Security in Android Applications

Master Thesis

Pascal Gadient
from
Bollingen, Switzerland

Faculty of Science
at the University of Bern

29. August 2017

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari

Software Composition Group
Institute for Computer Science
University of Bern, Switzerland

Abstract

The ubiquity of smartphones, and their very broad capabilities and usage, make the security of these devices tremendously important. Unfortunately, despite all progress in security and privacy mechanisms, vulnerabilities continue to proliferate. Research has shown that many vulnerabilities are due to insecure programming practices. However, each study has often dealt with a specific issue, making the results less actionable for practitioners. To promote secure programming practices, we have reviewed related research, and identified 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.

Contents

1 Introduction

Smartphones and tablets have recently overtaken the number of computers.[1] They provide powerful features once offered only by computers, however, the potential impact of malware 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. A recent survey on the Stackoverflow website shows that about 65% of mobile developers work with Android.[2] This platform has captured over 80% of the smartphone market,[3] 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[4] of billions of users. Making matters worse, this is particularly important with respect to Android as it is aiming beyond smartphones and tablets, appearing in smart TVs (*i.e.*, Android TV) and watches (*i.e.*, Android Wear), car navigation systems (*i.e.*, Android Auto), and home automation systems (*i.e.*, Android Brillo). As a result, it is also considered one of the most promising platforms for the growing Internet-of-Things (IoT) ecosystem. This highlights the importance of a platform-related security smell collection and contributes hopefully to platform's overall security.

The security of smartphones has been studied from various perspectives such as the device manufacturer [83], its platform [87], and end users [42]. Manifold security APIs, protocols, guidelines, and tools

[1] `http://www.pewinternet.org/fact-sheet/mobile`
[2] `http://insights.stackoverflow.com/survey/2017`
[3] `http://www.gartner.com`
[4] In short, referred to as security in this thesis

are proposed. Nevertheless, security concerns, in effect, are outweighed by other concerns [11]. Reaves *et al.* assessed Android app analysis tools, and found that they mainly suffered from lack of maintenance, and were often unable to produce functional output for applications with known vulnerabilities [64]. Linares-Vasquez *et al.* mined 660 Android vulnerabilities available in the official Android bulletins and the CVE-details[5] and acknowledged that most of them can be avoided by relying on secure coding practices [48]. Xie *et al.* interviewed 15 professional developers about their software security knowledge, and realized that many of them have reasonable knowledge but do not adopt it as they believe it is others responsibility [84]. As a result, apps still suffer from serious proliferating security issues.[6] For instance, analysing 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 [59]. Another weak point in Android's ecosystem is the version fragmentation, effectively preventing OS level approaches from getting integrated promptly into Android due to several vendor-specific concerns, *e.g.*, for compatibility or performance reasons, fostering the long-term prevalence of issues.

The primary goal of this work 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 introduce the notion of ***security code smells*** *i.e.*, *symptoms in the code that signal the prospect of a vulnerability*. We have identified avoidable vulnerabilities, their corresponding smells in the code; and 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 research questions:

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

- $\mathbf{RQ}_2$: How prevalent are security smells in benign apps? We have developed a lightweight tool that statically analyses 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 suffers from at least three different security smells.

- $\mathbf{RQ}_3$: 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, thus our tool reports primarily real security threats.

[5] Common Vulnerabilities and Exposures, a platform that assigns each vulnerability to a unique identifier shared among different security contributors together with details regarding the issue

[6] `http://www.cvedetails.com`

1.1 Contributions

To summarise, this work represents an initial effort to spread awareness about the impact of programming choices in making secure apps. We identified 28 security code smells in 5 different categories, namely *Insufficient Attack Protection*, *Security Invalidation*, *Broken Access Control*, *Sensitive Data Exposure* and *Lax Input Validation*. We collected for each security code smell its symptoms and the potential impact, together with basic remediation measures. We argue that this helps developers who develop security mechanisms or other sensitive code to identify frequent problems, and also provides developers inexperienced in security with caveats about the prospect of security issues in their code.

1.2 Outline

The remainder of this thesis is structured as follows. We summarize the work related to Android security in Chapter 2. Chapter 3 briefly explains the application level attacks and then presents 28 security code smells and associate each smell to the attack(s). Chapter 4 shows our empirical study and presents the obtained results. Finally, this thesis concludes in Chapter 5.

2
State of the Art

Impelled by Android's large-scale distribution, security research has become very popular in academia, hence many scientific papers have been published. Security in mobile devices involves many actors. In this chapter we particularly discuss *device vendors*, *operating systems* and *apps* in detail and present state of the art publications accordingly.

2.1 Vendor

Device drivers frequently need direct hardware access and thus utilise elevated privileges. Inappropriately developed drivers could expose system components such as cameras, microphones, GPS receivers, *etc.* to adversaries [95]. Pereira *et al.* unveiled exposed serial modem interfaces over USB connections that allow adversaries, without user consent, various low-level device manipulations, *i.e.*, reflashing of boot partitions or enabling of Android's debug bridge [61]. Weinmann discovered various issues within the compiled C code running on WWAN modem baseband firmwares [80]. Machiry *et al.* found multiple vulnerabilities in various Trusted Execution Environments (TEEs) which enable an attacker to gain full control of the host OS [51]. Similarly, Shen found kernel-level vulnerabilities in Huawei's HiSilicon TEE [68].

Pre-installed apps are very popular among vendors to introduce new functionalities and user interfaces to ease access to their services. In many cases these customisations severely collide with security requirements. For instance, researchers have analysed the inter-component communication (ICC) among

vendor apps with attention to numerous privilege escalation attacks and observed that the degree of vulnerability to such attacks correlates with the degree of vendor-based customisations [33, 35, 83]. Aafer *et al.* studied the distribution of dangling attribute references (*i.e.*, ICC accesses to destinations that do not exist among all device series) which they prevalently found in apps that were adapted to multiple device categories [1]. Building on the previous work, Aafer *et al.* performed differential analysis of inconsistent security configurations in custom Android ROMs and discovered severe vulnerabilities like private data exposures and privilege escalations [2]. Mitchel *et al.* got very similar results in their analysis performed with DexDiff and especially revealed the invasive Carrier IQ mobile intelligence software, a very common software package that provided comprehensive monitoring features for vendors and service providers [52].[1] Zhang *et al.* disclosed sensitive data residue in Android images after uninstallation of pre-installed apps [89].

Outdated and malicious OS releases are very common in the market. Vendors often try to reduce overall maintenance costs of a product by integrating profitable advertisement modules, shortening its life span through outdated initial firmware images and partial or complete denial of future firmware updates which actually leads to a huge OS fragmentation.[2] Thomas *et al.* examined predominant security flaws in different Android releases and their distribution between vendors, and found that each device on average was updated 1.26 times per year and 88% of all tested devices were exposed to at least one critical Android vulnerability at the time[72]. Zheng *et al.* tested 250 firmwares with 24,009 pre-installed apps and discovered that 7.6% of all images contained malware and 99.6% of firmwares suffer at least from one security issue. They further enlightened the distribution of the malware `CEPlugnew` with its geographical penetration and found that this threat primarily existed on low-cost devices. They concluded that "malware writers pay money to manufacturers of low-cost mobile devices to pre-install malware in their devices, or they release malicious firmwares with pre-installed malware to the wild" [93].

2.2 Operating System

Permission control is one of the key security features in Android's security architecture, however, various flaws arise through its complexity. Bagheri *et al.* distinguished three flaws regarding permission reservation, revoked URIs and permission re-delegation, in which each of these flaws could cause permission escalation attacks [10]. Conti *et al.* proposed CRePE, a system that extends the existing permission system by continuously adapting fine-grained permission policies during runtime [20]. Fragkaki *et al.* introduced secrecy and integrity policies, *i.e.*, permission flows among components that are either allowed or not, to the permission system [32]. Wang *et al.* proposed a framework called Compac that extends the existing permission system by linking permissions to components, hence they found a suitable methodology to effectively mitigate the risk of permission re-delegation among apps [78]. Subsequently, permission re-delegation and its remediation became a popular research topic [30, 92]. To identify unnecessary

[1] `http://www.computerworld.com/article/2499667/application-security/at-t--sprint-confirm-use-of-carrier-iq-software-on-handsets.html`

[2] `http://theunderstatement.com/post/11982112928/android-orphans-visualizing-a-sad-history-of`

permissions, Au *et al.* tried to extract Android's permission specification statically from source code and created the tool Pscout [7]. Bartel *et al.* elaborated on these extractions and proposed more sophisticated field-sensitive mappings that consider variable assignments for more precise flow graph creation in contrast to plain reachability analysis in Pscout [12]. Rasthofer *et al.* incorporated machine-learning approaches guided by a hand-annotated ground truth set with which they further improved accuracy of these mappings [63]. Xing *et al.* discovered that permission declarations introduced by developers themselves could be upgraded during Android platform upgrades [85].

Ren *et al.* found design flaws in Android's multitasking subsystem, named handling of tasks, *i.e.*, a collection of activities with which users interact to perform a job [65]. The flaws facilitated task hijacking attacks initiated by activation of the device's back button or a background service. Nadkarni *et al.* elaborated on the novel concept of system-wide sensitive data life-cycle confinement and developed a system called Aquifer that allows an application to retain control of data even after it was shared among other apps with support from a background kernel module [56]. Dietz *et al.* modified the Android platform to perform inter-process communication (IPC) tracking during execution, allowing a user finally the choice of operating with reduced privileges to increase security [24]. They further implemented a lightweight signature system that allowed any app to create and verify signed ICC calls facilitating originator verification mechanisms for ICC message receivers. Cooley *et al.* described activity spoofing attacks enabled by a background service which continuously monitors app launches [21]. As a mitigation, they proposed the concept of trusted activity chains that extends the Android Java framework to establish an exclusive interrupt lock.

Cao *et al.* analysed Android system service interfaces and identified 16 data validation vulnerabilities in these services [15]. Poeplau *et al.* investigated dynamic code loading of popular apps and found that 9.25% suffered from vulnerabilities caused by insufficient verification and filtering mechanisms [62]. Chen *et al.* proposed KARMA which enables live patching, *i.e.*, in memory patching of loaded vulnerable code, for Android kernel [17]. KARMA allowed patches to be written in LUA and was verified against several well-known exploits in native code which mostly had been mitigated with barely noticeable performance overhead. Mulliner *et al.* developed with PatchDroid a similar solution limited to managed app code [53].

2.3 Apps

Mutchler *et al.* studied several classes of vulnerabilities in a dataset of about 1M web apps, and found that 28% of these apps had at least one vulnerability [54]. Watanabe *et al.* classified Java libraries into official, private, and third-party; they studied the existence of several classes of vulnerabilities in each category, and found that third-party libraries were the most vulnerable ones. They further showed that at least 50% of paid and free vulnerable apps were actually vulnerable due to software libraries [79]. Li *et al.* studied the state-of-the-art work that statically analysed Android apps [45]. They found that much of this work supported detection of private data leaks and vulnerabilities, a moderate amount of research was dedicated to permission checking, and only three studies dealed with cryptography issues. Unfortunately, much state-of-the-art work did not publicly share their artefacts.

Ho *et al.* monitored native libraries by using a dynamic anomaly detection system called PREC with which they could mitigate 10 root exploits and substantially reduce the false positives over traditional malware detection algorithms [38]. Falsina *et al.* covered dynamic code loading threats with their tool Grab 'n Run, a drop-in Java library that securely loaded and verified external code [27]. Another approach was published by Schütte *et al.* with ConDroid, a tool that first performed static call path analysis to detect security-critical code sections and then analysed these sections by fully automated execution of every code path, enabling a thorough analysis of dynamically loaded code [66]. Vidas *et al.* tried to preserve Android developers from over-permissioning apps through an Eclipse plug-in that indicated unused permissions to the user [74]. On the task of estimating the prevalence of overprivileged permissions, Felt *et al.* developed Stowaway, a static byte code analysis tool that was able to detect overprivileged permissions [29]. Their test set consisted of 940 apps, in which they found about 33% overprivileged permissions. They concluded that developers try the least privilege principle, but fail because of bad documentation. Similar work was published by Backes *et al.* with their tool AppGuard that patched a security monitor into an app's installation file thus providing continuous permission checks during runtime without the need to change OS components [9]. Another major concern is being represented by weak cryptographic configurations, occurring in many different flavours, *e.g.*, usage of insecure cryptographic parameters. Egele *et al.* studied crypto implementation mistakes and found with CryptoLint that 88% of 11,748 apps made at least one mistake [25]. In 2014, Shuai *et al.* with regard to 13 generic crypto issues built CMA, a hybrid tool which uses static analysis for determination of crypto parts exploiting control flow and call graphs [69]. These parts were then run dynamically and they detected that more than 50% of all apps were vulnerable to man-in-the-middle (MITM) attacks. Arzt *et al.* recognized that many application developers are not cryptographic experts and thus introduced OpenCCE, an interactive Eclipse plug-in that allows developers to graphically assemble secure cryptographic configurations and export them to plain Java [5].

Fahl *et al.* discovered with MalloDroid that 8% (1,074) apps were vulnerable to potential MITM attacks caused by SSL certificate validation hacks resulting in potential data leakage or manipulation [26]. They also provided 41 attacks to verify their findings and performed an online survey revealing that half of 754 users were misinterpreting Android's SSL/TLS user-interface protection indicators, *i.e.*, users were not able to distinguish between secure and non-secure connections. Onwuzurike *et al.* found that almost 32% of the analysed apps contain partially wrong handling of SSL errors [59]. Similarly, Buhov *et al.* found missing certificate validation in 30% of all tested apps [14]. Zuo *et al.* investigated the soundness of WebView's certificate validation in 13,820 apps, unravelling that 4.7% of all tested apps suffer from insecure `onReceivedSslError` implementations [96].

Chin *et al.* used ComDroid for their work and found that 60% of tested real-world apps contained sensitive ICC data flows [18]. Lu *et al.* analysed with CHEX much larger datasets, their test set included 5,486 apps, and found 254 potential component hijacking vulnerabilities, *i.e.*, permission leakage, unauthorised data access and intent spoofing [50]. Wu *et al.* found that 17% of 7,190 apps in chinese markets were vulnerable to confused deputy attacks [82]. Zhongyang *et al.* developed a tool called DroidAlarm which identifies capability leak paths existed in unprotected files and network sockets [94]. Zhang *et al.* developed AppSealer to automatically create and deploy patches in byte code in order to mitigate several

component hijacking exploits [88]. The patch code represented on average 16% of the entire program and introduced merely 2% runtime overhead. Bugliesi *et al.* described π-Perms, a calculus for Android applications and provided the necessary plug-in Lintent, a security type-checker for Android Studio to verify and enforce modelled constraints that mitigate potential privilege escalation attacks [13]. Shao *et al.* developed SInspector tool that inspects data leaks in Unix domain sockets [67].

Li *et al.* investigated cloud service authentication issues and proposed Secomp that provides secure communication across different cloud platform providers, *i.e.*, Google Cloud Messaging and Amazon Device Messaging [46]. Similarly, Wang *et al.* used AuthDroid to find OAuth protocol implementation errors by analysing app code and network traffic, and found that 86.2% of all Chinese market apps, and 58.7% of all Google Play apps were vulnerable [75]. Jin *et al.* found that around 3% of PhoneGap apps are vulnerable to XSS-like code injection attacks caused by insecure `addJavascriptInterface` uses and developed NoInjection, a prototype to defend against these attacks [41]. Fang *et al.* were able to detect 37 input validation vulnerabilities in ICC including confused deputy as well as denial-of-service (DOS) [28].

3
App-level Security

In computer programming, the term *code smell* refers to symptoms in the code that could lead to a problem. Martin Fowler introduced several of such smells in his code refactoring book [31], and by then, many researchers studied the existence of various types of code smells. Particularly in Android, research on code smells has mostly focus on maintainability issues [37, 60], performance [36] or energy consumption [34].

In this thesis, we introduce the notion of *security code smells*, *i.e.*, symptoms in the code that signal the prospect of a security vulnerability, whereas security vulnerabilities are security issues that compromise user's security and privacy. Like traditional code smells, security code smells tend to exhibit technical dept, but with an explicit aspect of security. In this chapter we first briefly present common threats in Android-powered devices, and then introduce 28 security smells and associate their potential vulnerabilities to the attacks.

3.1 Threats

In this section we briefly explain major Android application level threats that we have seen during a state-of-the-art review.

- *Data exfiltration* threats are caused by stealing valuable data from a device through hardware or software connections. Hardware connections require direct access to the device. This issue can be mitigated by restricting physical access to the device and employing strong encryption mechanisms to protect data. Software connections, as used in man in the middle attacks (MITM), enable attackers

to eavesdrop and alter connections between two parties. The problem very frequently can be pinned down to certificate validation code that has been tampered with. In case certificates for potentially secure SSL connections are not validated accordingly, an attacker could replace them with forged ones without notice, enabling interception of all data transmitted. The attack surface can be reduced by using standard secure certificate validators and protocols. In addition, clickjacking, phishing, or spoofing attacks could cause data exfiltration threats as well. These attacks trick users with forged views. In Android, these can be of ordinary user interface elements or web content. Such forged layers are placed on top of existing benign ones, or arranged in another way so that users will get tricked.

- *Service inavailability* or system downtime threats are caused by denial-of-service (DoS) attacks to components, so they are forced into stalled state and cannot respond to further requests. Examples hereof are SMS validation flaws in Android's communication framework that lead to system reboots or temporal loss of data connections. Such threats originate from lack of proper input validation in system APIs, and can be resolved by adopting input validation strictly.

- *Miscellaneous risks* are caused by malicious apps or libraries that perform undesired operations. The impact varies from *privilege escalation e.g.*, by advertisement libraries that collect private information to creating *backdoors* for activating various sensors on the phone, *etc.*

3.2 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 finally identified 28 security smells that may lead to vulnerabilities in Android-powered devices. 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.

3.2.1 Insufficient Attack Protection

This category lists smells related to development decisions whose security impacts are essentially undermined by developers.

- **Unreliable Information Sources**
  Developers acquire their programming knowledge from various sources such as official documentations, books, crowd sources, *etc. Issue:* According to recent research, developers increasingly resort to studying code examples provided by informal sources like StackOverflow, which are easy to access and integrate, but often lack security concerns [3]. *Consequently*, vulnerabilities could make their way into apps in the absence of security expertise. *Symptom:* Existence of copy-pasted code from untrustworthy sources. Tools already exist that assist in detection of duplicated code,[1] however, they still require manual code snippet collections from external repositories. *Mitigation:* Use official sources which are more reliable, and vet the security of any external code before and after integration in your code.

- **Untrustworthy Libraries**
  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.*, introduce vulnerabilities and compromise user data [79]. *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 [23]. *Mitigation:* Solely use reliable libraries that are not known for any vulnerabilities [8], *e.g.*, not reported in any vulnerability database like CVE.[2]

- **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. *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 analyse; there is no distinction between code and data at the native level, and attackers can load and execute

[1] `http://pmd.sourceforge.net/pmd-4.3.0/cpd.html`
[2] `https://cve.mitre.org/`

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 [77]. *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 [43]. *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 [16]. *Symptom:* No technique (*e.g.*, watermarking, 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 Permissions**
  The use of protected features on Android devices requires explicit permissions, and developers occasionally ask for more permissions than necessary [70]. *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 [71]. *Symptom:* The manifest file contains permissions for APIs that are not used. The large number of different Android API levels and the incompletely documented mappings between permissions and method calls increase the complexity of the detection. *Mitigation:* Utilize tools like PScout[3] to exclude from the manifest file any permission whose corresponding API calls are absent in the app.

3.2.2 Security Invalidation

The smells within this category are basically due to misusing security features that in the end invalidate a desired security.

- **Weak Crypto Algorithm**
  The fundamental set of cryptograph 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. Up-to-date recommendations for encryption algorithms can be found on the Open Web Application Security Project (OWASP) website.[4] *Mitigation:* Consult the state of the art guidelines to choose an appropriate cryptography, and utilize expert systems [5].

- **Weak Crypto Configuration**
  The majority of security breaches come from exploiting developer's mistakes. *Issue:* Cryptography

[3] `http://pscout.csl.toronto.edu`
[4] `https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet`

APIs are widely perceived as being complex with many confusing options [55]. *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 1000 iterations, short keys and salts, or none 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 [4].

- **Unpinned Certificate**
  Digital certificates are needed to ensure secure communication. Unpinned certificates, *i.e.*, certificates not stored in any secure local key store, are easy to maintain and are frequently used in the appified world [58]. *Issue:* Ensuring the authenticity of a certificate is non-trivial, if it is not pinned. In such cases someone has to trust potentially compromised certificate authorities. *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 is always recommended to increase the security.[5]

- **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 certiifcate is self-signed, the OS devolves this validation process to the app itself. However, developers often fail to implement it properly [26]. *Consequently*, this leaves the communication channel over SSL/TLS insecure and susceptible to man-in-the-middle attacks [19]. *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 which 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"[6] 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

[5] Since Android 6.0 pinning can be enabled using the `Network Security Configuration` feature.
[6] `https://github.com/google/nogotofail`

checks, is neglected. *Consequently*, the risk of distributing a vulnerable app increases especially when the app utilises uncertified libraries, or in a worse case, an attacker can replace installation packages with malicious ones [93]. *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.

3.2.3 Broken Access Control

This category contains smells that arise when one party in a communication trusts the other party without any checks, *i.e.*, not enforcing any authentication nor authorisation controls. Especially insufficiently secured communication protocols, *e.g.*, used by ICC or debug interfaces, that are exposed to external code are prone to these kinds of smells.

- **Unauthorised Intent Receipt**
  An *intent* is an abstract specification of an operation that apps can use to utilise 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 [18]. *Symptom:* The existence of an intent with sensitive data (*e.g.*, tokens, credentials, *etc.*), but without a particular component name (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 provider, 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 access those operations [22, 81]. *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 [44].

- **Unprotected Unix Domain Socket**
  Android IPCs do not support cross-layer IPC, *i.e.*, communication between an app's Java and native processes/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 [67]. *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 Capabilities**
  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 [47]. *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 [39]. *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 they are also prohibited since Android 6.0.

- **Debuggable Release**
  During app development there exist two major build configurations, 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 [86]. *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 [40]. *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 (a.k.a. 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 tokens or other sensitive information [76]. Although custom schemes simplify some user interactions, these should be avoided where possible, as they increase the attack surface. *Symptom:* The registration of a URI scheme within the `intent-filter` in manifest file. The `SchemeRegistry.register` method is in the code. *Mitigation:* Adopt the dedicated system scheme *i.e.*, `Intent` which is harder to compromise.

3.2.4 Sensitive Data Exposure

Smells in this category are related to suffering from data protection mechanisms that prevent the disclosure of sensitive data to unauthorised parties.

- **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 [76]. *Symptom:* Calls like `HttpGet.addHeader()` are present in the code to store private data. *Mitigation:* Do not store sensitive data in headers, instead rely on dedicated mechanisms like OAuth2 protocol[7] 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. *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 user, but only within the same app identity.

- **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 [90]. *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 [57]. *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. 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. 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 esp. externally) stored sensitive data, and place the encryption key in KeyStore, protected with a user password that is not stored on the device.

[7] `https://oauth.net/2`
[8] `https://developer.android.com/guide/topics/data/data-storage.html`

- **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 [59]. *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 app from any usage of cleartext traffic.

- **Exposed Credentials**
  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. *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 [49]. *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 [91]. *Consequently*, such so-called "data residue" can be associated to another app and empower adversaries to access sensitive information [89, 91]. *Symptom:* The app calls system services that are known to be subject to data residue problem. *Mitigation:* Unfortunately, an app may not always be aware of its data being stored in system services, and the mitigation is to avoid sharing private data with these services, if possible.

3.2.5 Lax Input Validation

The smells listed in this category originate from the lack of input validation which consequently let a malicious input (*i.e.*, code or resources) compromise security.

- **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 like Chrome, FireFox, *etc.* which 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 [41]. *Symptom:* The `setJavaScriptEnabled` call with value `true` which 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 system. *Issue:* WebView provides an API, `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 [41]. *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 [62]. 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 [27]. *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 [73]. Analyse your app with the help of tools like *Grab 'n Run* [27].

- **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. *Symptom:* Inputs from untrustworthy sources are passed to the database without proper validation. *Mitigation:* Instead of dynamic SQL 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
Empirical Study

In this chapter we present the lightweight tool we developed to statically identify the presence of several security smells in the code. We apply the tool to two different repositories of benign and malicious apps and study the prevalence of the smells. We also manually inspect some apps to assess the performance of our lightweight analysis tool and the extent to which identifying smells indicate the presence of security vulnerabilities.

4.1 Prototype Implementation

We developed a lightweight analysis tool that statically detects 10 kinds of security smells in an app. We rely on the Apktool to reverse engineer Android app package installation (apk) files and generate Smali output that can be understood as human-readable Java bytecode.[1] We defined a set of rules to capture the symptoms of each security smell. In particular, we utilize a 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.

4.1.1 Tool Architecture

The tool itself is built upon Java 8 and takes the desired app's apk file as input parameter. The *Main* class shown in Figure 4.1 is the entrypoint and provides the setup for the analysis. It first decompiles the app

[1] `https://ibotpeaches.github.io/Apktool`

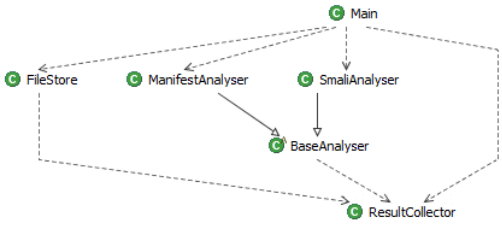


Figure 4.1: Composition of our lightweight tool

Security Code Smell	ManifestAnalyser	SmaliAnalyser
Debuggable Release	X	
Custom Scheme Channel	X	X
Header Attachment		X
Unique Hardware Identifier		X
Exposed Clipboard		X
Insecure Network Protocol		X
Improper Certificate Validation		X
Dynamic Code Loading		X
XSS-like Code Injection		X
Broken WebView's Sandbox		X

Table 4.1: Analyser class dependencies of the 10 detected smells

with Apktool before it initialises individual analysers. We have analysers for manifest and Smali code files, and each collects its data in a *ResultCollector*. Finally, the *FileStore* class stores the *ResultCollector* data in a persistent store. In Table 4.1 we present the dependencies of each security smell on each analyser.

4.1.2 Smali Code

The Android Dalvik executable format is not easily analysable,[2] thus inappropriate for our analysis. Instead, we worked with the Smali intermediate (IR) textual code representation, retrieved from the DEX code decompilation. This IR representation was superior to analyse with respect to its internal structure, *i.e.*, it is based on plain and human readable text rather than on byte code, and the translation even performed almost in linear time. Another benefit is the language's comprehensiveness, each command in bytecode[3] has a counterpart in Smali ensuring maximum compatibility. We show the single letter type identifiers used by Smali in Table 4.2 to foster interpretation of provided Smali code samples.

Listing 1 shows basic semantics of Smali code for a method invocation. The first element represents the bytecode operation for the Dalvik virtual machine (VM). It is followed by specific source or destination register pointers used in the register-based VM assigned by the specified bytecode operation. Next, a type

[2] `https://source.android.com/devices/tech/dalvik/dex-format`
[3] `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`

Identifier	Assigned Type	Primitive Type?
V	void (only return types)	yes
Z	boolean	yes
B	byte	yes
S	short	yes
C	char	yes
I	int	yes
J	long (64 bits)	yes
F	float	yes
D	double (64 bits)	yes
L	object	no

Table 4.2: Type declarations in Smali

and fully qualified Java class define the class affected by the Dalvik command. Subsequently, we have two different syntaxes, either the object's instance name together with a type and fully qualified class to prepare and load the specified object for the latter method call placement, or the method name combined with input and return parameters for the actual method invocation, yet again potentially consisting of fully qualified classes with their types. Furthermore, Smali identifiers and other features we did not use in our detection can be found in some listings. It is particularly interesting to note that in case of primitive types only the single letter identifiers are used without any separators, and arrays are represented by one or more opening square brackets according to the dimension, *e.g.*, an int[][] array would be defined as `[[I`.

Listing 2 and Listing 3 both represent the very same behavior. As we spot in the Smali code, the method names, try-blocks and object relations were maintained, allowing us to apply regular expression patterns to Smali code. Therefore, in Listing 3 line 5 loads the reference to the callee object in parameter register `p0` into value register `v7`. Line 6 calls the method `java.lang.String getDeviceID()` on the `TelephonyManager` reference in `v7`. Line 7 moves the result of the method call from the working register into the value register `v7`. Finally, line 8 calls the static method `IoHandler.saveData` with the two parameters `v6` and `v7`. We can further see the used types, *i.e.*, L for object types.

```
DalvikCommand {Register1, ..., RegisterN}, [Type][Packages/ClassName];->[[ObjectName:]
    ↪ [Type][Packages/ObjectClassName];] | [[method(ParameterType1;...;ParameterTypeN
    ↪ ;)][ReturnType][Packages/Classname];]
```

Listing 1: Smali code structure for method invocations

Smell	**Symptom in Manifest** and **Qualified Entities**
Debuggable Release	Debuggable attribute is set in manifest `application.android:debuggable = true`

Custom Scheme Channel	Scheme channels are registered in manifest
	`application.*.intent-filter.data.android:scheme`
	`↪ = "SCHEME_HERE"`
	(replace `*` with `activity|activity-alias|service|receiver`)

Table 4.3: XML elements and attributes inspected in our tool

```
public List<CellData> getAllCellDataCompatible(boolean saveCollectedMeasurements) {
...
try {
TelephonyManager tel = telephonyManager;
IoHandler.saveData(s.toString(), tel.getDeviceId());
} catch (Exception e) {}
...
}
```

Listing 2: Sample Java code of a traditional code smell in Android applications

```
.method public getAllCellDataCompatible(Z)Ljava/util/List;
...
:try_start_0
...
iget-object v7, p0, Linfo/smapper/smapper/logic/DataFetcher;->tel:Landroid/telephony/
    ↪ TelephonyManager;
invoke-virtual {v7}, Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/
    ↪ String;
move-result-object v7
invoke-static {v6, v7}, Linfo/smapper/smapper/logic/IoHandler;->saveData(Ljava/lang/
    ↪ String;Ljava/lang/String;)V
:try_end_0
.catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0
...
.end method
```

Listing 3: Java code from Listing 2 translated to Smali

4.1.3 XML Parser

Android's meta-information files (*i.e.*, manifest, layout resources and definitions) heavily rely on XML and were easily analysable through the standard Java XML libraries. As an additional benefit by using these reliable libraries we could increase the tool's robustness against the diverse XML structures that especially exist in malign apps, *e.g.*, issues with regard to incompletion or misused attributes. Mixed lower and upper

case definitions required us for compatibility reasons to normalise all strings to lower case. Several XML attributes that our tool inspects are presented in Table 4.3.

4.1.4 Regex Matching

We applied straightforward regular expression (regex) matching using Java's built-in regex implementation. Some of the smells mentioned in this section also relied on results from the *ManifestAnalyser*, *i.e.*, *Custom Scheme Channel*, or involved multiple method calls, increasing the overall complexity. Several regular expressions are noted in Table 4.4.

4.1.5 Website Data Scraping

Finally, to interpret data and get better insight about characteristics of each app we mined some meta-data features from Google Play. Table 4.5 shows the features and their corresponding regex statements. The scraper constructed for each app the URL according to a specific pattern,[4] downloaded the HTML page and matched it with regular expressions.

4.2 Study Design

The test subjects consisted of benign and malign Android app packages. An overview of our *app inspection workflow* is illustrated in Figure 4.2. We started by downloading apps from AndroZoo and VirusShare (steps 1 and 2) and filtered the latter ones, because the VirusShare packages were not only for Android apps (step 3). Next, we uploaded them onto the compute cluster of our university called Ubelix and let the analysis run (step 4). Once the process finished, we downloaded the results and exported them into a local database, simplifying the evaluation. This database allowed us to dynamically export tables containing the significant values for further analysis with GUI-based tools (step 5).

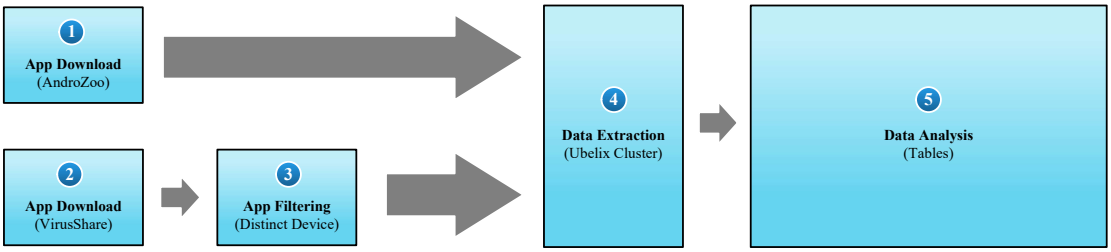


Figure 4.2: The workflow of our analysis with its different stages.

[4] `https://play.google.com/store/apps/details?id=my.packagename.here&hl=en`

Smell	**Symptom in Smali Code** and **Corresponding Escaped Regexes**

Custom Scheme Channel	Scheme registration code exists
	`Lorg/apache/http/conn/scheme/SchemeRegistry;->`
	`↪ register\(Lorg/apache/http/conn/scheme/Scheme;\)`
	`↪ Lorg/apache/http/conn/scheme/Scheme`

Header Attachment	Header attachment code exists
	`Lorg/apache/http/client/methods/HttpGet;->`
	`↪ addHeader\(Ljava/lang/String;Ljava/lang/String;\)`
	`↪ V`

Unique Hardware Identifier	Hardware identifier access code for MACs and IMEI exists
	`Landroid/telephony/TelephonyManager;->`
	`↪ getDeviceId\(\)`
	`↪ Ljava/lang/String`
	--- 	
	`Landroid/bluetooth/BluetoothAdapter;->`
	`↪ getAddress\(\)`
	`↪ Ljava/lang/String`
	--- 	
	`Landroid/net/wifi/WifiInfo;->`
	`↪ getMacAddress\(\)`
	`↪ Ljava/lang/String`

Exposed Clipboard	Clipboard manipulation code exists
	`Landroid/content/ClipboardManager;->`
	`↪ getPrimaryClip\(\)`
	`↪ Landroid/content/ClipData`
	--- 	
	`Landroid/content/ClipboardManager;->`
	`↪ setPrimaryClip\(Landroid/content/ClipData;\)`
	`↪ V`

Insecure Network Protocol	Http connection establishment code exists
	`Ljava/net/HttpURLConnection;->`
	`↪ <init>\(Ljava/net/URL;\)`
	`↪ V`

Improper Certificate Validation	Customised certificate validation code exists `.implements Ljavax/net/ssl/X509TrustManager;`

Dynamic Code Loading	Dynamic code loading mechanism exists
	`Landroid/content/Context;->`
	`↪ createPackageContext\(Ljava/lang/String;I\)`
	`↪ Landroid/content/Context`

XSS-like Code Injection	WebView JavaScript setting code exists
	`Landroid/webkit/WebSettings;->`
	`↪ setJavaScriptEnabled\(Z\)`
	`↪ V`

Broken WebView's Sandbox	WebView Java interface code exists
	`Landroid/webkit/WebView;->`
	`↪ addJavascriptInterface\(Ljava/lang/Object;`
	`↪ Ljava/lang/String;\)V`

Table 4.4: Regular expressions used in our tool containing Smali type identifiers as shown in Table 4.2

Feature	**Description** and **Escaped Regular Expression**
App Name	The name of the app `\"id-app-title\" tabindex=\"0\">([^<]+)<`
Category	The app's assigned Google Play Store category `\(\\w*)\"> <span itemprop=\"`
Contact Mail	Mail address of the app's developer `mailto:([^\"]+)\"`
Content Rating	Maturity rating of the app `itemprop=\"contentRating\">([\\w\\s,\\.-]+)<`
Current Version	Current version number of the app `itemprop=\"softwareVersion\">([\\w\\s,\\.-]+)<`
Last Update Date	The last time an update was released for the app `itemprop=\"datePublished\">([\\w\\s,]+)<`
Number of Downloads	The amount of downloads an app received, also referred to as popularity in our work `itemprop=\"numDownloads\">([\\w\\s,\\.-]+)<`
Offered by	The app's developer name, more precisely an individual or company `Offered By </div> <div class=\"content\">([^<]+)<`
Required Version	The Android API level an app requires `itemprop=\"operatingSystems\">([\\w\\s,\\.-]+)<`
Star Rating	An app's star rating provided from end users `Rated ([0-9.]+) stars out of five stars`

Table 4.5: Features and escaped regex strings thereof extracted from apps Google Play Store pages

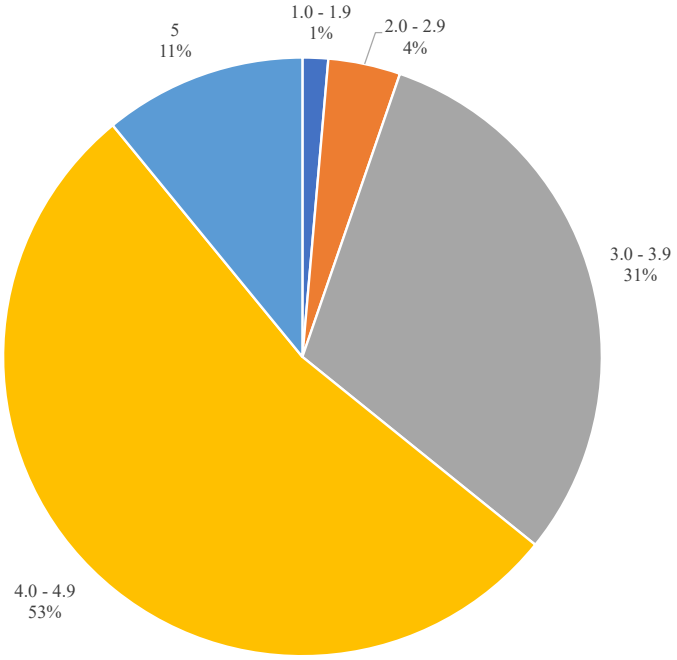


Figure 4.3: Distribution of apps' star ratings

4.3 App Collection

For our *benign dataset* we randomly selected our apps from the AndroZoo dataset[5] and downloaded them with a small Java tool using the AndroZoo publicly available index file as reference. 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 meta data information such as an app's category, its number of downloads, update cycle, and star rating we still needed to parse information from the Google Play website. Unfortunately, we could not access 25,000 apps for various reasons, for example, because they were no longer available on the store, or they were not accessible from Switzerland. In the end, we included 46,000 APK files of benign apps in our dataset with a total size of about 440GB. About 90% of these apps were released within 2014 to 2016, a quarter of apps was updated in less than 3 months, the median size of an apk was 5.5MB, the majority of the apps were rated with 4 or more stars (Figure 4.3), and slightly more than 26% of apps were downloaded more than 50,000 times from the official Google market (Figure 4.4).

For our *malign dataset* we relied on the VirusShare database.[6] Although the interface was not as sophisticated as in AndroZoo, we were able to retrieve around 81GB of data. However, we suffered from several issues leading to a more complex retrieval and separation process:

[5] `https://androzoo.uni.lu`
[6] `https://virusshare.com/`

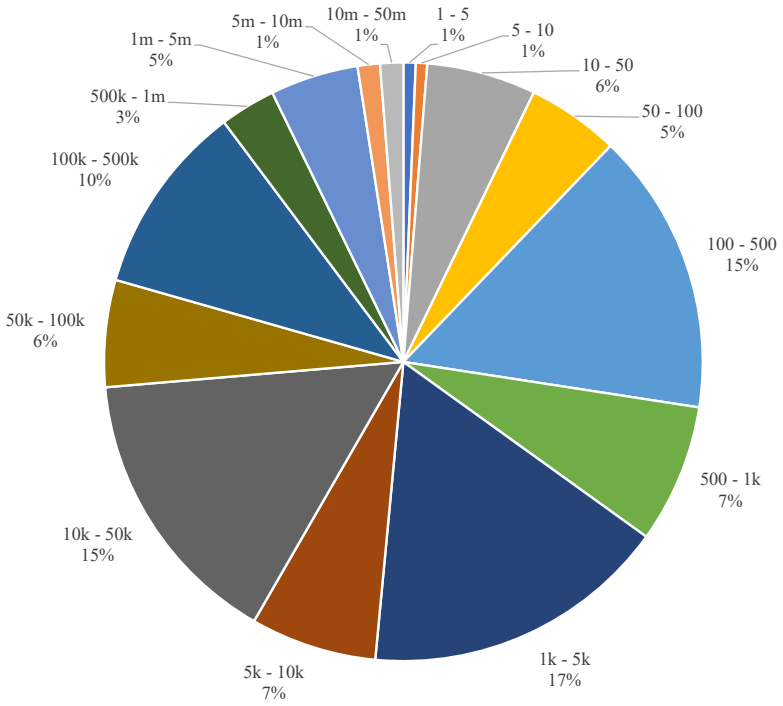


Figure 4.4: Distribution of apps' popularities

- *No direct HTTP or HTTPS downloads for archives*, but only BitTorrent, hence we had to set up a distinct machine with fast and unrestricted internet access. We used Vuze,[7] a popular free BitTorrent client, to download and share the encrypted collections.
- *No meta-data summaries were available*, thus we had to extract the meta-data (*e.g.*, package name, targeted Android version, app version number and many more) ourselves during the analysis in the cluster.
- *Explicit Android malware collections have been released until early 2014*, however, we were also interested in more recent malware samples. Therefore we had to download more recent ordinary collections containing malware for other platforms (*e.g.*, Windows, Linux, Apple macOS) as well.
- *Separation of Android malware files is not a trivial task*, because the risk of infection is omnipresent while working with such files. Hence, we had to separate them by parsing the malware files on a distinct device without any network access or window manager as a protective measure.

4.4 App Analysis

Although our analysis is light-weight, the sheer number of apps we analysed required parallel processing. However, we allocated much more resources than our tool actually needed as precautionary measure,

[7] `https://sourceforge.net/projects/azureus/`

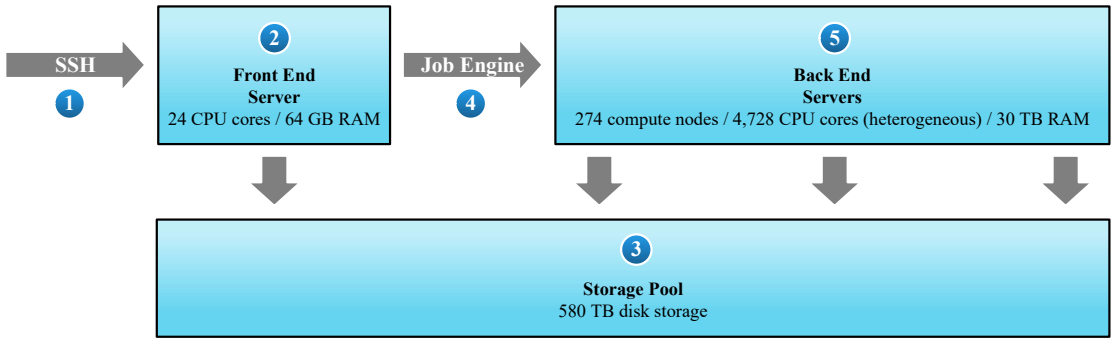


Figure 4.5: The architectural view on Ubelix, the compute cluster.

because the precise runtime requirements for each app were not available in advance. As a benchmark, the analysis of the "Facebook" app required 536 seconds on a traditional notebook PC with dual core Intel Core i7-4600U CPU and 8 GB of RAM. The university of Bern Linux *compute cluster* (Ubelix) provided resources for our static code analysis; maintaining many compute nodes with an architecture as shown in Figure 4.5. The system provided user-level access over SSH remote console (1) to a front end server for orchestrating the cluster (2). The front end server users home directory resided on the storage pool that was collectively used by front and back end servers (3). This server further ran the cluster management software (*i.e.*, Slurm[8]) with its job engine distributing compute jobs automatically among available nodes (4). These nodes then started working on the assigned jobs, as soon as they finished their remaining queues and got available. The results were afterwards written to the shared storage pool and ready for later retrieval through the SSH connection.

During our work we encountered some issues regarding the complex infrastructure of Ubelix:

- *Uploading huge data*, *i.e.*, 985GB, took much time and needed a fast and stable connection.
- *Change of workload manager* from Grid Engine to Slurm was forced by cluster operators and required modifications of job scripts and our custom job manager.
- *Outdated software packages* were installed on Ubelix, hence we had to compile and integrate our own packages (*e.g.*, Java virtual machine).
- *Restricted user-rights* caused serious issues with some tools expecting additional system applications available on the environment.
- *Best-effort assignment* of resources unexpectedly delayed transactions when other projects were causing heavy loads on the system.
- *Down-time* through periodical maintenance caused aborted experiments that had to be rerun.

[8] `https://slurm.schedmd.com/`

- *Size and quantity restrictions* on the storage pool forced us to implement additional routines for instantly cleaning up finished analyses.
- *Job queue length restrictions* enforced us to implement our own job monitor that was able to dynamically enqueue jobs below the threshold. Moreover, it was able to cope with system failures due to down-time by reuploading skipped jobs.

A traditional *Slurm Job Script* as we can see in Listing 4 configures the cluster node according to our specifications. We specified the maximum runtime on the cluster (*i.e.*, 45 minutes), the total amount of available RAM (*i.e.*, memory per CPU multiplied by CPU cores, thus we assigned 20GB in total), amount of CPU cores (*i.e.*, 8) and paths for storing standard and error messages into the same file. The last line starts the execution of our analysis tool through a Java 64-bit virtual machine. These resources were not always available as various other projects shared the same infrastructure.

```
#!/bin/bash
#SBATCH --time=00:45:00
#SBATCH --mem-per-cpu=2500M
#SBATCH --cpus-per-task=8
#SBATCH --job-name="executor_analysis_r001_000000001"
#SBATCH --error='logs/r001_000000001_sample.apk.log'
#SBATCH --output='logs/r001_000000001_sample.apk.log'
../../jdk1.8.0_101/bin/java -d64 -Xmx20g -jar Executor.jar './apks/sample.apk'
```

Listing 4: Sample job script for SLURM, ready to be assigned to the compute cluster with the command `sbatch`.

4.5 Results

We applied our lightweight tool to all apps in the dataset. Figure 4.6 shows how prevalent the smells are in our dataset, and in Subsection 4.5.5 we discuss the quality of the results. A majority of apps potentially suffer from XSS-like code injection (85%) followed by dynamic code loading (61%). About 40% use custom scheme channels and expose a unique hardware identifier. More than 12% use an insecure network protocol, and almost 11% 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 (see Figure 4.7). Only 9% of apps are free of 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 as the proportion of devices running different API versions varies. Figure 4.8 shows the distribution of smells within each API level. We noticed that the prevalence of *Debuggable Release* has been dramatically reduced. We believe this is mainly due to the fact that Google market no longer accepts apps in debug mode. We conjecture

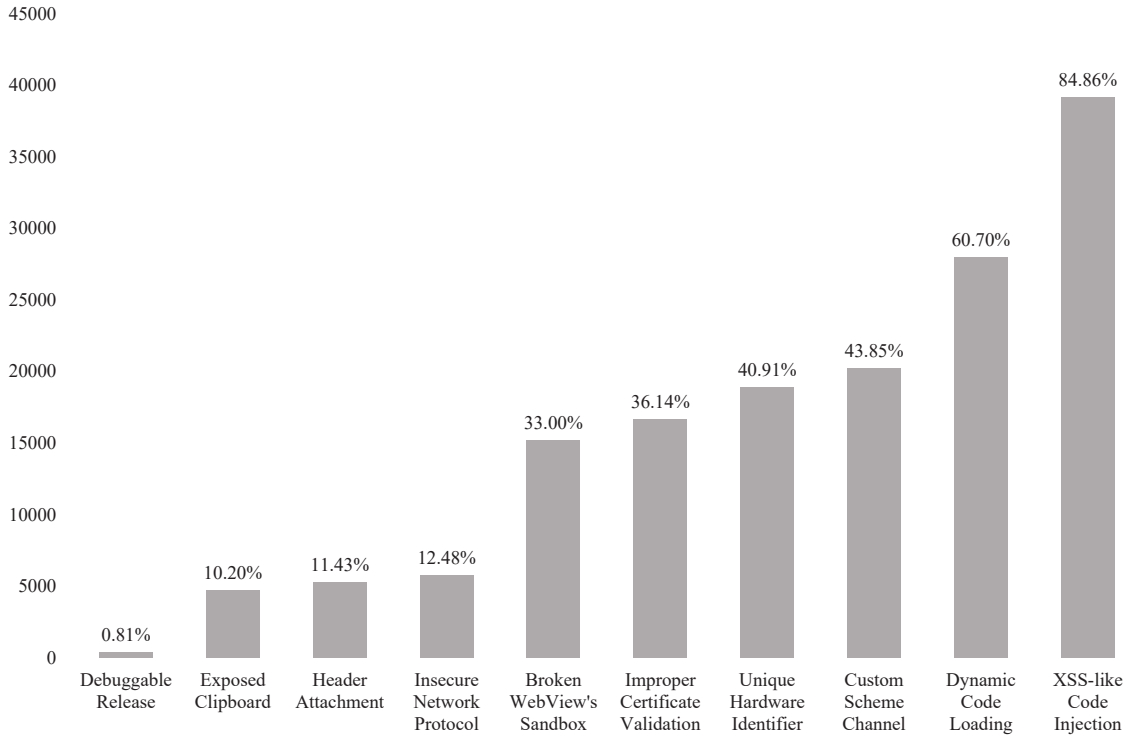


Figure 4.6: Distribution of security smells in the apps

this issue should have decreased also in other markets without this constraint as recent build platforms automatically disable the debug mode in the signed release version. 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 4.9 shows how many of these classes of smells appear within each API level. 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, which introduced amongst other things a social stream API in the Contacts provider enabling additional social media interactions for users.

In the remainder of this section we discuss our findings from a few more perspectives, since we collected more meta-data from the apps and Google's Play store, we are able to discuss other interesting correlations as well.

4.5.1 Category

Figure 4.10 shows the number of different security smells appearing in the apps in each category. 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

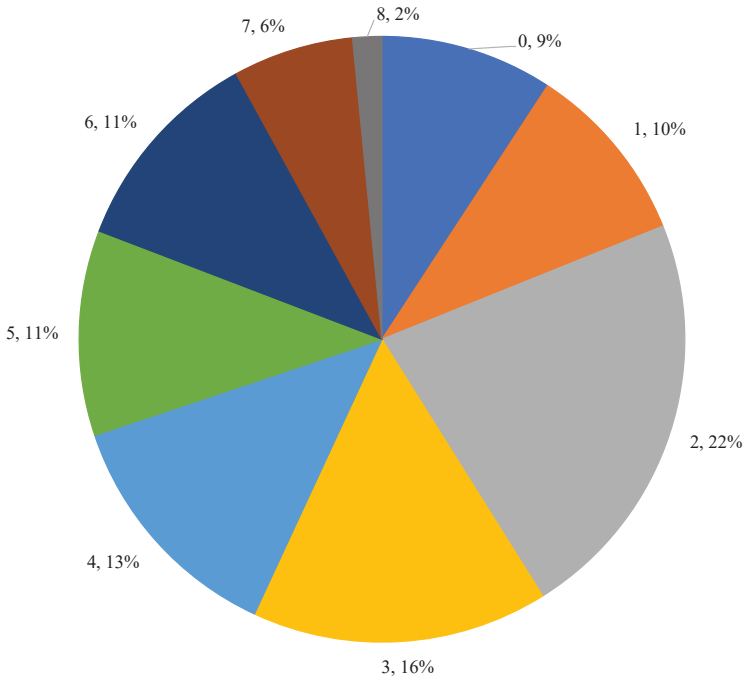


Figure 4.7: Partitioning apps by number of security smells

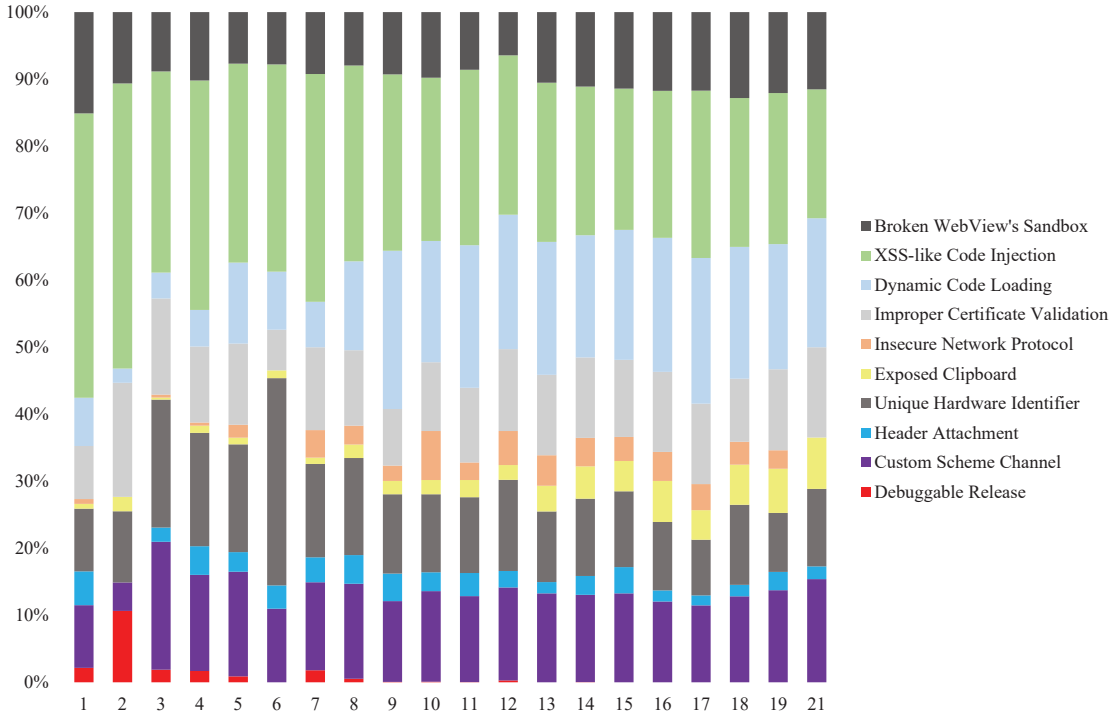


Figure 4.8: The distribution of security smells within each API level

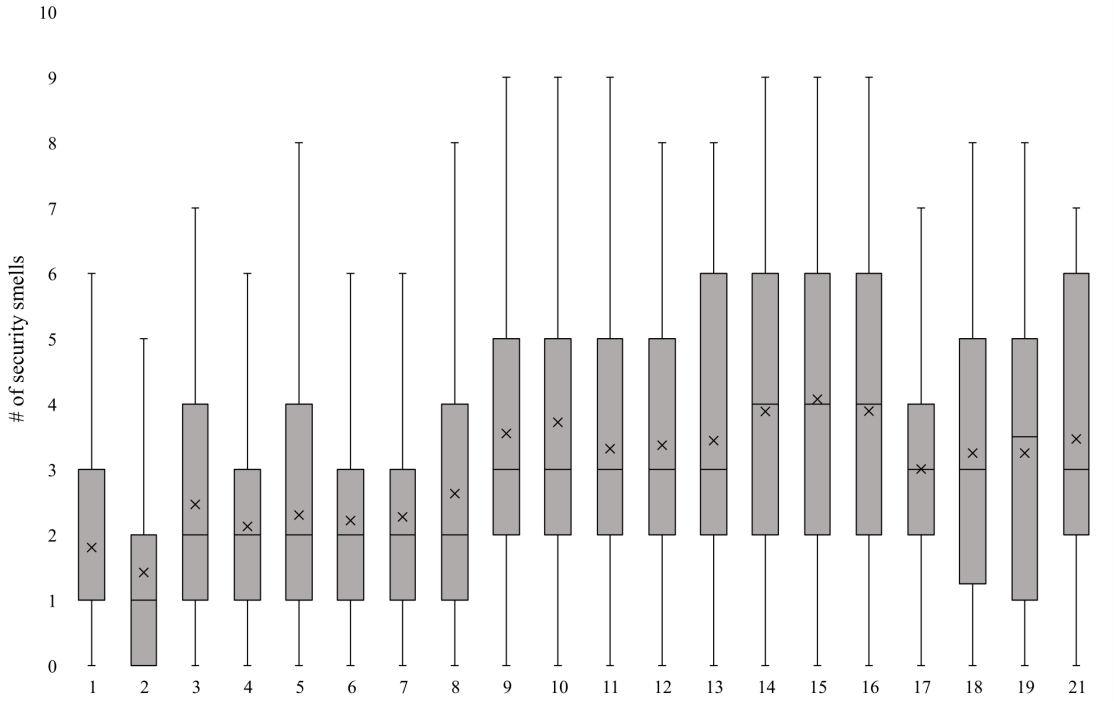


Figure 4.9: Number of smells within each API level

smells. We assume that Dating apps are not very restrictive when it comes to user privacy, as Dating apps track user's movements and various system sensors including cameras. Food and Drink apps, often published from franchises like McDonald's or Burger King, are surprisingly vulnerable as we believe this has been caused by the intense advertisement, tracking and online functionality they commonly use in current releases.

4.5.2 Popularity

Figure 4.11a shows the relationship between the number of downloads and the security smells. 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. However, it appears that we can observe a very interesting pattern: Up to 500M downloads apps begin to use more features, thus reducing protection, before the security finally increases when the apps get mature with more than 500M downloads. This pattern gets even more noticeable when working with smaller groups on the x-axis. We further noticed these apps vary in size, and there is no relationship between the number of downloads and the size of apps.

Figure 4.11b shows the relationship between the number of security smells and star ratings. 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

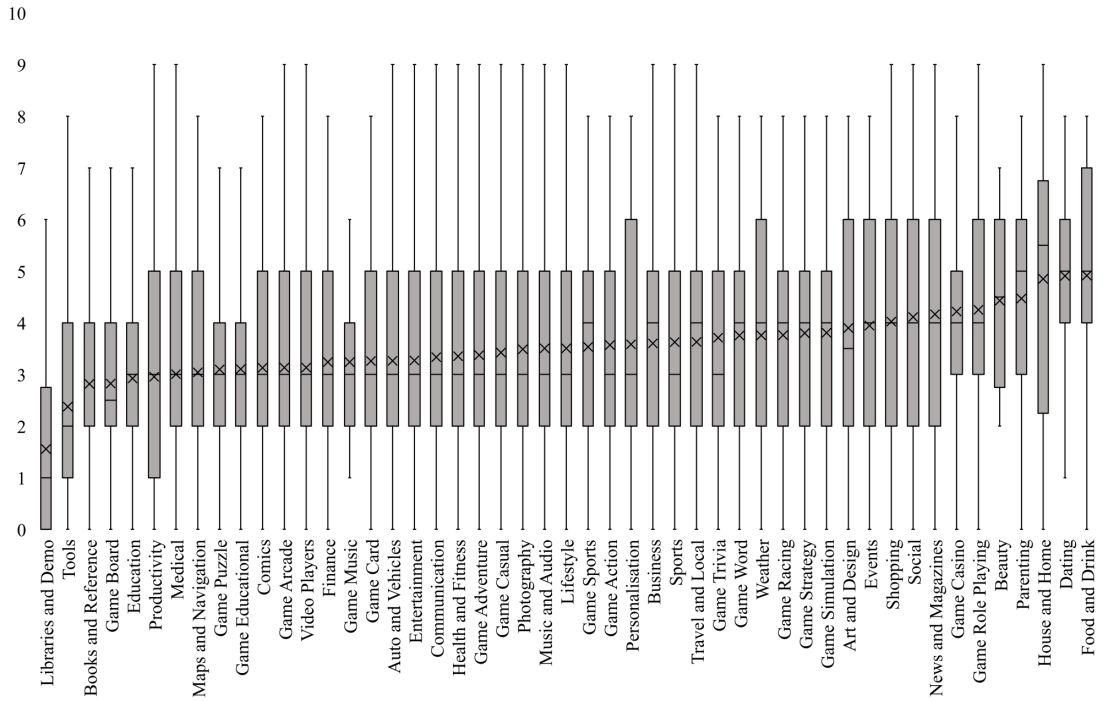
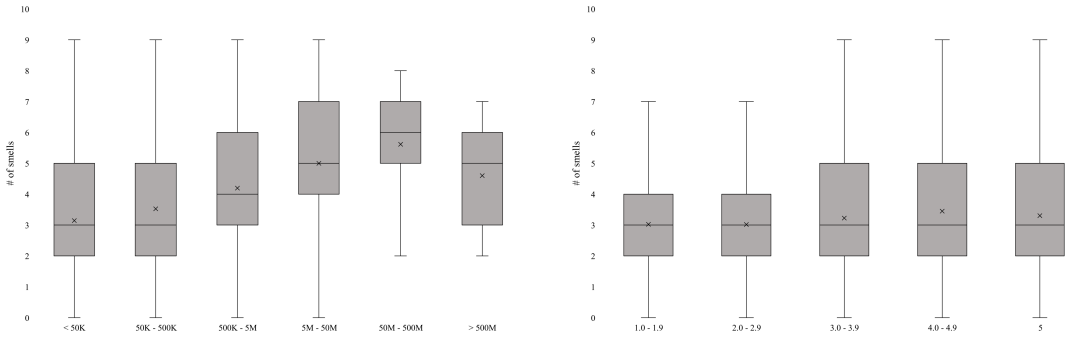


Figure 4.10: Distribution of smells in app categories



(a) (b)

Figure 4.11: A set of two subfigures: (a) The relationship between number of smells and number of downloads; (b) The relationship between number of smells and app star ratings

barely noticeable by end-users, hence they are not reflected in the ratings.

4.5.3 Release Date

We further studied whether the prevalence of security smells changes over time. In fact, with advances in developer 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 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.

4.5.4 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, 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 because they contain other resources such as image, video and audio content.

4.5.5 Manual Analysis

To assess how reliable these findings are to detect security vulnerabilities, we manually analysed 160 apps. For each smell, we inspected 20 apps manually and compared our findings to the result of the lightweight analysis tool. We did not consider any false negatives, because their evaluation would require manual large-scale analysis of random apps not reported by our tool. As is shown in Figure 4.12, the results were encouraging. The manual analysis completely agreed with the tool in the security risk associated with six security smells. In the case of the exposed clipboard smell 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 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.

4.5.6 Malicious Apps

We conjecture malicious apps should have more smells as their developers are incautious about user's security. Therefore, we also run our tool on a dataset of malicious apps to grasp knowledge about the dissimilarities between benign and malign code. Figure 4.13 shows our app corpus' API level distribution in which the benign corpus experiences a wider API level support due to its larger and more consistent

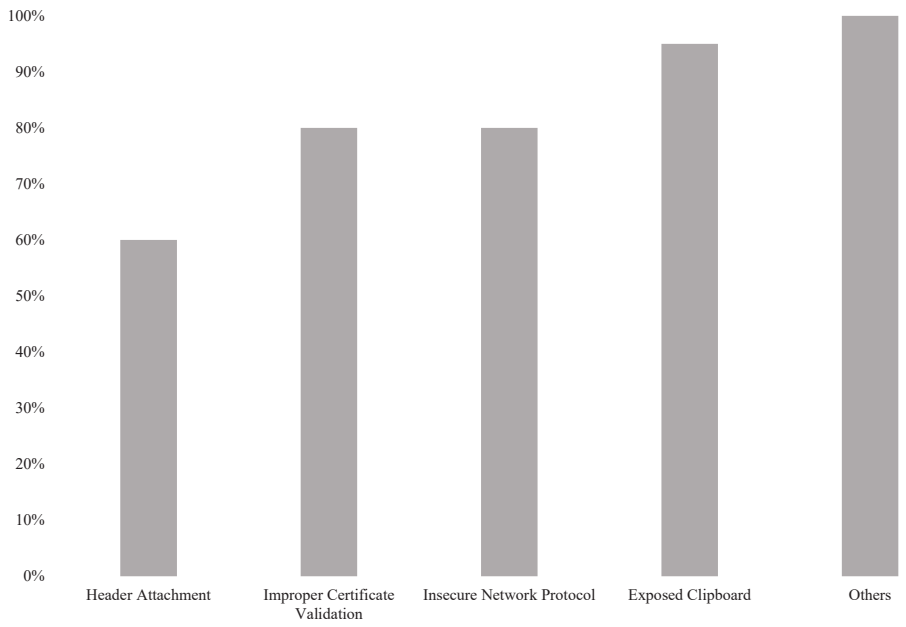
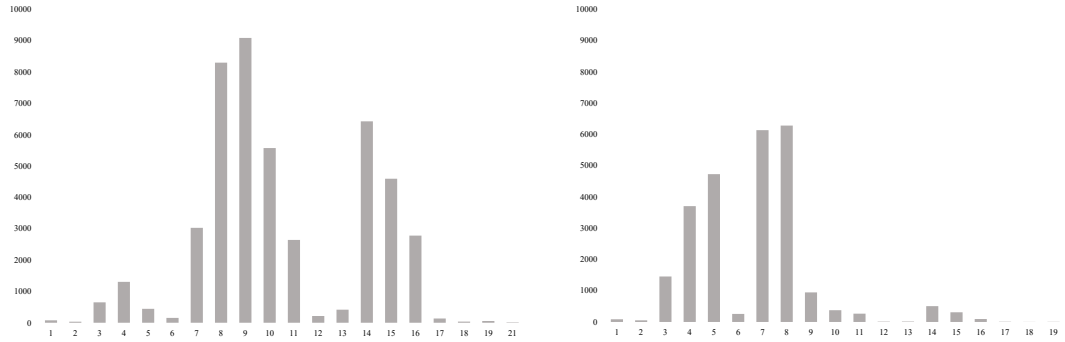


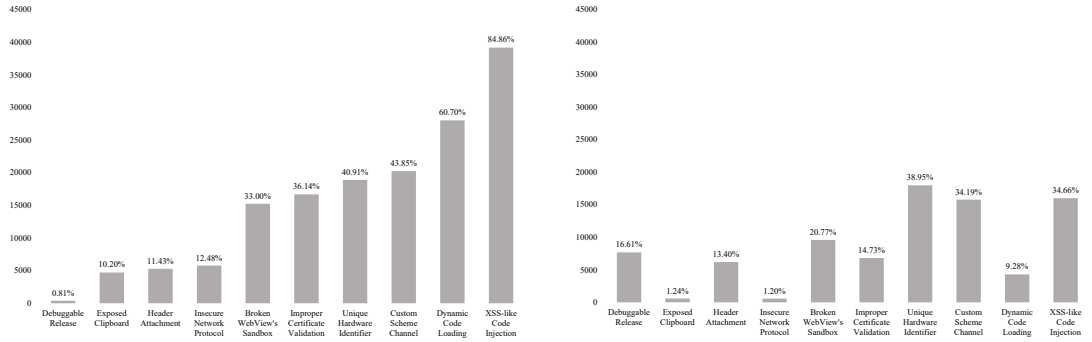
Figure 4.12: The precision of obtained results

dataset, while the malign datasets suffers a skew towards older API levels. Figure 4.14 reveals differences of security smell prevalences between both corpora. Whereas benign apps suffer heavily from XSS-like code injection, dynamic code loading and custom scheme channel, malign apps suffer especially from unique hardware identifier. This could be a side effect of the premium call or SMS misuse other researchers reported [6] that requires the very same permissions and classes. Surprisingly, dynamic code loading is much less prevalent in malicious apps, while the debuggable release appears more often in malign code. Similarly, Figure 4.15 presents the distribution of smells across API levels. It is interesting to see that exposed clipboard starts in malign apps to evolve much later than in benign ones, possibly with the introduction of piggybacked social media apps. Unlike for benign apps, malign applications behave different to the rest not only for the exposed clipboard issue, also improper certificate validation and dynamic code loading became popular at a later date with respect to the app's compilation timestamp as illustrated in Figure 4.16. In Figure 4.17, which illustrates the compilation dates of analysed apps, we observe an interesting anomaly around February-2008, seven months before release of Android 1.0. It appears that malign apps are being developed as soon as a platform gets announced. According to our results, legit app development entities, in contrast, await the rollout first and start to act on its success. It is intriguing to see in Figure 4.18 that for the malign corpus issues correlate to the code size. Nevertheless, we should consider that this could be a side-effect of our skewed malign corpus which contains more old and not that many recent (possibly piggybacked) apps with more complexity and thus more potential to suffer from vulnerabilities. A similar behavior can be seen in Figure 4.19 for Exposed Clipboard, that could be either induced by more recent or piggybacked apps.



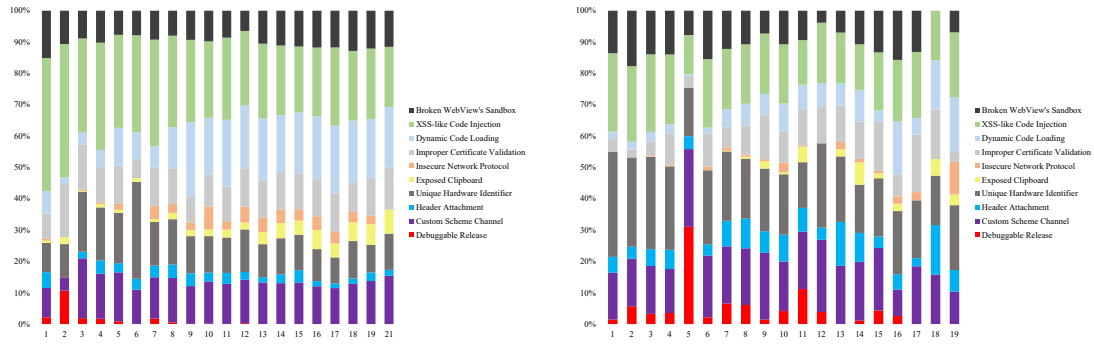
(a) (b)

Figure 4.13: A set of two subfigures: (a) The benign app distribution according to API levels; (b) The malign app distribution according to API levels



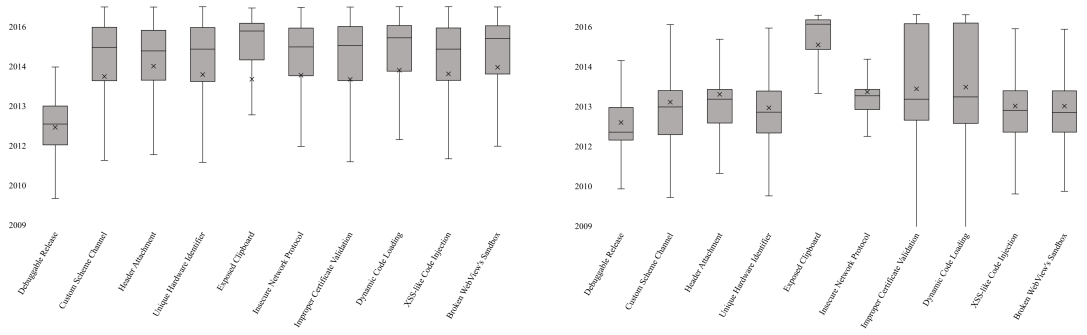
(a) (b)

Figure 4.14: A set of two subfigures: (a) Prevalence of security smell classes in benign apps; (b) Prevalence of security smell classes in malign apps



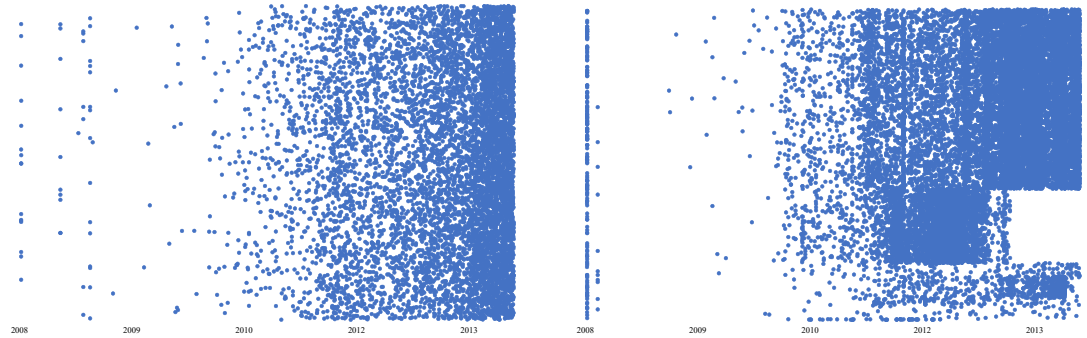
(a) (b)

Figure 4.15: A set of two subfigures: (a) Security smell class distribution among benign apps and API levels; (b) Security smell class distribution among malign apps and API levels



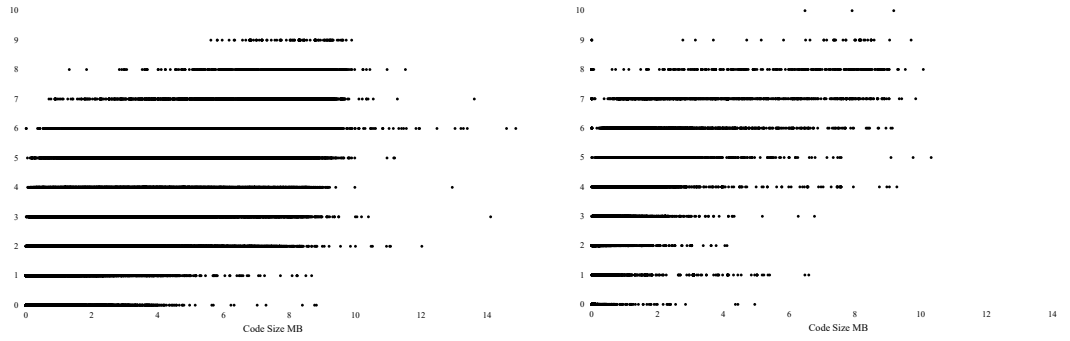
(a) (b)

Figure 4.16: A set of two subfigures: (a) Prevalence of security smell classes in benign apps with respect to compilation date; (b) Prevalence of security smell classes in malign apps with respect to compilation date



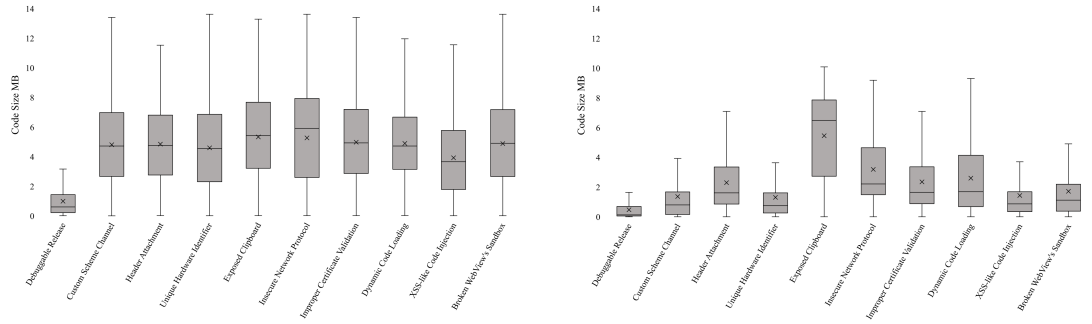
(a) (b)

Figure 4.17: A set of two subfigures: (a) Compilation date distribution of benign apps; (b) Compilation date distribution of malign apps



(a) (b)

Figure 4.18: A set of two subfigures: (a) Relation of code size and smell classes of benign apps; (b) Relation of code size and smell classes of malign apps



(a) (b)

Figure 4.19: A set of two subfigures: (a) Relation of smell classes and code size of benign apps; (b) Relation of smell classes and code size of malign apps

4.5.7 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 analysed 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.

5 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.

To fundamentally reduce the attack surface in Android, we promote the adoption of secure programming practices. We reviewed state of the art papers in security and identified 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 suffers 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.

The detection of some security code smell symptoms could easily be implemented in Android integrated development environment (IDE) plug-ins helping developers to improve code security already during development, as IDEs already construct internal dependency graphs containing all the necessary information. The security code smell reporting component could also leverage online web services and provide instant online analysis of apps.

A more thorough study of our collected but not yet detected smells, together with more specific symptoms, consequences and mitigation techniques for the already evaluated smells are left for future work.

To summarise, this work represents an initial effort to spread awareness about the impact of programming choices in making secure apps. We identified 28 security code smells in 5 different categories, namely

Insufficient Attack Protection, *Security Invalidation*, *Broken Access Control*, *Sensitive Data Exposure* and *Lax Input Validation*. We collected for each security code smell its symptoms and the potential impact, together with basic remediation measures. We argue that this helps developers who develop security mechanisms or other sensitive code to identify frequent problems, and also provides developers inexperienced in security with caveats about the prospect of security issues in their code.

Bibliography

[1] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1248–1259. ACM, 2015.

[2] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting inconsistent security configurations in custom Android ROMs via differential analysis. In *USENIX Security Symposium*, pages 1153–1168, 2016.

[3] 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.

[4] 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.

[5] 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!)*, pages 1–13. ACM, 2015.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[7] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[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] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard-enforcing user requirements on Android apps. In *TACAS*, pages 543–548. Springer, 2013.

[10] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. Detection of design flaws in the Android permission protocol through bounded verification. In *International Symposium on Formal Methods*, pages 73–89. Springer, 2015.

[11] R. Balebako and L. Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security Privacy*, 12(4):55–58, July 2014.

[12] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

[13] Michele Bugliesi, Stefano Calzavara, and Alvise Spanò. Lintent: Towards security type-checking of Android applications. In *Formal Techniques for Distributed Systems*, pages 289–304. Springer, 2013.

[14] Damjan Buhov, Markus Huber, Georg Merzdovnik, Edgar Weippl, and Vesna Dimitrova. Network security challenges in Android applications. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 327–332. IEEE, 2015.

[15] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. Towards analyzing the input validation vulnerabilities associated with Android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 361–370. ACM, 2015.

[16] 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 (USENIX Security 15)*, pages 659–674, Washington, D.C., 2015. USENIX Association.

[17] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[18] 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.

[19] 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.

[20] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for Android. In *ISC*, volume 10, pages 331–345. Springer, 2010.

[21] Brett Cooley, Haining Wang, and Angelos Stavrou. Activity spoofing and its defense in Android smartphones. In *International Conference on Applied Cryptography and Network Security*, pages 494–512. Springer, 2014.

[22] 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.

[23] 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.

[24] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, volume 31, 2011.

[25] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[26] 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, New York, NY, USA, 2012. ACM.

[27] 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, pages 201–210, New York, NY, USA, 2015. ACM.

[28] Zhejun Fang, Qixu Liu, Yuqing Zhang, Kai Wang, and Zhiqiang Wang. Ivdroid: Static detection for input validation vulnerability in Android inter-component communication. In *ISPEC*, pages 378–392, 2015.

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

[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, 2011.

[31] M. Fowler and K. Beck. *Refactoring: Improving the design of existing code*. Addison-Wesley object technology series. Addison-Wesley, 1999.

[32] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing Android's permission system. In *ESORICS*, 2012.

[33] Roberto Gallo, Patricia Hongo, Ricardo Dahab, Luiz C Navarro, Henrique Kawakami, Kaio Galvão, Glauber Junqueira, and Luander Ribeiro. Security and system architecture: Comparison of Android customizations. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 12. ACM, 2015.

[34] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. *GI-Jahrestagung*, 208:441–455, 2012.

[35] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*, volume 14, page 19, 2012.

[36] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of Android code smells. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, pages 59–69. ACM, 2016.

[37] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting antipatterns in Android apps. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149. IEEE Press, 2015.

[38] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. Prec: Practical root exploit containment for Android devices. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 187–198. ACM, 2014.

[39] Sungjae Hwang, Sungho Lee, Yongdae Kim, and Sukyoung Ryu. Bittersweet adb: Attacks and defenses. In *ASIACCS*, 2015.

[40] MWR InfoSecurity. Debuggable apps in Android market, jul 2011. https://labs.mwrinfosecurity.com/blog/debuggable-apps-in-android-market/.

[41] 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.

[42] 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.

[43] L. Li, D. Li, T. F. Bissyand, 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.

[44] 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.

[45] Li Li, Tegawend F. Bissyand, 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.

[46] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 978–989. ACM, 2014.

[47] Chia-Chi Lin, Hongyang Li, Xiao yong Zhou, and XiaoFeng Wang. Screenmilker: How to milk your Android screen for secrets. In *NDSS*, 2014.

[48] 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.

[49] 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.

[50] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[51] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. *NDSS Symposium 2017, San Diego, California*, 2017.

[52] Michael Mitchell, Guanyu Tian, and Zhi Wang. Systematic audit of third-party Android phones. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 175–186. ACM, 2014.

[53] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for Android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013.

[54] Patrick Mutchler, Adam Doup, John Mitchell, Christopher Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*, 2015.

[55] 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.

[56] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1029–1042. ACM, 2013.

[57] 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.

[58] 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.

[59] 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, New York, NY, USA, 2015. ACM.

[60] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. The aDoctor project. *SANER 2017*, 11 2016.

[61] André Pereira, Manuel Correia, and Pedro Brandão. USB connection vulnerabilities on Android smartphones: Default and vendors' customizations. In *IFIP International Conference on Communications and Multimedia Security*, pages 19–32. Springer, 2014.

[62] 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, February 2014.

[63] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *NDSS*, 2014.

[64] 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, October 2016.

[65] 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.

[66] Julian Schütte, Rafael Fedler, and Dennis Titze. Condroid: Targeted dynamic analysis of Android applications. In *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*, pages 571–578. IEEE, 2015.

[67] 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.

[68] Di Shen. Exploiting trustzone on Android. *Black Hat US*, 2015.

[69] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. Modelling analysis and auto-detection of cryptographic misuse in Android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, pages 75–80. IEEE, 2014.

[70] 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.

[71] 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.

[72] Daniel R Thomas, Alastair R Beresford, and Andrew Rice. Security metrics for the Android ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 87–98. ACM, 2015.

[73] 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, Washington, DC, USA, 2015. IEEE Computer Society.

[74] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing Android permission creep. In *Proceedings of the Web*, volume 2, pages 91–96, 2011.

[75] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability assessment of OAuth implementations in Android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 61–70. ACM, 2015.

[76] 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.

[77] 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.

[78] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in Android. In *CODASPY*, 2014.

[79] 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, Piscataway, NJ, USA, 2017. IEEE Press.

[80] Ralf-Philipp Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *Presented as part of the 6th USENIX Workshop on Offensive Technologies*, Bellevue, WA, 2012. USENIX.

[81] 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.

[82] Jianliang Wu, Tingting Cui, Tao Ban, Shanqing Guo, and Lizhen Cui. Paddyfrog: Systematically detecting confused deputy vulnerability in Android applications. *Security and Communication Networks*, 8(13):2338–2349, 2015.

[83] 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.

[84] 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.

[85] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 393–408. IEEE, 2014.

[86] Liang Xu, Shyhtsun Felix Wu, and Hao Chen. Techniques and tools for analyzing and understanding Android applications. In *Dissertation*, 2013.

[87] 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.

[88] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *NDSS*, 2014.

[89] 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.

[90] Xiao Zhang and Wenliang Du. Attacks on Android clipboard. In *DIMVA*, 2014.

[91] 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.

[92] Yuan Zhang, Min Yang, Guofei Gu, and Hao Chen. Finedroid: Enforcing permissions with system-wide application execution context. In *International Conference on Security and Privacy in Communication Systems*, pages 3–22. Springer, 2015.

[93] Min Zheng, Mingshen Sun, and John C. S. Lui. Droidray: A security evaluation system for customized Android firmwares. In *ASIACCS*, 2014.

[94] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Droidalarm: An all-sided static analysis tool for Android privilege-escalation malware. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 353–358. ACM, 2013.

[95] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in Android device driver customizations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 409–423. IEEE, 2014.

[96] Chaoshun Zuo, Jianliang Wu, and Shanqing Guo. Automatically detecting SSL error-handling vulnerabilities in hybrid mobile web apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 591–596. ACM, 2015.