CryptoShield - Automatic On-Device Mitigation for Crypto API Misuse in Android Applications

Florian Draschbacher
Graz University of Technology and Secure Information Technology Center Austria
Graz, Austria
florian.draschbacher@iaik.tugraz.at

Johannes Feichtner
Dynatrace Austria GmbH
Linz, Austria
johannes.feichtner@dynatrace.com

ABSTRACT

Misuse of cryptographic APIs remains one of the most common flaws in Android applications. The complexity of cryptographic APIs frequently overwhelms developers. This can lead to mistakes that leak sensitive user data to trivial attacks. Despite herculean efforts by platform provider Google, countermeasures introduced so far were not successful in preventing these flaws. Users remain at risk until an effective systemic mitigation has been found.

In this paper, we propose a practical solution that mitigates crypto API misuse in compiled Android applications. It enables users to protect themselves against misuse exploitation until the research community has identified an effective long-term solution. CryptoShield consists of generic mitigation procedures for the most critical crypto API misuse scenarios and an implementation that autonomously extends protection onto all applications on an unrooted Android device. Our on-device CryptoShield Agent injects an instrumentation module into application packages, where it can intercept crypto API calls for detecting misuse and applying mitigations. Our solution was designed for real-world applicability. It retains the update flow through Google Play and can be integrated into existing MDM infrastructure.

As a demonstration of CryptoShield's efficiency and efficacy, we conduct automated (1604 apps) and manual (99 apps) analyses on the most popular applications from Google Play, as well as measurements on synthetic benchmarks. Our solution mitigates crypto API misuse in 96 % of all vulnerable apps, while retaining full functionality for 92 % of all apps. On-device instrumentation takes roughly 11 seconds per application package on average, with minimal impact on package size (5 %) and negligible runtime overhead (571 ms on average app launches, 101 ms worst-case mitigation overhead per crypto API call).

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**; **Software and application security**.

KEYWORDS

Android, Security, Mitigation, Instrumentation, Crypto API Misuse

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0098-9/23/07.
https://doi.org/10.1145/3579856.3582832

1 INTRODUCTION

Several publications over the last years have uncovered that crypto API misuse causes severe vulnerabilities in a large number of mobile applications. Android, the most popular mobile operating system, was shown to be particularly affected. Most recently, Oltrogge et al. [12] found that *37 %* of 15,000 statically analyzed applications contained *mistakes in their code for validating TLS certificates*. For a broader definition of crypto API misuse that considers framework-provided TLS and crypto primitives[1], Rahaman et al. [15] even reported a misuse prevalence of 91 %.

As a reaction to the first alarming reports a decade ago, platform provider Google started multiple initiatives to eliminate TLS misuse and insecure parametrization of cryptographic APIs in general. The Android platform saw the introduction of improved APIs, reworked documentation, detailed linter rules, and automated checks for software uploaded to the Google Play Store. Still, researchers keep showing that all efforts so far were largely ineffective.

Despite the consequences of widespread crypto API misuse on data security and the ineffectiveness of official countermeasures, very few publications have so far attempted to mitigate crypto API misuse in compiled applications. However, until an effective long-term solution has been identified and implemented by Google, third-party solutions are user's only option for protecting against the severe security repercussions of crypto API misuse.

Buhov et al. [3] proposed a dynamic approach in the form of a Cydia Substrate module. However, their exclusive focus on retroactively adding TLS pinning to applications did not acknowledge the full scope of the problem, which affects a far broader range of cryptographic APIs. Additionally, the presented module utilized the obsolete Cydia Substrate hooking framework and required a *rooted device*, thereby considerably limiting its real-world applicability. An important first step towards mitigating other classes of crypto API misuse was made by Ma et al. [10]. They suggested an approach based on static analysis and static rewriting of Dalvik Executable (DEX) files. However, the reliance on static analysis of the control flow graph limited the practicality of their solution (19 seconds patch generation *per misuse*; only the first misuse per method could be mitigated).

We therefore propose CryptoShield as the first *practical* on-device method that mitigates both TLS and crypto primitive misuse on unmodified Android systems.

Our key contributions are:

- We devise *application-agnostic mitigation procedures* for 10 common classes of crypto API misuse, including flaws in

[1] We use the term *cryptographic APIs* to refer to Java APIs for TLS or crypto primitives.

the integration of TLS, Ciphers, Password-Based Encryption (PBE), Key Storage and Cryptographically Secure Random Number Generators (CSRNG). Our mitigations operate transparently to application code and require only runtime information. Thus, our method does *not* suffer from the imprecisions and resource-intensiveness of traditional patching approaches that employ static analysis of the control flow graph.
- We propose a self-contained *on-device component* named CryptoShield Agent (CSA) for deploying our mitigations to all applications on a target Android device. An instrumentation module intercepts crypto API calls to detect misuse and employ our mitigation procedures. Our implementation works on unrooted Android devices and does not require any interaction from the user. Application updating through Google Play remains fully functional. [2]
- We demonstrate the efficiency and efficacy of CryptoShield through automated and manual analyses on the 1604 and 99 most popular free applications, respectively, from the Google Play store. Additionally, we evaluate CryptoShield's detection precision on a synthetic benchmark, provide exact per-API runtime overheads and carry out a case study that illustrates how our solution can mitigate critical security vulnerabilities in two widely-used Android applications.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Android OS and its cryptographic APIs. Section 3 outlines the addressed problem and the assumed attacker model. Section 4 describes our mitigation procedures, before Section 5 covers our prototype implementation, which is subsequently evaluated in Section 6. We discuss limitations of our system and plans for future work in Section 7, highlight related publications in Section 8, and conclude this paper in Section 9.

2 BACKGROUND

In this section, we provide a short overview of cryptographic APIs in the Android framework, introduce the Android package format, and outline the OS's application runtime.

2.1 Android Crypto APIs & Common Misuse

Android applications can take advantage of Java Cryptography Architecture (JCA) APIs for cryptographic functionality. The most common pitfalls with JCA APIs are that some default to insecure configurations, while others completely rely on developers' choice of secure parameters. Together with a lack of clarity in the official documentation, this has led to developers making serious mistakes in protecting their application's data. Since the exact ramifications of the particular misuse classes have already been thoroughly discussed in previous publications [4, 5, 10], we only provide a very condensed overview of the relevant points here.

For SSL/TLS (provided by the `SSLSocket` class), applications often implement `TrustManagers` that fail to properly verify the certificate chain presented by the server or `HostnameVerifiers` that don't confirm correspondence between a server's hostname and its presented TLS certificate. Both issues leave applications vulnerable to Man-In-The-Middle (MITM) attacks that can compromise all transmitted data. In Android 7.0, Google introduced the Network Security Configuration (NSC) system that allows developers to conveniently set up certificate pinning and trusted self-signed certificates. Although this possibility eliminated most needs for custom `TrustManagers` or `HostnameVerifiers`, many applications remained vulnerable in some form [12]. Additionally, a considerable portion of Android applications still use the unprotected HTTP protocol in their communication with servers [16].

Applications that utilize the Cipher API for data encryption and decryption were found to commonly use hardcoded or predictable initialization vectors, which breaks the indistinguishability against chosen plaintext attacks (IND-CPA) property of symmetric ciphers. Two ciphertexts can reveal similarities between their original plain texts. Electronic Code Book (ECB) mode for symmetric block ciphers exhibits the same vulnerability to chosen plaintext attacks. Since ECB is the JCA's default mode of operation, it can still be found in a considerable amount of applications.

Password-Based Encryption (PBE) in the JCA can be accessed through the SecretKeyFactory API. If it is supplied with hardcoded passwords or reused or predictable salt values, derived keys can either be compromised immediately or with considerably less effort than in bruteforce attacks against properly parameterized PBE.

Hardcoded passwords pose a problem to the KeyStore API for generating and storing cryptographic keys as well. They may lead to compromitation of all stored keys and thus any related encrypted data.

Lastly, Java offers a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) through the SecureRandom API. Android's default implementation of the API has had a history of severe flaws caused by the reuse of seed material that led to the generation of a predictable sequence of numbers. Although these flaws have been mitigated at the framework level in recent versions of the Android OS, vulnerable legacy devices remain in use.

2.2 Android Application Package Format

Android applications are deployed in a special file format called Android Package (APK). On the outmost layer, it consists of a ZIP container signed by the application developer to ensure that only updates from the same author can replace an original installation. The content of the container follows a particular structure that is unique to the APK format. One or more Dalvik Executable (DEX) files store the class structure and program byte code of the application. The `AndroidManifest.xml` file encodes the contract between the OS and the application, declaring the supported functionality and required permissions or device capabilities. Native libraries are organized in dedicated folders within the APK archive. When obtained from Google Play, an app commonly is installed as a set of multiple Split APK files. The set comprises a base APK file storing DEX files, and several additional splits for device-specific resources or native code. As a key observation for the functionality of our solution, we point out that APK files reside in public storage on the Android device after their installation. PackageManager APIs can be used for locating all APK files for a given package. On Android versions up to 10, even completely unprivileged applications can access installed APK files, i.e. no permission is needed *at all*. Starting

[2] Source code is available at https://extgit.iaik.tugraz.at/fdraschbacher/cryptoshield

from Android 11, a suitable `<queries>` element in the application manifest or the `QUERY_ALL_PACKAGES` permission is required.

2.3 Android Runtime

Android applications are typically written in Java or Kotlin and compiled into architecture-independent Dalvik bytecode. In current Android versions, the Android Runtime (ART) takes over the responsibility of executing this bytecode on the device. Applications may additionally implement parts of their functionality in native shared libraries that are dynamically linked into the runtime process. It is worth noting that shared libraries are in a position that allows them to manipulate the internal data structures of the ART runtime.

3 PROBLEM STATEMENT, CHALLENGES, AND THREAT MODEL

In this section, we state the problem that needs to be addressed, derive a set of core goals, highlight the challenges involved in finding a solution and show how CryptoShield was designed to overcome them.

3.1 Problem Statement

We seek to provide a mitigation for crypto API misuse that enables users to protect themselves until the research community has identified an effective long-term solution for this widespread problem. Our goal is to cover as many classes of critical crypto misuse as feasible without breaking the functionality of target applications. We envision deployment by security-conscious individuals, as well as corporate organizations that wish to ensure data confidentiality on their complete fleet of Android devices no matter what applications employees utilize.

The core design goals of our solution are thus:

3.1.1 Practicality. We strive to propose a design that seamlessly integrates into the everyday workflows of real-world Android users.

3.1.2 User-Centricity. Our solution is intended to put users in control of application security. They no longer have to rely on developers properly securing their applications.

3.1.3 Compatibility. We consider compatibility with as many third-party applications as possible a crucial feature of a workable solution.

3.2 Attacker Model & Crypto Rules

Our attacker model assumes a malicious party that tries to gain access to a vulnerable application's stored or transmitted data. To this end, the attacker attempts to take advantage of vulnerabilities caused by the target application's misuse of cryptographic APIs. The modeled mode of attack depends on the specific crypto API misuse that is being exploited. For vulnerabilities in TLS host authentication or unprotected HTTP connections, the attacker is assumed to hold a Man-In-The-Middle (MITM) position somewhere between the client and server that enables the interception and modification of network traffic. Attacks against improper parametrization of cryptographic primitives are assumed to be carried out through a malicious application on the same device. It is worth noting that vulnerabilities in the *implementations* of the cryptographic APIs that can be exploited irrespective of their proper employment are considered out of scope. Additionally, we assume vulnerable applications to be benign in general, i.e. not actively trying to evade the mitigations put in place by our solution.

3.2.1 Crypto Rules. We define the vulnerabilities addressed in this paper as violations of well-established rules that consumers have to respect for ensuring the security of specific cryptographic APIs. As the basis for our rules, we use the comprehensive and recent set collected by Rahaman et al. [15], which we further refine for the scenario of vulnerability mitigation. We start this refinement process by ensuring the high level of precision needed for mitigating detected vulnerabilities while keeping negative side effects as rare as possible. Specifically, we modify our rules to only disallow non-cryptographic random number generators when their output is used in cryptographic operations. Similarly, we believe there are legitimate use cases for specifying custom TLS TrustManagers and HostnameVerifiers, so we strive to only mitigate cases where the custom implementations introduce vulnerabilities. Lastly, our experiments indicated that insecure cryptographic hash functions and hardcoded keys are so frequently used in communication with servers that mitigating them breaks most affected apps. Consequentially, we remove these two rules from our set. In their place, we introduce new rules concerned with reuse of nonce values, which has the same security repercussions as hardcoded values. The rules addressed by our mitigations are thus:

R01: Don't use unprotected HTTP connections.
R02: Don't verify TLS connections in insecure ways.
R03: Don't use predictable passwords for key stores.
R04: Don't use predictable passwords for key derivations.
R05: Don't use ECB mode for symmetric ciphers.
R06: Don't use predictable IVs for symmetric ciphers.
R07: Don't reuse IVs for symmetric ciphers.
R08: Don't use predictable salt values for key derivations.
R09: Don't reuse salt values for key derivations.
R10: Don't use predictable CSRNG seeds.

We use the term *predictable* to refer to values that were hardcoded or derived from an improper source of randomness. *Reused* values may have been securely generated but are used as a nonce multiple times, violating their uniqueness requirement.

3.2.2 Severity of Vulnerabilities. Our mitigations only address crypto misuse vulnerabilities of high or medium severity. For vulnerabilities that are hard to exploit or only offer limited gain for attackers, we argue that the potential of side effects introduced by mitigations (cf. Subsection 4.1) voids the small gain in security.

Severity ratings follow the reasoning by Rahaman et al. [15] that is guided by the difficulty and gain of exploitation. HTTP connections (**R01**) lack any form of protection for transferred data, so are trivially rated highly severe. Vulnerabilities in TLS connections (**R02**) immediately void all security properties of the protocol, so must be considered highly severe as well. Similarly, using predictable passwords (**R03**, **R04**) voids the confidentiality property of all operations the derived or stored keys are used for. ECB mode (**R05**) and predictable (**R06**, **R08**) or reused nonces (**R07**, **R09**)

considerably weaken the confidentiality properties of cipher operations, so are rated medium severity. Lastly, predictable CSPRNG seeds (**R10**) have a history of causing critical vulnerabilities in applications running on legacy Android systems [8]. Because recent Android versions are not affected, we assign medium severity.

3.3 Challenges

We identify the following key challenges to accomplishing the goals stated above.

3.3.1 Precision. Before a certain API misuse can be mitigated, it has to be detected in the first place. The precision of this detection is of utmost importance to the mitigation, since we do not want to modify proper use of crypto APIs by forcefully applying our mitigation, nor can we tolerate missing legitimate misuse and leaving it unmitigated. Most established methods for crypto API detection rely on static analysis of the control flow graph of an application. However, this method has proven to be imprecise in real-world scenarios [7, 13, 14], suffering from a high degree of false positives due to unreachable code and false negatives due to dynamically loaded code, complex data dependencies or obfuscation. To satisfy the requirements on precision, our solution employs dynamic runtime instrumentation. This approach enables us to guarantee that every crypto API call executed at runtime is detected and can be analyzed based on the concrete parameters passed from application code. In case of a detected misuse, parameters can be corrected at runtime, with no noticeable performance penalty. Additionally, relying on runtime instrumentation allows our solution to operate on-device in a self-contained manner, improving availability and privacy by not relying on any external server component.

3.3.2 Transparency to application code. All mitigations have to operate largely *transparently* to application code so that side effects on the functionality of affected programs can be avoided as much as possible. This is a particularly difficult task for cases where an additional parameter has to be transported between two separate API calls from code that has not provisioned for this possibility originally. CryptoShield solves this problem by taking advantage of a series of general observations about the JCA and its typical use in Android applications.

3.3.3 Practicality. We introduce our design as an emergency aid against the security vulnerabilities resulting from crypto API misuse. As such, we want it to be accessible and deployable to as large an audience as possible. Since maintaining a custom OS for the fragmented Android device landscape is not realistic and average Android users cannot be expected to be willing to flash a custom operating system, our concept needs to operate on stock firmware. Additionally, we consider rooting a device for deploying crypto API mitigation detrimental to the overall security. To surmount this challenge, our implementation employs a novel combination of performant package-level instrumentation and the Android DevicePolicyManager API.

4 MITIGATING CRYPTO API MISUSE

Conceptually, crypto API misuse can be described as a misparametrization of crypto APIs. For the crypto-savvy developer of a new application, avoiding crypto API misuse is usually as simple as choosing appropriate parameters for all employed crypto APIs. However, once such a misparametrization has found its way into a compiled application, its mitigation is a much more complex task. Retrofitted mitigations have to operate transparently to existing application code, so that the offending application's functionality remains intact. To this end, all our mitigations operate as transparent shims between application code and the Android framework, as illustrated in Figure 1. It is also worth noting that modern applications are usually not solitary pieces of software, but part of a much larger ecosystem consisting of web services and client applications that operate together. Our mitigations must fit into this bigger picture as seamlessly as possible, i.e. fix crypto API misuse in all parts of code executed by the client locally, while retaining web service interaction functionality.

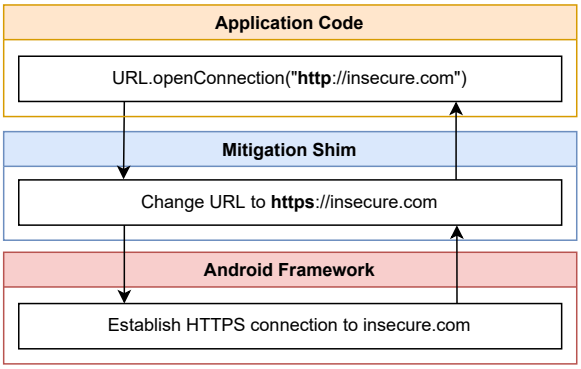


Figure 1: Our mitigations operate as shims between application code and the Android framework

4.1 General Observations and Strategies

Before we describe our mitigations for specific crypto API misuse cases, we highlight a series of general observations and strategies that guide the design of our mitigation procedures.

4.1.1 Correspondence of Operations. First, we highlight the basic observation that many cryptographic operations are carried out in ordered pairs, where the second operation either reverses, replicates or validates the first operation. This correspondence of two operations within the distributed ecosystem of a modern application yields three possible scenarios. It is worth pointing out that neither dynamic nor static analysis methods are capable of *reliably* foreseeing which of these scenarios will occur during further program execution.

In scenario Ⓘ, both of the corresponding operations are carried out locally in the application that is subject to the mitigations. In this case, we can trivially ensure that the mitigations applied during the first operation are correctly reversed or replicated in the second. An example would be changing a cipher mode of operation for both encryption and decryption of a local file. In scenario Ⓘ, the first of the corresponding operations is carried out in an external part of the ecosystem, while the second is carried out locally in the application that is subject to our mitigations. For covering this scenario, the second operation needs a way to know the origin

of the data that is to be processed, so that the mitigation is only applied if local data is processed. This can be accomplished e.g. by introducing an additional communication channel between local corresponding operations. Alternatively, for operations that have clearly defined failure cases, we can try applying our mitigations and fall back to an unmitigated operation if a failure is detected. In scenario (III), the first operation is carried out locally, while the second is not. This case is problematic, since the external processing entity will not have any knowledge about the mitigations applied in the first, local operation. Still, due to the inability to foresee the later destination of the output from the local operation, mitigations have to be carried out unconditionally in the first operation. In Subsection 6.2, we demonstrate that this scenario is reasonably rare in popular real-world applications for us to consider it an acceptable failure case of our solution in line with our goal of establishing an *emergency* aid.

4.1.2 Operation Context & Context Transfer. Even for purely locally corresponding operations, a communication channel between two corresponding operations is required. It has to supply the second operation with the context needed for reversing or replicating the mitigations applied in the first operation. For some mitigations, this context is specific to a particular execution, so it cannot be stored e.g. in a global variable. As an example, consider initialization vector reuse in a symmetric cipher encryption operation. As part of the mitigation, a fresh IV has to be generated to replace the insecure value supplied from application code. The same generated value later also has to be made available to the decryption operation of the resulting ciphertext. For our mitigations, we solve this problem by leveraging the output of the first operation as a communication channel to the corresponding second operation. In the IV reuse case described above, the IV generated by our mitigations is prepended to the resulting ciphertext and extracted at the corresponding decryption mitigation. Although this procedure changes the output size of the first operation, it still is transparent to most applications. This is because the Java Cryptography Architecture exposes functions for application code to query the output size of cryptographic operations. Due to the general opaqueness of cryptographic operations to the average application developer, the vast majority of Android applications uses these functions instead of hardcoding buffer sizes.

4.1.3 Introducing Fresh Entropy. Mitigations for misuse scenarios that involve an insecure key or password require the introduction of fresh entropy. To this end, our mitigations take advantage of the Android Keystore for generating and storing a fresh key we call key derivation key (KDK). Because the Android Keystore is limited to asymmetric cryptography on some devices, we use the KDK and the insecure key for deriving a fresh key that can be used for any cipher. This solution ensures that different installations of the same application always use unique keys.

4.1.4 Reuse Across Launches. Multiple covered vulnerabilities are concerned with reuse of nonces. For efficiently tracking used nonces, our mitigations maintain a simple nonce database.

4.1.5 Hardcoded Keys or Passwords. Our solution extracts hardcoded strings, character arrays and byte arrays from the target application package during instrumentation. This procedure can be implemented as a lightweight extraction algorithm that simply parses DEX data structures. It does not need any control flow information. For further reducing the runtime and memory overheads, the procedure is restricted to values that follow the allowed or typical format of passwords or keys. For DEX files dynamically loaded during execution, data extraction is performed at runtime.

4.1.6 Predictability. In addition to values that are hardcoded, predictability can also be caused by employing randomness sources other than those that are cryptographically secure (`SecureRandom`, `/dev/(u)random`) for cryptographic operations. For identifying this problem, our mitigations maintain a cache of the most recent output of the Java `Random` API, which is particularly convenient to use, but not cryptographically secure.

4.2 Specific mitigations

In the following, we describe the detailed procedures for mitigating misuse of specific cryptographic APIs. For all mitigations, we assume the ability to instrument (or intercept) Java methods of our choice.

4.2.1 SSL/TLS. For mitigating flaws in TLS use (**R02**), we focus on the `SSLSocket` interface. While only a minority of applications directly access this low-level API, it forms the backbone of the vast majority of HTTPS networking libraries, which means they can all be covered through this single point of interception. Our mitigation intercepts the creation of `SSLSocket` instances and returns a wrapper around the originally created object. When the application code signals the beginning of application-level data transfer through calls to the `getInputStream()` or `getOutputStream()` methods, our mitigation queries a configurable certificate trust policy for determining the trustworthiness of the connection. In the event of a negative assessment, we immediately abort the connection, without ever having sent any application-level data over the compromised channel.

By performing the certificate check immediately before the target program starts sending application data, we know that the former considers the connection secure. Combined with the 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.

The default certificate trust policy of our prototype implementation is based on the Trust-On-First-Use (TOFU) principle. When the first connection to a new host is made, its TLS certificate is cached by our policy code. Subsequent connections are then only allowed if the certificate presented by the TLS host is identical to the one encountered previously. Our prototype additionally supports a more complex certificate policy that builds on a notary web service. This web service takes the position of a certificate oracle, serving the legitimate TLS certificate for any host it is queried for. The certificate trust policy contacts the notary web service for any new host or certificate mismatch to determine whether its own connection to the target host is subject to an ongoing MITM attack. It is worth pointing out that both currently implemented certificate trust policies are only prototypes that illustrate the flexible nature of supported policies. Development of a more sophisticated policy

that fully considers privacy and reliability aspects is deemed out of scope for this work.

4.2.2 Ciphers. We intercept the creation of `Cipher` instances to return a wrapper object backed by a custom security provider. Once the wrapper has received all required parameters from application code, it enforces the rules established in Subsection 3.2. Implicit or explicit uses of ECB mode for symmetric block ciphers (**R05**) are automatically upgraded to CBC mode. Whenever a predictable (**R06**) or reused IV (**R07**) is identified, a fresh value is automatically generated and passed to the wrapped primitive instead. All parameters changed as part of our mitigations are encoded in a ciphertext prefix so they are later available in the corresponding decryption operation. Figure 4 in Appendix A displays a simplified mitigation flow for implicit ECB mode using a ciphertext prefix. Our mitigation also overrides the `Cipher.getOutputSize()` method so that applications that pre-allocate their buffers can correctly accommodate the prefix.

4.2.3 Password-Based Encryption. Mitigations for PBE are integrated into the custom security provider described above. Whenever a PBE key is passed to the Cipher API, its parametrization is checked for rule violations. For PBE keys that are passed to the Cipher API in serialized form, our mitigation code keeps a map between recently serialized PBE keys and the corresponding parameters that were used in their creation. If a predictable password (**R04**) is detected, a replacement password is derived from a key generated in the Android Keystore. Salt reuses (**R09**) detected through the nonce database or predictable salt values (**R08**) are mitigated by generating a fresh replacement value. All modified PBE parameters are encoded in the prefix of any ciphertext generated using the affected PBE key. While replacement salt values are completely embedded in the prefix, mitigations on other parameters are only encoded as flags interpreted in the corresponding decryption operation.

4.2.4 Random Number Generation. Our mitigation code installs a custom CSPRNG provider when an instrumented application launches. In the event of a predictable seed value (**R10**) passed from application code, we replace the explicit seeding with randomness obtained from the default system-provided entropy source.

4.2.5 HTTP. Most applications establish HTTP connections using either the `HttpUrlConnection` API or the open-source OkHttp library. Our mitigation intercepts calls to both of these implementations to check if a target server is contacted via the plain HTTP protocol (**R01**). If this is the case, the connection is automatically upgraded to HTTPS, i.e. an attempt is made to perform a TLS handshake with port 443 of the target host. If this handshake fails, our mitigation code falls back to the original unprotected HTTP url. We maintain a cache of hosts that were reachable over HTTPS in the past to thwart downgrade attacks that mask the availability of HTTPS on a server. A more sophisticated implementation may employ a pre-compiled list such as the one offered by DuckDuckGo Smarter Encryption [3].

4.2.6 KeyStore. When a predictable password is used for key storage (**R03**), a replacement password is derived by adding entropy from a key generated in the Android Keystore. For all load operations on key stores, our mitigation code tries the replacement key (derived in the same way as for store creation) and falls back to the key supplied from application code upon failure.

[3]DuckDuckGo Smarter Encryption: https://help.duckduckgo.com/duckduckgo-help-pages/privacy/smarter-encryption/

5 PROTOTYPE IMPLEMENTATION

The CryptoShield prototype demonstrates how our mitigations can augment the existing security architecture and user experience of the unmodified Android platform. The implementation consists of two core modules, described in the following and illustrated in Figure 2. In total, our prototype implementation consists of over 25.000 custom lines of code.

The first part of our prototype is comprised of an instrumentation module that implements the individual mitigations described in Subsection 4.2. Once injected into a target application, the instrumentation module sits as a mitigation shim between application code and framework calls. Parameters passed from application code are inspected and collected for monitoring purposes. If any of the rules established in Subsubsection 3.2.1 is violated, parametrizations are upgraded transparently to application code following the procedures put forth in Subsection 4.2.

The second part of our prototype is an Android application we call CryptoShield Agent (CSA). The CSA offers a convenient solution for automatically injecting our instrumentation module into all applications installed on an unrooted Android device. It consists of a daemon service that starts immediately after booting the OS and a user interface that allows managing instrumented applications, as well as inspecting detailed crypto API usage reports. By listening for installation events broadcast by the system, the background service can automatically generate an instrumented version of every application the user installs or updates on the device.

In the following, we highlight key aspects of our prototype implementation.

5.1 Instrumenting Applications on Unrooted Android Devices

Instrumenting running application processes on Android requires root privileges, which counteracts the goals established in Subsection 3.1. Instead of directly manipulating *processes*, our prototype instruments application *packages*. It takes advantage of the observation that the APK files of any installed application remain stored on the device and can be accessed by other processes. This opens the possibility for the CSA to generate modified copies of arbitrary installed application packages. Our instrumentation routine unpacks the APK files to add a DEX file and a native library. The modified APK files are then compressed and signed using a freshly generated signing certificate. A cache ensures that applications originally signed with the same signing certificate will retain this property across instrumentation. Once the instrumented APK files are installed and the app is launched on a device, the injected native library manipulates data structures in the ART runtime to reroute crypto API calls to the injected DEX file. Our mitigation code in the injected DEX file can analyze and modify all passed parameters before it invokes the original implementation of the intercepted method. At a high level, this instrumentation approach

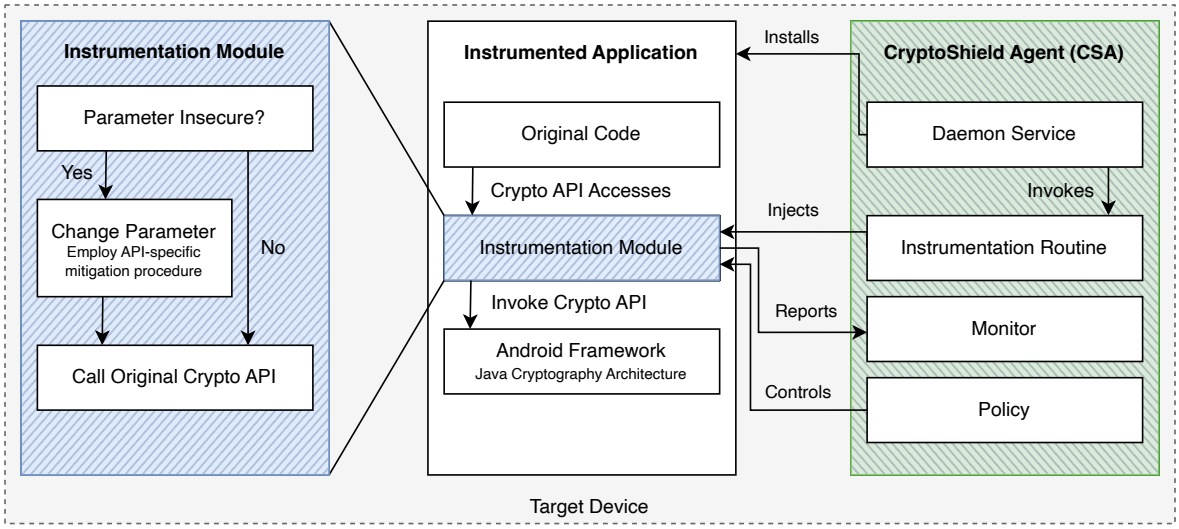


Figure 2: The two main components of our prototype implementation: *Instrumentation Module* and *CryptoShield Agent*

was first presented by Styp-Rekowsky et al. [20]. Our prototype uses a custom implementation on top of low-level primitives from the open-source SandHook library [4] that lazily instruments methods on demand as their defining classes are loaded into runtime. It supports modern APK features such as Multi DEX or Split APKs, and all existing APK signature schemes.

5.2 Updating Instrumented Applications through Google Play

Modifying an APK package and resigning it with a different certificate usually breaks the common update route through Google Play. This is because all updates will still be signed with the original signing certificate and thus cannot be installed on top of the instrumented variant. As a remedy to this problem, our prototype changes the package name of every instrumented application, so that it can be installed alongside the original version. The original version is then disabled, so that it remains installed on the device but can no longer be launched. Whenever an update to an application is available, the user can install it from Google Play. The CSA will then automatically pick up the new version of the original application and produce a corresponding update for the instrumented variant.

5.3 Changing Package Names of Instrumented Applications

Manipulating the package name of a compiled Android application requires additional changes to the application package. CryptoShield intercepts all framework methods where the package name is passed between application code and the outside world (e.g. in Intents or HTTP headers, even when accessed through Reflection APIs). Beyond the package name itself, all authorities of `ContentProviders`, custom permissions and account type identifiers are changed as well, so that they do not interfere with the respective components of the original version. Although our implementation goes to great lengths to cover as many scenarios as possible, a few issues remain in cases where it is unclear whether the original or modified package name yields correct behavior. For example, the original package name is sometimes used for fetching data from a proprietary configuration table in app resources. However, as detailed in Subsection 6.2, only a small number of real-world applications are affected by these issues. It is also worth pointing out that any issues caused by this tradeoff between practicality and application compatibility only affect our prototype implementation and are unrelated to the crypto API misuse mitigations themselves.

5.4 Installing and Disabling Applications without User Interaction

To install instrumented applications without user interaction, the CSA takes advantage of Android's DevicePolicyManager API. By implementing a `DeviceAdminReceiver` that is configured as the device owner, the CSA is granted access to the PackageInstaller API. Via 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. Additionally, the DevicePolicyManager API permits the disabling of installed applications. It is crucial to note that the device owner role grants CryptoShield precisely the privileges it needs for autonomous operation, while allowing it to seamlessly *integrate* into the security architecture of the Android OS. This represents a key advantage over solutions that require root access on the device, which effectively *circumvents* the system's security foundations, with potentially disastrous consequences for inexperienced users. In order to streamline the process of installing and configuring the CSA application as a device owner, our prototype includes an easy-to-use companion tool for desktop computers. Once the Android device is connected to the computer via a USB cable, our software asks the user for explicit consent to the start of the setup procedure. A single button click then automatically installs and configures the CryptoShield system.

[4] SandHook Android ART Hook: https://github.com/asLody/SandHook

5.5 Support for MDM Deployment

As the CryptoShield Agent is built upon Android's DevicePolicyManager APIs, it can easily be integrated into corporate Mobile Device Management (MDM) solutions. CryptoShield thus represents a novel device management policy that can be rolled out to all devices in a company's device fleet for offering full application choice to users without significantly compromising data security. We envision a system that can be fully configured by the MDM administrator. Configuration options could include the enforced crypto rules and additional rule-specific adjustments. For example, it would be possible to deploy custom certificate validation policies for TLS or configure a custom HTTPS upgrade host list.

5.6 Crypto API Call Monitoring

In addition to mitigating crypto API misuse, our prototype also implements functionality for monitoring crypto API invocations from third-party applications. This functionality covers the same crypto APIs as our mitigations, i.e. TLS, Cipher, PBE, CSPRNG, HTTP and KeyStore APIs. We elect to restrict monitoring to this set of APIs for consistency with our mitigations (which are the main focus of CryptoShield), but further expansion is feasible. Monitoring is designed to provide users with real-time feedback on the vulnerabilities that CryptoShield protects them from. The instrumentation module inside instrumented applications collects relevant calls and the respective parametrizations. It then communicates all findings to the CSA. Through a monitor UI, the user can inspect all reported crypto API use and misuse.

5.7 Fallback Mechanism

As described in 4.1, although our mitigations are designed to retain functionality as much as possible, we anticipate that they might lead to malfunction in a small number of instrumented apps. To accommodate for this possibility, our prototype implements a fallback mechanism: Upon noticing malfunction, a user may manually toggle CryptoShield's mitigations for a specific application. We have deliberately made disabling mitigations an explicit user choice, because it means that users will be susceptible to attacks exploiting crypto misuse in affected applications. Still, CryptoShield's monitoring functionality allows users to track crypto API invocations while mitigations are disabled, and issues alerts of concrete misuses that have been uncovered.

6 EVALUATION

The evaluation in this section is comprised of five parts. In an automated large-scale analysis, we first demonstrate our prototype's efficacy on 1604 applications. Secondly, through a manual examination on the 99 most popular free applications from Google Play, we validate that applications are still functional and usable after installing CryptoShield's mitigations. Moreover, we provide per-API call runtime overhead measurements and complement the analyses on real-world applications with results from a synthetic benchmark that enables comparison to static detection-only tools. Lastly, we present a detailed case study on how our mitigations fix critical security vulnerabilities in two widely-used applications.

6.1 Automated analysis

To obtain a quantitative measure for the effectiveness of our mitigations, we ran an automated security analysis on 1604 applications before and after injecting our instrumentation module. We based our automated analysis testbed on the CryLogger crypto API misuse detector by Piccolboni et al. [13], which we adapted to our crypto rules and augmented with concepts from Sounthiraraj et al. [18] for improved detection of TLS issues. CryLogger runs applications in an Android emulator driven by random user input (here: 10k input events ≈ 23 % line, 33 % class coverage). Its custom emulator image logs calls to various crypto APIs. In our testbed, network connections were additionally subjected to a MITM attack by rerouting the emulator's network traffic through a MITM proxy server that logs all successful TLS connections [5]. All collected logs were then scanned for violations of the rules stated in Subsection 3.2. If a particular misuse was logged while running an original application, but not in the instrumented version, we consider this a successful mitigation. Building our automated evaluation on the CryLogger tool by Piccolboni et al. enables comparison to an independent baseline for the vulnerable applications in our sample set and their respective violations of our crypto API rules.

For executing this analysis, we used a bare-metal-virtualized Ubuntu 20.04 LTS system with 128 CPU cores and 128 GB of RAM, backed by a Dual AMD EPYC 7502 CPU with 1 TB of RAM. This setup allowed us to run our analyses on 6 virtual devices in parallel. Since the SandHook library utilized in our prototype implementation does not support the x86 instruction set, we ran the automated analysis on a slightly modified version that intercepts method calls by rerouting invocations inside the DEX code ahead of execution. The mitigation procedures inside the instrumentation module were not changed, so that the effectiveness measured here directly translates to that of the implementation described in Section 5. The set of 1604 tested applications consists of the free top-100 for 33 categories on Google Play as of January 2022, cleared from entries incompatible with either the x86 instruction set or the Android emulator. All 1604 applications in our set use some cryptographic API, while 388 applications violate some of our crypto rules.

In total, CryptoShield was able to mitigate all vulnerabilities induced by rule violations (as identified by our testbed) in 348 applications, which corresponds to 89.7 % of the vulnerable subset. Some vulnerabilities were mitigated in 372 applications (95.9 % of the vulnerable subset). The most commonly mitigated vulnerabilities concerned violations of **R02** (164 apps), **R01** (75 apps) and **R06** (66 apps). Violations against multiple rules were detected and mitigated in 94 apps. Although CryptoShield was able to produce installable packages for all tested samples, 105 of all instrumented applications (6.5 %) crashed at some point during execution.

40 applications were still vulnerable after instrumentation due to plain HTTP communication or insecure TLS connections from native code. As discussed in Subsection 7.5, our prototype does not cover native code, although obfuscated code or invocations of Java code through Reflection APIs or JNI are supported.

Detailed statistics for individual application categories can be obtained from Figure 3.

[5] CryptoShield's notary web service certificate trust policy (see Section 4.2.1) was used in the automated analysis

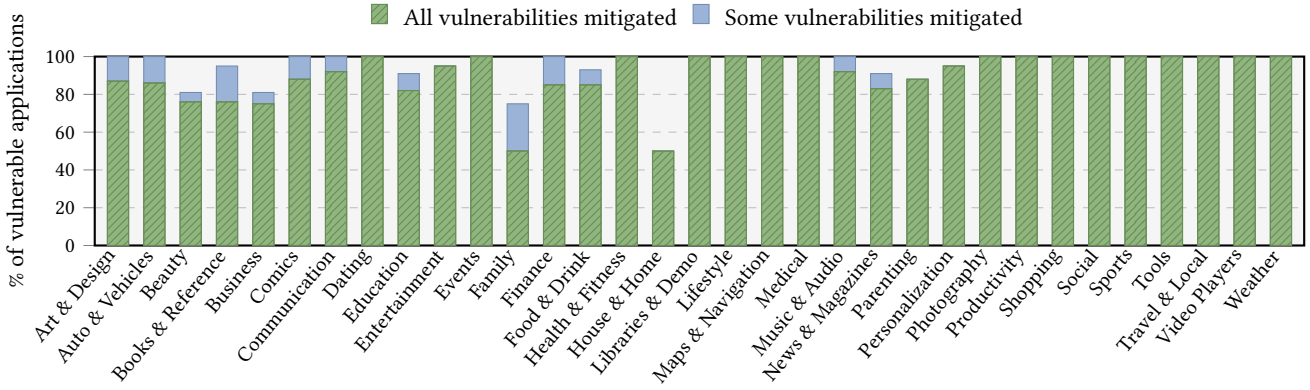


Figure 3: Automated analysis: Portions of vulnerable apps per category that could be partly or fully fixed

6.2 Manual analysis

Automated analysis is incapable of ensuring that applications retain their original functionality across our instrumentations and mitigations. Therefore, we manually collected additional compatibility and performance statistics for a smaller set of 99 applications on a physical Google Pixel 3 device running Android 11 (Build RQ3A.210605.005). As part of this analysis, every application was executed for two runs of 3 minutes, once in the original version and once in the instrumented version. In each run, a human operator navigated the application UI, simulating a typical end user exploring the software's functionality. The test set of 99 applications was obtained from Google Play in April 2022. It consists of the three most popular free packages for 33 different categories. For applications that required a paid membership or particular hardware peripheral, we picked a replacement application that immediately followed in the ranking. If a free user account was required, registration was completed before the timer was started.

6.2.1 Deployment Speed. Our prototype completed instrumentation of an application package in 10.7 seconds on average. Over all data collected, we can observe a strong correlation between the original APK size and the instrumentation duration ($\rho = 0.70$). This observation aligns well with the fact that the most computationally demanding task for our instrumentation routine is the compression and signing of the APK file. The distribution of instrumentation time across our test set is documented in Appendix B.

6.2.2 Package Size. The relative package size change caused by CryptoShield's instrumentation averaged at 5 %. It is worth pointing out that the size of the instrumentation module injected into all APK files was identical. The variance in instrumented APK size overhead results from our instrumentation routine's ZIP compression implementation that differs from the one found in official Android build tools. For several applications, the overall APK size even decreased during instrumentation. The distribution of absolute APK size changes across the test set is detailed in Appendix B.

6.2.3 Compatibility. From manually exploring the user interfaces of instrumented applications, we can report that 91.9 % of applications in our test set retained their original functionality. Although the instrumentation routine produced valid APK files for all tested applications, some showed signs of malfunction during execution. Through additional reverse engineering, we were able to trace back most of these cases to issues with package name modifications (3 applications), or signature checks that were intended to prevent malicious manipulation of the APK (3 applications). Two instrumented programs were no longer functional due to incompatibilities with CryptoShield's mitigation procedures (cipher mitigations broke server communication).

Given the non-exhaustive nature of manual UI exploration, the compatibility rate discussed here represents an upper bound. Still, we believe that manual exploration is the most accurate method for obtaining a representative compatibility measure in the absence of validation models that could be used for automated exploration.

6.2.4 Launch Overhead. CryptoShield's instrumentation module and mitigations are initialised during application launch. As the single most significant addition of instructions, this procedure has the potential to harm the user experience of instrumented applications. For our manual analysis, we therefore timed application launch time before and after instrumentation. We found the average launch time overhead of instrumented applications to be 571 ms, which is hardly noticeable to end-users. Appendix B documents the distribution of absolute launch time overhead across the test set.

6.3 Per-API Runtime Overhead

Intercepting crypto API methods, inspecting arguments and applying mitigations inevitably has some impact on their runtime performance. To obtain a measure for the concrete overhead for each affected crypto API method, we timed their execution (1) in an uninstrumented app, (2) while only monitoring is enabled and (3) while only mitigations are enabled. For each API, measurements were collected for the most expensive use case and mitigation. For each measurement, we calculated the average from at least 1000 samples.

Our measurements indicate that the worst-case runtime overhead of our mitigations is about 101 ms (Cipher API calls where a key has to be derived via the hardware-backed AndroidKeyStore). Measurement details can be found in Appendix C.

Please note that an average per-app runtime overhead cannot sensibly be provided for CryptoShield. Any such value calculated

Category	Cases TP	Cases TN	C.Shield FP	C.Shield FN	C.Guard FP	C.Guard FN
InsecureTrustManager (**R02**)	8	0	0	0	0	8
InsecureHostnameVerifier (**R02**)	4	1	0	0	1	1
EcbCrypto (**R05**)	7	2	0	0	1	1
Http (**R01**)	7	3	0	0	0	1
NoHostnameVerifier (**R02**)	2	0	0	0	0	1
PredictableIV (**R06**)	10	2	0	0	1	1
PredictableKeyStorePW (**R03**)	9	3	0	0	1	1
PredictablePBEPW (**R04**)	11	2	0	0	1	2
PredictableSeed (**R10**)	16	3	0	7	2	6
ReusedIV (**R07**)	9	2	0	0	0	9
ReusedSalt (**R09**)	8	2	0	0	0	8
PredictableSalt (**R08**)	11	3	0	6	1	1
Total	102	23	0	13	8	40

Table 1: Synthetic Benchmark test cases and results per misuse category. True Positive and True Negative indicate test case ground truth. False Positive and False Negative indicate errors in CryptoShield's and CryptoGuard's detections. Corresponding crypto rules in braces.

over the entire app lifecycle is entirely dependant on user behavior (how often intercepted methods are triggered).

6.4 Synthetic Benchmark and Comparison to Static Detectors

Although our manual and automated analyses use large datasets of popular real-world applications, they do not cover all our addressed crypto misuses. To provide a more complete picture of CryptoShield's mitigation robustness and to facilitate comparing its detection precision with state-of-the-art detection-only tools, we ran CryptoShield against a comprehensive synthetic benchmark. To the best of our knowledge, all existing crypto API benchmarks are designed for traditional *static* misuse *detectors*. In their original form, they are thus unsuitable for evaluating CryptoShield's *dynamic mitigation* capabilities. We therefore selected the CryptoAPI-Bench [1] benchmark as the best match for our misuse rules, and adapted it to our setting. Adaptations ensure executability of all test cases, and introduce checks for retained functionality. For example, one of these checks asserts that encryption followed by decryption yields the original plaintext even if CryptoShield replaces the key to mitigate its predictability. Furthermore, we added 29 test cases that closely follow the structure of existing cases, but fill CryptoAPI-Bench's gaps in coverage of misuse rules **R07** and **R09**. Lastly, we introduced new test cases for covering Crypto API invocations through Java Reflection APIs (13) and complex TLS certificate validation issues (3). Test cases not related to CryptoShield's rules were removed. In total, our benchmark comprises 125 test cases, of which 102 contain misuse and 23 were designed to catch false positives. For quantitatively evaluating CryptoShield's detection capabilities against a state-of-the-art static detection-only tool, we also analysed the same benchmark with CryptoGuard [15].

Benchmark details and results for both CryptoShield and CryptoGuard can be found in Table 1. CryptoShield's detection correctly identified 89 of the misuse cases (false negative rate: 13 %) and did not produce any false positives. All missed misuses were caused by hardcoded CSPRNG seeds (7) or PBE salts (6). Most issues can be traced back to a limitation in our prototype, which for efficiency reasons only considers hardcoded byte arrays of at least 4 bytes. This may be addressed in future work. For all test cases where CryptoShield detected a misuse, the consistency checks indicated that mitigation procedures retained existing functionality.

While CryptoGuard performed slightly better than CryptoShield on hardcoded values, it had higher false positive (9 cases ≈ 36 %) and false negative (41 cases ≈ 37 %) rates overall. The static detector was unable to identify insecure TLS certificate validation or nonce reuse at all. It is worth noting that CryptoAPI-Bench was designed for evaluating static misuse detectors, so some of its test case variations specifically challenge static detectors, but look identical to CryptoShield. A benchmark for objectively comparing static and dynamic crypto API misuse detectors has not been proposed yet. Dynamic *detectors* usually suffer from poor code coverage. However, for dynamic *mitigation*, code coverage is irrelevant, as long as we can ensure that all *executed* misuse is *mitigated* (see Section 6.1). CryptoShield is designed for preventing exploitation of crypto API misuse, not for generating an extensive list of misuse hidden in code bases.

Additional benchmark implementation details can be found in Appendix D.

6.5 Case Studies

To demonstrate how CryptoShield prevents leakage of sensitive user data in real-world scenarios, we conduct case studies on two popular applications that contain critical vulnerabilities caused by crypto API misuse. Case study samples were chosen based on *1)* number of installations, hence real-world representativeness, *2)* illustrative quality, i.e. suitability for demonstrating causes and consequences of crypto API misuse, and *3)* sensitivity of processed data. Displayed vulnerabilities were responsibly disclosed to the respective vendors.

6.5.1 Banggood. Banggood is a Chinese online retailer whose Android app has been downloaded more than 10 million times from Google Play as of August 2022. The application uses insecure `TrustManager` and `HostnameVerifier` implementations, allowing a MITM attacker to intercept the data exchanged between the Banggood client program and server. Although many of the application's requests are protected by a proprietary RSA scheme on top of TLS, the customer registration endpoint is not, meaning that complete user accounts can be compromised.

When installing the application on a system protected by CryptoShield, the data disclosure is successfully prevented. From the CSA monitor interface, we can confirm that the network request for registration fails during an ongoing MITM attack because our system aborts the connection after it detects the MITM attack.

6.5.2 Amaze File Manager. Amaze File Manager is an open-source file management application for the Android platform. Together with its derivates, the program has reached an audience of more than 10 million users according to Google Play. Beyond the basic feature set common in this software category, Amaze also supports encryption and decryption of sensitive files.

CryptoShield detects that the same IV is used for AES-GCM encryptions of multiple different files. Experiments with an unmodified version of the file manager confirm that from two ciphertexts C_1 and C_2 encrypted using the same key and one of the plaintexts P_1, we can calculate the other plaintext P_2 as $P_2 = C_1 \oplus C_2 \oplus P_1$. Since encrypted files are saved in public storage, this means that any other application with appropriate read access may decrypt files just by guessing one original plain text.

The flaw can also be confirmed from the source code of the Amaze File Manager application, where the corresponding class holds a comment about this exact problem. The example underlines that developers are often overwhelmed by the security requirements of modern applications, lacking the experience for ensuring proper data protection

When the same experiment is conducted after applying CryptoShield to the vulnerable app, fresh IVs are injected for every encryption, so that no information about P_2 is leaked.

7 DISCUSSION & FUTURE WORK

This section elaborates on the issues raised in Section 5 and highlights additional limitations. It also explores how our approach can be extended to related research fields.

7.1 Robustness

Although our mitigations were designed to keep side-effects at a minimum, some of them have the potential to lead to issues in an app's existing functionality. Example scenarios are applications that use insecure ECB mode in server communication, or that pass prefixed ciphertexts to fixed-size buffers in native code. However, evaluation on real-world apps (Section 6.2) shows that these issues combined only affect a very small number of applications (2 %), so that CryptoShield's security benefits far outweight its potential for introducting side-effects. For affected applications, users can take advantage of CryptoShield's fallback mechanism (Section 5.7). It is worth noting that our instrumentation may be incompatible with advanced app packers that hook Android Runtime APIs. However, to the best of our knowledge, these techniques are only used in malicious apps that seek to evade analysis. As discussed in Section 3.2, malicious applications are out of scope for CryptoShield.

7.2 Implementation Security

By utilizing the DevicePolicyManager APIs, CryptoShield assumes a security-sensitive role on the Android device. If an attacker was to find and exploit a vulnerability in CryptoShield, they could carry out operations on the device that might harm the user. For our prototype, we took great care to follow security best practises for the Android platform, and we publish our source code for examination by fellow security researchers[6]. Before deploying our solution in a real-world scenario, an independent code audit is advisable for ascertaining the implementation's security and trustworthiness.

7.3 Signature Checks

For instrumenting third-party applications on unrooted devices, our prototype implementation modifies APK files, re-signing them with a new certificate in the process. Like any other solution that takes advantage of this approach, it is affected by signature checks that some developers integrate into their programs as protection against maliciously modified redistributions. While it would be possible to work around many implementations of signature checks using our instrumentation technology, we respect the intention behind these barriers and refrain from pursuing ideas for circumventions. Only 3 % of applications in our manual evaluation test set were affected by this problem. We argue that developers who are versed enough to implement package signature checks are more likely to be aware of and follow best practices for cryptography in general. It is worth pointing out that this is a limitation of the prototype implementation and not our mitigation procedures. Another implementation may choose a different trade-off between usability and compatibility or take advantage of an entirely new instrumentation technique.

7.4 Bookkeeping Costs

CryptoShield maintains runtime databases for detecting and mitigating nonce reuses. Inevitably, this bookkeeping incurs some runtime overhead, e.g. when compared to the one-time cost of a static approach that analyses the control flow graph of an application. However, we argue that the bookkeeping runtime cost is not critical here - it is not noticable to the user in our prototype (see Section 6.3). Much more importantly, CryptoShield's runtime instrumentation can take advantage of data that static analysis never has access to. It enables mitigation of misuse cases that are not noticable to static analysis, such as reuse of dynamically generated nonces or non-trivial issues in TLS certificate validation.

7.5 API Coverage

Our prototype implementation only detects and mitigates misuse of cryptographic APIs provided by the system frameworks or by known third-party libraries. Applications that integrate from-scratch implementations of primitives or native third-party crypto libraries are not covered by our protection measures. Still, it is worth noting that our mitigation procedures can in principle be extended to any implementation of the covered APIs. Obfuscated code and invocations of Java code through Java Reflection APIs or the Java Native Interface are already supported in our prototype implementation.

7.6 Device Coverage

We implemented our prototype for the popular Google Pixel series of devices and the three most popular iterations of the Android OS (versions 9-11)[7] as of August 2022. The system was extensively tested on a Google Pixel 3 running Android 11. Given the slight variations of ART behavior between different devices and Android versions, we anticipate minor changes before support for other devices can be guaranteed. We also have to point out that our current implementation only works on the ARM instruction set, which covers the vast majority of available Android devices. Extending support to additional CPU platforms is feasible and only a matter of resources.

[6] Source code is available at https://extgit.iaik.tugraz.at/fdraschbacher/cryptoshield

[7] As indicated by Google's official Android version distribution chart inside the Android Studio IDE

8 RELATED WORK

In this section, we highlight previous publications in the domain of crypto API misuse on the Android platform.

8.1 Mitigating Crypto API Misuse

The only previous works capable of mitigating crypto API misuse in compiled Android applications are *Pin It!* by Buhov et al. [3] and *CDRep* by Ma et al. [10]. The first requires root permissions and only addresses issues with TLS misuse. CDRep follows similar high-level principles as our solution, but differs significantly in its approach and practicality. In contrast to CDRep, CryptoShield (1) can be directly deployed to protect a user's device in a self-contained manner, (2) addresses many more misuse cases, among them particularly wide-spread and critical TLS issues, (3) emphasizes retaining functionality, and (4) does not suffer from the resource-intensiveness and imprecision of static CFG analysis (CDRep only mitigates the first misuse per app method and takes 19 seconds per misuse for patch generation). Unfortunately, Ma et al. were unable to share their source code or data set with us, so we cannot provide quantitative comparisons.

Beyond these immediately related works, the closest publication to CryptoShield in terms of the addressed security vulnerabilities is FireBugs by Singleton et al. [17], which employs static analysis and pattern-based patching for retrofitting security best practices into applications that contain crypto API misuse. However, their solution only produces source-level patches that can serve as a guideline for application maintainers. It is not capable of protecting end users. Newbury et al. [11] combine static analysis and the desktop JVM's instrumentation API for hotpatching crypto API misuse in enterprise Java applications. However, their trivial mitigation strategies completely neglect the potential for side effects. Conceptually, our solution bears similarities to the work by Bates et al. [2], which proposes a solution for fixing SSL certificate validation on Ubuntu by dynamically linking a shim between third-party applications and SSL libraries. Some Android-specific publications suggest source-level mitigations for crypto API misuse, e.g. by facilitating the configuration of certificate pinning through XML files [6, 19].

8.2 Detecting Crypto API Misuse

Most previous works on crypto API misuse focus on detection in compiled application packages. The most recent major publications in this field of research have been provided in Section 1. The current state-of-the-art purpose-built tools ready for practice are generally considered to be CryptoGuard by Rahaman et al. [15] and CogniCrypt/CrySL by Krueger et al. [9].

9 CONCLUSION

Official countermeasures and previous efforts by the research community were ineffective in eliminating the common security vulnerabilities in popular Android applications that are induced by misuse of cryptographic APIs. In this paper, we devised a set of generic mitigation procedures for the most severe crypto API misuse scenarios. Based on prototype implementations of our mitigation procedures, we provide a tool that can protect users against crypto API misuse vulnerabilities until a long-term solution has been found. Our system's on-device daemon service is capable of automatically injecting an instrumentation module into all application packages installed on a device. This module subsequently monitors crypto API calls inside a target application and mitigates identified misuse transparently to application code. We showed that CryptoShield effectively mitigates vulnerabilities in real-world applications, is compatible with the vast majority of Android applications, and introduces minimal overhead.

REFERENCES

[1] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *IEEE Cybersecurity Development (SecDev)*.

[2] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Kevin R. B. Butler, and Abdulrahman Alkhelaifi. 2014. Securing SSL Certificate Verification through Dynamic Linking. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[3] Damjan Buhov, Markus Huber, Georg Merzdovnik, and Edgar R. Weippl. 2016. Pin it! Improving Android network security at runtime. In *IFIP Networking Conference, Networking and Workshops*.

[4] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[5] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. 2012. Why eve and mallory love android: an analysis of android SSL (in)security. In *ACM Conference on Computer and Communications Security (CCS)*.

[6] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL development in an appified world. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[7] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *35th International Conference on Software Engineering (ICSE)*.

[8] Alex Klyubin. 2013. Some SecureRandom Thoughts. https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html

[9] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: supporting developers in using cryptography. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[10] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*.

[11] Kristen Newbury, Karim Ali, and Andrew Craik. 2021. Hotfixing misuses of crypto APIs in Java programs. In *31st Annual International Conference on Computer Science and Software Engineering (CASCON)*.

[12] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. 2021. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In *30th USENIX Security Symposium*.

[13] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, and Simha Sethumadhavan. 2021. CRYLOGGER: Detecting Crypto Misuses Dynamically. In *42nd IEEE Symposium on Security and Privacy (SP)*.

[14] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *21st Annual Network and Distributed System Security Symposium (NDSS)*.

[15] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[16] Jingjing Ren, Martina Lindorfer, Daniel J. Dubois, Ashwin Rao, David R. Choffnes, and Narseo Vallina-Rodriguez. 2018. Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions. In *25th Annual Network and Distributed System Security Symposium (NDSS)*.

[17] Larry Singleton, Rui Zhao, Harvey P. Siy, and Myoungkyu Song. 2021. FireBugs: Finding and Repairing Cryptography API Misuses in Mobile Applications. In *IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*.

[18] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *21st Annual Network and Distributed System Security Symposium (NDSS)*.

[19] Vasant Tendulkar and William Enck. 2014. An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities. *CoRR* abs/1410.7745 (2014).

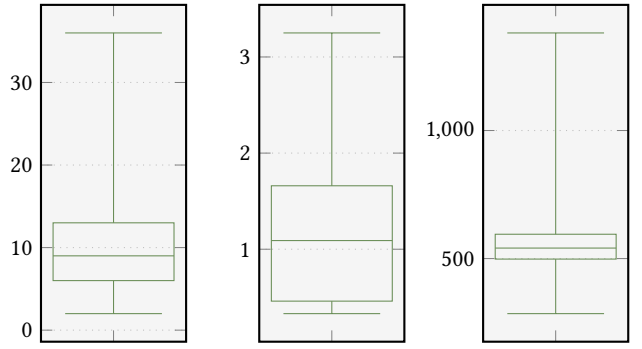
[20] Philipp Von Styp-Rekowsky, Sebastian Gerling, Michael Backes, and Christian Hammer. 2013. Idea: Callee-Site Rewriting of Sealed System Libraries. In *Engineering Secure Software and Systems - 5th International Symposium (ESSoS)*.

A MITIGATION PROCEDURES

Figure 4 illustrates the simplified mitigation procedure for implicit ECB mode in the Cipher API. ① Cipher creation is intercepted to return a wrapper object. ② In encryption mode, ECB mode is upgraded to CBC upon initialization and a fresh salt is generated. ③ The salt and an upgrade flag are prepended to the actual ciphertext ④ The prepended ciphertext is returned to application code, which later passes it to decryption. ⑤ Cipher creation is intercepted to return a wrapper object. ⑥ In decryption mode, the wrapped cipher is initialized with parameters from application code. ⑦ If a ciphertext prefix is found, the wrapped cipher is re-initialized with the parameters extracted from the prefix. Extracting the prefix from data streams fed through the cipher adds significant complexity to the complete implementation.

B MANUAL ANALYSIS

Figure 5 shows the distribution of instrumentation time, APK size overhead and launch time overhead across our manual analysis test set.



(a) Instrumentation Time (Seconds) **(b) Size Overhead (MB)** **(c) Launch Time Overhead (Milliseconds)**

Figure 5: Manual analysis: Distribution of performance characteristics across the test set

C PER-API RUNTIME OVERHEAD

This section provides details on our per-API runtime overhead measurements. For each API, measurements were collected for the most expensive use case. For example, `Cipher.update()` mitigation measurements are for the first decryption buffer in ECB mode and use a PBE key with hardcoded password and salt, so that it involves CryptoShield extracting the ciphertext prefix and deriving a fixed PBE key. For each measurement, we calculated the average from at least 1000 samples. Table 2 contains the exact per-API runtime overheads of CryptoShield's monitoring and mitigation. It is worth noting that monitoring was disabled while mitigation was measured and vice versa. The most significant overheads were observed for mitigations that involve deriving a key via the hardware-backed AndroidKeyStore. We attribute the speedup of `SecretKeyFactory.generateSecret()` in the instrumented app to measurement variations. CryptoShield's implementation of this API is only a wrapper of the original method. PBE key issues are mitigated once they are used in cipher operations.

API	Orig.	ℹ	✔
Cipher.getInstance()	0.01	0.02	0.02
Cipher.init()	0.01	1.54	101.00
Cipher.update()	0.03	1.20	95.12
Cipher.final()	0.00	1.19	95.42
Cipher.updateAAD()	0.00	0.00	0.00
KeyStore.store()	140.94	5.20	87.07
KeyStore.load()	140.60	3.85	86.99
SecretKeyFactory.getInstance()	0.00	0.01	0.01
SecretKeyFactory.generateSecret()	13.66	-0.08	-0.16
SecureRandom.getInstance()	0.00	0.02	0.02
SecureRandom.setSeed()	0.00	0.57	0.02
SSLSocketFactory.createSocket()	42.90	3.49	1.96
SSLSocket.getInputStream()	9.34	8.03	7.00
SSLSocket.getOutputStream()	10.09	9.04	5.60
URL.openConnection()	69.85	34.19	59.08
okhttp3.Request.Builder.url()	0.03	0.63	0.07

Table 2: Per-API runtime overheads in milliseconds. Orig. displays the API runtime in an unmodified app, while ℹ and ✔ contain overheads (relative to Orig.) for monitoring and mitigations, respectively.

D SYNTHETIC BENCHMARK DETAILS

In this section, we provide additional implementation details for the synthetic benchmark employed in our evaluation (Section 6.4).

D.1 Functionality Checks

Where possible, we implemented functionality checks by extending CryptoAPI-Bench's existing testcases to run a full sequence of corresponding operations, so that we can compare the final output to the original input. Listing 2 displays a simplified example of this principle for using ECB mode with a symmetric cipher. For cases involving TLS communication, we check for retained functionality by establishing a connection with and without an ongoing MITM attack, ensuring that the latter succeeds and the former is prevented by CryptoShield's mitigations. Test cases involving CSPRNG seeding are assumed to always stay functional, since CryptoShield never influences the structure of generated values.

D.2 Reflection Test Cases

We extended our benchmark to cover invocations of cryptographic APIs through Java Reflection APIs. For cases that involve interfaces implemented by application-level code, we utilize Java's Dynamic Proxy infrastructure to construct a proxy implementation that forwards all calls to a handler. A simplified example of this approach can be found in Listing 1.

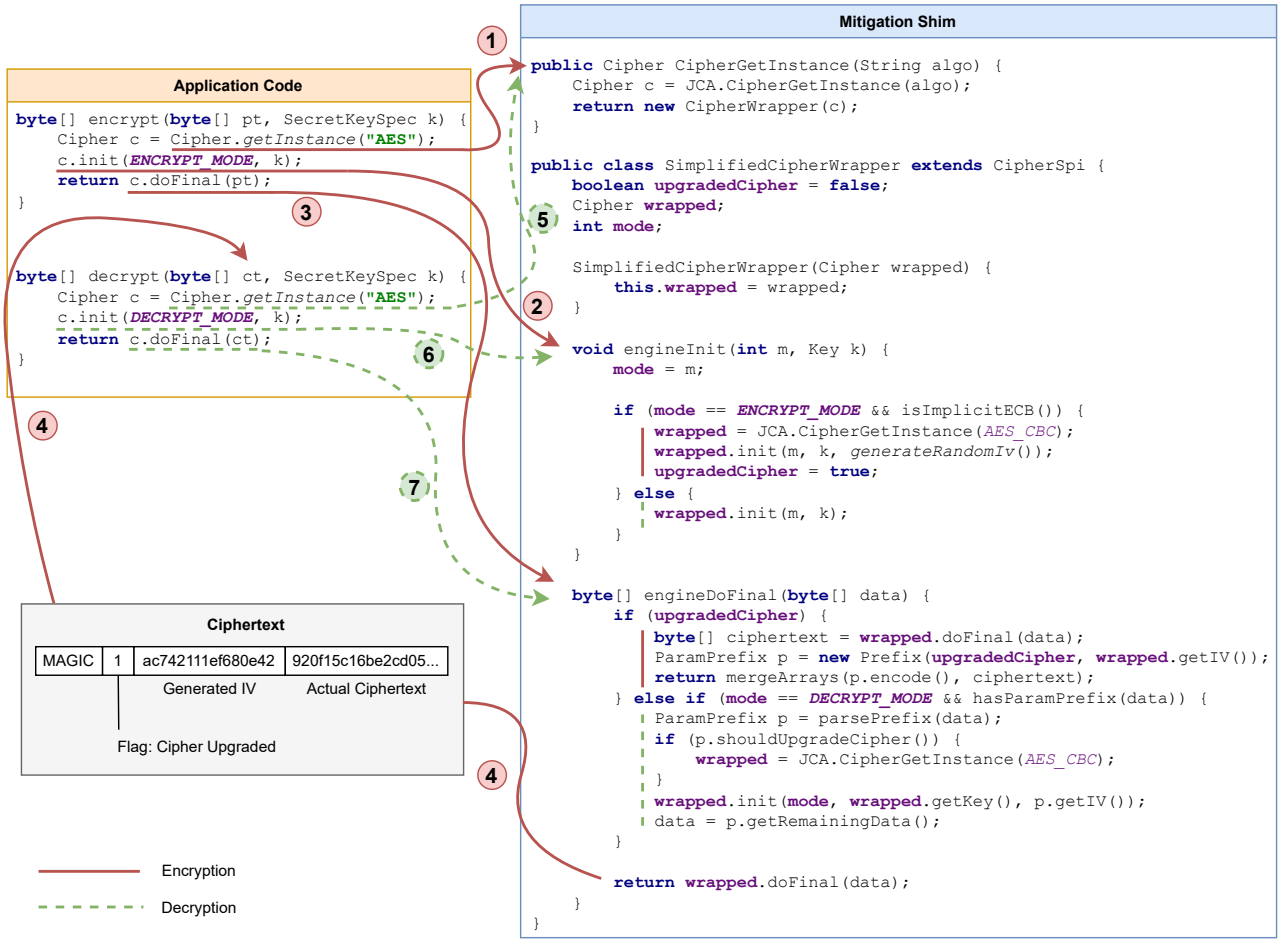


Figure 4: Simplified mitigation procedure for implicit ECB mode.

```
public class Case1 implements InvocationHandler {
    public static void main(String[] args) {
        HostnameVerifier hv = (HostnameVerifier)
        ↪ Proxy.newProxyInstance(...,
              new Class[] {HostnameVerifier.class},
               new Case1());
        // We're simulating a MITM attack here!
        if (hv.verify("mitm.com", session)) {
            ssl.getOutputStream().write(data);
        }
    }

    @Override
    public Object invoke(Object proxy, Method
    ↪ method, Object[] args) throws Throwable {
        return true; // Insecure!
    }
}
```

Listing 1: Example test case using `java.lang.reflect.Proxy`

```
public class EcbCase1 {
    public static void main(String[] a) throws ... {
        KeyGenerator keyGen =
        ↪ KeyGenerator.getInstance("AES");
        SecretKey key = keyGen.generateKey();
        Cipher c = Cipher.
        ↪ getInstance("AES/ECB/PKCS5Padding");
        c.init(Cipher.ENCRYPT_MODE, key);

        byte[] data = new byte[128];
        byte[] cipherText = c.doFinal(data);
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] plainText = c.doFinal(cipherText);
        if (Arrays.equals(plainText, data))
            System.out.println("Still functional");
    }
}
```

Listing 2: We adapted CryptoAPI-Bench to check retained functionality (adaptations highlighted).