CryptoPatcher: Automatic On-Device Patching of Crypto API Misuses in Android Applications

Florian Draschbacher, BSc

CryptoPatcher: Automatic On-Device Patching of Crypto API Misuses in Android Applications

Florian Draschbacher BSc

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Johannes Feichtner
Univ.-Prof. Dipl.-Ing. Dr.techn. Stefan Mangard
Institute for Applied Information Processing and Communications (IAIK)

Graz, January 2021

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The document uploaded to TUGRAZonline is identical to the present thesis.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Dokument ist mit der vorliegenden Arbeit identisch.

Date/Datum

Signature/Unterschrift

Abstract

As people are shifting an increasing amount of their sensitive private data onto smartphones and tablets, mobile security has moved into the focus of the research community. Numerous publications in recent years have uncovered serious security vulnerabilities in Android applications that stemmed from the misuse of cryptographic Application Programming Interfaces (APIs).

Despite major efforts by Google as the platform provider to educate application developers in the correct use of cryptographic primitives, a considerable portion of programmers remain unable or unwilling to follow the recommended best practices. Consequentially, they keep putting their users at risk of falling victims to trivial attacks. As a means to protect users who depend on potentially insecure applications, a third-party solution is needed.

In this thesis, we present CryptoPatcher as a solution to this problem. Once configured on an Android device, our system reacts to every new application installation by automatically generating and installing a patched package that transparently mitigates several classes of potential crypto API misuses. Since our solution does not require any user intervention for its operation, it is perfectly suited for untrained novices. Still, CryptoPatcher also offers advanced monitoring functionality for system administrators that allows gaining detailed insights into the accessed crypto APIs, chosen parameterisations and potential vulnerabilities of installed applications. Based on the provided information, advanced users can judge the trustworthiness of a specific program and toggle CryptoPatcher's mitigations.

Since the operation on an Android device places our system in tight performance constraints, we designed a custom Android Patch Development Kit (APDK) for constructing application-agnostic patches that can be deployed to arbitrary compiled Android packages. Through a custom XML patch format and a domain-specific language based on Java annotations, patch developers can express modifications to a target software's manifest file and intercept invocations of function calls. Furthermore, additional resources and native libraries can be injected into target packages. Since the specific code manipulation scheme is abstracted from the patch code, our system can provide two different rewriting backends, each with unique characteristics in terms of performance and compatibility.

As a proof of the efficacy and efficiency of CryptoPatcher, we analysed our approach on a selection of popular applications from Google Play, each risking disclosure of user data through misuse of cryptographic APIs. In our evaluation, we show how our solution successfully impedes attacks and confirm the correctness of CryptoPatcher's displayed warnings by locating the source of the identified vulnerabilities in the reverse-engineered source code of the test subjects.

Keywords: Android, Patching, Cryptography, Dynamic Analysis, Mobile Security

Kurzfassung

Während ein wachsender Teil der Bevölkerung seine sensiblen Daten auf mobile Endgeräte wie Tablets und Smartphones verlagert, rückt der Bereich Mobile Security auch immer mehr in den Fokus der Forschung. Mehrere Veröffentlichungen der letzten Jahre haben in einer Vielzahl von Android-Apps ernstzunehmende Sicherheitslücken entdeckt, die ihren Ursprung in der unsachgemäßen Verwendung von kryptografischen Programmierschnittstellen (APIs) haben.

Trotz großer Bemühungen von Seiten des Plattform-Anbieters Google, App-Entwickler im korrekten Umgang mit kryptografischen Primitiven zu schulen, zeigt sich ein beträchtlicher Teil der Programmierer immer noch unwillig oder unfähig, den Empfehlungen zu folgen. Um zu verhindern, dass sie die Daten ihrer Kunden weiterhin trivialen Attacken ausliefern, scheint eine Drittanbieter-Lösung unumgänglich.

In dieser Arbeit präsentieren wir CryptoPatcher als Lösung dieses Problems. Einmal auf einem Android-Gerät eingerichtet, reagiert das System auf jede App-Installation mit der automatischen Generierung und Installation eines modifizierten Programm-Pakets, das potentielle Fehler in der Verwendung von Krypto-APIs selbständig behebt. Weil dazu keine Intervention des Nutzers erforderlich ist, ist die Software auch für ungeschulte Anwender bestens geeignet. Für Administratoren bietet das System auch eine detaillierte Protokollierung der verwendeten Krypto-APIs, gewählten Parametrierung und möglichen Sicherheitslücken von installierten Apps. Auf Basis der so gewonnenen Informationen können fortgeschrittene Nutzer nach Einschätzung der Vertrauenswürdigkeit eines Programms Schutzvorrichtungen bei Bedarf deaktivieren.

Weil der Betrieb auf einem Android-Gerät unser System zu Sparsamkeit im Umgang mit System-Ressourcen zwingt, haben wir ein maßgeschneidertes Android Patch Development Kit (APDK) entworfen, das die Konstruktion anwendungs-agnostischer Patches erlaubt, die auf beliebige kompilierte Android-Pakete angewandt werden können. Über ein eigenes XML-Patch-Format und eine domänenspezifische Sprache basierend auf Java-Annotations können Patch-Entwickler Modifikationen der Manifest-Datei formulieren oder Funktions-Aufrufe abfangen. Auch das Einbetten von Ressourcen und Nativen Bibliotheken ist möglich. Da die genaue Methode der Code-Manipulation vom Patch-Code abstrahiert ist, kann unser System zwei verschiedene Rewriting-Backends integrieren, die jeweils unterschiedliche Charakteristika in Bezug auf Performance und Kompatibilität bieten.

Als Beweis der Effektivität von CryptoPatcher haben wir unsere Herangehensweise an einer Auswahl beliebter Apps aus dem Google-Play-Store untersucht, die alle durch Fehler im Umgang mit Krypto-APIs die Preisgabe von Nutzerdaten riskieren. In unserer Evaluierung zeigen wir, wie unsere Lösung erfolgreich Attacken verhindert und bestätigen die Korrektheit der von CryptoPatcher angezeigten Warnungen, indem wir den Ursprung der Sicherheitslücken im dekompilierten Quellcode der untersuchten Apps lokalisieren.

Schlüsselwörter: Android, Patchen, Kryptografie, Dynamische Analyse, Mobile Security

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

First and foremost, I would like to take this opportunity for thanking my thesis advisor Johannes Feichtner. He not only played a major role in the selection of this thesis' topic, but also did his best to help me stay on track as I worked. Whenever I had reached yet another unforeseen technical hurdle during the development of the CryptoPatcher system, he sent me valuable ideas and advise that always made me feel like I wasn't fighting the demons of technology on my own. During the writing work of my thesis, I was impressed with the speed of his email responses and the amount of thoughts Johannes had poured into his feedback.

Furthermore, I wish to express my sincere gratitude to my parents Heimo and Regine and my brother Thomas, who through the twists and turns of the extraordinary year 2020 ended up being my office colleagues during the majority of my work on this thesis. Their continuous encouragement and support carried me through the occasional difficult times, as did the excellent meals my father prepared whilst himself struggling with the challenging situation of working from home.

Florian Draschbacher
Graz, Austria, January 2021

Chapter 1

Introduction

As adoption rates of smart phones have grown over the last decade, people are gradually shifting larger parts of their computing tasks onto the new platform. Today, mobile phones not only represent a centre piece of people's communication, but have also taken over responsibilities as a means of payment and medium for controlling banking accounts. A major factor enabling this development is the broad availability of applications that claim to offer easy and secure access to a wealth of privacy-sensitive functionality. To this end, they make use of special infrastructure in the operating system's *Application Programming Interface* (API), that provides cryptographic primitives for obtaining secure random numbers, performing encryption and decryption of data or establishing secure communication channels.

Several studies over the last few years have revealed that in contrast to people's belief, applications often fail to deliver the degree of security required for processing sensitive data, due to improper employment of crypto primitives. Frequently, apps use the correct API component for a specific task, but do not parameterise it correctly. As a consequence, many applications are prone to trivial attacks by adversaries that can extract sensitive information completely unnoticed by the user.

In an effort to improve the situation, we are presenting CryptoPatcher, a system that is designed to automatically protect users from crypto API misuses in apps on their Android devices. The CryptoPatcher system is comprised of three components: The **Android Patch Development Kit (APDK)** was built to aid in developing and applying patches to Android application packages. Using our kit, we constructed a patch we call **CryptoPatch**, to specifically target and fix commonly found crypto API misuses in third-party applications. Lastly, the **CryptoPatcher Android application** automatically applies CryptoPatch to all applications installed on an Android device. It also includes a monitor component that provides high-level descriptions of detected vulnerabilities and additional low-level details on an app's usage of cryptographic primitives that advanced users can use as a basis for deciding about the inspected program's trustworthiness.

1.1 Crypto API Misuses on the Android Platform

Mobile devices' foray into people's homes in recent years has also piqued curiosity of the research community. The first series of papers on the topic investigated the security of SSL/TLS implementations in Android applications and was started in 2012 by Fahl et al. [20] and Georgiev et al. [25]. The former focused on the user perspective and found that 8% of 13500 Android apps that were run through conservative automated static analysis were in some form vulnerable to a *Man-In-The-Middle* (MITM) attack. Additionally, the majority of apps was found to still use unprotected HTTP. Georgiev et al.[25] concentrated on the developer perspective, and found that a myriad of popular Java libraries was vulnerable to MITM attacks, thus affecting any Android application that integrated them. Sounthiraraj et al. [49] developed a system combining automated static and dynamic methods for analyzing SSL/TLS

implementations and found that only 3% of more than 23000 tested apps from Google Play were actually vulnerable to MITM attacks. In contrast, statically analyzing the 500 most popular apps in 20 Google Play categories of 2013, Tendulkar et al.[51] found that more than 15% [1] were vulnerable. As shown by Buhov et al.[8, 9], the general trend of growth of the rate of vulnerable apps continued, reaching more than 30% by 2014.

By early 2016, the scale of the problem alerted Google to take measures. An analysis step was added to Google Play and all apps affected by insecure SSL/TLS implementations were banned from the store starting from May 2016[2]. Additionally, the new Android 7.0 release introduced a feature called *Network Security Configuration*[3] that allowed developers to specify custom trusted SSL certificates via an XML configuration file. This addressed a very frequent cause of SSL/TLS misuse that stemmed from developers writing their own certificate verification code in order to use custom, often self-signed, certificates. Subsequent studies by Wang et al.[54], Shin et al.[48] and Tang et al.[50] revealed that these measures were very effective in clearing the Google Play Store of apps affected by SSL/TLS issues, but a considerable amount of apps in third-party app stores stayed vulnerable, particularly in Asia. Additionally, apps that still do not employ certificate pinning (figures published in 2017 by Razaghpanah et al. reported only 5% of apps used certificate pinning [44]) are prone to attacks if a Certificate Authority (CA) is compromised, which has happened in the past as documented by the *Open Web Application Security Project* (OWASP) Foundation [42].

Soon after the first issues with SSL/TLS in Android apps were discovered, researchers initiated a second area of research investigating misuse of other crypto-related APIs on Android, and presented even more alarming results: Egele et al. [18] found that 68% of the subset of collected Play Store apps that utilise crypto APIs made at least one mistake in doing so, thus weakening the security premises of the used primitives. The most common issue was that apps were using block ciphers in the *Electronic Code Book* (ECB) mode, which is considered insecure for most uses (as discussed in Section 2.1.2). Subsequent studies by Shao et al. [47], Chatzikonstantinou et al.[10], Gajrani et al. [23] and Muslukhov et al. [39] reported even higher rates of affected apps between 80 and 90% of those that used cryptography. While Muslukhov et al. [39] observed a general trend of improvements to app security, Gao et al. [24] recently reported that analysed app updates were more likely to introduce new crypto misuses than address existing ones. Unanimously, both studies suggest that the problem still affects a large portion of currently available Android applications.

As the criticality of the problem came to the attention of the research community, several potential mitigations were presented. In an effort to fix the matter in the long term, Google integrated additional Lint rules into the Android Studio *Integrated Development Environment* (IDE) that warn developers when compiling application source code that potentially supplies cryptographic primitives with insecure parameters. Still, warnings are easily ignored, and many applications were even already written before these checks were put in place and have rarely been updated since. Consequently, a solution is needed that works on compiled application binaries. In order to detect the misuse of crypto APIs in these, most of the existing solutions employ static analysis, which involves identifying function calls within the compiled application code and tracing back the used parameters. This approach is prone to imprecise or erroneous classification and computationally expensive, prohibiting its use in an on-device solution. Other suggestions employ dynamic analysis, which provides higher precision and performance, but usually comes at the cost of requiring a modified OS install with root access. Our new proposal, the CryptoPatcher system, solves the problem at hand with a flexible dynamic analysis approach that is performant enough for execution on an Android device itself, yet works on unmodified Android firmwares.

[1] Apps that only used SSL in third-party libraries were excluded

[2] About Those X509TrustManager Emails: https://commonsware.com/blog/2016/02/22/about-x509trustmanager-emails.html

[3] Changes to Trusted Certificate Authorities in Android Nougat: https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html

1.2 The CryptoPatcher System

Our contributions in this thesis are threefold: First, we introduce the **Android Patch Development Kit**, a new system for developing and deploying patches that can be injected into arbitrary applications available only in compiled form. Second, we present **CryptoPatch**, which we designed with the goal of automatically mitigating security vulnerabilities in third-party apps induced by their misuse of cryptographic APIs. In order to demonstrate the capability of both these new technologies, our third contribution **CryptoPatcher** packages them into a form that provides a practical benefit to end users, by running a system daemon on the target Android device itself that automatically applies the CryptoPatch patch to any new application that is installed on the system.

1.2.1 Android Patch Development Kit

The Android Patch Development Kit allows building application patches within the Android Studio IDE that developers are already accustomed to. Patches can intercept calls to system or library methods, inject resources or native code, and modify the target application's manifest file. The flexible design of the APDK allows to switch the underlying code rewriting backend without requiring modifications to the patch code. Particular focus has also been put on the performance characteristics of the deployment functionality, so that it can itself be executed as part of an Android application.

1.2.2 The CryptoPatch Patch

Using our Android Patch Development Kit, we implemented a patch that aims at automatically correcting misuses of cryptographic APIs in the target application. Based on the taxonomy suggested by Egele et al. [18], we identified four classes of crypto API misuses that can be fixed transparently to the target application. Our patch implements these mitigations and also tightly integrates with the CryptoPatcher application by collecting usage reports for a broader range of privacy-relevant APIs.

1.2.3 The CryptoPatcher Application

To demonstrate the practical applicability of our solution, we implemented the prototype for an application that targets end consumers. In the fashion of an antivirus program, it runs a daemon service on the Android device and constantly watches out for new app installations. When such an event is detected, CryptoPatcher automatically disables the original program and sets up a patched version of it by applying the CryptoPatch patch described above. An additional configuration panel allows to toggle protection on a per-app basis, and displays information on the specific cryptographic APIs accessed by each patched application.

1.3 Outline

The rest of this thesis is structured as described in the following:

Chapter 2 seeks to provide necessary background information for understanding the terminology used and arguments brought up throughout the remainder of this thesis. We touch on cryptographic primitives, provide a primer on the architecture of the Android OS and its application package format, and describe the cryptographic APIs available to app developers.

Chapter 3 summarises other researchers' efforts that share some aspects with this thesis. We discuss works that also aim to fix security vulnerabilities in applications only available in compiled form, and provide an overview of different approaches for modifying the program flow of third-party Android applications.

Chapter 4 provides an introduction and structural overview of the complete CryptoPatcher system. Not only does it introduce the various components, but it also highlights the connections between them and how they work together.

Chapter 5 is dedicated to the Android Patch Development Kit. It starts by summarising the requirements and goal for designing this component, before discussing how it can be used for modifying different aspects of existing applications. Additionally, we provide insights into the inner workings of the various building blocks, and the challenges we faced while implementing them.

In Chapter 6, we cover the CryptoPatch patch and the CryptoPatcher application. Every supported crypto API misuse is detailed, along with a description of the mitigation measures that were implemented. Moreover, we elaborate on the user-facing functionality and what technical hurdles had to be overcome in order to realise the envisioned plan.

Chapter 7 offers an evaluation of the complete CryptoPatcher system, as comprised by the Android Patch Development Kit, the CryptoPatch patch and the CryptoPatcher Android application. We provide performance characteristics of the patching process and an exemplary case study on a select set of popular applications from Google Play. Furthermore, we point out the inherent limitations stemming from the underlying methodologies and technologies we built upon.

Lastly, Chapter 8 concludes this work with a summary of our efforts and contributions in this thesis. We provide a glimpse into the future with respect to the described CryptoPatcher system in the form of possible improvements that could be further explored.

Chapter 2

Background

In this chapter, we provide the background information required for a complete understanding of the discussions in the rest of this thesis.

First, Section 2.1 starts with an overview of cryptographic primitives. It focuses only on the most relevant concepts, beginning with *Pseudo Random Number Generators*, continuing with different kinds of Ciphers for encryption and decryption, and finishing with an excerpt about *Password-Based Encryption*.

In a very similar way, the adjacent Section 2.2 puts the introduced primitives into the bigger context of the *TLS/SSL* layers commonly used for securing network communication and adds the concepts of *Public Key Infrastructure* and *TLS Pinning* to the picture.

In the following Section 2.3, we provide a primer about the *Android OS* architecture, spanning from a description of the underlying Linux kernel, across the low-level system services, frameworks and application APIs, until arriving at the actual user-facing applications on the highest level.

Next, the Android app package format is explained in Section 2.4, including crucial details about the contained *Android Manifest* file, the compiled *Binary XML* format and the *ARSC* resource index structure. The section also charts the specifics of the uncompressed resources contained in an app package archive, as well as the native shared libraries. Finally, the section reports recent changes to the packaging and signing process required for obtaining an installable file.

We continue with Section 2.5 and an account of the Android runtime, focusing on the more modern *ART* runtime that implements a hybrid between ahead-of-time and just-in-time compilation. Our discussions include the modalities of compiling bytecode to native code and touch on the different calling conventions involved during method invocations.

As a conclusion to this chapter, a short summary of the cryptography APIs offered by the Android framework in Section 2.6 shows the convergence point of all earlier topics. We briefly explain the *Java Security Provider* system for cryptographic primitives and enumerate the different cryptography implementations available on Android, before covering the core components that make up the SSL framework of the platform.

2.1 Cryptographic Primitives

This section describes a few of the most commonly used cryptographic building blocks, or primitives, that have some relevancy for this thesis. Please note we will only provide a rough overview for the topics covered here. The interested reader is advised to consult background literature on the respective subjects for more complete descriptions.

2.1.1 Pseudo Random Number Generators

For the most part, the biggest strength of a computer is its determinism, i.e. the fact that the same program will always lead to the same result given the same input data. While this is perfectly desirable in most situations, there are cases where some degree of indeterminism is required in order to make the computer run through a different sequence of instructions upon repeated execution of a program. Commonly, the solution to this problem are *Pseudo Random Number Generators* (PRNG), which, based on some initial seed value, will generate a sequence of seemingly random values. These are not true random number generators, because the probability of a specific value x to be drawn is not uniform across all possible values x. Although this is not a problem for applications like games, many cryptographic primitives rely on some form of unique value that has to be unpredictable by any means.

In order to fulfil this requirement, a special class of *Cryptographically Secure Pseudo Random Number Generators* (CSPRNG) was conceptualised. While the distribution of the generated values more closely resembles true uniformity, these mathematical devices still require seed material to initialise their internal state. Most commonly, the system state and environment measurements from hardware sensors can be combined into a seed that is reasonably suitable for use as a random seed. Additionally, many implementations of CSPRNGs allow application developers to provide extra seed material that usually augments the one coming from the system.

2.1.2 Ciphers

In cryptography, *ciphers* are mathematical devices that utilise a secret key for transforming information (the *plain text*) into an encrypted *cipher text* that looks like random data to an observer. Still, the process is reversible, so that the recipients of the cipher text are able to decrypt it, i.e. recover the original plain text from the cipher text if they are in possession of the correct key. Fundamentally, ciphers provide data confidentiality to their operators.

If the keys for encryption and decryption are identical, the cipher is called *symmetric*, otherwise it is termed *asymmetric*. The latter case is more computationally intensive than the former, but invaluable for specific applications. Asymmetric ciphers enable the implementation of signature schemes, where the hash fingerprint of some input data is encrypted with a private signing key. The receiver of the data can then use the public counterpart of the key for verifying the signature.

An additional taxonomy distinguishes between symmetric *stream ciphers* and symmetric *block ciphers*. While a stream cipher is capable of processing an arbitrary amount of data, a block cipher operates on equally-sized chunks of data at a predefined length. In order to use a block cipher on an arbitrary amount of data, the latter has to be split into a block size suitable for the chosen cipher, and it has to be padded, so that the last block of data is not left partially filled. A mode of operation then specifies how each individual block is to be processed before running it through the block cipher.

The most basic mode of operation is called *Electronic Code Book* (ECB) mode. Because it encrypts all blocks independently using the same key, this mode is inherently insecure for any application that processes more than one block. An attacker can trivially compare blocks of cipher text that were encrypted with the same key to determine whether they originated from the same plain text block.

Secure modes of operation incorporate the cipher text from the preceding block of data into the encryption of the block following. While this principle alone already guarantees unique cipher text blocks for repeated identical blocks of plain text, comparing the joined cipher text of multiple encryption runs that used the same key still allows an attacker to deduce knowledge about the plain texts. To mitigate this problem, the first block of data is augmented with a unique random data block called *Initialisation Vector* (IV), which can be handed to the data recipient together with the cipher text. It is crucial that this IV be unique for every encryption with the same key.

2.1.3 Password-Based Encryption

While ciphers require keys that utilise the full entropy available in the key size for highest effectiveness, humans prefer passwords that consist of displayable characters, such as letters, numbers or punctuation marks. *Password-Based Key Derivation Functions* (PBKDF) were designed specifically to bridge that gap. They expand a user-specified password into a key that can be used for operating a cipher. In order to compensate for the limited password entropy caused by the reduced set of human-readable passwords, the PBKDFs adds other measures that are intended to slow down a brute-force attacker.

The commonly used *PBKDF2* adds a unique random *salt* to the password to ensure that even two identical passwords yield different keys, considerably slowing down bulk brute force attacks. Moreover, the PBKDF allows for an iteration count argument that roughly specifies the time duration the derivation should take. This parameter is chosen so that it causes an unnoticeable delay when run for deriving a single key, but effectively impedes brute-forcing, where all potential keys have to be tried in rapid succession.

When PBKDF2 was first published in 2000, the recommended iteration count was specified at 1000 [31]. As computers have grown more powerful since then, the value was periodically updated. In recent years, the *National Institute of Standards and Technology* (NIST) has recommended a minimum iteration count of 10 000 for server applications [26]. It is an application developer's responsibility to keep up with these recommendations to ensure the end user's data stays properly secured.

2.2 Transport Layer Security (TLS)

In this section, we will provide a relatively high-level description of the *Transport Layer Security* (TLS) protocol, which today represents one of the backbones of the modern Internet.

The *Transport Layer Security* (TLS) protocol and its predecessor *Secure Socket Layer* (SSL) specify an encryption layer that can be used on top of a networking protocol in order to protect the transmitted information from being intercepted by a passive attacker on the same network. The encryption and decryption is carried out transparently to the application layer on top. While we will focus on describing the most commonly used *HTTPS* protocol combination (*Hypertext Transfer Protocol* over TLS), the same principles apply to any other application layer protocol alike.

In analogy to the typically used lower-level *Transport Control Protocol* (TCP), TLS communication involves a server and a client. After the client establishes a network connection to the target server, the two parties exchange a series of TLS *handshake messages* for confirming each other's identity and securely agreeing on an encryption scheme and key for use in all subsequent communication. In the case of HTTPS, servers usually don't authenticate the client, but the opposite operation is crucial for ensuring that no attacker is impersonating the target server. The authentication procedure is one of the most delicate pieces of the protocol.

2.2.1 Public Key Infrastructure

The TLS protocol is built on the core principle of asymmetric cryptography. A TLS server is required to be configured with an asymmetric key pair, consisting of a public key that acts as an identifier and a private key that is used to prove the claimed identity to the client during the TLS key exchange or key agreement.

Although this key pair mechanism establishes identities in the domain of the TLS protocol, it still does not provide a secure way to bind them to identities in the other layers of the protocol stack. An attacker could still interpose as an arbitrary host, just by generating its own valid key pair. Because the secured communication channel to the target host is only about to be arranged at the time its identity has to be confirmed, the client does not have a way to query it for its true SSL identity on demand. The *Public Key Infrastructure* (PKI) was designed to bridge this and similar gaps that arise in various applications of asymmetric cryptography.

Every program that is designated to participate in TLS communication maintains a set of trusted TLS identities in the form of TLS certificates that link a public key to the name of a host. These trusted identities are known as root *Certificate Authorities* (CAs) that notarise other server's identities, which in turn can act as CAs on a smaller scale. At the end of this tree-like construct, there are leaf certificates that an TLS server actually uses as its TLS identity. A signed certificate contains the identity of the signer, so that it is possible for an SSL client to follow a chain of trust from an identity presented by an TLS server to a trusted root CA.

While this hierarchical PKI provides a flexible solution, it still contains a number of flaws. Most prominently, the root CAs represent a very exposed point of failure. If the private key of a CA is compromised, attackers can issue valid certificates for arbitrary domains, which allows them to intercept any client's communication with the targeted host. While this may sound like an unrealistic scenario, exactly this has already happened multiple times in the past [42]. Although special mechanisms have been put in place to revoke leaf and intermediary CA certificates, revoking root CA certificates was not provisioned for when designing the system. Instead, TLS client applications have to adapt and remove the compromised certificate. Most programs rely on a set of trusted CAs provided by the operating system, which meant that in some cases, an OS update was required to patch the gaping security hole. While this is not too much of a problem on well-maintained desktop operating systems, devices running the Android operating system are often abandoned by the manufacturer a few years after release, which leaves millions of users unprotected.

2.2.2 TLS Pinning

To mitigate this problem, an additional protection scheme was introduced that ensures the authenticity of the TLS server a client application is communicating with even in absence of a well-maintained set of trusted CAs. *TLS Pinning* involves short-cutting a part of the chain of trust by not (only) relying on the relatively large set of trusted CAs included in the OS, but also shipping the TLS client application with the TLS identity of an intermediary CA or the precise leaf certificate of the target server itself.

Two slightly different implementations of SSL Pinning exist. *Certificate Pinning* operates on the whole certificate, meaning that every property of the certificate must exactly match a value hardcoded in the client application. While this provides the greatest degree of security, it is also a very inflexible solution, because it necessitates an application update whenever the server certificate changes. Since certificates are usually issued for only a very limited time span of a few months, this approach results in the need for very frequent app updates.

The other possibility is called *Public Key Pinning*, and compares certificates only in terms of their public key, ignoring mismatches in any other property. While this permits continued use of the application across server certificate updates that retained the key pair, it also provides less protection against the compromise of these keys. While Certificate Pinning limits the potential security hole to the lifespan of a single certificate, Public Key Pinning allows an attacker to keep intercepting connections of users who failed to install the application update by issuing new certificates that use the same key pair.

Both of these variants are adequate solutions for applications that only access a limited set of TLS servers known when shipping the product to customers. As noted by [41], organisations employing SSL pinning are also required to have tight control over both server and application development, so that they can keep the pinned certificates updated.

2.3 Android OS Architecture

This section covers details about the Android operating system, which the CryptoPatcher system is targeting.

Android is an *Operating System* (OS) originally designed for mobile devices such as smartphones or tablets. Since its release in 2007, its lightweight and flexible nature has lead it into other embedded environments as well, such as TVs and smart watches. The *Android Open Source Project* (AOSP) is developed by Google in annual release cycles for the most part, so that a major revision is rolled out to recent devices once a year. As of May 2019, there were more than 2.5 billion actively used Android devices worldwide[1]. Between October 2019 and October 2020, devices running the Android OS were estimated to account for more than 40% of worldwide Internet traffic[2].

The foundation of the Android OS is formed by the *Linux* kernel, which manages the underlying hardware and system resources. The next layer in the conceptual stack are the different system services and native libraries that together lay the ground for the Java frameworks and APIs, part of which already live inside the Java world established by the Android runtime. Lastly, at the top of the structural pyramid reside the consumer applications that can leverage all the underlying technologies to generate a real-world benefit for the user.

2.3.1 Linux Kernel

Like many modern embedded operation systems, the Android OS is based on a Linux kernel, which is responsible for interfacing with the low-level hardware. The kernel itself includes core functionality for controlling CPU speed, memory management, scheduling of processes, implementing access control and file input or output. Additionally, the different kernel subsystems can be extended with kernel modules.

A key kernel module added for the Android OS is the *Binder Inter-Process Communication* (IPC) mechanism. It provides a way for applications to call exposed functionality of other processes, copying call arguments and results across virtual memory regions of different processes. Other kernel modules represent device drivers for the different *Integrated Circuits* (ICs) in the system, such as network interfaces, sensors, displays and various I/O buses. Following the *UNIX* philosophy, all these drivers expose device files in a special virtual file system, so that user space applications can take advantage of the available hardware and kernel infrastructure.

Once the kernel has brought up its different subsystems, it hands over operation to user-space applications. These are executing in a less-privileged context on the CPU, so that they have to go through the kernel in order to access certain functionality, such as control of hardware components. The kernel provides a system call mechanism that allows user space applications to trigger a special kind of interrupt, which the kernel handles by executing a particular functionality on behalf of the caller depending on the arguments passed from the user space program. By having the kernel supervise all accesses to critical infrastructure, it is in a position to enforce configured access policies, so that a group of administrator (*root*) users can selectively grant permissions to other (non-administrator) users.

Because the Android operating system, in its designation for mobile devices such as smart phones, was expected to process very privacy-sensitive information, its developers chose to not grant root access to normal end users in official release builds. While this generally helps preventing malicious applications from taking over the whole device, it also impedes the deployment of advanced third-party protection measures against more specific attacks, for example against vulnerabilities introduced by misconfigured applications.

[1] Google I/O'19: https://youtu.be/TQSaPsKHPqs?t=3298

[2] Net Market Share: https://netmarketshare.com/operating-system-market-share.aspx

2.3.2 System Services

The first user space program started by Android's Linux kernel is the init process, which in turn spawns an array of additional system services. Many of these are native daemons, responsible for providing a very specific slice of relatively low-level functionality, such as USB debugging or Bluetooth communication. One of the started processes, named *Zygote*, takes on a very central role in the system.

The Zygote process can be considered the nucleus for all other processes that execute Java code, be it as part of a system component or a third-party user-installed application. When it is started, Zygote configures the *Java Virtual Machin*e (JVM) (the specific implementation depends on the Android version, as discussed in Section 2.5 below) and preloads a set of commonly used framework classes. Once the JVM is initialised, it spawns the system server covered in the next paragraph and starts a Unix domain socket, listening for fork requests issued by other processes, such as the *System Server*. In response to these requests, Zygote forks into a new process that executes a specified Java program.

Spawned by the Zygote process, the System Server process is another important corner stone in the operation of the Android OS. It acts as a registry for various different sub services, each responsible for managing a specific portion of the OS functionality and exposing it to other processes at a higher level. A considerable amount of the functions available in the API is constructed around proxy components that interact with the system server process via the Binder IPC mechanism. Similar to how the kernel conceptually shields user-space applications from having arbitrary hardware access in the Linux domain, the Android system server adds an additional barrier that is protected with Android's own permission system.

2.3.3 Frameworks and APIs

The Android OS ships with a wealth of APIs and extension points that third-party developers can leverage to implement specific application functionality and improve the over-all user experience of the platform. While most of these APIs are exposed in the form of Java classes, many of the interfaces that interact with the OS or hardware in some way are implemented on top of either the System Server or native shared libraries provided by the OS.

In contrast to most Linux distributions, the Android OS does not employ the *GNU* project's C library *glibc*. Instead, a custom C library called *bionic* was brought into existence, assembled from parts of other open-source projects and components that were implemented from scratch, specifically tailored for low-power embedded use. While bionic is compatible with the *POSIX* API in most parts, some functionality was purposefully omitted for improved security. Other underlying native libraries include *BoringSSL* (Android's fork of OpenSSL) and libraries for hardware-accelerated 3d graphics or media decoding.

In addition to these Android-specific APIs, Android applications can also take advantage of many packages of the Java platform taken over from the *OpenJDK*[3] project.

2.3.4 Applications

While all applications running on an Android device share their ancestry from the Zygote process, they generally run in separate processes. As an additional isolation measure, the OS also assigns different Linux user IDs to installed application packages. This mechanism is then used to provide each application with exclusive access to its own data folder. In order to access files outside of this private folder, applications have to be granted permissions from the OS, and ultimately from the user. Similar permissions also exist for managing Internet access or control over hardware sensors.

[3]OpenJDK: https://openjdk.java.net

Applications are most commonly developed in the *Java* or *Kotlin* programming languages, but may also include native machine code for performance-critical parts of their functionality. For deployment, they are packaged into a special *APK* container format that bundles all compiled code, as well as resources needed at runtime. While manual installation from any source is possible, most users download their programs exclusively from Google's official app marketplace *Play Store*, which also provides them with the possibility to update software packages once a new version becomes available.

2.4 App Package Format

The Android OS defines its own *Android Package* (APK) file format for deployment of applications onto devices. As part of the installation procedure, the file is copied into a special folder on the device, where it is accessed during the execution of the contained program. The file consists of a *ZIP* archive that contains a manifest declaring the app's interaction points with the system, a resources index file accompanied by the resources themselves and compiled native and Java code (or bytecode-compatible code such as that stemming from Kotlin code). All these components will be detailed in the following subsections.

2.4.1 Android Manifest

The *Android Manifest* is an *XML* file written by the app author, which can be considered the contract between the OS and the application, declaring the supported functionality and required permissions or device capabilities.

For every implementation of one of Android's core application building blocks, a corresponding entry has to be added to the manifest file. These components include *Activities*, which are full-screen application windows, *Services* that can provide functionality while the user is interacting with a different program in foreground and *Content Providers* that expose a well-defined public interface to an app's databases or files. An additional core element of the Android API are *Broadcast Receivers*, which listen to signals raised from the system or other applications via IPC.

In order to limit third-party application's access to the functionality exposed in the manifest file, the software developer has the possibility to extend Android's permission system with custom permissions that can for example be granted only to packages from the same author. For Content Providers, the manifest even allows specifying separate permissions for read and write access.

2.4.2 Binary XML Format

During compilation, XML files used in an Android application, such as the manifest described above, are transformed into a more space-efficient proprietary binary XML format. It mostly differs from the plain text XML format in that any strings (including attribute keys and values) are only stored as references into a common strings table at the beginning of the file. Moreover, every attribute holds an integer *Attribute Identifier* in addition to the attribute key string reference. The mapping between attribute IDs and keys is globally defined, so it can be used for quickly looking up a specific attribute without having to resolve the attribute key string through the strings table.

2.4.3 ARSC Format

Apps may contain many different types of resources, such as layout XMLs describing the structure of a user interface, vector or bitmap images for icons and theming, strings that can be provided in multiple languages, or configuration files. All of the individual resources are internally identified by a unique integer number, and must additionally be assigned a human-readable name. The Android SDK then uses these names to generate a Java class with a final static integer field for every resource, which application code can reference instead of hardcoding the integer identifiers. The Java compiler eventually inlines the

resource IDs, so that this solution greatly improves readability and maintainability of the app source code, while not affecting runtime performance in any way.

In order to save storage space on the target device, many resources of an application are compressed during compilation. As part of this process, an index table is constructed, which keeps track of all available resources. This index, as well as compiled text resources, are stored in a special data structure in the *ARSC* file that can be conveniently `mmap`ped into the process memory at runtime for fast access. Fundamentally, the data is structured into nested chunks, each consisting of a generic header, an additional header depending on the specific type of the chunk, and optional payload data or child chunks.

At the root of this tree-like construct sits the *Table Chunk*, which contains one or several *Package Chunks* and a *Value String Pool Chunk*. While the former essentially bundle a set of resources, the latter stores a list of actual resource string values, so that they may simply be referenced as an offset into the list by consumers inside the Package Chunks.

A *Package Chunk* holds two more *String Pool Chunks*, for the type names and the resource keys respectively, the latter of which are the human-readable resource names mentioned above. The type names are referenced inside other elements of the Package Chunk, so-called *TypeSpec Chunks*. These are for structuring the resources by their type, so that for example all string resources stay together. Lastly, *Type Chunks* are added for every combination of resource type and device configuration. For instance, a separate Type Chunk may exist for German string resources that actually includes entries for the individual string resources, each containing a reference into the Package Chunk's Key String Pool and the Table Chunk's Value String Pool.

While some resource values, most notably all string resources, are stored directly in the ARSC structure, images and compiled XML layouts reside as separate files inside the res folder within the APK archive. For these, only their path is used as the resource value inside the Type Chunks.

The integer identifiers used for addressing an app's resources directly translate to a path through the nested structures described above. For example, the resource id `0x7f0b001e` can be interpreted as the 30th (`0x001e`) resource entry of the 11th (`0x0b`) Type Chunk inside package `0x7f`. Note that by default, the Android build tools merge all resources of an app (even those included through library dependencies) into a single package chunk with ID `0x7f`.

2.4.4 Resources

As mentioned in Section 2.4.3 above, some resources are kept as separate files inside the APK file, albeit in a compressed form. In addition to these processed resources, developers may wish for some files to retain their original form. The Android SDK supports this case as well, by providing special folders whose contents are not touched during compilation, but are still included in the application package.

2.4.5 DEX Format

The centre piece of the APK file's contents are the *Dalvik Executable* (DEX) files[4], which contain the Java program code in a machine-independent intermediate representation called Dalvik *bytecode*. This bytecode is produced in a multi-step process during compilation. As a first step, the conventional Java compiler generates a traditional `.class` file for every Java class, which the Android build tools then transform into one or several DEX files. This deviation from standard Java formats was chosen for more efficient execution on the highly resource-constrained environment of mobile devices.

[4]Dalvik Executable format: `https://source.android.com/devices/tech/dalvik/dex-format`

2.4.5.1 File Structure

At the top level, a DEX file consists of a header section, several index table sections and a data section.

The header at the very start of the DEX file serves two main purposes. To begin with, by starting the file with a well-known magic value and listing several checksum fields, it helps consumer programs recognise the format and verify the integrity of the contained information. As part of the magic value, the specific version of the DEX file format the file adheres to is noted. Additionally, the header also works as a central index for the remaining sections of the file. For all parts described below, dedicated offset and size fields in the header aid in locating the respective data structures within the DEX file.

Immediately following the header, the string IDs table contains a list of string identifiers to be used as constants in the actual program code or as names of other DEX elements. For every identifier, a string data offset points to the exact position of the actual character data, which is encoded in a version of UTF-8 modified to work with zero byte terminators. For performance reasons, the string IDs table is sorted by the code point values and must not contain any duplicates, which also helps minimising the memory footprint.

In the next segment, the type IDs table stores a list of identifiers of primitive, array or class types the bytecode in the DEX file references. Every entry points to an element in the string IDs table, representing a type in the Java type descriptor format, which uses one-letter abbreviations for primitive types, an opening square bracket for arrays and a transformed fully-qualified identifier in the form of `Lcom/mypackage/myclass;` for classes. Similar organisational requirements as for the string IDs table apply.

The immediately adjacent sections together describe the class structure of the program code contained in the DEX file. The class definitions table is the central component in this bundle, declaring the inheritance, implemented interfaces and access flags of the class itself. It also holds references into the field identifiers table and the method identifiers table, both of which contain structures that provide type information of the respective entities through additional references into other tables within the DEX file. Finally, a class definition item also includes a pointer to a class data item, which, through the values encoded in its `virtual_methods` and `direct_methods` fields leads to the code items that contains the actual bytecode instructions. As with the sections described in previous paragraphs, the mentioned tables and lists (as well as those covered in the next paragraph) strictly follow a clearly specified ordering and must not contain any duplicates.

To conclude the identifier tables sections at the start of the file, the DEX format specifies a call site identifiers table and a method handle table, both of which are used in program code whenever execution flow is to be directed towards a different method contained in the same or another DEX file.

Following the identifier tables, the curious investigator will find the data section, which contains the actual values referenced by many of the structures described above. Finally, the map section at the very end of the file provides a similar overview to its contents as the header at the start, but is more tailored for iteration of the whole file, instead of the more random access pattern that occurs during execution.

2.4.5.2 Dalvik Bytecode

All program code for static, virtual or direct methods implemented in the DEX file is following the *Dalvik Bytecode* format[5]. This in essence is a machine code format for a register-based virtual machine that operates on 32-bit registers. Values spanning 64 bits in width are possible as well and are stored in two adjacent registers. While the machine theoretically supports an arbitrary amount of registers, most instructions have a limit to the number of registers they can actually address. The virtual machine processes the instruction stream in code units of 16 bits, although instructions themselves are not fixed

[5]Dalvik bytecode: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

size, but composed of an arbitrary multiple of such blocks. All instructions contain an 8-bit opcode and specify the affected registers or immediate values.

Since they are the most relevant for the rest of this thesis, we will only examine the family of invocation instructions in more detail here, consisting of the `invoke-static`, `invoke-direct`, `invoke-virtual`, `invoke-super` and `invoke-interface` instructions, each with a few minor variations.

The `invoke-static` instruction is used for calling a static class method. In its basic form, it contains a 4-bit argument count, a 16-bit method reference index and up to five 4-bit argument register indices. In effect, this allows the caller to arbitrarily choose five of the first 16 of the machine's registers for passing arguments. When more parameters have to be used, an additional `invoke-static/range` instruction is available that permits method calls to include up to 255 adjacent register indices for transporting the call arguments. The method reference index is an index into the method identifiers table described above.

Very similar in concept is the `invoke-direct` instruction, except that it is used for calling `instance` methods that are not virtual, i.e. private instance methods or constructors. While argument passing works almost identical to the `invoke-static` instruction, there is one important difference. Targeting instance methods, a direct method implicitly takes the this object as its first parameter. Like for `invoke-static`, a range variant is also available for `invoke-direct`.

Another member of the invocation family of instructions is `invoke-virtual`, which is applicable for any virtual instance method, i.e. a method that can be overridden by subclasses. When executed, it performs a lookup in the target object's *vtable* to find the most specific implementation of the method in question. Please note that vtable resolution cannot simply be circumvented by using an `invoke-direct` with a virtual method, since doing so is actively prohibited at least by the ART runtime, although the arguments passing works exactly the same.

A particularly interesting instruction is `invoke-super`, which serves the purpose of requesting execution of the next less specific implementation of the referenced method, i.e. the implementation of the closest ancestor class. The specified method is not executed in verbatim, but used as the basis for a lookup in the vtable of the class that contains the method using the `invoke-super` instruction. Consequentially, invoke-super cannot be used outside of the member methods of the target object's class. In fact, this example goes to show how closely the Dalvik bytecode instruction set is following the feature set of the higher level Java language. The argument transport is handled in the same way as described for `invoke-direct` above.

Lastly, `invoke-interface` is the instruction that allows calling interface methods of objects whose exact type is not known at compile time, so a lookup similar to that for virtual methods is necessary. The mechanism for parameter passing is identical to that of the other instance method invocations described above.

Please note additional `invoke-custom` and `invoke-polymorphic` instructions exist that will not be discussed in more detail here, since they were only added in Android 8.0 for implementing dynamic invocation functionality, which is not relevant here.

2.4.6 Native Libraries

A special Android *Native Development Kit* (NDK) permits building performance-critical portions of an application in native code. It also provides the opportunity to port existing C or C++ libraries to the mobile platform. The native code is compiled into a shared library object, which is embedded into the APK package and can be dynamically linked at runtime. For calls from Java to native machine code and vice versa, the *Java Native Interface* (JNI) is utilised.

2.4.7 Packaging

While traditionally, an Android application was deployed as one monolithic application package and DEX file that supported a wide range of devices, this scheme has been deviated from in recent years.

Originally, a limitation of the DEX file format required applications to stay under the boundary of 2^{16} referenced methods in their Java code. This meant that Android developers had to exercise great caution when introducing library dependencies into their code base. Over time, the situation was becoming increasingly more aggravated as Google kept adding more functionality into its Android Support Libraries to combat the fragmentation problem of the platform. Eventually, a special *MultiDex* library for integration into third-party apps was brought forth, that exploits clever tricks to get the Dalvik runtime to load multiple DEX files in parallel. Although the successor runtime ART was designed to support this functionality from the very start, the responsible header structure in the DEX file format itself has never been modified for reasons of backward compatibility. As a result, even modern apps typically consist of multiple DEX files inside the APK package.

The APK format has received more changes due to the addressing of another problem caused by the heavy fragmentation in the Android device landscape. Developers used to deliver universal APKs to Google Play, which contained resources and native code for any of the system configurations supported by the Android platform. This lead to large application packages that occupied storage space for resources that only some different device really benefitted from. To solve the problem, Google introduced the *Android App Bundle*[6] system that has publishers submit their applications to the Play Store as special intermediary compilation artefacts. Based on the specific device configuration of the end user, Google Play then serves multiple different APK files for code and resources. The infrastructure later was also expanded to allow the on-demand installation of feature modules onto a slim base package. As a consequence of this development, applications are today commonly installed as *Split APKs*, consisting of a base package containing the core functionality and several secondary APK files for the specific use case and system configuration.

2.4.8 Signatures

Once all application files are packaged, the resulting APK file is signed before it can be installed on an actual device. Although (in contrast to the rivalling iOS platform) Android does not require the executable code itself to be signed, having the developer sign the APK package is used as a means for tracing back the origin of the application at install time. Only updates signed with the same certificate as the original application can be applied onto an existing installation, which guarantees that a malicious party cannot access private app files by tricking the user into installing a tampered update. Still, the signing certificates are usually self-signed and do not have to be notarised by a central authority, which means that attackers are left with the possibility to modify existing applications, sign them with an arbitrary certificate and manoeuvre unsuspecting users into installing it, as long as the legitimate original is not installed on the device. For this reason, the Android OS requires explicit user intervention prior to allowing installation of apps from unknown sources (that is, obtained from any place other than Google Play).

The Android OS today supports several different APK signature schemes that each grew out of the need for improvements over its respective predecessor. Until Android 7.0, the signature scheme of the *Java Archive* (JAR) format[7] for distribution of desktop Java programs was used. It operated on all the unzipped files inside the APK structure, noting down a hash fingerprint for any of them in a special index file inside a metadata folder within the archive, which was then used as the basis for generating the signature. The version 2 signature scheme[8] provided improved performance and security, by calculating the signature as

[6] About Android App Bundles: https://developer.android.com/guide/app-bundle
[7] JAR File Specification: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html
[8] APK Signature Scheme v2: https://source.android.com/security/apksigning/v2

a whole on the raw data blocks inside the archive instead of the individual uncompressed files and storing it inside a dedicated ZIP metadata block. The subsequently released version 3[9] expands on version 2, adding support for key rotation and more detailed metadata fields.

Since support of newer signature schemes is bound to recent versions of the Android OS, every one of them was designed with backwards compatibility in mind. Developers are advised to sign their applications with a combined scheme for best compatibility. The end user's device will then pick the most secure contained signature it supports for verification.

2.5 Android Runtime

Because Android applications are typically written in Java or Kotlin and compiled into architecture-independent bytecode instead of native machine code, executing them requires some sort of interpreter or runtime. The following section describes the components employed for this purpose by the Android OS.

Originally, application interpretation was the job of the *Dalvik* runtime, which was officially replaced in Android 5.0 with the more flexible and performant *Android Runtime* (ART). Most notably, recent versions of the newer product employ a combination of *Ahead-Of-Time* (AOT) compilation for translating bytecode into native code during installation and *Just-In-Time* (JIT) compilation for performing this step in a more fine-grained way at runtime. The deprecated Dalvik runtime was originally released as a pure interpreter and later updated to include a JIT that had grown considerable complexity by the time ART was released. Since Dalvik has been discontinued in 2014, we will concentrate our efforts on the ART runtime for this work.

The ART runtime was designed in a modular structure with an abstract compilation driver interface, which allows quickly exchanging the used backend. Originally, the *Quick* compilation driver was employed by default, which was based on functionality inherited from the Dalvik VM. In Android 7.0, a new *Optimising* driver was introduced as the default, which produces more performant code at the cost of longer compilation times.

Independent of the specific backend in use, compilation operates on a per-method level in ART. Depending on the runtime configuration, an app may be completely AOT-compiled during installation, or not at all at that point, in which case it will be ran purely by the interpreter when first launched. Over time, frequently invoked methods will be JIT-compiled for short-term performance improvements and noted in a usage profile, which a special background service then uses as a guideline for AOT compilation of the app at a later point of time.

Internally, the Android runtime maintains in-memory representations of the parsed portions of the DEX files that are augmented with additional metadata relevant for execution. Most notably, an instance of the `ArtMethod` C++ class is created for every method, which holds information about the specific method type and the compilation state, among others.

How an invoked method is executed depends on the compilation state of the caller and the callee methods. If both have been compiled to native code before, the method call may be inlined, which means that execution jumps directly to the compiled code of the callee, without requiring intervention of the ART runtime. Every compilation state follows its own unique calling convention, which means that bridge functions in the ART runtime are necessary for making calls between methods of differing compilation state. These bridge methods then bring the call arguments into the appropriate format for the callee. A similar parameter conversion also has to be performed when calling into native methods via JNI.

[9] APK Signature Scheme v3: https://source.android.com/security/apksigning/v3

2.6 Android Cryptography APIs

This section covers how the different concepts described in the previous sections come together in the various cryptography APIs of the Android platform.

2.6.1 Cryptographic Primitives

For the most part, the Android API follows common Java traditions when it comes to providing cryptographic primitives to application developers. In order to separate the actual implementors of the functionality from its consumers, the Java framework includes a special Security Provider infrastructure. For every cryptography API, the *Java Cryptography Extension* (JCE) defines a *Service Provider Interface* (SPI), which describes the functionality that a provider can supply. The most relevant for this thesis are the `CipherSpi` abstract class for the application-facing `Cipher` class (for encryption and decryption), the `SecureRandomSpi` as the backend of the `SecureRandom` API (CSPRNG functionality), and `KeyGeneratorSpi` for `KeyGenerator` (PBE).

A central provider component is part of every cryptography implementor project and installed with the JCE subsystem, so it can act as a registry for all implemented SPIs of the cryptography package. Application code can create instances of these implementations through factory methods in public-facing JCE classes, which allow either explicitly specifying the desired provider or letting the system chose the system default.

The Android framework provides several of these default providers. Generally the most secure provider is the `AndroidKeystore` provider, which implements particularly strong protection of key material. Cryptographic operations can be performed on a secure coprocessor, so that the actual keys are never loaded into RAM, where they could be compromised by an attacker program. Since this processor is usually limited in the supported algorithms, it is complemented with several more (software-based) providers, each focussing on a particular set of APIs.

The most common pitfall with these APIs is that some default to insecure configurations, while others completely rely on developers choosing safe parameters. In combination with a lack of clarity in the official documentation, this has lead to developers making grave mistakes in protecting their application's data.

2.6.2 TLS/SSL

Although the Android framework heavily borrows from Java practises for TLS/SSL as well, it does add a few unique refinements.

The core components of the *Java Secure Sockets Extension* (JSSE) are the `SSLSocketFactory` and `SSLSocket` classes and the `TrustManager` and `HostnameVerifier` interfaces. As the name suggests, the `SSLSocket` is the element that actually provides the encryption layer over a networking socket. It enables programs to trigger a TLS handshake before using the socket as any other unprotected connection, with all encryption happening transparently to the application code. The `TrustManager` has to be specified during construction of the `SSLSocketFactory` that later is used for instantiating `SSLSocket` instances. During the handshake, it is the `TrustManager`'s task to verify the TLS certificate presented by the server. The `HostnameVerifier` complements the `TrustManager` by making sure the presented certificate matches the server's hostname.

Although only the combination of `TrustManager` and `HostnameVerifier` ensure that a connection is properly secured from Man-In-The-Middle attacks, the Java SSL framework considers the `HostnameVerifier` part of the application layer on top of the TLS layer. As a result, programs that use the `SSLSocket` class directly have to invoke the `HostnameVerifier` in their own code, it is not called by the Java framework as part of the handshake. In the past, this has contributed to great confusion

among application developers trying to employ TLS/SSL in their apps. Additionally, the complexity of the APIs has frequently lead to developers inadvertently leaving their apps unprotected when trying to use self-signed certificates.

In order to address a number of deficiencies in the Java TLS stack, Android introduced various improvements over the standard Java APIs over the years. The most noteworthy addition was the *Network Security Configuration* system that allows developers of apps targeting Android 7.0 or newer to configure trusted TLS certificates in an XML file that is parsed into a special `TrustManager` at runtime. This permits easy integration of self-signed certificates or TLS pinning without the potential vulnerabilities of custom implementations.

Chapter 3

Related Work

Over the past years, the security of Android applications has been subject of numerous scientific publications. In this chapter, we honour these efforts and highlight their differences to our work.

This chapter is structurally separated in two sections. The first one focuses on related works that share the most similarities with our approach, for they also attempted to fix cryptographic API misuses in applications only available in binary form.

Section 3.2 covers a wider range of projects that employ some form of Android application patching. We discuss the strengths and weaknesses of various different approaches, classify them into several categories and compare them with our own implementation.

3.1 Fixing Crypto Misuses

In terms of addressed security vulnerabilities, *CDRep*, published in 2016 by Ma et al. [37], is the existing project that is most similar to ours. It describes a system capable of locating crypto API misuses in compiled third-party Android applications and applying patch templates for fixing them. While CDRep employs offline static analysis for identifying issues, our solution uses a dynamic approach that is performant enough to work on the device itself and benefits from increased accuracy and usability.

In [8], Buhov et al. introduced a Cydia Substrate module for adding certificate pinning to third-party Android applications. Although it shares the dynamic on-device nature, their system differs from our solution in that it requires a rooted device and builds on the Trust-On-First-Use principle. Additionally, our solution is conceived with a broader scope of fixing multiple crypto misuse problems instead of just TLS/SSL.

Bates et al. [6] proposed a solution for fixing SSL certificate verification on Ubuntu by dynamically linking a shim between third-party applications and SSL libraries that implements various kinds of proper certificate verification. Although they had to overcome some similar challenges as we did and also implemented partial support for the Java Secure Socket Extension, their solution is not translatable to Android applications for two reasons. Firstly, it employs the Java Instrumentation infrastructure that is not supported by Android's JVM and secondly, it interposes on internal functionality of the `TrustManagerImpl` that is specific to the desktop JSSE implementation.

Finally, Fahl et al. [21] and Tendulkar et al. [51] suggest solving SSL certificate verification problems on Android by letting developers configure trusted server certificates through XML files. A similar solution was adopted in Android 7.0, as described in Section 2.6.2.

3.2 Android Application Patching

Application patching generally refers to the process of modifying a third-party program's code sequence after it has been compiled. A common application in the context of the Android operating system is intercepting framework methods, so that injected code is executed in place of a called system framework function.

The work that is most similar to our solution in terms of application patching is *AppGuard* by Backes et al. [5], which describes an on-device system for injecting policy checks into security-critical parts of third-party Android applications installed on the same device, effectively creating a more fine-grained permission system. However, our work differs from Backes et al.'s in that it seeks to patch crypto misuses instead of adding a permission system, automates the patching process and installation of patched applications to a higher degree and is capable of dealing with new technologies such as *App Bundles*[1], *Split APKs*[2], dynamic feature modules and their consequences for app patching.

Other solutions for patching Android applications and/or intercepting framework methods can be divided into six categories:

3.2.1 Rewriting Dalvik Bytecode or Machine Code

This option usually involves modifying an application's DEX file(s) and the contained Dalvik bytecode instructions. Since this works on unmodified stock Android devices, it is generally the most popular approach. Its disadvantages are that some method calls (e.g. calls to system methods from native code) cannot be captured and that modified applications need to be resigned, which means they cannot be installed as updates to the unmodified version, and which allows apps to detect and prevent the applied modifications.

The most similar works to our Android Patch Development Kit in this category are those of Davis et al. [17, 16] and of Ki et al. [32]. Both propose general-purpose frameworks for intercepting method calls in compiled Android applications, focussing on calls to Android APIs. Ki et al. go to great lengths for covering corner cases such as constructors and faithful super call patches, neglecting the performance effects caused by this attention to detail. As a consequence, they report that patching takes over a minute on average on a top-tier desktop machine, so is far away from being able to run on mobile devices. Davis et al. who similarly strike for very accurate patching claim their product to be more performant, averaging at a total patch time of 5 seconds on a desktop computer, but these figures remain questionable in the light of those presented by Ki et al. of a similar approach on newer hardware.

Several solutions employ Java bytecode instrumentation tools for intercepting method calls in Android applications. Although this allows reusing existing infrastructure, it also requires a translation layer between Dalvik bytecode and Java bytecode, which adds considerable runtime and memory consumption overhead during the patching process. Hao et al [28] employ this approach to reuse the *BCEL* Java bytecode manipulation library for implementing an Android app instrumentation tool, while Arzt et al. [2] and Ali-Gombe et al. [1] utilise aspect-oriented programming and the AspectJ system to this end.

Yet another possibility for implementing Dalvik bytecode rewriting is taking advantage of the *apktool* program for transforming the DEX file into the *Smali* intermediate representation[3], which can then be parsed into memory as an `Abstract Syntax Tree` (AST) for easy manipulation. Dai et al. [15] and Liu et al. [35] follow this approach for more convenient register reorganisation when injecting logging calls into sensitive APIs in order to detect malicious behaviour in compiled apps. What is preventing

[1] About Android App Bundles: https://developer.android.com/guide/app-bundle/
[2] What a new publishing format means for the future of Android: https://medium.com/googleplaydev/2e34981793a
[3] Smali Wiki: https://github.com/JesusFreke/smali/wiki

this concept from being applicable for on-device patching is its high memory usage for the AST and the runtime overhead of Smali parsing.

Applications making use of Dalvik bytecode rewriting are manifold. Lee et al. [34] use it to enforce a more fine-grained permission system onto compiled Android apps by inserting a call to a security check function before any of an activity's methods. Rasthofer et al. [43] follow a more sophisticated approach with the same goal, implementing a specification language and static analysis for precise control over data flows from sensitive data sources to sinks. Jeon et al. [30] patch apps to reroute all sensitive API accesses through their policy decision application and remove the permissions from manifests as part of the patching, effectively creating a fail-safe default scenario for their custom permission system.

Several publications use this approach for patching security vulnerabilities in third-party applications. For example, Zhang et al. [61] and Xie et al. [56] developed solutions that specifically target inter-app vulnerabilities, where a malicious application collaborates with a benign but insecure or another malicious program to extract user data.

Rewriting machine code, i.e. the native parts of Android apps, is rarely attempted, since it is much more complex than rewriting Dalvik bytecode as Ha et al. [27] discuss in their 2018 paper *REPICA*.

3.2.2 Adapting the Android Framework

Some solutions opt for modifying the Android framework itself in order to run custom code in place of or in addition to some method within the framework. While this is usually the most reliable and performant solution, it generally comes at the big cost of requiring a custom-built Android version or a rooted device.

A common use case for this concept is implanting a more fine grained permission system. Nauman et al. [40] modify the Android package manager to allow selectively granting permissions and add additional mechanisms for the user to specify detailed policies for limiting runtime access to critical resources based on factors like current time and location or frequency of previous accesses. Heuser et al. [29] realised a more radical change to the existing Android permissions model, presenting a permission system composed of authorisation hooks spread throughout the Android framework and Linux kernel that can be controlled by user-installable *Security Module* applications. Backes et al. [3] suggest a very similar approach, except that Security Modules are designed as a more trustworthy part of the system that can also completely override system default policies. The *MOSES* system by Russello et al. [45] implements a system for maintaining separate software-isolated profiles that have different security policies attached to them for controlling whether and how data generated and linked to one profile can be accessed from another. Finally, Wu et al. [55] went even further and show a lightweight solution for running untrusted Android applications in an isolated virtual runtime side-by-side on an actual Android device.

Modifying the Android framework components also allows researchers to monitor unmodified applications' behaviour for the purpose of identifying malicious activities. To this end, Enck et al. [19] modified the Dalvik virtual machine and the Binder library to implement taint tracking for monitoring whether Android applications leak sensitive private data. In a similar spirit, Cho et al [11] adapted the runtime to log all instructions it executes. This allows to analyse code dynamically loaded or decrypted only at runtime.

You et al. [59] work around the requirement of a custom and/or rooted OS by bundling modified system components with the target app and manually restarting the application process with a custom version of Zygote. However, their solution adds a startup delay to the app, is more prone to break with new Android releases and still requires resigning the target app, thus inheriting the same disadvantages as the code rewriting approach described above.

3.2.3 Manipulating Runtime Structures

Another possibility is using native code to overwrite function pointers in the internal data structures of the Dalvik and ART runtimes. In contrast to the bytecode rewriting approach discussed above, this allows rewriting framework methods instead of just their call sites. Additionally, the technique is capable of intercepting system library calls from JNI code and patching essentially just involves injecting a native library, allowing for very performant patch deployment. Unfortunately, because it is dealing with internal data structures instead of a public API, the approach is very prone to break with new or modified Android versions and it also still requires either repackaging and resigning target apps or root privileges.

Von Styp-Rekowsky et al. [52] presented this approach for the Dalvik runtime in 2013, injecting policy checks into arbitrary apps that go beyond Android's native permission system. While they chose to repackage target applications, Fan et al. [22] decided to inject their runtime modification code for the ART runtime into running processes using *ptrace*. Their system then collects information about the target app, such as calls to security-sensitive APIs. A very similar approach was presented by Mulliner et al. [38] for attacking Google Play in-app billing.

Popular hooking frameworks for modifying applications and system components on rooted devices, such as *Cydia Substrate*[4] and *Xposed*[5], are also part of this category. They modify the Zygote process to inject their code into every started Java process, where they then manipulate the internal runtime structures to implement their hooking functionality. In an academic context, Costamagna et al. [13] presented a very similar general-purpose solution that utilises Samsung's *Android Dynamic Binary Instrumentation* toolkit to inject their native library into arbitrary processes.

3.2.4 Intercepting libc or System Calls

Android being a Linux-based system, most important operations involve a call into some libc function that in turn performs some system call. Modifying the pointers in the *Global Offset Table* (GOT) maintained by the dynamic linker allows to intercept calls to libc, which can be used for hijacking IPC, among others. However, given the low-level nature of the libc functions in question, much of the higher-level context has to be manually reconstructed, which is prone to fail with future OS revisions. Additionally, this solution either requires a rooted device or repackaging and resigning the target application.

Backes et al. [4] implemented a sandboxing mechanism for unmodified apps on unmodified systems by executing third-party applications inside a low-privileged isolated process controlled by a central broker running in a normal Android application. The broker uses libc hooking in order to intercept all system calls made by the target application and is able to reconstruct and modify Binder IPC transactions. The broker could then be used for enforcing security policy decisions. A similar system was realised even earlier by Xu et al. [57], but it applied the libc GOT modification code from within the target process itself, in a native library repackaged into the target application.

To a similar effect, Russello et al. [46] and Zheng et al. [62] present solutions that use ptrace for intercepting system calls and implementing an improved permission system or observing potentially malicious applications. Bianchi et al. [7] managed to work around the requirement for root access when employing ptrace in their sandboxing solution, by executing the target application as a child process to a generated stub application.

[4]Cydia Substrate: http://www.cydiasubstrate.com

[5]Xposed - General info: https://forum.xda-developers.com/xposed/xposed-installer-versions-changelog-t2714053

3.2.5 Container Applications

Recent years have seen the emergence of a new category of Android applications that allow to run third-party applications within a container application through clever use of Java's *Dynamic Proxy* API[6] and proxying Android framework components. Although it allows to modify the execution flow of an application without manipulating the application package, this solution has major shortcomings. As extensively documented by Zhang et al. [60], it effectively bypasses the Android OS's application isolation, because a single host application communicates with the system on behalf of all its contained guest applications. Since for the OS, the host and its guest form one entity, they all have access to the same app data folder and share their permissions. In order to prevent a malicious virtualised application from accessing the other application's data, the host solution would have to take additional safety measures, which all tested commercial software failed to address. Moreover, the technique adds a runtime overhead noticeable on the device's battery life and limits integration of virtualised apps into the OS.

Despite the severe security implications, several implementations are readily available to end consumers through Google Play (e.g. *Parallel Space*[7]) and as open source libraries (e.g. *VirtualApp*[8]) for integration into third-party products.

Dai et al. [14] published a compilation of knowledge and research around the topic, covering attack vectors from the perspective of the host and guest, conceptual design, available implementations and possible mitigations. Complementarily, Luo et al. [36] provide more detailed documentation on the concepts required for designing such a system, and recommend methods for apps to actively prevent being executed in a virtual container.

The work by Xuan et al. [58] illustrates how a malicious host application can hijack legitimate applications in order to obtain user data such as login credentials.

3.2.6 Hybrid Solutions

Finally, some publications use a combination of some of the aforementioned techniques in an attempt to compensate one solution's weaknesses with another's strengths.

Zhou et al. [63] utilise manipulation of runtime structures in conjunction with native code rewriting in order to prevent target applications from breaking out of the sandbox through a JNI library. This allows them to construct a much tighter system for enforcing security policies onto the target application without requiring modifications to the OS or root access.

In a similar fashion, the *DeepDroid* solution presented by Wang et al. [53] takes advantage of both runtime structure manipulation and system call interception in order to enforce enterprise policies in Dalvik bytecode and native parts of third-party Android applications. In comparison to Zhou et al.'s work, this approach permits more control over native code while still not requiring modifications to the Android system libraries. However, it does need root privileges.

[6]Dynamic Proxy Classes: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html
[7]Parallel Space: https://play.google.com/store/apps/details?id=com.lbe.parallel.intl&hl=en
[8]VirtualApp: https://github.com/asLody/VirtualApp

Chapter 4

CryptoPatcher System Overview

The purpose of this chapter is to provide a high-level introduction to our contributions presented in this thesis.

In Section 4.1, we formulate and justify the objectives chosen for the rest of this thesis. We draw the line from the problem stated in Chapter 1 to the various mitigation possibilities discussed in Chapter 2, before explaining what is required to develop our solution in a way beneficial for future research in slightly different fields.

The following Section 4.2 is dedicated to presenting our approach as the solution to the objectives stated earlier. We report how the need for flexibility lead us to the structure of the project and highlight how every part is tailored to cater to a specific portion of the overall goal.

4.1 Objectives

From the various studies summarised in Section 1.1, it becomes evident that a considerable number of Android applications leave their user's data at risk by not properly implementing security. More specifically, this includes vulnerabilities that make mistakes in the use of TLS/SSL for securing communication over a network, ciphers for encryption and decryption of data, password-based encryption and random number generators.

While Google as the de-facto owner of the Android platform has slowly started to take measures, they still are not addressing the wealth of applications that are not regularly updated despite being in real-world use. Additionally, some application developers keep maintaining their code, but simply lack the skillset required for properly implementing cryptography. This situation is particularly unacceptable for corporate organisations, where leaking sensitive information could put the company's existence at stake. As long as the Android OS does not integrate any form of automated security upgrade to potentially insecure third-party applications, an external solution is the best bet for protecting the data of users who depend on the operation of programs that misuse cryptography APIs.

Although it would be possible to develop a custom variant of the Android OS in an organisation outside of the Android Open Source Project, such an endeavour could never reach the critical mass of end consumer devices. Given the largely fragmented Android ecosystem and device revisions released in tight product cycles, maintaining a third-party operating system that covers all possible handsets represents a task that could only be accomplished with substantial financial resources. Even if such an aftermarket operating system could be provided for the end user's device, manufacturers typically sell their devices tightly sealed, so that frequently, security vulnerabilities have to be exploited in order to replace the preinstalled firmware image. While such a procedure can be followed by enthusiasts willing to take a risk, the task represents an insurmountable challenge for the typical consumer, and is even linked with legal consequences under

some jurisdictions. Together, these troubles effectively rule out the option of developing and rolling out a custom OS for large-scale organisations.

Since Android is based on a Linux kernel, root privileges on the device are enough for applying many modifications to the OS. However, because most vendors do not grant the owner of the device root access, the goal of performing system modifications through this route can usually only be achieved by exploiting a security vulnerability. Even if this feat could be accomplished, a rooted environment means that any misbehaving program can have grave consequences for the whole system. For rollout to the device fleet of a company, this option cannot be seriously taken into consideration.

Given that modifying the OS and/or utilising root privileges are no viable options, the remaining methods have to be carefully evaluated. As elaborated in Section 3.2.1, all sandboxing approaches that do not require root permissions severely restrict the integration of applications into the system. Consequentially, an invasive solution must be chosen, which mitigates the security vulnerabilities inside the application code itself.

Still, the system needs to protect against security vulnerabilities present in any of the apps installed on a device. The process of mitigating an installed application's vulnerabilities must happen as transparently to the user as possible, so that no manual intervention is required for keeping the system secure. If an enterprise rolls out the protection solution to the devices of its employees, it is critical that the latter do not have to be instructed on its operation.. The solution must also be capable of covering applications installed from any source, while ensuring that updates through the most commonly used distribution channel Google Play still work without any interferences.

Furthermore, the ideal solution works on the device itself, without depending on an external server for modifying target applications' execution flow. Android devices can be utilised in remote areas or restricted environments, where the functionality of installed applications must be guaranteed despite lack of a network connection. Obviously, for mitigations to the TLS API, which itself only makes sense in a networking scenario, a web service can be accessed without limiting the applicability of the overall solution.

In addition to providing automatic protection for unexperienced consumers, more advanced users and administrators will be interested in the possibility to learn more about the APIs an application accesses, and possibly use the acquired knowledge as the basis for making decisions about the trustworthiness of a given application. It is thus desirable to integrate functionality for monitoring the cryptographic primitives and their parameterisations used by target applications, as well as an option for bypassing the added security checks after having confirmed from the logs that an application can be trusted. This capability might be critical for apps that for some reasons fail to operate when subjected to CryptoPatcher's automatic mitigations.

Although the use case in this thesis is strictly focused on mitigating vulnerabilities induced by misuse of cryptographic APIs, it is very likely that similar principles can be applied to the fight against other classes of vulnerabilities commonly found in Android applications. It is thus desirable to structure the project in a way that permits later adaptation for other use cases. Similarly, the system needs to be able to accommodate to novel techniques for rewriting application bytecode that might arise in the future, so that developers do not need to reimplement their patches in this case.

To summarise, the objectives of this thesis are:

- Mitigation of security vulnerabilities induced by misuse of cryptographic APIs in compiled third-party Android apps
- Implementation of a solution applicable to as many devices as possible, without requirement for modifying the OS or obtaining root privileges
- Providing automatic protection so as little manual intervention as possible is needed

- Realising a standalone system that executes on the target Android device itself
- Allowing advanced users to gain insights into an application's API usage and to modify the default policy
- Developing a flexible design that can be adapted both in terms of covered security vulnerabilities and rewriting technique

4.2 Approach

We designed the CryptoPatcher system to meet the goals stated above. With the need for high flexibility and adaptability in mind, we structured it in three components, each adding features more tailored to the specific task of automatically patching crypto API misuses. Figure 4.1 displays how these three components interact inside the CryptoPatcher system.

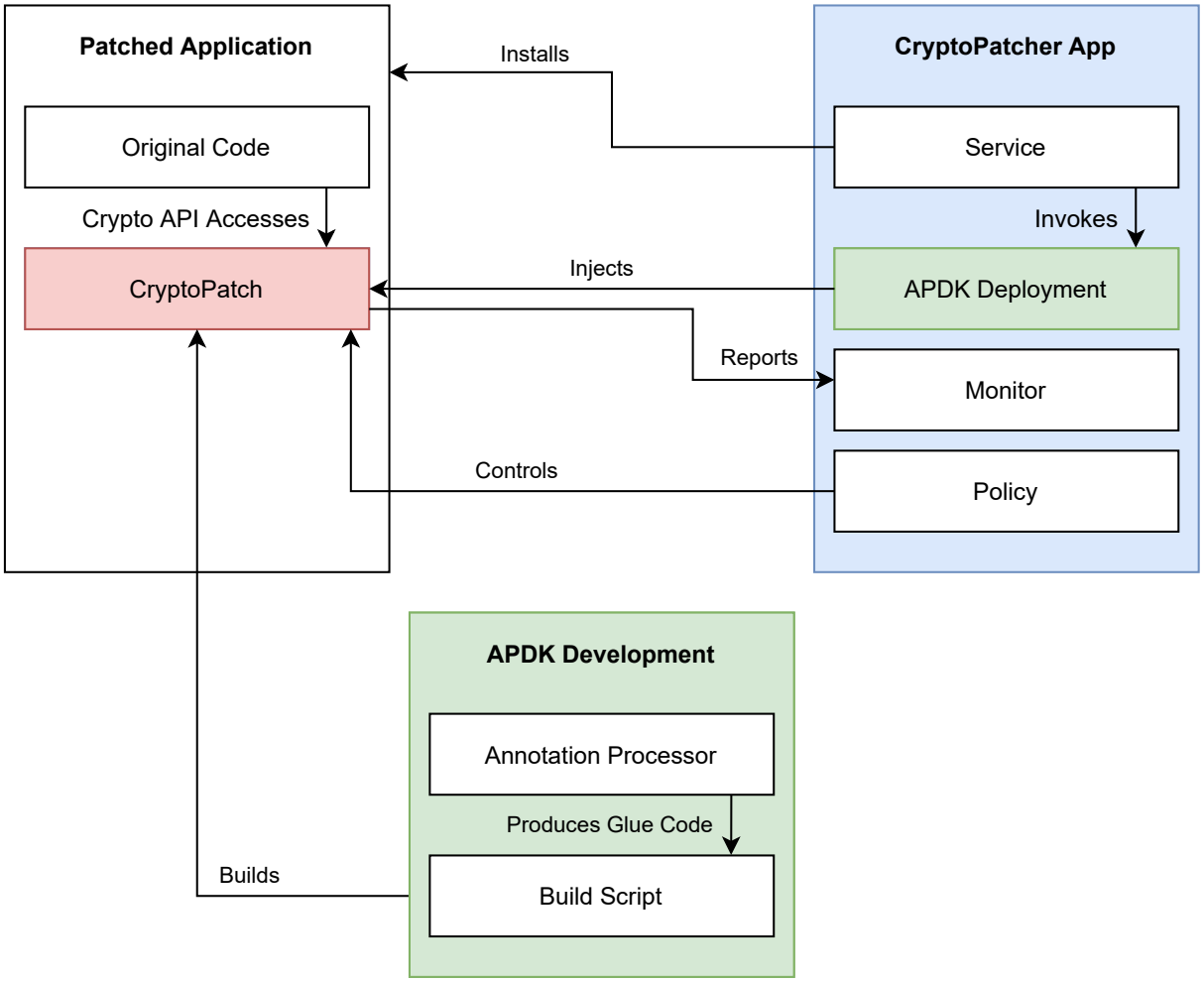

Figure 4.1: The three main components of the CryptoPatcher system (encoded in different colours)

4.2.1 Android Patch Development Kit

The *Android Patch Development Kit* (APDK) implements the functionality required for developing and deploying patches for third-party Android applications that are only available in binary form. Maintaining

these parts of the CryptoPatcher system individually lays the foundation for the flexibility required for applying the developed concepts onto different research questions. Our APDK provides patch developers with means to intercept arbitrary method calls in the Dalvik bytecode, modify the Android manifest, add resources or inject native libraries.

The Dalvik bytecode rewriting and patch deployment infrastructure is flexible enough to develop patches completely agnostic to the target application, meaning a compiled patch can be applied to any application. Additionally, the Dalvik rewriting backend can be exchanged transparently to the patch code. Our system currently includes a stable static rewriting backend that modifies the actual Dalvik bytecode and an experimental dynamic rewriting backend that implements the runtime structure manipulation approach for the ART runtime.

Our APDK is also equipped with a patch deployment system designed in a performant and portable fashion, so that it fits the constrained performance envelope of an Android device. It includes the whole tool chain needed for unpacking, modifying, repacking and signing an Android application. By injecting the patches into the target application's package, patch deployment does not require any privileges apart from a way to install repackaged apps onto a device.

4.2.2 CryptoPatch

The CryptoPatch patch is the part of the CryptoPatcher system responsible for implementing the specific mitigations for the covered misuses of cryptographic APIs. It consists of Dalvik bytecode patches and Android manifest modifications that add an entry point called by the Android framework when the target app is started and gives CryptoPatch a chance to initialise its state.

CryptoPatch's bytecode patches intercept calls to the `SSLSocket` API for TLS/SSL, the `Cipher` API for encryption or decryption, the `KeyGenerator` API involved in deriving keys for Password-Based Encryption and the `SecureRandom` API, as well as a few utility classes and alternative frontends to the same functionality. These are the four API classes that are most frequently subject to misuse and can be mitigated transparently to the application code.

In addition to automatically mitigating crypto API misuses wherever possible, the patch also collects call parameters on a set of additional methods of the Android framework, so that the CryptoPatcher application can be supplied with information for its monitor display.

4.2.3 CryptoPatcher

The CryptoPatcher application is implemented as an Android program that can be installed on unmodified devices. It consists of a service that is started immediately after booting the OS and a user interface that allows managing the installed patched packages and monitoring events.

By listening for installation events broadcast by the system, the background service can automatically start the patching process whenever the user installs or updates an application on the device. To this end, the patch deployment functionality of the APDK is directly integrated into the CryptoPatcher application.

In order to obtain permission for automatically installing patched application packages, the CryptoPatcher service utilises the *Device Administrator* API[1]. Originally targeted at enterprise *Mobile Device Management* (MDM), this API grants elevated permissions to a single application on the device. Since this represents a possibility for obtaining some degree of control over the system on a device running an official production build of the Android OS, it is a perfect solution compatible with a broad range of devices. Original applications are disabled but kept on the device, so that the common update flow via Google Play remains unaffected by the CryptoPatcher protection.

[1]Device Administration overview: `https://developer.android.com/guide/topics/admin/device-admin`

CryptoPatcher's monitoring user interface is tailored towards the requirements of two rather contrary user groups. For curious but inexperienced amateur users, it provides high-level descriptions for an application's API accesses. Events that were found to put user data at risk are highlighted in colours that encode the severeness of the issue. For advanced operators, CryptoPatcher augments the simplified output with extensive details about the specific framework function that was called, along with the exact parameterisation chosen. Based on this information, experienced administrators can then disable CryptoPatcher's mitigations in cases where an application whose trustworthiness was manually confirmed does not work properly when patched, for example because it communicates with a corporate server that cannot be reached by CryptoPatcher's TLS notary web service.

Chapter 5

Android Patch Development Kit

In this chapter, we shed light on our Android Patch Development Kit (APDK) from various perspectives. To begin with, Section 5.1 lays down the requirements that were set up at the start of the APDK's design and how they dictated the decisions taken during implementation. Section 5.2 continues with a brief user documentation of the APDK system, exploring how patch developers can take advantage of the APDK's capabilities in terms of patching Android manifest and DEX files or injecting native libraries and resources. Section 5.3 details some of the technical challenges we encountered during the development of the APDK and the solutions we found in order to overcome them. Lastly, Section 5.4 concludes the chapter.

5.1 Introduction

In Chapter 4 we identified the need for developing a flexible patching infrastructure in order to be able to apply the technological knowledge gained in this thesis to other similar fields in future work. The Android Patch Development Kit is the sensible step in order to meet this goal. In the following, we lay out our objectives for this general-purpose patching system and discuss the design chosen for accomplishing the stated mission.

5.1.1 Objectives

Fundamentally, we require our patching framework to be capable of intercepting framework method calls made in application code. This mechanism is what enables the later implementation of the CryptoPatch patch. More specifically, we are interested in rerouting control flow to an injected wrapper method when a specified framework method is called. From the wrapper method, the patch developer should then have the possibility to add custom functionality in addition or in replacement to that of the originally called method. Since most patching goals applicable to a broader range of third-party programs can be accomplished by interposing on public methods, we can ignore private methods for this matter. We are thus focussing on public static methods, virtual (public instance) methods and constructors. Ideally, calls to one of the aforementioned functions can be intercepted independently of the specific call procedure, i.e. even when called via Java *Reflection*, from native code, or as a super call from an instance to a method of one of its ancestor classes.

In addition to the ability to modify control flow, many patches will want to add their own components to an existing application. Since the Android OS requires installed applications to declare their main interaction points in the manifest file, there is an evident need for the capability to perform modifications to this file at the time a patch is applied. Specifically, we want to be able to add, modify or remove elements and attributes at precise locations inside the manifest's XML structure.

Similarly, feature-rich patches commonly present their own user interfaces. For example, as described in Section 3.2.1, several publications injected an extended permission system into applications that displays a

dialog asking for user's permissions on certain actions. In order to accommodate such projects, our system needs a way to add resources into an application. Further pursuing this idea, it would be desirable to also let patches add native libraries into existing applications, in case they need to perform a performance-critical task or want to leverage an existing C or C++ code base.

For the purpose of the CryptoPatcher project, it is essential that developing a patch is entirely decoupled from deploying it to an application. Only if patch development does not require insights into the implementation of a target app, it is possible to produce patches compatible to a wider range of programs. Patch deployment also needs to be implemented with a particular focus on portability and performance, since it has to run in environments with very tight memory and processing resources on Android devices.

Still, we want patches targeting methods of a class to be inferred onto subclasses in the framework as well as those declared inside a target application's code. Only by covering this case as well can we ensure a particular method is reliably intercepted. For example, some Android APIs allow application developers to pass a callback to the framework. We want to be able to intercept this callback, even if the specific implementation resides in the target application.

All of these features must be made available to patch developers in a way that is easy to understand with experience in Android development. To this end, it should integrate and make use of existing tools and assist the developer in preventing bugs. For example, the patch compilation should fail if the patch targets a method that does not exist or tries to pass arguments of the wrong type. Otherwise, applying the patch will either crash the target application or just not effect any change at all. Still, it must be possible to add libraries to the compilation classpath in case a patch optionally wants to intercept methods with parameter types not defined by the Android framework.

5.1.2 Approach

We designed the Android Patch Development Kit to meet the requirements stated above. It completely decouples patch development from its deployment, thus allowing the development of generic patches applicable to any third-party Android software. Figure 5.1 provides a rough overview of the process.

5.1.2.1 Patch Development

Patches are developed inside the *Android Studio Integrated Development Environment* (IDE) Android application developers are already accustomed to. They are compiled from a set of Java source files containing static methods that method calls will be redirected to. The method to be intercepted is specified via a set of Java *annotations*. Resources and native libraries can be included just like in an application project. Modifications to Android manifest files are formalised in a special set of XML elements loosely based on the *XML Patch* format as specified in *RFC 5261*[1].

The core component of the patch development process is the APDK annotation processor. It is invoked as part of compilation and performs multiple jobs. Firstly, it checks whether all methods targeted by the patch actually exist in the respective framework class. It then produces *glue code* for linking the developer-supplied patch code with the exchangeable rewriting backend. The two currently supported rewriting methods are compatibility-focused *static rewriting* via Dalvik bytecode modification at patch deployment and more performant *dynamic rewriting*, which works through manipulation of ART virtual machine data structures at runtime.

For compiling patch packages, the standard *Gradle* tool is utilised, but a custom build script modifies the process, so that two different ZIP archives are generated from the same patch source, one for each of

[1] An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors: https://tools.ietf.org/html/rfc5261

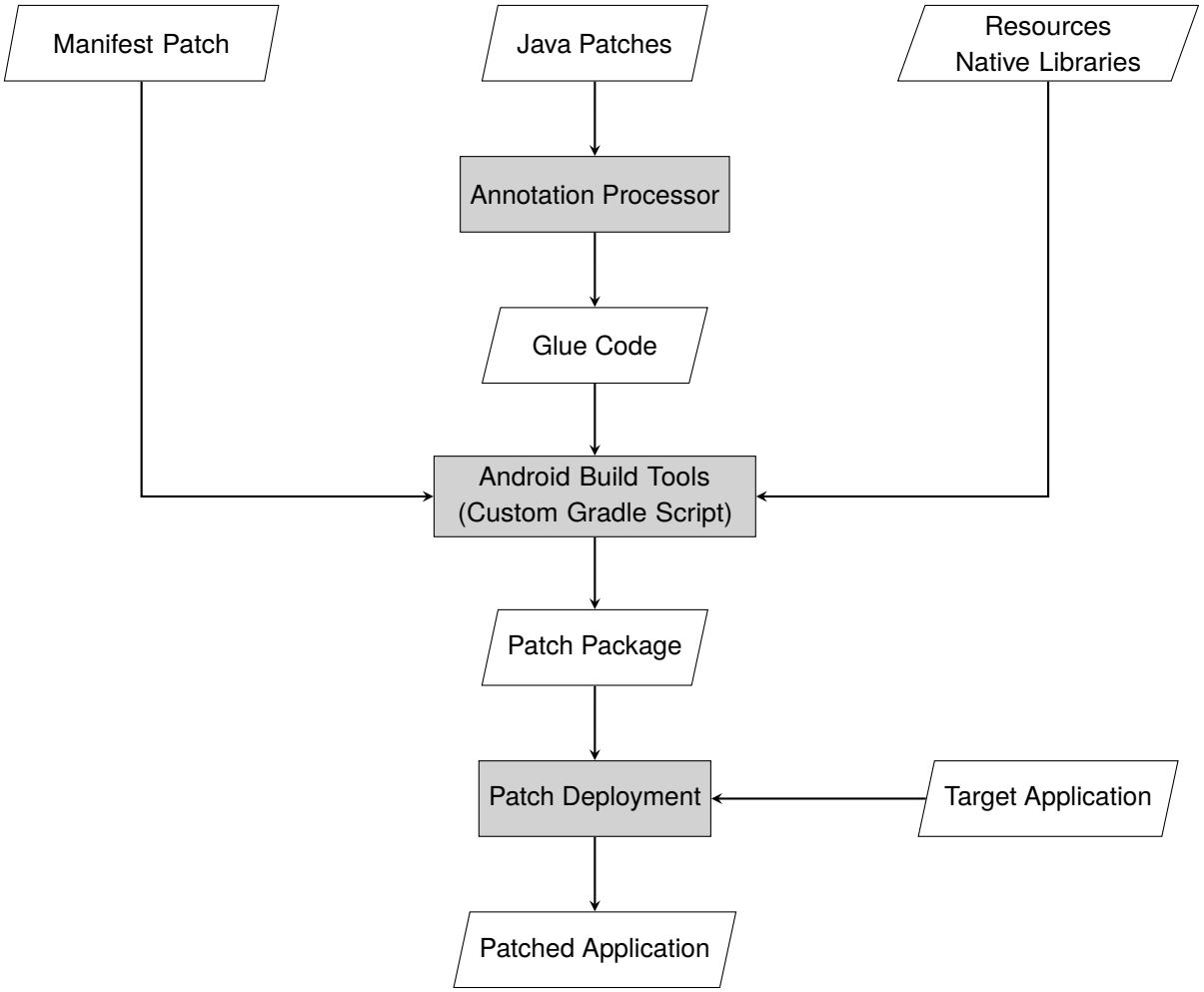

Figure 5.1: The build and deployment process of an APDK patch

the supported rewriting backends. Every ZIP archive includes a complete description of the patch ready for deployment.

5.1.2.2 Patch Deployment

While both rewriting backends share the deployment steps of unpacking the target application before and repacking and resigning it after applying the modifications, they differ considerably in the way their Dalvik bytecode modifications are applied.

The static rewriting backend infers subclass patches while the patch is applied to the target application. This involves iterating through all classes contained in the application's DEX files and examining their inheritance structure. If a class is found to inherit a method targeted by a patch, its corresponding method is intercepted analogously. While executing the inference during patch deployment is a very reliable solution and does not affect runtime performance of the produced application, it noticeably increases the duration the user has to wait for the patched program to be operational.

For this reason, the experimental dynamic rewriting backend defers patch inference to the target application's runtime. During patch deployment, only an extra DEX file and an additional entry point for loading it are injected into the application package. The added components then infer patches onto classes as they are loaded by the runtime. While this speeds up the patch deployment considerably, the added workload slightly slows down the target application and is much more prone to fail, since it manipulates runtime structures not meant to be externally tampered with.

5.2 Usage

Patches utilising the Android Patch Development Kit are developed as Android application projects in the Android Studio IDE. Since the system currently requires a specific project structure, patch developers are advised to base new patches on a copy of our template project. This ensures the different dependencies, build scripts and tools are correctly configured.

The organisation of the patch project follows the same structure as that of a regular Android application project. Dalvik bytecode patches are developed inside Java classes and annotated so that the build system and deployment mechanism can identify the respective target methods. The modifications to the Android manifest file are noted inside the manifest file of the patch project, using a special *Domain Specific Language* (DSL) based on custom XML elements. Lastly, resources and native libraries reside in their usual folders.

5.2.1 Android Manifest Patches

Modifications to the Android manifest file of a target applications are expressed as changes inside the manifest XML file of the patch project. The syntax and semantics for formalising changes are loosely based on RFC 5261, but were adapted for compatibility with the proprietary Android binary XML format. The format defines a change in terms of a selector for specifying the precise subject of the modification (either an element, an attribute or a text node) and a description of the actual operation to be performed. Available options are deletion, replacement or insertion of elements or attributes.

The selector is formulated as an *XML Path Language* (XPath) statement. In their most basic form, XPath expressions represent a path into the XML tree structure, very similar to how file system paths address a specific file in a folder structure. However, XPaths can make use of a number of very powerful extensions, such that they can reference a node's attributes or position relative to another node.

For each of the supported operations, a specific XML element can be placed inside the patch XML. The `<add sel="..." pos="..."/>` tag is used for adding attributes or elements to the selected node. Its `sel` attribute contains the selector, while an optional `pos` attribute allows specifying where exactly in relation to the selected node or its child nodes the added node should be placed. Possible values are `prepend`, `before` and `after`. If elements are to be added in this operation, they are passed as the child nodes of the `<add/>` node in the patch XML. In case of the absence of child nodes, all the attributes of the `<add/>` node other than `sel` or `pos` will be copied to the selected node.

The `<replace sel="..."/>` element is particularly powerful. When targeting an element, it will be replaced with the first child node of the `replace` element in the XML patch document, including all its attributes and children. While the `replace` tag can additionally be used for text nodes, its true strengths show when being applied onto attribute nodes. In this case, the replacement value (the text content of the `replace` tag) can contain *placeholder expressions*. The `$${xpath(...)}` placeholder allows including the result of an additional XPath expression evaluated in relation to the replaced element, `$${globalize(...,...)}` forms a fully-qualified class name from a relative class name (second argument) and its package name (first argument). Lastly, the `$${appendeach(..., ...)}` placeholder splits its first argument string at semicolon characters, appends the string passed in the second argument to every obtained substring and rejoins them with a semicolon character. Placeholder expressions can be nested, so that very complex combinations are possible.

Finally, the `<remove sel="..."/>` element permits patch developers to delete the selected element, attribute or text nodes.

Figure 5.2 shows an example patch utilising all tags introduced above. Its first element adds a new metadata node to the manifest element. The three following patch elements demonstrate the use of the replace tag and the supported placeholder expressions, modifying the package name, resolving relative

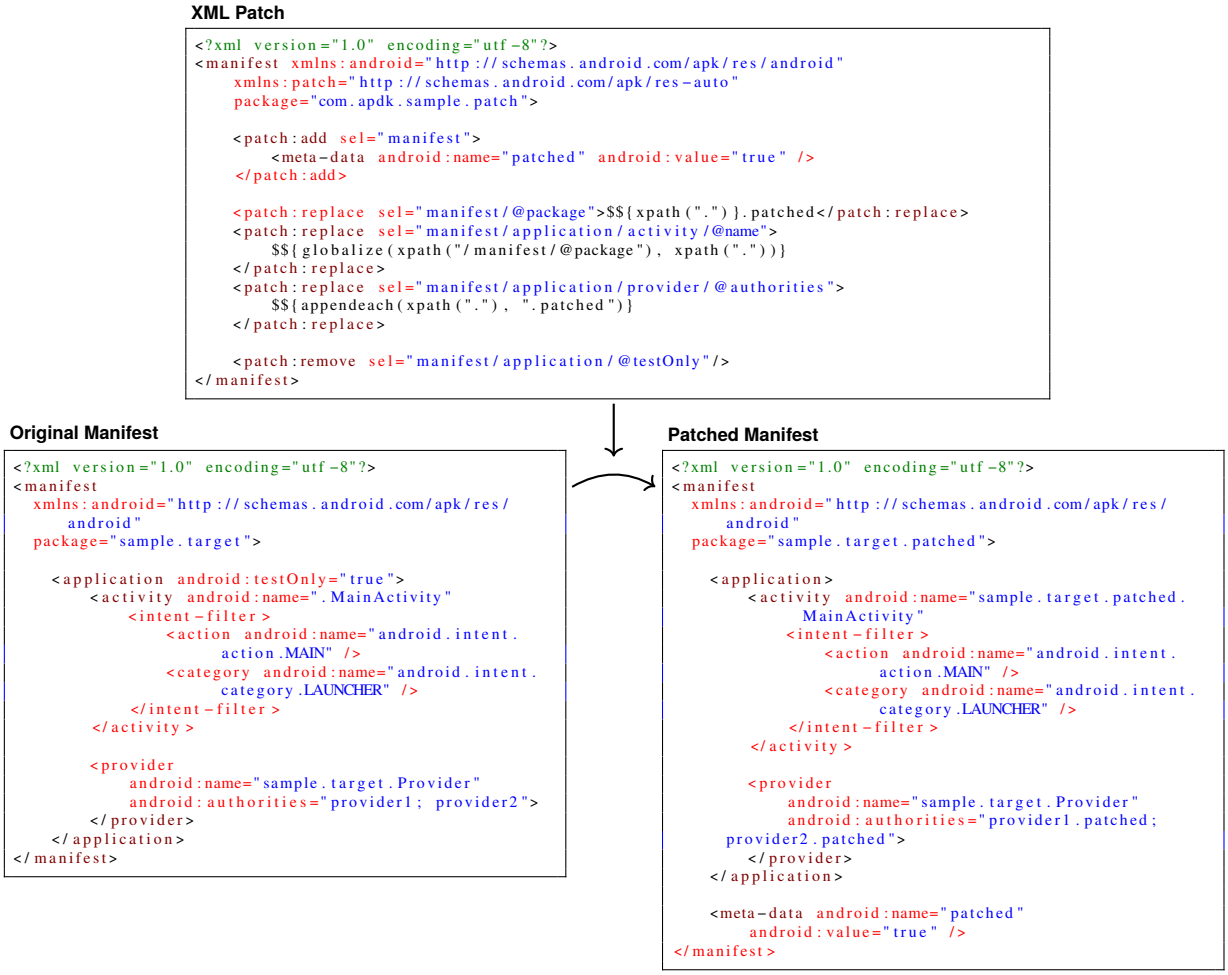

Figure 5.2: Applying an XML patch to the Android manifest file

activity names and altering the authorities property of all content providers. Lastly, the `testOnly` attribute is removed from the application node in the manifest.

5.2.2 Java Code Patches

Dalvik bytecode methods can be patched via a system of custom annotations that, enriched with information generated by the APDK's annotation processor, encode all information needed during patch deployment.

For every method that patch developers wants to intercept, they have to write a public static wrapper method exposing the same argument types, the return type and the thrown exceptions as the targeted method. In case an instance method is targeted, an additional parameter of the instance type has to be prepended to the argument list. The APDK currently offers `@PatchStaticMethod`, `@PatchInstanceMethod` and `@PatchConstructor` annotations, each for specifying a target method of the type implied by the respective name. All of these annotations permit passing an optional String argument encoding the target method's name. If not explicitly specified, the target method is assumed to carry the same name as the patch wrapper method.

In addition to the target method name and signature, the patch deployment components also need to know the class that defines the target method. To this end, an additional `@PatchClass("...")` annotation is available, which can be applied to either an individual wrapper method or a wrapper collection class containing multiple wrapper methods that all target the same class. Generally, wrapper methods may be spread throughout the whole project, although the use of wrapper collection classes is highly recommended

for maintaining a clean project structure. In case a target method is not fully qualified, does not exist or its parameters or thrown exceptions differ from those specified in the wrapper method, the annotation processor will throw an error during compilation.

Code inside a patch wrapper method can call through to its original target method, although they have to make use of special infrastructure to do so. For every class targeted by a wrapper method, the APDK annotation processor will add an inner class inside the generated `OriginalMethods` class that is named after the fully-qualified class name of the target, except that dots will be replaced with underscores. This inner class will expose a proxy method for any of the target class's original methods. For example, a wrapper method targeting `java.io.File.getName()` will make the annotation processor produce an inner class `OriginalMethods.java_io_File` containing a method titled `getName()`. This layer of indirection is needed for being able to transparently exchange the used rewriting backend. The example patch utilising this mechanism in Listing 5.1 intercepts calls to the `Intent.setData()`, `Intent.setDataAndType()` and `Intent.addFlag()` instance methods..

```
@PatchClass("android.content.Intent")
public static class IntentPatch {
    @PatchInstanceMethod
    public static Intent setData(Intent thiz, Uri uri) {
        return OriginalMethods.android_content_Intent.setData(thiz,
            patchContentUri(uri));
    }

    @PatchInstanceMethod
    public static Intent setDataAndType(Intent thiz, Uri uri, String
         type) {
        return OriginalMethods.android_content_Intent.setDataAndType
            (thiz, patchContentUri(uri), type);
    }

    @PatchInstanceMethod
    public static Intent addFlags(Intent thiz, int flags) {
        OriginalMethods.android_content_Intent.setDataAndType(thiz,
            patchContentUri(thiz.getData()), thiz.getType());
        OriginalMethods.android_content_Intent.addFlags(thiz, flags)
            ;
        return thiz;
    }
}
```

Listing 5.1: Java patch example

Sometimes, a patch needs to intercept calls to a method whose argument or return types are not defined by the Android framework. When written using the techniques described so far, such a patch would fail to compile, not least because the annotation processor would fail to confirm the target method's existence. As a mitigation to this problem, the APDK provides an additional `@PatchType("...")` annotation that allows to concretise an object's type to the APDK, while identifying it with a more generic type to the Java compiler. A JAR or DEX file can then be supplied to the annotation processor via the gradle build file, so that it can still ascertain the existence of the targeted method. After such a patch is deployed to an application through static rewriting, the resulting Dalvik executable will only reference the annotated type if it was already referenced in the original bytecode.

Although best efforts are being made to offer the same functionality for all rewriting backends, the patch developer is advised to keep in mind there are some inherent differences to the currently supported techniques that cannot be worked around automatically, so particular corner cases are best avoided when writing patches.

For example, there are limitations to the methods that a patch can target. Specifically, virtual methods have to be targeted at the class that first defines them. If a particular subclass's implementation is to be intercepted, it is the wrapper method's responsibility to check the instance type at runtime, and selectively apply modifications depending on the found type. Additionally, constructor patching is currently only supported by dynamic rewriting, because it would require computationally expensive register reorganisation in the static rewriter.

Moreover, there is a difference in how the rewriting backends handle different call types of the intercepted methods. While dynamic rewriting correctly captures method invokes via Reflection, from native code or as part of super calls, static rewriting does not support them at all.

From our experience, these limitations do not hinder the APDK's practical utility in any way because they can be worked around relatively easily for most use cases. Support for Reflection can be added to the static rewriter, but was considered out of scope for this thesis.

5.2.3 Resources and Native Libraries

Developers can integrate resources and native libraries into their patch project just like they would in an application project. Please note that it is currently not possible to modify or remove existing resources of a targeted application package.

5.2.4 Deployment

All of the APDK's deployment functionality is implemented inside a Java library for convenient integration into other projects. The CryptoPatcher distribution provides two programs utilising this library, namely the CryptoPatcher application and a general-purpose utility program for applying patches to an APK file on a desktop computer via a command-line interface.

The main interface to the APDK's deployment functionality are its `ApkPatcher` and `SplitApkPatcher` classes. For starting the deployment process, these components have to be supplied an instance of the `ApkPatchTaskInfo` or `SplitApkPatchTaskInfo` classes. These structures are responsible for holding all data required as input and produced as output of the process. Both `*Patcher` classes support publishing detailed progress reports to a listener specified by the client application.

5.3 Implementation

Most of the custom functionality in the build system for APDK patch projects resides in the annotation processor. The artefact resulting from patch compilation is later fed to the patch deployment process along with a target application, which may consist of a single APK file or a Split APK collection of a base package and several secondary APK files. In the latter case, deployment processing is largely the same as for a self-contained application, except that all steps have to be carried out on every individual file. Patch deployment starts by unpacking the application package. Since an APK file is essentially a ZIP archive, Java's standard ZIP infrastructure can be utilised for this task and the corresponding repackaging that takes place after the patch has been applied. Following the unpacking of the target APK file, a similar procedure is applied on the patch package. The patch resources and native libraries are then copied into the unpacked target package, before the manifest and DEX files are patched. After repackaging, the zipsigner[2] open source project is employed for signing the resulting APK file. The following sections detail some of the implementation challenges that had to be overcome in the course of the project.

[2]Official source of the *ZipSigner* app for Android: `https://github.com/kellinwood/zip-signer`

5.3.1 Annotation Processor

The APDK annotation processor is invoked during patch compilation. Its job is to examine the organisation of the Java patch code and generate glue code for linking it with the used rewriting backend.

As its first task, the annotation processor constructs an inheritance graph of all classes in the Android framework and classpath libraries, along with their declared methods. To this end, it is capable of parsing Java or Dalvik bytecode in *Java Archive* (JAR), *Dalvik Executable* (DEX), *Optimised Dalvik Executable* (ODEX) or *Ahead Of Time* (OAT) files through the integration of the *ASM*[3] and *dexlib2*[4] open source libraries. It is the patch developer's responsibility to pass the relevant files to the annotation processor. At a minimum, the simplified framework file contained in the Android SDK distribution must be supplied. From the obtained data, the annotation processor is then able to check the correct parameters, return types and thrown exceptions of all patch wrapper methods.

Next, the annotation processor generates additional Java files. Independent of the selected rewriting backend, an additional layer of wrapper methods termed *glue methods* is added, all merged into the same container class and explicitly annotated to simplify patch deployment. These glue methods will be called in place of an intercepted method at runtime and forward execution to the wrapper methods provided by the patch developer.

Additionally, the processor loops through all classes affected by a patch wrapper method and infers the patch onto relevant methods of subclasses inside the Android framework or classpath libraries. The inference information is saved to the glue methods as additional annotations for consumption by the patch deployment components.

The rest of the Java code produced by the annotation processor largely differs between the two supported rewriting backends. For the static case, the `OriginalMethods` class produced by the annotation processor just consists of simple calls to the original method. This simple solution is possible since static rewriting only affects the code of the target application. In contrast, dynamic rewriting equally applies to the target program code and the patch code, so simply invoking the target method from the patch wrapper would lead to a stack overflow. Instead, each proxy method in the `OriginalMethods` class generated for this case calls a backup method that points to the original implementation as found before applying the patch.

5.3.2 Manifest XML Patching

As elaborated in Section 5.2.1, the APDK supports specifying modifications to the target application's Android manifest file in terms of an XML patch file. While the general concepts are loosely based on RFC 5261, the format had to be heavily adapted for extending its feature set and ensuring its compatibility with Android's proprietary binary XML format.

In order to facilitate the XML patch deployment, the APDK starts by parsing the compiled manifest into its own in-memory tree representation. For this purpose, a custom parser was implemented on top of the open-source *axml* project[5].

While the tree representation already exposes manipulation functionality, the data is converted into plain XML format before applying the XML patch. This is achieved by serialising the structure into plain text XML and parsing it into an instance of Java's XML DOM `Document` class. The additional transformation was made necessary for being able to take advantage of existing Java infrastructure for evaluating XPath expressions on the manifest XML structure.

[3] ASM Java Bytecode Manipulation Framework: `https://asm.ow2.io`
[4] dexlib2: `https://github.com/JesusFreke/smali/tree/master/dexlib2`
[5] axml: `https://github.com/Sable/axml`

Still, extra measures have to be taken for mirroring all information from the binary format into the plain XML document. In an effort to speed up parsing at runtime, the binary XML format includes an integer identifier and a type qualifier for every attribute. The APDK retains this metadata by injecting *shadow attributes* into the plain XML data, so it does not have to completely regenerate this structure when transforming the patched XML data back into the binary format. Since the patch XML is stored inside the manifest file of the patch project, it is automatically converted to binary XML format during compilation and goes through similar processing as the target manifest as preparation for the patching. This ensures that nodes copied from the patch to the target manifest while deploying the patch are already enriched with proper shadow attributes.

An additional obstacle that has to be overcome for manifest patching are resource references. The Android framework allows many attributes in the manifest to be filled with references to a resource instead of explicitly hardcoding the value. For example, this enables developers to display the app name in the language of the user. However, a patch may want to modify such a value or use it as part of its XPath selector. In order to support this use case, the APDK resolves all resource string references before applying the patch, and adds shadow attributes for the original references and values. Once the patch has been deployed, the APDK scans through the resulting XML structure and identifies values that were modified, so that the changes can be propagated back into the referenced resource. Please note that this functionality has not yet been fully implemented because it is not needed by our CryptoPatch patch. In particular, the current implementation stores the modifications inside the Android manifest for performance reasons, instead of propagating them back to the resources file.

Furthermore, the manifest patching infrastructure collects a mapping between original and patched `ContentProvider` authorities and adds them to the manifest in a dedicated metadata field. The field can then be queried by the injected Java code at runtime. These modifications are necessary in cases where the patched application is to be installed alongside the original version, because applications can only be installed if their contained `ContentProvider`'s authorities do not collide with those of a package already known to the system.

As a minor caveat worth mentioning, our XML patching only follows a very minimalistic strategy for handling XML namespaces in the patch. In particular, entries added to the target manifest from the patch can not introduce new namespaces. However, given that Android manifests generally only contain a very limited set of known namespaces, this should not affect real-world use in any way.

5.3.3 Rewriting Backend Differences

Because the APDK attempts to provide an abstraction layer that separates the patch code from the underlying rewriting technology, it has to establish a common ground between the supported rewriting backends. However, the two backends work fundamentally different. Static rewriting manipulates control flow at the call site of a method during patch deployment, replacing the reference to the method that is to be invoked. On the contrary, dynamic rewriting operates at runtime, modifying the target method's implementation itself.

The most salient consequence of these differences can be found in the way the two backends handle class inheritances. Given that the concrete type of an object is usually only known at runtime, static rewriting, performing light static analysis, in many cases can only determine one of the more generic ancestor classes. Because of the lack of more detailed type information, a method patch can only be applied to all subclasses of the target class, independent of whether they override the implementation. On the other hand, a method patch applied through dynamic rewriting only affects subclasses as long as they do not override the target method's implementation.

In order to bridge these gaps, we stipulated patch semantics that both rewriting backends are capable to support and enforce them in the annotation processor. All virtual method patches have to aim at the class of the target method's first declaration, i.e. the most generic class exposing that function, and further

inspect the runtime instance type from there. With regards to static rewriting, this condition alone ensures that calls to a more specific subclass implementation cannot be missed even if the object is cast to one of its ancestor types. The APDK's annotation processor then infers the patch onto all subclasses of the targeted class in the Android framework, so that cases with more specific type information are covered as well. Lastly, the inference is extended onto classes of the target application during patch deployment (static rewriting) or runtime (dynamic rewriting).

As an example, consider the class structure depicted in Listing 5.2 and Listing 5.3 and assume a patch wants to target the `FileInputStream.read()` method. Because the call in `MainActivity.onCreate()` casts the concrete `BetterFileInputStream` instance to its `InputStream` ancestor type, it would escape static rewriting if `FileInputStream.read()` is targeted directly. Instead, the APDK requires `InputStream.read()` to be targeted by the patch, because that is the place the method is first defined. While this properly captures the call in `MainActivity.onCreate()` when deployed with static rewriting, dynamic rewriting fails to do so because the `BetterFileInputStream` class overrides the `read()` method and thus does not share the implementation with its ancestor class. Additionally, intercepting only `InputStream.read()` still misses the call in `MainActivity.onPause()` even for static rewriting, because the created object is cast to the more concrete `FileInputStream` type there. In order to cover these case as well, the APDK needs to infer the patch onto all subclasses of the generic target class in the framework (`FileInputStream`) and in the target application (`BetterFileInputStream`).

```
abstract class InputStream {
  abstract int read();
}

class FileInputStream extends InputStream {
  @Override
  int read() {
    ...
  }
}
```

Listing 5.2: Rewriting backend differences example: Framework code

```
class BetterFileInputStream extends FileInputStream {
  @Override
  int read(){
    ...
  }
}

class MainActivity {
  void onCreate() {
    InputStream input = new BetterFileInputStream(new File(...));
    int value = input.read();
  }

  void onPause() {
    FileInputStream input = new BetterFileInputStream(new File(...))
        ;
    int value = input.read();
  }
}
```

Listing 5.3: Rewriting backend differences example: Application code

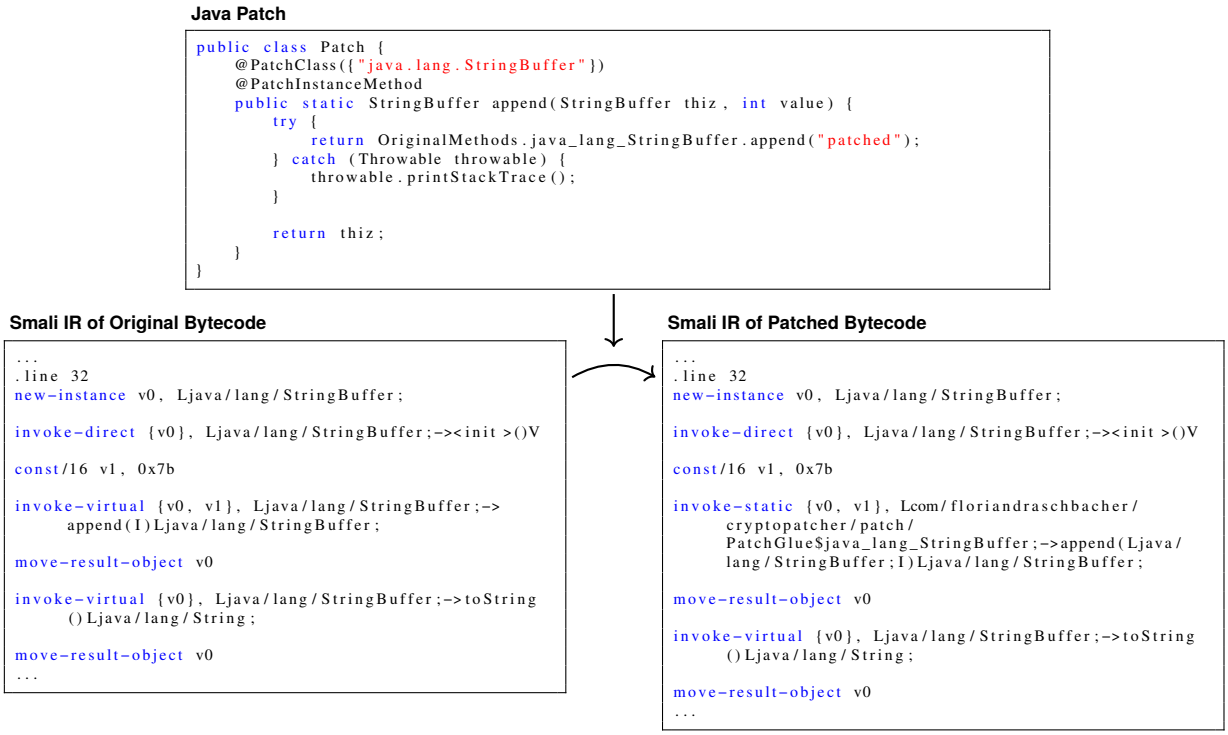

Figure 5.3: Applying a Java patch through static rewriting

5.3.4 Static Rewriting

In contrast to most existing Android patching frameworks implementing DEX modifications, the APDK's static rewriting backend does not disassemble Dalvik bytecode into the Smali *Intermediate Representation* (IR) and operate on a convenient in-memory representation of that format. Instead, it opts for a streaming design based on the lower-level *dexlib2* library, where patches are applied on the bytecode directly and most data is read from the DEX file only at the time it is needed. This ensures that the process consumes a bare minimum of memory on the device at any time.

As the first step in the static rewriter's patch deployment, the DEX file inside the patch package is parsed. Specifically, the APDK deployment components extract annotations in order to construct a list of all contained glue methods and the respective target functions. Next, the program iterates through all classes defined in the DEX files of the target application. For every class that extends a known target class, the corresponding patches are extended onto the subclass. As described in Section 2.4.5, the classes listed inside a DEX file are ordered by dependency so that the whole operation can be accomplished as a single pass.

A target application's control flow is diverted by modifying the call points to a target method. To this end, the static rewriter iterates through all bytecode of the source application. Once a call to one of the targeted methods is found, the `invoke-*` instruction is replaced with an `invoke-static`, and the referenced method is exchanged for the corresponding static glue method. Since `invoke-virtual` calls implicitly pass the instance as their first argument, the corresponding static patch glue and wrapper methods' parameters need to be explicitly augmented accordingly in this case. These basic principles allow the implementation of a very performant yet powerful system.

Figure 5.3 illustrates how a Java patch expressed through the APDK's annotation system is applied to the Dalvik bytecode inside the DEX file via static rewriting.

However, there are some cases that the static rewriting backend is incapable of handling, due to limitations inherently linked to the technique or for reasons of performance tradeoffs that had to be made

in the implementation. Most notably, static rewriting is unable to intercept constructor invocations. This restriction is caused by peculiarities of the virtual machine in regards to uninitialised objects. As illustrated in Listing 5.4, constructors are usually called via an `invoke-direct` immediately after a new instance is allocated. If the APDK was to replace the `invoke-direct` with an `invoke-static`, it would have to pass the uninitialised instance as an explicit argument. However, the Android runtime specifically disallows passing an object as a function parameter before it has been initialised by a constructor. A possible workaround is calling the glue method in an added invoke instruction immediately after the constructor, but that would involve compute-intensive register reorganisation. Alternatively, an additional wrapper class could be added whose constructor calls the glue method, although this requires more thorough static analysis for retaining the class hierarchy and would thus decrease patch deployment performance considerably.

```
new-instance v0, Ljava/util/ArrayList;
invoke-direct {v0}, Ljava/util/ArrayList;-><init>()V
```

Listing 5.4: Smali IR of object creation in Dalvik bytecode

Another more subtle deficiency of the static rewriter concerns how `invoke-super` instructions are handled. These are used for and in fact limited to cases where an instance method invokes a method of one of the instance's ancestor classes. If the APDK was to intercept these by replacement with an `invoke-static` to a glue function, it could potentially destroy the program's control flow irreparably. This is because although the APDK offers wrapper methods the possibility of calling the original method they replaced, these calls always make use of the `invoke-virtual` instruction, which has different semantics in terms of the vtable lookup than `invoke-super`, as discussed in Section 2.4.5.2. A wrapper method reached through an `invoke-super` would thus end up in an infinite recursion if it called the original method. As a remedy, the APDK leaves `invoke-super` instructions untouched by static rewriting.

5.3.5 Dynamic Rewriting

The dynamic rewriting backend reduces application package modifications to a minimum, deferring patch inference in the target application's code to its runtime. During patch deployment, an additional DEX file and native library are injected into the target package. Furthermore, a `ContentProvider` is inserted into the Android manifest file so the added dynamic rewriter components can be initialised before the application code starts executing.

During runtime initialisation, the dynamic rewriter extracts the patch code from the injected DEX file, parsing the annotation data for obtaining information regarding the target methods. It then loads the list of framework classes preloaded in the Zygote process that the Android OS stores inside a special publicly-readable system file. From this information, the dynamic rewriter is able to infer patches not unlike it is done in the static rewriter, except that the list of preloaded class names is not sorted by dependencies. As a result, we have to manually confirm patches have been inferred onto all ancestor classes before processing a given subclass. Once all preloaded classes have been dealt with, a function in the `ClassLinker` component involved in runtime class loading is hijacked, so that we can install a listener that is invoked immediately before a new class is first used. It is through this listener that patches can be continually inferred to cover all loaded classes.

The inference of patches onto framework classes at runtime is necessitated by the fact that the `android.jar` framework library included in the Android SDK distribution lacks some classes that are not considered part of the public interface and simplifies inheritance structures. This poses no problem for static rewriting, because the Dalvik bytecode generated by the Android build tools will only reference the public framework classes known at compile time. However, it is problematic for dynamic rewriting, which operates on the much more detailed type information available at runtime. As an example, the abstract `android.content.ContentResolver` class in the framework is public, but its most commonly

used concrete implementation `android.app.ContextImpl$ApplicationContentResolver` is not, and thus excluded from the `android.jar` file. Consequentially, patches written for `ContentResolver` methods can not be transferred to the `ApplicationContentResolver` class by the annotation processor. At runtime, calls to `ContentResolver` methods are actually resolved to the corresponding concrete subclass methods, so will not be intercepted if the patch was not inferred onto the concrete class.

Method rewriting itself is accomplished through manipulation of internal *Android Runtime* (ART) data structures. Since all the runtime and virtual machine libraries reside inside the same process as the application code, the latter can inspect and modify the former's internal state. More specifically, the APDK's dynamic rewriter employs the *SandHook*[6] library for taking control of the `ArtMethod` structures the runtime maintains for the target methods. To start with, the whole structure is copied to a backup method, which can later be used for invoking the original method. Next, the APDK makes use of the *libffi*[7] project for generating a native closure function in executable memory and sets the function pointer in the `ArtMethod` structure to its address. When executed, this closure function forwards its received parameters and a static userdata pointer to a specified dispatcher function, where the originally called method is identified through the userdata and the corresponding wrapper method is invoked via the *Java Native Interface* (JNI).

The dynamic generation of a separate closure function for every intercepted method is crucial to the correct operation of patch inference. Because no new Java methods are added during inference at runtime, multiple classes end up sharing a single wrapper function. However, this is problematic when the latter wants to execute the original method it intercepted, which requires explicitly referencing the backup method mentioned above and thus effectively bypasses the normal virtual method resolution through the vtable. This problem is very similar in nature to that found in `invoke-super` instructions for static rewriting, but due to the greater leverage of the dynamic approach, it can be solved. The implementation of the `OriginalMethods` class for dynamic rewriting maintains a mapping from fully qualified original method name to backup method that is filled as part of method hooking. However, when one of the `OriginalMethods` methods is executed, it still needs to know the method that was originally called in order to look up the corresponding backup method. This is where the generated closure function comes into play. Because every intercepted method is effectively turned into a dynamically generated native method that calls through to the patch glue method, the originally called method remains visible on the call stack. The `OriginalMethods` implementation can thus use the call stack element at a known offset as the key for its lookup into the backup methods map.

While it may seem that a similar effect can be achieved in a simpler way, this is actually not true when inspected more closely. The function pointer contained in an `ArtMethod` instance could simply be exchanged to point to the wrapper function directly. However, this approach turns out ineffective on recent Android versions due to the fact that (as briefly discussed in Section 2.5) method calls are frequently inlined by the ART compiler for performance reasons, so that the `ArtMethod` structure does not have to be consulted. As a remedy, Android hooking frameworks usually resort to overwriting the first machine code instructions of the compiler-generated code for the target method, inserting a jump to a trampoline that in turn redirects execution to the (JIT- or AOT-)compiled version of the hook method. However, this process effectively replaces the call stack entry of the originally called method with that of the glue method in the course of the latter's invocation, voiding its applicability for our project.

Please note the dynamic rewriting backend is still considered to be in an experimental state and not working reliably. In particular, many patched third-party applications crash with a memory fault immediately after startup. While the issue appears to be linked with the lookup of the `OatQuickMethodHeader` runtime structure for a patched method, we were so far unable to pinpoint the exact cause in a reasonable time frame due to the difficulty of debugging release builds of the ART runtime.

[6] Android ART Hook/Native Inline Hook/Single Instruction Hook: https://github.com/ganyao114/SandHook

[7] A portable foreign-function interface library: https://github.com/libffi/libffi

5.3.6 Resources

The requirement of the possibility for patches to add resources into the target application posed another technical challenge when developing the APDK system. As discussed in Section 2.4.3, the Android build tools bundle all the application resources into an ARSC file and assign integer identifiers for referencing them in code. Since these identifiers are hardcoded within the instructions referencing them, very elaborate static analysis would be needed for changing them. Because the APDK utilises the standard Android tool chain for patch development, all the patch resources and their references in code are processed in the same way as those of the target application. As the resource identifiers are assigned by the build tools from a per-type counter starting at zero for every application, simply merging the ARSC index of the patch into that of the target application is not possible due to identifier clashes. Although the identifiers in the patch ARSC could be modified relatively easily, the values hardcoded in the patch code can not.

As the solution to this problem, the APDK configures the *aapt* resource compiler from the Android build tools to assemble the patch project's resources in a package with ID `0x8f`, diverging from the default ID `0x7f` used in normal applications. As described in Section 2.4.3, the resource IDs include the containing package ID, so that this modification prevents resource identifiers of the patch package from clashing with those of the target application.

Patch deployment then just involves copying the package chunk from the patch's ARSC file to that of the target application. Additionally, all referenced strings have to be copied between the string pool chunks of the two files, and the references in the patch package chunk are adjusted accordingly.

5.4 Conclusion

In this chapter, we introduced our custom patching framework that was tailored specifically to the requirements identified for the CryptoPatcher system, albeit with the clear goal of delivering a solution that can be conveniently adapted for further use in related fields. The need for a clear separation between patch development and deployment was justified as key to a wide applicability of patches. Additionally, we demonstrated how developers can leverage the capabilities of the APDK for implementing their own patches capable of modifying the Android manifest file and Dalvik bytecode, as well as injecting resources and native libraries into third-party applications. These possibilities form the foundation for enabling the addressing of specific cryptographic API misuses in the CryptoPatch patch described in the remainder of our thesis.

Chapter 6

CryptoPatch and CryptoPatcher App

This chapter covers both the CryptoPatch patch and the user-facing CryptoPatcher application. Since the two components are very tightly coupled and their depiction from various perspectives perfectly complement each other, they perfectly lend themselves to a holistic description.

We start with an overview of the objectives for the two components in Section 6.1, before we discuss the approach that enabled us to meet the stated goals. Next, we focus on the CryptoPatcher application from a user perspective, detailing its configuration and usage in Section 6.2. We then dive into the technicalities of our specific implementations in Section 6.3, elaborating the technical challenges experienced when realising the patches for TLS, Cipher, PBE and Secure Random API misuses. At the end of the chapter, Section 6.4 provides a short conclusion of the covered aspects of the CryptoPatcher system.

6.1 Introduction

Most of the general requirements for the CryptoPatch patch and the CryptoPatcher application are congruent with or directly derived from those already described in Chapter 4 for the CryptoPatcher system as a whole. This section will thus only provide more detailed reasoning and refine the technicalities of the specific patches for mitigating crypto API misuses and their deployment within the CryptoPatch application.

6.1.1 Objectives

As a guideline for the security vulnerabilities we intend to address, we use the publications discussed in Section 1.1. While we overall strive for correcting as many application flaws introduced by misuse of cryptographic APIs as possible, there is a need to do so transparently to the application code and the user. This is because of limitations to the degree an application can accommodate functionality not originally provisioned for by its original authors. For example, in order to mitigate an application's use of a constant *Initialisation Vector* (IV) for symmetric block ciphers, it would have to be made aware of an additional token (a proper random IV) that needs to be supplied to both decryption and encryption of the correct ciphertext and corresponding plaintext. Our CryptoPatch aims at staying invisible to the user, so that existing use flows of target applications are not affected by the patching. Consequentially, we do not consider it a viable solution to inject an additional user interface for users to manually provide input to the patch, which stresses the need for an alternative solution.

It is also crucial for our system to offer the possibility to pass over all collected data such as call parameters for display in the user interface. This is because there are cases where a dynamic approach as chosen by CryptoPatcher is unable to determine whether a given operation poses a security risk or not. Consider an application's use of a hardcoded encryption key. While dynamic analysis is able to recognise the repeated use of the same key, there is no way to distinguish between a static key securely derived from unique system properties and one that is baked into the application code and thus the same for all

installations of the program. While this distinction cannot be made in an automated fashion, expert users can often make educated guesses based on previous experience, if they have access to detailed information regarding the operation in question.

Besides, our patch needs to cover all possible API endpoints to the cryptographic primitives addressed. For some functionality, such as HTTPS communication, a wide range of different open source middleware libraries exists. However, as described in Section 2.6.2, the `HostnameVerifier`, a crucial component of the Java TLS stack for ensuring the confidentiality of a connection, has to be invoked from the application code, since it is not automatically called by the framework during the handshake. Consequentially, the invocation can happen in various different ways and places, making it difficult for a patch to bring it into relation with the `TrustManager` used for the same connection. However, it is only with knowledge of all parameters that an authoritative judgement about the trustworthiness of a connection can be made. A clever solution has to be discovered for this problem.

Other requirements are more specific to the exact security vulnerabilities we want to cover. For TLS patching, we need a way to effectively prevent *Man-in-the-Middle* (MITM) attacks, even if the original application code does not perform proper server certificate validation. Since the server may employ a self-signed certificate, simply injecting a default implementation based on Public-Key Infrastructure could render the target application inoperational. In order to solve this problem, some publications as discussed in Section 3.1 proposed to build on the Trust-On-First-Use principle, which assumes that the application receives the legitimate certificate from the server at the time of their first contact. However, we believe that this base assumption is inherently flawed, so that an alternative solution needs to be found.

6.1.2 Patch Approach

From those common security vulnerabilities that were found to be induced by crypto API misuses in Android applications, we identified four different classes perfectly eligible for patching transparently to the application code and user. In the following, we describe how the objectives stated above dictated the design of our patch.

6.1.2.1 TLS/SSL

As documented by multiple sources summarised in Section 1.1, Android applications very commonly make some mistake when implementing secure network communication via the TLS/SSL protocol family. Specifically, they frequently utilise a `TrustManager` that trusts all servers irrespective of the validity of the presented certificate or implement `HostnameVerifiers` that do not properly ensure that the subject of the server's certificate matches the host name it was accessed under. As a consequence, application users are left unprotected to MITM attacks compromising potentially sensitive information such as login credentials.

In order to correct this flaw in a way that covers all possible implementations, the CryptoPatch patch opts for a practical approach based on an assessment whether an attacker is actively intercepting a given connection, instead of just judging about the possibility of such an incident to happen. This provided us the opportunity to reliably target any possible use of TLS connections in an application, irrespective of the specific HTTP stack it builds upon.

More specifically, our solution employs a trusted notary web service that is assumed to remain uncompromised under any circumstances. Whenever a target application has established a TLS connection, our notary web service is queried for the legitimate certificate of the contacted server. If the response does not match with the certificate directly obtained from the server, the connection is immediately dropped, without ever having sent application data through the compromised channel. By performing this check immediately before the target program starts sending application data, we know that the former considers the connection secure. Combined with our information about the legitimacy of the connection, we can take this knowledge as a basis for deducing whether an application uses insecure certificate validation logic.

6.1.2.2 Ciphers

Another set of common vulnerabilities is linked to an application's use of the `Cipher` API. In particular, developers frequently fail to specify a mode of operation for symmetric block ciphers, which causes a fallback to *Electronic Code Book* (ECB) mode. As discussed in Section 2.1.2, ECB mode is considered insecure for applications that encrypt multiple blocks of data with the same key. An additional problem is software that repeatedly uses the same combination of key and IV for any block cipher configuration.

In order to mitigate these problems, our CryptoPatch solution is capable of automatically upgrading ciphers that use ECB mode to the more secure CBC mode. Additionally, it collects a database of previously used IVs, so that it can detect reuses and autonomously generates a proper random value in this event. In order to supply the same IV to the decryption functionality, it is embedded within the ciphertext.

6.1.2.3 Password-Based Encryption

A similar mis-parameterisation issue exists with regard to Password-Based Encryption, where it is crucial to initialise the Key Derivation Function with an adequate iteration count and a unique random salt. Programs that fail to meet these basic requirements put their user's passwords at risk of getting compromised through basic brute-force attacks.

While an insecure iteration count can trivially be upgraded in our patch, injecting a secure salt in place of a constant is currently not supported. The only channel that could be used for transporting the same random salt to the decryption as was used for encryption would be embedded inside the ciphertext. This idea however poses major challenges because it is difficult to bring the salt in relation to the produced ciphertext within the patch code. The reason for this lies buried in the design of the specific APIs involved here. For the `Cipher` API, the IV is set directly on the object that is later used for encryption or decryption. For PBE however, the salt is used for generating the key in an entirely separate operation and can not be extracted at the place where the ciphertext is produced.

6.1.2.4 Secure Random

The `SecureRandom` API on the Android OS has had a history of severe flaws. As documented by Google [12], in releases prior to Android 4.2, if an application developer explicitly seeded the random number generator, the supplied value would replace the default entropy derived from the system state. As a result, applications that used a constant seed risked all generated randomness to be predictable. Although this original issue was fixed in Android 4.2, the introduced changes opened the door to even more severe problems. Specifically, the system failed to properly seed the random number generator in fresh application processes forked from the Zygote process, so that applications could predict each other's supposedly secure random values. The developers of the Android OS [33] even acknowledged that the vulnerability has been actively exploited for compromising a bitcoin transaction. This incident underlines the importance of ensuring proper seeding of a *Cryptographically Secure Pseudo Random Number Generator* (CSPRNG). Although both flaws have been corrected in recent versions of the Android OS, what remains invariantly important for today's application developers on any platform is avoiding the bad practise of seeding a CSPRNG with a constant seed.

Since the CryptoPatcher application that automatically deploys our CryptoPatch patch takes advantage of functionality for its automated operation that only became available with Android 5.0, we chose not to specifically address the two vulnerabilities described above that concern earlier revisions of the OS. Instead, we strive to detect an application's reuse of CSPRNG seeds as an indicator of a general lack of diligence from the developer's part. In the event of a discovered reuse, the CryptoPatch patch will automatically prevent the explicit seeding and instead fall back to the system-provided entropy generation, which is generally considered secure enough for most applications.

6.1.3 Application Approach

The key to enabling the envisioned degree of automation in the CryptoPatcher application is its implementation as a special device administrator program destined for the role of the device owner. This special position on the device permits our program to install or disable packages without requiring the usual interaction from the user. In combination with the on-device patch deployment functionality inherited from our APDK, our solution is thus equipped with all tools required for autonomously generating patched versions of third-party applications and installing them to the system. The original software is only disabled, not removed from the device, so as to prevent its execution while still supporting the common update flow through the Google Play store.

When it comes to the structure of the user-facing CryptoPatcher application, a particular focus of our effort was closely following the idioms and concepts present in other parts of the Android OS. The goal was for our application's screens to resemble user interfaces that consumers are already accustomed to for managing aspects of their device similar to those dealt with by the CryptoPatcher application. In particular, that meant that the interface for managing installed patched applications was deliberately designed similar to the Android OS's application management system settings, and the screen for displaying patching progress resembles that of the system's application installer. Consequentially, users can take advantage of the knowledge already gathered for navigating the rest of the system.

6.2 Usage

In this section, we concentrate on describing the configuration and operation of the CryptoPatcher application. Since the CryptoPatch patch does not offer any immediately user-facing functionality, it is not covered here.

6.2.1 Installation

The first step of setting up the CryptoPatcher Android application is installing the APK file on the target device. In theory, our software could be deployed through Google Play to facilitate this task. Alternatively, users could just download the APK file from the Internet and manually install it on their device, although this requires explicitly enabling software installs from unknown sources in the system settings.

If any accounts have been set up on the device before, they need to be removed in order for the next step to succeed. This affects Google accounts and any other account visible in the accounts section of the Android system settings. Please note that keeping the device clear of accounts is only necessary for the duration of the installation procedure. All removed accounts can simply be restored once the CryptoPatcher software is configured.

In the next and last step, the CryptoPatcher application has to be promoted to the role as a *Device Owner* on the device. To this end, a command has to be executed on the terminal interface of the Android OS's `DevicePolicyManager`, which can be reached from a computer through a USB connection and the *Android Debug Bridge* (ADB) tool. The exact command to be issued can be found in Listing 6.1.

```
adb shell dpm set-device-owner com.floriandraschbacher.cryptopatcher
    /.CryptoPatcherDeviceAdminReceiver
```

Listing 6.1: Enabling the CryptoPatcher application as a device owner

For corporate use, the installation procedure can be automated and integrated into the device setup by utilising Android Enterprise infrastructure.

6.2.2 Basic Use

Once installed and configured on a device, the CryptoPatcher application is constantly running in a background service listening for app installation events emitted from the system. Every freshly installed application is immediately disabled, so that the user cannot accidentally mistake it for the patched version. The latter is automatically generated by deploying the CryptoPatch patch and installed through CryptoPatcher's privileges as Device Owner application. A system notification keeps the user informed of the ongoing patching. Because the original application is left on the device, the normal update flow through Google Play stays largely unaffected by CryptoPatcher's operation. Whenever the user chooses to install an app update, CryptoPatcher's service automatically picks up the change and generates a new patched version.

6.2.3 User Interface

Since the default operation already provides automatic protection, inexperienced users do not have to interact with the CryptoPatcher user interface at all. Still, it offers curious explorers and administrators multiple opportunities for gaining insight and taking control of the app's functionality.

6.2.3.1 Monitor Screen

The main screen of the CryptoPatcher application displays a list of all monitor events produced by the CryptoPatch components inside patched programs. The entries are sorted chronologically and updated automatically, so that the most recent events are always available at the very top. For every list element, the app's icon and name are displayed, so that users can trace back individual events to the respective program they originated from. Additionally, CryptoPatcher evaluates the specific protection and security status, as well as detailed information on the exact API usage in order to provide a brief summary for every event. For identified crypto API misuses, the background colour of the item in the list encodes the severity as assessed by CryptoPatcher. A yellow background signifies that the CryptoPatch patch was able to mitigate the vulnerability, while a red highlight signals that this was not possible, either because the user explicitly disabled protection for the respective application, or because technical limitations require a source-level fix from the provider of the third-party software.

Upon tapping of a list entry, the user is presented with the event details screen, providing much more comprehensive information about the specific incident. For issues related to cipher operations, the exact algorithm, key and initialisation vector (if any) are reported, while SSL communication events include the destination host name and its presented certificate chain. Independent of the specific API, the exact method and parameters that lead to the report are disclosed.

Figure 6.1 demonstrates a screenshot of the monitor list screen and the details interface that is shown when one of the events is selected.

6.2.3.2 Apps Screen

CryptoPatcher's apps screen, depicted in Figure 6.2, provides an overview of all patched applications installed on the device. Augmenting every entry with the application's icon, package name and version number provides clear distinction and allows ascertaining the installed revision at a glance.

Tapping on one of the items leads the user to a details screen for the corresponding application. Two buttons provide the possibility to access a system-provided package information screen and to uninstall the program. The following section allows users to toggle CryptoPatcher's protection for the specific application. Lastly, a list of only the monitoring events emitted from the particular software allows quickly inspecting its trustworthiness.

While a patching operation is under way, the details screen provides additional information about its progress and current state. If a problem was encountered, the user can obtain an error message from this place.

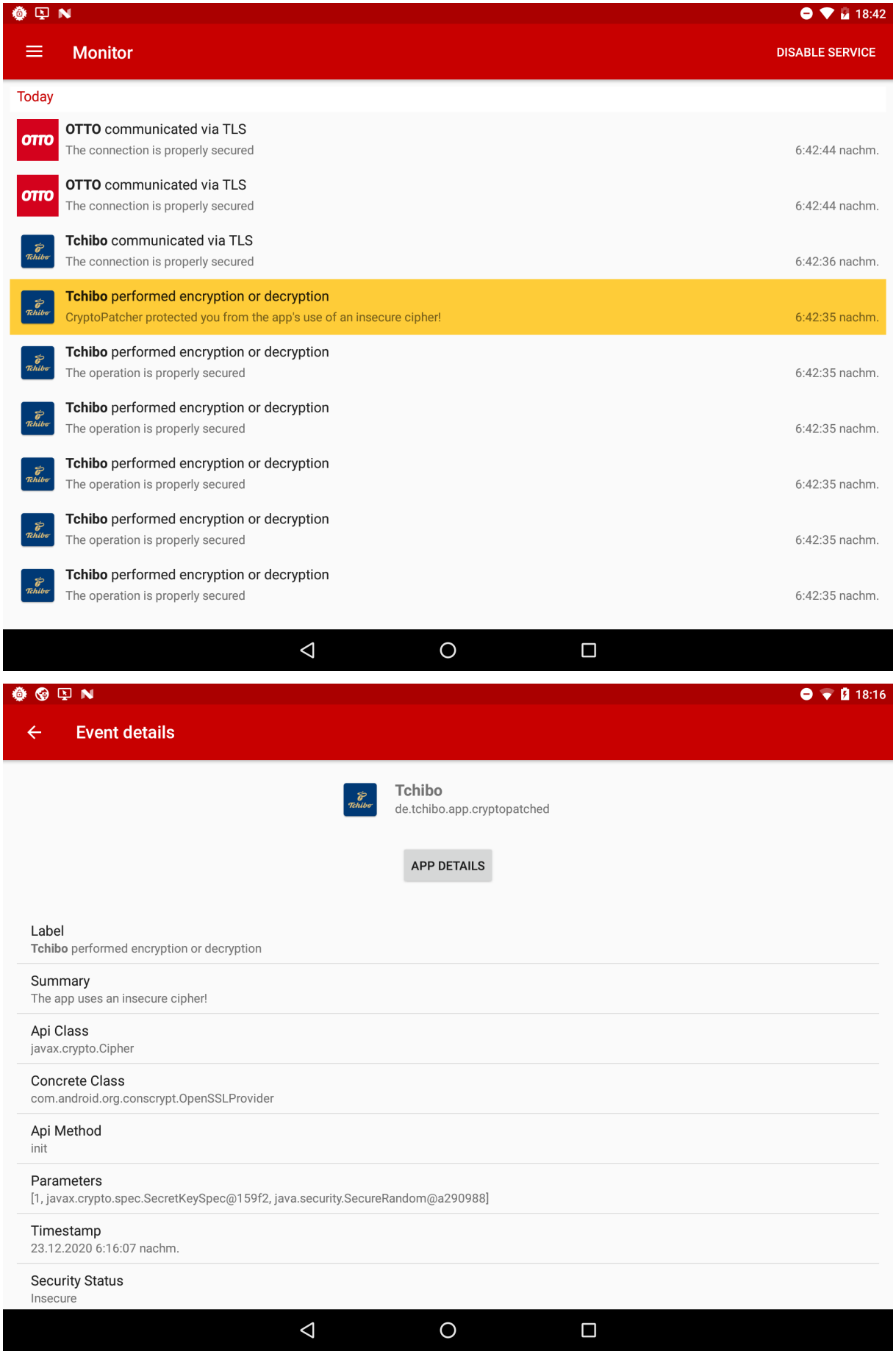

Figure 6.1: Overview and detail display of the monitor screen

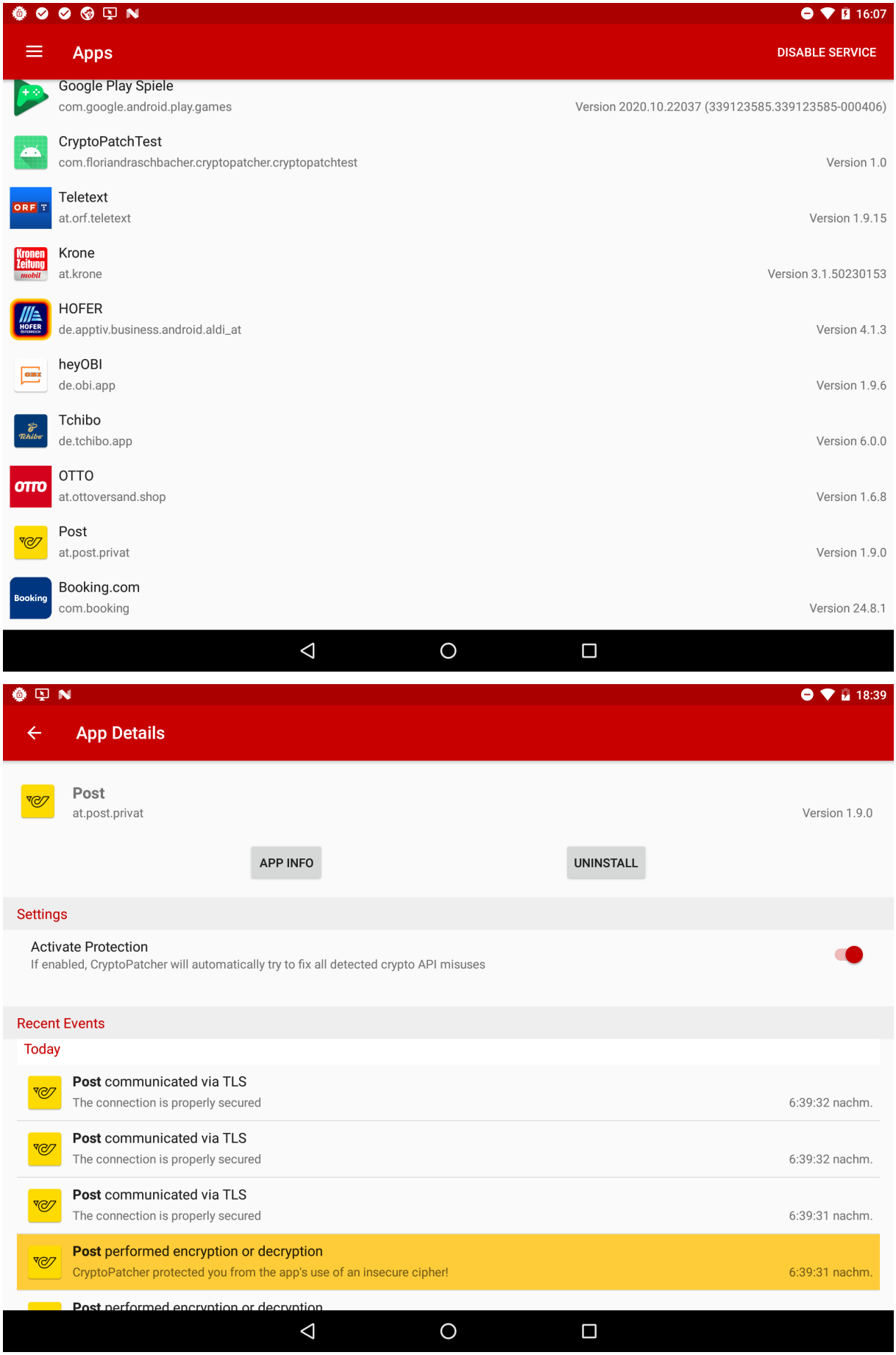

Figure 6.2: Overview and detail screen of the apps list

6.3 Implementation

6.3.1 CryptoPatch Patch

As covered in the preceding section of this thesis, the CryptoPatch patch includes mitigations for four classes of cryptography API misuses that can be corrected transparently to the target applications. In the following, we discuss the specifics of their implementations and highlight several difficulties we had to overcome in the process.

6.3.1.1 TLS/SSL

For mitigating an application's use of an insecure `HostnameVerifier` or `TrustManager` independently of the used higher-level protocol stack, our patch targets the `SSLSocket.getInputStream()` and `SSLSocket.getOutputStream()` methods. Since these must be called immediately before transport of application data begins, we can infer that the target program considers the connection secure if they are invoked. While the TLS handshake can be started implicitly and not just from an application's call to the `SSLSocket.handshake()` method, there is no way for a program to send or receive application data other than going through either of these two functions for obtaining an `InputStream` or `OutputStream`. They thus present a perfect opportunity for interposition by our patch.

However, just by intercepting the aforementioned methods, the patch does not have a chance for extracting the contacted host's address, port and name. All of this data is crucial for fetching the expected certificate from our notary web server. While the necessity for obtaining the address and port is trivial to understand, it is less obvious that the host name is similarly important for ensuring reliable operation. The reason for this can be found in today's common use of the *Server Name Identification* (SNI) extension of the TLS protocol. It is typically employed so that a single host computer can serve website data for multiple different domains. In this scenario, the client program includes the target host name in the TLS handshake, so that the server can present the certificate with a matching subject. If we were unable to identify the host name of the contacted server, our notary web server could not query the correct certificate from the latter, thus leading to a certificate mismatch for no reason.

As a solution to this problem, our patch goes a little beyond just intercepting the two methods mentioned above. In fact, it targets the creation of the `SSLSocket`, so that a wrapper object can be put in its place. To this end, the patch interposes on every method of the `SSLSocketFactory` class, where all information concerning the host is available in the call parameters. From here, it is passed to the constructor of the wrapper class, which now serves as a container for all data collected for a specific connection and as a possibility for executing our own certificate validation logic. The wrapper class extends the `SSLSocket` class and also forwards some of the private methods exposed by the system-default Conscrypt `SSLSocket` implementation to the wrapped object. This approach is necessary because some HTTP libraries access these methods via Java Reflection and do not fail gracefully for implementations other than the system default. By respecting these cases, our patch can do its job transparently to the application code. Listing 6.2 exemplarily demonstrates what the patch for intercepting one of `SSLSocketFactory`'s `SSLSocket` creation methods looks like. Please note that for the sake of brevity, all other similar methods and the code of the `CryptoPatcherWrappedSSLSocket` class that actually implements the certificate validation logic were omitted here.

```
@PatchClass("javax.net.ssl.SSLSocketFactory")
public static class SSLSocketFactoryPatch {
    @PatchInstanceMethod
    public static Socket createSocket(SSLSocketFactory thiz, Socket
        s, String host, int port, boolean autoClose) throws
        IOException {
         boolean protect = CryptoPatchProtectionStateResolver.
            getInstance().getProtectionActive();
```

```
        Socket socket = OriginalMethods.
            javax_net_ssl_SSLSocketFactory.createSocket(thiz, s, host
            , port, autoClose);
        return new CryptoPatcherWrappedSSLSocket((SSLSocket) socket,
             host, port, protect);
    }
}
```

Listing 6.2: Excerpt from the patch for injecting CryptoPatch's SSLSocket wrapper

Since the static rewriting backend of our APDK system is subject to technical limitations regarding the interception of method calls from code inside the framework, extra care had to be taken with regards to the `HttpsURLConnection` API. Although this class makes use of the `SSLSocketFactory` system for creation of its `SSLSockets`, its integration into the Android framework means that it is not covered by the solution described above. In an effort to work around this problem, our patch sets a custom default `SSLSocketFactory` for the `HttpsURLConnection` API, which in essence wraps the system default implementation and redirects execution to the patch wrapper methods covering all remaining uses of the `SSLSocketFactory` APIs as described above. In order not to disturb an application's explicit specification of a default or connection-specific `SSLSocketFactory`, the corresponding setters on the `HttpsURLConnection` API are intercepted so that the supplied object can be wrapped before being passed to the original setter.

The CryptoPatch's custom certificate validation logic contacts our notary web service, which we implemented as a Java Jetty servlet presenting a simple JSON REST interface over HTTPS. The service in turn initiates a TLS connection to the specified target server and queries it for the certificate of the given host. Please note the latter is not validated in any way by the notary service so as to support self-signed certificates, which can be found in real-world use in some cases. The obtained certificate is then forwarded to the patched application, where it is compared to the one received through the direct connection to the target server. In case of a mismatch (and activated protection in the CryptoPatcher application), the connection is immediately aborted by throwing an `IOException`. Since the `SSLSocket.getInputStream()` / `SSLSocket.getOutputStream()` methods are known to throw these types of exceptions for example in the event of a connection loss, the target application is guaranteed to gracefully handle the error. In order to ensure that no *Man-In-The-Middle* (MITM) attack can be mounted on the connection between the patch and notary web service, we utilise certificate pinning. The basic sequence executed by our TLS patch is illustrated in Figure 6.3.

Please note that we currently are not maintaining a dedicated web server for our notary servlet, so that the program must be running on a local host known to and reachable from the Android device that runs CryptoPatcher. We would like to stress that this setup is only suitable as a general proof of concept. For real-world use, a dedicated server is imperative, not least to ensure that compromising the connection between the notary service and target servers is exacerbated for an attacker.

In order to reduce the load on our notary web service, we integrated a caching system into the CryptoPatch. It stores every certificate obtained through the service in a dedicated file inside an application-private cache folder. If a mismatch is detected between the cached entry and the certificate presented by the target server, the notary web service is queried as the authoritative last resort. This procedure is necessary in case the certificate of the target server has been updated.

Since the notary web service usually runs on a remote web server, it is incapable of determining the legitimate certificate of a target server running on a local host. Consequentially, a special scheme has to be devised to account for this scenario. In its current implementation, our patch assumes that communication with a local server is less likely to be attacked and less likely to involve sensitive information. Since the possibilities of a remote notary service are technically limited for this specific corner case, a fallback to

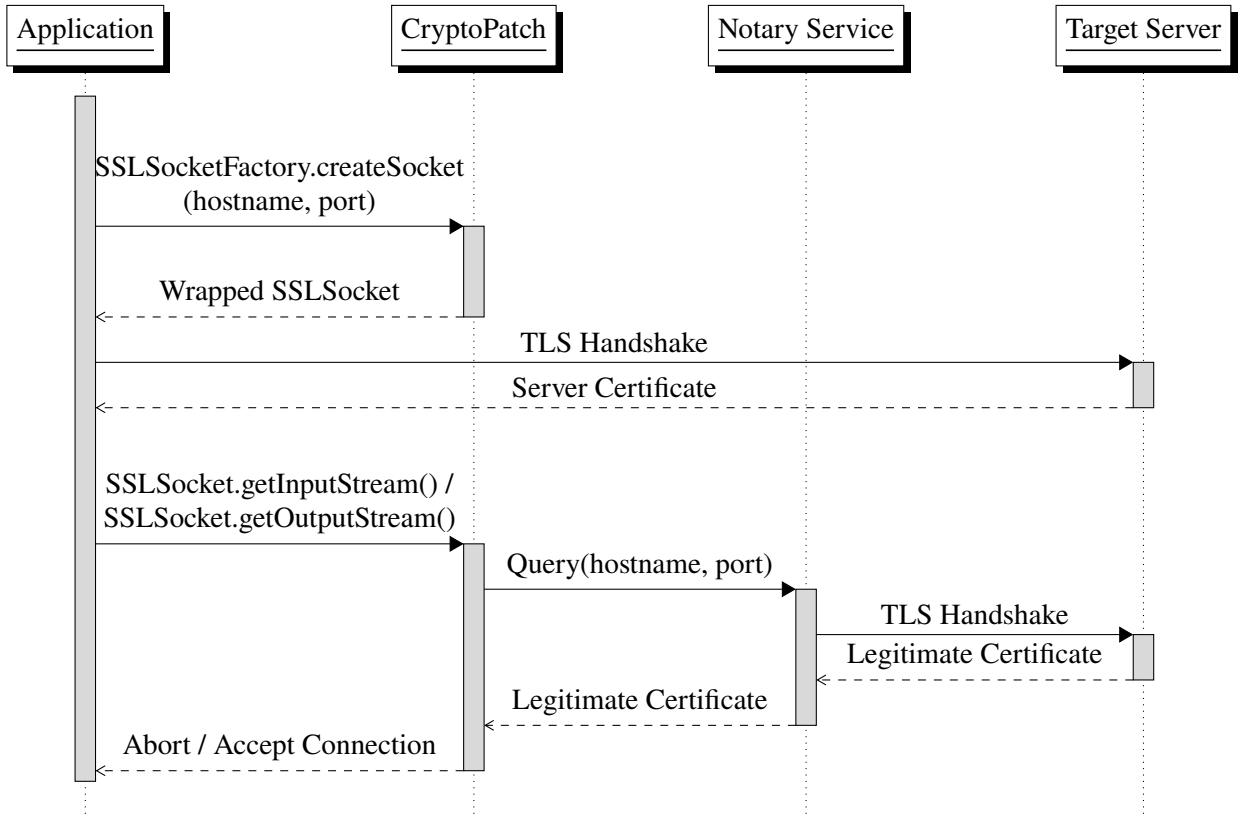

Figure 6.3: Simplified illustration of the procedure followed in the TLS patch

the *Trust-On-First-Use* (TOFU) principle could be investigated in further works, although we considered it out of scope for this thesis.

6.3.1.2 Cipher

Although our patch for the Cipher API follows a similar structural principle as that in the patch for TLS/SSL, its implementation was slightly more challenging since the `Cipher` class, in contrast to the `SSLSocket` class, is final and thus cannot trivially be wrapped. As a consequence, CryptoPatch has to get more deeply involved with the Provider architecture of the *Java Cryptography Extension* (JCE) described in Section 2.6.1. Specifically, the patch implements its own `Provider` that wraps another `Provider`'s implementation of a specific cipher algorithm. By intercepting the `Cipher.getInstance()` method, we are able to replace the default or user-specified `Provider` with an instance of our wrapper. When the latter is then requested to create a new `Cipher` object, it returns an `CryptoPatchCipherSpi` object, which is a custom wrapper `CipherSpi` that implements CryptoPatch's actual vulnerability mitigation functionality for the specific API. Listing 6.3 shows an illustrative excerpt from the patch code responsible for injecting our custom wrapper `Provider`. Please note the actual implementation of the latter and the `CipherSpi` are too extensive for detailed listing.

```
@PatchClass("javax.crypto.Cipher")
public class CipherPatch {
    @PatchStaticMethod
    public static Cipher getInstance(String transformation, Provider
        provider) throws NoSuchAlgorithmException,
            NoSuchPaddingException {
        return OriginalMethods.javax_crypto_Cipher.getInstance(
           transformation,
                new CryptoPatchCipherProvider(provider,
```

```
                    transformation, nonceStore));
        }
}
```

Listing 6.3: Excerpt from the patch for injecting our custom `CryptoPatchCipherProvider`

During its construction, the `CryptoPatchCipherSpi` automatically upgrades the supplied transformation string that encodes the requested algorithm, mode of operation and padding. If the insecure *Electronic Code Book* (ECB) mode is explicitly specified, or in case no mode is specified for an algorithm that is known to default to ECB mode, CryptoPatch opts to use the secure CBC mode instead. However, this mode requires a unique random *Initialisation Vector* (IV) to be supplied to both encryption and decryption.

A similar problem arises when a target application relies on an insecure IV. In order to detect these cases, the CryptoPatch maintains a database of previously used IVs. It is organised as two simple plain text files stored in application-private storage. One of the files saves all IVs that occurred exactly once, while the other keeps track of all values that were found to be used repeatedly. Since applications that properly implement IV generation are expected to produce a new value for every encryption, the first of the mentioned files could be prone to growing very large over time. In order to compensate for this problem, our solution stores the IVs used exactly once inside the application's private cache directory, so that it can be automatically evicted by the system, or manually by the user in low-memory situations. When an application-supplied IV was found inside the list of used IVs, it is permanently moved to the list of reused IVs.

Whenever CryptoPatch has to inject a fresh random IV, it prepends the value to the produced ciphertext. For all decently well-written third-party applications, the change in ciphertext length should not be a problem, since they should already be using the `Cipher.getOutputSize()` method for appropriately sizing their buffers. Our implementation of `CipherSpi` accounts for the space for the prepended IV in this method if needed.

Since an IV could have been tagged as reused between the time it was used for encryption and the time the produced ciphertext is decrypted, we can not rely on information from our IV database for ascertaining when an IV is to be extracted from the ciphertext. Instead, we need a way to encode in the ciphertext itself whether an IV has been prepended. To this end, the IV in the ciphertext is itself prefixed with a constant header. During decryption, the state machine inside our `CryptoPatchCipherSpi` class delays initialisation of the wrapped cipher until enough ciphertext material has been supplied for determining the presence of IV data from the found prefix.

It is worth noting that the solution discussed above has limitations. Namely, in some cases, an encrypted value is sent to a server for decryption, or vice versa. In this scenario, upgrading the requested encryption scheme transparently to the application code could render the server unable to decrypt the produced ciphertext. It is for this reason that our CryptoPatcher application allows users to disable the automatic mitigation measures. This permits administrators to trade off the need for protection against that of retaining a specific application's full feature set.

6.3.1.3 Password-Based Encryption

The patch for mitigating common flaws in the implementation of *Password-Based Encryption* (PBE) is less sophisticated than those for the two APIs described above. In general, it follows the same principle as our patch for the `Cipher` API, in that it intercepts the `SecretKeyFactory.getInstance()` method for returning a `SecretKeyFactory` instance based on a custom wrapper `SecretKeyFactorySpi` class, as illustrated in Listing 6.4.

```
@PatchClass("javax.crypto.SecretKeyFactory")
public static class SecretKeyFactoryPatch {
    @PatchStaticMethod
```

```
    static SecretKeyFactory getInstance(String algorithm) throws
        NoSuchAlgorithmException {
        boolean protect = CryptoPatchProtectionStateResolver.
            getInstance().getProtectionActive();
        return new CryptoPatchSecretKeyFactory(new
            CryptoPatcherSecretKeyFactorySpi(algorithm, nonceStore,
            protect), new CryptoPatchPbeProvider(), algorithm);
    }
}
```

Listing 6.4: Excerpt from the patch for injecting our custom `SecretKeyFactorySpi`

When the target application requests the `SecretKeyFactory` to generate a secret, the supplied key specification is inspected for improper values for the salt or iteration count parameters. In case the iteration count is lower than the current minimum recommendation of 1000, the patch automatically raises it to meet this best practise.

For reasons described in Section 6.1.2.3, our patch does not have an opportunity to automatically correct the use of a constant salt. Instead, it is limited to producing a critical event for display in the CryptoPatcher application's monitoring interface. The mechanism for identifying reused values works analogously to that discussed for IVs above.

Since some applications utilise the *BouncyCastle* library or its Android-specific *SpongyCastle* fork for PBE, CryptoPatch covers the relevant APIs as well. Specifically, this encompasses the `PBEParameterGenerator.init()` method that takes the salt and iteration count as its immediate arguments. The patch itself works identical to that for the standard JCE PBE API.

6.3.1.4 Secure Random

For correcting the bad practise of seeding the `SecureRandom` API with a constant value, CryptoPatch hooks the `SecureRandom.getInstance()` method for injecting its own wrapper `Provider` and a custom `SecureRandomSpi`. Since the default implementation can also be accessed without going through the static factory method by just creating a new instance of the `SecureRandom` class, our patch also installs a custom default `Provider` for the *SHA1PRNG* algorithm. Listing 6.5 displays the implementation of our `SecureRandomSpi` class, including monitor event creation and seed reuse checks.

```
public static class WrappedSecureRandomSpi extends SecureRandomSpi {
    private final SecureRandom wrapped;
    private boolean protect = true;

    public WrappedSecureRandomSpi(String algorithm, Provider
        wrappedProvider, Boolean protect) throws
        NoSuchAlgorithmException {
        this.wrapped = OriginalMethods.java_security_SecureRandom.
            getInstance(algorithm, wrappedProvider);
        this.protect = protect;
    }

    @Override
    protected void engineSetSeed(byte[] seed) {
        SecureRandomApiUsageReport report = new
            SecureRandomApiUsageReport(
                SecureRandom.class, wrapped.getProvider().getClass()
                    , "setSeed", Arrays.toString(seed));
        report.setProtectionStatus(protect ? ProtectionStatus.
            Protected : ProtectionStatus.Unprotected);
        report.setAlgorithm(wrapped.getAlgorithm());
```

```
        report.setSeed(seed);

        if (checkSeedReuse(seed)) {
            // Let wrapped seed itself.
            if (!protect) wrapped.setSeed(seed);
            report.setSecurityStatus(SecurityStatus.Insecure);
        } else {
            report.setSecurityStatus(SecurityStatus.Secure);
            wrapped.setSeed(seed);
        }

        AppMonitorClient.getInstance().reportApiUsage(report);
    }

    @Override
    protected void engineNextBytes(byte[] bytes) {
        wrapped.nextBytes(bytes);
    }

    @Override
    protected byte[] engineGenerateSeed(int numBytes) {
        return wrapped.generateSeed(numBytes);
    }
}
```

Listing 6.5: CryptoPatch's `SecureRandomSpi` implementation

The main task of the custom `SecureRandom Provider` besides generating monitoring events is preventing seed reuses. To this end, it employs a simple database as devised above. Whenever a seed reuse is detected, the entropy material is not forwarded to the underlying wrapped implementation, so that the system default seeding is used as the sole source of entropy.

6.3.2 CryptoPatcher Application

The backbone of the CryptoPatcher application is formed by its main service. Started immediately after boot, it promotes itself to a persistent service that is automatically reinstantiated in the event of a crash, and displays a notification to inform the user of its current state. Moreover, it installs a `BroadcastReceiver` that constantly listens for changes to the installed packages. An execution queue collects all patching tasks that have to be processed, so that the application can gracefully handle the installation of additional programs while patch deployment for a previous installation is in progress.

For obtaining some sort of control over the system, the CryptoPatcher application takes advantage of the `DevicePolicyManager` API. By implementing a `DeviceAdminReceiver` that is configured as device owner (through the instructions detailed in Section 6.2.1), our application is granted access to the powerful `PackageInstaller` API. Through this interface, it can install and uninstall applications, both in the form of a single APK file or multiple Split APKs supplied to a single installation session. Our application uses the opportunity for installing a progress listener in order to reflect the patch installation progress in its user interface. An array of additional package management functionality is available through the `DevicePolicyManager` API, one of which is the possibility for disabling an installed application. In this special state, the program's icon in the launcher is grayed out and tapping it just yields an error message screen. However, disabled applications are still eligible for updates through Google Play, presenting a perfect way for CryptoPatcher to support the usual update flow for patched software.

Throughout the whole application, a particular focus was put on closely following best practises for the Android platform. To this end, the CryptoPatcher software makes extensive use of the Android *Jetpack* components that facilitate the design of lifecycle-aware, data-centric applications. For the user interface,

all `Fragments` were implemented as lightweight containers for `ViewModel` classes that deal with the heavy lifting of keeping the screen elements synchronised to the backing data stores. Wherever possible, the access to the latter was abstracted through *Database Access Objects* (DAOs) and realised in accordance to the reactive programming paradigm via observable `LiveData` holders. Populating the actual on-screen display takes advantage of the view binding mechanism for reducing boilerplate code.

In order to keep track of all installed patched applications, as well as a package's status both in terms of ongoing patch deployments and configured protection level, the CryptoPatcher application integrates the *Rooms* library for SQL database interaction without getting involved in the lower-level details. Since the library already supports yielding query results as `LiveData` objects that automatically notify their observers whenever the data changes, it facilitates our goal of immediately reflecting updates into the user interface.

A central feature of the CryptoPatcher system is the possibility for gaining insights into a patched application's operation through a monitor display. This functionality is implemented by means of special broadcast messages exchanged between the CryptoPatch inside a patched target application and the CryptoPatcher software. Since these messages contain sensitive information such as encryption keys, they need to be kept unaccessible to other processes. To this end, we employ custom permissions so that patched applications are only allowed to produce monitor events, but not consume them. A `BroadcastReceiver` inside the CryptoPatcher application collects all emitted messages inside a circular buffer that is reflected into the monitor user interface.

Managing the protection status on a per-app level similarly takes advantage of Android IPC infrastructure. Specifically, a `ContentProvider` was implemented that allows the patch code inside an application process to query the current configuration from the CryptoPatcher application. In order to restrict insights into the protection level to only the affected package, the `ContentProvider` inspects the metadata provided by the Android Binder IPC mechanism to identify the calling component.

6.4 Conclusion

In this chapter, we completed the picture of our CryptoPatcher solution with the introduction of the CryptoPatch patch and the CryptoPatcher application. These are the components that bring the powerful foundations laid by the Android Patch Development Kit to a practical use for the benefit of end consumers.

Starting from a refinement of the specific objectives for these two entities, we discussed how our patch strives to mitigate the covered four categories of vulnerabilities induced by crypto API misuse and how the design of the CryptoPatcher application seeks to bring it into a user-friendly form. After highlighting the user-facing functionality and depicting its operation, we covered the actual implementation of the system.

Chapter 7

Evaluation

In this chapter, we evaluate efficacy and efficiency of our CryptoPatcher system. To compensate for the fact that the dynamic nature of our solution impedes large-scale automated analysis, Section 7.1 instead presents case studies that show how our software protects users' data from being disclosed through an application's misuse of cryptography APIs in practical scenarios. Specifically, we demonstrate how our CryptoPatcher solution prevents a simulated Man-In-The-Middle attack on a susceptible application's TLS traffic. Additionally, we reverse-engineer several third-party programs to pinpoint the location of crypto API misuses identified by our system. The subsequent Section 7.2 sheds some light on performance characteristics of our implementation, before Section 7.3 lists its limitations, respective causes and potential improvements. Finally, Section 7.4 concludes the chapter with a summary of our findings.

7.1 Case Studies

All case studies are executed on a Google Nexus 9 tablet running the official Android Nougat 7.1.1 firmware released by the manufacturer in 2016, as the last for the device. Although this means the device has not received the latest revisions of the Android OS, the same holds true for a considerable portion of active devices. In fact, as can be observed from official figures accessible through Android Studio[1], more than a third of devices is still running on the same Nougat version or an even older release. In terms of CPU performance and RAM size, the tablet is still comparable to recent low- to mid-range phones.

The target applications subject to our case study were all chosen from Google Play. Selection criteria included their popularity, confidentiality of the processed data, exhibition of at least one of the vulnerabilities covered by CryptoPatch, and compatibility with our APDK deployment infrastructure.

7.1.1 TLS

As discussed in Section 1.1, the prevalent misconfiguration of TLS in Android applications prompted Google to introduce the *Network Security Configuration* (NSC) system as a global, easy-to-integrate countermeasure. As its most fundamental achievements, the NSC solution provides convenient TLS pinning possibilities and a safe default that distrusts user-installed *Certificate Authorities* (CAs). Both components help in preventing *Man-In-The-Middle* (MITM) attacks where an attacker can decrypt a victim's TLS communication.

However, through the flexible NSC system, applications can still actively enable support for user-installed certificates. While this option is generally intended for use in isolated debugging environments during

[1]Android Version Distribution statistics will now only be available in Android Studio: https://www.xda-developers.com/android-version-distribution-statistics-android-studio/

app development, a considerable amount of apps ship to customers in this configuration, effectively dismissing a cornerstone of the protection measures provided by the NSC scheme. Depending on the specific circumstances, this mistake may open the door to a certain class of MITM attacks, where a user is tricked into installing an adversary's CA that can be used to transparently decrypt and re-encrypt intercepted TLS communication.

Even worse, it seems a small fraction of applications available through Google Play still employ unsafe `HostnameVerifier` or `TrustManager` implementations. It is unclear how these managed to pass the automated checks described in Section 1.1 that were put up by Google in 2016.

7.1.1.1 Facebook Messenger Lite

An example of an unsafe NSC can be found in the *Facebook Messenger Lite*[2] application, installed on more than 500 million devices as of January 2021, placing it among the most popular programs available through Google Play. The program integrates text messaging, voice calling and video chatting functionality that allows communicating with other Facebook users via the Internet. From the software's Network Security Configuration, included in Listing 7.1, we can observe that all user-installed certificates are trusted, and even more problematically, they are allowed to override the pinned certificates specified for some hosts.

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config cleartextTrafficPermitted="true">
        <trust-anchors>
            <certificates src="system" />
            <certificates overridePins="true" src="user" />
        </trust-anchors>
    </base-config>
    <domain-config cleartextTrafficPermitted="false">
        <domain includeSubdomains="true">facebook.com</domain>
        ...
        <pin-set expiration="2021-07-1">
            <pin digest="SHA-256">lCppFqbkrlJ3EcVFAkeip0+44
                VaoJUymbnOaEUk7tEU=</pin>
            ...
        </pin-set>
        ...
    </domain-config>
</network-security-config>
```

Listing 7.1: The Network Security Configuration of the Facebook Messenger Lite application (ellipses mark points of truncation)

As a consequence of this explicit opt-out of the default protection measures, the application needs to integrate a custom implementation of certificate pinning for preventing susceptibility to MITM attacks. In order to ascertain the existence and effectiveness of the custom approach, we can try to mount a MITM attack by configuring a system-wide HTTPS proxy. Instead of just relaying traffic, our proxy server tries to impersonate the target server to the Facebook Messenger Lite application and vice versa, so that it gets a chance to decrypt the communication. The proxy's certificate, which it uses for re-encrypting the traffic, is installed on the target Android device. For our study, we use the *Proxyman*[3] proxy server running on a computer on the local network.

Once the login credentials entered in the welcome screen of the app's user interface are submitted to

[2]We used Facebook Messenger Lite (com.facebook.mlite) 120.0.0.1.118, the newest version at the time of our testing
[3]Modern Web Debugging Proxy: `https://proxyman.io`

the Facebook server, we can indeed observe various HTTPS requests in the Proxyman application, as illustrated in Figure 7.1. It appears like the Facebook Messenger Lite application does not implement any TLS pinning functionality beyond the system-provided NSC scheme, which is bypassed as detailed above. Consequentially, an attacker can observe user name and password, which are enough for a permanent account takeover.

To summarise, an attacker would have to perform these steps:

1. Trick the user into installing the attacker's CA certificate
2. Intercept network traffic (e.g. through ARP spoofing)
3. Take over Facebook account with obtained credentials

In order to shield users of the Facebook Messenger Lite application from this critical information disclosure, we install the program onto a device protected with our CryptoPatcher solution. With the service running in the background, we simply initiate the download of the target application through the Google Play Store and wait until it has installed. Next, we can observe the CryptoPatcher notification informing us of the ongoing patching operation. Once the patched version is installed, it can be started from the application launcher. Again, we enter the credentials of our test account and initiate the login procedure. This time, as illustrated in Figure 7.3, the login within the Facebook Messenger Lite program fails with a message notifying the user of an unexpected error. From CryptoPatcher's monitor screen depicted in Figure 7.4, we can confirm that this is because our system detected a mismatch between the server certificate presented by the proxy server and the expected one as obtained through the notary web service, so that the connection was forcefully aborted. Since the CryptoPatch terminated the connection before any data was exchanged, no information leaked to the MITM attacker, as can be verified from a glance at the display of the proxy server program in Figure 7.2. Only the request to our notary web server can be observed, which is protected through TLS pinning in CryptoPatch.

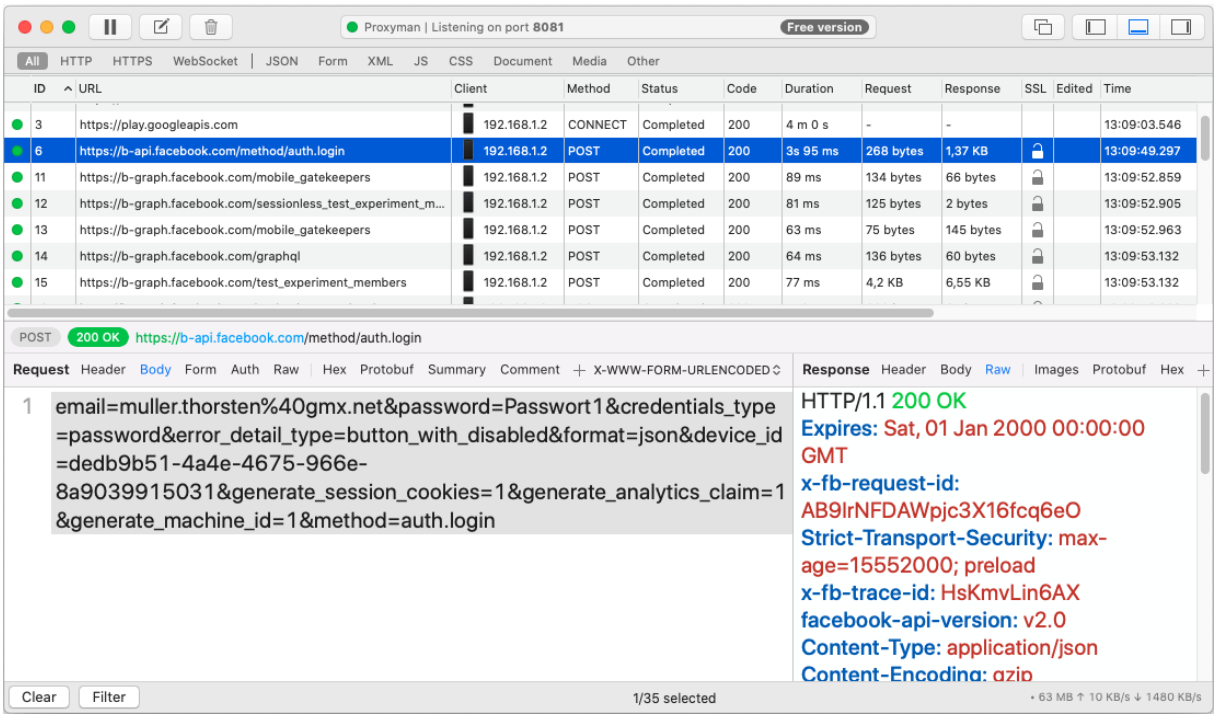

Figure 7.1: In the unpatched application, login credentials can be intercepted through the proxy server

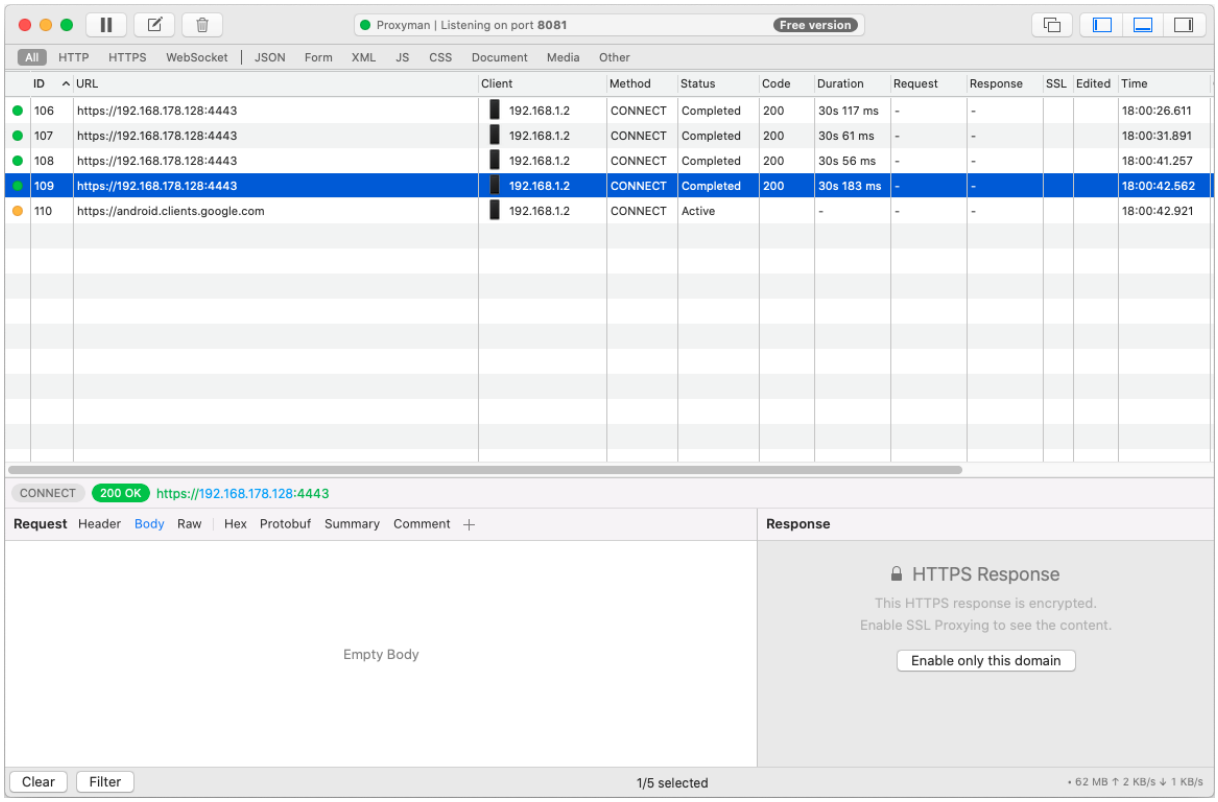

Figure 7.2: With active CryptoPatcher protection, no information is leaked at all

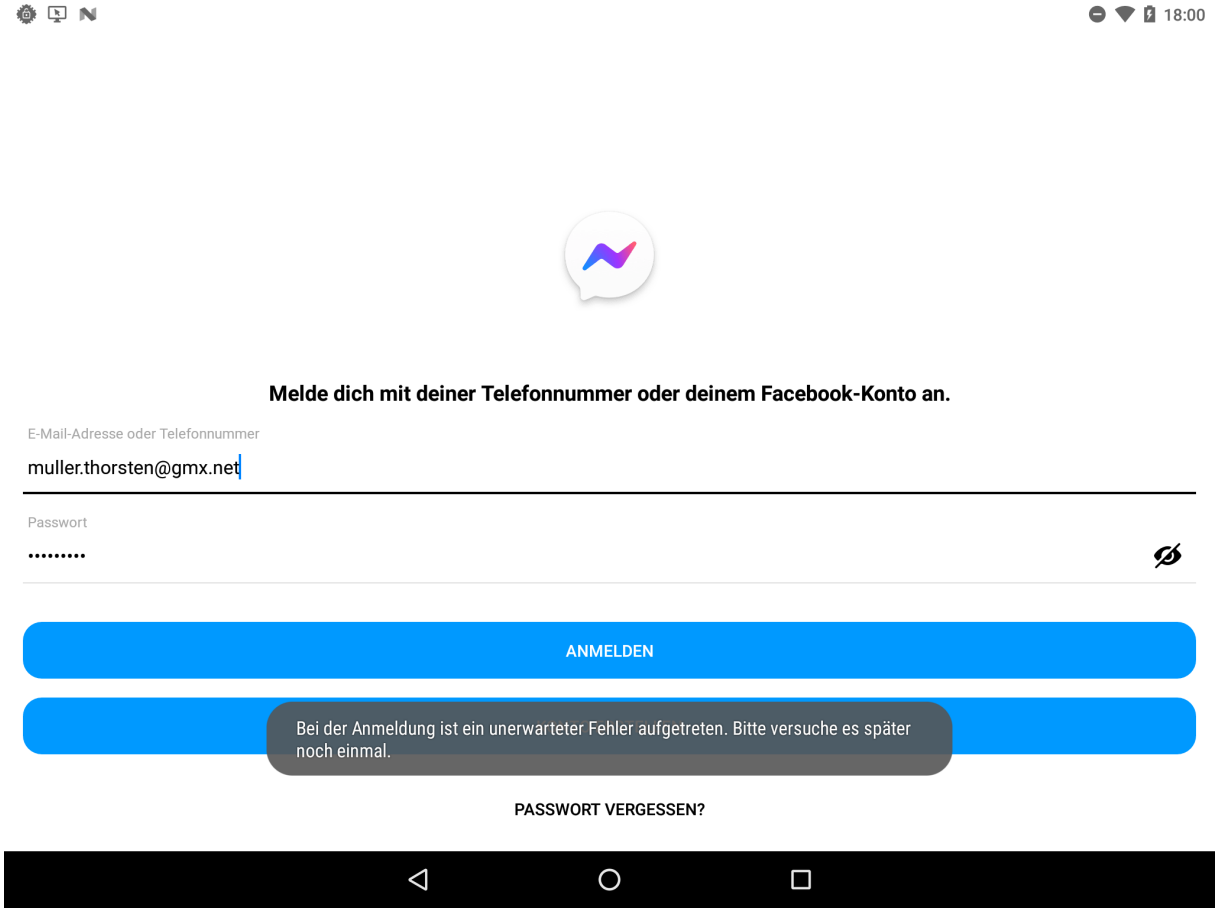

Figure 7.3: Login fails while an attack is mounted under active CryptoPatcher protection

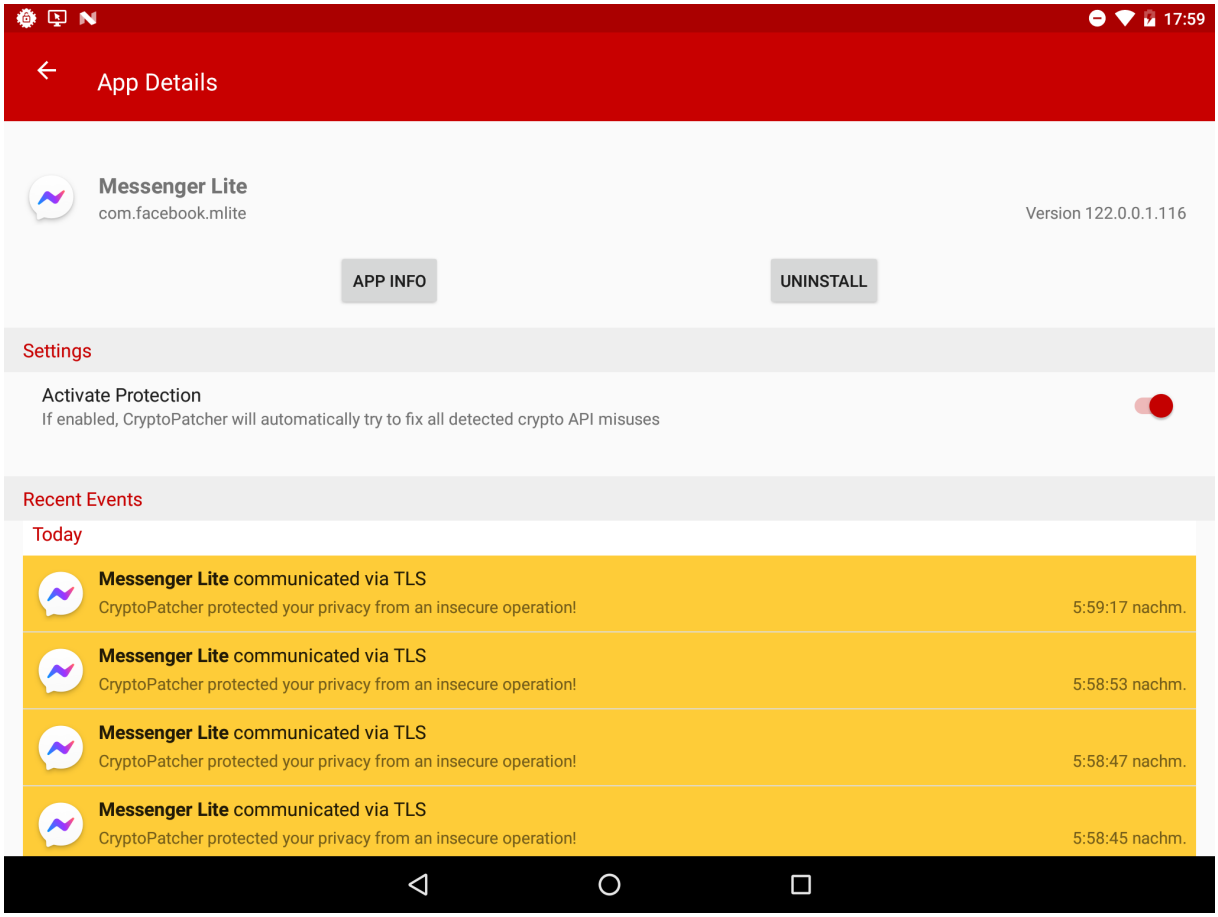

Figure 7.4: CryptoPatcher noticed and terminated the compromised connection

7.1.1.2 Banggood

An example of a critically unsafe application in terms of TLS communication is the *Banggood* Android client[4]. Banggood is a Chinese online retailer that offers low-priced items shipped directly from China. Its Android app has been downloaded more than 10 million times from Google Play as of January 2021.

The software not only uses a similar NSC as that of Facebook Messenger Lite described above, but more importantly completely dismisses the NSC system altogether by utilising a custom `TrustManager` implementation that accepts any certificate. As a result, the attack sketched above is tremendously simplified for an adversary, who no longer needs to trick the user into installation of a custom CA certificate.

When the registration procedure is executed inside the Banggood application as obtained from Google Play on a stock device, a MITM attacker can trivially extract the used login credentials, as shown in Figure 7.5. Worsening the criticality of the vulnerability is the fact that the unassuming user has no chance to notice the account takeover.

From reverse-engineering the Banggood APK, we conclude that the application is using a stub `TrustManager` implementation that does not perform any checks on the server's certificate at all. Since the application is heavily obfuscated, we considered a full investigation of its exact operation out of scope for this thesis.

[4]We used Banggood (com.banggood.client) 7.15.0, the newest version at the time of our testing

Still, when installing the application on a system protected by CryptoPatcher, the data disclosure is successfully prevented. From CryptoPatcher's monitor interface depicted in Figure 7.6, we can confirm that the network request for registration failed because our system aborted the connection after the certificate mismatch was detected.

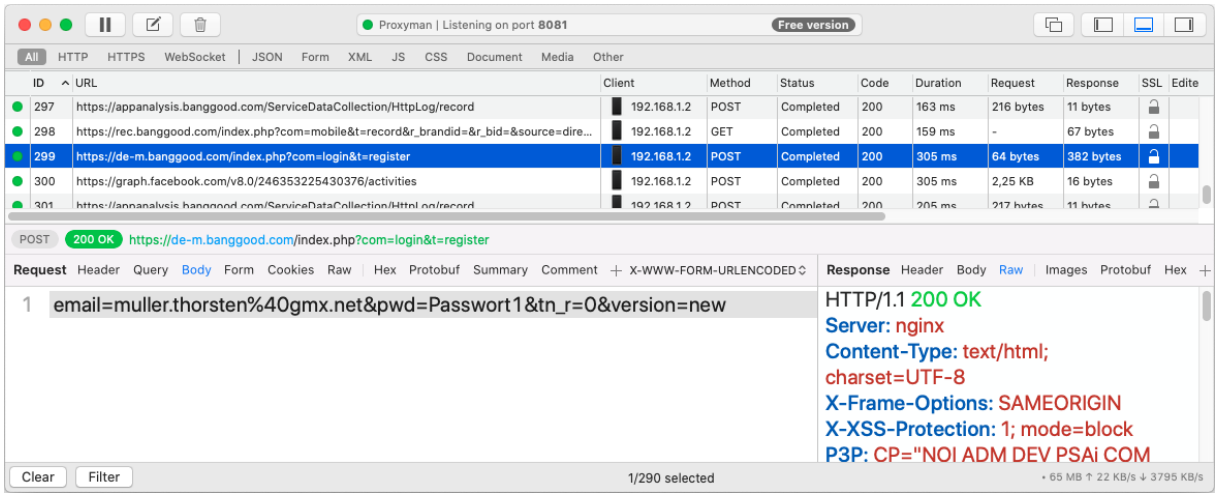

Figure 7.5: A MITM attacker can trivially extract Banggood login credentials

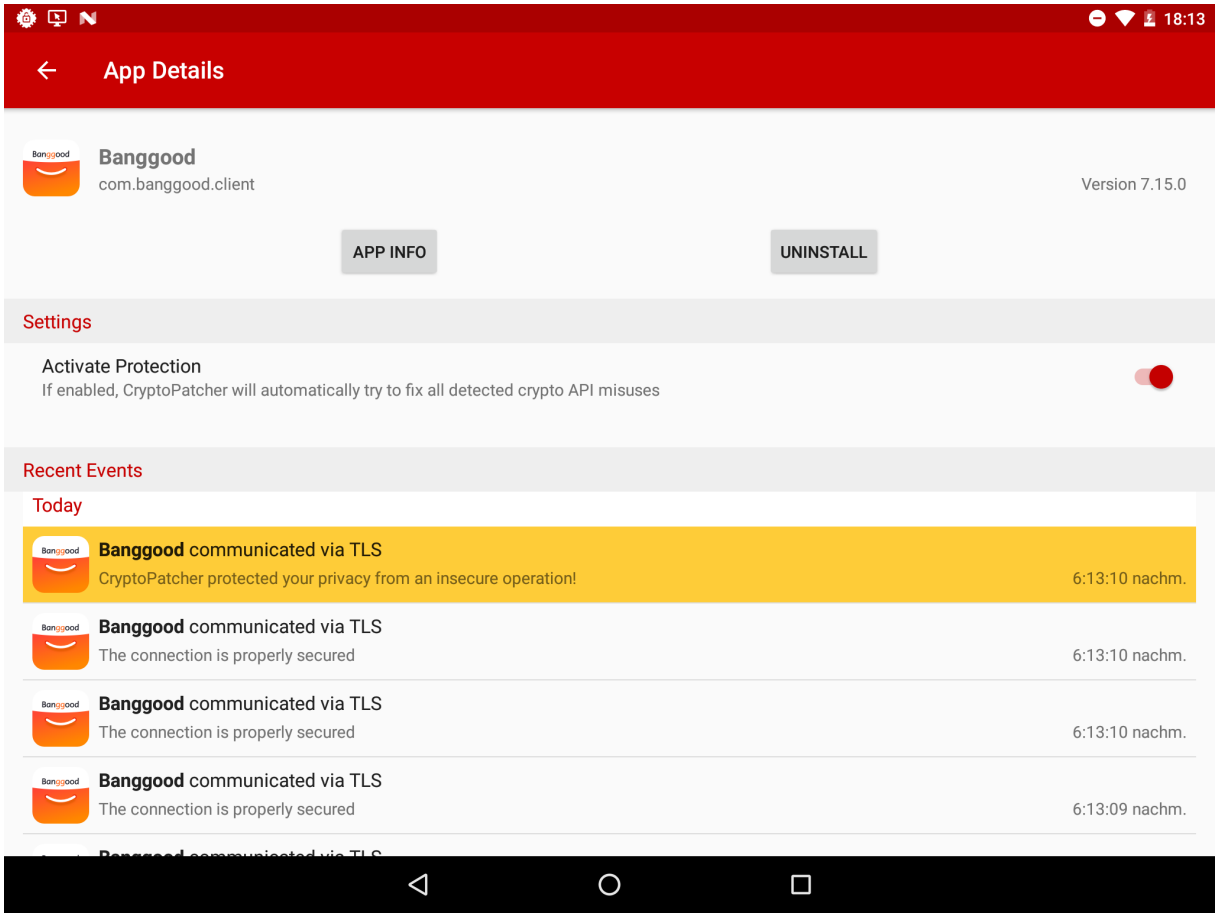

Figure 7.6: CryptoPatcher detected and prevented the MITM attack on the vulnerable app

7.1.2 Cipher

The only official countermeasure set up to prevent developers from making mistakes in the use of the `Cipher` API are Lint rules inside the Android Studio IDE. Since the produced warnings can easily be ignored or disabled, a considerable amount of applications on Google Play still use the unsafe ECB mode or a static IV.

7.1.2.1 Password Saver

Our first case study in this group investigates the *Password Saver* application[5]. With more than 500 000 downloads from Google Play as of January 2021, the software is one of the more popular choices in the category of password vaults, allowing the organisation of all the user's passwords in one place, encrypted by a single master password.

Despite the bold claims in its listing on Google Play, the software does not properly protect the entered data, as can be found out through CryptoPatcher's monitoring functionality. Specifically, our solution finds that for all its invocations of the `Cipher` API, the Password Saver program fails to specify a mode of operation, so that the AES algorithm is executed in the default *Electronic Code Book* (ECB) mode.

In order to demonstrate the validity of CryptoPatcher's claims, we augment the automated dynamic analysis with a manual static examination using the *JADX* DEX decompiler[6]. From a search in the reverse-engineered Java code, we can locate the source of the problem in the `CryptoAesMaster` class. In the functions that are apparently used for encryption and decryption of most of the user-entered data fields, the transformation string "AES" is supplied, as shown in Listing 7.2.

```
package com.gsonly.passbook;

class CryptoAesMaster {
    public static String encrypt(String str, String str2) {
        String str3 = "ISO-8859-1";
        String str4 = "AES";
        try {
            SecretKeySpec secretKeySpec = new SecretKeySpec(
                getRawKey(str2), str4);
            Cipher instance = Cipher.getInstance(str4);
            instance.init(1, secretKeySpec);
            str = toHex(instance.doFinal(new String(str.getBytes(
                UrlUtils.UTF8), str3).getBytes(str3)));
            return str;
        } catch (Exception e) {
            try {
                e.printStackTrace();
            } catch (Exception unused) {
            }
            return str;
        }
    }
}
```

Listing 7.2: Excerpt from the `CryptoAesMaster` class of the Password Saver application

Since our CryptoPatch broadcasts the monitor event at the same place that the insecure value is corrected, we can conclude that our solution not only successfully detected the crypto API misuse, but also protected

[5]We used Password Saver (com.gsonly.passbook) 2.13, the newest version at the time of our testing
[6]JADX Dex To Java Decompiler: `https://github.com/skylot/jadx`

the user by automatically mitigating the vulnerability.

7.1.2.2 aWallet

A slightly different `Cipher` API misuse can be found in the *aWallet* application[7], another popular password organiser available from Google Play. As of January 2021, the software has accumulated more than 1 million downloads.

From CryptoPatcher's monitor screen depicted in Figure 7.7, we are informed that although the program utilises the secure "AES/CBC/PKCS7Padding" transformation, its use of a constant IV still puts the stored data at risk.

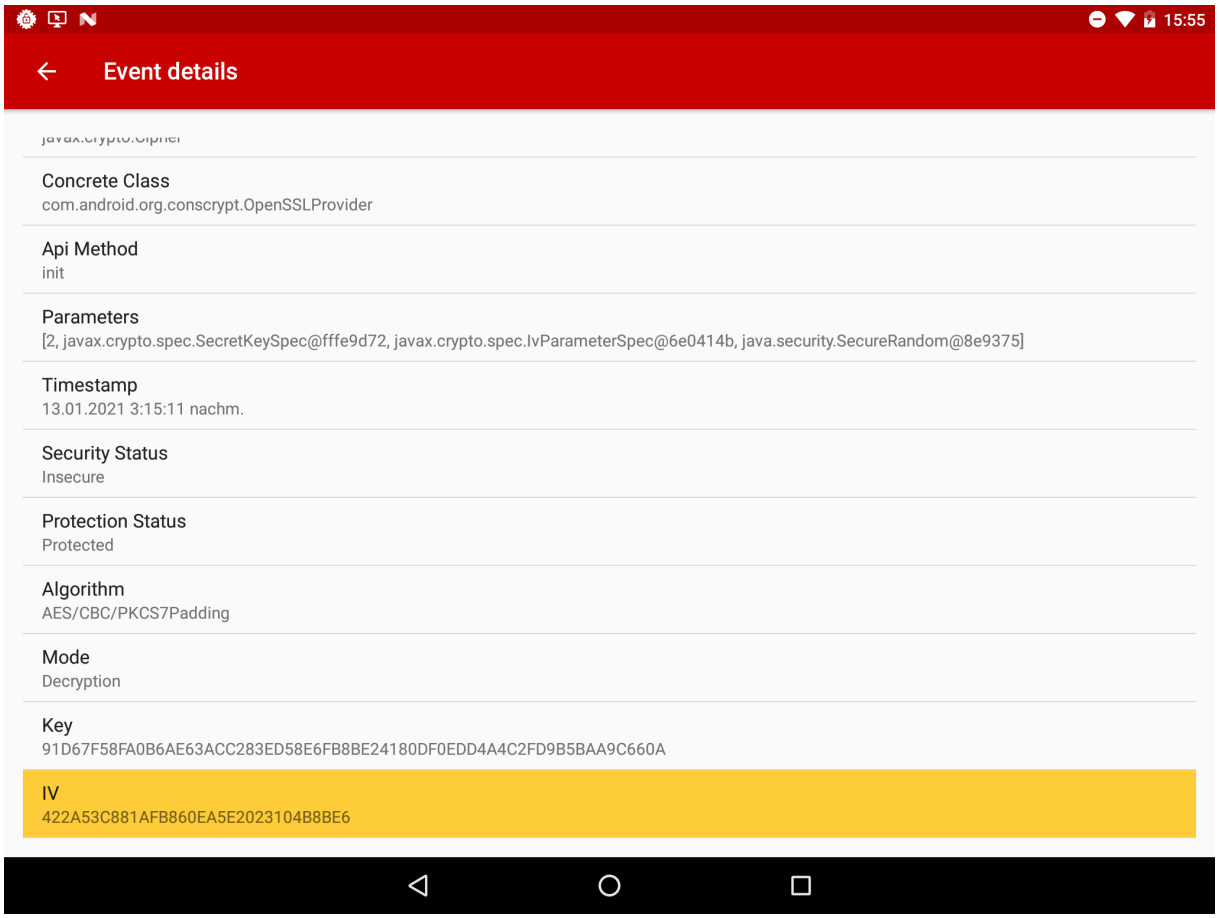

Figure 7.7: CryptoPatcher reported an IV reuse

For investigating the cause of the report, we again take advantage of the JADX decompiler. Through a search for references to the `IvParameterSpec` class and by following the parameters passed to its constructor, we eventually reach the class `org.awallet.c.g.m`, partly shown in Listing 7.3, which defines a static final byte array holding the IV reported by CryptoPatcher. The constant is accessed from other places in the code through the `org.awallet.c.g.m.p()` method.

```
package org.awallet.c.g;
```

[7]We used aWallet (org.awallet.free) 8.5.1, the newest version at the time of our testing

```
public final class m {
    ...
    private static final byte[] l = new byte[]{(byte) 66, (byte) 42,
        (byte) 83, (byte) -56, (byte) -127, (byte) -81, (byte) -72,
       (byte) 96, (byte) -22, (byte) 94, (byte) 32, (byte) 35, (byte
       ) 16, (byte) 75, (byte) -117, (byte) -26};
    ...

    private byte[] p(int i, boolean z) {
        ...

        if (i != 8) {
            if (i != 16) {
                ...
            } else if (z) {
                ...
            } else {
                return l;
            }
        } else if (!z) {
            ...
        } else {
            ...
        }
    }
}
```

Listing 7.3: The `org.awallet.c.g.m` class defines the static IV CryptoPatcher reported

The fact that the IV is not only reused but also hardcoded into the application's source code further raises the severity of the vulnerability, rendering CryptoPatcher's mitigation even more critical for protecting the user's data.

7.1.3 Password-Based Encryption

Although the frequent problem of insecure parameterisation of *Password-Based Encryption* (PBE) has been reported numerous times before, no official countermeasure has been integrated into the Android SDK or Play Store. As a result, a countless number of applications are still putting their user's data at risk.

For our case study of this vulnerability, we have picked the *My Passwords* application[8], with a total of more than 1 million installs according to Google Play as of January 2021. Like the two products examined above, the application belongs to the group of password managers.

When the My Passwords program is installed on a system protected with our CryptoPatcher system, the latter issues a warning as soon as the user enters the master password. Specifically, as shown in Figure 7.8, we are informed that the My Passwords application derives a cryptographic key from our entered master password, but only specifies an iteration count of 100. The low value can be considered hazardous from a security standpoint in the light of the recommendations discussed in Section 2.1.3.

[8]We used My Passwords (com.er.mo.apps.mypasswords) 20.12.00, the newest version at the time of our testing

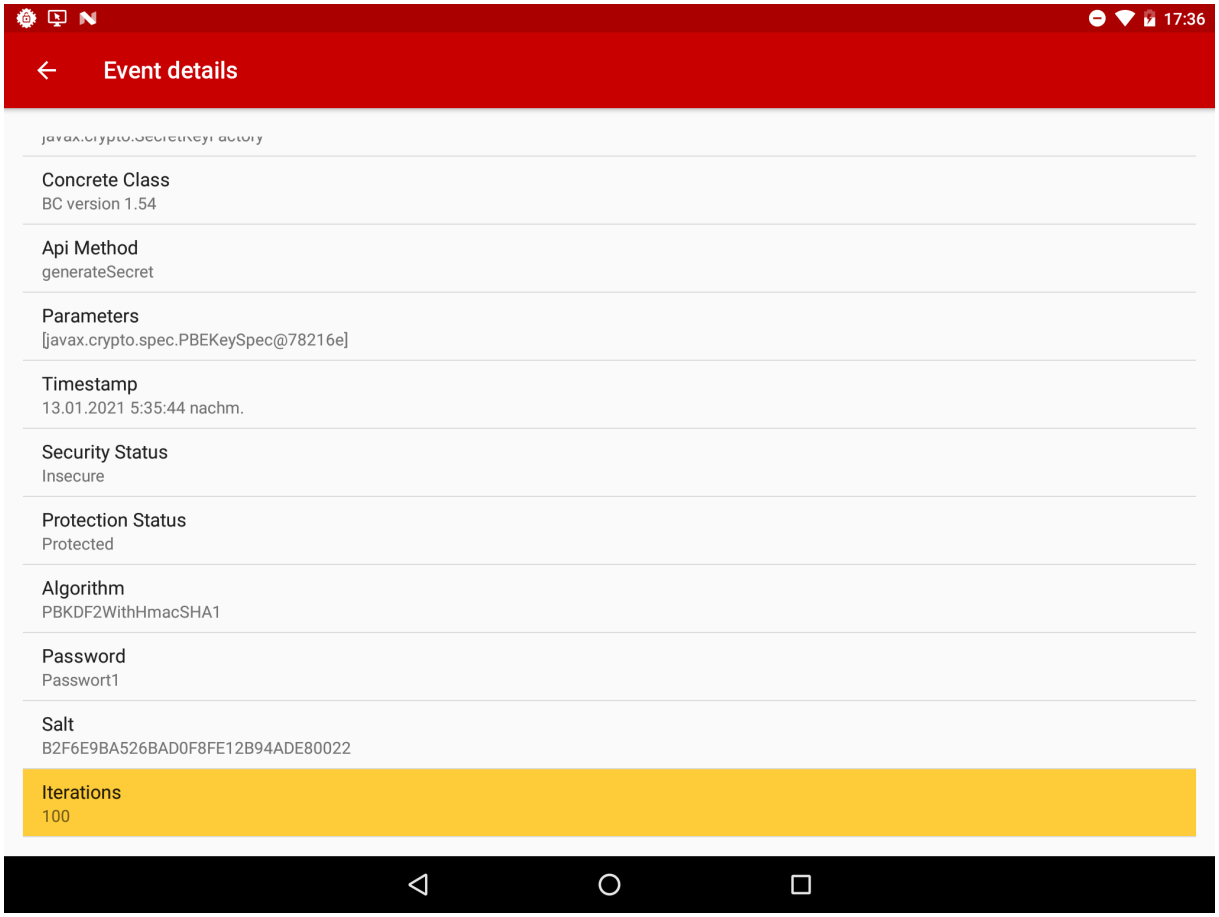

Figure 7.8: CryptoPatcher detected an insecure iteration count parameter

In order to confirm the correctness of this finding, we once more utilise the JADX decompiler. From the obtained reverse-engineered Java source code, we can identify the place that instantiates the `PBEKeySpec` object for use in the `SecretKeyGenerator` API. The relevant parts are illustrated in Section 7.1.3. We can observe that the method `com.er.mo.libs.secureutils.e.b()` calls the `com.er.mo.libs.secureutils.d.a()` function, which uses its third parameter as the iteration count in the creation of the `PBEKeySpec` object. Since the caller here passes the value 100 as the third parameter, we have proven that CryptoPatcher's claims were indeed correct.

```
package com.er.mo.libs.secureutils.e;

public class b {
    private void h(int i, String str) {
        byte[] a = c.a(3, str);
        byte[] bArr = new byte[16];
        System.arraycopy(a, 0, bArr, 0, 16);
        byte[] bArr2 = new byte[16];
        System.arraycopy(a, 16, bArr2, 0, 16);
        try {
            this.a.init(i, com.er.mo.libs.secureutils.d.a(str, bArr,
                100, 256), new IvParameterSpec(bArr2));
        } catch (InvalidAlgorithmParameterException |
            InvalidKeyException e) {
            throw new CryptoRuntimeException(e);
        }
```

```
    }
}
```

```
package com.er.mo.libs.secureutils;

public class d {
    public static SecretKeySpec a(String str, byte[] bArr, int i,
        int i2) {
        StringBuilder stringBuilder;
        if (VERSION.SDK_INT < 19) {
            throw new RuntimeException("Currently the KeyFactory
                supports android v-19 and newer");
        } else if (str == null || str.length() == 0) {
            throw new IllegalArgumentException("Invalid password!");
        } else if (bArr == null || bArr.length == 0) {
            throw new IllegalArgumentException("Invalid salt!");
        } else if (i <= 0) {
            stringBuilder = new StringBuilder();
            stringBuilder.append("Invalid iteration count: ");
            stringBuilder.append(i);
            throw new IllegalArgumentException(stringBuilder.
                toString());
        } else if (i2 == 128 || i2 == 192 || i2 == 256) {
            try {
                return new SecretKeySpec(SecretKeyFactory.
                    getInstance("PBKDF2WithHmacSHA1").generateSecret(
                    new PBEKeySpec(str.toCharArray(), bArr, i, i2)).
                    getEncoded(), "AES");
            } catch (NoSuchAlgorithmException |
                InvalidKeySpecException e) {
                throw new RuntimeException(e);
            }
        } else {
            stringBuilder = new StringBuilder();
            stringBuilder.append("Invalid key size: ");
            stringBuilder.append(i2);
            throw new IllegalArgumentException(stringBuilder.
                toString());
        }
    }
}
```

Listing 7.4: The code responsible for key derivation inside the My Passwords application

7.2 Performance

Injecting additional instructions into an existing application inevitably has some effect on the size of the resulting package. Within the boundaries of technical possibility, we are trying to keep the overhead as low as possible, so as not to overly strain a device's limited memory resources. Similarly, we are aiming to keep patch deployment durations as short as possible, since they have direct implications for the overall user experience of our approach. In order to provide some sense of the performance characteristics of our solution in these regards, Table 7.1 shows the APK sizes for all applications covered in our case studies before patch deployment, as well as the resulting file sizes and deployment durations for both dynamic and static rewriting.

Package	APK Size (Bytes)			Deployment Dur. (Seconds)	
	Original	Dynamic Rewr.	Static Rewr.	Dynamic Rewr.	Static Rewr.
Messenger Lite	11 052 557	11 868 500	11 678 834	20.7	59.1
Banggood	38 328 752	38 749 610	38 665 655	60.8	214.1
Password Saver	4 883 335	5 922 587	4 958 304	10.5	66.3
aWallet	2 705 974	3 733 992	2 752 903	9.8	21.8
My Passwords	3 133 775	4 111 466	3 177 335	8.2	33.8

Table 7.1: APK sizes and patch deployment durations for the different rewriting backends

As can already be observed from this small sample set, patch deployment durations are still somewhat detrimental to the overall user experience, given that a user has to wait for the patched version of a freshly installed application to be generated before it can be used. Although dynamic rewriting works faster than static rewriting, as mentioned in Section 5.3.5, it is still far too unreliable for general use. Consequentially, users frequently have to wait up to several minutes until the patch has been deployed via static rewriting.

Please note that the above figures can only serve as rough guidelines for the expected performance characteristics of a patching operation and its results. In addition to the differences between rewriting backends, these characteristics largely depend on a multitude of factors, such as the total size of DEX files within the APK package, the number of individual compressed files in the APK or the amount of calls to patched APIs.

Since the effect of the injected CryptoPatch patch on runtime performance of patched applications similarly varies with the number of calls to intercepted APIs, we refrain from including any benchmark results here. However, from our experience, we are confident that the minimal performance degradation caused by CryptoPatcher is not noticeable to end users for the vast majority of target applications.

7.3 Limitations

Although the above case studies approve the general efficacy of our approach for automatically mitigating security vulnerabilities induced by misuse of cryptographic APIs, there still are a few limitations that are worth pointing out here.

7.3.1 Compatibility

Many high-profile applications integrate some form of signature checks in order to prevent malicious modifications to their software's functionality from third parties. Through this measure, developers seek to protect the intellectual property contained in the application's executable code and try to prevent piracy. The latter problem commonly affects commercial applications and involves criminals re-releasing manipulated versions of popular legitimate software as a means to spread malware. Since this act on a technical level shares similarities with our approach, all countermeasures that try to prevent piracy affect CryptoPatcher as well. It thus has to be accepted that a compatibility rate of 100% can not be achieved, because a benign solution such as CryptoPatcher should not waste resources in the ethically questionable cat-and-mouse game of working around these countermeasures.

In addition to these cases where an application actively prevents being tampered with, CryptoPatcher currently is similarly incompatible to another set of programs that for some reason fail to function under the modifications applied by our system. A considerable source of problems can be found in the solution's

alteration of package names, particularly in conjunction with `ContentProvider` URIs derived from the application's package name. In order to improve the situation in this regard, future work could expand our patch to cover all possible interaction points between a patched app and the rest of the system that reference package names.

Furthermore, the problem that native code and custom implementations of cryptographic primitives are not covered by CryptoPatcher's monitoring and protection can not realistically be remediated. For these cases, the only possibility for misuse mitigations is a fix from the original developer.

7.3.2 Malicious Targets Applications

It is also worth noting that our solution currently does not assume target apps to have any malicious intentions. Instead, they are simply considered vulnerable due to ignorance or lack of competency on their developer's part. Consequentially, all our countermeasures rely on the presupposition that target applications make no attempts to actively evade the restrictions put up by the patching process.

A target application aware of being patched could for example dynamically alter its own executable code to get rid of CryptoPatcher's modifications. While this is generally more difficult for static rewriting where the CryptoPatch patch is baked into the target application's DEX file, the runtime manipulations of dynamic rewriting could be reverted relatively easily.

Although the CryptoPatcher system could be hardened against malicious target packages to some degree, it is to be expected that not all cases could be covered due to the very fact that our patch is running within the same process as the target application.

7.3.3 Usability

While we specifically designed our application to work as transparently to the user as possible, some minor nuisances of the Android system that are beyond our control slightly worsen the overall user experience.

In particular, recent versions of the Google Play Store include a service called *Play Protect*[9], which compares all installed applications with known malicious packages through a web server. In the default configuration, a dialog is shown whenever an unknown package is installed through sources other than Google Play, asking whether the application file should be uploaded to Google's servers for further examination. Since this dialog is also shown for the patched app versions produced by CryptoPatcher, the user has to manually intervene for every installation. Fortunately, Play Protect can be disabled in the Google Play application, so that it no longer interferes with CryptoPatcher. As long as the user only obtains programs through Google Play, the disabled setting has no implications for the device's security. This is because all software available from this official repository is scanned already when uploaded by developers.

A few minor usability issues stem from the fact that CryptoPatcher keeps the original applications installed alongside the patched versions, so as to maintain compatibility with the common update route via Google Play. As a result of this practise, two icons are shown in the app launcher for every installed application. Only the removed colours of the black-and-white icon that signify the suspended state of the original version provide visual distinction. As a solution to this minor nuisance, a custom launcher could be developed that hides these suspended applications.

[9]Help protect against harmful apps with Google Play Protect: https://support.google.com/accounts/answer/2812853?hl=en

7.4 Summary

In this chapter, we demonstrated two case studies that proved how CryptoPatcher is effective in preventing attacks against applications that misuse cryptographic primitives. It was shown that our solution is capable of prohibiting a MITM attack on a third-party program's insufficiently protected TLS communication. Furthermore, we traced down API calls inside reverse-engineered target applications in order to validate CryptoPatcher's claims. Performance-wise, we come to the conclusion that while the effects of patching on APK sizes are minimal, patch deployment speed still leaves room for improving the overall user experience. Lastly, we highlighted the limitations of our solution, including some that are inherently linked to the technicalities of our approach, and others that can be addressed in future work.

Chapter 8

Conclusion

As mobile security and the Android OS in particular are moving into the focus of security researchers, an increasing number of vulnerabilities are being uncovered that are caused by applications' misuse of cryptographic primitives. Despite all efforts of Google as the platform provider to eradicate this critical threat to end user's data confidentiality, a considerable portion of programs stay susceptible to attacks due to plain negligence or incompetence on their developers' part. In order to still guard users that are for one reason or another reliant on the operation of an insecure product, some form of third-party protection is needed.

In this thesis, we address this problem with our CryptoPatcher solution, a software capable of patching security vulnerabilities induced by crypto API misuses in third-party Android packages. As an Android application targeted for installation on end consumers' devices, CryptoPatcher runs a background service observing changes to the packages installed on the system. For every newly found application, our system automatically produces and deploys a patched version without requiring any intervention from the user. As a result, our solution can be used by novice and advanced users alike. For the latter group, the CryptoPatcher control application integrates detailed monitoring functionality, offering insights into the specific used cryptographic APIs, the chosen parameterisation and potential vulnerabilities of installed programs.

For developing and deploying the CryptoPatch patch that forms the core of the CryptoPatcher system, we devised our own Android Patch Development Kit. Through an annotation processor and custom deployment infrastructure, the APDK provides all functionality needed for modifying compiled Android APK files in a wide variety of ways. It is possible to perform sophisticated changes to the manifest of a target application, intercept execution of precisely targeted functions in the DEX code or inject native libraries and resources. Through abstraction of the code rewriting functionality, patches developed with the APDK can be deployed using either a static or a dynamic rewriting backend, each with unique characteristics in terms of compatibility and performance. These efforts not only laid a solid foundation for the task at hand in this thesis, they also provide a good starting ground for designing solutions to similar problems in related fields.

The specific crypto API misuses covered by our solution include those revolving around the `SSLSocket` API for TLS communication, the `Cipher` API for encryption and decryption, the `SecretKeyFactory` API for derivation of cryptographic keys from user-supplied passwords and the `SecretRandom` API for generating random numbers. For most of the known mistakes made in programs utilising these interfaces, CryptoPatcher can offer mitigations that are completely transparent to the targeted application and the user. For those vulnerabilities that cannot possibly be automatically remediated, our system still produces extensive information and warnings in its monitor UI.

In order to demonstrate how CryptoPatcher can effectively protect user data against exploits of the security shortcomings described above, we selected a set of five vulnerable applications as the subjects

of case studies. Through simulated attacks and reverse-engineering of the target softwares' source code, we were able to not only validate all claims our software made with regards to the test subject's security, but could also prove that our solution successfully prevented the disclosure of critical data caused by the identified problems.

8.1 Future Work

Since extensibility has been a core goal of our implementation from the very start, CryptoPatcher can be trivially expanded to cover a broader set of security vulnerabilities and also to further its compatibility and performance. Specifically, we envision the following future work:

8.1.1 Cover more Crypto API Misuses

Although CryptoPatcher already covers the most common misuses of cryptographic APIs, there still are a few more candidates for inclusion in our system. Most prominently, a patch could dynamically upgrade all plain HTTP connections to HTTPS where available. The concept was not realised for this thesis due to the anticipated difficulties in ascertaining whether a found HTTPS API endpoint actually matches the one on the original HTTP server. An additional idea could be to automatically replace references to the unsafe MD5 hashing algorithm with a safe alternative.

8.1.2 Improve Compatibility

As of now, CryptoPatcher is still incompatible to some applications, even beyond those that actively prevent manipulations. A whole group of additional software is not supported either due to use of proprietary technology in their build process or because of some specific way in which they depend on their original package name. In order to remediate these issues, the system has to be tested in a larger scale, and the identified problems be investigated. Additionally, the current issues of the dynamic rewriter on most production packages have to be solved.

8.1.3 Optimise Performance

Deployment durations have an immediate influence on the user experience of the CryptoPatcher application. In order to further reduce the delay before users can launch installed packages, it is imperative to benchmark the current implementation and identify those portions that can be optimised. For further improvements, a potential solution could be the transfer of some parts to more performant native code.

8.1.4 Enhance Customisability

Another opportunity for further work is provided by the limitations of the current configuration options within the CryptoPatcher control application. Specifically, a single switch is used for toggling the state of all the mitigations included in the CryptoPatch patch. More selective control could allow users to strike a more precise balance between keeping a target program fully operational and mitigating all its vulnerabilities.

8.1.5 Additional Patches

Since our APDK is entirely decoupled from the CryptoPatch patch, it could be used for any other solution that requires modification of third-party applications. For example, the system provides a powerful foundation for research into various aspects of applications' runtime behaviour. More practical applications are similarly conceivable that add functionality to existing software. Further pursuing this idea, another option is a system where third-party patches can be distributed through a (curated) central marketplace, initiating a whole ecosystem similar to that of the Xposed framework.

Bibliography

[1] Aisha I. Ali-Gombe, Irfan Ahmed, I. I. I. Golden G. Richard, and Vassil Roussev. *AspectDroid: Android App Analysis System*. Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016. ACM, 2016, pages 145–147. doi:10.1145/2857705.2857739 (cited on page 20).

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. *Instrumenting Android and Java Applications as Easy as abc*. Runtime Verification - 4th International Conference, RV 2013. Proceedings. Volume 8174. Lecture Notes in Computer Science. Springer, 2013, pages 364–381. doi:10.1007/978-3-642-40787-1_26. https://doi.org/10.1007/978-3-642-40787-1%5C_26 (cited on page 20).

[3] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. *Android security framework: extensible multi-layered access control on Android*. Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014. ACM, 2014, pages 46–55. doi:10.1145/2664243.2664265 (cited on page 21).

[4] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. *Boxify: Full-fledged App Sandboxing for Stock Android*. 24th USENIX Security Symposium, USENIX Security 15. USENIX Association, 2015, pages 691–706. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/backes (cited on page 22).

[5] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. *AppGuard - Enforcing User Requirements on Android Apps*. Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013. Proceedings. Volume 7795. Lecture Notes in Computer Science. Springer, 2013, pages 543–548. doi:10.1007/978-3-642-36742-7_39. https://doi.org/10.1007/978-3-642-36742-7%5C_39 (cited on page 20).

[6] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Kevin R. B. Butler, and Abdulrahman Alkhelaifi. *Securing SSL Certificate Verification through Dynamic Linking*. Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014, pages 394–405. doi:10.1145/2660267.2660338 (cited on page 19).

[7] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. *NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running stock Android*. Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2015. ACM, 2015, pages 27–38. doi:10.1145/2808117.2808122 (cited on page 22).

[8] Damjan Buhov, Markus Huber, Georg Merzdovnik, and Edgar R. Weippl. *Pin it! Improving Android network security at runtime*. 2016 IFIP Networking Conference, Networking 2016 and Workshops. IEEE Computer Society, 2016, pages 297–305. doi:10.1109/IFIPNetworking.2016.7497238 (cited on pages 2, 19).

[9] Damjan Buhov, Markus Huber, Georg Merzdovnik, Edgar R. Weippl, and Vesna Dimitrova. *Network Security Challenges in Android Applications*. 10th International Conference on Availability,

Reliability and Security, ARES 2015. IEEE Computer Society, 2015, pages 327–332. doi:10.1109/ARES.2015.59 (cited on page 2).

[10] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. *Evaluation of Cryptography Usage in Android Applications*. BICT 2015, Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS). ICST/ACM, 2015, pages 83–90. http://dl.acm.org/citation.cfm?id=2954820 (cited on page 2).

[11] Haehyun Cho, Jeong Hyun Yi, and Gail-Joon Ahn. *DexMonitor: Dynamically Analyzing and Monitoring Obfuscated Android Applications*. IEEE Access 6 (2018), pages 71229–71240. doi:10.1109/ACCESS.2018.2881699 (cited on page 21).

[12] Fred Chung. *Security Enhancements in Jelly Bean*. Feb 2013. https://android-developers.googleblog.com/2013/02/security-enhancements-in-jelly-bean.html (cited on page 47).

[13] Valerio Costamagna and Cong Zheng. *ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime*. Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security, IMPS 2016, co-located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2016). Volume 1575. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pages 20–28. http://ceur-ws.org/Vol-1575/paper%5C_10.pdf (cited on page 22).

[14] Deshun Dai, Ruixuan Li, Junwei Tang, Ali Davanian, and Heng Yin. *Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android*. Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, SACMAT 2020. ACM, 2020, pages 25–32. doi:10.1145/3381991.3395608 (cited on page 23).

[15] Shuaifu Dai, Tao Wei, and Wei Zou. *DroidLogger: Reveal suspicious behavior of Android applications via instrumentation*. 2012 7th international conference on computing and convergence technology (ICCCT). IEEE. 2012, pages 550–555 (cited on page 20).

[16] Benjamin Davis and Hao Chen. *RetroSkeleton: retrofitting android apps*. The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13. ACM, 2013, pages 181–192. doi:10.1145/2462456.2464462 (cited on page 20).

[17] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. *I-arm-droid: A rewriting framework for in-app reference monitors for android applications*. Mobile Security Technologies 2012.2 (2012), pages 1–7 (cited on page 20).

[18] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. *An empirical study of cryptographic misuse in android applications*. 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13. ACM, 2013, pages 73–84. doi:10.1145/2508859.2516693 (cited on pages 2–3).

[19] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones*. 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010. USENIX Association, 2010, pages 393–407. http://www.usenix.org/events/osdi10/tech/full%5C_papers/Enck.pdf (cited on page 21).

[20] 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*. the ACM Conference on Computer and Communications Security, CCS'12. ACM, 2012, pages 50–61. doi:10.1145/2382196.2382205 (cited on page 1).

[21] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. *Rethinking SSL development in an appified world*. 2013 ACM SIGSAC Conference on Computer and Com-

munications Security, CCS'13. ACM, 2013, pages 49–60. doi:10.1145/2508859.2516655 (cited on page 19).

[22] Wenhao Fan, Yaohui Sang, Daishuai Zhang, Ran Sun, and Yuan'an Liu. *DroidInjector: A process injection-based dynamic tracking system for runtime behaviors of Android applications*. Computers & Security 70 (2017), pages 224–237. doi:10.1016/j.cose.2017.06.001 (cited on page 22).

[23] Jyoti Gajrani, Meenakshi Tripathi, Vijay Laxmi, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. *sPECTRA: A precise framEwork for analyzing CrypTographic vulneRabilities in Android apps*. 14th IEEE Annual Consumer Communications & Networking Conference, CCNC 2017. IEEE, 2017, pages 854–860. doi:10.1109/CCNC.2017.7983245 (cited on page 2).

[24] Jun Gao, Pingfan Kong, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. *Negative results on mining crypto-API usage rules in Android apps*. Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019. IEEE, 2019, pages 388–398. doi:10.1109/MSR.2019.00065 (cited on page 2).

[25] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. *The most dangerous code in the world: validating SSL certificates in non-browser software*. the ACM Conference on Computer and Communications Security, CCS'12. ACM, 2012, pages 38–49. doi:10.1145/2382196.2382204 (cited on page 1).

[26] Paul A. Grassi, James L. Fenton, Elaine M. Newton, Ray A. Perlner, Andrew R. Regenscheid, William E. Burr, and Justin P. Richer. *NIST Special Publication 800-63B: Digital Identity Guidelines - Authentication and Lifecycle Management*. Online. Jun 2017. https://pages.nist.gov/800-63-3/sp800-63b.html#sec5 (cited on page 7).

[27] Dongsoo Ha, Wenhui Jin, and Heekuck Oh. *REPICA: Rewriting Position Independent Code of ARM*. IEEE Access 6 (2018), pages 50488–50509. doi:10.1109/ACCESS.2018.2868411 (cited on page 21).

[28] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. *SIF: a selective instrumentation framework for mobile applications*. The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13. ACM, 2013, pages 167–180. doi:10.1145/2462456.2465430 (cited on page 20).

[29] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. *ASM: A Programmable Interface for Extending Android Security*. Proceedings of the 23rd USENIX Security Symposium. USENIX Association, 2014, pages 1005–1019. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/heuser (cited on page 21).

[30] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd D. Millstein. *Dr. Android and Mr. Hide: fine-grained permissions in android applications*. SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012. ACM, 2012, pages 3–14. doi:10.1145/2381934.2381938 (cited on page 21).

[31] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. Online. Sep 2000. https://tools.ietf.org/html/rfc2898 (cited on page 7).

[32] Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. *Reptor: Enabling API Virtualization on Android for Platform Openness*. Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17. ACM, 2017, pages 399–412. doi:10.1145/3081333.3081341 (cited on page 20).

[33] Alex Klyubin. *Some SecureRandom Thoughts*. Online. Aug 2013. https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html (cited on page 47).

[34] Sung-Hoon Lee, Seung-Hyun Kim, Soohyung Kim, and Seung-Hun Jin. *AppWrapper: Patching Security Functions with Dynamic Policy on Your Insecure Android Apps*. 2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops. IEEE Computer Society, 2018, pages 36–41. doi:10.1109/ISSREW.2018.00-34 (cited on page 21).

[35] Jierui Liu, Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. *InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications*. IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017. IEEE Computer Society, 2017, pages 502–506. doi:10.1109/SANER.2017.7884662 (cited on page 20).

[36] Tongbo Luo, Cong Zheng, Z. Xu, and Xin Ouyang. *Anti-Plugin: Don 't Let Your App Play As An Android Plugin*. Black Hat Asia 2017. 2017 (cited on page 23).

[37] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. *CDRep: Automatic Repair of Cryptographic Misuses in Android Applications*. Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016. ACM, 2016, pages 711–722. doi:10.1145/2897845.2897896 (cited on page 19).

[38] Collin Mulliner, William K. Robertson, and Engin Kirda. *VirtualSwindle: an automated attack against in-app billing on android*. 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14. ACM, 2014, pages 459–470. doi:10.1145/2590296.2590335 (cited on page 22).

[39] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. *Source Attribution of Cryptographic API Misuse in Android Applications*. Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018. ACM, 2018, pages 133–146. doi:10.1145/3196494.3196538 (cited on page 2).

[40] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. *Apex: extending Android permission model and enforcement with user-defined runtime constraints*. Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010. ACM, 2010, pages 328–332. doi:10.1145/1755688.1755732 (cited on page 21).

[41] 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*. 24th USENIX Security Symposium, USENIX Security 15. USENIX Association, 2015, pages 239–254. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/oltrogge (cited on page 8).

[42] OWASP. *Certificate and Public Key Pinning*. Online. Accessed on: Feb. 12, 2020. https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning (cited on pages 2, 8).

[43] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. *DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android*. Ninth International Conference on Availability, Reliability and Security, ARES 2014. IEEE Computer Society, 2014, pages 40–49. doi:10.1109/ARES.2014.13 (cited on page 21).

[44] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. *Studying TLS Usage in Android Apps*. Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017. ACM, 2017, pages 350–362. doi:10.1145/3143361.3143400 (cited on page 2).

[45] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. *MOSES: supporting operation modes on smartphones*. 17th ACM Symposium on Access Control Models and Technologies, SACMAT '12. ACM, 2012, pages 3–12. doi:10.1145/2295136.2295140 (cited on page 21).

[46] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. *FireDroid: hardening security in almost-stock Android*. Annual Computer Security Applications Conference, ACSAC '13. ACM, 2013, pages 319–328. doi:10.1145/2523649.2523678 (cited on page 22).

[47] Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. *Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications*. IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, DASC 2014. IEEE Computer Society, 2014, pages 75–80. doi:10.1109/DASC.2014.22 (cited on page 2).

[48] Dongwan Shin and Jiangfeng Sun. *An Empirical Study of SSL Usage in Android Apps*. 2018 International Carnahan Conference on Security Technology, ICCST 2018. IEEE, 2018, pages 1–5. doi:10.1109/CCST.2018.8585431 (cited on page 2).

[49] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. *SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps*. 21st Annual Network and Distributed System Security Symposium, NDSS 2014. The Internet Society, 2014. https://www.ndss-symposium.org/ndss2014/smv-hunter-large-scale-automated-detection-ssltls-man-middle-vulnerabilities-android-apps (cited on page 1).

[50] Junwei Tang, Jingjing Li, Ruixuan Li, Hongmu Han, Xiwu Gu, and Zhiyong Xu. *SSLDetecter: Detecting SSL Security Vulnerabilities of Android Applications Based on a Novel Automatic Traversal Method*. Security and Communication Networks 2019 (2019), 7193684:1–7193684:20. doi:10.1155/2019/7193684 (cited on page 2).

[51] Vasant Tendulkar and William Enck. *An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities*. CoRR abs/1410.7745 (2014). arXiv: 1410.7745. http://arxiv.org/abs/1410.7745 (cited on pages 2, 19).

[52] Philipp Von Styp-Rekowsky, Sebastian Gerling, Michael Backes, and Christian Hammer. *Idea: Callee-Site Rewriting of Sealed System Libraries*. Engineering Secure Software and Systems - 5th International Symposium, ESSoS 2013. Proceedings. Volume 7781. Lecture Notes in Computer Science. Springer, 2013, pages 33–41. doi:10.1007/978-3-642-36563-8_3. https://doi.org/10.1007/978-3-642-36563-8%5C_3 (cited on page 22).

[53] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. *DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices*. 22nd Annual Network and Distributed System Security Symposium, NDSS 2015. The Internet Society, 2015. https://www.ndss-symposium.org/ndss2015/deepdroid-dynamically-enforcing-enterprise-policy-android-devices (cited on page 23).

[54] Yingjie Wang, Xing Liu, Weixuan Mao, and Wei Wang. *DCDroid: automated detection of SSL/TLS certificate verification vulnerabilities in Android apps*. Proceedings of the ACM Turing Celebration Conference - China, ACM TUR-C 2019. ACM, 2019, 137:1–137:9. doi:10.1145/3321408.3326665 (cited on page 2).

[55] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. *AirBag: Boosting Smartphone Resistance to Malware Infection*. 21st Annual Network and Distributed System Security Symposium, NDSS 2014. The Internet Society, 2014. https://www.ndss-symposium.org/ndss2014/airbag-boosting-smartphone-resistance-malware-infection (cited on page 21).

[56] Jiayun Xie, Xiao Fu, Xiaojiang Du, Bin Luo, and Mohsen Guizani. *AutoPatchDroid: A framework for patching inter-app vulnerabilities in android application*. IEEE International Conference on Communications, ICC 2017. IEEE, 2017, pages 1–6. doi:10.1109/ICC.2017.7996682 (cited on page 21).

[57] Rubin Xu, Hassen Saïdi, and Ross J. Anderson. *Aurasium: Practical Policy Enforcement for Android Applications*. Proceedings of the 21th USENIX Security Symposium. USENIX Association, 2012,

pages 539–552. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu%5C_rubin (cited on page 22).

[58] Chaoting Xuan, Gong Chen, and Erich Stuntebeck. *DroidPill: Pwn Your Daily-Use Apps*. Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017. ACM, 2017, pages 678–689. doi:10.1145/3052973.3052986 (cited on page 23).

[59] Wei You, Bin Liang, Wenchang Shi, Shuyang Zhu, Peng Wang, Sikefu Xie, and Xiangyu Zhang. *Reference hijacking: patching, protecting and analyzing on unmodified and non-rooted android devices*. Proceedings of the 38th International Conference on Software Engineering, ICSE 2016. ACM, 2016, pages 959–970. doi:10.1145/2884781.2884863 (cited on page 21).

[60] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. *App in the Middle: Demystify Application Virtualization in Android and its Security Threats*. Proc. ACM Meas. Anal. Comput. Syst. 3.1 (2019), 17:1–17:24. doi:10.1145/3322205.3311088 (cited on page 23).

[61] Mu Zhang and Heng Yin. *AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications*. 21st Annual Network and Distributed System Security Symposium, NDSS 2014. The Internet Society, 2014. https://www.ndss-symposium.org/ndss2014/appsealer-automatic-generation-vulnerability-specific-patches-preventing-component-hijacking (cited on page 21).

[62] Min Zheng, Mingshen Sun, and John C. S. Lui. *DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability*. International Wireless Communications and Mobile Computing Conference, IWCMC 2014. IEEE, 2014, pages 128–133. doi:10.1109/IWCMC.2014.6906344 (cited on page 22).

[63] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. *Hybrid User-level Sandboxing of Third-party Android Apps*. Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15. ACM, 2015, pages 19–30. doi:10.1145/2714576.2714598 (cited on page 23).