Dibris

Dipartimento di Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi (DIBRIS)

Novel Attacks and Defenses in the Userland of Android

by

Simone Aonzo

Ph.D. Thesis

DIBRIS, Università di Genova

Via Opera Pia, 13 16145 Genova, Italy http://www.dibris.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi (DIBRIS)

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Novel Attacks and Defenses in the Userland of Android

by

Simone Aonzo

December, 2019

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Università di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
http://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. INF/01)

Submitted by Simone Aonzo – DIBRIS, University of Genoa, Italy
Date of submission: December, 2019

Title: Novel Attacks and Defenses in the Userland of Android

Advisor: Alessio Merlo – DIBRIS, University of Genoa, Italy

External Reviewers:
Davide Balzarotti – Eurecom Graduate School and Research Center, France
Camil Demetrescu – Sapienza University of Rome, Italy

Abstract

In the last decade, mobile devices have spread rapidly, becoming more and more part of our everyday lives; this is due to their feature-richness, mobility, and affordable price. At the time of writing, Android is the leader of the market among operating systems, with a share of 76% and two and a half billion active Android devices around the world (Cut19). Given that such small devices contain a massive amount of our private and sensitive information, the economic interests in the mobile ecosystem skyrocketed. For this reason, not only legitimate apps running on mobile environments have increased dramatically, but also malicious apps have also been on a steady rise. On the one hand, developers of mobile operating systems learned from security mistakes of the past, and they made significant strides in blocking those threats right from the start. On the other hand, these high-security levels did not deter attackers. In this thesis, I present my research contribution about the most meaningful attack and defense scenarios in the userland of the modern Android operating system. I have emphasized "userland" because attack and defense solutions presented in this thesis are executing in the userspace of the operating system, due to the fact that Android is slightly different from traditional operating systems.

After the necessary technical background, I show my solution, RmPerm, in order to enable Android users to better protect their privacy by selectively removing permissions from any app on any Android version. This operation does not require any modification to the underlying operating system because we repack the original application. Then, using again repackaging, I have developed Obfuscapk; it is a black-box obfuscation tool that can work with every Android app and offers a free solution with advanced state of the art obfuscation techniques – especially the ones used by malware authors. Subsequently, I present a machine learning-based technique that focuses on the identification of malware in resource-constrained devices such as Android smartphones. This technique has a very low resource footprint and does not rely on resources outside the protected device. Afterward, I show how it is possible to mount a phishing attack – the historically preferred attack vector – by exploiting two recent Android features, initially introduced in the name of convenience. Although a technical solution to this problem certainly exists, it is not solvable from a single entity, and there is the need for a push from the entire community. But sometimes, even though there exists a solution to a well-known vulnerability, developers do not take proper

precautions. In the end, I discuss the Frame Confusion vulnerability; it is often present in hybrid apps, and it was discovered some years ago, but I show how it is still widespread. I proposed a methodology, implemented in the FCDroid tool, for systematically detecting the Frame Confusion vulnerability in hybrid Android apps. The results of an extensive analysis carried out through FCDroid on a set of the most downloaded apps from the Google Play Store prove that 6.63% (i.e., 1637/24675) of hybrid apps are potentially vulnerable to Frame Confusion. The impact of such results on the Android users' community is estimated in 250.000.000 installations of vulnerable apps.

Contents

Chapter 1

Introduction

The role of mobile devices in our lives has been exponentially increasing in the last decade. The advent of the Android operating system has fostered the development of an open ecosystem of developers eager to grasp the opportunities offered by the widespread adoption of smartphones. Nowadays the landscape of mobile devices is mostly divided between Android and iOS, with a market share of 76% and 22% respectively in the July of 2019 (Sta19b). The reason behind this success is due to the fact that Android is general purpose and can be freely adopted and customized by device manufacturers. As expected, such pervasive spread of Android-based smartphones had a significant impact on the number of applications (hereafter, *apps*) developed for Android. At the time of writing, the set of available apps on the Google Play Store has reached 2.7M (sta19a), thereby supporting almost every activity of both personal and professional users. Unleashing the smartphone application market was beneficial for end-users as they can now choose among a large variety of apps, but while most of these applications have legal and fair behaviors, some of them hit the media for being quite useful in affecting the user's security and privacy (SFK$^+$10). As a consequence, the spread of malicious apps rose, and it is still exponentially increasing.

In this thesis, I present my research contribution about the most meaningful attack and defense scenarios in the *userland* of the modern Android operating system. It is essential to point out that attackers or defenders presented in this thesis are executing in the userland (userspace), not in kerneland (kernel space), because it emphasizes their intrinsic limited capabilities. The userland is the memory area where application software runs, while the kerneland is strictly reserved for running a privileged operating system kernel. Moreover, the userland of Android is slightly different from traditional operating systems. The Android platform takes advantage of the Linux user-based protection to identify and isolate app resources from each other and protects apps and the system from malicious apps. To do this, Android assigns a unique user ID (UID) to each Android application and runs it in its own process. In general, Android implements the principle of least privilege so that

each app, by default, has access only to the components that it requires and nothing more. This creates a very secure environment in which an app cannot access parts of the system for which it does not hold the corresponding permission, while there are few controlled ways for an app to share data with other apps and to access system services. A central design point of the Android security architecture is that no app, by default, has permission to perform any operations. An app must explicitly publicize the permissions it requires, for example, reading or writing the user's private data (such as contacts or emails), reading or writing another app's files, performing network access, keeping the device awake, and so on. However, Chapter 2 discusses in detail all the necessary background about the topics presented in the rest of the thesis.

The rest of the thesis is structured as follows: in Chapter 3, we[1] propose a defensive approach that allows users to selectively remove permissions from apps before installing them, without any change to the underlying OS, i.e., fully in userland. Prior to Android 6, users can install an app only by accepting *all* its requested permissions, while newer Android versions allow users to grant/deny groups of permissions dynamically. Since some of them impact users' privacy, we argue that users should be granted control at the granularity of the single permission, not just an entire group. Besides, some apps do not work until the user grants a specific permission: sometimes it is a legitimate request (e.g., it is reasonable that a photo editing app needs to access your storage), while in other cases it is an attempt to violate the privacy of the user (e.g., when a single-player video-game requests the access to your contact list). We developed RmPerm, an open-source tool, that implements our methodology, and we present the viability of our approach via an empirical assessment carried out on 81K apps, showing that, in the worst case, up to 86% of the apps can execute without crashing when *none* of the requested privacy-related permissions are granted.

Then, given the technical background introduced in the previous chapter about application rewriting, Chapter 4 moves on a very current topic for the Android app ecosystem: obfuscation. We present Obfuscapk, an open-source automatic obfuscation tool for Android apps that works in a black-box fashion (i.e., it does not need the app source code). Obfuscapk is the result of an in-depth study about state-of-the-art obfuscation techniques. It is impossible to define if obfuscation is an attack or defense technique because it is also considered as a double-edged sword by the security community because both software developers and malware authors frequently use obfuscation. Thereafter, we explore the evolution of Android obfuscation using a real-world implementation, then we discuss an actual use-case for Obfuscapk, and an empirical assessment on the reliability of the tool on a set of 1000 "most downloaded" APKs from the Google Play Store. Obfuscapk aims at becoming a useful tool for the research community on mobile security with its modular architecture that could be straightforwardly extended to support new techniques. Using

[1]Since my findings are the result of a joint work with other people, I use the first person plural in the rest of this thesis.

it as a black-box obfuscation tool allows users to obfuscate apps and malware samples for several aims, like building or attacking a machine learning model, improving program analysis techniques w.r.t. obfuscation transformations, just to cite a few.

Deceiving automated code analysis, especially malware detection analysis, is one of the aims of obfuscation. As previously explained, the current Android architecture limits the efficacy and the applicability of anti-malware techniques because both attacker and defender face each other in userland. In fact, Android antivirus apps run in userland like normal apps. For this reason, given a target app, existing antivirus solutions can just perform static analysis (signature checks mainly) on the device, and upload the app to their remote server for an extensive privileged analysis (usually using an emulator with the root permission) at a later stage (RCJ13). This approach requires an internet connection, and it often does not react quickly enough. In Chapter 5, we propose BadDroids, a mobile application leveraging machine learning techniques for detecting malware on mobile –resource constrained– devices. BadDroids runs in background and transparently analyzes the applications as soon as they are installed, i.e., before infecting the device. BadDroids relies on static analysis techniques and features provided by the Android OS to build up sound and complete models of Android apps in terms of permissions and API invocations. It uses ad-hoc supervised classification techniques to allow resource-efficient malware detection. Resource-constrained systems are becoming more and more common as users migrate from PCs to mobile devices and as IoT systems enter the mainstream. At the same time, it is not acceptable to reduce the level of security; hence, it is necessary to accommodate the required security into the system-imposed resource constraints. By exploiting the intrinsic nature of data, it has been possible to implement a state-of-the-art data-driven model that provides deep insights on the detection problem and can be efficiently executed directly on the device itself as it requires a minimal computational effort. Besides its limited resource footprint, BadDroids is exceptionally effective: an extensive experimental evaluation shows that it outperforms the currently available solutions in terms of accuracy, which is around 99%.

In Chapter 6, we switch from defense to offense, and we show how two of modern Android features introduced in the name of convenience (mobile password managers and Instant Apps), can be abused to make phishing attacks that are significantly more practical than existing ones. We have studied the leading password managers for mobile, and we uncovered several design issues that leave them open to attacks. For example, we show it is possible to trick password managers into auto-suggesting credentials associated with arbitrary attacker-chosen websites. We then show how an attacker can abuse the recently introduced Instant Apps technology to allow a remote attacker to gain full UI control and, by abusing password managers, to implement an end-to-end phishing attack requiring only few user's clicks. We also found that mobile password managers are vulnerable to "hidden fields" attacks, which makes these attacks even more practical and problematic. We conclude the chapter switching back to the defensive side, by proposing a new secure-by-design

API that avoids common errors, and we show that the secure implementation of autofill functionality will require a community-wide effort.

In the end, Chapter 7 deals with Frame Confusion, which is a vulnerability affecting hybrid applications (web applications in the native browser), which allow circumventing the isolation granted by the Same-Origin Policy. The detection of such vulnerability is still carried out manually by application developers, but the process is error-prone and often underestimated. In this chapter, we propose a sound and complete methodology to detect the Frame Confusion on Android as well as FCDroid, a publicly-released tool that implements such methodology and allows to detect the Frame Confusion in hybrid applications, automatically. We also discuss an empirical assessment carried out on a set of 50K applications using FCDroid, which revealed that a lot of hybrid applications suffer from Frame Confusion. Finally, we show how to exploit Frame Confusion on a news application to steal the user's credentials.

Publications.

The research presented in this dissertation produced peer-reviewed papers accepted to conferences and journals. In what follows, I report the complete list of published works:

S. Aonzo, G. Lagorio, A. Merlo. "*RmPerm: a Tool for Android Permissions Removal*", in Proc. of the 14th International Conference on Security and Cryptography (SECRYPT 2017), Madrid, Spain. (ALM17)

S. Aonzo, A. Merlo, M. Migliardi, L. Oneto, F. Palmieri. "*Low-Resource Footprint, Data-driven Malware Detection on Android*", IEEE Trans. on Sustainable Computing, Vol. PP, no. 99, pp. 1-1, DOI: 10.1109/TSUSC.2017.2774184. (AMM+17)

S. Aonzo, A. Merlo, G. Tavella, Y. Fratantonio. "*Phishing Attacks on Modern Android*", in Proc. of the 25th ACM Conference on Computer and Communications Security (CCS 2018), Toronto, Canada. (AMTF18)

D. Caputo, L. Verderame, S. Aonzo, A. Merlo. "*Droids in Disarray: Detecting Frame Confusion in Hybrid Android Apps*", in IFIP Annual Conference on Data and Applications Security and Privacy (DBSec 2019) (pp. 121-139). Springer, Cham. (CVAM19)

While the paper:

S. Aonzo, G.C. Georgiu, L. Verderame, A. Merlo. "*Obfuscapk: an open-source black-box obfuscation tool for Android apps*"

is currently under review at SoftwareX journal.

Chapter 2

Technical Background

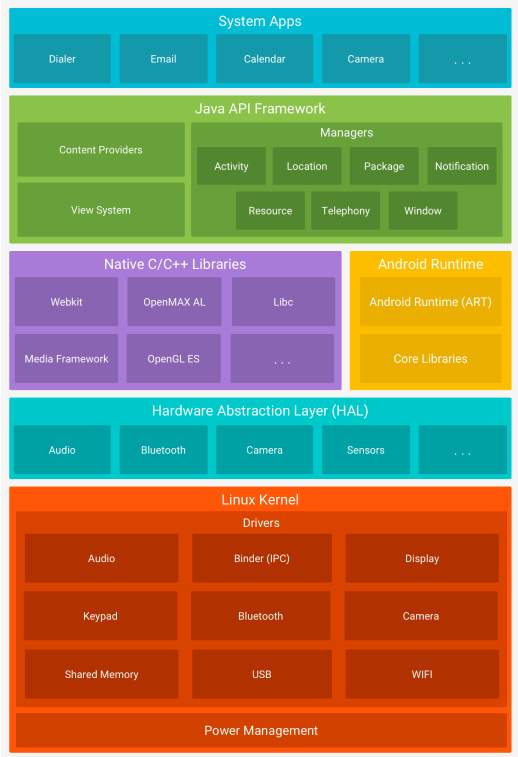


Figure 2.1: The Android software stack.

Android is an open-source, Linux-based software stack (as shown in Figure 2.1) created for a wide array of devices and form factors. Even though Android is built on top of the Linux kernel, it is slightly different from a regular kernel of a desktop machine or a

non-Android embedded device. The differences are due to a set of new features called "Androidisms" (Ele14). Some of the main Androidisms are the Low Memory Killer Daemon (lmkd), wake-locks (integrated as part of wakeup sources support in the mainline Linux kernel), anonymous shared memory (ashmem), Paranoid Networking, and Binder. For example, Binder implements IPC and an associated security mechanism, and Paranoid Networking restricts access to network sockets to applications that hold specific permissions. Android is a modern and sophisticated operating system; the purpose of this chapter is to give the necessary background to the reader about the topics that will be discussed in the rest of the thesis. However, this chapter just introduces the general technical concepts, while the following chapters have their dedicated section to its specific ones.

2.1 Application PacKages (APK)

Android applications are distributed and installed in the form of application package (APK) files. The APK format is an extension of the Java JAR format, which in turn, is an extension of the popular ZIP file format.

Listing 2.1 shows the contents of a typical APK file after it has been extracted.

Listing 2.1: Contents of a typical APK file

```
1  apk/
2  |-- AndroidManifest.xml
3  |-- classes.dex
4  |-- resources.arsc
5  |-- assets/
6  |-- lib/
7  |   |-- armeabi/
8  |   |    \-- libapp.so
9  |   \-- armeabi-v7a/
10 |        \-- libapp.so
11 |-- META-INF/
12 |   |-- CERT.RSA
13 |   |-- CERT.SF
14 |   \-- MANIFEST.MF
15 \-- res/
16    |-- anim/
17    |-- color/
18    |-- drawable/
19    |-- layout/
20    |-- menu/
21    |-- raw/
22    \-- xml/
```

Basically, APK files are archive files that include both application code and resources, as well as the application manifest file `AndroidManifest.xml` (line 2). The manifest file declares the application's package name, version, components, permissions, and other metadata. The `classes.dex` (line 3) file contains the executable code of the application in the DEX bytecode format, organized in classes and methods. The `resources.arsc` (line 4) packages all of the application's compiled resources such as strings and styles. The `assets` (line 5) directory is used to bundle raw asset files with the application, such as fonts or music files. Applications that take advantage of native libraries via Java Native Interface (JNI) contain a `lib` (line 6) directory, with subdirectories for each supported platform architecture (lines 7-10). Resources that are directly referenced from Android code, either directly using the `android.content.res.Resources` class or indirectly via higher-level APIs, are stored in the `res` (lines 15) directory, with separate directories for each resource type (animations, images, menu definitions, etc.). Like JAR files, APK files also contain a `META-INF` directory (line 11). The APK must contain precisely the entries listed in `MANIFEST.MF` (line 14) and where all entries must be signed by the same set of signers. Then the `CERT.SF` (line 13) contains the list of all files along with their SHA-1 digest and the `CERT.RSA` contains the signed contents of the CERT.SF file along with the certificate chain of the public key used for signing the contents. The protection chain is thus CERT.RSA → CERT.SF → MANIFEST.MF → contents of each integrity-protected JAR entry.

2.2 APK Signing

Signing an APK allows verifying its integrity and authenticity. Before executing any third-party program, the user wants to be sure that it has not been tampered with (integrity) and that it was created by the entity that it claims to come from (authenticity). These features are usually implemented by a digital signature scheme, which guarantees that only the entity owning the signing key can produce a valid code signature. The signature verification process verifies both that the code has not been tampered with and that the signature was produced with the expected key. Nevertheless, one problem that code signing does not solve directly is whether the code signer (software publisher) can be trusted. The usual way to establish trust is to require that the code signer holds a digital certificate and attaches it to the signed code. The verifiers decide whether to trust the certificate based on a trust model (such as PKI or web of trust) or on a case-by-case basis. Because Android code signing is based on Java JAR signing, it uses public-key cryptography and X.509 certificates like many code signing schemes, and usually, code signing certificates must be issued by a CA that the platform trusts. While there are many CAs that issue code signing certificates, it can be quite challenging to obtain a certificate that is trusted by all targeted devices. Android solves this problem quite simply: it does not care about

the contents or signer of the signing certificate. Thus there is no need to have it issued by a CA, and virtually all code signing certificates used in Android are self-signed. Additionally, the developer does not need to assert her identity in any way: she can use pretty much anything as the subject name of the certificate. The Google Play Store does have a few checks to weed out some common names, but not the Android OS itself. This lack implies that on Android the authenticity of the developer is not ensured!

2.3 Installation and Repackaging

There are several ways to install Android applications. Usually, end-users use an application store client, such as the *Google Play Store*, *Amazon Appstore*, *Aptoide*, *Tencent My App*, just to cite a few. However, it is also possible to do it directly on the device by opening a downloaded APK file (this method is commonly referred to as "sideloading" an app), or from a connected computer using the Android Debug Bridge (`adb`) command-line tool. Since the Google Play Store is the official Android store, there is a security restriction that blocks installing applications outside the Google Play Store. For these reasons, in order to install apps hosted outside the Play Store or sideloaded, users need to enable a security option called *Unknown sources* manually (off by default).

The package name of an Android app, defined in the `AndroidManifest.xml`, is a developer-specified string that acts as the primary app identifier, thus uniquely identifying an app on the device. While it is commonly believed that package names are analogous to web domain names for mobile apps, they are very different for what concerns security guarantees. The only constraint is that the package name needs to be unique i) across the apps published on the Play Store (on some markets as well) and ii) across the apps installed on a given device. No other security guarantees are provided. It is worth to emphasize that, even if some Android markets use the package name as a unique identifier, there are no guarantees that two apps with the same package name on two different markets are the same. The phishing attack presented in Chapter 6 abuse the fact that the package name is also attacker-controlled.

During the installation phase, if the package manager (the system component in charge of the APK management) detects that there is another app with the same package name installed on the device, it manages this operation like an update. However, the updates of applications are only allowed when the updated APK is signed with the same key of the previous one; therefore, the system "trusts" an APK on its first install. From that moment, another installation of an APK with the same package name is considered an update.

Combining all the above-described peculiarities of the Android operating system leads to an open problem: the APK repackaging (ZWZJ12; ZZG$^+$13). APK are zip files, and their files can be extracted, modified and repackaged – breaking their signature. However, the

repackaged APK can be re-signed with a new certificate. Despite the fact that certificates are self-signed, everybody can be a developer on Android markets and being able to sign apps for publications. Therefore, it is easy for malware authors to modify legitimate apps injecting their malicious code (if needed, changing the package name), repackaging, and redistribute them. Moreover, modifying the embedded advertising client ID or replacing it with new advertising libraries, an attacker can make profits through apps developed by others (GSC+13). Nowadays repackaging is one of the primary attack vectors to distribute malware through markets and siphon advertising revenue from original developers. Consequently, there is a lot of research effort to detect repackaged Android applications (ZZJN12; SLQ+14; ZHZ+14). Given that, repackaged apps are often protected by obfuscation or hardening systems.

In Chapter 3, we exploit app repackaging in order to create an unprivileged variant of the same application, while in Chapter 4 we have developed a framework to obfuscate Android apps (repacking the original one) in order to study the effects of such code transformations.

2.4 The permission-API mapping

The main security mechanisms in Android are the *app sandbox* and the usage of *permissions*. Android executes each app in a sandboxed environment, built by taking advantage of the multi-user nature of the Linux Kernel. In a nutshell, upon installation, each app gets assigned a Linux User ID (UID), and whenever the app executes, it runs in its own userspace. The aim of sandboxing is to improve the separation among apps by leveraging the isolation granted natively by the Linux Kernel to different system users. Beyond sandboxing, Android requires that each app declares, in the `AndroidManifest.xml`, its resource requirements as a set of *permissions* upon installation. Without loss of precision, permissions can be seen as strings that denote the possibility for an app to require specific functionalities offered by the operating system. For example, the `RECEIVE_SMS` perission allows an application to receive SMS messages. Some basic permissions are automatically granted at install time and never revoked. Other permissions, defined as *dangerous* (DAN), handle the privacy of the user (i.e., they allow the app to profile the user by accessing his contacts, messages and call logs, as well as activate camera and audio recording or access the user's position) and they have to be granted/denied at runtime by the user himself. Such permissions are divided into 11 groups (Goo19) that represent the granularity at which they may be granted/denied: `ACTIVITY_RECOGNITION`, `CALENDAR`, `CALL_LOG`, `CAMERA`, `CONTACTS`, `LOCATION`, `MICROPHONE`, `PHONE`, `SENSORS`, `SMS`, `STORAGE`. When a dangerous permission is requested for the first time, the user is prompted to grant/deny the corresponding permission group to the app, automatically granting all the dangerous permissions in the corresponding group. This choice allows limiting the number of permission requests at runtime (that can be annoying for the user), at the cost of a more coarse-grained

control over dangerous permissions. It is also worth noticing that once a group of dangerous permissions is granted, it is never revoked by the system; however, the user can remove a permission from an app by explicitly modifying the app settings. We argue that Android should also offer an advanced management of permission, allowing fine-grained control. In Chapter 3, we propose a userland solution to this problem that works in userland.

An app can obtain a system functionality by invoking a specific Android API (AAPI from now on) from the Java API Framework (Figure 2.1). The set of AAPI invocations can be mostly inferred from the app code (i.e., the executable app code in DEX format) through static analysis techniques. An AAPI invocation may require specific permissions, so an AAPI is adequately executed if the app has been granted the corresponding dangerous permissions. Google does not provide an official mapping between AAPI methods and permissions, but some works have empirically inferred this mapping (AZHL12; BBD$^+$16). An example of the correspondence between AAPIs and permissions is the following: the class `android.net.wifi.WifiManager` provides the AAPI method `isWifiEnabled()` that returns true or false whether the Wi-Fi interface is enabled or disabled and needs the permission `android.permission.ACCESS_WIFI_STATE` to check the state of the Wi-Fi.

Since a mapping (albeit potentially incomplete) exists (BBD$^+$16), checking both the requested permissions and the AAPI invocations could appear redundant, as considering the AAPI invocation inferred through static analysis could suffice. Unfortunately, the inference of the AAPI set by static analysis alone may be incomplete, in fact, an app could also leverage advanced Java mechanisms such as Reflection (REF) and Java Native Interface (JNI) to execute code that is not directly identifiable in the bytecode (i.e., it is impossible to infer all the specific invocations through static analysis alone). However, an AAPI invocation, especially a dangerous one, can still require some permissions to execute correctly when invoked at runtime.

Chapter 3

Defend the Privacy by Removing Permissions

3.1 Introduction

Apps are the main attack vector for Android devices; therefore, they should require the minimum set of permissions to work properly, while satisfying the least privilege principle to reduce the attack surface. However, apps are generally over-privileged (FCH+11) since developers tend to require more permissions than necessary to reduce the probability that their app crashes. Furthermore, it is worth noticing that some permissions are particularly important for the privacy of the user, like, for instance, those that allow apps to profile the user by accessing her contacts, messages and call logs. These permissions are called *dangerous* by the Android documentation, and we argue that users should be granted more control over them.
Android versions prior to 6, that have still an adoption of 25% (and19), grant a very coarse-grained control over permissions, i.e., the user cannot remove permissions from apps, and should grant *all* permissions requested by any app in order to install it. Newer versions (i.e., 6 and later) support dynamic management of groups of permissions, but they do not allow the user to grant/deny single permissions. To overcome these limitations, we put forward an approach allowing users to selectively remove permissions from apps that does not require any modification to the underlying operating system and is compatible with all Android versions. A key strength of our approach is that, when a user decides to remove certain permissions from an app, we can guarantee, by design, that no Java nor any native code could ever exploit such permissions, no matter what. The worst case scenario is a crash of the less-privileged app, but not a privacy leak. We have implemented our methodology in a tool, `RmPerm` (rmp17), that we use to extensively assess the viability of the approach on a set of $81,000$ apps.

RmPerm is an *open-source* project implemented in Java, and consists of a console application and a library. In this respect, we also implemented an Android app, `ApkMuzzle` (apk17), using RmPerm as external library. We argue that releasing a tool like RmPerm as open-source is a liability, as any tool that repackages apps can subtly add malicious code. By releasing RmPerm as open source we grant anyone the possibility to verify its behavior, by inspecting the source code.

The rest of the chapter is organized as follows. Section 3.2 describes our methodology for permission removal, while Section 3.3 assesses its viability and performance. Section 3.4 sets our proposal within the current state-of-the-art. Finally, Section 3.5 concludes and points out some future work.

3.2 A Methodology for Permission Removal

A quite direct way to remove a set of permissions $P = \{p_1, \ldots, p_n\}$ from an app A is simply to modify its manifest, contained in the APK of A, obtaining A'. This action has two consequences:

1. the digital signature s, for the APK of A, cannot be reused for A' since its manifest, and so the resulting APK, differs from the original;
2. A' may crash due to unexpected exceptions, thrown by the invocation of some API method that needs some permission p_i to run. Indeed, prior to Android 6, an app asking for the set of permissions P gets installed only if the user grants the whole set P, so apps could assume to be granted all asked permissions.

Since the signature s has been produced using an unknown secret key, we have no choice but to sign A' with another (secret) key. The only user visible effect is that Android will consider A and A' two different apps; however, since they have the same name, only one of them can be installed at any time. We do not consider this a problem; in some sense they *are different*: A' is probably safer than A! To avoid that A' crashes because of an unexpected exception, due to a missing permission p, we carry out a customization to all invocations of API methods that need the permission p. This, in turn, means that we need a mapping between permissions and the API methods that require such permissions. Surprisingly, this mapping is not provided by the official documentation. However, we were able to obtain the mapping from the Androguard Project (and17), which is based on *PScout* (AZHL12).

Listing 3.1: Example of method redirection

```
@CustomMethodClass
public class CustomMethods {
  @MethodPermission(
   permission="android.permission.INTERNET",
   defClass="java.net.URL")
  public static InputStream openStream(URL u)
    { return new FakeInputStream(); }

  @AuxiliaryClass
  public static class FakeInputStream
              extends InputStream {
      @Override
      public int read() throws IOException
        { return 0; }
  }
}
```

Using *Dexlib2* (dex17) we have implemented RmPerm, a tool to *redirect* selected API method invocations to our own alternative implementations.[1] Custom methods typically just return some fake data to the app, to let it proceed. Consider, for instance, the method `execute`, declared in `org.apache.http.impl.client.DefaultHttpClient`: it needs the `INTERNET` permission and returns a `org.apache.http.HttpResponse`; in this case, we cannot just remove the invocation or return a `null` reference, because that would likely make the app crash. In such cases we must mock a "reasonable" return value. In the general case, when we want to redirect the invocation for an instance method m of class C, with the signature $T_r\ m(T_1, \ldots, T_n)$, we define a new static method $T_r\ m(C, T_1, \ldots, T_n)$, defined in a class X that we add to the APK. The first parameter, of type C, plays the role of the *receiver* of the corresponding instance method. This kind of setup is similar to C# extension methods. Then, we rewrite all the invocations of the form $e_0.m(e_1, \ldots, e_n)$, where e_0 has static type C, with $X.m(e_0, e_1, \ldots, e_n)$[2].

To make writing custom replacement methods as easy as possible, our tool reads a DEX file and searches for classes and methods that are annotated with Java custom annotations, that we have defined: `@CustomMethodClass`, `@MethodPermission` and `@AuxiliaryClass`. The annotation `@CustomMethodClass` simply marks classes that contain replacement methods; this is simply an optimization to avoid to process each and every method of the input file. The annotation `@MethodPermission(p, defClass)` indicates the involved permission

[1] As an optimization, RmPerm can automatically remove the invocations of `void` methods, when such methods do not have an explicit custom replacement.

[2] Here, to make the explanation simpler, we use a source-like syntax but, of course, we work directly at DEX bytecode level.

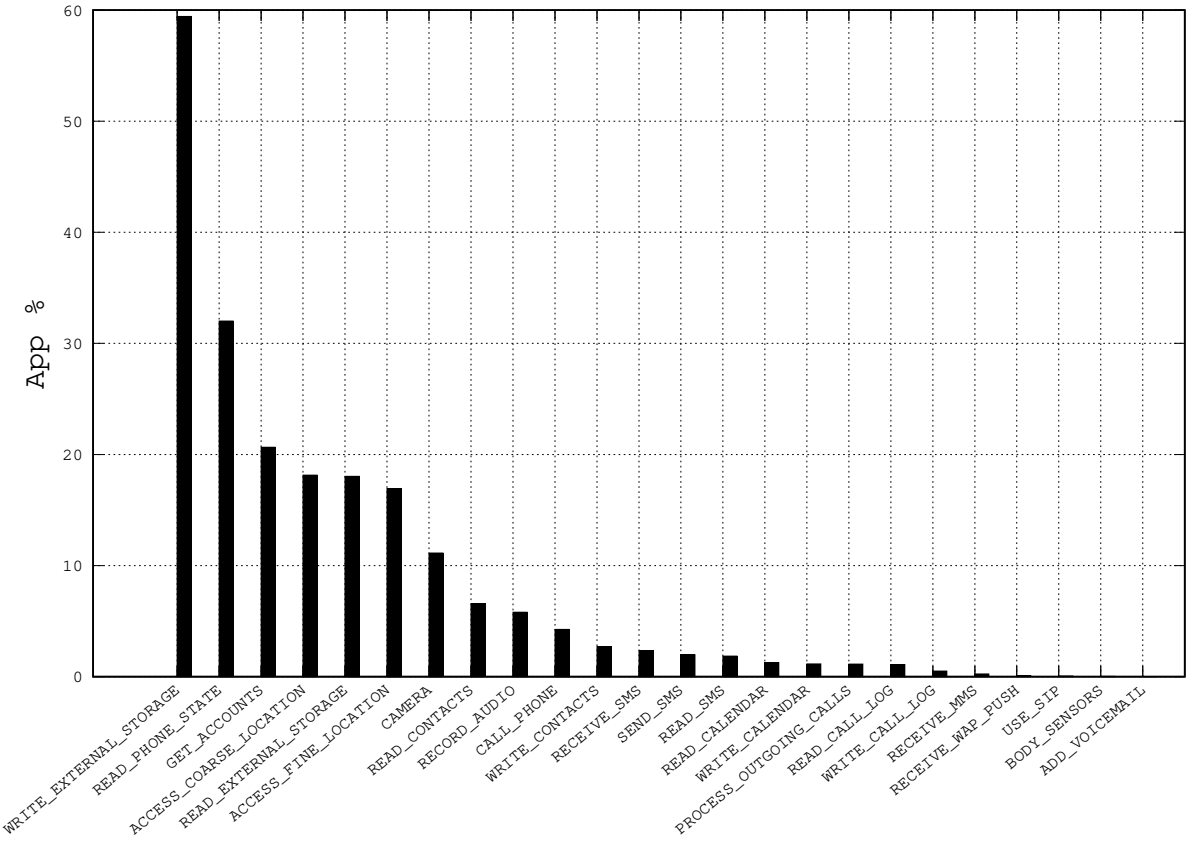


Figure 3.1: Distribution of dangerous permissions in the dataset.

`p` and the defining class (in the Android API) `defClass`. Finally, `@AuxiliaryClass` marks the classes that must be copied into the repackaged app, because they are needed by the custom methods. For instance, Listing 3.1 shows class `CustomMethods`, which contains a redirection for method `openStream` defined in class `java.net.URL`.

3.3 Experimental Assessment

To assess the effectiveness and efficiency of the proposed methodology, we have carried out an empirical assessment by executing RmPerm on a dataset of 81,000 APKs randomly downloaded from three different markets, namely Google Play (70,000 APKs), Aptoide (5,500 APKs) and Uptodown (5,500 APKs). Since we could not conceivably choose, for each app of the dataset, a different set of permissions to remove, we have decided to assess the worst-case scenario, that is, to remove *all dangerous permissions* from each app A,

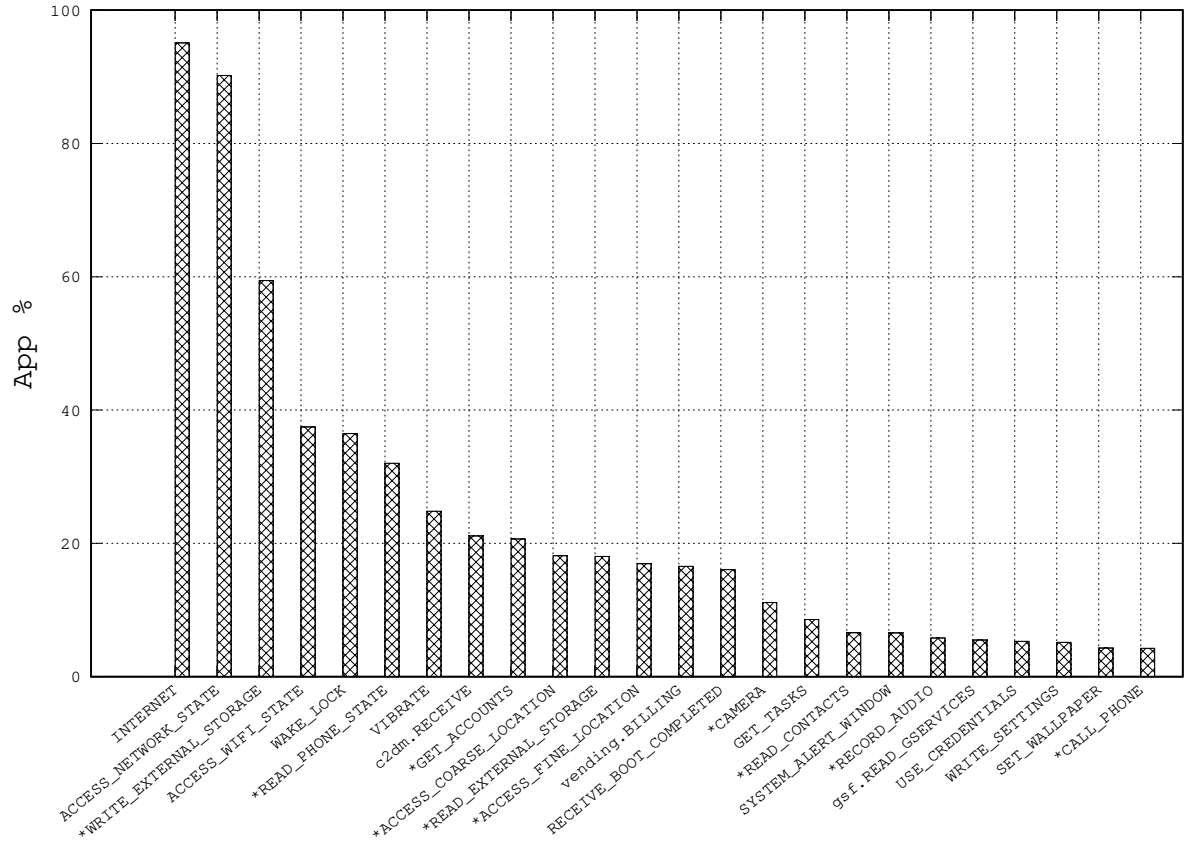


Figure 3.2: Top 24 requested permissions in the dataset. Dangerous permissions are labeled with the asterisk (*) character.

producing a new app A'; then, we have checked whether the less-privileged app A' could be installed and execute properly. This process is detailed below.

Dangerous permissions refer to the Android classification (req17); there currently are 24 dangerous permissions. They are strictly related to the user's privacy as they allow app to access storage, camera, GPS coordinates, user's calendar and contacts, just to cite a few. We extracted all permissions requested by the apps in our dataset by systematically parsing the app manifest file, that contains all permissions required by an app. Fig. 3.1 shows the distribution of dangerous permissions in the dataset. Intuitively, the x-axis shows the 24 permissions ordered accordingly to their frequency on the dataset. The y-axis indicates the percentage of apps requesting the permission. Fig. 3.2 plots the top 24 permissions requested by apps in the same way.

Table 3.1: RmPerm: Performance and size statistics.

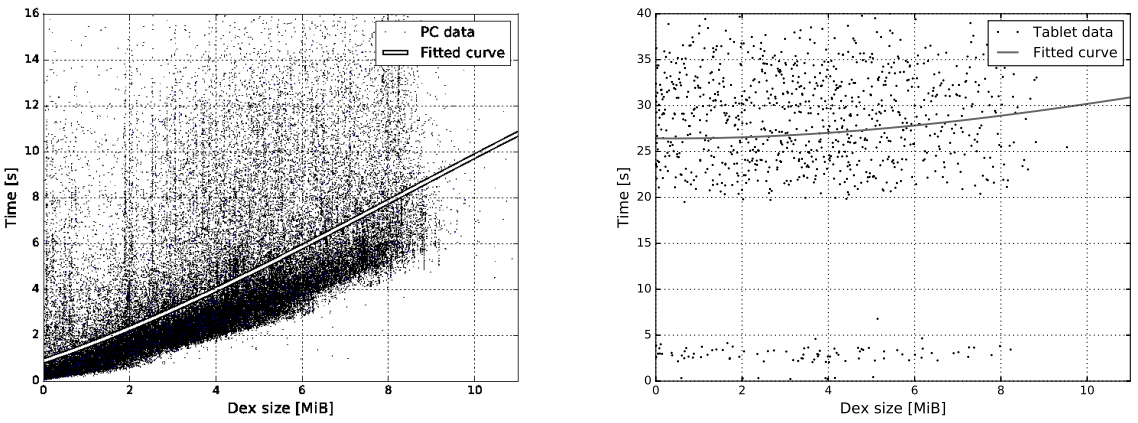
	#Permissions			SizeRatio		Exec. time[s]		Test results	
AppName	**original**	**new**	**Δ**	**APK**	**DEX**	**PC**	**Tablet**	**Inst.**	**Monkey**
AdobeReader	6	3	3	1.16	1.00	5.70	27.41	✓	✓
CandyCrushSaga	10	8	2	1.08	0.99	6.17	30.83	✓	✓
Facebook	48	33	15	1.03	0.94	4.04	27.73	✓	✓
Instagram	25	17	8	1.19	1.01	7.13	30.15	✓	✓
LedFlashLight	9	8	1	1.06	1.00	6.60	27.47	✓	✓
MGuard	35	33	2	0.97	1.00	8.46	32.57	X	n. a.
Shazam	20	13	7	1.04	1.00	7.25	35.49	✓	X
Skype	45	33	12	1.13	1.00	11.49	44.82	✓	✓
Snapchat	15	7	8	1.07	1.00	5.14	25.17	✓	✓
Spotify	23	19	4	1.11	1.00	8.49	35.50	✓	X
SwiftKey	14	11	3	1.18	1.00	9.62	31.90	✓	X
Telegram	32	21	11	1.12	1.00	5.94	23.94	✓	X
Twitter	31	22	9	1.45	1.00	8.76	28.19	✓	✓
Viber	56	41	15	1.21	1.00	7.65	29.35	✓	✓
Whatsapp	50	38	12	1.45	1.00	7.38	26.95	✓	X
...	...	...	...	...	...	...	...	...	...
Average	6.8	4.9	2.1	1.1	1.1	3.9	27.1		
Std deviation	5.9	4.4	2.4	0.2	3.1	3.3	9.8		

3.3.1 Testing RmPerm.

A single automated test, for each app A, runs as follow:

1. RmPerm gets the APK of app A, and builds up a new app A' that does not require any dangerous permission.
2. A' is installed on an actual Android device.
3. If this step fails, the original A is installed, in order to verify whether the failure is due to the modification carried out by RmPerm or it is independent from the permission removal.
4. If the installation of A' has been successful, its behavior is tested by generating a stream of 512 pseudo-random user events with *Monkey* (mon17), seeded by a random number n. Using different seed values leads to generate distinct sequences of user events. If A' fails, this can be due either to the removal of permissions or to the presence of bugs in the original app A.

5. To ascertain this, we stimulate A with the same stream of events, generated by seeding *Monkey* with the same seed n.



(a) PC (b) Tablet

Figure 3.3: Time required to repackage an app

Through previous steps, we can empirically assess whether the removal of dangerous permissions through RmPerm leads to failures. We carried out the experimental assessment on a Dell XPS 9530 (Ubuntu 16.04, Intel i7-4712HQ @ 2.30GHz, 16GB RAM), as well as on two Asus Z170CG (Android 5.0.2, Intel Atom x3-C3200 @ 900MHz, 1GB RAM) that we used to install and (automatically) stimulate the apps.

Our results indicate that on $81,000$ samples, $2,358$ repackaged apps failed in step 2; that is, they could not be installed successfully. Among these, 572 failed the step 3 too; this means that the corresponding original APKs were already broken in some way. Therefore, we discarded them and we considered a new set, consisting of the original samples except for the already broken APKs; that is, $81,000 - 572 = 80,428$ apps. On this set, the $98\% = 78,642/80,428$ of repackaged apps have been successfully installed. Then, we stimulated the installed apps according to step 4. Among these, $66,051$ were repackaged and stimulated without crashes, while $14,377$ $(= 80,428 - 66,051)$ failed and required further analysis. Therefore, we applied step 5 to such apps obtaining that $3,633$ original apps crashed, thereby proving that the same problems affected the original app, too. For this reason, we also discarded these APKs. Summing up, the $86\% = 66,051/76,795$ of working apps have been successfully modified, installed and executed properly after removing all dangerous permissions.

3.3.2 Discussion on global statistics.

We analyzed the average values for the whole dataset in terms of permission removal, size of APKs and DEX after repackaging, as well as the time required for the whole process. To this aim, Table 3.1 summarizes global and some per-app statistics; the database containing all the data can be freely downloaded (sql17). Regarding permissions, removing dangerous ones has led to the removal, on average, of 38% of all app permissions, as original apps have on average 6.8 permissions while repackaged ones have 4.9, with a large standard deviation value in both cases.
Repackaging phase does not alter the size of the APKs significantly; on average, repackaged APKs are smaller by a factor of 1.1, with a standard deviation of 0.2. Modified DEX files are smaller on average, with a factor of 1.1 and a large standard deviation of 3.1. This is due to the fact that, when removing a set of permissions P, we also remove all invocations to `void` API methods, requiring some permission $p \in P$, for which we do not have explicitly defined a redirection. Furthermore, the size of the original manifest file is decreased, but the size of DEXs is increased due to the addition of custom classes/methods.

3.3.3 RmPerm performance.

We have measured the time needed to remove all dangerous permissions from an app, in two different use cases: when RmPerm is running on a PC, and when RmPerm is running on an Android device. We have measured the running time on all APKs of our sample set when running on the PC, while we have randomly picked $1,000$ apps when RmPerm was run on the Android device. Indeed, we were only interested to check whether running RmPerm on an Android device was practical and if the running time was still linear in the size of the DEX file. Fig. 3.3 shows the performance results: the DEX size, say s, is generally a sensible parameter for predicting the running time t; indeed, as shown by the fitted curve, $t(s)$ is roughly a linear function. However, there are cases where a relatively small DEX file is contained in a large APK; for instance, this is the case for apps containing graphical/multimedia resources, like games. In these cases, the time to copy the resources from the original APK into the new one may prevail the time needed to process and rewrite the bytecode. This is the reason for the jitters in both graphs. Results have been positive in both regards: while obviously slower, running RmPerm on the device requires less than 30 seconds on average, and the times are still linear in the size of the DEX file, even though the slope is less steep and there is a constant cost, presumably due to the start-up time of RmPerm on Android.

3.4 Related Work

As shown by previous work (FGW11; FCH+11) , many Android apps are *over-privileged*, that is, they request more permissions than they actually need, thereby making the built-in permission system rather inadequate to protect the users and their privacy. To address these concerns, some authors have proposed to enrich the built-in security framework, by modifying the underlying operating system, and requiring changes to the app sources, in order to exploit the new features.

One of the top problems related to Android permissions is the fact that, up to Android 6, the protection offered by the system was an *all-or-nothing* choice at installation time, when the user was asked to accept *all* permissions requested by the app, or to abort the installation altogether. Moreover, with the current permissions management, if an app requests a dangerous permission, belonging to a certain permission group, and the user agrees on its usage, then the user is actually agreeing on accepting *all* permissions of the same group. That is, any subsequent update of the app can request, and be silently granted, any other permission belonging to an accepted group. Clearly, a fine-grained control was needed. For this reason, many proposals, including ours, tackle the built-in permission system directly.

Apex (NKZ10) is a policy enforcement framework that allows users to selectively grant permissions to apps, as well as impose constraints on the usage of resources. This implementation requires some changes to the Android code base so, while we share a similar goal, the striking difference is that we require no changes to the underlying system.

Some work addresses privacy concerns directly: *AppFence* (HHJ+11) retrofits the Android operating system to protect private data from being exfiltrated, by replacing *shadow data*, in place of data that the user wants to keep private, and by blocking network transmissions that contain data the user marked for on-device use only. *MockDroid* (BRSS11) modifies the Android operating system to allow users to *mock* the app's accesses to a resources . We use a similar *trick* to avoid that apps crash due to unexpected exceptions, once we have removed some of their permissions. However, we repackage apps and leave the operating system untouched. Finally, *TISSA* (ZZJF11) is a privacy-mode implementation in Android. All the above proposals allow running unmodified apps more safely, at the cost of modifying the underlying Android operating system, which severely hampers the widespread adoption of these solutions.

Other proposals (XSA12; JMV+12; DSKC12; BGH+13; DC13; RJV+11) bypass the need to modify the underlying operating system by repackaging arbitrary apps to attach user-level sandboxing and policy enforcement code. These proposals, like ours, use static analysis to identify the usage of API methods and instrument the bytecode to control the access to these invocations. However, with the exception of (JMV+12), discussed below, all of these do not remove permissions from the manifest of the original app A, when creating the repackaged app A'; thus, the underlying OS process that runs A' retains all permissions of the original app A. This means that incomplete/flawed implementations of bytecode

rewriting can lead to bypassing access control mechanisms, e.g., by using Java reflection and/or native code (HSD13).
Dr. Android (JMV+12) is a tool that uses bytecode rewriting to replace Android permissions with a specified set of fine-grained versions, that are accessed through a separate service, called *Mr. Hide*. In this case bytecode rewriting is adopted for replacing API calls, used by the original app, with interprocess communication primitives to query *Mr. Hide* service. These primitives are rather expensive, so there is a significant slowdown on API invocations. With our approach, instead, API invocations are "short-circuited" or removed altogether, making repackaged apps slightly *faster* than the original.
Finally, *Boxify* (BBH+15) has introduced a concept of app sandboxing on stock Android, based on app virtualization and process-based privilege separation. While this approach eliminates the need to repackage apps, it requires a lot of additional code (about 12 K lines of Java code, plus 3.5 K LoC of C/C++, according to the chapter), which should be carefully audited – authors promised to make the source code available, but at the time of writing, more than a year later, it is still unavailable. On the contrary, our approach simply requires to customize 57 trivial Java methods. Moreover, Boxify requires the presence of a fully privileged controller process, called *Broker*, which is an attractive target for privilege escalation attacks.

3.5 Concluding Remarks

We have presented a novel approach, and its supporting tool – RmPerm–, to enable Android users to better protect their privacy by selectively removing permissions from any app, on any Android version. Then we assessed the effectiveness of our idea on a set of 81,000 real-world samples. The experimental results have been encouraging; indeed, we have blindly removed, from these apps, *all* dangerous permissions obtaining that 86% of these rewritten apps can be installed and executed without crashing; we could not expect a 100% success-rate, some apps do need some of the permissions they request. However, the majority of them can be run equally fine with a strict subset of the permissions they originally requested. By using RmPerm, users can freely decide where to draw the "privacy line" and can run virtually any app without disclosing more personal information than they want to.
A limitation of our current implementation is that apps using anti-tampering techniques can detect that they have been rewritten. However, our experiments indicate that, for the time being, Android apps very seldom adopt anti-tampering techniques. Another limitation is that RmPerm currently redirects only "direct" API invocations, that is, we do not even try to redirect API invocations executed through the use of Java reflection or native code. We are considering how to extend our approach to intercept those reflective invocations too; however, this limitation is not severe as it may sound. With our approach, no Java nor any native code could ever exploit a removed permission, no matter what, since

involved permission requests are actually removed from the app manifest. As remarked, the worst case scenario is a crash of the less-privileged app, but not a privacy leak. Android 6 has introduced the possibility of both installing apps without granting all requested permissions at once, and toggling permissions at a later time. So, the usefulness of RmPerm could appear as dramatically reduced through the growing adoption of the latest Android versions, but RmPerm offers a finer grained permission selection, which is unavailable in the Android user interface. In fact, Android 6 allows the user to only grant/deny *groups* of permissions. For instance, because the user-level permission group *contacts* consists of the set of permissions `READ_CONTACTS`, `WRITE_CONTACTS` and `GET_ACCOUNTS`, a user cannot grant an app the ability to read his/her contacts, without granting the ability to write them too. While, by using RmPerm, such a policy is easily enforceable.

Chapter 4

The Importance of Obfuscation

4.1 Motivation and significance

Obfuscation is a *security through obscurity* technique that modifies the code in order to counteract automatic or manual code analysis. However, it is considered as a double-edged sword by the security community because both software developers and malware authors frequently use obfuscation. In fact, on the one hand, obfuscation keeps developers' competitors away from copying the code and makes it difficult for attackers to alter the regular flow of the software (e.g., *cracking*). On the other hand, it also helps malware authors to circumvent automated code analysis and manual inspection. As an example, consider a malware sample that is being recognized by anti-virus engines. In this case, the malware author needs to quickly build a variant of the original malware that could go undetected. Since creating a variant from scratch is time-consuming, obfuscating the code of the original malware is often considered a good compromise.

The challenges around obfuscation have attracted many researchers, especially in the field of mobile apps, given the astonishing growth of such markets. In the mobile world, especially the one focusing on the Android platform, obfuscation is rather common. Disassembling (or decompiling) and rebuilding an Android app is more straightforward w.r.t. other binary code, like, e.g., x86 executables. So far, most of the studies on Android app obfuscation focus on how: i) to build reliable obfuscation techniques (AN14; SLZG14), ii) obfuscation can be handled by state-of-art code analysis tools (RCJ13), iii) to automatically deobfuscate the code (BRTV16), and iv) developers actually adopt obfuscation nowadays (DLD+18; WHA+18). The recent research works suggest that many developers are unable to apply advanced obfuscation techniques and that free off-the-shelf obfuscators support only basic obfuscation or are very difficult to configure. Furthermore, there are several academic works (WMW+12; AZ13; SSL+13; LL14; ASH+14; BTG+16; IRC+17;

AMM+17) that take advantage of machine learning techniques for malware detection; however, only few of them (STDA+17; GHP+15) take into consideration obfuscation in their models. It is also worth to point out that there is an increasing trend (DMB+17; CLW+18) on applying adversarial machine learning on mobile, with the aim to study how an attacker could modify an app in order to evade an existing ML model. However, there exist only a couple of tools (i.e., ADAM (ZLL12) and AAMO (DPM17)) that allow applying (some) obfuscation techniques *automatically*. Unfortunately, both of them were never updated since the year they were released, ADAM does not implement advanced obfuscation techniques, and the current version of AAMO does not seem to work correctly. To overcome the limitation of previous tools, we have developed **Obfuscapk**, a free Python tool that is able to obfuscate compiled Android apps (i.e., without the need of the source code). Obfuscapk supports advanced obfuscation features (e.g., string encryption and native libraries encryption), and its modular architecture easily allows to add new obfuscation techniques by the community. We tested Obfuscapk on 1000 APKs among the most installed apps from the Google Play Store; our experiments indicate that Obfuscapk automatically generates obfuscated full working apps in the 83% of the cases. Obfuscapk aims at becoming a useful tool for both the developers' and the research communities. On the one hand, developers can use Obfuscapk in cooperation with ProGuard, which is the default optimizer and obfuscator included in the Android SDK and supported by the official Android Studio IDE. First, the developer uses the Android SDK with ProGuard to release the APK, and then she can apply more advanced obfuscation techniques through Obfuscapk. On the other hand, the research community on mobile security can apply Obfuscapk as a black-box obfuscation tool to apps and malware samples for several aims, like building or attacking a machine learning model, improving program analysis techniques w.r.t. obfuscation transformations, just to cite a few. Finally, each user can extend the current tool by adding her own or other obfuscation techniques at state of the art.

4.2 Supported Obfuscation Techniques

In this section, we describe the techniques supported by Obfuscapk, with a specific focus on their impact on malware detection. There exists a classification (MAC+15; DPM17) of obfuscation techniques for the Android ecosystem, which divides the techniques in two main categories: trivial and non-trivial.
The following techniques are the result of an in-depth study about state-of-the-art obfuscation techniques for the Android platform.

4.2.1 Trivial techniques

Trivial techniques are the simplest ones. They have no real obfuscation effects on the APK, but they can trick some signature-based anti-malware tools (MAC$^+$15). Obuscapk implements four existing trivial techniques, namely: **Align**, **Re-sign**, **Rebuild**, and **Randomize Manifest**.

4.2.1.1 Align and Re-sign

These techniques implement the last mandatory steps for building a working Android APK. The alignment is done by using `zipalign`, a specific tool of the Android SDK. The result is a reorganized application in which the structure of the files is optimized for running on an Android device. Android requires all APKs to be digitally signed with a certificate before they can be installed on a device (or updated), so the Re-sing step is the last mandatory step after applying obfuscation.

4.2.1.2 Rebuild

The bytecode contained in `classes.dex` file can be disassembled and reassembled to obtain a different version of the file. Such technique transforms the bytecode without changing its semantic, in order to preserve the original behavior of the app. This rebuild aims to fool anti-malware tools that use the signature of *classes.dex* file.

4.2.1.3 Randomize Manifest

Such technique randomly rearranges the entries in the `AndroidManifest.xml`, without modifying the XML tree structure. The goals are twofold: i) change the hash of the manifest file, and ii) fool the N-gram analysis (SFE10).

4.2.2 Non-trivial techniques

Non-trivial techniques are more complex, but they grant a more profitable gain in terms of detection rate and robustness (MAC$^+$15). The targets of obfuscation are both bytecode and resources (XMLs, asset files, and external libraries). Non-trivial obfuscation techniques can be divided into four subcategories: **Renaming**, **Encryption**, **Code**, and **Resources**.

4.2.2.1 Renaming

In software development, the names of identifiers (variable names, function names, and so forth) should be meaningful to provide good code readability and maintainability. However, such clear names may leak information about code functionalities. Furthermore, since the package name uniquely identifies an Android app, its modification amounts to put a new app in the Android ecosystem. Therefore, the renaming technique substitutes each identifier with an obscure and meaningless one. While the methods and fields renaming has no drawbacks, the classes and package name renaming is more complicated because the `AndroidManifest.xml` must be updated accordingly.

4.2.2.2 Encryption

An APK file may contain resources that can be requested at run-time by the developer. Those files can be native libraries or even strings. Such resources can be encrypted and decrypted at run-time. In this case, the attacker needs another step to find the decryption key before reading the resources, but there is also an obvious disadvantage: the app performances get worse because it needs extra calculations when it accesses its resources. When Obfuscapk starts, it automatically generates a random secret key (32 characters long, using ASCII letters and digits) that can be used to encrypt:

- **LibEncryption**. Native libraries;
- **AssetEncryption**. Asset files (like videos, photos, text files, etc.);
- **ResStringEncryption**. Strings contained in the `strings.xml` resource file.
- **ConstStringEncryption**. Constant strings in the code.

4.2.2.3 Code

This category contains all obfuscation techniques that affect instructions inside the `classes.dex`. There exist several techniques that hide the behavior of the application, each of which is applied to a different aspect of the code.

DebugRemoval. This technique just removes debug meta-data. The removal of debug information, such as line numbers, types, or method names, reduces the amount of useful information for the reverse engineering process.

CallIndirection. This technique modifies the control-flow graph (CFG from now on) without impacting the code semantics; it adds new methods that invoke the original ones.

For example, an invocation to the method m_1 will be substituted by a new wrapper method m_2, that, when invoked, it calls the original method m_1.

Goto. First, given a method, it inserts a `goto` instruction pointing to the end of the method and another `goto` pointing to the instruction after the first `goto`; it modifies the CFG by adding two new nodes. Then it randomly re-arrange the code abusing `goto` instructions.

Reorder. This technique consists of changing the order of basic blocks in the code. When a branch instruction is found, the condition is inverted (e.g., "branch if lower than", becomes "branch if greater or equal than") and the target basic blocks are reordered accordingly.

ArithmeticBranch. This is the first technique that belongs to the *junk code insertion* category, that aims at adding some useless and semantic-preserving instructions to the code. In this case, the junk code is composed by arithmetic computations and a branch instruction depending on the result of these computations, crafted in such a way that the branch is never taken.

Nop. Nop, short for *no-operation*, is a dedicated instruction that does nothing. This technique just inserts random `nop` instructions (i.e., junk code) within every method implementation.

MethodsOverload. It exploits the overloading feature of the Java programming language. Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Given an already existing method, this technique creates a new void method with the same name and arguments, but it also adds new random arguments. Then, the body of the new method is filled with random arithmetic instructions.

4.2.2.4 Invocation by Reflection

The Reflection is a feature of the Java programming language that allows examining or modifying the run-time behavior of a class during execution. In this context, this feature is used to invoke methods of a given object.

Reflection. This technique analyzes the existing code looking for method invocations of the app, ignoring the calls to the Android framework. If it finds an instruction with a suitable method invocation (i.e., no constructor methods, public visibility, enough free registers, ...) such invocation is redirected to a custom method that will invoke the original method using the Reflection APIs.

AdvancedReflection. This technique is complementary to the previous one because it works in the same way, but it targets the invocations of dangerous APIs. In order to

find out if a method belongs to the Android Framework, Obfuscapk refers to the mapping discovered by Backes et al. (BBD+16).

4.2.3 Summary of techniques by categories

Table 4.1 summarizes the aforementioned techniques implemented in Obfuscapk, grouped by categories.

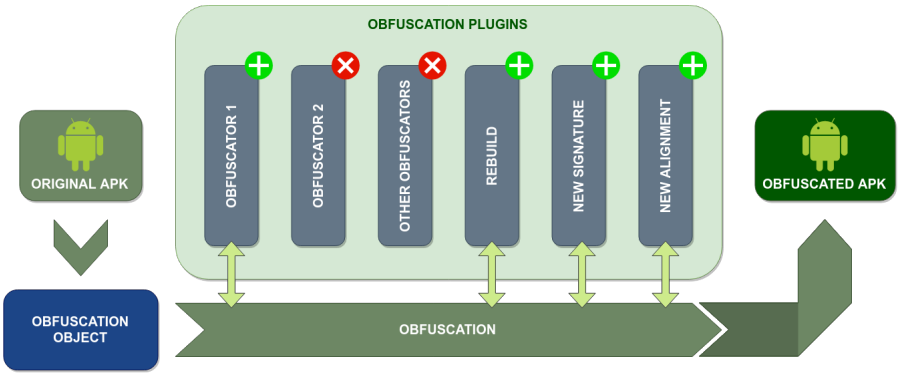
Table 4.1: Obfuscapk implemented obfuscators

Category	Obfuscapk obfuscator
Trivial	`RandomManifest`, `Rebuild`, `NewAlignment`, `NewSignature`
Renaming	`ClassRename`, `FieldRename`, `MethodRename`
Encryption	`LibEncryption`, `ResStringEncryption`, `AssetEncryption`, `ConstStringEncryption`
Code	`ArithmeticBranch`, `Reorder`, `CallIndirection`, `DebugRemoval`, `Goto`, `MethodOverload`, `Nop`
Reflection	`Reflection`, `AdvancedReflection`

4.3 Software description

4.3.1 Software Architecture

Figure 4.1: Obfuscapk Architecture



Obfuscapk is designed (see Figure 4.1) to be modular and easy to extend, so it is built on *Yapsy*, a plugin management system. Consequently, each obfuscator is a plugin that

inherits from an abstract base class and needs to implement the method `obfuscate`. When the tool begins to process an APK, it creates an `obfuscation` object to store all the needed information (i.e., the location of the decompiled code) and the internal state of the operations (i.e., the list of already used obfuscators). Then, the `obfuscation` object is passed, as a parameter to the obfuscate method, to all the active plugins/obfuscators sequentially. The list and the order of the active plugins are specified through command-line options.

The tool is easily extensible with new obfuscators: it is enough to add the source code implementing the obfuscation technique and the plugin metadata (a `<obfuscator-name>.obfuscator` file) in the `src/obfuscapk/obfuscators` directory. The tool will automatically detect the new plugin, with no need of further configuration steps.

A limitation of our current implementation is that apps using anti-tampering techniques can detect that they have been rewritten. However, our experiments indicate that, for the time being, Android apps very seldom adopt anti-tampering techniques.

4.3.2 Tool Functionalities

The complete set of Obfuscapk functionalities is provided by the following help message:

```
obfuscapk [-h] -o OBFUSCATOR [-w DIR] [-d OUT_APK]
          [-i] [-p] [-k VT_API_KEY]
          <APK_FILE>
```

There are two mandatory parameters: `<APK_FILE>`, the path (relative or absolute) to the apk file to obfuscate and the list with the names of the obfuscation techniques to apply (specified with the `-o` option). The remaining parameters are optional.

-o the list with the names of the obfuscation techniques (previously described in Section 4.2 and summarized in Table 4.1) to apply; e.g., `-o Rebuild -o NewSignature -o NewAlignment`.

-w `DIR` is used to set the working directory used to store the intermediate files (generated by apktool). If not specified, a directory named `obfuscation_working_dir` is created in the same directory of the input app. This option can be useful for debugging purposes.

-d `OUT_APK` is used to set the path of the destination file, i.e., the apk file generated by the obfuscation process. If not specified, the final obfuscated file will be saved inside the working directory. Existing files will be overwritten without any warning.

-i is a flag for ignoring known third party libraries during the obfuscation. This option could be useful to improve performances and reduce the risk of errors. The list of libraries to ignore is obtained from the LiteRadar project (lit17)

-p is a flag for showing progress bars during the obfuscation operations.

-k `VT_API_KEY` is only needed when using VirusTotal obfuscator, to set the API key(s) to be used when communicating with Virus Total. It can be set multiple times to cycle through the API keys during the requests (e.g., `-k VT_KEY_1 -k VT_KEY_2`).

4.4 Illustrative Example

As an example, we use Obfuscapk to obfuscate an Android malware discovered in early 2019, a Trojan-Banker named *CometBot*. In this example, we have obfuscated our specimen (vtcf) using the different sets of techniques implemented in Obfuscapk (summarized in Table 4.1). Then, we have uploaded the obfuscated sample to Virus Total (VIR); results are reported in Table 4.2, ordered by detection ratio.

Table 4.2: Detection ratio of different obfuscated versions

Category	**Detection ratio**	**Percentage**
Original	32/58 (vtcf)	55%
Trivial	18/58 (vtce)	31%
Renaming	16/58 (vtcd)	28%
Reflection	15/58 (vtcc)	26%
Code	8/58 (vtca)	14%
Encryption	0/58 (vtcb)	0%

This example shows how different types of obfuscation influence, more or less, the detection ratio. In this particular case, the techniques of the Encryption category allowed to build an *undetectable variant*. Since an Android APK is an archive that contains several files and a malicious component might be implemented almost everywhere, it is not possible to establish which is the most effective subset of techniques *a priori*, since each technique has different effects on the files within the APK.

Listing 4.1 shows the command line parameters for obfuscating the CometBot malware using the Encryption techniques.

Listing 4.1: Obfuscating Cometbot malware using encryption

```
obfuscapk \
-o LibEncryption -o ResStringEncryption \
-o AssetEncryption -o ConstStringEncryption \
-o Rebuild -o NewAlignment -o NewSignature \
-d encryption.apk cometbot.apk
```

4.5 Testing the Stability of Obfuscapk

We empirically evaluated the stability of Obfuscapk by obfuscating, using each implemented technique, a dataset of 1000 APKs randomly downloaded from the Google Play Store, among the top free apps by the number of installations (andb). Then, we have tested if the modified APK can be still installed and if it runs properly.

A single automated test, for each APK A, executes as follow:

1. Obfuscapk obfuscates A, and builds up a new APK A'.
2. A' is installed on an actual Android device.
3. If this step fails, the original A is installed, in order to verify whether the failure is due to the modification carried out by Obfuscapk or it is independent.
4. If the installation of A' has been successful, its behavior is tested by generating a stream of 1024 pseudo-random user events with *Monkey* (mon17), seeded by a random number n. Using different seed values leads to generate distinct sequences of user events. If A' fails, this can be due either to Obfuscapk transformations or to the presence of bugs in the original app A.
5. To discriminate, we stimulate A with the same stream of events, generated by seeding *Monkey* with the same seed n.

Through previous steps, we can empirically assess whether the obfuscation process leads to failures. We carried out the experimental assessment on a Dell XPS 9530 (Ubuntu 18.04, Intel Core i7-4712HQ @ 2.30GHz, 16GB RAM), as well as on a OnePlus 6 (Android 9, Snapdragon 845 @ 2.8 GHz, 8GB RAM) that we used to obfuscate, install and (automatically) stimulate the apps.

Our results indicate that on 1000 samples, 47 repackaged apps failed at Step 2; that is, they could not be installed successfully. Among these, 8 failed at Step 3 too; this means that the corresponding original APKs were already broken in some way. Therefore, we discarded

them, and we considered a new set, consisting of the original samples without the broken APKs; that is, $1000 - 8 = 992$ apps. On this set, the 95% (940/992) of repackaged apps have been successfully installed. Then, we stimulated the installed apps according to 4. Among these, 817 were repackaged and stimulated without crashes, while $123 = 940 - 817$ failed and required further analysis. Therefore, we applied Step 5 to such apps obtaining that 6 original apps crashed, thereby proving that the same problems affected the original app, too. For this reason, we also discarded these APKs, reaching a total of $992 - 6 = 986$ of working original apps. Summing up, the $83\% = 817/986$ of working apps have been successfully obfuscated, installed, and executed properly after the obfuscation process. It is worth pointing out that *Monkey* (mon17) is not a comprehensive tool for dynamically testing Android APK; therefore, such 83% must be considered an upper-bound.

Chapter 5

Malware Detection Using Permissions and API invocations

5.1 Introduction

In the field of computing the problem of sustainability may be tackled from several different angles. A first approach takes into account the problem of reducing the resource consumption of computing centers (LZ11), while a second one is dedicated to the greening of the network infrastructures (BCRR12). At the same time, the need to control the resource consumption cannot interfere with the need to guarantee the desired levels of security (MMC15). However, as the number of users relying on mobile devices for their daily routines increases and IoT systems enter the mainstream, sustainability cannot be seen only as an effort to reduce the resource footprint of systems that are intrinsically not constrained, but also as the need to develop new methodologies that allow both performing traditionally resource hungry activities on resource constrained devices and reduce the impact of attacks on the energy consumption of the device (CM12)(PRF11)(FCDSP17). In particular, the problem of providing appropriate security levels without depleting the resources of devices is of paramount importance. To this aim, in this chapter we focus on the problem of providing a methodology to detect malicious software on resource constrained devices with a very low resource footprint. Furthermore, to prove the efficacy of our methodology we also provide an actual tool implementing it on the Android operating system. As the traditional signature-based mechanism cannot cope with the malware evolution, in recent years several efforts have been put forward by the research community to define new approaches to malware detection. Among others, the most promising ones rely on data-driven techniques whose aim is to learn how to classify apps into two sets, namely *legal* and *malware* apps, based on the analysis of already classified apps.

This new approach, however, is usually resource hungry and has some drawbacks that may limit its actual applicability. The first issue is that data-driven techniques need to extract meaningful features from the apps that have to be classified. A very rich set of features may allow building a very precise classification model but it may also overload the device beyond acceptable usability, thereby preventing the implementation of model on the device. For this reason, it is necessary to extract a set of features which is large enough to support the definition of a reliable model, but also small enough to be computed on the resource constrained device in a sustainable way. Another limitation of this kind of models arises when the behavior of the program to be classified is evaluated at runtime (i.e., dynamic analysis). In fact, in such a situation, there is always the risk to recognize a threat only after the system has been compromised: therefore, this kind of analysis is generally carried out off-device, in secure *sandboxed* environments, albeit some proposals for anomaly detection at runtime on mobile have been recently put forward. When dealing with the need of on-device analysis, the safest solution is to rely only on features that may be statically extracted from the program code without the need to execute it (i.e., static analysis). Finally, from a data-scientist point of view, there is the need to deal with the *overfitting* problem that occurs when the model excessively adapts to the set of apps on which the model has been trained, without guaranteeing an adequate generalization performance needed to detect previously unseen apps (i.e., zero-day malware).

In this chapter we present BadDroids, an Android app capable of analyzing apps as soon as they are installed on the device, thereby allowing to classify an app as legal or malware before its execution, with a high degree of accuracy and a low resource footprint. We adopted state-of-the-art data-driven machine learning techniques for building the malware detection system, in such a way that they can efficiently execute on a resource-constrained mobile device. In detail, we studied both qualitatively and quantitatively how the model works and how malware can be detected. Our tests have been performed by merging the most recently updated malware databases for a total of more than seven thousand malware samples, and have demonstrated to correctly classify the apps in approximately the 99% of the cases. The performance of BadDroids suggests that it can be adopted on actual mobile devices to execute on-the-fly analysis of new apps with a very limited impact on the user experience.

Structure of the chapter. The rest of the chapter is organized as follows: Section 5.2 discusses some related work, while Section 5.3 presents the data-driven model at the basis of BadDroids. Section 5.4 discusses the experimental results of BadDroids on the field. Finally, 5.5 concludes the chapter by discussing some future development of BadDroids.

5.2 Related work

The aim of this work is to define a new approach to binary classification of malware on Android through data-driven techniques that take into account as features both the permissions required by apps as well as the Android API (AAPI, see Chapter 2) they invoke. Several examples of data-driven Android malware detection systems exist in literature. One of the earliest work is KIRIN (EOM09). It defines a set of security rules describing potentially dangerous permission patterns. For instance, an app requiring both the `RECEIVE_SMS` and the `SEND_SMS` permissions is considered risky by KIRIN. The approach has been assessed on a very limited set of apps (i.e., 311), and allowed to recognize 10 apps violating all inferred security rules. Among them, 5 has been proved to be real malware through manual code inspection.

Table 5.1: Related work comparison

Paper	DroidMat	PBMD	PUMA	TLPD	Drebin	MDLS	PIndroid	**BadDroids**
Reference	(WMW+12)	(AZ13)	(SSL+13)	(LL14)	(ASH+14)	(BTG+16)	(IRC+17)	
Year	2012	2013	2013	2014	2014	2016	2017	2017
Sustainability	L	L	H	M	M	L	H	H
Static features								
Req. perm.	✓	✓	✓	✓	✓	✓	✓	✓
Used perm.		✓		✓	✓			
App components	✓				✓			
Intents	✓				✓		✓	
API calls*	✓				✓	✓		✓
Inter-Comp Comm.	✓							
Market desc.						✓		
String pattern					✓			
Dynamic anal.		✓						
N°of apps	1738	21684	606	30084	129013	78649	1745	14988
Legal	1500	20548	357	28548	123453	52251	1300	7494
Malware	238	1136	249	1536	5560	26398*	445	7494
Accuracy	97.87	98	78	98.6	94	94	99.8	98.9

Table 5.1 provides a comparison among some of the most influential works that use ML techniques for malware detection, ordered by year of publication. We defined three labels, namely L(ow) M(edium) H(igh), describing the sustainability of the approach. The labeling is based on the resource footprint of the described tool, taking into account how much computational power is needed for collecting the features and apply the model to them. We always assume that the model is generated only once, during the training phase, not on the resource constrained device, hence the cost of the model generation is not used to evaluate the sustainability of the approach.

DroidMat(WMW+12), PBMD(AZ13) and MDLS(BTG+16) are considered Low-sustainable for the following reasons: DroidMat requires to create the Inter-procedural control flow graph of the app, PBMD uses dynamic analysis, and MDLS downloads and parses the market description. In this respect, we choose to avoid considering the whole invocation

chain our work as this would require to statically build a data structure (e.g., a flow graph) that is expensive in terms of memory and computational power.

TLPD (LL14) and Drebin(ASH+14) are considered Medium-sustainable because for generating the *Used permissions* set they must check if every method invocation in the code requires some permissions.

PUMA(SSL+13), PIndroid(IRC+17) and BadDroids are considered High-sustainable because they collect the features with a linear analysis of the bytecode and the Android Manifest file.

Previous works focus on the extraction of the following static features: requested permissions, used permissions, app components, intents, inter-component communication (ICC), meta data extracted from online market description, string patterns (e.g. URLs, IP addresses, base64, etc.) and API calls. For the sake of clarification, ours is the only approach that considers *every* API, i.e. every method invocation (that cannot be obfuscated) given by the language and the Android framework. For example we also consider the `java.lang.String` constructor. Usually authors check different subsets of API, often related to privacy or permissions declared in the manifest, but this selection lacks of important feature like Reflection and the loading of native code.

We chose a small subset of the most significant features among the ones that were already being researched extensively in previous publications and we demonstrated that they are enough for a very good classification. Obviously, other alternatives can be taken into consideration, but they would require too much memory or computation in order to be efficiently usable on a mobile device.

None of the cited articles use our set of features but our approach has higher accuracy with respect to any other chapter in literature except PIndroid(IRC+17). However, the high level of accuracy granted by PIndroid is calculated on a very reduced dataset of malware samples. Furthermore, no other work in literature also provides a freely available app for testing and the whole set of data allowing to replicate and check the presented statements. Finally, it is worth noting that our testbed is the second biggest malware dataset (7494) w.r.t. other works. Indeed, only in the MDLS experience presented in (BTG+16) the authors tested the solution on a dataset of 26398 malware samples.

5.3 The Data Driven Classification Model

For our specific malware classification purposes, we consider the supervised learning framework, with particular reference to the binary classification problem, where an input space $\mathcal{X}$ and an output space $\mathcal{Y}$ are available (Vap98). In our case $\mathcal{X} = \{0,1\}^d$, where each element of the space represents the presence or the absence of a particular declared per-

mission or AAPI invocation (as AAPI invocations are retrieved from code through static analysis we will call them retrieved AAPI), and $\mathcal{Y} = \{\pm 1\}$, since the possible labels are legal $(+1)$ or malware (-1). Note that, the same problem can be faced as a novelty detection task (STC04). In fact, in real world situations, the number of malware applications is much lower with respect to the number of legal ones. We made a preliminary study on the available data by exploiting the most recent tools in the novelty detection context (SMS$^+$16), but results were not satisfying both in terms of accuracy and also in terms of resource requirements because of the need for the use of the kernel and a huge number of legal apps. In the supervised learning framework, the goal is to estimate the unknown rule $\mu : \mathcal{X} \to \mathcal{Y}$ which associates a label $Y \in \mathcal{Y}$ to an element $X \in \mathcal{X}$. In general, μ can be non-deterministic (Vap98) (i.e.,different apps may have the same sets of declared permissions and/or retrieved AAPI invocation but different label). A data driven technique estimates μ through a learning algorithm $\mathscr{A}_{\mathcal{H}} : \mathcal{D}_n \times \mathcal{F} \to f$, characterized by its set of hyperparameters $\mathcal{H}$, which maps a series of examples of the input/output relation, contained in a dataset of n samples $\mathcal{D}_n : \{(X_1, Y_1), \cdots, (X_n, Y_n)\}$ sampled from μ (or in other words n different labelled Android apps), into a function $f : \mathcal{X} \to \mathcal{Y}$. The error that f commits, in approximating μ, is measured with reference to a loss function $\ell : \mathcal{X} \times \mathcal{Y} \times \mathcal{F} \to [0, \infty)$. In our case, we will make use of the Hard loss function which counts the number of errors $\ell_{\mathrm{H}}(f(X), Y) = [f(X) \neq Y] \in \{0, 1\}$ (Vap98). The purpose of any learning procedure is to select the best set of hyperparameters $\mathcal{H}$ such that the expected error $L(f) = \mathbb{E}_{\mu}\ell(f(X), Y)$ – which unfortunately is unknown since μ is unknown – is minimum. Obviously, the error that f commits over $\mathcal{D}_n$ is optimistically biased since $\mathcal{D}_n$ has been used for building f itself. For this reason another set of fresh data, composed of m samples and called test set $\mathcal{T}_m = \{(X_1^t, y_1^t), \cdots, (X_m^t, y_m^t)\}$, needs to be exploited. Note that, $X_i^t \in \mathcal{X}$ and $Y_i^t \in \mathcal{Y}$ with $i \in \{1, \cdots, m\}$, and the association of Y_i^t to X_i^t is again made based on μ.

Many different algorithms exist in literature such as the Kernel-based method (STC04), the Neural Network-based one (Bis95; GBC16; HWL11), the Ensemble Methods (ZM12), the Bayesian approaches (RW06b), the Local Methods (CH67), among others. In our case, we need to keep in mind that the classification model $f = \mathscr{A}_{\mathcal{H}}(\mathcal{D}_n)$ needs to run on a mobile device. For this reason we have to exploit a model which requires as little computational effort as possible (ORA16). In particular, the computational requirements of the training phase, namely the time needed to build f, are not important since the training phase can be performed offline. What is instead crucial are the computational requirements needed in order to compute $f(X)$ since it must be done on the device. In this context Kernel-based method are usually the best suited choice (AGO$^+$12; ORA16). Other alternatives exist such as Extreme Learning Machines (TDH16), Deep Neural Networks (S.15), Random Forests (Bre01), or Gaussian Processes (RW06a) but the forward phase of these methods usually requires much more memory and computations with respect to our proposal (ORA16).

We use two learning algorithms, one linear and one nonlinear by carefully considering the

computational requirements of computing $f(X)$. Moreover, we will try to get insight on the problem of detecting a malware based on declared permissions and retrieved AAPI invocations by detecting the most important subset of them and their weight (i.e., if the presence of an invocation or a declared permissions is an indication of malware or not). Finally, we will show how to tune the hyperparameters of the different algorithms and how not to simply get a binary answer from $f(X)$ (legal or malware) but also a reliability estimation of such response.

For what concerns the linear approach, let us define $\mathcal{F}$ as the set of all the possible linear separators in the space $\mathcal{X}$: $f(X) = W^T X + b$ with $W \in \mathbb{R}^d$ and $b \in \mathbb{R}$ (Vap98). Based on this choice the most intuitive way of choosing W and b is to choose the solution which minimizes the error over the available data (Vap98):

$$(W, b) : \arg\min_{W,b} \sum_{i=1}^{n} \ell_{\mathrm{H}}(W^T X_i + b, Y_i). \tag{5.1}$$

Unfortunately, the Problem (5.1) has two drawbacks: (i) it is NP-Hard since the loss function is non-convex (ORA16) and (ii) it is ill-posed and may overfit the available data and have large expected error (BG12). In order to solve issue (i) it is necessary to approximate ℓ_{H} with one of its convex relaxations. The most suited one is the Hinge loss function $\ell_\xi(f(X_i), Y_i) = \max[0, 1 - Y_i f(X_i)]$ (Vap98), the simplest convex upper bound of ℓ_{H}, which is also the best choice in this context (RDVC$^+$04). By solving issue (i) we can also address the issue (ii) since, by exploiting ℓ_ξ, it is possible to introduce a regularization term, inspired by the Tikhonov regularization principle (TA77), which allows to derive a well posed alternative to Problem (5.1). Several regularization terms exists, form the L1 to the L2 and Lp norms (Tib96; SLS06; ZH05), but in this work we will exploit the combination of the L1 and the L2 regularization schemes (ZH05). This choice is made since, in our case, $d \gg n$ and the presence of many declared permissions and retrieved AAPI invocations are correlated with each other. L1L2 regularization schema, also called elastic net regularization (ZH05), is both a regularization and variable selection method. L1L2 often outperforms the L1, while providing a similar sparsity of representation. In addition, the L1L2 encourages a grouping effect, where strongly correlated features tend to be in or out of the model together. L1L2 is particularly useful when $d \gg n$ (as in our case) and, contrarily to L1, it is a very satisfactory variable selection method when $d \gg n$. Consequently Problem (5.1) can be reformulated as follows:

$$\begin{aligned}(W, b) : \arg\min_{W,b} \lambda \|W\|_2^2 + (1 - \lambda)\|W\|_1 \\ + \frac{C}{n} \sum_{i=1}^{n} \max[0, 1 - (Y_i W^T X_i + b)],\end{aligned} \tag{5.2}$$

which is a convex problem than can be solved with many tools developed in recent years (ZH05; AGO$^+$13). Note that $\lambda \in (0, 1)$ is a constant that balances sparsity characteristics

with feature selection ability (ZH05) while $C \in (0, \infty)$ is another constant which balances the tradeoff between underfitting and overfitting tendency (TA77). The sparsity effect of the L1 regularizers also allows to reduce the number of $W_{j \in \{1,\cdots,d\}} \neq 0$ and then to obtain a model f which can run with reduced computational requirements (ORA16).

The shape of the model f, built by solving Problem (5.2), together with the sparsity effect of the L1 regularizers and the fact that $\mathcal{X} = \{0,1\}^d$, allows us to derive a simple yet effective and efficient feature selection and ranking method which also has the ability to infer if a feature is an indicator of malware or legal (GE03). In fact, if some W_j with $j \in \{1,\cdots,d\}$ are equal to zero the meaning is straightforward: that feature j-th is not meaningful for distinguishing between legal or malware. If, instead, a particular W_j with $j \in \{1,\cdots,d\}$ is different from zero, since $X_j \in \{0,1\}$, and $W_j > 0$ the feature j-th is an indication that the app is a malware. Analogously, if $W_j < 0$ the feature j-th is an indication that the app is legal. Finally, the magnitude of W_j gives its raw importance.

The limitation of Problem (5.2) is the shape of f which is linear (Vap98). In order to overcome this limitation, we can define f as a nonlinear function $f(X) = W^T\Phi(X) + b$ where $\Phi : \mathbb{R}^d \to \mathbb{R}^D$ with $D \gg d$ (since with d features we were not able to find a good classifier), $W \in \mathbb{R}^D$, and $b \in \mathbb{R}$. Then, we can substitute the new f in Problem (5.2) and exploit the representer theorem (SHS01) in order to observe that the solution of Problem (5.2) can be expressed as $W = \sum_{i=1}^n \alpha_i \Phi(X_i)$ with $\alpha_i \in \mathbb{R}$ where $i \in \{1,\cdots,n\}$. By substituting these results in Problem (5.2) we obtain the following problem:

$$\begin{aligned}(W,b,\alpha) : \arg\min_{W,b,\alpha}\ & \lambda\|W\|_2^2 + (1-\lambda)\|W\|_1 \\ & + \frac{C}{n}\sum_{i=1}^n \max[0, 1 - (Y_i W^T\Phi(X_i) + b)] \\ s.t.\ & W = \sum_{i=1}^n \alpha_i \Phi(X_i).\end{aligned} \tag{5.3}$$

Note that, Problem (5.3) suffers from the curse of dimensionality since the size of the problem depends on D. If D is large it may become intractable. For this reason, if we exploit the kernel trick (Sch01), and, instead of applying the regularization over W we equivalently apply the regularization to α, we obtain the following problem:

$$\begin{aligned}(\alpha,b) : \arg\min_{\alpha,b}\ & \lambda\|\alpha\|_2^2 + (1-\lambda)\|\alpha\|_1 \\ & + \frac{C}{n}\sum_{i=1}^n \max\left[0, 1 - \left(Y_i \sum_{j=1}^n \alpha_j K(X_i, X_j) + b\right)\right],\end{aligned} \tag{5.4}$$

where $K(X_i, X_j) = \Phi(X_i)^T\Phi(X_j)$, Φ can remain unknown, $f(X) = \sum_{i=1}^n \alpha_j K(X_i, X) + b$, and the problem is still convex. We opt for a Gaussian Kernel $K(X_i, X_j) = e^{-\|X_i - X_j\|_2^2/\sigma}$

for the reason described in (KL03). Obviously, the feature selection and ranking phase in this case is not possible but Problem (5.4) still takes into account the computational requirements of a mobile device. In fact, the L1 regularization allows to reduce the number of α_i with $i \in \{1, \cdots, n\}$ different from zero. The smaller the number of αs different from zero is, the less computational expensive is the computation of $f(X)$.

Another issue that we have to face is how to tune the hyperparameters of the proposed algorithms (λ, C, and σ for the nonlinear case) (AGOR12). The values of the hyperparameters deeply affect the performance of the final classification model $\mathscr{A}_{\mathcal{H}}(\mathcal{D}_n)$ and for this reason they must be tuned carefully. Resampling techniques like cross validation (AC10) and non-parametric bootstrap (ET94) (BOO) are often used by practitioners because they work well in many situations (AGOR12). Other alternatives exist, which are bases in the Statistical Learning Theory, which give more insight into the learning process. Examples of methods in this last category are: the seminal work of the Vapnik-Chervonenkis Dimension (Vap98), its improvement with the Rademacher Complexity (BM02; BBM05), the theory of compression (FW95; LM04), the Algorithmic Stability breakthrough (BE02), the PAC-Bayes theory (LLST13; GLLF15), and more recently the Differential Privacy theory (DFH$^+$15a; DFH$^+$15b).

In our specific case the BOO will be exploited since it is the most effective one in cases like the one described in the chapter, where the cardinality of the sample is reasonable (AGOR12). BOO relies on a simple idea: the original dataset $\mathcal{D}_n$ is resampled many (n_o) times with replacement, to build two independent datasets called training and validation sets, respectively $\mathcal{L}_l^o$ and $\mathcal{V}_v^o$, with $o \in \{1, \cdots, n_o\}$. Note that $\mathcal{L}_l^o \cap \mathcal{V}_v^o = \emptyset$. Then, in order to select the best set of hyperparameters $\mathcal{H}$ in the set of possible ones $\mathfrak{H} = \{\mathcal{H}_1, \mathcal{H}_2, \cdots\}$ for the algorithm $\mathscr{A}_{\mathcal{H}}$ or, in other words, to perform the model selection phase, the following procedure needs to be applied:

$$\mathcal{H}^* : \arg\min_{\mathcal{H} \in \mathfrak{H}} \frac{1}{n_o} \sum_{o=1}^{n_o} \widehat{L}^{\mathcal{V}_v^o}(\mathscr{A}_{\mathcal{H}}(\mathcal{L}_l^o)), \tag{5.5}$$

where $\widehat{L}^{\mathcal{S}}(f) = {}^1\!/\!_{|\mathcal{S}|} \sum_{(X,Y)\in\mathcal{S}} \ell_H(f(X), Y)$. Since the data in $\mathcal{L}_l^o$ are different with respect to the ones in $\mathcal{V}_v^o$, the idea is that $\mathcal{H}^*$ should be the set of hyperparameters which allows to achieve a small error on a data set that is independent from the training set. Note that, in BOO, $l = n$ and $\mathcal{L}_l^o$ must be sampled with replacement from $\mathcal{D}_n$, while $\mathcal{V}_v^o = \mathcal{D}_n \setminus \mathcal{L}_l^o$.

Finally, it is worth underlining that the classifier that we have just proposed gives, as an output, only the answer legal or malware. In a real world scenario this information is not enough. What it is important is also the reliability of the models' answers. For this reason we exploit the proposal of (Pla99) which is able to take the $f(X)$ (in our case $f(X) = W^T X + b$ for the linear model and $f(X) = \sum_{(X_i,Y_i)\in\mathcal{D}_n} \alpha_j K(X_i, X) + b$ for the

nonlinear one) and associates a probability to the choice of the model. In particular:

$$\mathbb{P}\{f(X) = +1\} = \frac{1}{1 + e^{\gamma f(X)+\beta}}, \tag{5.6}$$

where $\gamma \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are chosen by minimizing the negative log likelihood averaged over the different $\mathcal{V}_v^o$ which is a cross-entropy error function:

$$(\gamma, \beta) : \arg\min_{\gamma,\beta} - \sum_{o=1}^{n_o} \sum_{(X,Y)\in\mathcal{V}_v^o} \left[\left(\frac{Y+1}{2} \right) \log \left(\frac{1}{1 + e^{\gamma f(X)+\beta}} \right) \right. \tag{5.7}$$
$$\left. + \left(1 - \frac{Y+1}{2} \right) \log \left(1 - \frac{1}{1 + e^{\gamma f(X)+\beta}} \right) \right].$$

5.4 Empirical Evaluation

We empirically evaluated the proposed data driven model on a set of 14988 APKs, half of which (i.e., 7494) are malware samples, while the remaining 7494 are legal apps downloaded from the Google Play Store (GOO). We define the APKs in the latter set as *legal* as all of them have been previously analyzed through Virus Total (VIR) without being recognized as malware by any of its 59 different antivirus engines; therefore, it is reasonable to assume that they are not malware. The malware APKs have been downloaded from the AndroZoo dataset that contains officially recognized malware. Each entry in such dataset contains information about the source app market and the Virus Total scan result. We took into consideration any APK that has been recognized as malware by at least 30 engines in Virus Total. As previously pointed out, we considered as features the required permissions, the AAPI invoked in the app, and a combination of both. Regarding AAPI, we were not interested in building the chain of invocations in the app that leads to invoke the specific AAPI as discussed in Section 5.2. On the contrary, we were only interested in determining whether a specific AAPI is invoked somewhere in the app code, independently from how or when it is really invoked. Thus, we parsed the *AndroidManifest* file (i.e., the file that contains, among others, all the permissions required by the app) to extract the required permissions, and the DEX files to retrieve the AAPI invocations, thereby building, for each app A, the set P_A of required permissions, and the set I_A, of AAPI invocations.

It is worth pointing out that app developers can define their own custom permissions. Other apps declaring a custom permission can access to the specific functionality provided by the app defining it. This often happens for apps developed by the same developer. Even if there is a known attack (CUSb) (CUSa) that exploits a vulnerability in custom permission, we disregard them from our analysis because the exploited vulnerability has been fixed since Android 5, allowing only apps signed with the same signing key to define the same `<permission>` element, and the malware in our dataset very rarely use or define

custom permissions. For these reasons, we considered only the official Android permissions (ANDa), thereby discarding all custom permissions defined by apps.

Therefore, given ϕ_{And} as the set of all Android permissions, and an app A, we have that for each $p \in P_A$ then $p \in \phi_{And}$ and $P_A \subseteq \phi_{And}$. Regarding the extraction of AAPI invocations, we adopted *dexlib2* (DEX), a library that sequentially analyzes the bytecode and build up an abstract representation of the code. From this representation, we extracted all the AAPI invocations. In this respect, it is worth mentioning that we ignore method *overloading* for AAPI methods. In a nutshell, method overloading in Java is the possibility to have different methods in a class having the same name, as long as their arguments list is different. In our extraction, we consider overloaded AAPI invocations as semantically equivalent. This means that we consider $i \in I$ as the concatenation of the class and the method name even if there are multiple entries with the same method name in the class.

Experimental Results. We discuss the results achieved by applying the techniques proposed in Section 5.3 to the problem described in Section 5.2, based on the data described in Section 5.4.

In particular two approaches have been compared:

- LIN: the linear learning algorithm proposed in Problem (5.2);
- KER: the non linear learning algorithm proposed in Problem (5.4);

For what concerns LIN, we set $\mathcal{H} = \{\lambda, C\}$ and $\mathfrak{H} = \{10^{-4.0}, 10^{-3.8}, \cdots, 10^{0}\} \times \{10^{-4.0}, 10^{-3.8}, \cdots, 10^{3.0}\}$ while for KER we set $\mathcal{H} = \{\lambda, C, \gamma\}$ and $\mathfrak{H} = \{10^{-4.0}, 10^{-3.5}, \cdots, 10^{0}\} \times \{10^{-4.0}, 10^{-3.5}, \cdots, 10^{3.0}\} \times \{10^{-4.0}, 10^{-3.5}, \cdots, 10^{3.0}\}$ and $n_o = 10^3$. For what concerns (γ, β), the best solution is searched on the following grid $\{\pm 10^{-6.0}, \pm 10^{-5.9}, \cdots, 10^{6}\} \times \{\pm 10^{-6.0}, \pm 10^{-5.9}, \cdots, 10^{6}\}$.

Moreover, the three scenarios discussed in Sec. 5.4 have been investigated, namely:

- PER: where a classifier is built just based on the features related to the required permissions (i.e., $\{P_A\}$);
- INV: where a classifier is built just based on the features related to the AAPI invocations (i.e., $\{I_A\}$);
- PERINV: where a classifier is built based both on the features related to the declared permissions and the ones related to the AAPI invocations.

We split the $s = 14988$ samples in $\mathcal{D}_n$ and $\mathcal{T}_m$ such that $n + m = s$ and $\mathcal{D}_n \cap \mathcal{T}_m = \emptyset$ and $n \in \{750, 1500, 3000, 6000, 12000\}$. Experiments have been repeated 30 times in order to obtain statistically relevant results.

In Table 5.2 we reported the $\widehat{L}^{\mathcal{T}_m}(\mathscr{A}_{\mathcal{H}^*}(\mathcal{D}_n))$ of LIN and KER for problem PER, INV and PERINV when varying n. Based on the results reported in Table 5.2, it is possible derive some observation.

The first one is that the larger is the training set (the more app we use for training the model) the more effective the resulting model is. Moreover, in general, the KER is more powerful than LIN. As expected, the more information we provide to the learning algorithm the more effective the resulting model will be. In particular, the AAPI invocations have more predictive power with respect to the permissions and together they have even more predictive performance. Surprisingly, the difference of the two best performing models, LIN and KER with PERINV, is not statistically relevant (the two distributions of the errors cannot be distinguished with a t-test). Therefore, we chose the LIN model that is more suitable to be deployed on a smartphone device as it requires less computational resources in comparison to KER.

In Table 5.3 we reported the confusion matrices (in %) of the best performing models, namely LIN and KER for PERINV when n = 12000. From Table 5.3 it is possible to observe that the false positive and false negative rate is quite balanced in both models, thereby indicating that the models have a high quality. Furthermore, for the sake of completeness we also provided some indexes of performance in Table 5.4.

In Table 5.5 the confusion matrices (in %) of LIN and KER for PERINV and n = 12000 are reported. These matrices take into account also a *warning* class that represents the case when an app is classified as malware with a probability between 30% and 70%. In this case, the decision is left to the user; such alternative allows to remarkably reduce the number of false positives and false negatives at the expenses of letting the user decide in critical cases.

In Table 5.6 the *Top* 20 permissions and AAPI invocations, together with their raw importance (see Section 5.3), of LIN are reported. We consider only LIN as it is the only one that can provide such information, for PER, INV, and PERINV with n = 12000. From Table 5.6 it is possible to observe that a small amount of permissions and AAPI invocations have high importance for predicting the presence of a malware (strongly positive raw importance). Contrarily, a large amount of them have small importance for predicting the absence of a malware (weakly negative raw importance). This is reasonable since some permissions and AAPI invocations are a sort of strong indicator for a malware while the presence of many other permissions and AAPI invocations show that the app is performing common legal tasks. This underlines that innocuous permissions and AAPI invocations have high importance for predicting the absence of a malware with data-driven techniques while conventional approaches just search for malware behaviors. In general, the analysis of results suggests that the main goal of malware is to collect as much information as possible about the user and the phone, as well as getting access to the SMS service (i.e., to force the user to subscribe to some payment services).

Table 5.2: $\widehat{L}^{\mathcal{T}_m}(\mathscr{A}_{\mathcal{H}^*}(\mathcal{D}_n))$ of LIN and KER for problem PER, INV and PERINV when varying n.

n	PER		INV		PERINV	
	LIN	KER	LIN	KER	LIN	KER
750	12.6 ± 0.9	11.9 ± 0.8	5.3 ± 0.3	5.4 ± 0.3	5.1 ± 0.2	5.2 ± 0.2
1500	12.4 ± 0.8	10.5 ± 0.8	4.0 ± 0.3	4.1 ± 0.3	4.0 ± 0.2	4.0 ± 0.2
3000	12.3 ± 0.8	10.9 ± 0.8	3.4 ± 0.3	3.4 ± 0.3	2.9 ± 0.2	3.0 ± 0.2
6000	12.0 ± 0.8	10.2 ± 0.7	3.2 ± 0.2	2.9 ± 0.2	2.2 ± 0.1	1.7 ± 0.2
12000	11.7 ± 0.7	9.2 ± 0.6	2.5 ± 0.1	2.2 ± 0.2	1.1 ± 0.1	1.0 ± 0.2

Table 5.3: Confusion matrices (in %) of LIN and KER for PERINV when n = 12000.

		Truth	
	LIN	Legal	Malware
Prediction	Legal	49.4 ± 0.1	0.5 ± 0.1
	Malware	0.6 ± 0.1	49.5 ± 0.1

		Truth	
	KER	Legal	Malware
Prediction	Legal	49.5 ± 0.2	0.5 ± 0.1
	Malware	0.5 ± 0.1	49.5 ± 0.2

Table 5.4: Indexes of performance (in %) of LIN and KER for PERINV when n = 12000.

Index of Performance	LIN	KER
sensitivity or true positive rate	0.988 ± 0.001	0.990 ± 0.001
specificity or true negative rate	0.990 ± 0.001	0.990 ± 0.001
precision or positive predictive value	0.990 ± 0.001	0.990 ± 0.001
negative predictive value	0.988 ± 0.001	0.988 ± 0.001
false negative rate	0.012 ± 0.001	0.012 ± 0.001
fall-out or false positive rate	0.010 ± 0.001	0.010 ± 0.001
false discovery rate	0.010 ± 0.001	0.010 ± 0.001
false omission rate	0.012 ± 0.001	0.012 ± 0.001
accuracy	0.989 ± 0.001	0.990 ± 0.001
F1 score	0.989 ± 0.001	0.990 ± 0.001
Matthews correlation coefficient	0.978 ± 0.001	0.978 ± 0.001
informedness	0.978 ± 0.001	0.980 ± 0.001
markedness	0.978 ± 0.001	0.978 ± 0.001

Table 5.5: Confusion matrices (in %) of LIN and KER for PERINV when n = 12000 when the Warning class is introduced (apps classified with probability of being a Malware greater than 30% and less then 70%).

		Truth	
	LIN	Legal	Malware
Prediction	Legal	49.0 ± 0.1	0.2 ± 0.1
	Warning	0.3 ± 0.1	0.3 ± 0.1
	Malware	0.3 ± 0.1	49.1 ± 0.1

		Truth	
	KER	Legal	Malware
Prediction	Legal	49.1 ± 0.1	0.3 ± 0.1
	Warning	0.2 ± 0.1	0.2 ± 0.1
	Malware	0.3 ± 0.1	49.1 ± 0.1

Table 5.6: Top 20 permission and retrieved API invocations of LIN for PER, INV, and PERINV with n = 12000.

PER	
Raw Importance	Permission
1.00	android.permission.SEND_SMS
0.89	android.permission.READ_PHONE_STATE
−0.80	android.permission.ACCESS_NETWORK_STATE
0.75	com.android.launcher.permission.UNINSTALL_SHORTCUT
0.73	android.permission.CHANGE_WIFI_STATE
0.61	android.permission.READ_SMS
0.59	android.permission.WRITE_APN_SETTINGS
0.58	android.permission.DELETE_PACKAGES
−0.53	android.permission.READ_CALL_LOG
−0.51	android.permission.MODIFY_AUDIO_SETTINGS
0.51	android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
0.42	android.permission.WRITE_CALENDAR
−0.40	android.permission.READ_EXTERNAL_STORAGE
0.39	com.android.launcher.permission.INSTALL_SHORTCUT
0.38	android.permission.READ_LOGS
−0.38	android.permission.PACKAGE_USAGE_STATS
0.37	android.permission.RECEIVE_BOOT_COMPLETED
−0.37	android.permission.GET_ACCOUNTS
−0.35	android.permission.DISABLE_KEYGUARD
0.33	android.permission.STATUS_BAR
INV	
Raw Importance	Retrieved AAPI invocation
1.00	android.telephony.SmsManager->getDefault
0.56	android.content.BroadcastReceiver-><init>
0.51	android.app.admin.DeviceAdminReceiver-><init>
0.49	android.telephony.TelephonyManager->getDeviceId
0.45	android.telephony.TelephonyManager->getLine1Number
0.42	android.telephony.gsm.SmsManager->getDefault
0.42	java.lang.String-><init>
0.39	java.io.InputStreamReader-><init>
0.39	java.lang.reflect.Field->get
0.37	android.app.admin.DevicePolicyManager->isAdminActive
−0.36	android.content.Context->getPackageName
0.36	android.app.Application->attachBaseContext
0.36	android.app.ActivityManager->getRunningServices
0.35	android.app.PendingIntent->getBroadcast
−0.35	android.content.Intent-><init>
−0.35	java.lang.String->format
0.35	java.lang.String->valueOf
−0.33	android.content.Context->getSystemService
0.32	android.content.Context->getDir
0.32	android.os.Bundle->get
PERINV	
Raw Importance	Permission or Retrieved API invocation
1.00	android.permission.SEND_SMS
0.46	android.telephony.SmsManager->getDefault
0.44	android.content.BroadcastReceiver-><init>
0.42	android.app.Application->attachBaseContext
0.40	android.app.admin.DeviceAdminReceiver-><init>
−0.39	android.permission.ACCESS_NETWORK_STATE
0.37	android.telephony.TelephonyManager->getDeviceId
0.37	java.io.InputStreamReader-><init>
0.34	android.telephony.TelephonyManager->getLine1Number
−0.32	android.content.Context->getPackageName
0.30	java.lang.String-><init>
0.28	java.lang.String->valueOf
0.28	java.lang.reflect.Field->get
−0.28	java.lang.String->format
0.28	java.io.FileOutputStream->write
0.28	android.app.admin.DevicePolicyManager->isAdminActive
−0.28	android.content.Context->getSystemService
0.27	android.webkit.WebView->setDownloadListener
0.27	android.permission.RECEIVE_SMS
−0.27	java.util.Iterator->next

5.5 Concluding Remarks

We have presented a machine learning-based technique that focuses on the identification of malware in resource-constrained devices such as Android smartphones. Our technique has a very low resource footprint and does not rely on resources outside the protected device. Sustainability in the field of computing must not interfere with security. Hence, it is crucial that security systems and related measures are designed from the very start to be sustainable and compatible with the resource constraints of the target platform. This is important in the perspective of greening computing and networking but is paramount in the world of mobile devices and IoT, where resources in general and energy, in particular, represent tough constraints. The technique is at the basis of BadDroids, an Android app focused on early identification of malware, more in detail, directly at installation time, without heavily impacting the usability and the battery life of the mobile device. We adopted a data-driven approach capable of achieving a high level of accuracy in malware identification based on a set of features easily inferable from apps through static analysis techniques.

To validate our methodology we have implemented BadDroids (bad17) (see Fig. 5.1), and we have tested it on almost fifteen thousand different apps half of which were malware (Fig. 5.2 and Fig. 5.3 show an example of malware and non-malware analysis results). BadDroids showed an accuracy level equal to 98.9%, more in details 0.6% false positives and 0.5% false negatives. The complete dataset as well as further information on BadDroids are available at `http://baddroids.smartlab.ws`. The dataset has been also submitted to UCI (Lic).

Furthermore, to ensure that the usage of the tool on a mobile device was neither disruptive to the user experience, nor incompatible with less powerful devices, we tested it on a dated device, namely an LG Nexus 5. This device was released in 2013, runs Android 6.0.1, has a Qualcomm MSM8974 Snapdragon 800 CPU running at 2.3GHz, and 2GB of RAM.

We randomly chose 1000 APKs from our dataset and, since BadDroids starts whenever an app is installed or updated, we have installed them on our device logging the size of the DEX file and the time needed for the analysis. We obtained that the average DEX file size is 5539 KiB and the average time for analyzing an APK is 64474 ms.

It is worth noting that the specifications of our test device can be considered comparable with a mid-range mobile device of the current generation, thus BadDroids does not need a top-notch device to be actually used and on such a configuration and it requires a minute, on average, to analyze an app.

While the results in terms of accuracy are remarkable, the time required to perform the complete analysis is still clearly perceivable by the user, hence, in future work, we need to optimize the process to a further extent. Moreover, while the feature set adopted in this

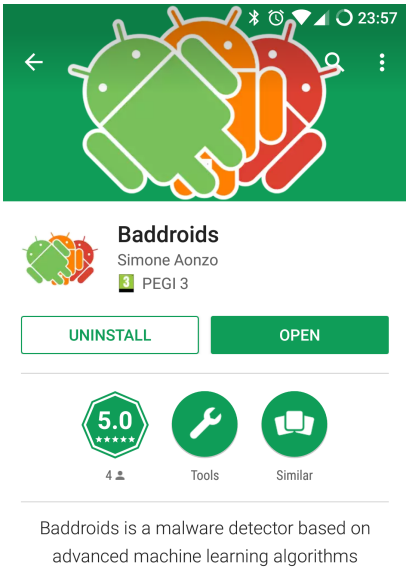


Figure 5.1: BadDroids on the Google Play Store

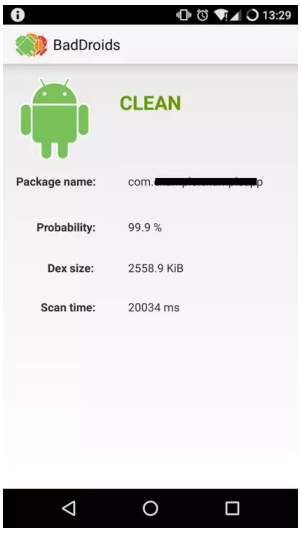


Figure 5.2: Clean classification

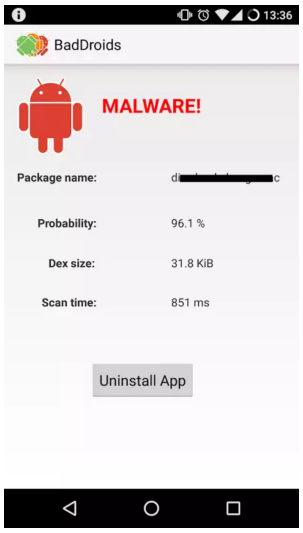


Figure 5.3: Malware classification

work has shown very good properties in terms of accuracy of the prediction, we need to verify its resilience to obsolescence and we also need to explore the possibility to adopt more sophisticated properties of the apps as independent features, like the interaction with other apps through intents, as well as the usage of Reflection and JNI.

Chapter 6

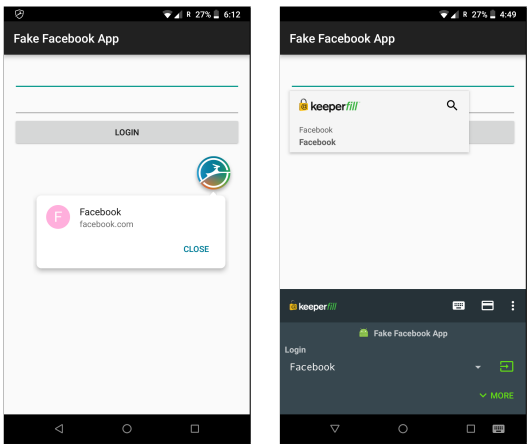
From New Features to a Phishing Attack

6.1 Introduction

Recent reports have shown that more than half worldwide website traffic has been generated via mobile devices (Sta18). Users take advantage of these devices not only to browse websites, but also to access social networks and other online services, such as online banking. Thus, to improve user experience, developers of web services often implement native Android apps, making mobile devices portals to their associated web backends. For example, a vast portion of Facebook accesses in the US is performed via mobile device (Dog18). According to these reports, this trend is forecasted to only increase in the future, and users are going to perform more and more often one of the most basic security-sensitive action: authenticate to mobile apps backends by inserting their credentials. On the one hand, this shift towards the mobile world pushed Google and platform developers to design new technologies and mechanisms to decrease the friction of these user interactions. On the other hand, unfortunately, the more frequently users will be asked to insert credentials on their mobile devices, the more attackers will find mobile phishing attacks rewarding.

In this chapter, we take a look at new features introduced in modern versions of Android, and we show that while they do simplify both users' and developers' lives, their weak design and implementation allow attackers to abuse them, making mobile phishing attacks significantly more practical than what previously thought.

Mobile password managers. The first aspect we look at is the growing popularity of *mobile password managers*. Password managers have been initially developed for the web, and the security community has long praised their many benefits. For example, they



(a) (b)

Figure 6.1: Android password managers (1a) Dashlane and (1b) Keeper, suggesting Facebook credentials to a fake malicious app.

provide a practical way for users to use different pseudo-random passwords for each web service they interact with, thus discouraging the use of simple, easy-to-guess, and shared passwords across accounts. In fact, the user has a chance to store her credentials and to associate them to specific websites: when the user later navigates to the same website, the password manager identifies the website through its domain name, and it then suggests (and in some cases automatically fills) the right credentials on behalf of the user.

As a way to support the increasing amount of mobile users, password managers are now also available for mobile devices. Mobile password managers are developed as apps, and they include advanced sync features, which allow suggesting (and filling) website-related credentials to their associated apps.

From a technical standpoint, these password manager apps either need to have support from the Android Framework, or they require modifications to their potential "clients" (e.g., the Facebook app). In fact, Android apps sandboxing mechanism prevents them to interact with external apps programmatically. To date, there are three mechanisms that act as necessary basic blocks to allow for their implementation. The first is the Accessibility Service (Goo18a) (a11y, in short): while, in theory, a11y is a mechanism to allow apps to be "accessible" to users with disabilities, it also allows apps to interact with others programmatically, and it thus provides the technical capability needed by password managers to implement their functionality. Since recent works have shown how a11y can be abused (FQCL17; JSC+14; Ami16b; Ami16a; Loo15; Lui16; Ven16), Google has recently implemented the Autofill Framework (Goo18b), a new component of the Android Framework specifically developed to allow password managers to suggest and autofill credentials to mobile apps (without the need to rely on a11y). The third mechanism is called

OpenYOLO, a recently-proposed protocol for storing and updating credentials for mobile apps (Goo17b). This mechanism is developed by Google and Dashlane, and it follows a different paradigm: it does not affect the Android Framework, but it requires modifications of each "client" (e.g., Facebook) and "server" app (e.g., the password manager).

In this chapter, we show that all these three mechanisms are affected by design and implementation issues. At the root of the problems is the need to *bridge the mobile world with the web world*: given an app with a login form, how can a password manager know whether this app is the legitimate Facebook app (and it is thus entitled to access Facebook credentials) or whether this is a malicious app attempting to appear as the legitimate one? How is it possible to know which app is linked to which domain name? The key design issue is that *all these three mechanisms use the app package name as the main abstraction to identify an app.* Password managers thus need to somehow *map* package names to associated websites.

While a technical solution to securely implement such mapping exists, this work shows that the poor design choices of the underlying mechanisms push to the implementation of vulnerable solutions. In particular, we have investigated the four leading third-party mobile password managers app (Keeper (Goo18f), Dashlane (das18), LastPass (las18), 1Password (1pa18)), as well as Google Smart Lock (GSL) (Goo18d): we have found that only GSL is securely implemented. Moreover, we have found that Keeper, Dashlane, and LastPass all implement various (vulnerable) heuristics, each of which misplaces trust in an app package name or other metadata. The net result is that *it is possible for a malicious app to systematically lure these password managers to leak credentials associated with arbitrary attacker-chosen websites.* To make it worse, we note that these attacks also work for websites for which an associated mobile app does not exist. These attacks effectively make mobile phishing more practical: differently than all previous works, the user is not even asked to type her credentials; the user is just asked to allow password managers to autofill the credentials on her behalf.

It is interesting to note how, on the web, password managers do not ease phishing attacks, but quite the opposite. In fact, web password managers check the current website domain name to determine whether to auto-fill (or auto-suggest) credentials: if the domain name does not match the expectations, no credentials are suggested. Thus, an attacker that uses particular Unicode characters to create a *facebook.com*-looking domain name may fool a human, but not a password manager: the malicious domain name will be different from the legitimate one, and the password manager suggestion will not trigger. We thus argue that the mere fact that a mobile password manager is suggesting credentials associated with the target website inherently adds legitimacy to the attack, making it even more effective.

Instant Apps. The second modern feature we explore in this chapter is called *Instant Apps*. This technology, implemented by Google, allows users to "try" Android apps at the touch of a click, without the need to fully install the app. Under the hood, the

system works by asking developers to upload small portions of their Android app, called Instant Apps, and to associate a URL pattern to it. The developer needs first to prove that she controls the domain name of the URL pattern. This is carried out through a multi-step procedure called App Link Verification (Goo18i) which relies on Digital Asset Links (Goo17a) protocol (it makes possible to associate an app with a website and vice versa, via verifiable statements). After this deployment step, the user will be able to click on a link (pointing to the specified URL), and, after a one-time confirmation, the Instant App is automatically downloaded and executed on the user's device.

In this chapter, we show that this technology, while indeed a very useful Android feature, can make phishing attacks more practical. The key observation is that *Instant Apps provide an attacker the ability to gain full control over the device UI, without the need of installing an app.* In a browser-only phishing scenario, the user would have a chance to notice the green lock and inspect the domain name. However, in an Instant Apps-based attack, the attacker has full capabilities to deceive the user. For example, an attacker could create a full-screen Facebook login view (as the real Facebook app would do). As reported in existing works (CQM14; BCI+15; RZX+15; AP17), users cannot distinguish between these. As another example, an attacker could simulate the view of a full browser; as the attacker controls every pixel of the screen, nothing prevents her to show the user a browser-like view with a spoofed *facebook.com* domain name and a green lock: once again, this attack is indistinguishable from a legitimate scenario. As highlighted by several recent works, the key insight is that the UI on mobile devices cannot be trusted, and Instant Apps provide a technical way for an attacker to move from a scenario where she does not fully control it (like a web page somehow constrained by the web browser security mechanisms) to a scenario where she fully does.

End-to-end attack. The combination of flawed mobile password managers and Instant Apps allow attackers to develop and mount mobile phishing attacks that are much more practical than what previously known (CQM14; BCI+15; RZX+15; FQCL17). In fact, we have found that, although Instant Apps are not "fully installed" apps, 1) password managers currently do not notice the difference, and that 2) their package name is the same as the associated full app. This means that the package name of the Instant App is attacker-controlled, and that *it is thus possible to trick password managers to auto-fill credentials for an attacker-chosen website even without requiring the installation of an additional app.* This allows an attacker to bootstrap an end-to-end phishing attack by luring the victim into visiting a malicious webpage: such webpage may contain, for example, a fake Facebook-related functionality. Upon clicking on it, the Instant App mechanism is triggered, the attacker can spoof a full-screen Facebook login form, at which point the password manager would offer to automatically fill the credentials on behalf of the victim.

To make things worse, we found that *current password managers fill hidden fields as well.* An attacker could thus create a form with a visible username field but a hidden password

field: while the unsuspecting user thinks she is autofilling only the username, her password manager will silently leak her password to the attacker.

To the best of our knowledge, the attacks presented in this chapter are the most advanced and practical phishing attack techniques to date. In fact, all existing approaches assume a malicious app installed on the user's device, ask the user to manually insert her credentials (which although not technically problematic, may reduce the attack success rate), or fall back to web-based phishing attacks (that are noticeable at least from the browser bar) (CQM14; BCI+15; RZX+15; FQCL17).

A look to the future. The future of these problems does not look encouraging. The current API has a design that is error-prone and does not force developers to take all necessary steps to avoid severe vulnerabilities. In this chapter we discuss the design of a new API, called *getVerifiedDomainNames()*, that *uses domain names as the main abstraction level* (instead of package names, which should not be trusted), and it hides behind a single, central implementation the necessary logic and security steps to establish that a requesting app does have authority over the credentials it is requesting. Internally, this new API relies on an existing technical solution based on Digital Asset Links (Goo17a) verification. This solution requires websites owners to publish an "assets" file on their website so that an app-website "link" can be established.[1] This is the same mechanism that Autofill Framework and OpenYOLO *suggest* developers to use: the difference is that our API *forces* them to use it, instead of leaving them open to implement vulnerable solutions—as they did.

Unfortunately, although we believe that this solution is technically sound, the current ecosystem is far from being ready. In fact, the App Link Verification requires collaboration from websites owners, as they would need to upload the appropriate assets file to their website. To determine the readiness of the ecosystem to this mechanism, we first built a dataset of 8,821 domain names extracted from the password managers we have analyzed (given the source of this dataset, these domain names are guaranteed to have at least one login form, otherwise they would not be relevant to password managers). We then checked how many websites already link themselves to an app: to our surprise, only 178 of them currently have an *assetlinks.json* compatible with the proposed solution, which is around 2%. This means that, to date, *password managers developers do not have the necessary information to securely implement their functionality, even if they wanted to*. One may then wonder how Google Smart Lock, which we found to be secure, implements such mapping. We found that, although a technical solution exists, this process is not automatic: according to the official documentation (Goo18e), the last step of the process requires developers to manually fill a Google Form (Goo18h) to provide the needed information. We conclude that the adoption of a secure mapping cannot be easily addressed by the single actors alone, but it requires a community-wide effort, which this work hopes to inspire.

[1] *Such "assets" file needs to be placed at a specific location: https://domain.name/.well-known/assetlinks.json*

In summary, this chapter provides the following contributions:

- We performed the first security analysis of mobile password managers and the three core technologies they rely on: a11y, Autofill Framework, and OpenYOLO; we have uncovered design and implementation issues that allow attackers to trick password managers to leak to malicious apps credentials associated to arbitrary attacker-chosen websites;
- We show how Instant Apps can be abused to gain full UI control and how they can be used to lower the bar for stealthy and practical phishing attacks;
- We present an end-to-end phishing attack that abuses password managers and Instant Apps, and we show that current implementations automatically fill hidden password fields. We believe this to be the most advanced and practical phishing attack to date;
- We propose a new secure-by-design API that moves the abstraction from package names to domain names;
- We provide empirical evidence that the current ecosystem is not ready yet to support secure autofill on mobile devices, and that a community-wide effort is required to address these issues.

6.2 Android Password Managers

A password manager (PM from now on) is a tool that stores and manages user's credentials like usernames and passwords. PMs aim to suggest to the user the right credentials to insert in login forms, thereby leveraging the same user from the burden of memorizing their sensitive data.

PMs have been originally conceived for the web domain and mostly implemented as browser extensions. They work as follows: the first time a user visits a website and inputs credentials in online forms, the PM stores such credentials on its backend and it maintains the association between the credentials and the domain name. When the user visits the same domain later on, the PM recognizes and verifies the domain, and it suggests the credentials to insert in the corresponding login form.

The increasing popularity of mobile apps acting as wrappers of their corresponding websites (e.g., email providers, online documents, social networks, home banking) has motivated the development of password managers for mobile devices. These are implemented as mobile apps, and they have the capability of helping managing and automatically filling user's credentials in other apps. Modern PM apps and browser extensions also provide advanced

sync functionalities between app and website credentials. For example, consider a user opening for the first time the Facebook app, which requires the users credentials: at this point, the PM identifies the app, determines which domain name this app is associated to (i.e., *facebook.com*), and checks whether it has credentials associated to it; if this is the case, it auto-suggests them to the user, who can thus authenticate herself with few clicks, without the need of manually inserting her credentials. Figure 6.1 shows two examples of password managers auto-suggesting credentials.

From the technical standpoint, filling credentials requires proper mechanisms allowing PMs to access the UI of other apps, thereby bypassing the isolation provided by the sandbox. To this end, modern Android versions offer three mechanisms to support the implementation of PMs apps: *Accessibility Service*, *Autofill Framework*, and *OpenYOLO*.

Accessibility Service. The Accessibility Service, a11y in short, is a framework that allows third-party apps to be accessible to users with disabilities (Goo18a). An app can make use of this framework by requesting the `BIND_ACCESSIBILITY_SERVICE` permission and by implementing a component that, while in the background, receives callbacks by the system when "Accessibility Events" are fired. These events are related to some specific transitions on the user interface, e.g., the focus is changed or a button has been clicked. This service has also access to relevant contextual information, the most important being which app the user is currently interacting with. This last information is made available by means of the package name of the app.

Even if a11y has been developed to assist users with disabilities, app developers have (benignly) abused this framework to implement a variety of different features, one of which is the implementation of password managers. In particular, PMs rely on a11y to determine which app the user is interacting with and whether there are text fields that could be filled with stored credentials; if that is the case, the PM then relies once again on a11y to programmatically interact with the target app and automatically fill the credentials fields on behalf of the user.

Unfortunately, while a11y is certainly useful, in the past few years there have been a number of research works from the industry and academic communities that show how a11y can be abused to perform a number of malicious functionality, from stealing user's personal information to the complete compromise of the device (FQCL17; JSC+14; Ami16b; Ami16a; Loo15; Lui16; Ven16). Due to these threats, Google has developed additional Android features so that apps do not need to have access to such powerful mechanism to implement their functionality. Since password managers are some of the most common and prominent use cases, Google has recently introduced the *Autofill Framework*.

Autofill Framework. The Autofill Framework (Goo18b) has been introduced in Android Oreo. This framework offers to password managers apps a technical solution to implement their core functionality without requiring access to a11y. In particular, the Autofill Frame-

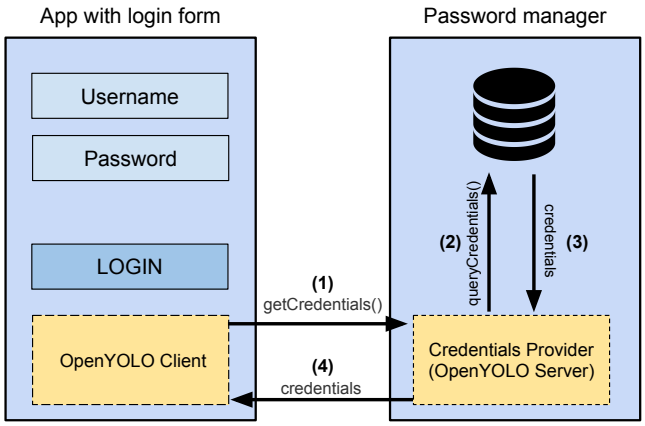


Figure 6.2: Deployment and workflow of OpenYOLO. We note that the interaction between the client and server is actually implemented via the Intent mechanism.

work allows an app to 1) determine which app the user is interacting with, and 2) fill credential fields programmatically.

The Autofill framework requires the developer to create an app that implements an *Autofill Service*, which allows filling out forms by injecting data directly into the views, such as the `EditText` widgets that store the credentials. To use that, the app needs to require the `BIND_AUTOFILL_SERVICE` permission. Android Oreo has also introduced some new XML attributes to assist password managers: `importantForAutofill`, which specifies whether the view is *autofillable*, `autofillHints`, which is a list of strings that suggests to the service what data to fill the view with, and `autofillType`, which tells the Autofill Service the type of data it expects to receive. Through these attributes, an app implementing an Autofill service is able to detect, classify, and fill form fields according to their types (e.g., username, email address, password). Note that an app that wants to be "compatible" with the Autofill Framework *must* use these XML attributes. Note also that only one Autofill service can be active at the same time (the user can select which one to use through a dedicated setting menu).

At run-time, when the user opens a supported app with a login form, the password manager is able to determine which app the user is interacting with (once again, through its package name) and it can offer the possibility to the user to automatically insert the corresponding credentials on her behalf.

OpenYOLO. OpenYOLO (YOLO stands for "You Only Login Once") is a recently developed protocol, supported by Google partnering with Dashlane, and it is available as an open-source library (Goo17b). OpenYOLO does not require neither a11y nor Autofill Framework, but it requires to modify each app that wants to support OpenYOLO-based PMs. This mechanism is constituted by two components: the *client* and the *credential*

provider (the server). The client is a component that needs to be embedded in each app that wants to support this protocol (e.g., Facebook). The credential provider, instead, is used within the password manager itself, and it is in charge of providing information to the password manager about the requester app identity. At run-time, the client seamlessly interacts with the credential provider (via the Intent mechanism), which, with the cooperation of the password manager, then returns to the client a set of credentials, if available. The interaction between the two components is depicted in Figure 6.2.

Note that OpenYOLO only helps PMs to interact with the target app. However, the implementation logic in charge of retrieving the correct credentials is left to the PM developers. In particular, the OpenYOLO credential provider exposes to the password manager the package name and the signature of the app requesting credentials. Once again, the PM is in charge of mapping the given package name to the appropriate domain names and credentials.

The central role of package names. Independently from which mechanism a password manager is relying on, the key information to identify which app the user is interacting with is the app *package name*. Unfortunately, in all these cases, the developers of the PM are left with the responsibility of securely mapping package names and domain names. As we will discuss in the rest of this chapter, this design choice has a severe negative impact on the security of password managers and of the entire ecosystem. In fact, while mobile password managers have access to package names (and thus apps), the user's credentials they manage are related to websites. And this begs the question: "*how do mobile password managers actually link apps to their respective websites?*"

6.3 Web and Mobile Apps Worlds

The three mechanisms discussed in the previous section allow PMs to feed website-related credentials to the corresponding mobile app counterparts. To work properly, a PM needs 1) to identify the app that requires credentials and 2) to bridge the mobile and the web worlds. Since all the available mechanisms use apps package names as the main abstraction, in order to determine the right credentials to suggest, PMs need to somehow define a mapping between these package names and their corresponding website. We argue that *package names are the wrong abstraction for PMs to work with.* This section discusses the many pitfalls associated with this process, and how it is likely to misplace trust in these package names.

6.3.1 The Mapping Problem

PMs have access to package names as the key information to identify apps and to determine whether to automatically suggest credentials and for which website. Given a package name, PMs need to bridge the gap between the mobile apps and the web worlds. There is thus the need of *mapping* package names to their associated web domain names.

One of the problems is that package names resemble URLs (e.g., the package name of the official Facebook app is *com.facebook.katana*), thereby suggesting to inexperienced Android developers the same level of trustworthiness of the associated domain name, *facebook.com*. As we will see later in this chapter, even developers of leading PMs severely misplace trust in package names, thus affecting the security of PMs and the entire ecosystem by making mobile phishing attacks more practical. We now discuss the main characteristics of domain names, package names, and the relation between them.

Domain names are trusted. In the modern web, domain names can be considered as trusted. With the wide adoption of robust DNS services and HTTPS, users and developers can determine whether they are securely visiting a given URL: the browser would verify the identity of the domain name by means of the PKI and the digital certificates ecosystem. Thus, web PMs do rightfully place trust in domain names. For example, a PM will automatically suggest Facebook's credentials whenever the user browses to *facebook.com*. Notably, PMs do *not* suggest Facebook credentials when the user visits a different domain name.

No authentication of package names. Differently than domain names, there is no authentication of package names. Anybody can create an app with a given package name, and it is possible for an attacker to create an app with the same package name of, for example, the legitimate Facebook app. However, one constraint must always be satisfied: there cannot be two apps with the same package name published on the Google Play Store or installed on the same device. In other words, package names act as unique keys. Note that third-party markets are not as controlled, and it may be possible to publish malicious apps with package names of legitimate apps. However, depending on the specific victim, it may be challenging to lure her to install such malicious apps from third-party stores.

No authority on "sub-packages." In the world of domain names, owners of the *example.com* are in control of sub-domains as well. In the world of package names, instead, this is not the case: the owner of *com.example* package name does not have any control over package names that may appear as "sub-packages." For example, nothing prevents anybody to create an app with package name *com.example.evil*: there is no relation between them. Thus, the sub-domain trustworthiness of the web world does not hold in the mobile counterpart. Unfortunately, as we will discuss later in the chapter, this false sense of safety is a key cause of security issues among PMs.

The mapping problem. In the vast majority of cases, credentials are associated to websites, not to mobile apps: in fact, credentials are generally used to authenticate to a web service backend, not to a mobile app. Thus, given an app package name, PMs need to answer the question "*which website is this package name associated to?*". This is not a trivial question to answer. To make things worse, PMs developers are left to implement their own "solution". Unfortunately, there are many pitfalls in implementing this mechanism, and we found that even leading PMs opted to rely on heuristics to solve this problem. It turns out that *most of these heuristics are vulnerable, and malicious apps can trick PMs to leak credentials associated to arbitrary websites.*

6.3.2 Attacker Practicality Aspects

From an attacker perspective, there are several aspects that would make a phishing attack more or less practical. In this section, we enumerate some questions related to the attacker capabilities. We will put them in relation to each vulnerable mapping in the next subsection.

Q1) Is the mapping vulnerable? The first question is, of course, about whether the mapping is vulnerable or not. We consider a mapping as *vulnerable* if an attacker can create an app that, although not being the legitimate one, can trick PMs into auto-suggesting credentials associated to a given website.

Q2) Can the legitimate and malicious apps co-exist? One of the most basic attack vectors is for a malicious app to have the same package name as the legitimate one. Since no two apps installed on the same device can have the same package name, this implies that, in this scenario, the legitimate and the malicious app cannot co-exist. This, in turn, implies that an attacker exploiting this package name-colliding technique would need to first lure the user to uninstall the legitimate app before the attack can be performed. Of course, this poses practicality issues. Thus, this question is about: can an attacker bypass this constraint? In other words, to give an example, can an attacker create a malicious app that can co-exist with the legitimate Facebook app and that, when opened, would trick PMs to auto-suggest the legitimate Facebook credentials?

Q3) Can the malicious app be hosted on the Play Store? In the general case, it is more difficult to lure the user to install an app that is not hosted on the Play Store. Thus, one relevant question is: is it possible for an attacker to upload her malicious app to the Play Store? The main constraint for an attacker is that no two apps with the same package name can be hosted on the Play Store at the same time. In other words, this question asks whether an attack requires creating an app with the same package name of an already-existing app on the Play Store. If yes, the only venue for the attacker is to

Table 6.1: This table systematizes vulnerable mapping implementations and puts them in relation with attacker practicality aspects.

	Q1	**Q2**	**Q3**	**Q4**
Secure mapping				
Static one-to-one mapping	✓			✓
Static many-to-one mapping	✓	✓		✓
Crowdsourced mapping	✓	✓	✓	✓
Heuristic-based mapping	✓	✓	✓	✓
No mapping (all credentials suggested)	✓	✓	✓	

lure the user to install the malicious app from a third-party market (via the side-loading process): although this attack is possible, it is less practical.

Q4) Can the attacker generate tailored suggestions? PMs have the capability to auto-suggest one or more set of credentials. Then, the user can choose one of them and, at the touch of a click, these credentials are automatically filled in the target app. Now, from an attacker perspective, the ideal situation would be to able to write a malicious app such that, for example, the PM would only suggest the credentials of *facebook.com* (or any other domain name chosen by the attacker). A less-ideal scenario is a PM where all the credentials are always suggested: although the user has the possibility to lure her credentials to the malicious app, this attack would be slightly less practical. Thus, the question is: can the attacker have fine-grained control over which and how many credentials are suggested?

6.3.3 Vulnerable Mappings

This section systematizes the different possible implementations of the package names → web domain names mapping. For each of them, we describe how such implementation is vulnerable, to which attacks, and how practical it is with respect to the questions discussed above. The insights presented in this section are systematized in Table 6.1.

Secure mapping. The safest way to implement a mapping consists in securely verifying whether the developers of the current app have authority over a given domain name: if that is the case, then it is safe to auto-suggest the credentials of such domain name to the current app. One known solution to achieve this mapping is called Digital Asset Links (Goo17a) (DAL from now on). From a conceptual point of view, DAL allows for the definition of authentication domain equivalence classes, and it makes it possible to associate an app with a website and vice versa, via verifiable statements. This mechanism works by asking websites owners to publish on their website an "assets" file that contains a list of apps that can be legitimately associated with it. In this case, each app is identified by its package

name and by the hash of its legitimate signing key. A third-party can then verify that an app is indeed legitimately linked to a website by checking whether the "assets" include a matching package name and the hash of the signing key.

Static one-to-one mapping. Consider a PM with a static one-to-one mapping, which maps one package name to exactly one domain name, and vice versa. As an example, consider the legitimate Facebook app, whose package name is *com.facebook.katana*, which is usually mapped to the *facebook.com* domain name. This simple mapping technique is vulnerable: in fact, Facebook credentials are suggested to any app whose package name is *com.facebook.katana*, even if the app is not the legitimate one. It would be possible to prevent this vulnerability by checking the certificate that signed the target app, and make sure it is one of the known, trusted one. Unfortunately, maintaining such list of known trusted certificates is a very challenging task. We consider this a vulnerability, but the attack is not very practical: in fact, the malicious app cannot co-exist with the legitimate one.

Static many-to-one mapping. Consider a PM with a mapping that maps n different package names $p_1, p_2, \ldots, p_n$ to the same domain name D. This can happen for different apps belonging to the same companies: while they are all different apps (and thus they have different package names), they are all associated with the same domain name. This typology of mapping is problematic because it is frequent that the user would install only *one* (or a subset) of these apps. Thus, a malicious app with one of the remaining package names is able to steal the credentials. This attacker is more practical than the previous one because it does not require the attacker to lure the user to uninstall the legitimate app. However, the package names specified in the mapping likely refer to real legitimate apps on the Play Store. This means that the attacker cannot upload her malicious app on the Play Store (because package names need to be unique across the store), and the app needs to be side-loaded.

Crowdsourced mapping. Given the scale of the problem—millions of apps and website to map one with each other—one possibility to create a comprehensive mapping is by means of crowdsourcing. Thus, one approach is the following: consider a user who inserts credentials for a domain D to an app with package name P, and assume that the given PM did not know about this mapping: in such case, a popup can ask the user whether she allows such association to be shared with other users, so that everybody can benefit. If the user allows for it, this new association is sent to the backend, which, depending on the specific implementation, could immediately make this mapping available to all its users, or wait until a number of users higher than a threshold report the exact same association. If an attacker is able to "inject" a new association, then she can mount an attack that is more practical than the two alternatives above. In fact, she could inject a new mapping $p_{attacker} \rightarrow D$ (where $p_{attacker}$ is an arbitrary attacker-chosen package name): in this way,

the PM would suggest credentials related to D to the malicious app with $p_{attacker}$ as package name. Since the package name is attacker-chosen, the attacker can choose a package name that does not yet exist, and she can upload the malicious app to the Play Store. Of course, this malicious app can also co-exist with the legitimate one, given the different package name.

Heuristic-based mapping. One last way to implement mapping is through heuristics. For example, one way is to infer which is the appropriate domain name by implementing heuristics on the package name of the app. One other strategy is to rely on some other metadata to take such decisions. From a security perspective, this is the most dangerous scenario. In fact, if such heuristics are implemented in a way that an attacker can game them, the attacker could create a malicious app that "maps" to an arbitrary attacker-chosen target. Also, in this case the attacker may be able to avoid constraints related to the package name of the malicious app, thus avoiding practicality issues.

No mapping. Another alternative for PMs is to not implement any mapping. In this case, the PM would always suggest *all* stored credentials associated with *all* websites. This option is simpler than all other alternatives, but it is not secure, especially when compared to what current web-based PMs do. As an example, consider the LastPass browser extension: in the current version, the extension does not allow a user to insert her Facebook credentials on a website that does not share the *facebook.com* domain name. This is done as a security protection against phishing: even if the domain name graphically looks like *facebook.com* (by, for example, using Unicode character, as it would be the case in advanced phishing attacks), the password will prevent the user to fall for this phishing attack: mobile PMs that do not implement mappings cannot protect from this threat. However, if no mapping is implemented and all credentials are suggested, such protection is not available.

6.4 Case Studies

We performed the security assessment of the top four third-party leading PM apps (i.e., *Keeper*, *Dashlane*, *LastPass*, and *1Password*), each of which has millions of users around the world. We have also considered the Google Smart Lock, a service integrated with Google Play Services, which currently implements, among many other features, a password manager. In particular, we wanted to study how these PMs address the challenges described in the previous sections, and we were interested in answering questions such as: how does the suggestion system work? How do these apps map apps and package names to their associated websites? Is it possible for a malicious app to trick PMs to provide credentials for arbitrary websites? How difficult is for an attacker to mount such attacks? Moreover, as three out of four PMs include the OpenYOLO library, we assessed the reliability of its

implementation.

This section describes the methodology we adopted and the details for each of the PM we have analyzed. Our findings, summarized in Table 6.2, are worrisome: three of the third-party PMs implement a mapping based on various heuristics that an attacker can easily game. In other words, an attacker can create an app so that the target PM auto-suggests credentials associated with an arbitrary attacker-chosen domain name. Note that, in such cases, an attacker can leak credentials even from websites that do not have an associated mobile app—as long as the attacker can game the auto-suggestion system, the attacker wins.

Last, it is worth noticing that all third-party PMs support both a11y and Autofill Framework (for Android 8+); more precisely, we note that each PMs keep asking for the a11y permission even on Android 8.0 for backward compatibility reasons, as many apps have not modified their layouts yet to include Autofill XML attributes. We have also noticed that from the perspective of a user who sees an app being auto-filled, sometimes the steps to get the credential are slightly different, or there are some graphical differences, between PM relying on a11y or the Autofill Framework. We will discuss them case-by-case; however, we underline that all attacks that we discuss here works independently from the supporting technique.

Table 6.2: Summary of findings for Keeper (K), Dashlane (D), LastPass (LP), 1Password (1P), and Google Smart Lock (GSL).

	K	**D**	**LP**	**1P**	**GSL**
Secure mapping					✓
One-to-one mapping	✓	✓	✓		✓
Many-to-one mapping		✓			
Crowdsourced mapping			✓		
Heuristic-based mapping	✓	✓	✓		
No mapping				✓	
Q1) Vulnerable?	✓	✓	✓	✓	
Q2) Can co-exist on device?	✓	✓	✓	✓	
Q3) Can co-exist on Play Store?	✓	✓	✓	✓	
Q4) Targeted suggestion?	✓	✓	✓		

6.4.1 Methodology

We developed a three-step methodology to investigate the security of each password manager. These analysis steps are performed using reverse engineering assisted by simple static

analysis (e.g., bytecode decompilation) and dynamic analysis (e.g., bytecode instrumentation, network analysis, etc.).

Step 1: Package name as app identifier. The first step is to determine whether a given PM uses the package name of the target app as the *only* information to auto-suggest credentials for a given website. This step is done in the following way: (1) Install the legitimate Facebook app and add the credentials to the PM; (2) Uninstall the Facebook app; (3) Install a malicious app that has the same package name as the Facebook app and contains a login form. This app is written so that the only aspect in common with the legitimate app is the package name, while everything else is intentionally changed; (4) Check whether the PM auto-suggests the real Facebook credentials.

Although this step is straightforward from the conceptual and technical standpoints, it is enough to reveal key information: since in our test we change all the aspects *except* the package name, if the PM provides the correct credentials, it means that the package name is the *only* information used by the PM to identify the requesting app.

Step 2: Mapping extraction. If the first step reveals that the package name is the only aspect that matters, we then proceed to our second step: we aim at determining which specific technique the PM uses to map package names to domain names. This step is performed by a number of black-box tests and by then supporting the findings via manual reverse engineering of the PM.

Step 3: Exploitation. The last step consists in developing techniques to game the system and exploit the peculiarities of a given mapping implementation, if vulnerable. In this scenario, a proof-of-vulnerability consists in an app written so that the PM under analysis is tricked to provide the credentials of an arbitrary attacker-chosen website. In the general case, this app will need to have a carefully crafted package name and, at the very least, a login form. In other cases, it may be required to tweak other additional metadata.

6.4.2 Keeper

The Keeper app is the most downloaded PM with more than ten million users on Play Store. Keeper supports both a11y and Autofill Framework (on Android 8+), but it does not support OpenYOLO yet. When the user selects a form, it shows an icon with a yellow lock close to the form. When the user clicks on this icon, if the app is recognized, the related credentials are suggested (see Figure 6.1b). Otherwise, it asks to create a new entry.

Keeper also downloads from its backend a configuration file with a list of known websites (and their names). This file, interestingly, does not contain any reference to known package

names. In fact, this list is only used to auto-suggest website names when the user manually inserts a new set of credentials.

Mapping implementation. When the user opens an app that can be auto-filled, Keeper obtains its package name, through a11y or Autofill Framework. Keeper then needs to determine which website is associated with the current package name. To this aim, Keeper builds a *heuristic-based* mapping as follows: it uses the app package name to infer the URL of the app webpage on the Play Store (e.g., when the user opens the Facebook app, whose package name is `com.facebook.katana`, Keeper tries to access the webpage at `https://play.google.com/store/apps/details?gl=us&id=com.facebook.katana`). Then, if the webpage exists, Keeper parses out the domain name of the URL specified in the "app developer website field." This is the domain name that Keeper considers as the rightful owner, and it then stores the package name → domain name association in its internal mapping database. Finally, Keeper auto-suggests the credentials associated with this just-retrieved domain name.

Exploitation. Unfortunately, this mechanism is trivial to exploit for an attacker. In fact, the app developer URL is not validated by the Play Store and it thus cannot be trusted. We were able to create an app (with an arbitrary package name) and to publish it on the Play Store specifying *facebook.com* as the developer's website. In this way, when a user opens our app, the Facebook credentials (and only these credentials) are suggested.

6.4.3 Dashlane

Dashlane has been installed by more than one million users, and it supports a11y, Autofill Framework, and OpenYOLO. When Dashlane uses a11y, it shows its icon close to the form to fill; when the user clicks on it, the app is recognized and Dashlane suggests the related credentials (see Figure 6.1a); otherwise it asks to create a new entry. Instead, with the Autofill Framework, it directly shows a window with the suggested credentials or the launcher for creating a new entry, saving one interaction with the user.

Mapping implementation. Dashlane implements the mapping by means of two layers. The first one is a hardcoded mapping package → domain names containing 81 entries. The second layer is a *heuristic-based* mapping that attempts to infer which domain name should be associated to a given package name (this layer is used only if the package name is not contained in the static mapping). Our analysis revealed that such heuristic works in this way: Dashlane first splits the package name in components separated by the dots (e.g., the *aaa.bbb.ccc* is split in the three components *aaa*, *bbb*, and *ccc*). Then, for each component, it checks whether at least *three* of its characters are contained in the "website" field of one (or more) of Dashlane entries. For example, the package name *xxx.face.yyy*

triggers an auto-suggestion for *facebook.com* credentials (as well as anything associated with *facts.com*, for example).

Exploitation. The static mapping is rather small and many entries are tied to well-known apps and websites. However, we noticed that such mapping is *many-to-one*. Therefore, there are multiple package names pointing to the same domain name. For example, we found that both `com.etrade.mobilepro.activity` and `com.etrade.tabletapp` point to `www.etrade.com`, the official website of the Etrade online banking platform: the two apps appear to be the smartphone and tablet versions of the same product, respectively.

Consider a user who has installed the smartphone version of the app. An attacker could then exploit the many-to-one mapping by luring the victim to install a malicious app having the package name of the tablet version (that the user did not already install): in this case, the attacker does not need to lure the victim to uninstall the first app (as it would be the case without the many-to-one mapping). We reported this attack for the sake of completeness, but we acknowledge it is affected by practicality issues.

However, the second layer of the mapping is severely vulnerable. In fact, it is sufficient to upload to the Play Store a malicious app whose package name contains three (or more) letters that overlap with the domain name the attacker wants to target; in this case, the malicious app will be auto-filled with the credentials of the victim domain. Furthermore, it is worth noticing that the malicious app can obtain credentials from multiple domains. For instance, we submitted to the Play Store an app with package name *com.lin.uber.face*: when opening this app, Dashlane promptly suggests Linkedin, Uber, and Facebook credentials.

Regarding OpenYOLO, Dashlane is exploitable exactly as a11y/ Autofill Framework, since the selection of credentials relies on the package name, which is parsed as previously described. Therefore, we wrote another malicious app embedding the OpenYOLO client library and we were able to obtain the credentials.

Interestingly, we have noticed that when Dashlane uses Autofill Framework instead of a11y, it performs some additional checks and it is able to determine that our simple proof-of-concept attempting to impersonate Facebook cannot be verified. In this case, a warning is shown to the user. To the best of our understanding, Dashlane employs a hardcoded list of well-known package name and signature pairs, and it checks our app against it. This is a promising step forward in the right direction. However, we found that these checks are easily bypassable. In fact, it is sufficient for a malicious app to *not* be compatible with the Autofill framework (this can be done by not using the new autofill-related XML attributes), and this will be enough to force Dashlane to rely on a11y and the vulnerable implementation.

6.4.4 LastPass

LastPass has been installed by more than one million users and it supports a11y, Autofill Framework, and OpenYOLO. With a11y, LastPass uses a permanent notification to alert the user if the currently active app has some form to fill; thus, she has to tap the notification to show a popup window with her credential; with the Autofill Framework, the user does not need to tap the notification and she will directly see the pop up window, as in Dashlane. This underlines that the support to OpenYOLO is still immature. However, the current implementation allows the user to select credentials and send them to any unidentified requesting app.

Mapping implementation. LastPass relies on two mappings. The first one is, once again, *heuristic-based*, and it works as follows. Given a package name, e.g., *aaa.bbb.ccc*, LastPass splits it in components separated by the dots (e.g., *aaa*, *bbb*, and *ccc*), and it builds a domain name pattern by using the first two in reversed order (e.g., *bbb.aaa*). LastPass will then suggest to the user all the credentials associated with domain names that end such pattern.

In case an entry does not exist, LastPass allows the user to search among her locally stored credentials and select (in case) one of them, thereby defining a new entry for the mapping. As such entries may be useful for other users worldwide, LastPass allows the user to share them with the community. This sharing step is at the basis of the second mapping, a *crowdsourced mapping*. LastPass downloads this global database at the first installation. At the time of writing, we found 19,273 crowdsourced mapping entries with repeated package names and domains, mostly many-to-one. For instance, we found a mapping between package names *com.tinder* and *com.tinderautoliker2* associated to the web domain *facebook.com*: Tinder is a dating app that needs Facebook credentials to authenticate the user, while TinderAutoLiker is an app available on alternative markets that automates some actions on Tinder services. It is also worth noting that the crowdsourced mapping contains errors, like invalid domains, domains with typos, and IP addresses belonging to local networks.

Exploitation. To exploit the first mapping strategy, the attacker can create an app with a package name beginning with the reverse of the target domain name. For example, we created an app with package name *com.facebook.evil* and we were able to upload it to the Play Store without problems: when the user opens this app, LastPass automatically suggests credentials related to *facebook.com*.

From the conceptual point of view, an attacker could exploit the second mapping as well. In fact, if the attacker is able to inject an arbitrary association, she can directly indicate to LastPass that, for example, her own package name should be associated to, say, *facebook.com*. For the sake of completeness, we tried to share with LastPass an association

from one of our package name app to one of our test websites. However, this association did not become public to all users. We assume that LastPass make these "new" associations available to all its users only when a number higher than a threshold suggested them. An attacker could try to create a high number of fake accounts and to automatically share these fake associations. However, we have opted not to do it for ethical reasons. Moreover, an attacker can already game LastPass suggestion mechanism by exploiting the first mapping.

6.4.5 1Password

1Password has been installed by more than one million users and it supports a11y, Autofill Framework and OpenYOLO. Differently from previously analyzed PMs, 1Password organizes its entries in categories (e.g., credit card, database, driver license, login, wireless router, etc.). We focused on the login category. Once the user selects a form, 1Password behaves differently with respect to the supporting methodology: on Autofill Framework, it shows a small windows bearing the imprint "Autofill with 1Password". Clicking on it, the user must insert the 1Password master password and search through all its previously saved credentials. With a11y, it directly loads the windows for searching among credentials. Albeit 1Password adopts the OpenYOLO library, the implementation contains just a stub that always returns empty credentials.

Mapping. 1Password does not provide any mapping, but it trivially suggests each stored credential through a searchable list, delegating the choice to the user. In other words, it is possible to autofill any requesting app with any stored credential.

Exploitation. The exploitation of 1Password was straightforward and did not require any further customization of the app. However, this attack is less practical than the other ones as the attacker does not have fine-grained control over the list of credentials that are auto-suggested.

6.4.6 Google Smart Lock

Google Smart Lock (GSL) is part of Google Play Services for Android. It was created to automatically keep the device locked when the user is not around and unlock it when specific user-defined constraints are met. For instance, the user can choose to have her device unlocked according to the presence of specific wireless connections, trusted locations, or when it recognizes the user's face or voice, or while the user is carrying the device. GSL has been equipped with the PM originally integrated into the Chrome browser. For this reason, GSL also offers a password-saving feature, taking advantage of Autofill Frame-

work (which works just with compatible apps), and a synchronization mechanism with the Chrome desktop browser.

Mapping. We believe that GSL mapping is securely implemented. However, the burden of mapping creation is delegated to the developer who has to provide all the necessary information to Google. In particular, the official documentation describes a multi-step process (Goo18e). From the technical standpoint, this process is based on Digital Asset Links (Goo17a), through which an app can be verifiably linked to a website (see Section 6.3.3, "Secure mapping"). However, this procedure is not fully automated, and developers are requested to fill a Google Form manually and to provide a set of information. We argue that such a process hardly scales, as it is centralized and it requires the manual intervention of the developer. To improve the current approach, Google should push the Digital Asset Links adoption and verify that it is correctly implemented. Moreover, we believe that Google would greatly benefit the community if it could make its current mapping database publicly available.

6.5 Instant Apps for Full UI Control

The attacks presented so far require a malicious app to be installed on the victim's device. This section discusses how this prerequisite can be waived by abusing the recently introduced *Instant Apps*. This technology, implemented by Google, allows users to "try" Android apps at the touch of a click, without the need for a full installation.

This mechanism works in several steps. First, the developer builds an Instant App, a small-but-functional version of her app, and she uploads it to the Play Store. The developer is also asked to associate a URL pattern to it (pointing to a domain name she controls). The idea is that when the user browses to a URL satisfying this pattern, the Android framework starts the process of downloading and running the Instant App associated with it. Of course, for security reasons, the app developer needs to first prove to Google that she controls the target domain name. This is carried out through a multi-step procedure called App Link Verification (Goo18i), which relies on Digital Asset Links (Goo17a) protocol (this makes possible to associate an app with a website and vice versa, via verifiable statements).

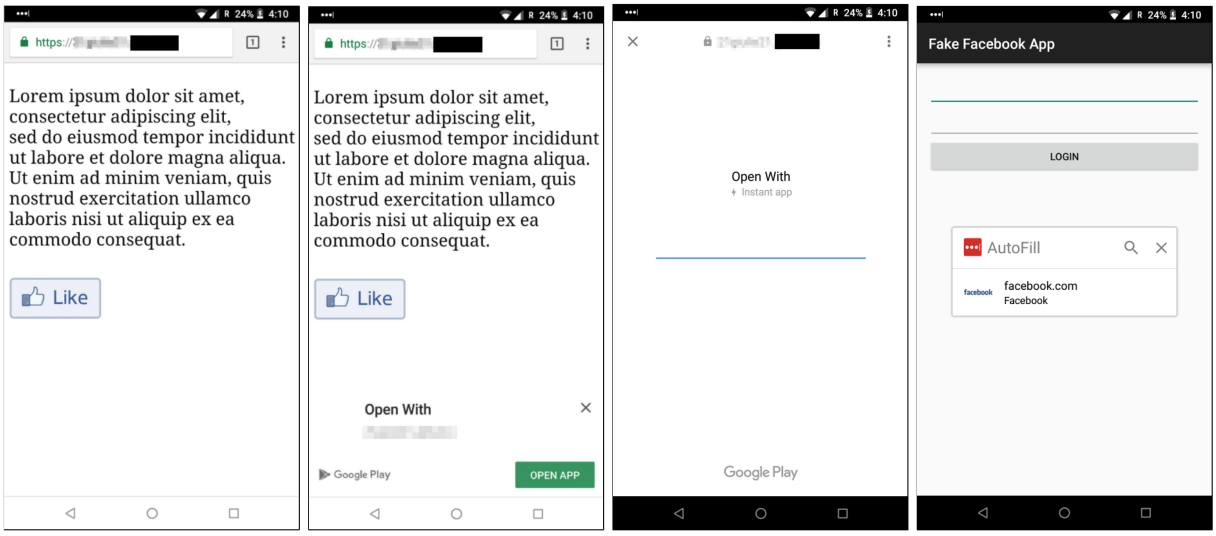
From the developers and users' usability perspective, Instant Apps is a great feature as it significantly lowers the friction for a user to test (and possibly fully install) an app. However, from the security point of view, *Instant Apps provide a venue for attackers to greatly facilitate phishing attacks.*

The key observation is that Instant Apps allow an attacker to move from *web phishing* to *mobile phishing*. Nowadays, web phishing is significantly more challenging than mobile phishing. On the web, the user can clearly see which website she is interacting with: she

has the chance to check the domain name, whether the connection is done via HTTPS, and whether there is a valid SSL certificate. In the mobile world, however, there are no such indicators. In fact, as previous works have pointed out (CQM14; BCI+15; RZX+15), there is currently no "green lock" or any space for any trusted indicator: these previous works have shown that a malicious app spoofing the Facebook UI can be made indistinguishable from the legitimate Facebook app—even for a security-savvy user. The key requirement for these pixel-perfect attacks is the ability to control all the pixels on the screen. A website cannot achieve that, but an attacker can use Instant Apps to do just that: gain code execution on the device outside the browser's JavaScript sandbox and gain the ability to fully control the UI (without requesting any permission).

Once the attacker has gained full UI control, there are many possibilities. One first example is that the Instant App could resemble the real Facebook app, which can be made indistinguishable from the legitimate one. A second example would be to *resemble the browser app itself*: as the attacker controls every pixel of the screen, nothing prevents her from showing the user a browser-like view with a spoofed *facebook.com* domain name and a green lock. Once again, this attack can be made indistinguishable from a legitimate scenario.

6.6 Practical Phishing Attacks



(a) Phishing website (b) IA popup (c) IA loading (d) LastPass is tricked

Figure 6.3: Instant Apps (IA) phishing attack PoC

The password managers flaws and Instant Apps "features" we have highlighted thus far are

independent of each other. However, we found that for what concerns phishing attacks, these two technologies are, in fact, complementary. In fact, we have shown that password managers can be tricked into revealing users' credentials, but these attacks require a malicious app (with an attacker-chosen package name) to be installed on the victim's phone: Instant Apps can be used to do just that.

We have found that Instant Apps, although they are not fully installed apps, do appear as they were to the Android framework and the components relying on it. The key insight is that even if the Instant App is not fully installed, the app somehow *lives* on the Android device, and its package name, application name, and icon are attacker-controlled (they are, in fact, the same as its associated full app on the Play Store). To make it worse, password managers currently do *not* notice the difference between full and Instant Apps, and they can thus be tricked to leak credentials even to them.

To make things worse, we have found that current password managers autofill hidden fields as well. This yet another "feature" that opens the possibility for a stealthy and practical end-to-end phishing attack, which we describe next.

6.6.1 End-to-end proof-of-concept

Consider a scenario where the user visits a website showing a spoofed Facebook "like" button, as in Figure 6.3a. Such button links to an attacker-controlled URL that is associated with her Instant App. Once the user clicks on the like button, the Instant Apps mechanism is triggered: the popup asking the user confirmation to start the Instant App is shown, as in Figure 6.3b. This popup shows the application name and the icon, which, however, are fully attacker-controlled. The reader can see from Figure 6.3b how it is easy to mislead the user: for this PoC we used "Open With" as the name of the app and a fully white square as the app's icon ("showed" on the left of the application name). Upon the user's click on the "Open app" button, the Instant App is automatically downloaded, while the user is shown for few moments (about one second) the view in Figure 6.3c. At this point, the malicious Instant App is running on the user's device, as shown in Figure 6.3d. At this point, since our app was created with a package name following the *com.facebook.** pattern (see Section 6.4.4), LastPass is tricked to automatically suggest the real Facebook credentials to the user: With a click on the autofill popup, the full credentials are leaked to the attacker.

We note that our app is a clearly "fake" Facebook app, just for clarity sake and for ethical and copyright concerns: as this is a "live" PoC (to test the Instant Apps we needed to publish it to the Play Store), we preferred to avoid having a real spoofed Facebook UI.

Practicality considerations. We have shown how the user can be lured to leak her credentials in just a few clicks. We also note that the click on "Open app" (6.3b) and the

"Loading" view (6.3c) are only shown the first time. That is, an attacker could make this attack even more practical by luring the user to approve and download the Instant App beforehand and for phishing-unrelated, seemingly innocuous reasons, to then make the transition from "Click to the like button" to "Spoofed Facebook UI" really seamless. We believe this attack strategy significantly lowers the bar, with respect to all known phishing attacks on the web and mobile devices: to the best of our knowledge, this is the first attack that does not assume a malicious app already installed on the phone and that does not even require the user to insert her credentials. These attacks are strictly more practical than all currently known mobile phishing works (CQM14; BCI+15; RZX+15; FQCL17).

6.6.2 Hidden Password Fields

We have carried out further experiments with the aim of assessing whether current mobile password managers are vulnerable to automatically filling hidden fields. We refer to fields as *hidden* if the field is, for one reason or another, not visible to a user. This is relevant because an attacker could create a form with a username field and a hidden password field: if the victim uses her password manager to autofill this form, her password will be silently leaked to the attacker. This is similar to what previous research has attempted with web-based password managers (Rod13): To the best of our knowledge, we are the first to show that these attacks work with mobile password managers as well. For this work, we considered four different techniques to make a password-related `EditText` seemingly invisible: 1) transparency, 2) small size, 3) same-color background and foreground, and 4) the *invisible* flag.

Transparency. To create a transparent `EditText` in Android, it is possible to set its alpha value accordingly (via the `setAlpha()` API). We note that if the alpha value is set to zero, both the a11y and Autofill Service cannot autofill the `EditText` because it is not visible anymore. However, setting an alpha value of 0.01 is enough to keep the field invisible and make the autofill mechanisms work.

Small size. One other venue to make a field invisible to the human eye is to make it very small. We found that password managers autofill password fields even if their size is 1dp × 1dp, independently from whether they are using a11y or Autofill Service.

Same-color background and foreground. If the text color is the same of the background color, the field (and its content) will not be visible. This technique works well with a11y. However, unexpectedly, it is not enough to trick Autofill Service. In fact, upon autofilling, the Autofill Service would overlay the autofilled fields with a yellow overlay, thus making the hidden field visible to the user. However, it would be possible for an attacker to create in-app overlays (which do not require additional permissions) to cover this yellow overlay, thus making this artifact not visible to the user.

Invisible flag. It is possible to make a field hidden by setting its visibility to `View.INVISIBLE`. We found that a11y-based password managers do *not* autofill these "invisible" fields, but those ones using Autofill Service do so.

Discussion. We believe these additional techniques make end-to-end phishing attacks even more practical and problematic. While the unsuspecting user will use password managers and instant apps to quickly provide her email address or username, her credentials could be silently leaked to the attacker, with only few clicks. We also note that while some of the above techniques are not working with both a11y and Autofill Service, there is nothing preventing an attacker to *combine* these techniques at her will and adapt given the attack scenario. Finally, we note that these password-stealing attacks are possible only because current password managers implement a vulnerable mapping algorithm: without such vulnerability, no credentials can ever be leaked to non-legitimate apps.

6.7 Secure-by-Design API

We believe that the attacks presented in this chapter are due to design problems of the current mechanisms to support autofill, from a11y, to the more recent Autofill Framework and OpenYOLO. The key design issue is that all these mechanisms use package names as the main abstraction to work with, thus leaving developers of password managers with the daunting task of mapping apps to their associated domain names. Given the number of security issues and misplaced trust assumptions we have identified in leading password managers, we believe third-party developers should not be asked to implement this critical step.

The *getVerifiedDomainNames()* API. We propose a new API that implements a secure-by-design mechanism by using domain names as the only abstraction that password managers need to interact with. Since credentials are created for websites, we argue this is a better abstraction level. In stark difference concerning existing proposals, this API, called *getVerifiedDomainNames()*, would directly provide to password managers a list of domain names that a given app is legitimately associated to. The API internal implementation would then be responsible for performing all the needed security checks. We envision this API to be used following the paradigm of OpenYOLO (as in Figure 6.2). The main difference is that password managers would directly query for domain names, and not for package names.

Integration and implementation. The request for auto-filling a form follows several steps. First, the client sends an Intent to the password manager to request credentials. Then, the password manager can invoke *getVerifiedDomainNames()*, passing the received Intent as argument. At this point, our API performs a number of steps, whose sequence

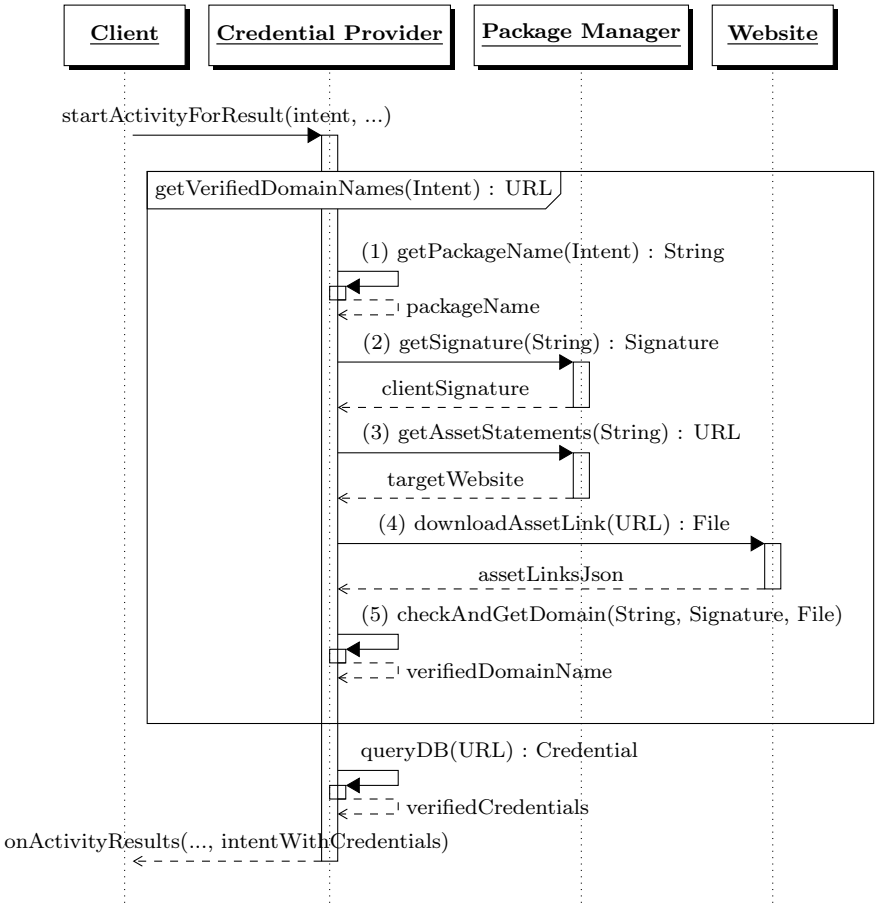


Figure 6.4: *getVerifiedDomainNames()* API sequence diagram

diagram is depicted in Figure 6.4. First, it retrieves the sender's package name from the Intent. The package name is used to extract the client's app signing key. Then, *getVerifiedDomainNames()* extracts from the client's manifest file the list of domain names the app claims to have access to (this list should be specified according to the standard App Link Verification (Goo18i) and Digital Asset Links (Goo17a) protocols). The API internally downloads, for each of these domain names, the associated DAL file (assetlinks.json) and it verifies that the requesting app (package name + hash of the app signing key) is listed in it. The API includes in its return value to the password manager the list of all domain names that satisfy such security checks. Given these domain names, the PM can then safely query its internal database for associated credentials and send them back to the requesting client.

Avoiding side-channel vulnerabilities. We have noticed that the current OpenYOLO client implementation opens apps to side channel attacks. In particular, the current im-

plementation sends a Broadcast Intent to request credentials from the credential provider, thereby making all other apps aware of such request. A malicious app can use this side-channel to infer that the user is about to login in a specific account: this information can be used for the attacker to know *when* to spawn its spoofed phishing UI (CQM14; BCI+15; RZX+15). Even if side channels are not required to mount phishing attacks (AP17), they do make them easier. For this reason, we argue that the communication between the client and the credential provider must remain confidential—not only the content, but even the mere fact that this communication is taking place. To this end, we believe that each client should have access to a (configurable) list of trusted password managers apps (e.g., Dashlane, LastPass, ...), so that explicit intents can be used instead of broadcast intents. This list could be stored as pairs of package names and hash of signing keys. This is analogous to what browsers do with trusted certificates.

Practicality of adoption. Independently from the API we propose, we were interested in determining how ready the ecosystem is in terms of information required to build a secure app-to-web mapping. Given that the current standard is DAL, we set to analyze the adoption rate by querying a dataset of domain names for their related *assetlinks.json* DAL file. As a dataset, we considered all domain names from all mapping we extracted from the password managers we have inspected. This list is constituted by 8,821 domain names. Note that since they are extracted from password managers, we know that these domain names host at least one page with a login form, thus making them relevant to our analysis.

To our surprise, only 8% (710/8,821) of them host an associated DAL file, and only 2% (178/8,821) specify an Android app in accordance with Google documentation (Goo18c). This low adoption rate is worrisome: password managers would have compatibility problems in securely implementing their solution even if they were fully aware of the problems discussed in this chapter. Google Smart Lock has addressed these problems by not relying on a fully automatic technique (developers need to manually fill a Google form) and by supporting app-to-web sync only when a secure mapping exists. We argue that the rest of password managers should follow a similar approach and warn the user about potential problems when a secure app-to-web association cannot be established.

6.8 Related Work

Phishing is a well-known problem and it has received the attention of the security community for several years. In the realm of mobile devices, there have been a number of works focusing on task hijacking (CQM14; RZX+15; FW11), and UI confusion (BCI+15; AP17). We built on the insights provided by these works and we have shown how features implemented for convenience can make mobile phishing attacks significantly more practical

than what previously thought: we do not assume a malicious app is already running on the victim's device and, for the first time, the user is not even required to type her credentials. Few works also proposed defense mechanisms for mobile phishing (BCI+15; FCP+16), which are unfortunately not finding adoption due to the invasive framework modifications they require. Another interesting research direction is the automatic identification of app widgets that contain user's sensitive info (NYY+15; HLX+15; AAL+17).

The problem of phishing has also been extensively studied in the browser context (CLT+04; DT05; KK05). In this context, protection mechanisms are usually implemented in forms of blacklist (Goo18g).

Another class of UI-related attacks is tapjacking (also called clickjacking). Some works have shown how an attacker can abuse the overlay system to lure users into unknowingly perform security-sensitive operations (NS12; WBDJ16; FQCL17). Other works show how accessibility service can be abused to bypass user interaction and perform UI-related attacks (FQCL17; JSC+14; Ami16b; Ami16a; Loo15; Lui16; Ven16). These are very powerful attacks, but they differ from phishing: they are about luring a user to perform a sensitive operation, while phishing focuses on luring them to leak their credentials.

A few recent works have focused on the security analysis of browser password managers (LHAS14; SJ14b). In those works, the authors conduct a security analysis of popular web-based password managers, and some of them were found exploitable, allowing an attacker to leak user credentials. The root-causes of the vulnerabilities were ranging from logic and authorization mistakes to traditional web vulnerabilities like CSRF and XSS. Our work, instead, focuses on *mobile* password managers. We also note that we have not focused on identifying classic implementation bugs, but we aimed at uncovering systemic design issues.

Silver et al. show several attacks aimed at retrieving passwords from in-browser PMs, by exploiting their autofill policies (SJB+14); the most powerful attack they uncovered does not require any human intervention and it allows to automatically auto-complete password fields. Several prior works show how combining innocuous visible fields and sensitive invisible fields trigger PMs to autofill, and, consequently, provide sensitive information to the attacker (dV13; Cop17). This is similar to our experiment with hidden password `EditText` widgets.

For what concern the security of Android password managers, the work by Fahl et al. is one of the few in the area (FHO+13): in this paper, the authors studied 21 popular password managers and show how password managers would somehow push users to "copy" their passwords to their clipboard: this has security implications since the device clipboard can be accessed by any app installed on the user's device. Interestingly, we note that password managers using a11y or Autofill Service are not affected by these problems: passwords shared via these "modern" features do not go through the clipboard. However, we have

shown that even these modern mechanisms are affected by security problems as well.

6.9 Responsible Disclosure

We have responsibly disclosed our findings to the security teams of the password managers we found vulnerable. We would like to acknowledge their quick and professional handling of the matter. The affected vendors are in the process of deploying countermeasures.

Chapter 7

Detecting Frame Confusion in Hybrid Android Apps

7.1 Introduction

Most of the mobile devices in the market use an Android operating system, whereas a substantial percentage of mobile devices use the iOS operating system: a market share of 76% and 22% respectively in the July of 2019 (Sta19b). The two operating systems obviously differ when it comes to software development and operating system architecture. The diverseness between these two operating systems has a negative impact on the application development process. Companies incur more cost than what is expected; this is because companies must rely on the services of different developers to build applications for those different architectures. The result of this is a high maintenance and development costs. A promising way to overcome the limitation posed by such multi-platform development process is a *cross-platform* framework, which allows to implement an app using a unique programming language and automatically generate a corresponding Android and iOS version. The cross-platform framework has played a critical role in reducing the time and cost required to build these applications. Cross-platform frameworks, such as Cordova (Cor19), allow developers to utilize standard web technologies (HTML, CSS and JSJavaScript), for cross-platform development of *hybrid* applications applications. Hybrid applications are, therefore, popular due to the feature that allows them to be used on two different platforms and work almost similarly to web apps because they are built with HTML and JavaScript programming languages. Hybrid applications also incorporate standard native features using a wrapper that is deployed to act as a bridge between the two platforms. The bridge layer in the hybrid applications allows JavaScript code to access the device capabilities that would be virtually impossible to obtain from the mobile browser. Through such interfaces, the developer can use callbacks to grant access to hybrid applications, making them similar

to the native app. However, from a security standpoint, the interaction between the native and the web worlds (which rely on different security models and requirements) can expose hybrid apps to ad-hoc and complex vulnerabilities, like those described in (Hu18; LHD+11; NLP13; CW14; LWZ+17). Among them, the *Frame Confusion* vulnerability (LHD+11) in hybrid apps has been discovered some years ago and it has been fixed on iOS[1] but not on Android (neither in the latest version, i.e., Android Pie 9.0). To this regard, we argue that a lot of hybrid apps still suffer from such vulnerability and that there is still a lack of i) an extensive analysis of Frame Confusion, ii) a methodology to automatically detect Frame Confusion in hybrid apps, and iii) a reliable solution to mitigate the problem. The Frame Confusion vulnerability arises due to the invoking of the native code by JavaScript through web pages containing at least an *iframe* element. Such element allows loading external contents (e.g., advertisements, video and payment systems) from domains which differ from the domain of the hybrid app. For this reason, any *iframe* is in charge of *containerizing* the rendered sub-page, and should execute content only within the scope of its own domain, as prescribed by the Same-Origin Policy (SOP). However, in case of web pages with multiple iframes, the WebView is unable to identify the iframe that invokes a function in the native code, and thus the result of the invocation is always executed in the main app page, thereby inducing the confusion problem. Such misbehavior occurs as the *JavaScriptInterface* is bound by the OS to the entire WebView element, without any distinction among the domains (and thus the iframes) that invoke the function calls. Therefore, the Frame Confusion vulnerability allows to bypass the isolation granted by the iframe security model and to build a communication channel between web pages belonging to different domains, (i.e., the main app page and the inner iframes). As a consequence, such vulnerability can affect the confidentiality and the integrity of hybrid apps: a malicious iframe can, for instance, force the main app to expose private information (like session cookies or internal app files) or mount sophisticated phishing attacks.

Contribution. In this work, we focus on the Frame Confusion vulnerability on the Android OS.
Our contribution is three-fold.

1. We propose a methodology for systematically detecting the Frame Confusion vulnerability in hybrid apps on Android.

2. We present *FCDroid*, a tool that implements such methodology to automatically identify hybrid apps on Android that suffer from the Frame Confusion vulnerability.

3. We discuss the results of an extensive analysis carried out through FCDroid on a set of $50,000$ apps downloaded from the Google Play Store.

[1] `https://cordova.apache.org/docs/en/latest/guide/appdev/security/index.html#iframes-and-the-callback-id-mechanism`

The experimental results indicate that the 49.35% of the analyzed apps are hybrid, as they use the WebView component and enable JavaScript execution, while about 6.63% of them (i.e., 1637 apps) were found to be vulnerable to Frame Confusion for a total of more than 250.000.000 app installations worldwide. To further validate the proposed methodology, we have manually analyzed some of these vulnerable apps to find out possible attacks exploiting the Frame Confusion vulnerability. To this regard, we show how to successfully exploit Frame Confusion in an application that has more than 1M users worldwide, and how to mount a phishing attack from this vulnerability. The rest of the chapter is organized as follows: Section 7.2 introduces some technical background on hybrid apps and the Frame Confusion, while Section 7.3 discusses the detection methodology. Section 7.4 presents FCDroid, while Section 7.5 shows the experimental results. Section 7.6 discusses the exploitation of the Frame Confusion on an actual news app, and Section 7.7 presents some related work. Finally, Section **??** concludes the chapter.

7.2 The Frame Confusion vulnerability

7.2.1 App typologies

Mobile apps can be divided into three categories: i) *native*, ii) *web*, and iii) *hybrid* apps.

Native apps are developed for a specific mobile platform using particular programming languages and technologies. IOS apps, for example, are written in Objective-C and Swift, Android apps in Java or Kotlin. This means that a separate version of the app must be developed for each platform. Developers cannot reuse any piece of code from another platform version, as it is written in a completely different programming language. For this reason, native app development is considered to be the most time-consuming and most expensive. Native app development is preferred for their high-performance apps and because they can easily interact with a set of API calls exposed by the mobile OS. On the other hand, they need to be re-implemented to execute on a different mobile OS. As this is a complicated and expensive task mostly for small-medium enterprises, there is a growing trend towards web or hybrid apps in order to lower maintenance and support prices.

Web apps render HTML5 and execute Javascript code within the device browser (which is a native app). For this reason, they are highly portable and platform-independent, but the interaction with the underlying OS is limited to the API accessible by the browser itself. Consequently, they have restricted functionalities and, in general, limited performance.

Hybrid apps are technically web apps packed in a native app container using the WebView. Like a web app, it is written in HTML, CSS, and JavaScript. The WebView may

also allow the interaction between the web and the native part, acting as a bridge between the web code and the host OS API, thereby allowing to render HTML/CSS content, execute JavaScript code, and get access to the full OS API. Accordingly, hybrid mobile apps are cross-platform with the great advantage of is their lower price development. The downsides, compared to native apps, are their limited performance and their different "look and feel".

7.2.2 WebView

On the Android platform, WebView is a subclass of View (an object that is the basic building block for user interface components), and it is used to display web pages. Using WebView, Android applications can easily embed a browser and using it not only to display web contents, but also to interact with web servers. Once a WebView is created, Android applications can use its `loadUrl` method to load a web page if given a URL string. Moreover, WebView provides a mechanism for its JavaScript code to invoke the Java code of the app. The API used for this purpose is called `addJavascriptInterface`. Android applications can register Java objects to WebView through this API, and all the public methods, with the annotation `@JavascriptInterface`, in these Java objects can be invoked by the JavaScript code from inside WebView. For the sake of precision, the DEX code is executed, not the Java code, but we refer to the Java code because we are presenting these concepts from the developer point of view. The communication between the JavaScript and the Java code is handled by the WebView using *asynchronous* callbacks. In detail, when some JavaScript code invokes Java code through an interface bounded to the WebView, it does not wait for the result: instead, when the result is ready, the Java code outside the WebView invokes a JavaScript callback function, passing the result back to the web page. This mechanism provides improved app performance and responsiveness, particularly in the case of long-running operations that would block the UI. JavaScript interacts with Java object on a private, background thread of this WebView. Care is therefore required to maintain thread safety. Because the object is exposed to all the frames, any frame could obtain the object name and call methods on it. There is no way to tell the calling frame's origin from the app side, so the app must not assume that the caller is trustworthy unless the app can guarantee that no third party content is ever loaded into the WebView even inside an iframe. This feature allows cross-platform frameworks (e.g., Cordova, PhoneGap) to design a set of plugins that can be embedded in apps and offer platform-specific functionality, such as the API for the file-system or the GPS location.

WebView Security Mechanisms. As the WebView deals with web content that can include untrusted HTML and JavaScript code, it can suffer from well-known web security vulnerabilities such as cross-site scripting (Bha13; JHY+14; BYZW17) or file-based cross-zone scripting (CW14). As countermeasures, the Android OS includes a set of mech-

anisms aimed at limiting the capability of the WebView to the minimum functionality required by hybrid apps. By default, the WebView does not execute JavaScript, thus requiring developers to enable this feature using the `setJavascriptEnabled` method. Besides, the application can either enable or disable the access of the WebView to specific resources like files, databases or geolocation (Web19a) through the `WebSettings` object. Since API 17, in order to expose a Java method it must be be explicitly annotated with the `@JavascriptInterface` (Jav19) annotation. The aim is to restrict the access to the OS API, in order to prevent the invocation of any public Java method through code reflection (Tho15). To further increase the resilience of the WebView component against untrusted contents, since API level 21 the Android OS implements the WebView as an independent app, thus offering a centralized update mechanism that relieves the developer from the burden of manually updating each hybrid app (Web19b). Moreover, since API Level 26, the WebView renderer executes in a separate process (Web17). Finally, since Android 8, the WebView incorporates Google's Safe Browsing protections to detect and warn users about potentially dangerous websites. Unfortunately, this option needs to be explicitly enabled by the developer through a specific tag in the Android Manifest (Web18).

7.2.3 Frame Confusion

In the Android system, interactions with several components of the system are asynchronous and require a callback mechanism to let the initiator know when the task has completed. Therefore, when the JavaScript code inside WebView initiates such interactions through the interface bound to WebView, JavaScript code does not wait for the results; instead, when the results are ready, the Java code outside WebView will invoke a JavaScript function, passing the results to the web page. Let us use `DroidGap`'s `ContactManager` interface as an example: after the binded Java object has gathered all the necessary contact information from the mobile device, it calls `processResults`, which invokes the JavaScript function `contacts.droidFoundContact`, passing the contact information to the web page. The invocation of the JavaScript function is done through WebView's loadUrl API, as shown in Listing 7.1.

Listing 7.1: Example of loadUrl API

```
public void processResults(Cursor paramCursor){
        String result = paramCursor.decode();
        String str8 = new StringBuilder().append("javascript:
        navigator.contacts.droidFoundContact(...)").
        localWebView.loadUrl(str8);
}
```

The JavaScript function `contacts.droidFoundContact` in the example is more like a call-

back function handler registered by the LivingSocial web page. The use of the asynchronous mode is quite common among Android applications. Unfortunately, if a page has frames (e.g., iframes), the frame making the invocation may not be the one receiving the callback. This exciting and unexpected property of WebView becomes a source of attacks.

In a web page with multiple frames, we refer to the main web page as the main frame, and its embedded frames as child frames. The following example, presented in Listing 7.2, demonstrates that when a child frame invokes the Java interface bound to the WebView, the code loaded by loadUrl is executed in the context of the main frame.

Listing 7.2: Example of addJavascriptInterface API

```
Object obj = new Object(){
    @JavascriptInterface
        public void showDomain()
        {mWebView.loadUrl("javascript:alert(document.domain)");}
};
mWebView.addJavascriptInterface(obj, "demo");
```

The code above registers a Java object to the WebView as an interface named "demo", and within the object, a method "showDomain" is defined. Using loadUrl, this method immediately calls back to JavaScript to display the domain name of the page. When we invoke `window.demo.showDomain()` from a child frame, the pop-up window actually displays the domain name of the main frame, not the child frame, indicating that the JavaScript code specified in loadUrl is actually executed in the context of the main frame. Whether this is an intended feature of WebView or oversight is not clear. As results, the combination of the `addJavascriptInterface` and `loadUrl` APIs creates a channel between child frames and the main frame, and this channel opens a dangerous Pandora's box: if application developers are careless, the channel can become a source of vulnerability, one that does not exist in the real browsers.

Attack from Child Frame. In this attack, we look at how a malicious web page in a child frame can attack the main frame. We use the `LivingSocial` app as an example. This app loads `LivingSocial`'s web pages into its WebView (in the main frame), and we assume that one of their iframes has loaded the attacker's malicious page. This is not uncommon because that is exactly how most advertisements are embedded. The main objective of the attacker is to inject code into the main frame to compromise the integrity of `LivingSocial`. Web browsers enforce the Same Origin Policy (SOP) by completely isolating the content of the main frame and the child frame if they come from different origins. For example, the Javascript code in the child frame (`www.advertisment.com`) cannot access the DOM tree or cookies of the main frame (`www.facebook.com`). Therefore, even if the content inside iframe is malicious, it cannot and should not be able to compromise the page in the main frame. As we have shown earlier, `LivingSocial` binds `CameraLauncher` to its WebView.

In this class, a method called `failPicture` is intended for the Java code to send an error message to the web page if the camera fails to operate (Listing 7.3).

Listing 7.3: Example of addJavascriptInterface API

```
public class CameraLauncher{
    @JavascriptInterface
        public void failPicture(String paramString){
                String str = "javascript:navigator.camera.fail('";
                str += paramString + "');";
                this.mAppView.loadUrl(str);
    }
}
```

Unfortunately, since `failPicture()` is a public method in `CameraLauncher`, properly annotated with `@JavascriptInterface`, the method is accessible to the JavaScript code within WebView, from both child and main frames. In other words, JavaScript code in a child frame can use this interface to display an error message in the main frame, opening a channel between the child frame and the main frame. At the first look, this channel may not seem to be a problem, but those who are familiar with the SQL injection attack should have no problem inserting some malicious JavaScript code in 'paramString', like the following:

```
x'); malicious JavaScript code;
```

As results, the malicious code embedded in `paramString` will now be executed in the main frame; it can manipulate the DOM objects of the main frame, access its cookies, and even worse, send malicious AJAX requests to the web server. This is exactly like the classical cross-site scripting attack, except that in this case, the code is injected through WebView, as illustrated in Figure 7.1a.

Attack from Main Frame. In this attack, we look at how a malicious web page in the main frame can attack the pages in its child frames. We still use the `LivingSocial` as an example. We assume that the attacker has successfully tricked the `LivingSocial` app into loading his/her malicious page into the main frame of its WebView. Within the malicious page, `LivingSocial`'s web page is loaded into a child frame. The attacker can make the child frame as large as the main frame, effectively hiding the main frame. Suppose that `DroidGap` uses tokens to prevent unauthorized JavaScript code from invoking the interfaces registered to WebView: the code invoking the interfaces must provide a valid token; if not, the interfaces will just do nothing. An example is given in Listing 7.4.

Listing 7.4: Example of an information-leak channel

```
public class Storage{
    public void QueryDatabase(SQLStat query, Token token){
        if(!this.checkToken(token)) return;
        else { /* Do the database query task and return result*/ }
    }
}
```

With the above token mechanism, even if the JavaScript code in the malicious main frame can still access the `QueryDatabase` interface, its invocation cannot lead to an actual database query. However, if the call is initiated by the `LivingSocial` web pages—which have the valid token—from the child frame, the invocation is legitimate and will lead to a query. Unfortunately, when the query results are returned to the caller by the app, using `loadUrl`, because of the frame confusion problem, the query results are actually passed to the main frame that belongs to the attacker. This creates an information-leak channel. Figure 7.1b, taken from (LHD+11), illustrates the attack.

(a) From the child frame

(b) From the main frame

Figure 7.1: Exploitation of Frame Confusion

The exploitation of the Frame Confusion vulnerability requires the attacker to affect any web domain in the main or a child iframes that has access to the JavaScript interfaces. This condition is achieved through:

- *The direct control of a web page.* In such a scenario, the attacker can be able either to take control over an existing web domain or to create an ad-hoc website, e.g., a malicious advertisement campaign.
- *The injection of malicious code in an existing web page.* In this case, the attacker can exploit a weakness in the communication protocol of the hybrid app, e.g., a clear-text communication or a misconfigured SSL connection, to mount a Man-In-The-Middle

attack[2] and inject malicious code in the loaded web pages.

It is worth noticing that the presence of other vulnerabilities in the JavaScript code, e.g., the adoption of JavaScript libraries with known vulnerabilities (OWA17) or the presence of XSS vulnerabilities (SJ14a; BGS11; BYZW17), further boosts the exploiting capabilities of the attacker.

7.2.4 Mitigations

As described above, the Frame Confusion allows violating the SOP by circumventing the sandbox of iframes. Unfortunately, despite the recent security mechanisms added in the WebView component, the Frame Confusion is still unfixed at any Android API level. Still, the web world offers an extra set of security mechanisms that are able to restrict the communication among the main frame and the child frames, thus preventing the Frame Confusion vulnerability, i.e.:

- the iframe `sandbox` attribute (w318), which enables a set of extra restrictions on any content hosted by an iframe and, among them, it allows blocking the execution of JavaScript code. Although effective in principle, this mechanism completely prevents the execution of *any* JavaScript code, thus limiting the functionalities of the web page.
- the Content Security Policy (CSP) (Con16) that allows for the definition of fine-grained restrictions on the execution of JavaScript code, including the possibility to define a set of trusted domains that are able to execute JavaScript, in a white-listing fashion. Although effective against the loading of an undesired web domain, the CSP cannot prevent the injection of the malicious code in a white-listed domain, thereby resulting ineffective against the Frame Confusion.

Furthermore, previous security mechanisms are not enabled by default, thus leaving the burden of their configuration to the developer. All in all, at the current state of the art, none of the existing security mechanisms are able to effectively prevent the Frame Confusion.

7.3 A Frame Confusion Detection Methodology

Despite the fact that the state of the art lacks a methodology for the automatic identification of the Frame Confusion vulnerability in Android, we have developed an automatic and

[2] `https://www.owasp.org/index.php/Man-in-the-middle_attack`

rigorous analysis flow. To achieve such result, we first define a blueprint of the Frame Confusion vulnerability, and then we build an analysis flow that can detect it automatically, by exploiting a fruitful combination of static and dynamic analysis techniques.

7.3.1 Vulnerability Blueprint

First of all, we need to select a set of features that *enable* the vulnerability. To this aim, we argue that a minimal set of such features is the following:

1. the app requires the `INTERNET` permission in order to access web domains using a WebView component;
2. the app uses at least a WebView that is configured to execute JavaScript code;
3. such WebView binds a Java class using the `addJavascriptInterface()` method;
4. in case of an app targeted to API level 17 or higher, such Java class must contain at least a method annotated with the `@JavascriptInterface`
5. the WebView loads at least a web page that contains one or more iframe elements;
6. such a web page does not enforce any mitigation technique among those described in the previous section.

7.3.2 Detection Algorithm

The Frame Confusion detection methodology can be summarized by the pseudocode listed in algorithm 1. Given a generic Android app in the APK format, the algorithm begins by retrieving a list of the Android permissions used by the app (row 1). If the list does not include the Internet permission, then the app cannot use the WebView component, and therefore it is marked as *not vulnerable* (rows 2-4). Otherwise, the algorithm computes the list of all the invoked methods of the app (row 5) in order to locate the presence of `setJavaScriptEnabled`, and `addJavascriptInterface` APIs.
If a `setJavaScriptEnabled` invocation (row 9) is recognized, the algorithm further investigates the flag parameter of the call (rows 10-14). A `True` value indicates that the WebView enables the execution of JavaScript and thus its object reference is retrieved (row 12) and included in the list of those that enable JavaScript (row 13).
Instead, the presence of a `addJavascriptInterface` indicates that a WebView component is configured to expose a bridge between Java and JavaScript. If this is the case, the algorithm extracts *i)* the WebView object from which the `addJavascriptInterface`

Algorithm 1: Frame Confusion Detection

Input : APK Package
Output: vulnerable, notVulnerable

```
1  listPermissions = getPermissionFromApk(app);
2  if "android.permission.INTERNET" not in listPermissions then
3  |  return notVulnerable;
4  methodsList = getAllInvMet(app);
5  JSWebView = list();
6  IWebView = list();
7  foreach method in methodsList do
8  |  if method.getName == "setJavaScriptEnabled" then
9  |  |  flagParam = getFlagParam(method);
10 |  |  if flagParam == True then
11 |  |  |  webViewObj = getInvObj(method);
12 |  |  |  JSWebView.add (webViewObj);
13 |  else if method.getName == "addJavascriptInterface" then
14 |  |  webViewObj = getInvObj(method);
15 |  |  interface = getInterfaceObj(method);
16 |  |  if getSDK(app) > 17 then
17 |  |  |  if containAnnotatedPubMet (interface) then
18 |  |  |  |  IWebView.add (webViewObj);
19 |  |  else if containPubMet(interface) then
20 |  |  |  IWebView.add (webViewObj);
21 if len (JSWebView) == 0 or len (IWebView) == 0 then
22 |  return notVulnerable;
23 if len (IWebView ∩ JSWebView) == 0 then
24 |  return notVulnerable;
25 resourceFiles = getAllResourceApk(app);
26 dumpWebStat = getStaticUrl(methodsList);
27 dumpWebDyn = getDynamicUrl(app);
28 filesToCheck = dumpWebDyn union resourceFiles union dumpWebStat;
29 vulnerablePages = list();
30 foreach file in filesToCheck do
31 |  if isHTMLfile(file) or isJSfile(file) then
32 |  |  if containIframe(file) then
33 |  |  |  if not containCSP(file) and not containSandboxAtt(file) then
34 |  |  |  |  vulnerablePages.add (file);
35 if len (vulnerablePages) > 0 then
36 |  return vulnerable;
37 return notVulnerable;
```

method is invoked (row 17), and *ii)* the Java object injected in the `JavascriptInterface` (row 18). After that, the algorithm needs to detect if the Java object injected in the interface contains public methods that can potentially be accessed from JavaScript code (rows 19-27). Moreover, in case of apps targeted to API level 17 or above, the public methods of the object need to be further annotated with the `@javascriptinterface` tag (rows 19-22). If the injected Java object contains methods accessible from JavaScript, then the corresponding WebView instance can be added to the list of those that expose potentially vulnerable interfaces (row 21 or row 25).
Next, if the analysis is not able to find at least a WebView - with JavaScript enabled - that contains a JavaScript interface with exposed Java methods, then the app is marked as not vulnerable (rows 29-34). Otherwise, the analysis collects from the Website collector module all the website pages accessed by the WebView that are i) included in the resources of the *.apk* package (row 35), ii) statically invoked by `loadURL` methods (row 36), and iii) dynamically reached during the execution of the app (row 37).
After that, the algorithm collects every website that uses at least an iframe element that loads an external page (either embedded in HTML pages or generated by JavaScript), and that does not enforce any of the mitigation techniques discussed in the previous section (rows 40-48). Finally, if the app loads at least one vulnerable website, it is marked as *vulnerable*. On the contrary, if the app uses the appropriate security mechanisms or does not use any iframe is marked as *non vulnerable*.

7.4 The FCDroid tool

FCDroid[3] implements the proposed detection methodology to automatically identify the presence of the Frame Confusion vulnerability in Android apps. In this section, we describe the implementation of FCDroid and its architecture, emphasizing the underlying tools and technologies.

7.4.1 Implementation

The Frame Confusion detection methodology poses several challenges in terms of implementation. Indeed, an automatic detection tool needs to:

1. achieve maximum coverage by detecting all possible app execution paths that may lead to vulnerability;

[3]FCDroid is available at `https://www.fcdroid.com`.

2. recognize the actual configuration of WebView components, which may *dynamically* enable JavaScript or define new interfaces;

3. analyze all the web pages loaded inside some potentially vulnerable WebViews, by also considering those loaded according to i) the user's input, and ii) the value of runtime variables.

To address such challenges, FCDroid uses static and dynamic analysis techniques. Static analysis can examine all possible execution paths. However, it is not able to predict variables or resources only available at runtime, for example, user-input or dynamically loaded code, therefore introducing false positives (i.e., the static analysis detects a defect, the defect exists, but it is impossible to trigger such malfunction at runtime). On the other hand, dynamic analysis examines the actual behavior of the app, so when it finds a defect it is a (desired) true positive. However, it has restricted coverage and it is time-limited, thus producing potential false negatives (i.e., the dynamic analysis does not detect an existing defect).
For those reason, FCDroid combines static and dynamic analysis techniques to overcome the limitations of both techniques and achieve more accurate detection results.

7.4.2 FCDroid Architecture

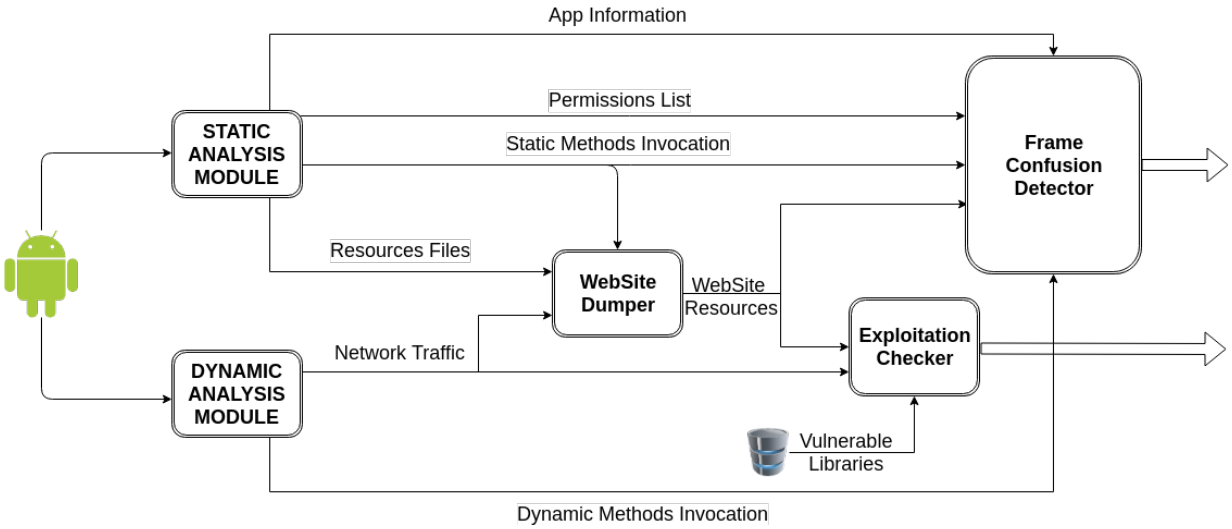


Figure 7.2: The FCDroid Architecture.

The FCDroid architecture, depicted in fig. 7.2, is composed by five main building blocks: the Static Analysis Module (SAM), the Dynamic Analysis Module (DAM), the WebSite Dumper (WD), the Frame Confusion Detector (FCD), and the Exploitation Checker (EC).

Static Analysis Module (SAM). The Static Analysis Module relies on *Apktool* (WT18) to disassemble the APK and translate the DEX bytecode into Smali (Gru19) language. In addition to that, SAM brings the resources contained in the app back to their original form, e.g., from binary compiled XML files into textual XML files. Then, the module extracts the list of permissions requested by the app and the target Android API level according to the content of the `AndroidManifest.xml` file. Finally, the SAM inspects each extracted Smali file in order to locate all the API method invocations of the WebView component. In detail, the module detects:

- `setJavaScriptEnabled` that enables the JavaScript code in a WebView object. If found, the SAM also extracts the variable containing the boolean flag passed as an argument;
- `addJavascriptInterface`, that creates a JavaScript interface object. In this case, the SAM retrieves the Java class of the injected object and the name assigned to the interface;
- `loadUrl` and `evaluateJavaScript`, that allows the loading of specific URLs or JavaScript code inside the WebView. In case, the module also extracts the URL address or the script code, if statically defined;

The collected pieces of information are then sent to the WebSite Dumper and the Frame Confusion Detector to continue the analysis.

Dynamic Analysis Module (DAM). The Dynamic Analysis Module is in charge of executing the app into a controlled testing environment in order to monitor the stimulation of the WebView components at runtime. To this aim, it installs the app into an Android Emulator and stimulates the app automatically, trying to explore its possible execution states. This allows the DAM to *i)* monitor the invocations of WebView-related API along with their execution parameters, and *ii)* intercept all the network traffic generated by the app. In order to stimulate the app automatically, the DAM relies on DroidBot (LYGC17), an open-source tool that can automatically explore the app UI and mimic the interaction with a user. Unlike many existing input generators that rely on static analysis and instrumentation of the app to generate inputs, DroidBot works in black-box mode, i.e., it does not need to know in advance the structure of the app, and it is resilient to obfuscation techniques. In order to keep track of API invocations, the DAM module provides the Android emulator with an ApiMonitor module. ApiMonitor, based on the Xposed[4] framework, allows the DAM to intercept and collect each method executed by the app during the analysis, saving its invocation and the value of parameters on a JSON file. Furthermore,

[4] `https://repo.xposed.info/`

the DAM module intercepts and stores all the network traffic generated by the app using the HTTP/HTTPs proxy *mitmproxy* (CHKc).

WebSite Dumper (WD). The WebSite Dumper module visits and retrieves all the websites invoked by the WebView components. Given the list of URLs accessed by app WebView components from the SAM and DAM modules, for each identified URL, the WebSite Dumper determines whether it refers to a local or a remote resource. In the first case, it collects and stores the static resource obtained by the app package. In the latter case, the WD module dumps the content of the remote website by downloading the web pages recursively, up to a maximum of 3 levels deep, by using the *wget* tool[5]. Finally, the module polishes the results and maintains only HTML and JavaScript files that will be inspected by both the FCD and the EC module.

Frame Confusion Detector (FCD). The Frame Confusion Detector module implements the core logic of FCDroid for the detection of the vulnerability. At first, FCD collects from the SAM the list of permissions of the app and verifies that the app requires the `INTERNET` permission. If so, the module analyzes the list of invoked APIs (both those statically extracted by the SAM module and those evaluated at runtime by the DAM) to verify the existence of at least a WebView instance that enables JavaScript and exposes a JavaScript interface. Furthermore, if an exposed interface is found, the FCD parses the class of the injected Java object to determine the existence of methods that can be accessed from JavaScript.
Finally, the FCD also needs to detect the amount of potentially-vulnerable webpages. To this aim, the module collects the websites dumped by the WD and checks whether a page contains at least an iframe element and does not enforce any mitigation techniques (i.e., the `Content-Security-Policy` meta tag in the HTML header or the sandbox attribute). At the end of the analysis, the FCD module marks the application as *vulnerable* or *not vulnerable*.

Exploitation Checker (EC). The Exploitation Checker is the module responsible for the detection of app configurations that can boost the exploitation of the Frame Confusion Vulnerability. In details, the EC can identify:

- *The adoption of unencrypted communication channels*, by analyzing the network traffic generated by the DAM module and by extracting the list of URLs that are accessed in plain HTTP.
- *The presence of buggy/vulnerable Javascript libraries* by relying on the RetireJS (Erl19) tool, which allows obtaining a list of known-to-be-vulnerable JavaScript libraries that are executed within the WebView.

[5] `https://www.gnu.org/software/wget/`

- *The presence of JavaScript code vulnerable to DOM-XSS attacks*[6], by including a customized implementation of JSPrime (Nis13). Such a tool inspects the JavaScript code in order to detect unsanitized input variables that could allow an attacker to execute arbitrary JavaScript code in the victim's WebView.

7.5 Experimental Results

We empirically assessed the reliability of the proposed methodology and its implementation in FCDroid, by systematically analyzing 50.000 apps [7]. The aforementioned apps have been downloaded from the Google Play Store in December 2018, and they were the top free Android apps ranked by the number of installations and average ratings according to Androidrank (andb). Our experiments were conducted using an Intel Xeon 3106@1.70 GHz, with 32GB RAM, running Ubuntu 18.04.

Frame Confusion Identification.

FCDroid discovered that 49.35% of apps (i.e., 24675 out of 50000) use at least a WebView component, thereby highlighting the wide adoption of such component in the Android ecosystem and the spread of Hybrid apps.
As shown in table 7.1, all the apps with at least one WebView component enable the execution of JavaScript, while 44.84% also attach (at least) a JavaScript interface, which contains a Java class with at least one method annotated with `@JavascriptInterface`. Therefore such methods can be invoked from the websites loaded inside the apps.
Furthermore, FCDroid inspected all the websites accessed by the hybrid apps obtaining the results described in table 7.2. In detail, 1.2% (87k/6.7M) of websites contain at least an iframe element; among those pages, the 27.96% include CPS policies while none of the visited pages enforces the sandbox attribute. Therefore, such findings indicate that most of the web pages that use iframe elements are potentially vulnerable to Frame Confusion. Finally, *our analysis identifies that 6.63% (i.e.,* 1637/24675*) of hybrid apps are potentially vulnerable to Frame Confusion.* To estimate the impact of such results on the Android users' community, we cross-referenced our findings with the Google Play Store meta-data, obtaining that the total sum of official installations for vulnerable apps is greater than 250.000.000.

FCDroid reported also the presence of other vulnerabilities in the app configuration that can make easier the exploitation of the Frame Confusion. As shown in table 7.3, 59.98% of vulnerable apps access websites using an insecure connection, i.e., plain HTTP, while 27.48% contain code vulnerable to XSS attacks. Finally, 79.96% of vulnerable apps include

[6] `https://www.owasp.org/index.php/DOM_Based_XSS`

[7] The complete list of experimental results is available at `https://www.fcdroid.com/results`.

Table 7.1: Statistics on the Frame Confusion blueprint.

	Percentage	**Ratio**
Internet Permission	96.45%	$48226/50k$
Use WebView	49.35%	$24675/50k$
JavaScript Enabled	49.35%	$24675/50k$
JavaScript Interface	44.84%	$22420/50k$

Table 7.2: Statistics on the web pages accessed by hybrid apps.

	Percentage	**Ratio**
Web pages with iframes	1.2%	$87k/6.7M$
Web pages with iframes and CSP	27.96%	$24108/87k$
Web pages with iframes and sandbox attribute	0%	$0/87k$

Table 7.3: Statistics on the exploiting conditions of vulnerable apps.

	Percentage	**Ratio**
Insecure connections	59.98%	982/1637
XSS vulnerabilities	27.48%	450/1637
Vulnerable JS libraries	79.96%	1309/1637

JavaScript libraries with known security vulnerabilities.

Limitations of FCDroid.

The dynamic analysis is limited to the public surface of the app (i.e., the one that does not require any user authentication) and executes in a predefined time-frame (i.e., 60 seconds). Furthermore, the implementation of FCDroid does not detect dynamically-generated iframe elements, like, i.e., those created at runtime by the JavaScript code.

7.6 From Frame Confusion to a Phishing Attack

In this section we present an attack against the "*YTN News*" app[8] which has been found vulnerable by FCDroid. At the moment of writing, YTN is available on the Google Play Store and has more than 1M downloads. We responsibly disclosed our findings to the app developers in January 2019. In this attack we exploit the a Frame Confusion vulnerability from a child frame in order to mount a phishing attack.

We manually reverse-engineered the app, and we found that it uses the WebView component to load a home page with several iframes. The iframe at the bottom of the web page

[8] `https://play.google.com/store/apps/details?id=com.estsoft.android.ytn`

loads an advertisement (step 1 in fig. 7.3). Our manual investigation confirms the FCDroid findings: the WebView uses a cleartext HTTP connection, JavaScript is enabled, and there is an interface exposed through the `addJavascriptInterface` method. The interface exposes different methods that can get some information about the device. One of these exposed methods is named `liveLogin`. This method has three parameters of type *string*; the first two are converted into integers and used to customize the WebView, while the last one is passed as a parameter to the `loadUrl` method without any kind of sanitization. Therefore, an attacker can easily inject arbitrary JavaScript code or a web page that will be loaded in the main frame.

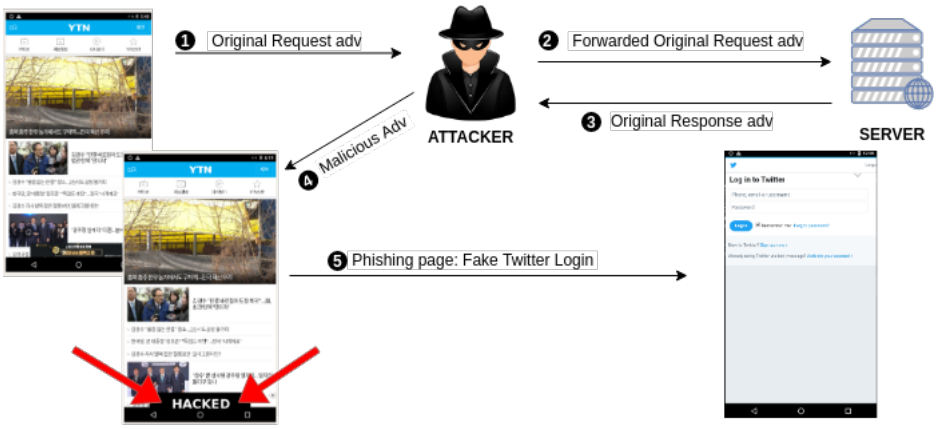


Figure 7.3: YTN News: Flow of the attack.

In order to exploit the vulnerability, the attacker must control an iframe. There are two approaches to achieve such a result: 1) if the attacker and the mobile device belong to the same network, the attacker can carry out a Man-in-the-Middle (MitM) attack; otherwise, 2) the attacker can create an ad-hoc advertising campaign. In our use case, we carried out a MitM attack, and we were able to control the advertisement (steps 2, 3, and 4 in fig. 7.3). For the sake of precision, since also the WebView uses a cleartext HTTP connection, the attacker can also target the main frame; however, in this example, we focused on a child frame, since we only aim to prove the exploitability of Frame Confusion. Thus, given the absence of any security mechanisms, we can access the exposed interface and exploit the Frame Confusion by invoking the `liveLogin` method with a URL pointing to our malicious web page (step 5 in fig. 7.3) containing a fake Twitter login page.

It is worth pointing out that the WebView is a promising vector attack for phishing, as there are no GUI components that prompt the actual URL and the transport protocol (e.g., HTTP/HTTPS), thereby making hard to distinguish between the legitimate Twitter website and a well-crafted phishing site. In Chapter 6, we have thoroughly discussed how the lack of graphical components for helping a user to unmask a fake GUI can be easily abused to mount powerful phishing attacks. As a final remark, this is just one among

a set of potential exploitation of Frame Confusion under such specific app settings. For instance, it is possible to download a large file (since the app has the `WRITE_EXTERNAL_STORAGE` permission) or continue to load the same pages within the WebView to carry out simple Denial-of-Service attacks.

7.7 Related Work

The steady growth of hybrid apps has attracted the attention of both academic and industrial security research communities. The main approaches for the security analysis of hybrid apps can be divided into static and dynamic. In static analysis methodologies, the hybrid app is analyzed according to its source (or binary) code without being executed. For instance, Lee at al. proposed HybriDroid (LDR16), a static analysis framework that examines the inter-communication between the native and the web counterpart of the app to identify development bugs or potential leaks of sensitive information. Other works, like (RCK17), (CLJW15), and (YXH+18) propose some detection methodologies for *code injection attacks* based on app-instrumentation or machine learning techniques. However, any of the proposed static analysis techniques suffer from the over-approximation of the app execution paths which drastically reduce the accuracy due to a high rate of false positives (LBP+17). On the other hand, dynamic analysis techniques aim at analyzing the security of the app runtime behavior in a controlled environment. The sole work based on dynamic analysis techniques for hybrid apps is BridgeTaint, proposed by Bai et al. (BWQ+19). BridgeTaint tracks sensitive data exchanged through the bridge and uses a cross-language taint mapping method to perform the taint analysis in both domains. Although dealing with the dynamic monitoring of the bridge between the Java and the JavaScript worlds, BridgeTaint is only focused on data analysis aimed at the identification of data leaks. Anyway, none of the approaches mentioned above is either able to identify the Frame Confusion vulnerability. The work proposed by Luo et al. (LHD+11) is the only research paper that explicitly discusses this vulnerability. Indeed, the authors – who also coined the term *"Frame Confusion"* – have also studied the security implications of the two-way interaction between the native and the web code in hybrid apps. Anyway, they focus only on detecting the security weaknesses of the WebView component and the JavaScript interfaces, as well as some statistics on the usage of the WebView API and JavaScript interfaces. Furthermore, their analysis is manual, and it has been carried out on a reduced dataset made by only 132 apps.
To the best of our knowledge, our methodology is the first approach allowing us to systematically detect the Frame Confusion vulnerability. Furthermore, the adoption of both static and dynamic analysis techniques allows overcoming the limitations of both approaches.

Chapter 8

Conclusions

In the last decade, we faced the rising of mobile devices as a fundamental tool in our everyday life. According to a blog post published in May 2019 of the Android director Stephanie Cuthbertson, there are more than two and a half billion active Android devices around the world today (Cut19). Android and its ecosystem of multi-purpose applications are pervasive in users' life and allow them to perform operations that would only be possible through a bulky Personal Computer. Besides, users heavily rely on them for storing their most sensitive information. This aspect pushes companies and prowlers to collect such a golden mine of personal data. Moreover, cyberwar has left the pages of science fiction and the desks of Pentagon war games to become a reality. It is nowadays clear that the threat of cyberattacks goes beyond petty vandalism, criminal profiteering, and even espionage to include the sort of physical-world disruption that was once possible to accomplish only with military attacks and terroristic sabotage.

I firmly believe that the purpose of academic research is to seek the truth with new verified knowledge, in order to enhance social development. It is for this reason that in this thesis, when possible, we have developed free tools to let our community evaluate and use the result of our methodologies. Furthermore, we have tackled such open-research problems from both the attacker and the defender points of view. The reason can be briefly explained by a quote from the military strategist Sun Tzu: "*Attack is the secret of defense; defense is the planning of an attack*" (Tzu14).

I conclude my thesis with my personal opinion, developed during the past three years. I argue that the neediest research areas of Android are the detection and prevention of malware and app plagiarism (which are often related to each other). The main issues in the Android ecosystem arise from the fact that the authenticity of the app developer is not ensured. The web-world is more mature from this point of view, thus providing – and forcing – a secure mapping (as discussed extensively in Chapter 6) will help not only

the end-users but also the app markets. Even though this solution is not a silver bullet, cross-checking the authenticity of a mobile app with a domain/website can only increase the security levels, helping fighting malware and detecting plagiarized apps.

Security considerations become especially highly important in scenarios where a malicious process on a mobile device may not just steal private data or inject malicious code, but can physically affect the user's safety or security. What policies and mechanisms allow granting that the information presented to a user is indeed trustworthy? How are the user's actions protected to ensure that no malicious app overtakes the controls without the user's awareness? These questions need reliable answers: autonomous vehicles, home automation, smart healthcare, and, in general, future research challenges must develop security solutions that take into account the environment in which devices operate.

On the other hand, it seems that system developers of modern operating systems learned from security mistakes of the past, and they made significant strides in blocking those threats right from the start. Zerodium is a famous American information security company whose primary business is acquiring premium zero-day vulnerabilities with functional exploits from security researchers and companies. At the time of writing, in their acquisition program (Zer19), Android has the highest payout (up to $\$2,500,000$) for a Zero-Click[1] full exploit chain with persistence, followed by iOS (up to $\$2,000,000$); instead the highest reward for the Desktops/Servers world is a Windows Remote Code Execution (up to $\$1,000,000$). Those huge rewards for exploiting mobile operating systems are a clear indication of the economic interests in the mobile ecosystem and how their kerneland is well defended. It is our duty to invent solutions for secure userland too.

[1] A Zero-Click technique takes over a device with no interaction from the user.

Bibliography

1Password. https://1password.com/, 2018.

Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. UiRef: analysis of sensitive user inputs in Android applications. In *WISEC*, 2017.

S. Arlot and A. Celisse. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79, 2010.

D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz. Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine. In *International Workshop on Ambient Assisted Living*, 2012.

D. Anguita, A. Ghio, L. Oneto, J. L. Reyes-Ortiz, and S. Ridella. A novel procedure for training l1-l2 support vector machine classifiers. In *International Conference on Artificial Neural Networks*, 2013.

D. Anguita, A. Ghio, L. Oneto, and S. Ridella. In-sample and out-of-sample model selection and error estimation for support vector machines. *IEEE Transactions on Neural Networks and Learning Systems*, 23(9):1390–1406, 2012.

Simone Aonzo, Giovanni Lagorio, and Alessio Merlo. Rmperm: A tool for android permissions removal. In *SECRYPT*, pages 319–326, 2017.

Yair Amit. 95.4 percent of all android devices are susceptible to accessibility clickjacking exploits. https://www.skycure.com/blog/95-4-android-devices-susceptible-accessibility-clickjacking-exploits/, 2016.

Yair Amit. "Accessibility Clickjacking" — The Next Evolution in Android Malware that Impacts More Than 500 Million Devices. https://www.skycure.com/blog/accessibility-clickjacking/, 2016.

Simone Aonzo, Alessio Merlo, Mauro Migliardi, Luca Oneto, and Francesco Palmieri. Low-resource footprint, data-driven malware detection on android. *IEEE Transactions on Sustainable Computing*, 2017.

Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. Phishing attacks on modern android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1788–1801. ACM, 2018.

Axelle Apvrille and Ruchna Nigam. Obfuscation in android malware, and how to fight back. *Virus Bulletin*, pages 1–10, 2014.

Android permissions. `https://developer.android.com/reference/android/Manifest.permission.html`. Accessed 2019-12-19.

Androidrank – open android market data. `https://www.androidrank.org/app/ranking?price=free`. Accessed 2019-12-19.

Androguard permission mapping. `https://github.com/androguard`, 2017. Accessed 2019-12-19.

Android: distribution of market shares at march 2017. `https://developer.android.com/about/dashboards`, 2019. Accessed 2019-12-19.

Efthimios Alepis and Constantinos Patsakis. Trapped by the UI: The Android Case. In *RAID*, 2017.

Apkmuzzle app. `https://github.com/packmad/ApkMuzzle`, 2017.

D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

Zarni Aung and Win Zaw. Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.

K. W. Y. Au, Y. F. Zhou, Z. Huang, and D: Lie. Pscout: analyzing the Android permission specification. In *ACM conference on Computer and communications security*, 2012.

Baddroids app. `https://github.com/packmad/BadDroids`, 2017.

Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 1101–1118, 2016.

Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium*, 2015.

Peter L Bartlett, Olivier Bousquet, and Shahar Mendelson. Local rademacher complexities. *Annals of Statistics*, 33(4):1497–1537, 2005.

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 *Proc. of the IEEE Symposium on Security and Privacy*, 2015.

A. P. Bianzino, C. Chaudet, D. Rossi, and J. L. Rougier. A survey of green networking research. *IEEE Communications Surveys Tutorials*, 14(1):3–20, First 2012. doi: 10.1109/SURV.2011.113010.00106.

O. Bousquet and A. Elisseeff. Stability and generalization. *Journal of Machine Learning Research*, 2:499–526, 2002.

A. Bakushinskiy and A. Goncharsky. *Ill-posed problems: theory and applications*. Springer Science & Business Media, 2012.

Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard – enforcing user requirements on Android apps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013.

Michael Backes, Sebastian Gerling, and Philipp Von Styprekowsky. A Local Cross-Site Scripting Attack against Android Phones. *Saarland University*, pages 1–6, 2011. URL: http://www.infsec.cs.uni-saarland.de/projects/android-vuln/, arXiv:arXiv:1106.4184v1.

A B Bhavani. Cross-Site Scripting attacks on Android webView. *International Journal of Computer Science and Network*, 2(2):1–5, 2013. arXiv:arXiv:1304.7451.

C. M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

P. L. Bartlett and S. Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3:463–482, 2002.

L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*. ACM, 2011.

Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355. ACM, 2016.

Tao Ban, Takeshi Takahashi, Shanqing Guo, Daisuke Inoue, and Koji Nakao. Integration of multi-modal features for android malware detection using linear svm. In *Information Security (AsiaJCIS), 2016 11th Asia Joint Conference on*, pages 141–146. IEEE, 2016.

Junyang Bai, Weiping Wang, Yan Qin, Shigeng Zhang, Jianxin Wang, and Yi Pan. BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. *IEEE Trans. Inf. Forensics Secur.*, 14(3):677–692, 2019. doi:10.1109/TIFS.2018.2855650.

Wenying Bao, Wenbin Yao, Ming Zong, and Dongbin Wang. Cross-site Scripting Attacks on Android Hybrid Applications. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy - ICCSP '17*, pages 56–61, New York, New York, USA, 2017. ACM Press. URL: http://dblp.uni-trier.de/db/conf/iccsp/iccsp2017.html, doi:10.1145/3058060.3058076.

T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 4.0]. URL: https://mitmproxy.org/.

Yen Lin Chen, Hahn Ming Lee, Albert B. Jeng, and Te En Wei. DroidCIA: A novel detection method of code injection attacks on HTML5-based mobile apps. *Proc. - 14th IEEE Int. Conf. Trust. Secur. Priv. Comput. Commun. Trust. 2015*, 1:1014–1021, 2015. doi:10.1109/Trustcom.2015.477.

N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. C. Mitchell. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *arXiv preprint arXiv:1808.04218*, 2018.

Luca Caviglione and Alessio Merlo. The energy impact of security mechanisms in modern mobile devices. *Network Security*, 2012(2):11 – 14, 2012. URL: http://www.sciencedirect.com/science/article/pii/S1353485812700156, doi:http://dx.doi.org/10.1016/S1353-4858(12)70015-6.

Content Security Policy. Content security policy, 2016. URL: https://content-security-policy.com,https://developers.google.com/web/fundamentals/security/csp/.

Mark Coppock. Your browser might be filling in hidden fields and giving away your secrets. https://www.digitaltrends.com/computing/browser-bug-can-fill-in-personal-information-in-hidden-fields/, 2017.

Apache Cordova. https://cordova.apache.org, 2019. URL: https://cordova.apache.org/.

Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. Peeking into your app without actually seeing it:{UI} state inference and novel android attacks. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 1037–1052, 2014.

Android custom permissions leak user data. [http://blog.trendmicro.com/trendlabs-security-intelligence/android-custom-permissions-leak-user-data/. Accessed 2019-12-19.

The custom permission problem. https://github.com/commonsguy/cwac-security/blob/master/PERMS.md. Accessed 2019-12-19.

Stephanie Cuthbertson. Sharing what's new in Android Q. https://www.blog.google/products/android/android-q-io/, 2019.

Davide Caputo, Luca Verderame, Simone Aonzo, and Alessio Merlo. Droids in disarray: Detecting frame confusion in hybrid android apps. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 121–139. Springer, 2019.

Erika Chin and David Wagner. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8267 LNCS, pages 138–159. 2014. URL: http://link.springer.com/10.1007/978-3-319-05149-9_9.

Dangerous permissions. https://developer.android.com/guide/topics/permissions/requesting.html#normal-dangerous. Accessed 2019-12-19.

Dashlane. https://www.dashlane.com/, 2018.

Benjamin Davis and Hao Chen. Retroskeleton: retrofitting Android apps. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 181–192. ACM, 2013.

Dexlib2. https://github.com/JesusFreke/smali/tree/master/dexlib2. Accessed 2019-12-19.

dexlib2. https://github.com/JesusFreke/smali/tree/master/dexlib2, 2017. Accessed 2019-12-19.

C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth. Preserving statistical validity in adaptive data analysis. In *Annual ACM Symposium on Theory of Computing*, 2015.

C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015.

Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *International Conference on Security and Privacy in Communication Systems*, pages 172–192. Springer, 2018.

Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 2017.

Artyom Dogtiev. Facebook Revenue and Usage Statistics. http://www.businessofapps.com/data/facebook-statistics/, 2018.

Mila Dalla Preda and Federico Maggi. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 13(3):209–232, 2017.

Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. *Mobile Security Technologies*, 2012(2):17, 2012.

Rachna Dhamija and J. D. Tygar. The Battle Against Phishing: Dynamic Security Skins. In *Proceedings of the Symposium On Usable Privacy and Security (SOUPS)*, pages 77–88, New York, NY, USA, 2005. ACM. doi:10.1145/1073001.1073009.

Joost de Valk. Why you should not use autocomplete. https://yoast.com/autocomplete-security/, 2013.

Nikolay Elenkov. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press, 2014.

William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

Oftedal Erlend. RetireJS - Scanner detecting the use of JavaScript libraries with known vulnerabilities, 2019. URL: `https://retirejs.github.io/retire.js/`.

B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

Ugo Fiore, Aniello Castiglione, Alfredo De Santis, and Francesco Palmieri. Exploiting battery-drain vulnerabilities in mobile smart devices. *IEEE Transactions on Sustainable Computing*, 2(2):90–99, 2017.

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. Android UI Deception Revisited: Attacks and Defenses. In *Proc. of Financial Cryptography and Data Security (FC)*, 2016.

Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 7–7, 2011.

Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. Hey, you, get off of my clipboard. In *International Conference on Financial Cryptography and Data Security*, pages 144–161. Springer, 2013.

Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057. IEEE, 2017.

S. Floyd and M. Warmuth. Sample compression, learnability, and the vapnik-chervonenkis dimension. *Machine learning*, 21(3):269–304, 1995.

Adrienne Porter Felt and David Wagner. Phishing on Mobile Devices. In *Proc. of IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*, 2011.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3:1157–1182, 2003.

Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*, 2015.

P. Germain, A. Lacasse, M. Laviolette, A. ahd Marchand, and Roy J. F. Risk bounds for the majority vote: From a pac-bayesian analysis to a learning algorithm. *Journal of Machine Learning Research*, 16(4):787–860, 2015.

Google play store. `https://play.google.com/`. Accessed 2019-12-19.

Google. Digital Asset Links. `https://developers.google.com/digital-asset-links/v1/getting-started`, 2017.

Google. OpenYOLO for Android. `https://openid.net/specs/openyolo-android-03.html`, 2017.

Google. Accessiblity Service. `https://developer.android.com/guide/topics/ui/accessibility/services`, 2018.

Google. Autofill Framework. `https://developer.android.com/guide/topics/text/autofill`, 2018.

Google. Enable automatic sign-in across apps and websites. `https://developers.google.com/identity/smartlock-passwords/android/associate-apps-and-sites/`, 2018.

Google. Google Smart Lock. `https://get.google.com/smartlock/`, 2018.

Google. Google Smart Lock - Associate apps and sites. `https://developers.google.com/identity/smartlock-passwords/android/associate-apps-and-sites`, 2018.

Google. Keeper. `https://keepersecurity.com/`, 2018.

Google. Safe Browsing. `http://www.google.com/transparencyreport/safebrowsing/`, 2018.

Google. Smart Lock for Passwords affiliation form. `https://docs.google.com/forms/d/e/1FAIpQLSc3FCn8ccGpgXd1jtLBVRlNJ6EhWQK50hNO5jT_9nuqHI79pg/viewform`, 2018.

Google. Verify Android App Links. `https://developer.android.com/training/app-links/verify-site-associations/`, 2018.

Google. Manifest.permission_group. `https://developer.android.com/reference/android/Manifest.permission{_}group`, 2019.

Ben Gruver. Smali - Assembler/Disassembler for the dex format, 2019. URL: http://github.com/JesusFreke/smali/.

Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 431–444. ACM, 2013.

Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *USENIX Security Symposium*, 2015.

Hao Hao, Vicky Singh, and Wenliang Du. On the effectiveness of api-level access control using bytecode rewriting in android. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 25–36. ACM, 2013.

Jiajun Hu. A Tale of Two Cities : How WebView Induces Bugs to Android Applications. 1:702–713, 2018. doi:10.1145/3238147.3238180.

G. B. Huang, D. H. Wang, and Y. Lan. Extreme learning machines: a survey. *International Journal of Machine Learning and Cybernetics*, 2(2):107–122, 2011.

Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M Chen, and Yogachandran Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.

JavascriptInterface. https://developer.android.com/reference/android/webkit/javascriptinterface, 2019. URL: https://developer.android.com/reference/android/webkit/JavascriptInterface.

Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code Injection Attacks on HTML5-based Mobile Apps. *Proc. 2014 ACM SIGSAC Conf. Comput. Commun. Secur. - CCS '14*, pages 66–77, 2014. URL: http://dl.acm.org/citation.cfm?doid=2660267.2660275, arXiv:arXiv:1505.06836v1, doi:10.1145/2660267.2660275.

Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. Android and Mr. Hide: fine-grained permissions

in Android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

Java native interface on android. `https://developer.android.com/training/articles/perf-jni.html`. Accessed 2019-12-19.

Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. A11y Attacks: Exploiting Accessibility in Operating Systems. In *Proc. of the Conference on Computer and Communications Security (CCS)*, 2014.

E. Kirda and C. Kruegel. Protecting users against phishing attacks with AntiPhish. In *Proceedings of the Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 517–524 Vol. 2, July 2005. `doi:10.1109/COMPSAC.2005.126`.

S. S. Keerthi and C. J. Lin. Asymptotic behaviors of support vector machines with gaussian kernel. *Neural computation*, 15(7):1667–1689, 2003.

LastPass. `https://www.lastpass.com/`, 2018.

Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.*, 88:67–95, 2017. `doi:10.1016/j.infsof.2017.04.001`.

Sungho Lee, Julian Dolby, and Sukyoung Ryu. HybriDroid: static analysis framework for Android hybrid applications. *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng. - ASE 2016*, pages 250–261, 2016. URL: `http://dl.acm.org/citation.cfm?doid=2970276.2970368`.

Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security Symposium*, pages 465–479, 2014.

Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android system. *Proceedings of the 27th Annual Computer Security Applications Conference on - ACSAC '11*, page 343, 2011. URL: `http://dl.acm.org/citation.cfm?doid=2076732.2076781`, `doi:10.1145/2076732.2076781`.

M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml"`. Accessed 2019-12-19.

Literadar. `https://github.com/pkumza/LiteRadar`., 2017. Accessed 2019-12-19.

Xing Liu and Jiqiang Liu. A two-layered permission-based android malware detection scheme. In *Mobile cloud computing, services, and engineering (mobilecloud), 2014 2nd ieee international conference on*, pages 142–148. IEEE, 2014.

G. Lever, F. Laviolette, and F. Shawe-Taylor. Tighter pac-bayes bounds through distribution-dependent priors. *Theoretical Computer Science*, 473:4–28, 2013.

J. Langford and D. McAllester. Computable shell decomposition bounds. *Journal of Machine Learning Research*, 5:529–547, 2004.

Lookout. Trojanized adware family abuses accessibility service to install whatever apps it wants. `https://blog.lookout.com/blog/2015/11/19/shedun-trojanized-adware/`, 2015.

Spandas Lui. Accessibility Service Helps Malware Bypass Android's Beefed Up Security. `http://www.lifehacker.com.au/2016/05/accessibility-service-helps-malware-bypass-androids-beefed-up-security/`, 2016.

Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, Xiaofeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. Unleashing the Walking Dead : Understanding Cross-App Remote Infections on Mobile WebViews. *Ccs*, pages 829–844, 2017. URL: `https://acmccs.github.io/papers/p829-liA.pdf`, `arXiv:1708.08510`, `doi:10.1145/3133956.3134021`.

Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: A lightweight UI-guided test input generator for android. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pages 23–26, 2017. `doi:10.1109/ICSE-C.2017.8`.

Qilin Li and Mingtian Zhou. The survey and future evolution of green computing. In *Proceedings of the 2011 IEEE/ACM International Conference on Green Computing and Communications*, GREENCOM '11, pages 230–233, Washington, DC, USA, 2011. IEEE Computer Society. URL: `http://dx.doi.org/10.1109/GreenCom.2011.47`, `doi:10.1109/GreenCom.2011.47`.

Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

Alessio Merlo, Mauro Migliardi, and Luca Caviglione. A survey on energy-aware security mechanisms. *Pervasive and Mobile Computing*, 24:77 – 90, 2015. Special Issue on Secure Ubiquitous Computing. URL: `http://www.sciencedirect.com/science/article/pii/S1574119215000929`, `doi:http://dx.doi.org/10.1016/j.pmcj.2015.05.005`.

Monkey. `https://developer.android.com/studio/test/monkey.html`, 2017. Accessed 2019-12-19.

Nishant Das Patnaik, Sarathi Sabyasachi Sahoo . JSPrime, 2013. URL: https://dpnishant.github.io/jsprime/.

Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010.

Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A View to a Kill: WebView Exploitation. *Leet*, 2013. URL: http://publik.tuwien.ac.at/files/PubDat{_}223415.pdf.

Marcus Niemietz and Jörg Schwenk. UI Redressing Attacks on Android devices. *Black Hat Abu Dhabi*, 2012.

Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and Xiaofeng Wang. UIPicker: User-Input Privacy Identification in Mobile Applications. In *USENIX Security Symposium*, 2015.

L. Oneto, S. Ridella, and D. Anguita. Learning hardware-friendly classifiers through algorithmic stability. *ACM Transaction on Embedded Computing*, 15(2):23:1–23:29, 2016.

OWASP. Using components with known vulnerabilities, 2017. URL: https://www.owasp.org/index.php/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities.

J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

Francesco Palmieri, Sergio Ricciardi, and Ugo Fiore. Evaluating network-based dos attacks under the energy consumption perspective: new security issues in the coming green ict area. In *Broadband and Wireless Computing, Communication and Applications (BWCCA), 2011 International Conference on*, pages 374–379. IEEE, 2011.

Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. 2017. URL: http://arxiv.org/abs/1709.05690, arXiv:1709.05690.

L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, and A. Verri. Are loss functions all the same? *Neural Computation*, 16(5):1063–1076, 2004.

Android reflection. https://developer.android.com/reference/java/lang/reflect/package-summary.html. Accessed 2019-12-19.

Requesting permissions. https://developer.android.com/guide/topics/permissions/requesting.html, 2017. Accessed 2019-12-19.

Nikhilesh Reddy, Jinseong Jeon, J Vaughan, Todd Millstein, and J Foster. Application-centric security policies on unmodified android. *UCLA Computer Science Department, Tech. Rep*, 110017, 2011.

Rmperm tool. https://github.com/CSecLab/RmPerm., 2017. Accessed 2019-12-19.

Ricardo Martín Rodríguez. How to take advantage of Chrome autofill feature to get sensitive information. https://blog.elevenpaths.com/2013/10/how-to-take-advantage-of-chrome.html, 2013.

C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. MIT press Cambridge, 2006.

E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards Discovering and Understanding Task Hijacking in Android. In *Proc. of USENIX Security Symposium*, 2015.

Jürgen S. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

B. Scholkopf. The kernel trick for distances. *Advances in neural information processing systems*, 2001.

Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *2010 International Conference on Computational Intelligence and Security*, pages 329–333. IEEE, 2010.

Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google android: A comprehensive security assessment. *IEEE Security & Privacy*, 8(2):35–44, 2010.

B. Schölkopf, R. Herbrich, and A. J. Smola. A generalized representer theorem. In *International Conference on Computational Learning Theory*, 2001.

Stanzein Sedol and Rahul Johari. Survey of Cross-site Scripting Attack in Android Apps. *International Journal of Information and Computation Technology*, 4(11):1079–1084, 2014.

Ben Stock and Martin Johns. Protecting Users Against XSS-based Password Manager Abuse. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 183–194. ACM, 2014.

David Silver, Suman Jana, Dan Boneh, Eric Yawei Chen, and Collin Jackson. Password Managers: Attacks and Defenses. In *USENIX Security Symposium*, pages 449–464, 2014.

Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 56–65. ACM, 2014.

F. Schöpfer, A. K. Louis, and T. Schuster. Nonlinear iterative methods for linear ill-posed problems in banach spaces. *Inverse Problems*, 22(1):311, 2006.

Junliang Shu, Juanru Li, Yuanyuan Zhang, and Dawu Gu. Android app protection via interpretation obfuscation. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 63–68. IEEE, 2014.

L. Swersky, H. O. Marques, J. Sander, R. J.G.B. Campello, and A. Zimek. On the evaluation of outlier detection and one-class classification methods. In *IEEE International Conference on Data Science and Advanced Analytics*, 2016.

Empirical assessment with rmperm. `https://github.com/CSecLab/BatchRmPerm/tree/master/dbDump`, 2017. Accessed 2019-12-19.

B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez. PUMA: Permission Usage to detect Malware in Android. In *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions*, 2013.

Statista. Percentage of all global web pages served to mobile phones from 2009 to 2018. `https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/`, 2018.

Number of available applications in the google play store from december 2009 to june 2019. `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`, 2019. Accessed 2019-12-19.

Statcounter. Mobile Operating System Market Share Worldwide. `https://gs.statcounter.com/os-market-share/mobile/worldwide`, 2019. Accessed 2019-12-19.

J. Shawe-Taylor and N. Cristianini. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

A. N. Tikhonov and V. I. A. Arsenin. *Solutions of ill-posed problems*. Halsted Press, New York, 1977.

J. Tang, C. Deng, and G. B. Huang. Extreme learning machine for multilayer perceptron. *IEEE transactions on neural networks and learning systems*, 27(4):809–821, 2016.

The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface. volume 9379, pages 126–138. 2015. URL: `http://link.springer.com/10.1007/978-3-319-26096-9_13`.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

Sun Tzu. The art of war. In *Strategic Studies*, pages 86–110. Routledge, 2014.

V. N. Vapnik. *Statistical learning theory*. Wiley New York, 1998.

Dinesh Venkatesan. Malware may abuse android's accessibility service to bypass security enhancements. http://www.symantec.com/connect/blogs/malware-may-abuse-android-s-accessibility-service-bypass-security- enhancements, 2016.

Virus total. `https://www.virustotal.com/`. Accessed 2019-12-19.

Cometbot – obfuscated with code manipulation. `https://www.virustotal.com/gui/file/356a5c92670b825d0bf3e2e927ce3f2ff3a407ad1b6e91119a8056391e665b0c/`. Accessed 2019-12-19.

Cometbot – obfuscated with encryption. `https://www.virustotal.com/gui/file/cc394ba746f55630d97f06df89d851438c866c6179e39eb5d706969ca7a40de0/`. Accessed 2019-12-19.

Cometbot – obfuscated with reflection. `https://www.virustotal.com/gui/file/feddd8ac2ce246105c5df050061ea5dad8cb5da8411010646f3f9eb8dbbc1b44/`. Accessed 2019-12-19.

Cometbot – obfuscated with renaming. `https://www.virustotal.com/gui/file/e58332461b8151e842369a635fa01822289f45128c5d5afcc981c7cb2ba170d4/`. Accessed 2019-12-19.

Cometbot – obfuscated with trivial techniques. https://www.virustotal.com/gui/file/1fe6ad3bd534bf9f42cbdefa66e99db1760bb110d978dfb28517bd61fb5e9a16/. Accessed 2019-12-19.

Cometbot original. https://www.virustotal.com/gui/file/642da73bc4c78004304dfed2e6e704ebb352ff9f1db19a19cc2296c86164e723/. Accessed 2019-12-19.

w3. Sandbox attribute, 2018. URL: https://www.w3.org/wiki/Html/Elements/iframe.

Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 55–63, 2016.

WebViewSecurity. https://android-developers.googleblog.com/2017/06/whats-new-in-webview-security.html, 2017. URL: https://android-developers.googleblog.com/2017/06/whats-new-in-webview-security.html.

WebViewSafeBrowsing. https://developer.android.com/guide/webapps/managing-webview, 2018. URL: https://developer.android.com/guide/webapps/managing-webview.

WebSetting. https://developer.android.com/reference/android/webkit/websettings, 2019. URL: https://developer.android.com/reference/android/webkit/WebSettings.

WebView. https://play.google.com/store/apps/details?id=com.google.android.webview&hl=en, 2019. URL: https://play.google.com/store/apps/details?id=com.google.android.webview&hl=en.

Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 222–235. ACM, 2018.

Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

Ryszard Wiśniewski and Connor Tumbleson. Apktool A tool for reverse engineering Android apk files, 2018. URL: http://ibotpeaches.github.io/Apktool/.

Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552, 2012.

Ruibo Yan, Xi Xiao, Guangwu Hu, Sancheng Peng, and Yong Jiang. New deep learning method to detect code injection attacks on hybrid applications. *Journal of Systems and Software*, 137:67–77, 2018. URL: `https://doi.org/10.1016/j.jss.2017.11.001`, `doi:10.1016/j.jss.2017.11.001`.

Zerodium. Exploit Acquisition Program. `https://zerodium.com/program.html`, 2019.

H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014.

Min Zheng, Patrick PC Lee, and John CS Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 82–101. Springer, 2012.

C. Zhang and Y. Ma. *Ensemble machine learning*. Springer, 2012.

Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM, 2013.

Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming information-stealing smartphone applications (on Android). In *International conference on Trust and trustworthy computing*, pages 93–107. Springer, 2011.

Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.