DroidChecker: Analyzing Android Applications for Capability Leak

Patrick P.F. Chan
The University of Hong Kong
Pokfulam, Hong Kong
pfchan@cs.hku.hk

Lucas C.K. Hui
The University of Hong Kong
Pokfulam, Hong Kong
hui@cs.hku.hk

S.M. Yiu
The University of Hong Kong
Pokfulam, Hong Kong
smyiu@cs.hku.hk

ABSTRACT

While Apple has checked every app available on the App Store, Google takes another approach that allows anyone to publish apps on the Android Market. The openness of the Android Market attracts both benign and malicious developers. The security of the Android platform relies mainly on sandboxing applications and restricting their capabilities such that no application, by default, can perform any operations that would adversely impact other applications, the operating system, or the user. However, a recent research reported that a genuine but vulnerable application may leak its capabilities to other applications. When being leveraged, other applications can gain extra capabilities which they are not granted originally. We present DroidChecker, an Android application analyzing tool which searches for the aforementioned vulnerability in Android applications. DroidChecker uses interprocedural control flow graph searching and static taint checking to detect exploitable data paths in an Android application. We analyzed more than 1100 Android applications using DroidChecker and found 6 previously unknown vulnerable applications including the renowned Adobe Photoshop Express application. We have also developed a malicious application that exploits the previously unknown vulnerability found in the Adobe Photoshop Express application. We show that the malicious application, which is not granted any permissions, can access contacts on the phone with just a few lines of code.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Verification

General Terms

Algorithms, Security, Verification

Keywords

Android, Capability Leaks, Privilege Escalation Attack, Taint Checking, Control Flow Checking

WiSec'12, April 16-18, 2012, Tucson, Arizona, USA.

1. INTRODUCTION

Smartphones have evolved rapidly over the last few years. The number of smartphone users is also increasing tremendously. According to figures for 2010 released by Gartner [17], smartphones accounted for 297 million (19%) of the 1.6 billion mobile phones sold that year. That is 72.1% more smartphone sales than in 2009. Another survey conducted by thinkmobile with Google showed that 68% of the participants expressed that they used an app during the week before [40]. This shows that more and more people are using smartphones these days and most of them are using applications on their smartphones.

A recent research from Nielsen shows that Android now owns 29% market share of smartphone users in the US and is pulling ahead of RIM Blackberry (27%) and Apple iOS (27%) [30]. Moreover, the Android Market is the fastest growing mobile application platform. According to a recent report released by mobile security firm Lookout, the Android Market is growing at three times the rate of Apple's App store [25]. However, unlike the Apple's App store, there is no screening process of the apps being published on the Android Market. Occasionally, Google needs to take down some malicious apps from the Android Market after they are found containing malicious code.

Along with the increasing prevalence of smartphones, the security threats to them are also growing. In August this year, an uncovered trojan was found recording phone calls and the recorded calls could be uploaded to a server maintained by the attacker [41]. In June, another trojan, called GGTracker, was uncovered. It sends SMS messages to a premium-rate number and may also steal information from the device [26, 38]. According to a blog entry from lookout [26], after the victims click on a malicious in-app advertisement, they are directed to a malicious website which imitates the Android Market installation screen. The malicious website then lures the victims to download and install a malicious application. The victims are then subject to unpredicted charges.

Android is basically a privilege-separated operating system [2]. Every application runs with a distinct system identity in its own Davik virtual machine. This mechanism isolates applications from each other and from the system. By default, an application is not capable of performing any operations that would adversely impact other applications, the operating system, or the user. In order to obtain extra capabilities, an application needs to declare the permissions that it needs in its *AndroidManifest.xml*. At application installation time, the user decides whether to grant the permissions

to the application or not. During the runtime of the application, no further checks with the user are done. The application either was granted a particular permission when installed and can use that feature as desired or the permission was not granted and any attempt to use that feature will fail without prompting the user. For instance, to be able to access the Internet, an application needs to have the *INTERNET* permission.

However, recent research has showed that an application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee) [7]. Such attack is called privilege escalation attack or confused deputy attack. It allows a malicious application to indirectly acquire more capabilities leaked from a benign application which fails to guard the permissions granted to it. To prevent this attack, applications must enforce additional checks on permissions to ensure that the callers have the required permissions before doing any dangerous actions. Since most of the application developers are not security experts, delegating the task of performing these checks to them is an error prone approach.

The aim of this research is to develop an automatic analysis system for detecting capability leaks in Android applications and find out how prevalent they are in existing applications. The resulting system is useful for developers to make sure that their applications, while not malicious on their own, do not leak capabilities to other applications. Moreover, Android users can use our system to search for capability leaks in an application before installing it. We propose DroidChecker, an automatic capability leaks detection tool for Android applications.

DroidChecker first parses the *AndroidManifest.xml* file which defines, among other things, the permissions an application uses and the permissions other applications need in order to access the components of that application. Every Android application must include the *AndroidManifest.xml* in its Android package (APK) file as required by the Android system. After that, it identifies components that are potential sources of capability leaks. For each of such components, it looks into its source code and uses interprocedural control flow searching to follow the taint propagation. It then obtains data paths that lead to capability leaks. It raises an alarm whenever such paths are found.

We downloaded 1179 Android applications from the wild and scanned them using DroidChecker. 6 applications were found to have capability leaks with one of them being *Adobe Photoshop Express 1.3.1* (APE). We inspected the source code of APE and found that when it is leveraged, an application with no capability granted can read e-mail addresses of the contacts on the phone by calling a component of it. We illustrate such an attack with a simple application written by us.

The primary contributions of this paper are:

- **Algorithm.** We propose a novel approach to automatically detect capability leaks in Android applications. To the best of our knowledge, we are the first to use interprocedural control flow graph searching coupled with taint checking to detect capability leaks. This significantly increases the granularity and hence the accuracy of the detection. Since any alarms raised by the detection system will likely to be checked manually, a more accurate detection system means less human effort involved and more useable.
- **Evaluation.** We implement a prototype of the proposed approach and use the tool to scan 1179 Android applications from the wild. We demonstrate an attack with a simple application which leverages the capability leak identified by DroidChecker in *Adobe Photoshop Express 1.3.1* to get e-mail addresses from the contacts on the phone. This shows that our system can successfully detect previously unknown capability leak in real-world applications.
- **A Scalable Implementation.** We fully automate every step of the checking tool to make it scalable. It took less than an hour to finish checking all the 1179 Android applications in our testing set. Moreover, we do not achieve such scalability by limiting the method invocations tracing to a certain depth when doing interprocedural checking. Instead, we only avoid running into infinite loop by not checking the same method with argument(s) in the same tainted state twice.

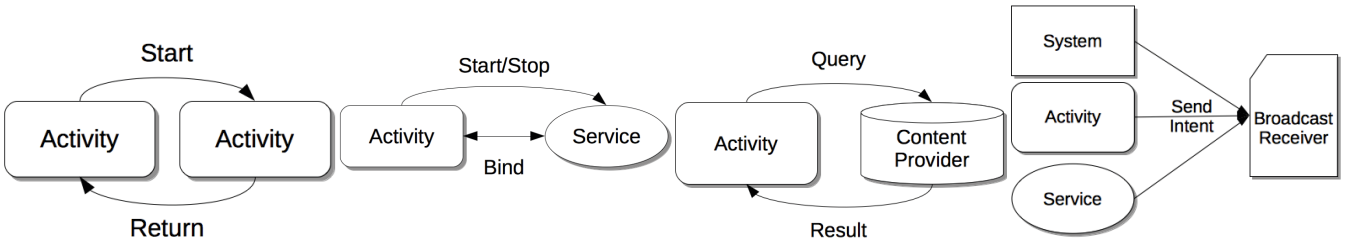
The rest of this paper is organized as follows: Section 2 provides some background information of Android; Section 3 describes the privilege escalation attack on Android; Section 4 provides the details of the design of our system; Section 5 reports the evaluation results; Section 6 illustrates an example attack; Section 7 discusses the limitations and future work. Section 8 surveys related work and compares it with our work; Section 9 concludes the paper.

2. BACKGROUND OF ANDROID

Android is a free, open source mobile platform based on the Linux kernel. Android applications are written in Java and composed of four types of application components: activities, services, content providers, and broadcast receivers. Components can communicate to each other and to components of other applications through an inter component communication (ICC) mechanism called intent messaging. An intent is a passive data structure holding an abstract description of an operation to be performed. To build performance-critical portions of the application in native code, developers can make use of the Android NDK companion tool which provides headers and libraries when programming in C or C++. However, inclusion of C or C++ libraries kicks away the security guarantees provided by the Java programming language. Several different vulnerabilities in native code of the JDK (Java Development Kit) have been identified [39]. In the rest of this section, the various ICC mechanisms for the four types of components will first be discussed. Following that, we will discuss about the four cornerstones of Android security.

2.1 Inter Component Communication

Figure 1 shows the ICC for the four types of components. There are different kinds of intent messages for each of them. For activity components, it can be started by *startActivity* method. The caller can attach some data into the intent message passed to the target activity. However, once the target activity is started, the caller is suspended and cannot interact with it. The control is passed back to the caller once the target activity has finished. In order to get data from the callee, the caller can use the *startActivityForResult* method instead. The difference between it and the *startActivity* method is that, the callee can pass an intent message



(a) Activity components ICC (b) Service components ICC (c) Content providers ICC (d) Broadcast receiver ICC

Figure 1: The ICC for the four types of components

back to the caller when it has finished. That is the only way the caller can get some data from the callee.

For service components, there are two ways to trigger their executions. The first way is to start the service with the *startService* method. But the service will run in background and the caller has no way to interact with it. The second way is to bind to the service and the caller can interact with it with Remote Procedure Call (RPC). The caller can send commands to the callee and retrieve data from it.

For content providers, it can be treated as a database and an activity component can issue queries to it and get back a result set. This is the only way to share data across applications. There are some content providers shipped with Android which provide convenient access to common data such as contact informations, calendar information, and media files.

Broadcast receivers receive intents sent by *sendBroadcast*. Such intents can be sent by the system or by other applications.

2.2 Cornerstones of Android Security

2.2.1 Sandboxing

Android is a privilege-separated operating system. Each application runs within its own distinct system identity and its own Dalvik virtual machine (DVM). System files are owned by either the "system" or the "root" user. As a result, an application can only access files it owns or files of other applications that are marked as readable / writable /executable for others explicitly. This provides a sandbox for each application which isolates it from other applications and from the system. Applications signed with the same signature can request for being assigned to the same user ID by using the *sharedUserId* attribute in the *AndroidManifest.xml*'s *manifest* tag of each package. Consequently, the two packages are then treated as the same application and can access the same set of files.

2.2.2 Application Signing

Each application must be signed with a certificate and the corresponding private key is held by its developer. The certificate is used solely for distinguishing application authors. It does not need to be signed by a certificate authority. Typically, it is a self-signed certificate. It is used by the Android system to decide whether to grant or deny application access to signature-level permissions and whether to grant or deny an application's request to be given the same Linux identity as another application. The certificate is included in its APK file such that the signature made by the developer can be validated at installation time.

2.2.3 Permissions

Additional finer-grained security features are provided through a "permission" mechanism that enforces restrictions on the specific operations that a particular process can perform. Also, applications can make use of per-URI permissions to grant ad-hoc access to specific pieces of data. By default, an application has no permission to perform any operations that would adversely impact other applications, the operating system, or the user. To share resources and data with other applications, an application must declare the permissions it needs for additional capabilities not provided by the basic sandbox. The permissions an application requires are declared in the *AndroidManifest.xml* file which is compulsory for all applications. At installation time, the Android system prompts the user for consent. It relies on the user to judge whether he or she permits the application to use all the permissions it requires or refuses to install the application.

2.2.4 Accessibility of Components

Application components can be specified as public or private. A public component can be accessed by other applications. However, it can still perform permission checking to restrict access to only applications that own certain permissions. On the other hand, a private component is only accessible by components within the same application. Making a component private simplifies the security specification. The application developer does not need to assign permission label to it and care about how another application might acquire that label.

3. PRIVILEGE ESCALATION ATTACK ON ANDROID

In this section, we give the details of the privilege escalation attack or confused deputy attack on Android. The attack was first proposed by Lucas Davi et al. in [7]. They stated the problem as follows:

An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee).

Figure 2 illustrates an example of privilege escalation attack on Android. In the figure, there are three applications running in their own DVMs. Application 1 owns no permissions. Since components in application 2 is not guarded by

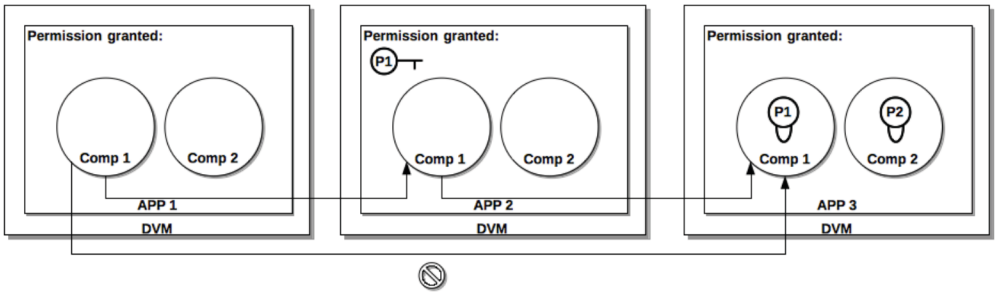


Figure 2: Privilege Escalation Attack

any permissions, they are accessible by components of any other applications. As a result, both components of application 1 can access components 1 in application 2. Application 2 own permission *P1*, Therefore, both components of application 2 can access component 1 of application 3 which is protected by permission *P1*.

We can see that component 1 of application 1 is accessing component 1 of application 2. However, since it does not have permission *P1*, it is not allowed to access component 1 of application 3. On the other hand, application 2 owns permission *P1*. Hence, component 1 of application 2 is allowed to access component 1 of application 3. Therefore, although component 1 of application 1 is not allowed to access component 1 of application 3, component 1 of application 1 can access it via component 1 of application 2. Therefore, the privilege of application 2 is escalated to application 1 in this case.

In order to prevent this attack, component 1 of application 2 should enforce that components accessing it must possess permission *P1*. This can be done at code level using the *checkPermission* API call or by guarding component 1 by permission *P2*. However, most developers are not security experts. They may not be aware of the possibility of leaking capabilities through their applications and the consequences of it. Even if they are aware of it, developers are not motivated to take measures to prevent leaking capabilities as the deputy itself is not harmed in the attack.

The consequences of a capability leak can be serious as Android provides a set of functionality-rich API calls which can get the current location, send text messages, make calls, and reading information on NFC cards. Recently, android applications can also be used to control devices using the Android ADK framework. [19]. Applications granted with permissions to perform such dangerous actions should be carefully protected.

It is obvious that in order to launch such an attack, the existent of application 2 (the deputy) is crucial as it serves as the stepping stone for application 1 to acquire extra capability. More specifically, it leaks to other applications its capability of accessing component 1 of application 3. Our research goal is to identify such capability leaks in applications.

4. SYSTEM DESIGN

4.1 System overview

Figure 3 shows the overview of our system. The APK file of the Android application to be analyzed is first converted into a JAR file. This is done by *dex2jar* which is a tool for converting Android's *.dex* format to Java's *.class* format [20]. It takes the whole APK file as input and gives a JAR file which contains the *.class* files as output. After that, the manifest file (*AndroidManifest.xml*) is extracted from the JAR file for further inspection. The manifest file defines, among other things, the permissions an application uses and the permissions other applications need in order to access the components of that application. Since the manifest file is in binary XML format, it is first converted back to human-readable XML using another tool called AXML-Printer2 [18]. By looking at the manifest file, we know the answers to the following questions:

1. Is the application asking for additional capabilities by requiring permissions?
2. What are the components that are publicly accessible by other applications?
3. Are those publicly accessible components guarded by permissions?

We then obtain a list of components that have the potential of leaking capabilities. These are the components that have extra capabilities, are publicly accessible, and are not guarded by any permissions. Finally, using interprocedural control flow graph searching and static taint checking, we look for data paths in the source files of these components that will lead to capability leaks. The source files are obtained by decompiling the class files in the JAR archive using Java Decompiler which is the latest Java decompiler [10].

In the rest of this section, we will discuss in details the two key modules, manifest file parsing and capability leak detection.

4.2 Manifest File Parsing

Before going into technical details of the manifest file parsing module, we discuss the structure of the manifest file and briefly explain the various tags found in it. After that, the high-level checking policy is introduced. Finally, we give the low-level details of the manifest file parsing.

4.2.1 The AndroidManifest.xml

Figure 4 shows an abstract of the general structure of the manifest file. The *uses-permission* tag requests a permission that the application must be granted in order for it to operate correctly. The *permission* tag declares a security permission that can be used to limit access to specific components or features. The *android:permission* attribute of the application tag declares the permission that components of

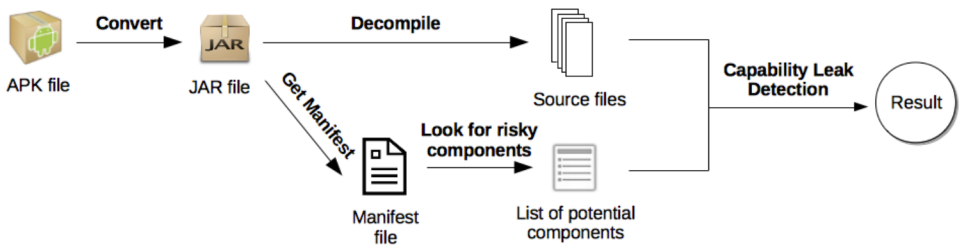


Figure 3: System Overview

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
  <uses-permission />
  <permission />
  <application android:permission="...">
    <activity android:permission="..." android:exported="...">
      <intent-filter> ... </intent-filter>
    </activity>
    <service android:permission="..." android:exported="...">
      <intent-filter> ... </intent-filter>
    </service>
  </application>
</manifest>
```

Figure 4: General structure of AndroidManifest.xml

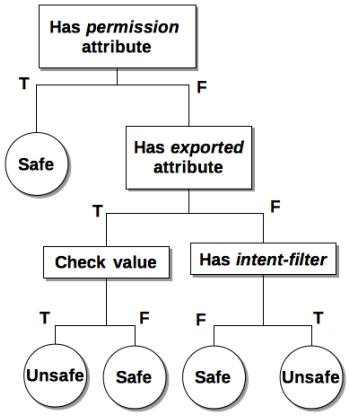


Figure 5: The decision tree of XML parsing

other applications must have in order to interact with the application. Moreover, each component can require extra permission for accessing it. They are declared in the *android:permission* attribute of the tag that declares the component. A component can also be made private by setting the *android:exported* attribute to *false*. Such a component is not accessible by components of other applications. If the *android:exported* attribute is absent, the default value of it depends on whether the component contains intent filters (except for Content Providers, which have the default value being *true*). If there is no intent filters, the default value is *false*. Otherwise, the default value is *true*. An intent filter specifies the types of implicit intents that an activity, service, or broadcast receiver can respond to. However, explicit intents can always target at a specific component no matter how the intent filters of that component are set. Therefore, intent filters cannot be relied on for security.

4.2.2 The Checking Policy

The checking system parses the *AndroidManifest.xml* to find out if:

1. The application uses at least one permission and;
2. There exists an activity or service component that does not require any permission and is publicly visible

If the above two are both satisfied, components satisfying the second requirement are components that have the potential of capability leak.

We focus on activity and service components as they are the components that can be directly leveraged to launch a privilege escalation attack among the four types of components. The activity component provides a user interface and is what the user will see on the screen. On the other hand, the service component performs long-running operation in the background and does not provide a user interface. Both kinds of components can be protected by permissions.

4.2.3 The Checking Process

The system starts with scanning the *uses-permission* tags to see if there is any permission required by the application. If no, the system terminates. Next, the system parses the *android:permission* attribute of the *application* tag to find out if the application is guarded by any permissions. If yes, the system concludes that the application is safe. Otherwise, it goes on to perform the following checking for each activity or service component.

Figure 5 shows the decision tree for the checking. First, the system checks if there is any permissions declared in the *android:permission* attribute of the tag that declares that component. If yes, the component is safe as it is protected by permission(s). If no, the system checks if the *android:exported* attribute is present. If yes and the value of it is *true*, the component is potentially vulnerable for capability leak and requires further checking. If the *android:exported* attribute is absent, the visibility of the component hinges on the presence of intent filters. If there is no intent filters, the component is not visible to other applications by default, and hence, it is safe. Otherwise, the component is visible to other applications. In that case, it is also potentially vulnerable for capability leak and requires further checking.

Figure 6: A Deputy Application

4.3 Capability Leak Detection

We build the capability leak detection module based on existing techniques. We look for data paths that lead to capability leaks by following taint propagation [21, 29, 37, 15, 24] in the interprocudural control flow graph [27, 8, 42, 23, 34] built from the source code of an application component. The capability leak detection is performed for each risky component reported by the manifest file parsing module. We will first talk about the nature of the API calls of Android. Then, we will discuss the two kinds of data paths that can be found in a deputy. After that, we will introduce the details of the mechanism we use to search for such data paths.

4.3.1 API Calls of Android

We focus on API calls that are protected by permissions at the dangerous level. They are the API calls that give an application access to private user data or control over the device that can negatively impact the user. Permissions at the dangerous level are not granted to applications automatically even if they request them.

We classify API calls into action calls and data calls. Action calls are API calls that have side effects. Data calls are API calls that will return data to the caller without side effects. For example, the *sendSMS* is an action call that will send an SMS when called but will not return any data. The *managedQuery* is a data call that is used to retrieve records in content providers.

4.3.2 The Two Kinds of Data Paths in a Deputy

Figure 6 shows the bird's-eye view of an application that is acting as a deputy for other applications. Arrow A represents an intent message from another application. Upon receiving the intent message, the deputy invokes an API call represented by arrow B. The API call can be an action call or a data call as explained in previous section. In the case of an action call, the job of the deputy is done. In the case of an data call, the deputy will get the return value of the API call as represented by arrow C. The return value will then be passed back to the application who sent the intent. This is represented by arrow D.

Figure 7a shows an example of a deputy invoking an action call. The *getExtras* method retrieves data from the intent message that triggers this component. At line 20 and line 21, this method is called and the return value is stored in variable *receiver* and *msg*. At line 23, these two variables become the arguments to the *sendTextMessage* method which will send an SMS message to the telephone number stored in *receiver* with the content stored in *msg*. Therefore, even though the application that triggers this component does not have the capability to send SMS, it acquires such capability through this component. It can send an SMS message to whoever it wants with any contents. Therefore, the capability of this component is leaked.

Figure 7b shows an example of a deputy invoking a data

```
20  String receiver = getIntent().getExtras()
      .getString("receiver");
21  String msg = getIntent().getExtras().getString("message");
22  SmsManager sm = SmsManager.getDefault();
23  sm.sendTextMessage(receiver, null, msg, null, null);
```

(a) Deputy for an Action Call

```
30  Location loc =
     getLastKnownLocation(LocationManager.GPS_PROVIDER);
31  String long = loc.getLongitude();
32  String alt = loc.getAltitude();
33  Intent resultIntent = new Intent(null);
34  resultIntent.putExtra("Longitude", long);
35  resultIntent.putExtra("Altitude", alt);
36  setResult(Activity.RESULT_OK, resultIntent);
37  finish();
```

(b) Deputy for a Data Call

Figure 7: Code Example of a Deputy

call. At line 30, the method call *getLastKnownLocation* returns the last known location and the location is stored in the variable *loc*. Eventually, the location is passed to the intent object *resultIntent* which will be returned to the application that started this component. As a result, even that application does not have the capability to get the location information, it can obtain such information from this component. In other words, the capability of getting location information is leaked from this component.

DroidChecker searches for two kinds of data paths corresponding to these two kinds of API calls. The first kind of data paths begin with getting the content of the intent from the sender and end in using it as the input argument(s) to an action call. This corresponds to arrow A followed by arrow B in figure 6. We do not consider dangerous action calls with arguments not coming from the sender of the intent as we consider such applications not working as deputies for other applications. The second kind of data paths begin with getting the return value from a data call and end in passing back the return value to the sender of the intent. This is represented by arrow C followed by arrow D in figure 6. Since application developers can perform dynamic checking on the permissions owned by the sender of an intent message, we consider such checking as a way to prevent capability leak. As a result, a data path is considered safe if such checking is found in the control flow along with the data path.

4.3.3 Path Searching

Each method in the component is represented by a control flow graph (CFG). A node of a control flow graph represents a basic block of code and an edge represents a jump in the control flow. Figure 8 shows two example CFGs which correspond to two methods. We will explain the figure in greater detail later. The system starts with default entry points of the component. For activity components, they are *onCreate*, *onStart*, *onResume*, *onPause*, *onStop*, and *onDestroy* methods. For service components, they are *onCreate*, *onStartCommand*, *onBind*, *onUnbind*, and *onDestroy* methods. There exist some other entry points such as event han-

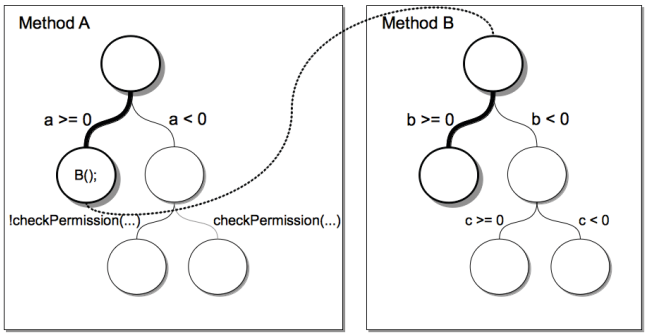


Figure 8: Example CFGs

dlers. Such entry points are also covered in the checking after the default entry methods are checked. After locating the entry methods, the system starts traversing their CFGs.

At anytime, there are two lists of tainted variables. The first list contains variables that are derived from the content of an incoming intent message. The second list contains variables that are derived from the return value of a data call. The system looks for API calls that get content from an intent (e.g. *getExtras*). The return values of such API calls are put into the first list. The system also looks for API calls that are data calls. Their return values are put into the second list. Whenever it comes across an assignment statement or a variable declaration, it checks if any variables on the right-hand side are in the two lists of tainted variables. Any taints of the variables on the right-hand side will be propagated to the variable on the left-hand side.

The are two kinds of sinks for the two kinds of tainted variables. An action call is the sink for those on the first list of tainted variables. The API calls used to attach data to an intent (e.g. *putExtra*) are the sinks for those on the second list of tainted variables. Whenever the system comes across a sink, it checks the two lists of tainted variables to see if the input argument(s) are tainted. If the nature of the taint matches that of the sink, the system raises an alarm for the risky data path found.

To avoid capability leak, application developers can check if the sender of the intent possesses the required permission before invoking an API call. This is done by invoking the *checkPermission* method which checks whether a particular package has been granted a particular permission. If such a checking is found in the control flow, the traversal for graph after the checking is skipped. For example, on Figure 8, the two nodes at the bottom of the CFG of method A are skipped as they are after a conditional check of the return value of the call *checkPermission*.

Most of the time, a method will call another method and the source and sink of a data path may reside in different methods. Therefore, when coming across a method call, the definition of the method is located and the checking moves to that method. Take the path searching shown on Figure 8 as an example. The path searching involves two methods. It starts at method A on the left. It first comes across a conditional check on the value of variable *a*. It then checks the branch where *a* is greater than or equal to 0. In that branch, there is a invocation of method B. The checking then goes to the CFG of method B. However, to avoid running into infinite loop in the case of a recursive method call, we maintain a list of checked methods and checking is skipped if that method has already been checked. A method is identified by its name, the class path of the class in which it is defined, the number of parameters, and the parameters that are tainted. For example, a method has two parameters and the first time when it is checked, the first parameter is tainted but the second is not. Next time, when it is invoked again but with the second parameter being tainted, it needs to be checked again. Therefore, a method with n parameters will be checked at most n times.

5. EVALUATION

5.1 Implementation and Environmental Setup

We implemented a prototype of DroidChecker. The entire system consists of 1013 lines of Java code. Since Java decompiler does not have a command-line interface, we automated the decompiling process by using Sikuli, a visual technology to automate and test graphical user interfaces using images [5]. We made use of ANTLR v3 to build a parser that could generate an abstract syntax tree from Java code [33]. The entire system runs under Ubuntu 11.04.

5.2 Experiment Results

We downloaded 1179 Android applications from Android Freeware[1], an Internet community aimed at collecting and categorizing truly free software, to test our prototype system. Running on a computer with Intel Core2Duo 2.66GHz and 3GB of ram, DroidChecker took about an hour to finish checking all the applications. 711 applications was found potentially risky by the manifest file parsing module. These applications then went through the capability leak detection module. 23 applications were caught for having capability leak problems. We manually checked these 23 applications to find out the data paths that lead to capability leaks. Table 1 summarizes the investigation results of them. For the 8 applications that have real capability leak problems, we confirmed all the data paths found by DroidChecker in their source code and identified capability leaks. However, we found that 2 of them were actually designed that way and their capability leaks are not harmful to the user. For the 15 false alarms, 4 of them are due to nonexistent paths that DroidChecker mistakenly considered. This is caused by reverse engineering problem of the applications. 8 of the false alarms are due to data passed between local application components that DroidChecker mistakenly thought that they were between components from different applications. For the remaining 3 applications, the first one writes data from another application to its logging file. The second one gets data from the caller and prints that to the standard output. This is not harmful but since the *print* method is sensitive, it was caught by DroidChecker. The third one stores some location data in an array and the whole array was tainted by DroidChecker. Eventually, one of the elements in the array is passed to another application. Although DroidChecker was not sure if that element is storing sensitive data, it still raised an alarm under such circumstances to make sure that it did not miss it. This is a generally accepted weakness of static taint checking techniques.

To show how our scheme improves the granularity of the checking, we repeated the experiment but with the second

[1] `http://www.freewarelovers.com/android`

Total no. of Apps	True alarms		False alarms		
1179	Not intended	Intended	Non-existent path	Leaked to local components	Other reasons
	6	2	4	8	3

Table 1: Experiment results

```
27  protected void onCreate(Bundle paramBundle)
28  {
39    Uri localUri = Contacts.ContactMethods.CONTENT_EMAIL_URI;
40    Cursor localCursor1 = localContentResolver.query(localUri,
        arrayOfString1, null, null, "name ASC");
41    mCursor = localCursor1;
58  }
66  protected void onListItemClick(ListView paramListView,
      View paramView, int paramInt, long paramLong)
67  {
68      boolean bool = mCursor.moveToPosition(paramInt);
75      String str = mCursor.getString(i);
76      Intent localIntent3 = localIntent2.putExtra("email", str);
77      setResult(-1, localIntent2);
78      finish();
80  }
```

Figure 9: Excerpt of source code of Adobe Photoshop Express

```
11  public void onCreate(Bundle savedInstanceState) {
23    Intent intent = new Intent();
24    intent.setClassName("com.adobe.psmobile",
        "com.adobe.psmobile.ContactEmailList");
25    intent.setAction("android.intent.action.PICK");
26    intent.addCategory("android.intent.category.DEFAULT");
27    intent.setDataAndType(null, "vnd.android.cursor.dir/email");
28    startActivityForResult(intent, 1);
29  }
31  protected void onActivityResult(int requestCode,
      int resultCode, Intent data) {
34    TextView t = (TextView)findViewById(R.id.textview);
35    t.setText("E-mail address obtained:\n" +
        data.getExtras().getString("email"));
37  }
```

Figure 10: Excerpt of source code of our example attack application

module disabled. This means it scanned the manifest file only and did not perform the interprocedural control flow graph searching and taint checking. We found that module one alone reported 852 applications as vulnerable. In contrast, only 23 alarms were raised when coupled with module 2. This shows that module 2 eliminated 829 false alarms which accounts for 97.3% of the total number of alarms raised.

6. SAMPLE ATTACK

In this section, we demonstrate a sample attack using a tiny Android application developed by us. The application does not have any capability on its own. However, exploiting the capability leak we found on *Adobe Photoshop Express 1.3.1* (APE), it is able to retrieve e-mail addresses of the contacts on the phone. In the rest of this section, we first illustrate the capability leak using the source code we obtained by reverse engineering APE. After that, we show how we can leverage that to obtain extra capability.

6.1 Capability Leak on APE

Figure 9 shows an excerpt from the source code of a component in APE. We extracted the minimum number of lines of code to save space. When the component is started, the *onCreate* function is first executed. At line 40, a query is executed to get the e-mails from the contacts. The *mCursor* reference variable now points to a Cursor object which is positioned before the first entry of the result set. At this point, the program shows the contact e-mails and waits for the user to select an e-mail on the list shown. After a contact e-mail is selected, the *onListItemClick* method is executed. At line 68, the *mCursor* is moved to the entry in the result set representing the contact selected. At line 75, the e-mail is stored in variable *str*. At line 76, the e-mail is put into the intent object *localIntent3* which is then returned to the application starting this component. Since this component is publicly accessible by other applications, by leveraging it, any applications can acquire access to e-mail addresses of the contacts even though they do not have that capability. Next, we will illustrate such an exploit using a simple Android application developed by us.

6.2 Exploiting the Capability Leak

Our attack application and APE are installed on an emulator running Android 2.3.3 on top of Android SDK Tools revision 11. First, the attack application is launched. After that, it starts the component in APE that has the capability leak. The user is then tricked to select a contact on the contact list. Next, the control is passed back to the attack application. The e-mail address selected is also passed back to the attack application. It can then send it back to a server owned by the attacker. Obtaining valid e-mail addresses is key to successful spamming. We have also tried our attack on a stock Nexus S Android phone running Android 2.3.4 and it worked.

Figure 10 shows an excerpt of the source code of our example attack application. When the application is launched, the method *onCreate* is executed. From line 23 to line 28, an explicit intent message is sent to start the component in APE. After that component has finished, the callback function *onActivityResult* is executed. It then gets the e-mail address from the data returned by APE and shows it on the screen. Note that the attack application does not have any capability originally. This shows that our application has acquired extra capability by exploiting the capability leak in APE.

7. DISCUSSION

7.1 Limitations

Since DroidChecker performs the checking on decompiled source code, it depends on the completeness of the decompilation tool. Designing decompilers for Android applications is an active research area. Dedexer is a disassembler tool for DEX files and decompiles Android applications into an "assembly-like format" [32]. The ded decompiler can decompile Android applications into Java source code [12]. Their work is orthogonal to ours and a better decompiler can help us get a more accurate picture of the Android applications being checked.

Since we use static taint analysis as the underlining technique, we also suffer from the inherent limitations of it. One of them is that it is not able to take into account dynamic features of the Java language like polymorphism. A reference variable may point to an object that inherits it. When a method of that object is invoked, our anaylsis tool will check the method in the superclass but not the one in the subclass. This will harm the soundness and completeness of the checking.

As the *permission* attribute of a component only allows one permission to be set, an application may still be risky even if it has set the *permission* attribute. However, we cannot overcome this limitation as it stems from the design of the Android system. Besides, we currently skip the path searching for code blocks after a conditional check of the return value of a call to *checkPermission*. This is based on the assumption that the presence of *checkPermission* means the developer effectively checks that the caller application has the required permissions for subsequent API calls. However, this may not be the case if the *checkPermission* is not used properly. However, we cannot take into account how it is used as that depends much on the dynamic behavior of the application. It is a balance between false negative and false positive. The assumption we made here is to make the checking more sound. However, it is true that it also makes the checking less complete. We make this as a parameter in our prototype implementation to let the user decide whether the assumption should be made or not.

7.2 Security Guidelines

To make sure that applications do not leak their capabilities to other applications, developers should avoid making the components of their applications accessible by components in other applications. This can be done by either not declaring any intent filters or setting the *android:exported* attributes of components to *false*. In case a component has to be made public, it should be protected by a permission such that other applications have to be privileged in order to access it. Besides, developers should be aware of the explicit intent messages sent to their applications. They should not rely on intent filters to protect a component since other applications can always send an explicit intent message to it. Therefore, a publicly accessible component should not perform security sensitive operations upon receiving an explicit intent message. Before invoking dangerous API calls upon receiving an intent message, developers should check if the sender possesses the required permissions using the *checkPermission()* system call.

7.3 Future Work

Some future work remains. Capability leaks can happen indirectly through content providers. A possible scenario is that an application, which has the required permissions, retrieves private data and stores them in a content provider of its own. Due to certain reasons, it grant the read permission of that content provider to other applications using the URI permissions mechanism. Leakages of this type will not be revealed by our checking tool since it is not able to determine whether a content provider contains private data or not. As a preliminary idea, we can address this by tainting a content provider whenever private data is written into it. Another direction is to make the system run on stock Android phones such that users can check an application for vulnerabilities before installing it.

8. RELATED WORK

The privilege escalation attack on Android was first proposed by Davi et al. [7] in which they demonstrated an example of the attack. They showed that a genuine application exploited at runtime or a malicious application can escalate granted permissions. However, they did not suggest any defense for the attack in the paper.

Some recent work tried to proposal security extensions for Android to remedy the attack. XMandDroid [3] prohibits IPC between applications which own certain combinations of capabilities. For example, an application which can retrieve location information is not allowed to communicate with an application which can access the internet. Permission re-delegation, which is the same as privilege escalation, for web applications and Android applications was studied by Felt et al. in [16]. They proposed IPC Inspector to defend against permission re-delegation. IPC Inspector reduces the permission set of an application after it receives messages from another application which does not own certain permissions the receiver owns. Their mechanism is considered restrictive as applications cannot deliberately receive messages from a less privileged application. Moreover, to avoid multiple reductions of the permission set of an application after communicating with several applications that do not have one or more permissions in its permission set, they create a new instance of an application whenever it encounters in IPC with a new application. This significantly drains the system resources and provides an attack surface for denial of service attack. They also carried out a study to search for vulnerabilities in current Android applications. However, they used call-graph analysis which is coarse-grained and likely to produce false positives for those applications which invoke dangerous calls not under the influence of the intent sender. At about the same time, another group of researchers proposed QUIRE to address the same problem [9]. QUIRE tracks the call chain of on device inter-component communication. When an application receives a message from a less privileged application, it extracts the information about the call chain from the caller. It relies on the application to pass information along the call chain. Provided that application developers are not security experts, it is an error-prone approach to rely on them to carefully pass the information to each receiver of the IPC calls invoked by the application. The advantage of this design is that it allows the developer to choose whether to reduce the set of privileges of the application to that of the sender or to

exercise its full privilege set by acting explicitly on its own behalf. This grant applications the freedom to communicate with less privileged applications without being taken away their own privileges. However, their mechanism requires existing applications to be re-compiled in order to enjoy the security service provided. While these work serve the purpose of introducing security measures into the Android system to prevent the attack, our work aims to provide users and developers with a tool to search for vulnerabilities in applications and let them take actions before these security measures are adopted.

There are some work in the literature that use static or dynamic approach to check Android applications for vulnerabilities or malicious behaviors. Chan et al. proposed a static analysis tool of Android applications to determine whether an application can be leveraged to launch privilege escalation attacks [4]. However, their tool looks into the manifest file only. Hence, their tool is more coarse-grained and produces more false positives. Enck et al. proposed a static analysis tool to evaluate the security of Android applications [12]. Their tool makes use of control flow analysis, data flow analysis, structural analysis, and semantic analysis to dangerous functionality and vulnerabilities. Their analysis focuses on a single Android application rather than the interaction between an Android application and other applications. Their work is orthogonal to ours. ComDroid [6] focuses on sniffing, modification, stealing, replacing, and forgery of messages passing between applications. It performs flow-sensitive, intraprocedural static analysis on disassembled assembly-like source code to search for vulnerabilities. Their approach can also be applied to search for capability leaks in applications but would be more coarse-grained and produce more false positives as it looks into the manifest file only. TaintDroid [11] makes use of dynamic taint tracking to keep track of the flow of privacy sensitive data through third-party applications. It monitors in real-time how applications access and manipulate users' personal data. It helps detect when sensitive data leaves the system via untrusted applications. Evaluation of it showed that it incurs significant performance overhead. ScanDroid [1] uses modular data flow analysis to look for data flows in Android applications and make security-relevant decisions automatically based on such flows.

There were some work on security extensions to Android security architecture. Saint [31] is a modification of Android to enable application providers to express the application security polices that regulate the interactions among them. It allows an application to control which applications can be granted the permissions it declares. Moreover, when an application needs to access an component of another application, both parties can assert controls of the communication between them through defining run-time interaction policies. In particular, the caller application selects which application's interfaces it uses and the callee application controls how its interface is used by other applications. Saint policy provides certain protection against privilege escalation attacks as the application can control which applications can access it. However, Saint assumes that access to components is implicitly allowed if no Saint policy exists. This put the burden of enforcing security to application developers which is error prone as most of them are not security experts. Kirin [13] is an application certification service to mitigate malware at installation time. It uses existing security requirements engineering techniques as a reference to identify dangerous application configurations in Android. The rules are a set of combinations of permissions that an application must not be granted at the same time. For example, an application being granted permissions to record audio and access location information may be an voice and location eavesdropping malware. Similar to our approach, their certification process relies on the manifest file in the APK of the application. However, their approach cannot identify applications that can be leveraged to launch privilege escalation attack. Instead, their work is orthogonal to our work and is targeting at a different kind of attack vector.

There are some other work on security of the Android system. Schmidt et al. [35] walked through the smartphone malware evolution. They provided possible techniques for creating Android malware(s). Their approach involves usage of undocumented Android functions enabling them to execute native Linux application even on retail Android devices. They also showed that it is possible to bypass the Android permission system by using native Linux applications. Enck et al. [14] gives a description of the security model of the Android system. Jakobsson et al. [22] proposed a software-based attestation approach to detect any malware that executes or is activated by interrupts. Based on memory-printingof client devices, it makes it impossible for malware to hide inRAM without being detected. Nauman et. al. [28] improved the installation process of Android applications to allow user to selectively grant permissions to applications and impose constraints on the usage of resources. Shabtai et al. [36] makes use of Security-Enhanced Linux (SELinux) to help reduce potential damage on the Android system from a successful attack.

9. CONCLUSION

Android is an open system with some state-of-the-art security measures. Although the idea of privilege-separating the applications can avoid them adversely impacting other applications or the system, it does not prevent applications from leaking capabilities. In this research, we proposed a methodology for detecting capability leaks in Android applications. We utilized interprocedural control flow graph searching and static taint checking technique to identify data paths that will lead to capability leaks. A prototype system was implemented and tested with more than 1000 applications. 6 of applications were found to have capability leaks. We developed a tiny Android application to demonstrate a sample attack. Using one of these 6 applications, our application successfully obtained the extra capability of retrieving e-mail addresses from the contacts on the phone. This showes that our proposed method is effective in identifying the potential risk. As a side product, some security guidelines were provided for Android application developers to avoid capability leaks in their applications.

Acknowledgment

The work described in this paper was partially supported by the General Research Fund from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. RGC GRF HKU 713009E), the NSFC/RGC Joint Research Scheme (Project No. N_HKU 722/09), HKU Seed Fundings for Applied Research 201102160014, and HKU Seed Fundings for Basic Research 201011159162 and

200911159149. The authors would like to thank Echo Zhang and anonymous reviewers for their insightful comments.

10. REFERENCES

[1] J. S. F. Adam P. Fuchs, Avik Chaudhuri. Scandroid: Automated security certification of android applications. Technical report, University of Maryland, College Park, 2009.

[2] Android Open Source project. Security and permissions. `http://developer.android.com/guide/topics/security/security.html`, April 2011.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr 2011.

[4] P. P. Chan, L. C. Hui, and S. Yiu. A privilege escalation vulnerability checking system for android applications. In *13th IEEE International Conference on Communication Techonologies (ICCT)*, 2011.

[5] S. T.-H. Chang and T. Yeh. Sikuli. `http://sikuli.org/`.

[6] 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.

[7] 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, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] S. K. Debray and T. A. Proebsting. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Trans. Program. Lang. Syst.*, 19:568–585, July 1997.

[9] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[10] E. Dupuy. Java decompiler. `http://java.decompiler.free.fr/`, Aug 2010.

[11] W. Enck, P. Gilbert, B.-G. 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[12] 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.

[13] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[14] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1):50 –57, jan.-feb. 2009.

[15] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19:2002, 2002.

[16] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[17] Gartner. Gartner says worldwide mobile device sales to end users reached 1.6 billion units in 2010; smartphone sales grew 72 percent in 2010. `http://www.gartner.com/it/page.jsp?id=1543014`, February 2011.

[18] Google. Axmlprinter2. `http://code.google.com/p/android4me/`, October 2008.

[19] Google. Android adk. `http://developer.android.com/guide/topics/usb/adk.html`, December 2011.

[20] Google. dex2jar. `http://code.google.com/p/dex2jar/`, June 2011.

[21] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[22] M. Jakobsson and K.-A. Johansson. Retroactive detection of malware with applications to mobile platforms. In *Proceedings of the 5th USENIX conference on Hot topics in security*, HotSec'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[23] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 89 –103, 1999.

[24] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[25] Lookout. App genome report. `https://www.mylookout.com/appgenome`, February 2011.

[26] Lookout. Security alert: Android trojan ggtracker charges premium rate sms messages. `http://blog.mylookout.com/2011/06/security-alert-android-trojan-ggtracker-charges-victims-premium-rate-sms-messages/`, June 2011.

[27] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 287–298, New York, NY, USA, 2009. ACM.

[28] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[29] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[30] Nielsen. Who is winning the u.s. smartphone battle? `http://blog.nielsen.com/nielsenwire/online_mobile/who-is-winning-the-u-s-smartphone-battle/`, March 2011.

[31] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel. Semantically rich application-centric security in android. In *In ACSAC '09: Annual Computer Security Applications Conference*, 2009.

[32] G. Paller. Dedexer. `http://dedexer.sourceforge.net/`, August 2009.

[33] T. Parr. Antlr. `http://www.antlr.org/`.

[34] M. Pistoia, R. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *In Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 362–386. SpringerVerlag, 2005.

[35] A.-D. Schmidt, H.-G. Schmidt, L. Batyuk, J. Clausen, S. Camtepe, S. Albayrak, and C. Yildizli. Smartphone malware evolution revisited: Android next target? In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 1 –7, oct. 2009.

[36] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *Security Privacy, IEEE*, 8(3):36 –44, may-june 2010.

[37] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[38] Symantec. Android.ggtracker. `http://www.symantec.com/security_response/writeup.jsp?docid=2011-062208-5013-99`, June 2011.

[39] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Proceedings of the 17th conference on Security symposium*, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[40] thinkmobile with Google. The mobile movement study. `http://www.gstatic.com/ads/research/en/2011_TheMobileMovement.pdf`, April 2011.

[41] D. Venkatesan. A trojan spying on your conversations. `http://totaldefense.com/securityblog/2011/08/26/A-Trojan-spying-on-your-conversations.aspx`, August 2011.

[42] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM.