Johannes Feichtner

Semantic Analysis of Mobile Applications

DOCTORAL THESIS

to achieve the university degree of
Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Prof. Reinhard Posch

Institute for Applied Information Processing and Communications
Graz University of Technology

Graz, July 2020

AFFIDAVIT

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

Date, Signature

Abstract

Applications for mobile devices constantly interact with a multitude of sensors, communication networks, and personal user data. While this paves the way for many innovative use-cases, mobile applications also bear the potential to compromise the security and privacy of user passwords, encryption keys, and other sensitive information. Due to the increasing integration of mobile devices into our daily lives, it is crucial to thoroughly assess and measure the risks associated with entrusting sensitive data to mobile applications.

Heterogeneous runtime environments, permission systems, and the absence of firewalls and powerful anti-virus solutions do not only expose mobile applications to new attack vectors but also require different assessment strategies. The rising complexity and size of nowadays applications impede conclusive security analyses and often limit inspections to single implementation aspects. If a flaw is found, it still needs to be tracked down using manual verification. Some flaws may turn out to be bad practice but not undermine the overall security level. Others could have a serious impact. The distinction of these is not always trivial.

With this thesis, we contribute to a more sophisticated understanding of the implementation and security-critical behavior of Android and iOS applications. Rather than evaluating only the existence of problematic program statements, our research meets the requirements to exactly pinpoint security-relevant weaknesses, tackles code transformation techniques that impede effective fingerprinting and similarity checks, and captures the semantics of source code and app metadata. We present novel analysis solutions to disclose privacy-invasive leaks of passwords and to precisely uncover improper usage of crypto APIs on Android and iOS. To assess and measure the similarity of code, we elaborate technically astute comparison strategies that successfully tackle code obfuscation. Finally, we leverage advanced neural networks to describe the purpose of applications based on their implementation and to understand privacy awareness in app descriptions.

The outcome of this work represents a notable contribution towards a holistic analysis of mobile applications. It helps researchers and users to evaluate the impact of personal data being supplied to mobile programs and to foster an understanding of what is actually executed within mobile applications.

Kurzfassung

Anwendungen für mobile Betriebssysteme interagieren ständig mit einer Vielzahl von Gerätesensoren, Netzwerken und persönlichen Nutzerdaten. Während dies den Weg für viele innovative Anwendungsfälle ebnet, können mobile Anwendungen auch die Sicherheit und die Schutzfunktion von Passwörtern, kryptographischen Schlüsseln und anderen vertraulichen Informationen gefährden. Angesichts der zunehmenden Integration mobiler Geräte in unser tägliches Leben ist es essentiell, die Risiken, die mit dem Anvertrauen sensibler Daten an mobile Anwendungen verbunden sind, gründlich zu untersuchen und zu bewerten.

Heterogene Laufzeitumgebungen, Berechtigungssysteme, sowie das Fehlen von Firewalls und leistungsfähigen Antiviren-Lösungen setzen Apps nicht nur neuen Angriffsvektoren aus, sondern erfordern auch neu gedachte Bewertungsstrategien. Die zunehmende Komplexität und Größe heutiger Anwendungen erschweren schlüssige Sicherheitsanalysen und reduzieren sie häufig auf einzelne Aspekte in Implementierungen. Wird ein Problem festgestellt, muss es nach wie vor manuell verifiziert werden. Dabei kann es vorkommen, dass sich einzelne Mängel als schlechte Praxis herausstellen, jedoch nicht das Sicherheitsniveau an sich unterminieren. Andere Schwächen wiederum können schwerwiegende Auswirkungen haben. Eine genaue Einordnung ist dabei nicht immer einfach möglich.

Mit dieser Arbeit tragen wir zu einem differenzierteren Verständnis der Implementierung und des sicherheitskritischen Verhaltens von Android- und iOS-Apps bei. Anstatt nur die Existenz problematischer Programmteile zu überprüfen, erfüllt unsere Forschung die Anforderungen, sicherheitsrelevante Schwachstellen genau zu lokalisieren, befasst sich mit Code-Transformations-Techniken, welche ein exaktes Erfassen und Ähnlichkeitsprüfungen erschweren, und erfasst die Semantik von Quellcode und Metadaten von Anwendungen. Wir präsentieren neuartige Analyselösungen, um Datenlecks von Passwörtern aufzudecken und die unsachgemäße Verwendung von Krypto-APIs unter Android und iOS präzise festzustellen. Um die Ähnlichkeit von Code zu bewerten, entwickeln wir technisch ausgereifte Vergleichsstrategien, die Code-Verschleierungstechniken erfolgreich umgehen. In weiterer Folge setzen wir hoch entwickelte neuronale Netze ein, um den Zweck von Anwendungen anhand ihrer Implementierung zu beschreiben und das Datenschutzbewusstsein in App-Beschreibungen zu verstehen.

Diese Arbeit trägt wesentlich zu einer ganzheitlichen Analyse mobiler Anwendungen bei. Das Ergebnis dient Forschern und Anwendern, um die Auswirkungen der Bekanntgabe persönlicher Daten an Apps zu bewerten und hilft dabei, besser zu verstehen, was in mobilen Anwendungen tatsächlich ausgeführt wird.

Table of Contents

List of Tables

List of Figures

1 Introduction

Android and iOS have evolved as the two leading mobile operating systems for smartphones, TVs, and many other devices. Designed to regularly interact with user-supplied personal information, sensors, and sensitive input peripherals, such as a camera or microphone, mobile platforms set out the conditions for innovative use-cases. To leverage the extensive technological capabilities of modern devices, platforms like Google Play or the Apple App Store distribute applications that serve a variety of different purposes. Among them, we also find mobile apps to manage passwords, track our health and fitness, organize bank transactions, enable secure messaging, and others where security aspects play an integral role.

While applications for Android and iOS have to undergo an automated or manual check before being published on the corresponding market platforms, users decide upon the trustworthiness of an application primarily based on the developer-provided description text and screenshots. Typically, this information is marketing-oriented, rather than security-centered. To avoid using potentially unknown technical terminology, even security-affine apps like password managers or secure messengers are being advertised as using "military-grade encryption", to be "as beautiful as secure", or with phrases like "your vault is encrypted with bank-level encryption". Evidently, these statements provide only a limited insight into how responsibly critical information, such as passwords and encryption keys, are really protected. Users can hardly be sure whether applications make use of available security features to handle secrets. Developers, in contrast, are constantly requested to align and update their implementations with regards to existing and new security mechanisms offered by a variety of different mobile hard- and software platforms. To protect confidentiality, integrity, and availability of sensitive user data, a profound platform-provided security architecture and correctly implemented applications are of vital importance.

1.1 Problem Statement

Researchers commonly apply techniques for program inspection to uncover security-critical implementation weaknesses in mobile applications. Analyzing a reverse-engineered representation of source code or monitoring the execution behavior of apps reveals a targeted insight into how critical functionality has been realized. On Android, supported by the openness of the platform, app archives can easily be dissected and searched for the existence of vulnerabilities, common attack vectors, and security-relevant implementation mistakes. iOS applications, in contrast, are compiled down to machine code, requiring inspections to operate on disassembled code. In combination with peculiarities of the iOS platform, such as dynamic compile-time decisions and the use of a pointer-aware language, this can be a challenging endeavor. Overall, for apps of both platforms, a steady increase in terms of implementation complexity and code size can be observed.

Manually analyzing apps involves examining security-relevant code parts in thousands of classes, being confronted with meaningless variable identifiers and type signatures due to the widespread use of code transformation techniques, and a non-distinguishable mix of code from third-party libraries and original implementation. Automated alternatives are often tailored to check particular security parameters but cannot conclusively highlight the exact origin of problematic program statements. The heterogeneity of Android and iOS aggravates this issue and requires approaches to be developed individually for each platform.

Existing research in the field of mobile application analysis concentrates on low-level implementation aspects. Tools for static and dynamic application analysis, by design, can only reveal security deficiencies in particular program statements that are specified in advance as being problematic, e.g., the use of a key derivation function with too few iterations [Ege+13] or the implementation of a defective trust manager for TLS connections that simply skips certificate validation [OC15]. The results typically fall into two categories: firstly, a classification into malevolent or harmless or, secondly, execution traces of predefined criteria the inspection has evaluated. While both types may be adequate with regards to the particular objectives, they barely evolve to a superior level where the actual context of problematic statements is considered. In practice, this leads to situations where researchers disclose security flaws in execution traces produced by automated analysis solutions but are unable to reason about the practical impact and relevance of findings. Due to the high complexity of nowadays apps, it can be difficult to verify whether problematic code statements are indeed actively used within the execution of an app and for which functionality they are necessary.

To better estimate the risk that emerges from security problems in Android and iOS applications, it is necessary to also analyze the context in which code statements occur. For example, from an objective point of view, it is alarming if a constant and hard-coded key is employed for encryption purposes. However, if this finding is located within an advertisement library where encryption is only employed for code obfuscation, the impact needs to be assessed differently. Likewise, it depends on the concrete purpose of an application whether requesting certain dangerous system permissions is privacy-invasive or functionally reasonable.

1.2 Motivation

In this thesis, we shift the analysis of mobile applications to a higher level by applying concepts that form structures over disassembled source code. By learning the semantics of code fragments, we cannot only draw conclusions about the semantic relationships of coherent code parts but also set up against obfuscation techniques. The overall goal is to elaborate solutions that do not only allow to verify the presence of particular properties in mobile applications but to deliver a generalized understanding of the main purpose and functionality of an application. For this to achieve, this work comprises two parts.

Until a high-level picture of application behavior can emerge, the semantics at a low level have to be understood. Precisely, in the *first part* of this thesis, we elaborate approaches for a target-oriented reverse-engineering process that fills missing gaps by revealing the exact origin of problematic statements. As pointed out previously, the heterogeneity of currently used platforms in terms of available security features and code formats imposes a significant challenge and demands for individually-designed solutions. We tackle challenges that are still unresolved in the analysis of Android and iOS applications and provide novel insights into the prevalence of password leaks and the improper usage of cryptography APIs.

Having built an environment capable of interpreting the effects of low-level instructions on Android and iOS applications, in the *second part* of this thesis, we establish a platform that enables us to understand the semantics of applications. Therefore, several challenges have to be tackled: (1) Developers often apply code transformation techniques that alter the control and data flow. This makes it difficult to identify code patterns that are known to be vulnerable or to verify how vendors have patched security-critical issues. (2) In addition, the widespread use of third-party libraries and applications with thousands of classes impede to infer a semantic understanding about the purpose of individual code fragments and the context they are embedded in. We elaborate solid strategies that tackle these challenges and that enable a semantic comparison of code fragments, even if a high degree of code obfuscation is used. We propose a practically usable approach for code fingerprinting and recognition and develop a similar concept to identify developer-induced changes in Android applications.

The variety of ways how developers can express semantically similar program statements make it difficult to understand the relevance of individual parts in source code. In addition, the widespread use of third-party libraries and code from external sources do not only enrich apps with additional functionality but also raise the risk to introduce vulnerabilities [Fis+17]. To better estimate the main purpose and functionality of Android applications and to assess whether it seems reasonable that an application requests potentially privacy-invasive system permissions, we leverage advanced machine learning concepts that allow to associate source code with contextual information. In our work, we envisage the use of deep neural networks to correlate vendor-provided app metadata with selected code features. Augmenting the inspection of mobile applications with context-awareness paves the way for a more sophisticated insight into security-critical implementation aspects.

1.3 Contribution and Outline

Our research contributes to a more distinctive understanding of how mobile applications access security- and privacy-sensitive resources. Organized in two successive parts, we first inspect low-implementation aspects in Android and iOS apps and then focus on bridging the semantic gap between the developer-described functionality and the behavior that is actually realized in source code.

Part I: Security-Critical Analysis of Mobile Applications

Chapter 2 provides an introduction to the key aspects of mobile security testing and outlines the analysis strategies that are commonly applied to Android and iOS applications. We explain required preliminaries and summarize implementation weaknesses when using cryptography in mobile applications.

Chapter 3 introduces a static analysis framework that excels in identifying and tracking security-relevant code in Android applications. We develop a strategy to precisely reveal how apps process passwords and show how flexibly definable slicing criteria can be targeted to cover any inspection scenario that involves program slicing in forward or backward direction. As also presented in a paper [Fei18] at *IFIP SEC 2018*, in a case study, we investigate the prevalence of password leaks in Android applications and find that 36% or 182 out of 509 tested apps expose sensitive user input to files or pass them to log output.

Chapter 4 contributes a solution to tackle the complicated and error-prone analysis of iOS applications. This involves handling low-level machine code, dynamic control-flow decisions, and data flows with pointers. Our multi-step approach allows to generically decompile ARMv8 binaries to LLVM IR code, performs static program slicing with pointer analysis, and checks for security-critical implementation weaknesses. For this to achieve, our framework reconstructs data types that were stripped during decompilation, can reliably model the control and data flow of relevant code segments, and allows to verify whether execution paths meet predefined inspection criteria. Together with David Missmann and Raphael Spreitzer, our solution was published in a paper [FMS18] at *WiSec'18* and has received the **Best Paper Award**.

Chapter 5 proposes a streamlined process to analyze the improper usage of cryptography APIs in Android and iOS applications. In a case study with 417 iOS applications, also presented in [FMS18], we find that 82% or 343 tested apps were subject to at least one major security issue. In a subsequent paper [Fei19] at *SECRYPT 2019*, we compare another set of 775 applications that vendors distribute for both mobile platforms and find that 604 apps for iOS (78%) and 538 apps for Android (69%) violate essential security rules. These results do not only highlight the wide applicability of our static analysis frameworks but also point out the high prevalence of security-critical implementation mistakes overall.

Part II: Understanding Mobile Application Behavior

Chapter 6 introduces our approach towards understanding the semantics of coherent code parts. We highlight the limitations of current approaches for application analysis and elaborate a novel concept involving neural networks to model the security-critical behavior of Android applications.

Chapter 7 presents a code recognition technique that excels in identifying third-party libraries as well as code fragments in Android applications, even if a very high degree of code obfuscation is applied. We thoroughly evaluate our solution with obfuscated, shrunken, and optimized code in real-world apps and demonstrate its practical ability to fingerprint and recognize code with high precision and recall. Elaborated together with Christof Rabensteiner, we presented this work in a paper [FR19] at *ARES 2019*.

Chapter 8 extends our goal to assess semantic relationships in code with a solution to identify similarities and differences in the Dalvik bytecode of two given Android apps. Together with Lukas Neugebauer, we developed a novel iterative comparison approach based on Merkle trees and published it in a paper [FNZ19] at *SECRYPT 2019*. Our solution enables to extract developer-induced code changes and succeeds in matching pairs of semantically related code fragments.

Chapter 9 introduces a machine learning-based system that infers the main purpose of apps based on their actual code. Having previously worked on approaches to assess and measure the similarity of code parts, this work focuses on explaining the functionality of Android apps using natural language descriptions. We capture semantic relationships of resource identifiers, string constants, and API calls, and leverage a dense neural network to predict human-readable keywords and short phrases that indicate the main use-cases apps are designed for. We evaluate our solution on 67,040 real-world apps and find that with a precision between 69% and 84% we can identify keywords that also occur in the developer-provided description in Google Play. This work has been done together with Stefan Gruber and is presented in a paper [FG20a] at *IFIP SEC 2020*.

Chapter 10 concentrates on the semantic relation between developer-provided description texts and the use of dangerous permissions. On Android, permissions play a crucial role in protecting users' privacy and descriptions should imply why certain permissions are necessary. To assess privacy awareness, we combine state-of-the-art techniques in natural language processing and deep learning and design a convolutional neural network for text classification. As also published in a paper [FG20b] at *CODASPY'20* together with Stefan Gruber, we apply our solution on 77,000 app descriptions and find that we can identify individual groups of dangerous permissions with a precision between 71% and 93%.

Part I

Security-Critical Analysis of Mobile Applications

2

Security Testing of Android and iOS Applications

In this chapter, we provide an overview about key terminology and security testing principles that are commonly applied to mobile applications. We start with an introduction to security analysis in Section 2.1 and examine the structure of Android and iOS applications in Section 2.2. We outline the basic techniques that can be used to inspect mobile apps and highlight practical challenges of code analysis in Section 2.3. Finally, we explain key aspects of cryptographic APIs on mobile platforms and summarize typical implementation weaknesses in Section 2.4. Parts of this chapter are taken verbatim from [Fei19; Fei18; FMS18].

Publication Data and Contribution

Johannes Feichtner. "A Comparative Study of Misapplied Crypto in Android and iOS Applications." In: *Security and Cryptography – SECRYPT 2019*. SciTePress, 2019, pp. 96–108. DOI: 10.5220/0007915300960108

Johannes Feichtner. "Hunting Password Leaks in Android Applications." In: *ICT Systems Security and Privacy Protection – IFIP SEC 2018*. Springer, 2018, pp. 278–292. DOI: 10.1007/978-3-319-99828-2_20

Johannes Feichtner, David Missmann, and Raphael Spreitzer. "Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications." In: *Security & Privacy in Wireless and Mobile Networks – WiSec'18*. **Best Paper Award**. ACM, 2018, pp. 236–247. DOI: 10.1145/3212480.3212487

Contribution: Main author; Initial prototype implemented by David Missmann. Raphael Spreitzer contributed to the paper introduction.

2.1 Introduction

Android and iOS have emerged as the two leading operating systems for mobile devices. Restricted runtime environments and granular permission models ensure that applications cannot adversely impact other apps, the platform, or the user. Nevertheless, problems can still arise if applications do not correctly address security-relevant properties. To protect confidentiality, integrity, and availability of data processed, e.g., by password managers, secure messengers, and other applications that perform security-critical tasks, it is crucial that developers use platform-provided APIs properly, as otherwise sensitive information might unintentionally be leaked via network traffic, system logs, backups, or more covert channels. In practice, it is often not obvious how thoroughly security aspects have been considered during app development and can only be verified by performing a targeted analysis of potential vulnerabilities.

In recent years, the research of implementation security on Android and iOS has received a lot of attention. A majority of publications in this field focus on the Android ecosystem where the openness of the platform promotes program inspection. This process of reverse engineering is predominantly driven by the motivation to check the existence and correct implementation of security mechanisms. Manually verifying how critical functionality has been realized can be challenging due to the rising complexity and size of today's programs. Automated solutions, on the other side, are often tailored to the inspection of particular parameters but fail to perform a conclusive identification and analysis of relevant program parts. This issue is aggravated by the heterogeneity of Android and iOS, which does not only cause distinct attack vectors but also prevents inspection tools from being re-used for both platforms.

The security-critical analysis of mobile applications is basically done by performing static or dynamic analysis. Depending on the threat model and platform specifics, one or even both approaches can be pursued for analysis purposes. In the case of *static analysis*, a predefined set of inspection procedures is applied to an application. Thereby, although the program is never executed and regardless of the actual behavior, it is possible to draw conclusions based on selected features. *Dynamic analysis*, in contrast, runs the application in a protected environment and tries to recognize previously defined patterns by means of interaction. A major advantage of static analysis involves the possible coverage of application code. For example, when analyzing the data flow of password inputs, dynamic analysis would only match those password fields that were actually visited during execution. Static analysis, on the other hand, is instantly applicable to the totality of password fields, contained within an application.

As mobile applications regularly interact with a multitude of sensors, networks, and user-provided data, security testing has to consider a wide attack surface. Key areas include the secure processing and storage of sensitive data, proper usage of encryption and security mechanisms, communication with trusted network endpoints, and resilience against injection attacks. By carefully following security principles during application development, many of these challenges can be tackled and help to lower the possible impact of successful attacks.

2.2 Reverse-Engineering Mobile Applications

Applications for Android and iOS run on a multitude of devices, ranging from smartphones, tablets, and wearables to TVs, cars and smart home components. Tailored to a specific operating system, apps are executed in sandboxed runtime environments that limit access to the underlying platform and define the boundaries how apps can interact with each other. The heterogeneity of Android and iOS also extends to the file formats used by apps on the corresponding platform. In the following, we point out individual characteristics and describe how apps have to be transformed to enable perform program inspection.

2.2.1 Android

Android applications are mostly developed in Java or Kotlin and build on APIs that are exposed by the platform framework. During compilation, stack-based JVM bytecode is translated to register-based Dalvik bytecode that is later interpreted by the Dalvik Virtual Machine (DVM). By reusing and eliminating repetitive function signatures, code blocks, and string values, the Dalvik compiler manages to effectively reduce the uncompressed bytecode size. As a result, all parts are merged into a single executable file named `classes.dex`.

A `classes.dex` executable consists of multiple sections. Starting with a header, specifying the file type using a magic number, and the format version number, the subsequent sections reference all strings, type identifiers, class and method signatures, fields, and method identifiers using unique IDs. The actual program code is stored in a separate *data* section.

The process of reverse-engineering Android applications usually consists in converting the Dalvik file to Java's `.class` format. Tools, such as *dex2jar*[1] and *enjarify*[2] include predefined rules and constraint solving mechanisms to translate register-based Dalvik bytecode to stack-based JVM code. As this process is non-deterministic, it often leads to spurious or wrong code translations.

Alternatively, instead of decompiling Dalvik bytecode to Java, it can be represented as Smali code. Smali is a mnemonic language to represent Dalvik bytecode in a human-readable and parseable format. As it keeps the semantics of code very close to the original, it is often a preferable choice over more intricate decompilation. The tool *baksmali*[3] disassembles the `classes.dex` file and replicates the structure of the original Java source code, the original directory structure and outputs each class into its own `.smali` file. The same applies for nested classes which are renamed and stored separately. Internally, the tool pursues a recursive traversal approach [SDA02] to find instructions. Thereby, the disassembler continuously follows the control flow of each branch and function call, as encountered within the application.

[1] https://github.com/pxb1988/dex2jar
[2] https://github.com/Storyyeller/enjarify
[3] https://github.com/JesusFreke/smali

2.2.2 iOS

Applications for the iOS operating system, developed in Swift and Objective-C, are distributed in the Mach-O file format, which allows a single binary file to contain multiple Mach-O executables for different CPU architectures. As all iOS devices manufactured after 2013 are equipped with a 64-bit ARMv8 CPU, current applications include at least a binary slice supporting this processor architecture. Internally, each Mach-O executable consists of three regions:

- **Header:** Identifies the executable as Mach-O file and highlights information about the target CPU architecture.
- **Load Commands:** Describes the remaining file layout and designated memory location of segments.
- **Data:** Consists of different sections that are loaded into memory at runtime. The section *Dynamic Loader Info* defines where dynamic symbols should be stored during execution.

A Mach-O file can be processed by parsing all segments in the *Data* region that are referenced within *Load Commands*. These segments contain all information needed for execution, including machine instructions, pointers, strings, and data.

Method invocations are handled by a dynamic dispatch function in the Objective-C runtime library, which requires the type of the object and the name of the method to be called. This involves the following sections in the binary:

- **___objc_classlist:** List of pointers to description of classes in a binary. The memory addresses refer to the section *___objc_data*.
- **___objc_data:** For each class, this section includes pointers to a super class and references to *___objc_const* that involve class meta information, such as initialized variables, strings, integers, and data arrays.
- **___objc_const:** Includes details about all classes, the name and virtual memory location of all implemented methods, instance variables and constants defined in the binary.
- **Dynamic Loader Info:** Includes pointers to classes that are not present in the binary, e.g., references to the Objective-C runtime library. By traversing this structure, it is possible to reconstruct a complete class hierarchy.

During execution, iOS applications can invoke externally defined library functions using *indirect symbols*. Occurring as *lazy* or *non-lazy* symbols, any reverse-engineering approach has to detect and handle these calls correctly as they may create or modify data. Non-lazy symbols are resolved when the binary is loaded using the binding information stored within the *Dynamic Loader Info* section. In contrast, lazy symbols are followed when they are first accessed.

A Mach-O executable defines method signatures for all included Objective-C and Swift methods. Type definitions of regular methods are stored in a

compressed way, prefixing all objects with the identifier symbol '@'. Methods that are to be implemented by classes via so-called *protocols* are denoted using their full signature. For reverse-engineering iOS applications, this information is essential in order to build accurate call graphs, since parameters, passed to protocol methods, are potentially not allocated within the code of the binary but included from system libraries. Without resolving these definitions, types of externally allocated parameters cannot be determined and inferred call graphs will inherently be incomplete.

2.3 Vulnerability Analysis

After transforming mobile applications into a representation enabling further investigation, static and dynamic analysis techniques can be applied in order to check specific security-critical properties. While dynamic approaches work by monitoring the live execution of applications during runtime, static techniques apply taint tracking on a reverse-engineered representation of a program's source code. Targeted at their individual use-case, the majority of analysis-related work aim to disclose possible leaks of private data [CZ15; CHY12; Gib+12; YY12], identify malware [Poe+14; Gas+13; Gra+12b; Fen+14], or to uncover security deficiencies [Bia+15; Fah+12; Dav+10; Fel+11].

On Android, solutions for dynamic application analysis, such as *Frida*[4], *TaintART* [SWL16], or *TaintDroid* [Enc+10] enable live information-flow tracking while a program is executed and can detect privacy leakage in the current execution path. While the real-time monitoring of program behavior can provide a valuable insight into how sensitive data is processed, dynamic security testing, by design, misses code paths that are not visited at runtime.

Supported by the fact that Dalvik bytecode in Android applications can be decompiled to Java code, existing tools for static analysis are easily applicable. Having a source-code like representation, the primary challenge then is to follow arbitrary execution traces as sound and precisely as possible. This objective is tackled by frameworks, such as *FlowDroid* [Arz+14] and *IccTA* [Li+15] that model the entire Android lifecycle, including callbacks, asynchronously executing components, and multiple entry points. Nevertheless, the generation of accurate call graphs remains challenging due the fact that many control flow transitions are made implicitly, i.e., by using Reflection or event listeners. To also consider such method calls, implicit data flows have to be resolved [Cao+15; Bar+15].

Although static application analysis is a well-established practice, only few contributions target the iOS platform. In [Ege+11], the authors approach this field by studying privacy threats in iOS applications using the disassembly of binaries. They create a control flow graph and perform a reachability analysis to identify possible privacy leaks. Other works [Zhe+15; Den+15; Li+14] survey the usage of private APIs or pursue a source-to-sink analysis using a combination of static and dynamic methods.

[4] https://frida.re

2.3.1 Program Slicing

Static slicing can be used to determine all code statements of a program that may affect a value at a specified point of execution (*slicing criterion*). The resulting *program slices* cover all possible execution paths and allow conclusions to be drawn about the functionality of a program.

Weiser [Wei81] introduced an *intra-procedural* method that enables to describe dependencies between program statements using data flow equations. Relevant variables and instructions are determined in an iterative manner. The presented algorithm can be adopted to create slices of code and to find paths from the origin of a parameter to its use, e.g., in cryptographic functions. As summarized by Tip [Tip95], the workflow basically involves two steps:

1. **Follow data dependencies:** This step is executed iteratively, if control dependencies are found.

2. **Follow control dependencies:** Includes relevant variables of control flow statements. The first step is repeated for affected variables.

To create slices over multiple functions, the approach can be extended to *inter-procedural* slicing in two steps: first an intra-procedural slice of a function P is computed, followed by the generation of new slicing criteria for every function that calls P or is called by P. In Weiser's concept, the generation of new criteria is described as $DOWN(C)$ for the callers of P and as $UP(C)$ for functions that are called by P based on the slicing criterion C.

2.3.2 Pointer Analysis

When identifying variables with an impact on program statements, it is essential to also know where they might point to during execution. Pointer analysis can be used to support slicing with accurate information about pointer states.

Introduced by Andersen [And94], pointer analysis is described as a *set-constraint* problem in which a constraint system C is created for a given program. By solving the system, it is possible to determine the locations $loc(v)$ a variable v might point to during program execution. The resulting *points-to set* for v is represented as $pts(v)$. All constraints are of the type $a \supseteq b$, which means that information flows uni-directional from b to a. As shown in Table 2.1, there are four different types of constraints that may be added to C.

Table 2.1: Pointer constraints and their meaning [HL07b].

Program Code	Constraint type	Meaning
a = &b	a $\supseteq$ {b}	loc(b) $\in$ pts(a)
a = b	a $\supseteq$ b	pts(a) $\supseteq$ pts(b)
a = $*$b	a $\supseteq$ $*$b	$\forall$v $\in$ pts(b) : pts(a) $\supseteq$ pts(v)
$*$a = b	$*$a $\supseteq$ b	$\forall$v $\in$ pts(a) : pts(v) $\supseteq$ pts(b)

Andersen defined a *context-sensitive* and a *context-insensitive* version of his algorithm. While the former leads to more precise results by separating information originating from different paths, its time complexity is exponential [WL95; EGH94]. Therefore, context-sensitive pointer analysis does not scale well for larger programs with a large number of calling contexts. Considering the size of today's mobile applications, a context-insensitive method for the computation of points-to sets yields better performance.

Flow-sensitive pointer analysis takes the control flow of a program into account and computes an individual points-to set for each instruction. The *flow-insensitive* pendant disregards branches and collects all information in a single points-to set. Hind and Pioli [HP01] have shown that a flow-sensitive pointer analysis does not improve the precision of the results if a context-insensitive algorithm is used.

2.4 Cryptography in Mobile Applications

A common method to protect sensitive information in Android and iOS apps is the use of system-provided APIs that expose cryptographic functionality. While these high-level interfaces reduce the burden of the developer to understand how cryptographic primitives work internally, it is still vital to use APIs with parameters that do not give a false sense of security. In the following, we introduce platform-provided APIs for cryptography-related applications and provide an outline of common implementation issues.

2.4.1 Android

By including the *Java Cryptographic Architecture* (JCA), Android supports a well-established set of security APIs. The JCA is provider-based which means that interfaces can be implemented by different cryptographic engines in the background. This flexibility also causes distinct providers and algorithms to be usable with each version of Android. To ensure that applications are compatible over multiple releases, developers are advised to choose from a list of recommended algorithms[5] and to not explicitly specify provider names.

Symmetric and asymmetric encryption schemes are exposed to applications through the `Cipher` class[6]. Developers can request a specific scheme by specifying a transformation string as an argument to the `Cipher.getInstance()` method. The parameter encloses the algorithm name, cipher mode, and padding scheme to use within the returned `Cipher` instance object.

To derive cryptographic key material from a given secret while impeding dictionary and brute-force attacks, Android provides the key derivation function PBKDF2 [MKR17] via the `PBEKeySpec` API. By iteratively transforming a given password and salt value using the pseudo-random function HMAC-SHA1, applications can obtain a cryptographic key of specified length.

[5] https://developer.android.com/guide/topics/security/cryptography
[6] https://developer.android.com/reference/javax/crypto/Cipher

2.4.2 iOS

On the iOS operating system, the platform libraries *CommonCrypto* and *Security* expose APIs to perform security-related operations. The first library provides symmetric ciphers, hash functions, and the key derivation function PBKDF2. The second library includes functionality for asymmetric ciphers, certificate and key management, i.e. access to the system's keychain.

In practice, cryptography-related operations in iOS applications that rely on system-provided APIs will typically involve the following three functions provided by the *CommonCrypto* library:

- **CCCryptorCreate:** Provides access to stream and block ciphers. Each call to this function initializes a cryptographic handle, named `CCCryptorRef`, that can subsequently be used for encryption and decryption operations. `CCCryptorCreate` takes the following security-relevant parameters:
  - `op`: Defines whether data should be encrypted or decrypted.
  - `options`: Defines whether to use ECB or CBC mode.
  - `key`: A pointer to the key material.
  - `iv`: A pointer to the initialization vector (IV).
- **CCCrypt:** Similar to the `CCCryptorCreate` function, `CCCrypt` is a self-contained alternative that expects all data and cipher-related options to be passed immediately. The security-critical parameters are the same as for the `CCCryptorCreate` function.
- **CCKeyDerivationPBKDF:** Derives a cryptographic key using PBKDF2. The subsequent parameters are crucial for the security:
  - `password`: Specifies the passphrase to derive a key from.
  - `salt`: The salt byte values used as input to the derivation function.
  - `rounds`: The number of iterations.

2.4.3 Common Implementation Weaknesses

Android and iOS support the protection of sensitive data by providing a variety of algorithms for hashing, encryption, digital signatures, and key derivation. Leveraging the corresponding high-level interfaces enables developers to access security-related functionality. Irrespective of whether the underlying implementation of a cryptographic primitive is correct, it has be supplied with input parameters that guarantee strong security.

Due to the open design of system-provided APIs for cryptography, developers are able to configure integral cipher parameters deliberately. While this supports flexibility in a multitude of different deployment scenarios, the selection of weak parameters can significantly impair the targeted level of security. The following non-exhaustive summary highlights platform-independent configuration issues that are commonly [MBB18; Li+14] encountered with mobile applications.

Symmetric Encryption

Block ciphers, such as AES or DES, split a given plaintext into blocks of a fixed-size and perform encryption individually on each block. By choosing ECB as a *mode of operation*, the result of encrypting previous blocks does not affect subsequent blocks. In case multiple blocks contain identical plaintext, they will be enciphered into identical ciphertext blocks. Consequently, patterns in data are easier to identify and message confidentiality may be compromised.

Another frequently used mode of operation is CBC, in which the previously encrypted block of ciphertext is XORed with the subsequent block of plaintext to be encrypted. To ensure that data patterns are hidden, the first plaintext block is XORed with an IV. Specifying a predictable or non-random IV renders the encryption scheme deterministic and stateless. This, in turn, makes it susceptible to a chosen-plaintext attack that allows an adversary to generate ciphertexts for arbitrary plaintext messages without knowing the key.

As the security of symmetric encryption depends on the secrecy of the key, it is paramount not to store keys in the source code as static, hard-coded values but only within secure areas of a mobile device that are designated for this purpose.

Password-Based Encryption

Schemes for password-based encryption (PBE), such as PBKDF2, are intended to derive cryptographically secure keys from potentially weak passwords. Concatenating a given password with a salt value and iteratively applying a pseudo-random function should hinder table-based attacks. For this to be effective, the used salt value must not be hard-coded, as otherwise the effect is equivalent to not making use of a salt value at all. Also, specifying less than 1,000 iterations, as recommended by RFC 8018 [MKR17] in 2017, facilitates password guessing attacks. Analogous to the key with symmetric encryption, with PBE it is vital to keep the password, that is used as an input for the KDF, secret.

Random Number Generation

Cryptographically secure pseudo-random number generators (PRNG) are designed to generate non-deterministic output. If initialized with a constant *seed value*, the result will be predictable and unsafe for use in cryptographic operations.

2.5 Conclusion

To assess the implementation security of Android and iOS applications, program code has to be reverse-engineered, followed by a vulnerability analysis that typically involves program slicing and, on iOS, pointer analysis. Despite significant research in recent years to uncover security deficiencies in apps, problems, such as insecure data storage or improper usage of cryptographic APIs can still appear, if not considered during development.

3

Static Analysis on Android

Static analysis helps to review the correct implementation of security-critical functionality in Android applications. Applied on reverse-engineered Java source code, existing tools for automated program inspection typically do not consider the characteristics of Android applications and, e.g., miss the opportunity to trace sensitive user inputs, such as passwords and PINs. In this chapter, we introduce a new approach to identify and follow the trace of user input right from the point where it enters an application. By performing static program slicing in forward and backward direction, we are able to reveal potential data leaks and can pinpoint their exact origin. The open design of our solution also covers other use-cases, such as the analysis of improper usage of cryptographic APIs.

In Section 3.1, we highlight challenges of static application analysis on Android. To overcome limitations of existing solutions, in Section 3.2, we propose an approach for program slicing on Smali code. In Section 3.3, we study password fields in Android applications and elaborate slicing criteria to track the data flow of secret inputs. We present a strategy to find potential leaks of sensitive user inputs in Section 3.4 and evaluate its practical applicability within a case study. Parts of this chapter are taken verbatim from [Fei18; Fei19].

Publication Data and Contribution

Johannes Feichtner. "Hunting Password Leaks in Android Applications." In: *ICT Systems Security and Privacy Protection – IFIP SEC 2018*. Springer, 2018, pp. 278–292. DOI: 10.1007/978-3-319-99828-2_20

Johannes Feichtner. "A Comparative Study of Misapplied Crypto in Android and iOS Applications." In: *Security and Cryptography – SECRYPT 2019*. SciTePress, 2019, pp. 96–108. DOI: 10.5220/0007915300960108

3.1 Introduction

Static vulnerability analysis of Android applications involves examining the code of security-critical program parts. A manually conducted inspection typically focuses on indicators for potential problems by searching for specific keywords or API calls, such as TLS-related methods like `setDefaultHostnameVerifier()` or `checkServerTrust()`. Code fragments surrounding these words likely implement security-relevant functionality and might eventually be vulnerable.

The increasing complexity and size of Android applications impede a target-oriented vulnerability analysis and often prevent manual security testing from being practicable. Tools for automated code analysis, as an alternative, attempt to uncover potential vulnerabilities by inspecting data flows or performing a taint analysis. After applying program slicing to determine the control and data flows of relevant code segments, it is possible to check whether implementations comply with a set of predefined rules. The result is typically a summary of warnings, potential flaws, and detected rule violations.

Ideally, tools for vulnerability analysis would automatically disclose security weaknesses with high confidence and point out appendant code statements. While this is feasible in other domains with, e.g., buffer overflows, SQL injections, and cross-site scripting flaws, with mobile applications many types of vulnerabilities are difficult to diagnose and often it is challenging to immediately assess the impact of found flaws. For instance, when looking for hard-coded encryption keys, automated solutions struggle to recognize byte arrays that are assembled only during runtime. Likewise, if data flows exhibit that user-entered passwords or GPS coordinates are sent to remote servers, it depends on the purpose of the corresponding application whether this happens for a legitimate reason.

Solutions for automated code analysis of Android applications operate either directly on Dalvik bytecode [Fan+20; Zha+18a; PS17] or use reverse-engineered source code. As it is challenging to verify found anomalies or suspicious behavior in low-level code, compiled applications are not convenient for vulnerability analysis. Although it is viable to translate Dalvik bytecode to Java source code, purposefully applied tricks against reverse-engineering, compiler-induced code obfuscation, and heuristics during decompilation often tamper with data flows. As a remedy, the code of Android applications can be disassembled to Smali code, which preserves the semantics of Dalvik bytecode but provides an intermediate representation in a human-readable and parseable format (see Section 2.2).

The ability to follow control and data flows is essential in order to identify and analyze security-critical code in Android applications. For this to achieve, we propose a solution to perform program slicing on Smali code. As presented in Section 3.2, we adopt the algorithm of Weiser [Wei81] for Smali code and present a flexible approach to produce data flow graphs for user-definable slicing criteria. By implementing methods for program slicing in forward and backward direction, we gain an insight into the prevalence of security-critical misconceptions in Android applications. Applicable on arbitrary applications, our solution enables to track the data flow of user-entered secrets, uncovers improper usage of cryptographic APIs, and pinpoints problematic code statements.

Privacy Leaks of Sensitive User Inputs

Many Android applications require users to input sensitive data, such as PINs or passwords. Given the ubiquitous and security-critical role of credentials, it is indispensable that programs process secrets responsibly and do not expose them to unrelated parties. Typically, it is unknown whether an input undergoes a cryptographic transformation and if it is safe for a user to enter secrets. To find out how Android applications process sensitive user inputs, all possible execution paths have to be determined in forward direction.

Resilient to dynamic code loading and code obfuscation, solutions, such as *TaintART* [SWL16], *TaintDroid* [Enc+10], or *Mobile Sandbox* [Spr+13] can analyze and detect privacy leakage in the current execution path. Nevertheless, they inherently miss code paths that are not visited at runtime. Leaks of password inputs would, thus, only be detectable for input fields where a password was actively provided by the user. The solution of Cox et al. [Cox+14] mimics this task and inspects the flow of sensitive data in a sandbox. Other works, also based on TaintDroid, uncover privacy leaks based on used permissions [Gib+12] or by enforcing previously elaborated policies [MS12]. More targeted solutions for similar challenges [Bac+16; Ege+13] are tailored to their specific use case and cannot handle the characteristics of both XML resources and dynamically generated input fields. The same applies to the subsequently conducted analysis of potential security-relevant problems where all possible execution paths have to be checked individually. Self-contained implementations and specific output formats make it difficult to extend existing tools with new capabilities. In experiments, we also noticed that some solutions are powerful in general but each present different drawbacks when it comes to aiming them at a specific purpose, such as following the trace of user-entered secrets.

To bridge this gap, we present a framework that features static analysis on definable slicing patterns in order to identify and highlight the improper usage of security-relevant functionality. With the ability to automatically extract, disassemble, and investigate programs, we focus on identifying and tracking user input right from the point where it enters an application.

Starting at predefined lookup patterns, the automated process first aims to derive concrete slicing criteria. Then, we follow the data flow throughout an application and obtain all execution paths that influence an input field under consideration. While applying static forward slicing to track all occurrences of password fields, we simultaneously map visited code lines to a graph-based representation. Subsequently, the results are examined involving predefined rules in order to detect potential security problems. For this to achieve in a most accurate way, backward slicing enables the discovery of influencing parameters. Having determined all relevant variables, the mined context is inspected regarding cryptography-related misconceptions and possible data leaks. To evaluate the applicability of our solution, we conducted a manual and automated inspection of security-related Android applications that process user-entered secrets. As presented in Section 3.4, we found that 182 out of 509 (36%) applications insecurely stored given credentials in files or passed them to a log output.

3.2 Static Slicing of Smali Code

The ability to track data flows in forward and backward direction is crucial in order to isolate those parts of an application that are relevant with regard to a specific slicing criterion. In the following, we present the implemented techniques for static slicing and highlight practical challenges.

3.2.1 Slicing Patterns

The slicing process naturally depends on a slicing criterion referencing a specific line of program code. Considering our objective to track arbitrary data flows matching predefined criteria, a more generic representation is needed. Therefore, we propose so-called slicing patterns that conceptually describe a type of resource or object to track in XML format.

Slicing patterns do not refer to individual program statements and, thus, do not represent slicing criteria by themselves. Instead, they comprise all needed information to dynamically build slicing criteria that coincide with the scheme specified by a pattern. In case relevant statements occur repetitively within applications, slicing criteria are also deduced multiple times and are tracked individually. Depending on the intended focus and level of adaptation, a pattern can be customized for only one specific application, or to be generally applicable for a larger set of targets. To satisfy different requirements, slicing patterns support the specification of *method invocations* and *resource objects* for tracking.

Method Invocations

To follow the trace of dynamically generated input fields, we need to be able to address particular method invocations. Corresponding slicing criteria are identified by looking for all `invoke` statements that match a given pattern. Thereby, all instructions contained within an Android application are scanned and compared with the indicated method signature. For each match, the appendant program statement is considered as a starting point for slicing. Next, the name of the register to track is identified by associating the index of each occurring register with the given parameter of interest. As a result, a set of matching slicing criteria is returned that complies with the initial pattern.

Resource Objects

Resources in Android applications, such as the user interface, layouts and strings, are usually externalized from the program code. For every outsourced element, the developer has to assign a unique resource ID which can later be referenced in code. Tracking concrete IDs would require us to manually modify the slicing pattern for every inspected application. As a remedy, we propose to address specific resource objects by using generalized XPath queries.

As Android resources are typically stored in XML format, XPath is well suited to select elements by means of their node type and a chosen predicates.

Consequently, it is feasible to compile slicing patterns that focus on specified resources that occur in many applications. Instead of tracking individual resource IDs, the formulated XPath queries are intended to cover one or multiple resource elements. By leveraging the flexibility of XPath, queries can be adapted to select arbitrary resource elements that match given properties. In practice, this feature enables slicing patterns to be generalizable to an extent that the characteristics of individual applications become entirely extraneous.

3.2.2 Implementation

By performing program slicing on Smali code, our solution is able to determine the control and data flow of relevant code segments in Android applications. Based on a given slicing pattern, an analysis is conducted in forward or backward direction, storing the results in an object-based graph representation.

Having derived one or multiple slicing criteria from a given pattern, they are added to a queue for processing. This to-do list serves as input for both the forward and backward slicer and collects all registers, fields, return values, and arrays that are subject to tracking. Furthermore, it holds references to all objects that have already been tracked and excludes them from being re-processed. Upon request by the slicing algorithm, the queue delivers the subsequent object to track, which describes the register to follow and the location of the corresponding instruction. Using this iterative approach, we manage to acquire data flows and prevent the repeated analysis of already inspected code branches.

Internally, backward and forward slicing are technically distinct components that process the input from the to-do list and represent slicing results in a dynamically built tree. On top, the slicing criterion is set as root node, followed by all code statements that are contained in the slice. The generated graph serves as an for further analysis, e.g., to evaluate security rules on password fields.

3.2.3 Graph-Based Output

At first, it appeared to be the obvious choice to combine all data flows emerging from one slicing pattern in a single output graph. However, since a pattern can result in multiple slicing criteria, this approach would cause incoherent flows of various criteria to be mixed into a single representation. Besides reducing the expressiveness of resulting graphs, it would also cause inconsistent results as overlapping data flows might occur repetitively. As a remedy, one graph is generated per slicing criterion.

The top node of the tree-like graph output is always the slicing criterion, as it represents the root of all possible execution paths that can follow. Subjacent nodes stand for all code lines which are contained in the slice. In case there are multiple execution paths, e.g., an `if-else` statement, a slice node can have links from multiple predecessor nodes. If code statements are executed repetitively, e.g., via `for` or `while` instructions, loop cycles are denoted between vertices. Each intermediate node refers to a set of all predecessor nodes, including the originating registers as well as registers that occur in the current instruction.

A slice tree can represent one or multiple leaf nodes, whereas each indicates either a constant or shows an abruptly terminated slicing process. Assuming that a constant value, such as an integer, an array, or a string, was copied into the tracked register, slicing may stop as the register value is overwritten. For static backtracking this implies that slicing has led to one or more values that affect the slicing criterion. For forward slicing it means that the currently tracked register will not affect any subsequent variable and, thus, the data flow has reached an endpoint. Leaf nodes are also inserted in case slicing loses track. This happens, for instance, when registers are set as parameters in calls to unresolvable methods, such as calls to system APIs or native libraries.

As all found path endpoints are invariant, we can consider them as constant values. Besides containing information about values that are assigned to registers, constants also explain why paths end at certain points. This is achieved by retaining metadata from slicing. For example, each constant is assigned a category which clearly defines the type of the underlying value. Similarly, in case tracking stops abruptly, constants are put in place to describe the reason.

3.2.4 Slicing Accuracy

The previously mentioned queue ensures accurate analysis results by filtering registers that exceed a predefined threshold of fuzziness. Each tracked register is assigned a *fuzzy level* that indicates its accuracy in accordance with the slicing criterion. In practice, it expresses the likelihood that the value of a currently tracked register still equals the value of the initially followed register. The fuzzy level is also assigned to found constants and nodes within the slice tree in order to highlight their relevance with respect to the slicing criterion. A value of zero means that the result is absolutely precise and has not been modified on its way to the slicing criterion. Higher values indicate less accurate results and a reduced expressiveness of the results.

Although the fuzzy level enables to measure uncertainty in analysis results, it does not draw conclusions about the quality of found constants. For instance, a high value does not necessarily imply that a constant has only marginal impact on a slicing criterion. Likewise, it is feasible that a register exhibits a low fuzzy level but does not correlate with the initial register at all.

3.3 Passwords on Android

The analysis of data flows from input fields for passwords starts with the definition of a suitable slicing pattern. Based on the provided parameters, concrete password field usages are searched in program code, added to slicing criteria and can then be tracked in forward direction. In view of our analysis objectives, the following case study illustrates the derivation of an eligible pattern. With the intention of tracking any password field occurring in practice, we also identify possible shortcomings of elaborated slicing patterns.

Basically, password fields in Android applications are either statically defined as XML resources or generated from program code during runtime. Since both options refer to the same implementation internally, their capabilities and produced outputs are identical. As an initial trigger for slicing, however, it is not feasible to cover both forms by a single slicing pattern. This also coincides with our patterns' types that focus either on method invocations or resource objects. In the following, we will examine both cases and highlight their characteristics.

3.3.1 XML Resources

Password input fields in XML resources typically make use of the element class `EditText` that enables editable input fields to be displayed. Depending on the provided attributes, differently shaped fields and keyboards are presented to the user during interaction. Concise XPath queries facilitate the selection of corresponding input fields for analysis purposes.

Until the release of Android 1.6 (API level 4), the default way to declare password input fields consisted in adding the property `password=true` to an `EditText` element. Although considered deprecated now, the technique can still be found in applications that maintain compatibility with the eldest versions of Android. Referring to the previous section, an XPath statement is suited to specifically match this password input field description. The first-mentioned slicing pattern in Listing 3.1 illustrates the assembled XPath query.

On current versions of Android, password fields are declared by setting a corresponding constant value to the `EditText` element property `inputType`. Alongside with other input types, the change also introduced more fine-grained descriptors for password input fields. For instance, developers can specify the type `numberPassword` in order to restrict possible user input to numerical values only. For the subsequent static slicing process, this implies that the initially tracked value is also numeric and, hence, likely to be subject to integer transformations. If the property `maxLength` is also set, conclusions about the achievable security grade could be drawn even without slicing.

The most obvious descriptor for an arbitrary password combination is the input type value `textPassword`. Considering the previously formulated pattern, the same scheme is applicable to the input type property. The resulting adaptation is depicted in Listing 3.1. In the current state the XPath statement is designed to match *exactly* the given predicate and fail for any deviation. Although it is suited for practical application, the precision is comparably low as other relevant and legitimate input type values are not taken into account. In particular, this concerns all other descriptors, designated for password input, such as `textWebPassword`, `textVisiblePassword`, and `numberPassword`. A possible remedy is to add the listed options to the XPath statement. The resulting query is then able to return all elements with an exactly matching input type value.

Another possible application scenario is the combined use of multiple input types. For example, the value `textNoSuggestions|textPassword` causes the user-shown keyboard to omit the display of any dictionary-based suggestions. Without adaptation to this circumstance our XPath query would not match input type

combinations at all. A pragmatical approach to this issue consists in refining the pattern in a way that it focuses on verifying the occurrence of a password type, disregarding further options. This can be achieved by simply checking whether the property *contains* a known value. In contrast to the previously stipulated exact conformity, we weaken the statement to a containing match. The final slicing pattern is denoted in Listing 3.1. It covers all relevant forms of password types while refraining from matching unrelated values.

Listing 3.1: Forward slicing pattern for password fields.

```
<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
    pattern="//EditText[@password='true']"
    description="EditText XML fields with attribute 'password'" />

<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
    pattern="//EditText[@inputType, 'textPassword']"
    description="EditText fields with inputType textPassword" />

<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
    pattern="//EditText[contains(@inputType, 'textPassword')
      or contains(@inputType, 'textWebPassword')
      or contains(@inputType, 'textVisiblePassword')
      or contains(@inputType, 'numberPassword')]"
    description="EditText fields with password inputType" />
```

3.3.2 Generated Input Fields

Another possibility to display password fields is to generate them dynamically during runtime. Rather than embedding monolithic `EditText` elements in XML resources, editable fields can also be defined using program code. Accordingly, a variety of properties and actions is assignable on each instance of the class `EditText`. A slicing pattern should, hence, be suited to identify generated password fields reliably and to convey slicing criteria for the subsequent tracking process. In order to achieve this, we have to cope with three essential problems:

- How is it possible to distinguish between ordinary `EditText` elements and those that are configured for password input?
- What are the implications of tracking the entire element instead of the password value only?
- How to design a slicing pattern that adapts to the given constraints?

These questions were equally relevant for password fields in XML resources. Nevertheless, in the former case it has shown to be fairly simple to derive a pattern that matches particular properties of one corresponding XML element. With generated input fields, more complex prerequisites apply since password fields cannot be reduced to a single program statement, enclosing all relevant attributes. In the following, we will gradually answer the previously listed questions by examining the sample code provided in Listing 3.2.

Password Field Identification

Initialized within the corresponding Android application context, a dynamically created input field is an instance of the class `EditText`. In order to hide the user-entered text by asterisks, an input field has to be assigned an appropriate *password transformation method*. Similar to XML resources, an optionally added *input type property* restricts the possible input value to a predefined set of characters and advises the keyboard not to save the password for spelling correction. Although not recommended from a security-aware perspective, specifying the input type may be omitted. Consequently, we can conclude that the only irrevocable indicator for a password field (with asterisks) is the assignment of a `PasswordTransformationMethod` class instance. In order to identify an employed transformation object and input type constant, the arguments of `setTransformationMethod()` and `setInputType()` have to be tracked in backward direction.

With visible (non-hidden) password input fields, an entered text undergoes no transformation and, hence, in that case the value of the input type property remains the sole indicator for a password input field. As illustrated in Listing 3.2, the type is declared by a constant value which first points to the possible user-entered values (e.g. text or number) and secondly specifies the particular type of the input field. Accordingly, for visible passwords the second descriptor would be `TYPE_TEXT_VARIATION_VISIBLE_PASSWORD`. The constant states whether an input field is designed to handle passwords and indicates the processed type.

Being assembled at runtime, it might occur that the input type is not immediately assigned to the `EditText` instance upon initialization. Similarly, it is probable that the transformation method changes during execution. This is likely the case with Android applications that offer users the option to toggle the password visibility by clicking on a button. Internally, this is achieved by switching the transformation method, e.g. from `PasswordTransformationMethod` to `HideReturnsTransformationMethod` (or any other non-hiding option) and vice-versa. Unless the password transformation is already registered upon initialization, it is evident that *all* transformation method assignments to an `EditText` instance need to be backtracked in order to determine whether the element acts as an input for passwords at any point of execution. Of course, this process becomes redundant and can be skipped if an input type is set, already referring to a password or PIN code. Overall, the workflow to find generated password input fields can be summarized as follows:

1. Find instances of `EditText` objects and, using forward slicing, verify whether the methods `setTransformationMethod` and `setInputType` are invoked directly upon initialization.

2. Based on the obtained results, backtrack the arguments passed to the found methods. An input field for passwords is found if at least one of the following conditions is met:

   (a) The tracked transformation method is an instance of the class `PasswordTransformationMethod`.

(b) The tracked input type constant value indicates a matching field type for a visible, numeric, web, or general password.

3. If still undecided, track all transformation method or input type assignments appendant to a particular `EditText` instance and perform the evaluation as outlined in the previous step.

Listing 3.2: Kotlin example of a dynamically generated input field.

```
1  val alert = AlertDialog.Builder(this)
2
3  val input = EditText(this)
4  input.setTransformationMethod(PasswordTransformationMethod.getInstance())
5  input.setInputType(InputType.TYPE_CLASS_TEXT or InputType.TYPE_TEXT_VARIATION_PASSWORD)
6
7  input.addTextChangedListener(object: TextWatcher {
8    override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
9      val password = s.toString()
10   }
11
12   override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {}
13
14   override fun afterTextChanged(s: Editable?) {
15     val password = s.toString()
16   }
17 })
18 alert.setView(input)
19
20 val submitButton = Button(this)
21 button.setText("Submit credentials")
22 button.setOnClickListener(object: View.OnClickListener {
23   override fun onClick(view : View?) {
24     val password = input.getText().toString()
25   }
26 })
```

Tracking Password Inputs

Having successfully identified an `EditText` element as a container for password input, the subsequent task consists in tracking the data flow of a user-entered password. Beforehand, a suitable slicing criterion is needed in order to trigger this process. In the following, we highlight the available options and point out possible implications on slicing results.

Basically, it is conceivable to compose a criterion from the previously found `EditText` instance and track the object in forward direction. The resulting slice would, in theory, comprise all code statements that refer to the input field or any of its properties. Applied to the sample code provided in Listing 3.2, the result *should* include the code lines 9, 15, 18, and 24 since they reference the input field object or a derivative. However, as opposed to the directly visible data flow from the `EditText` instance to the `AlertDialog` in line 18, the affiliation with the other code lines is not immediately obvious. To resolve these traces, our slicer is aware of *implicit control flows*, internally handled by the Android framework.

As depicted in Listing 3.2, `EditText` objects support the registration of event-triggered methods. They enable a predefined callback to be invoked whenever

the event is signaled. The sample code demonstrates this feature by means of the `addTextChangedListener` listener. In practice, it causes the method `onTextChanged` (line 8) to be called with the current input field text wrapped as a `CharSequence`, as soon as the text of the input field changes. Another listener method is attached to a button (line 22) and brings the method `onClick` to access the value of the input field (line 24), once the button is clicked. The actual control and data flow in these two examples is carried out internally and beyond the scope of the underlying program code. For static slicing, this means that neither a consecutive nor a coherent data flow is determinable due to missing links in the execution chain. For instance, without being able to track into Android's `TextWatcher` class, a slicer cannot know that the `CharSequence` encloses the value of the input field. More generally, the slicer will miss all information flows that are handled within a listener-callback system, leading to considerable imprecision and false negatives in the overall output.

One way to address the issue consists in statically linking callbacks and their registrations. For instance, assuming that a call to `addTextChangedListener` is encountered by the slicing process, a previously learned mapping could disclose that the actual input value is made available through a `CharSequence` or `Editable` parameter. The downside of this approach, however, is that all probable associations have to be known in advance. Considering the extensive amount of possible listeners and callbacks on the Android ecosystem, a manually managed database is likely to cover only a subset of all implicit control flows.

Instead of tracking `EditText` instances, another approach is to track methods that are known to access the password value. E.g., by defining invocations of `EditText->getText()` as slicing criterion for forward tracking, it can safely be assumed that the initially sliced register holds the actual password value. Employing the same criterion for backward slicing reveals whether the originating `EditText` instance sets an appropriate transformation method or input type. Compared to the formerly described method, this combination of slicing into both directions enables the resulting slice to start with the password value itself (instead of the input field) and ascertains that it is not influenced by unrelated properties of the originating `EditText` object. However, the focus on specific methods, such as `getText()`, also causes other accessors to be excluded a priori.

The following key points can be concluded from the described approaches:

- The slicing criterion has to be assigned an `EditText` element or an access method, such as `getText()`, in order to track password input fields.
- Depending on the initial trigger, the slicing results may include code statements that are not related to the input field value at all.
- User-entered passwords are typically passed to event-triggered callbacks.
- By implicitly referring to an `EditText` instance, password values are made available via different data types and access descriptors. The slicing process has to know these characteristics in advance.

3.4 Finding Password Leaks

Evaluating the data flow of passwords regarding security aspects is challenging since the severity of problems may depend on the context of an application. For example, it might be inappropriate to flag an application insecure due to the fact that a password does not undergo a cryptographic transformation. Of course, the opposite can be true for applications where cryptography is inevitable in order to protect sensitive data.

Considering passwords as sensitive information, our security rule focuses on general misconceptions that substantially affect its secrecy. For instance, one paradigm states that passwords must not be written to a logging function. This emerges from the fact that the mandatory confidentiality is no longer given as soon as an unintended party is able to learn secret credentials. By analyzing the data flow between a password field and one or multiple endpoints, we aim to answer the following questions:

- Is an entered password written to an output file?
- Is a password leaked to a logging function / logfile?
- Is a cryptographic transformation applied to an input?

If one of the first two conditions is satisfied, the security of an entered password is clearly impaired. The latter question specifically depends on the investigated application. For example, under normal circumstances there is no need for a Mobile Banking application to transform a password in order to login to the service behind. In contrast, a program intending to securely store data protected by a user password undoubtedly should apply cryptography for key derivation and data encipherment.

3.4.1 Detection Strategy

Using the following workflow, we intend to evaluate the questions listed before:

1. Identify available password fields by applying the patterns, elaborated in Listing 3.1. For each occurrence, track all resource usages in forward direction.
2. From each computed slicing graph, extract all feasible execution paths and evaluate the following conditions:
   (a) Raise an alert if the data flow includes calls to `write(...)` methods of the (sub)classes of `java.io.OutputStream` and `java.io.FileWriter`. Also detect when passwords are exposed using `java.io.PrintWriter`.
   (b) Check if a password is sent to log output or leaked to a logfile using methods of the `android.util.Log` API. Issue a warning if corresponding calls have been found.

(c) Verify if a password is processed by security-related APIs, exposed in `java.security.*` and `javax.crypto.*`. If found, emit a notification.

The detection workflow starts by obtaining the slicing graphs for all password fields. Initially containing the offset of the password resource, the data flow of an execution path models all program statements that are affected by the input field. Inspecting the graph enables us to search specific accessors that are known to implement the questioned behavior.

3.4.2 Case Study

The goal of this study is twofold. First, we intend to assess the practical feasibility of our analysis solution. Therefore, we manually contrast the output of our framework with the actual source code of real-world applications. This helps us to identify possible weaknesses in our approach and implicitly highlights the framework's reliability. Second, by applying our tool on a larger number of current applications that include password inputs, we gain a valuable insight into the prevalence of potential security problems.

For the case study, we conducted both a manual and an automated analysis on the same dataset. In the following, we explain the applied methodology, the individual goals, and what applications were analyzed. Lastly, we combine both approaches into a single representation and point out notable findings.

Methodology

Before testing the automated analysis, we manually reverse-engineered and examined the source code of 522 applications that use input fields for secrets. All of them were downloaded from the official Google Play Store and had at least 10,000 installations. 206 applications were "password managers", intended to protect user-entered credentials by means of cryptography. The remaining applications served different purposes: mobile banking (145), cloud storage (68), secure data container (12), messenger functionality (91).

The idea of the manual analysis was primarily to collect a ground truth about what our approach should later find automatically. Meanwhile, we repetitively refined the implementation where we recognized deficiencies and ensured all components would interact well enough with each other. Besides identifying opportunities for future improvement, we also benefited from seeing what our security checks would be able to (not) cover in a real-world scenario.

In the second step, we applied our framework on the dataset. For each automatically inspected application, we obtained a generated report that included all found input fields for secrets, for each of them the possible execution paths and the result of the performed security checks.

Results

In total, we applied our framework to 522 selected Android applications. As listed in Table 3.1, among the investigated programs, 10 could not be analyzed

Table 3.1: Case study on password leaks.

	Count	[%]
Downloaded from Google Play Store	522	
Failure during static slicing	10	2%
Out of memory	3	1%
Analyzable with password inputs	509	97%
Input fields for secrets	2,874	
Secrets passed to crypto-related functions	1,181	41%
Secrets leaked through `android.util.Log`	577	20%
Secrets written to a file output	346	12%
Input fields leaking secrets	923	32%
Apps with unsafe input fields	182	36%

automatically as the slicing process was either aborted after the defined threshold of 25 minutes or it surpassed the limit of 80,000 tracked registers. The analysis of another set of three applications failed due to limitations in the amount of usable memory. Precisely, during the pre-processing step, the automated analysis ran out of memory while parsing the Smali code into an object-oriented representation. A manual review of the affected programs revealed that their Dalvik bytecode contained tricks to hamper reverse-engineering. Apart from that, we could verify that these apps process secrets safely. As a result, for 97% or 509 out of 522 applications the analysis workflow terminated successfully.

During our study, we disclosed a total of 2,874 input fields for passwords or PINs. The manual review revealed that the amount of fields used correlates with the program's category. While, on average, mobile banking applications include 2, messengers provide up to 8 input fields for secrets.

Overall, we found that 41% or 1,181 entered secrets were processed by security-related APIs. Clearly, it depends on the purpose of the individual input field whether a cryptographic transformation is appropriate. However, of 206 inspected password managers, we observed that in 38% or 78 applications none of the available input fields for secrets was linked to security-related APIs. Although this does not immediately imply security issues in all affected programs, a further inspection seems advisable.

The secrecy of the user-entered data is only preserved if the associated data flows do not allow an attacker to learn credentials. Unfortunately, we found that in 20% or 577 input fields the secret was passed to a log output. Likewise, the input to another 12% or 346 fields was written to files. Interestingly, as also confirmed by the manual analysis, no credentials were leaked both to log output and files. In summary, we observed that 32% or 923 out of 2,874 inspected input fields leaked input data either to files or log output. With regard to the set of 509 investigated applications, it can be subsumed that 36% or 182 are subject to an issue that substantially affects the secrecy of entered passwords.

Evidently, the precision of our analysis results is strongly linked to the accuracy of the inspected data flow graphs. The manual analysis step ensured that there are neither false positives, nor false negatives with regard to our dataset. Nevertheless, from the obtained results we conclude that our solution qualifies for use with an arbitrary dataset. Conversely, this does not imply the security checks are exhaustive and there is no more room left for improvement. In fact, additional patterns and checks could be suited to reveal further misconceptions.

3.5 Conclusion

The analysis of security-critical execution paths in Android applications is crucial in order to uncover leaks of sensitive data, improper usage of security APIs, and vulnerable code fragments. In practice, many flaws are difficult to diagnose and cannot easily be verified in reverse-engineered source code.

We presented a target-oriented approach to track the data flow of input fields in Android applications by means of static analysis. Based on the proposed concept of slicing patterns and a combination of program slicing on Smali code, our framework excels in following user-provided input right from the point where it enters an application. We assessed our approach by analyzing 509 applications manually and automatically. We detected that 36% or 182 applications leak sensitive user input either to files or log output. This result does not only highlight the viability of our solution but also underlines that misconceived processing of secrets is a common issue in Android applications.

In the context of this thesis, our contribution strives to make the semantics of Android applications understandable at a low level. Our solution takes into account the deficiencies of existing state-of-the-art tools for static analysis. By performing slicing on Smali code in forward and backward direction, we disclose potential data leaks and can exactly pinpoint their origin in code. Due to the open design of our solution, an application is not limited to user inputs but can, basically, cover any inspection scenario that involves an analysis of data flows. Our results highlight the need for approaches to automatically uncover implementation weaknesses and stress the need for further research that helps to understand the logical context of code fragments in mobile applications.

4

Static Analysis on iOS

Static application analysis on iOS involves working with low-level machine code. In this chapter, we introduce a multi-step approach to facilitate the security-critical inspection of iOS binaries. First, we present a solution to decompile machine code for 64-bit ARMv8 CPUs to a higher-level representation. Based on the reverse-engineered code, we can perform program slicing and pointer analysis to reconstruct the control and data flow of relevant code segments. As a result, we are able to verify whether execution paths meet predefined inspection criteria.

We start by explaining the challenges of static analysis on iOS in Section 4.1 and outline how they can be tackled. In Section 4.2, we propose a workflow for the static analysis of iOS binaries and highlight the involved steps. We describe the decompilation process in Section 4.3 and examine how to recover type information from the binary that was stripped during compilation. Having acquired code in a higher-level representation, Section 4.4 describes our approach to resolve pointer states and to obtain an accurate call graph. Section 4.5 elaborates on how we can leverage this information for program slicing and parameter backtracking. Parts of this chapter are taken verbatim from [FMS18].

Publication Data and Contribution

Johannes Feichtner, David Missmann, and Raphael Spreitzer. "Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications." In: *Security & Privacy in Wireless and Mobile Networks – WiSec'18*. **Best Paper Award**. ACM, 2018, pp. 236–247. DOI: 10.1145/3212480.3212487

Contribution: Main author; Initial prototype implemented by David Missmann. Raphael Spreitzer contributed to the paper introduction.

4.1 Introduction

Many applications for the iOS platform perform sensitive tasks and process user data. An effective method to verify how thoroughly security-critical functionality has been implemented is to perform static analysis on a reverse-engineered representation of the application archive.

While Android applications are provided in a reversible bytecode format, programs for iOS are compiled to machine code that is tailored to a particular CPU architecture. Manually inspecting the disassembled code of an executable can be a challenging endeavor as the increasing complexity and size of today's applications impede a conclusive analysis. Automated tools for binary inspection, in contrast, are typically not aligned to the characteristics of the iOS platform and fail to perform a thorough data flow analysis. Among these characteristics are, for example, dynamic control-flow decisions and the use of a pointer-aware language (see Section 2.3.2), where pointers may point to different memory locations and memory locations may be referenced from different pointer variables (*aliasing*). Especially the use of a pointer-aware language requires particular attention in order to focus on code parts that are essential for a particular computation by means of program slicing (see Section 2.3.1).

In the following, we introduce a solution that enables such an analysis on iOS applications by addressing the following challenges.[1]

1. *Decompiling and simplifying machine code*: The analysis of 64-bit binaries compiled for the ARMv8 architecture is error-prone and tedious. Therefore, we transform the binary into higher-level LLVM intermediate representation (IR)[2], where all low-level CPU instructions have to be modeled appropriately. This allows to re-use existing LLVM-based tools, such as KLEE [CDE08], PAGAI [HMM12], and LLBMC [MFS12].

2. *Language peculiarities*: iOS applications are developed in runtime-oriented languages such as Objective-C and Swift, and the majority of control-flow decisions are made during runtime. Instead of calling methods of objects directly or through virtual method tables (*vtables*), this task is delegated to a dynamic dispatch function in the Objective-C runtime library. To recover a semantically correct control flow from the binary, we reconstruct the hierarchies of classes, methods, and types from binaries. This information allows to resolve the target of a function call through the dispatch routine.

3. *Pointer analysis:* Computing control flow and data dependencies, as well as the identification of instructions and variables that have an impact on a particular program statement, requires information about where different variables (and CPU registers) point to during execution. Since computing points-to sets is an undecidable problem [And94], we propose a solution addressing the trade-off between the accuracy of program slices and the runtime overhead.

[1] The framework is available at: https://github.com/IAIK/ios-analysis
[2] https://llvm.org/docs/LangRef.html

We propose and implement the necessary building blocks for a framework that tackles these challenges and enables a static analysis of iOS executables. With our approach, arbitrary ARM 64-bit binaries can be decompiled to LLVM IR code. Instead of being constrained to a low-level inspection of machine code, our solution simplifies reasoning about instructions using LLVM-based analysis tools. Our contribution includes the following key components:

- **Decompiler.** As there are no decompilers available for iOS binaries yet, we introduce a generic decompiler transforming ARMv8 binaries to LLVM IR code. With applications developed in Objective-C or Swift, it is significantly harder to obtain a meaningful program control flow on iOS compared to Android. In addition, the need for pointer analysis and the reconstruction of information from the binary are major differences that required new approaches to be pursued.

- **Static Slicing.** We use static slicing to extract a subset of the program affected by a specific variable. Introduced by Weiser [Wei81], the idea was to describe dependencies between statements using data flow equations. Subsequent works extended the concept to Program and System Dependency Graphs (PDG, SDG) [OO84; HRB90], defined as a reachability problem. Agrawal et al. [ADS91] proposed an approach to create program slices using PDGs that handle pointers and arrays in intra-procedural programs. In [BH04], the supportive impact of pointer analysis on program slicing has been underlined.

- **Pointer Analysis.** For a reliable analysis of data flows, it is inevitable to determine what values are referenced and modified by pointers. Shapiro and Horwitz [SH97b] compared the precision of different techniques for pointer analysis [And94; Ste96; SH97a]. Of the three tested approaches, Andersen's [And94] was the most precise but had a runtime of $O(n^3)$. Steensgard's algorithm [Ste96] runs in almost linear time having less precise results. The third algorithm [SH97a] is a compromise between runtime and precision. The main difference between Andersen's and Steensgard's algorithms is that Andersen uses so-called *inclusion relations*, while Steensgard builds on *equality relations*.

  Since iOS applications usually consist of a very large code base, Andersen's initial algorithm would lead to a poor overall performance. However, various improvements [HT01; PKH07; Ber+03; HL07b; HL07a] have been proposed to tackle this issue. Hardekopf and Lin [HL07b] improved the approach to an almost linear runtime while still providing results similar to Andersen's algorithm. In our solution, we use the constraint optimizations proposed in [HL07a]. By merging the definitions of similar pointer variables, the input size for constraint solving becomes smaller. In combination with another approach [HL07b], we further optimize constraint solving as strongly connected components in a graph are detected and collapsed to a single node as they form cycles in which each node still points to the same location.

4.2 System Design

In this section, we provide an overview about our analysis concept and describe the individual steps. As depicted in Figure 4.1, analysis starts with a 64-bit Mach-O binary executable for the ARMv8 platform. The overall workflow can be summarized as follows:

1. **Disassemble:** After extracting a 64-bit ARMv8 binary from the Mach-O file, we can leverage the disassembler of the LLVM framework to generate assembly code.
2. **Decompile:** We translate the ARMv8 assembly code into LLVM IR code by extending an existing decompiler framework with instruction semantics for ARMv8.
3. **Pointer Analysis:** The points-to sets are computed for all pointers using an enhanced context-insensitive approach that scales well for arbitrary iOS applications of any size.
4. **Static Slicing:** Relevant segments are identified in LLVM IR code based on the slicing criteria derived from the predefined security rules.
5. **Parameter Backtracking:** All execution paths a parameter can take are backtracked to the slicing criterion. This enables us to verify whether encountered statements meet predefined inspection criteria.

Except for the initial disassembly step that is pursued by LLVM, we contribute new approaches and augment existing frameworks. In the subsequent sections of this chapter, our solution is explained in detail.

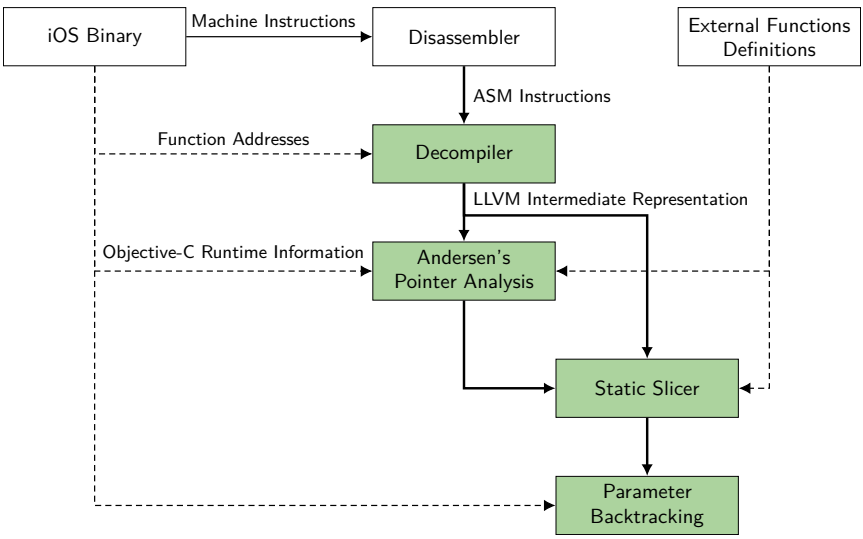


Figure 4.1: Analysis workflow of an iOS binary

4.3 Decompilation to LLVM IR

In this step, we translate 64-bit ARMv8 binaries into LLVM IR code. The aim is to obtain a simpler representation that still models a semantically correct control flow and data flow of the application.

The LLVM compiler *frontend* takes source code as input. After the code has been fully tokenized, parsed, and analyzed, LLVM IR code is emitted. The compiler *backend* is then responsible to optimize it, assemble machine code and link the resulting object. Within this process, LLVM tailors the output to a specific CPU architecture using the corresponding register and instruction descriptions. While the former specifies the processor's register types and relations, the latter includes pattern definitions used to select machine instructions in place of IR instructions during code generation[3].

In order to decompile binaries, we apply the patterns in the opposite direction. Therefore, we build on the reverse-engineering framework *Dagger*[4]. In contrast to similar approaches[5][6], it extends LLVM and relies on *instruction semantics*[7] to translate machine instructions to LLVM IR code based on target descriptions of registers and instructions. The semantics describe the different types of operands a machine instruction can take and what operations have to be applied to get an equivalent in LLVM IR code. For most machine instructions, semantics can be generated from the descriptions. However, since not all instructions necessarily have a counterpart in LLVM IR, we manually supplemented missing definitions for the 64-bit ARM architecture.

The ARMv8 instruction set knows various control flow statements that all have to define a target where the branch should link to. While for most instructions the address is statically defined, the unconditional branch statements `BR`, `BLR`, and `RET` read the destination from a register whose value cannot be resolved during decompilation. As a remedy, we perform a pointer analysis on the register and then update the branch targets in the control flow.

Function calls, i.e. the `BL` and `BLR` instruction on ARMv8 CPUs, are translated to the `call` instruction in LLVM IR. Passing parameters to functions is done by storing values in registers and/or on the stack[8]. Further steps are not required since the called function retrieves the values from the registers and the stack via the register set. This, however, means that we do not have information about parameters or return values. As static analysis requires this information to reconstruct the data flow between functions, we have to restore the missing type information from the binary (see Section 4.5.1).

As shown in Algorithm 1, the decompilation itself consists of two steps. First, all *MachineInstructions* are grouped into basic blocks that resemble the control flow (lines 1 to 16). If an instruction is a control flow statement, its target

[3] https://llvm.org/docs/CodeGenerator.html
[4] https://github.com/repzret/dagger
[5] https://github.com/trailofbits/mcsema
[6] https://github.com/draperlaboratory/fracture
[7] https://llvm.org/docs/TableGen/
[8] http://infocenter.arm.com

address is determined. The second step iterates over *MachineFunctions* and its basic blocks of disassembled instructions and decompiles them to LLVM IR using the semantics (lines 17 to 24). If the exit point of a basic block is reached, a terminator instruction is added to the LLVM IR code that defines the subsequent basic blocks or returns from the function.

Algorithm 1: Decompilation workflow

```
 1 for MI in MachineInstruction do
 2 |   if MI.address in FunctionStarts then
 3 |   |   createMachineFunction(MI.address)
 4 |   |   createMachineBasicBlock(MI.address)
 5 |   end
 6 |   DIS ← disassemble(MI)
 7 |   if branchInstruction(DIS) then
 8 |   |   if not callInstruction(DIS) then
 9 |   |   |   if getTarget(DIS) then
10 |   |   |   |   createMachineBasicBlock(getTarget(DIS))
11 |   |   |   end
12 |   |   |   createMachineBasicBlock(MI.address + InstructionSize)
13 |   |   end
14 |   end
15 |   addToMachineBasicBlock(DIS)
16 end
17 for MF in MachineFunctions do
18 |   switchToFunction(MF)
19 |   createAllBasicBlocks(MF)
20 |   for MBB in MF.MachineBB do
21 |   |   switchToBasicBlock(MBB)
22 |   |   decompileInstruction(MBB)
23 |   end
24 end
```

4.3.1 Recovering Lost Information

During compilation, the LLVM backend strips function prototypes, local variables, and other type information from the LLVM IR code. Without this knowledge, it is non-trivial to generate a valid call graph for use with static analysis. In this section, we describe how the needed information can be reconstructed.

Intraprocedural Control Flow

Grouping statements into basic blocks helps to understand the control flow in a function. Branch instructions always indicate the exit point of one basic block and point to succeeding statements. Since they are referenced by their instruction offset in the binary, we can leverage this address to unambiguously find the entry points of the subsequent basic blocks. The immediate predecessors of entry points can be defined as exit instructions of basic blocks. As a result, we obtain an accurate control flow graph from basic blocks. Without the binary addresses, successors would not always be clearly visible.

Function Parameters and Return Values

Knowing about these definitions is essential for the data flow analysis. Since function prototypes are completely removed during compilation, we can only make assumptions based on the ARMv8 calling convention. A function parameter can be assumed if a value is read from a memory location where a parameter may have been stored previously. To find such locations, we traverse the control flow graph from the entry point of a function to the `load` instruction. A parameter is also assumed if no instruction is found that stores a value to a location.

To identify return values, the control flow graph is not traversed from the function entry point but from the `call` instruction to which the program counter will return after executing the function.

External Symbols

iOS applications include placeholders for symbols that refer to external libraries. At runtime, the placeholders are replaced with the correct associations from the loaded libraries. For the purpose of static analysis, we can imitate this behavior and store a reference to a symbol's name and its library association within the decompiled code.

4.3.2 Implementation

As the *Dagger* framework is only capable of translating x86 binaries to LLVM IR, we have extended it with the definition of instruction semantics for 64-bit ARMv8. By also considering specifics of iOS binaries, we succeeded in modeling the correct control flow and can complement the subsequent data flow analysis with accurate type information. In the following, we highlight some adaptations that were made to optimize further analysis.

Registers

Dagger relies on the LLVM register description of a CPU to create a data structure that simulates internal storage. However, in practice, registers of a 64-bit ARMv8 CPU overlap with the LLVM definitions and storing multiple sub registers in a single super register will lead to ambiguities when computing data dependencies. The largest super registers contained the values of four physical registers and each of them was stored in four different super registers. Thus, we modified Dagger's model such that a single super register may represent only the value of exactly one physical register. This means that three values of each of the largest super registers can be removed and no data is lost.

In case an instruction touches multiple physical registers by accessing using a super register, which now holds only the value of a single physical register, multiple super registers need to be combined into a single value that can then be used by translated instructions. Adapting Dagger's strategy for this problem still keeps the values stored in registers unchanged but facilitates to compute data dependencies of super registers.

Non-Volatile Registers

According to the 64-bit ARMv8 calling convention, the content of non-volatile or *callee-saved registers* must be preserved across function calls. Although the callee accesses these values only for storing and loading them, this operation causes data dependencies between the caller and callee. Since the values are not used for anything else, we can optimize the code by cutting such data dependencies.

Tail Calls

In ARMv8, a *branch-and-link* (`BL`) instruction is replaced with `branch` (`B`) if `call` is the last instruction of a function before returning to the caller. Since `branch` cannot leave the scope of a function in LLVM IR, Dagger replicates the target's code to the current function. This leads to a wrong call graph as code inlining prevents that an edge is added for the call. We resolve this issue by checking whether the target address of a branch instruction is outside the function body. In that case, we replace the tail call with a regular `call` and return statement.

4.4 Pointer Analysis

Pointer analysis is required to support the subsequent slicing step with information about pointers and to compute an accurate call graph, needed for data flow analysis. We rely on context- and flow-insensitive constraint generation with a focus on LLVM IR and the characteristics of Objective-C (and implicitly Swift).

4.4.1 Iterative Constraint Generation

The algorithm presented by Andersen [And94] generates constraints once for each instruction, which only works if all values are already provided within an instruction. However, in case of function pointers and calls to Objective-C methods, points-to knowledge is required before being able to process these calls. Therefore, we adapted the constraint generation process to handle call instructions with pointers in an iterative manner.

Focusing on the C language only, Andersen's analysis does not consider an important feature that occurs in decompiled code: All memory accesses are done by converting an integer to a pointer using the `inttoptr` instruction. Without considering these instructions, constraint generation would produce a points-to set with all pointers referencing an unknown location. We modify the rules for constraint generation to cover the `inttoptr` instruction. Based on the constraints by Hardekopf and Lin [HL07b] (see Table 2.1), we identify patterns in the decompiled code for all possible applications of `inttoptr`:

- **Binary:** offsets are accessed by `inttoptr` via static addresses. This is the simplest memory access since the operand of `inttoptr` includes the source address as a constant integer value.

- **Heap:** memory addresses are created via allocation functions, such as `malloc()` or `calloc()`. Since their return values describe an abstract location, it can immediately be used for pointer analysis.
- **Stack:** memory is usually accessed by adding a static offset to the stack or frame pointer. However, different instructions that access the same location on the stack are not easily identifiable since a separate pointer is created for each of them. Thus, we have to match stack locations with all variables which might contain the same value during execution. This is done by determining all operations that use the stack pointer in an `add` or `sub` operation with a constant integer. As these operations are mainly responsible to create the stack address for a subsequent use with `inttoptr`, we can match different pointers to the same stack location.

As shown in Algorithm 2, we extend the original algorithm by Anderson [And94], to distinguish between `call` instructions and others. After generating constraints for all regular statements (line 5), the subsequent loop solves the constraints by propagating points-to sets through the program. The gained knowledge is then used to update the constraints for `call` statements. This step is repeated until no more edges are added to the call graph.

Algorithm 2: Constraint generation

```
1  for I in Instructions do
2  |  if isCallInstruction(I) then
3  |  |  addInstruction(I, CallInstructions)
4  |  else
5  |  |  generateConstraints(I)
6  |  end
7  end
8  repeat
9  |  solveConstraints()
10 |  for I in CallInstructions do
11 |  |  updateConstraints(I)
12 |  end
13 until no new constraints added
```

4.4.2 Objective-C Peculiarities

As iOS applications are compiled from source code in the runtime-oriented languages Objective-C and Swift, the LLVM frontend rewrites direct method invocations to use a dynamic dispatcher function instead. `objc_msgSend()` is a function in the Objective-C runtime library, responsible to decide at runtime what method to call. Therefore, it takes two parameters that specify the class or object, and the name of the method to call. Both arguments, `X0` and `X1` in ARMv8, hold pointers to locations in the binary where the actual values are retrieved from. This also affects the call graph generation which naturally fails to add edges if neither type information nor selector names are defined in the LLVM IR code. We, thus have to restore the correct call graph by adding edges

for the methods referenced in calls to `objc_msgSend()`. Other language-specific peculiarities that need special processing include the Objective-C features *Blocks* and *Fast Enumeration*. For both cases, constraint generation has to be adapted to identify instructions that access locations on the stack.

Algorithm 3 describes how the original call graph is restored using the points-to sets of the class or object (PtsTo_a), and method name (PtsTo_b) parameters passed to `objc_msgSend()`. The algorithm iterates over the possible values of the $X0$ and $X1$ parameters and based on the locations they point to, it determines the method that can be called. The $X0$ parameter has to point either at a location where class infos are stored or to a dynamically allocated location. If its value points to class infos, the corresponding class method will be called. Otherwise, if it points to a dynamically allocated memory location and is annotated with information about an object type, an instance method is invoked. The overall result is a call graph that also models all calls that are performed at runtime via `objc_msgSend()`.

Algorithm 3: Call graph reconstruction using points-to sets

Input: CallInstruction, PtsTo_a, PtsTo_b

```
1  for loc_a in PtsTo_a do
2  |  ClassMethod ← false
3  |  if PointsToClassInfo(loc_a) then
4  |  |  ClassMethod ← true
5  |  end
6  |  Type ← GetTypeName(loc_a)
7  |  for loc_b in PtsTo_b do
8  |  |  if not PointsToClassInfo(loc_a) then
9  |  |  |  continue
10 |  |  end
11 |  |  Selector ← GetSelectorName(loc_b)
12 |  |  if not KnownMethod(Type, Selector, ClassMethod) then
13 |  |  |  continue
14 |  |  end
15 |  |  if HasEdge(CallInstruction, Type, Selector, ClassMethod) then
16 |  |  |  continue
17 |  |  end
18 |  |  AddEdge(CallInstruction, Type, Selector, ClassMethod)
19 |  end
20 end
```

4.5 Static Slicing

The purpose of static slicing is to compute all code segments that affect a slicing criterion. We adopt the algorithm of Weiser [Wei81] to work with LLVM IR code by considering characteristics of ARMv8.

As parameters can be passed using both the stack or registers on ARMv8, Weiser's approach does not immediately work for decompiled LLVM IR code: Each function has exactly one formal parameter that represents the register set.

Since this set is the same for all functions, no parameters could be substituted to generate a new set of slicing criteria when a function is called. Our solution for this issue is to extend the collection of all relevant variables $ROUT(i)$ with information about `load` and `store` operations that read or modify registers before a function call.

Let $S(v, r)$ be a `store` instruction that saves a variable v to the register r and $L(v, r)$ the corresponding `load` instruction. We now define $STORE(i)$ to return the set of preceding variables that were stored to a register before the instruction i and $LOAD(i)$ to return the set of first variables, loaded from a register after the instruction i:

$$STORE(i) = \Big\{(v,r) \mid \exists\, S_i(v,r) \rightarrow_{CFG}^{*} i, \nexists\, L_i(v,r) \rightarrow_{CFG}^{*} i\ S_j(v',r) \rightarrow_{CFG}^{*} i\Big\} \tag{4.1}$$

$$LOAD(i) = \Big\{(v,r) \mid \exists\, i \rightarrow_{CFG}^{*} i\ L_i(v,r), \nexists\, i \rightarrow_{CFG}^{*} L_j(v',r')\ {}_{CFG}^{*}\!\rightarrow L_i(v,r)\Big\} \tag{4.2}$$

Using these sets, parameter substitution from formal to actual parameters is possible even if all functions use the same register set. If the stack is used for parameter passing between functions, inter-procedural slicing is achieved using the points-to analysis. If a stack parameter is a relevant variable, the corresponding memory location will be the same in the caller and callee functions.

4.5.1 Restoring Missing Type Information

A conclusive data flow analysis requires knowledge about all object types. After recovering function parameters and return values in the decompilation step, type information is still missing for instance variables and protocol methods. If available in the code of the binary, the type definitions can easily be resolved (see Section 2.2.2). However, if instance variables are allocated by a function of an external library, finding their type information can be a challenging task.

Instance variables are always accessed by loading an offset from an individual static address in the binary. This enables us to find all instructions that refer to a particular variable and a single abstract location [SH97a] can be specified. Similarly, the binary has no precise type information for parameters of methods that are declared by a protocol in Objective-C or Swift. Although creating abstract locations for each parameter does not allow for a precise pointer analysis, it is still possible to identify calls made using these objects.

4.5.2 Parameter Backtracking

Program slices summarize all relevant statements and variables that influence a certain parameter. This information can now be split into single execution paths. In the following, we explain our solution to find predecessors, isolate execution paths, and how to avoid cycles while backtracking.

Finding Predecessors

The static single assignment form (SSA) of LLVM IR already represents a effective way to backtrack program statements that do not use pointers for memory access. However, so far if a value is read from a memory location referenced by a pointer, it is not feasible to identify the preceding store instruction by inspecting only the current statement. For program slicing, it is necessary to add the memory location to the relevant set and traverse the control flow graph backwards to find a modification of this location. Overall, for backtracking we need to specify more parameters than just the relevant location, as explained in the following.

In case a memory location l is added to the set of relevant variables, we also add the statement s, which induced this location, to a separate set $R_{Sources}(i, l)$. It includes all statements that were added by a relevant variable l at statement i and allows to track value changes with according read instructions. In the following, we show the formal definition of these sets that apply to each instruction and the approach how sources of relevant variables are propagated through a program. Analogous to the $LOAD(i)$ and $STORE(i)$ definitions that were used with slicing, instruction j is an immediate successor of instruction i in the CFG ($i \rightarrow_{CFG} j$):

$$\begin{aligned} R_{Sources(i,l)} = &\{i \mid i \in S_C,\ l \in REF(i)\} \\ &\cup \{i \mid i \in R_{Sources}(j, l),\ i \notin S_C\} \end{aligned} \tag{4.3}$$

Leveraging this information, it is feasible to find the preceding changes of a memory location, in case a value is accessed using a pointer. Assuming the statement r reads a value from a memory location l, the according set of predecessors $Pred'(r)$ for backtracking this value is defined as:

$$Pred'(r) = \{s \mid r \in R_{Sources}(s, l),\ l \in DEF(s)\} \tag{4.4}$$

The set of predecessors $Pred(i)$ that contains all program statements modifying a referenced variable can then be written as:

$$\begin{aligned} Pred(i) = &\{s \mid r \in R_{Sources}(s, l),\ l \in DEF(s),\ l \in Loc\} \\ &\cup \{op \mid op \in Operators(i),\ op \in Instructions\} \end{aligned} \tag{4.5}$$

Extracting Execution Paths

The set of predecessors $Pred(i)$ comprises all statements before instruction i. Any possible execution path of a value v back to its initial definition can be formulated as a graph $G = (V, E)$. While V represents vertices with all program statements, the edges E can be found by recursively adding all reachable predecessor instructions:

$$\begin{aligned} E^{(0)} &= \{(v, j) \mid j \in Pred(v)\} \\ E^{(k)} &= \{(i,\ j) \mid (j,\ k) \in E^{(k-1)},\ i \in Pred(j)\} \end{aligned} \tag{4.6}$$

Avoiding Cycles

If program statements are run within a loop, execution paths resemble cyclic dependencies. To avoid infinite loops during analysis, we discard the branch of a path immediately as soon as an instruction is found that is already included in the path. Consequently, execution branches without cyclic dependencies remain being tracked backwards and lead to the initial definition of a variable.

4.5.3 Implementation

We implemented a solution for program slicing and parameter backtracking on LLVM IR code that considers language peculiarities of iOS applications. The slicing step is required to integrate with points-to information, and the subsequent backtracking step has to support the call graph restored during pointer analysis.

We build on *LLVMSlicer*[9], an implementation of Weiser's algorithm for static slicing, which also includes Andersen's algorithm for pointer analysis. While this works fine for smaller programs, it leads to a poor performance for iOS applications that usually consist of a very large code base. We tackle this issue by replacing the implementation with constraint optimization techniques[10] proposed by Hardekopf and Lin [HL07a]. Taking up this idea, we look for pointers that have an equivalent points-to set and merge their representations. Constraint solving is optimized by detecting and collapsing strongly connected components in the constraint graph [HL07b]. We further augment the slicer with a functionality for parameter backtracking in order to extract single execution paths. As demonstrated in Section 5, this feature enables us to verify whether data flows meet certain predefined inspection criteria.

4.6 Conclusion

Thoroughly analyzing data flows in applications for the iOS platform is challenging due to low-level machine code, the use of a pointer-aware language, dynamic control-flow decisions, and missing data types. To perform a conclusive inspection of implementation security it is crucial to address these challenges.

We developed a multi-step approach to facilitate the static analysis of iOS applications. Instead of inspecting the low-level representation of an executable, we proposed a solution for a generically applicable decompiler that translates 64-bit ARMv8 binaries to LLVM IR code. By reconstructing data types from the binary, resolving indirect method calls, and performing a pointer analysis, we are able to precisely model control and data flow graphs for use with program slicing and parameter backtracking. After extracting and evaluating individual execution traces regarding predefined criteria, it is possible to draw conclusions about security-related properties in iOS applications.

[9] https://github.com/jirislaby/LLVMSlicer
[10] https://github.com/grievejia/andersen

5

Misapplied Crypto in Android and iOS Applications

In this chapter, we study the wrong application of platform-provided crypto APIs. We motivate our analysis in Section 5.1 and present a concept for automatically uncovering weak security parameters in Section 5.2. In Section 5.3, we formulate low-level properties to find misapplied APIs and evaluate them in two case studies: (1) We inspect the prevalence of crypto misuse on iOS in Section 5.4 and disclose that 343 out of 417 apps (82%) are subject to at least one security misconception. (2) In Section 5.5, we focus on differences between Android and iOS concerning the proper usage of platform-specific APIs for cryptography. We find that out of 775 investigated applications that vendors distribute for both operating systems, 604 apps for iOS (78%) and 538 apps for Android (69%) violate at least one basic security principle. Parts of this chapter are taken verbatim from [Fei19; FMS18].

Publication Data and Contribution

Johannes Feichtner. "A Comparative Study of Misapplied Crypto in Android and iOS Applications." In: *Security and Cryptography – SECRYPT 2019*. SciTePress, 2019, pp. 96–108. DOI: 10.5220/0007915300960108

Johannes Feichtner, David Missmann, and Raphael Spreitzer. "Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications." In: *Security & Privacy in Wireless and Mobile Networks – WiSec'18*. **Best Paper Award**. ACM, 2018, pp. 236–247. DOI: 10.1145/3212480.3212487

Contribution: Main author; Initial prototype implemented by David Missmann. Raphael Spreitzer contributed to the paper introduction.

5.1 Introduction

Many applications for Android and iOS process sensitive and private user data. A common method to protect this information is the use of security APIs and cryptographic functionality, provided by the platform. For this to be effective, essential rules need to be obeyed, as otherwise the attainable level of security would be weakened or, in the worst case, entirely defeated. Among these integral principles are, for example, the need to preserve confidentiality of encrypted data by not using electronic code book (ECB) mode for block ciphers, the need to prevent static keys or credentials, which are in the worst-case hard-coded into the application, or the need to avoid static seeds for generating random numbers.

An evaluation of reports in Common Vulnerabilities and Exposures (CVE) revealed that 83% of all findings related to cryptography in mobile applications were due to the wrong usage of security-related system APIs [Laz+14]. Focusing on the usability of different crypto libraries, other studies [Aca+17; Nad+16] conclude that, aside from too complex interfaces and insecure defaults, typical reasons for faulty implementations are often to be found in poor documentation, missing code samples, and further features required by the library to work securely. Irrespective of whether an erroneous implementation is a result of a developer's ignorance, lack of knowledge, or a too complex documentation of the crypto API, the identification of misapplied crypto APIs is of utmost importance to protect sensitive data and to provide the intended functionality, e.g., in case of password managers or secure messengers. As developers usually do not provide detailed information and the source code of mobile applications is not made available, the correct implementation of cryptographic functionality can only be verified by reverse-engineering the final application.

While analyzing 11,748 Android applications in 2013, Egele et al. specified six common types of mistakes in using cryptographic APIs and confirmed that issues were present in 88% of all inspected apps [Ege+13]. Based upon the reverse-engineering framework Androguard[1], the proposed analysis solution first derived a control flow graph over all functions and subsequently performed static program slicing on Dalvik bytecode to inspect the parameters passed to cryptographic operations. Inspired by this approach, Shao et al. [Sha+14a] compiled a set of cryptography-related Java APIs where implementation flaws would potentially have a severe impact. Besides incorporating the rules of Egele et al., the authors also suggested to consider further security-relevant rules targeting certificate handling and key management. Since then, approaches have been elaborated to better detect [MBB18] and mitigate [Ma+16a] the wrong use of crypto APIs on Android. To inspect whether iOS applications are subject to similar weaknesses, Li et al. [Li+14] proposed a concept for dynamic analysis, aimed at identifying weak security properties in execution traces captured during runtime. As the results of existing work suggest that the correct usage of system-provided APIs for cryptography-related purposes is rather the exception than the norm, it stresses the need for further research to reliably disclose vulnerable implementations.

[1] https://github.com/androguard/androguard

Existing research focuses on highlighting the widespread of crypto misuse rather than pinpointing the exact origin of problematic statements in code. While there is undoubtedly a need for tools to warn about these kinds of security problems, so far they are unable to give clear advice to developers and analysts about the origin of rule-violating code. Being able to directly compare execution traces of API invocations is important in order to see why methods are invoked with cryptographically weak values. With regards to the platform, execution traces can also provide a valuable insight into why some rules are significantly more violated than others. To assess and compare the occurrence of misapplied crypto APIs in Android and iOS applications, we pursue a three-stage process:

1. Streamline the security-critical application analysis for both platforms by establishing a workflow and output format that ensures inspection results on Android and iOS are comparable.

2. Based on a general set of basic cryptographic rules not to violate, we specify concrete detection strategies that hold on both platforms. Due to the heterogeneous APIs on Android and iOS, individual constraints apply regarding parameters that cryptographic methods can take.

3. After incorporating the rules into our approaches for static analysis on Android, as introduced in Chapter 3, and iOS, as presented in Chapter 4, we perform two case studies on misapplied crypto APIs using two distinct sets of mobile applications that rely on cryptography to fulfill their purpose.

Case Study on iOS Applications

Multiple studies have demonstrated that security-critical vulnerabilities due to the wrong application of crypto APIs are a common problem on Android. Although one would expect that the situation is similar on the iOS platform, an equivalent analysis on iOS applications has not been performed so far. Low-level machine code, dynamic control flow decisions during runtime, and the use of a pointer-aware language complicate a security-critical inspection of binaries for the iOS platform. By addressing these challenges in our concept for static analysis on iOS (see Chapter 4), we pave the way for an automated inspection of iOS applications regarding misapplied crypto APIs.

Based on the platform-provided *CommonCrypto* library (see Section 2.4), we specify and implement security rules for the use of cryptographic APIs, and automatically test several hundred closed-source apps regarding their compliance to essential security principles. As a result, we are not only able to confirm the existence of problematic statements in iOS applications but can also pinpoint their origin in code. Overall, we make the the following key contributions:

1. We formulate the low-level properties required to identify misapplied crypto APIs on iOS. When used within the slicing process, we can verify whether execution paths meet or violate a certain conditions predefined by rules. Besides this use case, the rules might also assist developers in identifying security-critical code within their applications.

2. Using open-source iOS applications, we iteratively refine the implementation of our static analysis solution to reduce the number of false positives.

3. We apply our framework on more than 400 iOS applications that rely on cryptographic APIs and identify security-critical misconceptions in the majority of these applications.

We evaluated the practical viability of our solution by analyzing real-world iOS applications. A manual security analysis confirmed the applicability of the pursued approach and helped to identify possible limitations. With automated testing, we detected that 82% or 343 out of 417 applications with cryptographic functionality violate at least one security rule. This result shows that misconceived usage of security-related APIs is a very prevalent issue in iOS applications. Among the most common flaws are the usage of non-random initialization vectors and constant encryption keys as input to cryptographic primitives.

Case Study on Applications for Both Platforms

Available research of misapplied crypto in mobile applications usually does not cross the line between platforms and so it is still unclear whether apps that are available for both Android and iOS suffer from similar implementation weaknesses. Consequently, it is hard to argue whether API misuse is the result of a developer's lack of knowledge or ignorance, or more likely due to problematic design decisions in the system-provided API and its too complex documentation. Our objectives and key contributions are as follows:

1. We derive concrete detection strategies for general security rules and assess how they can be violated on each platform. Used in combination with analysis frameworks for Android and iOS, problems can reliably be identified based on given data flows and the inspection results from both platforms are made comparable.

2. We present the first comparative evaluation of misapplied crypto in Android and iOS apps. Applied on a set of 775 apps for iOS and their counterparts for Android, we assess the spread of common mistakes in apps for both platforms. We ask if developers know how to use the system-provided APIs correctly and check if apps that are considered to be secure on one platform, typically fulfill the same expectations on the other.

We manually collected a set of 775 mobile applications that apparently originate from vendors that distribute individual versions of the same apps in the official stores of Android and iOS. All of them had at least 1,000 installations or ratings on each platform, and their use of cryptography seemed obvious, such as with password managers and secure messengers. We disclosed that 78% or 604 apps for iOS and 69% or 538 apps for Android violated at least one basic security rule. The analysis also showed strong indications that the prevalence of some issues is almost solely based on design decisions of the system API.

5.2 Analysis Workflow

In previous chapters, we presented static analysis approaches for a security-critical program inspection of Android and iOS applications. For both platforms, we designed and implemented modular frameworks that offer the ability to automatically extract, disassemble and investigate programs. Our concepts also integrate definable slicing patterns that can be applied to identify and highlight the improper usage of security-relevant functionality. By applying static backtracking techniques, the control and data flow of relevant code segments is opened up and serves as input for further evaluation. Therefore, our solutions incorporate essential rules aimed at disclosing common implementation weaknesses and to make reliable assumptions about the degree of security, security-related code fragments are able to provide. Besides analyzing these constructions, it is determined whether they deviate from being employed correctly. In addition to an automated inspection, our workflows for program inspection on both platforms also support the manual analysis of mobile applications by delivering well-arranged graphs, representing the static slices of user-definable patterns.

In contrast to approaches, such as CryptoLint [Ege+13] or CMA [Sha+14a], we can give clear advice about the origin of rule-violating code, rather than being only able to confirm its existence. In experiments, we also noticed that existing work tend to flag all apps that violate any rules as insecure. While this is true in the strict sense, it disregards the practical impact of rule violations: mobile applications might use cryptography APIs also for other purposes than encryption. Without looking at the actual execution paths, it might go unnoticed if encryption is only employed for obfuscation. A similar case occurs if, e.g., libraries used in apps initialize variables holding encryption keys, salt values, or IVs with insecure default values but overwrite them before actual usage. As a consequence, we do not draw a binary conclusion on the (in)security of applications but provide reports with *potentially problematic* execution paths, as identified by the conditions in our security rules. The overall analysis strategy, as pursued in the subsequent case studies, can be summarized as follows:

1. **Preprocessing:** We translate the Dalvik bytecode of Android applications to Smali code and parse the resulting files to an object representation. With iOS applications, we extract the ARMv8 64-bit binary executable, disassemble machine code, and decompile it into LLVM IR code.

2. **Static Slicing:** Based on abstract slicing patterns that are defined within our security rules, we derive concrete slicing criteria. We perform static backtracking, follow all possible execution paths that affect a certain slicing criterion, and organize the resulting data flows in a graph representation that enables further analysis.

3. **Security Rule Evaluation:** Using a breadth-first search on slice trees, we traverse individual execution paths and check whether they comply with our predefined security rules.

5.3 Evaluating Security Properties

Egele et al. [Ege+13] described six general rules in cryptography that have to be considered by Android developers in order to avoid security issues, and they also pointed out possible security implications. In our work, we conceptually adopt these rules and elaborate platform-specific detection strategies for each of them.

We investigate whether applications for Android and iOS that use system-provided APIs for cryptography achieve a cryptographic notion of IND-CPA security. *Indistinguishability under a chosen plaintext attack* or IND-CPA states that attackers are unable to extract even a single bit of plaintext from a ciphertext within a certain amount of time. Encryption can be considered secure if it is IND-CPA secure [KL14; Bel+97]. In practice, faulty implementations and wrongly-chosen parameters, e.g., the use of AES in ECB mode, using CBC mode with a non-random IV, or a hard-coded encryption key, thwart IND-CPA security.

To ensure that platform-provided crypto APIs are only used with parameters that guarantee strong security and do not violate the IND-CPA property, we formulate individual evaluation patterns for Android and iOS. For Android, we rely on API methods exposed by the *Java Cryptographic Architecture*. For iOS, in turn, we define analogous rules based on the anatomy of *CommonCrypto* methods. We consider custom crypto implementations out of scope, as cryptography should not be implemented by developers themselves anyways.

Rule 1: Do not use ECB mode for encryption. In ECB mode, data blocks are enciphered independently from each other and cause identical message blocks to be transformed into identical ciphertext blocks. Consequently, data patterns are not hidden and confidentiality may be compromised.

On Android, applications can request an instance of a particular cipher by passing a suitable *transformation* value as a parameter to `Cipher->getInstance()`. Typically, this value is composed of the algorithm, an operation mode, and the padding scheme to use. E.g., to request an instance providing AES in ECB mode with PKCS#5 padding, `AES/ECB/PKCS5Padding` has to be specified. While it is indispensable to declare an algorithm, explicitly setting mode and padding may be omitted. In that case, the underlying provider will implicitly assume ECB mode. Besides AES, this affects all symmetric block ciphers.

1. Determine all invocations of the method `Cipher->getInstance()` and for each occurrence, backtrack the first parameter, containing the cipher transformation value.
2. Find all possible execution paths, where the endpoint resembles a transformation or specifies a symmetric block cipher, such as AES or DES.
3. For each selected path, verify whether it includes parts of a transformation value, e.g. `/OFB/NoPadding`. If found, complete the algorithm name in the path endpoint with the determined mode and padding descriptor.
4. Raise an alert if the transformation value either explicitly declares ECB mode, or specifies only the algorithm name.

On iOS, by default CBC is preferred over ECB mode. ECB mode is only used when the developer explicitly specifies to use this mode of operation.

1. For each invocation of `CCCryptorCreateWithMode()`, `CCCryptorCreate()`, or `CCCrypt()`, backtrack the third parameter, `options`, specifying whether to use ECB or CBC mode.

2. Raise an alert if any of the possible execution paths ends with a constant value of 2, i.e., `kCCOptionECBMode`, which would indicate ECB mode.

Rule 2: No non-random IVs for CBC encryption. Initialization vectors (IVs) that are constant or predictable lead to a deterministic and stateless encryption scheme susceptible to chosen-plaintext attacks. If CBC mode is selected for encryption, developers should provide a cryptographically secure, non-random IV. If no IV is specified at all, the cipher uses an all-zeros IV, which is at least as bad as a constant value.

By analyzing byte arrays set as IVs, we learn if IVs are composed of static values or deduced from constants, such as hard-coded strings or assembled arrays. IVs can also be predictable when weak pseudo-random number generators (PRNG) are employed. Besides probing for specific indicators, we are naturally unable to make assumptions about the predictability of values. Similarly, non-static IVs cannot implicitly be assumed unpredictable.

On Android, to specify an IV for encryption, an `AlgorithmParameterSpec` object is typically passed as an argument to `Cipher->init()`. If it encapsulates an object of the type `IvParameterSpec`, an initialization vector is manually defined rather than being generated randomly.

1. For all invocations of `Cipher->init()` with an `AlgorithmParameterSpec` object as second argument, backtrack the value of this parameter.

2. Using the found list of constants, verify whether an object of the type `IvParameterSpec` is created by calling its constructor. Abort, if none is found. This implies that no IV is set explicitly for this `Cipher` instance.

3. From each available slicing path, extract the subpath that begins at the `iv` argument, passed to the constructor of the `IvParameterSpec` object.

4. Raise an alert if the `iv` parameter is derived from a statically defined byte array, a string, e.g., via `String->getBytes()`, or by calling the cryptographically insecure `Random` API.

On iOS, we assert that an IV is indeed generated using a cryptographically secure pseudorandom number generator (PRNG).

1. For each invocation of `CCCryptorCreateWithMode()`, `CCCryptorCreate()`, or `CCCrypt()`, we backtrack the 6th parameter, specifying the IV.

2. Alert if a possible execution path does not call `CCRandomGenerateBytes()` in *CommonCrypto* or `SecRandomCopyBytes` in the *Security* library.

Rule 3: Do not use constant encryption keys. Keeping encryption keys secret is a vital requirement to prevent unrelated parties from accessing confidential data. Statically defined keys clearly violate this basic principle and render encryption useless. Therefore, the security rule aims to detect hard-coded keys and raises an alert if a constant key is employed for symmetric encryption.

On Android, `SecretKeySpec` can be used to specify a key. If derived from a constant, it must not be used for symmetric encryption. However, with public-key crypto, the encryption key is not a secret and can also be wrapped as a `SecretKeySpec` object. Thus, we distinguish between keys for symmetric and asymmetric encryption and also check the second parameter of `SecretKeySpec`, which specifies the algorithm to use.

1. For all invocations of the method `SecretKeySpec->init()`, backtrack the first parameter, holding the key.
2. Verify for all contained constants if the `key` parameter is derived from a statically defined byte array, or a string, e.g., using `String->getBytes()`.
3. If at least one possible key has been found, also backtrack the 2nd parameter of the corresponding `SecretKeySpec` object and extract all data flows.
4. For each execution path, verify whether one of the following asymmetric encryption schemes is specified: *DHIES, ECIES, ElGamal, RSA*. If the algorithm is not a known public-key algorithm, we conclude that the statically defined key is used for symmetric encryption and raise an alert.

On iOS, we assure that any byte array specified as key does not exclusively consist of constant values.

1. For each invocation of `CCCryptorCreateWithMode()`, `CCCryptorCreate()`, or `CCCrypt()`, we backtrack the 4th parameter, holding the key.
2. For any path, we ensure that key elements do not originate from a constant or hard-coded source.

Rule 4: Do not use constant salts for PBE. A randomly chosen salt value ensures that a password-based key is unique and slows down brute-force and dictionary attacks. Consequently, salts passed to key derivation functions (KDF) must not exclusively depend on constant values.

On Android, using the `PBEKeySpec` API parameters to use with password-based encryption (PBE) can be declared. A `SecretKeyFactory` instance then transforms the password to an encryption key by invoking `generateSecret()`.

1. Find all calls to `PBEKeySpec->init()` or `PBEParameterSpec->init()` and backtrack the parameter with the `salt` value.
2. Raise an alert if any execution path providing the `salt` parameter, is derived from a statically defined byte array, a string, e.g., via `String->getBytes()`, or by calling the cryptographically insecure `Random` API.

On iOS, `CCKeyDerivationPBKDF()` must not be provided with a salt from an entirely constant source.

1. For all calls to `CCKeyDerivationPBKDF()`, backtrack the 4th parameter, specifying the `salt` value.
2. For any execution path, we ensure that byte arrays with the salt originate from a non-constant origin.

Rule 5: Do not use $<$ 1,000 iterations for PBE. A low iteration count significantly reduces the costs and computational complexity of table-based attacks on password-derived keys. We expect applications to use $\geq$ 1,000 rounds in KDFs, as recommended by RFC 8018 [MKR17].

On Android, the iteration count for PBE is declared by using the `PBEKeySpec` or `PBEParameterSpec` API.

1. Find all calls to `PBEKeySpec->init()` or `PBEParameterSpec->init()` and backtrack the parameter with the `iteration count` value.
2. Raise an alert if any execution path terminates at a constant integer whose value is less than 1,000.

On iOS, the `rounds` parameter of the method `CCKeyDerivationPBKDF()` specifies the amount of iterations to use for key derivation.

1. For all calls to `CCKeyDerivationPBKDF()`, backtrack the 7th parameter with the `iteration` count.
2. For any execution path, we raise an alert if it does not end at a constant integer value $\geq$ 1,000.

Rule 6: Do not use static seeds for random-number generation. If a PRNG is seeded with a statically defined value, it will produce a deterministic output that is not suited for use with security-critical applications.

With Android 4.2, API level 16, the default PRNG provider has been changed from Apache Harmony to the native `AndroidOpenSSL`. Before that, it was possible to override the internally designated seed with a custom value which, in case it was constant, caused the generation of deterministic output values.

1. For all invocations of the method `SecureRandom->init()`, backtrack the first parameter, holding a byte array with the seed. Likewise, for all calls to `SecureRandom->setSeed()` compute slices for the `seed` argument, which may consist of a byte array or eight bytes stored in a `long` integer value.
2. For all found execution paths, check if any of them supplies the `seed` parameter with constant input values.
3. An alert is raised if the parameter is derived from a statically defined byte array, a string, e.g. via `String->getBytes()`, or a 64-bits integer value.

On iOS, this rule cannot be violated as APIs do not support seeding the PRNG.

5.3.1 Implementation

Each security rule is implemented individually for Android and iOS using JSON definitions that allow for easy extensibility. In the following, we explain how the previously presented rules can be formalized and provide two example definitions.

Each rule consists of a `name`, at least one `criterion` that serves as a starting point for static slicing, and a set of `conditions` that define the requirements each execution trace of a parameter has to fulfill. Each condition specifies either the type `OK` – a *positive test* to ensure that some call occurs in the trace, e.g., a call to a secure PRNG, or `NOK` – a *negative test* to check for some value that would violate the rule, e.g., the occurrence of a constant byte array or string value. The condition value itself depends on what we expect: either we want an execution path to end with a specific variable, e.g., a constant integer having a certain value, or we evaluate whether the call graph includes some function call.

Each rule can consist of multiple subgroups that are organized in additional JSON objects. Instead of listing a set of criteria, it is possible to reference another group. This is useful if a criterion does not only depend on the presence of one function call but a nested call hierarchy, which can then recursively be checked.

Listing 5.1 exemplifies a security rule implementation to check the number of iterations for password-based encryption on Android. Multiple criteria serve as a starting point for slicing. The `name` attribute indicates a class and method, with `<init>` representing the constructor of methods in Smali code. The `parameter` value refers to the index of the method argument, starting with zero. In this example, we want to track the third parameter passed to the constructor of `PBEKeySpec`, as it holds the iteration count. Besides considering PBE-related methods, the Listing also demonstrates how rules can be extended with additional criteria to cover further method signatures, such as the PBKDF2 implementation of a third-party crypto library. The security rule is satisfied if each obtained execution trace ends with a constant integer having a value greater than 1,000. An alert will be raised if this condition is not fulfilled, including the according execution trace as a reference for manual analysis.

Listing 5.1: JSON rule to evaluate the number of iterations for PBE on Android.

```
{
  "name": "Rule 5: Minimum iteration count >= 1,000",
  "criterion": [
    { "name": "javax/crypto/spec/PBEKeySpec-><init>", "parameter": "2" },
    { "name": "javax/crypto/spec/PBEParameterSpec-><init>", "parameter": "1" },
    { "name": "org/spongycastle/crypto/generators/PKCS5S2ParametersGenerator->init",
      "parameter": "2" },
    { "name": "org/spongycastle/crypto/generators/PKCS5S1ParametersGenerator->init",
      "parameter": "2" },
    { "name": "org/spongycastle/crypto/generators/PKCS12ParametersGenerator->init", "parameter":
      "2" }
  ],
  "conditions": [
    { "type": "OK", "conditionType": "ConstInt", "greater": 1000 }
  ]
}
```

Listing 5.2 presents the JSON implementation of a security rule to evaluate the use of non-random IVs for encryption on iOS. In contrast to the previously shown example, the criterion is not specified directly but referenced via a subgroup. The rule requires the IV to be random for all calls to functions defined in the `Cipher-IV` group. It, furthermore, illustrates how to define multiple conditions to be fulfilled within a single rule: (1) As a precondition, the main group requires that a call is made to a function defined in `Cipher-Operation`, where parameter `X0` is set to the constant integer value 0. This implies a symmetric encryption operation and can be verified by backtracking from a `Cipher-IV` criterion to a `Cipher-Operation` call. (2) The rule also requires that execution traces for any IV parameter, as identified by the functions and parameters listed in the subgroup `Cipher-IV`, contain a call to a secure PRNG. This condition is satisfied if an execution trace includes a call either to `CCRandomGenerateBytes()` from the *CommonCrypto* library or `SecRandomCopyBytes()` from the *Security* library. If no call is found to any of these methods, a rule violation will be reported, including the output of execution traces not conforming to these conditions.

Listing 5.2: JSON rule to evaluate the IV used for encryption on iOS.

```
{
  "name": "Rule 2: No non-random IV for encryption",
  "criterion": "Cipher-IV",
  "conditions": [
    {
      "type": "PRE",
      "criterion": "Cipher-Operation",
      "name": "Encrypt",
      "conditions": [
        { "type": "OK", "conditionType": "ConstInt", "equal": 0 }
      ]
    },
    { "type": "OK", "calls": "SecureRandom" }
  ]
},
{
  "name": "Cipher-IV",
  "calls" : [
    { "name": "CCCryptorCreate", "parameter": "X5" },
    { "name": "CCCryptorCreateWithMode", "parameter": "X4" },
    { "name": "CCCrypt", "parameter": "X5" }
  ]
},
{
  "name": "Cipher-Operation",
  "calls" : [
    { "name": "CCCryptorCreate", "parameter": "X0" },
    { "name": "CCCryptorCreateWithMode", "parameter": "X0" },
    { "name": "CCCrypt", "parameter": "X0" }
  ]
},
{
  "name": "SecureRandom",
  "calls" : [
    { "name" : "CCRandomGenerateBytes", "parameter": "X0" }
    { "name" : "SecRandomCopyBytes", "parameter": "X0" }
  ]
}
```

5.4 Case Study 1 - iOS Applications

The goal of this study is twofold. First, by manually investigating the output of our solution for static analysis on iOS (see Chapter 4) with the source code of real-world applications, we identify and fix possible weaknesses in our approach. Second, applying our tool on a large number of closed-source applications, we strive to uncover security-critical misconceptions.

5.4.1 Method and Dataset

We conducted both a manual and an automated analysis. In the following, we introduce the methodology applied, point out the pursued goals, and explain what applications were analyzed.

Manual Analysis

The objective of this step was primarily to test whether all components interact appropriately with each other and to refine the implementation where needed. Therefore, we applied the framework to open-source applications and checked the obtained results against the source code. Besides identifying opportunities for improvement, we also benefited from seeing what our security rules were able to (not) cover in a real-world scenario.

For this analysis, we downloaded 15 open-source applications from GitHub that use *CommonCrypto* for encryption. 8 of them were password managers, intended to protect user-entered credentials by means of cryptography. The remaining applications belonged to different categories, aimed at providing secure e-mail, data container, cloud storage, or messenger functionality.

We supplied the applications to the framework and analyzed them with respect to the defined security rules. We manually observed the analysis and improved the framework, e.g., by supplementing new *instruction semantics* that were not covered by the decompiler yet. After checking all security rules, the framework generated a report including all paths to statements that modified a specific parameter. We then verified and iteratively refined the accuracy of the analysis by studying the applications' source code. To facilitate this process, we leveraged a utility in Apple's IDE Xcode that enables the generation of call hierarchies for selected functions. A hierarchy is basically a subgraph of the call graph, containing only the nodes from which the targeted function was reached.

Automated Analysis

The primary goal of this step was to investigate whether iOS application developers know how to apply cryptographic APIs correctly. Inspired by the work of Egele et al. [Ege+13], we performed a similar empirical study for iOS applications.

We downloaded 634 free applications from the official iOS App Store and focused on programs where the use of cryptography seemed indispensable, e.g., password managers, (secure) messengers, cloud storage, and data containers, each

with more than 10,000 installations. Due to the fact that most were closed-source, we had to rely on the developer-provided descriptions to select those which might employ cryptography. After fetching them via iTunes, we used the tool *Clutch*[2] on a jail-broken iPhone to decrypt the applications. It turned out that 78% or 495 of the crawled applications included calls to the cryptographic API and were relevant for further analysis. Interestingly, the remaining set of 139 applications without *CommonCrypto* also included password managers and applications where the use of cryptography seemed appropriate. Aside from a faulty or absent implementation, this could be caused by the use of third-party libraries that implement crypto routines themselves.

After extracting all downloaded applications, we checked whether their library bindings indicate the usage of *CommonCrypto*. If this condition was fulfilled, we sequentially supplied the ARMv8 binaries to our framework. By inspecting the generated output reports, we ensured that any claimed security rule violation was indeed the consequence of a problematic execution path.

False Positives and False Negatives. In general, all static analysis approach suffer from a trade-off between minimizing false negatives and, worst-case, false positive rates of 100%. We avoid false negatives as our tool cannot miss calls to *CommonCrypto* API methods. We verified all identified problems found in open-source applications and iteratively refined the framework to eliminate false positives. In contrast, due to the nature of closed-source applications, we cannot formally exclude the existence of false positives. Manually studying execution traces of tested applications enabled us to also consider false positives that could have occurred, e.g., if a backtracked value was used a parameter to a function that had never been invoked (dead code).

5.4.2 Results

We evaluated 495 closed-source and 15 open-source applications that included calls to *CommonCrypto*. Iteratively refining the components during manual analysis ensured that all open-source applications could be decompiled and inspected. Afterwards, for a total of $417 + 15 = 432$ applications, the analysis workflow terminated successfully. As summarized in Table 5.1, the inspection of the remaining 78 closed-source applications failed due to one of three reasons. First, 7 iOS applications contained only binaries for the ARMv7 platform, which are considered as deprecated by Apple and are not supported by our decompiler. Second, 9% or 46 applications could not be decompiled due to missing *instruction semantics*. Third, for 25 applications constraint solving ran out of memory.

Automated Analysis

Of 417 successfully inspected closed-source applications, we found that 82% or 343 applications violate at least one rule. Table 5.2 summarizes our observations of violated security rules. In the following, we discuss the findings in more detail.

[2] https://github.com/KJCracks/Clutch

Table 5.1: Reliability for closed-source applications.

	Count	[%]
Downloaded from iOS App Store	634	
No *CommonCrypto* calls	139	22%
With *CommonCrypto* calls	495	78%
Binary only for ARMv7	7	1%
Not decompilable	46	9%
Out of memory	25	5%
Analyzable with *CommonCrypto* calls	417	84%

Table 5.2: Violations of security rules.

Violated Rule	**# Applications**	[%]
Rule 2: Uses non-random IV	289	69%
Rule 3: Uses constant encryption key	268	64%
Rule 1: Uses ECB mode	112	27%
Rule 4: Uses constant salts for PBE	72	17%
Rule 5: Uses < 1,000 iterations (PBE)	49	12%
Applications with ≥ 1 rule violations	343	82%
No rule violation	74	18%

Rule 2: Do not use a non-random IV for CBC encryption. This was the most commonly violated rule: 69% or 289 iOS applications used a cryptographically insecure initialization vector with CBC encryption. Among them, 92% or 265 applications specified a constant or NULL IV. The remaining 24 applied the hash value of a constant string as IV.

Rule 3: Do not use constant encryption keys. 64% or 268 applications used constant data as key material. Although not immediately applicable for encryption, we also consider constant passwords passed to a key derivation function as misuse. Table 5.3 highlights the provenience of key material. The total number of violations is higher than the number of applications due to the fact that some applications violated the rule multiple times. 193 keys were plain C strings that did not undergo any form of key derivation.

Rule 1: Do not use ECB mode for encryption. Overall, we found that 27% or 112 applications explicitly declared to use this mode of operation for symmetric encryption with AES or DES.

Rule 4: Do not use constant salts for PBE. We identified that 17% or 72 applications specified constant salt values as input to the key derivation function. The use of constant salt values is problematic as it effectively undermines the protection of password-based encryption against table-based attacks.

Table 5.3: Origin of constant secrets.

	# Violations
Constant string used as encryption key	193
Constant password for PBKDF2	84
Hash value of constant string	18
Secret retrieved from *NSUserDefaults*	14
Constant key data	6
Applications violating rule 3	268

Rule 5: Do not use fewer than 1,000 iterations for PBE. This was the least violated rule with only 12% or 49 applications applying less than 1,000 rounds in key derivation functions. Among them, 59% or 29 applications used exactly 1 round, 14% or 7 applications specified 100 iterations. The remaining 13 applications applied other values below the threshold of 1,000.

Manual Analysis

While most rule violations were found where expected, the analysis also pointed out deficiencies in our concept. Relying on a context-insensitive pointer analysis means that the points-to information of a pointer is independent of its calling context and might also include wrong locations. This further results in spurious paths being followed during parameter backtracking. Nevertheless, besides wrong values (or values belonging to a different calling context), the output will also always include an actually correct execution path.

In the following, we exemplify the analysis process based on one of the 15 open-source apps, namely the *Damn Vulnerable iOS App (DVIA)*[3]. We explain the security-critical weaknesses and contrast the source code with the results of our framework. DVIA is designed for penetration testing and includes common mistakes on purpose. Among other issues, it contains the kind of cryptographic misuse we are looking for. However, DVIA does not invoke functions of the library *CommonCrypto* directly but relies on the wrapper framework *RNCryptor*[4]. Nevertheless, violations should be detected by our framework.

Constant Password for PBE. The automated analysis of DVIA further reported two different origins of a password used for key derivation. One path ends up at a call to `-[UITextField text]`, which indicates that the password was retrieved from a text field. This signifies no rule violation and was also not detected as such. At the end of the second path, a hard-coded string was found. As highlighted in Listing 5.3, the constant *Secret-Key* was directly passed to the `encryptData()` method in *RNEncryptor* where it was subsequently used as a password input for `CCKeyDerivationPBKDF()`. Our framework correctly uncovered this security rule violation.

[3] https://github.com/prateek147/DVIA
[4] https://github.com/RNCryptor/RNCryptor

Listing 5.3: Constant password in DVIA.

```
NSData *encryptedData = [RNEncryptor encryptData:data
                                    withSettings:kRNCryptorAES256Settings
                                    password:@"Secret-Key"
                                    error:&error];
```

As can also be seen in the listing, the argument used as a second parameter is an object with the name `kRNCryptorAES256Settings`. If set, *RNCryptor* performs the password-based key derivation with 10,000 rounds, using a salt value generated by the previously described helper function `+[RNCryptor randomDataOfLength:]`. With regard to the corresponding rules, these settings do not represent a wrong application, as has also been correctly determined by our framework.

Constant IV and Salt Value. Our framework identified multiple execution paths that violate security rule 2, concerning the use of a non-random IV for CBC encryption, and rule 4, targeting constant salt values with password-based encryption. In both rules, the corresponding input parameter should have been generated using a cryptographically secure random number generator. DVIA, however, calls `+[RNCryptor randomDataOfLength:]`, a function belonging to *RNCryptor*. For all function calls using an IV or salt value, the report included three separate paths, of which only the one using `SecRandomCopyBytes()` was considered secure. Verifying these results manually revealed that *RNCryptor* only relied on `SecRandomCopyBytes()` if this function was actually defined, which is always the case on iOS devices. However, due to the nature of static analysis, we also found two alternative execution paths that tried to read bytes from `/dev/random`. This could fail due to two reasons: first, if the file descriptor was not available, and second, if the number of bytes to read was zero. In both cases, the IV or salt would have consisted of NULL values. Although this represents a correctly identified rule violation, its security impact is negligible. Based on this observation, we learned that reports generated during the automated process should be manually inspected in order to confirm critical rule violations.

Dead Code. The automated analysis reported rule violations by a function that was included in the binary but never invoked. Related to the generation of an initialization vector for symmetric encryption, it applied the method `+[RNCryptor randomDataOfLength:]` to generate random data and caused the same execution paths to be emitted as found before with constant IVs and salt values.

5.4.3 Limitations

As a result of analyzing 15 open-source applications by manual and automated means, we identified possible limitations that may affect the reverse-engineering process. In the following, we summarize the most relevant drawbacks that could prevent a successful analysis of iOS applications.

User Interface Elements. Missing type information of objects can result in an incomplete call graph. As a consequence, the data flow between functions cannot entirely be modeled. This circumstance is primarily caused by calls to external libraries that cannot be resolved, e.g., in case of user interface events. Since we also cannot determine the function signatures and parameter types, backtracking them becomes infeasible.

Defining event listeners and actions for UI elements works either by creating the user interface directly in the code or using the Interface Builder in Xcode. With the latter option, elements are stored in *.nib* files that are parsed during application start. As we only consider information retrieved from the binary, it is not aware of user interface actions declared elsewhere. In the context of static slicing, this might lead to situations where, e.g., an input parameter to the function `CCKeyDerivationPBKDF()` cannot be fully backtracked due to missing declarations of the associated *UITextField* object. Nevertheless, user input represents dynamic information and can never be captured by static analysis.

Polymorphism. Under certain program conditions, pointer analysis might follow implausible function calls if classes apply polymorphism. As a result, the call graph contains spurious edges that could have a negative influence on the accuracy of parameter backtracking. As exemplified in Listing 5.4, depending on `someCondition`, `object` has either type `ClassA` or `ClassB`. Determining the actually used type can only be done inside the branches of the allocation statement. Since our approach for pointer analysis is flow-insensitive, the points-to set of the subsequent variable `foo` includes both loc_A and loc_B. Assuming an instance method is now called using this variable, both types have to be considered.

To represent polymorphism more accurately in the call graph, we would have to use a flow-sensitive pointer analysis. Practically, this is rarely an issue as both allocations would have to be assigned to the same variable and parameter backtracking only considers paths where a variable of interest is modified.

Listing 5.4: Pointer analysis with polymorphism.

```
1  @class BaseClass {
2    - (void) foo;
3    - (void) foobar;
4  }
5  @class ClassA : BaseClass {}
6  @class ClassB : BaseClass {
7    - (void) bar;
8  }
9
10 void someFunction() {
11   BaseClass *object = nil;
12   if (someCondition) {
13     object = [[ClassA alloc] init];   // { loc_A }
14   } else {
15     object = [[ClassB alloc] init];   // { loc_B }
16   }
17   [object foo];                       // { loc_A, loc_B }
18 }
```

C Arrays. By using Andersen's field-insensitive solution [And94] for pointer analysis, individual fields of an array or struct cannot be distinguished from each other. Elements of these constructs are typically accessed via the stack pointer, which is in fact a base pointer to which an offset is added. The distinction between individual array entries is only possible based on this offset. If it is omitted, the points-to set would include the entire stack frame of a function, rather than building an individual set for each variable. Although, in general, this causes less precise analysis results, it usually does not affect parameter backtracking as arrays are always passed to functions using only their base pointer.

5.4.4 Discussion

In the manual study scenario, we focused on refining the framework to cope with closed-source iOS applications. While most of the rule violations were found where expected, we also discovered cases that could not be handled correctly due to limitations in our static analysis approach. In a context-insensitive pointer analysis, the points-to set of a variable is always independent of the underlying calling context. Consequently, parameter backtracking might traverse and report spurious execution paths. With regard to the overall analysis workflow, we recognized that it is crucial to restore function signatures and types from the binary during the decompilation step, as they facilitate further analysis.

The automated analysis of 417 closed-source applications revealed that 82% were subject to at least one security-critical implementation flaw. Besides these findings, inspecting the execution paths in all reports also showed that specific rule violations often result from a similar misunderstanding of the intended API usage, perhaps due to insecure default settings. For instance, regarding the use of non-random IVs for encryption in CBC mode, the same API method is also used with ECB mode. Therefore, setting the IV parameter is optional by design and the fallback to an all-zeros IV, when applied with CBC mode, does not have any immediately noticeable effect on application behavior.

As an alternative to specifying the number of iterations for key derivation explicitly, we identified applications that rely on the system-provided method `CCCalibratePBKDF()` to compute a suitable number of rounds with respect to the current device. This implies that applications can omit defining the number of iterations as a constant integer value. In its current form, our security rule does not actively consider the use of this API as a *positive test*, although this could easily be achieved by adding a JSON definition with this condition.

Related to constant encryption keys, we repeatedly noticed an execution path that transformed a password to a key without using a genuine derivation function. A password string of arbitrary length was either truncated to the block size of the used cipher or if too short, filled up with zero bytes. This behavior did not violate our rules and, thus, was also not reflected in the previously presented results. Nonetheless, in practice, it significantly weakens security by facilitating attacks on the encryption key. Although not possible automatically, uncovering such configuration issues is still feasible by manually studying execution traces of parameters passed to security-critical APIs.

5.5 Case Study 2 - Platform Comparison

In the following study, we present the first comparative evaluation of misapplied cryptography APIs in Android and iOS applications. Related research regarding crypto misuse never crossed the line between platforms. As it has already been demonstrated that security-critical implementation weaknesses are prevalent in both Android or iOS, we focus our analysis specifically on mobile applications that vendors distribute in separate versions for both platforms.

The goal of this case study is twofold. First, we ask if developers know how to use system-provided crypto APIs correctly and give an impression of how often security misconceptions occur in a representative set of applications. To prevent false positives or out-of-context findings, we verify all report execution traces in a manual study. Second, we ask how likely it is that applications which vendors distribute for both platforms, also violate the same security rules. Therefore, we compare the findings of security-critical mistakes in applications for Android with those in their iOS counterparts.

5.5.1 Method and Dataset

For the automated study of Android apps, we employ our framework, as presented in Chapter 3. For each inspected application, it outputs a report, lists found rule violations and associated execution paths. For iOS, we rely on our solution for binary analysis, as shown in Chapter 4, which pursues an equivalent approach and emits a report with execution traces for all inspected security properties. To establish comparable conditions for misapplied crypto APIs, we apply platform-specific implementations of the security rules, described in Section 5.3.

Dataset

We manually compiled a set of mobile applications where the use of cryptography seemed inevitable to provide specific functionality. Empirically, we found that this requirement affects at least applications for password management, secure messaging, encryption, sensitive data exchange, and secure cloud storage.

While Google Play uniquely identifies applications by their package name, e.g., *com.example*, the iOS App Store features no comparable identifiers. We were, thus, looking for other descriptors suited to match applications that were provided for both platforms. We found that for the vast majority of multi-platform offered applications, the title and/or description text used as metadata was usually widely consistent over both platforms. Moreover, in both app stores, the corresponding vendors were at least identifiable by their company name, website, support details. To measure the similarity of an app title and description, we organized all text elements of selected application metadata in a bag-of-words approach and compared them by pairwise computing the Jaccard coefficient. By empirically defining a reasonable threshold, we were able to automatically filter applications that apparently had no matching counterpart.

Table 5.4: Dataset of tested mobile applications.

	Count	[%]
Downloaded from iOS App Store	1,322	
Matching Android apps in Google Play	976	
iOS: No *CommonCrypto* calls	172	18%
Android / iOS: With crypto API calls	804	82%
iOS: App not decompilable	21	3%
Android: App archive corrupted	4	0.5%
Android / iOS: Out of memory	4	0.5%
Analyzable apps with crypto API usage	775	96%

We identified and downloaded 1,322 free applications from the iOS App Store, where we assumed the use of cryptography. Using the textual metadata of each app, we were searching Google Play for an Android pendant and were successful for 976 apps. All of them had at least 1,000 installations or ratings as indicated by Google Play and the iOS App Store. With a version for Android and iOS each, in total, we have acquired $2 \times 976 = 1,952$ apps for analysis.

After fetching iOS apps via iTunes, we used *Clutch* on a jail-broken iPhone to decrypt them. By inspecting their library bindings, it became evident that only 82% or 804 iOS apps included calls to *CommonCrypto*. As the remaining set of 172 apps without calls to system-offered crypto APIs also provided functionality where the use of cryptography seemed reasonable, security might be missing or provided via third-party libraries, which our rules are not designed to cover.

False Positives and False Negatives

As our analysis solutions for Android and iOS do not only warn about the existence of problematic statements but can also pinpoint their origin, we leverage the data flow seen in execution paths to assess the soundness of found issues. By *manually* validating all obtained analysis reports, we prevent findings of false positives and ensure that all rule violations indeed occur in reachable code that has a real practical impact on the security of applications.

False positives or out-of-context results can occur, e.g., if encryption is only used for obfuscation and not for actually enciphering secret messages. Likewise, applications might include code where crypto routines are only initialized with insecure default attributes but never used. By manually examining the execution traces before acknowledging a rule violation, we minimize false positives that could have occurred, e.g., if a backtracked value was a parameter to a function that had never been invoked during runtime.

As our implementations of security rules evaluate the use of system-provided crypto APIs based on their method signatures, we can be confident that these calls are always found, even if applications obfuscate their program code. For both platforms, this enables us to effectively avoid false negatives.

5.5.2 Results

In total, we studied 976 applications where vendors provided a version for Android and iOS via the official app stores. As summarized in Table 5.4, we did not find references to *CommonCrypto* in 18% or 172 applications for iOS. Irrespective of whether they actually contain cryptography-related code, we excluded these apps from further analysis, since we knew a priori that our security rules would not detect any violations in them. Interestingly, for all remaining 804 iOS apps that use the *CommonCrypto* API, also all corresponding Android pendants included calls to methods in `java.security.*` or `java.crypto.*`. This strongly indicates that crypto is actually needed for the correct functionality of these applications, rather than being just included incidentally, e.g., via used third-party libraries.

The inspection of 24 iOS and 5 Android applications failed due to errors in processing the program archives. Of them, 21 apps for iOS could not be decompiled from ARMv8 to LLVM IR code due to missing mappings of instruction semantics. Also, our Android framework was unable to process four apps, where the archive was damaged, despite repeated tries to re-download a functional version from Google Play. For another four apps, out of memory errors occurred during pointer analysis on iOS or register tracking on Android.

For 775 apps supplied to the iOS framework and our solution for Android, the analysis workflow terminated successfully. For each inspected application, we obtained a generated report that included the result of the performed security checks and for all rule violations, listings with problematic execution paths.

Violations of Security Rules

We found that 78% or 604 apps for iOS and 69% or 538 apps for Android commit at least one security-critical mistake. Among them, we identified 52% or 404 apps, where the Android and iOS versions of the same app commit at least one mistake on both platforms. Table 5.5 lists our observations of violated security rules. In the following, we discuss the findings in more detail.

Rule 1: Do not use ECB mode for encryption. On Android, we observed that 77% or 587 apps use instances of symmetric ciphers where the underlying mode of operation is ECB. Manually studying found execution path clarifies that block ciphers are mostly declared without explicitly specifying the mode and padding to use. This causes the underlying provider to apply ECB mode implicitly. Although the use of AES is predominant, at times we also noticed `Cipher` objects, specifying the nowadays weak DES algorithm.

CBC being the default mode on iOS, 25% or 192 applications explicitly declared to use ECB. In 22% or 172 apps, this mode was specified in the iOS version and also deployed in Android pendant of the same app.

Rule 2: No non-random IVs for CBC encryption. 35% or 271 applications on Android specified a static IV originating from hard-coded byte arrays or

Table 5.5: Security rule violations in 775 applications for Android and iOS.

Rule (R)	**Overall rule violations in apps**				**Rule violations in same apps on both platforms**	
	Android		**iOS**			
R1: Do not use ECB mode for encryption	598	(77%)	192	(25%)	172	(22%)
R2: No non-random IVs for CBC encryption	271	(35%)	494	(64%)	200	(26%)
R3: Do not use constant encryption keys	320	(41%)	455	(59%)	243	(31%)
R4: Do not use constant salts for PBE	112	(14%)	84	(11%)	49	(6%)
R5: Do not use $<$ 1,000 iterations for PBE	119	(15%)	145	(19%)	104	(13%)
R6: Do not use static seeds for PRNGs	25	(3%)	-		-	
Applications with $\geq$ 1 rule violations	538	(69%)	604	(78%)	404	(52%)
No rule violation	237	(31%)	171	(22%)	371	(48%)

constant values. Manually verifying these results, we found that the majority derived a cryptographically secure IV using the API `SecureRandom`. Some Android applications relied on the `Random` API instead, which leads to predictable IVs.

On iOS, this was the most prominent problem: 64% or 494 apps used a constant IV. Mostly, no IV was declared at all, implicitly causing an all zeros IV.

Rule 3: Do not use constant encryption keys. Constant or hard-coded encryption keys were identified as an issue in 41% or 320 Android applications and 59% or 455 iOS applications. Regarding the prevalence in apps for both platforms, 31% or 243 apps used constant data as key material.

Table 5.6 highlights the provenience of key material. The according execution traces show that most apps which employ constant keys do so by deriving them from string values or by declaring byte arrays with constants. In practice, apps typically create distinct instances of `Cipher` and `CCCryptor` objects on Android and iOS respectively for encryption and decryption. However, as they usually refer to the same constant key material, the total number of hard-coded secrets exceeds the number of applications with this issue.

Besides being entirely variable or static, from studying the execution paths, we learned that encryption keys can also be mixed, consisting of a statically defined key concatenated with a non-constant value. Interpreting these keys as constants would be inaccurate as we are unable to make assumptions about the entropy provided by the non-constant part.

Rule 4: Do not use constant salts for PBE. We identified that 14% or 112 Android apps and 11% or 84 inspected iOS apps passed constant salt values as input to key derivation functions. This effectively undermines the protection of password-based encryption against table-based attacks. On both platforms, most apps violating this rule declared a static byte array initialized with zero values. In 6% or 49 apps, this issue occurred in both versions of the same app.

Rule 5: Do not use < 1,000 iterations for PBE. On Android, we found that 15% or 119 apps employ less than the minimally advised amount of 1000 iterations for Password-Based Encryption (PBE). Likewise, 19% or 145 apps for iOS and 13% or 104 apps on both platforms declared a too small value. Regarding the distribution of the iterations, most rule-violating apps specified a count of either 20, 50, 64, or 100. In execution traces of iOS apps without this issue, we could observe a significant prevalence of apps using `CCCalibratePBKDF()` API to dynamically derive an iteration count, rather than hard-coding a value.

Rule 6: Do not use static seeds for random-number generation. As the platform APIs on iOS do not offer a seedable PRNG, this rule can only be violated by Android applications. Changes in the underlying PRNG implementation have globally fixed this vulnerability for systems running Android 4.2, API level 16, or newer. Due to this and only 3% or 25 Android apps declaring a constant seed for use with `SecureRandom`, the practical impact of this rule violation is limited.

Table 5.6: Origin of constant secrets used as key material.

# Violations on	Android	iOS
Constant string used as encryption key	238	305
Constant byte array as key	96	164
Hash value of constant string	9	36
iOS: Secret retrieved from *NSUserDefaults*	-	28
Applications violating rule 3	320	455

5.5.3 Discussion

The study of 775 applications that were distributed for both platforms revealed that in 52% or 404 apps security-critical mistakes were present in both the Android and iOS version. This suggests that the wrong application of cryptographic APIs is caused by a developer's lack of proficiency in handling cryptographic properties. Although this implies that the likelihood for configuration issues is independent of a particular API design, the distribution of rule violations per platform also exhibits indicators that some misconceptions are specifically promoted by the security architecture implemented in Android and iOS.

The most significant difference in rule violations across platforms was observed in the first rule targeting the use of ECB mode for encryption. The high number of 77% or 598 Android applications with this issue can be attributed to the fact that, unless a mode is specified, an implicit fallback to ECB occurs. In contrast, on iOS only 25% or 192 apps explicitly specified ECB instead of the default mode CBC. This circumstance strongly indicates that the prevalence of this problem on Android is related to the insecure default setting. Another notable difference between the two platforms has shown regarding the use of non-random IVs for CBC encryption. On Android, if no IV is specified, a `Cipher` instance will automatically generate a strong IV that can be retrieved via `Cipher->getIV()`. As opposed to that, not declaring an initialization vector on iOS will trigger a fallback that causes an all zeros IV to be used for encryption. This again underlines the importance of secure default and fallback settings.

The most frequently violated security rule in applications for both platforms affects constant encryption keys. Compared with other security violations, the key parameter always has to be specified explicitly. According to the high prevalence of this misconception, it seems that developers of both Android and iOS applications, are not aware of the radical consequences of hard-coding secrets.

Summarizing, we have seen that the origins of mistakes fall into two categories: firstly, those which are based on insecure default values in the corresponding API, e.g., implicit ECB mode on Android, or a NULL IV on iOS, and secondly, security problems that occur due to developers not carefully handling security-critical parameters, such as encryption keys or salt values. Our study confirms that neither Android nor iOS prevents developers from specifying weak parameters and shows that the wrong application of crypto APIs is still a widespread issue.

As a remedy, we propose the implementation of a two-fold strategy:

1. **API Changes:** Unsafe default values in APIs, such as ECB mode on Android, should be replaced by secure alternatives. In occasions, where omitting arguments impairs security, e.g., a NULL value as IV on iOS, a cryptographically secure random value should be generated instead. Although such profound changes might break compatibility with existing code, they would require developers to improve their implementations and implicitly minimize the widespread of problematic code.

2. **Raising Awareness:** Modern IDEs for application development, such as Android Studio or Apple Xcode, feature sophisticated code inspection. Following our detection strategies (see Section 5.3) to evaluate security-critical properties, code analysis in IDEs should be extended to warn about harmful practices, such as hard-coded encryption keys, the use of ECB mode, or a too low number of iterations for password-based encryption.

5.6 Conclusion

Many security-critical implementation weaknesses in mobile applications emerge from a wrong use of system-provided security APIs. While multiple studies have pointed out that such problems are common with Android applications, it was unclear whether similar configuration issues were also present with iOS applications. Although it was known that insecure default settings or wrong developer decisions can promote vulnerable implementations, existing research did not address the role of platform API designs in a comparative context.

We proposed a streamlined process to inspect security-critical aspects in Android and iOS applications. By performing a target-oriented analysis using our approaches for static analysis, we were able to identify the exact origin of weak or insecure parameters. To automatically evaluate execution traces, we presented platform-tailored detection strategies for common cryptographic principles and demonstrated concrete implementations using JSON definitions. We performed two case studies with mobile applications that include cryptography and found vulnerabilities in the majority of them. For the first time, we showed that the wrong application of cryptographic APIs is also a widespread problem in iOS applications. Focusing on applications that vendors distributed for Android and iOS, we compared the prevalence of individual security rule violations on a platform level. Our results indicate that a simplification of APIs, the replacement of unsafe default values with secure alternatives, and increasing awareness could substantially help to lower the widespread of vulnerable implementations.

In the context of this thesis, the study of misapplied crypto APIs in mobile applications underlines the practical relevance of a low-level application analysis and provides a valuable insight into the prevalence of implementations that rely on weak security properties. Considering the increasing complexity and size of nowadays applications, it is crucial to have automated solutions at hand that can successfully identify and pinpoint problematic code statements.

Part II

Understanding Mobile Application Behavior

6

Learning Application Semantics

Having gained an understanding of low-level implementation aspects in reverse-engineered source code, we now aspire to build a platform that enables us to reason about the effects of coherent code parts in Android applications. Despite their versatility in tracking and uncovering security-critical implementation weaknesses, approaches for static and dynamic program analysis do not cover all use-cases that are relevant for a holistic security analysis of mobile applications.

In this chapter, we describe an approach towards augmenting the inspection of mobile applications with contextual information. We start in Section 6.1 by highlighting security-relevant analysis scenarios that require a broader view on coherent code statements. Section 6.2 explains the challenges of modeling semantic relations between words and introduces two modern NLP techniques to capture the linguistic context of words in documents. To detect the relevance of semantic features in large amount of input data, in Section 6.3, we summarize key aspects of machine learning and point out the advantages of convolutional neural networks for efficient text classification. In our research, we envisage to apply these concepts to the source code and metadata of Android applications in order to obtain a more sophisticated understanding of their purpose and security-critical behavior. Parts of this chapter are taken verbatim from [FG20b].

Publication Data and Contribution

Johannes Feichtner and Stefan Gruber. "Understanding Privacy Awareness in Android App Descriptions Using Deep Learning." In: *Conference on Data and Application Security and Privacy – CODASPY'20*. ACM, 2020, pp. 203–214. DOI: 10.1145/3374664.3375730

Contribution: Main author; Prototype implemented by Stefan Gruber.

6.1 Introduction

By performing a target-oriented security-analysis with approaches for static and dynamic program inspection, we succeed in identifying and tracking individual code statements. Evaluating execution traces and parameters of appendant code statements helps us to reliably uncover improper usage of crypto APIs and other security-relevant functionality. At the same time, working only with excerpts of application code also precludes certain inspection scenarios that would be relevant for a more in-depth program understanding. For instance, to find anomalies or problematic code patterns, a broader view on coherent code statements is needed.

For an efficient workflow and to boost productivity, many developers [Der+17] of Android applications rely on functionality provided by third-party libraries and code snippets from discussion platforms, such as Stack Overflow. Usually [SSE15], the search starts with only a few keywords, e.g., "Android trusting all certificates" and quickly leads developers to discussion posts that provide query-related code snippets or public repositories with libraries that involve the desired functionality. As highlighted by multiple studies [Yas+19; Fis+17; Li+16], code pieces from external sources often contain serious vulnerabilities that compromise the security of millions of apps. However, even if a specific code snippet or library is known to be problematic, it remains difficult to reveal if a specific mobile application is affected. If a flaw is not contained within a single program statement, e.g., a particular hard-coded encryption password or low iteration count used with key derivation, it cannot be detected via static or dynamic program analysis. In addition, when developers apply code transformation techniques, such as obfuscation, inlining, or shrinking, control and data flows change and it becomes infeasible to *fingerprint and accurately recognize code fragments or libraries.*

Whenever developers provide updates for their mobile applications, e.g., to fix a security- or usability-related bug, it would also be interesting to see how thoroughly changes have been implemented. For instance, in 2018, the disclosure of a vulnerability named Efail [Pod+18] (CVE-2017-17689) required vendors of email applications with S/MIME support to provide a patch for a critical flaw in the S/MIME end-to-end encryption standard. Again, due to the widespread use of code obfuscation and the inherent indistinguishability between program libraries and developer code, it is very time-consuming to verify how security-relevant fixes, such as for the Efail vulnerability, have been realized. Techniques for code optimization and transformations, e.g., renaming all variables to meaningless pseudo-identifiers, prevent to effectively *identify the differences among Android apps* and, thereby, raise the costs and effort needed for a holistic security analysis.

In the case of Stack Overflow or code sharing platforms like GitHub, the *semantic understanding* and classification of information is provided by users. For a better categorization of posts or repositories, individual tags can be assigned that summarize the core functionality in only a few keywords. This again creates metadata, builds links to similar questions or projects, and supports developers in efficiently finding code snippets that fulfill a particular purpose. While it is trivial for developers to abstract their understanding of code into single keywords, extracting and describing the purpose of code remains challenging for computers.

In recent years, new architectures for neural networks have evolved to discover relevant features in large amount of data. Supported by the computing power of distributed cloud instances, novel concepts for *deep learning* make a manual, time-intensive feature engineering obsolete and introduce new capabilities for computers to derive a contextual understanding of information. E.g., to model the context of individual words in text, solutions, such as word2vec [Mik+13] and GloVe [PSM14] have been proposed. The common idea is to represent the meaning of words as vectors (embeddings) in a multi-dimensional space. Vectors close to each other indicate similar semantics. Technically, these approaches work by transforming large amount of input, e.g., sentences, paragraphs, documents, into a representation that can be further processed by machine learning algorithms.

As source code resembles regular text in many linguistic aspects, intuitively, techniques for *natural language processing* should also be applicable to the reverse-engineered code of Android applications. However, the underlying syntactical, grammatical, and structural forms of text and code are different. E.g., in addition to "word-like" tokens, such as strings and identifiers, the semantic expressiveness of source code is made up of keywords, constants, special symbols, and operators. In contrast to pages and chapters in a regular book, code methods can easily be moved to other classes, and classes to different packages. In addition, embeddings only map the context of individual entities, i.e., they cannot be used to reason about the functionality of classes, packages, or libraries in mobile applications.

To build structures over coherent parts of code, infer their main purpose, and find what characterizes an application, we propose to leverage the distinctive abstraction capabilities of neural networks. State-of-the-art architectures for recurrent neural networks (RNN) and convolutional neural networks (CNN) excel in learning generalizations of local properties in input data. In combination with meta information extracted from sources that surround a particular code of interest, i.e., selected attributes that are invariant to transformation techniques and compiler peculiarities, techniques for knowledge discovery in text should also work well on source code. Ultimately, the use of advanced neural networks could help to discover what code parts are significant, what distinguishes mobile applications from each other, and to *model the behavior of mobile applications.*

In order to associate a semantic model with a concrete meaning, we need to supplement it with information that describes code in the best way possible. In practice, for all Android applications that are distributed via Google Play, vendors indicate the category the app belongs to and provide a user-oriented functionality description. The distribution platform, in addition, lists the system permissions an app requests to be granted. In the course of machine learning, one or multiple of these attributes could be linked with the source code of an application. Evidently, rather than assigning a higher-level description to every individual line of code, this approach could be employed to find a suitable program category based on the implementation or measure the correlation between a developer-provided functionality description and implemented app behavior. Likewise, deep learning techniques could assist in correlating description texts with the use of *dangerous permissions* to better *understand the privacy awareness of Android applications.*

6.2 Natural Language Processing

Computers understand natural language if the underlying syntax and semantics can be translated into mathematical models. Besides speech recognition, research domains of NLP include information retrieval, text classification, and sequence-to-sequence translation. Despite significant improvements throughout the last years, it remains challenging for machines to correctly interpret the meaning of text. Systems for *language modeling* enable computers to learn sequences of words as discrete values, collect statistics about word frequencies, and, depending on the purpose, can also be targeted to capture contextual information.

A trivial form of word representation for machine learning is *one-hot-encoding*. Each word of a predefined vocabulary is represented as a vector, with the vector dimension being the total number of words. All vector elements are zero, except for one, whereas the corresponding index refers to a certain word. Encoding each word as a discrete binary value in a dense vector allows us to transform regular text to an input that can be processed by machine learning classifiers. However, the sparse encoding also has some disadvantages. The memory needed to represent full document samples increases with the size of the vocabulary and the length of the words. If a neural network then processes one-hot vectors as an input, all operations and internal parameters have to span over the entire dictionary, although most vector elements are zero. In addition, there is no natural notion of similarity modeled within these vectors, e.g., the words "cat" and "cats" are considered unrelated to the same extent as "bird" and "truck". To tackle this issue, commonly referred to as *curse of dimensionality*, statistical models have been proposed that map the probabilistic distribution of words [BDV00]. By arranging one-hot word vectors as data points (embeddings) in a continuous vector space, the vocabulary can be processed much more memory-efficient and arithmetic operations [GAM17] allow to assess semantic relations between words.

6.2.1 Word Embeddings

Vector space models, in particular word embedding models, support NLP tasks with memory-efficient and arithmetically meaningful word representations. In the following, we briefly introduce the idea behind embeddings and present the approaches we rely upon in our work, namely word2vec and GloVe.

Word2vec

Mikolov et al. [Mik+13] proposed an embedding technique that calculates the probabilistic distribution of words via skip-grams. The algorithm relies on a shallow neural network to learn word vectors so that we can predict the context in which a certain word occurs. For instance, as depicted in Figure 6.1, given a document containing the sentence "I bought a red car today", the embedding for "red" is trained by providing the model with surrounding words. During learning, the probability to predict "car" as a context word, is maximized by minimizing the loss function. For this to achieve, word2vec is designed such that at a specific

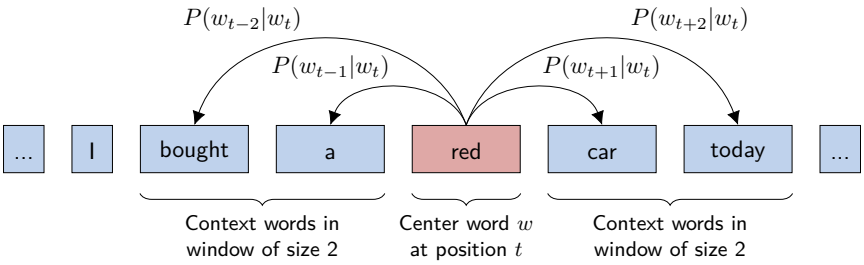


Figure 6.1: Skip-gram model to predict surrounding words via $P(w_{t+j}|w_t)$.

point in the chain of calculations, all information needed to make a prediction is represented as a small low-dimensional vector. We can store this dense vector as the *word embedding* for a certain word. Typically, models are trained until the average predictive accuracy of context words no longer improves. By supplying a single term to a trained embedding model, we are able to predict the set of words that frequently appear close-by. These surrounding words define the context of a word within a certain neighborhood that is confined by a fixed-size window.

Word embeddings are trained using neural networks and require a significant amount of training data. In practice, the use of words is not uniformly distributed and contextual relations are constrained by the content of the documents used for learning. To reduce the number of parameters neural networks have to learn, numerous databases with pre-trained word embeddings are available and can be employed as an alternative to generating them from scratch[1].

GloVe

GloVe [PSM14] is an embedding model that combines a context-based skip-gram model, such as seen with word2vec, with a count-based matrix factorization. The overall idea is that words in the same contexts exhibit similar or related semantic meanings. Therefore, the algorithm captures the ratios of co-occurrence probabilities rather than the probabilities themselves. While word2vec traverses a corpus word-by-word and records the co-occurrence of words one at a time, GloVe directly counts them for a full document.

Assuming a large matrix M, each cell $M_{i,j}$ refers to how often word i is used in the local context of word j (n-gram). To map such count-based statistics between neighboring words within small, dense word vectors, Pennington et al. propose to use a simple weighted least squares regression model instead of a predictive neural network. According to the authors of GloVe, the count-based approach ensures fast training, a good performance even with small training corpora, and in some cases outperforms purely context-based models, such as word2vec. As it is difficult to say, in general, whether to employ a count-based or a probability-based embedding model, the performance of both techniques has to be evaluated with regard to a specific task.

[1] https://code.google.com/archive/p/word2vec/

6.3 Neural Networks

Neural networks have been used with considerable success in solving complex tasks like speech or image recognition. A strong increase in computational power and the rapid development of advanced machine learning concepts have introduced the era of so-called *deep learning*. Machine learning frameworks, such as TensorFlow [Aba+16] or PyTorch [Pas+19] feature a variety of algorithms, run them efficiently on GPU, and simplify building big computational graphs.

Traditional feed-forward neural networks are composed of a collection of neurons that are organized in layers, whereas each has its own learnable weights and biases. Given a single vector as an input, a network transforms it via a sequence of hidden layers and ultimately passes it to an output layer, which maps the result to a set of predefined classes. Each hidden layer is built upon a structure of neurons that are connected to all neurons in the previous layer, i.e., a *fully-connected* or *dense* layer. The network learns by repeatedly updating the weights that are assigned to these links between neurons. By back-propagating the error, i.e., *loss*, a model tries to find weights that generalize the properties seen in samples during training. For predictions, the network then leverages the learned abstraction and can map a previously unseen input to a certain output.

Many algorithms that are used with neural networks need configurations, i.e., *hyperparameters*, to control the learning process. Finding these values is an empirical process that can sometimes be facilitated using grid or random search [BB12]. The most relevant hyperparameters to adjust include:

- **Activation function:** Models the firing rate of a neuron.
  Examples: Sigmoid, Softmax, tanh, Rectified Linear Unit (ReLU)
- **Optimizer:** Efficiently finds parameters that minimize the loss function.
  Examples: Stochastic gradient descent, ADAM, RMSProp
- **Regularization term:** Penalizes the complexity of the model.
  Examples: L1, L2, batch normalization, Dropout (typical for deep learning)
- **Weight Initialization:** Prevents vanishing or exploding activation output.
  Examples: Xavier/Glorot, random (normal or uniform distribution)

6.3.1 Convolutional Neural Networks

Ordinary neural networks are inherently limited by their processing capability. Within dense layers, each neuron is connected with all neurons in adjacent layers. This connectivity implies that input data follows a persistent structure, in which all features always occur at a fixed global position. It also means that all features are considered equally relevant for processing. In practice, there are some domains, such as speech, images, or text, where it seems reasonable to extract only useful attributes, i.e., learn local properties of input features. Convolutional neural networks (CNN) [LeC+89] correspond to this need by encoding selected properties of input data directly into the architecture.

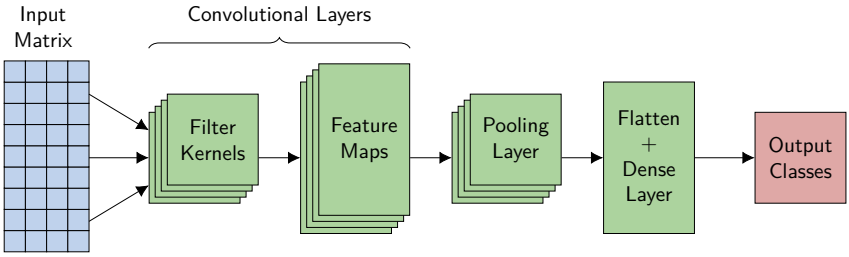


Figure 6.2: General CNN architecture for classification tasks.

When processing images, CNN layers can be pre-arranged with neurons that capture, e.g., the red, green, and blue values of pixels. In a regular neural network, a single fully-connected neuron in the first hidden layer would be connected to all color channels of all pixels. Assuming an image of size $250 \times 250 \times 3$, it would, thus, manage 187,500 weights, whereas back-propagating the error through the network causes each weight to be adjusted in every epoch. Evidently, this does not scale and the large number of parameters would quickly lead to overfitting. The architectural transition from fully-connected neural networks to CNNs is entailed to fulfill three properties [GBC16, p. 329–335]:

1. **Sparse interactions:** A filter with a kernel smaller than the whole input is windowing the input data. This leads to fewer parameters being held in memory, less computing operations, and higher training efficiency.

2. **Parameter sharing:** A learned pattern can be used for multiple inputs; it is not tied to the value of a weight applied elsewhere.

3. **Equivariant representations:** While the detection of patterns is not strictly limited to input regions, e.g., an image position, a translation is still detectable in the output. Precisely, a convolution function is equivariant if any changes to the input are likewise reflected in the output.

Figure 6.2 shows the general architecture of CNNs. One or multiple *convolutional layers* include *filters* of a predefined pixel size (*kernel size*) that are applied to input data in order to learn spatial properties. These filters will be activated by the network when they see some type of visual feature, e.g., small edges within an image. As the filters slide over the input data using a fixed-size window, they each produce an individual 2-dimensional activation map. The results of all these convolution operations are stored in *feature maps*.

A *pooling layer* then performs a downsampling operation on the feature maps. It progressively reduces the number of parameters by taking the global maximum (*max pooling*) or average value of several feature map entries. In many state-of-the-art CNNs, different combinations of convolutional / pooling layers are nested on top of each other, resulting in a deep convolutional architecture.

One or multiple fully-connected *dense layers* typically unroll (*flatten*) the pooled values and compute the final classification or regression output.

6.3.2 Text Classification

In regular neural networks, samples are learned independently from each other. This implies that any structure in the original data, e.g., the position of pixels in images or the order of words in sentences, is not preserved. With text, this issue can be compensated to some extent by breaking down sentences into n-grams, where n refers to the number of words to be combined. However, the fixed input structure of ordinary neural networks prevents them from working with n-grams of arbitrary size, i.e., they cannot adapt to the variable length of sentences.

To process variably-sized input, several architectures for recurrent neural networks (RNN) have been proposed. The general idea is to process a sequence of vectors by applying a recurrence formula at every time step. Whenever the RNN processes a new input, it updates its internal hidden state and, thereby, allows sharing features learned across different positions of text. Among the most commonly used systems that incorporate this concept are long short-term memory (LSTM) [HS96] and gated recurrent unit (GRU) [Cho+14] networks. Results are often intuitively comprehensible, e.g., by simply reading a sequence from left to right, making RNNs a preferable choice for language models [Mik+10], sequence tagging [HXY15] and translation [SVL14]. However, due to the recursive structure, training cannot be parallelized as efficiently as with CNNs.

Although originally invented for computer vision, CNNs have shown to also perform well with NLP tasks, such as classification and search query retrieval. By design, CNNs are efficient, easy to parallelize on GPU and excel in tasks where positional features have to be extracted, which is also the case with source code or text in natural language. CNN architectures for text classification have been proposed both on character-level [ZZL15] and sentence-level [Kim14].

In our work (see Chapter 10), we extend the sentence classification network by Kim et al. that is shown in Figure 6.3. First, words are transformed to low-dimensional word embeddings, e.g., using word2vec or GloVe, and concatenated to form a matrix. Each row of the matrix then corresponds to one embedding.

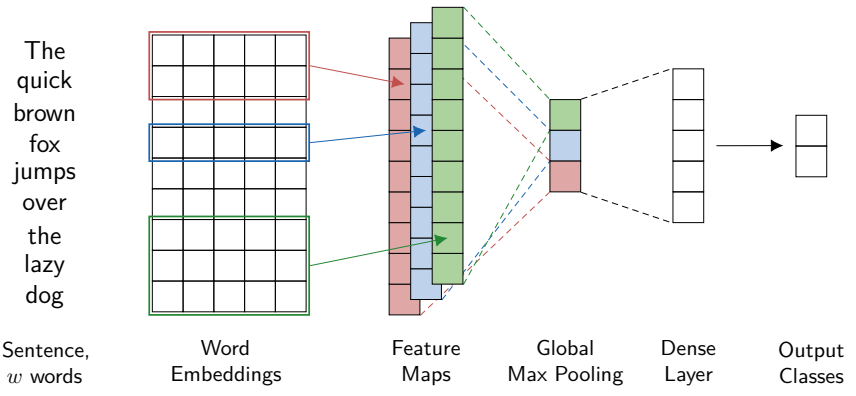


Figure 6.3: CNN for text classification by Kim et al. [Kim14].

A sentence with 9 words using 100-dimensional embeddings would result in a 9×100 input matrix. If the matrix would be an image, filter windows would slide over local regions, trying to find specific information. With word embeddings, filters typically slide over full rows, i.e., embeddings, of the matrix. Thus, the width of filters usually corresponds to the width of the input matrix and the height is set to the amount of words to be captured at once, similar to n-grams.

Word embeddings support the *filter kernels* in learning generalizations of the input data. Every filter performs convolutional operations on the input matrix and stores the 1-dimensional results in feature maps of variable length. Performing the convolutions allows the model to disclose the impact of single words as well as of word formations. In practice, there are often several filters of a specific kernel size, such that multiple words or combinations can be captured.

A *pooling layer* subsequently identifies input features with a high relevance by performing a global *max pooling* operation. Thereby, the largest number from each feature map is retrieved and passed on to a fully-connected, *dense layer*. Based on this feature vector, that is a combination of single word embeddings from local contexts, the dense layer performs a multi-class classification. Therefore, a softmax function is used, as it can determine the probability of multiple classes at once. For each trained output state, the network yields a probability that classifies the sentence previously provided as an input.

6.4 Conclusion

Static and dynamic approaches for program analysis enable researchers to verify low-level implementation aspects in Android and iOS applications. Typically, this involves a targeted vulnerability analysis that often succeeds in uncovering security-critical mistakes. However, traditional program inspection suffers from a lack of context-awareness and disregards the use-case applications are designed for. While many issues can be tackled by identifying and tracking relevant code statements, the complexity of modern applications prevents existing approaches from revealing the context problematic statements occur in. Understanding the behavior of mobile apps on a higher-level is challenging due to the widespread use of code obfuscation, functionality that spreads over thousands of classes, and the many ways how developers can express semantically similar code.

For computers to learn the semantics of mobile applications, they need to be able to assess the relevance of individual parts in source code. Since program statements remind of text in many aspects, in this chapter, we introduced modern techniques in the fields of NLP and machine learning that excel in capturing the context of natural language and might also perform well in a semantic analysis of mobile apps. In the following chapters of this thesis, we explore different approaches to gain a more sophisticated understanding of the behavior of Android applications. This involves tackling code obfuscation, assessing the similarity of program statements, and applying the building blocks, presented in this chapter, on the metadata and source code of mobile applications.

7

Code Recognition on Android

Many developers take advantage of third-party libraries and code snippets from public sources to add functionality to applications. Besides making development more productive, external code can also be harmful, introduce vulnerabilities, or raise critical privacy issues that threaten the security of sensitive user data and amplify an app's attack surface. Reliably recognizing such code fragments in apps is challenging due to the widespread use of obfuscation techniques and a variety of ways, how developers can express semantically similar program statements.

In this chapter, we propose a code recognition technique that is resilient against common code transformations and that excels in identifying code fragments and libraries in Android applications. In Section 7.1, we motivate our research and explain how state-of-the-art solutions identify libraries. Section 7.2 introduces our system design and highlights our selection of code features to overcome program obfuscation. In Section 7.3, we elaborate how to generate well-characterizing fingerprints by combining features from the Abstract Syntax Tree of methods with invariant attributes from method signatures. We thoroughly evaluate how well our solution tackles obfuscated, shrunken, and optimized code in Section 7.4 and demonstrate its practical ability to fingerprint and recognize code with high precision and recall. Parts of this chapter are taken verbatim from [FR19].

Publication Data and Contribution

Johannes Feichtner and Christof Rabensteiner. "Obfuscation-Resilient Code Recognition in Android Apps." In: *Availability, Reliability and Security – ARES'19*. ACM, 2019, pp. 1–10. DOI: 10.1145/3339252.3339260
Contribution: Main author; Christof Rabensteiner elaborated the fingerprint inclusion check and significance score, and implemented a prototype.

7.1 Introduction

Most Android applications are bundled with third-party libraries that potentially include vulnerable or outdated code [Der+17]. The Apache Cordova library, e.g., was affected by a vulnerability that enabled an attacker to interfere with an application's behavior by sending malicious intents[1]. Providing the building blocks for a majority of cross-platform applications, this flaw immediately put the security of all of them at risk. Likewise, adversaries can leverage advanced reverse engineering techniques to tamper with popular libraries, add harmful code, and republish compromised versions that preserve the original functionality. The problem of uncovering such small differences is commonly referred to as clone or plagiarism detection and conceptually exhibits requirements similar to code recognition. Techniques for clone detection work on semantic and syntactic features of programs and measure the similarity of code based on tokens [VHP16; CGC12], parsing trees [AI13; Bax+98], or dependency graphs [CLZ14; Luo+14].

Although third-party libraries on Android undoubtedly introduce potential security problems, insecure code snippets can also be located within app-specific code. If developers copy ready-to-use code snippets, e.g., from programming discussion platforms like Stack Overflow, they unknowingly might also introduce weaknesses. In 2017, a study [Fis+17] has revealed that 15.4% of 1.3 million inspected Android applications included security-related code snippets from Stack Overflow, whereas 97.9% of them contained severe security problems. Apart from introducing vulnerabilities, multiple studies [Seo+16; Li+16; Gra+12a] have demonstrated that code from external sources can also leak private information, exploit their privileges, or forward sensitive data to unauthorized parties.

Since the use of third-party libraries and the integration of code snippets from external sources have evolved to common practices in app development, it is of utmost importance to find vulnerable code fragments. Despite significant research efforts to dissect apps and uncover such threats, a reliable identification of insecure program parts remains challenging. Developers can express semantically similar program statements in a variety of ways and very often, control and data flows are altered when code transformation techniques, such as obfuscation, identifier renaming, shrinking, or method inlining, are applied during compilation.

Existing approaches for code recognition in Android applications mostly target third-party libraries and involve either whitelisting or a similarity-based strategy. In the former case, a precompiled whitelist of directories or package names is used as a reference to individual libraries. This concept is well suited to investigate the security risks associated with using specific third-party components, e.g., for advertising [Liu+15] or usage statistics [Liu+20]. However, an inherent problem of whitelists is that they usually have to be gathered manually [BBD16] and need active maintenance to stay up-to-date. Considering the constant intervention and the fact that it is practically infeasible to cover all relevant libraries by their vendor name and version number, this approach does not scale and is only suited for analysis scenarios where it is viable to check for exact code matches.

[1] https://cordova.apache.org/announcements/2015/05/26/android-402.html

The second approach consists in identifying Android libraries without prior knowledge [Zha+18b; Che+16; Ma+16b]. Therefore, applications are decompiled and split into sets of potential library candidates. A custom similarity metric or hash-based comparison then measures the difference to candidates that have previously been extracted from other applications. If the similarity score exceeds a predefined threshold, candidates are considered to be the same libraries.

As such approaches are still prone to fail if code transformation techniques are employed, more elaborate solutions based on machine learning and clustering have been proposed. PEDAL [Liu+15] trains a support-vector machine model to detect obfuscated advertisement libraries by extracting only selected features from code. Evidently, possible classification outcomes are limited to the set of labeled training samples due to the choice of a technique for supervised learning. AdDetect [NCC14], AnDarwin [CGC13] and WuKong [Wan+15] build on the assumption that a library consists of only one package and segregate package hierarchies into distinct clusters. LibRadar [Ma+16b] augments the approach by assigning each cluster a unique profile representing a library. However, the lack of ground truth in all these solutions and the use of heuristics come at the cost of precision and recall. None of the listed solutions is able to handle partial library inclusions, e.g., as a consequence of shrinking or dead code elimination.

Although research has demonstrated the practical feasibility to identify code, existing work still leaves room for improvement:

1. Current approaches for code recognition in Android applications focus on detecting individual third-party libraries by name and version. They require large amounts of ground truth for training and do not work effectively if the reference codebase is small or a priori incomplete. Trained on Pinpointing specific code snippets instead of full libraries is, thus, infeasible.

2. State-of-the-art methods strongly depend on Java package names, preserved directory hierarchies, and unaltered method signatures. However, package structures and names can be different in multiple versions of the same library. Also, during compilation, code can mutate as it undergoes automated performance-related code optimizations, such as constant propagation, method inlining, duplicate code merging, and removal of unused method parameters. Focusing too much on such auxiliary information can, thus, give a false sense of a good classifier that is only reliable if trained libraries and tested applications exhibit the required attributes and do not apply code optimizations.

3. Android apps commonly apply code transformation techniques, including obfuscation, identifier renaming, and shrinking not only to optimize code but also to harden against various forms of abuse, such as tampering, reverse engineering, and intellectual property theft. While existing classification approaches based on similarity metrics might yield useful results despite such modifications, the recognition rate with real-world applications could significantly be improved if techniques were resilient to common types of obfuscation and code mangling.

We address shortcomings of existing approaches and introduce a solution that is able to recognize arbitrary code fragments in Android applications, even if code transformation techniques, like shrinking or obfuscation, are applied. We overcome the aforementioned limitations by extracting and processing features from the Abstract Syntax Tree (AST) of methods. Our approach does not rely on identifiers of packages, classes, and methods and uses them only as supplementary information. Instead of a hash-based comparison, we measure the similarity of methods using vectorized fingerprints we derive from the AST of methods and transformation-invariant representations of method signatures. This enables us to recognize not only full libraries but also individual code snippets and library parts. To compare code segments, we design a scoring metric that accurately determines inclusion within other code parts and can express the similarity of classes and packages based on an aggregation of fingerprints.

Compared to previous research, our solution excels in reliably recognizing code fragments, even if a very high degree of code obfuscation is applied and if the majority of originally trained code is no longer present, e.g., due to code merging or inlining. Our approach is scalable and succeeds in accurately matching individual small code snippets at method level as well as entire libraries. Aimed at conditions that can be found with real-world apps, our solution is suited for arbitrary tasks that involve code recognition in Android applications.

Our key contributions are as follows:

- We present a framework to reliably recognize arbitrary code fragments in Android apps. Our solution can overcome various limitations present in existing research and represents an effective method to identify used libraries, recognize specific code snippets, or find semantically similar code.
- We study features in code that are invariant to widely used code transformation techniques and propose a novel feature matching process that is resilient to code mangling, identifier renaming, shrinking, and optimizations, such as inlining, code merging, or removal of unused method parameters.
- We evaluate the quality of our algorithm by testing it with a set of open-source libraries. We compile all libraries multiple times with different forms of code transformations enabled and assess the impact on classification. Moreover, we ensure the soundness of our solution by thoroughly comparing the expressiveness of chosen features and threshold values for matching confidence and package significance.

In the context of this thesis, tackling the problem of code recognition in Android apps is an integral step towards identifying semantically equivalent program parts. To better understand the behavior of applications, it is crucial to distinguish functional changes by developers from non-functional adaptations induced by compile-time optimizations and obfuscation techniques. In a broader scope, our work represents an effective solution for code fingerprinting and pattern recognition. By extracting transformation-invariant features from Abstract Syntax Trees and method signatures, we succeed not only in identifying individual code snippets but can also measure their semantic similarity.

7.2 System Design

We design a static analysis framework to recognize code in Android app archives. The primary functionality can be split into two parts: In the **learning phase**, our tool is trained with arbitrary code fragments. In the **matching phase**, we automatically analyze an app and try to recognize code parts using previously learned data. The objectives of our solution can be summarized as follows:

1. If an app includes a library or code fragment, the tool should identify it using an unambiguous label, if known.
2. The tool should work equally with obfuscated code.
3. After analyzing an app, the tool should list packages that resemble previously learned packages or code fragments with a score indicating how much code has been matched.

With these properties in mind, our fingerprinting approach, as detailed in Section 7.3.1, is based on **AST Vectors** and **Sanitized Signatures**. AST vectors are compact vector-style representations of attributes that are gathered from interpreting structural dependencies in the AST of a method. Sanitized signatures are built by removing all transformable or variable identifiers from a method signature. Such elements include a method's name, access modifiers, and all references to class objects in parameter types and the return type. We combine an AST vector and a sanitized signature to form a fingerprint that describes an individual method. Consequently, an aggregated set of fingerprints can be employed to represent a class or a full package hierarchy.

7.2.1 Overcoming Obfuscation

In regular Android applications, code fragments or libraries can be recognized with reasonable certainty by matching the names of packages, classes, and methods. However, if code transformation techniques, such as obfuscation, shrinking, or inlining are applied, these identifiers become inconclusive.

In a survey from 2018, Wermke et al. [Wer+18] analyzed 1.7 million Android apps regarding the use of obfuscation techniques. According to the authors, 24.92% of apps are obfuscated, whereas the most prevalent obfuscation system is ProGuard. While the authors confirm that identifier renaming of classes, methods, and fields is among the most popular features, they make no statements about minified or shrunken apps. Most research of obfuscation in Android applications concentrated on reversing [BPM17; Bic+16] and analyzing applications in spite of obfuscation [Gla+17; Vás+14; Zha+14a]. More recent studies specifically focus on obfuscated malware [HGM18] and address the impact of obfuscation on Android anti-malware products by inspecting 7 obfuscation strategies and 29 techniques. Also in this context, the research of Park et al. [Par+19] and Garcia et al. [GHM18] aims at identifying obfuscation-resilient properties as an input to machine learning classifiers in order to uncover malware.

In our work, we address obfuscation as well as common code transformation techniques by focusing solely on invariant code attributes. This involves features that *(1)* are suited to identify a code segment and remain the same for semantically similar sections of code, and *(2)* cannot be tainted if multiple revisions of the same app are compiled with different obfuscation settings. In the following, we point out how our solution handles some widely used transformation techniques.

Identifier Renaming

If activated, the obfuscator replaces names of variables, methods, classes, and packages with randomly-generated, meaningless characters. Fingerprints that are based on such identifiers would fail to recognize semantically equivalent but differently named code segments. As our solution does not rely on identifiers at all, it cannot be affected by this modification.

Shrinking

Shrinking involves removing unused code. Our approach is to some extent resilient to this operation as methods, classes, and packages are matched by similarity.

Optimizations

Optimizations are applied to reduce the size of the resulting Dalvik bytecode and to increase performance. Among other operations, this involves forced inlining, outlining, constant propagation, `switch-case` rewriting, etc. While some of these features can taint our fingerprints, in our evaluation we show that for most of them the practical impact is marginal. As we measure similarity among fingerprints based on the distance in the vector space, it is still possible to recognize cases where two slightly different fingerprints refer to a semantically equivalent code.

The obfuscator ProGuard provides developers with 29 possible optimizations[2]. Some of them cause code to be transformed such that the control and data flow is altered. In these cases, our solution loses track when matching code fragments:

- **Inlining:** If enabled, ProGuard substitutes a method invocation with the body of the called method. This change of the control flows tampers with the AST vector. However, as this operation is performed at method-level only and typically only for short methods, it is still feasible to accurately identify specific classes or packages using additional method fingerprints.
- **Merging:** With this optimization activated, ProGuard merges duplicated blocks of code by modifying branch targets. This operation affects AST vectors as it prunes nodes from a method's AST.
- **Method Parameter Removal:** If unused parameters are eliminated, methods exhibit different type signatures and this leads to different sanitized signatures. This operation prevents us from matching methods, as matching this feature type requires strict equality.

[2] https://www.guardsquare.com/en/products/proguard/manual/usage/optimizations

7.3 Workflow

Our approach starts by converting code provided as a method, class, or package into the `.dex` format. This task is delegated to the build tool `d8`, which is included in the Android SDK. Based on the Dalvik bytecode obtained, for each available method, we generate a fingerprint and arrange all of them within a package hierarchy. When **learning** code, we stop here and store the results in a database. When **recognizing** code, we compute a similarity score between zero and one, indicating whether given code fragments can be matched fully, partially, or not at all by comparing with learned fingerprints and known package hierarchies.

7.3.1 Fingerprinting Code

Potharaju et al. [Pot+12] proposed a technique to compute fingerprints based on features extracted from the AST of methods. Therefore, the Android app archive was first transformed into a custom assembly language, followed by pruning the code of each method body, keeping only references to method calls and replacing all variable identifiers with placeholders. Of all method signatures, only the number of used arguments was preserved. The remaining instructions were then arranged as AST and formed the basis to derive a fingerprint vector.

We adopt the algorithm of Potharaju et al. as it solves a problem that is close to ours. However, as opposed to their work, we cannot create a fingerprint based on the sum of all method fingerprints, since we have to assume that code parts may be incomplete or could have been removed during compilation. We, thus, design our own similarity metric that is resilient to common code transformations.

Algorithm 4: Building a minimal AST from a method body

Input : Method Body
Output : Abstract Syntax Tree (AST)

```
1  AST ← createRootNode();
2  foreach instruction ∈ method body do
3  |  Skip instruction unless opcode in {INVOKE_DIRECT, INVOKE_VIRTUAL};
4  |  instructionNode ← createNode(instruction.opcode);
5  |  foreach parameter ∈ instruction do
6  |  |  parameterNode ← createNode(parameter.type);
7  |  |  instructionNode.addChild(parameterNode);
8  |  end
9  |  AST.addChild(instructionNode);
10 end
11 return AST
```

AST Vectors

We derive an AST vector by first generating an AST and then transforming this tree to a vector. One advantage of working with feature vectors instead of full ASTs is the fact that comparing methods becomes substantially simpler than detecting graph isomorphism or computing the tree edit distance [Ngu+09].

Algorithm 5: Conversion of an AST to an AST vector

Input : Abstract Syntax Tree
Output : Abstract Syntax Tree Vector

```
1  vector ← createVector()
2  //count horizontal features
3  foreach invokeStmtNode ∈ AST.getChildren() do
4      #locals ← |{c ∈ invokeStmtNode.getChildren() | c.type = local}|
5      #params ← |{c ∈ invokeStmtNode.getChildren() | c.type = param}|
6      vector[local_local] ← (#locals 2)
7      vector[param_param] ← (#params 2)
8  end
9  //count vertical features
10 foreach lvl1Node ∈ AST.getChildren() do
11     increment(vector[lvl1Node])
12     foreach lvl2Node ∈ lvl1Node.getChildren() do
13         increment(vector[lvl2Node])
14         increment(vector[lvl1Node, lvl2Node])
15     end
16 end
17 return vector
```

As demonstrated in Algorithm 4, we start by building a minimal AST over a method body. Starting at the top node of a tree (line 1), we iterate over all program statements contained in the method (line 2) and filter instructions of the type `INVOKE_DIRECT` and `INVOKE_VIRTUAL` (line 3). We focus on these two as they are the most commonly used method invocation calls, according to Potharaju et al. As a next step, we create an AST node for the current method call (line 4) and attach a child node for each method parameter (lines 5-8). Finally, we add the instruction node as a child to the tree root (line 9).

Having derived an AST, we transform it into an AST vector by counting *horizontal* and *vertical features*, as introduced by Potharaju et al. [Pot+12]: A *horizontal feature* is a pair of leaf nodes with the same parent node, whereas a *vertical feature* is a directed path of arbitrary length, starting at the root node.

As shown in Algorithm 5, starting with an empty vector (line 1), we first identify horizontal features by traversing all first-level nodes of the AST and check the number of leaf pairs in each node (lines 3-8). For each invocation call, we count the number of *local registers* and *method parameters* (lines 4-5), and also store the number of pairs of types `local-local` and `param-param` in the AST vector (lines 6-7). Hereby, we leverage the fact that finding the number of possible pair combinations from a set of n local registers or method parameters is equivalent to computing the binomial coefficient with $k = 2$. Subsequently, we determine all vertical features by traversing all first-level nodes of the AST again (lines 10-16) and for each, increment its the occurrence count. (line 11). Finally, we also iterate over all child nodes of the current node and remember occurrences of both paths, i.e., we separately increment and store second-level relations as well as edges in the tree that connect first- with second-level nodes (line 12).

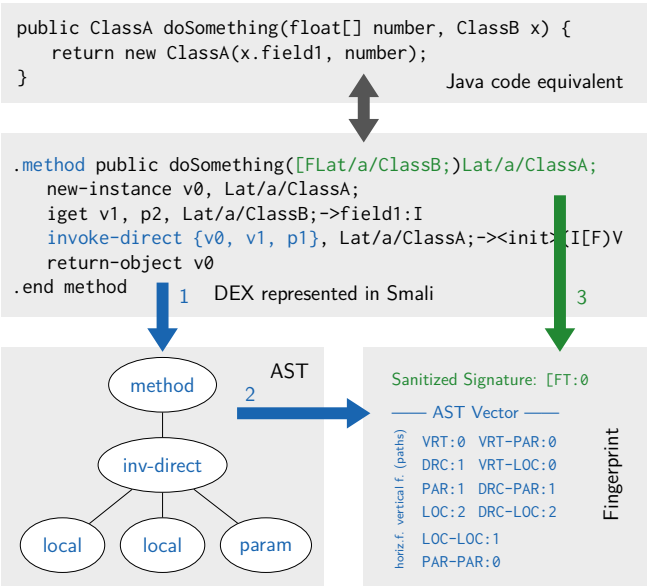


Figure 7.1: Fingerprint generation example.

Sanitized Signature

Sanitized signatures are built from data in type signatures that cannot be altered by code transformations. To obtain obfuscation-invariant signatures, we omit access modifiers and method names, and prune identifiers of parameters and return types. For primitive types, we keep the abbreviations as used in Smali[3]. If non-primitive types, such as classes or interfaces, are used as parameters or return types, we substitute the arbitrarily named object with a single character code: If the type equals the local class, i.e., represents a reference to itself, we assign the placeholder `T`. Likewise, if a class is affiliated with the same package as the current class, we assign `0` and, otherwise, we set `E` to denote external origin.

Fingerprints in Package Hierarchies

Fingerprints are built by combining AST vectors and sanitized signatures. While a single fingerprint can already be used to match individual methods, it does not necessarily carry enough semantic information to unambiguously identify classes and packages. E.g., due to their general design, *getter* and *setter* methods share similar fingerprints, which makes it hard to associate them with a specific class.

Instead of matching a maximum of individual methods, we propose to identify classes and packages by aggregations of fingerprints. Even in case that two unrelated methods exhibit an identical fingerprint, it is unlikely that two unrelated classes also share all method fingerprints. Likewise, we can assume that two packages are similar or equal if the contained classes share the same fingerprints.

[3]https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields

Example

Figure 7.1 shows a complete example of a fingerprint generation. Our objective is to build a fingerprint for a method that is referred to as `doSomething`, located in a class named `ClassB` that belongs to the package `at.a`. Therefore, we derive an AST vector and a sanitized signature from a representation of Dalvik bytecode in Smali format, as shown in the middle box. For a better understanding, the top box highlights an equivalent code snippet in Java. As marked by arrows, fingerprint generation involves the following steps:

1. **Convert method body to minimal AST:** We first create the top node `method` and add a node `inv-direct` that represents the instruction `invoke-direct {v0,v1,p1}`. The registers `v0,v1` used in the `invoke-direct` statement are local parameters. `v0` holds an instance of `ClassA` and `v1` an `Integer` value that is read from the instance field `ClassB.field1`. For the local registers `v0,v1` and the parameter `number` in `p1`, we create two `local` and one `param` node and assign them as children to the node `inv-direct`.

2. **Convert AST to AST vector:** *Vertical features* describe a sequence of nodes that are connected by a directed edge in the AST. In the given example code snippet, we count the following directed paths of length 1: `DRC:1`, since there is only a single `inv-direct` node in tree; `LOC:2`, `PAR:1` because of the child nodes for two local registers and one method parameter; `VRT:0` as there exists no `invoke-virtual` node in the AST. Vertical paths of length 2 are `DRC-LOC:2` and `DRC-PAR:1`. The features `INV-LOC` and `INV-PAR` are set to zero, due to the lack of corresponding nodes. Finally, we determine *horizontal features* by counting the pairs of `local` and `param` nodes. There is one `local` pair that can be mapped as `LOC-LOC:1` and no `param` pair.

3. **Generate sanitized signature:** The signature is derived from the method parameters `[F`, describing an array of floating-point numbers, `at/a/ClassB`, an instance `ClassB`, and an instance of `ClassA`, specified as a return type. We keep `[F` as-is and substitute `at/a/ClassB` with character `T`, as the object identifier refers to the local class. To distinguish method parameters from the return type, we insert a colon and substitute `at/a/ClassA` with the character code `0`, as the affected class is located within the `at/a` package. As a result, we obtain the the sanitized signature `[FT:0`.

4. **Form fingerprint from AST vector and sanitized signature**

7.3.2 Recognizing Code

In the recognition phase, a given package hierarchy P_a, including an arbitrary amount of classes and methods, should be matched with previously learned code. We compute fingerprints for all packages $p_a \in P_a$ and pursue the following steps:

1. All fingerprints of classes and methods in p_a are ordered by their *significance*. This helps to quickly identify those code parts that carry most semantic information. An explanation of significance is provided below.

2. Each fingerprint, composed of an AST vector and a sanitized signature, is looked up in the database of previously learned fingerprints. In case we encounter package hierarchies that exhibit similar fingerprints, we assign them to a candidate set P_C.

3. For all potential matches found in the database, we test whether $p_a \subseteq P_C$, i.e., if the code package is *included* within the previously learned package.

4. If the package p_a is indeed included within P_C, we compute the extent of *similarity* $s(p_a, P_C)$. If AST vectors in p_a and P_C are related, they are assigned a positive score, or zero otherwise.

5. Potential matches are ordered by their similarity score. The package that yields the highest score and appears to be a confident match, is returned as a recognition result.

Fingerprint Significance

As methods can be of arbitrary length and their signature may or may not include parameters and non-primitive return types, AST vectors and sanitized signatures sometimes cannot capture enough semantic information to form fingerprints that are clearly distinguishable from each other. Especially shorter methods that take no arguments and return `void` can lead to rather general fingerprints that are more likely match with unrelated methods, resulting in *false positives*.

To approximately express the amount of semantic information a fingerprint carries, we assign it a score that we derive from the length of both features. Thereby, we make the simple assumption that longer features are also more expressive. Let $m = (v, s)$ be a method fingerprint formed from an AST vector v and a sanitized signature s. The *significance score* of m can then be defined as:

$$\text{score}(m) := w_v \cdot \|v\|_1 + w_s \cdot \text{strlen}(s) \tag{7.1}$$

We quantify v using its L1-norm $\|v\|_1$ representation and s by the number of characters contained in the string, as retrieved via the function strlen(s). Both values are weighed with regard to their frequency of occurrence.

Inclusion

We apply the inclusion relation $\subseteq$ to denote when a code package p is a subset of package p'. In contrast to checking for strict equality, this enables us to also match packages where code fragments are missing, e.g., due to different versions of particular library or compiler-driven elimination of dead code. Formally, this means that a package p can only be contained within a package p' *if and only if* there exists an injective mapping f_c for each class in p to a class in p':

$$p \subseteq p' \Leftrightarrow \exists f_c : p \mapsto p',\ f_c \ldots \text{ injective.} \tag{7.2}$$

For all classes in f_c, we add constraints regarding fingerprints of contained methods. Assuming $c \in p$ to be a class c in a package p and $c' \in p'$, we have:

$$f_c(c) = c' \Rightarrow c \subseteq c'. \tag{7.3}$$

Analogous to packages, we can use the inclusion relation to express when a class c is included within another class c'. This is the case, *if and only if* there exists an injective mapping f_m for all methods m in c to methods in c':

$$c \subseteq c' \Leftrightarrow \exists f_m : c \mapsto c',\ f_m ... \text{ injective.} \tag{7.4}$$

However, we require that a mapping f_m for a method $m \in c$ to fingerprint of a method $m' \in c'$ is only possible if their sanitized signatures match exactly:

$$f_m(m) = m' \Rightarrow \text{Sanitized signatures of } m \text{ and } m' \text{ are equal.} \tag{7.5}$$

By formally defining these inclusion relations, we have set the dependencies for methods to be contained in classes, and classes to be included in packages. In the following, we explain how to implement these relations for classes and packages.

Class Inclusion

Let $c = \{t_1, ..., t_n\}$ be a class comprising t_n fingerprints that are built from AST vectors and sanitized signatures. We can evaluate whether c is contained within class c' by following a *greedy* lookup strategy, as outlined in Algorithm 6. The overall idea is to match each fingerprint contained in c with its counterpart in c', unless a mismatch occurs. Upon a successful match, we remove t_i from the set of fingerprints in c', as it is no longer a potential candidate. If all fingerprints from c are found in c', the relation $c \subseteq c'$ is fulfilled.

Algorithm 6: *Greedy* check if class c is included in c'

Input : Class c, Class c'
Output : true if $c \subseteq c'$

```
1 foreach t_i ∈ c do
2     if t_i ∈ c' then
3         Remove t_i from c'
4     else
5         return false
6     end
7 end
8 return true
```

Package Inclusion

To check whether package p is contained within a package p', we need an injective mapping f_c for all classes in p to classes in p'. Due to the missing symmetry in the class inclusion relation, i.e., p' is not required to have a counterpart for each class in p, a greedy inclusion check is not feasible in this case.

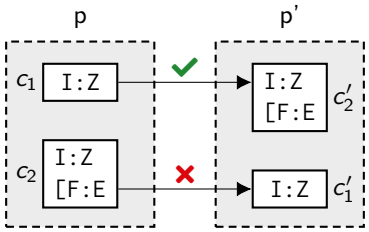


Figure 7.2: Failing package inclusion using a *greedy* strategy.

Figure 7.2 illustrates an example mapping of classes, where a greedy inclusion check, as shown in Algorithm 6, would fail. Assuming two packages p and p', it can be seen that p includes p', as there is an injective mapping f_c with $f_c(c_1) = c'_1$ and $f_c(c_2) = c'_2$. In this case, a greedy inclusion lookup would fail, since class c_1 can also be matched with c'_2, since $c_1 \subseteq c'_2$. However, if we would match c_1 with c'_2, class c_2 would not be assignable at all due to the fact that c_2 is not in $c'1$.

For situations where an assignment is not unambiguously possible, we want to find a mapping between packages such that a maximum of contained classes is matched. To solve this combinatorial optimization problem, we can leverage the *Hungarian algorithm* [Kuh10]. Therefore, we build a cost matrix M over all possible combinations of classes in two packages and set the element, i.e., the cost of assignment to 0, if a class is contained within another one, or 1 otherwise:

$$M \in \{0,1\}^{|p|\times|p'|},\ M[i,j] = \begin{cases} 0 & \text{if } c_i \subseteq c'_j \\ 1 & \text{otherwise} \end{cases} \tag{7.6}$$

After applying the algorithm on matrix M, we obtain an assignment mapping f_c. As costs should be minimized, the algorithm prefers assignments that cost 0 to those that cost 1. The total costs are, thus, the sum of non-matching classes:

$$\text{cost}(M, f_c) := \sum_{i=1}^{|p|} M[i, f_c(i)] \tag{7.7}$$

If assignment costs are 0, there exists an injective mapping from p to p':

$$\text{cost}(M, f_c) = 0 \Leftrightarrow \exists f_c : p \mapsto p',\ \ f_c\text{... injective.} \Leftrightarrow p \subseteq p' \tag{7.8}$$

7.3.3 Similarity Score

To recognize fingerprints of different code fragments, methods, classes, and packages in a given Android app, we have to match them against the set of previously learned candidates. As structural changes at, e.g., basic block-level can affect fingerprints at method-, class-, and package-level, searching only for exact matches is no viable option. To evaluate the extent of similarity between two arbitrary packages hierarchies p and p', we introduce a score that numerically expresses the distance between two AST vectors. In the following, we explain how to measure similarity for the different fingerprint entities.

AST Vectors

We can describe the similarity between two AST vectors v and v' by the distance of their coordinates in the vector space. As a measure, we rely on the L1-norm that is calculated from the sum of the absolute length values of two vectors.

$$s(v, v') = \max(\|v\|_1 - \|v - v'\|_1, 0) \tag{7.9}$$

Equation 7.9 satisfies the following requirements:

- We expect the similarity score to be zero if AST vectors are not close to each other, i.e., the difference between them exceeds the length of the vector.
- Similarity must not fall below zero. We ensure this condition by selecting the maximum value among the absolute vector distance and zero.
- We expect similarity to be maximized when both AST vectors are equal. In such a case $\|v - v'\|_1$ becomes zero, causing the score to be $s(v, v') = \|v\|_1$.

Classes

We evaluate similarity between two classes $s(c, c')$ using the Hungarian algorithm, as it allows us to find matches between AST vectors with maximum similarity. Let $c = \{m_1, ..., m_n\}$ be a class with a set of fingerprints of methods m_i, whereas each is formed from an AST vector v_i and a sanitized signature s_i. We build a cost matrix C that holds the similarity scores of all possible AST vector combinations:

$$C \in \mathbb{R}^{|c| \times |c'|},\ C[i, j] = \begin{cases} s(v_i, v'_j) & \text{if } s_i = s'_j \\ 0 & \text{otherwise} \end{cases} \tag{7.10}$$

By design, the algorithm minimizes the cost of assignments. As our goal is to find assignments that yield the maximum cost, i.e., the highest similarity scores among AST vectors, we solve the problem by negating and shifting the matrix:

$$C_{\text{negated}} = (\max(C) - C[i, j])_{ij} \tag{7.11}$$

After applying the algorithm, we can use the resulting mapping f_c to compute a package similarity score using the $\text{cost}(C, f_c)$ function, as defined in 7.7.

Packages

Likewise, we can apply the Hungarian algorithm to assess similarity between two code packages $s(p, p')$ and find pairs of classes with maximum similarity. Therefore, we build a cost matrix P and fill it with the similarity scores $s(c, c')$ of all classes c and c' respectively, that are included in the two packages p and p':

$$P \in \mathbb{R}^{|p| \times |p'|},\ P[i, j] = \begin{cases} s(c, c') & \text{if } c_i \subseteq c'_j \\ 0 & \text{otherwise} \end{cases} \tag{7.12}$$

We negate P analogous to Equation 7.11 and run the algorithm. To finally compute the similarity score $s(p, p')$, we again use $\text{cost}(P, f_c)$ from Definition 7.7.

7.4 Evaluation

The goal of this evaluation is twofold. First, we investigate how well AST vectors and sanitized signatures fingerprint code and tackle obfuscation (see Section 7.4.2). Second, by applying our solution on open-source Android applications, we assess *(1)* how much code of a package is needed to reliably recognize it (see Section 7.4.3), *(2)* how much confidence a match should have to be significant (see Section 7.4.4), and *(3)* how well we can recognize individual packages when different obfuscation techniques are applied (see Section 7.4.5).

7.4.1 Method and Dataset

In the **learning phase**, we seek to assign similar fingerprints to semantically similar code fragments. Distinctive features should characterize unrelated code. To better understand how well AST vectors and sanitized signatures identify code, as a first step, we test our features using a home-made app that is obfuscated, includes two libraries, and 150 packages. The results give an intuition on how changes in the codebase or parameters impact accuracy in matching code.

To evaluate how our solution can **recognize code** in real-world applications, we opted to crawl the *F-Droid* Repository[4]. Since this app repository offers only *Free and Open Source Software (FOSS)*, we can download the source codes of apps including configuration files for the Gradle build system. From these files, which are needed for compilation only but are not included in final Android application archives, we are able to extract a list of used library packages. On their basis, we split the code into different parts to derive fingerprints for each of them. Moreover, we adapted the build files of all downloaded FOSS apps to compile multiple versions with different code transformation techniques enabled: *shrunken, obfuscated, shrunken + obfuscated, shrunken + obfuscated + optimized.* Each version incorporates the same functionality but realizes it using code fragments that vary in terms of control and data flows, variable names, and the set of involves class methods. We leverage these builds to assess the extent to which our code recognition strategy is resilient to obfuscation and optimization measures. For this evaluation, we crawled source codes of 800 apps and compiled one regular and four transformed versions, resulting in 4,000 app samples overall.

The collected dataset enables us to unambiguously verify how well learned code can be recognized. Based on the assumption that FOSS apps exhibit the same code structure and perform library inclusion analogous to other apps, our results should also hold for arbitrary Android applications.

7.4.2 Fingerprint Quality

Our algorithm derives a summarized representation of methods, classes and packages by combining *AST vectors* and *sanitized signatures*. For these objects to be unambiguously recognizable by their fingerprints, it is essential that our features focus only on code attributes that are invariant to code transformations.

[4] https://f-droid.org

Question: *How well can our features describe individual code fragments?*

To assess the quality of fingerprints, we evaluate the degree of ambiguity that emerges from applying our techniques to average code samples. As a reference, we build a confusion matrix $M = (m_{ij})$ that depicts for how many distinct packages of an average Android library, we are able to derive uniquely assignable fingerprints. At the same time, it is suited to visualize incorrectly recognized packages. Each row represents the label assigned to a package, each column stands for a single package. The color of a cell m_{ij} highlights the confidence that package i of an arbitrary app matches package j of the test package. We are able to interpret the quality of fingerprints from the structure of M: Only if the main diagonal is confident, we are able to identify code segments without ambiguity.

Setup

To build a confusion matrix, we use the sample set of 150 packages included in our obfuscated test application. The set is large enough to feature a variety of different packages and sufficient to visualize confusion. We compute the similarity score between each app package with each reference package we learned before.

Results

As illustrated in Figure 7.3, we derived three matrices that reveal confusion when only AST vectors or sanitized signatures are applied for code identification, and when both features are used in combination. The x- and y-axes are ordered by the significance of fingerprints that serves as a measure for expressiveness. This means that packages with a small significance score are shown on the top-left, whereas packages with high significance are to be found on the bottom-right.

With *AST vectors*, we see that code similarity is confident on the main diagonal but, overall, shows a high degree of confusion. The top-right segment of the matrix outlines many packages that were mislabeled with high confidence. This results from the fact that the code of small packages often re-occurs in multiple larger packages and, thereby, impedes an unambiguous attribution. Conversely, we observe that larger packages are not confused with smaller ones.

The confusion matrix for *sanitized signatures* reveals that the similarity measure is either confident that a code package is *contained* (see Section 7.3.2) within another one, or not at all. In contrast to AST vectors, we observe only little confusion in the top-right of the matrix, where code packages with a lower significance score are compared with packages that exhibit higher scores. Nonetheless, similar to AST vectors, the amount of confusion remains high for small packages as their fingerprints are often also part of larger packages.

With both features combined, confusion is mostly eliminated aside from code packages with low significance in rows at the top. Smaller clusters near the main diagonal in the top-left area result from packages that implement similar interfaces and are, thus, semantically closely related to each other. Overall, we see that the combination of AST vectors and sanitized signatures can identify code almost without confusion and, thereby, paves the way for accurate recognition.

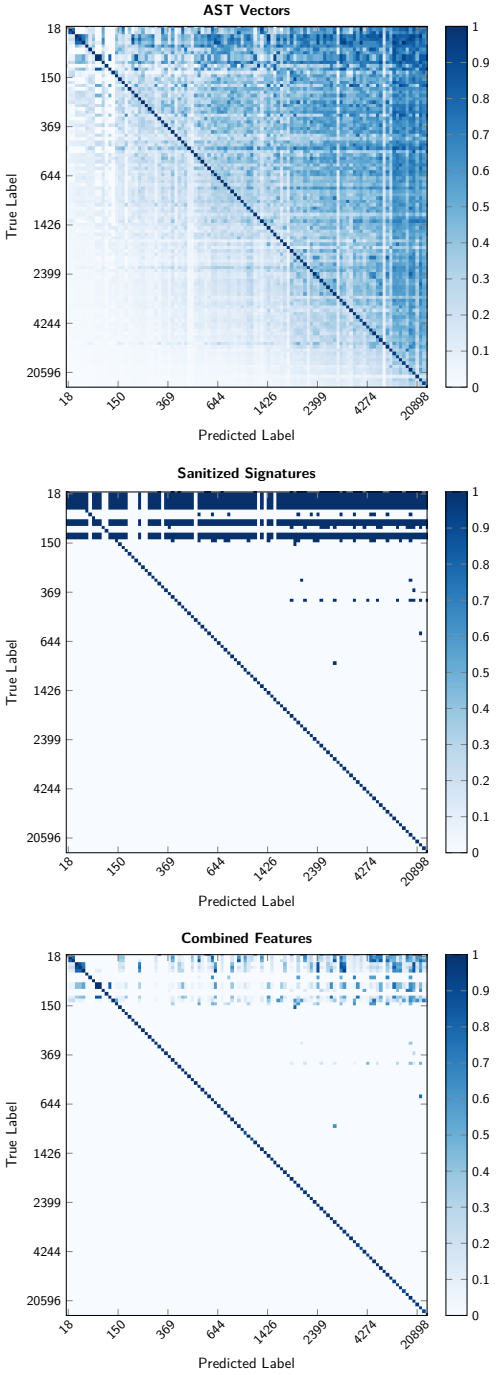


Figure 7.3: Confusion matrices based on AST vectors (top), sanitized signatures (middle), and a combination of both features (bottom).

7.4.3 Threshold for Package Significance

The confusion matrices for AST vectors and sanitized signatures combined showed that the remaining confusion was caused by packages with a low significance score. As a remedy for confusion and to increase accuracy, we want to define a lower bound for the *minimum package significance* score t_{ps}. It decides whether a particular package is sufficiently expressive to be used for learning and matching.

Before processing a code package, we verify its significance. If a package carries only little semantic information, we can ignore it assuming that matching would not be possible unambiguously. The higher we set t_{ps}, the more accurate recognition becomes. However, a high threshold also leads to more packages being ignored. In the best case this means that fingerprinting omits generic code parts, such as *getter* and *setter* methods, as well as others that consist only of a few lines of code. In the following, we define this impact on t_{ps} as the *keep ratio*:

$$\text{keep ratio} = \frac{|\text{Packages learned from app}|}{|\text{Total number of packages in app}|} \tag{7.13}$$

Our objective is to determine a reasonable value for t_{ps} that represents a trade-off between recognition accuracy and keep ratio.

Question: *How much significance is necessary for precise recognition?*

Setup

We use our set of 800 real-world applications and iteratively try recognition with values between 0 and 200 for t_{ps}. After each round, we build a confusion matrix and compute the impact on accuracy and keep ratio.

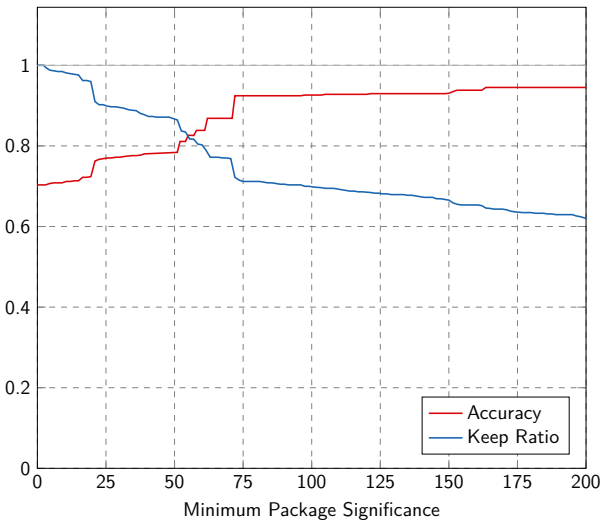


Figure 7.4: Influence of t_{ps} on accuracy and keep ratio.

Results

Figure 7.4 illustrates the effect of different t_{ps} values on accuracy and keep ratio. If almost no minimum package significance is required, code packages are barely dropped and the keep ratio remains close to 1. Recognition accuracy in the top-left area reaches 0.7, which implies that 30% of matches are wrong. The higher we set the threshold for the minimum package significance, the more code packages with a low score are ignored and, at the same time, accuracy increases. This observation reaffirms that it is inevitable to ignore insignificant packages, as otherwise low quality packages would hamper a precise recognition overall. At a package significance value of 75, accuracy attains 0.9 and remains steady, whereas the keep ratio continues to decrease, as long as the threshold is elevated.

As also seen previously in Figure 7.3, confusion increases below a particular package significance score but tends to decrease again if a certain upper bound is exceeded. Consequently, we see that a t_{ps} value of 80 delivers the highest accuracy without ignoring more packages than necessary. With $t_{\text{ps}} \leq 80$, accuracy is sacrificed for matching code with packages that are insignificant for recognition.

7.4.4 Threshold for Matching Confidence

As explained in Section 7.3.3, we describe the similarity between an arbitrary code package p_a and a comparison object p_b with the *similarity score* $s(p_a, p_b)$. These scores are built upon the similarity of a set of fingerprints that are derived from classes and methods contained within a corresponding package. As each set can comprise fingerprints of varying significance, it is challenging to decide on what makes a recognition result reliable. If we compare two packages that include only classes that are highly individual by themselves, i.e., their fingerprints exhibit high significance, but they match the reference object due to true similarity, then $s(p_a, p_b)$ indicates a high score when comparing these two packages. On the other hand, if two packages contain mostly classes with low significance, they are more easily matched with unrelated candidates, resulting in false positives.

This bias is problematic when it comes to recognizing semantically related code. In practice, this issue occurs when matching differently obfuscated code fragments with each other. Changed method signatures, inlining, shrinking, and other transformation techniques can negatively affect fingerprint significance and, in the worst case, cause a package to be ignored from matching overall. By setting the required minimum package significance to a constant t_{ps} value, we prefer packages with high scores but disregard others with low significance, although they might be similar. As a countermeasure, we assign each recognition match a *confidence* value that we derive from package similarity:

$$\text{confidence}(p_a, p_b) = \frac{s(p_a, p_b)}{s(p_a, p_a)} \tag{7.14}$$

The resulting value is between zero and one because $0 \leq s(p_a, p_b) < s(p_a, p_a)$. Our objective is to determine a reasonable threshold t_{mc} that can indicate whether a recognition result is sufficiently robust to be considered as a plausible match.

Question: *How much confidence makes a recognition result reliable?*

Setup

To determine the optimal value for t_{mc}, we use our sample set of FOSS apps and transform the multi-class problem into a binary classification problem by following a one-vs.-all strategy [Bis07, p. 182–184]. We define our binary classifier to output whether a given code package has been learned and is, thus, known (*positive*, *+*) or unknown (*negative*, -) to the system. Each match is reduced into the binary classification problem by expressing learned code packages as *positive*, and all others, e.g., unlearned packages or code packages, as *negative*.

With our recognition matches transformed into binary classification results, we can visualize the performance of our multi-class model using *Receiver Operational Characteristics* (ROC) curves. This enables us to disclose how the matching threshold for confidence impacts both true and false positive rate. ROC curves, thereby, illustrate the separability of known and unknown code packages and assist in identifying a reasonable threshold value t_{mc} for matching confidence. We repeat this transformation for all app samples to determine how well we can distinguish known from unknown packages if code transformations are used.

Results

Figure 7.5 summarizes the ROC curves for different build configurations. As can be seen, the classifier distinguishes known from unknown code packages with high accuracy, even when code transformation techniques, such as obfuscation and shrinking are enabled. For these app samples, the *Area Under the Curve* (AUC) exceeds 99.5%. The only type where our binary classifier outputs less optimal results is a configuration involving obfuscation, shrinking and code optimizations. Nonetheless, the AUC for this type of apps is still sufficiently high at 87.7%.

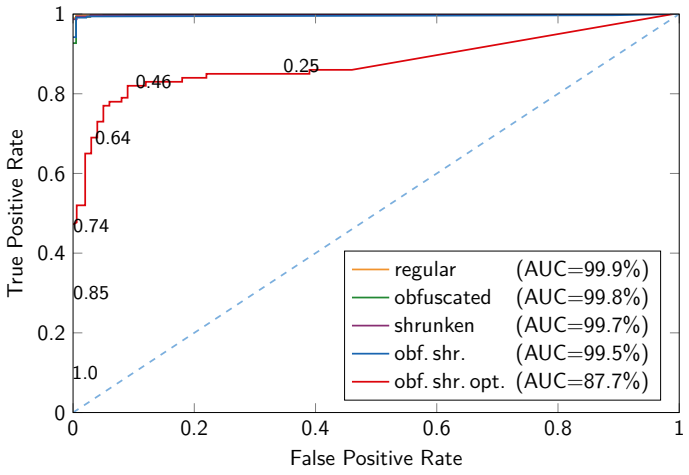
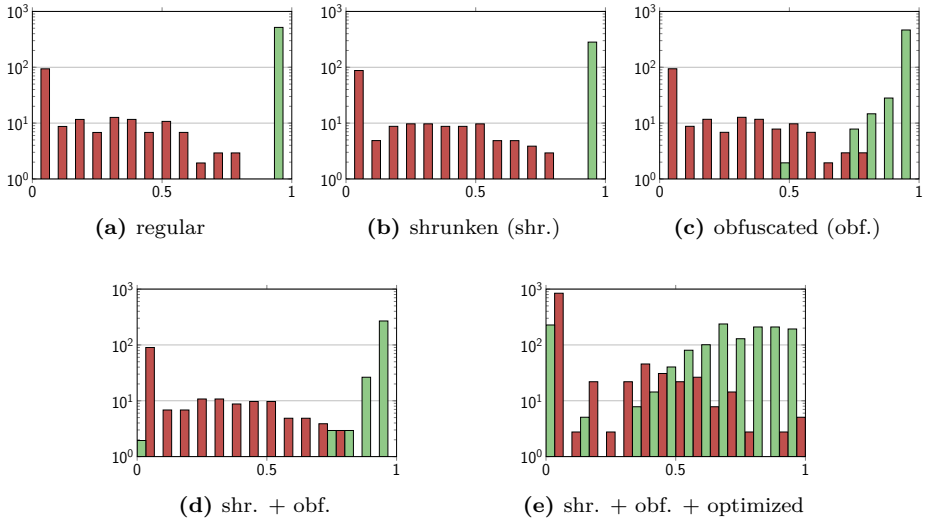


Figure 7.5: ROC curves of apps with different code transformations applied.



(a) regular **(b)** shrunken (shr.) **(c)** obfuscated (obf.)

(d) shr. + obf. **(e)** shr. + obf. + optimized

Figure 7.6: Confidence histograms for known (green) and unknown (red) packages.

For each build type, Figure 7.6 presents how known and unknown recognition results are distributed according to matching confidence. The red bar indicates the frequency of unknown packages, the green bar shows confidence for known packages. As confidence(p_a, p_b) tends to be zero or one, i.e., recognition results are either very reliable or not at all, we use a logarithmic scale for the y axis.

We see that *regular* and *shrunken* code packages can be distinguished very well at $t_{\text{mc}} = 0.8$. With *obfuscated* builds, class separability is still ensured as there is almost no intersection among the distributions. Similar conditions apply to samples with a *shrunken + obfuscated* configuration. The histogram for builds with all possible transformations enabled confirms that they are the most challenging to separate. According to the ROC curve, a matching confidence value near 0.5 works best for this configuration to optimize recognition accuracy.

7.4.5 Code Recognition

In Section 7.3.2, we elaborated a workflow to recognize learned code by computing fingerprints of methods, aggregating them in package hierarchies and testing for similarity with known packages.

Question: *How well does our approach recognize app packages?*

Setup

For this scenario, use our FOSS sample set and analyze all applications in all build configurations. Aimed at highest recognition accuracy, we set the thresholds for matching confidence t_{mc} to 0.5 (see Section 7.4.4) and for package significance t_{ps} to 80 (see Section 7.4.3). All matches are evaluated by applying multi-class performance metrics [SL09] to all known packages, as shown in Figure 7.7.

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{l} tp_i \qquad \text{Precision}_{\text{M}} = \frac{1}{l}\sum_{i=1}^{l} \frac{tp_i}{tp_i + fp_i}$$

$$\text{Recall}_{\text{M}} = \frac{1}{l}\sum_{i=1}^{l} \frac{tp_i}{tp_i + fn_i} \qquad \text{F}_1\text{score}_{\text{M}} = 2 \cdot \frac{\text{precision}_{\text{M}} \cdot \text{recall}_{\text{M}}}{\text{precision}_{\text{M}} + \text{recall}_{\text{M}}}$$

Figure 7.7: Multi-class classification measures [SL09] for l known code packages p_i and n recognition matches. tp_i are true positives, fp_i false positives, fn_i false negatives and the M index indicates macro-averaging over all classes.

Based on a generalization of performance metrics that typically employed for binary classifiers, we assess the number of correctly recognized code packages (*true positives*), samples where packages were incorrectly matched with unrelated candidates (*false positives*) and packages that were not recognized although they were learned (*false negatives*). For each build type, we determine accuracy, precision, recall and evaluate the quality of the classification using the F_β-score.

- **Accuracy.** Determines how many code packages have been identified correctly with regard to the total amount of recognition matches.
- **Precision.** Ability of our solution to not assign wrong labels to packages.
- **Recall.** Ability to determine all instances of a code package.
- **F_β-score.** Harmonic mean of precision and recall.

Results

The recognition results are summarized in Table 7.1. As shown, all metrics perform well in all build types except for the set of *obfuscated, shrunken, and optimized* applications. By manually investigating classifications, we noticed that the optimization technique *Method Parameter Removal* causes the weaker performance with this build type (see Section 7.2.1). With this code transformation enabled, ProGuard prunes method signatures from unused parameters, leading to different sanitized signatures. Nonetheless, the use of AST vectors ensures that recognition remains feasible for the majority of packages.

Table 7.1: Code recognition performance on real-world apps.

	regular	**obfuscated**	**shrunken**	**obf.,shr.**	**obf.,shr.,opt.**
Accuracy	96.76%	96.61%	93.30%	93.40%	78.83%
Precision	98.03%	97.80%	94.81%	94.69%	70.64%
Recall	99.15%	98.92%	97.05%	96.80%	71.95%
F1 Score	98.17%	97.94%	94.94%	94.80%	70.32%

7.4.6 Summary

We examined how AST vectors and sanitized signatures align with real-world Android applications that apply code transformations. First, we assessed how well our techniques can fingerprint obfuscated code fragments. Three confusion matrices revealed that although each of our techniques is capable of identifying obfuscated code by itself, the quality of fingeprints and matching results are significantly improved when both features are used in combination.

We also noticed that most confusion occurs in code packages with a low significance score. Therefore, we introduced a threshold t_{ps} value that indicated how much code was relevant to keep high accuracy high while not ignoring too many packages that carry only little semantic information. We also proposed a matching confidence threshold t_{mc} to decide on what makes a recognition result reliable. To find a reasonable value for t_{mc}, we reduced the problem of multi-class classification to multiple binary classification problems. ROC curves underlined the extent to which we could distinguish known from unknown packages despite code transformation techniques being applied. In our final study, we tested code recognition with a set of Android app samples and found that our solution delivers high values for accuracy, precision, recall, and F-score in all scenarios.

7.5 Conclusion

The use of third-party libraries and the integration of code snippets from public sources have become common practices in Android application development. Security issues and vulnerabilities in such components reach a high number of end-users and put sensitive data at risk. However, a precise recognition of such program parts is challenging if code transformation techniques are applied.

We proposed a solution that can reliably recognize code packages comprising individual snippets or entire libraries, even if obfuscation, shrinking, or similar techniques are used. By extracting fingerprints from the Abstract Syntax Tree of methods and combining them with obfuscation-resilient features of method signatures, we succeed in accurately characterizing code. We thoroughly evaluated the applicability of our technique and demonstrated that can we describe and recognize arbitrary code fragments with high precision and recall.

The elaborated techniques help to obtain a representation of source code that cannot be tainted by code transformation techniques. Usable in conjunction with single methods, classes, as well as large-scale package hierarchies, we presented a general purpose solution to reliably recognize code. Possible practical use-cases range from fingerprinting and searching individual code snippets that are known to be harmful, to the detection of entire libraries that potentially include outdated or vulnerable code. In the context of this thesis, our work represents a notable step towards distinguishing actual implementation behavior from low-level code semantics. The ability to identify specific transformation-invariant patterns in source code is crucial on our way to model the behavior of Android applications.

8

Identifying Differences Among Android Applications

Android applications often receive updates that introduce new functionality or tackle problems, ranging from critical security issues to usability-related bugs. Although developers tend to briefly denote changes when releasing new versions, it remains unclear what has actually been modified in the program code. Verifying even subtle behavioral changes between two given applications is challenging due to the widespread use of code transformations and obfuscation techniques.

In this chapter, we present a multi-level comparison strategy to distinguish functional changes by developers from structural compiler modifications. We start by motivating our research in Section 8.1 and discuss existing solutions for code comparison. Followed by that, in Section 8.2, we introduce a new approach to find similarities and differences in the code of two given Android applications. In Section 8.3, we elaborate our matching process at class, method, and basic block level in more detail. In a case study in Section 8.4, we demonstrate the practical applicability by verifying how updates have been deployed to fix security issues in real applications. Parts of this chapter are taken verbatim from [FNZ19].

Publication Data and Contribution

Johannes Feichtner, Lukas Neugebauer, and Dominik Ziegler. "Mind the Gap: Finding What Updates Have (Really) Changed in Android Applications." In: *Security and Cryptography – SECRYPT 2019*. SciTePress, 2019, pp. 306–313. DOI: 10.5220/0008119303060313

Contribution: Main author; Prototype implemented by Lukas Neugebauer. Dominik Ziegler contributed to the explanation of class matching.

8.1 Introduction

Developers regularly distribute and update their Android applications via Google Play or third-party distribution channels. Featured by descriptions, screenshots, and further promotional information, users can pick from a large pool of often similar applications. Many vendors reuse their own code and offer multiple revisions of the same app for different devices or with adjustments, e.g., for learning various languages, local weather, and city travel guides. While these applications are usually clearly distinguishable by their visual appearance, the opposite is the case when third-party developers distribute repackaged versions of existing apps with barely noticeable adaptations. These changes often introduce malware [Tia+20; Lin+16] or code to hijack revenues for advertisements [ZZT19].

Upon the release of new app versions, developers can provide a changelog that should include a list of modifications, fixed issues, and new functionality. In practice, these descriptions are often kept rather generic, stating, e.g., "stability improvements and bug fixes" instead of giving clear advice to marketplace maintainers and users of what precisely has been improved. Even if provided, release notes are not necessarily complete and accurate. For instance, if an author mentions that a known security-critical privacy issue has been fixed, users are to believe that all relevant program parts have carefully been reworked to mitigate the problem. Besides not being able to check whether indicated changes have indeed and thoroughly been implemented, additional code modifications that are not disclosed in the release notes are impossible to reproduce.

Whenever updates are published for Android applications, or apps appear to be repackaged versions or clones, it is crucial to see what has actually changed in the program code. To obtain an insight, marketplaces, such as Google Play, employ approaches for static and dynamic program analysis and check whether all uploaded applications comply with predefined security and privacy policies that are enforced by the distribution platform. Typically, this review process also involves machine learning in order to evaluate a collection of attributes from an app's code, metadata, and user reviews for signs of anomalous behavior and potentially harmful content [May+19]. While these measures are of significant value to counteract malware spread as cloned or repackaged applications, they inherently miss implementation weaknesses that do not infringe any protection policy, security rule or cause security-critical bugs, such as a wrong application of cryptography. The ability to isolate code differences between a previously tested application and an update candidate would add another valuable perspective on behavioral program changes with a possible impact on security aspects.

From a research perspective, there is a strong need for a solution that can reliably highlight differences among various app versions while distinguishing between functional changes by developers and structural adaptations by compilers. This involves an interpretation of the underlying program syntax and semantics. Especially, regarding the increasing complexity and size of today's apps, as well as the frequent use of code obfuscation techniques, a comprehensive comparison could help to lower the costs and efforts needed for application analysis and would allow for easier verification of security-critical bugfixes.

When comparing Android applications, we aim to disclose changes in the implemented program behavior, rather than finding stylistic or other non-functional changes without influence on execution or visual appearance. Manually identifying similarities and differences between two implementations is challenging due to the widespread use of code transformation techniques, including obfuscation, identifier renaming, and shrinking that are applied at compile time not only to optimize code but also to harden against reverse engineering and tampering. Automated solutions, on the other side, are usually targeted to return a binary decision on whether an app appears to be a cloned or repackaged version of another one [Che+15; Wan+15]. In very simple cases, such a conclusion can be drawn by checking if the majority of files of one app sample are also contained in the comparison object. However, if code transformation techniques are used, a content or hash-based comparison will lead to spurious results. The outcome will also be tainted if obfuscation and related techniques are not taken into account.

Comparing the code of Android apps involves finding a metric that assigns scores to implementation differences. Relying on a static or dynamic analysis approach, related work for similarity analysis extract features from metadata, such as permissions or reviews, code, e.g., as call graphs or instruction subsets, or obtain them during runtime from execution traces. Existing state-of-the-art efforts concentrate on detecting repackaged or cloned apps based on heuristics but are unable to highlight individual code parts that are similar or different between two Android applications. Mostly operating on the Dalvik bytecode of apps, these work derive a verdict based on a custom similarity metric that yields accurate but often irreproducible results [ZZT19; Gua+16]. Other approaches for similarity-based analysis, as implemented by *DroidSD* [Akr+20], *FSquaDRA* [Zha+14b], and solutions for malware detection [Tah+20; HKC19], deliver a deterministic score but give no explanation on how and where code relates to each other.

The reverse-engineering framework *AndroGuard*[1] first included an algorithm for pairwise comparison, entitled *Normalized Compression Distance* [Des12]. By comparing hashing-based fingerprints of methods, a normalized value between 0 and 1 was derived to indicate the extent of similarity. Variants of this approach focus on dependency graphs [CGC12], layout information [SLL15], or combine different features [Sha+14b; Zha+14a] to also address code obfuscation. In order to improve the scalability of pairwise comparisons, other approaches summarize extracted features in vector representations. For example, instead of operating on code, *PiggyApp* [Zho+13] fills vectors with semantic information and uses them to compute the distance between apps. For faster comparisons, abstract code parts can also be organized in graph-like representations [CLZ14; DNL14].

Among the various approaches for similarity analysis, the most significant differences are the algorithm used as a distance metric, as well as the features fingerprints are derived from. Although the detection of repackaged applications is conceptually related, existing work does not *(1)* tackle the problem of uncovering code differences instead of deriving a summarizing similarity score and *(2)* cannot distinguish functional program changes from compile-time modifications.

[1] https://github.com/androguard/androguard

The ability to highlight concrete code similarities and differences among Android applications is essential in order to understand what has really been changed by updates and in repackaged versions. In addition, a reliable comparison can provide valuable insight for analysts and curious users into how developers have handled security-critical implementation weaknesses.

For this to achieve, we design a framework that enables a pairwise comparison of the code contained in two given Android applications. We implement similarity checks at file, class, method, and basic block level and present code differences in a format that is well-known from source code versioning systems, such as Git or Subversion. As a comparison strategy, we propose an iterative solution that takes compiler peculiarities and code transformation techniques, such as shrinking and identifier obfuscation into account. Therefore, we extract different representations of basic blocks from Dalvik bytecode using an adapted version of the *baksmali*[2] reverse-engineering tool. To make code elements comparable despite obfuscation and dynamic compiler decisions, we derive segments where registers, labels, and re-arrangements cannot influence similarity matching. With this strategy, we excel in detecting matching pairs of code blocks and outperform existing state-of-the-art solutions for similarity analysis. To demonstrate the practical applicability, we perform case studies on real-world applications and verify how updates have been deployed to fix security-critical issues.

To identify and extract semantic code differences between two given Android applications, we propose a multi-level comparison strategy:

1. We model the hierarchy of packages with code and resources in Merkle trees and prune parts that are identical among the two versions.

2. From the remaining code, we extract and process all classes, methods, and basic blocks. While preserving the original code semantics and data flow, we perform multiple rounds of comparison that accurately identify newly added, changed, moved, and deleted code elements.

3. We tackle common code transformation techniques by comparing only selected features that are invariant to obfuscation and arbitrary compiler modifications.

We implement an analysis framework that can be used for various purposes involving the direct comparison of two Android applications. Considering the fact that nowadays applications comprise thousands of classes, the choice of a data structure for an efficient pairwise comparison of files is crucial. By using Merkle trees, we are able to precisely identify classes and packages that are identical among two versions. In a case study, we highlight the practical applicability of our solution by verifying the developer-provided changelog of real-world applications. We also demonstrate that our tool cannot only help to identify repackaged apps but also to precisely isolate code that might have been added or modified by malware authors.

[2] https://github.com/JesusFreke/smali

8.2 System Design

We design a framework to list differences and similarities between two given Android app archives. As depicted in Figure 8.1, we first convert the extracted Dalvik bytecode of both comparison objects into Smali code, derive a hash of each object, and organize the resulting hierarchy of files and directories as Merkle trees. Considering the large amount of code in today's applications, we leverage this data structure to quickly filter classes and packages that are equal among both applications. For files without a matching candidate, we continue with a more fine-grained comparison at class, method, and basic block level (see Section 8.3).

In a multi-round approach, we rewrite the Dalvik bytecode of both apps and produce different code representations that mitigate possible compiler peculiarities or effects of code obfuscation techniques in interfaces, classes, or methods. Due to the wide range of possibilities how code can be transformed, we sequentially elaborate semantically equivalent code fragments that address distinct aspects of why files can differ. By iteratively comparing the various representations, it is possible to describe how apps are different from each other and to finally highlight precisely what code segments have been added, modified, or removed.

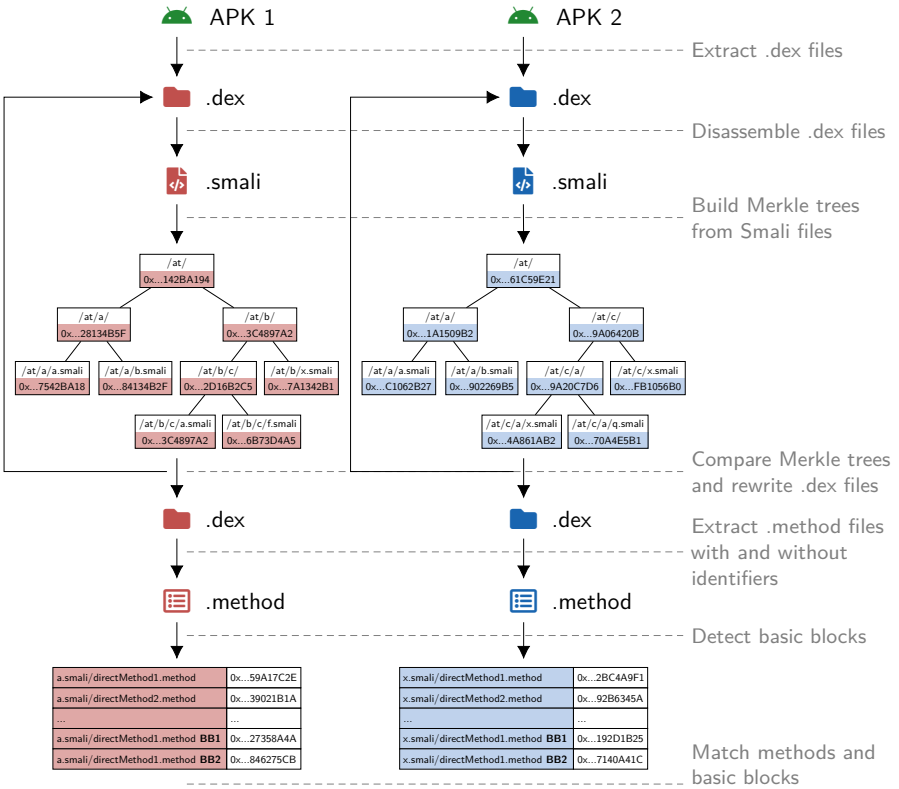


Figure 8.1: Iterative comparison of two given Android applications.

8.3 Code Similarity in Android Apps

Our overall concept is based on comparing Smali code to detect differences and similarities. The idea is to iteratively split parts that do not match between two applications into smaller chunks. In each step, we gradually reduce the number of identifiers, substitute them with placeholders, and look for matching candidates in the comparison object. After an initial preprocessing (see Section 8.3.1), we pursue a repeated comparison on class level (see Section 8.3.2) and filter matching classes. Subsequent comparisons on method and basic block level (see Section 8.3.3) highlight more and more fine-grained implementation differences.

8.3.1 Preprocessing

As a first step to reliably determine what code fragments are shared among two applications, we filter semantically superfluous information that have no effect on the control or data flow during execution. These *preliminary operations* also ensure that the hashes of files generated in subsequent steps cannot be tainted by trivial aspects, such as changed labels in `goto` instructions, debug information or the order of fields and methods:

- **Removing debug information:** Debug information typically has no impact on code execution. Still, it might differ for two identical versions of a file, depending on the used compiler and build settings. In a preliminary step, we strip debug information, such as line numbers and parameter names from Smali code, derived from Dalvik bytecode using the tool *baksmali*.

- **Implicit method and field references:** When generating Smali code, we rely on implicit method and field references for elements of a current class. Instead of prepending the current class name as a prefix for method and field names, we omit this reference. E.g., `La/b;->a:Ljava/lang/Object;` will be rewritten as `a:Ljava/lang/Object;`. The reduction is important to identify identical code snippets in differently named classes and packages.

- **Sequential numbering of labels:** Instead of relying on arbitrarily named references that are typically used in Dalvik bytecode, we apply a sequential numbering scheme for all labels, e.g., used as targets in `goto` instructions. By redefining labels with deterministic substitutes, we prevent findings of code differences without any effect on program semantics.

- **Sorting fields and methods:** To further avoid code diffs due to reordered methods and fields, we sort them in an unambiguous sequence. We achieve this by linking each field and method in a class with name-invariant values. Therefore, we derive a unique deterministic value, depending on the access flag, number of parameters and return type for methods, and access flag and type of field for fields. To determine the order of elements, we also leverage the method and field index stored in Dalvik bytecode. The resulting name-invariant value is finally used as a key to sort methods and fields.

While preprocessing operations are primarily aimed at obtaining consistent comparison results, further measures are needed to also handle *dynamic compile-time decisions*. The following operations are designed to address these peculiarities:

- **Deterministic register labels:** Using an adapted version of *baksmali*, we enforce the assignment of registers in a deterministic way instead of using variable, compiler-chosen register labels. For this purpose, we incrementally reassign all register names in ascending order, according to their first usage. This auxiliary procedure helps to detect two semantically identical blocks where only the register names were named differently during compilation. However, without additional checks, a sequential labeling could still lead to false results. E.g., assuming that a simple instruction would be inserted in between two identical comparison objects, all registers might be shifted and consequently, all remaining basic blocks would not be found as matches.
- **Static register labels:** As an alternative approach to replacing register names with deterministic assignments, our solution also supports static register placeholders. This enables hashes of basic blocks to be independent of used register names. However, this measure can influence matching accuracy. For instance, if a method changes the return value by referencing another register, this change would be unnoticed by this approach. Thus, we apply this step in conjunction with others to prevent possible mismatches.
- **Resolve resource identifiers:** Another measure to facilitate comparison in spite of dynamic compiler decisions is to replace used resource identifiers with the original type and content that is defined in `/res/default.xml`. For example, the value `const v0, 0x42cafe123` could be replaced by `const v0, **APKDIFF.<some_value>**`. Due to arbitrary assignments of resource identifiers during compilation, these replacements are inevitable for a distinctive comparison of code with references to application resources.
- **Excluding:** To filter possibly irrelevant differences, we propose to exclude classes of the `android` package from analysis. As these classes are typically provided by the Android runtime and not changed by developers, they do not contribute to showing functional differences between applications.

8.3.2 Matching Classes

Matching classes is based on the idea of gradually rewriting and trimming classes. For each application under comparison, we build Merkle trees that store hashes of all packages, classes, and methods. As depicted in Figure 8.2, this process is repeated multiple times. In each iteration, already matched classes are pruned and only those classes remain that exhibit differences after all comparison rounds.

1. **Unmodified files:** We first compare the hash value of unmodified `.smali` files. Unmodified means that no measures, besides those necessary for a coherent generation of classes, have been applied. Subsequently, we remove all classes from the Merkle trees that are identical in both applications.

Figure 8.2: Multi-round code comparison at class level.

2. **Files with replaced super class names:** We strive to detect classes that differ only in the names of the used superclasses. Hence, we generate `.smali` files, where occurrences of superclasses have been substituted with placeholders. Matching classes are again pruned from the Merkle trees.

3. **Files with replaced class names:** After comparing all classes with changed super class names, we now substitute only the class name with a placeholder. This approach enables us to find renamed classes. After this step, our analysis set remains with classes that either implement a different interface or that differ in their class signature.

4. **Files with replaced interfaces names:** We introduce a placeholder for all used interface names in our class files. As in previous steps, we then reduce our analysis set by all classes matching in both applications.

5. **Files with replaced class signatures:** For class definitions for which no corresponding match could be found so far, we repeat the search with a combination of all previous variants but also introduce placeholders for class and super class names as well as implemented interfaces.

As illustrated in Figure 8.2, steps 1-5 can be repeated multiple times. Between each iteration, the Dalvik bytecode of both apps is rewritten and additional operations are applied to increase the chance for possible matches. These steps can be repeated as long as new matches are found. However, so far the first five steps did not affect classes where fields or methods have been renamed by developers or obfuscation tools, such as ProGuard. As a remedy, we introduce one final step where we also replace all remaining identifiers with placeholders:

6. **Files with all identifiers replaced:** With a focus on the sequence of individual instructions, we remove all identifiers and substitute the names of classes, methods, fields, and type names with placeholders. An example of this operation is depicted in Figure 8.3. To prevent possible ambiguities, we omit rewriting for some methods, such as `<init>`, packages, e.g., `android/` or basic data types that, by design, cannot be affected by code obfuscation.

```
.method public final run()V
.registers 3

iget-object v0, p0, b:Lb/f;
iget-object v0, v0, Lb/f;->a:Lb/e;
iget-object v1, p0,
  a:Lrenamed/by/apkdiff139;

invoke-static {v0, v1}, Lb/e;->
  a(Lb/e;Lrenamed/by/apkdiff124;)V
return-void

.end method
```

```
.method public final _METHOD_()V
.registers 3

iget-object v0, p0, _FIELD_:L_TYPE_;
iget-object v0, v0, L_CLASS_;->_FIELD_:L_TYPE_;
iget-object v1, p0,
  _FIELD_:Lrenamed/by/apkdiff139;

invoke-static {v0, v1}, L_CLASS_;->
  _METHOD_(L_TYPE_;Lrenamed/by/apkdiff124;)V
return-void

.end method
```

Figure 8.3: Direct comparison of obfuscated Smali code before (left) and after (right) replacing all identifier names with placeholders.

8.3.3 Matching Methods and Basic Blocks

The previously described approach for a pairwise comparison of classes is able to recognize identical classes, even if identifiers are obfuscated or differ from each other due to dynamic compile-time decisions. However, it does not yet produce positive matches if methods have been added, removed, or changed. Therefore, for all code classes that did not match in the previous step, we pursue a more fine-grained inspection that focuses on individual methods and basic blocks.

As a first step, we leverage *baksmali* to extract all methods from remaining classes for which no match could be determined. For each method, we retrieve two representations: (1) the original code with all identifiers and, (2) to overcome code transformations, we rewrite the Dalvik bytecode and generate a semantically equivalent version where identifiers are replaced with placeholders.

While the original code of a method has an unambiguous affiliation to a specific class and package, this association is no longer present within the obfuscation-invariant representation. This could be problematic when matching methods that only consist of very few code lines. If the method to compare with is not clearly distinguishable from others, mismatches can occur. To prevent possible misclassifications, we consider a configurable lower bound of code lines that methods should have to be used for comparison. Preliminary tests have shown that methods with less than four instructions can no longer be matched accurately if obfuscation-invariant features are used. In fact, this threshold particularly affects so-called *getter* and *setter* methods as they usually exhibit the same implementation pattern that is strictly tied to concrete identifiers. As can be observed with Git, Subversion, and other tools that compare code, similar effects occur. Due to the little amount of information in very small methods, the practical impact of this issue is negligible. In our concept, we propose to highlight corresponding methods and remark that they are exempted from comparison.

For all methods that exceed the lower bound of code lines needed for matching, we derive a hash value of the method signature and the body with all basic blocks. This implies that the order to basic blocks is also considered and causes re-arranged, moved, added, or deleted code segments to thwart a successful match. As a remedy, in cases where methods cannot be matched, we propose to extend the matching process to the level of basic blocks for a more in-depth comparison. Summarizing, the comparison procedure concerning methods and basic blocks can be split into four consecutively executed steps:

1. **Methods with identifiers:** After extracting all methods of unmatched classes from Dalvik bytecode in their original format with potentially obfuscated identifiers, their hash representation is stored in a sorted list. Methods that fall below the configured minimum threshold for needed code lines are winnowed and not considered. Then, for all generated hash values of the first Android application a match is looked up in the set of methods in the second application, serving as a comparison object.

   Purpose: In this step, we determine all methods that have not been altered but were moved to another class.

2. **Methods without identifiers:** For each remaining method for which no corresponding match could be found in the first step, the search is repeated with a semantically equivalent but obfuscation-invariant representation of the method body. We replace all method identifiers with generic placeholders and, thereby, decouple methods from concrete labels.

   Purpose: This step finds all methods that exhibit the same control and data flow with differently named identifiers. In practice, this is the case, if one version of an Android application is obfuscated and the comparison object is not. Likewise, this comparison step matches method bodies that were applied different code transformation techniques and were e.g., compiled using different obfuscation settings.

3. **Basic blocks with identifiers:** If a method body has changed between two application versions, a more fine-grained matching using the contained basic blocks is inevitable. Similar to the first step, we derive the hash value of all basic blocks in so far unmatched methods, collect them in a sorted list, and compare them with the hash values of basic blocks of a comparison object. If basic blocks are found in multiple methods, the resulting candidates are sorted by the overall amount of matching basic blocks in the same method. This decision logic resembles the matching behavior of the Git source code versioning system.

   Purpose: This step discloses basic blocks that were moved to other methods without changing any identifiers.

4. **Basic blocks without identifiers:** Finally, the hash values of obfuscation-invariant basic blocks are compared with the hash values of basic block representations where identifiers have been replaced by placeholders.

   Purpose: In this step, we find deleted and newly added basic blocks. Due to the lookup without identifiers, we also discover basic blocks that have been moved to other methods but where identifiers have been renamed, e.g., due to different obfuscation settings or membership in a different code package. This operation also covers typical obfuscation settings, such as *code merging* and *inlining*. To preserve control flow integrity, code transformations are usually applied on entire basic blocks, rather than single code lines. By comparing representations without identifiers, our approach enables to effectively keep track of affected basic blocks.

The described comparison strategy evolves from matching the body of entire methods with unaltered identifiers to a fine-grained analysis on basic block level with replaced identifiers. Considering that today's Android applications often include tens of thousands of methods, our approach reasonably reduces the set of objects to compare in each step. Evidently, newly added or deleted basic blocks always come at full analysis cost as they have to be compared with the hash values of all other basic blocks, until we can conclude that they exist in only one of the two given Android applications.

8.4 Case Study

We study the practical feasibility of our concept for differential code analysis by comparing the implementation of two subsequent versions of Android applications. For demonstration purposes, we select two apps that work with a large code base: the messenger app *Skype*[3] and the password manager *1Password*[4]. For both apps, the source code is non-public, which means that only changelog information and reverse-engineered Smali code can be used for analysis. Nonetheless, since the vendors of these specific applications provide security-relevant release notes, we can leverage our comparison strategy and verify if changes in code correlate with statements made in changelogs. The subsequently shown Listings with code diffs represent the output of our framework after comparing two given app samples.

8.4.1 1Password

1Password is a popular password manager application for Android. Besides the brief changelog denoted in Google Play, full release notes are presented at the developer website[5]. In version 6.4.1, the authors addressed several security vulnerabilities that were present in previous versions. We apply our comparison framework on versions 6.4 (build 58) and 6.4.1 (build 59) of *1Password* and validate how security issues were handled in code. According to the developer-provided changelog, the security update introduced functional code changes:

- **Improved domain matching**

  In versions 6.4 and earlier, *1Password* did not consider subdomains when parsing URLs due to a mismatching regular expression. The update to version 6.4.1 replaced the check with a call to the newly added method `getLoginsForUrl`, which was successfully located by our similarity analysis in class `Utils`. As revealed by the code diff shown in Listing 8.1, the vendor added new code fragments to fix the domain matching problem.

- **New default scheme: HTTPS instead of HTTP**

  In earlier versions, the web browser functionality included in the application applied HTTP as the default scheme, if no full URL was specified by the user. As depicted in the lower part of Listing 8.1, the security update added code to prepend URLs with the *https://* prefix.

- **Prevent access to non-web URLs**

  Previous versions of *1Password* enabled users to read private data from the app folder by calling URLs with the *file:///* scheme in the built-in web browser. As shown in Listing 8.2, the security update changed the processing of user-entered URLs and added an alert dialog that presents an error message for all non-conforming inputs.

[3] https://play.google.com/store/apps/details?id=com.skype.raider
[4] https://play.google.com/store/apps/details?id=com.agilebits.onepassword
[5] https://app-updates.agilebits.com/product_history/OPA4

Listing 8.1: *1Password*: Added method `getLoginsForUrl`.

```
@@ diff: com/agilebits/onepassword/support/Utils.java <->
@@       com/agilebits/onepassword/support/Utils.java
...
+  public static List<GenericItemBase> getLoginsForUrl(
+    List<GenericItemBase> paramList, String paramString) {
+    paramString = PublicSuffix.registrableDomainForUrl(paramString);
+    ArrayList localArrayList = new ArrayList();
+    if ((paramList != null) && (!TextUtils.isEmpty(paramString))) {
+      paramList = paramList.iterator();
+      while (paramList.hasNext()) {
+        GenericItemBase localGenericItemBase = (GenericItemBase)paramList.next();
+        if ((!TextUtils.isEmpty(mLocation)) &&
+          (paramString.equals(PublicSuffix.registrableDomainForUrl(mLocation)))) {
+          localArrayList.add(localGenericItemBase);
+        }
+      }
+    }
+    return localArrayList;
+  }
...
+  public static URI parseURIFromUrl(String paramString) {
+    ...
+    return createURIFromUrlStr("https://" + paramString);
+  }
...
```

- **Informative dialogs in the case of TLS errors**

  According to the release notes, the vendor-provided fixes also improved messages that were displayed to users upon the event of SSL/TLS errors. Inspecting the newly added class `CommonWebViewClient` enabled us to verify how corresponding code improvements have been realized.

The changelog verification of *1Password* in version 6.4.1 confirmed that all security-relevant changes have indeed been carried out as described. The comparison also pointed out that no further modifications were made apart from those indicated in the release notes. As the app did not apply obfuscation, as can be seen by the names of files and variables, replacements of identifiers by placeholders were unnecessary in this case and would have had no effect on the comparison result.

Listing 8.2: *1Password*: Changed URL input check.

```
@@ diff: com/agilebits/onepassword/activity/AutologinActivity.java <->
@@       com/agilebits/onepassword/activity/AutologinActivity.java
...
  public void loadUrl(String paramString) {
-   paramString = Utils.uriFromUrl(paramString);
+   paramString = Utils.parseURIFromUrl(paramString);
    if (paramString != null) {
-     mWebView.loadUrl(paramString.toString());
+     mWebView.loadUrl(paramString.toASCIIString());
+     return;
    }
+   ActivityHelper.getAlertDialog(this, 2131281548, 2131281547).show();
  }
...
```

8.4.2 Skype

In 2011, the developers of the *Skype* messenger app distributed an update via Google Play that raised the version from 1.0.0.831 to 1.0.0.983. The provided changelog reported the fix of a severe security issue that caused sensitive profile data, such as the account balance, date of birth, email address, etc. to be stored unencrypted on the device. In addition, the formerly vulnerable version of the messenger also chose wrong access permission for files, allowing other applications to read and write them. An update resolved the vulnerability by securing the permissions of existing files and setting safer access rights for newly created ones.

Listing 8.3: *Skype*: Added method `getLoginsForUrl`.

```
@@ diff: com/skype/ipc/SkypeKitRunner.smali <-> com/skype/ipc/SkypeKitRunner.smali
  ...
  .end method

+ .method private fixPermissions([Ljava/io/File;)V
+   .registers 7
+
+   array-length v0, p1
+   ...
+
+ .end method
+
+ .method private chmod(Ljava/io/File;Ljava/lang/String;)Z
+   .registers 7
    ...
    const-string v6, "csf"
-   const/4 v7, 0x3
+   const/4 v7, 0x0

    invoke-virtual {v4, v6, v7}, Landroid/content/Context;->
     openFileOutput(Ljava/lang/String;I)Ljava/io/FileOutputStream;
    ...

    invoke-direct {v2}, Ljava/lang/StringBuilder;-><init>()V

-   const-string v4, "chmod 777 "
+   const-string v4, "chmod 750 "
    ...

    move-result-object v1
+   move-object/from16 v3, p0
+
+   iget-object v3, v3, mContext:Landroid/content/Context;
+   move-object v2, v3
+
+   invoke-virtual {v2}, Landroid/content/Context;->getFilesDir()Ljava/io/File;
+   move-result-object v2
+
+   invoke-virtual {v2}, Ljava/io/File;->listFiles()[Ljava/io/File;
+   move-result-object v2
+
+   move-object/from16 v3, p0
+   move-object v18, v2
+
+   invoke-direct {v3, v18}, fixPermissions([Ljava/io/File;)V

    invoke-static {}, Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;
    ...
.end method
```

Although this implementation weakness has been resolved long time ago, we specifically selected this sample for our case study, as it has previously been inspected by Desnos et al. [Des12] pursuing a similar approach. We consider their findings as a reference and can contrast them with the results of our framework.

By applying our comparison strategy on both versions of *Skype*, we disclose that the update introduced changes to the class `com.skype.ipc.SkypeKitRunner`. All vendor modifications that were found as code diff are depicted as Smali code in Listing 8.3. As can be seen in the upper part of the Listing, the update added two methods, named `fixPermissions()` and `chmod()`. Invoked from within existing code, their purpose is to reset permissions for files that have been created with previous versions of *Skype* and that were insecurely stored on the device. Another change affects an integer variable that is stored in register `v7` and gets passed to the method `openFileOutput()` as a second argument. According to the Android documentation[6], this change indicates that the mode used to access files has been changed from a combination of the modes `MODE_WORLD_READABLE` (1) and `MODE_WORLD_WRITEABLE` (3) to the more secure choice `MODE_PRIVATE` (0). In the second half of the listing, we see that a string value used to define the Unix access permissions for files was reset from `777` (*rwxrwxrwx*) to the value `750` (*rwxr-x—*), followed by an invocation of the newly added method `fixPermissions()`.

Summarizing, the study of two versions of the *Skype* application demonstrated that our solution is capable of highlighting all changes in code that were made to fix a security issue. The practical impact of these results is twofold: first, they show that the details provided in the release notes indeed correspond to the changes as found in the source code of *Skype* and second, they implicitly confirm that the changelog included all modifications that were made to the source code.

8.5 Conclusion

Android applications often receive updates that introduce new functionality and bugfixes. Even if developers summarize their modifications in release notes, it remains unclear how the underlying implementation has been altered. Compiler peculiarities, obfuscation, and other code transformation techniques make it challenging to filter only for changes with an effect on program execution and to verify whether bugs or security issues have been addressed.
We presented a solution to accurately assess similarities and differences in the code of two given Android applications. By using an iterative comparison approach, we are able to extract developer-induced code changes and succeed in matching pairs of semantically identical code fragments, even if they were moved or obfuscated. In a case study, we exemplified the practical applicability and verified how updates have been deployed to fix security-critical issues in real-world applications.

In the context of this thesis, our comparison strategy contributes to a semantic understanding of code fragments and supports application analysis with a novel solution to efficiently verify behavioral differences in two given app samples.

[6] https://developer.android.com/reference/android/content/Context.html

9

Modeling the Behavior of Android Applications

Static and dynamic program analysis are the key concepts researchers apply to uncover security-critical implementation weaknesses in Android apps. As it is often not obvious in which context problematic statements occur, it is challenging to assess their practical impact. While some flaws may turn out to be bad practice but not undermine the overall security level, others could have a serious impact. Distinguishing them requires knowledge of the designated app purpose.

We introduce a machine learning-based system that is capable of generating natural language text describing the core functionality of Android apps based on their code. We start in Section 9.1 by highlighting limitations of existing analysis approaches. To derive a high-level picture of implemented behavior, Section 9.2 elaborates the semantic relationships of resource identifiers, string constants, and API calls contained in apps. In Section 9.3, we design a dense neural network to predict precise, human-readable keywords and short phrases that indicate the main use-cases apps are designed for. We evaluate our solution on 67,040 real-world apps in Section 9.4 and find that with a precision between 69% and 84% we can identify keywords that also occur in the developer-provided description in Google Play. Parts of this chapter are taken verbatim from [FG20a].

Publication Data and Contribution

Johannes Feichtner and Stefan Gruber. "Code between the Lines: Semantic Analysis of Android Applications." In: *ICT Systems Security and Privacy Protection – IFIP SEC 2020*. Springer, 2020. In press
Contribution: Main author; Prototype implemented by Stefan Gruber.

9.1 Introduction

In recent years, researchers have elaborated various approaches to disclose possible leaks of private data [Ren+18; Li+15], identify malware [Alf+19; Mir+19], or to uncover security deficiencies [MBB18; Bia+15] in Android apps. Typically, the results of these analyses fall into two categories: firstly, a classification into malevolent or harmless or, secondly, concrete results of specific aspects the inspection has been aiming for. While both types may be adequate with regards to the intended objectives, they barely evolve to a superior level where the implemented behavior and context of instructions is also taken into account.

By performing a target-oriented program inspection using static and dynamic analysis techniques, we succeed in identifying and tracking relevant source code statements. However, we still fail to understand what the underlying code is actually executing and in what context particular code statements are employed. In practice, missing context awareness leads to situations where researchers disclose security flaws in execution traces but are unable to reason about the impact or relevance of the finding in terms of the actual purpose of an application. E.g., basically, it is problematic if a constant, hard-coded key is used for encryption. However, if this happens within an advertisement library where encryption is only used for obfuscation, the impact of the finding needs to be assessed differently.

Likewise, solutions aiming at uncovering leaks of sensitive data usually work with generalized assumptions about data flows and disregard the specific use-case applications are designed for. Most approaches are based on a source-sink analysis where the information flow between a particular source, e.g. a system API delivering a device's GPS coordinates, and a sink, e.g. an API for HTTPS requests, is examined [SKT17; RAB14]. Evidently, it depends on the designated purpose of an application whether supplying GPS information via HTTPS to an external entity, such as for assistance in traffic navigation or local weather forecasts, is a legitimate action or the undesirable leakage of sensitive data.

Besides leading to findings of *false positives*, missing context-awareness can also invoke the opposite effect. For example, under normal circumstances, there is no need for a mobile banking application to transform a password in order to login to an online service. In contrast, a program intending to securely store data protected by a user password undoubtedly should apply cryptography for key derivation and data encipherment. While, as shown in Chapter 5, it is possible to evaluate the soundness of parameters passed to key derivation functions via static or dynamic program analysis, existing solutions cannot highlight the need for cryptographic transformations in cases where their absence has a fatal impact on the attainable level of security.

In a broader sense, these examples highlight what analyses are currently unable to cover: the *semantic understanding* of applications. Rather than gaining a high-level picture of the functionality and security of a program, common approaches for inspection focus on single instructions at the lowest possible level. While this is undoubtedly a legitimate level to determine the immediate effects on memory calls and registers, we are still missing a platform that enables us to reason about the effects of coherent code parts on the overall program state.

Augmenting app analysis by contextual information, such as the intended purpose and designated functionality, is of utmost importance to obtain a holistic picture of app behavior. However, currently no solutions exist that could relate the metadata of an app with their actual implementation. This situation is aggravated by the fact that developer-provided descriptions are often minimal, inaccurate, and miss key information. Within this context, we formulate the following problems: *(1) Which attributes of an application describe its behavior? (2) How to identify the main purpose of an app? (3) What keywords and phrases should be included in a description text to represent an app's functionality?*

In this work, we introduce a solution that infers the main purpose of Android apps based on their implementation. Leveraging the recent advances in neural networks, our work attempts to capture and classify semantic relationships between apps. Our system works unsupervised, involves no labeling of data sets, and is trained with real-world app samples that are only coarsely pre-filtered, e.g., regarding the language of descriptions. The output is not only a prediction of what main functionality is implemented within an Android application. Using a model explanation algorithm, we also obtain an insight into what is relevant in apps, can explain the reasoning of predictions, and based on this knowledge, derive meaningful keywords and short phrases in natural language.

In summary, we make the following key contributions:

- To infer the functionality from implementations, we propose a combination of three dense neural networks that combine knowledge extracted from resource identifiers, string constants, and API calls. Our system delivers concise keywords and short phrases that describe the main purpose of apps.
- We train, validate, and test our models with 67,040 apps from Google Play. In a case study, we demonstrate the practical relevance and plausibility of predictions by contrasting them with the developer-provided description.
- To assess the quality of our system and to avoid incomprehensible black box predictions, we apply the model explaining algorithm SHAP [LL17]. It enables us to understand the influence of network input features on the derived output.

The outcome of this work represents a notable contribution towards a holistic analysis of Android applications. It helps researchers and users to foster an understanding of what functionality is actually implemented in Android apps.

9.2 Behavior Modeling of Android Apps

A naïve approach to identify functionality implemented in Android applications would be to statically define rules for classifying source code. However, the evolving nature of smartphone apps with constantly changing APIs and the usage of third-party libraries would make it cumbersome to spot and label specific behavior. As a remedy, our approach leverages modern methods of machine learning that work unsupervised and involve no prior labeling of data sets.

Before designing a neural network that predicts the main purpose of apps, we need to tackle a basic question: *Which attributes characterize the behavior of an app?* Users can answer this question intuitively by installing and testing an application. Vendors would refer to the specification to derive similar conclusions. Our approach is inspired by both perspectives and focuses on information sources that are included within the code and resources of Android app archives.

We attempt to model Android app behavior from two different angles. On the one hand, we consider static string resources that indicate what an app does from a user's and developer's perspective. On the other hand, we describe a program by the Android API calls it includes, e.g., to access sensitive information, draw UI effects, or implement event listeners. Based on the presence and co-occurrence of calls, we expect to see individual patterns that characterize different functionality.

In the following, we outline the features our neural network will use as an input to infer a semantic understanding of the purpose of applications:

- **App Resource Identifiers**: *Semantic information provided by developers.* In order to access resources, such as UI elements, graphics, or multilingual definitions from program code, Android relies on alphanumeric identifiers that unambiguously identify individual elements. Although these values can be chosen arbitrarily during development, they usually correspond semantically to the resource content.

- **String Constants**: *UI text and functional descriptions, shown to users.* Static UI elements, language variables, and URLs are typically stored within app resources. When shown to the user, these constants provide valuable semantic information regarding the purpose of an app and actions users can perform. E.g., if an app includes UI elements containing the string values "new transaction", "account balance", and "money transfer", its implemented functionality most likely targets financial transactions.

- **API Calls**: *Define how an app interacts with the Android OS environment.* The widespread use of third-party libraries, code obfuscation techniques, and the multitude of possible usage scenarios make it challenging to identify the individual semantics for every code block. We, thus, postulate that the behavior of applications is not (only) determined by the interaction of individual code fragments, but especially by their interaction with the operating system and users. Consequently, to infer implementation behavior, we focus on calls to APIs of the Android framework. By their modus operandi, they control, for instance, access to sensitive user data, device sensors, visual effects, media processing, and networks and, thus, clearly define the functionality of applications.

As each of these three feature types is embedded within a different semantic context, it is not viable to simply collect all occurrences and use them as a combined input for a neural network. For a more accurate representation, we propose to train three separate neural networks that take different input features but share the same underlying architecture and produce the same type of output.

9.3 Semantic App Analysis

We design a dense neural architecture to infer the implemented functionality from real-world Android applications. Our goal is to develop a system that can process an unknown app archive and delivers keywords and short phrases that describe the main purpose. To remediate the "black box" character usually associated with neural networks, we require that our solution provides an insight into which input features are decisive for predictions.

In the **training** phase, we train three separate neural networks with Android app archives and their developer-provided descriptions from Google Play. For each app, we first extract all relevant semantic features, weigh their importance using TF-IDF and use the resulting vector as input for the corresponding neural network. In parallel, we build a TF-IDF model with app description texts that will be used to derive a neural network output in natural language.

In the **prediction** phase, our system receives an app not seen during training. After deriving and processing a TF-IDF vector representation of all features included in a given archive, each neural network will return a list of key words and short phrases that commonly occur in app descriptions when certain input features are used. As shown in Figure 9.1, the output of the network for predicting description words based on given string constants may, e.g., consist of the words *sms*, *messenger*, and *friends*, with the adjacent decimal value expressing the relevance of the predictions. An algorithm for explaining neural networks called SHAP (see Section 9.3.4) is then applied to find out which input features contribute most to the prediction of these output tokens. Summarizing the outputs of all three models and sorting them regarding the shown relevance provides us with a ranked list of description fragments that describe the core functionality of an app as realized by the sum of included source code components.

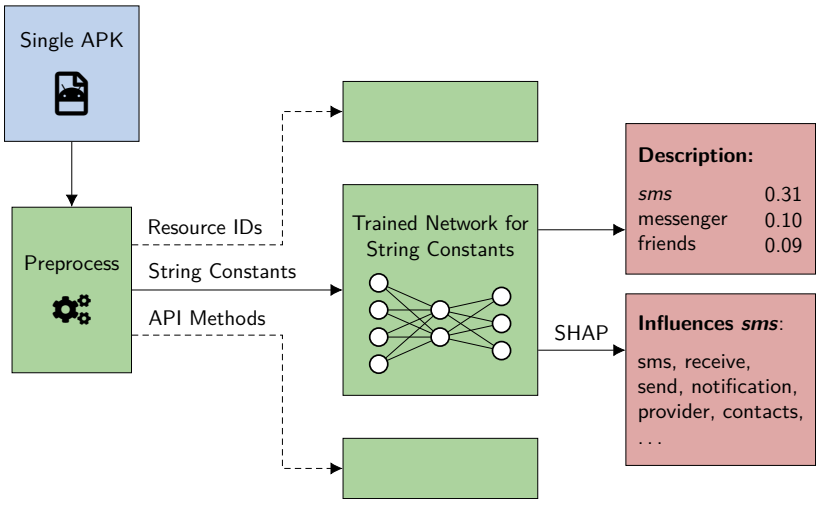


Figure 9.1: Prediction of implemented functionality using three dense neural networks.

9.3.1 Feature Preprocessing

Before training a neural network, it is essential to prepare the data for efficient learning. In the following, we cover the preprocessing steps that are applied to all developer-provided descriptions used in the training phase and the semantic input features processed by our networks after extracting them from app archives.

App Resource Identifiers

By parsing the XML files provided as resources in Android app archives, we obtain a list of identifiers consisting of alphanumeric characters and underscores. As opposed to variable or function names in source code, identifiers are typically not obfuscated but stored in the way app vendors define them during development. The name, or identifier, usually reflects its purpose to some extent and can also give hints about the overall app. In practice, values are mostly made up of words or word combinations that are linked either by underscores or formatted via camel-case, e.g., *select_photo_dialog*, *confirmDelete*, *pay_btn*, or *start_quiz_headline*. The challenge is therefore to decompose these values meaningfully in order to capture semantic relations. Without tokenization, e.g., it would not be possible to determine that the identifiers *select_photo_dialog* and *select_video_dialog* imply similar actions that differ only in *photo* and *video*. For a semantically more accurate representation, we split the words into smaller alphanumeric entities, i.e., *select*, *video*, *photo*, *dialog* and link them as n-grams.

String Constants

Android, by design, allows apps to display UI elements in different languages. Therefore, vendors have to provide translations for all UI-related string values that are referenced by language-agnostic resource identifiers. In this work, we aim to infer keywords and short phrases in English only. To achieve this, we mimic the behavior of the Android operating system and try to match identifiers with constants by primarily searching them in language files that are supposed to include values in English, i.e., `values-en.xml` or `values-en-us.xml`. Only in the case of mismatch, we fallback to default definitions in `values.xml`. This simple resolution strategy ensures that the corpora of values subsequently trained in TF-IDF models consist mainly of English words.

After extracting all relevant string constants from an app, we iteratively decompose each value into substrings by splitting at non-alphanumeric characters, e.g., whitespaces, HTML tag brackets, dots, etc. While most resulting tokens are likely app-specific, others supposedly occur frequently across multiple apps. To estimate the relevance of individual tokens in relation to all apps, we use the tokens and their occurrence count to build a TF-IDF model. Thereby, we leverage the property of TF-IDF that rarely occurring and very frequent tokens are ignored to maintain a reasonable dictionary size. As a result, for each app, we obtain a TF-IDF vector that can be used as input for a neural network.

API Calls

Inspecting the call graph of Android apps enables us to identify and count invocations of Android APIs. We process the reverse-engineered source code of the app archive and build a call graph based on static, explicit code statements. We enrich the graph with additional edges by resolving inheritance relations and implicit data flows using EdgeMiner [Cao+15] by Cao et al. As Android applications have no predefined entry points, *Activities*, *Services*, and *Providers* defined in the `AndroidManifest.xml` of each app are used as the starting point for modeling the call graph. This approach ensures that we capture only calls of API methods that are actively used and implement an app's main functionality.

After augmenting the call graph with implicit data flows, our objective is to measure the prevalence of implemented execution paths, i.e., data flows between app entry methods $E_{in,j}$ and API call methods $E_{out,k}$. For this purpose, we apply the Dijkstra algorithm to verify for each node $E_{in,j}$ in the graph, whether there exists a path to an API call in $E_{out,k}$. If this is the case, we increment a counter that reflects how often invocations of API method k are found. To unambiguously identify each call, we remember it by prepending the fully-qualified class identifier to the method name. As a result, we obtain a map of actively used API methods with numeric values that indicate their frequency of usage.

Similar to resource identifiers and string constants, we build a TF-IDF model for API methods. Applied on the acquired map of API calls and invocation counts, we use the TF-IDF measure to filter for relevant API calls based on their frequency. In practice, this enables us to identify small sets of methods that are descriptive for the functionality of individual app samples. TF-IDF weighs the information in our feature map and for each app, returns a fixed-size 1-dimensional vector with decimal numbers that can be used as a network input.

App Descriptions

For an arbitrary Android application provided as input, we want our neural networks to produce keywords and phrases that summarize the main functionality. In contrast to features in string constants or resource identifiers, where coherent tokens are enclosed within quotes or underscores, description texts include regular sentences in which semantic information can be spread over multiple words or word groups. If we would process descriptions in a bag-of-words approach, i.e., each word individually and without positional constraints, a lot of relevant connotation might be lost. For instance, assuming that the predicted functionality is only described by the keyword *recording*, it remains ambiguous whether this refers to capturing *call*, *screen*, *video* or any other possible type.

For a better understanding of the context in which words occur and to support a more intuitive interpretability of predictions, we focus on functionality that is expressed within word compositions. Therefore, we split the text into n-grams with $n = (1, 2, 3)$ representing phrases of one, two, and, three words. By applying Porter stemming [Por80] and removing stop words, e.g., *the, at, which, is*, we can quickly filter function words with comparably low importance for predictions.

Word combinations, such as *take a picture* and *take some picture* both become *take picture*. These techniques in combination with a sliding window approach using three different sizes enable us to build meaningful groups of words.

Extracted description tokens, irrespective of whether the model identifies single frequently occurring words or word formations, are stored in their stemmed representation. By reducing words to their stem, we can represent different word forms, e.g., *play* and *playing*, as a single token and, thereby, capture semantics of otherwise independent terms. While this contributes to optimizing the quality of predictions, stemming is not a bidirectional transformation and the result can be difficult to interpret for humans. Stripping the suffix of words also involves a loss of information and, thus, an accurate *un-stemming* method cannot exist.

As we want our models to infer expressive and human-readable description fragments, we apply a greedy algorithm to recover original words from their stemmed format. For this to achieve, we keep track of all stemming transformations, i.e., whenever a token T is processed and reduced to its stemmed version $\widehat{T} = f_{\text{stem}}(T)$. Stripping the suffix of words results in $\widehat{T}$ having multiple potential original tokens T. As some original tokens are by far more prevalent than others, we also remember the occurrence count of $T \to \widehat{T}$. After preprocessing all description texts, we obtain a mapping that reveals how often the stemmed form $\widehat{T}$ originated from a particular T. For instance, assuming the stemmed form of a token equals *locat*, our approach for *greedy un-stemming* will substitute it by *location*, irrespective of whether the original token was *location*, *located*, *locating*, or *locate*. In practice, this means that by counting how often an original token was transformed to a particular stemmed form, we maximize the likelihood that the un-stemmed form matches the most common original word.

The preprocessing of each description text results in a list of tokens that are ready to be processed via TF-IDF. The input features comprise single tokens, 2-grams, and 3-grams. By finally applying the TF-IDF measure on all tokens of a description, we retrieve a 1-dimensional vector with normalized decimal numbers between zero and one that can be set as a target for our machine learning tasks.

9.3.2 Model Architecture

We propose a combination of three models of dense neural networks to predict keywords and short phrases that characterize a given Android application. Each model produces n-grams as output and receives TF-IDF vectors with either resource identifiers, string constants, or method names as input. In this section, we highlight the advantages of dense neural networks for our problem and present our network architecture regarding the set of chosen layers and hyperparameters. Figure 9.2 illustrates our network architecture that is equally applicable to all input features. While resource identifiers and string constants are derived from n-grams, API calls cannot be tokenized and only include a limited set of methods. To make the results produced by these heterogeneous features comparable among each other, we aimed for a common network design and decided for TF-IDF vectors as the most suitable form of input representation. In contrast to other possible input types, e.g., word embeddings or a bags-of-words representation,

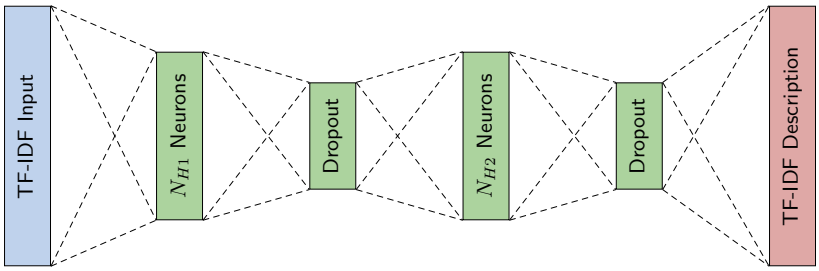


Figure 9.2: Dense neural architecture to infer TF-IDF vectors with descriptive keywords from resource identifiers, string constants, and API calls.

TF-IDF is less memory intensive, vectors can immediately be built from any feature type, and it needs no external training data. In addition, TF-IDF relies on a sparse matrix that features positional constraints, i.e., a single value is always mapped to the same element. We leverage this property in our network design to build a standard dense neural architecture instead of relying on a deep convolutional or recurrent structure that would determine sequential or positional information by itself. This simplifies our model and allows for efficient training.

Between the dense layers, we add dropout for regularization. Since the output produced by each neuron is a continuous numerical value, we are solving a regression task and use a linear activation function for the final output layer, as well as mean-squared error as a loss function. To prevent overfitting, we apply early stopping for 6 epochs and require the $F_{0.5}$-score to change by at least 0.5%, or otherwise training will be aborted as there is no more improvement.

The size of the input and output vectors corresponds to the length of the token list that is passed to TF-IDF. As the amount of extracted string constants, resource identifiers, and API calls is different for each app, the size of the dictionaries to process also varies in terms of the minimum and the maximum document frequency. This unequal distribution and, in particular, the minimum document frequency have an influence on the training process. If the lower boundary is too high, i.e., many apps make use of a large variety of features, we miss information that would be needed to precisely correlate individual input tokens, e.g., certain string constants or API calls, with description predictions. On the other hand, if the minimum is too low, i.e., many apps are highly individual in terms of the features they use, our models would learn too many rarely occurring tokens, and the dictionary would become very large. Thus, the selection of reasonable TF-IDF model parameters is crucial for the training process. To find suitable network architectures, we used random search. For this non-exhaustive search, we trained networks with one to three hidden layers and 1,000 to 15,000 hidden neurons. Dropout was randomly set between 0% and 40%.

We choose hyperparameters empirically by testing different setups and by observing the resulting dictionary sizes and model performance. Thereby, we set the minimum document frequency for each of the three input types to 2% of the total number of documents (apps), and the maximum frequency to 20%.

Table 9.1: Neural network configurations of the three models.

	Input TF-IDF # Features	**Network Hidden Layers**	**Description TF-IDF # Features**
Resource Identifiers	3315	2968, 3265, 1393 (3 Layers)	6140
String Constants	6391	2898, 3105 (2 Layers)	6140
API Methods	11735	5891 (1 Layer)	6140

This range means that if, e.g., we have a dataset of size N and a token occurs n times, it only ends up in the dictionary if it occurs in $0.02N \leq n \leq 0.2N$ apps. Table 9.1 lists our final network configurations and the TF-IDF dictionary sizes.

9.3.3 Model Training

We train our models using the mean-squared error (MSE) as a loss function. As our underlying problem is a regression task, MSE is not suited as a performance measure since it does not provide a meaningful insight into how well a model can predict the expected outcome. In order to apply standard classification metrics, such as precision, recall, and F-score, we need to model our regression task as a binary classification problem. Therefore, we discretize predicted TF-IDF vectors using a threshold θ, above which the element is set to one, or zero otherwise. We empirically define the value for θ such that noise from the network is ignored, i.e., values that are near but not exactly zero because of approximation errors. By measuring the model performance based on binary vectors, we are able to compare the TF-IDF vectors of actual app descriptions with those of predictions.

Our correlation-based learning approach aims to disclose real similarities between apps and, therefore, neglects app-specific terms in descriptions. From a performance point of view, this implies that we can expect a lower recall than precision because we are not specifically trying to predict the same words that are used in the original app description. For early stopping, we need to define a decisive performance metric that accounts whether training should stop or proceed. We, thus, apply a weighed F-measure of $\beta = 0.5$ that rates precision twice as high as recall. Consequently, we apply the $F_{0.5}$-score metric for early stopping to find a good final training state.

9.3.4 Explaining Predictions

Based on resource identifiers, string constants, and API calls, we build three semantic models that aim to infer the main functionality of Android applications by learning patterns and correlating similarities. Due to the inner complexity of neural networks, it is not always evident why certain description tokens are predicted. Especially, when multiple layers are stacked upon each other, the higher-level generalization of data precludes an insight into what features at the first layer were significant for decisions. To find out which input items contribute to the prediction of keywords and to remediate the black box character of network decisions, we apply the model explainer SHAP [LL17].

SHAP is an algorithm to explain the output of arbitrary machine learning classifiers. The method builds upon Shapley values, a concept from coalitional game theory: among n potential players, multiple combinations of $k \leq n$ players are possible, i.e., can play together against another football team. Each player formation leads to a different game score. Shapley values allow to interpret the impact an individual player has on the final score in relation to all possible player combinations. Lundberg et al., the authors of SHAP, proposed to combine this approach with additive feature attribution. By masking parts of the input features, the model yields different outputs per sample. Transforming SHAP values that were computed for smaller components of the network into values that hold for the whole model is computationally expensive. However, SHAP features several approximation methods, e.g., one for neural networks called Deep SHAP. By leveraging knowledge about the network's hyperparameters and architecture, rather than treating it as a black box, Deep SHAP creates an approximated, non-heuristic model.

9.4 Evaluation

The goal of this evaluation is twofold. First, we investigate the performance of our neural network with real-world Android apps. Second, applying our solution on a hand-picked set of applications, we compare predictions about the presumed functionality of apps with the actual description text from Google Play.

9.4.1 Dataset

We evaluate our approach using real-world applications from the PlayDrone dataset [VGN14]. We opted for this repository of apps as it does not only feature raw app archives but also makes the vendor-provided app description available.

After downloading 115,294 Android apps and their corresponding metadata, we removed cross-platform apps as they implement their core functionality with web technologies and lack the corresponding resource identifiers, string constants, and API calls. From the remaining set of 85,915 apps, we filtered apps that had no descriptions in English language and ensured that preprocessing each description text resulted in at least 20 TF-IDF vectors. This boundary was set to reduce the potential impact of insignificant samples on the training process.

Table 9.2: Subsets of Android apps used as neural network input.

	# Apps
Android apps crawled	115,294
Cross-platform apps	29,379
English descriptions and $\geq$ 20 TF-IDF tokens	67,040
Training set	66,040
Validation set (20%)	13,208
Test set	1,000

Table 9.3: Performance on the test set of the three neural network input types via discretized TF-IDF vectors. Discretization threshold: $\theta = 0.05$.

	Resource Identifiers	**String Constants**	**API Calls**
Precision	79%	84%	69%
Recall	27%	19%	18%
$F_{0.5}$-score	57%	50%	44%

Table 9.2 highlights the final set of Android applications we used to train, validate, and test each network input feature. 20% of apps used for training are randomly picked to be also part of the validation set. This partitioning scheme is required to prevent overfitting of our machine learning model and to ensure meaningful predictions. The test set includes 1,000 randomly chosen apps that are not used during training. We build the set such that it only includes apps with a comprehensive description text. Instead of manually selecting samples, we make the simplifying assumption that applications with higher installation count presumably also feature more qualitative descriptions. Therefore, we sort all applications in the crawled dataset by their popularity in descending order and select every third app until we reach 1,000 test samples.

9.4.2 Results

We trained neural networks for resource identifiers, string constants, and API calls, each with a set of 66,040 apps. To ensure an unbiased evaluation, the three models were validated using 20% of training data and tested individually with 1.5% of previously unseen data to confirm their final performance.

The evaluation results on the test set are summarized in Table 9.3. Comparing the $F_{0.5}$-scores, we observe that resource identifiers deliver the best results, while API calls perform worse, and string constants in between. While precision values range between 69% and 84%, the recall column presents low values for all models. This issue can be explained by mainly two reasons. First, description texts often include words that characterize the app purpose very specifically and only make sense in the local context. To also capture such words that do not generalize for a wider variety of samples, our models would have to memorize training data (*overfitting*), which generally should be prevented whenever possible. While app descriptions can be correct and sound nonetheless, reconstructing rarely occurring words remains challenging. Second, the TF-IDF model used to generate description output does not consider synonyms. For instance, if the original text included the word *picture*, but the word *photo* is predicted, it counts as a mismatch and lowers the recall performance despite the semantic correctness.

In practice, these results mean that our trained neural networks can well predict keywords and short phrases that also occur in the developer-provided description. High precision and low recall imply that the rate of false negatives is higher than the rate of false positives. This is desirable in our setting because a lower false positive rate also produces fewer false attributions of app functionality.

9.4.3 Case Study

For a better understanding on the practical relevance of functionality predictions, in the following, we take a closer look at each model's output regarding two music-related apps that were not used during training. We visualize the top 8 predictions and relevance values via word clouds. The font size of each token is set with respect to the weight (relevance) the models assign to all outputs.

Figure 9.3 illustrates the top-ranked predictions of the three models for the music video streaming app *Vevo*. Besides the word *video* being top-ranked, the essence of a platform for video streaming and sharing is described by the phrases *tv show / tv channels*, *movies*, *subscription*, *music*, *live* and *content*. By the prediction of phrases that include the keyword *tv*, we notice that our semantic models have not learned to differentiate between traditional television and online video streaming. However, as the term occurs in many other trained samples, we can assume that the neural networks understand the domain of the input and learn to cluster video-related applications internally. It can also be observed that tokens derived from API calls are less specific than those inferred via resource identifiers or string constants. Although the general domain of the app is still revealed by terms, such as *tv*, *video*, and *watch*, the top-predictions via API calls do not exhibit *n*-grams, like *tv channels* or *tv show*. Overall, despite their independent reasoning, the three models each yield descriptive information and can correctly identify an application's main purpose.

The app *4shared Music* is a music player that accesses audio files stored on the cloud storage provider 4shared. In Figure 9.4, we contrast the developer-provided description with the summarized predictions of our three models. Our neural networks correctly found that *4shared Music* is a *music player*, interacting with *playlists* and *albums*. They also correctly disclosed the secondary purpose of the app, operating as an online storage platform, by the terms *cloud*, *backup*, and *files*. While all these token predictions are comprehensible, not all of them appear in the original description text. E.g., the tokens *player*, *cloud*, and *backup* were not explicitly mentioned by the developer. As the description does not contain these words literally, the measurable performance (see Section 9.4.2) decreases despite the good generalization. These examples also point out that a precise but abstracted word cloud is not always intuitively interpretable for humans.

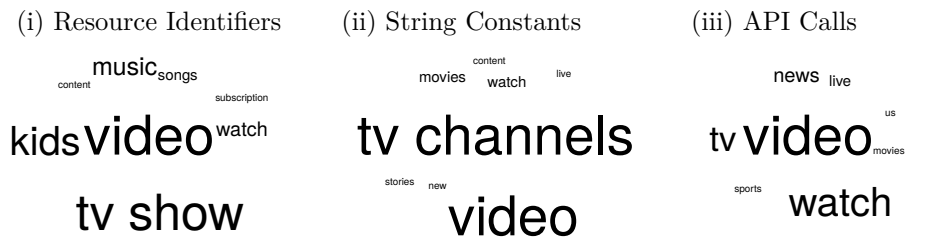


Figure 9.3: Word clouds with functionality predictions for the video streaming app *Vevo* based on resource identifiers, string constants, and API calls.

(i) Predicted Functionality

(ii) Actual Description

4shared Music was created for those, who can't live without music and don't want their attention to be attracted with anything else, but music while listening to it. Using your 'Search' menu item you can look for music files you like and add them to your playlist at 4shared Music. Moreover, you can upload tracks from your Android device to your 4shared Music. With 4shared Music you can enjoy 15GB of space for your music and nothing out of place. Upload and add all music files you like and make your life even more enjoyable with 4shared!

Figure 9.4: Comparison of the actual description text of *4shared Music* and our models' predictions.

9.4.4 Prediction Explanation

Each of our three machine learning models predicts a list of keywords and short phrases based on a given Android app archive. Apart from seeing this result, we also want to know which word predictions are caused by which input items. Therefore, we apply Deep SHAP (see Section 9.3.4) to all model predictions.

If, e.g., our resource identifier model outputs the word *dictionary*, we want to find what input data influences this prediction. A reasonable, for humans understandable relation would be input tokens, such as *search*, *word*, or *translate*. In case meaningless tokens were predicted instead, the model would have learned this correlation as "noise" from similar apps but not from a particular app feature.

To determine correlations between model input and output, we take the TF-IDF vector of an app sample as an input and infer the top prediction for it, i.e., from all network outputs, we pick the description word or phrase that yields

Table 9.4: SHAP algorithm applied on two predictions for the app *Slacker Radio*.

(i) Resource Identifiers

Description Token(s)	Input Tokens	SHAP
music player	artist	0.0122
	album	0.0107
	playlist	0.0071
	art	0.0034
	lyrics	0.0032

(ii) String Constants

Description Token(s)	Input Tokens	SHAP
music	playlist	0.0321
	song	0.0230
	stations	0.0125
	songs	0.0096
	tracks	0.0039

the highest probability. Then, we compute the Shapley values for all inputs and sort the associated input features by their SHAP values in descending order. Table 9.4 exemplifies this process for the app *Slacker Radio*. The description tokens predicted with the highest probability were *music player* for resource identifiers and *music* for the string constants model. Applying SHAP enables us to disclose the input tokens that contribute most to these predictions. For both prediction tokens, we see that the two network models make a decision based on comprehensible inputs. The SHAP value assigned to each input feature expresses its impact and responsibility for a change in the model output.

From applying SHAP to many samples, we noticed that for resource identifiers and string constants, found correlations are mostly self-evident. Although we also noticed many Android apps where the model based on API calls returned very accurate keywords, the associated SHAP values were not intuitively traceable. For example, for the *Vevo* app (see Figure 9.3), the term *video* is predicted with the highest probability. Nonetheless, the corresponding SHAP values refer to generic methods belonging to the `Activity` class from the Android API that, by their design, are not specifically related to multimedia applications. We suppose that in such cases, implementations leverage a characteristic set of methods that only occur in combination with video-related applications and, thus, acts as a sort of identifying fingerprint. With other samples, SHAP explanations for API calls expose very evident correlations. For instance, we found the description keyword *shake* to be very closely linked with the `SensorManager` class of the Android API. Considering that this class is responsible for measuring device sensor values, e.g., from the accelerometer or gyroscope, the association is also intuitively understandable. Overall, our qualitative analysis using the SHAP model explanation algorithm confirmed that all our models could very well outline the main purpose of most real-world Android applications.

9.5 Conclusion

We presented a solution to describe the main purpose of Android applications in natural language by analyzing resource identifiers, string constants, and API calls contained in app archives. Based on a combination of three dense neural networks, our approach accurately captures semantic relationships among apps.

In the context of this thesis, our work represents a significant contribution towards a more holistic analysis of Android applications. If app archives are subject to inspection for which no metadata is available, e.g., due to the fact that the apps under test do not originate from official sources, the main functionality can still be inferred and explained in natural language keywords.

We carefully evaluated our approach on 67,040 real-world Android apps and showed that with a precision between 69% and 84% our neural networks could predict keywords and short phrases that also occur in the developer-provided description in Google Play. Our solution provides an effective method to describe the behavior of unknown implementations.

10 Privacy Awareness in Android App Descriptions

Permissions are a key factor in Android to protect users' privacy. As it is often not obvious why applications require certain permissions, developer-provided descriptions in Google Play and third-party markets should explain to users how sensitive data is processed. Reliably recognizing whether app descriptions cover permission usage is challenging due to the lack of enforced quality standards and a variety of ways developers can express privacy-related facts.

In this chapter, we introduce a deep learning approach to identify discrepancies between developer-described app behavior and permission usage. We start by highlighting the semantic gap between described functionality and access to privacy-sensitive resources in Section 10.1. Followed by that, in Section 10.2, we propose a convolutional neural network that captures the relevance of words and phrases in app descriptions in relation to the usage of *dangerous permissions*. Section 10.3 describes required preprocessing steps and Section 10.4 explains the model architecture. In Section 10.5, we evaluate our solution on 77,000 real-world applications and find that we can identify permission groups with a precision between 71% and 93%. Parts of this chapter are taken verbatim from [FG20b].

Publication Data and Contribution

Johannes Feichtner and Stefan Gruber. "Understanding Privacy Awareness in Android App Descriptions Using Deep Learning." In: *Conference on Data and Application Security and Privacy – CODASPY'20*. ACM, 2020, pp. 203–214. DOI: 10.1145/3374664.3375730

Contribution: Main author; Prototype implemented by Stefan Gruber.

10.1 Introduction

Google Play and third-party markets enable Android users to choose from a vast amount and diversity of apps. For a better overview, markets organize apps in categories, outline their purpose in a one-sentence summary, and upon selection provide a more extensive description, screenshots, and the set of permissions an app requests to be granted. Based on this metadata, users make a decision on whether an Android application seems trustworthy enough to install and use it.

To meet users' expectations and to set adequate boundaries of behaviors, Android protects security-critical device functionality. Whenever an app requires access to the camera, microphone, or other sensitive features, users are requested to explicitly grant or deny the according permission. To alleviate privacy concerns, with Android 6 permissions have been reorganized into groups, whereas *dangerous permissions* subsume all those that carry a higher risk for abuse by giving an application access to sensitive data features and the user's personal data.

Recent studies [PPK18; Sco+18] point out that many users cannot assess the security risk associated with granting permissions. Although it may seem intuitive that a messenger application requires access to a user's contacts, it may not be obvious why the same app also requests the permission to fetch the current GPS position or to make phone calls. The situation is aggravated by the fact that many malware and privacy-invasive applications also infamously claim more permissions than their functionality warrants [AP18; PCJ18]. Despite recurring attempts to improve how meaning and purpose of system permissions are presented to users, permission requests often remain incomprehensible.

Making access to sensitive user data and device features transparent to users is of utmost importance to prevent unknowing or unconscious leaks of personal data. For this to achieve, descriptions of Android apps should imply the usage of dangerous permissions and highlight how and why they are required. Unfortunately, in practice, descriptions are often minimal, inaccurate, and lack statements about security-related aspects overall. As descriptions mostly promote functionality from a usability point of view rather than being security-centric, there is also only little incentive for developers to point out security aspects. We aim to build a system that can assess and improve privacy awareness in descriptions of Android apps. This involves answering the following questions: *(1) How does an app description reflect privacy-related permission usage?* and *(2) What is the relevance of individual words, word groups, and phrases?*

The results of prior work in this direction confirm the existence of correlations between permission usage and individual word combinations [Qu+14; Pan+13]. A viable approach to relate them would be to apply established NLP techniques for sentence analysis. However, the arbitrariness of real-world app descriptions and the lack of enforced quality standards impede a conclusive extraction of semantic information. The constantly evolving nature of smartphone apps with frequently changing descriptions also make it cumbersome to constrain text-permission-relationships to a limited set of semantically similar vocabulary. The key challenge, thus, is to find an approach that works reliably with noisy, real-world app descriptions and can map privacy-related text to permission usage.

We propose an innovative method to identify semantic correlations between description text and permission usage within real-world Android applications. Our incentive is to develop a system that can infer permission usage from the functionality described in arbitrary text fragments. Therefore, we design a deep neural network to reliably assess and grade the need for permissions by analyzing noisy, potentially incomplete, and inaccurate descriptions. For each permission, a score is delivered that indicates the likelihood that a certain permission is required according to a given text. To remediate the black box character usually associated with deep learning, we require that our solution provides an insight into which words are decisive for different network outputs. Comparing obtained predictions with the set of actually requested permissions enables users, developers, and markets to draw conclusions about privacy-relevant aspects in app descriptions. Our contribution includes the following key components:

- **Correlating permission usage with app descriptions.** We introduce a deep learning-based approach to predict the likelihood that Android applications needs certain permissions based on their description texts. We extend the architecture of a convolutional neural network (CNN) for text classification to identify sample-based correlations between parts of the description and the permission groups an app requests. Our solution overcomes various limitations present in existing research and can work effectively with unlabeled, potentially flawed descriptions of real-world Android applications.

- **Extracting semantic knowledge from app descriptions.** We study the expressiveness of textual properties in app descriptions and employ an embedding model to capture semantic links between words, word groups, and phrases. Instead of linking individual permissions only with frequently occurring words, we perform a contextual text analysis using the state-of-the-art techniques word2vec [Mik+13] and GloVe [PSM14] and, as introduced with Android 6, focus on groups of *dangerous permissions*. In our evaluation, we compare the performance of the two NLP techniques in their role as an input preprocessor for a deep neural network.

- **Evaluation.** We train our network with 77,000 descriptions and sets of permission usage from Google Play. We validate the performance of our model and evaluate the prediction quality for each group of *dangerous permissions*. In a case study, we demonstrate that our system can successfully identify text fragments that point to individual permission, identify missing permission explanations, and also reveal non-intuitive links between permissions and text phrases.

- **Explaining Predictions.** To assess the quality of our neural network and to avoid incomprehensible black box predictions, we employ the model explaining algorithm LIME [RSG16]. We calculate a score for each word that shows a significance for the output. Visualized as heatmaps, our solution can highlight the influence of particular words in description texts with regard to a predicted permission group.

As Android permissions play a crucial role in protecting the users' privacy, aligning their usage with the alleged functionality of applications has become a growing field of research. In the following, we present existing work on correlating permission usage with app metadata and program code.

Android Permissions. To validate whether permissions are indeed used within apps, Pandita et al. [Pan+13] parse descriptions using an NLP parser and build semantic graphs of API calls associated with permissions. Based on the names of classes and methods, keywords are identified and compared with tokens in the semantic graph. A similar approach is presented by Qu et al. [Qu+14], who perform a sentence analysis, identify textual patterns based on frequency analysis, and correlate them with permissions. The output can finally be used to annotate privacy-relevant sentences in the description text. As their system may raise false alerts when patterns that are typically found with privacy-critical permissions are not present, Yu et al. [Yu+18] propose to handle such cases by cross-verifying an app's privacy policy, description, permissions, and bytecode.

Code Analysis. Determining whether a permission request is necessary can also be achieved via static and dynamic program analysis. In a combination of static code inspection and text analysis, Watanabe et al. [Wat+15] present a keyword-based technique to correlate access to privacy-relevant resources with app descriptions. With a focus on potential abuse of sensitive APIs, Gorla et al. [Gor+14] derive app clusters based on pre-labeled description topics. Related to that, the approach of Gao et al. [Gao+19] infers expectable permissions by applying statistical correlation coefficients after mining topics from descriptions using NLP techniques and Latent Dirichlet Allocation (LDA). Similar to our solution, the authors address the usage of *dangerous permissions* and derive a score for each group to assess whether a declared permission has a close relevance with the app's presumed functionalities. As topics are derived from words that frequently occur together, the absence of trigger keywords and the lack of further contextual information may provoke false negatives.

Machine Learning. Kong et al. [KCJ15] and Wu et al. [WYL17] propose to predict security-related app behavior by supplying the words of user reviews to Support Vector Machines (SVM). To infer how specific permissions are used in code, Wang et al. [WHG15] apply text analysis and different supervised classifiers on a manually labeled set of 622 apps. McLaughlin et al. [McL+17] interpret source code analysis as a form of textual processing and design a convolutional neural network (CNN) that captures semantic information from opcodes in Dalvik bytecode to detect malware. Also targeted at finding malware sequences in Android apps, Huang et al. [HK18] propose to organize bytecode as images and learn them in CNNs. Both approaches highlight the efficiency of CNNs to detect contextual patterns in a large amount of training data. Our solution confirms that this type of deep neural network is also very well-suited to infer *dangerous permissions* from real-world app descriptions with high precision.

10.2 System Overview

We design a deep neural network to find relationships between permissions and text within a large dataset of real-world Android applications. Our goal is to develop a system that can process a given description text in human language and based on this text, deliver a score for each permission. A better score corresponds to a higher likelihood that an app requests a certain permission. In order to remediate the *black box* that is usually associated with deep learning, we require that our solution provides an insight into which parts of description texts are causative for individual scores. A comparison of obtained predictions with the set of actually requested permissions enables users, developers, and markets to draw conclusions about privacy-relevant aspects in app descriptions.

The primary functionality of our system can be split into two parts: In the **training phase**, our network learns correlations between permissions used in apps and parts of description texts. Using backpropagation, our convolutional neural network iteratively adjusts its internal weight kernel to achieve correct predictions of an app's permissions. Training continues until the performance decreases or plateaus, i.e., the quality of permission scores does not improve any further. The model giving the best performance in training is then used for testing. In the **prediction phase**, our system receives app descriptions not seen during training. As depicted in Figure 10.1, the trained network outputs scores indicating whether a given description reveals hints about particular permissions. To estimate which words in descriptions have an impact on permission scores, we leverage the model explanation algorithm LIME (see Section 10.4.3).

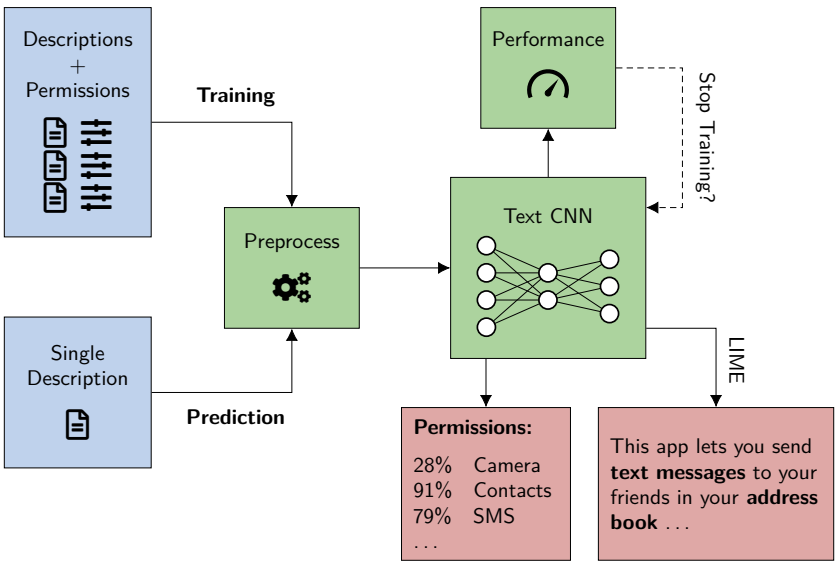


Figure 10.1: Training on combinations of app descriptions and permissions to eventually predict permissions based on the text only.

10.3 Feature Preprocessing

Before training a neural network, it is essential to prepare the data for efficient learning. In the following, we explain the preprocessing steps that are applied to developer-provided app description texts and permission sets after extracting them from Android applications.

10.3.1 Descriptions

For our network to model the relation of individual words in descriptions with used permissions, we first have to find a strategy to transform the contained character arrays into individual tokens. Since we strive to also capture co-occurrences of words, a bag-of-words approach would not be feasible for this task as it does not preserve the word order. To describe the different types of contexts words appear in, it is necessary to parse a given text word-for-word and to represent extracted tokens as vectors in a dense low-dimensional space.

For this translation, we can leverage a *pre-trained model* of word embeddings that reduce the number of parameters our neural network has to learn from scratch. Pre-trained word embedding models consist of key-value associations, whereas a key represents a single or combined word formation and a value the corresponding embedding vector. Typically, such dictionaries are built from alpha-numeric tokens that are extracted from text corpora which presumably carry a vast amount of semantic information, e.g., encyclopedias or news articles [Nor+17]. As an embedding model might not include all tokens contained in app descriptions, we pursue a *trial-and-error lookup strategy* and gradually build dense matrices of a fixed size for each description text. Alternatively, it would be possible to extend the CNN with an *embedding layer* to learn embeddings on-the-fly. While this would enable us to also process additional tokens with e.g., company brands or web addresses, the extra step would significantly prolong training.

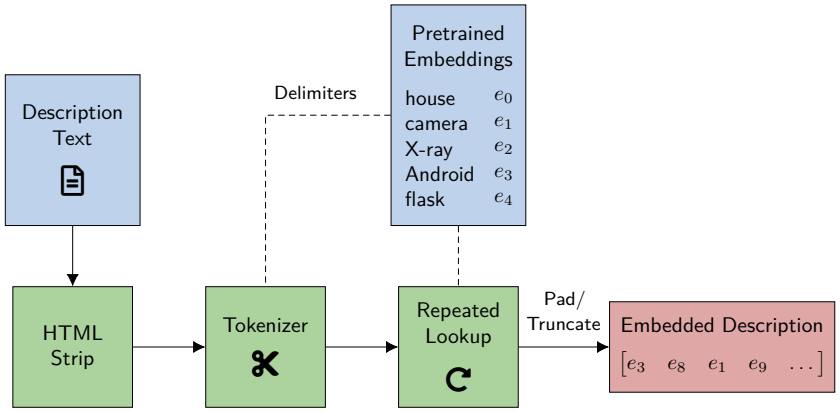


Figure 10.2: Iterative tokenization and embedding lookup for each word contained in the description of an Android application.

Relying on a pre-trained word embedding model implies that any token which is not present as a dictionary key, cannot be considered for training. In practice, this constraint acts as a double-edged sword: if an embedding model was trained on a sufficiently large corpus, it also serves as a filter to ignore potentially irrelevant tokens, such as hyphenated expressions, trademarks, or email addresses. At the same time, this means that we have to adapt the tokenizer such that it matches a maximum amount of possible embeddings.

In Figure 10.2, we illustrate our approach to address this problem. From a given description text, we first remove all contained HTML tags, as they are easily detectable and would not have embedding counterparts anyway. Before dividing the string into smaller entities, we inspect the character sets of all pre-trained embeddings and build a set of distinct characters available in the dictionary. The idea is to subsequently apply the inverse of this set as a delimiter for tokenization. While individual words and word formations typically consist of alphanumeric ASCII characters, they may also include non-alphanumeric or Unicode characters. Instead of dividing descriptions into substrings by always splitting at predefined delimiters, such as hyphens or colons, relying on the inverse character set of pre-trained embeddings ensures that word compositions can be matched in full.

After deriving individual sub-strings, each token is looked up in the dictionary of pre-trained embeddings: first, it is checked whether an embedding exists for the entire token. If a match is found, the token is replaced by the dictionary value. Otherwise, we attempt to strip surrounding characters that might hamper a successful match but have no influence on semantic expressiveness. After trimming leading and trailing non-alphanumeric characters, the pruned token is searched again. Upon mismatch, the token is further divided into sub-strings by splitting at remaining non-alphanumeric delimiters. Each sub-token is then looked up separately and if found, the corresponding embedding value can be used to substitute the sub-token. Overall, this approach ensures that a maximum of tokens can be matched in full via pre-trained embeddings, while being conservative enough to not taint semantic relationships by replacing tokens with potentially unrelated vectors that might emerge from breaking word combinations.

As convolutional neural networks operate on feature maps with a fixed-size input, it is necessary to define an upper bound for the possible maximum amount of processable embedding vectors. In practice, application descriptions can be of arbitrary length and, thus, a reasonable threshold can only be identified with regard to a concrete set of texts. We determine this value empirically by inspecting the dataset, described in Section 10.5.1, and require that it should allow to process 95% of texts until the last token. For descriptions that exceed this value, we discard vectors that go beyond. Conversely, for smaller descriptions, padding tokens are inserted that carry no semantic information and do not taint the network training process. At the end, we obtain a constant-sized matrix, with the row indices referring to the corresponding description tokens, and each row containing a vector from the pre-trained word embedding model. The final matrix represents the contextualized description of an Android application, ready to be passed as input to an arbitrary neural network.

10.3.2 Permissions

Introduced with Android 6, the protection level[1] of *dangerous permissions* comprises all API calls with access to private information and sensitive device features. Considering their critical role and high impact on users' privacy, we focus on permissions included in this group and disregard other protection levels. As apps declare requested permissions by means of listing predefined string identifiers, we need to represent these values in a way that a neural network can process them.

Every Android application includes a *AndroidManifest.xml* file that denotes requested permissions within `<uses-permission>` tags. We extract all permission identifiers, filter for *dangerous permissions* and assign them into groups[2]. This leaves us with nine permission groups: `CALENDAR`, `CALL_LOG`, `CAMERA`, `CONTACTS`, `LOCATION`, `MICROPHONE`, `PHONE`, `SMS`, and `STORAGE`.

We can express these nine permission groups in a vector, whereas each column represents a different group. For each app, we then verify whether it requests one or multiple individual permissions pertaining to a group. If it does, the value of the corresponding column is set to 1, or 0 otherwise. As a result, we obtain a binary vector with 9 values that describe the privacy-critical permissions an app makes use of. In the neural network, the vectors are set as learning targets.

10.4 Model Construction

We propose a convolutional neural network (CNN) to predict the likelihood that an Android app requires a certain permission. By applying different filters, we capture the local properties of individual words, word groups, and phrases and apply them within a multi-label classification task, in which each label corresponds to a permission group. In this section, we highlight the advantages of CNNs for our problem and present the architecture of our network with regard to the set of chosen filters, layers, and hyperparameters.

In contrast to recurrent networks, such as LSTMs or GRUs, convolutional neural networks allow for faster training and can more intuitively be interpreted. If we only wanted to correlate the use of permissions with the occurrence of words in descriptions, a machine learning solution would not be necessary. The architecture of CNNs, however, promotes to not only measure the impact of individual words but also enables us to capture their co-occurrence. In our model, we leverage this feature to determine whether certain words occur together and assess how relevant each of them is regarding the use of individual permissions.

Moreover, the concept of CNN filters allows for generalization. As we use word embeddings, the 1-D filters of our network do not immediately operate on individual words in description texts. Instead, a particular filter captures groups of words with similar properties in the word embedding subspace, e.g., the semantically related words "photo", "picture", and their plurals. These properties fostered our decision to use CNNs for this multi-label classification task.

[1] https://developer.android.com/guide/topics/permissions/overview
[2] https://developer.android.com/reference/android/Manifest.permission.html

10.4.1 Architecture

As a basis, we employ a word-based CNN model for text classification, proposed by Kim et al.[Kim14], and augment it for our task. Table 10.1 summarizes our set of used hyperparameters. To find good parameters for the number of layers, dense neurons, and dropout, we used grid search. Figure 10.3 depicts the adapted architecture of our network and highlights all involved layers and operations.

The network input consists of a $d \times m$ matrix, where d denotes the size or *dimension* of embeddings and m the *maximum* amount of embeddable tokens we process for each description text. Based on the data retrieved from the input layer, the network performs 1-D convolutions with kernels of size 1, 2, and 3, i.e., sliding windows that can process one to three words of input. For each kernel, we allocate 1024 filters that learn the semantic context words appear in. After feature extraction, the three filter bundles are reduced via *global max pooling* operations. As a result, we obtain the 3×1024 largest values of each kernel slice and concatenate them into a one-dimensional vector to fit the input of a *dense layer*. To prevent overfitting, we apply a dropout regularization of 0.2 after a fully-connected layer with 5000 neurons and another one with 2500 neurons. This means that in every training iteration, 20% of feature weights are randomly set to zero. The output layer represents the 9 permissions groups by single neurons, whereas each describes the usage probability with a score between 0 and 1.

Table 10.1: Hyperparameters used for CNN training.

Filters	Kernels: 1x1, 1x2, 1x3 (1024 each) Padding: Same Stride: 1
Hidden Dense Layers	5000 Neurons, 2500 Neurons
Optimizer	Adam, $\eta = 0.0001$
Batch Size	32
Weight Initialization	Glorot (Uniform)
Loss Function	Binary Cross-Entropy
Hidden Activations	ReLU
Final Activations	Sigmoid
Early Stopping	Patience: 6 epochs Delta: 2% F_β-score (macro)

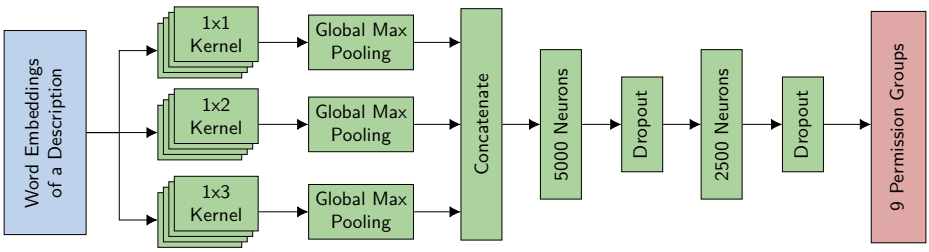


Figure 10.3: CNN architecture to process embedded app descriptions and get the likelihood for the use of *dangerous permissions*.

10.4.2 Model Training

Designed to maximize accuracy and reduce error, our network performs best when the number of samples in each class are about equal. However, in real-world Android apps permission usage is not equally distributed. As there are in practice, e.g., by far more apps that request access to a user's location than to the microphone, class imbalance occurs. As a countermeasure, we weigh the loss function and, as subsequently described, scale individual performance metrics. Moreover, we split our dataset into a training, validation, and test set. This allows us to adopt early stopping and to find a good final training state that prevents both over- and underfitting.

Class Imbalance

In many real-world training datasets, the distribution of samples across known classes is biased or skewed. This difference is commonly referred to as class imbalance, affects deep learning techniques to a similar extent [JK19] as traditional classifiers [JS02], and can result in a poor predictive performance. In the case of mobile applications, the need for particular permission groups as *classes*, as well as their co-occurrence, is strongly influenced by the designated app purpose. For instance, in practice, it can be observed that all Android applications with a request to the `CAMERA` permission also require `EXTERNAL_STORAGE`, as they would otherwise not be able to store captured photos. As our neural network produces an individual output for each permission group in arbitrary co-occurrence formations, the impact of class imbalance is inherently more significant than in other domains.

A common approach [BMM18] to counter class imbalance that also works well with CNNs is *oversampling*. However, for this to effectively enlarge minority classes, synthetic samples of app descriptions would have to be generated. As we do not consider this a viable option for our task, we aim to mitigate class imbalance by assigning individual weights to all permission groups. As shown in Equation 10.1, therefore, we first have to evaluate their actual distribution. For all samples in our dataset, we add up the 9 binary values p_i of the permission vector and store the sum in vector s. Then, we divide the inverse of s by the total number of app samples. We also normalize the vector by dividing by 9, such that the class-weighted loss and the non-weighted loss can immediately be compared. As a result, we obtain a vector c with 9 elements that can be multiplied with the actual error value (*loss*) that is back-propagated for every single permission output. In practice, this means that for permission groups with only few samples (*low support*), the error rate is increased, while it is decreased for those with high support. This effectively reduces class-dependent overfitting in a trade-off with the fact that particular samples are considered more important than others.

$$
\begin{aligned}
p_i &= \begin{bmatrix} p_{i,0} & p_{i,1} & \dots & p_{i,8} \end{bmatrix} && \text{for } 0 \leq i < N_{\text{samples}} \\
s &= \sum_{N_{\text{samples}}} p_i && c = \frac{N_{\text{samples}}}{s} \cdot \frac{1}{9}
\end{aligned}
\tag{10.1}
$$

As a fundamental prerequisite, our neural network has to be capable of working with descriptions of real-world Android applications. This implies that parts of texts may not provide enough semantic information to describe all used permission groups. For cases where humans would not be able to intuitively link requested permissions with text fragments, we demand that a machine learning algorithm should also not attempt to identify insignificant correlations. In practice, it is challenging to tackle this problem with an unfiltered dataset or without a metric that indicates the quality of a description. We can mitigate the issue to some extent by adjusting the β parameter of the F-score. With an $F_{0.5}$-score, *precision* is weighed twice as high as *recall*. This alleviates the influence of false negatives and causes training to prefer models that produce more confident predictions.

Balancing individual classes alters the weight that each training sample carries when computing the loss. In addition to adjusting the error value during back-propagation, class imbalance also has to be considered with evaluation metrics. While we can explain the performance for each class using precision, recall, and F_β-score, we need a single number with the classifier's overall F-score to compare different network architectures and training states independent from class support. As the *macro F_β-measure* factors in class imbalance analogous to F_β-scores of individual classes, we use it to identify the best-performing network architecture.

Overfitting

To prevent overfitting, we apply early stopping using a distinct set of training, validation, and test data. The network attempts to learn the relationship on a majority of samples in the training data by back-propagating the loss. To determine how well the model generalizes, we measure the performance on the validation set after each epoch. The test dataset finally provides an unbiased evaluation of the model. All sets are compiled randomly and the order of elements is permuted in each epoch. Instead of training a fixed number of epochs, we strive to find a suitable count dynamically by continuously monitoring how the model performs on training data. While the error rate decreases, learning continues for at least 6 more epochs as long as new local minima are determined. If this is not the case, weights are reset to the previously best-performing network state. The final weight parameters also correspond to the last state of the trained model.

10.4.3 Explaining Predictions

Besides obtaining the likelihood that certain permission groups are represented within app descriptions, we are interested in the individual words and word formations that contribute to the prediction of a certain permission usage. To better understand which observations our CNN deems relevant, we employ the model explaining algorithm LIME [RSG16]. The overall idea is to assign each word an individual score that expresses its relevance with regard to some output.

LIME works model-agnostic and interacts with an arbitrary black box classifier by reflecting its behavior around an instance being predicted. Explanations are shown as regular words, even if word embeddings were used as a network input.

Unrelated to our feature preprocessing approach, LIME splits app descriptions into individual tokens and uses them as a basis to generate variations with minor alterations. For each sample, the algorithm removes features that contribute the most to the predicted class until the output changes. The impact that each input perturbation causes is accumulated per classification target, i.e., permission group. The LIME algorithm relies on two configuration parameters: (1) the number of features to vary and (2) the amount of input perturbations to generate. We determined these values experimentally and found that 100 input combinations represent a reasonable trade-off between computational effort and result accuracy. Based on these settings, LIME builds sparse linear models around individual words and computes their significance on the output. For a given app description, the algorithm returns a list with impact scores per class and for each permission group, sublists with words and scores that indicate their local relevance.

Weighing the Impacts of Words

LIME describes the impact on the model output for every single word in all learned app descriptions using positive and negative decimal numbers. If the removal of a token leads to a more confident prediction, it is penalized with a score below zero. Conversely, if a word is found to contribute to a prediction, it is assigned a positive score. As we intend to explain the relevance of words not globally but individually per app description, we have to set a lower bound and normalize all LIME values with respect the tokens of a given text. In addition, we have to factor in the prediction of our model that a certain permission group is actually being used. This usage probability is expressed with a value between 0 and 1. In practice, this means that if, e.g., the likelihood that an app needs a particular permission is 30%, a word associated with this class has to be weighed lower, even if its LIME score accounts to 90% of the global maximum score.

$$\widetilde{L}_{p,t} = \frac{L_{p,t}}{\max_k \left(L_{p,k}\right)} \qquad \text{for k ... } 0 \leq t < N_{\text{tokens}}$$
$$h_{p,t} = \widetilde{L}_{p,t} \cdot y_p \tag{10.2}$$

Equation 10.2 shows the normalization and scaling formula we apply to $L_{p,t}$, the raw LIME score produced for each predicted permission p and token t in a description text with N words. First, we eliminate all negative values and divide each impact score by the maximum value $\max_k$ for that sample and permission p. The normalized score $\widetilde{L}_{p,t}$ can then be scaled by the usage probability y_p the neural network predicted for this permission group. As a result, we obtain a heat value $h_{p,t}$ for each token t between 0 and the prediction output for permission p.

Heat values $h_{p,t}$ quantify the relative impact a particular token has on a predicted permission group and can be visualized as a heatmap over text. Since LIME is only capable of operating on single words, we cannot use it to evaluate our filter kernels that capture up to three tokens. However, in Section 10.5.3, we demonstrate that the technique can still be employed to effectively illustrate good and bad correlations learned by our convolutional neural network.

10.5 Evaluation

The goal of this evaluation is twofold. First, we investigate the performance of our neural network with real-world Android applications. Second, applying our solution on a hand-picked set of apps, we underline the quality of permission predictions and demonstrate how the impact of individual words can be assessed.

10.5.1 Dataset

For the training and evaluation of our network model, we require a pre-trained word embedding model and sample applications.

Android Applications

We evaluate our approach using real-world applications from the PlayDrone dataset [VGN14]. We opted for this repository of apps as it does not only feature raw app archives but also provides metadata from Google Play, including the app description text, category, and download count.

We downloaded 115,294 Android apps and corresponding metadata from the PlayDrone dataset. Then, we filtered apps based on their text embeddability: After preprocessing each description, it had to consist of at least 20 embeddings from the pre-trained model to be used further on. This boundary was set to reduce the potential impact of insignificant samples on the training process.

As highlighted in Table 10.2, we split the resulting set of 77,758 apps into three subsets. The smallest one, the test set, includes 1,000 randomly chosen apps that are not used during training. The remaining apps are added to the training set, and 20% of them are randomly picked to be also part of the to the validation set. This partitioning scheme is required to prevent overfitting of our machine learning model and to ensure meaningful predictions.

Word Embeddings

As our approach requires descriptions to be provided as word embedding vectors, for performance reasons we leverage a pre-trained model that covers a large set of words in English language. Among the two most common architectures for word embeddings are word2vec [Mik+13] and GloVe [PSM14]. The authors of

Table 10.2: Subsets of Android apps used as network input.

	# Apps
Apps crawled	115,294
English descriptions	81,803
20+ embeddable tokens	77,758
Training and validation set	76,758
Test set	1,000

Table 10.3: Properties of word embedding models and performance comparison when applied on the descriptions of 81,803 Android applications. Scores are macro-averaged over all labels and five folds.

	Word2vec	**GloVe**
Architecture	Predictive	Co-occurrence
Text corpora trained on	Wikipedia, UMBC, and statmt.org	Wikipedia
Total words in corpus	$\approx$ 16 billion	$\approx$ 5 billion
Output embeddings	999,994	400,000
Characters	Lowercase	Lowercase
Precision	76%	81%
Recall	54%	55%
F_β**-score**	70%	77%

both techniques provide pre-trained models[3][4] that include the text corpora of Wikipedia, news articles, and crawled websites. As summarized in Table 10.3, both models exhibit substantial differences in terms of the underlying architecture that might have an influence on the representation of words in our neural network. Before employing word-vector pairs for tokenization and word representation, we perform a qualitative comparison between word2vec and GloVe.

We assess the performance of both embeddings models by applying them on all crawled app descriptions. After tokenization and repeated lookups in the pre-trained models, we find that there is no noticeable difference between word2vec and GloVe in terms of matching words with pre-trained embeddings. As GloVe yields marginally better values for precision, recall, and F_β-score, we proceed with GloVe and do not consider word2vec embeddings any further.

10.5.2 Results

We evaluated the performance of our CNN with the descriptions and sets of requested permissions for a total of 77,758 apps across training, validation and test set. The test set results, comprising 1,000 apps, can be seen in Table 10.4, whereby the particular values are averaged independently across the five folds. The rightmost column shows the *support*, indicating the number of true positives found by the model. The varying number for the individual permission groups is mainly caused by their unequal usage in real-world applications. As described in Section 10.4.2, we counter the effects of this imbalance by weighing the error value of each class during training. Over all permission groups, *recall* shows significantly lower values than *precision*. This emerges from the fact that our dataset includes only unfiltered samples, of which many have low-quality descriptions or miss permission-related keywords. In practice, low recall means that the words and context we can rely on to identify a corresponding target class is more limited.

[3] https://code.google.com/archive/p/word2vec/
[4] https://nlp.stanford.edu/projects/glove/

Table 10.4: Test set performance, 5-fold average.

	Precision	**Recall**	F_β**-score**	**Support**
EXTERNAL_STORAGE	93%	76%	89%	699
CALENDAR	55%	24%	42%	36
CALL_LOG	70%	44%	62%	108
CAMERA	71%	56%	67%	199
CONTACTS	80%	65%	76%	369
LOCATION	82%	56%	74%	327
MICROPHONE	83%	64%	78%	125
PHONE	73%	68%	72%	440
SMS	83%	52%	73%	100
Macro Average	77%	56%	70%	
Micro Average	81%	65%	77%	

Table 10.5: Recurring words with high impact, according to their LIME score.

CALENDAR	calendar
CALL_LOG	contact(s), call(s), phone, sms
CAMERA	qr, barcode, scanning, photo, flash(light)
CONTACTS	contact(s), call(s), ringtone(s), friends, sync
EXTERNAL_STORAGE	photo(s), files, sync, mp3, voicemail
LOCATION	gps, map(s), near(est), location, weather
MICROPHONE	microphone, walkie / talkie, record, voice, calls
PHONE	call(s), ringtones, voice, personalized, phone
SMS	sms, text, message, call

Precision varies between 71% and 93%. With the averaged macro precision set at 77%, some permission groups significantly outperform this value, while others are inferior. EXTERNAL_STORAGE has the highest precision, concurrently with the highest support. While CAMERA, PHONE, and CALL_LOG are on the lower end, CALENDAR is worst due to its rare usage. As the scores suggest that some permissions are more clearly identifiable than others, our case study takes a closer look at this assumption and the reasoning of our neural network.

Although the model makes predictions based on formations of up to three words and their local contexts, it is plausible that the frequent occurrence of particular words highly influences the decision process. To evaluate the impact of repetitively used words on predictions, we apply the model explanation algorithm LIME. Therefore, we extract all tokens from all app descriptions in our dataset, sort them by their frequency and LIME score, and repeat this process for each permission group. Table 10.5 presents the most frequently occurring words per prediction class. All listed words are assigned a comparably high LIME score, i.e., have a high impact on true positives, and occur in multiple description texts. Some of these correlations are self-evident, e.g., the word *calendar* occurring when the CALENDAR permission is used, *map* being used in combination with LOCATION permission, and *ringtones* appearing in the context of the PHONE permission.

Keyword overlaps between groups indicate that writers of description texts obviously tend not to distinguish when describing some permission requests. At the same time, it can imply that the implementation of particular functionality typically involves a very precise combination of permissions. Considering the underlying purpose of the CONTACTS, PHONE, and CALL_LOG permissions, such as providing telephone numbers, calling, and storing the record in the call log, a strong inter-relationship seems highly plausible.

The analysis of impact scores shows that the usage of permissions groups, such as CAMERA and LOCATION, is often self-explaining by the use of corresponding keywords. The good interpretability is supported by frequent textual reflections within these groups. A similar situation can also be observed with the CALENDAR and MICROPHONE groups, which tend to have a narrower spectrum of use cases and a lower class support, leading to a semantically very unambiguously assigned set of keywords. The EXTERNAL_STORAGE permission, in contrast, is located at the other end of this spectrum, as it is the most frequently requested permission and commonly lacks good reflections. Comparable effects exist for the CONTACTS, PHONE, and CALL_LOG permissions that partially share the same trigger words. Other correlations between words and permission usage may seem less obvious:

- Weather applications tend to request a user's geographic location, which promotes *weather* as a trigger word for the LOCATION permission group.
- The CONTACTS permission group is often employed for synchronizing a user's address book with online services. Another frequent use-case includes the setting of specific *ringtones* for different contacts.
- The widest range of different tokens occur with EXTERNAL_STORAGE requests. Keywords, such as *screenshot*, *photos*, or *files*, in particular, represent meaningful correlations as read or write access to physical memory will usually depend on that permission.

While these findings are evident for software vendors and experienced users, the link of different keywords to the usage of specific permission groups might not be immediately comprehensible for regular users. Certain relations between, e.g., *weather* and the LOCATION permission exhibit no natural semantic relationship but are entirely the effect of correlation-based learning as is possible with CNNs.

10.5.3 Case Study

For a more in-depth understanding on how our deep neural network operates with real-world descriptions and their qualitative shortcomings, we provide an insight by comparing predicted permissions with actually requested permissions. To obtain knowledge which correlations the model has learned and whether they are meaningful or coincidental, we employ the model explanation algorithm LIME.

For demonstration purposes, we select three real-world apps and highlight interesting findings: the video editing app *AndroMedia*, the messaging app *Snapchat*, and the food delivery app *Lieferheld*. For better readability, we reduce

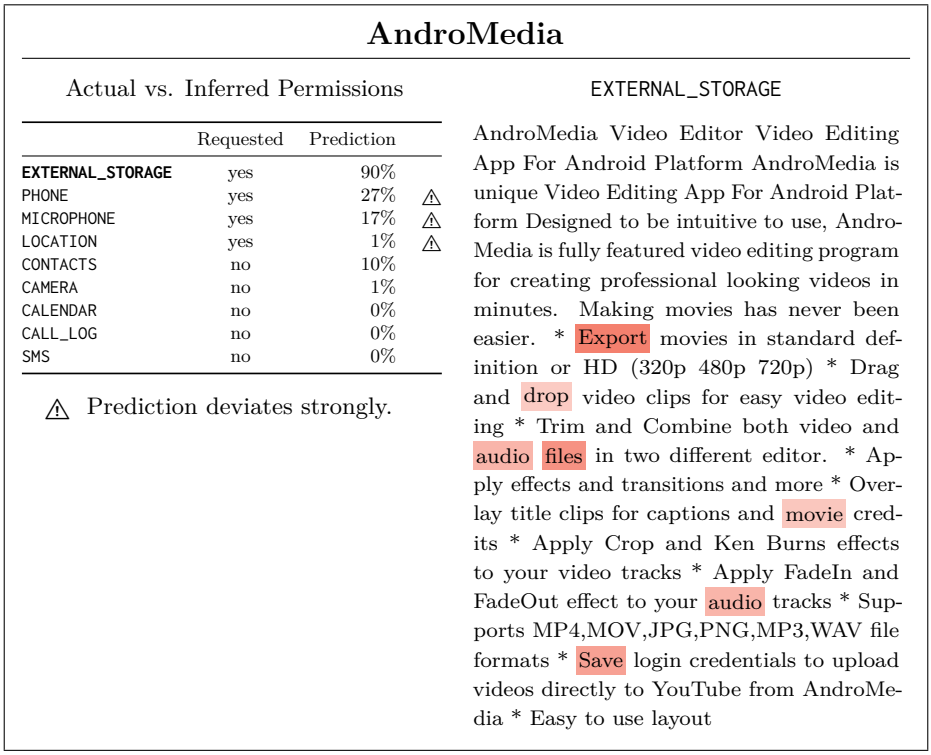


Figure 10.4: *AndroMedia*: actual vs. inferred permission groups based on the app description. Predictions are explained for the permission group `EXTERNAL_STORAGE` using a LIME heatmap overlay.

the presented description texts and show only those sentences that have been identified as relevant for the prediction of permissions.

We repeatedly apply the LIME algorithm for all predictions and words in a description text and derive scores that indicate the individual relevance of words with regard to a prediction outcome. For the text fragments shown in Figures 10.4-10.6, we adjust the background color's opacity of all words, such that the values align with the LIME impact scores. A darker color implies a higher impact on the prediction output. As a result, we can determine single words of descriptions that our system deems to be significant, and see how accurate the prediction is compared to the actual permission.

AndroMedia

Designed for the purpose of video editing, we present the evaluation of this application in Figure 10.4 and highlight words associated with the prediction of the `EXTERNAL_STORAGE` permission. We selected this app for our case study as it underlines the capabilities of our system to identify privacy-critical discrepancies between a described app purpose and actual permission usage.

As pointed out in the description text, the keywords *export*, *drop*, *audio*, *files*, *movie*, and *save* stress the need for the EXTERNAL_STORAGE permission. The *AndroMedia* app also requires granting the LOCATION, MICROPHONE, and PHONE permissions but omits explaining their usage within the description. In fact, there is no obvious argument for a video editing app to need privacy-related permissions. In case they are required from a functional perspective, developers are encouraged to augment the text with an explanation on how they are used.

Snapchat

This app is designed as a social messenger that enables users to send photos, videos, to share the current location and to make calls. In Figure 10.5, we summarize the permissions the app requests and contrast them with the predictions our system makes. To validate the reasoning of our neural network, we highlight the three permission groups CAMERA, EXTERNAL_STORAGE, and MICROPHONE.

The words *snap*, *photo*, *video*, and *capturing* are assigned a high LIME score with regard to the CAMERA permission. Analogous to that, for the prediction of EXTERNAL_STORAGE, *share* and words with a semantic relationship to CAMERA are dominant. This seems reasonable as file sharing typically requires access to the phone's storage. The heat values of the words are comparably strong, which underlines their relevance for the prediction output and a high LIME score overall. For MICROPHONE, the prediction confidence of 77% shows to be mainly caused by the words *chat*, *video*, and *capture*. From the study of recurring words and their affinity to permission usage in Table 10.5, it can be assumed that these values would be higher if the text included words like *voice* or *audio*. Although requested, the SMS and LOCATION permissions do not appear to be linked to the text at all. A manual look at the text confirms this lack and the network outputs of 8% and 1% for these groups. CONTACTS and PHONE are predicted with a score lower than 50%, due to the obvious lack of textual reference. Summarizing, in the description text of *Snapchat*, particular keywords confirm the need for the CAMERA, MICROPHONE, and EXTERNAL_STORAGE permissions. At the same time, it misses semantic indicators that could explain the remaining subset of permissions.

Lieferheld

Depicted in Figure 10.6, *Lieferheld* aims to connect users to a food delivery platform. Our neural network predicts the likelihood that the LOCATION permission is needed with a confidence of 90% and points out the words *current* and *location*. As these two tokens occur adjacent to each other, it seems likely that the word formation *current location* has been repeatedly captured by a 1x2 or 1x3 filter kernel during training. However, since both terms may also appear in other contexts, they exhibit different heat values. The prediction of the CAMERA permission is primarily influenced by the words *barcode*, *camera*, and *QR*. Again, we observe that two words in direct vicinity, *QR* and *barcode*, are highlighted. Interestingly, in the local context of the *Lieferheld* application, these two tokens contribute more actively to the prediction of this permission than the word *camera* itself.

Snapchat

Actual vs. Inferred Permissions

	Requested	Prediction	
CAMERA	yes	90%	
EXTERNAL_STORAGE	yes	90%	
MICROPHONE	yes	77%	
CONTACTS	yes	46%	
PHONE	yes	39%	
LOCATION	yes	8%	⚠
SMS	yes	1%	⚠
CALL_LOG	no	1%	
CALENDAR	no	0%	

⚠ Prediction deviates strongly.

CAMERA

Enjoy fast and fun mobile conversation! Snap a photo or a video, add a caption, and send it to a friend. They'll view it, laugh, and then the Snap disappears from the screen – unless they take a screenshot! You can also add a Snap to your Story with one tap to share your day with all of your friends. [...] if you're both Here, simply press and hold to share live video - and Chat face-to-face! Happy Snapping! *** Please note: even though Snaps, Chats, and Stories are deleted from our servers after they expire, we cannot prevent recipient(s) from capturing and saving the message by taking a screenshot or using an image capture device.

EXTERNAL_STORAGE

Enjoy fast and fun mobile conversation! Snap a photo or a video, add a caption, and send it to a friend. They'll view it, laugh, and then the Snap disappears from the screen – unless they take a screenshot! You can also add a Snap to your Story with one tap to share your day with all of your friends. [...] if you're both Here, simply press and hold to share live video - and Chat face-to-face! Happy Snapping! *** Please note: even though Snaps, Chats, and Stories are deleted from our servers after they expire, we cannot prevent recipient(s) from capturing and saving the message by taking a screenshot or using an image capture device.

MICROPHONE

Enjoy fast and fun mobile conversation! Snap a photo or a video, add a caption, and send it to a friend. They'll view it, laugh, and then the Snap disappears from the screen – unless they take a screenshot! You can also add a Snap to your Story with one tap to share your day with all of your friends. [...] if you're both Here, simply press and hold to share live video - and Chat face-to-face! Happy Snapping! *** Please note: even though Snaps, Chats, and Stories are deleted from our servers after they expire, we cannot prevent recipient(s) from capturing and saving the message by taking a screenshot or using an image capture device.

Figure 10.5: *Snapchat*: actual vs. inferred permission groups based on the app description. Predictions are explained for the permission groups CAMERA, EXTERNAL_STORAGE, and MICROPHONE using LIME heatmap overlays.

For CONTACTS, our network correctly identified that permission usage was not reflected within the text. Overall, the heat values generated for the description of *Lieferheld* show that some permissions are sufficiently well explained. For others, explanations are obviously missing, as also predicted correctly by our network, and developers might be advised to address potential privacy concerns of users.

Lieferheld

Actual vs. Inferred Permissions

	Requested	Prediction	
LOCATION	yes	90%	
CAMERA	yes	90%	
EXTERNAL_STORAGE	yes	71%	
CONTACTS	yes	14%	⚠
PHONE	no	49%	
CALL_LOG	no	1%	
MICROPHONE	no	0%	
CALENDAR	no	0%	
SMS	no	0%	

⚠ Prediction deviates strongly.

LOCATION

With Lieferheld you can conveniently order food at over 7,000 food delivery services throughout Germany – on your Android smartphone! [...] - Fast and convenient food ordering on your smartphone - Over 7,000 food delivery services throughout Germany - Plenty of choice – pizza, pasta, sushi, burgers and much more - Online payment by request - Helpful restaurant reviews Ordering food from a delivery service via app has never been easier! [...] Enter your delivery address or use your current location and Lieferheld will display a clear list of the food delivery services in your area. Reviews from other customers can help you to find the right food delivery service. Build your own menu in the twinkling of an eye and Lieferheld forwards your order to the food delivery service. [...] New feature: When you see a Lieferheld QR-voucher code somewhere, you can simply scan it with the integrated QR barcode scanner and save the voucher to your account. The camera function is solely used for the QR barcode scanner! [...]

CAMERA

With Lieferheld you can conveniently order food at over 7,000 food delivery services throughout Germany – on your Android smartphone! [...] - Fast and convenient food ordering on your smartphone - Over 7,000 food delivery services throughout Germany - Plenty of choice – pizza, pasta, sushi, burgers and much more - Online payment by request - Helpful restaurant reviews Ordering food from a delivery service via app has never been easier! [...] Enter your delivery address or use your current location and Lieferheld will display a clear list of the food delivery services in your area. Reviews from other customers can help you to find the right food delivery service. Build your own menu in the twinkling of an eye and Lieferheld forwards your order to the food delivery service. [...] New feature: When you see a Lieferheld QR-voucher code somewhere, you can simply scan it with the integrated QR barcode scanner and save the voucher to your account. The camera function is solely used for the QR barcode scanner! [...]

Figure 10.6: *Lieferheld*: actual vs. inferred permission groups based on the app description. Predictions are explained for the permission groups LOCATION and CAMERA using LIME heatmap overlays.

10.5.4 Summary

We studied the practical benefit of our neural network from three different angles. First, we showed its metric-based performance. Second, we demonstrated the plausibility of learned words for each permission group. Third, we highlighted single description texts for which our system correctly predicted permission configurations and found discrepancies between the text and the actual permissions.

Our approach for identifying permission-related phrases has stressed the employment of a dynamic, self-learning model. Current approaches strongly rely on static relations, e.g., the word *camera* has to appear in the description text if an Android SDK method includes the exact same word, or the model training process relies on a limited set of hand-picked description sentences. A quick look at high-impact words for the camera permission group above illustrates the superiority of our approach: The use of pre-trained word embeddings and a large description dataset allow the system to learn a broad range of plausible relations that are very specific, from words like *capturing* over *QR/barcode* and *snap*. Manually labeling all these words and their combinations before training is practically infeasible. Our deep learning approach especially addresses the ever-changing nature of app use cases and their evolving peculiarities.

The aspect that sets our approach apart from current solutions arises from the filter-based convolutional architecture, which finds local properties, i.e., words and word groups, and makes global decisions based on them. Selected examples demonstrate that the way a CNN processes text is not only limited to a sentence-based search but takes description-wide information into account. For AndroMedia, trigger words are *export*, *save*, and *files*, but they are be spread across multiple sentences. A system limiting the scope to one sentence cannot interpret this context. Utilizing multiple filter sizes increases the effectiveness by also training on word groups of two and three words. Our evaluation makes clear that crucial information can span over multiple description sentences and that we are capable of capturing these relations.

10.6 Conclusion

Android users often fall for apps with intriguing descriptions and grant them permissions that could adversely impact their privacy. Cross-checking an app's description and its permission usage is challenging due to the lack of enforced quality standards for descriptions and the nonexistent verification of their content.

We presented a solution to reliably assess and grade the need for *dangerous permissions* by analyzing real-world description texts. Based on a convolutional neural network, our approach accurately captures contextual properties in potentially incomplete text and can reliably predict individual groups of semantically related permissions. We carefully evaluated our approach on more than 77,000 real-world apps and uncovered discrepancies between the actual and predicted usage of permissions with a precision between 71% and 93%. Our solution provides an effective method to assess privacy awareness in descriptions of Android apps.

11 Conclusions

With more and more sensitive data being stored and processed by applications for the Android and iOS mobile operating systems, inspecting the implementation and behavior of apps is of vital importance to disclose security-critical problems. The increasing complexity of nowadays applications raises the costs and efforts needed for a thorough security analysis. While existing solutions allow for testing of privacy leaks, implementation flaws, and common attack vectors at the level of individual program statements, their results do not provide an insight into what functionality apps realize and in which context problematic statements occur.

This thesis contributes to obtaining a more sophisticated view on problematic code parts and augmenting the security analysis of mobile applications with a *semantic understanding*. Rather than conducting a security-critical inspection of applications only on the basis of commonly known vulnerabilities, our work broadens the scope to pinpointing the exact origin of problematic statements and to enriching privacy- and security-critical analyses with contextual information.

To understand semantics at a low-level, we filled missing gaps in the static analysis of mobile applications and proposed new frameworks that excel in identifying and tracking security-relevant code in Android and iOS applications. In a case study of 509 Android apps, we disclosed that 36% or 182 out of 509 tested apps expose sensitive user input to files or pass them to log output. Two additional case studies that focused on the improper usage of cryptographic APIs in Android and iOS apps revealed blatant security-critical implementation weaknesses in a majority of them. We also inspected the role of the underlying platform and noticed that the design of APIs on Android and iOS promotes certain mistakes. Our results underline the high relevance and practical benefit of reliable solutions for low-level analysis and stress the need for further research that helps to understand the logical context of code fragments in mobile applications.

For a more holistic analysis of apps, we concentrated on elaborating a semantic understanding of coherent code parts in Android applications. The goal was to lift the analysis of apps from the level of individual program statements to a state that would allow us to reason about the security of functional code dependencies. We started by assessing and measuring the similarity of code and noticed that obfuscation and similar transformation techniques impose a significant challenge in reliably recognizing potentially vulnerable code parts. By fingerprinting code based on transformation-invariant patterns, we accomplished to distinguish actual implementation functionality from low-level code semantics. To efficiently verify behavioral differences between apps, we elaborated a comparison strategy based on Merkle trees. Our iterative comparison approach succeeded in identifying semantically identical code fragments, even if they were moved or obfuscated.

Thousands of program classes, the widespread use of obfuscation, and a variety of ways how developers can implement similar program functionality, make it challenging to summarize the actual main purpose of applications. Augmenting existing approaches for mobile security analysis with contextual information is essential in order to better estimate the practical risks that emerge from found vulnerabilities. In this thesis, we learned that modern techniques from the fields of natural language processing and machine learning can significantly assist in assessing the relevance of individual parts in source code. We leveraged dense and convolutional neural networks to correlate the metadata of apps with the underlying implementation. Our solutions excel in capturing the context of words and phrases in description texts and can link them with features extracted from source code and used system permissions. Our work provides valuable insights into the privacy awareness of Android apps and succeeds in accurately identifying the main purpose of arbitrary apps based on their actual implementation.

Our research provides a strong contribution towards augmenting the security analysis of mobile applications with context-awareness. While our thesis was able to show new perspectives on highly security-critical implementation aspects of Android and iOS applications, further research is needed for a holistic analysis. Considering the continuing hype that mobile devices are experiencing these days, it is to be expected that the amount, complexity, and size of applications will still increase. Accordingly, a rise in terms of monetary costs and time needed for manual inspection can also be foreseen. Hence, there is an evident need to assess the security of mobile programs in an automated way in order to impede possible attack vectors, prevent personal data from being leaked, and to verify that security features are applied in a correct manner. It will not suffice to uncover only specific implementation aspects rather than gaining a more coherent understanding of what applications are actually executing. Recent developments in the field of deep learning also involve algorithms to summarize code, improve code completion systems, predict code properties, and assist in disclosing critical implementation faults. The security analysis of mobile apps can greatly benefit from adopting these achievements to efficiently identify bugs, recognize insecure programming practices, e.g., hard-coded keys, already during development, and to finally better understand the semantic behavior of mobile applications.

Publications

[Fei18] **Johannes Feichtner**. "Hunting Password Leaks in Android Applications." In: *ICT Systems Security and Privacy Protection – IFIP SEC 2018*. Springer, 2018, pp. 278–292. DOI: 10.1007/978-3-319-99828-2_20.

[Fei19] **Johannes Feichtner**. "A Comparative Study of Misapplied Crypto in Android and iOS Applications." In: *Security and Cryptography – SECRYPT 2019*. SciTePress, 2019, pp. 96–108. DOI: 10.5220/0007915300960108.

[FG20a] **Johannes Feichtner** and Stefan Gruber. "Code between the Lines: Semantic Analysis of Android Applications." In: *ICT Systems Security and Privacy Protection – IFIP SEC 2020*. Springer, 2020. In press.

[FG20b] **Johannes Feichtner** and Stefan Gruber. "Understanding Privacy Awareness in Android App Descriptions Using Deep Learning." In: *Conference on Data and Application Security and Privacy – CODASPY'20*. ACM, 2020, pp. 203–214. DOI: 10.1145/3374664.3375730.

[FMS18] **Johannes Feichtner**, David Missmann, and Raphael Spreitzer. "Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications." In: *Security & Privacy in Wireless and Mobile Networks – WiSec'18*. **Best Paper Award**. ACM, 2018, pp. 236–247. DOI: 10.1145/3212480.3212487.

[FNZ19] **Johannes Feichtner**, Lukas Neugebauer, and Dominik Ziegler. "Mind the Gap: Finding What Updates Have (Really) Changed in Android Applications." In: *Security and Cryptography – SECRYPT 2019*. SciTePress, 2019, pp. 306–313. DOI: 10.5220/0008119303060313.

[FR19] **Johannes Feichtner** and Christof Rabensteiner. "Obfuscation-Resilient Code Recognition in Android Apps." In: *Availability, Reliability and Security – ARES'19*. ACM, 2019, pp. 1–10. DOI: 10.1145/3339252.3339260.

Bibliography

[Aba+16] Martin Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning.” In: *Symposium on Operating Systems Design and Implementation – OSDI’16*. USENIX Association, 2016, pp. 265–283.

[Aca+17] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. “Comparing the Usability of Cryptographic APIs.” In: *Security and Privacy – S&P’17*. IEEE Computer Society, 2017, pp. 154–171. ISBN: 978-1-5090-5533-3.

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. “Dynamic Slicing in the Presence of Unconstrained Pointers.” In: *Symposium on Software Testing and Analysis – ISSTA’91*. ACM, 1991, pp. 60–73. ISBN: 0-89791-449-X.

[AI13] Marat Kh. Akhin and Vladimir M. Itsykson. “Tree slicing: Finding intertwined and gapped clones in one simple step.” In: *Automatic Control and Computer Sciences* 47 (2013), pp. 427–432.

[Akr+20] Junaid Akram, Zhendong Shi, Majid Mumtaz, and Ping Luo. “DroidSD: An Efficient Indexed Based Android Applications Similarity Detection Tool.” In: *J. Inf. Sci. Eng.* 36 (2020), pp. 13–29.

[Alf+19] Emily Alfs, Doina Caragea, Nathan Albin, and Pietro Poggi-Corradini. “Identifying Android Malware Using Network-Based Approaches.” In: *Conference on Artificial Intelligence – AAAI’19*. AAAI Press, 2019, pp. 9911–9912. ISBN: 978-1-57735-809-1.

[And94] L. O. Andersen. “Program Analysis and Specialization for the C Programming Language.” (DIKU 94/19). PhD thesis. DIKU, University of Copenhagen, May 1994.

[AP18] Anshul Arora and Sateesh Kumar Peddoju. “NTPDroid: A Hybrid Android Malware Detector Using Network Traffic and System Permissions.” In: *Trust, Security and Privacy in Computing and Communications – TrustCom’18*. IEEE, 2018, pp. 808–813. ISBN: 978-1-5386-4388-4.

[Arz+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-aware Taint Analysis for Android Apps." In: *Programming Language Design and Implementation – PLDI'14*. ACM, 2014, pp. 259–269. ISBN: 978-1-4503-2784-8.

[Bac+16] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. "R-Droid: Leveraging Android App Analysis with Static Slice Optimization." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2016, pp. 129–140. ISBN: 978-1-4503-4233-9.

[Bar+15] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. "Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T)." In: *Knowledge-Based Software Engineering Conference – KBSE'15*. IEEE Computer Society, 2015, pp. 669–679. ISBN: 978-1-5090-0025-8.

[Bax+98] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. "Clone Detection Using Abstract Syntax Trees." In: *International Conference on Software Maintenance – ICSM'98*. IEEE Computer Society, 1998, pp. 368–377. ISBN: 0-8186-8779-7.

[BB12] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization." In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305.

[BBD16] Michael Backes, Sven Bugiel, and Erik Derr. "Reliable Third-Party Library Detection in Android and its Security Applications." In: *Conference on Computer and Communications Security – CCS'16*. ACM, 2016, pp. 356–367. ISBN: 978-1-4503-4139-4.

[BDV00] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. "A Neural Probabilistic Language Model." In: *Neural Information Processing Systems – NIPS'00*. MIT Press, 2000, pp. 932–938.

[Bel+97] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. "A Concrete Security Treatment of Symmetric Encryption." In: *Symposium on Foundations of Computer Science – FOCS'97*. IEEE Computer Society, 1997, pp. 394–403. ISBN: 0-8186-8197-7.

[Ber+03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. "Points-to analysis using BDDs." In: *Programming Language Design and Implementation – PLDI'03*. ACM, 2003, pp. 103–114. ISBN: 1-58113-662-5.

[BH04] David Binkley and Mark Harman. "A Survey of Empirical Results on Program Slicing." In: *Advances in Computers* 62 (2004), pp. 105–178.

[Bia+15] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. "What the App is That? Deception and Countermeasures in the Android User Interface." In: *Security and Privacy – S&P'15*. IEEE Computer Society, 2015, pp. 931–948. ISBN: 978-1-4673-6949-7.

[Bic+16] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. "Statistical Deobfuscation of Android Applications." In: *Conference on Computer and Communications Security – CCS'16*. ACM, 2016, pp. 343–355. ISBN: 978-1-4503-4139-4.

[Bis07] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

[BMM18] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. "A systematic study of the class imbalance problem in convolutional neural networks." In: *Neural Networks* 106 (2018), pp. 249–259.

[BPM17] Richard Baumann, Mykolai Protsenko, and Tilo Müller. "AntiProGuard: Towards Automated Deobfuscation of Android Apps." In: *Workshop on Security in Highly Connected IT Systems – SHIS*. ACM, 2017, pp. 7–12. ISBN: 978-1-4503-5271-0.

[Cao+15] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework." In: *Network and Distributed System Security Symposium – NDSS'15*. The Internet Society, 2015.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Symposium on Operating Systems Design and Implementation – OSDI'08*. USENIX Association, 2008, pp. 209–224. ISBN: 978-1-931971-65-2.

[CGC12] Jonathan Crussell, Clint Gibler, and Hao Chen. "Attack of the Clones: Detecting Cloned Applications on Android Markets." In: *European Symposium on Research in Computer Security – ESORICS'12*. Vol. 7459. LNCS. Springer, 2012, pp. 37–54. ISBN: 978-3-642-33166-4.

[CGC13] Jonathan Crussell, Clint Gibler, and Hao Chen. "AnDarwin: Scalable Detection of Semantically Similar Android Applications." In: *European Symposium on Research in Computer Security – ESORICS'13*. Vol. 8134. LNCS. Springer, 2013, pp. 182–199. ISBN: 978-3-642-40202-9.

[Che+15] Jian Chen, Manar H. Alalfi, Thomas R. Dean, and Ying Zou. "Detecting Android Malware Using Clone Detection." In: *J. Comput. Sci. Technol.* 30 (2015), pp. 942–956.

[Che+16] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. "Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS." In: *Security and Privacy – S&P'16*. IEEE Computer Society, 2016, pp. 357–376. ISBN: 978-1-5090-0824-7.

[Cho+14] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." In: *Empirical Methods in Natural Language Processing – EMNLP'14*. ACL, 2014, pp. 1724–1734. ISBN: 978-1-937284-96-1.

[CHY12] Patrick P. F. Chan, Lucas Chi Kwong Hui, and Siu-Ming Yiu. "DroidChecker: Analyzing Android Applications for Capability Leak." In: *Security and Privacy in Wireless and Mobile Networks – WISEC'12*. ACM, 2012, pp. 125–136. ISBN: 978-1-4503-1265-3.

[CLZ14] Kai Chen, Peng Liu, and Yingjun Zhang. "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets." In: *International Conference on Software Engineering – ICSE'14*. ACM, 2014, pp. 175–186. ISBN: 978-1-4503-2756-5.

[Cox+14] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. "SpanDex: Secure Password Tracking for Android." In: *USENIX Security'14*. USENIX Association, 2014, pp. 481–494.

[CZ15] Xin Chen and Sencun Zhu. "DroidJust: Automated Functionality-aware Privacy Leakage Analysis for Android Applications." In: *Security and Privacy in Wireless and Mobile Networks – WISEC'15*. ACM, 2015, 5:1–5:12. ISBN: 978-1-4503-3623-9.

[Dav+10] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. "Privilege Escalation Attacks on Android." In: *Information Security – ISC'10*. Vol. 6531. LNCS. Springer, 2010, pp. 346–360. ISBN: 978-3-642-18177-1.

[Den+15] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. "iRiS: Vetting Private API Abuse in iOS Applications." In: *Conference on Computer and Communications Security – CCS'15*. ACM, 2015, pp. 44–56. ISBN: 978-1-4503-3832-5.

[Der+17] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. "Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android." In: *Conference on Computer and Communications Security – CCS'17*. ACM, 2017, pp. 2187–2200. ISBN: 978-1-4503-4946-8.

[Des12] Anthony Desnos. "Android: Static Analysis Using Similarity Distance." In: *Conference on Systems Science – HICSS'12*. IEEE Computer Society, 2012, pp. 5394–5403. ISBN: 978-0-7695-4525-7.

[DNL14] Luke Deshotels, Vivek Notani, and Arun Lakhotia. "DroidLegacy: Automated Familial Classification of Android Malware." In: *Program Protection and Reverse Engineering Workshop – PPREW*. ACM, 2014, 3:1–3:12. ISBN: 978-1-4503-2649-0.

[Ege+11] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. "PiOS: Detecting Privacy Leaks in iOS Applications." In: *Network and Distributed System Security Symposium – NDSS'11*. The Internet Society, 2011.

[Ege+13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. "An Empirical Study of Cryptographic Misuse in Android Applications." In: *Conference on Computer and Communications Security – CCS'13*. ACM, 2013, pp. 73–84. ISBN: 978-1-4503-2477-9.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers." In: *Programming Language Design and Implementation – PLDI'94*. ACM, 1994, pp. 242–256. ISBN: 0-89791-662-X.

[Enc+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *Symposium on Operating Systems Design and Implementation – OSDI'10*. USENIX Association, 2010, pp. 393–407. ISBN: 978-1-931971-79-9.

[Fah+12] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. "Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security." In: *Conference on Computer and Communications Security – CCS'12*. ACM, 2012, pp. 50–61. ISBN: 978-1-4503-1651-4.

[Fan+20] Yong Fang, Yangchen Gao, Fan Jing, and Lei Zhang. "Android Malware Familial Classification Based on DEX File Section Features." In: *IEEE Access* 8 (2020), pp. 10614–10627.

[Fel+11] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. "Permission Re-Delegation: Attacks and Defenses." In: *USENIX Security'11*. USENIX Association, 2011.

[Fen+14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. "Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis." In: *Foundations of Software Engineering – FSE'14*. ACM, 2014, pp. 576–587. ISBN: 978-1-4503-3056-5.

[Fis+17] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security." In: *Security and Privacy – S&P'17*. IEEE Computer Society, 2017, pp. 121–136. ISBN: 978-1-5090-5533-3.

[GAM17] Alex Gittens, Dimitris Achlioptas, and Michael W. Mahoney. "Skip-Gram - Zipf + Uniform = Vector Additivity." In: *Association for Computational Linguistics – ACL'17*. Association for Computational Linguistics, 2017, pp. 69–76. ISBN: 978-1-945626-75-3.

[Gao+19] Hongcan Gao, Chenkai Guo, Yanfeng Wu, Naipeng Dong, Xiaolei Hou, Sihan Xu, and Jing Xu. "AutoPer: Automatic Recommender for Runtime-Permission in Android Applications." In: *Conference on Computers, Software and Applications – COMPSAC'19*. IEEE, 2019, pp. 107–116. ISBN: 978-1-7281-2607-4.

[Gas+13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Structural Detection of Android Malware Using Embedded Call Graphs." In: *Artificial Intelligence and Security – AISec*. ACM, 2013, pp. 45–54. ISBN: 978-1-4503-2488-5.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `https://www.deeplearningbook.org`. MIT Press, 2016.

[GHM18] Joshua Garcia, Mahmoud Hammad, and Sam Malek. "Lightweight, obfuscation-resilient detection and family identification of Android malware." In: *International Conference on Software Engineering – ICSE'18*. ACM, 2018, p. 497. ISBN: 978-1-4503-5638-1.

[Gib+12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale." In: *Trust and Trustworthy Computing – TRUST'12*. Vol. 7344. LNCS. Springer, 2012, pp. 291–307. ISBN: 978-3-642-30920-5.

[Gla+17] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. "CodeMatch: obfuscation won't conceal your repackaged app." In: *Foundations of Software Engineering – FSE'17*. ACM, 2017, pp. 638–648. ISBN: 978-1-4503-5105-8.

[Gor+14] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. "Checking app behavior against app descriptions." In: *International Conference on Software Engineering – ICSE'14*. ACM, 2014, pp. 1025–1035. ISBN: 978-1-4503-2756-5.

[Gra+12a] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. "Unsafe exposure analysis of mobile in-app advertisements." In: *Security and Privacy in Wireless and Mobile Networks – WISEC'12*. ACM, 2012, pp. 101–112. ISBN: 978-1-4503-1265-3.

[Gra+12b] Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. "RiskRanker: Scalable and Accurate Zero-Day Android Malware Detection." In: *Mobile Systems – MobiSys'12*. ACM, 2012, pp. 281–294. ISBN: 978-1-4503-1301-8.

[Gua+16] Quanlong Guan, Heqing Huang, Weiqi Luo, and Sencun Zhu. "Semantics-Based Repackaging Detection for Mobile Apps." In: *Engineering Secure Software and Systems – ESSoS'16*. Vol. 9639. LNCS. Springer, 2016, pp. 89–105. ISBN: 978-3-319-30805-0.

[HGM18] Mahmoud Hammad, Joshua Garcia, and Sam Malek. "A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products." In: *International Conference on Software Engineering – ICSE'18*. ACM, 2018, pp. 421–431. ISBN: 978-1-4503-5638-1.

[HK18] TonTon Hsien-De Huang and Hung-Yu Kao. "R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections." In: *Conference on Big Data – BIGDATA'18*. IEEE, 2018, pp. 2633–2642. ISBN: 978-1-5386-5035-6.

[HKC19] Masoud Reyhani Hamedani, Gyoosik Kim, and Seong-je Cho. "SimAndro: an effective method to compute similarity of Android applications." In: *Soft Comput.* 23 (2019), pp. 7569–7590.

[HL07a] Ben Hardekopf and Calvin Lin. "Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis." In: *Static Analysis Symposium – SAS'07*. Vol. 4634. LNCS. Springer, 2007, pp. 265–280. ISBN: 978-3-540-74060-5.

[HL07b] Ben Hardekopf and Calvin Lin. "The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code." In: *Programming Language Design and Implementation – PLDI'07*. ACM, 2007, pp. 290–299. ISBN: 978-1-59593-633-2.

[HMM12] Julien Henry, David Monniaux, and Matthieu Moy. "PAGAI: A Path Sensitive Static Analyser." In: *Electr. Notes Theor. Comput. Sci.* 289 (2012), pp. 15–25.

[HP01] Michael Hind and Anthony Pioli. "Evaluating the Effectiveness of Pointer Alias Analyses." In: *Sci. Comput. Program.* 39 (2001), pp. 31–55.

[HRB90] Susan Horwitz, Thomas W. Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs." In: *ACM Trans. Prog. Lang. Syst.* 12 (1990), pp. 26–60.

[HS96] Sepp Hochreiter and Jürgen Schmidhuber. "LSTM can Solve Hard Long Time Lag Problems." In: *Neural Information Processing Systems – NIPS'96*. MIT Press, 1996, pp. 473–479.

[HT01] Nevin Heintze and Olivier Tardieu. "Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second." In: *Programming Language Design and Implementation – PLDI'01*. ACM, 2001, pp. 254–263. ISBN: 1-58113-414-2.

[HXY15] Zhiheng Huang, Wei Xu, and Kai Yu. "Bidirectional LSTM-CRF Models for Sequence Tagging." In: *CoRR* abs/1508.01991 (2015).

[JK19] Justin M. Johnson and Taghi M. Khoshgoftaar. "Survey on deep learning with class imbalance." In: *J. Big Data* 6 (2019), p. 27.

[JS02] Nathalie Japkowicz and Shaju Stephen. "The class imbalance problem: A systematic study." In: *Intell. Data Anal.* 6 (2002), pp. 429–449.

[KCJ15] Deguang Kong, Lei Cen, and Hongxia Jin. "AUTOREB: Automatically Understanding the Review-to-Behavior Fidelity in Android Applications." In: *Conference on Computer and Communications Security – CCS'15*. ACM, 2015, pp. 530–541. ISBN: 978-1-4503-3832-5.

[Kim14] Yoon Kim. "Convolutional Neural Networks for Sentence Classification." In: *Empirical Methods in Natural Language Processing – EMNLP'14*. ACL, 2014, pp. 1746–1751. ISBN: 978-1-937284-96-1.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[Kuh10] Harold W. Kuhn. "The Hungarian Method for the Assignment Problem." In: *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer, 2010, pp. 29–47.

[Laz+14] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. "Why does cryptographic software fail?: a case study and open problems." In: *Asia-Pacific Workshop on Systems – APSys'14*. ACM, 2014, 7:1–7:7. ISBN: 978-1-4503-3024-4.

[LeC+89] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition." In: *Neural Computation* 1 (1989), pp. 541–551.

[Li+14] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. "iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications." In: *Network and System Security – NSS'14*. Vol. 8792. LNCS. Springer, 2014, pp. 349–362. ISBN: 978-3-319-11697-6.

[Li+15] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." In: *International Conference on Software Engineering – ICSE'15*. IEEE Computer Society, 2015, pp. 280–291. ISBN: 978-1-4799-1934-5.

[Li+16] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "An Investigation into the Use of Common Libraries in Android Apps." In: *Software Analysis, Evolution, and Reengineering – SANER'16*. IEEE Computer Society, 2016, pp. 403–414. ISBN: 978-1-5090-1855-0.

[Lin+16] Zimin Lin, Rui Wang, Xiaoqi Jia, Shengzhi Zhang, and Chuankun Wu. "Analyzing Android Repackaged Malware by Decoupling Their Event Behaviors." In: *International Workshop on Security – IWSEC'16*. Vol. 9836. LNCS. Springer, 2016, pp. 3–20. ISBN: 978-3-319-44523-6.

[Liu+15] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. "Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps." In: *Mobile Systems – MobiSys'15*. ACM, 2015, pp. 89–103. ISBN: 978-1-4503-3494-5.

[Liu+20] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. "Privacy Risk Analysis and Mitigation of Analytics Libraries in the Android Ecosystem." In: *IEEE Trans. Mob. Comput.* 19 (2020), pp. 1184–1199.

[LL17] Scott M. Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions." In: *Neural Information Processing Systems – NIPS'17*. 2017, pp. 4765–4774.

[Luo+14] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection." In: *Foundations of Software Engineering – FSE'14*. ACM, 2014, pp. 389–400. ISBN: 978-1-4503-3056-5.

[Ma+16a] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. "CDRep: Automatic Repair of Cryptographic Misuses in Android Applications." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2016, pp. 711–722. ISBN: 978-1-4503-4233-9.

[Ma+16b] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. "LibRadar: fast and accurate detection of third-party libraries in Android apps." In: *International Conference on Software Engineering – ICSE'16*. ACM, 2016, pp. 653–656. ISBN: 978-1-4503-3900-1.

[May+19] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. "The Android Platform Security Model." In: *CoRR* abs/1904.05572 (2019).

[MBB18] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. "Source Attribution of Cryptographic API Misuse in Android Applications." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2018, pp. 133–146.

[McL+17] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Y. Yerima, Paul C. Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupe, and Gail-Joon Ahn. "Deep Android Malware Detection." In: *Conference on Data and Application Security and Privacy – CODASPY'17*. ACM, 2017, pp. 301–308. ISBN: 978-1-4503-4523-1.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR." In: *Verified Software: Theories, Tools, Experiments – VSTTE'12*. Vol. 7152. LNCS. Springer, 2012, pp. 146–161. ISBN: 978-3-642-27704-7.

[Mik+10] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. "Recurrent neural network based language model." In: *International Speech Communication Association – INTERSPEECH'10*. ISCA, 2010, pp. 1045–1048.

[Mik+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. "Distributed Representations of Words and Phrases and their Compositionality." In: *Neural Information Processing Systems – NIPS'13*. 2013, pp. 3111–3119.

[Mir+19] Omid Mirzaei, Guillermo Suarez-Tangil, José Maria de Fuentes, Juan Tapiador, and Gianluca Stringhini. "AndrEnsemble: Leveraging API Ensembles to Characterize Android Malware Families." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2019, pp. 307–314. ISBN: 978-1-4503-6752-3.

[MKR17] Kathleen M. Moriarty, Burt Kaliski, and Andreas Rusch. "PKCS #5: Password-Based Cryptography Specification Version 2.1." In: *RFC* 8018 (2017), pp. 1–40.

[MS12] Christopher Mann and Artem Starostin. "A framework for static detection of privacy leaks in android applications." In: *Symposium on Applied Computing – SAC'12*. ACM, 2012, pp. 1457–1462. ISBN: 978-1-4503-0857-1.

[Nad+16] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. "Jumping through hoops: why do Java developers struggle with cryptography APIs?" In: *International Conference on Software Engineering – ICSE'16*. ACM, 2016, pp. 935–946. ISBN: 978-1-4503-3900-1.

[NCC14] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. "AdDetect: Automated detection of Android ad libraries using semantic analysis." In: *International Conference on Intelligent Sensors, Sensor Networks and Information Processing – ISSNIP'14*. IEEE, 2014, pp. 1–6. ISBN: 978-1-4799-2843-9.

[Ngu+09] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. "Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection." In: *Fundamental Approaches to Software Engineering – FASE'09*. Vol. 5503. LNCS. Springer, 2009, pp. 440–455. ISBN: 978-3-642-00592-3.

[Nor+17] Thanapon Noraset, Chen Liang, Larry Birnbaum, and Doug Downey. "Definition Modeling: Learning to Define Word Embeddings in Natural Language." In: *Conference on Artificial Intelligence – AAAI'17*. AAAI Press, 2017, pp. 3259–3266.

[OC15] Lucky Onwuzurike and Emiliano De Cristofaro. "Danger is my middle name: experimenting with SSL vulnerabilities in Android apps." In: *Security and Privacy in Wireless and Mobile Networks – WISEC'15*. ACM, 2015, 15:1–15:6. ISBN: 978-1-4503-3623-9.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. "The Program Dependence Graph in a Software Development Environment." In: *Software Engineering Symposium on Practical Software Development Environments – SDE'84*. ACM, 1984, pp. 177–184. ISBN: 0-89791-131-8.

[Pan+13] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. "WHYPER: Towards Automating Risk Assessment of Mobile Applications." In: *USENIX Security'13*. USENIX Association, 2013, pp. 527–542. ISBN: 978-1-931971-03-4.

[Par+19] Minjae Park, Geunha You, Seong-je Cho, Minkyu Park, and Sangchul Han. "A Framework for Identifying Obfuscation Techniques applied to Android Apps using Machine Learning." In: *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 10 (2019), pp. 22–30.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Neural Information Processing Systems – NIPS'19*. 2019, pp. 8024–8035.

[PCJ18] Jungsoo Park, Hojin Chun, and Souhwan Jung. "API and permission-based classification system for Android malware analysis." In: *Conference on Information Networking – ICOIN'18*. IEEE, 2018, pp. 930–935. ISBN: 978-1-5386-2290-2.

[PKH07] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. "Efficient Field-Sensitive Pointer Analysis of C." In: *ACM Trans. Program. Lang. Syst.* 30 (2007), p. 4.

[Pod+18] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels." In: *USENIX Security'18*. USENIX Association, 2018, pp. 549–566.

[Poe+14] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications." In: *Network and Distributed System Security Symposium – NDSS'14*. The Internet Society, 2014.

[Por80] Martin F. Porter. "An algorithm for suffix stripping." In: *Program* 14 (1980), pp. 130–137.

[Pot+12] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. "Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques." In: *Engineering Secure Software and Systems – ESSoS'12*. Vol. 7159. LNCS. Springer, 2012, pp. 106–120. ISBN: 978-3-642-28165-5.

[PPK18] Anthony Peruma, Jeffrey Palmerino, and Daniel E. Krutz. "Investigating user perception and comprehension of Android permission models." In: *Conference on Mobile Software Engineering and Systems – MOBILESoft@ICSE*. ACM, 2018, pp. 56–66.

[PS17] José Gaviria de la Puerta and Borja Sanz. "Using Dalvik opcodes for malware detection on android." In: *Log. J. IGPL* 25 (2017), pp. 938–948.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global Vectors for Word Representation." In: *Empirical Methods in Natural Language Processing – EMNLP'14*. ACL, 2014, pp. 1532–1543. ISBN: 978-1-937284-96-1.

[Qu+14] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. "AutoCog: Measuring the Description-to-permission Fidelity in Android Applications." In: *Conference on Computer and Communications Security – CCS'14*. ACM, 2014, pp. 1354–1365. ISBN: 978-1-4503-2957-6.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *Network and Distributed System Security Symposium – NDSS'14*. The Internet Society, 2014.

[Ren+18] Jingjing Ren, Martina Lindorfer, Daniel J. Dubois, Ashwin Rao, David R. Choffnes, and Narseo Vallina-Rodriguez. "Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions." In: *Network and Distributed System Security Symposium – NDSS'18*. The Internet Society, 2018.

[RSG16] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier." In: *Conference on Knowledge Discovery and Data Mining – SIGKDD'16*. ACM, 2016, pp. 1135–1144. ISBN: 978-1-4503-4232-2.

[Sco+18] Gian Luca Scoccia, Stefano Ruberto, Ivano Malavolta, Marco Autili, and Paola Inverardi. "An investigation into Android run-time permissions from the end users' perspective." In: *Conference on Mobile Software Engineering and Systems – MOBILESoft@ICSE*. ACM, 2018, pp. 45–55.

[SDA02] Benjamin Schwarz, Saumya K. Debray, and Gregory R. Andrews. "Disassembly of Executable Code Revisited." In: *Software Analysis, Evolution, and Reengineering – SANER'02*. IEEE Computer Society, 2002, pp. 45–54. ISBN: 0-7695-1799-4.

[Seo+16] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. "FLEXDROID: Enforcing In-App Privilege Separation in Android." In: *Network and Distributed System Security Symposium – NDSS'16*. The Internet Society, 2016.

[SH97a] Marc Shapiro and Susan Horwitz. "Fast and Accurate Flow-Insensitive Points-To Analysis." In: *Principles of Programming Languages – POPL'97*. ACM Press, 1997, pp. 1–14. ISBN: 0-89791-853-3.

[SH97b] Marc Shapiro and Susan Horwitz. "The Effects of the Precision of Pointer Analysis." In: *Static Analysis Symposium – SAS'97*. Vol. 1302. LNCS. Springer, 1997, pp. 16–34. ISBN: 3-540-63468-1.

[Sha+14a] Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications." In: *Conference on Dependable, Autonomic and Secure Computing – DASC'14*. IEEE Computer Society, 2014, pp. 75–80. ISBN: 978-1-4799-5079-9.

[Sha+14b] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. "Towards a scalable resource-driven approach for detecting repackaged Android applications." In: *Annual Computer Security Applications Conference – ACSAC'14*. ACM, 2014, pp. 56–65. ISBN: 978-1-4503-3005-3.

[SKT17] Julian Schütte, Alexander Kuechler, and Dennis Titze. "Practical Application-Level Dynamic Taint Analysis of Android Apps." In: *Trust, Security and Privacy in Computing and Communications – TrustCom'17*. IEEE Computer Society, 2017, pp. 17–24. ISBN: 978-1-5090-4906-6.

[SL09] Marina Sokolova and Guy Lapalme. "A systematic analysis of performance measures for classification tasks." In: *Inf. Process. Manage.* 45 (2009), pp. 427–437.

[SLL15] Mingshen Sun, Mengmeng Li, and John C. S. Lui. "DroidEagle: seamless detection of visually similar Android apps." In: *Security and Privacy in Wireless and Mobile Networks – WISEC'15*. ACM, 2015, 9:1–9:12. ISBN: 978-1-4503-3623-9.

[Spr+13] Michael Spreitzenbarth, Felix C. Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. "Mobile-sandbox: having a deeper look into android applications." In: *Symposium on Applied Computing – SAC'13*. ACM, 2013, pp. 1808–1815. ISBN: 978-1-4503-1656-9.

[SSE15] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian G. Elbaum. "How developers search for code: a case study." In: *Foundations of Software Engineering – FSE'15*. ACM, 2015, pp. 191–201. ISBN: 978-1-4503-3675-8.

[Ste96] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time." In: *Principles of Programming Languages – POPL'96*. ACM Press, 1996, pp. 32–41. ISBN: 0-89791-769-3.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks." In: *Neural Information Processing Systems – NIPS'14*. 2014, pp. 3104–3112.

[SWL16] Mingshen Sun, Tao Wei, and John C. S. Lui. "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime." In: *Conference on Computer and Communications Security – CCS'16*. ACM, 2016, pp. 331–342. ISBN: 978-1-4503-4139-4.

[Tah+20] Rahim Taheri, Meysam Ghahramani, Reza Javidan, Mohammad Shojafar, Zahra Pooranian, and Mauro Conti. "Similarity-based Android malware detection using Hamming distance of static binary features." In: *Future Gener. Comput. Syst.* 105 (2020), pp. 230–247.

[Tia+20] Ke Tian, Danfeng Yao, Barbara G. Ryder, Gang Tan, and Guojun Peng. "Detection of Repackaged Android Malware with Code-Heterogeneity Features." In: *IEEE Trans. Dependable Secur. Comput.* 17 (2020), pp. 64–77.

[Tip95] Frank Tip. "A Survey of Program Slicing Techniques." In: *J. Prog. Lang.* 3 (1995).

[Vás+14] Mario Linares Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. "Revisiting Android reuse studies in the context of code obfuscation and library usages." In: *Mining Software Repositories – MSR'14*. ACM, 2014, pp. 242–251. ISBN: 978-1-4503-2863-0.

[VGN14] Nicolas Viennot, Edward Garcia, and Jason Nieh. "A measurement study of google play." In: *Measurement and Modeling of Computer Systems – SIGMETRICS'14*. ACM, 2014, pp. 221–233. ISBN: 978-1-4503-2789-3.

[VHP16] Mario Linares Vásquez, Andrew Holtzhauer, and Denys Poshyvanyk. "On automatically detecting similar Android apps." In: *International Conference on Program Comprehension – ICPC'16*. IEEE Computer Society, 2016, pp. 1–10. ISBN: 978-1-5090-1428-6.

[Wan+15] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. "WuKong: a scalable and accurate two-phase approach to Android app clone detection." In: *Symposium on Software Testing and Analysis – ISSTA'15*. ACM, 2015, pp. 71–82. ISBN: 978-1-4503-3620-8.

[Wat+15] Takuya Watanabe, Mitsuaki Akiyama, Tetsuya Sakai, and Tatsuya Mori. "Understanding the Inconsistencies between Text Descriptions and the Use of Privacy-sensitive Resources of Mobile Apps." In: *Symposium On Usable Privacy and Security – SOUPS'15*. USENIX Association, 2015, pp. 241–255. ISBN: 978-1-931971-249.

[Wei81] Mark Weiser. "Program Slicing." In: *International Conference on Software Engineering – ICSE'81*. IEEE Computer Society, 1981, pp. 439–449. ISBN: 0-89791-146-6.

[Wer+18] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. "A Large Scale Investigation of Obfuscation Use in Google Play." In: *Annual Computer Security Applications Conference – ACSAC'18*. ACM, 2018, pp. 222–235. ISBN: 978-1-4503-6569-7.

[WHG15] Haoyu Wang, Jason I. Hong, and Yao Guo. "Using text mining to infer the purpose of permission use in mobile apps." In: *Conference on Pervasive and Ubiquitous Computing – UbiComp'15*. ACM, 2015, pp. 1107–1118. ISBN: 978-1-4503-3574-4.

[WL95] Robert P. Wilson and Monica S. Lam. "Efficient Context-Sensitive Pointer Analysis for C Programs." In: *Programming Language Design and Implementation – PLDI'95*. ACM, 1995, p. 1. ISBN: 0-89791-697-2.

[WYL17] Jingzheng Wu, Mutian Yang, and Tianyue Luo. "PACS: Pemission abuse checking system for android applictions based on review mining." In: *Conference on Dependable and Secure Computing – DSC'17*. IEEE, 2017, pp. 251–258. ISBN: 978-1-5090-5569-2.

[Yas+19] Tatsuhiko Yasumatsu, Takuya Watanabe, Fumihiro Kanei, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. "Understanding the Responsiveness of Mobile App Developers to Software Library Updates." In: *Conference on Data and Application Security and Privacy – CODASPY'19*. ACM, 2019, pp. 13–24. ISBN: 978-1-4503-6099-9.

[Yu+18] Le Yu, Xiapu Luo, Chenxiong Qian, Shuai Wang, and Hareton K. N. Leung. "Enhancing the Description-to-Behavior Fidelity in Android Apps with Privacy Policy." In: *IEEE Trans. Software Eng.* 44 (2018), pp. 834–854.

[YY12] Zhemin Yang and Min Yang. "LeakMiner: Detect Information Leakage on Android with Static Taint Analysis." In: *World Congress on Software Engineering – WCSE 2012*. 2012, pp. 101–104.

[Zha+14a] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. “ViewDroid: towards obfuscation-resilient mobile application repackaging detection.” In: *Security and Privacy in Wireless and Mobile Networks – WISEC’14*. ACM, 2014, pp. 25–36. ISBN: 978-1-4503-2972-9.

[Zha+14b] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. “FSquaDRA: Fast Detection of Repackaged Applications.” In: *Data and Applications Security and Privacy – DBSec’14*. Vol. 8566. LNCS. Springer, 2014, pp. 130–145. ISBN: 978-3-662-43935-7.

[Zha+18a] Jixin Zhang, Zheng Qin, Kehuan Zhang, Hui Yin, and Jingfu Zou. “Dalvik Opcode Graph Based Android Malware Variants Detection Using Global Topology Features.” In: *IEEE Access* 6 (2018), pp. 51964–51974.

[Zha+18b] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. “Detecting third-party libraries in Android applications with high precision and recall.” In: *Software Analysis, Evolution, and Reengineering – SANER’18*. IEEE Computer Society, 2018, pp. 141–152. ISBN: 978-1-5386-4969-5.

[Zhe+15] Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John C. S. Lui. “Enpublic Apps: Security Threats Using iOS Enterprise and Developer Certificates.” In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2015, pp. 463–474. ISBN: 978-1-4503-3245-3.

[Zho+13] Wu Zhou, Yajin Zhou, Michael C. Grace, Xuxian Jiang, and Shihong Zou. “Fast, scalable detection of "Piggybacked" mobile applications.” In: *Conference on Data and Application Security and Privacy – CODASPY’13*. ACM, 2013, pp. 185–196. ISBN: 978-1-4503-1890-7.

[ZZL15] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. “Character-level Convolutional Networks for Text Classification.” In: *Neural Information Processing Systems – NIPS’15*. 2015, pp. 649–657.

[ZZT19] Xian Zhan, Tao Zhang, and Yutian Tang. “A Comparative Study of Android Repackaged Apps Detection Techniques.” In: *Software Analysis, Evolution, and Reengineering – SANER’19*. IEEE, 2019, pp. 321–331. ISBN: 978-1-7281-0591-8.