Identifying Android Inter App Communication Vulnerabilities Using Static and Dynamic Analysis

Biniam Fisseha Demissie, Davide Ghio, Mariano Ceccato, Andrea Avancini
Fondazione Bruno Kessler
Via Sommarive, 18
Trento, Italy
{demissie,ceccato,anavancini}@fbk.eu

ABSTRACT

The Android platform is designed to facilitate inter-app integration and communication, so that apps can reuse functionalities implemented by other apps by resorting to delegation. Though this feature is usually mentioned to be the main reason for the popularity of Android, it also poses security risks to the end user. Malicious unprivileged apps can exploit the delegation model to access privileged tasks that are exposed by vulnerable apps.

In this paper, we present a particularly dangerous case of delegation, that we call the Android Wicked Delegation (AWiDe). Moreover, we compare two distinct approaches to automatically detect inadequate message validation, respectively based on static analysis and on dynamic analysis. We empirically validate our approaches on more than three hundred popular apps. Vulnerabilities detected by us lead to the implementation of successful proof-of-concept attacks, and the app developers have confirmed one of them.

1. INTRODUCTION

Android is a popular [13] operating system for smart phones that offers a programming interface (API) to host and run applications written in Java, commonly called apps. Android has a centralized app market that offers more than one million apps to download and install. It is common to find several apps in the market having similar purposes. Therefore, when a new idea becomes apparent, an app developer needs to rush before similar apps become available on the market by competitors. Since the first apps that appear on the market usually get accepted by the users and are rated better, posting an app early can help the developer gain market share.

Due to the need to develop an app as fast as possible, developers usually spend more time on the app's user experience and appealing user interface, but they overlook the need for quality and security of the app [11]. Other quality aspects (possibly including security issues) are delayed for future updates (if there are any). Therefore, an effective, fast and automated tool to spot defects in apps' code would be highly beneficial for app developers.

MobileSoft'16, May 16-17, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4178-3/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2897073.2897082

Though the Android operating system enforces barriers to isolate apps that are provided by different vendors, poorly developed apps might still pose security risks to the end user. Legitimate apps might unintentionally expose their access to sensitive resources to attackers, such as phone contacts, and privileged actions, such as making phone calls. This problem is known as *permission re-delegation* [12]. However, delegation can be legitimate, when performed according to the Android design, or malicious, when intended to exploit permissions in a way that was different from the developer's intentions.

The novel contribution of this work consists in extending the permission re-delegated threat model to clearly distinguish the malicious cases. We call this extension the Android Wicked Delegation (AWiDe for short). The vulnerability consists in executing privileged tasks on behalf of another app (similar to the re-delegation case) but with the additional preconditions that (i) the privileged task is executed with data coming from attacking apps, controlled by the attacker, and (ii) without performing adequate validation of such data.

While static and dynamic taint analysis have been already used in the past to spot inadequate input validation (e.g., in web applications), our novelty consists in formulating malicious delegation as a problem of inadequate input validation in the context of inter-app communication in Android.

We compare two automated detection techniques for AWiDe vulnerabilities. In the first approach, we instantiate static taint analysis to detect whether values used in privileged actions depend on potentially malicious data. The second approach relies on dynamic analysis with automatically generated input values. Privileged actions are detected by comparing execution traces of the same app with and without the corresponding special permissions. Dynamic taint analysis tracks data dependencies during execution and detects when data from the attacking apps is used in privileged actions.

These approaches have been empirically validated on more than three hundred apps. Results show that popular apps are affected by AWiDe vulnerabilities. We also demonstrate the relevance of this problem by elaborating actual attacks based on the vulnerabilities detected by our approaches. The corresponding app developers have confirmed one of these vulnerabilities.

The paper is structured as follows. Section 2 compares this work with the state of the art and Section 3 presents the threat model. Section 4 and Section 5 describe the approaches for vulnerability detection, respectively based on static and dynamic analysis. Then, experimental evaluation is presented in Section 6 and discussed in Section 7. Eventually, Section 8 closes the paper.

2. RELATED WORK

The most related work deals with vulnerabilities connected to different threat models, i.e. permission re-delegation or information leak. Other related work is about the automatic generation of security test cases.

Permission re-delegation. The most related work is by Felt et al. [12]. They presented the permission re-delegation problem, i.e. an app with special permissions that exposes a service that does not require the same permissions to be consumed. As a consequence, the exposed service could be used by a second app without permissions to ask the former to act on its behalf and take privileged actions. A vulnerability is detected whenever there exists a path from a public entry point to a privileged API call. Our work presents an extension of this threat model, by checking additional preconditions. Our threat model, in fact, requires that inadequately validated (potentially malicious) data is used in the restricted API call. In our opinion, this extension is fundamental to distinguish malicious attacks from legitimate delegation.

Permission re-delegation is also detected by ComDroid, a tool developed by Chin et al. [8], which additionally detects cases of *unauthorized intent receipt*. The latter cases are malicious apps that define intent filters similar to those of the apps under attack, to intercept the intents sent to them. However, both Felt et al. and Chin et al. acknowledge that their approaches cannot distinguish between legitimate and malicious delegation.

Zhang et al. [26] proposed a runtime patch to mitigate re-delegation problem. They perform static data flow analysis to determine sensitive data flows from sources to sinks and apply the patch before the invocation of the privileged API such that the app informs the user of a potential re-delegation attack and requests if the user wants to continue. The authors used the same taint policy as TaintDroid (see below) but they apply the taint propagation directly on the Dalvik bytecode instead of modifying the Dalvik virtual Machine.

Information leak. Information flow in mobile apps is analysed either statically [22, 25, 18, 17] or dynamically [10], to detect disclosure of sensible information. Tainted sources are system calls that access private data (e.g., global position, contacts entries), while sinks are all the possible ways that make data leave the system (e.g., network transmissions). An issue is detected when privileged information could potentially leave the app through one of the sinks.

Amandroid [25] has been proposed to detect privacy leaks related to inter-component communication in Android apps. IccTA [18] attempts to improve static taint analysis by modelling the life-cycle and callback methods by instrumenting the code of the app. However, it is also limited to privacy leak. DidFail [17] attempts to detect data leaks between activities through implicit intents. It misses other components and explicit intents. A similar work has been done by Lu et al. [19], where they try to identify all possible entry points of an Android app and then performing data flow analysis starting from the entry point until a sensitive API is reached. Grace et al. [14], perform static analysis in stock Android apps released by different vendors, to check the presence of any information leak. Since vendors modify or introduce their own apps, they might also introduce new vulnerabilities. This work, however, is limited to stock apps on specific vendor devices.

TaintDroid [10] is a tool to perform dynamic taint analysis. It relies on a modified Android installation that tracks taintedness at run-time. The implementation showed minimal size and computational overhead, and was effective in analysing many real Android apps. A complementary approach is based on static analysis [22], where a type system is implemented to track security levels. A static analyser applies the type system to byte-code and it detects violations when privileged information could potentially leave the app through a sink.

All these works adopt a threat model different from AWiDe. Their common objective is to list candidate vulnerable cases of information leak, rather than malicious cases of permission re-delegation among apps.

Security test case generation. While works mentioned above report a list of vulnerable points, Maji et al. [21] analyse inter-app messaging with the objective of generating executable scenarios, where the communication is tested to spot app (or system) crashes. The JarJarBinks tool has been implemented to study when invalid intents (i.e. that violate intent filters) make apps crash. Test cases, however, reveal generic app crashes rather than focusing on security defects.

Automatic testing of peculiarities of Android apps has been addressed from the point of view of the graphical user interface [16, 2, 1], to detect events and event sequences that make the app crash. Hu et al. [16] presented an approach for testing Android apps' GUI. Random graphical events are generated and patterns are used to detect bugs in the system log, such as app crashes, type exceptions and violations in the activity state machine. Amalfitano et al. developed *AndroidRipper* [2] and *A2T2* [1] to test Android GUI. These tools dynamically analyse the apps to get a list of fireable events in the GUI widgets, they then generate sequences of graphical events and sensor events. Code is instrumented to record crashes and to eventually translate event sequences into JUnit test cases.

A work that directly generates security test cases for Android apps has been presented by Mahmood et al. [20]. Test cases are generated by using random graphical events and input values. Objective of the work, however, is just to spot crashes due to communication errors and violations of access permissions. Our approach, instead, generates test cases that expose issues due to inadequate input validation, which could threat the security of mobile devices.

This paper extends a previous workshop paper [5] that proposed the initial intuition on the identification of AWiDe vulnerabilities. However, such preliminary work was incomplete because taint analysis was not considered (neither static nor dynamic). The present paper proposes and validates an improved threat model, by requiring that intent data is used in privilege protected API calls. Moreover, static and dynamic taint analysis are presented and used on a large set of real Android apps.

3. THREAT MODEL

3.1 Background

Many apps are available on the official Android app store (called *Google Play*). However, apps are provided by various developers with different levels of trust. The Android framework has been designed with the two-fold objective of (i) allowing the integration and collaboration of apps from different vendors but still (ii) guaranteeing a certain level of separation to enforce security and confidentiality. Separation among apps is achieved by enforcing *sand-boxing* and *firewalling* and by adopting a permission system to regulate the access to sensitive resources.

Sand-box: Sand-boxing apps means isolate them from system resources. In order to access any resource (such as the network, the GPS position, the contact lists), apps have to explicitly request proper permissions that the final user evaluates and authorizes at installation time. The list of authorizations requested by an app is specified in an XML manifest file that is part of its packaging. Figure 1 shows a fragment of manifest for a running example app. In this example, the app is granted the special permission *INTERNET* to access the network.

Firewall: Firewalling apps means separating them from each other. Apps receive a distinct Unix user-id, so they run in their own private space and memory. Communication among apps is possible through the mediation of the operating system by the so-called Inter-Process Communication mechanism (IPC). The framework is designed such that apps can collaborate, integrate and complement each other. For instance an app able to take pictures can be delegated to make photos (and elaborate them) on behalf of other apps that, thus, do not need to reimplement this feature.

Integration: An app can delegate a specific task to another app, without actually knowing which apps are available in the current device to accomplish that task. Different final users might have different installed apps that are able to take pictures, but the requester app does not need to know which one to delegate. For the requester, it is enough that the delegated app can take pictures.

Android has been developed such that a requester app has just to specify what should be done (and with what data), and the framework will identify an app able to accomplish it. To delegate an action, apps use inter-process communication (IPC) messages, called *intents*. Intents are messages that contain the description (in a specific syntax) of the operation that the requester needs to perform. Apps specify in their XML manifest files what services they expose to other apps, with the so-called *intent filters*. The framework relies on the manifest to decide what app to delegate.

Figure 1 shows a fragment of the manifest file of an Android app that provides a service to expand URLs shortened by using *goo.gl*. URL shortening is a service to substantially reduce long and complex URLs to few characters but still pointing to the original locations. URL shortening is useful in social networks (e.g., Twitter), in messaging apps or to reduce typing when manually entering URLs. The official market contains apps that offer this feature (e.g., *URL Expandroid*[1] and *Short URL Evaluator*[2]).

This app defines an intent filter to accept short URL to expand to the original full URL. According to the filter definitions, intents for this app must specify the *VIEW* action and the *DEFAULT* category. The data part of the intent specifies the short URL to expand with scheme *https* and host *goo.gl*.

Intents can be either *implicit* or *explicit*. *Implicit* intents just specify the task to be performed. The system inspects the intent content to decide the most appropriate destination app(s). To decide the destination, the system compares the content of the intent with the *intent filters* (i.e., with the service definitions) that are specified in the manifest files of the currently installed apps.

A second app, for example, sends implicit intents to ask for URL expansion. An implicit intent with action *VIEW*, category *DEFAULT* and data *https://goo.gl/IhH0Ix* would match the intent filter in Figure 1 and therefore it would be delivered to the corresponding app. This app will contact the appropriate URL shortening service to retrieve the original URL that is returned as result.

In *explicit* intents the sender app specifies the receiver name as part of the intent. This assumes that the requester knows exactly what app to delegate. Different Android users, however, may have a wide diversity of installed apps, therefore a specific app may not be available. Implicit intents, instead, work on the wide heterogeneity of device configurations.

```
<activity android:name="ExpandUrl">
  <intent-filter>
       <action android:name="android.intent.action.VIEW" />
       <category android:name="android.intent.category.DEFAULT" />
       <data android:scheme="https" android:host="goo.gl" />
  </intent-filter>
</activity>

<uses-permission android:name="android.permission.INTERNET" />
```

Figure 1: Example of Android manifest file.

3.2 Motivating Example

Apps that are granted special privileges should not expose vulnerabilities, otherwise special privileges could be the target of attacks. Non-privileged apps could, in fact, exploit such vulnerabilities by crafting malicious intent messages intended to make a vulnerable app misuse its permissions to leak sensitive data (e.g., GPS position or contacts), write sensitive information (such as contacts or app private data) or perform costly operations (calls or SMS to premium numbers).

Figure 2 shows an example of attack scenario in which the vulnerability is due to inadequate validation of an intent message whose data is used in a privileged operation.

The scenario includes two apps: a benign victim *URL Expander* app U and an *attacker* app A. Let's assume that U defines the manifest file in Figure 1. U is granted the special permission *INTERNET* to query the URL shortening service. An intent filter is defined to offer the URL expansion service to other apps. U extracts the URL to expand from an incoming intent message. The app queries the URL shortening service and expands the URL. U is meant to query only trusted URL shortening services, those listed in the intent filter (*goo.gl* in the example).

However, U fails to correctly validate intent data, and when a server controlled by the attacker is (explicitly) specified as a shortened URL, instead of rejecting the message, U tries to expand the URL by directly connecting to it.

Even if the attacking app A is not fully trusted, it was installed by the final user because it requested no permission,

[1]net.studiofly.android.yuzu
[2]com.github.nicolassmith.urlevaluator

thus it was assumed harmless. Moreover, firewalling and sand-boxing are expected to preserve security by guaranteeing separation among apps. This is true until U contains no vulnerability.

In our attack context, A sends a malicious message as explicit intent to U. The intent data is *https://evil.com/abcd* and it corresponds to a host controlled by the attacker. However, U's validation is defective. U does not check the host name to block untrusted domains, but queries it directly. The malicious host returns a long URL that encodes a piece of malware as ASCII characters. If the size of a URL is not large enough, several similar intents can be used to collect all the parts of the malware. Expanded URL(s) are returned to A that can decode and reconstruct the piece of malware.

As a result of inadequate message validation, U takes a privileged action on behalf of the attacker A using data controlled by A. Eventually, A is able to download malware on demand (a most recent version and, possibly, device specific) without INTERNET permission[3].

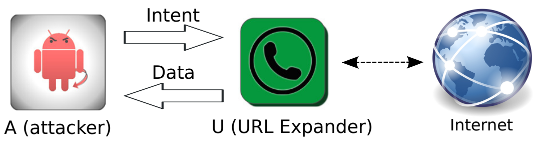


Figure 2: Example of attack scenario.

3.3 Vulnerability Preconditions

Based on the previous attack scenario, we identify the preconditions that a vulnerability should meet in order to be exposed to attacks based on inadequate message validation.

Sensitive actions can be performed only by apps that are granted the permission to access the corresponding resource. An attacker app that misses the permission to access sensitive resources might resort to other vulnerable apps that hold the needed access right. The goal of the attacker is to make the vulnerable app execute privileged actions on its behalf. Thus, the first precondition of this vulnerability is the following:

> ***Precondition PR_1: Privileged API call.*** *While performing the action requested by the intent message, the vulnerable app calls a privileged API.*

Using the example of Figure 1, this corresponds to an app that, after receiving an intent from a requester attacking app, accesses the Internet invoking the API *HttpURLConnection.connect()* that requires the special permission INTERNET. This is a case of permission re-delegation, as described by Felt et al. [12], because the vulnerable app performs a privileged task on behalf of a second app that misses the required permission.

However, as acknowledged by Felt et al., permission re-delegation includes also cases of legitimate delegation. Our threat model goes beyond that and requires additional preconditions to distinguish between legitimate delegation and attacks. A first precondition to assume to make delegation non-legitimate is on data used in the privileged action. Data used in the privileged action should come from the intent message, which is controlled by the attacker. Thus, the following precondition should be also satisfied:

> ***Precondition PR_2: Attacker data.*** *A privilege API is called using data coming from the intent message sent by the attacker app.*

In the running example, the string with the query to the URL shortening service contains the shortened URL, which comes from the intent message. However, this can be still a case of legitimate delegation. A real attack would use a malicious data that a properly implemented app would normally discard.

Despite intent filters have not been defined explicitly for security, nonetheless intent filters reflect the intention of the developer on the format of incoming data. In case data are received that clearly violate the filters, they violate the developer intention, so they are likely cases of not legitimate delegation, i.e. malicious intents meant to attack the app.

The Android framework inspects intent filters before delivering *implicit* intent messages to identify the most appropriate destination app. Thus, the destination app is confident to receive only valid messages, i.e. messages that are compatible with the protocol defined in the intent filter. However, *explicit* intent messages already contain the name of the destination app, therefore Android delivers the messages immediately without verifying their content.

While in implicit intent messages validation is performed by the (well tested and mature) Android framework, on explicit intents validation of messages is completely in charge of the receiving app, which may implement incomplete or partial validation.

The final precondition needed for delegation to be malicious is message validation. A vulnerable app accepts and processes an invalid message that does not satisfy the conditions defined in the intent filter.

> ***Precondition PR_3: Incomplete data validation.*** *The validation of intent data implemented in the app code is defective, because it accepts a message that violates the app intent filter.*

An intent message requesting to expand the "short" URL *https://evil.com/abcd* clearly violates the intent filter of Figure 1, because the host is a server controlled by the attacker, *evil.com*, instead of the trusted server *goo.gl*.

A defect that satisfies all of these three preconditions represents a security issue: an invalid intent message is processed as if it was valid and intent data is used to call privileged API. In this way, an attacker app manages to control the execution of a privileged action without having the corresponding permission. We call this an *Android Wicked Delegation*, (AWiDe).

The objective of the rest of the paper is to elaborate and assess two automated approaches to identify when an app contains AWiDe vulnerabilities. Our analysis consists in tracing (either statically or dynamically) malicious data from intent messages, and in detecting when a malicious data is used in sensitive operations, namely privileged API calls.

4. STATIC ANALYSIS FOR VULNERABILITY DETECTION

[3]Even if A could access directly the network, this attacks allows downloading a malware stealthily, because network usage would be imputed to U.

The first approach we propose is based on static taint analysis. We instantiate static flow analysis to trace the dependencies on intent values through the app control flow. When (potentially malicious) intent values are used in privileged API calls, the tool reports a candidate AWiDe vulnerability.

4.1 Taint Analysis

Taint analysis is intended to track the tainted/untainted status of values throughout the app control flow. A problem is reported whenever a tainted value is used in a security sensitive statement, called the *sink*. Tainted status is propagated on assignments to the variable on the left hand side, when an expression on the right hand side uses a tainted value. Tainted variables become untainted upon sanitization, by means of special functions or when they are assigned untainted values, such as a constant value or an expression that contains no tainted value.

Two variants of taint analysis, static [24] and dynamic [9], are possible. Static taint analysis is formulated as a flow analysis [23] problem, where the information propagated in the control flow graph is the set of variables potentially holding tainted values. Static taint analysis detects candidate vulnerabilities as paths in the control flow that make tainted values reach a sink (i.e., a vulnerable statement). However, static taint analysis is conservative, because flow information is propagated through all the paths, potentially including infeasible paths that cannot be executed.

Dynamic taint analysis, instead, updates the tainted/untainted flag of program variables during execution. Often, taint tracking is implemented by using a modified execution environment (e.g., a virtual machine), deployed to keep track of the tainted flag of program variables. Dynamic taint analysis is more precise but incomplete, because it analyses the app just with respect to the current observed execution, at the cost of some runtime overhead. A problem is reported when a value tagged as tainted is used in a vulnerable statement (the sink).

4.2 Static Analysis

In our analysis, FlowDroid [3] was used for static taint analysis. FlowDroid is a static analysis tool that reasons about information flow at the level of variables, by finding paths from sources to sinks. This tool was initially conceived to trace data flow to detect privacy leaks in the form of sensitive data sources (e.g., from the contact lists) that are disclosed on sinks, for example through network messages. It also accepts options to run analysis on different precision levels.

We instantiated static taint analysis by changing the configuration of sources and sinks to meet our analysis requirements. In our threat model, we are interested in tracing how intent data flows in the app and potentially reaches a privileged instruction, i.e. an API that requires a special permission to be executed.

Sources: According to precondition PR_2, tainted data comes from intents sent by other apps. The API call used to read the incoming intents is the method `Intent.getIntent()`. This call returns an *Intent* object that contains the message payload. All fields of this object will be considered tainted.

Sinks: According to precondition PR_1, the sinks are all the method calls that require a special permission. However, there is no complete documentation of what are the privileged APIs in Android. Thus we reused the result of the work by Au et al. [4], who discovered these APIs using dynamic analysis. Sinks consist of 32,203 method calls, spread across the entire Android library classes.

Decreasing Precision Level: FlowDroid supports analysis with different levels of precision. So, we set a time limit for the analysis at the most precise level. We stop the analysis of an app that can not complete within 10 minutes, and we restart the analysis with a less precise configuration. We iterate this process until the analysis complete or if all the levels are attempted. We set this time budget by observing that FlowDroid is in general very fast and that when it does not complete in 10 minutes, it is unlikely that it will ever finish.

Filtering: The report filled by FlowDroid specifies what pair of source-sink is connected by a data flow.

Due to over-tainting (and possibly less precise analysis option), the report sometimes contains cases where our source (i.e., method `Intent.getIntent()`) is not involved at all. This can be also caused by a known bug[4] on FlowDroid's taint wrapping which, in particular conditions, propagates taintedness from a field to its class.

Thus, we have to filter the cases reported by FlowDroid and keep only those source-sink pairs that explicitly mention `Intent.getIntent()` as source.

Intent Filter Violation: With respect to vulnerability preconditions of Section 3.3, so far static analysis checked just preconditions PR_1 and PR_2, i.e. data from the intent is used in a privileged API call. However, static analysis does not check precondition PR_3, i.e., the validity of intent data. The flows reported as vulnerable are supposed to be checked manually. This consists in verifying if the source code of the app checks intent data for the same conditions that are specified in the intent filter. Moreover, to qualify the security report as a true positive, an input should be elaborated that satisfies all the vulnerabilities preconditions.

5. DYNAMIC ANALYSIS FOR VULNERABILITY DETECTION

The second approach we propose relies on dynamic analysis. The execution of the app under analysis requires proper input data, i.e. intents. Intents are automatically generated that satisfy precondition PR_3 (see Section 3.3), by checking whether they violate the intent filters of the app under analysis. Then, trace analysis is used to identify what executions hit a privileged API (precondition PR_1). Eventually, precondition PR_2 holds for those intents whose data is used in privileged API calls. Dynamic taint analysis is applied to check this last condition.

5.1 Data Generation

Vulnerability precondition PR_3 requires input data (intents) to violate the intent filters defined by the app under analysis. An intent filter specifies a set of conditions and *all* the conditions should hold for an intent to be considered valid. Consequently, to violate a filter, an intent needs to violate just one condition. In order to maximize condition coverage, we adopt a heuristic approach to generate input data. Each generated input violates at least one condition in the intent filter.

[4]https://github.com/secure-software-engineering/soot-infoflow-android/issues/43

Before starting the generation of test data, the manifest file of the app is parsed to extract information about the intent filters. Since there could be many filters, each of them is subject to test data generation. The first intent to be generated is the *prototype*, a valid intent that *does* satisfy all the conditions in the filter.

For example, for the intent filter in Figure 1 we automatically create a prototype intent with action=*VIEW*, category=*DEFAULT* and data=*https://goo.gl/IhH0Ix*.

A new testing intent is generated by mutating the prototype such that it negates one of the conditions in the filter. The mutation operators are:

- *Change action*: The action of the intent is changed into a new action that is not specified in the intent filter. For example, the action is changed from *VIEW* to *EDIT*;

- *Change category*: The category of the intent is changed into a new category that is not specified in the intent filter. For example the category of the prototype is changed from *DEFAULT* to *PREFERENCE*;

- *Change data*: The data in the intent is changed such that just one filter condition on data is not satisfied. Since the data has the format *scheme://host:port/path*, a change is applied on any of the composing parts (i.e., MIME type, scheme, host name, port, path, etc.).

  In the case of the running example, we could negate the condition on the scheme and change it from *https://* to *http://*. Alternatively, the host might be changed from *goo.gl* to *evil.com*.

After mutation, we need to check whether the test intents violate the intent filter (precondition PR_3). To assess this, we rely on the Android framework. We send test intents as implicit intents (without specifying any destination). The Android framework will rely on intent filters to decide the proper destination. We only keep those intents that were *not* delivered to the app under test, because they violate its intent filter. If these intents are explicitly sent to the test app, then the app should discard them. If the app processes these intents we can say that precondition PR_3 is met.

5.2 Trace Analysis

Generated intents need to be sent to the app under test to verify if they satisfy precondition PR_1, i.e. the app fails to reject invalid intent messages and the execution triggers a privileged API call. A way to achieve this is by comparing the behaviour of the app under test with trace analysis. We revoke all permissions from an app and we check if its execution on a given input raises errors due to denied permission.

In more details, trace analysis is performed by running two variants of the app under analysis, P and P'. P is the original app, while P' is the original app with its permissions removed. P and P' are instrumented to trace method execution, exceptions and errors (both handled and unhandled). Instrumentation is done directly at the byte-code level by using an AspectJ[5] aspect that injects new code to record all the method executions and all the exceptions thrown at runtime.

Each test intent is sent as explicit intent to P and to P' and execution traces are compared. Three cases are possible:

[5]AspectJ is the aspect oriented version of Java, available at https://eclipse.org/aspectj/.

1. *No error*: In case of equal traces with no errors, it means that the execution related to a given intent with and without permission does not pose any difference. Therefore, the intent is either discarded by both P and P', or it is not discarded but no protected API is called. Thus, according to our threat model, no defect is found in the intent validation routine;

2. *Same errors*: In case the two traces present the same error, with and without permission, it means that the malformed intent makes P and P' fail in the same way, but without triggering any privileged API call (otherwise there would have been a difference in the traced errors, i.e., an error in P' that involves the missing access permission). Therefore, even if the intent is not rejected, no sensitive action has been performed.

3. *Different errors*: If the execution traces are different, it means that there has been an error in one of the executions. As the only difference between P and P' is in the permissions, a variation in the trace means that the specific intent makes app P' trigger a privileged API call that causes an error due to an insufficient permission. In this case, the test could expose potential security vulnerability because the invalid intent is not rejected (precondition PR_3) and forces the app to perform a privileged action (precondition PR_1).

At the end of trace analysis we know exactly what privileged APIs are executed, these are the calls that caused a permission violation difference between P and P'. The calls detected by us might also include APIs that are potentially missed by Au et al. [4] (as acknowledged by Au et al., their list could be partial).

The only remaining check is to verify if intent data has been used in these privileged calls (precondition PR_2).

5.3 Dynamic Taint Analysis

An invalid intent that makes the app execute a privileged instruction does not necessarily represent a vulnerability. In fact, the app may perform privileged actions for reasons that are not directly related to the received intent. For example, the app may need to access the Internet to update a local cache or to check for updates. We register a potential security problem when precondition PR_2 is also satisfied, i.e. when the privileged API call uses potentially malicious data that comes from the invalid intent.

To verify if intent data is used in a privileged API call, we instantiate dynamic taint analysis. All data fields from the intent are marked as tainted and taintedness is propagated on assignments during the execution. The privileged APIs that caused error during trace analysis are then checked to see whether they are called using any tainted data.

TaintDroid [10] is used for dynamic taint analysis, with some modifications to adapt the tool to our security analysis. TaintDroid is a dynamic taint analysis tool for Android, intended to reveal sensitive information disclosure during execution. In TaintDroid, data is tagged as tainted when it comes from a privacy sensitive source, such as the address book or the GPS. Sinks are represented by all those possible ways a piece of information can leave the device, such as through network transmissions or outgoing text messages. Dynamic taint tracking relies on a modified version of Android that checks the tainted flag of variables on assignments in byte-code, and conservatively in native calls, inter-process messaging and file system.

First, taint sources of TaintDroid differ from the ones we need for our analysis. In fact, TaintDroid was intended to monitor the flow of private data, while our analysis is intended to track intent data. Second, while sinks in TaintDroid are points where data leaks from the device (e.g., via the network), in our analysis sinks are all the privileged API calls (that have been identified by trace analysis), including also access to local private databases not considered originally by TaintDroid. For example, the act of writing a new contact is a privileged action (it fails if the app is not granted the proper permission), but it is not a sink for TaintDroid, because no information leaves the device. Despite these differences, the propagation of tainted/untainted values in TaintDroid perfectly fits our needs.

To adapt TaintDroid to our needs, we deploy an aspect written in AspectJ, that sets the taint flag of data from the incoming intent (sources) and that checks the taint flag of data on privileged API calls (sinks). While the sources are constant, i.e. they are the calls to `Intent.getIntent()`, sinks might change because they depend on the result of trace analysis. An aspect is automatically generated to intercept the privileged API calls detected by trace analysis. After this aspect is applied to the app byte-code, the app is executed with test intents on TaintDroid to check whether tainted data is used in sink calls. The automatically generated aspect reports if test intents satisfy precondition PR_2.

6. EMPIRICAL RESULTS

This section reports a study that assesses our vulnerability detection, when analysing a set of 329 apps to identify potential AWiDe security defects.

6.1 Research Questions

The aim of this section is to investigate the following research question:

- *RQ_1:* How do static and dynamic analysis compare in detecting Android Wicked Delegation vulnerabilities?

This question aims at studying the detection accuracy of static and dynamic analysis. Answering this research question would suggest what is the most appropriate tool to support developers of Android apps.

6.2 Metrics

To answer this question, we apply the proposed detection techniques to the same case studies. We then validate detection results by manually filtering the reported vulnerabilities. During this process we record these metrics:

- *True positives:* Number of vulnerable apps correctly detected as vulnerable by a tool;
- *False positives:* Number of safe apps incorrectly detected as vulnerable by a tool (false alarms);

True positives and *False positives* are meant to quantify the accuracy of the techniques.

6.3 Subject Apps

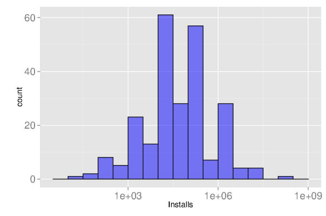
Our approaches work on compiled apps; therefore, availability of source code is not a requirement for the analysis. However, we decided to opt for open source projects because they offer the possibility to inspect the app code and manually verify the correctness of vulnerability detection.

The F-Droid repository[6] represents an ideal setting for our experimentation, because (i) it includes real world and popular apps that can be found also in the official market, and (ii) apps can be downloaded with their source code, for manual validation of the security reports delivered by the tools.

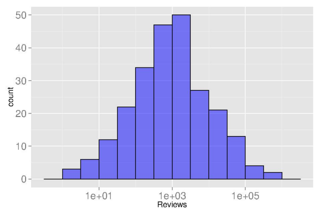
The whole app catalogue was downloaded in October 2014, and it consisted of a total of 1,216 apps. Sometimes, multiple versions were available for the same app. In these cases we considered the most recent version.

Apps that specify no intent filters do not support delegation, so they can not be target of delegation attacks. In the empirical assessment we only consider apps that define intent filters. Among the 329 apps that do define intent filters, more than 70% are also present in the official Google Play Store and some of them (in particular games) are also quite popular there. We queried the official Android store to study the popularity of the apps considered in our study. Figure 3 shows the number of user reviews and the number of installs reported in the official Android store. While the number of installs ranges from 30 to 300 million, the average number of installs of the case study apps is 2.3 million. The minimum number of reviews is 0 and the maximum is 700.000, while the average number of reviews is 12,803.

The number of installs and reviews supports the claim that our study considered popular apps that can be found in the official store.



(a) App installs



(b) App reviews

Figure 3: Popularity (installs and reviews) of case study apps according to the official Android store.

6.4 Results

Table 1: Detailed intermediate results of the static and dynamic analysis.

(a) Static analysis.

Analysed apps	With source-sink path	Filtered for "getIntent"
329	141	53

(b) Dynamic analysis.

Analysed apps	Apps calling privileged APIs	Tainted data on sink
329	84	10

[6]`http://f-droid.org/`

Table 2: Apps analysed with different precision level.

Options used	Number of apps
Default configuration	222
–NOCALLBACKS	40
–ALIASFLOWINS	4
–PATHALGO SOURCEONLY	1
–ALIASFLOWINS, –NOCALLBACKS, –NOEXCEPTION	6
(Timeout)	56
Total	329

Detailed results of static analysis are shown in Table 1(a). Out of the 329 apps that can actually be targeted by delegation attacks, i.e. they define intent filters and permission requests in their manifest XML files, static analysis could complete only on 273 apps (time out in 17% of the cases), because of limitations of the tool we used. Among them, 141 apps were reported as containing a flow from the source to a sink. Further automatic filtering was required to exclude vulnerability reports that did not include `getIntent` as source. The final result of static analysis consists of 53 apps classified as affected by the AWiDe vulnerabilities.

Table 2 shows the level of precision used in the analysis. 222 apps were analysed with the default analysis option. A slightly less precise analysis (without considering callbacks) allowed completing static taint analysis in 40 apps. Even less precise analysis was required in fewer cases. However, on 56 apps, even the less precise analysis did not deliver any result by the time out (17% of the cases).

Table 1(b) summarizes the results of dynamic analysis. The 329 apps with intent filters and permissions were instrumented and analysed. Trace analysis revealed 84 apps that invoked privileged APIs after receiving an invalid intent that violated their intent filters. Among them, our modified version of TaintDroid detected 10 cases as vulnerable, because a tainted data from an intent was actually used in privileged API calls.

Table 3: Final results of automatic tools in detecting AWiDe vulnerabilities.

Metric	Static analysis	Dynamic analysis
True positives	9	9
False positives	44	1

Manual inspection of source code was required to assess the accuracy of the analysis, to check whether reported vulnerabilities were true positives. Manual inspection of source code consists of (i) verifying that intent data is used in the privileged call; (ii) that no (or partial) validation of intent data is enforced before using it on privileged call; and (iii), only for static analysis, elaborating an input value that satisfies all the vulnerability preconditions.

The comparison of the results of the two approaches is summarized in Table 3. Among the 53 apps classified as vulnerable by static analysis, we identified 9 apps as true positive because they contain at least one vulnerable flow from source to sink that misses adequate data validation. The remaining 44 cases were false positives due to the reasons mentioned in Section 4 (conservative analysis and overtainting).

After manual inspection of the results of the dynamic analysis, we classified the reported cases as 9 true positives and 1 false positive. The false positive case was due to a bug in our implementation that was difficult to solve, because it involves interference between taint tags for our analysis and privacy leak taint tags originally propagated by TaintDroid. Table 4 shows details of the actual discovered vulnerabilities, with the special permission related to the privileged API call. Only 3 apps have been correctly classified as vulnerable by both static and dynamic analysis.

We analysed more in depth the other cases, detected only by one or the other approach. Among the 6 cases missed by dynamic analysis, in 5 cases none of the test case generated by dynamic analysis could execute the vulnerable path detected by static analysis. The remaining case is a limitation of our implementation on a particular filter protocol.

Conversely, considering the 6 cases missed by static analysis, in 2 cases static analysis got stuck and did not complete. In the remaining 4 cases, the vulnerable data flow traverses a system library not modelled by the static analysis tool, limited just to intra-component analysis. For example, *Zirco Browser* receives (potentially malicious) URLs from other apps through intent messages and displays them using WebView, a system component. This inter-component interaction is missed by FlowDroid hence reporting no flow.

Then, we compared the amount of time required to complete the analysis. While static analysis was quite fast (135' for the 273 apps that completed), dynamic analysis, even if fully automated, took more time (5 days). Long execution time of dynamic analysis is due to the large set of input values to test. Each intent required at least two distinct executions in the instrumented environment (trace analysis requires running an app with and without permissions). An additional execution with TaintDroid was required for those apps that passed the trace analysis phase. The set of inputs for the 329 apps consisted of 9,756 intents. It should be noted here that we do not perform combinatorial testing, because each test intent violates just one condition among those from the intent filter.

A major difference was also observed in number of apps with incomplete analysis. While dynamic analysis always delivered a result, FlowDroid (static analysis) could not complete the analysis on 56 apps (17%), for reasons related to limitations of the tool version we used for the experiment.

6.5 Outcome

All in all, experimental results allow us to answer the research question in the following way:

- Both static analysis and dynamic analysis report the same number of true positives (both 9 cases, however, the overlap is on 3 apps). On the other hand, static analysis reports many more false positives than dynamic analysis (44 vs. 1, respectively).

The outcome of the experiments also allows us to formalize the following considerations:

- Static analysis is faster than dynamic analysis (minutes compared to days).
- Android Wicked Delegation vulnerabilities affect real-world apps. In a set of 329 apps, we found 15 apps that match the threat model defined by us.

Table 4: Apps correctly detected as vulnerable by static and dynamic analysis.

App name	Static Analysis	Dynamic Analysis
DAAP	Internet	—
Scid on the go	Internet	—
Short URL Evaluator	Internet	—
TunesViewer	Internet	—
OpenSudoku	Internet	Internet
Call Meter 3G	Internet	Internet
ACV	Internet	—
Moss	Internet	Internet
ZooBorns	Set Wallpaper	—
SipDroid	—	Call Phone
Lumicall	—	Call Phone
Car Cast	—	Internet
Ermete SMS	—	Read Contacts
Zirco Browser	—	Internet
Crosswords	—	Internet

6.6 Attacks

To show that the vulnerabilities discovered by us can be exploited to port serious attacks, we elaborated attacks[7] starting from the results of our analysis techniques. Table 4 summarizes the permissions that can be abused on vulnerable apps. For space reason, in this section we present only some of them.

The first attack is on *SipDroid*, an app with more than 1 million downloads from the Android market. It is an open-source SIP client that is granted the `CALL_PHONE` permission to make phone calls. A particular activity of this app defines an intent filter to accept implicit intents with data schemes related to messaging (schemes are `sms://` and `smsto://`). However, the app's implementation fails to enforce these schemes on incoming intents. Therefore, it accepts and processes explicit intents with scheme `tel://` that does not match the intent filter. As a result, the app dials any number specified in the intent.

This vulnerability can be used to port complete attacks either by, (i) making the vulnerable app dial a premium number or, (ii) performing a USSD/MMI attack [15, 6]. USSD/MMI codes are numeric strings between the "*" or "#" characters. They are meant to access services supplied by the mobile operator or to access phone functions. This attack can lead to the factory reset of the device on a very popular device from a mainstream vendor [7], or play with operator configurations (e.g., #793#, which resets the voicemail password on T-Mobile USA).

Lumicall is another telephony app that borrows code from SipDroid, which makes it vulnerable to the same attack. Vulnerabilities in both these apps have been reported to the authors[8].

An attack related to information leak can be ported to an app detected by both the techniques. The app is *OpenSudoku*, an open-source version of the popular *Sudoku* game. Besides the normal game play, one of its functions is the possibility to import new sudoku schemes from the Internet, using the special permission `INTERNET`.

While the intent filter allows to open streams only towards a subset of URLs (e.g., a specific MIME-Type or *.opensudoku* file extension), these restrictions are not enforced by the app code. The URL from an invalid intent is directly interpreted as a source to load the sudoku scheme definition data.

A malicious app without permission to access the Internet can still leak sensitive data (e.g., the device serial number *1234*) by sending data in parameter as GET request. The URL should refer to a server controlled by the attacker, for example *evil.com*. The complete attack contains the URL *http://evil.com/?deviceid=1234*. When receiving this intent, *OpenSudoku* loads the URL and leaks the device serial number to the malicious server. The vulnerability we spotted in this app has been reported to the developer who confirmed it[9].

Another interesting vulnerability that might allow an attacker publish as well as download file from a malicious host exists in the app *Short URL Evaluator*. The details of the attack are omitted because of space constraint, but they are similar to the running example of Section 3.2. They can be found in the vulnerability report filed by us[10].

7. DISCUSSION

7.1 Considerations

Static analysis reports more false positives: Static analysis and dynamic analysis report the same number of true positives, but static analysis reports a larger number of false positives. This result suggests that both analysis methods are effective in detecting defects. However, it is important to consider the large number of false positives that are also involved (44 versus 1). Valuable development time should be invested in inspecting each potential security problem. High false positive rate could potentially make app developers miss their time-to-market objective. Thus, dynamic analysis, with lower number of false positives, is an effective option. In fact, vulnerabilities that are reported by dynamic analysis are more likely to be instances of real problems that deserve high priority in manual investigation.

Dynamic analysis requires more time: We observed a huge difference in the amount of time required by the two analyses to complete. While static analysis is very fast (minutes), dynamic analysis took a lot more (days). This probably reduces the possibility of using dynamic analysis as a tool to check code quality by app-store managers, i.e., where a large set of apps has to be considered. In that context, the more lightweight approach of static analysis is probably more appropriate. However, if we consider the case of an app developer, who just needs to inspect the quality of a single app during development, both static and dynamic analysis are viable approaches. In fact, dynamic analysis of one app took on average one hour, which is still acceptable in this second context.

[7]These attacks are not meant to cause real damages or steal real data, their purpose is to demonstrate that a vulnerability is exploitable. In security terms they are called *proof-of-concept attacks*.

[8]https://code.google.com/p/sipdroid/issues/detail?id=1183
https://github.com/opentelecoms-org/lumicall/issues/32

[9]https://code.google.com/p/opensudoku-android/issues/detail?id=170
https://code.google.com/p/opensudoku-android/issues/detail?id=171

[10]https://github.com/nicolassmith/urlevaluator/issues/42

Manual filtering of security reports: Apart from the different false positive rate, security reports differ also in terms of information provided. Reports from static analysis just lists what *sink* statement is reachable with tainted input data. The developer needs to figure out by herself/himself how data can flow from the source to the sink. In our experience, this was quite easy when the source and the sink were located in the same class, or in the same method, because the control flow was quite easy to understand. However, it was much harder when the source and the sink were in different classes. In some cases the flow was complicated by the presence of multiple methods in multiple classes, and manual inspection was more time consuming.

The report of dynamic analysis, instead, consists of an executable scenario. It is a concrete instance of an input value that triggers the discovered vulnerability in a candidate vulnerable flow. To classify one of these cases, the full understanding of the whole control flow was not required, and the developer could just focus on the particular execution flow taken by the test. Often, step-wise execution in debug mode was enough to quickly figure out the vulnerable data flow and confirm or discard the reported case.

Fast understanding and filtering of executable scenarios makes dynamic analysis reports more appropriate for a fast time-to-market business model.

User interaction: Static analysis considers all the potential flows, including those that may potentially require external events (e.g., user intervention or system event) to execute a privileged API call. On the other hand, since it is not included in our execution scenarios, dynamic analysis does not consider any user interaction.

For this reason, AWiDe vulnerabilities detected by dynamic analysis are more direct and stealthy. A malicious app could exploit them directly, just by sending an accurately crafted intent, without the help of any subsequent event to execute the intended privileged API. Secondary events (user or system event) might be required to exploit vulnerabilities reported by static analysis. However, a user who might become suspicious could block these second cases of attacks when the confirmation required to perform an attack pops up out of the blue.

Vulnerabilities reported by dynamic analysis should be considered with higher precedence, and they would require immediate attention by the developers.

7.2 Limitations

We acknowledge these limitations on the two approaches we propose.

Static analysis checks all the vulnerability preconditions (see Section 3.3) but one. The only non automated check is whether input data violates the intent filter (i.e., precondition PR_3). As such, the report of static analysis is not fully compliant with our threat model. Manual inspection was required to identify cases of missing or incomplete validation in the source codes.

The current implementation of FlowDroid is affected by over-tainting, resulting in false positives. Communications with the maintainers of FlowDroid revealed that over-tainting would be reduced after they fixed some implementation limitations. Indeed, some cases of over-tainting that we observed in our experiment are connected to a documented bug of this tool.[11]

The approach based on static analysis cannot propagate taintedness through constructs that FlowDroid cannot resolve statically, such as reflective calls and calls to native code, which can be present in Android apps. Although some support could be implemented to model native code, reflective calls are hard to be handled statically in the general case. This limitation does not affect dynamic analysis, where actual calls can be observed.

In our dynamic analysis tool, trace analysis runs the app without permissions. An error due to insufficient privileges reveals the first protected API call. As a result, for each input value, only the first AWiDe vulnerability is detected by dynamic analysis. This defect should be fixed by developers, to make our dynamic analysis tool able to detect potentially subsequent security defects. Security checks performed just before release could pose a limitation to the applicability of dynamic analysis to a fast time-to-market development model. Security verification should be more integrated in the daily development activity.

Our dynamic analysis tool does not model user interaction. On one side, this allows detecting stealthy attacks that a final user would not notice. On the other side, this restricts the number of attacks that dynamic analysis would detect. For this reason, we recommend to use static analysis to complement the limitations of dynamic analysis.

Eventually, in future versions of Android, a different permission scheme will be adopted. Common permissions, such as the Internet access, will be granted by default to all the apps, without the need of end-user confirmation. This change will probably reduce the number of apps that are vulnerable according to the AWiDe threat model, because many cases of vulnerability were related to the Internet permission (see Table 4). However, less than 1% of the Android devices run the latest version of the framework (Android M) [12] that automatically grants Internet permission to all apps. Thus, misuse of the Internet permission is still a threat on most of the Android devices and our approach is still relevant to identify security defects related to Internet. Moreover, our approach is effective on the remaining set of sensitive permissions, for example to make phone calls.

8. CONCLUSION

Smart phone apps are often developed under a high time-to-market pressure. For this reasons, they are sometimes delivered while they still contain defects. Automatic support for fast quality verification is highly advisable.

In this paper, we present a novel case of malicious delegation, the Android Wicked Delegation. We also proposed two approaches to automatically identify this class of security defects. Static analysis showed to be faster than dynamic analysis, but less reliable and affected by more false positives.

In future work, we intend to integrate static and dynamic analysis to combine their strong points and overcome their limitations. Moreover, we plan to validate the effectiveness of our security reports with human studies to check what piece of information is the most important for developers when validating true vulnerabilities within a limited time budget.

[11]https://github.com/secure-software-engineering/soot-infoflow-android/issues/43

[12]http://developer.android.com/about/dashboards/index.html

9. REFERENCES

[1] D. Amalfitano, A. Fasolino, and P. Tramontana. A GUI crawling-based technique for Android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252 –261, march 2011.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[5] A. Avancini and M. Ceccato. Security testing of the communication among Android applications. In *Proceedings of the 8th International Workshop on Automation of Software Test*, pages 57–63. IEEE Press, 2013.

[6] R. Borgaonkar. Dirty use of USSD codes in cellular networks. In *Ekoparty Security Conference*, 2012.

[7] R. Borgaonkar. Demo: Dirty use of USSD codes, https://www.youtube.com/watch?v=q2-0b04hphs. last accessed August 2015.

[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] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[10] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th Usenix Symposium on Operating Systems Design and Implementation*, 2010.

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[12] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th Usenix Security Symposium*, 2011.

[13] Gartner. Smartphone sales in 2014, http://www.gartner.com/newsroom/id/2996817, last accessed August 2015.

[14] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*. The Internet Society, 2012.

[15] L. Hattersley. Samsung Galaxy S III secret USSD reset code discovered, http://www.macworld.co.uk/news/apple/samsung-galaxy-s-iii-secret-ussd-reset-code-discovered-3400408/, 2012.

[16] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[17] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[18] 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 (ICSE 2015)*, pages 280–291, 2015.

[19] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[20] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST)*, pages 22–28, 2012.

[21] A. Maji, F. Arshad, S. Bagchi, and J. Rellermeyer. An empirical study of the robustness of inter-component communication in Android. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1 –12, june 2012.

[22] C. Mann and A. Starostin. A framework for static detection of privacy leaks in Android applications. In *27th Symposium on Applied Computing (SAC): Computer Security Track*, pages 1457–1462, 2012.

[23] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two approaches to interprocedural data flow analysis, pages 189–233. Prentice Hall, 1981.

[24] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[25] F. Wei, S. Roy, X. Ou, and Robby. AmAndroid: A

precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[26] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. 2014.