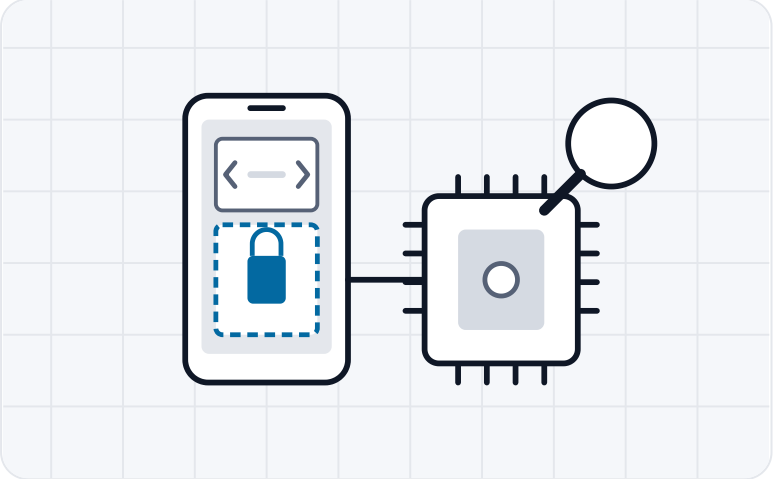
A-SIT Technology
E-Government Innovation Center

10 February 2025 Analysis · Mobile Security · Privacy & Crypto · 13 min read

Security Analysis of Native Cryptography in Android Apps

Florian Draschbacher



Authors	Florian Draschbacher
Version	2025-01-31
Source	https://technology2.a-sit.at/en/publications/security-analysis-of-native-cryptography-in-android-apps/
Note	Machine translation.

Key findings

- Of 2,700 analysed Google Play apps, more than 60 % (1,640) integrate native libraries.
- In 212 of the 1,640 apps with native code (almost 13 %), the OpenSSL version string was found; an estimated almost 5 % of all apps use OpenSSL.
- Most static tools are limited to Java code; only NativeSpeaker examines native code, but it identifies crypto functions only by their exported names.
- The dynamic approach of Piccolboni et al. does not detect misuse in native libraries that each app brings along itself.

- The proposal is dynamic hooking of the OpenSSL API; where OpenSSL is included indirectly, pattern recognition in the compiled code is also needed.

Although Android apps are primarily developed in Java and compatible languages for portability reasons, there is the possibility of implementing particularly performance-critical functionality in native languages. This option is increasingly being used to integrate native cryptographic libraries such as OpenSSL. However, a significant issue is that existing tools for detecting errors in the use of cryptography only consider Java code.

As part of this project, this problem is systematically analyzed and addressed with a solution. First, a brief overview of current tools is provided. Then, we evaluate the prevalence of apps among the most popular Android applications that cannot be analyzed by existing tools due to their use of native cryptography. Finally, we propose a concept for a solution that enables the automated detection of errors in the use of native cryptography.

1. Introduction

Android apps are mainly developed in Java or compatible languages to ensure high portability and compatibility with the Android runtime environment. However, there are numerous use cases in which particularly performance-critical functionality is implemented in native languages such as C or C++. A key motivation is the possibility of using existing, optimized native libraries, especially for resource-intensive operations such as cryptography.

A prominent example of the integration of native cryptography is the use of OpenSSL or comparable libraries that operate outside the Java environment. While this enables more efficient processing in many cases, it also brings challenges: Most existing tools for analyzing the correct use of cryptography focus exclusively on Java code and do not consider logic implemented in native languages. This leads to a blind spot in the security assessment of Android apps.

The insufficient coverage of native cryptography by existing security and analysis tools poses a considerable risk. Faulty implementations of cryptographic methods can undermine security guarantees. Examples include insecure key storage, the use of outdated or insecure algorithms, or faulty random number generators. Since these problems often only manifest at a low code level, conventional static or dynamic analysis tools do not reliably detect them.

The central challenge is therefore to improve the detection of such vulnerabilities. This requires a systematic analysis of existing tools as well as an investigation of the prevalence of apps that use native cryptography without it being covered by previous testing methods.

To capture the problem systematically and develop a solution, this project follows a multi-stage approach. First, an overview of currently available tools for detecting cryptographic errors in Android apps is created. It is examined to what extent these tools are able to identify errors in native libraries. In a second step, it is analyzed how widespread the use of native cryptography in Android apps is. This is done by static analysis of a representative sample of apps from different categories. Finally, a concept for improved automated

detection of errors in native cryptography is developed. The tool can be used by interested users to assess the security of their apps, by developers for independent verification of their products, and by security researchers for large-scale security analysis of the app landscape. Deployment at app stores is also possible, where all apps submitted by developers for publication can be checked before release.

2. Background

This section explains the technologies that are necessary for understanding the later discussion.

2.1. Android

Android is the most popular operating system for mobile devices such as smartphones and tablets. The system is developed within the Android Open Source Project (AOSP) by a number of device manufacturers under the leadership of Google. From a technical point of view, it is an operating system based on a Linux kernel. However, the userspace components differ significantly from those of any other Linux distribution. They were newly developed specifically for the requirements of mobile devices.

2.2. OpenSSL

OpenSSL is a widely used open-source library that provides basic cryptographic functions and the implementation of secure communication protocols such as SSL (Secure Sockets Layer) and TLS (Transport Layer Security). It includes a large number of cryptographic algorithms for symmetric and asymmetric encryption, digital signatures and certificate management. OpenSSL is frequently used in server and client applications to secure data transmissions and is increasingly used in mobile applications, especially when performance-critical cryptographic operations are required.

In Android apps, OpenSSL is typically integrated via the Java Native Interface (JNI). JNI enables communication between the Android code written in Java and native C/C++ libraries such as OpenSSL. This is done via so-called native methods, which are declared in Java and linked to the native implementation by JNI. Through JNI, developers can call OpenSSL functions directly and thus efficiently execute computationally intensive cryptographic operations.

2.3. Crypto-API-Misuse

Crypto-API-Misuse describes the faulty integration of cryptographic functionality. If, for example, parameters are poorly chosen by the app developer, the cryptographic properties provided by correct use (such as confidentiality or integrity) may not be achieved. The most common variants of Crypto-API-Misuse include:

- Use of insecure or outdated algorithms: Many older cryptographic algorithms, such as MD5 or SHA-1, are nowadays considered insecure because they are vulnerable to collisions or attacks. If developers use these outdated algorithms in their app without switching to more modern and secure alternatives such as SHA-256 or AES, attackers can exploit the cryptographic weaknesses and endanger the integrity and confidentiality of the data.
- Statically stored or predictable keys or nonces: Some cryptographic primitives require byte sequences that act as keys or random input. An important requirement for these values is that they are freshly and randomly generated at least for each app instance (keys) or each operation (salt). If

such values are instead stored as constants in the app, an attacker can easily extract them from there and easily decrypt the data or communication protected by them. Applications that do not use cryptographically secure random number generation but instead rely on predictable values such as timestamps are similarly vulnerable.

2.4. Analysis of Native Android Libraries

Static analysis has mostly been used in the past for analyzing Android applications for Crypto-API-Misuse. While this technique works well for Android apps written in Java or Kotlin, the analysis of native libraries developed in C or C++ poses a particular challenge.

A central problem arises from the diversity of binary formats and optimizations. Native libraries are provided in compiled form as ELF binaries (Executable and Linkable Format), which means that conventional static analysis tools specialized in Java bytecode cannot be applied to them. In addition, various compiler optimizations are used that make the analysis even more difficult. Functions are merged through inlining, for example, unused code is removed and control flows are restructured, which considerably reduces the traceability of program flows.

Another problem is the lack of symbolic information. While Java bytecode often contains metadata such as method names and class hierarchies, native libraries frequently lack this information, especially if they were compiled with high optimization levels or without debug symbols. This makes it difficult to determine the exact function of a code section and to reliably identify cryptographic APIs.

The interaction between Java code and native libraries via the Java Native Interface (JNI) presents an additional hurdle. JNI allows Java methods to access native functions. Static analysis tools must therefore consider both the Java and the native code together in order to detect relevant connections. This requires specialized tools that support both Java and native binary analysis.

Finally, there is the challenge of dynamic code generation and obfuscation. Some apps use techniques such as just-in-time (JIT) compilation, dynamic loading of native libraries (e.g., via dlopen) or code obfuscation to deliberately make static analysis more difficult. As a result, critical security-relevant code paths are not directly visible to static tools.

Since many existing security analysis tools examine only Java code, cryptography code embedded in native libraries often goes unnoticed. This leads to potential vulnerabilities that must be uncovered by specialized analysis tools in order to ensure the security of native cryptography in Android applications

3. Detection of Crypto-API-Misuse on Android

In this section, we describe the existing solutions presented in the current literature for automatically detecting Crypto-API-Misuse in Android apps. The discussion places particular focus on addressing Crypto-API-Misuse in native code.

3.1. Static Analysis Tools

The first findings on Crypto-API-Misuse on Android were provided by Egele et al. [1]. They obtained these findings by means of static analysis. In this approach, the program code is analyzed without executing it. Function calls and parameters can only be traced to a limited extent by computing all possible control flows.

Although this removes the need for lengthy interaction with the app under examination, the statements that can be made are vague. For example, code is reported as faulty even if it can never be reached when the app is actually executed.

Conversely, problems that actually exist are not detected, since, for example, the concrete target instances of method calls on interfaces cannot be traced statically. The approach has similar problems when tracing data dependencies via class or method variables or operating-system-specific mechanisms. Finally, static analysis is a very computationally intensive method.

Another serious challenge arises when static analysis is used to examine native libraries of Android apps. Here, not only the Dalvik bytecode format but also the native machine instructions must be traced. These differ significantly between processor architectures and also reveal considerably less information about the control flow (see Section 2.4). Last but not least, Android has no uniform native interface for cryptographic operations. Rather, each application must either implement these from scratch itself or bring its own library build (such as of OpenSSL). Due to the large number of possible libraries and differences even between versions of the same library, even the second case still poses a considerable challenge for the analysis.

For these reasons, most static tools for detecting Crypto-API-Misuse in Android applications are limited to Java code or the Dalvik bytecode resulting from it. Only Wang et al. [2] present a tool that focuses on native code. However, they identify cryptographic functions merely by their exported name. This approach fails in a large number of scenarios, for example when names cannot be mapped or are concealed by obfuscation.

3.2. Dynamic Analysis Tools

Piccolboni et al. [3] were the first to present a dynamic analysis approach for detecting Crypto-API-Misuse in Android apps. To do so, they extended the implementation of various cryptographic programming interfaces of the Android framework with logging functionality that recorded the parameters passed. Applications to be checked were installed on an Android emulator that used the modified Android framework. They were then started automatically and confronted with random input events to ensure that different parts of the app were reached. After a few minutes of runtime, the recordings could be used to determine whether misuse had occurred.

Since the approach only made changes to crypto implementations in the Android framework, misuse in native libraries, which every app has to bring along itself, could not be detected. Nevertheless, this approach allows method calls and parameters to be traced significantly more reliably.

4. Prevalence of OpenSSL in Android Applications

To narrow down the problem of detecting Crypto-API-Misuse in native Android libraries, we put forward the hypothesis that many applications do not implement cryptographic primitives themselves but rely on existing implementations. In particular, we assume that the choice frequently falls on the well-known OpenSSL library. To test this hypothesis, we analyze a dataset of 2700 applications from Google Play. The dataset consists of the most popular free applications from 35 different app categories.

First, we checked the applications for their use of native libraries. We found that more than 60 %, namely 1640 applications, integrate native libraries. For the further analysis of these libraries, we extracted the constant strings they contain. If OpenSSL is integrated, the respective version identifier is always contained

in the library. It always begins with the string “OpenSSL”. This string was indeed found in 212 (just under 13 %) of the 1640 apps with native code.

To confirm the findings, we randomly selected 11 applications from these 212 candidates and examined them manually using Ghidra. In 7 of the 11 apps, we found OpenSSL statically linked. In 5 of 11 apps, all OpenSSL symbols were exported directly, while in 2 cases only a higher-level API was exported. Based on the findings of the manual analysis, we can therefore estimate that OpenSSL is used in about 8 % of all apps with native code, or just under 5 % of all apps. The results are illustrated in Figure 1.

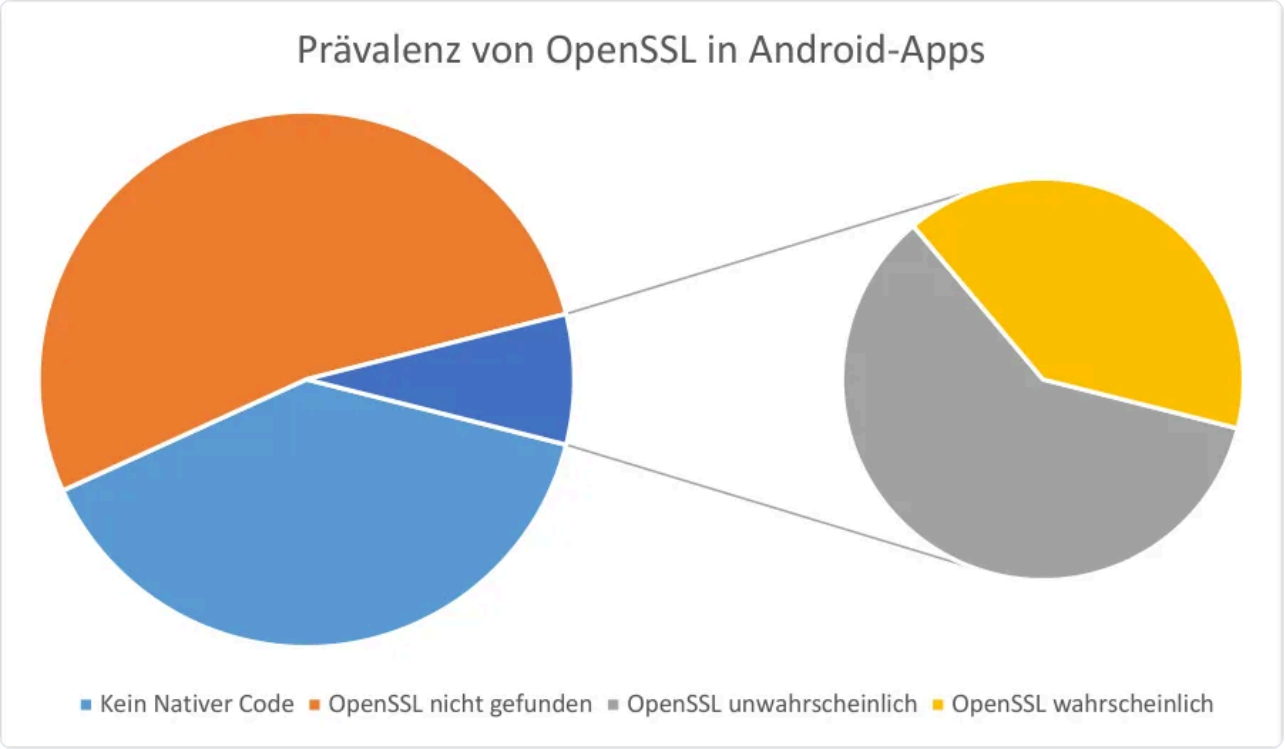


Figure 1: Prevalence of OpenSSL in popular Android apps

5. Automated Detection of Crypto-API-Misuse in Native Android Libraries

From the investigation of the prevalence of OpenSSL in Android apps, we can infer that focusing on OpenSSL alone already allows a considerable number of apps to be covered. We therefore propose here a concept for a tool that can detect Crypto-API-Misuse in apps that use OpenSSL.

5.1. Concept

Following the work of Piccolboni et al., dynamic analysis is also suitable for detecting Crypto-API-Misuse in native Android libraries. However, modifications in the Android framework cannot be relied on. Instead, dynamic hooking must be used to intercept the method calls in the OpenSSL API. All passed parameters are then also available there. In contrast to the cryptographic primitives offered via interfaces in the Android framework, however, the parameters have more complex formats that must first be interpreted. Based on the OpenSSL source code, the offset of the relevant fields in the data structures can be determined and then used to extract the corresponding values.

Particular attention must be paid to identifying the relevant entry points of the OpenSSL library. If the library is integrated directly as a static dependency, the methods of the API are usually exported as symbols of the ELF file. Differences between OpenSSL versions can be captured and handled by means of the version string.

However, if OpenSSL is integrated indirectly, for example because another native library uses it as a hidden dependency, the symbols are not necessarily exported. In this case, the relevant API must first be identified by means of the version string and then located by means of patterns in the compiled code. To do so, a pattern recognition must be implemented that is not influenced by the specifics of different compilers. If the corresponding functions are inlined, further work is needed to identify the parameters. Dynamic hooking alone is then no longer sufficient; selective rewriting of individual locations in the machine code may be necessary.

6. Summary

In this project, the security of native cryptography in Android apps was investigated. To this end, an overview of existing solutions for detecting Crypto-API-Misuse on Android was first created, and their support for, or suitability for, native libraries was evaluated. We then evaluated, using a representative dataset of Android apps, how widespread native code is in Android apps and whether OpenSSL can be identified as a widely used library. Based on these findings, we then presented a concept for a methodology that makes it possible to detect Crypto-API-Misuse in apps that integrate OpenSSL.

References

[1] M. a. B. D. a. F. Y. a. K. C. Egele, „An empirical study of cryptographic misuse in android applications," in Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, 2013.

[2] Q. a. L. J. a. Z. Y. a. W. H. a. H. Y. a. L. B. a. G. D. Wang, „NativeSpeaker: Identifying Crypto Misuses in Android Native Code Libraries," in Information Security and Cryptology, 2018.

[3] G. D. G. L. P. C. a. S. S. L. Piccolboni, „CRYLOGGER: Detecting Crypto Misuses Dynamically," in EEE Symposium on Security and Privacy, 2021.