SCAM 2017

Security Smells in Android

Mohammad Ghafari, Pascal Gadient, Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{ghafari, gadient, oscar}@inf.unibe.ch

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.

I. Introduction

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

The recent survey on the Stackoverflow website shows that about 65% of mobile developers work with Android.[1] This platform has captured over 80% of the smartphone market,[2] and just its official app store contains more than 2.8 million apps. As a result, a security mistake in an in-house app may jeopardize the security and privacy[3] of billions of users.

The security of smartphones has been studied from various perspectives such as the device manufacturer [41], its platform [44], and end users [16]. Manifold security APIs, protocols, guidelines, and tools are proposed. Nevertheless, security concerns, in effect, are outweighed by other concerns [6]. Many developers undermine their significant role in providing security [42]. As a result, apps still suffer from serious proliferating security issues.[4] For instance, the analysis of 100 popular apps downloaded at least 10M times, revealed that over 90% of them, due to development mistakes, are prone to SSL vulnerabilities that allow criminals to access credit card numbers, chat messages, contact list, files, and credentials [27].

Given these premises, 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 security 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 three research questions:

- **RQ**$_1$: What are the security code smells in Android apps? We have reviewed major related work, especially those appearing in top-tier conferences/journals, and identified 28 avoidable vulnerabilities and the smells that indicate their presence. We thoroughly discuss each smell, the risk associated with it, and its mitigation during app development.
- **RQ**$_2$: How prevalent are security smells in benign apps? We have developed a lightweight tool that statically analyzes apps for the existence of ten security smells. We applied the tool to a repository of about 46 000 apps hosted by Google. We realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security smells.
- **RQ**$_3$: To which extent identifying security smells facilitates detecting vulnerabilities? We manually inspected 160 apps, and compared our findings to the result of the tool. Our investigation showed that the identified smells are in fact a good indicator of security vulnerabilities.

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

The remainder of this paper is structured as follows. We present the identified vulnerabilities and corresponding security smells in section II. We study the prevalence of these smells and discuss the results in section III. We summarize the work closely related to this paper in section IV, and we conclude the paper in section V.

[1] http://insights.stackoverflow.com/survey/2017
[2] http://www.gartner.com
[3] In short, referred to as security in this paper
[4] http://www.cvedetails.com

II. Security Smells

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

We identified 28 smells that may lead to vulnerabilities in Android-powered devices.[5] 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.

A. Insufficient Attack Protection

- **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. *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 [37]. *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 [11]. *Mitigation:* Solely use reliable libraries [5].
- **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 analyze; there is no distinction between code and data at the native level, and attackers can load and execute code from native executables, in a variety of ways much easier than in Java. *Consequently*, native code is susceptible to severe vulnerabilities like buffer overflow, and an attacker could exploit such vulnerabilities, for instance, to execute malicious code [36]. *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 [5] 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 [19]. *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 [7]. *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 [32]. *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 [33]. *Symptom:* The manifest file contains permissions for APIs that are not used. *Mitigation:* Utilize tools like PScout[6] to exclude from the manifest file any permission whose corresponding API calls are absent in the app.

[5] We define vulnerability as a security issue that compromises user's security and privacy.

[6] http://pscout.csl.toronto.edu

B. Security Invalidation

- **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. *Mitigation:* Consult the state of the art guidelines to choose an appropriate cryptography, and utilize expert systems [4].
- **Weak Crypto Configuration**
  The majority of security breaches come from exploiting developer's mistakes. *Issue:* Cryptography APIs are widely perceived as being complex with many confusing options [23]. *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 [2].
- **Unpinned Certificate**
  Digital certificates are needed to ensure secure communication. Unpinned certificates are easy to maintain and are frequently used in the appified world [26]. *Issue:* Ensuring the authenticity of a certificate is non-trivial, if it is not pinned. *Consequently*, an app may inadvertently end up trusting a certificate issued by an adversary who has intercepted network communication. *Symptom:* The app uses unpinned certificates. *Mitigation:* Pinning certificates are always recommended to increase the security.[7]
- **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 [12]. *Consequently*, this leaves the communication channel over SSL/TLS insecure and susceptible to man-in-the-middle attacks [9]. *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"[8] to examine your communication.
- **Unacknowledged Distribution**
  Google Play, Google's official marketplace for Android, strives to identify potential security enhancements when an app is uploaded to it. However, developers may distribute their packages via other channels to circumvent out-of-order updates, bypassing the slow release cycles and security restrictions of this market place. *Issue:* The protection provided by Google, including code and signature checks, is neglected. *Consequently*, the risk of distributing a vulnerable app increases especially when the app utilises uncertified libraries, or in a worse case, an attacker can replace installation packages with malicious ones [48]. *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.

C. Broken Access Control

- **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 [8]. *Symptom:* The existence of an intent with private data, 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

[7] Since Android 6.0 pinning can be enabled using the `Network Security Configuration` feature.

[8] https://github.com/google/nogotofail

operations [40], [10]. *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 [17].

- **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 [31]. *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 [20]. *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 [14]. *Symptom:* The existence of adb-specific commands or TCP connection to local host in the code. *Mitigation:* Avoid using adb-level capabilities in your app, as it is also prohibited since Android 6.0.
- **Debuggable Release**
  During app development there exist two major build configurations, 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 [43]. *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 [1]. *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 [35]. *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.

D. Sensitive Data Exposure

- **Header Attachment**
  The header section of data transport protocols like HTTP comprises key/value pairs to store operational parameters. *Issue:* Developers may rely on headers to transfer sensitive data, *e.g.*, they store credentials to auto-login into a service. *Consequently*, any adversary eavesdropping on the network may easily access the attached data [35]. *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[9] 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 [46]. *Symptom:* The related calls on `ClipboardManager` exist in the code. The app uses the common `TextView`

[9] https://oauth.net/2

and `EditText` controls, which allow copy and paste to handle sensitive data [24]. *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.[10] *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.
- **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 [27]. *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 [22]. *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 [47]. *Consequently*, such so-called "data residue" can be associated to another app and empower adversaries to access sensitive information [45], [47]. *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 mere mitigation is to avoid sharing private data with these services, if possible.

E. Lax Input Validation

- **XSS-like Code Injection**
  WebView is an essential component that enables developers to use web technologies such as HTML and JavaScript to deliver web content within an app. Unlike Web browsers 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 [15]. *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 [15]. *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 [28]. 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 [13]. *Symptom:* Use of

[10] https://developer.android.com/guide/topics/data/data-storage.html

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 [34]. Analyze your app with the help of tools like *Grab 'n Run* [13].

- **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.

III. Empirical study

We developed a lightweight analysis tool that statically detects known security smells in an app. We rely on the Apktool to reverse engineer Android apk files and generate smali code.[11] We defined a set of rules to capture the symptoms of each security smell. In particular, we utilize Java XML Parser for parsing the Manifest files and use regular expressions to define and match the code pattern corresponding to the identified symptoms of each smell in the code.

We randomly selected our apps from the AndroZoo dataset.[12] 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 visit 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 benign apps in our dataset. About 90% of these apps were released between 2014 and 2016, a quarter of them were updated within three months, the majority were rated more than four stars, slightly more than 27% were downloaded above 50 000 times, and the median apk size was 5.5MB.

A. Result

We applied our lightweight tool to all apps in the dataset. Figure 1 shows how prevalent the smells are in our dataset. A majority of apps potentially suffer from XSS-like code injection (85%) followed by dynamic code loading (61%).

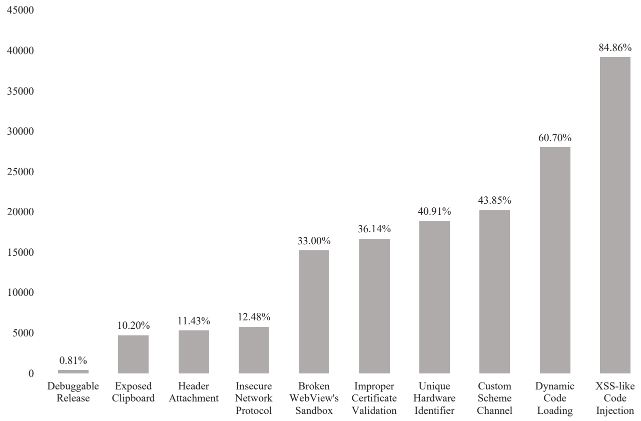

Fig. 1. Distribution of security smells in the apps

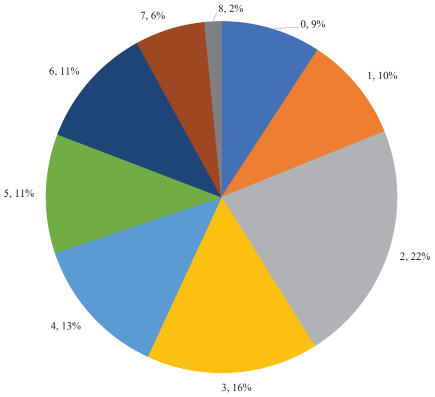

Fig. 2. Partitioning apps by number of security smells

About 40% use custom scheme channels and expose a unique hardware identifier. More than 12% use an insecure network protocol, and almost 10% are subject to header attachment as well as clipboard issues. Finally, just under 1% of the apps have debug mode enabled.

We also studied how many of security smells usually appear in the apps (see Figure 2). 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 3 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 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*

[11] https://ibotpeaches.github.io/Apktool
[12] https://androzoo.uni.lu

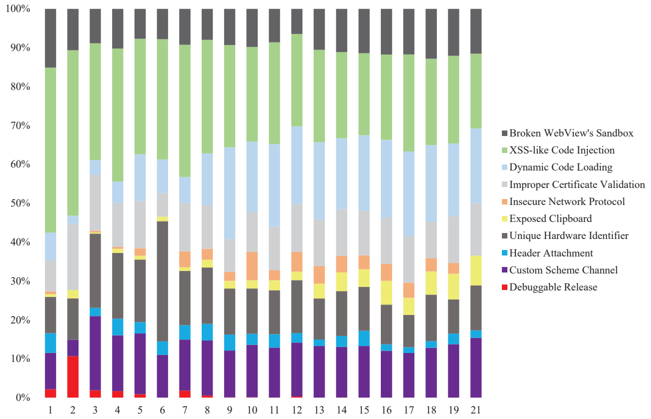

Fig. 3. The distribution of security smells within each API level

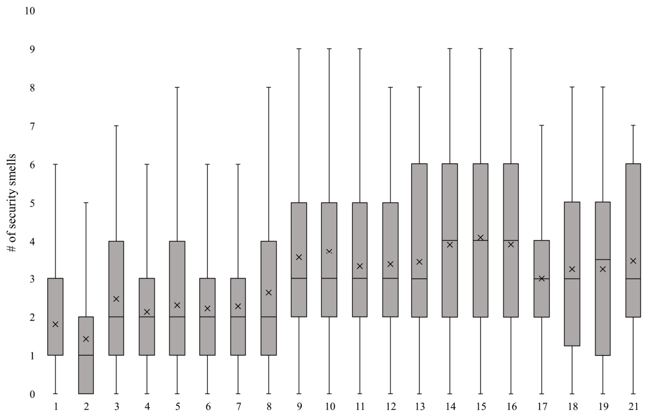

Fig. 4. Number of smells within each API level

Loading has become more common. We observed that many developers adopt this feature to implement their own update mechanisms.

Figure 4 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.

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

Category: Figure 5 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 smells.

Popularity: Figure 6 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. Figure 7 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 barely noticeable by end-users, hence they are not reflected in the ratings.

Release date: We further studied whether the prevalence of security smells changes over time. In fact, with advances in developers support (*e.g.*, tools, learning resources) we expected that security smells in more recent apps should be rarer than in older apps. Nonetheless, the result showed that neither the number of smells nor the likelihood of a particular smell relates to the release date of an app. Moreover, we noted that in general the security of apps with short update cycles is similar to those with longer update cycles. That is, either security issues in one release still remain in future releases, or they get fixed but new releases also introduce new smells.

App Size: We were interested to know whether 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 other resources such as image, video and audio content.

B. *Manual Analysis*

To assess how reliable these findings are to detect security vulnerabilities, we manually analyzed 160 apps. For each smell, we inspected 20 apps manually and compared our findings to the result of the lightweight analysis tool. As is shown in Figure 8, the results were encouraging. The manual analysis completely agreed with the tool in the security risk associated with six security smells. In case of exposed clipboard the tool achieved a very good performance *i.e.*, above 90% agreement with the manual analysis. The level of agreement in insecure network protocol and improper certificate validation was 80%. We realized some apps use http connections to exclusively load local contents which is legitimate in development frameworks like Apache Cordova or Adobe PhoneGap. And 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.

C. *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.

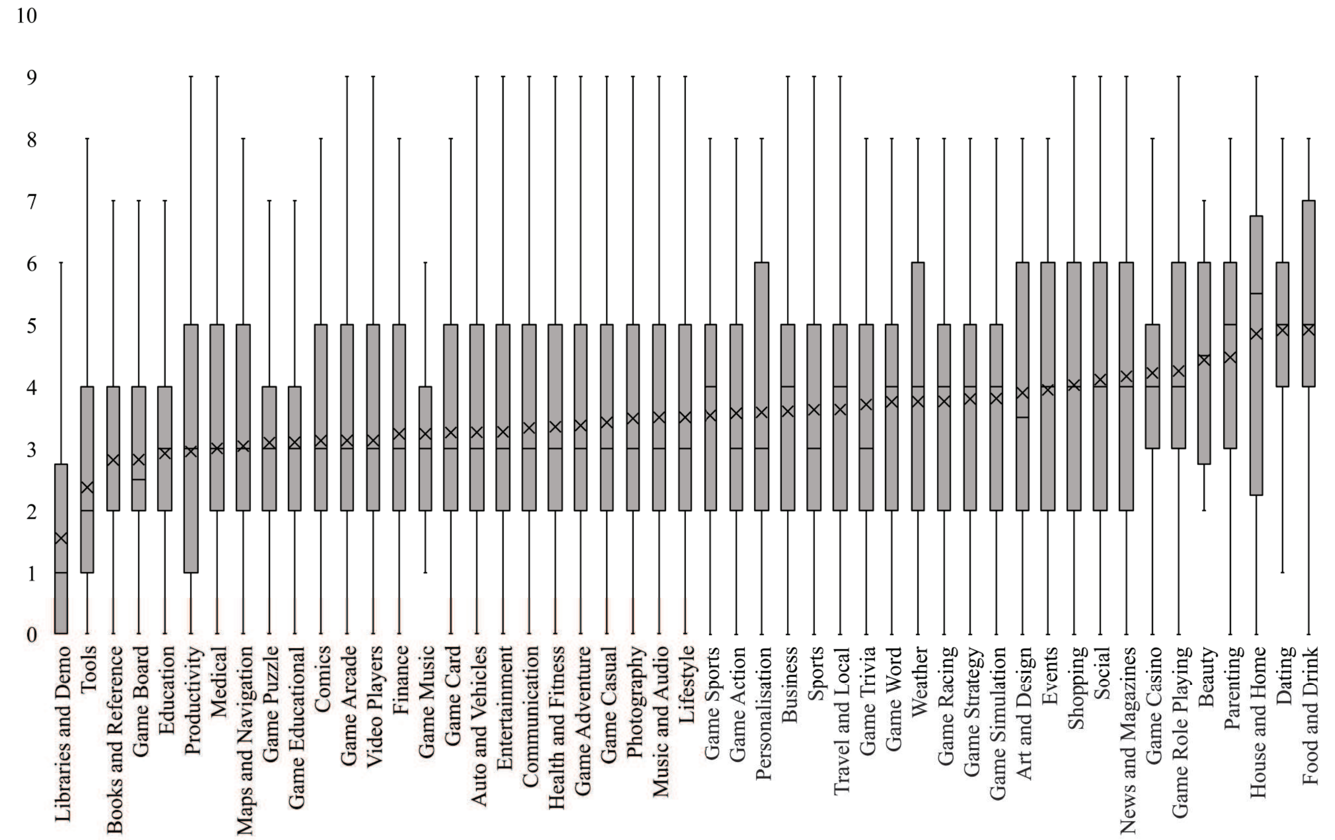

Fig. 5. Distribution of smells in app categories

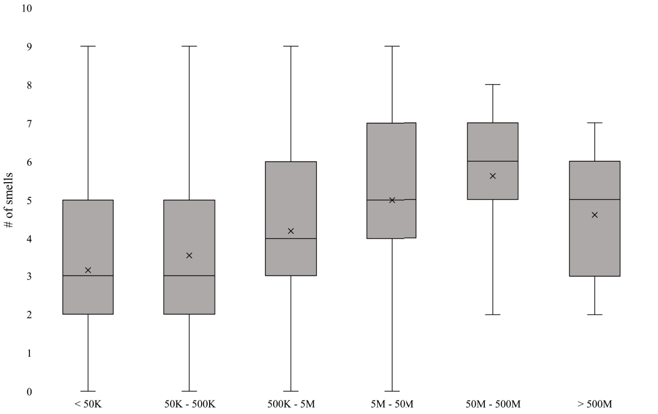

Fig. 6. The relationship between number of smells and number of downloads

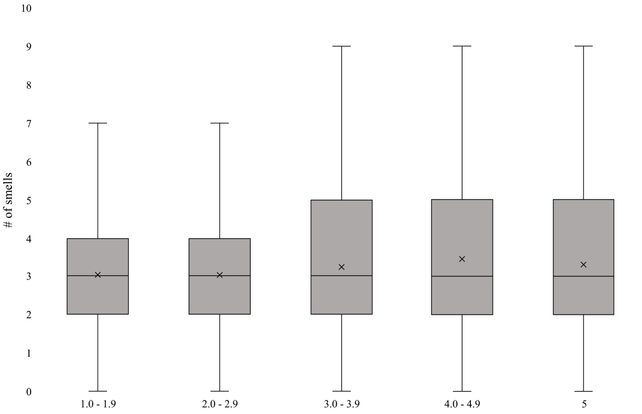

Fig. 7. The relationship between number of smells and app star ratings

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 GoogleScholar.

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

We were only interested in studying benign apps as in malicious ones developers may not spend any effort to accommodate security. Thus, we merely collected apps which were available on 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.

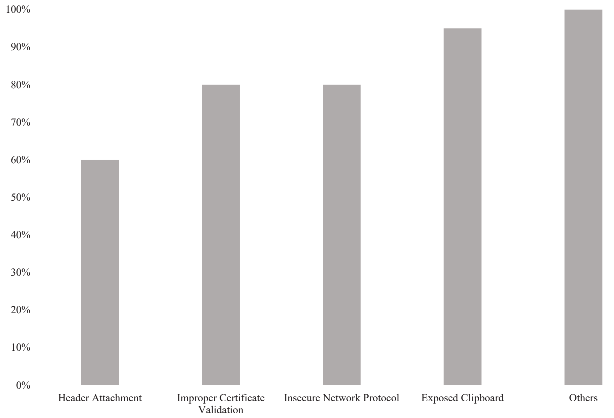

Fig. 8. The precision of obtained results

IV. Related Work

Reaves *et al.* discussed Android specific challenges to program analysis and assessed android application analysis tools, and found that they mainly suffer from lack of maintenance, and are often unable to produce functional output for applications with known vulnerabilities [29]. Li *et al.* studied the state-of-the-art work that statically analyses Android apps [18]. They found that much of this work supports detection of private data leaks and vulnerabilities, a moderate amount of research is dedicated to permission checking, and only three studies deal with cryptography issues. Unfortunately, much state-of-the-art work does not publicly share their artefacts. Linares-Vasquez *et al.* mine 660 Android vulnerabilities available in the official Android bulletins and the CVE-details and present a taxonomy of the their types; they report the presence of those vulnerabilities affecting the Android OS and acknowledge that most of them can be avoided by relying on secure coding practices [21]. Finally, Sadeghi *et al.* review 300 research papers related to Android security, and provide a taxonomy to classify and characterize the state-of-the-art research in this area [30]. They find that 26% of existing research is dedicated to vulnerability detection, but each study is usually concerned with specific types of security vulnerabilities. Our work expands on such studies to provide practitioners with an overview of the security issues that are inherent in insecure programming choices.

Some research is devoted to educating developers for secure programming. 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 [42]. Weir *et al.* performed open-ended interviews with a dozen app security experts, and identified that app developers should learn analysis, communication, dialectic, feedback and upgrading in the context of security [38]. Witschey *et al.* surveyed developers about their reasons for adopting and not adopting security tools [39]. Interestingly, they found the perceived prestige of security tool users and the frequency of interaction with security experts more important to promote security tool adoption. Acar *et al.* suggest a high-level research agenda to achieve usable security for developers. They propose several research questions to elicit developers' attitudes, needs, and priorities in the area of security [25]. Our work is complementary to these studies in the sense that provides an initial assessment of developers' security knowledge, and as well highlights the significant role of developers in making more secure apps.

V. 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 28 smells whose presence may indicate a security issue in an app. We developed a static analysis tool to study the prevalence of ten of such smells in 46 000 apps. We realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security smells. Moreover, the manual inspection of 160 apps showed that the identified security smells are actually a good indicator of security vulnerabilities.

VI. Acknowledgments

We gratefully acknowledge the funding of the Swiss National Science Foundations for the project "Agile Software Analysis" (SNF project No. 200020_162352, Jan 1, 2016 - Dec. 30, 2018). [13]

References

[1] https://labs.mwrinfosecurity.com/blog/debuggable-apps-in-android-market.

[2] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. Mazurek, and C. Stransky. Comparing the usability of cryptographic apis. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, 2017.

[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] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini. Towards secure integration of cryptographic software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015. ACM, 2015.

[5] M. Backes, S. Bugiel, and E. 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.

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

[7] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th USENIX Security Symposium*, pages 659–674, Washington, D.C., 2015. USENIX Association.

[13] http://p3.snf.ch/Project-162352

[8] E. Chin, A. P. Felt, K. Greenwood, and D. 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.

[9] 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.

[10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, 2011.

[11] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. 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.

[12] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61. ACM, 2012.

[13] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi. Grab 'n run: Secure and practical dynamic code loading for android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015. ACM, 2015.

[14] S. Hwang, S. Lee, Y. Kim, and S. Ryu. Bittersweet adb: Attacks and defenses. In *ASIACCS*, 2015.

[15] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. 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.

[16] B. H. Jones and A. G. 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.

[17] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. 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.

[18] L. Li, T. F. Bissyand, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67 – 95, 2017.

[19] L. Li, D. Li, T. F. Bissyand, J. Klein, Y. L. 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.

[20] C.-C. Lin, H. Li, X. yong Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *NDSS*, 2014.

[21] M. Linares-Vásquez, G. Bavota, and C. 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.

[22] 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.

[23] S. Nadi, S. Krüger, M. Mezini, and E. 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.

[24] 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.

[25] Y. A. nd Sascha Fahl nd Michelle Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *IEEE SecDev 2016*, 2016.

[26] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To pin or not to pin-helping app developers bullet proof their tls connections. In *USENIX Security Symposium*, 2015.

[27] L. Onwuzurike and E. De Cristofaro. Danger is my middle name: Experimenting with ssl vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 15:1–15:6. ACM, 2015.

[28] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[29] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. Butler, W. Enck, and P. Traynor. *droid: Assessment and evaluation of android application analysis tools. *ACM Comput. Surv.*, 49(3):55:1–55:30, 2016.

[30] A. Sadeghi, H. Bagheri, J. Garcia, and s. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[31] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. 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.

[32] V. F. Taylor and I. 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.

[33] V. F. Taylor and I. 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.

[34] D. Titze and J. Schütte. Preventing library spoofing on android. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 01*, TRUSTCOM '15, pages 1136–1141. IEEE Computer Society, 2015.

[35] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: threats and mitigation. In *ACM Conference on Computer and Communications Security*, 2013.

[36] T. Wang, K. Lu, L. Lu, S. Chung, and W. 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.

[37] T. Watanabe, M. Akiyama, F. Kanei, E. Shioji, Y. Takata, B. Sun, Y. Ishi, T. Shibahara, T. Yagi, and T. Mori. Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 14–24, 2017.

[38] C. Weir, A. Rashid, and J. Noble. Reaching the masses: A new subdiscipline of app programmer education. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 936–939. ACM, 2016.

[39] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 260–271. ACM, 2015.

[40] D. Wu, D. Gao, Y. Li, and R. H. Deng. Seccomp: Towards practically defending against component hijacking in android applications. *CoRR*, abs/1609.03322, 2016.

[41] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[42] 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.

[43] L. XU, S. F. Wu, and H. Chen. Techniques and tools for analyzing and understanding android applications. In *Dissertation*, 2013.

[44] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, et al. Toward engineering a secure android ecosystem: a survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.

[45] X. Zhang, Y. Aafer, K. Ying, and W. Du. *Hey, You, Get Off of My Image: Detecting Data Residue in Android Images*, pages 401–421. Springer International Publishing, Cham, 2016.

[46] X. Zhang and W. Du. Attacks on android clipboard. In *DIMVA*, 2014.

[47] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS*, 2016.

[48] M. Zheng, M. Sun, and J. C. S. Lui. Droidray: a security evaluation system for customized android firmwares. In *ASIACCS*, 2014.