pour le compte de l'application," c'est-à-dire que même si la logique d'interrogation n'est pas directement implémentée dans l'application, elle est toujours liée au contexte de l'application puisque le composant utilise l'identité de l'application pour les invocations suivantes. Ces cas, qui sont toutefois limités et mis en œuvre dans des sous-systèmes bien définis, tels que ceux liés aux graphiques, ne constituent pas une source de vulnérabilité et, par conséquent, après une analyse minutieuse et manuelle, nous avons décidé de ne pas les considérer comme des "comportements suspects." De plus, nous avons également remarqué que les applications bénignes présentent un pic des activités effectuées au début de l'application, mais que l'interaction avec les services du système diminue avec le temps. Nous notons que cette caractéristique est profondément différente du comportement des logiciels malveillants d'attaque par inférence d'état, c'est-à-dire qu'une fois que le comportement d'interrogation est lancé, il ne s'arrête jamais.

Le système que nous avons utilisé pour surveiller le comportement des applications malveillantes et bénignes, et le système que nous avons utilisé pour identifier les sondages au sein des applications bénignes, ont fourni la base du système de défense que nous avons mis en œuvre sous forme de prototype sur AOSP, qui s'est avéré efficace en termes d'efficacité, en identifiant correctement tous les logiciels malveillants, et de convivialité, car le nombre de faux positifs sur l'ensemble de test était minime (les 0,37% consistant en 30 applications) et l'overhead était négligeable. L'importance de ce système réside dans l'avantage de pouvoir détecter les tentatives d'exploitation même lorsque l'API exploitée n'est pas connue pour être vulnérable à l'avance.

A.4 Sécuriser la couche du Fabricant

Ce chapitre est basé sur la publication *"Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization"* [ASDY21].

Aujourd'hui, plus de deux milliards d'appareils mobiles fonctionnent sous Android OS. Au cœur de ce succès se trouvent la nature open source du Android Open Source Project et la capacité des fabricants à personnaliser la base de code et à l'expédier sur leurs propres appareils. L'ouverture et la flexibilité de l'AOSP ont été un facteur déterminant du grand succès de la plate-forme, qui a été adoptée par un grand nombre de fabricants qui commercialisent des appareils. Cela a donné lieu à une multitude de variantes différentes, un aspect connu sous le nom de "fragmentation."

Si la possibilité de personnalisation est bénéfique pour les fabricants, elle peut potentiellement conduire à des problèmes de compatibilité et de sécurité. En particulier, nous pouvons identifier deux catégories de problèmes de sécurité.

La première est que ces personnalisations peuvent affecter la sécurité du système global (par exemple, en rendant vains les efforts de durcissement de Google), augmenter la surface d'attaque et, dans certains cas, même introduire de nouvelles failles de sécurité. La deuxième catégorie de problèmes peut provenir des composants réels qui sont affectés par les personnalisations des OEM. En effet, les personnalisations qui modifient les composants essentiels du système d'exploitation Android peuvent entraîner des problèmes de compatibilité et des retards dans l'application des correctifs de sécurité, tels que ceux publiés dans le cadre des bulletins de sécurité mensuels d'Android.

Pour éviter ces problèmes, Google a travaillé dans deux directions parallèles et a développé un ensemble d'exigences qui doivent être satisfaites pour qu'un fabricant puisse marquer ses appareils comme "Android." La première est la conformité : alors que l'AOSP est un projet open source et peut donc être librement modifié, un équipementier qui souhaite apposer le label "Android" (qui est une marque de Google) sur ses appareils doit suivre un ensemble de règles bien définies. Le deuxième effort mené par Google pour contrer les répercussions sur la sécurité introduites par les personnalisations des OEM est le projet Treble, une réarchitecture du système d'exploitation Android. Cette réorganisation vise à séparer les composants spécifiques aux fabricants (par exemple, les pilotes pour des jeux de puces spécifiques et d'autres personnalisations) du cadre principal du système d'exploitation Android. La raison d'être de ce changement est de permettre aux OEM d'appliquer plus facilement des correctifs (de sécurité) à leurs AOSP personnalisés. Cependant, ce n'est pas parce qu'il existe des règles définies à suivre et à respecter qu'elles seront effectivement observées. Il est donc nécessaire de vérifier que les équipementiers respectent effectivement cet ensemble de règles imposées par Google.

D'un point de vue général, notre analyse se concentre sur deux aspects clés : le premier consiste à déterminer si un OEM donné respecte les diverses réglementations imposées aux appareils Android (par exemple, CDD, CTS, VTS), et le second à déterminer si et comment les diverses personnalisations de l'OEM affectent la posture de sécurité de l'ensemble de l'OEM. Pour étudier ces deux aspects, nous avons mené une analyse longitudinale et à grande échelle sur un ensemble de données de 2,907 ROMs publiées entre 2009 et 2020 et couvrant les ROMs de la version 2.3 à la version 9.0 d'Android, provenant de 42 OEM différents. Le pipeline d'analyse que nous avons développé nous permet de comparer une ROM modifiée avec la version AOSP de départ. Ce cadre nous permet donc d'effectuer de nombreuses analyses différentielles sur des composants communs et modifiés, mais aussi d'identifier et d'effectuer une analyse plus approfondie uniquement sur les composants insérés éventuellement par un fabricant.

A.4.1 Conformité : Analyse et résultats

Pour être compatible avec Android, un appareil doit répondre aux exigences présentées dans le document de définition de la compatibilité Android (CDD). D'un point de vue pratique, le CDD est une série d'exigences techniques et non techniques spécifiées en langage naturel. Chacune de ces exigences est assortie d'un label qui indique si elle doit être adoptée, si son adoption est fortement recommandée ou si elle est simplement recommandée. Pour vérifier si les équipementiers respectent ces règles, nous avons analysé tous les CDD publiés de la version 1.6 à la version 9.0 d'Android, et extrait toutes les règles relatives aux paramètres de sécurité qu'un fabricant doit, ou devrait, respecter. Ainsi, l'analyse effectuée dans cette phase ne compare pas les ROMs des fabricants avec les ROMs AOSP originales, mais extrait plutôt les configurations nécessaires des ROMs, et les compare aux règles définies dans les CDD.

Par ordre d'apparition, la première exigence de durcissement du système a été introduite dans le CDD d'Android 4.3, où Google a annoncé la prise en charge de SELinux pour les appareils Android. Par conséquent, toutes les ROMs basées sur Android 4.3+ doivent supporter et implémenter le contrôle d'accès obligatoire SELinux. Ensuite, à partir d'Android 7, la section Compatibilité des modèles de sécurité s'est principalement concentrée sur les options de configuration du noyau. Étonnamment, les exigences de sécurité du CDD ne mentionnent pas le durcissement de l'espace utilisateur avant Android 9, et la seule exigence de durcissement de l'espace utilisateur n'est définie que comme fortement recommandée.

Comme il est possible de le constater, il n'y a que trois domaines dans

lesquels la CDD définit des règles, à savoir le noyau, les politiques SELinux et certaines configurations binaires de l'espace utilisateur : notre analyse se concentre donc sur ces trois aspects. En ce qui concerne les noyaux, notre analyse basée sur la configuration et la récupération des symboles du binaire du noyau, a identifié que 7,9% (190 sur 2,396) des noyaux (de 10 fabricants différents) violent la CDD pour leur version spécifique d'Android car ils ne mettent pas en œuvre une ou plusieurs exigences de sécurité obligatoires. Parmi ceux-ci, 162 sont utilisés dans des ROMs ré-architecturées avec Project Treble, ciblant ainsi une version d'Android supérieure ou égale à 8.0. La violation la plus courante, trouvée sur 150 noyaux, concerne l'absence de protections de la mémoire du noyau visant à marquer les régions et sections de mémoire sensibles en lecture seule ou non exécutables. Nous avons également identifié que 10% (241 sur 2,396) des noyaux (de 10 fabricants) n'implémentent pas une ou plusieurs fonctionnalités fortement recommandées. Cette fois, nous avons remarqué que 160 fabricants n'ont pas activé la randomisation de l'espace d'adressage du noyau.

Pour chaque version d'Android qui prend en charge SELinux, l'AOSP fournit une politique standard que les fabricants peuvent utiliser comme base pour construire et personnaliser leur politique SELinux. L'analyse, effectuée sur 1,817 ROM qui définissent une politique SELinux, a montré que 7% (108 ROM) violent la spécification CDD pour leur version Android correspondante car ils définissent toujours un ou plusieurs domaines permissifs. Nous avons constaté que cette violation était répartie entre 16 fabricants différents. Sur les 1,817 ROM, 1,533 ciblent Android supérieur ou égal à 5 et 29% d'entre elles (443) violent la spécification CDD en définissant une ou plusieurs règles contrairement aux règles par défaut *neverallow*.

La dernière catégorie de durcissement du système définie par Google est liée aux binaires de l'espace utilisateur. Comme indiqué précédemment, les exigences relatives aux binaires n'ont été introduites que dans Android 9, et jusqu'à présent, elles ne couvrent que deux aspects : Control Flow Integrity (CFI) et Integer Overflow Sanitization (IntSan). Comme ces deux mécanismes de défense ont été introduits dans le CDD d'Android 9, nous n'avons considéré que les 196 ROMs dont le SDK est supérieur ou égal à 28. Parmi elles, 85 (43,37%) contenaient au moins un binaire désactivant CFI et 104 (53,06%) contenaient au moins un binaire désactivant IntSan. Dans ces cas, six fabricants uniques ont réduit la sécurité d'un binaire, par rapport à l'AOSP, violant ainsi la recommandation du CDD.

Les résultats de cette première analyse montrent que l'ensemble actuel de réglementations et de contrôles est clairement insuffisant.

A.4.2 Personnalisation : Analyse et résultats

L'analyse des dangers et des répercussions des personnalisations sur la sécurité du système étant un vaste sujet, nous avons décidé d'étudier des aspects spécifiques qui opèrent à différents niveaux et privilèges. Ainsi, dans cette analyse, nous avons pris en compte les binaires en espace utilisateur, les configurations des scripts "init", et enfin, les modifications des règles SELinux. Il est important de souligner que cette analyse ne vise pas à identifier les violations des règles définies par Google, mais plutôt à mesurer combien et quelles sont les modifications qui peuvent potentiellement impacter, positivement ou négativement, la sécurité d'un système.

A ce stade, notre analyse porte sur deux aspects. Le premier, qui est basé sur l'analyse différentielle, compare les modifications apportées aux composants déjà présents dans l'AOSP. Le second, en revanche, est lié à l'analyse des nouveaux composants ajoutés par le fabricant. En ce qui concerne les personnalisations sur les bibliothèques binaires livrées dans une ROM, notre analyse montre une tendance presque constante d'environ 80 nouvelles fonctions ajoutées à 20% des bibliothèques système, rendant ainsi vains les efforts de Project Treble. De plus, cette analyse a également mis en évidence le fait que les fabricants utilisent encore des fonctions anciennes et dépréciées de l'AOSP, probablement parce que leur code hérité en dépend encore. Cela pose un sérieux problème de sécurité car l'utilisation d'une fonction qui n'est plus maintenue dans l'AOSP ne reçoit aucun correctif de sécurité ni aucune vérification. Nous avons cependant réalisé que les modifications apportées aux composants binaires du système ne se limitent pas seulement au code, mais affectent également la façon dont il est compilé et distribué. Une analyse systématique des protections introduites par les compilateurs nous a permis d'identifier plusieurs problèmes : par exemple, bien que les canaris de pile soient parmi les plus anciennes fonctionnalités de sécurité présentées dans Android, environ 40% des binaires des fabricants sont dépourvus de cette fonctionnalité de base. Il en va de même pour l'adoption des mécanismes No-Execute (NX) et Full RELocation Read-Only (RELRO). Bien que ces mesures d'atténuation soient largement utilisées par l'AOSP pour renforcer ses binaires, nous avons remarqué que, parmi les binaires introduits par les fabricants, ces mécanismes de défense ne sont pas très répandus.

Le système d'exploitation Android s'appuie sur un système de script pour lancer les binaires et les services au démarrage. Malheureusement, ce composant a fait l'objet de nombreux problèmes de sécurité dans le passé, dans de nombreux cas, en raison de personnalisations introduites par les fabricants. L'analyse relative aux scripts "init" a révélé plusieurs tendances

inquiétantes de la part des fabricants. Nos résultats montrent comment, au fil des ans, les fabricants ont toujours apporté des changements considérables aux scripts init et, en particulier, comment le nombre total de services nouvellement définis est en constante augmentation—certaines ROMs définissant près de 250 services supplémentaires par rapport à l'AOSP. Pour exprimer cela en termes de chiffres absolus, par exemple, un AOSP 8.0 (SDK 26) avait, en moyenne, 59 services définis dans le script init, alors qu'une ROM moyenne en avait 90, avec un pic de fabricants définissant 195 services supplémentaires. De plus, le nombre étonnant de services qui sont lancés avec des privileges "root" rend la situation encore pire puisqu'une surface d'attaque importante sur-privilégiée est exposée, et les fabricants violent probablement le principe du moindre privilège.

La même tendance qui affecte les scripts init affecte également SELinux. Nous identifions trois grands types de personnalisation : le premier concerne les règles qui modifient une règle préexistante pour étendre les permissions et les opérations autorisées sur une ressource donnée. Le second consiste en des règles qui modifient un domaine de politique de base existant mais juste en l'étendant pour supporter de nouvelles ressources introduites par le fabricant, et enfin, des règles qui sont complètement nouvelles et qui opèrent sur des domaines et des ressources qui ne sont pas présents dans la politique AOSP originale. Parmi les différents changements identifiés, nous avons remarqué que certains fabricants étaient particulièrement agressifs dans le nombre de nouvelles règles ajoutées au système. Nous avons remarqué, par exemple, que la politique SELinux du SDK 27 contenait en moyenne 10,000 règles, mais que certains fabricants ont défini une politique contenant plus de 232,000 règles (c'est-à-dire une augmentation de plus de 20 fois). Compte tenu de la sensibilité de ce composant, du fait que même Google, en gérant les règles les plus restrictives pour l'AOSP, commet des erreurs, et du fait qu'au fil des ans, une seule règle erronée a suffi à compromettre l'intégrité d'un appareil en réintroduisant un bogue, il est difficile de croire que des règles 20 fois plus nombreuses sont exemptes de vulnérabilité.

Les résultats et les perspectives de cette analyse montrent qu'il existe encore plusieurs domaines de personnalisation qui, même s'ils ne violent aucune exigence stricte, sont à l'origine de graves problèmes de sécurité. Bien que le CDD soit un excellent point de départ, nous pensons qu'il devrait être considérablement étendu pour empêcher les fabricants de personnaliser leurs ROMs d'une manière qui va à l'encontre de nombreuses pratiques et principes de sécurité bien établis.

A.5 Conclusion

Dans cette thèse, j'ai voulu montrer comment l'écosystème complexe d'Android fait face à de nombreux défis concernant sa sécurité. Cette analyse multidimensionnelle nous a permis de comprendre comment les différents acteurs, contribuant à cet écosystème, voient et abordent la sécurité de manière significativement différente. Dans cette thèse, nous avons commencé par montrer comment la sécurisation des connexions réseau reste un problème ouvert pour les applications Android, et nous avons proposé une amélioration du nouveau mécanisme de défense Network Security Policy, qui, nous l'espérons, permettra une plus grande sécurité et une utilisation plus répandue. Ce parcours s'est poursuivi par l'analyse d'un autre problème qui affecte l'écosystème depuis les premières versions, à savoir le phishing. J'ai abordé cette question de deux manières complémentaires. La première visait à identifier, dans la phase de développement, les bogues utilisés par les attaquants pour monter cette attaque, et la seconde visait à détecter les attaques sur l'appareil au moment où elles se produisent. Ces deux approches se sont avérées prometteuses, et nous ont permis de trouver de nouvelles vulnérabilités et de bloquer, sur l'appareil, tous les malwares connus utilisant cette attaque. J'ai voulu conclure cette thèse par l'étude de ce qui est peut-être le plus gros problème qui représente le mieux l'ampleur de cet écosystème: la fragmentation. Avec cette étude, nous avons enfin mesuré l'impact sur la sécurité des changements apportés par les différents fabricants au fil des années, montrant que, dans la plupart des cas, ceux-ci ont un impact négatif sur la posture de sécurité du système.

Cette thèse doit toutefois être considérée comme un point de départ : nous espérons en effet avoir posé les bases permettant à d'autres chercheurs d'approfondir, d'étendre et d'améliorer nos idées. Nous espérons voir, à l'avenir, le développement d'outils qui permettent au développeur de vérifier la configuration de sa "Network Security Policy," ou que nous pourrons obtenir d'avoir une sandbox plus restreinte qui isole l'exécution du code reçu via HTTP et réduit ses capacités. En outre, nous espérons que les nouvelles approches et techniques de "Software Testing" pourront être utilisées pour trouver des vulnérabilités supplémentaires qui permettent aux logiciels malveillants sur Android d'exécuter des attaques de phishing, ou pour tester, de manière automatisée, si les changements de code effectués par un fabricant introduisent des vulnérabilités. Nous espérons que cette thèse pourra servir de base à ces futurs travaux.

J'espère que cette thèse aura fait progresser, ne serait-ce que légèrement, la

sécurité de l'écosystème Android, qu'elle aura fourni des indications utiles pour accélérer l'adoption du HTTPS partout sur les applications Android, qu'elle aura permis d'avancer dans l'effort de Google pour éradiquer le problème du phishing, et qu'elle aura inspiré de futurs travaux et analyses dans le domaine important des personnalisations OEM.

References

[52808] RFC 5280. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL). http://tools.ietf.org/html/rfc5280, 2008. Accessed: October 13, 2021.

[Ach16] Jagdish Prasad Achara. *Unveiling and Controlling Online Tracking. (Traçage en ligne : démystification et contrôle)*. PhD thesis, Grenoble Alpes University, France, 2016.

[AD] Official Documentation Android Developers. Android Manifest application. https://developer.android.com/guide/topics/manifest/application-element.html?#usesCleartextTraffic. 2019, Accessed: June, 2020.

[AD16] Official Documentation Android Developers. NetworkSecurityPolicy isCleartextTrafficPermitted, API. https://developer.android.com/reference/android/security/NetworkSecurityPolicy.html#isCleartextTrafficPermitted(), 2016. Accessed: October 13, 2021.

[AD19] Official Documentation Android Developers. Network security configuration. https://developer.android.com/training/articles/security-config, 2019. Accessed: October 13, 2021.

[AD20] Platform Documentation Android Developers. Android 8.0 Behavior Changes. https://developer.android.com/about/versions/oreo/android-8.0-changes, 2020. Accessed: October 13, 2021.

[Ada21] Alexandre Adamski. A Samsung RKP Compendium. https://blog.longterm.io/samsung_rkp.html, 2021. Accessed: October 13, 2021.

[Adk11] Heather Adkins. An update on attempted man-in-the-middle attacks. https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html, 2011. Accessed: October 13, 2021.

[ads21] Choose a monetization model for your app. https://developer.android.com/distribute/best-practices/earn/monetization-options, 2021. Accessed October 13, 2021.

[ADY21] Possemato Andrea, Nisi Dario, and Fratantonio Yanick. Preventing and detecting state inference attacks on android. In *Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium, February 2021.

[AGoAR17] HCL Technologies Alon Galili of Aleph Research. Cordova-Android MiTM Remote Code Execution, CVE-2017-3160. https://alephsecurity.com/vulns/aleph-2017013, 2017. Accessed: October 13, 2021.

[Ale16] Klyubin Alex. An Update on Android TLS Adoption. https://security.googleblog.com/2016/04/protecting-against-unintentional.html, 2016. Accessed: October 13, 2021.

[And11] AndroidRank. AndroidRank, open android market data since 2011. https://www.androidrank.org, 2011. Accessed: October 13, 2021.

[and20a] Android compatibility definition document. https://source.android.com/compatibility/cdd, 2020. Accessed October 13, 2021.

[and20b] Android Init Language. https://android.googlesource.com/platform/system/core/+/master/init/README.md, 2020. Accessed October 13, 2021.

[and20c] Android ONE. https://www.android.com/one/, 2020. Accessed October 13, 2021.

[and21] Android Codenames, Tags, and Build Numbers. https://source.android.com/setup/start/build-numbers#source-code-tags-and-builds, 2021. Accessed October 13, 2021.

[ant20] Verifying Boot - Rollback protection. https://source.android.com/security/verifiedboot/verified-boot, 2020. Accessed October 13, 2021.

[AP17] Efthimios Alepis and Constantinos Patsakis. Trapped by the UI: the android case. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, volume 10453 of *Lecture Notes in Computer Science*, pages 334–354. Springer, 2017.

[App11a] AppBrain. AppBrain: Monetize, advertise and analyze Android apps. Ad Networks. https://www.appbrain.com/stats/libraries/ad-networks, 2011. Accessed: October 13, 2021.

[App11b] AppBrain. AppBrain: Monetize, advertise and analyze Android apps. Network Libraries. https://www.appbrain.com/stats/libraries/tag/network/android-network-libraries, 2011. Accessed: October 13, 2021.

[ASDY21] Possemato Andrea, Aonzo Simone, Balzarotti Davide, and Fratantonio Yanick. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In *IEEE Symposium on Security & Privacy*. IEEE Computer Society, May 2021.

[ATH+18] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise android API protection mapping derivation and reasoning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1151–1164. ACM, 2018.

[AZD16] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting inconsistent security configurations in custom android roms via differential analysis. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 1153–1168. USENIX Association, 2016.

[AZHL12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 217–228. ACM, 2012.

[BBD16a] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 356–367. ACM, 2016.

[BBD+16b] Michael Backes, Sven Bugiel, Erik Derr, Patrick D. McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 1101–1118. USENIX Association, 2016.

[BCI+15] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 931–948. IEEE Computer Society, 2015.

[Bel20] Maxime Rossi Bellom. CVE-2020-0069: Autopsy of the Most Stable MediaTek Rootkit. https://blog.quarkslab.com/cve-2020-0069-autopsy-of-the-most-stable-mediatek-rootkit.html, 2020. Accessed: October 13, 2021.

[BHMW16] Damjan Buhov, Markus Huber, Georg Merzdovnik, and Edgar R. Weippl. Pin it! improving android network security at runtime. In *2016 IFIP Networking Conference, Networking 2016 and Workshops, Vienna, Austria, May 17-19, 2016*, pages 297–305. IEEE Computer Society, 2016.

[bK21] SecureList by Kaspersky. Mobile malware evolution 2020. https://securelist.com/mobile-malware-evolution-2020/101029/, 2021. Accessed: October 13, 2021.

[BPW13] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal analysis of android ad library permissions. *CoRR*, abs/1303.0857, 2013.

[Bro16] Broadcom. Android malware finds new ways to derive current running tasks. https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=d3231e0f-67a0-4b31-8adb-4247ca23243d&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments, 2016. Accessed: October 13, 2021.

[Bru16] Chad Brubaker. Changes to Trusted Certificate Authorities in Android Nougat. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html, 2016. Accessed: October 13, 2021.

[Cim19] Catalin Cimpanu. Over 58,000 Android users had stalkerware installed on their phones last year. https://www.zdnet.com/article/over-58000-android-users-had-stalkerware-installed-on-their-phones-last-year/, 2019. Accessed: October 13, 2021.

[CK18] Hyunwoo Choi and Yongdae Kim. Large-scale analysis of remote code injection attacks in android apps. *Secur. Commun. Networks*, 2018:2489214:1–2489214:17, 2018.

[con20] MDM contributors. Web technology for developers: Strict-Transport-Security. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security, 2020. Accessed: October 13, 2021.

[cor20] Core Kernel Requirements. https://source.android.com/devices/architecture/kernel/core-kernel-reqs, 2020. Accessed October 13, 2021.

[CQM14] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: UI state inference

and novel android attacks. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 1037–1052. USENIX Association, 2014.

[cra18] OATmeal on the Universal Cereal Bus: Exploiting Android phones over USB. https://googleprojectzero.blogspot.com/2018/09/oatmeal-on-universal-cereal-bus.html, 2018. Accessed October 13, 2021.

[cts11] Add Test to Verify NX is Enabled. https://android-review.googlesource.com/c/platform/cts/+/21776, 2011. Accessed October 13, 2021.

[Cut19] Stephanie Cuthbertson. Sharing what's new in Android Q. https://www.blog.google/products/android/android-q-io/, 2019. Accessed October 13, 2021.

[cve18] CVE-2018-9488. https://nvd.nist.gov/vuln/detail/CVE-2018-9488, 2018. Accessed October 13, 2021.

[CWS14] Brett Cooley, Haining Wang, and Angelos Stavrou. Activity spoofing and its defense in android smartphones. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, volume 8479 of *Lecture Notes in Computer Science*, pages 494–512. Springer, 2014.

[Dat16] DataTheorem. TrustKit Android: Easy SSL pinning validation and reporting for Android. https://github.com/datatheorem/TrustKit-Android, 2016. Accessed: October 13, 2021.

[Dav17] Corbin Davenport. Google will remove Play Store apps that use Accessibility Services for anything except helping disabled users. https://www.androidpolice.com/2017/11/12/google-will-remove-play-store-apps-use-accessibility-services-anything-except-helping-disabled-users/, 2017. Accessed: October 13, 2021.

[Det18] CVE Details. Xiaomi Stock Browser: content provider injection. https://www.cvedetails.com/cve/CVE-2018-20523/, 2018. Accessed: October 13, 2021.

[Det19] CVE Details. Xiaomi Stock Browser: URL spoofind. https://www.cvedetails.com/cve/CVE-2019-10875/, 2019. Accessed: October 13, 2021.

[Dev16] Android Developers. Official Documentation NetworkSecurityPolicy, API. https://developer.android.com/reference/android/security/NetworkSecurityPolicy, 2016. Accessed: October 13, 2021.

[Dev19a] Appodeal Android SDK Developer. Appodeal Android SDK. Android SDK Integration Guide. https://wiki.appodeal.com/en/android/2-6-4-android-sdk-integration-guide, 2019. Accessed: October 13, 2021.

[Dev19b] HeyZap Android SDK Developer. HeyZap Android SDK. http://web.archive.org/web/20190615131844/https://developers.heyzap.com/docs/android_sdk_setup_and_requirements, 2019. Accessed: October 13, 2021.

[Dev20a] MoPub SDK Developer. Integrate the MoPub SDK for Android. https://developers.mopub.com/publishers/android/get-started/, 2020. Accessed: June 2020.

[Dev20b] Android Developers. AOSP Design Architecture: Conscrypt. https://source.android.com/devices/architecture/modular-system/conscrypt, 2020. Accessed: October 13, 2021.

[DMW15] Thurston H. Y. Dang, Petros Maniatis, and David A. Wagner. The performance cost of shadow stacks and stack canaries. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 555–566. ACM, 2015.

[dtm15] dtmilano. AndroidViewClient. https://github.com/dtmilano/AndroidViewClient, 2015. Accessed: October 13, 2021.

[DZJ+20] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. Bscout: Direct whole patch presence test for java executables. In Srdjan Capkun and Franziska Roesner,

editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1147–1164. USENIX Association, 2020.

[ext10] extract-ikconfig - Extract the .config file from a kernel image. https://github.com/torvalds/linux/blob/master/scripts/extract-ikconfig, 2010. Accessed October 13, 2021.

[FBX+17] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 121–136. IEEE Computer Society, 2017.

[FCP+16] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Zhuoqing Morley Mao, and Atul Prakash. Android UI deception revisited: Attacks and defenses. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*, volume 9603 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2016.

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: an analysis of android SSL (in)security. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 50–61. ACM, 2012.

[FHP+13] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 49–60. ACM, 2013.

[fir15] Firmware file. https://firmwarefile.com/, 2015. Accessed October 13, 2021.

[FQCL17] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. Cloak and dagger: From two permissions to complete control of the UI feedback loop. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 1041–1057. IEEE Computer Society, 2017.

[FW11] Adrienne Felt and David Wagner. Phishing on mobile devices. 05 2011.

[GB17] Project Zero Gal Beniamini. Lifting the (Hyper) Visor: Bypassing Samsung's Real-Time Kernel Protection. https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html, 2017. Accessed: October 13, 2021.

[GCY+16] Yacong Gu, Yao Cheng, Lingyun Ying, Yemian Lu, Qi Li, and Purui Su. Exploiting android system services through bypassing service helpers. In Robert H. Deng, Jian Weng, Kui Ren, and Vinod Yegneswaran, editors, *Security and Privacy in Communication Networks - 12th International Conference, SecureComm 2016, Guangzhou, China, October 10-12, 2016, Proceedings*, volume 198 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 44–62. Springer, 2016.

[GN19] Hogben Giles and Idika Nwokedi. Protecting against unintentional regressions to cleartext traffic in your Android apps. https://android-developers.googleblog.com/2019/12/an-update-on-android-tls-adoption.html, 2019. Accessed: October 13, 2021.

[Goo] Google. UsageStatsManager Documentation. https://developer.android.com/reference/android/app/usage/UsageStatsManager. Accessed: October 13, 2021.

[Goo12] Google. A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson, 2012. Accessed: October 13, 2021.

[goo21] Factory images for nexus and pixel devices. https://developers.google.com/android/images, 2021. Accessed October 13, 2021.

[GRR+20] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1039–1055. IEEE, 2020.

[Gru19] Leonid Grustniy. What's wrong with "legal" commercial spyware. https://www.kaspersky.com/blog/stalkerware-spouseware/26292/, 2019. Accessed: October 13, 2021.

[HTY+20] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R. B. Butler. Bigmac: Fine-grained policy analysis of android firmware. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 271–287. USENIX Association, 2020.

[Hua12] Huawei. Huawei EMUI. https://consumer.huawei.com/en/emui-11/, 2012. Accessed: October 13, 2021.

[HWC14] John Hubbard, Ken Weimer, and Yu Chen. A study of SSL proxy attacks on android and ios mobile applications. In *11th IEEE Consumer Communications and Networking Conference, CCNC 2014, Las Vegas, NV, USA, January 10-13, 2014*, pages 86–91. IEEE, 2014.

[ICW18] Bumjin Im, Ang Chen, and Dan S. Wallach. An historical analysis of the seandroid policy evolution. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 629–640. ACM, 2018.

[Jag18] Chandraiah Jagadeesh. Red Alert 2.0: Android Trojan targets security-seekers. https://news.sophos.com/en-us/2018/07/23/red-alert-2-0-android-trojan-targets-security-seekers/, 2018. Accessed: October 13, 2021.

[Kas16] Kaspersky. Asacub Android Trojan: From Information Stealing to Financial Fraud. https://www.kaspersky.com/about/press-releases/2016_asacub-android-trojan-from-information-stealing-to-financial-fraud, 2016. Accessed: October 13, 2021.

[ker92] Linux Kernel Banner. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/init/version.c, 1992. Accessed October 13, 2021.

[Kev17] Sun Kevin. BankBot Found on Google Play and Targets Ten New UAE Banking Apps. https://blog.trendmicro.com/trendlabs-security-intelligence/bankbot-found-google-play-targets-ten-new-uae-banking-apps/, 2017. Accessed: October 13, 2021.

[KN18] Jakob Lell Karsten Nohl. The Android ecosystem contains a hidden patch gap. https://srlabs.de/bites/android_patch_gap/, 2018. Accessed: October 13, 2021.

[Koz16] John Kozyrakis. CVE-2016-2402. https://koz.io/pinning-cve-2016-2402/, 2016. Accessed: October 13, 2021.

[Kra17] Nick Kralevich. Honey, I Shrunk the Attack Surface. Adventures in Android Security Hardening. https://www.blackhat.com/docs/us-17/thursday/us-17-Kralevich-Honey-I-Shrunk-The-Attack-Surface-Adventures-In-Android-Security-Hardening.pdf, 2017. Accessed: October 13, 2021.

[Lab13] MWR F-Secure Lab. Paypal Remote Code Execution, CVE-2013-7201, CVE-2013-7202. https://labs.f-secure.com/advisories/paypal-remote-code-execution/, 2013. Accessed: October 13, 2021.

[Lab14] MWR F-Secure Labs. Paypal Remote Code Execution. https://labs.f-secure.com/advisories/paypal-remote-code-execution/, 2014. Accessed: October 13, 2021.

[Ley11] John Leyden. Inside 'Operation Black Tulip': DigiNotar hack analysed. https://www.theregister.co.uk/2011/09/06/diginotar_audit_damning_fail/, 2011. Accessed: October 13, 2021.

[lin91] Linux Kernel. https://github.com/torvalds/linux/, 1991. Accessed October 13, 2021.

[lkm20] Loadable Kernel Modules. https://source.android.com/devices/architecture/kernel/loadable-kernel-modules, 2020. Accessed October 13, 2021.

[LLC08] Google LLC. Android API reference. https://developer.android.com/reference, 2008. Accessed: October 13, 2021.

[LLC20a] Google LLC. CVE-2020-0069. https://source.android.com/security/bulletin/2020-03-01/, 2020. Accessed: October 13, 2021.

[LLC20b] Google LLC. ODM Partitions. https://source.android.com/devices/bootloader/partitions/odm-partitions, 2020. Accessed: October 13, 2021.

[mac15] Android 6.0 Changes - Access to Hardware Identifier. https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-hardware-id, 2015. Accessed October 13, 2021.

[med16] Hardening the media stack. https://android-developers.googleblog.com/2016/05/hardening-media-stack.html, 2016. Accessed October 13, 2021.

[med20] Control Flow Integrity. https://source.android.com/devices/tech/debug/cfi, 2020. Accessed October 13, 2021.

[MJ20] Project Zero Mateusz Jurczyk. MMS Exploit: Samsung Qmage Coded. https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-1-introduction-to-qmage.html, 2020. Accessed: October 13, 2021.

[MKC17] Luka Malisa, Kari Kostiainen, and Srdjan Capkun. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 289–300. ACM, 2017.

[MTC+18] Huasong Meng, Vrizlynn L. L. Thing, Yao Cheng, Zhongmin Dai, and Li Zhang. A survey of android exploits in the wild. *Comput. Secur.*, 76:71–91, 2018.

[Nic18] Lorenz Nicole. MysteryBot - the Android malware that's keylogger, ransomware, and trojan. https://blog.avira.com/mysterybot-the-android-malware-thats-

keylogger-ransomware-and-trojan/, 2018. Accessed: October 13, 2021.

[nok21] Marin nokiamob. Large number of Modern Phones, including Nokia, are besigned by ODM companies. https://nokiamob.net/2021/01/17/large-number-of-modern-phones-including-nokia-are-designed-by-odm-companies/, 2021. Accessed: October 13, 2021.

[OAD+15] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. To pin or not to pin-helping app developers bullet proof their TLS connections. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 239–254. USENIX Association, 2015.

[OC15] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, pages 15:1–15:6. ACM, 2015.

[Opp13] Oppo. OPPO ColorOS. https://www.coloros.com/en/coloros7, 2013. Accessed: October 13, 2021.

[osm21] Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2021. Accessed October 13, 2021.

[oST19] National Institute of Standards and Technology. CVE-2019-15340. https://nvd.nist.gov/vuln/detail/CVE-2019-15340, 2019. Accessed: October 13, 2021.

[pae14] Practical Android Exploitation. http://theroot.ninja/PAE.pdf, 2014. Accessed October 13, 2021.

[PF20] Andrea Possemato and Yanick Fratantonio. Towards HTTPS everywhere on android: We are not there yet. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 343–360. USENIX Association, 2020.

[PFB+14] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android

applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[PFNW12] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David A. Wagner. Addroid: privilege separation for applications and advertisers in android. In Heung Youl Youm and Yoojae Won, editors, *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, pages 71–72. ACM, 2012.

[PNC17] Duy Phuc Pham, Croese Niels, and Han Sahin Cengiz. Exobot - Android banking Trojan on the rise. https://www.threatfabric.com/blogs/exobot_android_banking_trojan_on_the_rise.html, 2017. Accessed: October 13, 2021.

[pro17] SELinux for Android 8.0. https://source.android.com/security/selinux/images/SELinux_Treble.pdf, 2017. Accessed October 13, 2021.

[Qur20] Quram. QURAM SOFT. http://www.quramsoft.com/, 2020. Accessed: October 13, 2021.

[Rah20] Mishaal Rahman. Critical MediaTek rootkit affecting millions of Android devices has been out in the open for months. https://www.xda-developers.com/mediatek-su-rootkit-exploit/, 2020. Accessed: October 13, 2021.

[Rav20] Ole André Vadla Ravnås. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/docs/android/, 2020. Accessed: October 13, 2021.

[RBA17] Elena Reshetova, Filippo Bonazzi, and N. Asokan. Selint: An seandroid policy analysis tool. In Paolo Mori, Steven Furnell, and Olivier Camp, editors, *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017, Porto, Portugal, February 19-21, 2017*, pages 47–58. SciTePress, 2017.

[RBN+16] Elena Reshetova, Filippo Bonazzi, Thomas Nyman, Ravishankar Borgaonkar, and N. Asokan. Characterizing seandroid policies in the wild. In Olivier Camp, Steven Furnell, and Paolo

Mori, editors, *Proceedings of the 2nd International Conference on Information Systems Security and Privacy, ICISSP 2016, Rome, Italy, February 19-21, 2016*, pages 482–489. SciTePress, 2016.

[RLZ17] Chuangang Ren, Peng Liu, and Sencun Zhu. Windowguard: Systematic protection of GUI security in android. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[RNV+18] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Philippa Gill. Studying TLS usage in android apps. In *Proceedings of the Applied Networking Research Workshop, ANRW 2018, Montreal, QC, Canada, July 16-16, 2018*, page 5. ACM, 2018.

[RZX+15] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in android. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 945–959. USENIX Association, 2015.

[sam20] Issue 2002: Samsung Android multiple interactionless RCEs and other remote access issues in Qmage image codec built into Skia. https://bugs.chromium.org/p/project-zero/issues/detail?id=2002, 2020. Accessed October 13, 2021.

[SC13] Stephen Smalley and Robert Craig. Security enhanced (SE) android: Bringing flexible MAC to android. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[SCM+16] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[SDW12] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. Adsplit: Separating smartphone advertising from applications. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 553–567. USENIX Association, 2012.

[sec20] Security Enhancements. https://source.android.com/security/enhancements, 2020. Accessed October 13, 2021.

[sel14] Android 4.4.3 Patch Finally Closes Up An Ancient Vulnerability, Shuts Down Several Serious Security Exploits. https://www.androidpolice.com/2014/06/04/android-4-4-3-patch-finally-closes-ancient-vulnerability-shuts-several-serious-security-exploits/, 2014. Accessed October 13, 2021.

[SGC+] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries.

[She17] Di Shen. Defeating Samsung KNOX with zero privilege. https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege-wp.pdf, 2017. Accessed: October 13, 2021.

[SKC15] Hossain Shahriar, Tulin Klintic, and Victor Clincy. Mobile Phishing Attacks and Mitigation Techniques. In *Journal of Information Security*, volume 06, pages 206–212, 06 2015.

[SKC+16] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: enforcing in-app privilege separation in android. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[SKGM18] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 749–763. ACM, 2018.

[SKS16] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[SPM18] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. Scandroid: Automated side-channel analysis of android apis. In Panos Papadimitratos, Kevin R. B. Butler, and Christina Pöpper, editors, *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec 2018, Stockholm, Sweden, June 18-20, 2018*, pages 224–235. ACM, 2018.

[sta15] Experts Found a Unicorn in the Heart of Android. https://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android/, 2015. Accessed October 13, 2021.

[sto21] Stock rom. https://www.stockrom.net/, 2021. Accessed October 13, 2021.

[Stu20] Ruby Game Studio. Hunter Assassin. https://play.google.com/store/apps/details?id=com.rubygames.assassin, 2020. Accessed: October 13, 2021.

[TBR15] Daniel R. Thomas, Alastair R. Beresford, and Andrew C. Rice. Security metrics for the android ecosystem. In David Lie and Glenn Wurster, editors, *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2015, Denver, Colorado, USA, October 12, 2015*, pages 87–98. ACM, 2015.

[tcp20a] Tcpdump Common Vulnerabilities. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=tcpdump, 2020. Accessed October 13, 2021.

[tcp20b] Tcpdump Public CVE List. https://www.tcpdump.org/public-cve-list.txt, 2020. Accessed October 13, 2021.

[Tea14] Android Security Team. Google Report: Android Security 2014 Year in Review. https://source.android.com/security/reports/Google_Android_Security_2014_Report_Final.pdf, 2014. Accessed: October 13, 2021.

[THC+18] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Ruales, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. Attention spanned: Comprehensive vulnerability analysis of AT commands within the android ecosystem. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 273–290. USENIX Association, 2018.

[Thr18] ThreatFabric. BianLian - from rags to riches, the malware dropper that had a dream. https://www.threatfabric.com/blogs/bianlian_from_rags_to_riches_the_malware_dropper_that_had_a_dream.html, 2018. Accessed: October 13, 2021.

[Thr19] ThreatFabric. Anubis II - malware and afterlife. https://www.threatfabric.com/blogs/anubis_2_malware_and_afterlife.html, 2019. Accessed: October 13, 2021.

[Tom15] Federico Tomassetti. JavaParser - Parser and Abstract Syntax Tree for Java. https://github.com/javaparser/javaparser, 2015. Accessed: October 13, 2021.

[tra20] Trade Federation Overview. https://source.android.com/devices/tech/test_infra/tradefed, 2020. Accessed October 13, 2021.

[VAHD17] Eline Vanrykel, Gunes Acar, Michael Herrmann, and Claudia Diaz. Leaky birds: Exploiting mobile application traffic for surveillance. pages 367–384, 05 2017.

[VBP+14] Mario Linares Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do API changes trigger stack overflow discussions? a study on the android SDK. In Chanchal K. Roy, Andrew Begel, and Leon Moonen, editors, *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 83–94. ACM, 2014.

[ven21] Mobile vendor market share worldwide. https://gs.statcounter.com/vendor-market-share/mobile/

worldwide/#monthly-201003-202007, 2021. Accessed October 13, 2021.

[Vit19] Ventura Vitor. Gustuff banking botnet targets Australia. https://blog.talosintelligence.com/2019/04/gustuff-targets-australia.html, 2019. Accessed: October 13, 2021.

[vml19] vmlinux-to-elf. https://github.com/marin-m/vmlinux-to-elf, 2019. Accessed October 13, 2021.

[vol14a] Android Vulnerabilities: vold asec. https://androidvulnerabilities.org/vulnerabilities/vold_asec, 2014. Accessed October 13, 2021.

[vol14b] Root 4.4.X - Pie for Motorola devices. https://forum.xda-developers.com/moto-x/orig-development/root-4-4-x-pie-motorola-devices-t2771623, 2014. Accessed October 13, 2021.

[VSA+19] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. An empirical study of C++ vulnerabilities in crowd-sourced code examples. *CoRR*, abs/1910.01321, 2019.

[vts21] Vendor Test Suite (VTS) & Infrastructure. https://source.android.com/compatibility/vts, 2021. Accessed October 13, 2021.

[WDW14] Longfei Wu, Xiaojiang Du, and Jie Wu. Mobifish: A lightweight anti-phishing scheme for mobile phones. In *23rd International Conference on Computer Communication and Networks, ICCCN 2014, Shanghai, China, August 4-7, 2014*, pages 1–8. IEEE, 2014.

[Wel15] Ryan Welton. Remote Code Execution as System User on Samsung Phones. https://www.nowsecure.com/blog/2015/06/16/remote-code-execution-as-system-user-on-samsung-phones, 2015. Accessed: October 13, 2021.

[WER+15] Ruowen Wang, William Enck, Douglas S. Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. Easeandroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In Jaeyeon Jung and Thorsten Holz, editors, *24th*

USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015, pages 351–366. USENIX Association, 2015.

[WGZ+13] Lei Wu, Michael C. Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 623–634. ACM, 2013.

[wis19] Android Security: Taming the Complex Ecosystem. https://wisec19.fiu.edu/wp-content/uploads/wisec2019-keynote.pdf, 2019. Accessed October 13, 2021.

[WPN17] Gahr Wesley, Duy Phuc Pham, and Croese Niels. LokiBot - The first hybrid Android malware. https://www.threatfabric.com/blogs/lokibot_the_first_hybrid_android_malware.html, 2017. Accessed: October 13, 2021.

[Xia10] Xiaomi. Xiaomi MIUI. https://consumer.huawei.com/en/emui-11/, 2010. Accessed: October 13, 2021.

[XZ12] Zhi Xu and Sencun Zhu. Abusing notification services on smartphones for phishing and spamming. In Elie Bursztein and Thomas Dullien, editors, *6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings*, pages 1–11. USENIX Association, 2012.

[YHG19] Guangliang Yang, Jeff Huang, and Guofei Gu. Iframes/popups are dangerous in mobile webview: Studying and mitigating differential context vulnerabilities. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 977–994. USENIX Association, 2019.

[YLC+19] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and detecting overlay-based android malware at market scales. In Junehwa Song, Minkyong Kim, Nicholas D. Lane, Rajesh Krishna Balan, Fahim Kawsar, and Yunxin Liu, editors, *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys*

2019, Seoul, Republic of Korea, June 17-21, 2019, pages 168–179. ACM, 2019.

[ZAD13] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: isolating advertisements from mobile applications in android. In Charles N. Payne Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 9–18. ACM, 2013.

[Zer21] Google Project Zero. Introducing the In-the-Wild Series. https://googleprojectzero.blogspot.com/2021/01/introducing-in-wild-series.html, 2021. Accessed: October 13, 2021.

[ZLZ+14] Xiao-yong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 409–423. IEEE Computer Society, 2014.

[ZYH+18] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating insecure input validations in android services. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1165–1178. ACM, 2018.

[ZYN+15] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao-yong Zhou, and XiaoFeng Wang. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 915–930. IEEE Computer Society, 2015.