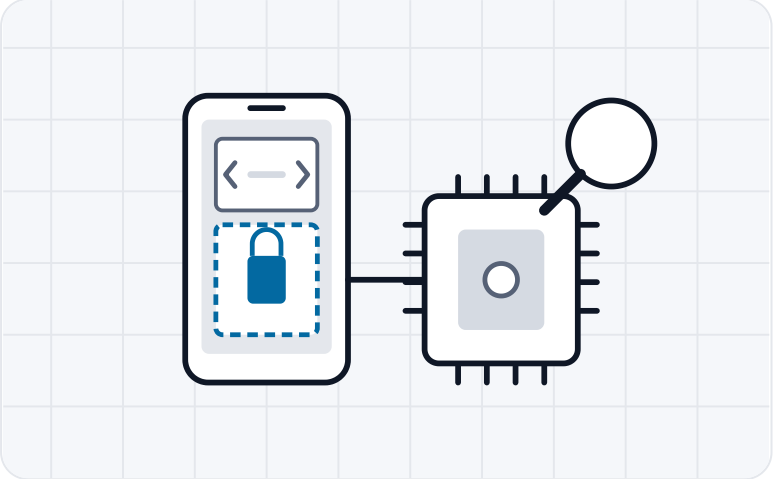
A-SIT Technology
E-Government Innovationszentrum

10. Februar 2025 Analyse · Mobile Sicherheit · Privatsphäre & Krypto · 12 Min. Lesezeit

Sicherheitsanalyse von Nativer Kryptografie in Android-Apps

Florian Draschbacher



Autorinnen und Autoren	Florian Draschbacher
Stand	2025-01-31
Quelle	https://technology2.a-sit.at/veroeffentlichungen/sicherheitsanalyse-von-nativer-kryptografie-in-android-apps/

Kernaussagen

- Von 2700 analysierten Google-Play-Apps integrieren mehr als 60 % (1640) native Bibliotheken.
- In 212 der 1640 Apps mit nativem Code (knapp 13 %) fand sich der OpenSSL-Versions-String; geschätzt nutzen knapp 5 % aller Apps OpenSSL.
- Die meisten statischen Tools beschränken sich auf Java-Code; nur NativeSpeaker prüft nativen Code, erkennt Krypto-Funktionen aber nur am exportierten Namen.
- Der dynamische Ansatz von Piccolboni et al. erkennt Misuse in nativen Bibliotheken nicht, die jede App selbst mitbringt.

- Vorgeschlagen wird dynamisches Hooking der OpenSSL-API; bei indirekt eingebundenem OpenSSL ist zusätzlich eine Mustererkennung im Kompilat nötig.

Obwohl Android-Apps aus Gründen der Portabilität vorrangig in Java und kompatiblen Sprachen entwickelt werden, besteht die Möglichkeit, besonders performance-kritische Funktionalität in nativen Sprachen zu implementieren. Diese Möglichkeit wird immer mehr auch dazu genutzt, native Kryptografie-Bibliotheken wie OpenSSL einzubinden. Problematisch ist hierbei, dass bestehende Tools zur Erkennung von Fehlern in der Nutzung von Kryptografie nur Java-Code berücksichtigen.

Im Rahmen dieses Projekts wird dieses Problem systematisch erfasst und einer Lösung zugeführt. Zunächst wird ein kurzer Überblick über aktuelle Tools geschaffen. Danach evaluieren wir anhand der populärsten Android-Anwendungen die Prävalenz von Apps, die aufgrund der Nutzung von nativer Kryptografie von bestehenden Tools nicht überprüft werden können. Schließlich schlagen wir ein Konzept für eine Lösung vor, mit der Fehler in der Nutzung von Nativer Kryptografie automatisiert aufgedeckt werden können.

1. Einleitung

Android-Apps werden vorwiegend in Java oder kompatiblen Sprachen entwickelt, um eine hohe Portabilität und Kompatibilität mit der Android-Laufzeitumgebung zu gewährleisten. Allerdings gibt es zahlreiche Anwendungsfälle, in denen besonders performance-kritische Funktionalitäten in nativen Sprachen wie C oder C++ implementiert werden. Eine zentrale Motivation hierfür ist die Möglichkeit, bestehende und optimierte native Bibliotheken zu nutzen, insbesondere für ressourcenintensive Operationen wie Kryptografie.

Ein prominentes Beispiel für die Einbindung nativer Kryptografie ist die Verwendung von OpenSSL oder vergleichbaren Bibliotheken, die außerhalb der Java-Umgebung operieren. Während dies in vielen Fällen eine effizientere Verarbeitung ermöglicht, bringt es zugleich Herausforderungen mit sich: Die meisten bestehenden Tools zur Analyse der korrekten Nutzung von Kryptografie konzentrieren sich ausschließlich auf Java-Code und berücksichtigen keine in nativen Sprachen implementierte Logik. Dies führt zu einem blinden Fleck in der Sicherheitsbewertung von Android-Apps.

Die unzureichende Erfassung nativer Kryptografie durch bestehende Sicherheits- und Analysetools stellt ein erhebliches Risiko dar. Fehlerhafte Implementierungen kryptografischer Verfahren können dazu führen, dass Sicherheitsgarantien unterlaufen werden. Beispiele hierfür sind unsichere Schlüsselspeicherung, die Verwendung veralteter oder unsicherer Algorithmen oder fehlerhafte Zufallszahlengeneratoren. Da sich diese Probleme häufig erst auf niedriger Code-Ebene manifestieren, werden sie von herkömmlichen statischen oder dynamischen Analysetools nicht zuverlässig erfasst.

Die zentrale Herausforderung besteht somit darin, die Erkennung solcher Schwachstellen zu verbessern. Dies erfordert eine systematische Analyse bestehender Tools sowie eine Untersuchung der Prävalenz von Apps, die native Kryptografie einsetzen, ohne dass diese in bisherigen Prüfverfahren erfasst wird.

Zur systematischen Erfassung des Problems und Entwicklung einer Lösung wird für dieses Projekt ein mehrstufiger Ansatz verfolgt. Zunächst wird ein Überblick über aktuell verfügbare Werkzeuge zur Erkennung kryptografischer Fehler in Android-Apps geschaffen. Dabei wird untersucht, inwiefern diese Tools in der Lage sind, Fehler in nativen Bibliotheken zu identifizieren. In einem zweiten Schritt wird analysiert, wie weit verbreitet der Einsatz nativer Kryptografie in Android-Apps ist. Dies erfolgt durch die statische Analyse einer repräsentativen Stichprobe von Apps aus unterschiedlichen Kategorien. Abschließend wird ein Konzept zur verbesserten automatisierten Erkennung von Fehlern in nativer Kryptografie entwickelt. Das Tool kann von interessierten Nutzern zu Erfassung der Sicherheit ihrer Apps, von Entwicklern zur unabhängigen Überprüfung ihrer Produkte, sowie von Sicherheitsforschern zur großflächigen Sicherheitsanalyse der App-Landschaft herangezogen werden. Außerdem ist eine Ausrollung bei App-Stores möglich, wo alle von Entwicklern zur Veröffentlichung übermittelten Apps vor der Freigabe überprüft werden können.

2. Hintergrund

In diesem Abschnitt sollen jene Technologien erklärt werden, die für das weitere Verständnis der späteren Ausführungen notwendig sind.

2.1. Android

Android ist das populärste Betriebssystem für Mobilgeräte wie Smartphones und Tablets. Das System wird im Rahmen des Android Open Source Project (AOSP) von einigen Geräte-Herstellern unter Federführung von Google entwickelt. Aus technischer Sicht handelt es sich um ein Betriebssystem auf Basis eines Linux-Kernels. Die Userspace-Komponenten weichen jedoch wesentlich von denen jeder anderen Linux-Distribution ab. Sie wurden speziell für die Anforderungen von Mobilegeraten neu entwickelt.

2.2. OpenSSL

OpenSSL ist eine weit verbreitete Open-Source-Bibliothek, die grundlegende kryptografische Funktionen sowie die Implementierung sicherer Kommunikationsprotokolle wie SSL (Secure Sockets Layer) und TLS (Transport Layer Security) bereitstellt. Sie umfasst eine Vielzahl kryptografischer Algorithmen für symmetrische und asymmetrische Verschlüsselung, digitale Signaturen und Zertifikatsverwaltung. OpenSSL wird häufig in Server- und Client-Anwendungen zur Absicherung von Datenübertragungen eingesetzt und findet zunehmend Verwendung in mobilen Anwendungen, insbesondere wenn performance-kritische kryptografische Operationen erforderlich sind.

In Android-Apps wird OpenSSL typischerweise über das Java Native Interface (JNI) eingebunden. JNI ermöglicht die Kommunikation zwischen dem in Java geschriebenen Android-Code und nativen C/C++Bibliotheken wie OpenSSL. Dies geschieht über sogenannte Native Methods, die in Java deklariert und durch JNI mit der nativen Implementierung verknüpft werden. Durch JNI können Entwickler OpenSSL-Funktionen direkt aufrufen und somit rechenintensive kryptografische Operationen effizient ausführen.

2.3. Crypto-API-Misuse

Crypto-API-Misuse beschreibt die fehlerhafte Integration von kryptografischen Funktionalitäten. Wenn etwa Parameter durch den App-Entwickler schlecht gewählt werden, kann es passieren, dass die bei korrekter Verwendung gegebenen kryptografischen Eigenschaften (etwa Vertraulichkeit oder Integrität)

nicht erzielt werden. Zu den häufigsten Varianten von Crypto-API-Misuse gehören:

- Verwendung unsicherer oder veralteter Algorithmen : Viele ältere kryptografische Algorithmen, wie z. B. MD5 oder SHA-1, gelten heutzutage als unsicher, da sie anfällig für Kollisionen oder Angriffe sind. Wenn Entwickler diese veralteten Algorithmen in ihrer App verwenden, ohne auf modernere und sicherere Alternativen wie SHA-256 oder AES umzustellen, kann dies dazu führen, dass Angreifer die kryptografischen Schwächen ausnutzen und die Integrität und Vertraulichkeit der Daten gefährden.

- Statisch hinterlegte oder vorhersehbare Schlüssel oder Nonces: Einige kryptografische Primitive benötigen Byte-Sequenzen, die als Schlüssel oder Zufalls-Input fungieren. Eine wichtige Anforderung an diese Werte ist, dass sie zumindest für jede App-Instanz (Schlüssel) oder jede Operation (Salt) frisch und zufällig generiert werden. Werden solche Werte stattdessen als Konstanten in der App hinterlegt, so kann ein Angreifer sie von dort einfach extrahieren und die damit geschützten Daten oder Kommunikation einfach entschlüsseln. Ähnlich angreifbar sind auch Anwendungen, die keine kryptografisch sichere Zufallszahlengenerierung verwenden, sondern stattdessen auf vorhersehbare Werte wie Timestamps zurückgreifen.

2.4. Analyse von nativen Android-Bibliotheken

Für die Analyse auf Crypto-API-Misuse in Android-Anwendungen wurde in der Vergangenheit meist auf statische Analyse zurückgegriffen. Während diese Technik für Android-Apps, die in Java oder Kotlin geschrieben sind, gut funktioniert, stellt die Analyse von nativen Bibliotheken, die in C oder C++ entwickelt wurden, eine besondere Herausforderung dar.

Ein zentrales Problem ergibt sich aus der Unterschiedlichkeit der Binärformate und Optimierungen. Native Bibliotheken werden in kompilierter Form als ELF-Binärdateien (Executable and Linkable Format) bereitgestellt, wodurch herkömmliche statische Analysetools, die auf Java-Bytecode spezialisiert sind, nicht auf sie angewendet werden können. Zudem kommen verschiedene Compiler-Optimierungen zum Einsatz, die die Analyse weiter erschweren. Funktionen werden beispielsweise durch Inlining zusammengeführt, ungenutzter Code wird entfernt und Kontrollflüsse werden umstrukturiert, was die Rückverfolgbarkeit von Programmabläufen erheblich reduziert.

Ein weiteres Problem ist die fehlende symbolische Information. Während Java-Bytecode oft Metadaten wie Methodennamen und Klassenhierarchien enthält, fehlen in nativen Bibliotheken diese Informationen häufig, insbesondere wenn sie mit hohen Optimierungsstufen oder ohne Debug-Symbole kompiliert wurden. Dies macht es schwierig, die genaue Funktion eines Codeabschnitts zu bestimmen und kryptografische APIs zuverlässig zu identifizieren.

Die Interaktion zwischen Java-Code und nativen Bibliotheken über das Java Native Interface (JNI) stellt eine zusätzliche Hürde dar. JNI erlaubt es, dass Java-Methoden auf native Funktionen zugreifen. Statische Analysetools müssen daher sowohl den Java- als auch den nativen Code gemeinsam betrachten, um relevante Verbindungen zu erkennen. Dies erfordert spezialisierte Werkzeuge, die sowohl Java- als auch native Binäranalyse unterstützen.

Schließlich gibt es die Herausforderung der dynamischen Code-Generierung und Verschleierung. Einige Apps verwenden Techniken wie Just-in-Time (JIT)-Kompilierung, dynamisches Laden von nativen Bibliotheken (z. B. über dlopen) oder Code-Verschleierung, um die statische Analyse gezielt zu erschweren.

Dies führt dazu, dass kritische sicherheitsrelevante Codepfade für statische Werkzeuge nicht direkt sichtbar sind.

Da viele bestehende Sicherheitsanalysetools ausschließlich Java-Code untersuchen, bleibt der in nativen Bibliotheken eingebettete Kryptografie-Code oft unbemerkt. Dies führt zu potenziellen Schwachstellen, die durch spezialisierte Analysetools aufgedeckt werden müssen, um die Sicherheit nativer Kryptografie in Android-Anwendungen zu gewährleisten

3. Erkennung von Crypto-API-Misuse unter Android

In diesem Abschnitt beschreiben wir die bestehenden Lösungen, die in der aktuellen Literatur vorgestellt werden, um Crypto-API-Misuse in Android-Apps automatisiert zu erfassen. Ein besonderer Fokus wird in der Diskussion auf die Adressierung von Crypto-API-Misuse in nativem Code gelegt.

3.1. Statische Analyse-Tools

Die ersten Erkenntnisse zu Crypto-API-Misuse unter Android lieferten Egele et al. [1]. Sie gewannen diese Erkenntnisse mittels statischer Analyse. Dabei wird der Programmcode analysiert, ohne ihn auszuführen. Funktionsaufrufe und Parameter können nur begrenzt nachvollzogen werden, indem alle möglichen Kontrollflüsse errechnet werden. Zwar besteht so kein Bedarf für langwierige Interaktionen mit der zu untersuchenden App, allerdings sind die Aussagen, die getroffen werden können vage. Es wird etwa Code als fehlerhaft gemeldet, selbst wenn er bei tatsächlicher Ausführung der App nie erreicht werden kann.

Im Gegenzug werden tatsächlich vorliegende Probleme nicht erfasst, da sich etwa die konkreten Ziel-Instanzen von Methodenaufrufen auf Interfaces statisch nicht nachvollziehen lassen. Ähnliche Probleme hat der Ansatz beim Nachvollziehen von Datenabhängigkeiten über Klassen- oder Methodenvariablen oder von betriebssystem-spezifischen Mechanismen. Schließlich handelt es sich bei der statischen Analyse um eine sehr rechenintensive Methode.

Eine weitere schwerwiegende Herausforderung zeigt sich, wenn statische Analyse zur Untersuchung von nativen Bibliotheken von Android-Apps verwendet wird. Hier muss nicht nur das Dalvik-Bytecode-Format, sondern auch die nativen Maschinen-Instruktionen nachvollzogen werden. Diese unterscheiden sich nicht nur wesentlich zwischen verschiedenen Prozessorarchitekturen, sondern verraten auch deutlich weniger Informationen über den Kontrollfluss (siehe Abschnitt 2.4). Nicht zuletzt verfügt Android auch über keine einheitliche native Schnittstelle für kryptografische Operationen. Vielmehr muss jede Anwendung diese entweder von Grund auf selbst implementieren, oder ihren eigenen Bibliotheks-Build (etwa von OpenSSL) mitbringen. Aufgrund der Vielzahl möglicher Bibliotheken und Unterschiede sogar zwischen den Versionen derselben Bibliothek, stell hier sogar der zweite Fall noch eine erhebliche Herausforderung für die Analyse dar.

Aus diesen Gründen beschränken sich die meisten statischen Tools zur Erkennung von Crypto-API-Misuse in Android-Anwendungen auf Java-Code, bzw. den aus diesem resultierenden Dalvik-Bytecode. Einzig Wang et al. [2] stellen ein Tool vor, das sich auf nativen Code konzentriert. Allerdings identifizieren sie kryptografische Funktionen lediglich anhand des exportieren Namens. Dieser Ansatz schlägt in einer Vielzahl an Szenarien fehl, etwa wenn sich Namen nicht zuordnen lassen, oder durch Obfuscation verschleiert sind.

3.2. Dynamische Analyse-Tools

Piccolboni et al. [3] stellten als erste einen dynamische Analyse-Ansatz für die Erkennung von Crypto-API-Misuse in Android-Apps vor. Dazu ergänzten sie die Implementierung verschiedener kryptografischer Programmierschnittstellen des Android-Frameworks um Logging-Funktionalität, die die übergebenen Parameter aufzeichnete. Zu überprüfende Anwendungen wurden auf einem Android-Emulator installiert, der das modifizierte Android-Framework verwendete. Sie wurden dann automatisiert gestartet und mit zufälligen Input-Events konfrontiert, um sicherzustellen, dass verschiedene Teile der App erreicht wurden. Nach einigen Minuten Laufzeit konnte anhand der Aufzeichnungen dann nachvollzogen werden, ob ein Misuse vorlag.

Da der Ansatz hier lediglich Änderungen an Krypto-Implementierungen im Android-Framework vornahm, konnte Misuse in nativen Bibliotheken, die jede App selbst mitbringen muss, nicht erkannt werden. Dennoch können mit diesem Ansatz Methodenaufrufe und Parameter deutlich verlässlicher Nachvollzogen werden.

4. Prävalenz von OpenSSL in Android-Anwendungen

Um die Problemstellung der Erfassung von Crypto-API-Misuse in nativen Android-Bibliotheken etwas eingrenzen zu können, stellen wir die Hypothese auf, dass viele Anwendungen kryptografische Primitive nicht selbst implementieren, sondern auf bestehende Implementierungen zurückgreifen. Insbesondere vermuten wir, dass die Wahl häufig auf die bekannte Bibliothek OpenSSL fällt. Um diese Hypothese zu überprüfen, analysieren wir einen Datensatz von 2700 Anwendungen von Google Play. Der Datensatz setzt sich aus den populärsten Gratis-Anwendungen aus 35 verschiedenen App-Kategorien zusammen.

Zunächst überprüften wir die Anwendungen auf ihre Nutzung von nativen Bibliotheken. Hier konnten wir feststellen, dass mehr als 60 %, nämlich 1640 Anwendungen, native Bibliotheken integrieren. Für die weitere Analyse dieser Bibliotheken extrahierten wir die enthaltenen konstanten Zeichenfolgen. Wird OpenSSL integriert, so ist auch immer der jeweilige Versions-Bezeichner in der Bibliothek enthalten. Dieser beginnt jeweils mit der Zeichenfolge „OpenSSL“. Tatsächlich konnte dieser String auch in 212 (knapp 13 %) der 1640 Apps mit nativem Code gefunden werden.

Zur Bestätigung der Funde wählten wir 11 Anwendung zufällig aus diesen 212 Kandidaten aus und untersuchten sie manuell mittels Ghidra. In 7 der 11 Apps fanden wir OpenSSL statisch verlinkt. Bei 5 von 11 Apps waren alle Symbole von OpenSSL direkt exportiert, während in 2 Fällen nur eine höherliegende API exportiert war. Anhand der Erkenntnisse aus der manuellen Analyse können wir also schätzen, dass in etwa 8 % aller Apps mit nativem Code, bzw. knapp 5 % aller Apps OpenSSL verwendet wird. Die Ergebnisse werden in Abbildung 1 veranschaulicht.

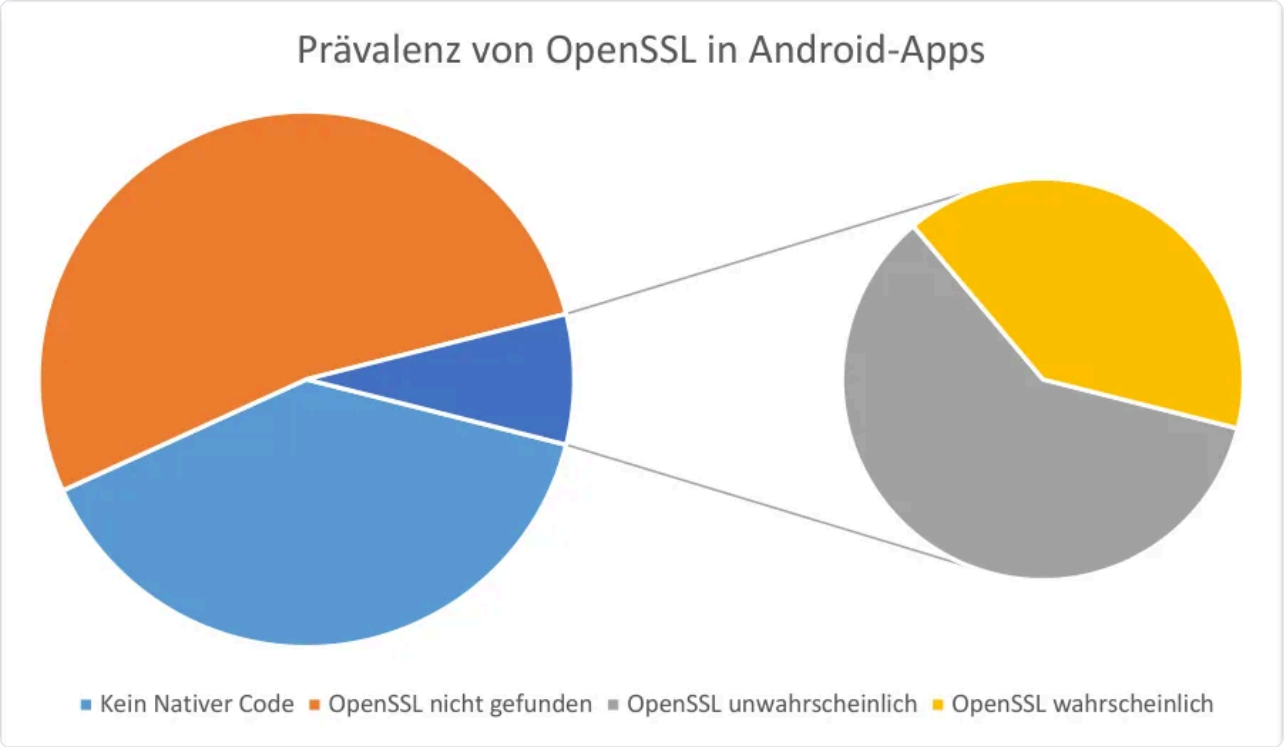


Abbildung 1: Prävalenz von OpenSSL in populären Android-Apps

5. Automatisierte Erfassung von Crypto-API-Misuse in nativen Android-Bibliotheken

Aus der Untersuchung der Prävalenz von OpenSSL in Android-Apps können wir ableiten, dass schon mit der Fokussierung auf OpenSSL eine erhebliche Anzahl von Apps abgedeckt werden können. Wir schlagen daher hier ein Konzept für ein Tool vor, das Crypto-API-Misuse in Apps erkennen kann, die OpenSSL verwenden.

5.1. Konzept

In Anlehnung an die Arbeit von Piccolboni et al. eignet sich die dynamische Analyse auch für die Erfassung von Crypto-API-Misuse in nativen Android-Bibliotheken. Es kann allerdings nicht auf Modifikationen im Android-Framework zurückgegriffen werden. Stattdessen muss dynamisches Hooking genutzt werden, um die Methoden-Aufrufe in der OpenSSL-API abzufangen. Dort stehen dann auch alle übergebenen Parameter zur Verfügung. Im Gegensatz zu den kryptografischen Primitiven, die über Schnittstellen im Android-Framework angeboten werden, verfügen die Parameter jedoch über komplexere Formate, die erst interpretiert werden müssen. Anhand des OpenSSL-Quellcodes kann der Offset der relevanten Felder in den Datenstrukturen bestimmt werden, der dann zur Extraktion der entsprechenden Werte genutzt werden kann.

Besondere Aufmerksamkeit muss der Identifikation der relevanten Einsprungspunkte der OpenSSL-Bibliothek zugemessen werden. Wird die Bibliothek direkt als statische Abhängigkeit eingebunden, so werden in der Regel die Methoden der API als Symbole des ELF-Files exportiert. Unterschiede zwischen verschiedenen OpenSSL-Versionen können anhand des Versions-Strings erfasst und gehandhabt werden.

Wird allerdings OpenSSL indirekt eingebunden, also etwa, weil eine andere native Bibliothek sie als versteckte Abhängigkeit nutzt, sind die Symbole nicht zwangsläufig exportiert. Hier muss dann anhand des Versions-Strings zunächst die relevante API identifiziert werden und dann mittels Muster im Kompilat lokalisiert werden. Dazu muss eine Mustererkennung implementiert werden, die nicht von Spezifika verschiedener Compiler beeinflusst wird. Falls die entsprechenden Funktionen ge-inlined werden, ist weitere Arbeit nötig, um die Parameter zu identifizieren. Hier kann dann nicht mehr nur mit dynamischem Hooken gearbeitet werden, eventuell ist das selektive Umschreiben einzelner Stellen im Maschinencode notwendig.

6. Zusammenfassung

In diesem Projekt wurde die Sicherheit von nativer Kryptografie in Android-Apps untersucht. Dazu wurde zunächst eine Übersicht über bestehende Lösungen zur Erkennung von Crypto-API-Misuse unter Android geschaffen, sowie deren Unterstützung bzw. Tauglichkeit für native Bibliotheken evaluiert. Anschließend evaluierten wir anhand eines repräsentativen Datensatzes von Android-Apps, wie weitverbreitet nativer Code in Android-Apps ist, bzw. ob sich OpenSSL als weitverbreitete Bibliothek feststellen lässt. Auf Basis der Erkenntnisse stellten wir dann ein Konzept für eine Methodik vor, die es erlaubt, Crypto-API-Misuse in Apps festzustellen, die OpenSSL einbinden.

Referenzen

[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.