The Dilemma of Security Smells and How to Escape It

Inaugural dissertation
of the Faculty of Science,
University of Bern

presented by

Pascal Gadient

from Flums-Grossberg SG, Switzerland

Supervisors of the doctoral thesis:

Prof. Dr. Oscar Nierstrasz
University of Bern, Switzerland

Prof. Dr. Mohammad Ghafari
University of Clausthal, Germany

The Dilemma of Security Smells and How to Escape It

Inaugural dissertation
of the Faculty of Science,
University of Bern

presented by
Pascal Gadient
from Flums-Grossberg SG, Switzerland

Supervisors of the doctoral thesis:

Prof. Dr. Oscar Nierstrasz
University of Bern, Switzerland

Prof. Dr. Mohammad Ghafari
University of Clausthal, Germany

Accepted by the Faculty of Science.

Bern, 10 May 2022

The Dean
Prof. Dr. Zoltan Balogh

Submitted to:
Institute of Computer Science
University of Bern
Hochschulstrasse 6
CH-3012 Bern

Typesetting:
Overleaf.com
LaTeX, LaTeX Project Public License (LPPL)
UZH/USZ habilitation template from J. von Spiczak, CC BY 4.0
ETH CADMO template from F. Mousset and H. Einarsson

Imprint:
Lulu Press, Inc., NC, USA
https://www.lulu.com

Availability:
First edition, May 2022

"Repairing old faults often costs more than acquiring new ones."

Wiesław Leon Brudziński (1920 – 1996)

Polish writer, satirist, and aphorist

Acknowledgment

I would like to thank for the great supervision by Prof. Dr. Oscar Nierstrasz who always remained very calm even during heated paper discussions within the group and let me work with students on my own ideas. Moreover, I would like to thank Prof. Dr. Mohammad Ghafari for his expertise and out-of-the-box thinking in several of my works. I am also very grateful to be able to work in such a great team over the last four years together with Pooja, Nataliia, Reza, Nitish, and Manuel.

A big thank you is also due to my friends and family, especially my fiancée Gabi and my mother Edith, who particularly suffered from my lack of sleep as deadlines approached. Finally, this work was written in memory of my father Jakob and my fiancée's father Sergio who passed away much too soon.

Abstract

A single mobile app can now be more complex than entire operating systems ten years ago, thus security becomes a major concern for mobile apps. Unfortunately, previous studies focused rather on particular aspects of mobile application security and did not provide a holistic overview of security issues. Therefore, they could not accurately understand the fundamental flaws to propose effective solutions to common security problems.

In order to understand these fundamental flaws, we followed a hybrid strategy, *i.e.*, we collected reported issues from existing work, and we actively identified security-related code patterns that violate best-practices in software development. Based on these findings, we compiled a list of security smells, *i.e.*, security issues that could potentially lead to a vulnerability.

As a result, we were able to establish comprehensive security smell catalogues for Android apps and related components, *i.e.*, inter-component communication, web communication, app servers, and HTTP clients. Furthermore, we could identify a dilemma of security smells, because most security smells require unique fixes that increase the code complexity, which in return increases the risk of introducing more security smells. With this knowledge, we investigate the interaction of our security smells with the 192 Mitre CAPEC attack mechanism categories of which the majority could be mitigated with just a few additional security measures. These measures, a String class with behavior and the more thorough use of secure default values and paradigms, would simplify the application logic and at the same time largely increase security if implemented appropriately.

We conclude that application security has to focus on the String class, which has not largely changed over the last years, and secure default values and paradigms since they are the smallest common denominator for a strong foundation to build resilient applications. Moreover, we provide an initial implementation for a String class with behavior, however the further exploration remains future work. Finally, the term "security smell" is now widely used in academia and eases the communication among security researchers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our society is dependent on smart devices that run a complex OS (operating system) and apps, which are highly connected and support a plethora of different sensors. Such devices are used to perform critical operations such as voting, e-finance, unlocking doors and cars, or to show and validate immunity certificates for Corona. Moreover, they allow users to communicate with others, browse the web, or consume digital media. Americans spent an average of 4.1 hours a day on mobile devices in 2021, more time than they spent watching TV.[1]

1.1 Motivation

The possibilities for misuse of smart devices are therefore very diverse and security has become a major concern. In fact, the protection of installed apps is very challenging since mobile app development is much more complex than traditional software development: every mobile platform maintains a different software architecture that is not interoperable, and the devices have countless unique hardware capabilities such as gyroscopes, lidar scanners, cameras, and fingerprint sensors. In fact, unlike desktop applications a mobile app must be at any time aware of the environment, *e.g.*, whether the OS has closed some app views to save battery. On top of this, apps have many opportunities to collect personal data and they often access servers from third parties, which are not maintained by the app vendors.

The global increase in code complexity indeed represents a problem, particularly for mobile apps. Shin *et al.* showed that complexity metrics can be successfully used to assess the likelihood of vulnerabilities [88].

[1] Forbes: Another blockbuster year for digital economy, `https://www.forbes.com/sites/roberthart/2022/01/12/record-38-trillion-hours-spent-on-mobile-apps-during-2021-in-another-blockbuster-year-for-digital-economy/?sh=715d48a32a42`, *accessed on 28-FEB-2022*

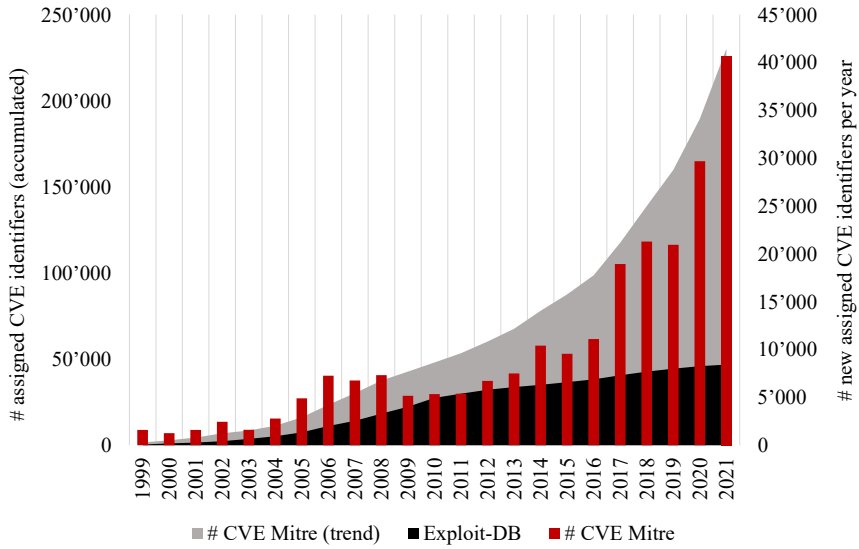


Figure 1.1: Annual increase in reported security issues and exploits

This suggests that more complex code is prone to security issues. Consequently, it is no surprise that we can see a major increase of recently reported security vulnerabilities in CVE Mitre,[2] *i.e.*, the most commonly referred vulnerability database funded by the United States National Cyber Security Division, and Exploit-DB,[3] *i.e.*, a database that leverages data from CVE Mitre, which contains exploits for vulnerabilities and is maintained by a security agency that provides consulting services.

Figure 1.1 shows the numbers of reported security issues and exploits. We collected the data in early September 2021 and thus the final numbers for the year 2021 will be higher than those presented in the plot where we shaded that particular section. The area plot uses the accumulated data, *i.e.*, the total of reported elements until the specified year in the y-axis. In contrast, the line plot indicates the new submitted reports per year, but only for CVE Mitre since it is the major data provider used for the reporting of vulnerabilities. We can clearly see that the number of reported issues has exponential tendencies, and that the number of exploits released to the public through Exploit-DB is constantly growing. In other words, the number of annually reported security issues in software has increased from about 11 000 to more than 40 000 within the last five years, which is an increase of 267%.

Unfortunately, the situation may be even worse for mobile apps. Figure 1.2 shows the numbers of reported security issues related to Android.

[2]CVE Mitre website, `https://cve.mitre.org`, *accessed on 28-FEB-2022*
[3]Exploit-DB website, `https://www.exploit-db.com`, *accessed on 28-FEB-2022*

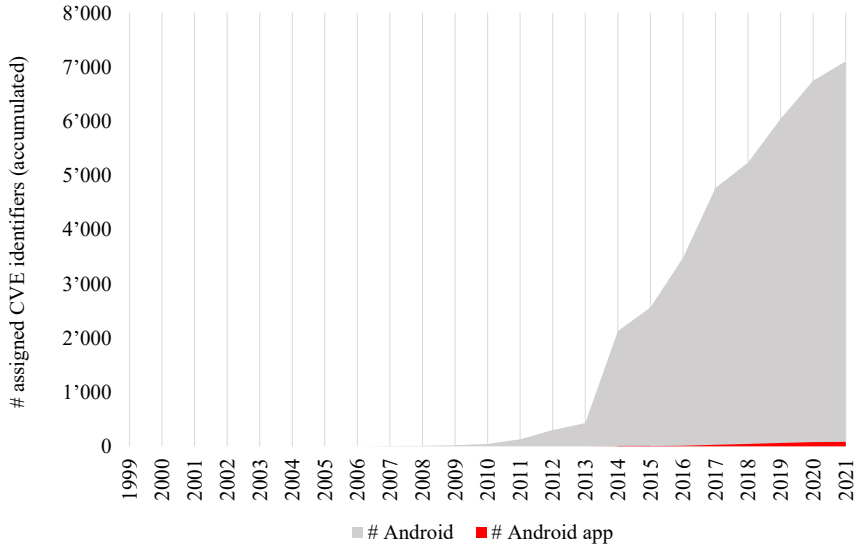


Figure 1.2: Annual increase in reported Android security issues and exploits

We use the same data and visualization as before, however we only counted reports that either contained the term "android", where we assume a bug report is related to the Android OS, or "android app," where we assume a bug report is related to an Android app. On the one hand, we can see that the Android OS received many bug reports and the climax seems to be reached within the next few years. On the other hand, however, there is only a *very low number* of vulnerabilities *represented in red* that have been reported for a few apps although there exist currently more than 2.5 million Android apps in the Google Play store.[4] *Therefore, we should ask ourselves if most of these mobile applications are secure, or if perhaps it is more due to a lack of interest from the security community.*

In this work, we will investigate common problems in Android apps and based on these findings we will determine effective measures against such threats.

1.2 Thesis Statement

The aim of this thesis is to improve the mobile app security with potential applications to desktop application security. We formulate our thesis as follows:

[4] AppBrain: Number of Android apps on Google Play, `https://www.appbrain.com/stats/number-of-android-apps`, *accessed on 28-FEB-2022*

Thesis Statement

The fundamental mitigation of security vulnerabilities in mobile apps requires a holistic understanding of security. This work achieves that by i) defining the notion of "security smell," ii) gathering common security smells in the context of mobile apps, and iii) classifying the identified smells into threat classes. With this knowledge, effective remediation strategies can be identified, implemented, and evaluated.

In the following we present all the investigated research questions (RQs) of this thesis separated by chapters. The presented RQs span eight chapters of this thesis.

Chapter 4: Security Smells in Android
- RQ 4.1: *What are the security code smells in Android apps?*
- RQ 4.2: *How prevalent are security smells in benign apps?*
- RQ 4.3: *To which extent identifying security smells facilitates detecting vulnerabilities?*

Chapter 5: Security Smells in Android ICC
- RQ 5.1: *What are the known ICC security code smells?*
- RQ 5.2: *How prevalent are the smells in benign apps?*
- RQ 5.3: *To which extent does identifying security smells facilitate detection of security vulnerabilities?*

Chapter 6: Security Smells in the Web Communication of Mobile Apps
- RQ 6.1: *Which API frameworks are used in Android mobile apps, and what is the nature of the data that apps transmit through these frameworks?*
- RQ 6.2: *What security smells are present in web communication?*

Chapter 7: Security Smells in Mobile App Servers
- RQ 7.1: *What is the prevalence of the server side security smells in the web communication of mobile apps?*
- RQ 7.2: *What is the relationship between security smells and app server maintenance?*

Chapter 8: Security Smells in Mobile App HTTP Clients
- RQ 8.1: *What is the support of the most common security-related HTTP header fields in existing HTTP clients?*

Chapter 9: Effective Holistic Security for Mobile Apps
- RQ 9.1: *Which security smells enable what attack mechanisms?*

RQ 9.2: *What are holistic security strategies that can effectively prevent attack mechanisms?*

Chapter 10: Default Values and Practices to Improve Application Security

RQ 10.1: *What are examples of default values and practices that could greatly improve application security?*

Chapter 11: A String-based Framework to Improve Application Security

RQ 11.1: *What are the restrictions when using an instrumented Java String class with existing code?*

RQ 11.2: *Can an instrumented String class offer protection against data leaks and remote code execution, and what are the security risks using such a technique?*

1.3 Contributions

We present nine distinct contributions with the aim to improve application security from different aspects.

- We review related research, and identify avoidable vulnerabilities in Android-run devices and the security code smells that indicate their presence. In particular, we explain the vulnerabilities, their corresponding smells, and we discuss how they could be eliminated or mitigated during development. Moreover, we develop a lightweight static analysis tool and discuss the extent to which it successfully detects several vulnerabilities in about 46 000 apps hosted by the official Android market.

- We review related research, and identify avoidable ICC (inter-component communication) vulnerabilities in Android-run devices and the security code smells that indicate their presence. We explain the vulnerabilities and their corresponding smells, and we discuss how they can be eliminated or mitigated during development. We present a lightweight static analysis tool called *AndroidLintSecurityChecks* on top of Android Lint that analyzes the code under development and provides just-in-time feedback within the IDE (integrated development environment) about the presence of such smells in the code. Moreover, with the help of this tool we study the relevance of security code smells in more than 700 open-source apps, and manually inspect around 15% of the apps to assess the extent to which identifying such smells uncovers ICC security vulnerabilities.

- We analyze the web communications found in mobile apps from the perspective of security. We first manually study 160 Android apps

to identify the commonly-used communication libraries, and to understand how they are used in these apps. We then develop a tool called *Jandrolyzer* to statically identify web API URLs (uniform resource locators) used in the apps, and restore the JSON (JavaScript object notation) data schemas including the type and value of each parameter.

- We analyze the prevalence of six security smells in mobile app servers, and we investigate the consequence of these smells from a security perspective. We use an existing dataset that includes 9 714 distinct URLs used in 3 376 Android mobile apps. We exercise these URLs twice within 14 months and investigate the HTTP (hypertext transfer protocol) headers and bodies. We find that more than 69% of tested apps suffer from three kinds of security smells, and that unprotected communication and misconfigurations are very common in servers. Moreover, source code and version leaks, or the lack of update policies expose app servers to security risks.

- We explore the adoption of security-related HTTP headers in mobile app communication by querying 9 714 distinct URLs that are used in 3 376 apps and collected each server's response information. We discover that support for secure HTTP header fields is absent in all major HTTP clients, and it is barely provided with any server response. Based on these results, we discuss opportunities for improvement particularly to reduce the likelihood of data leaks and arbitrary code execution.

- We manually investigated the impact of our 51 security smells on 192 attack mechanisms of the CAPEC taxonomy, which led to 9 792 combinations that we considered. We found that insecure algorithms, the abuse of existing functionality, data leaks, and user deception are the four major threats when using Android, and we elaborate strategies against them. That is, we see most potential in secure default values and safe practices to prevent feature misuse in the Android ecosystem. We further realized that string variables need increased protection, because they are responsible for most issues that relate to the employment of probabilistic methods and the injection of unexpected items.

- We reviewed every reported security smell and found that eight smells (16%) could be addressed with more secure default values, and that 36 smells (71%) could be addressed with safer practices. In fact, we only see for seven smells (14%) no potential in such measures, however they can be addressed using a better control of data.

- We present a flexible framework that can effectively prevent data leaks and other threats even if developers are inexperienced or un-

aware of potential security implications when they apply changes to their own code. Using this framework, we could successfully prevent data leaks, leverage off-memory encryption, or perform taint and data flow analyses even without changing the existing application logic.

1.4 Outline

In chapter 2 we explain the frequently used terms, and in chapter 3 we explain the current state of application security. In chapters four to eight we present security smells in Android (chapter 4), Android ICC (chapter 5), web communication (chapter 6), app servers (chapter 7), and HTTP clients (chapter 8). In chapter 9 we elaborate which security smells enable what attack mechanisms and identify characteristics of effective holistic remediation strategies. With this knowledge, we are able to propose effective remedies in chapter 10 and in chapter 11, before we present future work, visible impact, and the conclusions in chapter 12.

Chapter 2

Background

In this chapter we provide the necessary background to follow this thesis. In more detail, we elaborate the Android OS, the web communication, and we discuss the term *security smell*.

Declaration of Content Reuse

The content of this chapter contains elements from sections that correspond to paper submissions, *i.e.*, *Security Code Smells in Android ICC* (chapter 5) and *Security Smells in HTTP Headers* (chapter 8).

2.1 Android

The Android OS has become the most prevalent OS today with a mobile OS market share of more than 70% in early 2022,[1] and is used far beyond smartphones, *e.g.*, in cars, tvs, watches, and internet of things (IoT) devices.[2] Android was initially designed for "smart" digital cameras, which can communicate with other devices.[3] However, after Android was acquired by Google, the intention behind the operating system changed: it was to become an "open-source handset solution" with support for third-party apps. Such Android apps can be installed manually or with app stores, *e.g.*, the Google Play Store. An Android app consists of an .apk

[1] statcounter: mobile OS market share worldwide, `https://gs.statcounter.com/os-market-share/mobile/worldwide`, *accessed on 21-MAR-2022*

[2] Google documentation: Android devices, `https://developer.android.com/about`, *accessed on 21-MAR-2022*

[3] PCWorld: Android founder: we aimed to make a camera OS, `https://www.pcworld.com/article/451350/android-founder-we-aimed-to-make-a-camera-os.html`, *accessed on 28-FEB-2022*

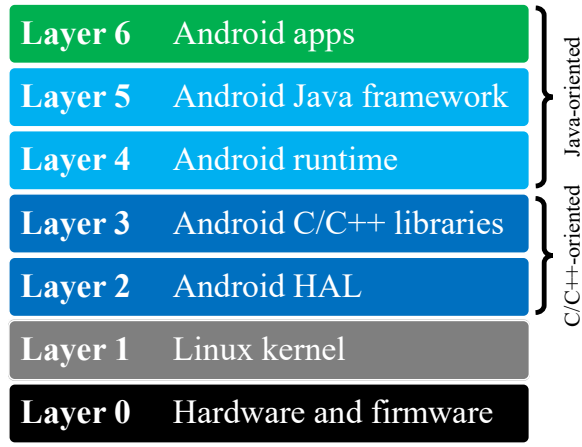


Figure 2.1: The Android architecture

file that contains the compiled byte code and, optionally, native code, additional metadata, and resource files. Moreover, each new major Android release introduces new Android application framework APIs, and to easily distinguish between the different Android releases a new "API level" is introduced, *i.e.*, an integer that corresponds to a particular Android OS release. Android apps are always optimized for a particular Android OS release, *i.e.*, the desired API level is specified in their configuration files.

2.1.1 Architecture

According to Google's documentation[4,5], the Android architecture comprises several layers with distinct responsibilities that are shown in Figure 2.1:

- **Layer 0:** Encompasses the hardware and other low-level machine code of which some can be implemented in silicon, *e.g.*, firmware that integrates logic for the hardware initialization to make it capable of executing compiled code.
- **Layer 1:** A customized Linux kernel that includes some system drivers.
- **Layer 2:** The hardware abstraction layer (HAL) that ensures uniform driver interfaces across different hardware platforms. There exist different HALs for different purposes that must be followed,

[4]Google documentation: Android architecture, `https://source.android.com/devices/architecture`, *accessed on 21-MAR-2022*

[5]Google documentation: platform architecture, `https://developer.android.com/guide/platform`, *accessed on 21-MAR-2022*

e.g., OpenGL and Vulkan for the acceleration of 3D content, or well-defined HALs for radio communication and biometric authentication hardware.[6]

- **Layer 3:** The native Android system libraries manage access to hardware components such as cameras, speakers, or displays. Besides, these libraries further provide access routines to the Webkit browser, various hardware accelerator chips, and they can be accessed by Android apps using the Android native development kit (NDK).[7]
- **Layer 4:** The Android runtime (ART) consists of the code execution component which is similar to a Java virtual machine, and the corresponding core runtime libraries that offer most of the functionality of the corresponding Java base libraries.
- **Layer 5:** The Android Java application framework offers the components required to build an app, *e.g.*, classes related to UI elements and resource management. Among other purposes, this layer drives the view system, a window manager, a notification manager, and an activity manager.
- **Layer 6:** Incorporates all the Android apps that can be installed by a user, and the apps pre-installed by Google or the device vendor.

The majority of the code that relates to the bottom layer (layer zero) originates from microchip and device vendors, whereas the code at the top layer (layer six) is written by Android app developers. The code of the intermediate layers essentially is required to build Android from the "official" Linux kernel, *i.e.*, the *mainline* kernel. The *Android Open-Source Project (AOSP)* maintains a dedicated *android-mainline* Linux kernel fork that continuously integrates the changes from the Linux mainline kernel.[8] Unfortunately, not all components required to build the Android OS have been open-sourced, *e.g.*, some firmware, device drivers, and proprietary frameworks like the *Google Play Services* still remain closed-source.

2.1.2 Crucial Components

Android apps do not require a traditional "main" method, instead they heavily rely on callback methods across system classes, which are called when the callback events are raised by the OS. We review two major

[6] Google documentation: HAL types, `https://source.android.com/devices/architecture/hal-types`, *accessed on 21-MAR-2022*

[7] Google documentation: native APIs, `https://developer.android.com/ndk/guides/stable_apis`, *accessed on 21-MAR-2022*

[8] Android Open-Source Project (AOSP) website, `https://source.android.com/`, *accessed on 28-FEB-2022*

system classes that are important for Android app development and the remainder of this thesis.

Activity

An `Activity` class instance typically describes the entire life-cycle of a single screen user interface of an app that a user can interact with. This class facilitates 71 different `on*(...)` callback methods, however only seven of them are important for developers: `onCreate()` which is called during the initialization process of the object, `onStart()` which is called just before the system shows the corresponding view to the user, `onResume()` which is called after the activity is shown to the user and ready for user input, `onPause()` which is called when the view loses focus, *e.g.*, due to a home button event, `onStop()` which can be used for clean up and executes if an activity is about to be closed, *i.e.*, the `Activity.finish()` method is called by the user or forcefully by the system, `onRestart()` which is called when the user navigates back to a previously closed activity, and finally, `onDestroy()` which is called before the activity enters the terminal state.

Service

A `Service` class instance typically describes the entire life-cycle of a background service with no user interface that is used by one or more apps. There exist two variants of starting a service: either an app can bind it and without any particular instructions it will terminate if no app is bound anymore, or it can be explicitly started using the method `startService()` that will allow a service to be run indefinitely even when the corresponding app is currently not executed. These two variants can also be combined. Therefore, the abstract service class facilitates eleven different `on*(...)` callback methods, however only four of them are essential for developers: `onCreate()` which is called during the initialization of the `Service` object, `onStartCommand()` which is executed after the service has been started, `onBind()` which is executed whenever an app initially connects to the service, `onUnbind()` which is executed when an app disconnects from the service, `onRebind()` which is executed when an app tries to reconnect after it disconnected, and finally, `onDestroy()` which is executed before the service terminates on behalf of the developer or the system that tries to free resources.

2.1.3 Data Facilities

Android supports a plethora of different data sharing[9] and storage facilities[10] of which this subsection provides an overview. A brief knowledge of these facilities is required to follow the remainder of this thesis.

Intent

The OS and its apps, as well as components within the same or across multiple apps, communicate with each other via inter-component communication (ICC) APIs. These APIs take an *intent object* as a parameter. An intent is either *explicit* or *implicit*. In an explicit intent, the source component declares to which target component, *i.e.*, `Class` or `ComponentName` instances the intent is sent, whereas in an implicit intent, the source component only specifies a general action to be performed, *i.e.*, represented by a text string, and the target component that will receive the intent is determined at run time. Intents can optionally carry additional data called *bundles*. Components declare their ability to receive implicit intents using "intent filters," which allow developers to specify the kinds of actions a component supports. If an intent matches any intent filter, it can be delivered to that component. *Broadcast receivers* receive system-wide "intents," *i.e.*, descriptions of operations to be performed, sent to multiple apps. Broadcast receivers act in the background, and often relay messages to activities or services.

Content Provider

A content provider manages access to a repository of persistent data that could be used internally or shared between apps. That is, it mediates access to one or more databases for various applications and components, *e.g.*, apps, widgets, and search suggestions. The used database structures can be arbitrary, however relational tables and SQLite are preferred.

Persistent Storage

Android can store data on internal storage, *e.g.*, integrated eMMC chips, and external storage, *e.g.*, memory cards. Such storage space can be accessed by traditional file system methods, by the *MediaStore* API to store photo and video content in media folders, by the *DataStore* and *Jetpack Preferences* APIs to save key-value pairs, or by the *Room* API to store data in a SQLite database. Moreover, Android can use such storage to automatically save debugging data of application crashes and *Application Not Responding (ANR)* errors.

[9] Google documentation: application fundamentals, `https://developer.android.com/guide/components/fundamentals`, *accessed on 21-MAR-2022*

[10] Google documentation: app data and files, `https://developer.android.com/guide/topics/data`, *accessed on 21-MAR-2022*

Volatile Storage

A developer can use application memory to store data during the execution of an app. Android uses circular in-memory buffers to store various log messages. Ordinary log messages can be generated in apps with the methods `Log.v()` (verbose message), `Log.d()` (debug message), `Log.i()` (informational message), `Log.w()` (warning message), `Log.e()` (error message), and `Log.wtf()` ("what a terrible failure" message).

IP-like Network Communication

Internet Protocol (IP)-like network connections in Android apps can be established using well-known Java classes such as `Socket`, `InetSocketAddress`, `HttpURLConnection`, *etc.* Therefore, data can be shared by using a variety of different classes. Many Java network APIs abstract the data carrier for simple requests and responses, because it is nowadays irrelevant whether a message is sent through a cellular or WiFi network since both technologies have converged in many aspects, *e.g.*, both natively support the IPv4 and IPv6 addressing schemes. Furthermore, data can be sent with Bluetooth that uses a *Media Access Control (MAC)* addressing scheme which is similar to the one used in IP, to enable the exchange of messages between one master and up to seven clients in a wireless *Personal Area Network (PAN)*. What is more, *Bluetooth Low Energy (BLE)* for slow but energy efficient data transmission supports mesh networking and has no device limit. If faster data rates are required, recent Bluetooth implementations can use a WiFi link to boost the transmission performance at the expense of an increased power consumption.

Other Communication

Communication that is not part of an IP network uses different addressing schemes and techniques than IP. For example, a smartphone can be identified with the *International Mobile Equipment Identity (IMEI)* that is globally unique and the user can be identified with the *International Mobile Subscriber Identity (IMSI)* that is stored inside a *Subscriber Identification Module (SIM)*, which corresponds to a globally unique phone number. Communication services that rely on such identifiers are traditional phone calls, the *Short Message Service (SMS)*, the *Multimedia Messaging Service (MMS)*, or the *Rich Communication Services (RCS)* that supersede the SMS and MMS. There exist numerous other communication protocols for point-to-point or point-to-multipoint connections, *e.g.*, *Near-Field Communication (NFC)* to communicate with another device over very short distances of typically less than ten centimeters, *Universal Serial Bus (USB)* to communicate through wires with up to 127 clients within a radius of few meters, *Infrared Data Association (IrDA)* to communicate to devices in sight with invisible infrared light, and most recently, the *Ultra-*

Wide Band (UWB) communication protocol that in theory supports fast data transfers between devices within a range of about 200 meters, which is currently used primarily for the precise tracking of objects. Rather unconventional communication protocols can further leverage speakers and microphones, screens and cameras, or possibly light-based senders and the corresponding camera-like receivers that are used in lidars and *Time of Flight (ToF)* systems, which can create a depth map of the surroundings.

2.1.4 Pillars of Security

Google employs various security mechanisms to ensure a secure Android app platform.[11]

App Screening

Before any app is published in the app store, it is screened by Google's Bouncer that is nowadays part of the Google Play Protect service, which combines cloud-based security with on-device protection that is similar to a traditional desktop malware scanner. In general, this is a continuously improved automated service,[12] however it can include a manual code review if Bouncer identifies potentially harmful features in an app or the corresponding metadata. If the app does not pass these screenings, it will be rejected and not listed in the app store, or if already listed, it will be unlisted from the app store and eventually removed from user devices. According to Google, Bouncer instruments the app code with static and dynamic code analysis tools that leverage machine learning algorithms.[13] That is, its algorithms monitor hundreds of features and compare behavior across Android apps to identify potentially suspicious behavior such as illicit or unexpected interactions with other apps on a device, accessing or sharing of personal data without any authorization, the aggressive installation of apps, requests to known malicious websites, or the bypassing of built-in security features. Additionally, metadata from a developer's Google account is considered like actions, history, billing details, device information, and more. To effectively identify potential threats, Bouncer scans apps that are reported by security researchers, users, or generally apps found on the internet. However, to let Bouncer inspect installed apps from other markets, users have to allow Google to review new apps

[11] Android security & privacy: 2018 year in review, `https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf`, *accessed on 19-MAR-2022*

[12] Ars Technica: new Play store rules block most apps from..., `https://arstechnica.com/gadgets/2021/04/new-play-store-rules-block-most-apps-from-scanning-your-entire-app-list`, *accessed on 11-APR-2022*

[13] Google documentation: cloud-based protections, `https://developers.google.com/android/play-protect/cloud-based-protections`, *accessed on 11-APR-2022*

by enabling the "Improve harmful app detection feature" in Google Play Protect on their devices.

Although Google tries to hide implementation details of their screening process to prevent adversaries from exploiting it, researchers could still obtain interesting findings. For example, Bouncer seems to use for the dynamic analysis a randomly modified QEMU Android emulator instance with some arbitrary user data that executes the app for a rather short amount of time, and Bouncer grants internet access to the apps that are being tested [66]. Therefore, Bouncer can assess byte code and native code, which is usually used for specific features that require heavy computation such as the conversion of image or video content. Researchers further found that app analyses are immediately performed after every submission, or weekly if no submission has been performed by the app author [71]. Moreover, Google Play Protect seems to be less strict compared to traditional malware scanners [43].

Since the time available for automated and manual code reviews is limited, Google Play Protect is particularly vulnerable for delayed attacks where the submitted application behaves benignly while it is being analyzed, but turns malicious otherwise.

App Side-loading

In recent Google Play Protect-certified devices, by default, the manual installation of apps from unknown sources is disabled, which is also called app "side-loading" of *Android PacKage (APK)* files. As result, Google has full control over all the available apps and can, if required, prevent the spread of malicious apps.

Permission System

Access to sensitive APIs is protected by a set of permissions that the user can grant to an app. If the user declines a permission request from an app, the app can either continue with limited functionality or terminate. In general, these permissions are text strings that correlate to a specific access grant, *e.g.*, `android.permission.CAMERA` for camera access. Apps can further declare custom permissions with appropriate protection levels to protect their exposed interfaces against unauthorized accesses from other apps. In more detail, apps can use the protection levels "normal," *i.e.*, the default permission level that is automatically granted at installation time, "dangerous," which must be requested during run time and usually leads to a UI notification that a user has to acknowledge, or "signature," which only grants access if the requesting app has been signed using the same certificate as the granter. If two apps are signed using the same certificate, they can be expected to be from the same vendor.

Security Hardware

Recent Android devices support hardware security features such as the *Trusted Execution Engine (TEE)*, which is part of the host CPU and runs a secure, isolated OS that can only exchange data with Android via the corresponding secure kernel driver. Therefore, that OS is preferably used as a safe enclave to protect sensitive information such as passwords and decryption keys. In addition, Android supports various authentication hardware such as fingerprint readers and facial recognition systems.

Security Software

The Android platform assigns a unique user identifier (UID) to each app at installation time that prevents apps from accessing other apps' data. Moreover, each app runs in a unique process within a sandbox so that every app runs in isolation from other apps. The used *Security-Enhanced Linux (SELinux)* kernel security module further improves the resiliency against various attacks by introducing system-wide security policies and a more fine grained set of access controls. In addition to that, user-space hardening protects against memory corruption threats by introducing *Address-Space Layout Randomization (ASLR)*, *Control Flow Integrity (CFI)*, and *Data Execution Prevention (DEP)*. Finally, the entire boot process relies on cryptographic signatures to prevent unauthorized changes to the OS.

Software Updates

There exist two kinds of Android software updates: app updates and OS updates. The Play Store is used to distribute app updates just in time across its users. However, the procedure for OS updates is entirely different and updates are released much less frequently, *i.e.*, usually every few weeks or months, because device vendors have to adopt the changes and must possibly re-certify a cellular modem in different countries if they have modified its baseband firmware. Therefore, Google is currently transitioning as many OS components as possible into the Play Store to facilitate more seamless future updates.

System Image Scanning

Android can be installed on a device with a system image file that must be flashed to the device's persistent memory. A vendor can perform arbitrary changes in the default image and, for instance, pre-install any Android application or include additional device drivers to support custom hardware. As a result, before any release of a new Android-powered device, it is recommended that the device vendor submit the customized system image to

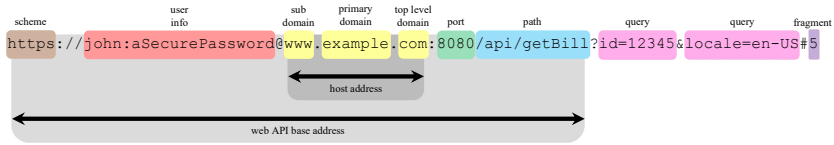


Figure 2.2: The structure of a URL

Google to let them perform a security audit to identify potential threats. If a problem is revealed, Google will assist the vendor in its remediation.[14]

2.2 Web Communication

The web communication of mobile apps accounted for more than 54% of the total internet traffic in 2021.[15] In this section, we briefly discuss the primarily used web addressing scheme, the communication protocol, and the web APIs that are built on top of them.

2.2.1 Web Addressing Scheme

A web resource is usually accessed with a *Unified Resource Locator (URL)*, which points to its location. The typical structure of a URL is shown in Figure 2.2. From the left to the right, the *scheme* denotes the desired protocol that usually corresponds to a particular *port*, the optional *user info* section holds the colon-separated user name and password required to access a protected resource, the *sub domain*s, the *primary domain*, and the *top level domain* identify the target computer(s) in the internet that store(s) the desired resource, the *port* number specifies the desired target service or process, the *path* indicates the desired resource in the host service's logical structure, the optional *query* elements further parameterize the web resource so that a server can respond accurately, and finally, the *fragment* specifies the desired page if the requested resource supports pagination. When we refer to a *host address*, we assume the fully-qualified domain name that includes any sub, primary, and top level domains, and when we refer to a *web API base address*, we assume the left part of the URL including the path. There further exists, compared to the URL, the more general concept of a *Uniform Resource Identifier (URI)*, which can describe entities beyond web resources. A URI can identify a particular physical or virtual resource that is unique in its kind, *e.g.*, an international

[14] Android security & privacy report for 2018, `https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf`, *accessed on 28-FEB-2022*

[15] Statista: percentage of mobile device website traffic worldwide, `https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices`, *accessed on 21-MAR-2022*

phone number or a book identifier such as the *International Standard Book Number (ISBN)*.

2.2.2 Hypertext Transport Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) is a client-server request-response protocol that was invented in the 1980s by Tim Berners-Lee[16] and has initially been planned as a general purpose "application-level protocol for distributed, collaborative, hypermedia information systems" [32], but it is best-known for its delivery of content to web browsers. In fact, web browsers commonly access websites through URLs that use the HTTP or HTTPS scheme. HTTP and its successor HTTP/2 provide facilities to encapsulate user data, *e.g.*, HTML, JSON, XML, or SOAP, and use plaintext messages to instruct the receiver on how to treat the transmitted data. HTTP Secure (HTTPS) is an extension of HTTP and thus follows the same principles, except that the messages are encrypted. As shown in Listing 1, requests and responses mostly follow the same structure, but there exist minor differences: the request always specifies the HTTP method (line one), *e.g.*, `GET`, `POST`, `PUT`, `DELETE`, and the requested fully qualified resource path, *i.e.*, lines one and two. On the contrary, the server response includes an HTTP status code (line eleven) to indicate whether the request was successful, but not any resource path.

```
1  GET /v2/networks/nextbike-leipzig HTTP/1.1
2  Host: api.citybik.es
3  User-Agent: Mozilla/5.0 (Windows NT 10.0)
4  Accept: text/html,application/xhtml+xml
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Connection: keep-alive
8  Upgrade-Insecure-Requests: 1
9  Cache-Control: max-age=0
10
11 HTTP/1.1 200 OK
12 Content-Type: application/json
13 Content-Length: 5613
14 Server: nginx/1.15.9 (Ubuntu)
15 Date: Mon, 14 Oct 2019 09:44:16 GMT
16 Access-Control-Allow-Origin: *
17 X-RateLimit-Limit-minute: 180
18 X-RateLimit-Remaining-minute: 179
19 X-Kong-Upstream-Latency: 61
20 X-Kong-Proxy-Latency: 1
21 Via: kong/1.2.1
```

Listing 1: Typical request and response communication flow between a client and a server

[16] Request For Comments (RFC) of the HTTP 1.0, `https://datatracker.ietf.org/doc/html/rfc1945`, *accessed on 28-FEB-2022*

HTTP Header Fields

In HTTP communication, header fields are used to set up the connection between the server and the client, *e.g.*, by specifying the used data encoding (line six) and content caching option (line nine), and to provide additional information, *e.g.*, the used infrastructure (lines three and fourteen) or the originating date of the response (line fifteen). Besides non-standard fields, there exist for the sake of interoperability 48 different header fields in the HTTP/1.1 specification; each header field consists of a key-value pair in textual form. For example, the header field key `Content-Type` (line twelve) declares the content type of the message body, *e.g.*, the value `text/plain` is used for plain text, `text/html` for websites, or `application/json` for JSON web API responses [32].

Security-related Header Fields

Header fields can pose a threat, *e.g.*, leak software version information, or they can mitigate a threat, *e.g.*, code execution, click-jacking, and data leak. Version information leaks are typically caused by `Server`, `X-Powered-By`, `X-AspNet-Version`, and `X-Powered-By-Plesk` header fields. These headers are widely used in web communications, and the risk is that if adversaries know the software version, they can research that software and find publicly disclosed vulnerabilities in old software versions to plan attacks on web servers accordingly. Conversely, header fields such as `X-Content-Type-Options`, `X-XSS-Protection`, or `Content-Security-Policy` can mitigate code execution attacks. Code execution attacks require two steps: the arbitrary injection of malicious code into an app, and its execution to steal sensitive data or to manipulate a rendered website. Moreover, `X-Frame-Options` mitigates potential click-jacking attacks that can be performed, *e.g.*, when several iframes are shown simultaneously, but one creates a view that overlays the others. This attack confuses users to accidentally click on clickable elements such as buttons and hyperlinks. Another important security-related header field is HTTP `Strict-Transport-Security` (HSTS), which ensures that clients access a certain URL only through secure HTTPS communication channels, thus mitigating potential man-in-the-middle attacks. However, to benefit from such protection the first page visit must not be tampered with unless the desired domain is already included in the pre-loaded list of domains that support HSTS, which instructs the HTTP client to use HTTPS without any prior communication over the insecure HTTP.

2.2.3 Web APIs

A web API can refer to a client side implementation, *e.g.*, in a web browser, or a server side implementation, *e.g.*, a web service that generates dynamic

responses in a machine-readable format. In this thesis, we exclusively focus on server side web APIs.

Server side web APIs commonly follow the *REpresentational State Transfer (REST)* architectural pattern, which is based on HTTP. In theory, a "RESTful service" is a well-documented stateless dynamic web server that can respond to requests with properly encoded JSON or XML data. However, some implementations are not well documented, return malformed responses, or do not scale well, because they are stateful, *i.e.*, they maintain connection information throughout the entire session, *e.g.*, a successful log-in. To access such a service, it is mandatory to know the web API base address, which is a URL, *e.g.*, `https://api.weather.com` to access an imagined weather service web API. The current weather at a particular location could then be requested by attaching an additional query parameter to the base URL, *e.g.*, `place=Paris`. The resulting URL string `https://api.weather.com?place=Paris` would then lead to a response like {`"condition":"clear sky","tempCelsius":25`} that is returned using HTTP. Usually, such services require either a unique query key-value pair that grants the requester specific usage rights, or require credentials attached to the HTTP request header. The information on how to use a given service can be found in the API documentation that should be made publicly available by the API operators.

2.3 Security Smell

In this thesis we do not particularly distinguish between a *security code smell* and a *security smell*, although their meaning is slightly different: We consider the *security code smell* a symptom in the code that signals the prospect of a security vulnerability, and more generally, we consider a *security smell* a symptom in the resources of an application that signals the prospect of a security vulnerability.

Chapter 3

State of the Art

In this chapter we discuss related work to our security analyses, which is relevant for this thesis. In particular, we present related work for our investigations on Android security, Android inter-component communication (ICC) security, and Android web security.

Declaration of Content Reuse

The content of this chapter contains elements from sections that correspond to paper submissions, *i.e.*, *Security Code Smells in Android* (chapter 4), *Security Code Smells in Android ICC* (chapter 5), *Security Smells in Web APIs* (chapter 6), *Security Smells in Mobile App URLs* (chapter 7), and *Security Smells in HTTP Headers* (chapter 8).

3.1 Android Security

Code analysis is important for assessing the security of Android apps, and one of these four strategies is usually pursued: i) a manual code analysis where researchers use a workflow that involves manual work requiring specialized knowledge, ii) a static code analysis where researchers use a tool to extract features from source code, iii) a dynamic code analysis where researchers use a tool to extract features during the execution of an app, or finally, iv) a hybrid code analysis where researchers employ both static and dynamic code analysis techniques.

Linares-Vasquez *et al.* mine 660 Android vulnerabilities available in the official Android bulletins and their CVE details,[1] and present a taxonomy of the types of vulnerabilities [57]. They report on the presence of those

[1] Common Vulnerabilities and Exposures (CVE), a public list of known cybersecurity vulnerabilities, `https://cve.mitre.org`, *accessed on 28-FEB-2022*

vulnerabilities affecting the Android OS, and acknowledge that most of them can be avoided by relying on secure coding practices. Li *et al.* studied the state of the art work that statically analyses Android apps [55]. They found that much of this work supports detection of private data leaks and vulnerabilities, a moderate amount of research is dedicated to permission checking, and only three studies deal with cryptography issues. Unfortunately, much state of the art work does not publicly share the concerned artifacts. Reaves *et al.* studied Android-specific challenges to program analysis, and assessed existing Android application analysis tools. They found that these tools mainly suffer from lack of maintenance, and are often unable to produce functional output for applications with known vulnerabilities [81]. Finally, Sadeghi *et al.* review 300 research papers related to Android security, and provide a taxonomy to classify and characterize the state of the art research in this area [84]. They find that 26% of existing research is dedicated to vulnerability detection, but each study is usually concerned with specific types of security vulnerabilities. Our work expands on such studies to provide practitioners with an overview of the security issues that are inherent in insecure programming choices.

Some research is devoted to educating developers in secure programming. Xie *et al.* interviewed fifteen professional developers about their software security knowledge, and realized that many of them have reasonable knowledge but do not apply it as they believe it is not their responsibility [110]. Weir *et al.* conducted open-ended interviews with a dozen app security experts, and determined that app developers should learn analysis, communication, dialectics, feedback, and upgrading in the context of security [102]. Witschey *et al.* surveyed developers about their reasons for adopting or not adopting security tools [106]. Interestingly, they found the perceived prestige of security tool users and the frequency of interaction with security experts to be important for promoting security tool adoption. Acar *et al.* suggest a high-level research agenda to achieve usable security for developers. They propose several research questions to elicit developers' attitudes, needs and priorities in the area of security [65].

Our work is complementary to these studies in the sense that we provide an initial assessment of developers' security knowledge, and we highlight the significant role of developers in making apps more secure.

3.2 Android ICC Security

Numerous researchers have dedicated their work to detecting common ICC vulnerabilities. Despite the fact that ICC has changed over time, for example, with the availability of new APIs such as the path-permission feature, the vulnerability classes have remained largely the same. Chin *et al.* discuss the ICC implementation of Android and examine closely the interaction between sent and received ICC messages [16]. Despite the fact that their work is based on a small corpus containing only twenty apps, they

were able to detect various denial-of-service issues in numerous application components, and conclude that the message-passing system in Android enables rich applications, and encourages component reuse, while leaving a large potential for misuse when developers do not take any precautions.

Felt *et al.* discovered that permission re-delegation, also known as confused deputy or privilege escalation attack, is a common threat, and they pose OS level mitigations conceptually similar to the same origin policy in web browsers [30]. The community aimed on the one hand for preciseness, as countless tools to detect these flaws in ICC have been released, notably Epicc [67] and IccTA [54] with a significantly improved precision. On the other hand, the app coverage began to play a major role, as in the work of Bosu *et al.* who recently discovered with their tool inadequate security measures, including privilege escalation vulnerabilities, among inter-app data-flows from 110 000 real-world apps [12].

Along with static analysis that does not require any execution of code, new kinds of attacks and run time countermeasures have emerged in the scientific community. Garcia *et al.* crafted a state of the art tool to automatically detect and exploit vulnerable ICC interfaces to provoke denial-of-service attacks amongst others [38]. They identified exploits for more than 21% of all apps appraised as vulnerable. Xie *et al.* presented a bytecode patching framework that incorporates additional self-contained permission checks avoiding privilege issues during runtime, generating a remarkably low computational overhead [111]. Ren *et al.* successfully investigated design glitches in the multitasking implementation of Android, uncovering task hijacking attacks that affected every OS release and were potentially duping user perception [82]. They considered in particular the `taskAffinity` and `taskParentReparenting` attributes of the manifest file that allow views to be dynamically overlaid on other apps, and provided proof-of-concept attacks. Wang *et al.* assessed the threat of data leakage on Apple iOS and Android mobile platforms and show serious attacks facilitated by the lack of origin-based protection on ICC channels [98]. Interestingly, they found effective attacks against apps from such major publishers as Facebook and Dropbox, and more importantly, indicate the existence of cross-platform ICC threats.

Researchers have found interest in reinforcing the Android ICC core framework. Khadiranaikar *et al.* propose a certificate-based intent system relying on key stores that guarantee integrity during message exchanges [49]. In addition to securing the ICC-based communication, Shekhar *et al.* proposed a separation of concerns to reduce the susceptibility for manipulation of Android apps, by explicitly restricting advertising frameworks [87]. Ahmad *et al.* elaborated on problematic ICC design decisions on Android, and found that missing consistent message types and conformance checking, unpredictable message interactions, and a lack of coherent versioning could break inter-app communication and pose a severe risk [3]. They recommend a centralized message-type repository that immediately

provides feedback to developers through the IDE.

In summary, existing studies have often dealt with a specific issue, whereas the work presented in this thesis is aimed at covering a broader range of issues, making the results more actionable for practitioners. For example, we investigate multiple different ICC APIs in chapter 5. Moreover, previous work often overwhelms developers with reports that contain many identified issues at once, whereas the work presented in this thesis aims to provide feedback during app development where developers have the relevant context. One result is the presented code linting tool in chapter 5, which offers immediate feedback while a developer writes code. Such feedback makes it easier to react to issues, and helps developers to learn from their mistakes [97].

3.3 Android Web Security

Related work to this field primarily pertains to app analyses that have been summarized by data transmissions with a particular interest in web communication and the used URLs, public service audits that improve the app server security, and the HTTP header support of popular websites.

3.3.1 APIs

Zhou *et al.* harvested free email and Amazon AWS cloud service credentials with their tool *CredMiner* from more than 36 500 apps from various Android markets [119]. In their case studies, they mention unprotected credentials within the app's source code, obfuscated credentials using a *Base64* encoding, and encrypted credentials, however, in those cases the decryption key has also been found in the app's source code. They alarmingly found that more than every second app using such a service leaked the developers' credentials in the apps' source code. Making matters worse, more than 77% of those collected credentials were valid at the time of the experiment. Such credentials will present a massive threat in the mid-term future, as many of those credentials cannot be easily replaced without temporarily abating the experience of millions of users, but in the meantime they can be easily exploited by attackers.

Rapoport *et al.* studied web requests in Android apps [78]. They demonstrated that a large number of web requests are not immediately traceable to source code and need dynamic analysis. For instance, URLs may originate in app resources, *e.g.*, XML files or Gradle build scripts, they may stem from the content received from previous web requests, or they might be assembled by JavaScript code at run time. In contrast, a significant proportion of URLs are only detected by static analysis: the dynamic analysis may simply fail to produce desired results due to a lack of code coverage during instrumentation.

Mendoza *et al.* studied the inconsistencies in input validation logic between apps and their respective web API services [61]. They developed a tool to extract requests to web API services from an app, and to infer sample input values that violate the implemented constraints found in the app, such as email address or JSON content validation executed on the client side. They then analyzed app-violating request logic on the server side via black box testing. From a set of 10 000 popular Android apps, they found 4 000 apps that do not properly implement input validation for web API services. Investigation of web API hijacking vulnerabilities in 1 000 apps showed that the security and privacy of millions of users are at risk.

In summary, unlike our investigation in chapter 6, existing work usually focused on the use of `java.net` APIs, and did not study several third-party libraries to implement network communication in Android apps. Finally, to the best of our knowledge, dissecting the distribution of elements that comprise the web APIs, and the use of embedded languages, is never studied.

3.3.2 URLs in Apps

Web communication in apps is usually initiated by the client, *i.e.*, the app that sends a request to a specific server. Therefore, apps can reveal interesting features used to establish such a connection. For example, Zuo *et al.* analyzed 5 000 top-ranked apps in Google Play and identified 297 780 URLs that they fed to the VirusTotal URL screening service [120]. The service identified 8 634 harmful URLs of which the majority were related to malware (43%), followed by malicious sites (37%), and phishing (23%). For the malware category, one interesting example they mention is an APK file download triggered by an app, which itself tries to obtain superuser access to the device by exploiting Linux kernel vulnerabilities. Mendoza *et al.* investigated the input validation constraints imposed by apps on outgoing requests to web API services from 10 000 popular free apps from the Google Play Store of which 46% suffered from inconsistencies that could be exploited by attackers [61]. Such inconsistencies allowed them to access app-related databases through various injection attacks, *e.g.*, they could misuse an app's email address field for an SQL injection attack, because its value did not receive additional server side validation.

Contrary to our analyses in chapter 6 and chapter 7, a comprehensive investigation of the URLs used in mobile apps has not yet been performed, *i.e.*, existing work primarily focuses on a particular security aspect and proposes a novel strategy to mitigate the potential threat, but it lacks a more general view on web communication security, which limits the scope of the proposed remediation strategies.

3.3.3 App Servers

App server security focuses on server side problems, configuration, or implementation. Zuo *et al.* found that 15 098 app servers are subject to data leakage attacks [121]. In particular, they suffer either from a broken key management, *i.e.*, the developers became confused about root and app keys, or from a broken permission configuration, *i.e.*, developers were overwhelmed when they had to choose appropriate permissions for their data. They assume that this is a direct consequence of the utterly complex interfaces to configure such services designed for developers. That is, Google even provides a language for developers to specify the desired user permissions. Moreover, Mendoza *et al.* found discrepancies between the use of such features in the mobile and desktop version of websites that enable various injection and spoofing attacks, although the affected websites remain in the realm of a few percent [60].

In chapter 7 we will advance existing research by performing an empirical study on the prevalence of security smells in app server configurations and we will assess the maintenance activity of app servers.

3.3.4 HTTP Headers

HTTP headers offer various security features and are widely used in traditional web communication. Lavrenovs *et al.* assessed HTTP security header fields in the top one million websites including those accessed by mobile apps and found that less popular websites tend not to implement security-related features for their users [52]. Notably, they found that nearly 38% of the top one thousand sites reported by Amazon Alexa implement HSTS, while it is the case for only 17.5% of the top one million HTTPS websites. This finding shows that large websites are twice as likely to implement an up-to-date, effective security feature as less popular websites. Buchanan *et al.* performed the same experiment and generally draw very similar conclusions, except that, according to them, *Let's Encrypt SSL* certificates are more than eight times more prevalent in minor sites compared to major sites, because the top sites apparently have the money to buy their own certificates and do not rely on free services [14].

Fahl *et al.* investigated the use of insecure HTTP configurations in Android apps [24] and found that almost every tenth app is potentially vulnerable to a man-in-the-middle attack. They could capture credentials from various credit cards, social media accounts, web blogs, *etc.*

Adopting new header fields seems to be an appropriate approach to address HTTP issues without breaking existing implementations. Two well-known instances are the cross-site scripting protection proposed by Stamm *et al.* [90] and the protection against downgrading HTTPS connections, which arose from the work of Marlinspike [59]. Both new header fields, *i.e.*, `Content-Security-Policy` and `Strict-Transport-Security`,

have become standardized, and they are now supported in major web browsers.

HTTP header fields used in web communication are well documented and received much interest in the scientific community, however their support in mobile app HTTP clients has not yet been comprehensively studied. We perform an in-depth study on HTTP header support in popular HTTP clients used by Android apps in chapter 8.

3.4 Conclusion

We can confirm many results of existing work, however the reported vulnerabilities are usually incomplete in one or more aspects: they lack a thorough discussion about symptoms, mitigation strategies, consequences, prevalence, or the required tools. Consequently, security engineers face difficulties to obtain all the required information to fully understand and mitigate potential threats.

Moreover, existing work focuses at a particular domain and does not consider a holistic view on the ecosystem. Therefore, such work cannot leverage synergies of security measures across different domains to propose effective holistic security strategies.

These two major limitations lead to the various open challenges we address in the five subsequent chapters in which we will investigate security smells in different domains, *i.e.*, Android, Android ICC, web communication of mobile apps, app servers, and finally, HTTP clients used in mobile apps. The remaining four chapters will then discuss the threats which arise from security smells and effective holistic countermeasures, before we reach the conclusion.

The open challenges are:

- **Chapter 4: Security Smells in Android**
  [Challenge 1] The establishment of a notion to describe potential vulnerabilities. [Challenge 2] The compilation of a comprehensive list of Android security smells that have been reported in the literature. [Challenge 3] The large-scale study regarding the prevalence of security smells in mobile apps.

- **Chapter 5: Security Smells in Android ICC**
  [Challenge 4] The compilation of a comprehensive list of ICC security smells that have been reported in the literature. [Challenge 5] The implementation of IDE tool support for the reported ICC security smells. [Challenge 6] The large-scale study of the prevalence of ICC security smells in mobile apps.

- **Chapter 6: Security Smells in the Web Communication of Mobile Apps**
  [Challenge 7] The implementation of a tool to extract data relevant

for the web communication of mobile apps. [Challenge 8] The large-scale study of web communication characteristics of mobile apps. [Challenge 9] The compilation of a comprehensive list of web communication security smells that have been reported in the literature.

- **Chapter 7: Security Smells in Mobile App Servers**
  [Challenge 10] The large-scale study of the prevalence of the server-side security smells in the web communication of mobile apps. [Challenge 11] The investigation of the relationship between security smells and app server maintenance.

- **Chapter 8: Security Smells in Mobile App HTTP Clients**
  [Challenge 12] The investigation of security-related HTTP header support in existing HTTP clients.

- **Chapter 9: Effective Holistic Security for Mobile Apps**
  [Challenge 13] The review of reported security smells with respect to how they enable attack mechanisms. [Challenge 14] The identification of holistic security strategies that can effectively prevent attack mechanisms.

- **Chapter 10: Default Values and Practices to Improve Application Security**
  [Challenge 15] The discussion of default values and practices that could greatly improve application security.

- **Chapter 11: A String-based Framework to Improve Application Security**
  [Challenge 16] The implementation of a String-based framework to improve application security. [Challenge 17] The investigation of the restrictions when using such a framework with existing code. [Challenge 18] The investigation whether such a framework can offer protection against data leaks and remote code execution, and what security risks could arise when using it.

Chapter 4

Security Code Smells in Android

Declaration of Content Reuse

The content of this chapter is based on the full paper *Security Code Smells in Android* that has been accepted for the *17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)* in 2017 [39].

Smartphones and tablets have recently overtaken the number of computers. They provide powerful features once offered only by computers, but the risk of vulnerability on these devices is not on a par with traditional desktop programs; smartphones are increasingly used for security sensitive services like e-commerce, e-banking, and personal healthcare, which make these multi-purpose devices an irresistible target of attack for criminals.

The recent survey on the *Stack Overflow* website shows that about 65% of mobile developers work with Android.[1] This platform has captured over 80% of the smartphone market,[2] and just its official app store contains more than 2.8 million apps. As a result, a security mistake in an in-house app may jeopardize the security and privacy of millions of users. In this chapter, we use the term "security" to refer to both security *and* privacy.

The security of smartphones has been studied from various perspectives such as the device manufacturer [108], its platform [113], and end users [47]. Manifold security APIs, protocols, guidelines, and tools have been proposed. Nevertheless, security concerns, in effect, are outweighed by other concerns [9]. Many developers undermine their significant role in providing security [110]. As a result, apps still suffer from serious pro-

[1] Stack Overflow: Developer Survey Results in 2017, `https://insights.stackoverflow.com/survey/2017`, *accessed on 28-FEB-2022*

[2] Website of Gartner, a technological research and consulting agency, `https://www.gartner.com`, *accessed on 28-FEB-2022*

liferating security issues.[3] For instance, the analysis of 100 popular apps downloaded at least 10M times, revealed that over 90% of them, due to development mistakes, are prone to SSL vulnerabilities that allow criminals to access credit card numbers, chat messages, contact list, files, and credentials [69].

Given these premises, the primary goal of this chapter is to shed light on the root causes of programming choices that compromise users' security. In contrast to previous research that has often dealt with a specific issue, we study this phenomenon from a broad perspective. We have identified avoidable vulnerabilities and their corresponding smells in the code, and we discuss how they could be eliminated or mitigated during development. We have also developed a lightweight static analysis tool to look for several of the identified security smells in 46 000 apps. In particular, we answer the following three research questions:

- **RQ**$_1$: *What are the security code smells in Android apps?* We have reviewed major related work, especially those appearing in top-tier conferences or journals, and identified 28 avoidable vulnerabilities and the smells that indicate their presence. We thoroughly discuss each smell, the risk associated with it, and its mitigation during app development.

- **RQ**$_2$: *How prevalent are security smells in benign apps?* We have developed a lightweight tool that statically analyzes apps for the existence of ten security smells. We applied the tool to a repository of about 46 000 apps hosted by Google. We realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security smells.

- **RQ**$_3$: *To which extent does identifying security smells facilitate detecting vulnerabilities?* We manually inspected 160 apps, and compared our findings to the result of the tool. Our investigation showed that the identified smells are in fact a good indicator of security vulnerabilities.

To summarise, this chapter represents an initial effort to spread awareness about the impact of programming choices in making secure apps. We argue that this helps developers who develop security mechanisms to identify frequent problems, and also provides developers inexperienced in security with caveats about the prospect of security issues in their code.

The remainder of this chapter is structured as follows. We present the identified vulnerabilities and corresponding security smells in section 4.1. We study the prevalence of these smells and discuss the results in section 4.2. We conclude this chapter in section 4.3.

[3]CVE Details: the ultimate security vulnerability data source, `https://www.cvedetails.com`, *accessed on 28-FEB-2022*

4.1 Security Smells

Although Android security is a fairly new field, it is very active, so researchers in this area have published a large number of articles in the past few years. We were essentially interested in any paper explaining an issue, or a countermeasure that involves the security of apps in Android. We used a keyword search over the title and abstract of papers in IEEE Xplore and ACM Digital Library, as well as those indexed by the Google Scholar search engine. We formulated a search query comprising *Android* and any other security-related keywords such as *security*, *privacy*, *vulnerability*, *attack*, *exploit*, *breach*, *leak*, *threat*, *risk*, *compromise*, *malicious*, *adversary*, *defence*, or *protect*. We read the title and, if necessary, skimmed the abstract of each paper and included security-related ones. We further read the introduction of these papers and excluded those whose concerns were not about app security. In order to extend the search, for each included paper we also recursively looked at both citations and cited papers. Finally, we carefully reviewed all remaining papers. During the whole process, we resolved any disagreement by discussion.

We identified 28 smells that may lead to vulnerabilities in Android-powered devices, *i.e.*, we consider a vulnerability a security issue that compromises user's security and privacy. We group these smells into five categories. We explain each smell, its consequence, *i.e.*, potential risk, and its symptom, *i.e.*, an identifiable property in the code. We also mention any possible resolution, *i.e.*, a more secure practice to eliminate or mitigate the issue during app development.

4.1.1 Insufficient Attack Protection

- **Unreliable Information Source**
  Developers acquire their programming knowledge from various sources such as official documentation, books, crowd sources, *etc. Issue:* According to recent research, developers increasingly resort to studying code examples provided by informal sources like *Stack Overflow*, which are easy to access and integrate, but often lack security concerns [1]. *Consequently*, vulnerabilities could make their way into apps in the absence of security expertise. *Symptom:* Existence of copy-pasted code from untrustworthy sources. *Mitigation:* Use official sources, which are more reliable, and vet the security of any external code before and after integration in your code.

- **Untrustworthy Library**
  Developers cope with the complexity of modern software systems and speed up the development process by relying on the functionalities provided by off the shelf libraries. *Issue:* Many third-party libraries are unsafe by design, *i.e.*, they introduce vulnerabilities and compromise user data [101]. *Consequently*, the ramification of adopting

such libraries could be manifold. *Symptom:* The app utilises unsafe libraries such as advertising libraries that are known to be prone to data leakage [21]. *Mitigation:* Solely use reliable libraries [8].

- **Outdated Library**
  The risk of using third-party libraries is not resolved by only using trusted libraries per se. *Issue:* Libraries usually offer various bug fixes and improvements in newer releases, but often different developers maintain libraries and apps, and their update cycles generally do not coincide [100]. *Consequently*, a security breach in an old library or a deprecated API could lead to serious issues. *Symptom:* An included library is behind the latest release, or the app exercises a deprecated API that is not maintained anymore (*e.g.*, the `SHA1` cryptographic hash function). *Mitigation:* Integrate the latest release of a library into your app and replace deprecated APIs with their newer counterparts. Publish an update not only when the app itself has some improvements but also when there is a new version of a library, which the app uses.

- **Native Code**
  Developers often incorporate native code in their apps to perform intensive computations or to use many third-party libraries, which exist in this form. *Issue:* Native code is hard to analyze; there is only little distinction between code and data at the native level, and attackers can load and execute code from native executables, in a variety of ways much easier than in Java. *Consequently*, native code is susceptible to severe vulnerabilities like buffer overflow, and an attacker could exploit such vulnerabilities, for instance, to execute malicious code [99]. *Symptom:* Existence of native code or a native code library in the app. *Mitigation:* Use native code only when necessary, and only integrate trustworthy libraries [8] into your code.

- **Open to Piggybacking**
  Android apps are often easy to repackage. *Issue:* Adversaries could add their malicious code to a benign app before repackaging it [53]. *Consequently*, depending on the original app's popularity, users can be infected when installing a seemingly benign app that has evaded the analyses of leading app markets [15]. *Symptom:* No technique, *e.g.*, watermarking or signature checking is applied to hardening repackaging. *Mitigation:* Leverage obfuscation to make retro-engineering of apps harder. Also, verify the app's authenticity before any sensitive operation.

- **Unnecessary Permission**
  The use of protected features on Android devices requires explicit permissions, and developers occasionally ask for more permissions

than necessary [94]. *Issue:* The more permission-protected features an app can access, the more sensitive data it can reach. *Consequently*, a more permission-hungry app may expose users to additional security risks [95]. *Symptom:* The manifest file contains permissions for APIs that are not used. *Mitigation:* Utilize tools like PScout[4] to exclude from the manifest file any permission whose corresponding API calls are absent in the app.

4.1.2 Security Invalidation

- **Weak Crypto Algorithm**
  The fundamental set of cryptographic algorithms can be categorized into symmetric, asymmetric, and hash functions. *Issue:* Each category includes several algorithms, each of which may have various features and attack resilience. *Consequently*, incautious adoption of an algorithm could subject to security issues. *Symptom:* The use of weak cryptographic hash functions like `SHA1` or `MD5`, insecure modes, *e.g.*, `ECB` for block ciphers. *Mitigation:* Consult the state of the art guidelines to choose an appropriate cryptography, and utilize expert systems [7].

- **Weak Crypto Configuration**
  The majority of security breaches come from exploiting developers' mistakes. *Issue:* Cryptography APIs are widely perceived as being complex with many confusing options [63]. *Consequently*, a strong but poorly configured algorithm could jeopardise the in-place security. *Symptom:* Each algorithm has different parameters, and cryptographic parameters in each library could have different defaults. PBE (password-based encryption) with fewer than 1 000 iterations, short keys and salts, or inappropriate random seeds and initialisation vectors are common mistakes. *Mitigation:* Use libraries that provide strong documentation and working code examples, and rely on simplified APIs with secure defaults [2].

- **Unpinned Certificate**
  Digital certificates are needed to ensure secure communication. Unpinned certificates are easy to maintain and are frequently used in the appified world [68]. *Issue:* Ensuring the authenticity of a certificate is non-trivial, if it is not pinned. *Consequently*, an app may inadvertently end up trusting a certificate issued by an adversary who has intercepted network communication. *Symptom:* The app uses unpinned certificates. *Mitigation:* Pinning certificates are always recommended to increase the security. Since Android 6.0 pin-

[4]PScout: analyzing the Android permission specification, `https://pscout.csl.toronto.edu`, *accessed on 28-FEB-2022*

ning can be enabled using the `Network Security Configuration` feature.

- **Improper Certificate Validation**
  Android provides a built-in process for validating the certificates signed by the trusted Certificate Authorities (CA). *Issue:* In other cases, *e.g.*, when a certificate is self-signed, the OS devolves this validation process to the app itself. However, developers often fail to implement it properly [24]. *Consequently*, this leaves the communication channel over SSL/TLS insecure and susceptible to man-in-the-middle attacks [18]. *Symptom:* The presence of a `X509TrustManager` or a `HostNameVerifier` that does not perform any validity check. The `TrustManager` may only use `checkValidity` to assess the expiration of a certificate without any further check, *e.g.*, verifying the certificate's signature or asking the user consent to trust a self-signed certificate. Overridden `onReceiveSslError` in WebView that ignores any certification errors. *Mitigation:* Ensure the certificate chain is valid, *i.e.*, the root certificate of the chain is issued by a trusted authority, none of the certificates in the chain are expired, and each certificate in the chain is signed by its immediate successor in the chain. Moreover, the certificate should match its designated destination, *i.e.*, the "Common Name" field or the "Subject Alternative Name" in the certificate should match the domain name of the server being connected to. Finally, utilize network security testing tools like "Nogotofail"[5] to examine your communication.

- **Unacknowledged Distribution**
  Google Play, Google's official marketplace for Android, strives to identify potential security enhancements when an app is uploaded to it. However, developers may distribute their packages via other channels to circumvent out of order updates, bypassing the slow release cycles and security restrictions of this market place. *Issue:* The protection provided by Google, including code and signature checks, is neglected. *Consequently*, the risk of distributing a vulnerable app increases especially when the app utilizes uncertified libraries, or in a worse case, an attacker can replace installation packages with malicious ones [118]. *Symptom:* The `android.permission.INSTALL_PACKAGES` permission exists in the manifest. *Mitigation:* Distribute your apps and updates exclusively through official app stores that perform security checks.

[5] GitHub project website: nogotofail, `https://github.com/google/nogotofail`, *accessed on 28-FEB-2022*

4.1.3 Broken Access Control

- **Unauthorised Intent Receipt**
  An *intent* is an abstract specification of an operation that apps can use to utilize the actions provided by other apps. An *explicit* intent guarantees communication with the specified recipient, but it is the Android system that determines the recipient(s) of an *implicit* intent among available apps. *Issue:* Any app that declares itself able to serve the requested operation is potentially eligible to fulfill the intent. *Consequently*, if such an app is malicious, a threat called *intent hijacking* could arise in which user information carried by the intent could be manipulated or leaked [16]. *Symptom:* The existence of an intent with private data, but without a particular component name, *i.e.*, the fully-qualified class name. *Mitigation:* Only use explicit intents for sending sensitive data. In addition, always validate the results returned from other components to ensure they comply with your expectation.

- **Unconstrained Inter-Component Communication**
  One app can reuse components, *e.g.*, activities, services, content providers, and broadcast receivers of other apps, provided those apps permit it. *Issue:* Android apps are independently restricted in accessing resources. *Consequently*, a threat called *component hijacking* arises when a malicious app escalates its privilege for originally prohibited operations through other apps that can perform those operations [107, 20]. *Symptom:* The existence of the `intent-filter` element or `android:exported = true` attribute in the manifest file without any permission check to ensure that a client app is originally permitted to receive that service. *Mitigation:* Exclusively export components that are meant to be accessed from other apps and avoid placing any critical state changing actions within such components. Enforce custom permissions with the `android:permission` attribute to prohibit access from apps with lower privileges. Finally, use tools like *IccTA*, which detects flaws in inter-component communication [54].

- **Unprotected Unix Domain Socket**
  Android IPCs do not support cross-layer IPC, *i.e.*, communication between an app's Java and native processes or threads. To circumvent this limitation developers resort to using Unix domain sockets. Moreover, developers may reuse Linux code that already utilizes such sockets. *Issue:* Developers are barely guided to protect Unix domain sockets with appropriate authentication. *Consequently*, adversaries are capable of abusing these exposed IPC channels to exploit vulnerabilities within privileged system daemons and the kernel [86]. *Symptom:* The server socket channel accepts clients without performing

any authentication or similarly a client connects to a server without properly authenticating the server. *Mitigation:* Enforce proper security checks when using the sockets.

- **Exposed Adb-level Capability**
  Android debug bridge (Adb) is a versatile tool that provides communication with a connected Android device. Many developers opt for Adb-level capabilities to legitimately access a subset of signature-level resources [56]. *Issue:* For this purpose, an app communicates locally with an Adb-level proxy through the TCP sockets opened on the same device, which exposes the Adb server to any app with the INTERNET permission. *Consequently*, a malign app with ordinary permissions can command the Adb and establish serious attacks [44]. *Symptom:* The existence of Adb-specific commands or TCP connection to local host in the code. *Mitigation:* Avoid using Adb-level capabilities in your app, as it is also prohibited since Android 6.0.

- **Debuggable Release**
  During app development there exist two major build configurations, *i.e.*, debug and release. The first is meant for active development, while the latter is for signed in-market releases. However, developers may forget to switch to release mode before publishing an app [112]. *Issue:* Apps shipped with debugging enabled always try to connect to a local Unix socket opened by the Android debug bridge (Adb). While Adb is not running on every consumer device, a malign app could disguise itself as an Adb service and connect to random debuggable apps. *Consequently*, a malicious app is able to gain full access to the Java process and can execute arbitrary code in the context of the debuggable app.[6] *Symptom:* The manifest file contains the attribute `android:debuggable = true`. *Mitigation:* The debug mode should be disabled in the signed release version, *i.e.*, either the debuggable attribute should not exist in the manifest file, or its value should be false. More recent build environments already perform this task automatically.

- **Custom Scheme Channel**
  Scheme channels, also known as protocol prefixes, like `fblite://` for Facebook allow seamless interactions between web and Android apps. *Issue:* The sender of a scheme message is not able to verify the recipient of the message so that malign apps could register themselves as a receiver of another app's unified resource identifier (URI) scheme. *Consequently*, adversaries could collect access

[6] F-Secure labs: debuggable apps in Android market, `https://labs.mwrinfosecurity.com/blog/debuggable-apps-in-android-market`, *accessed on 28-FEB-2022*

tokens or other sensitive information [98]. *Symptom:* The registration of a URI scheme within the `intent-filter` of the manifest file. The `SchemeRegistry.register` method is in the code. *Mitigation:* Adopt the dedicated system scheme, *i.e.*, `Intent`, which is harder to compromise.

4.1.4 Sensitive Data Exposure

- **Header Attachment**
  The header section of data transport protocols like HTTP comprises key-value pairs to store operational parameters. *Issue:* Developers may rely on headers to transfer sensitive data, *e.g.*, they store credentials to auto-login into a service. *Consequently*, any adversary eavesdropping on the network may easily access the attached data [98]. *Symptom:* Calls like `HttpGet.addHeader()` are present in the code to store private data. *Mitigation:* Do not store sensitive data in headers, but instead use dedicated mechanisms like OAuth2 protocol[7] that rely on tokens to authenticate to third-party services.

- **Unique Hardware Identifier**
  Each device often has a couple of globally unique identifiers such as the IMEI number, MAC address, *etc. Issue:* For various purposes like user profiling, apps utilize these IDs, which are tied to each device [50, 109]. *Consequently*, anyone in the possession of such IDs would be able to track the user's activities across various sources. *Symptom:* Method calls that return IDs from associated classes like `TelephonyManager` or `BluetoothAdapter` exist in the code. *Mitigation:* Use the `UUID.randomUUID()` API to ensure that the retrieved ID is globally unique for each app installation.

- **Exposed Clipboard**
  Users usually rely on a clipboard to copy and paste data across apps. *Issue:* The clipboard content is readable and writable by all apps. *Consequently*, a malign app could perform versatile attacks on the clipboard content from URL hijacking to data exfiltration and code injection [116]. *Symptom:* The related calls on `ClipboardManager` exist in the code. The app uses the common `TextView` and `EditText` controls, which allow copy and paste to handle sensitive data [64]. *Mitigation:* Never allow sensitive data to be copied and pasted in your app. Perform input validation before exercising any input from the clipboard.

- **Exposed Persistent Data**
  Android provides various storage options to store persistent data.

[7] OAuth 2.0: the industry-standard protocol for authorization, `https://oauth.net/2`, *accessed on 28-FEB-2022*

These options vary depending on the size, type, and accessibility of data.[8] *Issue:* Developers may opt for a particular option without considering its security implication. *Consequently*, they expose private data. *Symptom:* The existence of a private storage with global access scope, *i.e.*, `MODE_WORLD_READABLE` or `MODE_WORLD_WRITEABLE` in the app [45]. The app relies on `ContentProvider` to access data, but there is no access restriction for other apps. *Mitigation:* Specify permissions to protect who can access your shared data. Encrypt any internally or especially externally stored sensitive data, and place the encryption key in KeyStore, protected with a user password that is not stored on the device.

- **Insecure Network Protocol**
  Data transportation channels exist in various flavours, and insecure ones like HTTP are more prevalent and easy to maintain. *Issue:* Insecure channels transfer data without encryption per se. *Consequently*, an attacker can secretly relay the data and possibly alter it [69]. *Symptom:* APIs related to opening insecure network connections like `http` or `ftp` exist in the code. *Mitigation:* All app traffic should happen over a secure channel. Otherwise, any sensitive data should be encrypted before it is sent out. Android 6.0 or above provides the `cleartextTrafficPermitted` property, which protects apps from any usage of cleartext traffic.

- **Exposed Credential**
  Passwords, private keys, secret keys, certificates, and other similar credentials are commonly used for authentication, communication, or data encryption. *Issue:* In some circumstances such data is inadvertently disclosed to unauthorised parties. *Consequently*, this could break the intended security. *Symptom:* The app contains hard-coded credentials, or they are stored without any password protection such as when the `KeyStore.ProtectionParameter` is null [62]. *Mitigation:* Store such data in a KeyStore in a protected format, which restricts unauthorised accesses.

- **Data Residue**
  According to recent research, about 80% of abandoned apps are likely to be uninstalled in less than a week [58]. *Issue:* After an app is uninstalled, various types of data associated to the app, ranging from its permissions, operation history, configuration choices, and so on may still remain in a few system services [117]. *Consequently*, such so-called "data residue" can be associated to another app and empower adversaries to access sensitive information [115, 117]. *Symptom:* The

[8] Android documentation: data and file storage overview, `https://developer.android.com/guide/topics/data/data-storage.html`, *accessed on 28-FEB-2022*

app calls system services that are known to be subject to the data residue problem. *Mitigation:* Unfortunately, an app may not always be aware of its data being stored in system services, and the mere mitigation is to avoid sharing private data with these services, if possible.

4.1.5 Lax Input Validation

- **XSS-like Code Injection**
  WebView is an essential component that enables developers to use web technologies such as HTML and JavaScript to deliver web content within an app. Unlike Web browsers such as Chrome, Firefox, *etc.* that are developed by well-recognized companies that we trust, each app using a WebView is like a customized browser, which may not have undergone thorough security tests. *Issue:* An app may load web content unsafely, *i.e.*, without sanitising the input from any code. *Consequently*, an adversary could inject malicious code through any channel that the app uses to get web content [46]. *Symptom:* The `setJavaScriptEnabled` call with value `true` that enables execution of JavaScript exists in the code, and the app fetches web content from untrustworthy sources, *e.g.*, by calling `loadUrl` or `loadData` on `WebView` without applying proper sanity checks. *Mitigation:* Invoke the default browser to display untrusted data. Use a HTML sanitizer to filter out any code inside the data, and show plain text only using safe APIs that are immune to code injection, *i.e.*, do not execute JavaScript code. Beware of third-party libraries that employ WebView. Disable JavaScript if you do not need it.

- **Broken WebView's Sandbox**
  There is a sandbox inside WebView that separates its JavaScript from the rest of the system. *Issue:* WebView provides an API, *i.e.*, the `addJavascriptInterface` through which an app can access Java APIs, and therefore mobile resources from within JavaScript code inside the sandbox. *Consequently*, if the app renders the web content unsafely, a code injection attack is possible [46]. *Symptom:* In addition to the symptoms of the previous issue, the `addJavascriptInterface` call exists in the code. *Mitigation:* Take into account the suggestions of the previous issue, and as well use the `@JavascriptInterface` annotation to specify any method that is exposed by JavaScript to prevent reflection-based attacks.

- **Dynamic Code Loading**
  Android allows apps to load and execute external code and resources. *Issue:* Although dynamic code loading is widely adopted, developers are often unaware of the risks associated to this generally unsafe mechanism or fail to implement it securely [72]. An attacker can

replace the code that is to be loaded with a malicious one. *Consequently*, this can lead to severe vulnerabilities such as remote code injection [25]. *Symptom:* Use of any class loader in the code. In case of loading the code and resources of another installed app, a call to `createPackageContext()` on the `Context` object exists in the code. *Mitigation:* Either bundle the required resources within each app package, or verify the integrity and authenticity of the loaded code, *e.g.*, by imposing restrictions on its location or provenance [96]. Analyze your app with the help of tools like *Grab 'n Run* [25].

- **SQL Injection**
  Data-driven apps organize their data through a database. *Issue:* An app might directly use inputs to build a query that will be run by the database engine. *Consequently*, an adversary who succeeds at inserting malicious code into SQL statements, can access or modify database data [23]. *Symptom:* Inputs from untrustworthy sources are passed to the database without proper validation. *Mitigation:* Instead of dynamic SQL code generation, rely on parameterized queries and stored procedures, which let the database distinguish between code and data. Validate inputs and filter suspicious values, *e.g.*, *escape characters* to ensure they do not end up in the query.

4.2 Empirical Study

We developed a lightweight analysis tool that statically detects known security smells in an app. We rely on the Apktool to reverse engineer Android APK files and generate Smali code, which is a human readable representation of byte code.[9] We defined a set of rules to capture the symptoms of each security smell. In particular, we utilize the Java XML Parser for parsing the manifest files and use regular expressions to define and match the code pattern corresponding to the identified symptoms of each smell in the code. Since some smells require additional context, we could only assess ten of the 28 reported security smells with our lightweight analysis tool.

We randomly selected our apps from the AndroZoo dataset.[10] This dataset currently provides more than 5.5M apps collected from several sources. We initially collected a random subset of 70 000 apps whose sources are in Google Play. However, to collect more metadata such as an app's category, its number of downloads, update cycle, and star rating we still needed to visit the Google Play website. Unfortunately, we could not access 25 000 apps for various reasons, for example, because they were no

[9] Apktool: a tool for reverse engineering Android APK files, `https://ibotpeaches.github.io/Apktool`, *accessed on 28-FEB-2022*

[10] AndroZoo: a growing collection of Android apps, `https://androzoo.uni.lu`, *accessed on 28-FEB-2022*

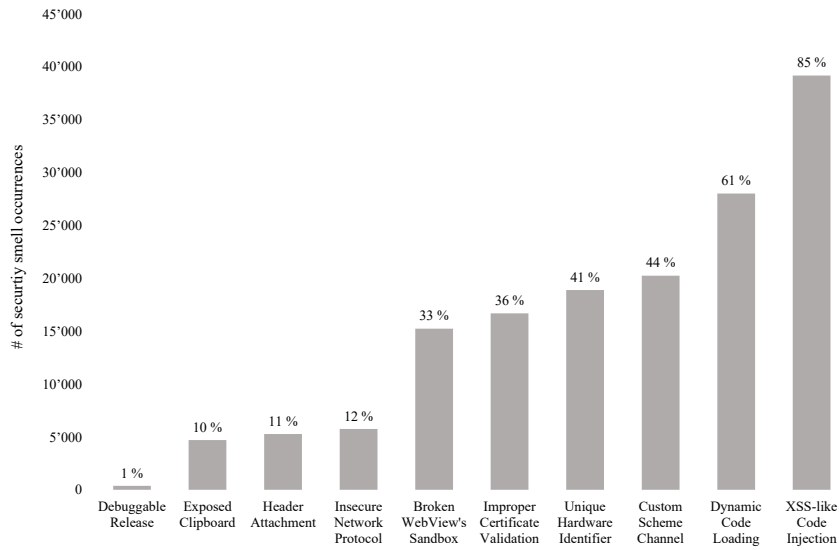


Figure 4.1: Distribution of security smells in the apps

longer available on the store, or they were not accessible from Switzerland. In the end, we included 46 000 benign apps in our dataset. About 90% of these apps were released between 2014 and 2016, a quarter of them were updated within three months, the majority were rated more than four stars, slightly more than 27% were downloaded above 50 000 times, and the median APK size was 5.5MB. The corresponding dataset is publicly available on Figshare.[11]

4.2.1 Result

We applied our lightweight tool to all apps in the dataset. Figure 4.1 shows how prevalent the smells are in our dataset, where the y-axis describes the number of security smell occurrences. A majority of apps potentially suffer from XSS-like code injection (85%) followed by dynamic code loading (61%). About 44% use custom scheme channels and expose a unique hardware identifier. More than 12% use an insecure network protocol, and almost 10% are subject to header attachment as well as clipboard issues. Finally, just under 1% of the apps have debug mode enabled.

We also studied how many of security smells usually appear in the apps. The results are shown in Figure 4.2, where the x-axis describes the number of different security smells an app suffers, *i.e.*, zero to eight smells, and the y-axis describes the number of apps. Only 9% of apps are free of

[11] Figshare: replication package security code smells in Android, `https://figshare.com/s/4f3b4bbc161d600b9ffb`, *accessed on 30-MAR-2022*

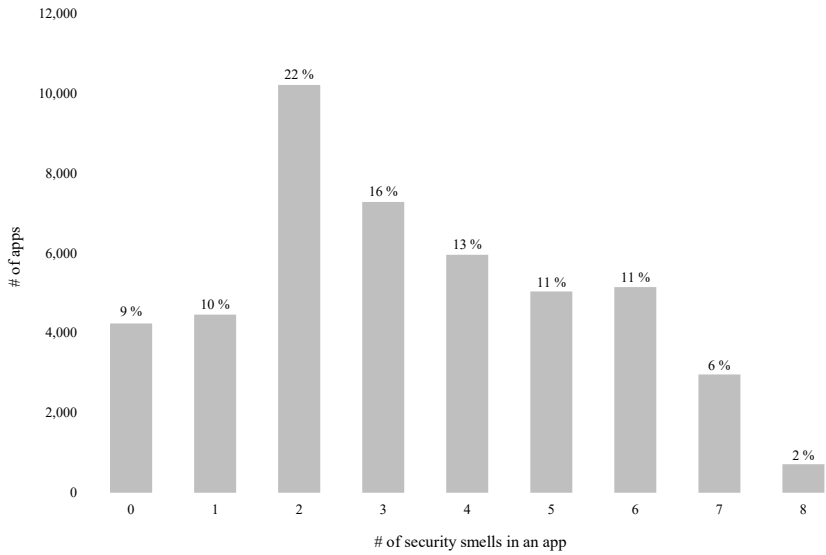


Figure 4.2: Partitioning apps by number of security smells

any smell, a majority, *i.e.*, above 50% suffer from at least three different smells, and over a quarter are subject to more than four smells, which is catastrophic.

We also investigated the prevalence of security smells at different API levels, *i.e.*, different releases of the Android OS apps are optimized for, as the proportion of devices running different API versions varies. Figure 4.3 shows the distribution of smells within each API level, *i.e.*, the x-axis denotes the different Android API levels that an app can target, and the y-axis shows the relative distribution of the different smells for a particular Android API level. We noticed that the prevalence of *Debuggable Release* has been dramatically reduced. We believe this is mainly due to the fact that the Google Play store no longer accepts apps in debug mode. We conjecture this issue should have decreased also in other app markets without this constraint as recent build platforms automatically disable the debug mode in the signed release version of an app. In contrast, there is an increase in the existence of the *Exposed Clipboard* security smell. This could stem from the many sharing options for social media in the apps. Similarly, the issue of *Dynamic Code Loading* has become more common. We observed that many developers adopt this feature to implement their own update mechanisms.

Figure 4.4 shows how many of these classes of smells appear within each API level, *i.e.*, the x-axis denotes the different Android API levels, and the y-axis shows the number of security smell classes an app suffers on average. The crosses represent the mean value of the number of different

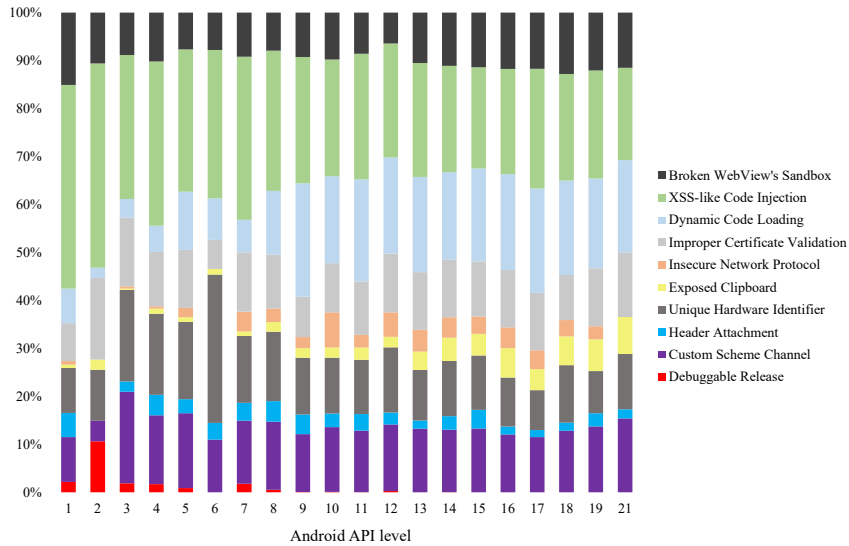


Figure 4.3: The distribution of security smells within each API level

security smell classes apps are suffering from each category, and, as we hid any outliers to increase readability, these values can exceed the first quartiles. All box plots in this chapter use the same configuration. There is a correlation between feature availability and feature usage, and apparently these uses have introduced more insecurity. It seems the peak of issues was reached around API level 15 at which an app suffers on average the most from security smells.

In the remainder of this subsection we discuss our findings from a few more perspectives.

Category

Figure 4.5 shows the number of different security smells appearing in the apps in each category, *i.e.*, the x-axis denotes the different Android Play store app categories, and the y-axis shows the number of security smell categories an app suffers on average. The apps in the *Libraries and Demo* category are the most secure ones as they usually rely on local content. We noted that security smells are prevalent in gaming apps, and that *Casino* and *Role Playing* games are more problematic. Finally, *Dating* as well *Food and Drink* apps suffer from the highest number of security smells.

Popularity

Figure 4.6 shows the relationship between the number of downloads and the security smells, *i.e.*, the x-axis denotes the number of app downloads

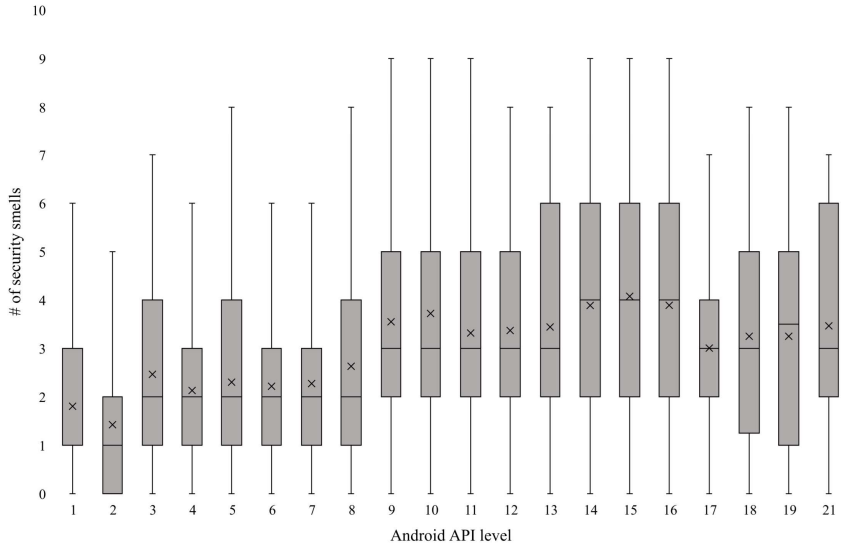


Figure 4.4: Average number of smells within an app targeting a particular API level

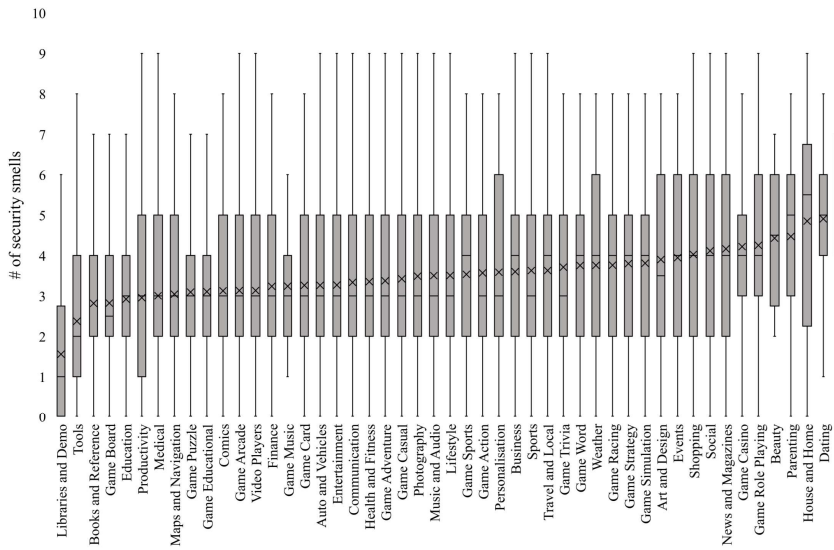


Figure 4.5: Distribution of smells in app categories

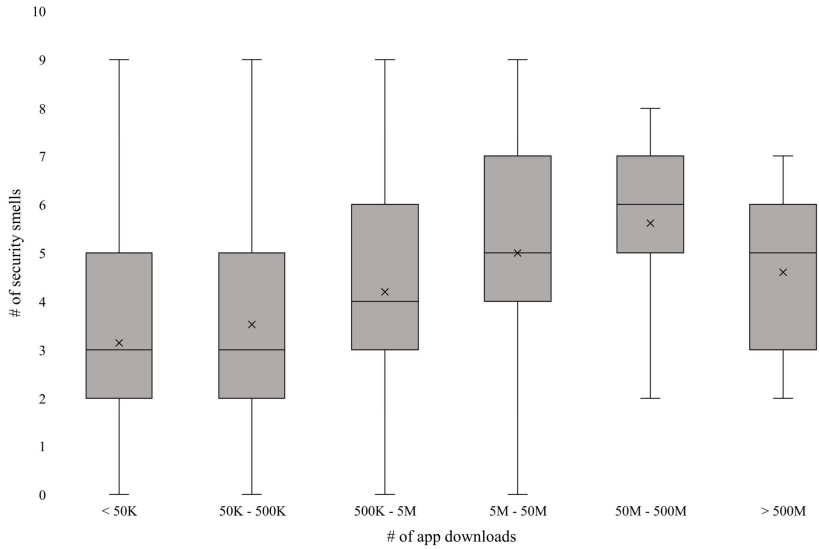


Figure 4.6: The relationship between number of smells and number of downloads

from the Google Play store, and the y-axis shows the number of security smells an app suffers on average. The majority of apps with millions of downloads suffer from five kinds of smells. Although about 73% of apps within our dataset were downloaded less than 50 000 times, there were still enough apps with more downloads to conclude that the number of downloads never guarantees security.

Figure 4.7 shows the relationship between the number of security smells and star ratings, *i.e.*, the x-axis denotes the Google Play store star rating of an app, and the y-axis shows the number of security smells an app suffers on average. Despite the number of stars, apps often suffer from three kinds of security smells. In particular, the star rating correlates negatively with the presence of security smells. We assume that the studied security smells are barely noticeable by end users, hence they are not reflected in the ratings.

Release date

We further studied whether the prevalence of security smells changes over time. In fact, with advances in developers' support, *e.g.*, tools, learning resources, we expected that security smells in more recent apps should be rarer than in older apps. Nonetheless, the result showed that neither the number of smells nor the likelihood of a particular smell relates to the release date of an app. Moreover, we noted that in general the security of apps with short update cycles is similar to those with longer update

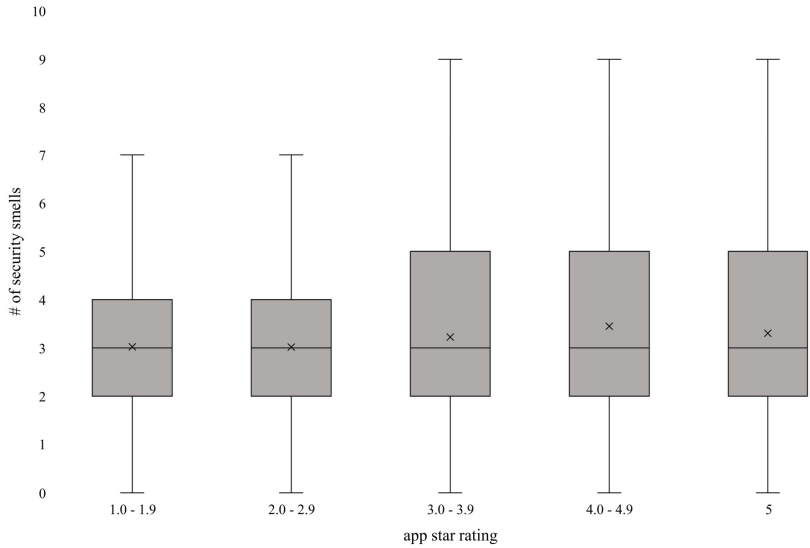


Figure 4.7: The relationship between number of smells and app star ratings

cycles. That is, either security issues in one release still remain in future releases, or they get fixed but new releases also introduce new smells.

App Size

We were interested to know whether the existence of security smells is ever related to the size of an app. Our investigation showed that an app may suffer from various kinds of security smells, despite its size. In fact, an increase in app size may only increase the frequency of a security smell. It is also worth mentioning that some apps are larger not because of having more code but other resources such as image, video and audio content.

4.2.2 Manual Analysis

To assess how reliable these findings are to detect security vulnerabilities, we manually analyzed 160 apps. For each smell, we inspected 20 random apps manually and compared our findings to the result of the lightweight analysis tool. As is shown in Figure 4.8, the results were encouraging. The manual analysis completely agreed with the tool in the security risk associated with six security smells. In case of exposed clipboard the tool achieved a very good performance, *i.e.*, above 90% agreement with the manual analysis. The level of agreement in insecure network protocol and improper certificate validation was 80%. We realized some apps use http connections to exclusively load local contents, which is legitimate in development frameworks like Apache Cordova or Adobe PhoneGap. And

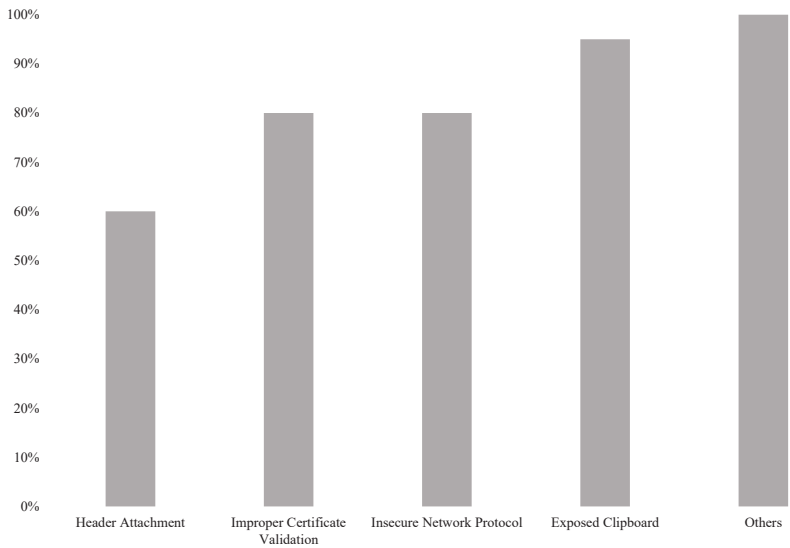


Figure 4.8: The precision of obtained results

some apps implemented their own custom `TrustManager`, which in fact was secure. Finally, our tool was unable to correctly detect the security risk associated with header attachment in 40% of cases, which is mainly due to the fact that discerning data sensitivity is non-trivial.

4.2.3 Threats to Validity

We note several limitations and threats to validity of the results pertinent to our research. One important threat is the completeness of this study, *i.e.*, whether we could identify and study all related papers in the literature. We could not review all the publications, but we strived to explore top-tier software engineering and security journals and conferences as well as highly cited work in the field. For each relevant paper we also recursively looked at both citations and cited papers. Moreover, to ensure that we did not miss any important paper, for each identified issue we further constructed more specific queries and looked for any new paper on Google Scholar.

We analyzed the existence of security smells in the source code of an app, whereas third-party libraries could also introduce smells.

We were only interested in studying benign apps as in malicious ones developers may not spend any effort to accommodate security. Thus, we merely collected apps, which were available on the official Google market. However, our dataset may still have malicious apps that evaded the security checks of the market.

Finally, the fact that the results of our lightweight analysis tool are validated against manual analysis performed by the authors is a threat to construct validity through potential bias in experimenter expectancy.

4.3 Conclusion

In contrast to all advances in software security, software systems are suffering from increasing security and privacy issues. Security in Android, the dominant mobile platform, is even more crucial as these devices often contain manifold sensitive data, and a security issue in a small home-brewed app can threaten the security of billions of users.

We reviewed state of the art papers in security and identified 28 smells whose presence may indicate a security issue in an app. We developed a static analysis tool to study the prevalence of ten of such smells in 46 000 apps. We realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security smells. Moreover, the manual inspection of 160 apps showed that the identified security smells are actually a good indicator of security vulnerabilities. Consequently, we promote the adoption of secure programming practices to fundamentally reduce the attack surface in Android.

Chapter 5

Security Code Smells in Android ICC

Declaration of Content Reuse

The content of this chapter is based on the journal extension submission *Security Code Smells in Android ICC* that has been accepted for the journal of *Empirical Software Engineering (EMSE)* in 2019 [34], which extends the previous chapter to better comprehend the aspect of inter-component communication.

In the previous chapter, we identified 28 security code smells, *i.e.*, symptoms in the code that signal potential security vulnerabilities. We studied the prevalence of ten such smells, and realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security smells, and such smells are in fact good indicators of actual security vulnerabilities.

To promote the adoption of secure programming practices, we build on our previous work, and identify security smells related to Android Inter-Component Communication (ICC). Android ICC is complex, largely unconstrained, and hard for developers to understand, and it is consequently a common source of security vulnerabilities in Android apps.

We have reviewed state of the art papers in security and existing benchmarks for Android vulnerabilities, and identified twelve security code smells pertinent to ICC vulnerabilities. In this chapter, we present these vulnerabilities and their corresponding smells in the code, and discuss how they could be eliminated or mitigated during development. We present a lightweight static analysis tool on top of Android Lint that analyzes the code under development, and provides just-in-time feedback within the IDE about the presence of such security smells in the code. Moreover, with the help of this tool we study the prevalence of security code smells in more than 700 open-source apps, and discuss the extent to which iden-

tifying these smells can uncover actual ICC security vulnerabilities. We address the following three research questions:

- **RQ**$_1$: *What are the known ICC security code smells?* We have reviewed related work, especially that appearing in top-tier conferences and journals, and identified twelve avoidable ICC vulnerabilities and the code smells that indicate their presence. We discuss each smell, the risk associated with it, and its mitigation during app development.

- **RQ**$_2$: *How prevalent are the smells in benign apps?* We have developed a tool that statically analyzes apps for the existence of ICC security smells, and we applied it to a repository of 732 apps, mostly available on GitHub. We discovered that almost all apps suffer from at least one category of ICC security smell, but fewer than 10% suffer from more than two categories of such smells. Interestingly, only small teams appear to be capable of consistently building software resistant to most security code smells. Furthermore, long-lived projects have more issues than recently created ones, and updates rarely have any impact on ICC security.

- **RQ**$_3$: *To which extent does identifying security smells facilitate detection of security vulnerabilities?* We inspected the identified smells in 100 apps, and verified whether they correspond to any security vulnerabilities. Our investigation showed that about half of the identified smells are in fact good indicators of security vulnerabilities.

To summarize, this work represents an effort to spread awareness about the impact of programming choices in making apps secure, and to fundamentally reduce the attack surface of ICC APIs in Android. We argue that this helps developers who develop security mechanisms to identify frequent problems, and also provides developers inexperienced in security with caveats about the prospect of security issues in their code. Existing analysis tool reports often overwhelm developers with too many identified issues at once. In contrast we provide feedback during app development where developers have the relevant context. Such feedback makes it easier to react to issues, and helps developers to learn from their mistakes [97]. This chapter goes beyond our earlier work in chapter 4 by (i) providing a completely new study on ICC vulnerabilities, one of the most prevalent Android security issues, and identifying the corresponding security smells, (ii) providing more precise, while still lightweight, static analysis tool support to identify such smells, (iii) integrating our analysis into Android Lint, thus providing just-in-time feedback to developers, (iv) experimentation on a new dataset of open-source Android apps, and (v) open-sourcing the Lint checks as well as the analyzed data. We further collaborate with Google to officially integrate these checks into Android Studio.

The remainder of this chapter is organized as follows. We introduce ICC-related security code smells in section 5.1, followed by our empirical study in section 5.2. We conclude the chapter in section 5.3.

5.1 ICC Security Code Smells

In this section we present the guidelines we followed to derive the security code smells from previous research. Finally, we explain each security smell in detail.

5.1.1 Literature Review

Although Android security is a fairly new field, it is very active, and researchers in this area have published a large number of articles in the past few years. In order to answer the first research question (RQ_1), and to draw a comprehensive picture of recent ICC smells and their corresponding vulnerabilities, our study builds on two pillars, *i.e.*, a literature review and a benchmark inspection.

We were essentially interested in any paper that matches our scope, *i.e.*, explaining an ICC-related issue, and any countermeasures that involve ICC communication in Android. Therefore, we considered for our analysis multiple online repositories, such as IEEE Xplore and the ACM Digital Library, as well as the Google Scholar search engine. In each repository we formulated a search query comprising *Android*, *ICC*, *IPC* and any other security-related keywords such as *security*, *privacy*, *vulnerability*, *attack*, *exploit*, *breach*, *leak*, *threat*, *risk*, *compromise*, *malicious*, *adversary*, *defence*, or *protect*. In addition to increase our potential coverage on Android security, we also collected all related publications in recent editions of well-known software engineering venues like the *International Conference on Software Engineering (ICSE)*. This search led initially to 358 publications.

In order to retrieve only relevant information that lies within our scope, *i.e.*, *Android application level ICC security*, we first read the title and abstract, and if the paper was relevant we continued reading other parts.

This process led to the inclusion of 47 papers in our study. We recursively checked both citations and cited papers until no new related papers were found. This added six new relevant papers in our list that in the end contained 53 relevant papers for an in-depth study, out a total of 430 papers. During the entire process, which was undertaken by two authors of this paper, we resolved any disagreement by discussions. The list of included papers in this study is available on the GitHub page of the project.[1]

[1]GitHub project website: AndroidLintSecurityChecks, `https://github.com/pgadient/AndroidLintSecurityChecks`, *accessed on 28-FEB-2022*

We further studied the well-known DroidBench[2] and Ghera[3] benchmarks for our evaluation, both built with a focus on ICC. We relied on their technical implementation, or description where possible, to extract the desired information, *i.e.*, issues under test, symptoms, and vulnerabilities. The inspection of these two benchmarks served two different purposes: on the one hand we wanted to ensure there are no smells that we might have missed to include in our list. On the other hand, we wanted to rely on some ground truth, while explaining and examining the vulnerability capabilities of the smells.

5.1.2 List of Smells

We have identified twelve ICC security code smells that are listed in Table 5.1. For each smell we report the security *issue* at stake, the potential security *consequences* for users, the *symptom* in the code, *i.e.*, the code smell, the *detection* strategy that has been implemented by our tool for identifying the code smell, any *limitations* of the detection strategy, and a recommended *mitigation* strategy of the issue, principally for developers. One of these ICC-related smells, namely *Custom Scheme Channel*, has been mentioned in subsection 4.1.3, however in this chapter we focus particularly on the aspect of ICC.

ID	Security code smells	ID	Security code smells
SM01	Persisted Dynamic Permission	SM07	Broken Service Permission
SM02	Custom Scheme Channel	SM08	Insecure Path Permission
SM03	Incorrect Protection Level	SM09	Broken Path Permission Precedence
SM04	Unauthorized Intent	SM10	Unprotected Broadcast Receiver
SM05	Sticky Broadcast	SM11	Implicit Pending Intent
SM06	Slack WebViewClient	SM12	Common Task Affinity

Table 5.1: The identified ICC security code smells

We mined this information from numerous publications and benchmark suites, but only few of these resources provided detailed information about a given security issue. Therefore, we put in a great manual effort to provide a comprehensive description for each smell, while consulting other resources such as the official Android documentation and external experts. For instance, authors who focused on vulnerability detection generally neglected the aspect of mitigation. This is very problematic, since it is very common for ICC-related issues to share strong similarities with only subtle differences, *e.g.*, regular directed inter-app communication

[2]GitHub project website: DroidBench, `https://github.com/secure-software-engineering/DroidBench`, *accessed on 28-FEB-2022*

[3]Bitbucket project website: android-app-vulnerability-benchmarks, `https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks`, *accessed on 28-FEB-2022*

and broadcasts both rely on intents. Furthermore, manifold vulnerability terms that are used in the literature insufficiently reflect the symptoms as they do not name the involved component, *e.g.*, "Confused Deputy" instead of "Unauthorized Intent." Better naming conventions would greatly ease the understanding of security vulnerabilities.

- **SM01: Persisted Dynamic Permission**. Android provides access to protected resources through a Uniform Resource Identifier (URI) to be granted at run time.
  Issue: Such dynamic access is intended to be temporary, but if the developer forgets to revoke a permission, the access grant becomes more durable than intended.
  Consequently, the recipient of the granted access obtains long-term access to potentially sensitive data [26].
  Symptom: `Context.grantUriPermission()` is present in the code without a corresponding `Context.revokeUriPermission()` call.
  Detection: We report the smell when we detect a permission being dynamically granted without any revocations in the app.
  Limitation: Our implementation does not match a specific grant permission to its corresponding revocation. We may therefore fail to detect a missing revocation if another revocation is present somewhere in the code.
  Mitigation: Developers have to ensure that granted permissions are revoked when they are no longer needed. They can also attach sensitive data to the intent instead of providing its URI.

- **SM02: Custom Scheme Channel**. A *custom scheme* allows a developer to register an app for custom URIs, *e.g.*, URIs beginning with `myapp://`, throughout the operating system once the app is installed. For example, the app could register an activity to respond to the URI via an intent filter in the manifest. Therefore, users can access the associated activity by opening specific hyperlinks in a wide set of apps.
  Issue: Any app is potentially able to register and handle any custom schemes used by other apps [58].
  Consequently, malicious apps could access URIs containing access tokens or credentials, without any prospect for the caller to identify these leaks [98].
  Symptom: If an app provides custom schemes, then a scheme handler exists in the manifest file or in the Android code. If the app calls a custom scheme, there exists an intent containing a URI referring to a custom scheme.
  Detection: The `android:scheme` attribute exists in the `intent-filter` node of the manifest file, or `IntentFilter.addDataScheme()` exists in the source code.
  Limitation: We only check the symptoms related to receiving cus-

tom schemes.
Mitigation: Never send sensitive data, *e.g.*, access tokens via such URIs. Instead of custom schemes, use system schemes that offer restrictions on the intended recipients. The Android OS could maintain a verified list of apps and the schemes that are matched when there is such a call.

- **SM03: Incorrect Protection Level**. Android apps must request permission to access sensitive resources. In addition, custom permissions may be introduced by developers to limit the scope of access to specific features that they provide based on the protection level given to other apps. Depending on the feature, the system might grant the permission automatically without notifying the user, *i.e.*, signature level, or after the user approval during the app installation, *i.e.*, normal level, or may prompt the user to approve the permission at run time, if the protection is at a dangerous level.
Issue: An app declaring a new permission may neglect the selection of the right protection level, *i.e.*, a level whose protection is appropriate with respect to the sensitivity of resources.
Consequently, apps with inappropriate permissions can still use a protected feature [62].
Symptom: Custom permissions are missing the right `android:protectionLevel` attribute in the manifest file.
Detection: We report missing protection level declarations for custom permissions.
Limitation: We cannot determine if the level specified for a protection level is in fact right.
Mitigation: Developers should protect sensitive features with dangerous or signature protection levels.

- **SM04: Unauthorized Intent**. Intents are popular as one way requests, *e.g.*, sending a mail, or requests with return values, *e.g.*, when requesting an image file from a photo library. Intent receivers can demand custom permissions that clients have to obtain before they are allowed to communicate. These intents and receivers are "protected."
Issue: Any app can send an unprotected intent without having the appropriate permission, or it can register itself to receive unprotected intents.
Consequently, apps could escalate their privileges by sending unprotected intents to privileged targets, *e.g.*, apps that provide elevated features such as camera access. Also, malicious apps registered to receive implicit unprotected intents may relay intents while leaking or manipulating their data [16].
Symptom: The existence of an unprotected implicit intent. For intents requesting a return value, the lack of check for whether the

sender has appropriate permissions to initiate an intent.
Detection: The existence of several methods on the `Context` class for initiating an unprotected implicit intent like `startActivity`, `sendBroadcast`, `sendOrderedBroadcast`, `sendBroadcastAsUser`, and `sendOrderedBroadcastAsUser`.
Limitation: We do not verify, for a given intent requesting a return value, if the sender enforces permission checks for the requested action.
Mitigation: Use explicit intents to send sensitive data. When serving an intent, validate the input data from other components to ensure they are legitimate. Adding custom permissions to implicit intents may raise the level of protection by involving the user in the process.

- **SM05: Sticky Broadcast**. A normal broadcast reaches the receivers it is intended for, then terminates. However, a "sticky" broadcast stays around so that it can immediately notify other apps if they need the same information.
Issue: Any app can watch a broadcast, and particularly a sticky broadcast receiver can tamper with the broadcast.
Consequently, a manipulated broadcast may mislead future recipients [62].
Symptom: Broadcast calls that send a sticky broadcast appear in the code, and the related Android system permission exists in the manifest file.
Detection: We check for the existence of methods such as `sendStickyBroadcast`, `sendStickyBroadcastAsUser`, `sendStickyOrderedBroadcast`, `sendStickyOrderedBroadcastAsUser`, `removeStickyBroadcast`, and `removeStickyBroadcastAsUser` on the `Context` object in the code and the `android.permission.BROADCAST_STICKY` permission in the manifest file.
Limitation: We are not aware of any limitations.
Mitigation: Prohibit sticky broadcasts. Use a non-sticky broadcast to report that something has changed. Use another mechanism, *e.g.*, an explicit intent, for apps to retrieve the current value whenever desired.

- **SM06: Slack WebViewClient**. A `WebView` is a component to facilitate web browsing within Android apps. By default, a `WebView` will ask the activity manager to choose the proper handler for the URL. If a `WebViewClient` is provided to the `WebView`, the host application handles the URL.
Issue: The default implementation of a `WebViewClient` does not restrict access to any web page.
Consequently, it can be pointed to a malicious website that entails diverse attacks like phishing, cross-site scripting, *etc.* [62].
Symptom: The `WebView` responsible for URL handling does not per-

form adequate input validation.
Detection: The `WebView.setWebViewClient()` exists in the code but the `WebViewClient` instance does not apply any access restrictions in `WebView.shouldOverrideUrlLoading()`, *i.e.*, it returns `false` or calls `WebView.loadUrl()` right away. Also, we report a smell if the implementation of `WebView.shouldInterceptRequest()` returns `null`.
Limitation: It is inherently difficult to evaluate the quality of an existing input validation.
Mitigation: Use a white list of trusted websites for validation, and benefit from external services, *e.g.*, SafetyNet API,[4] that provide information about the threat level of a website.

- **SM07: Broken Service Permission**. Two different mechanisms exist to start a service: `onBind` and `onStartCommand`. Only the latter allows services to run indefinitely in the background, even when the client disconnects. An app that uses Android IPC to start a service may possess different permissions than the service provider itself.
  Issue: When the callee is in possession of the required permissions, the caller will also get access to the service.
  Consequently, a privilege escalation could occur [62].
  Symptom: The lack of appropriate permission checks to ensure that the caller has access right to the service.
  Detection: We report the smell when the caller uses `startService`, and then the callee uses `checkCallingOrSelfPermission`, `enforceCallingOrSelfPermission`, `checkCallingOrSelfUriPermission`, or `enforceCallingOrSelfUriPermission` to verify the permissions of the request. Calls on the `Context` object for permission check will then fail as the system mistakenly considers the callee's permission instead of the caller's. Furthermore, reported are calls to `checkPermission`, `checkUriPermission`, `enforcePermission`, or `enforceUriPermission` methods on the `Context` object, when additional calls to `getCallingPid` or `getCallingUid` on the `Binder` object exist.
  Limitation: We currently do not distinguish between checks executed in `Service.onBind` or `Service.onStartCommand`, and we do not verify custom permission checks based on the user id with `getCallingUid`.
  Mitigation: Verify the caller's permissions every time before performing a privileged operation on its behalf using `Context.checkCallingPermission()` or `Context.checkCallingUriPermission()` checks. If possible, do not implement `Service.onStartCommand` in

[4] Android documentation: SafetyNet safe browsing API, `https://developer.android.com/training/safetynet/safebrowsing.html`, *accessed on 28-FEB-2022*

order to prevent clients from starting, instead of binding to, a service. Ensure that appropriate permissions to access the service have been set in the manifest.

- **SM08: Insecure Path Permission**. Apps can access data provided by a content provider using path specifications of the form `/a/b/c`. A content provider may restrict access to certain data under a given path by specifying so called path permissions. For example, it may specify that other apps cannot access data located under `/data/secret`. The Android framework prohibits access to unauthorized apps only if the requested path strictly matches the protected path. For instance, `//data/secret` is different from `/data/secret`, and therefore the framework will not block access to it.
  Issue: Developers often use the `UriMatcher` for URI comparison in the `query` method of a content provider to access data, but this matcher, unlike the Android framework, evaluates paths with two slashes as being equal to paths with one slash.
  Consequently, access to presumably protected resources may be granted to unauthorized apps [62].
  Symptom: A `UriMatcher.match()` is used for URI validation.
  Detection: We look for `path-permission` attributes in the manifest file, and `UriMatcher.match()` methods in the code.
  Limitation: We are not aware of any limitation.
  Mitigation: As long as the bug exists in the Android framework, use your own URI matcher.

- **SM09: Broken Path Permission Precedence**. In a content provider, more fine-grained path permissions, for example, on `/data/secret` take precedence over those with a larger scope, *e.g.*, on `/data`.
  Issue: A path permission never takes precedence over a permission on the whole content provider due to a bug that exists in the `ContentProvider.enforceReadPermissionInner()` method. For example, if a content provider has a permission for general use, as well as a path permission to protect `/data/secret` from untrusted apps, then the general use permission takes precedence.
  Consequently, content providers may mistakenly grant untrusted apps access to presumably protected paths [62].
  Symptom: A content provider is protected by path-specific permissions.
  Detection: We look for a `path-permission` in the definition of a content provider in the manifest file.
  Limitation: We are not aware of any limitation.
  Mitigation: As long as the bug exists in Android, instead of path permissions use a distinct content provider with a dedicated permission for each path.

- **SM10: Unprotected Broadcast Receiver**. Static broadcast receivers are registered in the manifest file, and start even if an app is not currently running. Dynamic broadcast receivers are registered at run time in Android code, and execute only if the app is running.
  Issue: Any app can register itself to receive a broadcast, which exposes the app to any other app able to initiate the broadcast.
  Consequently, if there is no permission check, the receiver may respond to a spoofed intent yielding unintended behavior or data leaks [62].
  Symptom: The `Context.registerReceiver()` call without any argument for permission exists in the code
  Detection: We report cases where the permission argument is missing or is `null`.
  Limitation: We are not aware of the permissions' appropriateness.
  Mitigation: Register broadcast receivers with sound permissions.

- **SM11: Implicit Pending Intent**. A `PendingIntent` is an intent that executes the specified action of an app in the future and on behalf of the app, *i.e.*, with the identity and permissions of the app that sends the intent, regardless of whether the app is running or not.
  Issue: Any app can intercept an implicit pending intent and use the pending intent's `send` method to submit arbitrary intents on behalf of the initial sender.
  Consequently, a malicious app can tamper with the intent's data and perform custom actions with the permissions of the originator. Relaying of pending intents could be used for intent spoofing attacks [62].
  Symptom: The initiation of an implicit `PendingIntent` in the code.
  Detection: We report a smell if methods such as `getActivity`, `getBroadcast`, `getService`, and `getForegroundService` on the `PendingIntent` object are called, without specifying the target component.
  Limitation: Arrays of pending intents are not yet supported in our analysis.
  Mitigation: Use explicit pending intents, as recommended by the official documentation.[5]

- **SM12: Common Task Affinity**. A *task* is a collection of activities that users interact with when carrying out a certain job.[6] A task affinity, defined in the manifest file, can be set to an individual

[5] Android documentation: PendingIntent class, `https://developer.android.com/reference/android/app/PendingIntent.html`, *accessed on 28-FEB-2022*

[6] Android documentation: Tasks and the back stack, `https://developer.android.com/guide/components/activities/tasks-and-back-stack.html`, *accessed on 28-FEB-2022*

activity or at the application level.
Issue: Apps with identical task affinities can overlap each others' activities, *e.g.*, to fade in a voice record button on top of the phone call activity. The default value does not protect the application against hijacking of UI components.
Consequently, malicious apps may hijack an app's activity paving the way for various kinds of spoofing attacks [82].
Symptom: The task affinity is not empty.
Detection: We report a smell if the value of a task affinity is not empty.
Limitation: We are not aware of any limitation.
Mitigation: If a task affinity remains unused, it should always be set to an empty string on the application level. Otherwise set the task affinity only for specific activities that are safe to share with others. We suggest that Android set the default value for a task affinity to empty. It may also add the possibility of setting a permission for a task affinity.

In summary, each security smell introduces a different set of vulnerabilities. We established a close relationship between the smells and the security risks with the purpose of providing accessible and actionable information to developers, as shown in Table 5.2.

Vulnerabilities	**Security code smells**
Denial of Service	SM01, SM02, SM03, SM04, SM06, SM07, SM10, SM12
Intent Spoofing	SM02, SM03, SM04, SM05, SM07, SM08, SM09, SM10, SM11
Intent Hijacking	SM02, SM03, SM04, SM05, SM10, SM11

Table 5.2: The relationship between vulnerabilities and security code smells

5.2 Empirical Study

In this section we first present the Lint-based tool with which we detect security code smells, and introduce a dataset of more than 700 open-source Android projects that are mostly hosted on GitHub. We then present the results of our investigation into RQ_2 and RQ_3 by analyzing the prevalence of security smells in our dataset, and by discussing the performance of our tool, respectively.

The results in subsection 5.2.3 suggest that although fewer than 10% of apps suffer from more than two categories of ICC security smells, only small teams are capable of consistently building software resistant to most security code smells. With respect to app volatility, we discovered that updates rarely have any impact on ICC security, however, in case they do, they often correspond to new app features. On the other hand, we found that long-lived projects have more issues than recently created ones,

except for apps that receive frequent updates, where the opposite is true. Moreover, the findings of Android Lint's security checks correlate to our detected security smells.

In subsection 5.2.4, our manual evaluation confirms that our tool successfully finds many different ICC security code smells, and that about 43.8% of the smells in fact represent vulnerabilities. We consequently hypothesize that the tool can offer valuable support in security audits, but this remains to be explored in our future work.

We performed analyses similar to our previous work, *e.g.*, exploring the relation between star rating and smells, or the distribution of smells in app categories, and we did not observe major differences with our past findings in chapter 4. Our results are therefore in line with our prior research that did not consider ICC smells, and found that the majority of apps suffer from security smells, despite the diversity of apps in popularity, size, and release date.

5.2.1 Linting Tool

Our Linting tool is built using Android Lint, a static analysis framework from the official Android Studio IDE[7] for analyzing Android apps. Android Lint provides various rich interfaces for analyzing XML, Java, and Class files in Android. Using these interfaces, one can implement a so-called "detector" that is responsible for scanning code, detecting issues, and reporting them. More specifically, each detector is represented by a Java class that implements Android Lint interfaces to access Android Lint's ASTs (abstract syntax trees) of the app built from XML, source, or byte code. In order to ease the AST traversal, Android Lint provides an implementation of the visitor design pattern with additional helper methods to support further interaction with the tree. The majority of methods use idiomatic names that closely resemble the developer's intention, *e.g.*, `UastUtils.tryResolve()` to resolve a variable, or the class `ConstantEvaluator` to evaluate constants. The latest Android Lint provides more than 300 different detectors to check several categories of issues such as, *e.g.*, Accessibility, Usability, Security, *etc.*

We extended Android Lint by developing twelve new detectors. These detectors implement `UastScanner` and `XmlScanner` interfaces to check the presence of security code smells in source code and manifest files, respectively. The `UastScanner` is the successor of the `JavaScanner`, and, in addition to Java, also supports Kotlin, a new programming language used in the Android platform. We implemented the detection strategies that we introduced for each security smell in section 5.1. The average size of a detector is 115 lines of code.

[7] Android documentation: improve your code with lint checks, `https://developer.android.com/studio/write/lint`, *accessed on 02-MAR-2022*

Android Lint brings analysis support directly into the Android Studio IDE. Developers can therefore receive just-in-time feedback during app development about the presence of security code smells in their code. Detectors are automatically run during programming in the latest Android Studio IDE, *i.e.*, the Canary build and notify developers about the security code smells once they appear in the code under development. Each notification includes an explanation of the smell, mitigation or elimination strategies, as well as a web link to some references.

Linting in batch mode is also possible through the command line interface, given the availability of the successfully built projects. In our experience, a successful build often entails changing build paths, and updating Gradle and its project configurations to a version that is compatible with the current release of Android Lint. We created a script to automate most of this non-trivial process. After a successful build of each project, another script runs the executable of Android Lint, and collects the analysis results in XML files.

The tool is publicly available for download from our GitHub repository.[8]

5.2.2 Dataset

We collected all open-source apps from the F-Droid[9] repository as well as several other apps directly from GitHub.[10] In total we collected 3 471 apps, of which we could successfully build 1 487 (42%). For replication of our results we explicitly provide the package names of all successfully analyzed apps,[11] instead of a binary compilation, because of the dataset's storage space requirements of more than 27 GBytes. In order to reduce the influence of individual projects, in case there existed more than one release of a project, we only considered the latest one. Finally, we were left with 732 apps (21%) in our dataset. The median project size in our dataset is about 1.2 MB, while the median number of data files per project is 108.

5.2.3 Batch Analysis

This section presents the results of applying our tool to all the apps in our dataset.

[8] GitHub project website: AndroidLintSecurityChecks, `https://github.com/pgadient/AndroidLintSecurityChecks`, *accessed on 02-MAR-2020*

[9] F-Droid: a catalogue of free and open-source apps, `https://f-droid.org/`, *accessed on 02-MAR-2022*

[10] GitHub project website: open-source-android-apps, `https://github.com/pcqpcq/open-source-android-apps`, *accessed on 02-MAR-2022*

[11] AndroidLintSecurityChecks: list of analyzed apps, `https://github.com/pgadient/AndroidLintSecurityChecks/blob/master/dataset/analyzed_apps.csv`, *accessed on 02-MAR-2022*

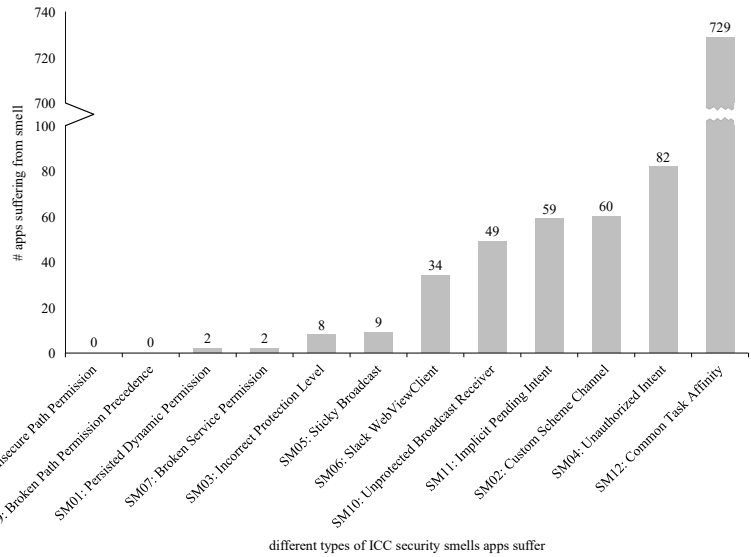


Figure 5.1: Distribution of security smells in the apps

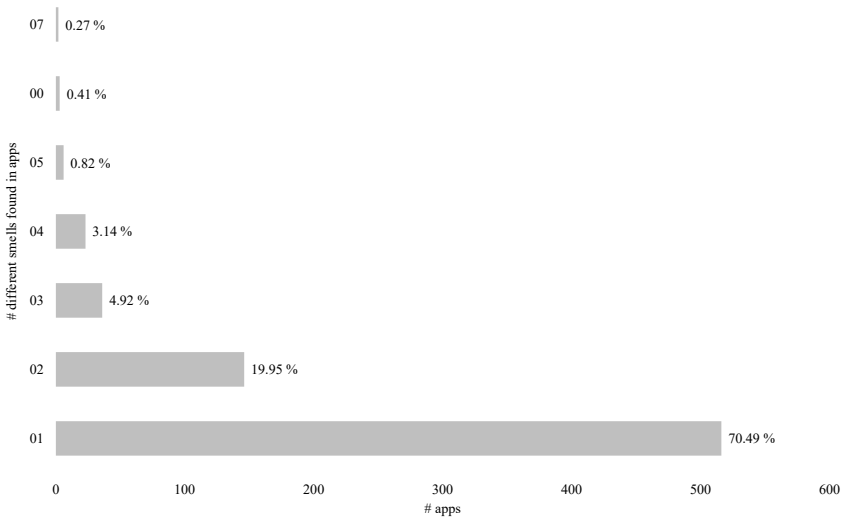


Figure 5.2: Prevalence of different security smells in apps

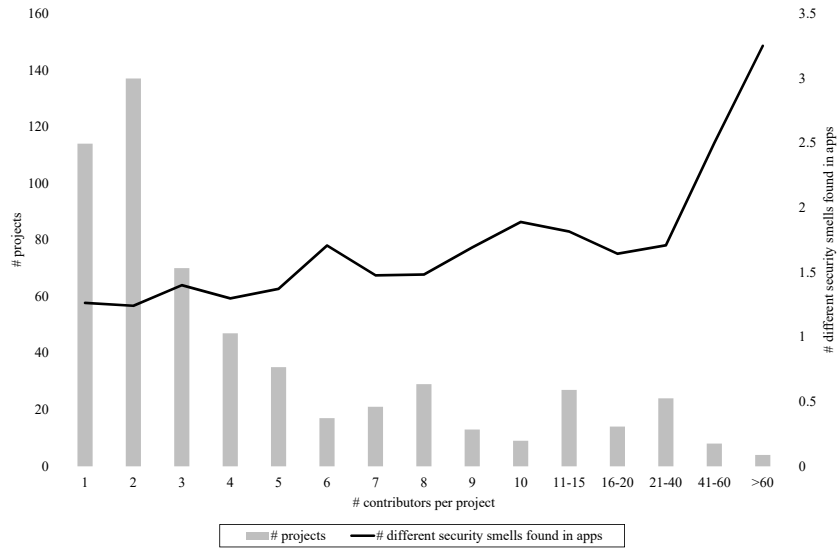


Figure 5.3: Relation between number of a project's participants, its prevalence, and the average number of different security smells found

Prevalence of Security Smells

Figure 5.1 shows how prevalent the smells are in our dataset. Almost all apps suffer from *Common Task Affinity* issues (99%) followed by the much less prevalent *Unauthorized Intent* smell (11%). The default value of task affinity configurations does not protect the application against high-jacking of UI components, and only few developers appear to be aware of the issue and set the property accordingly. *Custom Scheme Channel* and *Implicit Pending Intent* each contribute about 8% of the smells. Furthermore, *WebViewClient* is in line with our observation that apps increasingly rely on web components for their UI. At the other end of the spectrum, *Sticky Broadcast*, *Incorrect Protection Level*, *Broken Service Permission*, and *Persisted Dynamic Permission* cause less than 2% of all issues. The threat of path permissions is not very common, as no apps suffered from SM08 or SM09.

We were also interested in the relative prevalence of different security smells in the apps, which we reveal in Figure 5.2. Less than 1% did not suffer from any security smell at all, whereas the majority of apps, *i.e.*, over 90%, suffered from one or two different smells. 9% of all apps were affected by three or more smells. No apps, fortunately, suffered from more than seven different types of smells. It is important to recall that the more issues that are present in a benign app, the more likely it is that a malign app can exploit it, *e.g.*, with denial of service, intent spoofing, or intent hijacking attacks.

Contributor Affiliation

Figure 5.3 shows the relationship between the number of contributors participating in a project and the mean number of security smell categories apps suffer from. For example, the second last bar represents the number of all projects maintained by 41 to 60 participants, while the line chart shows that projects with this many participants suffer on average from 2.5 security smell categories. We see that most apps are maintained by two contributors, followed by projects developed by individuals. A trend exists that projects with many participants are less common than projects with only a few contributors. The more people are involved in a project the more the security decreases, especially for large teams. More precisely, we found statistical evidence that only small teams of up to five people are capable of consistently building projects resistant to most security code smells, by using the non-parametric Mann-Whitney U test that does not require the assumption of normal distributions for the dataset. The mean different smell occurrences in the groups "projects with one contributor" and "projects with six contributors" were 1.263 and 1.705; the distributions in the two groups differed significantly (Mann-Whitney U = -2.086, n1 < n2 = 0, P < 0.05 two-tailed). Similarly, we found that the distributions in the two groups "projects with six to forty contributors" and "projects with more than forty contributors" were diverse (Mann-Whitney U = -2.204, n1 < n2 = 0, P < 0.05 two-tailed) with mean different smell occurrences of 1.655 and 2.750, respectively.

App Updates

We investigated the smell occurrences in subsequent app releases. Of the 732 projects, 33 (4%) of them released updates that either resolved or introduced issues. By inspection of source code we noticed that many of the updates targeted new functionality, *e.g.*, addition of new implicit intents to share data with other apps, implementation of new notification mechanisms for receiving events from other apps using implicit pending intents, or registration of new custom schemes to provide further integration of app related web content into the Android system. We believe this is due to developers focusing on new features instead of security.

For the majority of the app updates that introduced new security smells, we found that the dominant cause for decreased security is the accommodation of social interactions and data sharing features in the apps updates. Hence, developers should be particularly cautious when integrating new functionality into an app.

Evolution

Every new Android version introduces changes that strengthen security. The targeting of outdated Android releases will not only limit the sup-

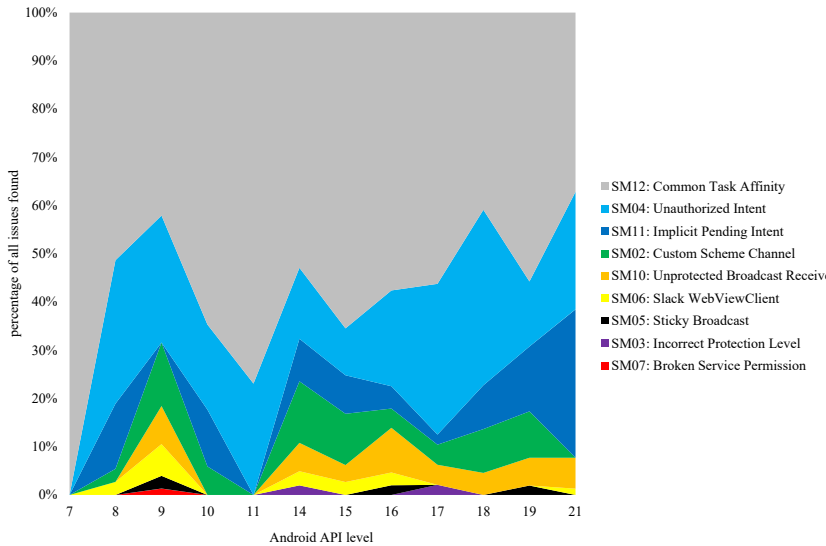


Figure 5.4: Evolution of security code smells in different Android releases

ported feature set to the respective release, but also introduce potential security issues as security fixes are continuously integrated into the OS with each update.

Figure 5.4 shows the evolution of security smells across different Android releases. For those apps that had more than one release in our dataset, we only considered the latest release. The horizontal axis shows the different Android releases apps are targeting in their configuration, whereas the vertical axis shows the contribution of a specific smell to the total amount of smells detected. As in chapter 4, we see changes in some of the security smells apps suffer from. We believe that the positive trend in *Unauthorized Intent* within apps is the consequence of built-in sharing functionalities to external services. The relative growth of *Implicit Pending Intent* could correlate to the introduction of a new storage access framework in Android release 19, which heavily relies on intents, and allows developers to browse and open documents, images, and other files with ease. Google's efforts to raise the developer's awareness of web-related security issues appears to be working: the occurrences of *Slack WebView Client* have decreased in more recent releases. Despite the lack of comprehensive data on API levels 10 and 11 due to the relatively few apps available for study, the occurrences of the majority of smells remain constant as a result of the early feature availability since API level 1.

Comparison to Existing Android Lint Checks

In order to compare our findings with other issues in the apps, we correlated the results from the existing Android Lint framework with security code smells. We wanted to explore whether frequent reports of specific Android Lint issue categories were also indicative of security issues, or in other words, if security checks by the Android Lint framework agree with our security smells and whether other quality aspects of an app could relate to its security level. We collected all available issue reports for each app and then extracted the occurrences of each detected issue.

We applied the Pearson product-moment correlation coefficient algorithm for each ICC security smell category combination according to the following formula:

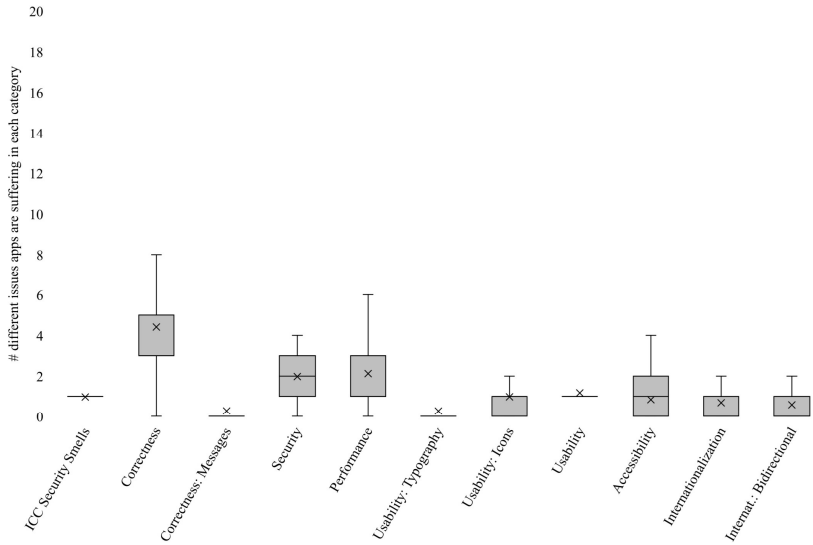
$$Pearson(x, y) = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \sum(y - \bar{y})^2}}, \text{where} \tag{5.1}$$

x is the array of all apps issue occurrences in category *ICC security code smells*,
y is the array of all apps issue occurrences in the respective Android Lint category, and
$\bar{x}, \bar{y}$ represent the corresponding sample means.

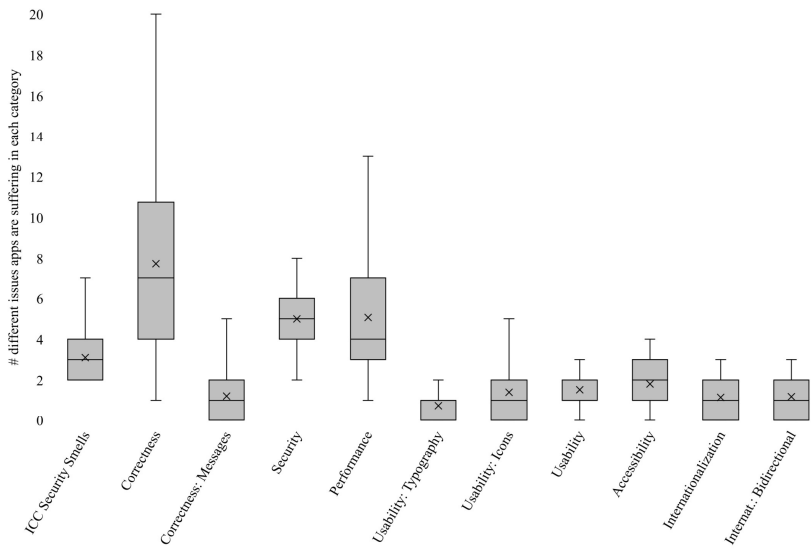
(5.2)

This formula provides a linear correlation between two vectors represented as a value in the range of -1, *i.e.*, a total negative linear correlation and $+1$, *i.e.*, a total positive linear correlation. The correlation of the Android Lint categories and our ICC smell category in Table 5.3 reveals several interesting findings: (1) Our ICC security category strongly correlates with the Android Lint security category (+0.72), which contains checks for a variety of security-related issues such as the use of user names and passwords in strings, improper cryptography parameters, and bypassed certificate checks in WebView components. (2) Another discovery is the minor correlation between the ICC security smells and the Android Lint correctness category (+0.29). This category includes checks for erroneously configured project build parameters, incomplete view layout definitions, and usages of deprecated resources. (3) Furthermore, we assume that usability does not impede security (+0.07), because issues in usability are closely related to UI mechanics. (4) Finally, minor correlations are shown for performance, accessibility, and internationalization. These three categories have in common that they rely heavily on UI controls and configurations.

To further assess how our tool performs on real world apps against the Android Lint detections, we take the 100 apps with the most and least prevalent ICC security smells and compare them to Android Lint's analysis results. We expect to see significant similarities in the increase of issues detected as our security smells correlate to Android Lint's security checks, *i.e.*, the least vulnerable apps should suffer less in both, the Android Lint checks and our security smell detectors. Figure 5.5 illustrates



(a) 100 least vulnerable apps



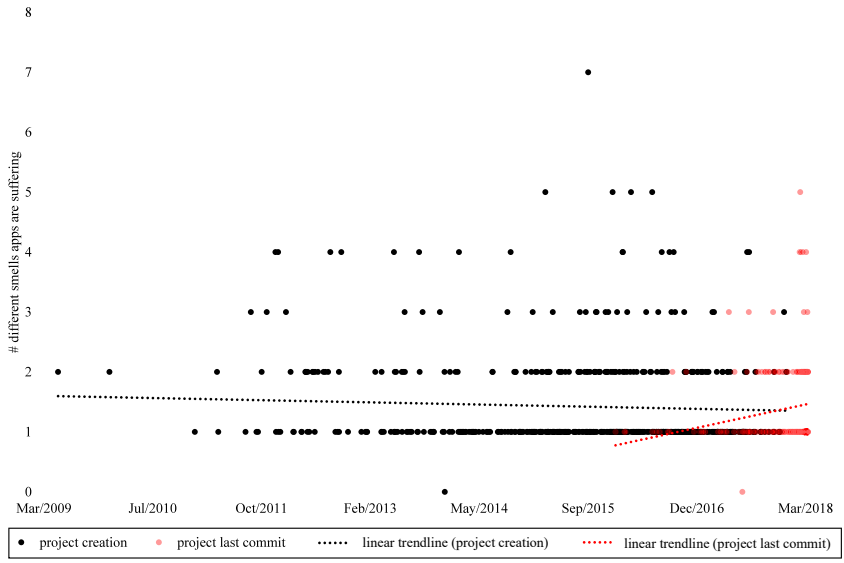
(b) 100 most vulnerable apps

Figure 5.5: Prevalence of Android Lint issues in the 100 most and least vulnerable apps

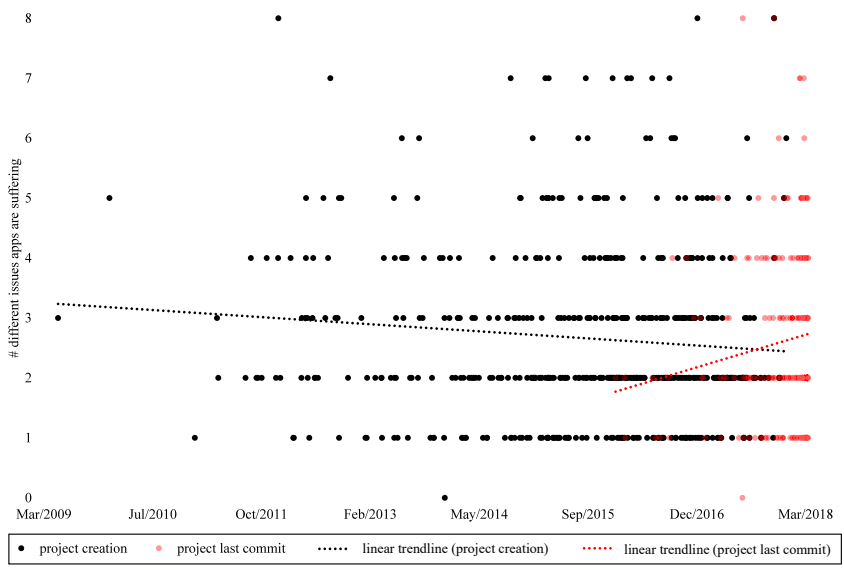
Android Lint category	Correlation with ICC security smells
Security	0.72
Correctness	0.29
Correctness: Messages	0.27
Accessibility	0.25
Performance	0.25
Usability: Typography	0.21
Internationalization	0.13
Internationalization: Bidirectional	0.11
Usability: Icons	0.11
Usability	0.07

Table 5.3: Correlation of ICC security smells with Android Lint issue categories

two plots, each presenting our analysis results for the 100 apps suffering the most and the least from ICC security smells, respectively. The vertical axis represents the condensed mean number of found issues, that is, we conflated all detected ICC security smell issues, regardless of their smell categories, into "ICC Security Smells." The remaining Android Lint categories on the x-axis are treated accordingly. The crosses represent the mean value of the number of different issues apps are suffering from in each category, and, as we hid any outliers to increase readability, these values can exceed the first quartiles. The least and most affected apps clearly correspond in terms of issue frequency among specific categories, that is, the mean number of issues found in *each* category is between 29% and 332% higher on behalf of the 100 most vulnerable apps. Besides the ICC Security Smells category with an increase of 219% in issues found, the Android Lint security category experienced an increase of 152%. The *Correctness: Message* and the *Usability: Typography* categories of Android Lint achieved, unexpectedly, an increase in issues found of about 332% and 174%, respectively. After manual verification, we discovered that these gains were mostly caused by flawed language dictionary entries used for internationalization, such as missing or misunderstood language dependent string declarations, spelling mistakes, and the use of strings containing three dots instead of the ellipsis character. While the 100 most vulnerable apps appear to prominently incorporate translations for several different languages, the 100 least vulnerable apps rarely make use of these features, hence, they suffer from much fewer issues. The remaining categories encountered an increase of less than 139%. Interestingly, the internationalization category does not encounter a noticeable increase in issues due to its limited scope, *i.e.*, it only covers five specific flaws regarding insufficient language adaption, and the use of uncommon characters or encodings. We propose that some of these issue detections should be reallocated to other categories, *e.g.*, spelling mistakes should be assigned to internationalization, and vice versa the issue *SetTextI18n* in the category internationalization that reports any use of methods that potentially fail with number conversions.



(a) Relation of dates to our ICC security smells



(b) Relation of dates to Lint security

Figure 5.6: GitHub project creation and last commit date in relation to each project's issue count

Influence of Project Age and Activity

To explore the effect of recent updates, which we believed would improve app security, we evaluated our ICC category as well as the Android Lint security and correctness categories according to time since the last commit. More precisely, we were interested in the question: Do recent updates improve app security? A related question arises from the age of a project, *i.e.*, are mature projects more secure than recent creations? We investigated these two questions based on available GitHub metadata, and brought the dates into perspective with the reported issues.

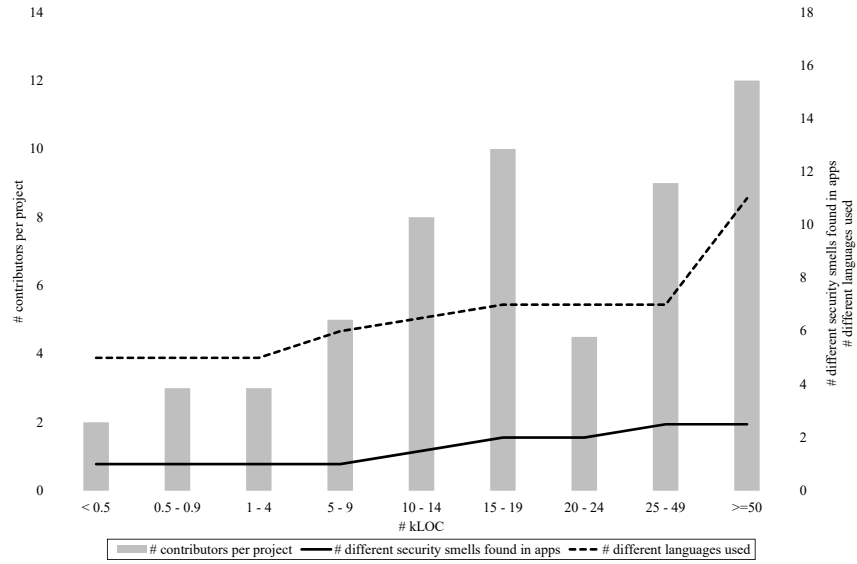
Figure 5.6 shows the mean number of detected issues per app on the vertical axis, either for the ICC security smells, or the Android Lint security category. The black dots reveal the app's project creation dates, whereas red dots indicate the most recent commit dates of projects, hence every app is represented by one black and one red dot in each plot. The creation date for the majority of apps dates back to less than 6.5 years. We can clearly see in every plot a correlation between both the creation date and the date of the last commit to the overall issue count, based on the pictured linear trends using dotted lines. These trends, which are very similar in terms of elevation, are a further indicator for the close relationship between our tool and the Android Lint checks. Moreover, the Lint security category shows strong evidence that mature projects have more security issues than recent ones. We assume that this is caused by the less comprehensive checks that older IDEs performed on the source code. Similarly, apps that frequently introduce changes, *i.e.*, receive updates, are prone to have more issues.

Influence of Code Size

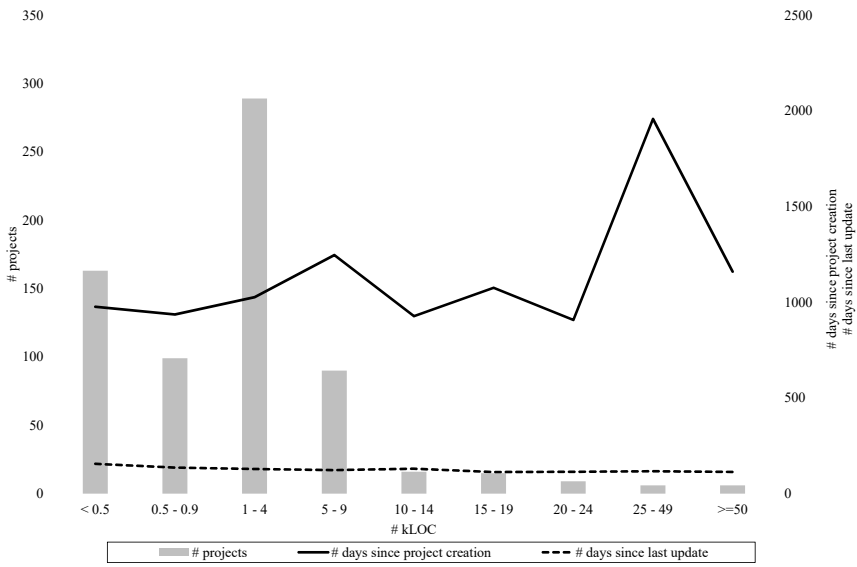
Another popular indicator used in software analysis is the code size, which we measured in thousands of lines of code (kLOC) with the open-source tool cloc.[12] As Android projects consist aside from source code of different configuration, resource and other utility files, we first ran the analysis of adopted software languages, *e.g.*, Java, Kotlin, XML that required each of those items, before we excluded all elements except the Java code in the main Java source folders for the kLOC measurements. We conjectured that we would see a trend of small teams developing small apps that are less likely to have problems. In contrast, we expect that aging projects are more likely to have smells as they are larger than more recent ones. Figure 5.7 illustrates the relation between the kLOC and other relevant properties.

In Figure 5.7a we categorized projects according to their size on the x-axis, while the left y-axis displays contributors per project, and the

[12] GitHub project website: cloc, `https://github.com/AlDanial/cloc`, *accessed on 02-MAR-2022*



(a) Relation of kLOC to contributors, ICC security smells, and used languages



(b) Relation of kLOC to number of projects, days since project creation, and days since last update

Figure 5.7: Different project properties in relation to kLOC

right y-axis the number of different categories of security smells found in apps, and the number of different languages used. We see a trend that larger projects rely on more contributors with a minor exception at 20-24 kLOC. Furthermore, it is interesting to see that projects of up to 10 kLOC are maintained by five or fewer developers. In addition, we see that larger projects tend to suffer from more smells, and those projects are also using more languages. After a manual inspection of apps exploiting different languages we discovered that those apps are rather collections of frameworks, *e.g.*, for network penetration tests using a plethora of different tools written in different languages.

Figure 5.7b uses the same feature for the x-axis, but presents on the left y-axis the number of projects, and on the right y-axis the number of days since project creation, and the number of days since last update. The majority of projects in our dataset consist of less than 10 kLOC, and especially projects with 1-4 kLOC have been very prevalent, followed by apps that are less than 500 LOC. Only six projects contained more than 50 kLOC. Interestingly, we cannot derive clearly any major trend regarding the age of projects and LOC, although projects of 25-49 kLOC evidently are older than the others. On the contrary, we can see a minor trend regarding the time since last update. It appears that smaller apps are updated less frequently than larger apps. We expect that the larger an application becomes, the more maintenance work is required due to library updates, obsolete external references, and content changes.

5.2.4 Manual Analysis

To assess the performance of our tool and show how reliable these findings are to detect security vulnerabilities, we manually analyzed 100 apps. We invited two participants to independently evaluate the precision and recall of our tool. Participant A is a senior developer with more than 5 years of professional experience in development and security of mobile apps. Participant B is a junior developer with less than two years of experience in Java and C# software development. We provided both participants an introduction to Android security, and individually explained every smell in detail. We subsequently selected the top 100 apps, that is more than 13% of the whole corpus, with most smells in accordance with our ICC security smell list, for which we can say with 95% confidence that the population's mean smell occurrences of the top 100 apps are between 3.04 and 3.48, while they are between 1.38 and 1.50 for the whole data set of about 732 apps. Then we provided the participants with our tool, the sources of the top 100 apps, and a spreadsheet to record their observations. Each participant was asked to import the sources of each app in Android Studio, which had been prepared to run a customized version of our analysis plug-in, to verify each reported smell according to the symptoms of any smell described in section 5.1. We were also interested in vulnerability

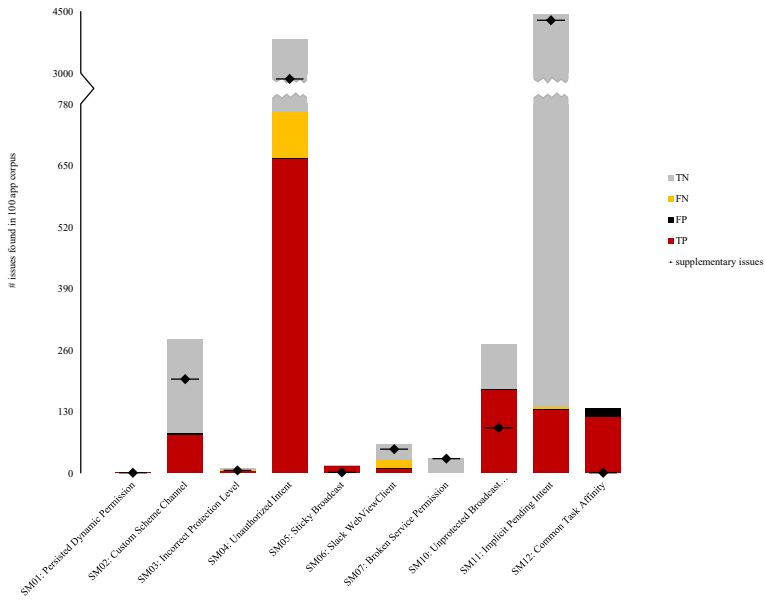


Figure 5.8: Tool evaluation results

detection capability of security smells, *i.e.*, the possibility a security issue can compromise a user's security and privacy, thus the participants were asked to investigate if a security smell indicates the presence of a security vulnerability based on the vulnerability information available in the benchmarks.

Tool Evaluation

While the assessment of true positives (TPs), *i.e.*, reported code that is a smell, and false positives (FPs), *i.e.*, reported code that is not a smell, requires participants to manually check only the tool's results, the extraction of true negatives (TNs), *i.e.*, unreported code that is not a smell, and false negatives (FNs), *i.e.*, unreported code that is a smell, is resource intensive and error prone. Therefore to avoid an exhaustive code inspection, we developed a relaxed analysis that shows ICC-related APIs in the code to support the participants.

We obtained relatively high smell detection rates, especially for SM02, SM04, SM10, SM11 and SM12, as indicated by the TPs in Figure 5.8. The reason is that these smells occur frequently and are straightforward to detect, mostly relying on some very specific method calls and permissions.

We encountered above average FPs in SM12 due to the intended use of task affinity features in apps that try to separate activities with empty task affinities. This smell would require additional semantical, architectural,

and UI information for proper assessment. While some of the exposed activities are non-interactive, and thus supposedly secure, some of them are interactive and could be misused in combination with other spoofing techniques, like clickjacking, in which an adversary unexpectedly shows the exposed activity to trick users into providing unintended inputs. In particular, call recorders and various client-server apps for chat, video streaming, home automation, and other network services have been affected by this issue.

Each participant had to check 7 241 locations in the code to examine the TNs and FNs in 100 apps. In more than 98.36% of cases participants confirmed that there are no security smells beyond what the tool could identify; we consider this very low proportion of FNs, *i.e.*, 1.7%, encouraging.

We are surprised to see only a few FNs in SM04 as we expected much more to appear due to the countless ways that intents can be created in Android. A substantial number of FNs were missed because of complex chained executions and calls initiated from sophisticated UI related classes containing URIs. For SM06, we discovered that the FNs have been frequently caused by lack of context, *e.g.*, unawareness of data sensitivity, or custom logic that does not mitigate the vulnerability. For example, our tool was unable to verify the correctness of custom web page white-listing implementations for `WebView` browser components, which would actually reduce security if implemented incorrectly.

We did not encounter any instances of the two smells SM08 and SM09, that is, we retrieved zero reports on both of them for our 100 app dataset, hence, we excluded them for all subsequent plots and discussions in this subsection.

We could find common security smells while reviewing the feedback from the two participants, for example, that some apps were using `shouldOverrideUrlLoading` without URL white-listing to send implicit intents to open the device's default browser, rather than using their own web view for white-listed pages, thus fostering the risk of data leaks. Another discovery was the use of regular broadcasts for intra-app communication. For these scenarios, developers should solely rely on the `LocalBroadcastManager` to prevent accidental data leaks. The same applies for intents that are explicitly used for communication within the app, but do not include an explicit target, which would similarly mitigate the risk of data leaks. Moreover, unused code represents a severe threat. Several apps requested specific permissions without using them, increasing the impact of potential privilege escalation attacks.

Tool Performance

Figure 5.9 presents the tool's performance based on the precision, recall, and lastly the F-measure for existing smells in 100 apps. All smells except

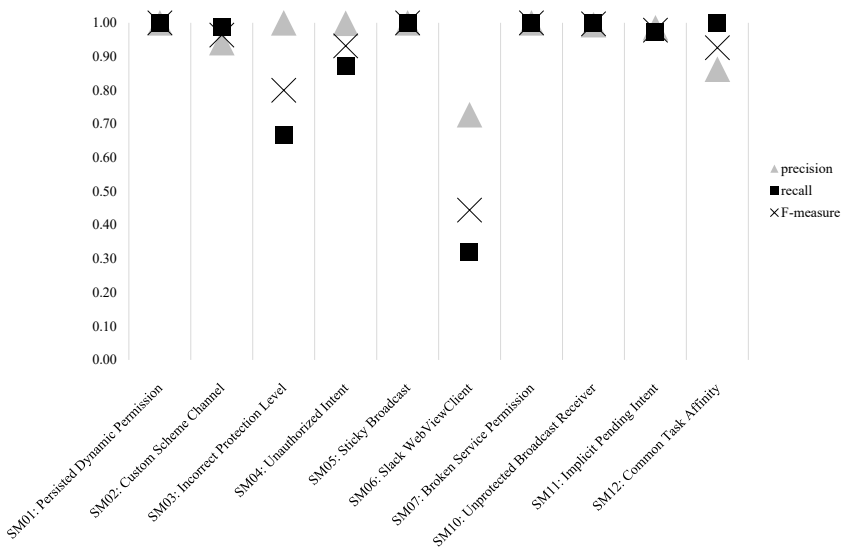


Figure 5.9: Tool performance

Figure 5.10: Vulnerability capability of detected issues

SM03 and SM06 show outstanding results, nonetheless, some of them could be biased as a result of their low occurrences, which is true for SM01 and SM07. We performed a follow up manual investigation of SM03 and SM06. Apparently, the detection of SM03 suffers from the difficulty to discern data sensitivity and the need to approximate the required protection level. Besides that, SM06 is heavily affected by custom web API implementations that (mis)use security features, which are, in fact, not secure.

Smells and Their Vulnerability Capability

Figure 5.10 shows the vulnerability capability perception of both participants against the reported smells. For each smell we show two grouped columns: the left column reports the results from the more experienced participant A (PA), and the right column reports the results from the less experienced participant B (PB). Each column consists of three different segments *yes*, *uncertain*, and *no*. The category *yes* is used for all reported smells that introduce critical risks, such as plain exposure of user passwords through network sockets. The *uncertain* category is used for risks that potentially exist, and are challenging to inspect manually, for instance, vulnerabilities that require prerequisites for successful exploitation such as potentially dangerous user-defined schemes. Finally, all smells assigned to the *no* category are not vulnerable to any attacks, either because they do not contain any user information, or because they are sufficiently secure with respect to the participant's opinion. Apps that send static non-sensitive information commonly match this category. For all our considerations the participants were told to treat any user data as sensitive, since they could potentially contain sensitive information at run time.

According to the reports by PA, 38.5% of smells represent potential threats, *i.e.*, *uncertain* category, and only 5.3% of smells represent critical threats, *i.e.*, *yes* category. In other words, only about 44% of security smells could lead to security vulnerabilities.

A further comparison of the reports between the two participants shows that they expect somewhat similar risks for the smell categories SM05, SM07, and SM12, whereas the participants tended to interpret diversely the threat caused by *Custom Scheme Channel*, *Unauthorized Intent*, and *Slack WebViewClient* smells. We reviewed the feedback of the participants and discovered that for *Custom Scheme Channel* predefined system schemes are considered less harmful for PA (category *no*), while PB assigns them to the category *uncertain*. For *Unauthorized Intent* PB assessed the risks similar to PA, however, PB encountered difficulties to predict adequately the threat capability of many intent instances, thus PB assigned them to section *uncertain*. For the smell *Slack WebViewClient* PB performed a conservative risk assessment by not assigning any custom security feature implementations to *no*, instead PB assigned them to *uncertain*, unlike PA who concluded many of them as secure. An example thereof is an

app with a network security penetration test suite that requires opening insecure web pages for security validation purposes.

It is interesting to observe that PB, in contrast to PA, does consider fewer instances as harmful for SM11 and SM12. For the first smell *SM11: Implicit Pending Intent* PB considers intents with assigned actions frequently as secure, while PA considers them as potential risk, which is more accurate. For the second smell *SM12: Common Task Affinity* PB considers most apps that used empty task affinity properties as secure, while PA performed a more thorough analysis of the UI and considered additionally the misuse capability of such exposed views, which resulted in many assignments to the category *uncertain*. We conclude that the very complex and flexible ICC implementation provided by Android overwhelms inexperienced developers, even worse, it could mislead those developers to create insecure code due to their misunderstanding.

Overall, most of the vulnerabilities seem to emerge from SM10 and SM11, which collectively contribute to more than 72% of all detected critical issues. On the other hand, SM04 on its own provides with 77% the largest proportion of false alarms regarding vulnerability capabilities.

5.2.5 Threats to Validity

A major threat to validity is the completeness of this investigation, *i.e.*, whether we explored every related paper from the scientific community.

We focused on top software engineering and security conferences, journals, and included other popular work in the field. For each relevant paper in the resulting corpus, we recursively iterated over both citing and cited papers. Additionally, for each identified issue we further constructed more specific queries and looked for any new paper on Google Scholar to prevent missing a major publication.

We solely focused on benign apps as we expect that for malicious apps developers usually do not dedicate much of their time to resolve security issues. Our dataset therefore contains apps from GitHub and the F-Droid app catalogue. Nevertheless, this dataset could still contain malicious apps that have not been reported by the community or the app stores.

Our investigations regarding the prevalence of security smells is limited to the source code of apps, whereas security smells could also manifest in third-party libraries.

Our analysis is intra-procedural and suffers from inherent limitations of static analysis. Moreover, many security smells actually constitute security risks only if they deal with sensitive data, but our analysis cannot determine such sensitivity.

The Android Lint tool we used for the analysis is prone to errors that could lead to FNs, for example, they can be introduced by an immediate termination of the inspection that occurs when Android Lint crashes due to file parsing issues.

Ultimately, there is a threat to construct validity through potential bias in experimenter expectancy since some tool results were manually reviewed by the people who created the tool. We mitigated this threat by including an external participant in the process in addition to the co-author who simultaneously played the senior developer's role.

5.3 Conclusion

We have reviewed ICC security code smells that threaten Android apps, and implemented a linting plug-in for Android Studio that spots such smells, by linting affected code parts, and providing just-in-time feedback about the presence of security code smells.

We applied our analysis to a corpus of more than 700 open-source apps. We observed that only small teams are capable of consistently building software resistant to most security code smells, and fewer than 10% of apps suffer from more than two ICC security smells. We discovered that updates rarely have any impact on ICC security, however, in case they do, they often correspond to new app features. Thus developers have to be very careful about integration of new functionality into their apps. Moreover, we found that long-lived projects suffer from more issues than recently created ones, except for apps that are updated frequently, for which that effect is reversed. We advise developers of long-lived projects to continuously update their IDEs, as old IDEs have only limited support for security issue reports, and therefore countless security issues could be missed.

A manual investigation of 100 apps shows that our tool successfully finds many different ICC security code smells, and about 43.8% of them in fact represent vulnerabilities. Thus it constitutes a reasonable measure to improve the overall development efficiency and software quality.

We recommend security aspects such as secure default values and permission systems to be considered in the initial design of a new API, since this would effectively mitigate many issues like the very prevalent *Common Task Affinity* smell.

Chapter 6

Security Smells in the Web Communication of Mobile Apps

Declaration of Content Reuse

The content of this chapter is based on the full paper *Web APIs in Android through the Lens of Security* that has been accepted for the *27th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* in 2020 [35].

Mobile apps increasingly rely on web communication to provide their services. Apps access the internet through web APIs in order to use an increasing number of public web services, or to communicate with private back-ends. Researchers have recently studied the use of such APIs in mobile apps, and, for instance, found that a large number of web requests are not directly traceable to source code [78], cloud and mail service credentials are hard-coded in the apps [119], many web requests are harmful [120], many web links targeting well-known advertisement networks impose serious risks on users [79], and lax input validation in many web APIs could compromise the security and privacy of millions of users [61].

We could not, however, find any publicly available tool that researchers can use to study web APIs. Also, there are several third-party libraries to implement network communication, but existing studies are mainly limited to `java.net` APIs. Finally, dissecting the distribution of elements that comprise the web API URLs is never studied, which is necessary for collecting security-related information stored in query keys and values, as well as to fuzz web APIs.

We manually studied the use of common web communication frameworks in 160 randomly selected Android mobile apps, *i.e.*, more than 4.7% of the whole dataset, and developed a static analysis tool to investigate

whether network communications in 3 376 closed-source and open-source apps differ. We manually inspected the tool's output for 100 random apps, and used the reported URLs to connect to the servers and to investigate their response. We found eight security code smells, *i.e.*, *symptoms in the code that signal the prospect of a security vulnerability*, on both ends, dominated by the use of embedded computer languages. We handcrafted regular expressions to automatically identify the use of those languages, and other languages prevalent on GitHub.

In this work we address the following research questions:

RQ$_1$: *Which API frameworks are used in Android mobile apps, and what is the nature of the data that apps transmit through these frameworks?* We identified six different web API communication libraries, and learned that open-source apps rely on simpler request paths including only one or two path segments, whereas closed-source apps mostly include two or three path segments. Unexpectedly, the opposite is true for key-value pairs: open-source apps frequently use one to three pairs, whereas closed-source apps mainly use one pair. Fragments have only been used very sparsely in both types of apps. We found that open-source and closed-source apps are similar in the choice of web communication libraries, but advertising services are more prevalent in closed-source apps.

RQ$_2$: *What security smells are present in web communication?* We found eight security smells in the apps and the server software. For instance, 500 apps use embedded computer languages, *e.g.*, SQL, and JavaScript commands in web API communications, thus introducing the threat of code injection attacks. A horrific 67% of the closed-source and 9.5% of the open-source apps communicate with servers over insecure HTTP connections. Many apps neglect to use the HTTP strict transport security policy. Finally, we observed a lack of authentication and authorization mechanisms for services that are supposed to be private.

In summary, this chapter attempts to shed more light on the use of web APIs in mobile apps, by studying what data the apps transmit, to whom, and for what purpose. The tool and the obtained results in this study are available online.[1]

The remainder of this chapter is organized as follows. We describe the methodology of our web API mining approach in section 6.1, and we present the results of our empirical study in section 6.2. We report numerous web API security smells in section 6.3. Finally, we recap the threats to validity in section 6.4, and we conclude this chapter in section 6.5.

[1] GitHub project website: jandrolyzer, `https://github.com/pgadient/jandrolyzer`, *accessed on 02-MAR-2022*

6.1 Web API Mining

We manually inspected Android apps to identify what APIs developers use to call web services, and how they are used. Then we took advantage of this information to develop a tool to automatically extract the web API URLs and their corresponding HTTP request data statically from the apps.

6.1.1 Library Inspection

An Android app can call a web API either with the help of the built-in Java classes, or by using external third-party libraries. We consulted the official Java and Android documentations to compile a list of built-in APIs that are relevant to network communication, and to establish how these APIs are used. We mainly focused on the `java.net` package, which includes a number of classes such as `Socket`, `HttpsURLConnection`, and `URLConnection` to implement network-related operations.

Next, we manually inspected 160 randomly selected apps from a dataset of 3 376 apps that request Android's `INTERNET` permission to investigate what third-party libraries they may use for web communication, and how. These libraries are often built on top of the built-in Java network APIs. Therefore, we first checked whether a call to such Java APIs exists, and, if so, we checked whether the call belongs to the app or an external library. For each library, we studied the documentation, and investigated how developers use the library in each app, *e.g.*, to construct URLs, and to attach headers to web requests. During the inspection of each app, we collected the web API URLs and any data that are transmitted to the servers to determine if what we collect from the source code is actually helpful to issue valid requests.

In this study, besides the native Java network libraries, we found that libraries such as *Apache HttpClient*, *Glide*, *Ion*, *OkHttp*, *Retrofit*, and *Volley* are used in the apps.

While studying the use of web communication libraries, we also noticed that besides the built-in *org.json* package, developers often use two external libraries, namely *Gson* and *Moshi*, for parsing and manipulating JSON data, which is commonly used for data exchange in web services.

6.1.2 API Miner

We then developed a tool that leverages our finding in the library inspection phase, and statically analyzes apps to extract web API URLs, query keys and the corresponding values where applicable. The tool takes the following steps:

Decompilation

Given an APK file, the tool first decompiles the app using the command line version of the *JADX* decompilation tool.[2] A successful decompilation will provide us with a project folder that contains decompiled Java source code of the app and the resource files. Although decompilation errors are common, JADX is quite robust and produces code with a correct syntax. In particular, method declarations and class structures remain intact with comments in place where the decompilation did not succeed completely.

Detection and Extraction

The tool uses the *JavaParser* framework to create an AST for every `.java` file within the project.[3] When the actual source code of an app is available, we use the information from the build and configuration files to accurately inject specific library versions into the *JavaParser* framework to enable the resolution of library dependencies in the subsequent app analysis. If the desired library version is unavailable in our collection, the next available more recent version is added instead. Closed-source apps packaged as APK files do not require those dependency injections as they already contain the required code themselves.

In principal, we need to track flows of data in relevant APIs, and several static analysis frameworks exist to track data flows in Android apps. Nevertheless, in our experience as well as according to recent studies, these tools may not perform as described in the relevant papers [75, 70, 19]. We therefore decided to implement our own lightweight analysis tailored to reconstruct web APIs in the code.

The tool traverses the AST to identify APIs, *i.e.*, `MethodCallExpression` nodes, that are used to access web APIs in a network library. For each method call, it recursively resolves the nodes on which the API depends, *e.g.*, the object on which the method is called, and its parameters. In detail, we rely on the *JavaSymbolSolver* framework to associate a variable in the code to its declaration.[4] We track all `Assignment`, and `MethodInvocation` constructs on each variable in each relevant `VariableDeclaration` node. Moreover, depending on the target library, the tool also tracks implicit dependencies, *e.g.*, the annotation-driven dependency injection.

URL and header construction largely depend on string concatenation, *e.g.*, the HTTP request header is a plain text record consisting of key-value pairs providing input details for the web API request. We therefore sup-

[2] GitHub project website: jadx, `https://github.com/skylot/jadx`, *accessed on 02-MAR-2022*

[3] JavaParser: project website, `https://javaparser.org`, *accessed on 02-MAR-2022*

[4] GitHub project website: javasymbolsolver, `https://github.com/javaparser/javasymbolsolver`, *accessed on 02-MAR-2022*

port the extraction of strings that are built using the `StringBuilder.append()` method, the `String.concat()` method, and the "`+`" operator.

Reconstruction

All web API URLs and JSON data structures that contain at least one unresolved value are further processed in the reconstruction stage. We set the value of variables whose types are number or boolean to `0` and `true`, respectively. For those variables, *i.e.*, JSON or query keys, whose types are `String`, and for which we did not find a concrete value during the extraction, we compute the *Jaro-Winkler* similarity distance [105] between the variable names and every variable declaration in the code. In the end, for each successful analysis, the tool reports the web API, as shown in Listing 2, and the corresponding request headers, as shown in Listing 3.

```
Path:
/Users/webproject/...
Library:
com.squareup.retrofit
Scheme:
http://
Authority:
retrofiturl.com
Base URL:
http://retrofiturl.com
Endpoints:
    Path: api/loadUsers
    Queries:
        Query key: position, query value: <String>
        Query key: order, query value: <String>
    Fragments:
    HTTP Methods:
        HTTP Method: GET
```

Listing 2: The tool's output for a successful web API extraction

```
Path:
.../User.java
Library:
com.squareup.moshi
JSON Object:
{"address":{"street":"<STRING>",
   "number": <NUMBER_INT>"},"name":"Bob"}
```

Listing 3: The tool's output for a successful JSON object extraction

Evaluation

We performed a lightweight evaluation of the tool on ten open-source and ten closed-source apps randomly selected from our dataset. In each app, we manually searched for the terms "http://" and "https://" in the source code. For each finding, we evaluated which entries were related to web

APIs, and then tried to understand what are the URLs and the other request parameters.

We manually identified 24 distinct URLs for web APIs in the apps, of which 21 were found in the Java source code. The tool reported 39, of which eighteen URLs referred to web services: seventeen were amongst the URLs identified manually, and the tool uncovered one new case that was overlooked due to complex string concatenation. The tool achieved a precision of 46% and a recall of 80%.

There are several reasons for the tool missing the remaining seven URLs, such as URLs in open-source apps being hidden in build scripts and XML resource files rather than Java code, and incomplete library injections for closed-source apps.

The tool reported 21 URLs that did not refer to a web service. In particular, 18 URLs referred to static HTML pages, and three suffered from invalid reconstruction.

6.1.3 Security Checks

We inspected the result of the tool on a random set of 100 apps in order to identify security smells in the code relevant to web API communications.

We implemented lightweight detection strategies for these smells, mainly using regular expressions. For instance, using search terms such as *username*, *password*, *etc.* we could find hard-coded passwords, tokens, and insufficiently protected authorization schemes in the results.

```
1 HTML:
2 String uiElement = "<html><body>" +
3     ↪ jsonObj.getText() + "</body></html>";
4
5 JavaScript:
6 String customScript = jsonObj.getResponse();
7
8 SQL:
9 String queryParameter = "SELECT * FROM weather";
```

Listing 4: Examples of embedded computer code in app source strings

In many apps we found code from various computer languages embedded in Java strings, such as that shown in Listing 4, thus potentially exposing the app or the server to code injection attacks. We compiled a list of commonly used computer languages based on our own findings, and the scripting languages found in the top ten used programming languages on GitHub.[5] For each language, we pragmatically developed regular expressions inspired by the relevant language specifications, with the aim to match as many occurrences as possible. With these regular expressions, shown in Table 6.1, we counted the key identifiers for each language

[5]GitHub project website: github-languages, `https://github.com/oprogramador/github-languages`, *accessed on 02-MAR-2022*

in each app report, to detect usages of embedded languages in the web communications.

Language	Regular expressions	Language	Regular expressions
Bash	sh[ ]+	SQL	alter[ ]+table
	%.sh		create[ ]+.*index
HTML	%<[ ]*html[ ]*%>		create[ ]+.*table
JavaScript	function[^%([]*%([^%)]*%)		create[ ]+.*trigger
	%<[ ]*script		create[ ]+.*view
	js[ ]*=		delete[ ]+from
PHP	%<%?		drop[ ]+index
Python	import[ ]+%(.*%)		drop[ ]+table
Ruby	require[ ]*%(.*%)		drop[ ]+trigger
			drop[ ]+view
			insert[ ]+.*into
			replace[ ]+into
			select[ ]+.*[ ]+from
			update[ ]+.+[ ]+set

Table 6.1: Regular expressions used to detect computer languages

In a subsequent step, we issued requests to each of the URLs extracted from the entire dataset, and observed unexpected responses, *e.g.*, stack traces, error messages, or status information, disclosing sensitive information regarding the API implementation, running software, or server configuration.

6.2 Study Result

We investigated the use of network communication in Android mobile apps. In particular, the focus is on the use of libraries, and the request characteristics.

We randomly collected apps that use the internet. For closed-source apps we mined the free apps on the *Google Play* store, and for the open-source apps we relied on the *F-Droid* app catalogue.[6] For each app, we removed the duplicates, *i.e.*, apps with the same package identifier, but different version numbers, and kept only the most recent version of the app. In the end, we collected 17 079 closed-source, and 432 open-source apps.

We applied our tool to these apps, and restricted each app analysis to 30 minutes processing time, with a node resolution limit of 15 iterations on a machine with two AMD Opteron 6272 16-core processors and 128 GB of ECC memory. The tool could completely analyze 293 open-source apps, and 2 410 closed-source apps. We also included the partial results of the apps whose analyses were incomplete, resulting in a total analysis result of 303 open-source, and 3 073 closed-source apps in our dataset. Only 2 587 apps (15%) were successfully decompiled, due to crashes of the tool

[6] F-Droid: a catalogue of free and open-source apps, `https://f-droid.org/`, *accessed on 02-MAR-2022*

caused by various bugs, and incomplete feature support, *e.g.*, reflection, native code, and customized app configurations.

The apps in our dataset come from 48 different Google Play store categories. Most of them belong to `EDUCATION` (317 apps) and `TOOLS` (292 apps), however, a majority (574) have a `GAMES`-related tag. Interestingly, work-related apps are common in our dataset (335 apps). The top five categories whose apps contain the largest number of distinct web API URLs are `EDUCATION` (1 555 URLs), `LIFESTYLE` (1 027 URLs), `BUSINESS` (995 URLs), `ENTERTAINMENT` (704 URLs), and `PRODUCTIVITY` (619 URLs).

The list of apps that we analyzed in this study is available online,[7] and we share the aggregated data for research purposes on request due to the contained sensitive information such as credentials, API keys, and email addresses.

We present our findings in the following, and conclude each focal point with a short discussion, which entails similarities or differences in open-source and closed-source apps.

6.2.1 Communication Libraries

We investigated the distribution of the seven communication libraries in 3 376 apps in our dataset.

Result

In *open-source apps*, we found that each app uses up to four network libraries. The `URLConnection` (37%), `HttpURLConnection` (24%), `Socket` (9.1%), and `HttpsURLConnection` (6.0%) classes included in *java.net* are the preferred choice of open-source developers, especially `URLConnection` and `HttpURLConnection` are omnipresent in projects. When considering third party network libraries, we found that *OkHttp* and *Retrofit* (each 5.6%) are used the most. It is interesting to see that libraries with specific support for image downloads are similarly used, *i.e.*, *Glide* and *Volley*. The *Ion* library is used only in three apps (1.0%).

In *closed-source apps* each app uses up to seven network libraries. We found that the classes included in *java.net* such as `URLConnection` (42%), `HttpURLConnection` (34%), `Socket` (10%), and `HttpsURLConnection` (4.3%) are the preferred choice. Interestingly, the *OkHttp* library is the most commonly used third-party library even surpassing the well-known *Glide* and *Retrofit* libraries. We found `org.apache.httpcomponents` and `com.loopj.android` are the two least used network libraries contributing only 0.9% and 0.5%, respectively.

[7] Figshare: list of the analyzed apps, `https://doi.org/10.6084/m9.figshare.14981061`, *accessed on 02-MAR-2022*

Discussion

We realized that one to three classes are usually responsible for network communication in an app. In open-source apps we found the use of up to four network libraries in each app, and in closed-source apps it was up to seven. Although each library may provide specific features, *e.g.*, JSON parsing, HTTP connection management, image caching, *etc.*, we expect the reason for the use of multiple libraries in an app is that many developers use the code snippets from other projects or online information sources.

We found fewer *java.net* libraries in open-source apps compared to closed-source apps. During decompilation, the bundled libraries are decompiled together with the app code. Therefore, what the tool reports is not only the network calls in the app code, but also the network APIs on top of which the third-party libraries are developed. However, this is not the case for the open-source apps whose dependencies are defined in Gradle, and are dynamically injected without adding the actual code to the project itself.

The libraries *Ion* and *Volley* have been used only in open-source apps, while *HttpComponents* and *LoopJ* have been used only in closed-source apps. Surprisingly, we did not find any instances of the well-known `AndroidHttpClient` and `SSLSocket` classes. Finally, the use of *Glide*, which supports exhaustive image downloading and caching features, seems much more prevalent on closed-source apps.

6.2.2 The Nature of Web Communication

Based on the analysis results for the apps in our database, we investigated the structure, dissemination and use of 13 276 web API URLs, of which 9 714 were unique.

Open-source Apps

The tool extracted 1 533 URLs from the open-source projects. We found that the majority of web APIs consist of one or two queries or path segments. We only found up to one fragment per web API. We further found that 209 web APIs exist with paths consisting of four or five segments to distinguish between resources, and the average number of segments in the web APIs is 2.36. Nevertheless, web APIs using more than five elements are rare. Web APIs contain an average of 2.3 key-value pairs in queries. The data do not follow a normal distribution.

Surprisingly, the top base URL was `https://github.com`, which we observed 29 times (1.8%). Likewise, *Google* services have been widely used, *e.g.*, `https://play.google.com` or `https://plus.google.com`, of which the tool could spot 42 instances (2.7%). Rather at the end of the ten most commonly used base URLs the tool found the *OpenWeatherMap* API

`http://openweathermap.org` (7, 0.4%) and the *Twitter* social network API `https://twitter.com` (6, 0.3%).

Furthermore, we found that the `https` URL scheme (1 012 occurrences, 66%) is much more commonly used than its insecure counterpart `http` (521 occurrences, 33%).

Closed-source Apps

The tool extracted 11 743 URLs from closed-source apps. We found that the majority of web APIs consist of one or two queries or path segments. On a second look, we observed that web APIs with two path segments are most prevalent. We further discovered that 2 116 web APIs exist with paths consisting of four to eight path segments to distinguish between resources, and the average number of segments in the web APIs is 2.44. Nevertheless, web APIs using more than four elements are rare. Additionally, we could identify that URL fragments are seldomly used in web APIs; although we found up to seven fragments in a single web API URL, we only discovered 183 web APIs in total using this feature, *i.e.*, 1.5%. Web APIs, on average, contain 2.9 key-value pairs in queries. The data do not follow a normal distribution.

Interestingly, all the most common URLs we could retrieve were pointing towards *Google* services. The top URL, `http://schemas.android.com`, was observed 1 303 times (11%). Two of the observed URLs were related to advertising distribution services, *i.e.*, `http://media.admob.com` (283, 2.4%) and `https://pagead2.googlesyndication.com` (271, 2.3%). *Google AdMob* is a popular advertising platform that provides SDKs to developers to integrate *Google* ads into their own apps to increase revenue.

We found that the `http` URL scheme (7 208 occurrences, 61%) is much more prevalent than its secure counterpart `https` (4 531 occurrences, 38%). Besides findings of the two common schemes we found few appearances of the `ws` schema that belongs to the WebSocket protocol (4 occurrences, 0.0%), which provides unprotected full-duplex web communication on top of HTTP TCP connections.

Discussion

The numbers of used path segments and query keys are indicators for the complexity of a specific request. Servers usually reject requests with incomplete or flawed parameter configurations, and thus the task of sending a successful request becomes harder the more path segments and query keys are involved.

Open-source apps relied on simpler request paths including only one or two path segments, while closed-source apps mostly included two or three path segments. Unexpectedly, the opposite is true for key-value pairs: open-source apps frequently use one to three pairs, whereas closed-source

apps mainly use one pair. Fragments have only been used very sparsely in both types of apps.

We did not expect to observe a difference between open-source and closed-source apps. Moreover, we did not expect to find many complex requests, because the idea of providing APIs is that they can be used by other developers who presumably prefer an easy to use interface. We conclude that the majority of the APIs provide a simple interface and are rather straightforward to access.

While the open-source apps contained no advertising services in the ten most used base URLs, the closed-source apps heavily used such services. We expect that the "Freemium" price model, *i.e.*, installation of apps is free but the user must later watch ads or pay a fee, is a major enabler of this setting.

The open-source community prefers the *Twitter* social network over *Facebook*.

We found one major difference in the URL schemes used in the apps. Open-source apps principally rely on secure `https` connections (66%). In contrast, closed-source apps largely use the insecure `http` protocol (60%). We see here much potential for improvement through stricter rejection of apps using insecure connections. The more efficient, but more complex WebSocket protocol seems to be without interest for the majority of developers.

6.2.3 Security Risks

We studied the kinds of data communicated through web APIs, and found that both credentials, *i.e.*, user name and password combinations and embedded code were very common in the web communications. As the former has been reported on extensively in the past, we focus here on the latter.

Open-source Apps

The tool extracted 458 JSON schemes in which `STRING` is the most used value type with 1 197 occurrences, followed by `NUMBER` with 234 occurrences.

We found that SQL (91%, ten affected apps) is by far the most used embedded language. HTML (5.5%, two affected apps) and JavaScript (2.7%, one affected app) are very rare. No instances of other embedded languages were detected.

Closed-source Apps

The tool extracted 14 606 JSON schemes where `STRING` is the most used value type with 40 017 occurrences, followed by `BOOLEAN` with 5 640 occur-

rences. `NUMBER` and `NULL` only represent a minority with 2 389 and 1 483 occurrences, respectively.

In contrast to open-source apps, we observed that JavaScript (76%, 170 affected apps) is very prevalent, and SQL (23%, 476 affected apps) is used less, but still frequently. HTML code is almost non-existent (0.7%, 27 affected apps).

Discussion

We found that the use of tokens in open-source apps is not as common as in closed-source apps. One explanation could be that the fees associated to web services do not pay off for open-source apps, which mostly do not generate any revenue.

Several embedded languages are actively used within mobile apps. While SQL is relatively common in both open-source and closed-source apps, JavaScript is much more commonly used in the latter.

6.3 Web Communication Security Smells

In this section, we present the security smells that we found in web communication during investigation of the tool's results, by manually investigating 100 apps, and by analyzing the responses from requests to each of the 9 714 web API URLs extracted from apps in our dataset. We classify the smells into client side, *i.e.*, within mobile apps, and server side, *i.e.*, on the API servers. For each smell we report the security *issue* at stake, the potential *consequences* for users, the *symptom* in the code, *i.e.*, the code smell, and the recommended *mitigation* strategy of the issue, principally for developers.

We used the results from the manual analysis explicitly to identify security issues, but not to perform any quantitative evaluation. In this section, we do not report any number of occurrences found in the tool's results, because those either have been discussed in the previous section, or the task would require additional research to gather quantitative results.

In our analysis, we could identify eight web communication security smells, of which three were in apps and five in server implementations. Two of the three web communication app security smells could be mitigated, if only secure HTTPS channels would be used for communication. We have not yet reported our findings to developers or marketplaces.

6.3.1 Client side

We identified three client side web communication security code smells.

- **Credential leak**
  We found hard-coded API keys, login information, and other sensitive data, *e.g.*, email addresses, in the source code. Several of the

retrieved data were valid at the time of our investigation: we could access *Google Maps*, *Mapquest*, *OpenWeatherMap*, the *San Francisco transit* API, and a *Telegram* bot.
Issue: Credentials issued to app vendors are prevalent in apps that use web APIs, and they are statically stored in the Java software to perform the queries. However, the software can be decompiled into source code, which renders the data extraction trivial [119].
Consequently, web services can be misused by people who have gained access to unique credentials. Such services allow impersonation, phishing, information leaks, fake messages, or financial infringements for the app developers due to API overuse or lockdowns.
Symptom: Query keys like `key`, `token`, `user`, `username`, `password`, `pw` are used in web requests and the corresponding values are statically stored in the apps.
Mitigation: Developers should avoid using access tokens and logins of corporate accounts for apps. Instead, a unique child token based on the corporate token should be assigned to every user. If this option is unavailable, web relay APIs can be provided to the apps, which forward the requests to the final destination without disclosing any credentials.

- **Embedded languages**
  We found apps that assemble CSS, HTML, or JavaScript code programmatically using external input. In many apps, such constructed code is executed within a `WebView` or Android's UI framework, which is inspired by *Java Swing* and supports HTML elements. Similarly, we found assembled SQL statements that are executed in the local SQLite database engine. In two apps we found assembled shell commands sent over an SSH connection.
  Issue: An attacker could gain control over the app's visual representation, the behavior, the data storage, or the corresponding server by exploiting such code [89].
  Consequently, for HTML and CSS, an attacker could change the appearance of existing web elements to make space for additional ones, *e.g.*, by reducing the font size of existing text to make it impossible to read and at the same time injecting additional text in regular size. Such changes can trick users into taking unintended actions. With JavaScript, an attacker could gain access to the DOM (document object model) of the app's web page and extract or alter the visible content. Such changes expose sensitive user data, or mislead users through altered information. SQL allows adversaries to perform arbitrary actions on the database, *e.g.*, altering and deleting existing data, or inserting new data. This leads to data loss, corruption, or leaks for the users. Through shell commands an adversary could potentially gain elaborated remote access to the server's operating sys-

tem. For example, the shell command `String command = "touch /home/" + username + "/.toolConfig/configuration";` allows an adversary to execute commands on a server in the context of the service by letting the variable `username` be `;echo 'executes on server';`. Threats range from DoS attacks to sensitive user information leaks and corporate network infiltration by disabling security measures and installing malicious software on the server.
Symptom: At least one statement is manually assembled with the help of external data, *e.g.*, `"<html><body>" + example + "</html></body>"` or `"color:" + color + ";"`. HTML/CSS: common tags or properties appear, *e.g.*, `"<html>"`, `"<body>"`, or `"color:"`. JavaScript: identifiers exist in the app, *e.g.*, `function()`, `<script`, `js=`. SQL: the corresponding keywords exist in statements that obey the SQL syntax. Such keywords are, *e.g.*, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REPLACE`, `TRUNCATE`. Shell: commands are not trivial to detect, because developers use a variety of different commands, *e.g.*, `sudo`, `rm`, `cp`, `mv`, `ls`, `exec`, `attrib`, `chmod`, `touch`, *etc.*
Mitigation: Developers should not use external input when assembling embedded languages, but try to embed the content into the app installation or update package. Static code should be used whenever possible. If dynamic code is required, the built-in sanitizing classes must be used, *e.g.*, `PreparedStatement` for SQL code. User input should *never* be trusted. In general, *any* untrustworthy input must not be used before it is properly escaped and sanitized.

- **Insecure transport channel**
  Web API communication relies on HTTP or HTTPS; both variants exist in apps.
  Issue: HTTP does not provide any security; neither the address, nor the header information or the payload are encrypted [69].
  Consequently, any attacker with access to the transmitted data can read or alter all plain text messages. User data leaks, corruptions, losses, or impersonation are probable.
  Symptom: HTTP URLs are used to establish connections to web APIs.
  Mitigation: HTTPS instead of HTTP URLs must be used for any web communication.

6.3.2 Server side

For every collected API in our dataset, we accessed the corresponding web server and stored the response. We were particularly interested in information such as operating system identifiers, used software modules, and version numbers, which we could initially identify during the manual analysis of a sample of the server responses. We then crafted a number of

search queries to detect occurrences of such features and applied them to our dataset.

We have identified five server side web communication security code smells.

- **Disclosure of source code**
  Error messages provide valuable information regarding the implementation of a running system. We found web APIs that leak internal error states and use *status codes* in a different way than what is specified by the RFC7231.[8] Although HTTP servers should reply with the status code `200` to indicate a successful request, we noticed that some servers use this status code when an error has occurred and instead return implementation code.
  Issue: Error messages that include the relevant stack trace are transmitted as plain text in the server's message response body. Such a message reveals information like the used method names, line numbers, and file paths disclosing the internal file system structure and configuration of the server [4].
  Consequently, adversaries can obtain detailed information about the service implementation, which may lead to an exploit.
  Symptom: When an invalid request is received, a server responds with a detailed error message containing information that is not required by any user of the API.
  Mitigation: If the used framework provides an option to turn off diagnostic or debug messages: this feature should be used. Otherwise, an API gateway in between the client and the server should filter such responses and deliver regular HTTP `500` messages to the client instead.

- **Disclosure of version information**
  Besides useful connection parameters, HTTP headers provide information regarding the software architecture and configuration of a running system. Their keys are case insensitive. We spotted in the reported HTTP headers version information of web server daemons and API implementation frameworks.
  Issue: We encountered outdated software that suffers from severe security vulnerabilities [52]. For instance, we observed a server that returned `X-Powered-By: PHP/5.5.23` in the response header. This PHP version is at the time of writing more than 6 years old, and a quick search in the Common Vulnerabilities and Exposures (CVE) database showed that this framework suffers from 69 known security vulnerabilities, six of which received the most severe impact score of

[8] Request For Comments (RFC) of the HTTP 1.1, `https://tools.ietf.org/html/rfc7231`, *accessed on 02-MAR-2020*

10.[9]
Consequently, the vulnerabilities range from simple DoS attacks, access control bypassing, and cross-site scripting to arbitrary code execution on the server.
Symptom: One of the following header keys exists in the response header: `Engine`, `Server`, `X-AspNet-Version`, or `X-Powered-By`.
Mitigation: If the used software provides an option to turn off the publishing of version information: this feature should be used. Otherwise, an API gateway in between the client and the server should remove the affected keys and deliver messages with sanitized HTTP headers to the client instead.

- **Lack of access control**
  Authentication by a user name and a password provides tailored experiences to end users, *e.g.*, individual chat logs or friend lists, and at the same time enables access control to separate and protect sensitive user data.
  Issue: The access to sensitive data or actions is not restricted by a sane authentication mechanism such as a user name and password pair, instead, easy-to-forge identifiers or no identification data at all are used to secure the access [40]. We found several APIs that did not use any authentication or authorization mechanisms, although they host sensitive data, *e.g.*, for car rental services and accounting. In one app we found code to access an exposed SQL database interface.
  Consequently, every internet user can access sensitive data or perform unauthorized actions including the reading, modification, and deletion of arbitrary user data. We could access information from such APIs, *e.g.*, real-time location data of rental cars and transaction histories on different bank accounts. In one case, we were also able to create new users in the system. Exposing database or other interpreter interfaces with broken authentication allows adversaries to execute arbitrary statements on the server.
  Symptom: A web API server hosts sensitive data or provides actions, which would require elevated access rights. The server responds without asking for any login information, that is, no HTTP headers or keys related to personal information are used in the API, *e.g.*, `username`, `password`, or `pw`. The server requires query keys with names of programming languages, *e.g.*, `sql`, and responds when such variables hold a statement in that language, *e.g.*, `SELECT table_name FROM all_tables;`. The decision finding of data sensi-

[9] National Vulnerability Database (NVD): vulnerabilities of PHP 5.5.23, `https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&isCpeNameSearch=false&cpe_vendor=cpe%3A%2F%3Aphp&cpe_product=cpe%3A%2F%3Aphp%3Aphp&cpe_version=cpe%3A%2F%3Aphp%3Aphp%3A5.5.23`, *accessed on 02-MAR-2022*

tivity or elevated actions is non-trivial and involves manual reasoning [114]. Therefore, we cannot infer general purpose terms.
Mitigation: Application architects have to implement authentication, favorably multi-factor authentication, whenever sensitive data or elevated operations are involved in the process. All user data, and location data in general, have to be considered as sensitive. Developers should never expose interpreter interfaces to a web service without prior authentication and input validation. REST interfaces for specific tasks should be created, preferably each using static statements that do not rely on any user input.

- **Missing HTTPS redirects**
  In contrast to HTTPS, HTTP does not provide any security: neither the URL, nor the header information and embedded content are encrypted. We found servers that do not redirect the clients to encrypted connections although they would have been supported.
  Issue: Web API servers do not redirect incoming HTTP connections to HTTPS when legacy apps try to connect, or users manually configure a URL without adding a proper `https://` prefix [29].
  Consequently, the transmitted data remains visible and changeable to anyone within the communication path.
  Symptom: For an HTTP web API request, a server does not deliver an HTTP `3xx` redirect message, which points to the corresponding HTTPS implementation of the web API.
  Mitigation: A server should not offer legacy HTTP services. If they are still required due to legacy clients with hardcoded HTTP URLs, redirects should be provided to guide all clients to the secure version.

- **Missing HSTS**
  HTTP header information is used to properly set up the connection by specifying various communication parameters, *e.g.*, the acceptable languages, the used compression, or the enforcement of HTTPS for future connection attempts, a feature which is called HSTS (HTTP strict transport security). HSTS provides protection against HTTPS to HTTP downgrading attacks, *i.e.*, when a user once accessed a web resource in a secure environment, *e.g.*, at home or work, the client knows that the resource needs to be accessed *only* through HTTPS. If this is not possible, *e.g.*, at an airport at which an attacker tries to perform MITM attacks, the client will display a connection error. Hence, HSTS should be used in combination with HTTP to HTTPS redirects, because the HSTS header is only considered to be valid when sent over HTTPS connections. We found servers that do not enforce clients to remain on the secure channel for future requests.
  Issue: Servers do not leverage the HSTS feature [51].
  Consequently, in unprotected public networks or networks under external supervision, if an attacker sets up a fake gateway which runs

SSLsniff,[10] the provided services remain vulnerable, because transmitted data is visible and changeable.
Symptom: A server does not deliver the HTTP HSTS header `Strict-Transport-Security: max-age=31536000; includeSubDomains` for an HTTPS request.
Mitigation: In combination with HTTP to HTTPS redirects, the HSTS header should be used in all HTTPS connections.

6.4 Threats to Validity

The main threat to validity is the completeness of this study, *i.e.*, it is not guaranteed that we found all major libraries used for web communication in Android apps.

There may be a bias in the apps that we selected for this study. We included all open-source apps that were available on *F-Droid*, but they may not be representative of the whole open-source app community. We collected random closed-source apps that were freely available on the *Google Play* store, but paid apps or the apps on third-party stores may have different characteristics.

We only mined web APIs that were available in the source code; our tool suffers from the inherent limitations that come with static source code analysis. We developed a lightweight analysis, which is not path sensitive. We opted for this design because, during the manual inspection of network APIs in the apps, we noticed that these APIs are usually free of conditional statements and loops. Moreover, we had to decompile closed-source apps for analysis, which introduces further threats to the validity of our results. For instance, the app code and its library code are not easy to discern automatically, and therefore the libraries in such apps may have influenced our findings.

We did not evaluate how complete the tool results are for every app, but just a small number. There is a threat to construct validity through potential bias in our expectancy. However, we examined the tool results for 50 apps, and confirmed that 90% led to successful web communication.

6.5 Conclusion

We manually reviewed 160 Android apps to compile a list of commonly used network and data conversion libraries and to learn how they are used in these apps. Based on our findings, we developed a lightweight static analysis tool that identifies network-related APIs, and extracts communication information such as the web APIs, and the associated JSON

[10] GitHub project website: sslsniff, `https://github.com/moxie0/sslsniff`, *accessed on 02-MAR-2022*

headers. With the help of our tool we successfully analyzed the network-related information within 450 closed-source and open-source apps. We found that in both open-source and closed-source apps network communication is mainly developed using *java.net* classes. Amongst the third-party libraries we found that *OkHttp* and *Retrofit* are used the most. By far the most used value type in JSON data is `STRING`.

We realized that closed-source apps substantially rely on advertisement services, and that they tend to have more complex URL paths consisting of more path segments. Surprisingly, the secure HTTPS protocol is used in the majority of extracted web APIs from open-source applications, but the opposite is true for closed-source apps. Obviously, when embedded languages are used along with manual string concatenations, the attack surface for code-injection attacks increases. Nevertheless, we could identify numerous such cases during the manual examination of the web APIs, *i.e.*, embedded SQL and JavaScript content was rather common within web communications. Even worse, we found many more issues on the server side: unnecessary disclosure of server configurations, outdated web servers and language interpreters with known security vulnerabilities, leaks of internal error messages, and other sensitive data. Finally, we also found private APIs without any kind of authentication or authorization mechanisms.

We conclude that a lightweight static code analysis is very helpful in mining web APIs, and that the impact of embedded code in web API requests and the process of securing servers has been deeply underestimated.

Chapter 7

Security Smells in Mobile App Servers

Declaration of Content Reuse

The content of this chapter is based on the full paper *Security Smells Pervade Mobile App Servers* that has been accepted for the *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* in 2021 [37].

Globally accessible, reliable, and scalable web apps that utilize web APIs are on the rise.[1] In fact, there exist already more than 24 000 known public web APIs.[2]

Unfortunately, many web APIs have flaws [61] and even worse, they may behave unpredictable [10] as some of them are hosted externally with the help of cloud operators that continuously try to maximize their profits at the expense of the API users. What is more, if any problem occurs, the majority of web API providers are unable to offer helpful support through their official communication channels [11] and some of their servers run dated software [35]. These as well as our other findings in chapter 6 indicate that many web APIs and particularly their corresponding app servers are not always well maintained.

Therefore, we investigate in this chapter the prevalence of five app server security smells that we identified in the previous chapter, and in addition, we reconsider the client side security smell "Insecure transport channel" from the server side. Moreover, we assess the server maintenance

[1] Forbes: the decline of the native app and the rise of the web app, `https://www.forbes.com/sites/victoriacollins/2019/04/05/why-you-dont-need-to-make-an-app-a-guide-for-startups-who-want-to-make-an-app/`, *accessed on 02-MAR-2022*

[2] ProgrammableWeb: the largest API directory on the web, `https://www.programmableweb.com/apis/directory`, *accessed on 02-MAR-2022*

activity based on the dataset that contains 9 714 distinct URLs that were used in 3 376 apps. We address the following research questions:

RQ$_1$: *What is the prevalence of the server side security smells in the web communication of mobile apps?* We found 231 URLs from 44 apps that leak the source code of the web service implementation if processing errors occur. We can further confirm that most app servers communicate with apps over insecure HTTP connections [74], and fail to enforce use of the HTTP strict transport security policy. Finally, we found that on average almost every second app server suffers from version information leaks.

RQ$_2$: *What is the relationship between security smells and app server maintenance?* In particular, we are interested in configuration changes, because they provide insights into established maintenance processes of mobile app servers. Based on the collected HTTP header information from two measurements over fourteen months, we evaluated what software changes are introduced by system administrators. We observed that servers are usually set up once and never touched again, yielding severe security risks. For instance, criminals can attack outdated app servers by exploiting vulnerabilities listed in public databases or illicit websites. On the positive side, we noted that version upgrades are much more common than version downgrades, and that developers occasionally use Cloudflare to protect their infrastructure against adversaries, especially for non-JSON-based app servers.

In summary, this work reveals the prevalence of insecure app server configurations accessed by Android mobile apps, and their maintenance protocol. The list of apps that we analyzed in this study is available online,[3] and we share the aggregated data for research purposes on request due to the contained sensitive information such as credentials, API keys, and email addresses.

The remainder of this chapter is organized as follows. In section 7.1, we present our empirical study about the prevalence of app server security smells and server maintenance. In section 7.2, we recap the threats to validity, and finally in section 7.3, we conclude this chapter.

7.1 Empirical Study

In this section, we investigate the prevalence of six of the eight security smells that we identified in chapter 6, *i.e.*, *Insecure transport channel*, *Disclosure of source code*, *Disclosure of version information*, *Lack of access control*, *Missing HTTPS redirects*, and *Missing HSTS*. The two remaining security smells, *i.e.*, *Credential leak* and *Embedded languages* are not within the scope of this study, because they already received much atten-

[3] Figshare: list of the analyzed apps, `https://doi.org/10.6084/m9.figshare.14981061`, *accessed on 02-MAR-2022*

tion in existing research [119] or they require a deep understanding of the app and the context where they occur.

For this empirical study we evaluated all URLs from the dataset according to the security smell symptoms described in the previous section. We collected the data twice: the initial download of HTTP headers and bodies was performed in June 2019 whereas additional data, *i.e.*, the authorization errors and up-to-dateness, was retrieved in August 2020. The duration of 14 months is arbitrary but long enough to ensure developers have to update their software infrastructure.

7.1.1 Dataset

We build on our previous work and dataset detailed in chapter 6 in which we manually inspected Android apps to identify which APIs developers use to call web services, and how they are used. We then took advantage of this information to develop a tool to automatically extract and reconstruct string variables and the assigned values as well as the server URLs and their corresponding HTTP request headers statically from the apps. Using this information, we analyzed the reconstructed app server data and tried to establish connections to the corresponding servers from which they gathered additional information for analysis, *i.e.*, from HTTP response headers.

The apps from the dataset are randomly collected from those that use Android's internet permission. For closed-source apps we mined the free apps on the *Google Play* store, and for the open-source apps we relied on the *F-Droid* software repository.[4] For each app, we removed the duplicates, *i.e.*, apps with the same package identifier, but different version numbers, and kept only the most recent version of the app. We also included the partial results of the apps whose analysis was incomplete and could not finish in time, ultimately resulting in an analysis result for 303 open-source, and 3 073 closed-source apps in the dataset.

The apps in the dataset come from 48 different Google Play store categories. Most of them belong to `EDUCATION` (317 apps) and `TOOLS` (292 apps), however, a majority (574) have a `GAMES`-related tag. Interestingly, work-related apps are common in the dataset (335 apps). The top five categories whose apps contain the largest number of distinct URLs are `EDUCATION` (1 555 URLs), `LIFESTYLE` (1 027 URLs), `BUSINESS` (995 URLs), `ENTERTAINMENT` (704 URLs), and `PRODUCTIVITY` (619 URLs).

As shown in Figure 7.1, almost 94% of the apps received a star rating of 3.0 or higher. Surprisingly, apps with a five star rating are more prevalent than apps in any other category. The apps have an average star rating of 4.2 stars and a median rating of 4.3 stars. Figure 7.2 presents the number of app downloads and the timeliness of app updates. The y-axis

[4] F-Droid: a catalogue of free and open-source apps, `https://f-droid.org/`, *accessed on 02-MAR-2022*

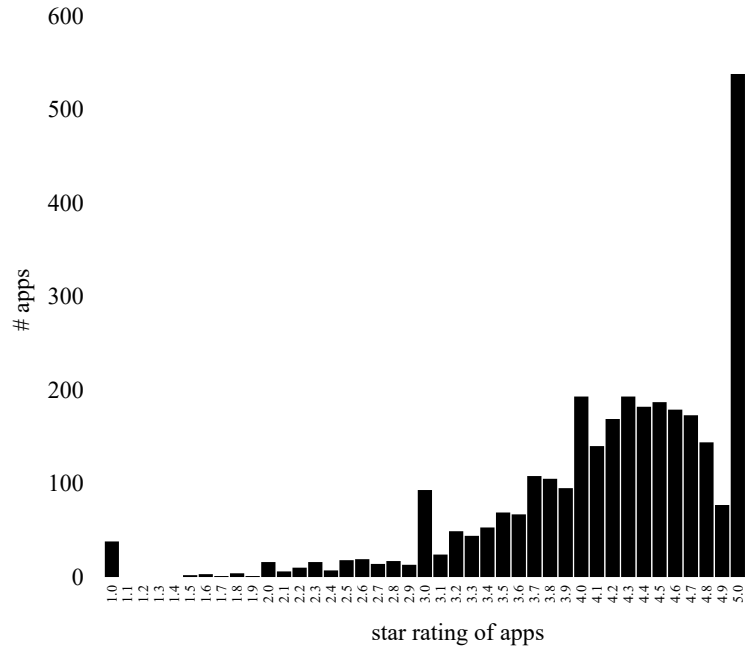


Figure 7.1: Star ratings for the Google Play apps in the dataset

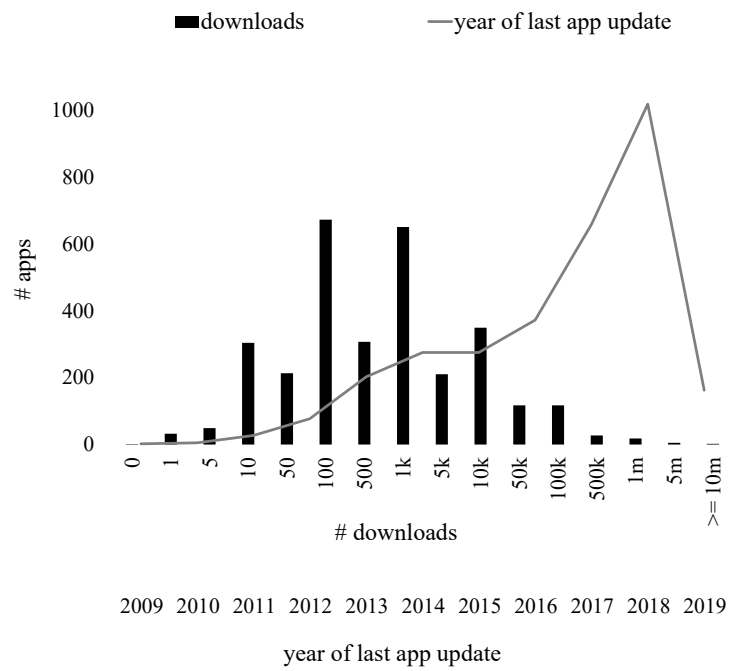


Figure 7.2: The popularity and developer support for the Google Play apps in the dataset

denotes the number of apps in each category. In contrast, the primary x-axis with the bars indicates the app downloads, and the secondary x-axis with the line indicates the time of the last app update. We can see that most apps achieved between 100 and 1 000 downloads, and barely any app was downloaded more than 1 million times. Regarding the app updates, most of the apps received an update in 2018. Therefore, we see that most vendors update their apps only a few times a year, because we collected the statistics separately in 2019.

We then exercised every URL in the dataset and collected the HTTP header and body of each server response. Eventually, we processed 1 230 open-source URLs and 8 486 closed-source URLs. We realized that many app servers do not leverage JSON, but instead they use, for example, XML or plain HTTP communication. Because we were interested whether there exist any differences for data-centric app servers, we split the results into four different groups. We report our findings based on closed-source and open-source apps, and we also separate JSON and non-JSON app servers. We decided to investigate the JSON data format, because it was much more commonly used for communication than the others, *e.g.*, comma-separated values (CSV) and XML. Therefore, we partitioned the open-source URLs into 1 171 non-JSON servers and 59 JSON servers. Accordingly, we partitioned the closed-source URLs into 7 997 non-JSON servers and 489 JSON servers. In addition, we matched these URLs against those found in the apps to see how many apps are suffering from a particular security smell.

We were particularly interested in information such as operating system identifiers, used software modules, and version numbers. Hence, we crafted a number of search queries to detect occurrences of such features. The relevant features, *i.e.*, security smells, and the results are part of the discussion in the subsequent subsections.

7.1.2 Prevalence of Security Smells

This subsection answers $\mathbf{RQ}_1$: *What is the prevalence of the server side security smells in web communication?* In Figure 7.3 and Figure 7.4 we report on the relative prevalence of app server security smells in apps for JSON and non-JSON web services, respectively. In Figure 7.3, the vertical axis indicates the percentage of apps that suffer from a specific app server security smell. In the following, we discuss the findings from different perspectives, *i.e.*, security smell categories, software development model, and technology.

By Security Smell Category

We report findings and provide actionable advice to mitigate the issue for each security smell.

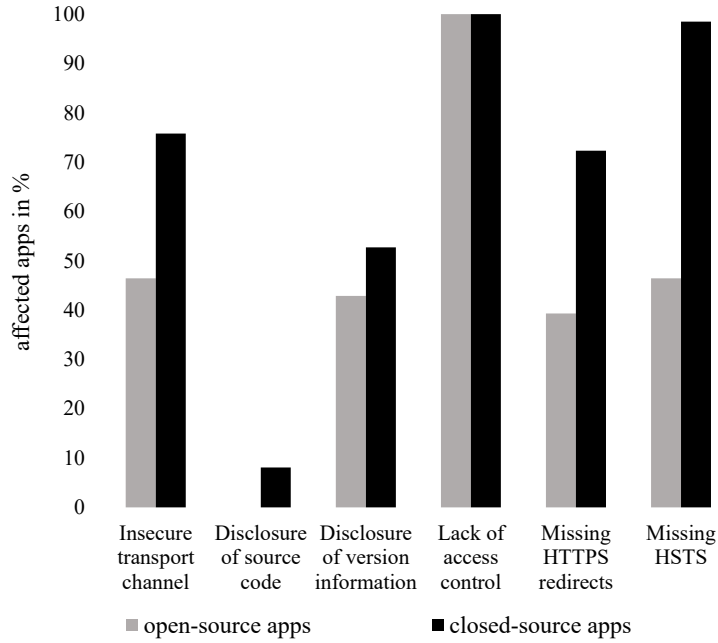


Figure 7.3: Prevalence of app server smells in apps considering JSON communication

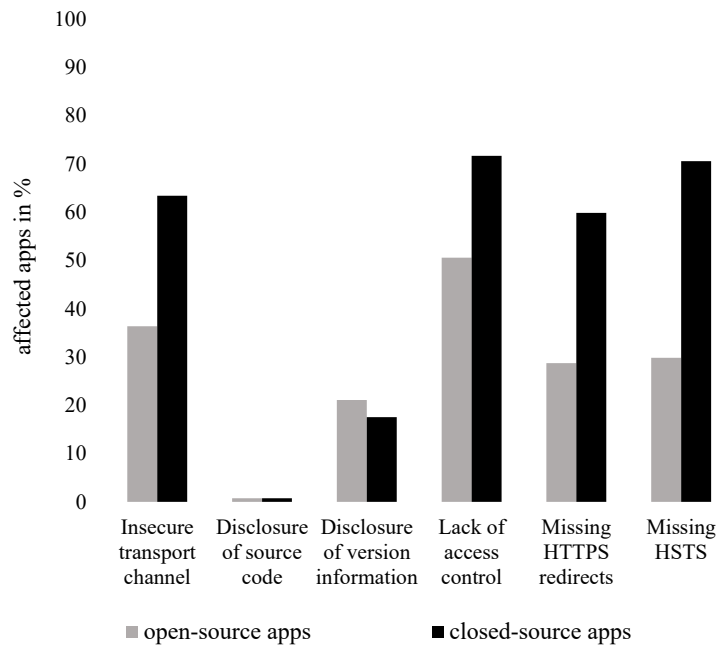


Figure 7.4: Prevalence of app server smells in apps considering non-JSON communication

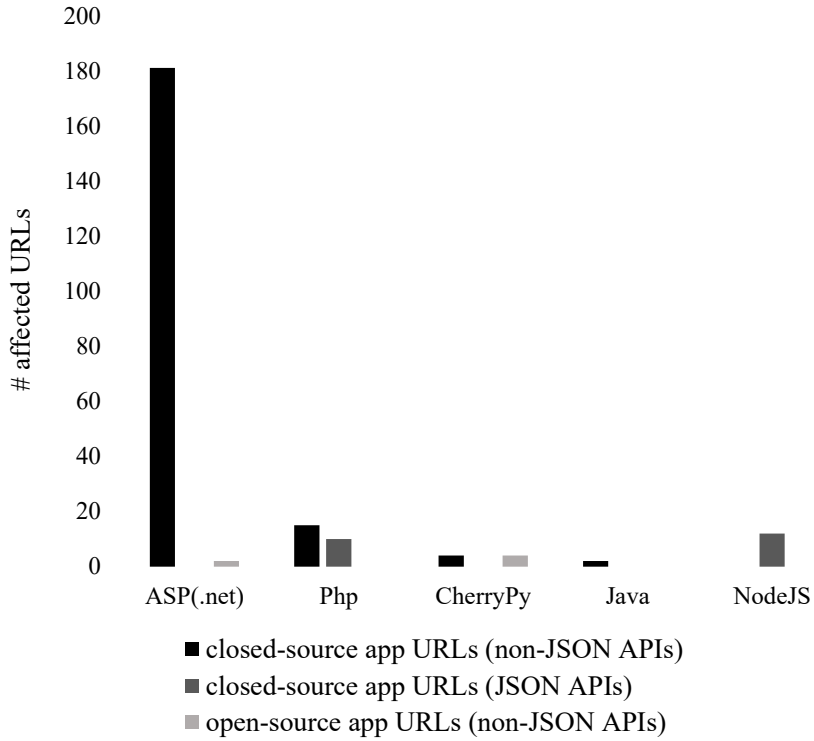


Figure 7.5: Frameworks that caused code leaks

Insecure transport channel. Communication through an insecure transport channel is prone to data leaks and manipulation, *e.g.*, an adversary could alter conversations. Hence, practitioners should avoid HTTP and instead focus on the secure HTTPS. Third-party libraries that require HTTP should be replaced with ones that support secure communication. With respect to URLs from open-source apps, we found that 582 non-JSON app servers (50%) did not use protected communication, respectively 100 open-source apps (36%). This is different for JSON app servers: only six JSON app servers (10%) used plain text communication, however they affect thirteen open-source apps (46%). We found worse results in closed-source communication. Secure communication was usually unavailable, *i.e.*, 5 639 non-JSON app servers (71%) used HTTP, respectively 1 783 closed-source apps (63%). A total of 245 JSON app servers were not protected (50%), respectively 197 closed-source apps (76%).

Disclosure of source code. Leaked code is valuable for adversaries to plot their attacks, or for competitors to glimpse into the source code and the architecture. Therefore, administrators should disable verbose error messages on production environments and review the default settings. We could identify stack traces from five different server frameworks, *i.e.*,

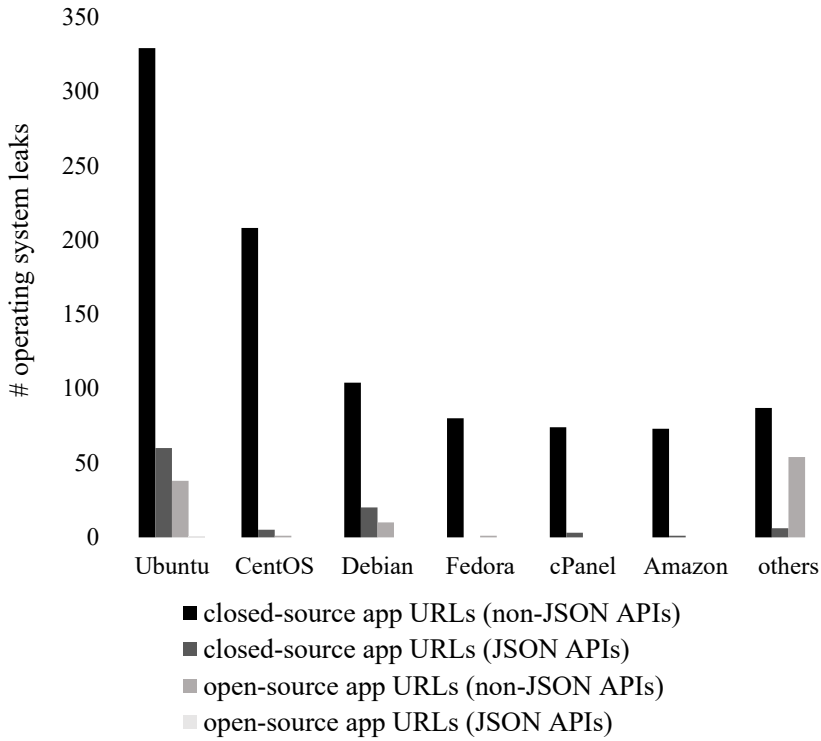


Figure 7.6: Disclosure of operating system information

ASP(.net), CherryPy, Java, NodeJS, and Php. As we can see in Figure 7.5, URLs from closed-source applications suffer the most from code leaks, *i.e.*, we found 225 instances (2.7%) where 182 instances can be assigned to the ASP(.net) framework. Considering URLs used in open-source software, we only found six instances (0.5%) primarily caused by ASP(.net) and CherryPy.

Disclosure of version information. The knowledge of what exact software runs on a server is crucial for successful attacks. Consequently, administrators should disable the self-promotion of services and review their default settings. In Figure 7.6, we present the found operating system leaks in app servers, where the y-axis denotes the number of leaks we found. We found 1 155 operating system leaks in our dataset. Ubuntu and Debian are the most prevalent operating systems for JSON app servers, and CentOS is rather used for non-JSON app servers. Customized Linux distributions, *i.e.*, cPanel and Amazon, are less commonly used among web application developers.

In Figure 7.7, we present the found service leaks in app servers, where the y-axis denotes the number of leaks we found. We found 8 707 service leaks in our dataset, including servers that pack up to three leaks into

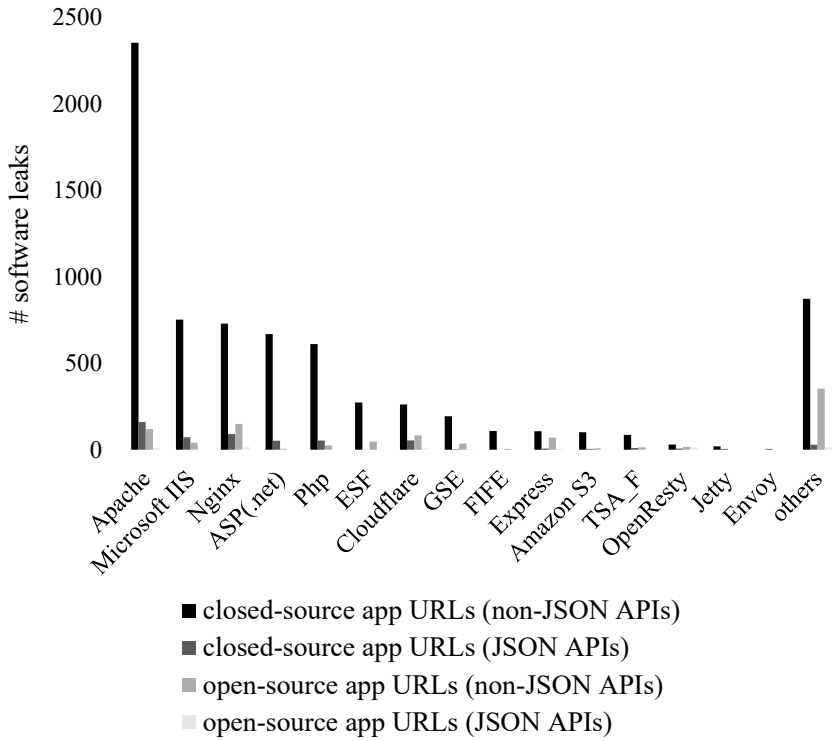


Figure 7.7: Disclosure of service information

a single HTTP response. Open-source and closed-source software behave similarly, *i.e.*, Apache and Nginx are among the top three web application gateway servers used, but Microsoft services, *i.e.*, Microsoft IIS and ASP(.net), remain a preferred choice for closed-source developers. Interestingly, the web security provider Cloudflare is used not only for numerous closed-source apps, but also for open-source apps, as we expect, due to their free plans. Furthermore, the service leaks indicate that most of the app servers do not use the Google Cloud API (ESF) or storage services such as Amazon S3.

In Figure 7.8, we present the found version leaks in app servers, where the y-axis denotes the number of leaks we found. We found 3 992 closed-source and 359 open-source software leaks in our dataset. Most version leaks occur for both closed-source and open-source app servers in the HTTP header field `Server`, followed by `X-Powered-By`, and `X-AspNet-Version`. The leaks in HTTP bodies, *i.e.*, `Apache`, `Nginx`, `Apache H3`, `OpenResty`, and `CherryPy` are less prevalent than those found in the headers.

Lack of access control. Unprotected information can be accessed by everyone on the internet. Since apps usually provide experiences tailored to each user, their servers should use well known authentication schemes to

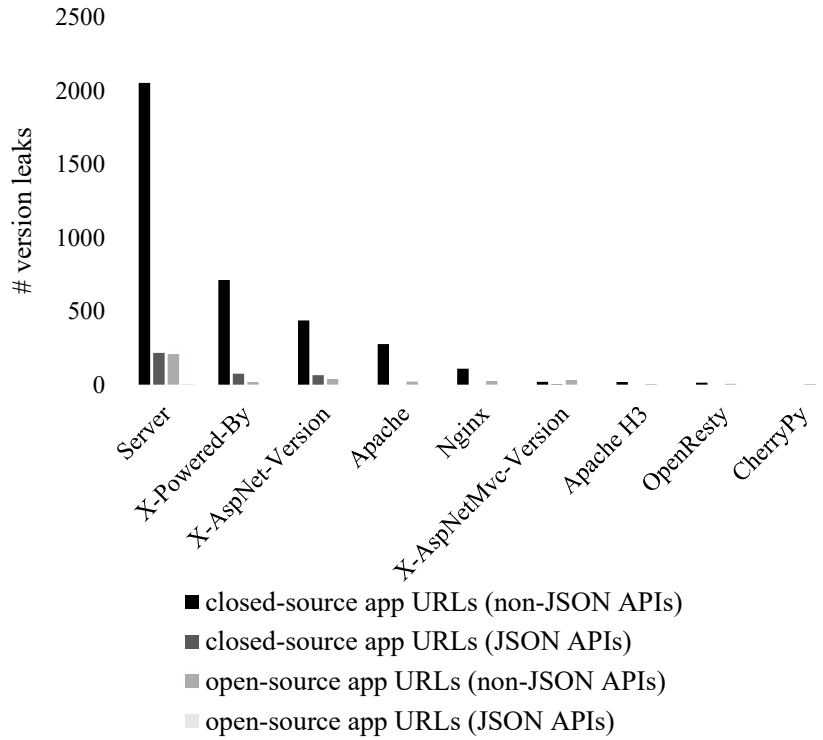


Figure 7.8: Disclosure of version information

prevent leaks of personal data. We encountered 53 HTTP authentication errors for closed-source non-JSON app servers, and 28 errors for open-source non-JSON app servers. We did not find any such errors for open-source or closed-source JSON app servers. However, there exist JSON web applications that returned arbitrary authorization errors in the JSON format, *e.g.*, using OAuth instead of the HTTP mechanism.

Missing HTTPS redirects. Missing redirects leave flawed or outdated clients vulnerable to eavesdropping. Redirects should always be set in place, if a server has ever been accessible through the insecure HTTP protocol. Redirects can be chained, but they should be used sparingly. As shown in Figure 7.9, we found server responses with missing HTTPS redirects in the URLs from 4 961 closed-source apps and from 387 open-source apps. Fortunately, we did not find any HTTPS to HTTP connection downgrades in JSON app servers, but we found 48 for closed-source non-JSON app servers and 15 in open-source non-JSON app servers. Concerning forwarded requests, closed-source app servers forwarded the requests on average 1.3 times, open-source non-JSON app servers 1.5 times, and open-source JSON app servers once. We found two request loops, *i.e.*, infinite redirects from a destination to itself, in each open-source and closed-source

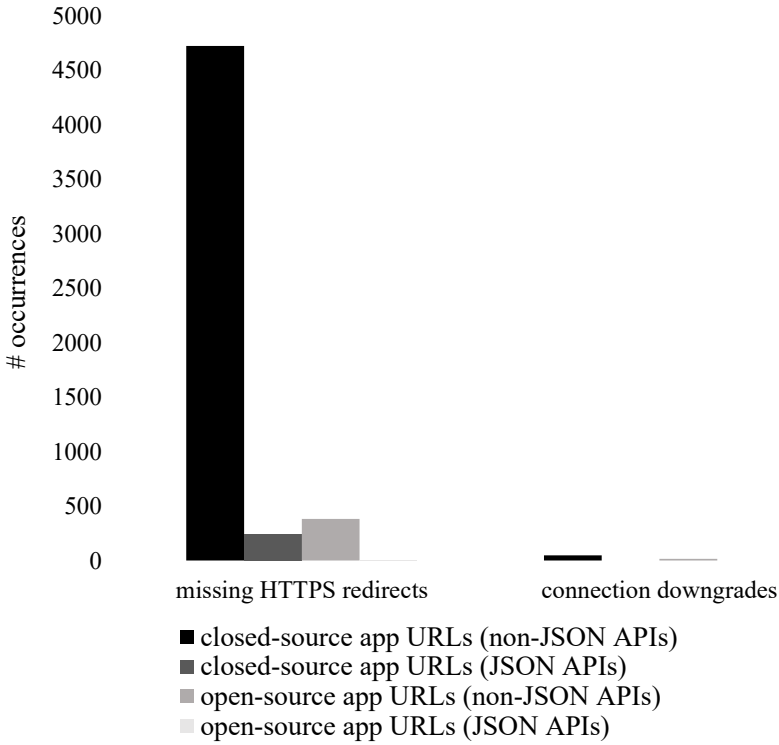


Figure 7.9: Missing HTTPS redirects in app servers

app servers. Without the request loops, open-source app servers redirected a request up to three times, and closed-source app servers up to seven times.

Missing HSTS. App servers without proper support for HSTS expose users to eavesdropping due to possible HTTPS to HTTP connection downgrades. Therefore, servers should deploy this feature to every subdomain and request the client side caching of this setting for at least one year. Ultimately, the protected URLs should be added to the publicly available HSTS preload list that is included in all major browsers. As shown in Figure 7.10, we found 7 494 closed-source app servers and 833 open-source app servers that miss HSTS HTTP headers. Only a minority of the connections are protected, that is 397 (34%) of all open-source app servers and 992 (12%) of all closed-source app servers. Contrary to recommended practices,[5] 432 app servers use `max-age` values shorter than one year, and 785 do not use the preload feature. In other words, 31% of the app servers that support HSTS have not sufficiently configured the protection for subdomains, and 57% lack the preload feature that enforces security already

[5]Google Chrome HSTS preload list submission form, `https://hstspreload.org/`, *accessed on 02-MAR-2022*

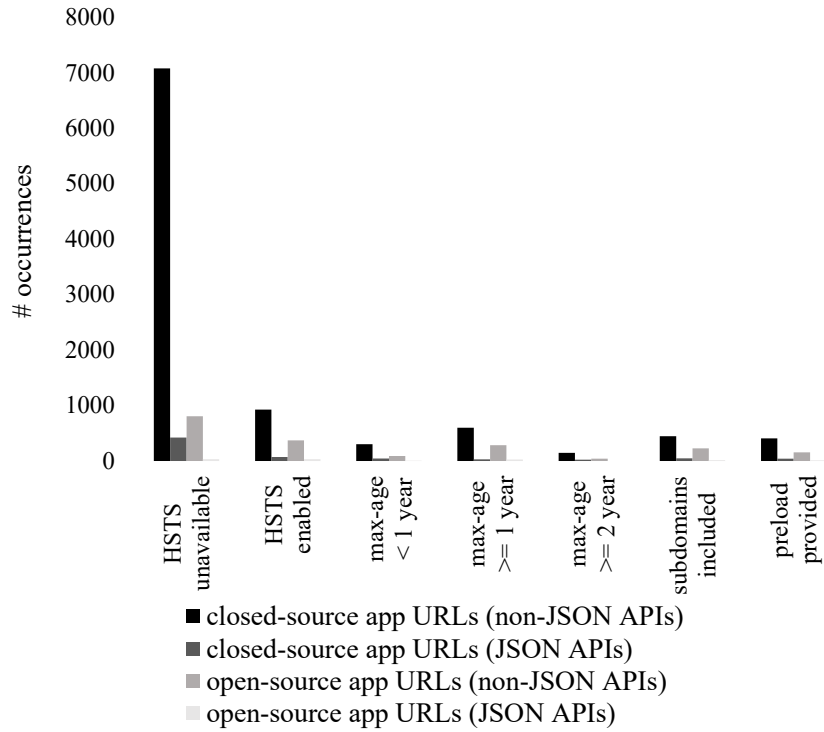


Figure 7.10: Missing HSTS protection for app servers

for the first request.

By Software Development Model

We report findings for two different software development models, *i.e.*, the open-source and the closed-source software development model. We can clearly see in Figure 7.3 and Figure 7.4 that closed-source apps generally suffer from more security smells than open-source apps. Furthermore, *Lack of access control* and *Missing HSTS* appear in the communication of almost all closed-source apps. Over all apps, the three smells *Insecure transport channel*, *Missing HTTPS redirects* and *Disclosure of version information* are less frequent, but exist still in more than 52% of all closed-source apps and in more than 39% of all open-source apps. Interestingly, *Disclosure of source code* primarily emerges in closed-source app communication.

By Technology

We report our findings for two different technologies, *i.e.*, the JSON and non-JSON-based web communication. According to Figure 7.3, access

control and unprotected HTTP communication constitute major threats for apps that use JSON web services. However, apps that do not rely on JSON communication are apparently more robust against security smells: such apps are on average about 19% less affected by them. Code leaks primarily occurred in JSON communication. For instance, *Disclosure of source code* only exists in less than 1% of the apps that use non-JSON web services, whereas it is more than 8% for the apps that use regular JSON web services. We only found code leaks in JSON app servers that use the Php or NodeJS framework, but in contrast, we found code leaks in non-JSON servers from almost every major framework.

Summary

App server security smells pose a severe threat. Most app server security smells affect more than 25% of all apps, regardless whether the app is open-source or closed-source, and whether it uses a JSON or non-JSON app server. Particularly alarming is the finding that apps using JSON app servers suffer 1.5 times more from app server security smells than non-JSON apps, and even worse, closed-source applications suffer 1.6 times more compared to open-source applications.

More than 50% of the servers accessed by mobile apps use unprotected HTTP communication. Since smart devices are becoming rather personal assistants, they carry much sensitive information that needs adequate protection.

Misconfigured app servers cause code leaks. Although only little code is revealed at a time, an attacker can replay requests and alter parameters to reconstruct the architecture and logic behind the service. Such information eases the search for bugs in the code.

The leaked information is devastating. Although intended for publicity purposes, the currently leaked data reveals very often not only the operating system running on the server, but also the installed services and their version number. Such information can be entered into vulnerability databases to find suitable security issues that could be exploited.

Based on our results, access control for JSON app servers is currently not implemented with HTTP status codes, but instead with arbitrary replies. A standardized approach would help in creating more service independent apps, and at the same time default authorization templates could be used from back-end developers.

HTTPS redirects are usually inexistent for HTTP-based app servers. Even worse, some downgrade a HTTPS connection to an insecure HTTP connection. Moreover, redirect loops exist occasionally, and few redirect implementations use more than five redirects, which is not recommended by RFC2068.[6]

[6]Request For Comments (RFC) of the HTTP 1.1 section 10.3, `https://tools.ietf.org/html/rfc2068#section-10.3`, *accessed on 02-MAR-2022*

Finally, HSTS is only set up for a minority of app servers, and for those it is common to have weak configurations.

In conclusion, we see that security smells are very prevalent in app servers. In fact, every app references on average more than three servers that suffer from at least one of these smells.

7.1.3 Maintenance of Server Infrastructure

In order to answer $\mathbf{RQ}_2$: *What is the relationship between security smells and app server maintenance?*, we investigate maintenance operations performed on the servers used by mobile apps. In particular, we are interested whether app server administrators have updated their infrastructure within the time period of fourteen months, and if we see a correlation between the number of identified security smells and the quality of server maintenance. The selected duration of more than a year covers multiple bug fixes including major releases of common server software, *e.g.*, Apache, Microsoft IIS, or PHP. We accessed the URLs by sending an HTTP GET request, and stored their HTTP header responses twice, *i.e.*, once in June 2019 and once in August 2020. We can only compare version numbers between the two datasets if we received some version information in the HTTP `Server` header. As a result, the data in this section are based on fewer responses, *i.e.*, from 309 open-source (JSON and non-JSON) app server URLs (25%) and 3 006 closed-source (JSON and non-JSON) app server URLs (35%).

During our manual analysis of the first 100 entries, we encountered eight different scenarios: i) no updates have been applied, *i.e.*, the software name and version remains identical, ii) the version has been downgraded, *i.e.*, the software name remains identical, but the version number decreased, iii) the version has been upgraded, *i.e.*, the software name remains, but the version number increased, iv) the version leak has been closed, *i.e.*, the software name remains, but the version number is not anymore available, v) the environment has changed, *i.e.*, the software has been replaced and it might use a different versioning scheme, vi) Cloudflare protection has been enabled, *i.e.*, the server has moved behind a Cloudflare protection gateway and does not anymore leak version information, vii) server spawned, *i.e.*, we received no software name in the first run, but we received one in the second run, viii) server shutdown, *i.e.*, we received a software name in the first run, but not anymore in the second run. We could not gather security-related changes for 1 254 app server URLs for several reasons: i) new server instances have been spawned without prior knowledge of software configurations, ii) existing server instances have been shutdown without the possibility to find any changes, or iii) the environment has changed using a different versioning scheme.

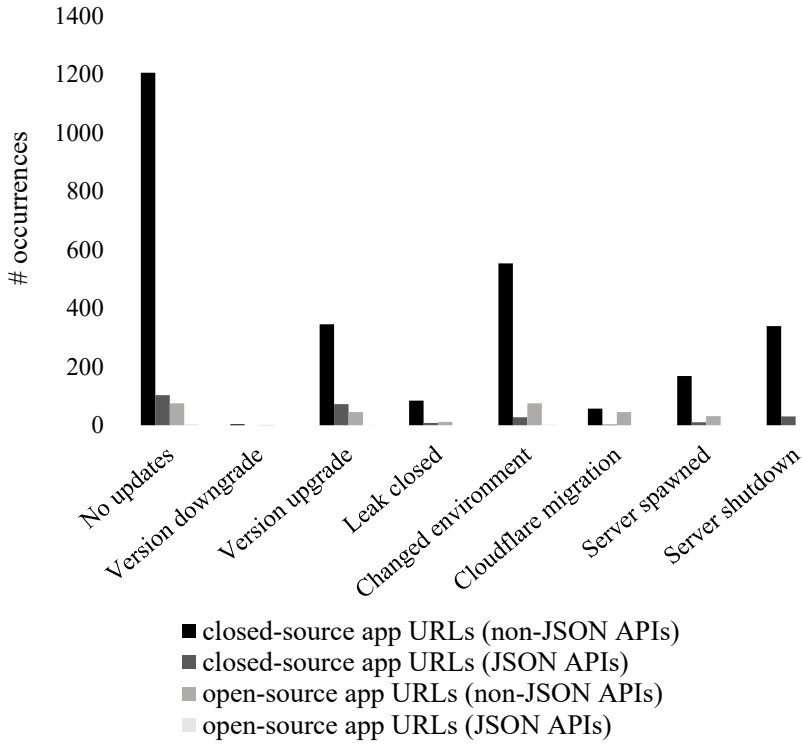


Figure 7.11: Configuration changes of app servers after fourteen months

Configuration Changes

In Figure 7.11, we show the results. From the app servers that leaked versioning information, by far most closed-source non-JSON app servers did not undergo any changes to the server software. Closed-source JSON app server infrastructure seems to be updated more frequently, however the majority still do not provide any updates. The same is true for open-source software although less evident. Version downgrades occurred sparsely, *i.e.*, four times, and not for JSON app servers. Only a fraction of the leaking servers, *i.e.*, 103 (4%), have been configured to mitigate the leaks. Interestingly, environment changes occur more frequently for open-source non-JSON app servers than no updates at all. In other words, open-source developers seem to replace app servers rather then updating them. Moreover, Cloudflare support has been enabled for 104 app servers, *i.e.*, for 45 open-source URLs and for 59 closed-source URLs. Finally, more servers are shut down than spawned.

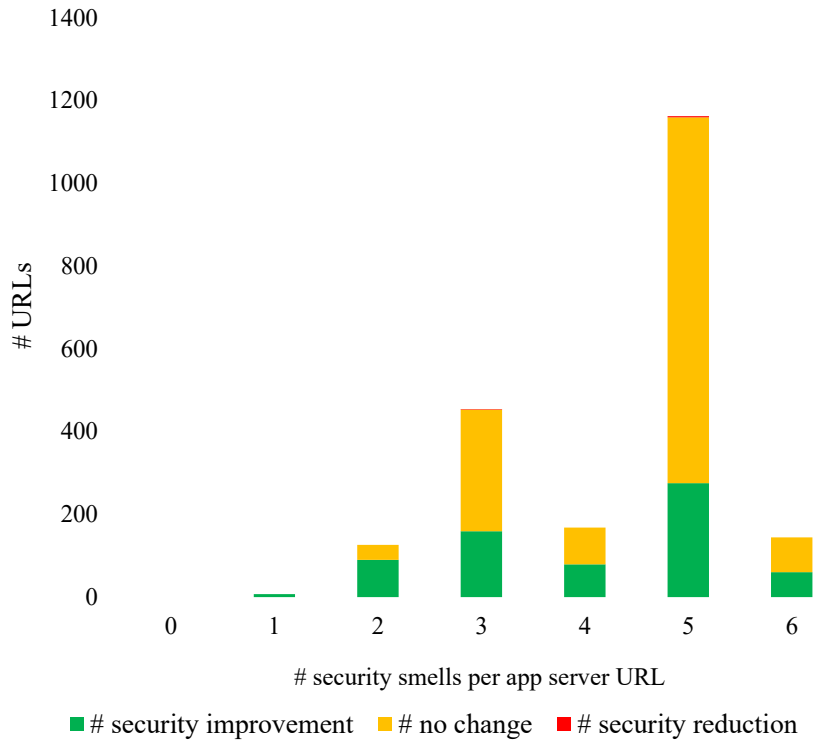


Figure 7.12: Correlation between app server security smells and configuration changes

Correlation of Security Smells

Figure 7.12 shows the correlation between app server security smells and administrative configuration changes. For this figure, we consolidated all app server categories, *i.e.*, open-source, closed-source, JSON, non-JSON due to the limited number of elements in some of them. The x-axis denotes the number of security smells from which a particular app server suffers, and the y-axis indicates how many such app servers exist in each category. Based on the versioning information from 2 061 URLs, we can see that app servers suffering from three or more smells are usually not well maintained, *i.e.*, they are set up once and then left alone. Although security improvements, *i.e.*, version upgrades, the removal of versioning information, and the migration to Cloudflare appear more frequently in instances that suffer from more than one smell, they only affect a minority. Security downgrades, *i.e.*, the change to a more dated version, appear only in app servers that massively suffer from security smells, *i.e.*, from three or more smells.

Summary

According to our findings, app servers are usually set up once and never touched again. This paradigm introduces severe security risks due to outdated software running on publicly accessible interfaces. Hence, sensitive user data could be exfiltrated when adversaries apply suitable exploits to such systems. Luckily, version upgrades are much more common than version downgrades, although they cannot at all compensate for the lack of change. We expect that downgrades were performed to circumvent new bugs or compatibility issues, because all downgrades considered only minor release changes, *e.g.*, from *nginx* release 1.14.1 to 1.12.1. Some developers shift to Cloudflare to protect their infrastructure especially for non-JSON app servers.

We conclude that app server security smells seem to be a good indicator for poor server maintenance. In fact, the more smells an app server has the more likely it is that server maintenance processes are broken.

7.2 Threats to Validity

Completeness. A major threat to validity is the completeness of the used dataset built from Android apps. Although state of the art decompilation tools have been used, only about 37% of all closed-source Android apps could be successfully decompiled for the subsequent analysis. Of these decompiled apps, the analysis for 22% could not finish in time and might have led to incomplete results. Moreover, the analysis tool skipped the evaluation of bundled build scripts and XML resources that could have pointers to additional app servers. This threat cannot be mitigated entirely, however the rather large and diverse set of included apps ensures that the results can be generalized.

Accuracy. Another important threat represents the accuracy of the used dataset. The tool that we used to build the dataset achieves a precision of 46% and a recall of 80%. However, this performance is the result of a manual analysis of decompiled code performed by the authors, which included only ten open-source and ten closed-source apps that comprised 22 web API URLs. In particular, it reported several URLs unrelated to web APIs but to static HTML pages, and the tool occasionally reconstructed invalid requests. In this work, we do not depend on accurate requests, *i.e.*, the investigated response headers are identical even for malformed requests. In fact, most of the reconstructed requests contained placeholders that we could leverage to see whether the app servers leak sensitive information in case of errors.

Data collection. The collected data might contain duplicates or suffer from temporal issues. Some requests we generated from the URL might have reached identical servers, which ultimately lead to duplicated connection information in the result set. Another problem is that of server

side outages or configuration changes that temporarily cause unexpected or erroneous results. To mitigate these threats, we filtered the URL list for duplicates, and we used rather long timeouts and a high retry count when we accessed the servers.

Selection bias. The data used for the investigation of server maintenance represents only a subset of the original dataset. This is an immediate result of the many servers that do not leak any data. Even more, for the qualitative analysis we require two responses, each containing versioning information. In order to reduce the impact of these threats, we manually reviewed the first 100 server responses to ensure that we do not miss any version information. We then designed the value extraction process for the individual version numbers based on the results of this initial exploration.

Recency. The data set contains apps that have been downloaded in 2018, and the corresponding metadata has been collected in 2019. This might change the results due to improved development processes and tools. However, recent works still identified a lack of security in web communication [5, 42].

Security risks. The risks associated with the security smells are not necessarily severe. We do not know what and how much data the web services hoard, and many of the risks directly correlate with the confidentiality of the data. Since we cannot easily obtain this information, we follow a defensive strategy, *i.e.*, we assume that every server might host at least some sensitive data.

Construct validity. There is a threat to construct validity through potential bias in our expectancy.

7.3 Conclusion

We analyzed the prevalence of six security smells in app servers and investigated the consequence of these smells from a security perspective. We used an existing dataset that includes 9 714 distinct URLs that were used in 3 376 Android mobile apps. We exercised the URLs twice over fourteen months, and stored the HTTP headers and bodies. We realized that the top three smells exist in more than 69% of all tested apps, and that unprotected communication and server misconfigurations are very common. Particularly alarming is the finding that apps using JSON app servers suffer 1.5 times more from app server security smells than non-JSON apps, and even worse, closed-source applications suffer 1.6 times more compared to open-source applications. Moreover, source-code and version leaks, or the lack of update policies foster future attacks against these data centric systems. We found that app server security smells are omnipresent and they indicate poor app server maintenance.

Chapter 8

Security Smells in Mobile App HTTP Clients

Declaration of Content Reuse

The content of this chapter is based on the short paper *Security Header Fields in HTTP Clients* that has been accepted for the *21st IEEE International Conference on Software Quality, Reliability, and Security (QRS)* in 2021 [36].

The most prominent clients for the HTTP protocol are web browsers. Modern web browsers receive regular updates every few weeks and protect users from threats with various techniques such as HTTP headers to set up HSTS. However, the support for such HTTP headers in HTTP client libraries, *e.g.*, `HTTPUrlConnection` in *OpenJDK* is missing.

Therefore, in this chapter, we study the presence of HTTP header fields in the communication of mobile apps to understand their use and the provided protection by the HTTP clients. Using the dataset presented in chapter 6, we sent an HTTP GET request to each server behind the 9 714 distinct URLs from 3 073 closed-source and 303 open-source apps. We investigated the server responses to understand the research question, *What is the support of the most common security-related HTTP header fields in existing HTTP clients?* In particular, we present the security-related HTTP header fields, their purpose, and prevalence, and we investigate which are supported in common HTTP libraries.

In the top 50 used header fields in the communication of mobile apps, we could identify sixteen well-known security-related fields. We found that on average 93% of the security-enabling pairs are not used in server responses. We discovered that all commonly used HTTP clients in Android apps lack proper support for the majority of such header fields. We discuss

these header fields and report where HTTP client libraries can benefit from them too. We publicly share our replication package to encourage further research in this direction.[1]

In the remainder of this chapter, we present the used methodology in section 8.1 before we investigate the current HTTP header support in HTTP clients and the resulting security smells in section 8.2. We explain the threats to validity in section 8.3, and we conclude this chapter in section 8.4.

8.1 Methodology

For the analysis we used the URL list that we compiled in chapter 6 to connect to servers and retrieve their responses, which we finally exercised. Based on these results, we manually investigated the HTTP client support for security-related header fields. Please note that the preliminary sourcing of the apps and the extraction of the URLs is not in the scope of this chapter, and we only provide a brief overview of the dataset.

8.1.1 Sourced Apps

The URLs in the dataset are extracted from random Android apps found in the Google Play Store (3 073 apps) and the F-Droid repository (303 apps) that request Android's `INTERNET` permission. Based on their package identifier, only the most recent version of each app has been kept in the dataset. The Play Store apps come from 48 different categories, the most prevalent categories being `EDUCATION` (317 apps) and `TOOLS` (292 apps). Moreover, the apps have an average star rating of 4.2 stars (median: 4.3 stars), and most apps reached between 100 and 1 000 downloads. Barely any app was downloaded more than one million times, and most of the apps were updated in 2018.

8.1.2 URL Extraction

The static analysis tool used to extract the URLs performed three steps for each analyzed app. First, it decompiled the source code that is distributed within the APK installation file to regular Java code. Next, it detected the used web communication APIs in the code and extracted the corresponding data, *e.g.*, the tool can assemble URLs from concatenated string variables within a class, and even reconstruct JSON data structures from JSON object class implementations. Finally, the key-value pairs in the identified URLs were enriched with possible values, which could be found in the code, or their value type if no value was available. This process was error-prone and time demanding. A 32-core machine with 128 GB RAM was allowed

[1] Figshare: replication package, `https://figshare.com/s/c57bb34cadcac225cadc`, *accessed on 02-MAR-2022*

to work on each app for up to 30 minutes before the process was killed, yet the analysis did not complete for some of them. After the analysis, the reported URLs were collected and duplicated URLs removed. URLs that point to the same server, but use a different path or query parameter were considered different to not miss any particular server configuration. In fact, there exist technologies that may hide different servers behind a single IP address, *e.g.*, the Anycast network addressing and routing methodology.[2] Therefore, we did not establish the number of unique servers. In the end, 1 230 open-source URLs and 8 486 closed-source URLs were available for further analyses.

8.1.3 Header Data Collection

The reported URLs consequently represent different kinds of HTTP servers, *e.g.*, for web APIs, media streaming, or website delivery. For every reported URL we issued an HTTP `GET` request and collected the response header information. An empty file has been created for those URLs for which we received no response in the process. After we collected the header information, we used simple pattern matching to gather the prevalence of the different response header fields.

8.1.4 HTTP Client Support

In order to understand whether an HTTP client supports the security-related header fields that we discovered, we searched each field name in the source code, the project website, documentation, and the forums where available. If we encountered matches, we started a manual investigation to determine the extent to which the support is available. For the web browser compatibility, we searched Mozilla's developer network directory, which provides browser compatibility matrices, and for non-standardized fields we had to use Google to find the relevant information. This task was performed by the author and required about fifteen hours.

8.2 Results

In this section we first discuss the prevalence of header fields in server responses for mobile apps, before we show our findings regarding their support in non-browser HTTP clients.

8.2.1 Identified Header Fields

We found 439 header fields, which we could identify in the server responses. We present the top 50 in Table 8.1. The first column denotes the rank

[2] Request For Comments (RFC) for the Operation of Anycast Services, `https://datatracker.ietf.org/doc/html/rfc4786`, *accessed on 30-MAR-2022*

Rank	# Occ.	Header field	Purpose	Relevant to security
01	7 567	Date	performance optimization	Minor: provides a timestamp
02	7 189	Content-Type	data presentation	Minor: is a CORS-safelisted response header
03	*6 978*	*Server*	*advertisement*	*Major: can leak sensitive information*
04	4 032	Content-Length	performance optimization	Minor: is a CORS-safelisted response header
05	3 479	Cache-Control	performance optimization	Minor: is a CORS-safelisted response header
06	3 065	Connection	performance optimization	Minor: provides connection-state
07	2 400	Expires	performance optimization	Minor: is a CORS-safelisted response header
08	2 111	Set-Cookie	cookie management	Minor: cookie transmission
09	1 811	Vary	performance optimization	Minor: enables fine-grained caching
10	1 788	Location	content redirection	Minor: redirect target
11	*1 770*	*X-Powered-By*	*advertisement*	*Major: can leak sensitive information*
12	*1 601*	*X-Content-Type-Options*	*security*	*Major: can prevent content sniffing from arbitrary data*
13	*1 519*	*X-XSS-Protection*	*security*	*Major: can prevent XSS attacks*
14	1 367	Accept-Ranges	performance optimization	Minor: enables partial downloads
15	*1 289*	*X-Frame-Options*	*security*	*Major: can prevent iframe attacks*
16	1 241	Pragma	performance optimization	Minor: is a CORS-safelisted response header
17	971	Last-Modified	performance optimization	Minor: is a CORS-safelisted response header
18	*916*	*Access-Control-Allow-Origin*	*security*	*Major: extends cross origin resource sharing*
19	834	Etag	performance optimization	Minor: document identifier
20	*787*	*Strict-Transport-Security*	*security*	*Major: can prevent HTTPS downgrade attacks*
21	659	Alt-Svc	performance optimization	Minor: HTTP/2 load balancing
22	*601*	*Upgrade*	*security*	*Major: upgrades connection protocol or security*
23	594	P3P	privacy	Minor: privacy web page
24	*538*	*X-AspNet-Version*	*advertisement*	*Major: can leak sensitive information*
25	*517*	*Content-Security-Policy*	*security*	*Major: can restrict access to particular origins*
26	471	Via	debugging	Minor: routing information
27	425	X-Cache	debugging	Minor: caching state at the CDN
28	410	CF-Ray	debugging	Minor: request identifier
29	*336*	*Access-Control-Expose-Headers*	*security*	*Major: exposes selected headers to a frontend*
30	*332*	*Expect-CT*	*security*	*Major: enforces certificate transparency (obsolete)*
31	318	Access-Control-Allow-Methods	performance optimization	Minor: lists supported HTTP methods
32	300	X-UA-Compatible	data presentation	Minor: lists compatible user agents
33	258	Age	performance optimization	Minor: proxy cache duration
34	*255*	*X-Powered-by-Plesk*	*advertisement*	*Major: can leak sensitive information*
35	250	Access-Control-Allow-Headers	performance optimization	Minor: lists supported HTTP headers
36	225	Status	debugging	Minor: server response status
37	*205*	*Access-Control-Allow-Credentials*	*security*	*Major: exposes credentials to a frontend*
38	196	Transfer-Encoding	performance optimization	Minor: specifies data encoding
39	192	X-GitHub-Request-Id	debugging	Minor: request identifier
40	*189*	*Timing-Allow-Origin*	*security*	*Major: can introduce side-channel attacks on personalized data*
41	*186*	*Referrer-Policy*	*security*	*Major: can restrict the exposed referrer information*
42	185	X-Amz-CF-Id	debugging	Minor: request identifier
	185	X-Amz-CF-Pop	debugging	Minor: server identifier
43	182	X-Request-Id	debugging	Minor: request identifier
44	175	X-Cache-Hits	debugging	Minor: server side caching statistics
45	174	X-Served-By	debugging	Minor: lists CDN caching servers
46	166	X-FB-Debug	debugging	Minor: request debugging information
47	163	Allow	performance optimization	Minor: lists supported HTTP methods
48	131	Content-Disposition	data presentation	Minor: specifies data presentation
	131	X-Amz-Id-2	debugging	Minor: request identifier
	131	X-Amz-Request-Id	debugging	Minor: request identifier
49	127	Content-Language	data presentation	Minor: is a CORS-safelisted response header
50	123	X-Timer	debugging	Minor: message transport statistics

Table 8.1: Top 50 HTTP headers in mobile app web communication

among the most prevalent header fields, the second column denotes the number of occurrences in our dataset, the third column presents the header field name, and the fourth column reveals the purpose. We collected this information from Mozilla's Developer Network,[3] or, if unavailable, from websites operated by the corresponding protocol designers that could be found using Google Search. We proceeded identically for the last column, which shows the relevance to security for each header field together with a brief explanation. We highlighted the header fields that are directly related to security.

We found that the most prevalent header fields serve eight different purposes, *i.e.*, sixteen (30%) allow *performance optimization*, fourteen (26%) support *debugging*, twelve (23%) address *security*, four (8%) perform *advertisement*, four (8%) request *data presentation*, one (2%) enables *cookie management*, one allows *content redirection*, and, finally, one clarifies *privacy*. As we can see, only a subset of them are security-related, *i.e.*, sixteen header fields (30%). Moreover, only four of them (8%) do not increase security, but pose a threat by leaking information.

8.2.2 Security-related Header Fields

Within the top 50 fields, we identified sixteen (30%) that may introduce or resolve a security threat, *e.g.*, fields that can prevent arbitrary code execution, click-jacking attacks, or data leaks. We considered fields that leak version information as also being security-related, because such data can make existing servers an easy target for adversaries. We present the details in Table 8.2. The first and second columns denote the header field name and its intended use. The third column cites the relevant specification document, where available. The fourth column indicates either the corresponding threat that the field mitigates, or the threat that a field can introduce, *i.e.*, a version leak. The fifth column shows the total number of URLs returning a specific header field in the order `total responses (open-source responses / closed-source responses)`, and the last column reveals affected percentage of URLs following the same order.

From a total of 9 714 responses, the `Server` header field is omnipresent and included in 72% of them. We observe that `X-Powered-By`, `X-AspNet-Version`, and `X-Powered-By-Plesk` are less frequently adopted fields, which were present in 18%, 6% and 3% of the responses, respectively. The fields `X-Content-Type-Options` and `X-XSS-Protection` (both 16%), and `X-Frame-Options` (13%) are among the top five most used header fields, but they only occurred in around every seventh response. `Strict-Transport-Security` and `Expect-CT` are rarely used (8% and 3%). Nevertheless, whether these fields are relevant for every app is not known and needs to be investigated in future research.

[3] Mozilla Developer Network (MDN): HTTP headers, `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers`, *accessed on 02-MAR-2022*

Header	Use	Specification	Threat	# Responses	% URLs
Server	Software used by the origin server to handle the request.	RFC 2616 [32]	version leak	6 978 (909 / 6 069)	70% (74% / 72%)
X-Powered-By	Specifies the technology supporting the web application.	non-standard	version leak	1 770 (95 / 1 675)	18% (8% / 20%)
X-Content-Type-Options	Can be used to require checking of a response's "Content-Type" header against the destination of a request.	WHATWG [48]	code execution	1 601 (330 / 1 271)	16% (27% / 15%)
X-XSS-Protection	Stops pages from loading when they detect reflected cross-site scripting (XSS) attacks.	non-standard	code execution	1 519 (321 / 1 198)	15% (26% / 14%)
X-Frame-Options	Indicates a policy that specifies whether the browser should render the transmitted resource within a <frame> or an <iframe>.	RFC 7034 [83]	click-jacking	1 289 (317 / 972)	13% (26% / 11%)
Access-Control-Allow-Origin	Indicates whether a resource can be shared based by returning the value of the Origin request header, "*," or "null" in the response.	WHATWG [48]	data leak	916 (141 / 775)	9% (11% / 9%)
Strict-Transport-Security	Indicates to a UA that it MUST enforce the HSTS Policy in regards to the host emitting the response message containing this header field.	RFC 6797 [41]	data leak	787 (251 / 536)	8% (20% / 6%)
Upgrade	Intended to provide a simple mechanism for transitioning from HTTP/1.1 to some other protocol on the same connection.	RFC 7230 [31]	data leak	601 (0 / 601)	6% (0% / 7%)
X-AspNet-Version	A state server implementation indicates which version of the state server is using this response header.	non-standard	version leak	538 (38 / 500)	5% (3% / 6%)
Content-Security-Policy	Preferred mechanism for delivering a policy from a server to a client.	W3C [104]	code execution	517 (176 / 341)	5% (14% / 4%)
Access-Control-Expose-Headers	Indicates which headers can be exposed as part of the response by listing their names.	WHATWG [48]	data leak	336 (23 / 313)	3% (2% / 4%)
Expect-CT	Allows web host operators to discover misconfigurations in their certificate transparency deployments.	IETF [91]	data leak	332 (147 / 185)	3% (12% / 2%)
X-Powered-By-Plesk	Advertises the used Plesk server software.	non-standard	version leak	255 (0 / 255)	3% (0% / 3%)
Access-Control-Allow-Credentials	Indicates whether the response can be shared when request's credentials mode is "include."	WHATWG [48]	data leak	205 (7 / 198)	2% (1% / 2%)
Timing-Allow-Origin	Defines an interface for web applications to access the complete timing information for resources in a document	W3C [103]	data leak	189 (16 / 173)	2% (1% / 2%)
Referrer-Policy	While the header can be suppressed for links with the noreferrer link type, authors might wish to control the Referer header more directly.	W3C [27]	data leak	186 (79 / 107)	2% (6% / 1%)

Table 8.2: Security-related HTTP header fields found in server responses sorted by their prevalence

When comparing open and closed-source app URLs, we observed that open-source apps are slightly better. For example, the `Strict-Transport-Security` field was more common in open-source app responses, even by considering the more extensive use of HTTPS connections. Similarly, some well-known security-related HTTP header fields were more prevalent in open-source server responses, *e.g.*, `X-Content-Type-Options`, `Content-Security-Policy`, `X-Frame-Options`, and `X-XSS-Protection`.

8.2.3 Security Smells in HTTP Clients

We report security smells in HTTP clients based on our findings in Table 8.3. One of these HTTP client-related smells, namely *Unsupported HSTS*, has been mentioned in subsection 6.3.2, however in this chapter we focus particularly in the context of a HTTP client. For a feature, *i.e.*, a security-related header field that is fully supported we use the symbol ✔, and for those with limited support (✔). Limited support refers to features that only have partial support, *e.g.*, only selected options are implemented, the implementation has not yet been released, or the corresponding logic is only available as stub, *i.e.*, the most frequent header fields[4] are parsed, but not evaluated. We use the symbol ✖ for unsupported features.

The following software releases have been evaluated in this study: *Glide 4.7.0* from February 2018, *HttpComponents 5.1* from August 2021, *Ion* from November 2020, *LoopJ 1.4.9* from January 2021, *OkHttp* from August 2021, *RetroFit* from August 2021, *Volley 1.2.1* from August 2021, *URLConnection*, *HttpUrlConnection*, *HttpsUrlConnection*, and *Socket*, each in OpenJDK 18 from August 2021, *Android WebView 90* from April 2021, *Google Chrome 92* from August 2021, *Microsoft Edge 92* from August 2021, and finally, *Mozilla Firefox 91* from August 2021. The only HTTP clients that contain closed-source components are Google Chrome and Microsoft Edge, which both rely on the open-sourced Chromium rendering engine, however they have propretiary customizations, *e.g.*, vendor account synchronization.

The support for versioning information features (`Server`, `X-AspNet-Version`, `X-Powered-By`, `X-Powered-By-Plesk`) remains incomplete across the different softwares: although the header field is accessible in almost all tested API clients, no further routines are available to dynamically react on them, *e.g.*, by disrupting connections to insecure servers.

HttpComponents has only stubs with no provided logic behind some security-related header fields, however it is one of the few clients that contains code to treat multiple different scenarios of the `Upgrade` field, *e.g.*, a connection protocol upgrade from HTTP to HTTPS, from HTTP/1.1 to HTTP/2, or from HTTP to WebSocket.

[4]Request For Comments (RFC) of the header compression for HTTP/2, `https://tools.ietf.org/html/rfc7541#appendix-A`, *accessed on 02-MAR-2022*

Header	Glide	HttpComponents	Ion	Loopj	OkHttp	Retrofit	Volley	HttpsURLConnection	HttpURLConnection	Socket	URLConnection	Android WebView	Google Chrome	Microsoft Edge	Mozilla Firefox
Server	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	✖	(✓)	✓	✓	✓	✓
X-Powered-By	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	✖	(✓)	(✓)	(✓)	(✓)	(✓)
X-Content-Type-Options	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
X-XSS-Protection	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
X-Frame-Options	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
Access-Control-Allow-Origin	✖	✖	(✓)	✖	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
Strict-Transport-Security	✖	✖	(✓)	✖	✖	(✓)	✖	✖	✖	✖	✖	✓	✓	✓	✓
Upgrade	✖	✓	✖	✓	✓	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
X-AspNet-Version	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	✖	(✓)	(✓)	(✓)	(✓)	(✓)
Content-Security-Policy	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
Access-Control-Expose-Headers	✖	(✓)	✖	(✓)	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
Expect-CT	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓	✓	(✓)
X-Powered-By-Plesk	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	✖	(✓)	(✓)	(✓)	(✓)	(✓)
Access-Control-Allow-Credentials	✖	(✓)	✖	(✓)	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
Timing-Allow-Origin	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
Referrer-Policy	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓

Table 8.3: Support of HTTP security-related header fields for frameworks, Java classes, and web browsers

Security Smell: Unsupported connection upgrades

Only a few HTTP clients support connection upgrades to secure communication.
Therefore, communication may remain unprotected [92].
Recommendation: If a client detects an insecure connection, it should try to upgrade to a secure connection.

Furthermore, within the past six months we could see that the number of security-relevant header field stubs has continuously increased in *HttpComponents* and *OkHttp*, *e.g.*, the `Access-Control-Allow-Origin` and the `Strict-Transport-Security` field are now explicitly parsed, although still not evaluated, which has not been the case before. Since *LoopJ* is based on *HttpComponents* it has access to the same feature set, but unfortunately it currently uses an older release with fewer features.

Security Smell: Abandoned HTTP client

Some of the used HTTP clients seem abandoned and do not anymore receive frequent updates.
Therefore, they increasingly lack support for essential security features [100].
Recommendation: They should not be used.

We can further see that the HSTS policy has been considered for the *OkHttp* client: there exists a feature request ticket from February 2017 in Square's GitHub product page asking for HSTS support.[5] Unfortunately, although the feature has apparently been implemented in some internal developer builds, it has not yet found its way into the production releases.

Security Smell: Unsupported HSTS

No evaluated HTTP client does fully support HSTS policies, although this feature has been discussed and partially implemented in one of the major HTTP clients.
Therefore, users remain unprotected against man-in-the-middle attacks [51].
Recommendation: HTTP clients should fully support this feature.

Finally, the `Socket` class is different from the other contenders, because it is mainly built for low-level network communication that does not consider any information from the ISO/OSI layers four or higher that are required for HTTP headers.

[5] OkHttp issue #3170: support HSTS, `https://github.com/square/okhttp/issues/3170`, *accessed on 02-MAR-2022*

To summarize, we can see that support for header fields barely exists among HTTP client libraries, however, the opposite is true for all web browsers: they support all features with only a few exceptions, *e.g.*, for obsolete header fields.

8.3 Threats to Validity

Selection of URLs. The foremost threat to validity of this study is the selection bias, *i.e.*, whether the selection of URLs is representative. The authors of the URL extraction study strived to collect URLs from more than three thousand random apps from various categories in order to obtain relevant results. The analyzed apps were popular and achieved many downloads. Our decision to include URLs that point to the same server, but use a different path or query parameter ensures that every server is accessed, however it might introduce false positives when the same server is serving multiple endpoints.

Dataset accuracy. Our analyzer that was used to build the list of URLs suffers from the inherent limitations of static code analysis. Moreover, the pre-processed list may suffer from inconsistencies due to their rather opportunistic approach, *e.g.*, broken variable assignments, *etc.* However, the URLs generally do not need to be very accurate to retrieve reasonable results; when the domain and the path are correct, the different query parameters generally point to the same server.

Selection of header fields. We only considered security header fields that were prevalent in the received HTTP responses and did not scrutinize other header fields. As a result, we missed, for example, the outdated header field `Public-Key-Pins`, which has been used for certificate pinning before `Expect-CT` became available. We only queried the servers with the most commonly used HTTP GET request method, but not other methods such as PUT, DELETE, or POST, which are less prevalent. Moreover, we used the default settings of the *curl* application, where not otherwise specified, to collect the headers, which can alter the results if a server reacts differently depending on the transmitted user agent.

Library support for security-related header fields. Where available, we searched in the source code for the support of particular header fields, *i.e.*, we checked whether a header field name exists. Although a match likely indicates that the header field is at least partially supported, its absence does not necessarily indicate that it is unsupported, although it is very likely. We tried to mitigate this risk by intensively investigating the found matches, and by manually skimming relevant classes for such logic.

Sensitive data. Finally, we assume that every communication involves the transmission of sensitive data and should be secure, which is not always true. However, the detection of data sensitivity is a non-trivial task and not part of this work.

8.4 Conclusion

We collected the HTTP response header information from 9 714 distinct URLs found in 3 376 Android apps. We discovered that, on average, 93% of the security-related headers are not used in server responses, indicating great potential for future improvements. We also found that unlike major web browsers, the support for such fields in HTTP client libraries is very limited, and that server responses for mobile apps frequently lack them. We encourage a more comprehensive use of existing HTTP headers and timely development of relevant web browser security features in HTTP client libraries.

Chapter 9

Effective Holistic Security for Mobile Apps

In the previous five chapters we investigated security smells in different domains, *i.e.*, Android, Android ICC, web communication, app servers, and HTTP clients used in mobile apps. We observed that many of the reported security smells have unique remedies and require additional context, and thus their consideration is time consuming and error prone.

A potential solution to this problem is that of effective security measures that can prevent or resolve multiple security smells at once. In an app, for example, where all user input containing distinct features of a computer language is blocked, many, if not most code injection attacks could be effectively repelled. However, we do not yet genuinely know how such measures should look since we first have to understand the interaction of security smells with common threats, *i.e.*, how large their support is for attacks that have been launched in practice. With this knowledge, we can then reason about the key characteristics required by security measures that can address several related attacks at once, and thus could provide holistic security for mobile apps.

To assess this information, we utilize a comprehensive classification taxonomy of known attacks that have been launched in the wild, *i.e.*, the Mitre Common Attack Pattern Enumeration and Classification (CAPEC) taxonomy.[1] The taxonomy defines nine major categories of attack mechanisms that group meta, standard, and detailed attack mechanisms based on some common characteristic. All these attack mechanisms are *categories* of attacks that are used to classify CVE vulnerability reports.

Therefore, in the remainder of this chapter we will simply use the term "category" wherever we refer to a particular category of attack mechanisms, *i.e.*, we refer to the standard category wherever we use the term

[1] CAPEC view: attack mechanisms (version 3.7), `https://capec.mitre.org/data/definitions/1000.html`, *accessed on 19-MAR-2022*

without the words "major," "meta," or "detailed" beforehand.

The terminology of major, meta, standard, and detailed categories originates from CAPEC and they are organized as hierarchy, however the resulting hierarchy is not balanced, and not always consistent in terms of hierarchy levels that correspond to a particular category, except for the major and the meta categories. The used taxonomy is shown in Table 9.1, which is continued in Table 9.2 where the nine major categories are emphasized in bold, the meta categories are emphasized in italics, and the standard or detailed categories are displayed using the default typeface. A typical example of a rather abstract major category is *Inject unexpected items*, which encompasses all categories related to data injection attacks. Meta categories are subordinate to the major categories and thus more concrete, *e.g.*, those related to *Inject unexpected items* are *Code inclusion*, *Code injection*, *Command injection*, *Hardware fault injection*, *Local execution of code*, and *Object injection*. Standard categories are subordinate to meta categories and are even more concrete with respect to a particular attack. For example, the meta category *Code injection* encompasses the standard category *Cross-site scripting (XSS)*, *Embedding scripts within scripts*, *File content injection*, *Generic cross-browser cross-domain theft*, and *Using meta-characters in email headers to inject malicious payloads*.

We manually investigated the impact of our 51 security smells on 192 standard categories, which led to 9 792 combinations that had to be evaluated. After that, we further condensed the results of all standard categories to the nine superordinate major categories to better identify important findings. We found four major threats that Android users face from our reported security smells. With these findings in mind, we start a discussion about effective holistic security and investigate suitable remediation strategies such as more secure default values, safer best practices, and smart data types, which we discuss in more detail in the two following chapters.

In this chapter, we address the following research questions:

RQ$_1$: *Which security smells enable what attack mechanisms?* After we established the correlation of all security smells with the attack mechanisms, we found that insecure algorithms, the abuse of existing functionality, data leaks, and user deception are the four major threats when using Android. Particularly dangerous are weak crypto algorithms and configurations, which enable the employment of attack mechanisms that depend on probabilistic techniques, and exposed Adb-level capability that fosters acquisition of sensitive resources.

RQ$_2$: *What are holistic security strategies that can effectively prevent attack mechanisms?* We explain the concept of effective and holistic security in the context of mobile apps and elaborate strategies with respect to the four most affected major categories considering the results of RQ$_1$. We see most potential in secure default values and safer practices to prevent feature misuse in the Android ecosystem. We realized further that string

variables are responsible for most issues that relate to the employment of probabilistic methods and the injection of unexpected items, and thus they need increased protection.

In the remainder of this chapter, we discuss the attack mechanisms in section 9.1, and present the related empirical study in section 9.2, before we discuss effective security in section 9.3 and holistic security in section 9.4. We then take a short detour in section 9.5 to focus on Android OS security where we establish a security triad that can illustrate the current conflict of concerns. We report the threats to validity in section 9.6, and conclude in section 9.7.

9.1 Attack Mechanisms

CVE is maintained by the Mitre corporation and is the largest publicly accessible vulnerability database [85]. Mitre used this knowledge to build a comprehensive dictionary and classification taxonomy of known attacks,[2] *i.e.*, *Common Attack Pattern Enumeration and Classification (CAPEC)* database. With nine major categories CAPEC encompasses 192 different standard categories that have been observed in the wild.

Although it is not explicitly mentioned for the categories, some of the reported categories can have more reach than others, *e.g.*, categories such as phishing can be used in distributed attacks to address millions of potential victims simultaneously. Preferred targets of such categories are international companies to extort ransom, the distribution of spam emails, and the creation of fake internet traffic such as simulated clicks on ads to increase any impact-based revenue for the adversary [6].

To make the comprehensive taxonomy more tangible to the reader, we use a popular scenario, *i.e.*, an adversary that tampers with crypto money, to briefly illustrate how typical categories relate to the nine major CAPEC categories, which are emphasized in bold.

- **Abuse existing functionality**. An adversary could trick a crypto service provider to illegally transfer funds, or flood network interfaces to delay transaction acknowledgments of block chain computations that can enable further attacks.
- **Collect and analyze information**. An adversary probes the API servers of a crypto provider to discover potential vulnerabilities that can be exploited to finally take them over. Another example is the interception of network traffic: an adversary intercepts communication between a user and the crypto provider to exfiltrate the used credentials that eventually enable the theft of the user's crypto wallet.

[2] CVE: CVE-CWE-CAPEC relationships, `https://cve.mitre.org/cve_cwe_capec_relationships`, *accessed on 19-MAR-2022*

Abuse existing functionality
Communication channel manipulation
Choosing message identifier
Exploiting incorrectly configured SSL/TLS
Excessive allocation
ICMP fragmentation
Oversized serialized data payloads
Regular expression exponential blowup
Serialized data with nested payloads
SOAP array blowup
TCP fragmentation
UDP fragmentation
Flooding
Amplification
BlueSmacking
HTTP flood
ICMP flood
SSL flood
TCP flood
UDP flood
XML flood
Functionality bypass
Calling micro-services directly
Evercookie
Transparent proxy abuse
Functionality misuse
Drop encryption level
Inducing account lockout
JSON hijacking (a.k.a. JavaScript hijacking)
Passing local filenames to functions that expect a URL
Password recovery exploitation
Interface manipulation
Exploit non-production interfaces
Exploit script-based APIs
Try all common switches
Using unpublished interfaces
Protocol manipulation
Client-server protocol manipulation
Data interchange protocol manipulation
Inter-component protocol manipulation
Reflection attack in authentication protocol
Web services protocol manipulation
Resource leak exposure
Sustained client engagement
HTTP DoS

Collect and analyze information
Excavation
Collect data as provided by users
Collect data from common resource locations
Pull data from system resources
Query system for information
Retrieve data from decommissioned devices
Fingerprinting
Active OS fingerprinting
Application fingerprinting
Passive OS fingerprinting
Footprinting
Account footprinting
File discovery
Group permission footprinting
Host discovery
Malware-directed internal reconnaissance
Network topology mapping
Owner footprinting
Peripheral footprinting
Port scanning
Process footprinting
Services footprinting
System footprinting
Information elicitation
Pretexting
Interception
Android intent intercept
Eavesdropping
Sniffing attacks
Protocol analysis
Cryptanalysis
Reverse engineering
Black box reverse engineering
White box reverse engineering

Employ probabilistic techniques
Brute force
Encryption brute forcing
Password brute forcing
Fuzzing
Fuzzing for application mapping

Engage in deceptive interactions
Action spoofing
Android activity hijack
Clickjacking
Tapjacking
Task impersonation
Content spoofing
Checksum spoofing
Counterfeit GPS signals
Intent spoof
Spoofing of UDDI/ebXML messages
Identity spoofing
Fake the source of data
Pharming
Phishing
Principal spoof
Signature spoof
Manipulate human behavior
Influence perception
Influence via incentives
Influence via psychological principles
Pretexting
Target influence via framing
Resource location spoofing
Establish rogue location
Redirect access to libraries

Inject unexpected items
Code inclusion
Local code inclusion
Remote code inclusion
Code injection
Cross-site scripting (XSS)
Embedding scripts within scripts
File content injection
Generic cross-browser cross-domain theft
Using meta-characters in email headers ...
Command injection
IMAP/SMTP command injection
LDAP injection
Manipulating writeable terminal devices
NoSQL injection
OS command injection
SQL injection
XML injection
Hardware fault injection
Mobile device fault injection
Local execution of code
Targeted malware
Object injection
(continues in Table 9.2)

Table 9.1: The attack mechanisms according to the Mitre CAPEC, version 3.7

Inject unexpected items (continued)
Parameter injection
Argument injection
Command delimiters
Email injection
Flash injection
Format string injection
Reflection injection
Resource injection
Cellular data injection
Traffic injection
Connection reset

Manipulate data structures
Buffer manipulation
Overflow buffers
Overread buffers
Input data manipulation
Integer attacks
Leverage alternate encoding
Path traversal
Pointer manipulation
Shared resource manipulation

Manipulate system resources
Configuration/environment manipulation
Data injected during configuration
Disable security software
Manipulate registry information
Manipulating writeable configuration files
Schema poisoning
Contaminate resource
File manipulation
Alternative execution due to deceptive filenames
Artificially inflate file sizes
Hiding malicious data or code within files
User-controlled filename
Hardware integrity attack
Malicious hardware update
Physically hacking hardware
Infrastructure manipulation
Audit log manipulation
Block logging to central repository
Cache poisoning
Contradictory destinations in traffic routing schemes
Force the system to reset values
Malicious logic insertion
Infected hardware
Infected memory
Infected software
Manipulation during distribution
Malicious hardware component replacement
Malicious software implanted
Rogue integration procedures
Modification during manufacture
Design alteration
Development alteration
Obstruction
Blockage
Jamming
Physical destruction of device or component
Route disabling
Software integrity attack
Alteration of a software update
Exploitation of transient instruction execution
Malicious software download
Malicious software update

Manipulate timing and state
Forced deadlock
Leveraging race conditions
Leveraging TOCTOU race conditions
Manipulating state
Bypassing of intermediate forms in multiple-...
Exploitation of transient instruction execution

Subvert access control
Adversary in the middle (AiTM)
Adversary in the browser (AiTB)
Application API message manipulation via ...
Application API navigation remapping
Leveraging active AiTM attacks ...
XML routing detour attacks
Authentication abuse
Reflection attack in authentication protocol
Unauthorized use of device resources
Authentication bypass
Escaping virtualization
Forceful browsing
Key negotiation of Bluetooth attack (KNOB)
Server side request forgery
Web services API signature forgery leverag. ...
Bypassing physical security
Bypassing electronic locks and access controls
Bypassing physical locks
Exploitation of trusted identifiers
Cross site request forgery
SaaS user request forgery
Session credential falsification through forging
Session hijacking
Exploiting trust in client
Create malicious client
Manipulating opaque client-based data tokens
Manipulating user-controlled variables
Removing important client functionality
Physical theft
Privilege abuse
Accessing functionality not properly constr. ...
Data serialization external entities blowup
Exploiting incorrectly configured access ...
Using malicious files
WebView exposure
Privilege escalation
Cross zone scripting
Hijacking a privileged process
Hijacking a privileged thread of execution
Subvert code-signing facilities
Target programs with elevated privileges
Use of known domain credentials
Credential stuffing
Remote services with stolen credentials
Use of known Kerberos credentials
Use of known Windows credentials

Table 9.2: The attack mechanisms according to the Mitre CAPEC, version 3.7 (continued)

- **Employ probabilistic techniques**. An adversary uses a probabilistic technique to decrypt a protected crypto wallet, *e.g.*, by performing a brute force attack on the used password. Another probabilistic method is fuzzing, which uses generated values to identify failures and ultimately weaknesses in a system, *e.g.*, crypto web API service.

- **Engage in deceptive interactions**. An adversary deceives a user in malicious interactions such as a tap on a link from a spam email that leads to a phishing site. The process of deception is very broad and includes, for example, social engineering with the help of incentives or psychological principles to steal crypto funds.

- **Inject unexpected items**. An adversary exploits a code injection vulnerability to the crypto provider's web API to access user data and funds. A related technique is code inclusion that involves the addition or replacement of file references to executable code, *e.g.*, when a crypto app stores the wallet data using a referenced XML schema. If the adversary could alter that reference to point to a destination of choice, the app may receive an unexpected response and behave irregularly.

- **Manipulate data structures**. An adversary manipulates a shared resource, *e.g.*, the crypto wallet of an app to perform illicit crypto transactions. Moreover, this major category is about categories that rely on the execution of native code and cause overflowed or overread buffers, or use any kind of malicious memory pointer manipulation.

- **Manipulate system resources**. An adversary performs software or hardware manipulations on a mobile device, *e.g.*, during the manufacture or distribution, to remotely control the device when it is used for crypto transactions. Such manipulations aim at the OS, application data, security configurations, or the infrastructure, to name a few.

- **Manipulate timing and state**. An adversary exploits application bugs to trigger a deadlock of the app or device to, for example, delay a crypto transaction or prolong the time available to steal a user's crypto wallet. Furthermore, this major category includes the manipulation of state, *e.g.*, the overriding of browser cookies or integrity checks that could reveal such adversarial actions.

- **Subvert access control**. An adversary tries to subvert access control features to steal the funds in a user's crypto wallet, *e.g.*, by using a malicious web browser plug-in that can perform an adversary in the middle (AiTM) attack to re-route crypto transactions. The adversary could further try to bypass broken authentication protocols.

9.2 Empirical Study

We first explain the used methodology before we report our findings.

9.2.1 Methodology

We read the description of each meta and standard category, *i.e.*, in total 192 different categories, and evaluated them for each of the 51 security smells that we identified in this work. Since not every meta category has a subordinate category, we consider such meta categories also as standard category to not miss any meta category in the subsequent analyses. In addition, we consider a detailed category a standard category, if and only if it resides in the same hierarchy level in the CAPEC taxonomy. Where a given category was lacking context to be fully understandable, we iterated over the peers and their descriptions to gather additional information. In the end, we evaluated 9 792 security smell and standard category combinations, and decided for each smell whether it is unlikely, neutral, or likely to enable a particular category. Since we had to perform discrete decisions to make it easier to abstract the data, we assigned the value zero to threats that are unlikely to emerge from a particular security smell, the value 0.5 to threats that we expect to emerge occasionally from a particular security smell, and the value one for a threat that is likely to emerge from a particular security smell. These decisions were taken objectively based on the descriptions of the categories and the security smells in the appropriate chapters. The entire dataset is available online.[3]

There are smell and standard category combinations, which are not trivial to decide. For example, on the one hand, we determined that the security smell "Unacknowledged distribution" is unlikely to have an impact on the category "Choosing message identifier" since the app package identifier should remain identical regardless whether an app is installed from the Play store, or side-loaded by the user. On the other hand, we determined the smell "Unauthorized intent receipt" likely to have an impact on that specific category, because the availability of an implicit intent receiver implies that other apps can choose a message identifier to access that receiver, which lacks authorization. 48 categories (25%) were unrelated to our work, *i.e.*, every security smell was determined to have unlikely an effect on all of these categories. Typical examples of unrelated categories are: *Physical theft*, *Bypassing physical locks*, or *Jamming*, *i.e.*, an adversary uses radio noise or signals in an attempt to disrupt communications. The reasoning for every combination may introduce threats to validity, and therefore a follow-up peer review is recommended to validate the following preliminary results.

[3]Figshare: evaluation of security smells against CAPEC, `https://figshare.com/s/deb1a1983b956677fa2d`, *accessed on 23-MAR-2022*

Domain	ID	Name	Abuse existing functionality	Collect and analyze information	Employ probabilistic techniques	Engage in deceptive interactions	Inject unexpected items	Manipulate data structures	Manipulate system resources	Manipulate timing and state	Subvert access control
Android	A01	Unreliable Information Source									
Android	A02	Untrustworthy Library									
Android	A03	Outdated Library									
Android	A04	Native Code									
Android	A05	Open to Piggybacking									
Android	A06	Unnecessary Permission									
Android	A07	Weak Crypto Algorithm									
Android	A08	Weak Crypto Configuration									
Android	A09	Unpinned Certificate									
Android	A10	Improper Certificate Validation									
Android	A11	Unacknowledged Distribution									
Android	A12	Unauthorised Intent Receipt									
Android	A13	Unconstrained ICC									
Android	A14	Unprotected Unix Domain Socket									
Android	A15	Exposed Adb-level Capability									
Android	A16	Debuggable Release									
Android	A17	Custom Scheme Channel									
Android	A18	Header Attachment									
Android	A19	Unique Hardware Identifier									
Android	A20	Exposed Clipboard									
Android	A21	Exposed Persistent Data									
Android	A22	Insecure Network Protocol									
Android	A23	Exposed Credential									
Android	A24	Data Residue									
Android	A25	XSS-like Code Injection									
Android	A26	Broken WebView's Sandbox									
Android	A27	Dynamic Code Loading									
Android	A28	SQL Injection									
Android ICC	B01	Persisted Dynamic Permission									
Android ICC	B02	Custom Scheme Channel									
Android ICC	B03	Incorrect Protection Level									
Android ICC	B04	Unauthorized Intent									
Android ICC	B05	Sticky Broadcast									
Android ICC	B06	Slack WebViewClient									
Android ICC	B07	Broken Service Permission									
Android ICC	B08	Insecure Path Permission									
Android ICC	B09	Broken Path Permission Precedence									
Android ICC	B10	Unprotected Broadcast Receiver									
Android ICC	B11	Implicit Pending Intent									
Android ICC	B12	Common Task Affinity									
Web comm.	C01	Credential Leak									
Web comm.	C02	Embedded Languages									
Web comm.	C03	Insecure Transport Channel									
Web comm.	C04	Disclosure of Source Code									
Web comm.	C05	Disclosure of Version Information									
Web comm.	C06	Lack of Access Control									
Web comm.	C07	Missing HTTPS Redirects									
Web comm.	C08	Missing HSTS									
HTTP clients	D01	Unsupported Connection Upgrades									
HTTP clients	D02	Abandoned HTTP Client									
HTTP clients	D03	Unsupported HSTS									
Ranking of major categories enabled by security smells			2	3	1	3	4	4	5	6	4

Figure 9.1: Security smells categorized by the CAPEC taxonomy

9.2.2 Findings

We present our findings in Figure 9.1, where the y-axis lists all of our security smells and the x-axis represents the CAPEC taxonomy. For each major category, we accumulated the values, *i.e.*, zero, 0.5, or one of all subordinate standard categories and divided them by their number to receive an average value, *i.e.*, a value in the range [0,1]. This linear value is then used to determine the red shade of the corresponding cell in the table, *i.e.*, white indicates no correlation, and bold red indicates maximal correlation. In other words, the colours only reflect correlation, but not frequency in practice. The bottom row presents the ranking from most (1) to least (6) impacted major CAPEC category and is based on the average value of the accumulated values in the corresponding column.

We can see that most affected are the major CAPEC categories "Employ probabilistic techniques," "Abuse existing functionality," "Collect and analyze information," and "Engage in deceptive interactions." Categories that relate to the employment of probabilistic techniques are primarily enabled by the security smells related to weak crypto algorithms or configurations, and by unreliable information sources as well as untrustworthy or outdated libraries, because the requirements for secure encryption may change over time. Moreover, exposed Adb-interfaces are major enablers of attacks related to data leaks, and attacks leveraging manipulated data structures seem to have more potential in apps that suffer from smells that relate to unreliable information sources and untrustworthy or outdated libraries, and similarly, attacks that engage in deceptive interactions especially seem to benefit from the task affinity security smell. On contrary, security smells barely introduce threats that leverage manipulated timing and state, and the smells regarding unique hardware identifiers, exposed clipboard content, and persisted dynamic permissions seem to favor only few categories.

9.3 Effective Security Measures

When we recap the findings of the previous section, we can observe a high correlation of some smells with selected major categories. Vice-versa, we can observe a high correlation of some major categories with selected smells. We argue that it is reasonable to evaluate strategies against major categories, because their subordinate categories do relate technically and thus we have an increased chance that one measure can prevent multiple standard categories. The four most affected major categories are responsible for more than 70% of the positive correlations. Therefore, we expect that remedies against these four major categories could greatly increase the mobile app security. We turn these major category titles into four questions that indicate our intended goals:

i) How can we prevent the use of probabilistic techniques?

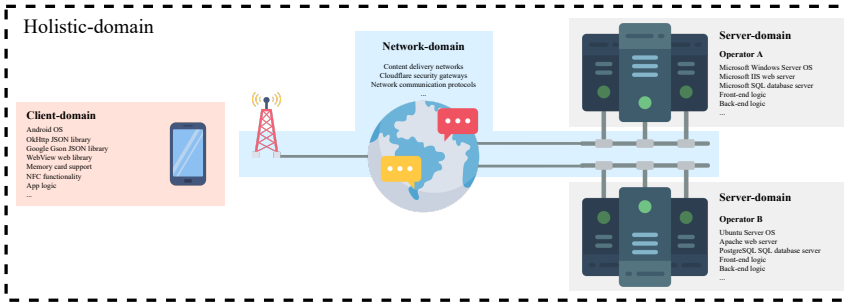


Figure 9.2: A holistic domain of mobile apps

ii) How can we prevent the abuse of existing functionality?

iii) How can we prevent the engagement in deceptive interactions?

iv) How can we prevent the collection and analysis of information?

Based on our observations during the data gathering for this thesis, we see for these four questions three effective remediation strategies:

i) **The use of secure default values**. Probabilistic techniques can be employed in unsafe configurations. Unsafe configurations can be a result of people that use the provided default values to make code just work with the least effort required. If a platform operator would consistently set secure default values, such issues would effectively be mitigated. This has already been proposed by researchers [80], however there still seem to exist problems in the Android ecosystem. *Examples of secure default values are discussed in the next chapter.*

ii) **The enforcement of safe practices**. Unsafe configurations, the abuse of existing functionality, and deceptive interactions can be a result of people that rely on flawed information from developer Q&A sites such as Stack Overflow or, for example, developers who access features that are not intended for them. If a platform operator would consistently enforce secure and easy to follow practices across the entire ecosystem, many of these issues could effectively be mitigated. Again, this security strategy has been proposed by researchers [28], however there still seem to prevail problems in the Android world. *Examples of safe practices are discussed in the next chapter.*

iii) **The use of smart data types**. According to our findings, sensitive data is usually stored as string. Therefore, if the String class would offer features to protect its value based on the accessing context, leaks could be prevented in many cases. *A smart data prototype is discussed in the next but one chapter.*

9.4 From Effective to Holistic Security Measures

Figure 9.2 shows the difference between security measures that target a traditional security-domain or measures that target, as we call it, a holistic-domain. Typical examples for security measures in the network-domain are firewall configurations, cloud security providers, *etc.* Typical security measures settled in the client-domain are OS settings, app updates, library updates, *etc.* Typical security measures settled in the server-domain are access control, certificate management, *etc.* Each of these traditional security measures increase the system security, however they seldom leverage synergies and although some of them are considered to provide "end-to-end" security, they do not. In fact, an encrypted communication channel may securely transmit data from the server to the client, but when the client receives the data it is immediately decrypted and can be intercepted by any involved library or method that is called in the process. Such a treatment curates an environment where data leaks can happen. Instead, holistic security takes the entire system into account, *i.e.*, the server, the network, and the client. Since all domains can share functionality, security measures can more effectively benefit from synergies. For example, if data is tagged as sensitive, the server, the network, and the client may behave appropriately and all together prevent such data from leaking.

9.5 The Conflict in Android OS Security

During our review of Android application framework APIs for the security smell investigations, we found that Android OS security has two main opposing forces, *i.e.*, changeability and compatibility as shown in Figure 9.3, resembling the well-known confidentiality, integrity, availability (CIA) triad. The triad illustrates that only one aspect can be optimized simultaneously, while the other two aspects will suffer the more the single aspect is preferred.

In other words, if the Android OS maintainers would particularly focus on security, continuous API changes would need to be prohibited and the compatibility with existing apps would be limited since they either conform with the higher security requirements, or fail to run. Moreover, if the Android OS is preferred to have the ability of continuous change, it will reduce app compatibility and security since apps from developers who do not continuously adopt the changes fail to run, and the continuous changes are likely to introduce new security vulnerabilities. Finally, if the Android OS has to maintain compatibility with existing code, it cannot leverage comprehensive code changes and thus it is likely that it also cannot evolve securely since, for example, existing apps may still require insecure crypto algorithms.

However, the Android OS is currently trying to offer the best of both worlds as it maintains compatibility with as many existing apps as possible,

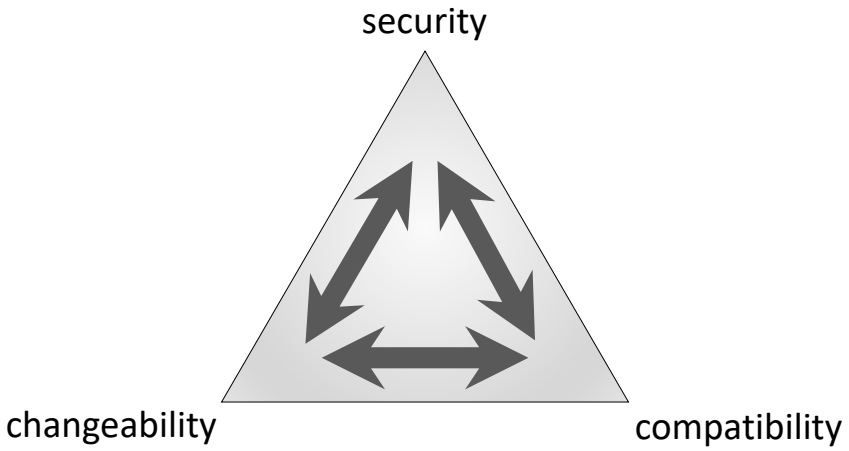


Figure 9.3: The triad of software security

however at the same time it fundamentally changes continuously. On the one hand, for example, the class `WebView` which is commonly used for the display of web content contains currently 25 deprecated elements, and the oldest was deprecated at API level 12, *i.e.*, Android "Honeycomb" that has been released in May-2011, which is more than ten years ago. On the other hand, Android application framework APIs fundamentally change every few months.[4,5,6] The resulting conflict severely impacts the security of the Android platform as security cannot be supreme without any sacrifices of the changeability and the compatibility property. In fact, a consequence of these contradicting requirements is that the development of any Android app is a very painful process since numerous compatibility targets must be dealt with, *i.e.*, code paths for different versions of the Android OS, which greatly contribute to code complexity and thus increase the risk of security smells. This increased risk is measureable: Zimperium assessed in their 2022 security report vulnerabilities across different mobile platforms and they consistently found more vulnerable apps on Android than on rivaling mobile platforms.[7]

[4]Google documentation: API changes between API level 29 and 30, `https://developer.android.com/sdk/api_diff/30/changes`, *accessed on 27-MAR-2022*

[5]Google documentation: API changes between API level 30 and 31, `https://developer.android.com/sdk/api_diff/31/changes`, *accessed on 27-MAR-2022*

[6]Google documentation: API changes between API level 31 and 32, `https://developer.android.com/sdk/api_diff/32/changes`, *accessed on 27-MAR-2022*

[7]Zimperium: Global Mobile Threat report 2022, `https://storage.pardot.com/66612/1646667936Ekpj8Pff/Zimperium_Global_Mobile_Threat_Report_2022.pdf`, *accessed on 27-MAR-2022*

9.6 Threats to Validity

Accuracy. A major threat represents the accuracy of the established dataset, because the data is the result of a careful manual review performed by the author. To mitigate this threat, the provided description and further information such as related threats, meta, standard, or detailed categories have been studied in detail if the purpose of a category was unclear. Moreover, he closely followed the description of the reported security smells to decide whether a smell can be a malefactor for certain categories. However, a follow-up peer review is recommended to further validate the preliminary results.

Correlation. Although we observed a high correlation between smells and some attacks, this does not necessarily mean that there is any practical relevance. To mitigate this threat, labeled data could be collected from CVE, however the assessment of such data remains future work.

Completeness. Another threat to validity is the completeness of the used classification scheme. He used CAPEC from Mitre, which is the largest public vulnerability database provider. Therefore, he expects that the established categorization in version 3.7 has come close to a stable state, and is reasonable to be used.

Selection bias. Based on the results of the smell classification, he identified four major areas that would massively benefit from improved security. However, what he has suggested is guided by the taxonomy proposed by Mitre, and thus there might exist other taxonomies, which could lead to another result.

Construct validity. There is a threat to construct validity through potential bias in his expectancy.

9.7 Conclusion

We observed that the consideration of security smells is time consuming and error prone, and therefore we are interested in finding more efficient holistic remedies. For that purpose, we manually investigated the impact of our 51 security smells on 192 attack mechanisms of the CAPEC taxonomy, which led to 9 792 combinations that we considered. We found that insecure algorithms, the abuse of existing functionality, data leaks, and user deception are the four major threats when using Android, and we elaborate strategies against them. That is, we see most potential in secure default values and safe practices to prevent feature misuse in the Android ecosystem. We further realized that string variables need increased protection, because they are responsible for most issues that relate to the employment of probabilistic methods and the injection of unexpected items.

Chapter 10

Default Values and Practices to Improve Application Security

Declaration of Content Reuse

The content of this chapter is partially based on the short paper *Security Header Fields in HTTP Clients* that has been accepted for the *21st IEEE International Conference on Software Quality, Reliability, and Security (QRS)* in 2021 [36].

In the previous chapter we established three measures that we expect could greatly improve mobile app security. We cover two of them in this chapter, *i.e.*, secure default values and safe practices.

A secure default value could be, for example, a boolean flag in a web API server that indicates whether secure communication channels should be preferred. If such a flag is enabled by default, every application that uses this server would immediately try to upgrade insecure communication channel requests to secure ones, *e.g.*, by forwarding the client to a HTTPS URL that points to the same resource. A safe practice differs from a secure default value in the sense that it does not leverage existing functionality that can be triggered with a simple boolean value, but instead it requires some additional logic or code changes. For example, a safe practice could be to disable the possibility to execute native code, or the removal of constructors that allow one to initialize unencrypted credential stores, which obviously represent a major risk.

These techniques are especially important as modern IDEs like Android Studio incorporate much knowledge about the Android application framework APIs and can propose default parameter values while a developer is writing code, *e.g.*, for method calls. Moreover, the implementation of Android application framework APIs determines how to treat missing

parameter values in methods used by apps, *e.g.*, whether they are allowed, and if they are, how they are set internally when not specified. Therefore, the selection of a default value can have a tremendous impact on app security, especially for novice developers who are not very familiar with every option and like to stay safe with the default. Besides default values, development practices are equally important, *e.g.*, the decision whether insecure APIs should be accessible for developers.

In this chapter, we shed light on different aspects based on the security smells that we previously established. In particular, we address the following research question:

RQ$_1$: *What are examples of default values and practices that could greatly improve application security?* We reviewed every reported security smell and reasoned whether it could be mitigated with improved platform security. We found that eight smells (16%) could be addressed with more secure default values, and that 36 smells (71%) could be addressed with safer practices. In fact, we only see for seven smells (14%) no potential in such measures, however they can be addressed using a better control of data and we present a potential solution to this problem in the following chapter.

In the remainder of this chapter, we discuss secure default values in section 10.1, and safe practices in section 10.2, before we report in section 10.3 the remaining security smells that have not been yet been considered in this chapter. We report the threats to validity in section 10.4, and conclude in section 10.5.

10.1 Secure Default Values

In general, wherever a developer is supposed to make a particular choice from a list of options, the option that exposes the least privilege should be proposed by default, or automatically used internally for calls that do not require a choice. For example, if a developer does not specify any access rights before reading an existing file from disk, it should be read-only to prevent accidental modifications. Nevertheless, it should still remain possible to modify the file when the access right is adjusted to provide read-write access, but then this change is requested explicitly and chances are much higher that a developer is aware of the potential implications.

10.1.1 Apps

With respect to apps, we identified one secure default value to address five security smells.

- **Android Permission System**. Android's permission system has already been discussed in section 2.1.4. The default protection level

of an Android app permission is `normal` and gives requesting applications access to isolated application-level features with minimal risk to other applications.[1] However, the default use of a `dangerous` protection level would prevent accidental data leaks since it has to be acknowledged explicitly, and could be occasionally revoked, if unused. The `signature` level permission should not be used by default as it does not require any user confirmation. *Relevant for: Unauthorized Intent Receipt, Unconstrained Inter-component Communication, Incorrect Protection Level, Unauthorized Intent, and Sticky Broadcast.*

10.1.2 App Servers

With respect to app servers, we identified three secure default values to address three security smells.

- **Disclosure of Source Code**. We discussed the disclosure of source code in subsection 6.3.2. The server side option to show error message output in the response may help during debugging, but it should certainly not be enabled on a production system. Therefore, if this feature would be disabled by default, lazy system administrators could not forget to disable this particular feature. In practice, the Apache Tomcat, *i.e.*, a widely used web app server offers the flag `showReport` that should be set to `false` by default to prevent stack traces in error pages.[2] *Relevant for: Disclosure of Source Code.*

- **Disclosure of Version Information**. We discussed the disclosure of version information in subsection 6.3.2. The server side option to include version information in the response may help during development, but it should certainly not be enabled on a production system. Therefore, if this feature would be disabled by default, lazy system administrators could not forget to disable this particular feature. In practice, the Microsoft internet information server (IIS) should be configured to not leak such information by setting in the `web.config` file the `enableVersionHeader` property to `false`.[3] *Relevant for: Disclosure of Version Information.*

- **Missing HTTPS redirects**. We discussed missing HTTPS redirects in subsection 6.3.2. The server side option to forward the client to the HTTPS version of a resource may complicate debugging, but

[1] Google documentation: permission element, `https://developer.android.com/guide/topics/manifest/permission-element#plevel`, *accessed on 27-MAR-2022*

[2] Apache Tomcat documentation: error report valve, `https://tomcat.apache.org/tomcat-9.0-doc/config/valve.html#Error_Report_Valve`, *accessed on 29-MAR-2022*

[3] Microsoft documentation: EnableVersionHeader property, `https://docs.microsoft.com/en-us/dotnet/api/system.web.configuration.httpruntimesection.enableversionheader`, *accessed on 29-MAR-2022*

it should certainly be enabled on a production system. Therefore, if this feature would be enabled by default, lazy system administrators could not forget to enable this particular feature. In practice, the URL rewrite module of Microsoft's IIS should by default contain a redirect rule that will transform every HTTP request into an equivalent HTTPS request.[4] *Relevant for: Missing HTTPS Redirects.*

10.2 Safe Practices

Safe practices that we propose in this section are supposed to prevent developers from making mistakes. However, if developers intentionally neglect to follow safe practices, they should be motivated to change their minds, *e.g.*, with additional badges shown on their apps' overview websites of the app stores, or they should earn more profit from their app sales. The practices we present in this section primarily comprise new IDE features, a revised public Android Java API, and improved methodologies to cope with Java API code changes. For example, it is important to remove deprecated Java APIs in a timely fashion by leveraging short grace periods, *e.g.*, at most few OS release cycles.

10.2.1 Apps

With respect to apps, we identified 22 safe practices to address 35 security smells.

- **Code Black-listing**. Some code snippets may suffer from well-known vulnerabilities as discussed in subsection 4.1.1, however developers are not forced to replace them. In Android Studio, a solution should involve the black-listing of well-known vulnerable code, *e.g.*, that can be found on Stack Overflow. *Relevant for: Unreliable Information Source.*

- **Code Obfuscation**. Continuously changing code obfuscation mechanisms increases the difficulty of reverse engineering apps as discussed in subsection 4.1.1, and therefore should be used by default. This measure would make it more difficult for adversaries to piggyback existing apps. *Relevant for: Open to Piggybacking.*

- **Custom Schemes**. Custom schemes also known as "deep links" required for URLs that can request certain views of a particular app are well-known to introduce various threats as discussed in subsection 4.1.3. However, they still exist in the Android ecosystem although a more secure implementation is available, which is called

[4] Microsoft documentation: using the URL rewrite module, `https://docs.microsoft.com/en-us/iis/extensions/url-rewrite-module/using-the-url-rewrite-module`, *accessed on 29-MAR-2022*

"Android App Links"[5] that relies on a certificate to validate the scheme ownership of an app, which cannot be forget easily. Therefore, the concept of deep links should be abandoned entirely. *Relevant for: Custom Scheme Channel (Android and Android ICC).*

- **Dynamic Code Loading**. The dynamic code loading API provides flexibility in programming as discussed in subsection 4.1.2 and subsection 4.1.5, however it is risky to use since it allows the execution of arbitrary code, which in turn can introduce various security vulnerabilities. Therefore, this interface should be disabled and developers should be forced to solely rely on the regular app store mechanisms to update their apps. Moreover, the side-loading of apps should be prevented for regular end user devices, and if side-loaded apps are still installed, they should be screened and updated by Google Play, which is currently the case.[6] *Relevant for: Unacknowledged Distribution, and Dynamic Code Loading.*

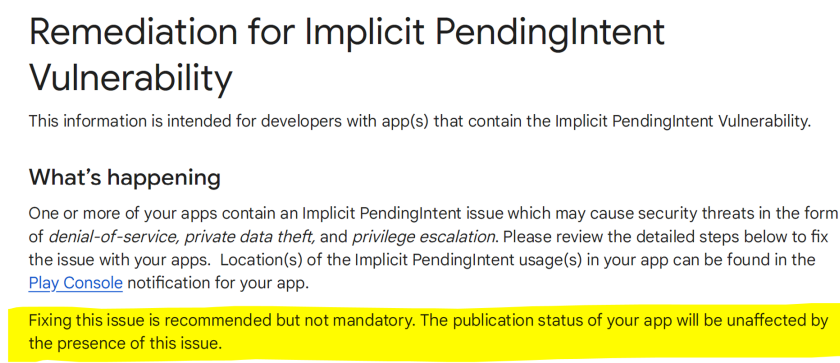
- **Dynamic Permissions**. Dynamic permissions allow the exposure of resources until the permission is revoked as discussed in section 5.1. However, developers may forget to revoke resources and thus expose resources longer than desired. Therefore, the API should be revised so that it does not require a revoke call, but instead use a timeout or an integer that defines how many times the resource can be accessed before it is revoked automatically. Therefore, developers would not have to mind revoke calls. *Relevant for: Persisted Dynamic Permission.*

- **Exclude Apps From App Stores**. Apps that perform risky or unauthorized operations as discussed in subsection 4.1.3 should be removed from the store and eventually from the devices, *e.g.*, when Unix domain sockets are accessed instead of Android APIs, when Adb sockets are exposed, when the debug mode is enabled, or when the OS terminal is exposed. Currently, Google does not consistently remove apps that suffer from well-known vulnerabilities, *e.g.*, the use of insecure implicit pending intent configurations as shown in Figure 10.1.[7] *Relevant for: Unprotected Unix Domain Socket, Exposed Adb-level Capability, and Debuggable Release.*

[5] Google documentation: handling Android App Links, `https://developer.android.com/training/app-links/`, *accessed on 27-MAR-2022*

[6] Google enterprise help: why does Play automatically..., `https://support.google.com/work/android/thread/67460799/why-does-play-automatically-update-sideloaded-emm-installed-apps-that-have-matching-bundle-ids?hl=en`, *accessed on 27-MAR-2022*

[7] Google documentation: remediation for implicit PendingIntent vulnerability, `https://support.google.com/faqs/answer/10437428?hl=en`, *accessed on 27-MAR-2022*



Remediation for Implicit PendingIntent Vulnerability

This information is intended for developers with app(s) that contain the Implicit PendingIntent Vulnerability.

What's happening

One or more of your apps contain an Implicit PendingIntent issue which may cause security threats in the form of *denial-of-service, private data theft,* and *privilege escalation*. Please review the detailed steps below to fix the issue with your apps. Location(s) of the Implicit PendingIntent usage(s) in your app can be found in the Play Console notification for your app.

Fixing this issue is recommended but not mandatory. The publication status of your app will be unaffected by the presence of this issue.

Figure 10.1: A typical vulnerability not considered for apps in the Google Play store

- **Exposed Java Code**. As discussed in subsection 4.1.5, the JavaScript API provides flexibility in programming, however it is risky to use since it exposes app code to the web, which is well-known to suffer from code injection attacks. Therefore, this interface should be disabled and developers should be forced to solely use Java or Kotlin code to build their apps. *Relevant for: Broken WebView's Sandbox.*

- **Exposed Private Storage**. As discussed in subsection 4.1.4, private storage can be exposed by using distinct parameters to grant access rights to the "world," *i.e.*, every other app on the device. However, if data needs to be shared it should not remain in private but public storage, and optionally leverage data encryption techniques. Therefore, this API should be erradicated. *Relevant for: Exposed Persistent Data.*

- **HTTPS Certificates**. As discussed in subsection 4.1.2, HTTPS certificates are nowadays available for free and, consequently, the pinning of HTTPS certificates should be enforced by default in combination with the disabling of custom validation mechanisms to bypass the certificate validation process. These measures could effectively prevent attackers from performing HTTP downgrade or certificate manipulation attacks. *Relevant for: Unpinned Certificate, and Improper Certificate Validation.*

- **Implicit Pending Intent**. Implicit pending intents can be altered by other apps and thus mislead subsequent accessors as discussed in section 5.1. Meanwhile Android introduced a flag `FLAG_IMMUTABLE` to make them immutable, however its use is still optional. Since explicit pending intents offer a more secure addressing scheme, but otherwise offer the same features, support for implicit pending intents should be removed. *Relevant for: Implicit Pending Intent.*

- **Insecure Network Protocol**. We already discussed insecure network communication in subsection 4.1.4, subsection 6.3.1, subsection 6.3.2, and subsection 8.2.3. The communication over HTTP is insecure and should be disabled by default. In the meantime, this measure has already been implemented by Google.[8] Moreover, the HTTP headers `Strict-Transport-Security` and `Upgrade` should be supported out of the box in major HTTP clients. *Relevant for: Insecure Network Protocol, Insecure Transport Channel, Missing HSTS, Unsupported Connection Upgrades, and Unsupported HSTS.*

- **Library White-listing**. As discussed in subsection 4.1.1 and subsection 8.2.3, some libraries may suffer from well-known vulnerabilities, however developers are not forced to upgrade them. In Android Studio, a solution could be a white-listing of recent libraries that are considered to be safe. *Relevant for: Untrustworthy Library, Outdated Library, and Abandoned HTTP Client.*

- **Native Code**. Native code is used by numerous apps to decrease the time required for complex computations. As discussed in subsection 4.1.1, since it is simpler to investigate more abstract managed code, native code should be prohibited, and instead frequently used native libraries integrated into the Android system so that they can be accessed straight from the Java framework. If custom machine code is still required, GPU acceleration APIs or the more secure Rust programming language should be used instead, which are more constrained.[9] Rust is a system programming language designed with performance and memory safety in mind to replace traditional C/C++ code. *Relevant for: Native Code.*

- **Path Permissions**. As discussed in section 5.1, path permissions are supposed to protect resources from unauthorized accesses, however their validation is flawed, *i.e.*, the parsing and precedence of paths is unexpected. Therefore, the parsing should be more robust and the path precedence feature abandoned as it can be replaced using more secure mechanisms, *e.g.*, a different path permission for every sub path. *Relevant for: Insecure Path Permission, and Broken Path Permission Precedence.*

- **Permission Clean up**. As discussed in subsection 4.1.1, unnecessary permissions can be exploited by attackers and therefore they

[8] Google documentation: opt out of cleartext traffic, `https://developer.android.com/training/articles/security-config#CleartextTrafficPermitted`, *accessed on 27-MAR-2022*

[9] Google security blog: Rust in the Android platform, `https://security.googleblog.com/2021/04/rust-in-android-platform.html`, *accessed on 27-MAR-2022*

should be reported by the IDE and removed before the release of an app. *Relevant for: Unnecessary Permission.*

- **Secure Authorization**. As discussed in subsection 6.3.1, hard-coded passwords should not be allowed in app code. Instead, credentials should be requested from the user during run time and used to dynamically acquire a token, before such information must be stored in a secure key store. *Relevant for: Credential Leak.*

- **Secure Key Store**. As discussed in subsection 4.1.4, Android's KeyStore API allows one to store key-value pairs, however the provided information is not necessarily encrypted since several methods allow the use of `null` as key parameter to request unencrypted key stores.[10] Such code paths should be removed so that every use of KeyStore results in an encrypted key store. *Relevant for: Exposed Credential.*

- **Service Permission**. In Android, there exist validation methods that depend on the execution context, and therefore behave differently in different app code locations as discussed in subsection 5.1.2. Such methods should not be exposed publicly since they introduce confusion among developers and thus can lead to vulnerabilities. *Relevant for: Broken Service Permission.*

- **Sticky Broadcast**. As discussed in subsection 5.1.2, other apps may tamper with a sticky broadcast. Therefore, the sticky broadcast should either be immutable for other apps, or, preferably, the feature be removed entirely since there exist other mechanisms to work around such a problem, *e.g.*, a non-sticky broadcast. *Relevant for: Sticky Broadcast.*

- **Task Affinity**. As discussed in subsection 5.1.2, the task affinity feature allows other apps to show overlays if they know the name of the used affinity. The used default value is the package name,[11] which is publicly available and therefore should be replaced with a random value, or even better, the entire task affinity implementation in the Android application framework should be protected with fine-grained permissions that can be revoked by the user at any time. *Relevant for: Common Task Affinity.*

- **Weak Crypto**. As discussed in subsection 4.1.2, APIs related to weak crypto algorithms and configurations can introduce major vulnerabilities. Therefore, they should be removed entirely so that

[10] Android documentation: KeyStore, `https://developer.android.com/reference/java/security/KeyStore`, *accessed on 27-MAR-2022*

[11] Google documentation: activity, `https://developer.android.com/guide/topics/manifest/activity-element#aff`, *accessed on 27-MAR-2022*

developers cannot anymore use them. *Relevant for: Weak Crypto Algorithm, and Weak Crypto Configuration.*

- **WebViewClient**. Arbitrary URLs can be accessed when using a `WebViewClient` instance. If such an instance would by default be restricted to a particular domain as discussed in section 5.1, it could effectively prevent arbitrary unsafe URLs from being accessed. *Relevant for: Slack WebViewClient.*

10.2.2 App Servers

With respect to app servers, we identified one safe practice to address one security smell.

- **Lack of Access Control**. As discussed in subsection 6.3.2, sensitive resources must always be protected by some form of authentication and authorization. Such validation measures should preferably rely on existing standards such as OAuth. *Relevant for: Lack of Access Control.*

10.3 Remaining Security Smells

There are seven remaining smells that we did not yet discuss, however all of them rely on plain text that can be analyzed, *e.g.*, the smell SQL injection relies on strings that contain fragments of a SQL query, which can be detected with regular expressions. Although secure default values or safe practices do not seem to be an option, we expect measures that operate on plain text, *i.e.*, strings can have an impact on them. We discuss a potential string-driven approach in the next chapter. *Relevant for: Header Attachment, Unique Hardware Identifier, Exposed Clipboard, Data Residue, XSS-like Code Injection, SQL Injection, and Embedded Languages.*

10.4 Threats to Validity

Lack of validation. The foremost threat to validity is the lack of validation. We could not implement and evaluate the proposed measures in the Android ecosystem and therefore the expected security gain remains speculation. However, the strategy to utilize secure parameters and to remove or replace insecure features is commonly used to improve overall security.

Completeness. Another major threat to validity of this study is the selection bias, *i.e.*, whether the proposed measures are comprehensive. We strived to identify effective measures that serve the purpose, and therefore the scope of evaluated measures does not necessarily have to be investigated entirely.

Construct validity. There is a threat to construct validity through potential bias in his expectancy.

10.5 Conclusion

We reviewed every reported security smell and found that eight smells (16%) could be addressed with more secure default values, and that 36 smells (71%) could be addressed with safer practices. In fact, we only see for seven smells (14%) no potential in such measures, however they can be addressed using a better control of data. We investigate a prototype that offers better control over data in the next chapter.

Chapter 11

A String-based Framework to Improve Application Security

Declaration of Content Reuse

The content of this chapter is based on the MSc thesis *BString: A String-based Framework to Improve Application Security* from Christian Zürcher [122], which was supervised by the author.

The Java String API does not provide any particular method related to security, *i.e.*, there is no method to validate the assigned data or to prevent data leakage. For example, there exists no embedded facility that can prevent email addresses and passwords stored in text strings from leaking to the console or to a log file. If additional protection is required, developers could, on the one hand, consider using additional checks at critical locations. Unfortunately, the thorough manual implementation of such checks is very error prone and introduces code duplication, which further complicates the problem. On the other hand, there exist static code analysis tools that cannot access problematic run time data, and there exist dynamic analysis tools which are rather limited and usually focus on a single security threat, *e.g.*, they can only perform variable tainting to determine potential data leaks. In other words, there is no comprehensive solution that can prevent sensitive data from leaking *and* remote code execution attacks, although these are among the top three major web application threats in 2021 according to the OWASP project.[1]

In this chapter, we present a framework called *BString* that offers a String class, which can react depending on the contained value. In consequence, the behavior of an application will adjust depending on the

[1] Open Web Application Security Project (OWASP): top 10 web application security risks, `https://owasp.org/www-project-top-ten`, *accessed on 02-MAR-2022*

contained String values and their locations. For this purpose, we introduced two additional methods in the *OpenJDK* Java String class implementation, which accept compiled byte code as an input parameter. The provided code will be executed before every read, respectively before every write on the object. For example, if such a String object's value is requested from within a `FileWriter` instance, it can either block the request and raise an exception, grant and log the request, or return a safe replacement value that indicates the need for protection. The use of such strings has only a minor average performance impact on accesses, of less than 16%, is completely optional and, if not used, it does not alter the behavior of existing code.

We investigate the following two research questions:

RQ$_1$: *What are the restrictions when using an instrumented Java String class with existing code?* We used an instrumented String instance as parameter in the API calls of the thirteen most popular Java libraries in the Maven repository, and we evaluated whether the behavior of the instrumented String was executed. If our code was not executed, we investigated the root cause. We found that only two libraries were not compatible with *BString*, *i.e.*, the *Gson* library, which used reflection to access String values, and the *SLF4J* library, which used a not instrumented custom byte buffer implementation for text content.

RQ$_2$: *Can an instrumented String class offer protection against data leaks and remote code execution, and what are the security risks using such a technique?* Using *BString*, we implemented several remediation strategies against the two major threats, data leak and remote code execution, that have been presented in existing literature. Moreover, we briefly summarize common threats that may arise when misusing *BString*. We found that this technique can protect particularly well against data leaks, *i.e.*, it can encrypt data on demand or restrict access for certain classes without the need for changing existing code in the used libraries. Furthermore, the provided interface supports various other security-related tasks, *e.g.*, data verification and validation.

In summary, this chapter investigates the utility of injecting arbitrary code into String instances to leverage additional functionality within Java applications. Our evaluation with commonly used Java libraries showed promising results for security-related tasks and moreover, we expect that this concept can further be helpful in the domain of, for example, logging and debugging.

The remainder of this chapter is organized as follows. We discuss the implementation in section 11.1, and we present the restrictions of *BString* in section 11.2. We show how *BString* can contribute to security in section 11.3. We recap the threats to validity in section 11.4, and finally, we conclude this chapter in section 11.5.

11.1 Prototype

We start with a motivating example to describe the core idea, before we elaborate on our prototype, *i.e.*, its implementation, the features, and the achievable performance.

11.1.1 Motivating Example

A developer has to implement an application that receives a user password from a web service and then securely stores it within an encrypted database. Moreover, the plain password must never be logged, stored to disk, or leaked through a network socket, and it must not contain any special characters that could enable remote code execution (RCE) attacks.

To comply with the requirements, a continuous monitoring of all variables that get directly or indirectly "in touch" with the password value would be required since the password string can be concatenated with multiple other strings, for example, when a random salt value is added which increases resilience against brute-force and rainbow table attacks, *i.e.*, attacks that leverage a large pre-computed table of hashed passwords so that efficient search algorithms can be used to identify a password candidate, before it reaches the destination. Therefore, a developer would currently require at least two different tools to solve this task: i) a dynamic analysis framework that can trace variables during run time, and ii) a library or manual code that will check the variables for consistency before they are accessed. Besides the high complexity of using multiple tools for this particular task, it is difficult to reuse such code and configuration rules across different projects due to the additional dependencies introduced by the tools.

In contrast, *BString* encourages developers to separate traditional application logic from String validation logic that can easily be reused and maintained across different projects. In Listing 5, we show an excerpt of an interface implementation that can protect a password value, which, using *BString*, can be attached to one or more String class instances. In particular, the implementation ensures that no data can leak through `java.net` network, `java.io` disk, and `java.system.out` console classes (lines two to nine), and at the same time it checks whether the string value contains only safe characters (lines eleven to fourteen) to prevent potential RCE attacks. Moreover, the attached interface implementation of a String instance automatically can be reused by every other String instance that is involved in a shared operation, *e.g.*, when they get "in touch" because of concatenation. In short, the presented code in Listing 5 can offer several benefits: i) it separates security-related concerns from traditional code, ii) it prevents duplicated validation logic being scattered across different classes, iii) it reduces the overall project complexity by centralizing code and by making several analysis libraries and frameworks obsolete, iv) it can

be reused across different projects, and finally, v) security-related changes are immediately visible when using a versioning system.

```
1  public String applyOnRead(String s) {
2          leakyClasses = "java.net", "java.io", "java.system.out";
3          stackTraceElements = Thread.currentThread().getStackTrace();
4
5          For each fully-qualified class name n in stackTraceElements do {
6                  if(n.startsWithOneOf(leakyClasses)){
7                          throw new LeakException("Data leak identified in
       ↪ class " + n);
8                  }
9          }
10
11         allowedCharacters = "A-Z", "a-z", "0-9", "-", ".", ",";
12         if (!s.containsOnly(allowedCharacters) {
13                 throw new RCEException("The provided text contains unsafe
       ↪ characters.");
14         }
15
16         return s;
17 }
```

Listing 5: Pseudo-code that illustrates the use of *BString* against data leaks and RCE attacks

11.1.2 Implementation

We only modified the built-in Java classes of the OpenJDK VM to maintain compatibility across different platforms. In particular, we modified besides the Java `String` class three more Java base classes, which are used by the Java implementation when a developer concatenates multiple strings, *e.g.*, by using the + operator: `StringBuilder`, `StringBuffer`, and `AbstractStringBuilder`. In particular, we were interested in intercepting all methods of the String class that internally operate on the String value using a `byte[]` or `this` reference, *i.e.*, `equals(Object)`, `getBytes()`, `charAt(int)`, `substring(int, int)`, `matches(String regex)`, and `toString()`. We check for each of these public methods whether there is a behavior attributed to the String object and act accordingly.

We define the term "behavior" as the required code that a user must provide to make use of *BString*, *i.e.*, an implementation of the `IStringBehavior` interface.

The intercepting code must follow the convention of the `IStringBehavior` interface shown in Listing 6 that we have added to the package `java.lang`. This interface describes five methods that can be implemented: i) `applyOnCreation(...)` that holds the code that is executed before the initialization of a String instance, ii) `applyOnRead(...)` that holds the code that is executed before a value is leaked, iii) `applyDerivationRule(...)` that specifies when the provided code should be attached

to a derived String instance, iv) `recordHistory()` that allows one to access each String transformation that occurs in the lifetime of a String, and finally, v) `getDescription()` that returns a textual description of the provided logic. Moreover, we provide the class `StringNotMatchingBehaviorException` to indicate an error state that can be raised by the developer, *e.g.*, if a password is about to be leaked, *etc.*

```
1 public interface IStringBehavior {
2     public String applyOnCreation(String s);
3     public String applyOnRead(String s);
4     public boolean applyDerivationRule(DerivationRule dr);
5     public boolean recordHistory();
6     public String getDescription();
7 }
```

Listing 6: Methods of the interface `IStringBehavior`

Whether a user has to use `applyOnCreation(...)`, `applyOnRead(...)`, or even both methods simultaneously depends on the task. If such a method is not implemented, the string with behavior will act like any regular non-modified string. If such a method is implemented, it must return a string that will be further used throughout the application instead of the original value. Four examples are listed in Listing 7: line two returns the original string and therefore does not alter the behavior, line three returns the string "NewString," line four returns a protected string encrypted by a custom method, and finally, line five throws the run time exception `StringNotMatchingBehaviorException` to block the current execution of the application.

```
1 public String applyOnRead(String s) {
2     return s;
3     return "NewString";
4     return EncryptionMechanism.encrypt(s);
5     throw new StringNotMatchingBehaviorException();
6 }
```

Listing 7: Behavior examples

We illustrate the signalling between the different methods when the `applyOn*(...)` APIs are used in Figure 11.1. First, we create a String object and then we add some behavior to it, *i.e.*, an implementation of the `IStringBehavior` interface. This implementation contains the relevant logic and remains active when the String is later accessed with, for example, the `toString()` method. Besides the manual use of such String objects, we further provide the class `StringBehaviorController` that can programmatically assign a behavior to selected String instances, *e.g.*, inside a specified package, method, or class. This class is particularly important when working with closed-source code, however it only works when the support for modules is disabled in the JDK.

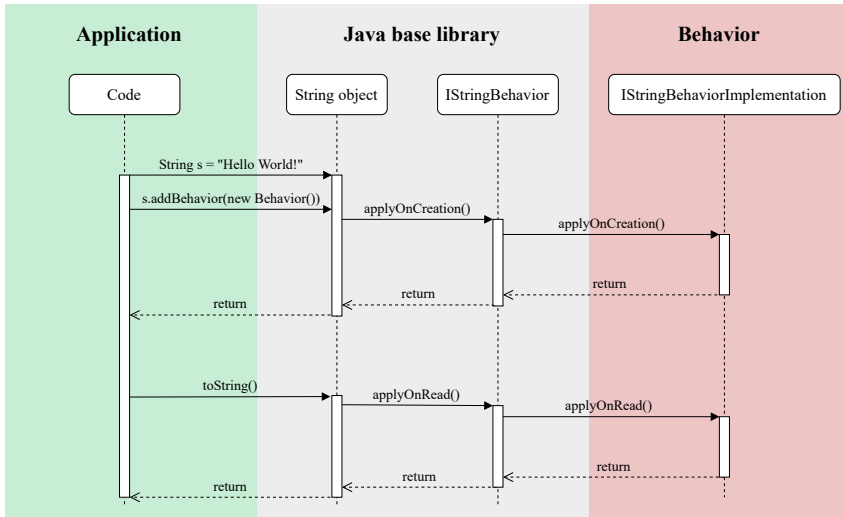


Figure 11.1: Message flow between software components

11.1.3 Features

The implementation adds three major features to the String class, *i.e.*, mutable strings, derivation of string behaviors, and a value history.

Mutable Strings

Strings are immutable in the Java VM and a change in a String value will always return a new, derived String instance. However, we can use the `applyOnCreation(...)` method to set a different value of the derived String instance during its initialization by accessing the internal fields `byte[] value` and `byte coder`, or we could at any time change the returned value of the string with the help of a behavior, *e.g.*, to prevent data leaks.

We present two examples that illustrate the use and usefulness of a mutated string value:

- Listing 8 shows the code that is required to add a prefix to a String of which the prefix even can be altered during run time to, for example, inform users that the application is a debug release that should not be used for production.

```
1 public static String prefix = "DebugRelease:";
2 public String applyOnRead(String s) {
3     return prefix + s;
4 }
```

Listing 8: Read behavior that adds a prefix

- Listing 9 shows the code that is required to escape all apostrophes inside a string to prevent SQL injection attacks.[2] Only the resulting escaped string value will be saved into memory.

```
1 public String applyOnCreation(String s) {
2     return s.replaceAll("[0\t\n\r\"%\'_\\\\]", "''");
3 }
```

Listing 9: Initialization behavior to prevent SQL injection attacks

Derivation of String Behaviors

A user can precisely control to which derived String instances a behavior is attached with the method `applyDerivationRule(...)`, which takes a `DerivationRule` enumeration as argument. Four different derivation rules are available: i) `COPY` only attaches the behavior to the derived String if it is a copy of the original String, ii) `ADD` only attaches the behavior to the derived String if it is the result of a concatenation from the original String with another String, iii) `DELETE` only attaches the behavior to the derived String if it is the original String with one or more removed characters, and iv) `REPLACE` only attaches the behavior to the derived String if at least one character is altered from the original String. The derivation rules are useful if certain string operations reduce the sensitivity of information, *e.g.*, a credit card number from which numbers were removed, might require less protection since the shortened numbers may not anymore unique.

Listing 10 shows typical examples of the different derivation rules. In line one we copy a string to another string, which the framework will consider a COPY operation. In line two we add a character to an existing string, which will be classified as ADD operation. In line three we derive a new string without the first three characters, which will be classified as DELETE operation. Finally, in line four we replace the letter “a” with “b” in a string, which will be classified as REPLACE operation. We distinguish the REPLACE operation although it could be seen as a mixture of add and delete operations, because it enables a developer to react more easily on common changes.

```
1 String derivative = original;                    // derivation rule: COPY
2 String derivative = original + "!";              // derivation rule: ADD
3 String derivative = original.substring(3);       // derivation rule: DELETE
4 String derivative = original.replace('a','b');   // derivation rule: REPLACE
```

Listing 10: Typical examples of different derivation rules

We show the use of such derivation rules in Listing 11. During run time, the provided variable `dr` in line four holds the detected string operation

[2] Stack Overflow: What characters have to be escaped..., `https://stackoverflow.com/questions/1086918/what-characters-have-to-be-escaped-to-prevent-mysql-injections`, *accessed on 28-MAR-2022*

by *BString*. For whatever string operation a developer returns true, the behavior will be derived to the resulting String instance. For example, in line five we specify that the behavior must be attached to derived String instances if they are copied or include additional characters, but not in any other case. In other words, if we want to always or never attach a behavior to derived String instances, we can simply return `true` or `false`, respectively.

```
1 public class BehaviorWithDerivationRule implements IStringBehavior {
2     ...
3     @Override
4     public boolean applyDerivationRule(DerivationRule dr) {
5         return (dr.equals(COPY) || dr.equals(ADD));
6     }
7     ...
8 }
```

Listing 11: Use of derivation rules

Listing 12 shows the interplay between a behavior and the derivation rules using more complex examples together with the behavior from Listing 11. We initially assign a behavior to the String instance `s1` (line one). The behavior will be derived from `s1` to `s2` (COPY, line two) and `s3` (ADD, line three) since both derivation rules matched the specified condition in the used behavior. However, all the other strings will remain without any attached behavior, *i.e.*, `s4`, `s5`, `s6`, and both instances inside `arr`.

```
1 String s1 = new String("Hello World", new BehaviorWithDerivationRule());
2 String s2 = new String(s1);     // derivation rule: COPY
3 String s3 = s1 + "!";           // derivation rule: ADD
4 String s4 = s1.substring(6);    // derivation rule: DELETE
5 String s5 = s1.toLowerCase();   // derivation rule: REPLACE
6 String s6 = s1.toUpperCase();   // derivation rule: REPLACE
7 String[] arr = s1.split(" ");   // derivation rule: DELETE
```

Listing 12: Interplay between a behavior and the derivation rules

Value History

BString can record the value of every instrumented String object and generate a tree representation of their lineages. In more detail, the String history node class `SHNode` represents a node of a tree that reflects the tracked string transformations, *e.g.*, when two strings are concatenated the resulting string will contain a tree with at least two nodes, *i.e.*, a parent node that points to one or more child nodes, which refer to the originating strings. To use this feature, the implementation of the method `recordHistory()` must return `true` and then the developer can access the tree through the `SHNode` variable in the behavior interface. With this feature a user can track the changes that have been applied over time to a particular String, and act accordingly.

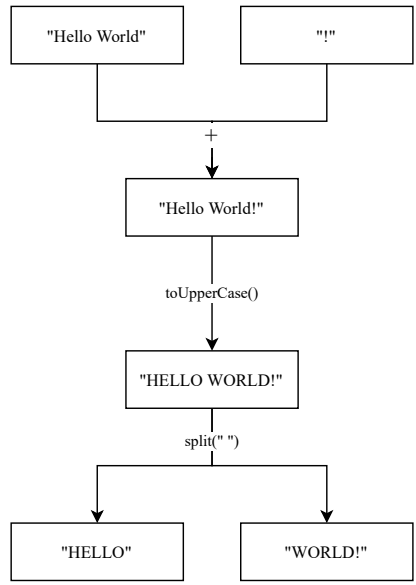


Figure 11.2: The resulting value history tree for Listing 13

Listing 13 illustrates the use of the value history feature. The content of the String `s1` is reused several times before the final result arrives in the String variables in `arr`. The corresponding value history tree is shown in Figure 11.2. First, we perform an ADD operation on "Hello World" and "!", before we perform a REPLACE operation with `toUpperCase()`. Next, we perform a DELETE operation that splits the string into the two distinct strings "Hello" and "World." For the original and every derived String instance a new history value node is attached to their existing tree after each operation, which we can then access with the method `String.getHistoryNode()`. We can further navigate the tree by calling the `getParents()`, `getChildren()`, and `getValue()` methods of `SHNode`. Alternatively, the tree can be inspected by any debugger. In Figure 11.3, we show a screenshot of the debugger available in Visual Studio Code[3] that currently inspects the variable `historyNodeOfS1` at the end of the execution (line six in Listing 13).

```
1 String s1 = new String("Hello World", new HistoryRecordBehavior());
2 String s2 = s1 + "!";
3 String s3 = s2.toUpperCase();
4 String[] arr = s3.split(" ");
5 SHNode historyNodeOfS1 = s1.getHistoryNode();
6 SHNode historyNodeOfArr = arr[1].getHistoryNode();
```

Listing 13: Use of the value history feature

[3]Visual Studio Code Website, `https://code.visualstudio.com`, *accessed on 06-MAR-2022*

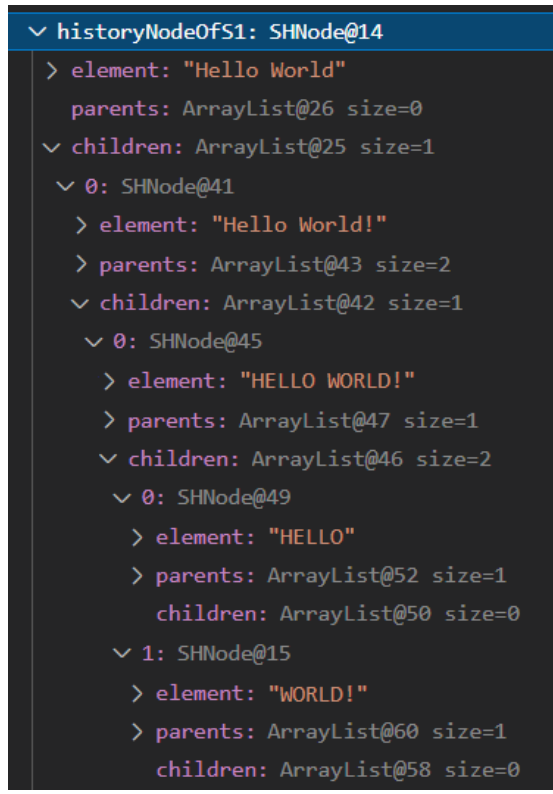


Figure 11.3: Value history tree visualized from a debugger

11.1.4 Application Support

BString supports three different kinds of software:

- **Open-source applications.** If the source-code of an application is available, a user can implement the `IStringBehavior` interface to specify the desired behavior and either assign it to individual strings with the method `String.setBehavior(IStringBehavior)` or assign it more generally with the provided class `StringBehaviorController` to, for example, all strings within a specified class. In more detail, the class `StringBehaviorController` supports the methods `addMethodBehavior(...)`, `addClassBehavior(...)`, and `addPackageBehavior(...)`, which all require a string parameter that either define the relevant method (*e.g.*, `some.package.class.Method`), the relevant class (*e.g.*, `some.package.class`), or the relevant package (*e.g.*, `some.package`), and an instance of the behavior that should be applied. This will force all String constructors inside the defined scope to execute the `applyOnCreation()` method of the be-

havior. In cases where method, class and package behavior conflict, the method behavior is prioritized, then the class behavior and finally the package behavior.

- **Closed-source libraries.** A user can add custom String behavior to compiled libraries. The process is the same as for open-source applications, *i.e.*, a String with behavior must be created and then used as parameter in public library methods.

- **Closed-source applications.** The process to instrument compiled code that is packaged as a runnable `.jar` file or as class files is more complex: the interface implementation must be compiled manually, and then referenced in an `.xml` file that must be parameterized at the start of the VM, which then will load the configuration from disk. An example of such a configuration file is shown in Listing 14 where the root tag `<behaviors>` allows one to specify one or more `IStringBehavior` implementations that will be used in the application. Each behavior is enclosed by a `<behavior>` tag that must contain the name of the behavior including the package name if necessary, and where it applies to. We can define each package, class, or method in which the String instances must use the provided behavior within the behavior's `<applyTo>` tag. The use of such a configuration file will force all String constructors within the defined scope to execute the `applyOnCreation(...)` method of the specified behavior. Whenever a conflict arises between different behaviors, package-level behaviors will have the least priority and method-level behaviors will have the highest priority.

```
<?xml version="1.0">
<behaviors>
    <behavior>
        <name>SomeBehaviorClass</name>
        <applyTo>
            <package>name.of.package</package>
            <class>name.of.other.package.Class</class>
            <method>name.of.package.Class.method1</method>
            <method>name.of.package.Class.method2</method>
        </applyTo>
    </behavior>
    <behavior>
        ...
    </behavior>
    ...
</behaviors>
```

Listing 14: Typical configuration file for closed-source application analyses

	initialization	read	derivation
baseline	4 ms	166 ms	11 ms
without behavior	13 ms	171 ms	20 ms
with empty behavior	16 ms	193 ms	22 ms
with derivation rules	16 ms	191 ms	32 ms
with history	59 ms	213 ms	87 ms
with encrypt on init	2 388 ms	9 331 ms	19 ms
encrypt on init and decrypt on read	5 112 ms	7 982 ms	timeout

Table 11.1: String performance evaluation

11.1.5 Performance

To assess the performance of the implementation, we instantiated one million String objects with random values using seven different configurations and measured the required time: i) a vanilla JDK without any changes in the String class, ii) a custom JDK where we used no behaviors, iii) a custom JDK where we used empty behavior stubs, iv) a custom JDK where we used empty behavior stubs and the behavior derivation feature, v) a custom JDK where we used empty behavior stubs, the behavior derivation feature, and the history feature, vi) a custom JDK where we used a behavior that encrypts the provided value during the initialization, and finally, vii) a custom JDK where we used a behavior that encrypts the provided value during the initialization and decrypts the value when it is read. For each assessment we disabled any output to `System.out` to prevent potential biases.

We present the results in Table 11.1. We can clearly see that a read from a String object consumes much more time compared to its initialization, *i.e.*, a read is between 1.6 and 41.5 times slower. This seems to be caused by inefficient look-ups in the string pool that may involve value conversions between native and interpreted code. In general, encryption and decryption methods are computationally demanding and can prolong the initialization or read tasks by more than three orders of magnitude, *e.g.*, 5.1 s compared to 4 ms. Attaching behavior to other Strings is relatively cheap, *i.e.*, it usually prolongs the initialization task about 50%. Interestingly, it seems that the vanilla Java VM is performing additional background optimizations for String objects when they are reassigned, which is 2.75 times slower than the initialization of the original String instance. Overall, we observed that the use of strings with rather simple behavior has only a minor average performance impact on reads of less than 16%, which is very close to what other researchers have achieved when they added support for value tainting where they measured an overhead of less than 15% [17].

11.2 Restrictions

In this chapter we investigate the first research question, *i.e.*, *What are the restrictions when using an instrumented Java String class with existing*

Project Name	Package	Version	Compatible?
Apache Commons (IO)	commons-io	2.8.0	✔
Apache Commons (Logging)	commons-logging	1.2	✔
Apache HttpClient	org.apache.httpcomponents	4.5.13	✔
Gson	com.google.code.gson	2.8.5	✖ (reflection)
JavaMail	com.sun.mail	1.6.0	✔
Log4J (core)	org.apache.logging.log4j	2.14.1	✔
Logback (classic)	ch.qos.logback	1.3.0-alpha5	✔
SLF4J	slf4j-simple	2.0.0-alpha1	✖ (cust. byte buf.)
SLF4J (API)	org.slf4j	2.0.0-alpha1	✔
Spring	org.springframework	5.3.6	✔
Square Okhttp	com.squareup.okhttp3	5.0.0-alpha.2	✔
Square Okio	com.squareup.okio	2.10.0	✔
Square Retrofit	com.squareup.retrofit2	2.9.0	✔

Table 11.2: Evaluation of popular Java libraries

code? For that purpose we tested our implementation with some of the most popular libraries in the Maven repository.

11.2.1 Methodology

We searched in the Maven repository for the thirteen most popular Java web communication libraries and downloaded for each the most recent version. Next, we set up a Java project for each library that uses our custom JDK and prepared a String instance with a behavior that logs a potential value leak to the console. Then we provided this String instance to the library and checked whether the behavior was executed within the library code. Whenever a behavior did not work as intended, *i.e.*, we received no message in the console, we kept notes and started to investigate the root cause.

11.2.2 Compatibility

Table 11.2 presents the results of this evaluation. The first column states the project name, the second column the Maven package identifier, the third column lists the tested library version, and finally, the last column indicates whether the library is compatible with *BString*, and if not, it shows the reason why not. We can see that only two of thirteen libraries are incompatible with *BString*, *i.e.*, Gson and SLF4J. Gson uses reflection to access String data internally, which is not supported by *BString*. SLF4J uses custom byte buffers that are not instrumented in *BString*.

11.2.3 Limitations

We discuss the observed limitations of *BString* in more detail, and how these shortcomings could be mitigated in future work, *e.g.*, by extending the framework or the Java environment.

Native Code

BString relies on the Java class system and therefore cannot track strings that leave the object boundary and are forwarded to native code, which can only work with primitive data types. However, a string with behavior might still detect whether it is accessed in a native method and take action before it is transformed to a primitive string type. Conversely, a user can add behavior, *i.e.*, use the method `addBehavior(...)` to attach an implementation of `IStringBehavior` to a new String that is created within a native method. This limitation could be removed entirely, however this demands many changes. On the one hand, the original C and C++ languages only know a string in the form of a character array, which is a primitive type that is not extensible. If behavior features are required, they must be improved to support such mechanisms. On the other hand, the C++ boost library offers a string class that could be adapted, *i.e.*, to *BString*. Nevertheless, this requires that all native C++ code is recompiled and uses that particular class. In the end, the required native behavior interfaces and their implementations would not be interoperable with our Java implementation.

Value Conversion

Value conversions are not supported without additional support in the relevant classes, *e.g.*, `ByteArray`, *etc.* In other words, classes that do not use the String class to work with text and rather prefer primitive collections such as `char` arrays, `byte` streams, *etc.* cannot benefit from *BString* and the attached behavior will be lost when the value conversion is performed. To be clear: we can detect a value conversion in a behavior, but the behavior will be lost after it. This limitation could be removed if additional classes and primitive types would support behaviors, *e.g.*, the `Object` class or the `char` type. However, a generic implementation for `Object` would require type-specific code, which may be difficult to maintain, and primitive types would become more complex and thus slower.

Reflection

In Java, the reflection mechanisms eventually rely on native code to interact with the system that can bypass our behavior. In other words, a behavior will be lost if a String instance is constructed using reflective methods. Therefore, *BString* does not entirely support applications that use reflective methods although it is possible to detect an access from a reflection class in a behavior to act accordingly. This limitation could be removed if the native Java VM code that is responsible for the reflection feature would support behaviors.

Concurrency

BString is currently not thread-safe. This limitation could be removed by improving the existing code of *BString* to leverage concurrency features, *e.g.*, using concurrent locks to safeguard the offered methods. However, such a change would decrease the performance since string operations would then first need to acquire a lock before any further operation, which is very expensive.

Scope

BString only instruments commonly used methods in the String-related classes `String`, `StringBuffer`, and `StringBuilder`. There is currently no support for the `Object` class or wrapper classes like `Integer`, `Float`, or `Boolean`. Moreover, if enabled, the module system introduced in Java 9 prevents the use of *BString* for closed-source projects since such code will not reside in the same module. In order to use *BString* for such analyses, the module system must first be disabled or bypassed. This limitation could be removed by additional changes to the Java VM, and by adding support for behaviors in `Object` instances. However, the latter change would massively increase the complexity for such behaviors.

11.3 Security Gains

In this section, we explore the second research question, *i.e.*, *Can an instrumented String class offer protection against data leaks and remote code execution, and what are the security risks using such a technique?* For that purpose, we implemented selected well-known security measures to prevent data leaks and remote code execution attacks. In particular, we show how a developer can use *BString* to implement type systems, encrypt values, and perform code analyses. Finally, we reason about potential threats that could arise when using *BString*.

11.3.1 Data Type Emulation

Two major type systems that are used in practice are linear types and liquid or refinement types. Whereas a linear type can be accessed once at most, refinement types do not have such a limitation. However, they allow value constraints, *e.g.*, the corresponding value must be shorter than five characters or only contain letters.

Security Gain

Such type systems can be used to prevent remote code execution by specifying value patterns that must not occur (refinement types), or to prevent data leakage and to improve performance (linear types), because a value

can be accessed at most once and immediately after the access it can be safely deleted.

Implementation

The code snippet in Listing 15 shows a String behavior that imitates a linear type, *i.e.*, a boolean flag is checked (line two) to ensure the returned value has not been accessed before in which case the application will proceed as expected. However, if it already has been accessed before, the check will fail and thus raise a `LinearTypeException` (line six). The code snippet in Listing 16 shows a String behavior that imitates a refinement type, *i.e.*, the value is checked whether it meets the requirements (line two) when it is created. If the value does not meet the requirements, the check will fail and thus raise a `RefinementTypeException` (line three).

```
1 public String applyOnRead(String s) {
2     if (!this.hasBeenRead) {
3         this.hasBeenRead = true;
4         return s;
5     } else {
6         throw new LinearTypeException();
7     }
8 }
```

Listing 15: Code that simulates a typical linear type behavior

```
1 public String applyOnCreation(String s) {
2     if (s.length < 5) {
3         throw new RefinementTypeException();
4     }
5     return s;
6 }
```

Listing 16: Code that simulates a typical refinement type behavior

11.3.2 In-memory Encryption

String data can be securely encrypted using a robust encryption algorithm together with a complex password. Such data remains encrypted in the memory and is only decrypted when used.

Security Gain

The in-memory encryption of data values increases the difficulty for adversaries to understand and exfiltrate sensitive data from collected memory dumps.

Implementation

A String value is encrypted in the `applyOnCreation(...)` method and decrypted in the `applyOnRead(...)` method. The simplified code of such

a behavior implementation is shown in Listing 17. The behavior will encrypt the original string and store the encrypted value in memory instead, when a string is created or the behavior attached to an existing string (line four). If the string is requested, the `applyOnRead(...)` method is called and the string is decrypted for the requester (line nine), but it still remains encrypted in memory. The behavior will further attach itself to all derived String instances (lines thirteen to fifteen), *e.g.*, a substring will also have the identical behavior attached and hence be encrypted in-memory. The entire code of a working implementation can be found online in the corresponding GitHub repository.[4]

```
1  public class InMemoryEncryptionBehavior implements IStringBehavior {
2     @Override
3     public String applyOnCreation(String original) {
4        return encrypt(original);
5     }
6
7     @Override
8     public String applyOnRead(String encryptedStringInMemory) {
9        return decrypt(encryptedStringInMemory);
10    }
11
12    @Override
13    public boolean applyDerivationRule(DerivationRule dr) {
14         return true;
15     }
16 }
```

Listing 17: A simplified behavior that illustrates the use of *BString* for in-memory encryption

11.3.3 Off-memory Encryption

Off-memory encrypted data remains as plain text in the memory and is only encrypted when it leaves the memory.

Security Gain

The off-memory encryption of data values ensures that adversaries cannot understand sensitive data that leaves a system, *e.g.*, through a network socket. Developers do not need to encrypt data manually, instead they can attach behavior to sensitive content that will automatically encrypt itself in a transparent process.

Implementation

Contrary to the in-memory encryption, the `applyOnCreation(...)` is not needed and the value encryption in the `applyOnRead(...)` method must

[4] GitHub project website: StringWithBehavior, `https://github.com/pgadient/StringWithBehavior`, *accessed on 23-MAR-2022*

only take place in certain execution contexts, *e.g.*, where a value is stored to disk or transmitted through a network socket. A strong encryption set up, *e.g.*, based on the symmetric-key algorithm AES should be preferred since only a secure crypto configuration can prevent an adversary from recovering encrypted data, and therefore the receivers of the encrypted data must know the key. Listing 18 shows the simplified code for such an implementation. It is not required to implement the `applyOnCreation(...)` method, because the value must only be encrypted when it is read in classes from network and file input or output packages. Two packages related to network and disk storage have been selected to enable the encryption: `java.net` and `java.io` (line two). The current execution context that contains the used packages is determined with the stack trace (line six). If a package name matches one of those that have been previously selected (lines seven to nine), the behavior will automatically encrypt the value before it is returned (line ten). Since it is reasonable to apply this behavior to every derived string for protecting the data, the `applyDerivationRule(...)` method always returns `true` (line nineteen) without checking for a specific `DerivationRule`. The entire code of a working implementation can be found online in the GitHub repository.

```
1  public class OffMemoryEncryptionBehavior implements IStringBehavior {
2     private String[] ioPackages = { "java.net", "java.io" };
3
4     @Override
5     public String applyOnRead(String s) {
6        StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
7        for(int i = 1; i < contexts.length; i++) {
8           for(String ioPackageName: ioPackages) {
9              if(contexts[i].getClassName().startsWith(ioPackageName)) {
10                return encrypt(s);
11             }
12          }
13       }
14       return s;
15    }
16
17    @Override
18    public boolean applyDerivationRule(DerivationRule dr) {
19         return true;
20     }
```

Listing 18: A simplified behavior that illustrates the use of *BString* for off-memory encryption

11.3.4 Taint Analysis

Taint checking or tainting is the concept of adding a marker to a specific variable that indicates its trustworthiness, *i.e.*, either the origin of the value is safe or unsafe. For example, a value received from an insecure

network socket should be tainted as unsafe, because an adversary might have altered its value to exploit the application.

Security Gain

Properly tainted values can prevent arbitrary code execution or data leaks. However, it is important to accurately specify the contexts, *i.e.*, packages, classes, or methods for which unsafe tainted data represents a security threat.

Implementation

Listing 19 shows a typical implementation that provides taint support for a String variable. The use of a boolean marker is optional, because we can selectively assign this behavior to String instances that must be tainted. However, a taint variable can be introduced, if desired, to support use cases that require the `StringController` class. The String array (line two) denotes the packages in which the particular String instance must not be read, *i.e.*, network and input or output-related classes. The current execution context is retrieved from the stack trace (line five), and the package names are validated for each element in it (lines seven and eight). If an unauthorized package is found, a run time exception will be thrown (line nine). Otherwise, the application will proceed as expected (line thirteen). Since the tainting behavior should "taint" every derived string, we just return true and do not check for a specific `DerivationRule` (line seventeen). Please note that the caching of stack traces is not a viable option with the *existing* Java mechanisms, because the returned stack trace instance is different for every look up and therefore a manual comparison, which we implemented, is required anyway.

```
1  private String[] unauthorizedPackages = {"java.net", "java.io"};
2  private ArrayList<String> list = Arrays.asList(unauthorizedPackages);
3
4  public String applyOnRead(String s) {
5      StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
6      for (int i = 1; i < contexts.length; i++) {
7          for (String packageName: list) {
8              if (contexts[i].getClassName().startsWith(packageName)) {
9                  throw new TaintException();
10             }
11         }
12     }
13     return s;
14 }
15
16 public boolean applyDerivationRule(DerivationRule dr) {
17     return true;
18 }
```

Listing 19: A behavior that simulates taint checking

Moreover, a behavior can behave responsibly for different kinds of threats as illustrated in Listing 20. For example, it could ignore them (lines four and five), log the use of a tainted value to the console for minor threats (lines six to eight), or interrupt the further code execution for more major threats by throwing a `TaintException` (lines nine and ten). Further possibilities could involve the manipulation of the String value before it is returned, or the termination of the entire application.

```
1  public String applyOnRead(String s) {
2      ...
3      switch(taintAction)  {
4          case NO_ACTION:
5              return s;
6          case LOG:
7              logLeak(stElements[i].getClassName(), stElements, i);
8              return s;
9          case BLOCK:
10             throw new TaintException("This String must not be used in this
       ↪ package!");
11     }
12     ...
13 }
```

Listing 20: Behavior extension to support different severity levels

11.3.5 Data Flow Analysis

Whereas taint analyses are usually rather limited in the scope and the level of automation, data flow analyses are much more sophisticated and can trace data throughout the system from a source that provides sensitive data to a sink that can leak sensitive data. A data flow analysis typically requires a list of relevant data sources and sinks to report the desired traces.

Security Gains

Data flow analyses can prevent arbitrary code execution or data leaks. However, it is important to accurately specify the contexts, *i.e.*, data sources and sinks for which an exchange of data represents a security threat, *e.g.*, a password field as data source and the console output as disallowed data sink. Using the history feature of our implementation, the originating method and class where a String value was instantiated is always known, every access can be tracked and, if necessary, prevented for selected components of the application.

Implementation

The analysis of data flows is very similar to taint checking, and thus it is straightforward to reuse that code as shown in Listing 21. In essence,

the behavior must be automatically attached to relevant String sources by using the `StringController` class, and every relevant sink must be added to the list of unauthorized contexts (lines one and two). In addition, access to the origin and every subsequent state is required, which can be enabled using the String history feature (lines twenty to twenty-two). The use of history nodes has already been discussed in section 11.1.3.

```
1  private String[] unauthorizedPackages = {"java.net", "java.io"};
2  private ArrayList<String> list = Arrays.asList(unauthorizedPackages);
3
4  public String applyOnRead(String s) {
5      StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
6      for (int i = 1; i < contexts.length; i++) {
7          for (String packageName: list) {
8              if (contexts[i].getClassName().startsWith(packageName)) {
9                  throw new TaintException();
10             }
11         }
12     }
13     return s;
14 }
15
16 public boolean applyDerivationRule(DerivationRule dr) {
17     return true;
18 }
19
20 public boolean recordHistory() {
21     return true;
22 }
```

Listing 21: A behavior that enables data flow analyses

11.3.6 Discussion

We discuss potential threats and summarize the security gains.

Potential Threats

In this subsection, we briefly mention two threats that can arise when using *BString*, how they present a security risk, and how they could be mitigated.

String or Application Hijacking. We call string hijacking the attack that allows an adversary to collect the content of string variables. If adversaries gain access to the behavior files of an application, they could induce malicious behavior that could, for example, send the value of an instrumented string over the internet to any recipient. Consequently, sensitive information may leak. Even worse, malicious behavior classes would also allow an entire application to be hijacked, *i.e.*, adversaries could execute their own code in the context of the Java VM, *e.g.*, to download and execute arbitrary malicious code from the internet to take over the computer that runs the application and eventually the corporate network. This threat can only be mitigated by protecting the behavior files from unauthorized accesses and instituting regular code reviews.

Developer Confusion. A developer who encounters unexpected behavior might become confused and start to debug the application, *e.g.*, when the code does not function as expected. Therefore, the productivity

of such a developer will decrease. This threat cannot entirely be mitigated, however a developer who has to work on projects that rely on behavior classes should be made familiar with the mechanism before writing any code.

Summary

We elaborated five different measures to prevent data leaks and remote code execution that would require several distinct tools, and showed that our proposed framework can provide initial support for all of them. Moreover, different functionality could be combined to provide new solutions to common problems. For example, a developer could create a behavior that validates whether a variable access originates from a network socket class and block the subsequent storage into a database, or transparently encrypt sensitive data to prevent a potential data leak.

11.4 Threats to Validity

In this section we discuss the threats to validity that might affected our results.

The generalization of our implementation might be limited. BString currently only offers support for String-related classes and we do not genuinely know whether such functionality can be provided for other programming languages or classes since their implementation may differ. This is an inherent threat, because we cannot guarantee that the proposed approach can be generalized beyond what we have presented. To reduce the impact of this threat, we applied only few changes to the Java classes to avoid unnecessary dependencies and increase the portability.

We introduced arbitrary decisions during the implementation and evaluation of our work. We selected the String-related classes in our work, because they are used very frequently and can hold alphanumeric credentials unlike numeric types that can only hold numbers. However, other value types may still be valuable for some developers, and we expect that an instrumentation of `Object` would allow more flexibility. For the evaluation, we randomly chose top listed libraries from the Maven repository, which may not represent well the libraries used in practice. We tried limit the impact of this threat by selecting popular classes and libraries that are very likely representative of the current development practices.

The applicability of our results might be limited. We did not test *BString* in large-scale and distributed applications and thus we cannot predict the performance and reliability for such applications. Thread-safety is not yet provided. However, we expect that a well-defined scope can effectively reduce the performance overhead.

The authors have carried the work themselves. Finally, the fact that the evaluation is performed by the authors is a threat to construct validity through potential bias in experimenter expectancy.

11.5 Conclusion

In this work, we modified the String class of a recent OpenJDK release to enable the execution of arbitrary Java code whenever a value is assigned to, or read from a String object. This new interface enables the adoption of several existing security concepts that have been proposed by numerous researchers such as linear and refinement types, or taint and data flow analyses. Moreover, it supports more complex use cases that are hardly possible without such a framework, *e.g.*, transparent in-memory or off-memory data encryption, email notifications, *etc.* Since the implemented behaviors consist of regular Java code, they can be easily integrated into existing development processes and even used beyond a single project. Therefore, such an API would greatly enhance the capabilities of existing Java VMs, and the corresponding behaviors could increase application security at a reasonable cost. We conclude that the proposed framework provides obvious benefits and should be adapted to other languages that use a particular entity to represent text strings.

Chapter 12

Conclusions, Impact, and Future Work

In this thesis we established a comprehensive catalogue of 51 security smells that we have sourced from existing literature, and we assessed their prevalence in publicly accessible apps. The catalogue comprises five different domains of the major mobile app ecosystem, *i.e.*, Android, Android ICC, web communication of mobile apps, app servers, and HTTP clients in mobile apps. The tools that we used for the analyses are open-source and publicly available on GitHub.

Carrying on with these results, we investigated what threats may be introduced by which smell. Moreover, we propose how to mitigate such threats at large and discuss three potential effective holistic remediation strategies, *i.e.*, the use of secure default values, the enforcement of safe practices, and the use of smart data types.

In the remainder of this chapter, we revisit the challenges from chapter 3, before we end this thesis with a closing remark.

12.1 Security Smells in Android

[**Challenge 1**] **The establishment of a notion to comprehensively describe potential vulnerabilities.** We introduced the notion of a "security smell" to describe a bad practice that may turn into a vulnerability. Moreover, the manual inspection of 160 apps showed that the identified security smells are a good indicator of security vulnerabilities.

[**Challenge 2**] **The compilation of a comprehensive list of Android security smells that have been reported in the literature.** We reviewed state of the art papers in security and identified 28 security smells whose presence may indicate a security issue in an app.

[**Challenge 3**] **The large-scale study regarding the prevalence of security smells in mobile apps.** We developed a static analysis tool to study the prevalence of ten of such smells in 46 000 apps. We realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security smells.

12.1.1 Visible Impacts

[**Impact 1**] **Adoption of the term "security smell."** The term has been adopted by other researchers after we published our initial work, *e.g.*, in the realm of code scripts [76, 77], smart contracts [22], and microservice-based application [73].[1]

12.1.2 Future Work

We see four major aspects that should be considered in future work: time series, more heavyweight analyses, the expansion to different mobile platforms, and improved tool support. In our analysis, we only covered the most recent version of an app in the dataset. However, an extended analysis could offer valuable insight if repetitively performed over time, *e.g.*, to identify the adoption rate of security smell mitigation strategies. Whereas we used a lightweight analysis based on regular expressions to match method use patterns, a heavyweight analysis could provide a more thorough view on the existence of security smells. For example, a static or dynamic analyzer could leverage additional context to offer more precise feedback. We focused solely on Android apps, however there exist other major mobile platforms such as Apple's iOS, iPadOS, watchOS, and tvOS, which would also benefit from such data gatherings. Finally, we did not implement any developer tool support for the reported security smells. Android app security can be expected to improve if the Android Studio IDE could detect and report such security smells just-in-time to developers.

12.2 Security Smells in Android ICC

[**Challenge 4**] **The compilation of a comprehensive list of ICC security smells that have been reported in the literature.** We reviewed state of the art work that discusses ICC-related vulnerabilities that threaten Android apps and compiled a list of twelve ICC security smells.

[1] Google Scholar: "security smell" search, `https://scholar.google.com/scholar?q=security+smells`, *accessed on 26-MAR-2022*

[**Challenge 5**] **The implementation of IDE tool support for the reported ICC security smells.** We implemented a linting plug-in for Android Studio that helps developers to spot and understand such smells by linting affected code parts and providing just-in-time feedback about their presence. A manual investigation of 100 apps shows that our tool successfully finds many different ICC security code smells, and about 43.8% of them in fact represent vulnerabilities. Thus it constitutes a reasonable measure to improve the overall development efficiency and software quality.

[**Challenge 6**] **The large-scale study of the prevalence of ICC security smells in mobile apps.** We applied our analysis to a corpus of more than 700 open-source apps. We observed that only small teams are capable of consistently building software resistant to most security code smells, and fewer than 10% of apps suffer from more than two ICC security smells. We discovered that updates rarely have any impact on ICC security, however, in case they do, they often correspond to new app features. Moreover, we found that long-lived projects suffer from more issues than recently created ones, except for apps that are updated frequently, for which that effect is reversed.

12.2.1 Visible Impacts

[**Impact 2**] **Collaboration with Google.** We collaborated with Google's Android Studio team and submitted the revised ICC security smell linters for further integration into Android Studio.

[**Impact 3**] **Code linting for ICC security smells.** Patrick Frischknecht carried out further research on ICC security smells and contributed to a security smell linting plug-in for Android Studio that can provide just in time feedback [33].

[**Impact 4**] **Quick fixes for ICC security smells.** Dominik Briner carried out further research on ICC security smells and contributed to a quick fix plug-in for Android Studio that can not only provide just in time feedback, but also resolve many security smells with just a few clicks [13].

12.2.2 Future Work

We see two major aspects that should be considered in future work: time series, and the expansion to different mobile platforms. In our analysis, we only covered the most recent version of an app in the dataset. However, an extended analysis could offer valuable insight if repetitively performed over time, *e.g.*, to identify the adoption rate of ICC security smell mitigation

strategies. Moreover, we focused solely on Android apps, however there exist other major mobile platforms such as Apple's iOS, iPadOS, watchOS, and tvOS, which would also benefit from such data gatherings.

12.3 Security Smells in the Web Communication of Mobile Apps

[**Challenge 7**] **The implementation of a tool to extract data relevant for the web communication of mobile apps.** We manually reviewed 160 Android apps to compile a list of commonly used network and data conversion libraries and to learn how they are used in these apps. Based on our findings, we developed a lightweight static analysis tool that identifies network-related APIs, and extracts communication information such as the web APIs, and the associated JSON headers.

[**Challenge 8**] **The large-scale study of web communication characteristics of mobile apps.** With the help of our tool we successfully analyzed the network-related information within 450 closed-source and open-source apps. We found that in both open-source and closed-source apps network communication is mainly developed using *java.net* classes. Amongst the third-party libraries we found that *OkHttp* and *Retrofit* are used the most. By far the most used value type in JSON data is `STRING`. We realized that closed-source apps substantially rely on advertisement services, and that they tend to have more complex URL paths consisting of more path segments. Surprisingly, the secure HTTPS protocol is used in the majority of extracted web APIs from open-source applications, but the opposite is true for closed-source apps.

[**Challenge 9**] **The compilation of a comprehensive list of web communication security smells that have been reported in the literature.** During the analysis we identified bad coding habits from the literature that we could relate to eight web communication security smells. Major findings were the numerous cases of embedded language use during the manual examination of the web APIs, *i.e.*, embedded SQL and JavaScript content was rather common within web communications, and the many issues in the server side communication: unnecessary disclosure of server configurations, *e.g.*, outdated web servers and language interpreters with known security vulnerabilities, leaks of internal error messages, and other sensitive data. We further found private APIs without any kind of communication authentication or authorization mechanisms.

12.3.1 Visible Impacts

[**Impact 5**] **Web communication analysis of Android applications.** Marc-Andrea Tarnutzer carried out further research on the web communication analysis of Android apps and contributed to a tool that can reconstruct used web data structures from source code [93].

12.3.2 Future Work

We see two major aspects that should be considered in future work: the expansion to different mobile platforms, and improved tool support. We focused solely on Android apps, however there exist other major mobile platforms such as Apple's iOS, iPadOS, watchOS, and tvOS, which would also benefit from such data gatherings. Moreover, we did not implement any developer tool support for the reported security smells. Android app security can be expected to improve if the Android Studio IDE could detect and report such security smells just-in-time to developers.

12.4 Security Smells in Mobile App Servers

[**Challenge 10**] **The large-scale study of the prevalence of the server-side security smells in the web communication of mobile apps.** We used an existing dataset that includes 9 714 distinct URLs that were used in 3 376 Android mobile apps to analyze the prevalence of six security smells in app servers, and to investigate the consequence of these smells from a security perspective. We realized that the top three smells exist in more than 69% of all tested apps, and that unprotected communication and server misconfigurations are very common. Particularly alarming is the finding that apps using JSON app servers suffer 1.5 times more from app server security smells than non-JSON apps, and even worse, closed-source applications suffer 1.6 times more compared to open-source applications.

[**Challenge 11**] **The investigation of the relationship between security smells and app server maintenance.** We exercised the URLs twice over fourteen months, and stored the HTTP headers and bodies. We found that app server security smells are omnipresent and they indicate poor app server maintenance.

12.4.1 Future Work

We see two major aspects that should be considered in future work: the expansion to different mobile platforms, and improved tool support. We focused solely on Android apps, however there exist other major mobile platforms such as Apple's iOS, iPadOS, watchOS, and tvOS, which would also benefit from such data gatherings. Moreover, we did not implement

any server side tool support for the reported security smells. Server security can be expected to improve if the server management tools could detect and report security smells just-in-time to system administrators.

12.5 Security Smells in Mobile App HTTP Clients

[**Challenge 12**] **The investigation of security-related HTTP header support in existing HTTP clients.** We collected the HTTP response header information from 9 714 distinct URLs found in 3 376 Android apps. We discovered that, on average, 93% of the security-related headers are not used in server responses, indicating great potential for future improvements. We also found that unlike major web browsers, the support for such fields in HTTP client libraries is very limited, and that server responses for mobile apps frequently lack them.

12.5.1 Future Work

We see two major aspects that should be considered in future work: the expansion to different mobile platforms, and improved library support. We focused solely on Android apps, however there exist other major mobile platforms such as Apple's iOS, iPadOS, watchOS, and tvOS, which would also benefit from such data gatherings. Finally, we did not implement mitigation strategies in libraries for the reported smells. Android app security can be expected to improve if the used libraries would support the proposed features.

12.6 Effective Holistic Security for Mobile Apps

[**Challenge 13**] **The review of reported security smells with respect to how they enable attack mechanisms.** We manually investigated the impact of our 51 security smells on 192 attack mechanisms of the CAPEC taxonomy, which led to 9 792 combinations that we considered. We found that insecure algorithms, the abuse of existing functionality, data leaks, and user deception are the four major threats that users face when using Android.

[**Challenge 14**] **The identification of holistic security strategies that can effectively prevent attack mechanisms.** We explain the concept of effective and holistic security in the context of mobile apps and elaborate strategies with respect to the four most affected major attack mechanisms considering the results of the previous challenge. We see most potential in secure default values and safer practices to prevent feature misuse in the Android ecosystem.

We realized further that string variables are responsible for most issues that relate to the employment of probabilistic methods and the injection of unexpected items, and thus they need increased protection.

12.6.1 Future Work

We see three major aspects that should be considered in future work: a follow-up peer review to validate the preliminary results, a follow-up study of how the attack mechanisms correlate with concrete attacks in the wild, and the consideration of additional taxonomies. We recommend a follow-up peer review to identify and clear misclassifications, because the current analysis has been solely performed by the author and has not been subject to a thorough review. Furthermore, we urge to investigate the correlation of attack mechanisms with concrete attacks, because although we observed a high correlation between smells and some attack mechanisms, this does not necessarily mean that there is any practical relevance. The labeled data required to assess the correlation could be collected from CVE. Moreover, we established effective holistic remedies to address few major CAPEC attack mechanism categories, however there might exist other classification schemes that could reveal additional remedies.

12.7 Default Values and Practices to Improve Application Security

[**Challenge 15**] **The discussion of default values and practices that could greatly improve application security.** We reviewed every reported security smell and reasoned whether it could be mitigated with improved platform security. We found that eight smells (16%) could be addressed with more secure default values, and that 36 smells (71%) could be addressed with safer practices. In fact, we only see for seven smells (14%) no potential in such measures, however they can be addressed using a better control of data and we implement a potential solution to this problem in chapter 11.

12.7.1 Future Work

We see two major aspects that should be considered in future work: the implementation and evaluation of the proposed measures in practice, and the more thorough exploration of mitigation strategies for the reported problems. We could not implement and evaluate the proposed measures in the Android ecosystem, however a collaboration with several distinct teams working on Google's Android ecosystem would be required for the implementation of all the proposed measures. Furthermore, we strived to identify effective measures that serve the purpose, but there may exist

additional pragmatic measures that could be identified in literature or in discussions with a diverse crowd of security professionals.

12.8 A String-based Framework to Improve Application Security

[**Challenge 16**] **The implementation of a String-based framework to improve application security.** We modified the String class of a recent OpenJDK release to enable the execution of arbitrary Java code whenever a value is assigned to, or read from a String object. This new interface enables the adoption of several existing security concepts that have been proposed by numerous researchers such as linear and refinement types, or taint and data flow analyses. Moreover, it supports more complex use cases that are hardly possible without such a framework, *e.g.*, transparent in-memory or off-memory data encryption, email notifications, *etc.* Since the implemented behaviors consist of regular Java code, they can be easily integrated into existing development processes and even used beyond a single project.

[**Challenge 17**] **The investigation of the restrictions when using such a framework with existing code.** We searched in the Maven repository for the thirteen most popular and most recent Java web communication libraries and downloaded each of them, which we instrumented with our custom JDK. Whenever the instrumentation did not work as intended, we kept notes and started to investigate the root cause. We found that eleven libraries were compatible and only two libraries were failing our framework, *i.e.*, due to an unsupported use of reflection or the lack of instrumentation in custom data classes. However, these two limitations inherently reveal further limitations, *i.e.*, the lack of support for native code, arbitrary objects and the corresponding conversions, and finally, concurrency.

[**Challenge 18**] **The investigation whether such a framework can offer protection against data leaks and remote code execution, and what security risks could arise when using it.** We implemented five security measures reported in existing literature, *i.e.*, data type emulation, in-memory and off-memory encryption, taint and data flow analysis, and discuss two major threats that may arise when using the framework, *i.e.*, string or application hijacking, and developer confusion.

12.8.1 Visible Impacts

[**Impact 6**] **Protecting string-based sensitive information.** Christian Zürcher carried out further research on the protection of string-

based information in Java applications and contributed to a prototype that offers flexible security mechanisms that can be used in existing applications [122].

12.8.2 Future Work

We see four major aspects that should be considered in future work: improving the current implementation, the expansion to different platforms, and improved tool support. Our framework supports currently only the String class and adding support for additional classes may increase the utility of the framework for more developers. We particularly used the Java VM for demonstration purposes, however we expect that it is possible to adopt the concept in other object-oriented programming languages. Finally, we did not implement any particular developer tool support that would ease the work with such strings, nor did we yet propose these features to the well-known OpenJDK project. If Java VMs and major IDEs would offer such features out of the box, it would be much more convenient for developers to use them.

12.9 Closing Remarks

This study condenses information regarding good and bad development practices for an entire mobile ecosystem, and is to our knowledge the first of its kind. This comprehensive and unique view allows one to reason beyond existing work and thus advances the understanding of software security in the highly complex mobile domain.

Bibliography

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. How internet resources might be helping you develop faster but less securely. *IEEE Security Privacy*, 15(2):50–60, March 2017.

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle Mazurek, and Christian Stransky. Comparing the usability of cryptographic APIs. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, 2017.

[3] Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich. Inter-app communication in Android: Developer challenges. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 177–188. IEEE, 2016.

[4] Huyam AL-Amro and Eyas El-Qawasmeh. Discovering security vulnerabilities and leaks in ASP.NET websites. In *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, pages 329–333, 2012.

[5] Eman Salem Alashwali, Pawel Szalachowski, and Andrew Martin. Exploring HTTPS security inconsistencies: A cross-regional perspective. *Computers & Security*, 97:101975, 2020.

[6] Brian Anderson and Barbara Anderson. *Seven deadliest USB attacks*. Syngress, 2010.

[7] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015. ACM, 2015.

[8] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in Android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 356–367, New York, NY, USA, 2016. ACM.

[9] R. Balebako and L. Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security Privacy*, 12(4):55–58, July 2014.

[10] David Bermbach. Quality of cloud services: Expect the unexpected. *IEEE Internet Computing*, 21(1):68–72, 2017.

[11] David Bermbach and Erik Wittern. Benchmarking web API quality – revisited. *arXiv preprint arXiv:1903.07712*, 2019.

[12] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.

[13] Dominik Briner. Developer tool support for security code smells. Bachelor's thesis, University of Bern, July 2021.

[14] William J Buchanan, Scott Helme, and Alan Woodward. Analysis of the adoption of security headers in HTTP. *IET Information Security*, 12(2):118–126, 2018.

[15] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale. In *24th USENIX Security Symposium*, pages 659–674, Washington, D.C., 2015. USENIX Association.

[16] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[17] Erika Chin and David Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the 2009 ACM workshop on Secure web services*, pages 3–12, 2009.

[18] M. Conti, N. Dragoni, and V. Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys Tutorials*, 18(3):2027–2051, thirdquarter 2016.

[19] Claudio Corrodi, Timo Spring, Mohammad Ghafari, and Oscar Nierstrasz. Idea: Benchmarking Android data leak detection tools. In Mathias Payer, Awais Rashid, and Jose M. Such, editors, *Engineering Secure Software and Systems*, pages 116–123, Cham, 2018. Springer International Publishing.

[20] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, 2011.

[21] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. Free for all! assessing user data exposure to advertising libraries on Android. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[22] Mehmet Demir, Manar Alalfi, Ozgur Turetken, and Alexander Ferworn. Security smells in smart contracts. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 442–449, 2019.

[23] Ehsan Edalat, Babak Sadeghiyan, and Fatemeh Ghassemi. ConsiDroid: A concolic-based tool for detecting SQL injection vulnerability in Android apps. *arXiv preprint arXiv:1811.10448*, 2018.

[24] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61. ACM, 2012.

[25] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab 'n run: Secure and practical dynamic code loading for Android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015. ACM, 2015.

[26] Zheran Fang, Weili Han, Dong Li, Zeqing Guo, Danhao Guo, Xiaoyang Sean Wang, Zhiyun Qian, and Hao Chen. RevDroid: Code analysis of the side effects after dynamic permission revocation of Android apps. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 747–758, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Dominic Farolino, Jochen Eisinger, and Emily Stark. W3C editor's draft: Referrer policy, 2021.

[28] Johannes Feichtner. A comparative study of misapplied crypto in Android and iOS applications. In *ICETE (2)*, pages 96–108, 2019.

[29] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring HTTPS adoption on the web. In *26th USENIX security symposium (USENIX security 17)*, pages 1323–1338, 2017.

[30] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, page 88, 2011.

[31] R Fielding and J Reschke. RFC 7230: Hypertext transfer protocol (HTTP/1.1): Message syntax and routing, 2014.

[32] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. RFC 2616: Hypertext transfer protocol–HTTP/1.1, 1999.

[33] Patrick Frischknecht. Security in android icc. Bachelor's thesis, University of Bern, June 2018.

[34] Pascal Gadient, Mohammad Ghafari, Patrick Frischknecht, and Oscar Nierstrasz. Security code smells in Android ICC. *Empirical Software Engineering*, 24:3046–3076, 2019.

[35] Pascal Gadient, Mohammad Ghafari, Marc-Andrea Tarnutzer, and Oscar Nierstrasz. Web APIs in Android through the lens of security. In *27th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2020.

[36] Pascal Gadient, Oscar Nierstrasz, and Mohammad Ghafari. Security header fields in HTTP clients. In *21st IEEE International Conference on Software Quality, Reliability, and Security (QRS)*, December 2021.

[37] Pascal Gadient, Marc-Andrea Tarnutzer, Oscar Nierstrasz, and Mohammad Ghafari. Security smells pervade mobile app servers. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, October 2021.

[38] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for Android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 661–671. ACM, 2017.

[39] M. Ghafari, P. Gadient, and O. Nierstrasz. Security smells in Android. In *2017 IEEE 17th International Working Conference on*

Source Code Analysis and Manipulation (SCAM), pages 121–130, Sept 2017.

[40] Md Maruf Hassan, Shamima Sultana Nipa, Marjan Akter, Rafita Haque, Fabiha Nawar Deepa, Mostafijur Rahman, Md Asif Siddiqui, Md Hasan Sharif, et al. Broken authentication and session management vulnerability: a case study of web application. *International Journal of Simulation Systems, Science & Technology*, 19(2):6–1, 2018.

[41] Jeff Hodges, Collin Jackson, and Adam Barth. RFC 6797: HTTP strict transport security (HSTS). *Internet Engineering Task Force (IETF)*, 2012.

[42] Qinwen Hu, Muhammad Rizwan Asghar, and Nevil Brownlee. A large-scale analysis of HTTPS deployments: Challenges, solutions, and recommendations. *Journal of Computer Security*, Preprint:1–26, 2021.

[43] Shinelle Hutchinson, Bing Zhou, and Umit Karabiyik. Are we really protected? an investigation into the Play Protect service. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4997–5004. IEEE, 2019.

[44] Sungjae Hwang, Sungho Lee, Yongdae Kim, and Sukyoung Ryu. Bittersweet ADB: Attacks and defenses. In *ASIACCS*, 2015.

[45] Vineeta Jain, Shweta Bhandari, Vijay Laxmi, Manoj Singh Gaur, and Mohamed Mosbah. SniffDroid: Detection of inter-app privacy leaks in Android. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 331–338. IEEE, 2017.

[46] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[47] Beth H. Jones and Amita Goyal Chin. On the efficacy of smartphone security: A critical analysis of modifications in business students' practices over time. *International Journal of Information Management*, 35(5):561 – 571, 2015.

[48] van Anne Kesteren. WHATWG: Fetch living standard, 2019.

[49] Babu Khadiranaikar, Pavol Zavarsky, and Yasir Malik. Improving Android application security for intent-based attacks. In *Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2017 8th IEEE Annual*, pages 62–67. IEEE, 2017.

[50] Youngho Kim, Tae Oh, and Jeongnyeo Kim. Analyzing user awareness of privacy data leak in mobile applications. *Mobile Information Systems*, 2015, 2015.

[51] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air. In *Proceedings of thz 2015 Network and Distributed System Security Symposium. NDSS*, 2015.

[52] Arturs Lavrenovs and F. Jesús Rubio Melón. HTTP security headers analysis of top one million websites. In *2018 10th International Conference on Cyber Conflict (CyCon)*, pages 345–370, 2018.

[53] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro. Understanding Android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, June 2017.

[54] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[55] 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. *Information and Software Technology*, 88:67 – 95, 2017.

[56] Chia-Chi Lin, Hongyang Li, Xiao yong Zhou, and XiaoFeng Wang. Screenmilker: How to milk your Android screen for secrets. In *NDSS*, 2014.

[57] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on Android-related vulnerabilities. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 2–13, Piscataway, NJ, USA, 2017. IEEE Press.

[58] X. Liu, X. Lu, H. Li, T. Xie, Q. Mei, H. Mei, and F. Feng. Understanding diverse usage patterns from large-scale appstore-service profiles. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[59] Moxie Marlinspike. More tricks for defeating SSL in practice. *Black Hat USA*, 2009.

[60] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. Uncovering HTTP header inconsistencies and the impact on desktop/-mobile websites. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, page 247–256, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[61] Abner Mendoza and Guofei Gu. Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 756–769. IEEE, 2018.

[62] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of Android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 43–52. ACM, 2017.

[63] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946, New York, NY, USA, 2016. ACM.

[64] Y. Nan, Z. Yang, M. Yang, S. Zhou, Y. Zhang, G. Gu, X. Wang, and L. Sun. Identifying user-input privacy in mobile applications at a large scale. *IEEE Transactions on Information Forensics and Security*, 12(3):647–661, March 2017.

[65] Yasemin Acar nd Sascha Fahl nd Michelle Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *IEEE SecDev 2016*, 2016.

[66] Jon Oberheide and Charlie Miller. Dissecting the Android Bouncer. *SummerCon2012, New York*, 95:110, 2012.

[67] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, Washington, D.C., 2013. USENIX.

[68] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. To pin or not to pin-helping app developers bullet proof their TLS connections. In *USENIX Security Symposium*, 2015.

[69] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: Experimenting with SSL vulnerabilities in Android apps. In

Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec '15, pages 15:1–15:6. ACM, 2015.

[70] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do Android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 331–341, 2018.

[71] Nicholas J Percoco and Sean Schulte. Adventures in Bouncerland. *Black Hat USA*, 95:110, 2012.

[72] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[73] Francisco Ponce. Towards resolving security smells in microservice-based applications. In *European Conference on Service-Oriented and Cloud Computing*, pages 133–139. Springer, 2020.

[74] Andrea Possemato and Yanick Fratantonio. Towards HTTPS everywhere on Android: We are not there yet. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 343–360, 2020.

[75] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 176–186. ACM, 2018.

[76] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 164–175, Piscataway, NJ, USA, 2019. IEEE Press.

[77] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. Security smells in ansible and chef scripts: A replication study. *ACM Trans. Softw. Eng. Methodol.*, 30(1), jan 2021.

[78] Marianna Rapoport, Philippe Suter, Erik Wittern, Ondřej Lhóták, and Julian Dolby. Who you gonna call?: Analyzing web requests in Android applications. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 80–90, Piscataway, NJ, USA, 2017. IEEE Press.

[79] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS*, 2016.

[80] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying TLS usage in Android apps. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 350–362, New York, NY, USA, 2017. Association for Computing Machinery.

[81] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. *Droid: Assessment and evaluation of Android application analysis tools. *ACM Comput. Surv.*, 49(3):55:1–55:30, 2016.

[82] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in Android. In *USENIX Security Symposium*, pages 945–959, 2015.

[83] David Ross, Tobias Gondrom, and T Stanley. RFC 7034: HTTP header field X-Frame-Options. *Internet Engineering Task Force (IETF)*, 2013.

[84] A. Sadeghi, H. Bagheri, J. Garcia, and s. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[85] Brian Schweigler. An investigation into vulnerability databases. Bachelor's thesis, University of Bern, May 2020.

[86] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z. Morley Mao. The misuse of Android Unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 80–91, New York, NY, USA, 2016. ACM.

[87] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, 2012.

[88] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, 2008.

[89] Yunsik Son. A study on software vulnerability of programming languages interoperability. In Tai-hoon Kim, Hojjat Adeli, Rosslin John

Robles, and Maricel Balitanas, editors, *Advanced Computer Science and Information Technology*, pages 123–131, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[90] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930. ACM, 2010.

[91] Emily Stark. IETF draft: Expect-CT extension for HTTP, 2018.

[92] Meenakshi Suresh, PP Amritha, Ashok Kumar Mohan, and V Anil Kumar. An investigation on HTTP/2 security. *Journal of Cyber Security and Mobility*, pages 161–180, 2018.

[93] Marc-Andrea Tarnutzer. Web communication analysis of Android applications. Master's thesis, University of Bern, April 2019.

[94] Vincent F. Taylor and Ivan Martinovic. SecuRank: Starving permission-hungry apps using contextual permission analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '16, pages 43–52, New York, NY, USA, 2016. ACM.

[95] Vincent F. Taylor and Ivan Martinovic. To update or not to update: Insights from a two-year study of Android app evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 45–57, New York, NY, USA, 2017. ACM.

[96] Dennis Titze and Julian Schütte. Preventing library spoofing on Android. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 01*, TRUSTCOM '15, pages 1136–1141. IEEE Computer Society, 2015.

[97] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. JIT feedback — what experienced developers like about static analysis. In *Proceedings of the 26th IEEE International Conference on Program Comprehension (ICPC'18)*, 2018.

[98] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: threats and mitigation. In *ACM Conference on Computer and Communications Security*, 2013.

[99] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on iOS: When benign apps become evil. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 559–572, Washington, D.C., 2013. USENIX.

[100] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[101] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 14–24, 2017.

[102] Charles Weir, Awais Rashid, and James Noble. Reaching the masses: A new subdiscipline of app programmer education. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 936–939. ACM, 2016.

[103] Yoav Weiss and Noam Rosenthal. W3C editor's draft: Resource timing level 2, 2021.

[104] M West. W3C working draft: Content security policy level 3, 2021.

[105] William E Winkler and Yves Thibaudeau. *An application of the Fellegi-Sunter model of record linkage to the 1990 US decennial census*. Citeseer, 1991.

[106] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 260–271. ACM, 2015.

[107] Daoyuan Wu, Debin Gao, Yingjiu Li, and Robert H. Deng. SecComp: Towards practically defending against component hijacking in Android applications. *CoRR*, abs/1609.03322, 2016.

[108] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on Android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[109] Wenjia Wu, Jianan Wu, Yanhao Wang, Zhen Ling, and Ming Yang. Efficient fingerprinting-based Android device identification with zero-permission identifiers. *IEEE Access*, 4:8073–8083, 2016.

[110] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164, Sept 2011.

[111] Jiayun Xie, Xiao Fu, Xiaojiang Du, Bin Luo, and Mohsen Guizani. AutoPatchDroid: A framework for patching inter-app vulnerabilities in Android application. In *Communications (ICC), 2017 IEEE International Conference on*, pages 1–6. IEEE, 2017.

[112] LIANG XU, Shyhtsun Felix Wu, and Hao Chen. Techniques and tools for analyzing and understanding Android applications. In *Dissertation*, 2013.

[113] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. Toward engineering a secure Android ecosystem: a survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.

[114] George O. M. Yee. Model for reducing risks to private or sensitive data. In *Proceedings of the 9th International Workshop on Modelling in Software Engineering*, MISE '17, pages 19–25, Piscataway, NJ, USA, 2017. IEEE Press.

[115] Xiao Zhang, Yousra Aafer, Kailiang Ying, and Wenliang Du. *Hey, You, Get Off of My Image: Detecting Data Residue in Android Images*, pages 401–421. Springer International Publishing, Cham, 2016.

[116] Xiao Zhang and Wenliang Du. Attacks on Android clipboard. In *DIMVA*, 2014.

[117] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. Life after app uninstallation: Are the data still alive? data residue attacks on Android. In *NDSS*, 2016.

[118] Min Zheng, Mingshen Sun, and John C. S. Lui. DroidRay: a security evaluation system for customized android firmwares. In *ASIACCS*, 2014.

[119] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in Android apps. In *WISEC*, pages 1–12, 2015.

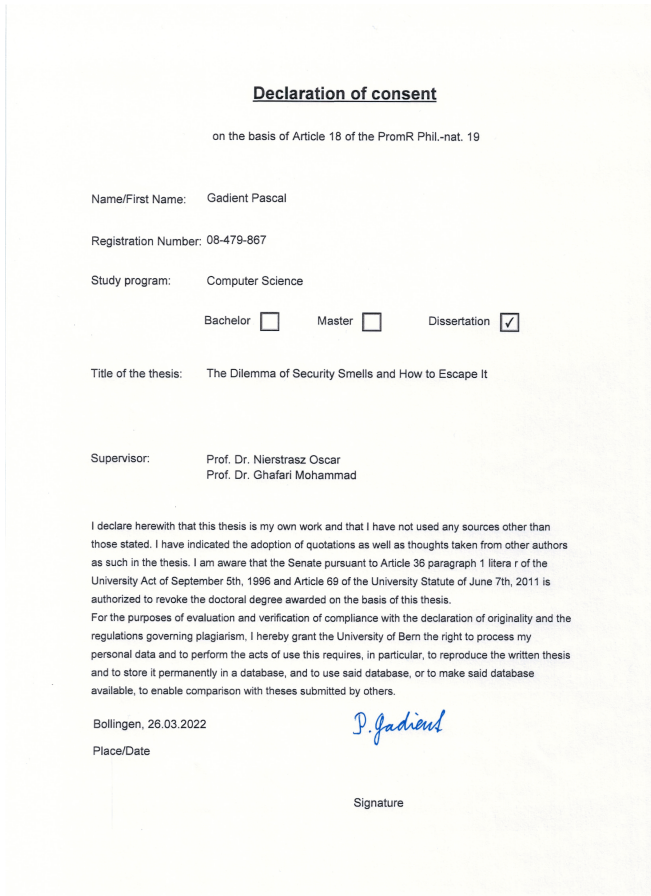
[120] Chaoshun Zuo and Zhiqiang Lin. SMARTGEN: Exposing server URLs of mobile apps with selective symbolic execution. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 867–876, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[121] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310, 2019.

[122] Christian Zürcher. BString: A String-based framework to improve application security. Master's thesis, University of Bern, February 2022.

Appendix A

Declaration of Consent



Declaration of consent

on the basis of Article 18 of the PromR Phil.-nat. 19

Name/First Name: Gadient Pascal

Registration Number: 08-479-867

Study program: Computer Science

Bachelor ☐ Master ☐ Dissertation ☑

Title of the thesis: The Dilemma of Security Smells and How to Escape It

Supervisor: Prof. Dr. Nierstrasz Oscar
Prof. Dr. Ghafari Mohammad

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of September 5th, 1996 and Article 69 of the University Statute of June 7th, 2011 is authorized to revoke the doctoral degree awarded on the basis of this thesis.
For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with theses submitted by others.

Bollingen, 26.03.2022 P. Gadient

Place/Date

Signature

Appendix B

Curriculum Vitæ

B.1 Academic Education

10/2017 - 05/2022 **Dr. rer. nat. (PhD)**.
Software Composition Group (SCG)
University of Bern, Switzerland (part-time)
Thesis advisors:
Prof. Dr. Oscar Nierstrasz
Prof. Dr. Mohammad Ghafari

09/2014 - 09/2017 **Swiss Joint Master of Science in Computer Science (MSc)**.
Universities Bern, Neuchâtel and Fribourg, Switzerland (part-time)

08/2008 - 02/2013 **Bachelor FHO in Computer Science (BSc)**.
University of Applied Sciences (OST), Rapperswil-Jona, Switzerland (part-time)

08/2006 - 08/2007 **Vocational Baccalaureate (BMS)**.
AKAD Profession, Zürich-Oerlikon, Switzerland (part-time)

B.2 Professional Experience

10/2017 - 12/2021 **Part-time employment**
Software Composition Group, University of Bern, Switzerland

08/2006 - today **Part-time employment**
Paul Morger AG, Rüti ZH, Switzerland

08/2002 - 08/2006 **Apprenticeship as IT professional**
Paul Morger AG, Rüti ZH, Switzerland