Noname manuscript No.
(will be inserted by the editor)

Towards a More Dependable Hybrid Analysis of Android Malware Using Aspect-Oriented Programming

Received: date / Accepted: date

Abstract The growing threat to user privacy by Android applications (app) has tremendously increased the need for more reliable and accessible analysis techniques. This paper presents *AspectDroid*[1] – an app-level hybrid analysis system designed to investigate Android applications for possible unwanted activities. It leverages static bytecode instrumentation to weave in analysis routines into an existing application to provide efficient data flow analysis, detection of resource abuse and analytics of suspicious behaviors, which are then monitored dynamically at runtime. Unlike operating system or framework dependent approaches, *AspectDroid* does not require porting from one version of Android to another, nor does it depend on a particular Android runtime, making it a more adaptable and easier to use technique. We evaluate the strength of our data flow algorithm on 105 apps from the DroidBench corpus, with experimental results demonstrating that *AspectDroid* can detect tagged data with 95.29% accuracy. Furthermore, we compare and contrast the behavioral patterns in 100 malware samples from the Drebin dataset [7] and 100 apps downloaded from Google Play. Our results showed more traces of sensitive data exfiltration, abuse of resources, as well as suspicious use of programming concepts like reflection, native code and dynamic classes in the malware set than the Google Play apps. In terms of runtime overhead, our experiments indicate *AspectDroid* can comprehensively log relevant security concerns with an approximate overhead of 1MB memory and a 5.9% average increase in CPU usage.

Address(es) of author(s) should be given

[1] A poster version of this paper appears in CODASPY 2016 [4].

1 Introduction

Android malware and over-privileged applications are well-known for privacy violation and data leakage [22]. For instance, they transfer personal data outside the devices of end-users without their consent. In a report published by GDATA [30], the Android platform is estimated to account for 97% of all malware on mobile devices in 2014. Over 2 million trojan applications have been detected in 2015, representing a 50% increase from 2014. Modern malware is in use on an industrial scale by crime organizations and its development is often highly professional. In another report, Andrubis [33] performed an analysis on over a million (malicious and benign) apps, and found that 38.79% of the apps have data leakage. The percentage further increases from 13.45% in 2010 to 49.78% in 2014, and is also noted by Yajin *et al.* [38]. In many respects, this presents an even greater threat to users than before, as mobiles are entrusted with the most private of information and mobile malware can very effectively spy on users in real time. Overall, the security and privacy concerns surrounding these revelations increases the need for reliable and accessible app analysis systems.

Traditionally, Android apps are analyzed using either static or dynamic approaches. Static analysis involves the use of predetermined signatures and/or other semantic artifacts such as API calls, strings etc. Enck et al. developed Kirin [18] which evaluates privacy risks based on the set of permissions requested, while [20, 39] analyzed Android applications by evaluating fine-grained API calls in addition to the permissions set. Other semantic-based analysis tools [21,34] examine components and intents in addition to the permissions and API calls made within the application binary.

Dynamic analysis on the other hand executes a target application in a contained environment [37,36,6,9, 10,19,25,17,16,29]. In general, static analysis has the advantage of high performance and coverage. Conversely, simple obfuscation can hinder the extraction of important data such as API names. Dynamic analysis on the other hand provides a better view of an app's behavior, although it is usually limited in scope to observed execution paths.

Most comprehensive dynamic analysis techniques either require instrumentation of the underlaying operating system code [17,29,16] or involve virtual machine introspection [35]. They provide effective sandboxing for the analysis of the target applications, but unfortunately, such techniques are heavily dependent on OS versions and the Android runtime. Porting and flashing a new build on real devices for various versions of Android is not an easy task, which can limit the number and kind of applications that can be analyzed. Existing application-level techniques like [6,9,10,19,25] are mostly constrained to performing only API monitoring. Although systems like Capper[37,36] can perform app-level taint analysis, their heavy reliance on static analysis for the extraction of taint slices makes it equally vulnerable to simple obfuscation.

In this paper, we present *AspectDroid*, a hybrid analysis system for Android applications based on the AspectJ instrumentation framework. *AspectDroid* performs static bytecode instrumentation at the application level, and does not require any particular support from the operating system or the Dalvik virtual machine. It weaves in monitoring code at compile time using a set of predefined security concerns, such as data/resource abuse and other non-traditional behaviors like reflective calls and native code execution. The target application is then executed on any Android platform of choice for which behavioral patterns are monitored and logged dynamically.

In summary, *AspectDroid* is a new hybrid analysis system for Android applications that has the following salient features:

Android Platform Independent: *AspectDroid* does not introduce code at the operating system level. Instrumented applications can run without any restrictions on both emulators and original Android devices.

Adaptable to all Android Runtimes: *AspectDroid* is not restricted to the Dalvik virtual machine or Android Runtime (ART).

Explicit Data Exfiltration: *AspectDroid* uses an efficient algorithm to track data propagation dynamically from source to sink.

Behavioral Tracing: We monitor applications for possible unwanted activities like telephony abuse, use of reflection, dynamic classes and native code execution.

To determine the effectiveness and efficiency of *AspectDroid*, we carry out two different tests. In the first experiment, we analyze 105 Android apps from the DroidBench project for possible data exfiltration. The results show that the *AspectDroid* data flow algorithm can accurately follow the propagation of target data from source to sink with 95.29% F-score accuracy. The second experiment analyzes the dynamic behavior of 100 malware samples from Drebin's dataset [7] and 100 apps downloaded from Google Play. Our findings are itemized based on data exfiltration, use of reflection, dynamic class loading, native code, and telephony abuse. The results of our analysis indicate that while phone-related data like IMEI are equally exfiltrated in both malware and the Google Play apps, that's not the case for user-related data like Contacts where leak traces were more common in malware samples than the Google Play apps. Five malware families use reflection for malicious purposes, such as invoking the methods of a background service to spoof user accounts and passwords. On the other hand, all the reflective call invocations in the Google Play samples did not result in any sensitive API call. Furthermore, we have seen more telephony abuse in malware than the Google Play apps, e.g., SMS was sent to all contacts on the phone without the user's consent. Nine families invoke native processes about 72 times, as compared to 6 for the Google Play apps.

We further use the malware dataset to measure the instrumentation overhead for dynamic execution. The results show that *AspectDroid* has limited memory overhead of around 1MB and a reasonable 5.9% average CPU usage overhead.

The rest of the paper is organized as follows: Section 2 presents background on the AspectJ instrumentation framework; Section 3 provides an overview of *AspectDroid*'s design and associated algorithms; Section 4 presents the implementation of *AspectDroid*; Section 5 contains testing and evaluation of results; Section 6 enumerates some challenges and discusses limitations and future work; Section 7 reviews the related literature followed by section 8 that concludes the paper.

2 Background

Instrumentation is the process of analyzing programs by adding trace code to their source code, binary code, or execution environment. This provides mechanisms for an analyst to define concerns related to program verification, enforcement, monitoring, error-checking, performance, debugging, or tracing. Instrumentation tech-

niques do not necessarily modify code but rather tamper with the execution or behavior based on defined constructs. In recent years, instrumentation techniques have gained momentum in the cybersecurity community for vulnerability [36], malware [6,16,17,19,25,29] and privacy analysis [9,10,24,37].

Aspect oriented programming (AOP), first introduced by [27], is a modularized programming model allowing the separation of cross cutting concerns [31], which are difficult to capture in traditional programming models. AOP encapsulates the concerns, defined as `aspects`, by instrumenting extra behavior in the existing code. These aspects are special constructs forming the building blocks of AOP. Their designs can be generic, which allows for reuse throughout program execution. Implementation of AOP can be performed in two distinct ways:

1. Static instrumentation allows for code to be injected at compile time. This technique merges both the aspects and the original code into one binary, which then executes in the execution environment of the original code.
2. Dynamic instrumentation, on the other hand, injects code at runtime. In most instances, this requires a custom classloader to enable the interpreter to understand and implement the AOP features.

In 2001 [26], PARC developed an extension for AOP designed specifically for the Java programming language, called AspectJ. Its Java-like syntax, coupled with its ease of use, makes AspectJ a very popular instrumentation tool for Java programs. Aspects in AOP are defined by some key terms:

1. Pointcuts are defined by `kinded` constructs such as function `call`, method `execution`, `within` class, `cflow` etc., which match some specified `signatures` or `modifiers`.
2. Signatures are semantic definitions which can be decoded by the AspectJ compiler during Play creation. It can encapsulate both broad and narrow definitions, giving an analyst ample flexibility.
3. Joinpoints are points within the execution of the program that are interesting and/or defined by the concerns of an analyst. These are chosen based on constructs defined in a pointcut.
4. Advice is the piece of code that gets executed when a certain Play is reached during program execution

In addressing security concerns, advice defined for a Play adds some functionalities such as logging, code injection, value manipulations, execution rerouting, skipping execution, etc. to an instrumented program. Advice to be executed can target `before, after` and `around` the execution of a particular Play. As the name implies, execution of `before` advice precedes the execution of the target Play. In this advice, parameters and the target object can be retrieved, in addition to signatures, source location, etc. For `after` advice, in addition to the information extracted in before advice, the return value can also be retrieved and evaluated. The most interesting is `around` advice which, although potentially expensive to use, allows code injection and modification of arguments, variable values, and return values.

The code snippet in Listing 1 shows a sample aspect that defines a pointcut which picks a Play at the call to `getDeviceId`. When instrumented, this aspect will pick the method `getDeviceId` from the class `TelephonyManager`. The Play is picked because the signature matches the method in that class and it is the only class in the Android SDK with such a method. However, if within the application there also exists a library class with such a method, our broad signature will automatically capture such a Play, too.

Listing 1: Simple Aspect

```
public aspect Logger{
  pointcut myId():
     call(* *..*.getDeviceId());
  after()returning(String id): myId(){
     log.v("AspectJ", "DeviceId="+id);
  }
}
```

2.1 Bytecode Weaving

In the Java compilation process, an intermediate representation called `bytecode` is generated when the original source code is compiled. This bytecode is contained in .class files representing each source class. More specific to Android, the system has added another level of abstraction to its compilation process, where the class files are further compressed into one dex file. Bytecode weavers are tasked with weaving together class files (both Java classes and aspect classes). In this paper, our chosen bytecode weaver is AspectJ [27]. Its robust framework allows us to define and inject security concerns related to Android apps for the purpose of logging and monitoring. More so, its programming syntax and semantics is identical to that of the Java language, allowing us to tie and weave the monitoring code into a target Android application with better precision.

AspectJ compilers (ajc) can accept both raw sources and class files for compile-time weaving and thus have the capability to compile and weave the aspect/Java sources and/or class files to produce a new woven class. The resulting merged Java bytecode has to be com-

patible with the execution platform's VM. However, in load-time weaving AspectJ exposes an interface that facilitates the weaving process between the target bytecode and a custom `classloader`.

3 System Design

AspectDroid is a hybrid system that uses static instrumentation to inject monitoring code into the target app based on some specific cross-cutting concerns. A requirements of our system involve analyzing unknown binaries where there is no available source code. The core of *AspectDroid* is built based on compile-time bytecode weaving. This form of static instrumentation takes the advice defined in an aspect and weaves them at specified joinpoints in a target class file. For Android apps, the resulting binary is `dexed` and re-packaged into a new apk. Since this new application does not need a custom `classloader`, it has the flexibility of executing on any device emulator without changes to the underlying OS.

With *AspectDroid*, the new injected code executes alongside the original code and performs custom logging and other analytical functions. The instrumentation engine (IE) which is the primary component, forms the backbone of *AspectDroid* and is designed to address three objectives:

1. Dataflow Analysis
2. Resource Abuse Tracing
3. Analytics of Suspicious Behavior

Our instrumentation code is encapsulated in an aspect and is tailored for each of the objectives mentioned above. The aspects are weaved into the target app using AspectJ's `ajc` compiler, producing the instrumented version used to perform the analysis. The instrumentation process is done in-vitro on a host machine. At successful re-compilation the target app is then pushed onto the test bed for dynamic execution.

3.1 Dataflow Analysis

AspectDroid performs application-level tainting of target data source(s). Our approach is built around the fact that standard Java and Android libraries use specific method naming conventions to express common types of operations. Thus, we utilize the consistent use of specific verbs, such as `read, open, write, put, connect`, and `execute`, to define broad signatures to capture actions such as file/stream/network access. More specific signatures, such as `getLongitude` are used to define narrower joinpoints. Based on all the signatures, we define pointcuts to select various `source, sink, propagation` joinpoints. With the help of AspectJ APIs, a joinpoint's data, such as the `target object`, `parameters`, `return values`, etc. (as shown in Figure 1) can be extracted at runtime. Java programming semantics categorize data types as primitive, object, and arrays. Although beyond the scope of this paper, it is important to note that the JVM stores and processes these data types very differently. Therefore, our data sources, propagation and sink for each data type are handled differently. To simplify the terminology, we refer to `primitive` data types, such as `string` and `character`, as `low-level data types`.

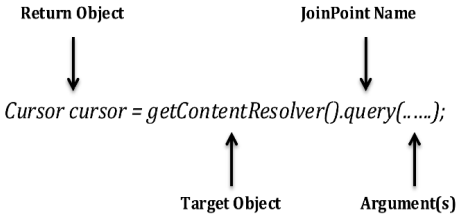

Fig. 1: Parts of a Method joinpoint

Our dataflow analysis is limited to explicit propagation, where the tainted data must be in the sink call, as shown in Listing 2. On the other hand, Listing 3 illustrates an example of an implicit flow which exfiltrates inferred data based on the real tainted data.

Listing 2: Explicit Data Flow

```
TelephonyManager telephonyManager = (TelephonyManager)
   getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = telephonyManager.getDeviceId();
if (!IMEI.equals("00000000"){
   String id =
        Base64.encodeToString(myString.getBytes(), 0);
   SmsManager sms = SmsManager.getDefault();
   sms.sendTextMessage("5556", null,id, null, null);
   }
```

Listing 3: Implicit Data Flow

```
TelephonyManager telephonyManager = (TelephonyManager)
   getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = telephonyManager.getDeviceId();
if (!IMEI.equals("00000000"){
   String val = "Device not emulator";
   SmsManager sms = SmsManager.getDefault();
   sms.sendTextMessage("5556", null,val, null, null);
   }
```

The AspectJ API used by our system is not designed to create joinpoints on conditional/branch instructions. Nonetheless, in the analysis of Android applications, sensitive data leaving the device is the real threat, not inferred data. As illustrated in Listing 2, the real device IMEI was exfiltrated compared to "Device not Emulator" in Listing 3.

3.1.1 Taint sources

We are interested in sources that are relevant to the privacy and security of the user. We define `vital` sources as phone-related data, content provider objects, file reads and user input. In Android, most important data are guarded by permissions and only accessible to the user through specialized Android API calls. Other relevant data not guarded by permission, such as data read from files and user input from text boxes, are also accessed via the standard Java/Android APIs. Specialized pointcuts are created using signatures to intercept these vital API calls. After execution, the return value is stored as a key in a `taint map` with a corresponding special tag for each unique source as the value. Depending on the return type, low-level data types are stored in raw form, while every other object is stored in hash form. This storage design is very significant in reducing the overhead associated with checking if tainted data is part of an object. It allows us to check if the object is tainted using its hashcode at propagation or sink joinpoints.

3.1.2 Taint sinks

Taint sinks are defined as points where the target application communicates with an external component, either within the device or the outside world. In our data flow analysis, we seek to monitor only those sinks that form a `possible` exfiltration point for the data sources defined above. The data sinks are broadly categorized as network, e.g., writing to a `Socket`, `URLConnection`, etc.; SMS sends; file writes (both ordinary files and shared preferences); and IPC. We use the same signature semantics to pick the sink joinpoints. We also leverage the `around advice` of such joinpoints to check if its arguments, or target, contain tainted data. WHY??This process is relatively straightforward for parameters; however, for targets, the object needs to be parsed and the associated fields checked against the keys in the taint map. Data exfiltration is detected if a tainted piece of data is found either within the sink joinpoint's parameters/parameter's fields or within the target/target's fields.

3.1.3 Taint Propagation

Knowing data sources and sinks alone cannot accurately determine data exfiltration; we also need to identify the data propagation process as represented by the sequence of variable assignments along the path from source to sink. The tainted data can be part of an object's field and the object can be manipulated in different ways. For every joinpoint, we check if it contains tainted data; if so, the appropriate propagation rule is picked based on the respective joinpoint's return type as enumerated in the 7 point rules below:

1. Rule 1: Joinpoint that returns a low-level data type and contains a tainted argument.
2. Rule 2: Joinpoint that returns a low-level data type and contains a tainted target.
3. Rule 3: Joinpoints that convert a tainted array to other data types.
4. Rule 4: Joinpoints that create an array from other tainted data types.
5. Rule 5: Object constructor joinpoint that contains a tainted argument.
6. Rule 6: All joinpoints with object return type that contains tainted arguments.
7. Rule 7: All joinpoints with object return type that contains tainted target.

For joinpoints targeting low-level data types, their return values and target object are the same. However, for object joinpoints, the target is always a reference to the location of the object in memory while the return type could be anything. For example, an object's joinpoint could return a `Boolean` indicating success of a method call, `void`, a low-level data type, or other objects. This distinction forms the basis of how our taint tag/map is updated after the execution of the joinpoint. Table 1 gives a taint propagation example for each of the flow rules, and shows the taint tag/map update after the joinpoint's execution. Propagation rule 7 can create a weaving process that might get out of hand, thus we included some optimizations for joinpoints associated with that rule, based on the object's class.

EXTREMELY hard to parse the following–rewrite!

To optimize the weaving process and reduce the complexity of the instrumentation, the propagation's joinpoints for every source are created on method calls that fall within the control flow path of the source call's enclosing method. For example, if the data source IMEI returned by `getDevideId` is found within the body of an Activity's `onCreate` method, then the propagation joinpoints will only be created for methods in that control flow that have satisfied the propagation rules. This optimization greatly enhances our weaving process and eliminates the need for redundant joinpoints.

3.2 Resource Abuse Tracing

Access to some vital functionalities such as Telephony (SMS and Calls) on the mobile devices are requested

Table 1: Flow rules examples for updating taint/tag map

Rules	joinpoint Example	Taint Data	Taint Tag/Map Update
Rule 1	Int myInt = System.identityHashCode(val)	valueOf(val), tag = DeviceID	valueOf(myInt), tag = DeviceID
Rule 2	String str1 = myInt.toString()	valueOf(myInt), tag = DeviceId	valueOf(str1), tag = DeviceID
Rule 3	char arr[] = str1.toCharArray()	valueOf(str1), tag = DeviceID	Hashcode(arr), tag = DeviceID Elements of arr, tag = DeviceID
Rule 4	Str str2=Arrays.toString(arr)	Hashcode(arr), tag = DeviceID	valueOf(str2), tag = DeviceID
Rule 5	StringBuilder stb = new StringBuilder(str2)	valueOf(str2), tag = DeviceID	Hashcode(stb), tag = DeviceID
Rule 6	stb.append(val2)	Hashcode(stb), tag = DeviceID valueOf(val2) tag = LineNum	HashCode(stb) tag = DeviceID and LineNum
Rule 7	Vector vec = new Vector() vec.add(str2)	New empty vector is created valueOf(str2), tag = DeviceID	Hashcode(vec), tag = DeviceID

through specialized API calls. According to a 2012 Trend Micro report [23], resource abuse is the most common category of Android malware. Thus it is imperative for an analysis system to trace and report such abuse. With *AspectDroid*, the system instruments the telephony methods invocations and have their target object, parameters and return value logged. This information is significant in determining the phone number used (premium service or device contact), the message content, settings and format.

3.3 Analytics of Suspicious Behaviors

Programming practices such as native call invocation, dynamic class loading, native code execution and reflective call invocation add flexibility to software development. Although these concepts may be benign, malware can often hide its behavior using these practices to hinder static analysis or for malicious purposes such as privilege escalation.

Reflection, for instance, allows method calls to be resolved dynamically at runtime. Malware can use this technique to hide calls to sensitive APIs. With *AspectDroid*, we instrument reflective calls and analyze the target object at runtime for possible tainted data sources, propagation, sinks or any sensitive API calls. We also check their parameter arrays for possible taint propagation.

Android apps can load extra classes at runtime using a dynamic class loading API. One of the drawbacks of static bytecode instrumentation (and by extension all static analysis) is that only available classes are processed at compile time; extra classes loaded at runtime are not affected by the weaving. To address this, *AspectDroid* implements `dynamic class instrumentation`: at the joinpoint where `Dexclassloader` loads the new dex file, the weaved advice captures the absolute path to the file, sends it to the host machine via an Asynchronous task, and waits for notification to proceed. On the host machine, *AspectDroid* has a server side component that receives the dynamic class, instruments it and pushes it back to its original path on the testbed. On return of this an Async task, normal program flow resumes. Although this wait time slows down the process, it considerably expands the code coverage of our analysis.

AspectDroid also logs native code invocation, both for simple processes like `Logcat` or through the Java native interface. Although it does not trace the activities within the native code, it does log the name, object, parameters, and return value.

4 Implementation

4.1 Prototype implementation

We implemented a working prototype of our system in Python, Java and PHP. The instrumentation engine is setup on a host machine (64-bit Ubuntu system) for the initial `dex` weaving and dynamic class instrumentation. Our software dependencies includes external tools; `dex2jar`[12], `AspectJ-ajc` [3] compiler version 1.8, and external libraries; `Apache Web Server`[1], `aspectjweaver`[3], `Apache Commons`[2] and `Android SDK`[5]. Our experiments were carried out on both a physical device (a rooted Motorola Droid2 with Android 2.2) and two emulated devices (Android 4.1.2 and 4.4.2). The execution environments are loaded with text messages, calls, contacts, one Gmail account, and some browser history.

4.1.1 Helper component

AspectDroid includes a "helper" component containing modules that automate key actions. In particular, it implements unpacking, re-packaging and application

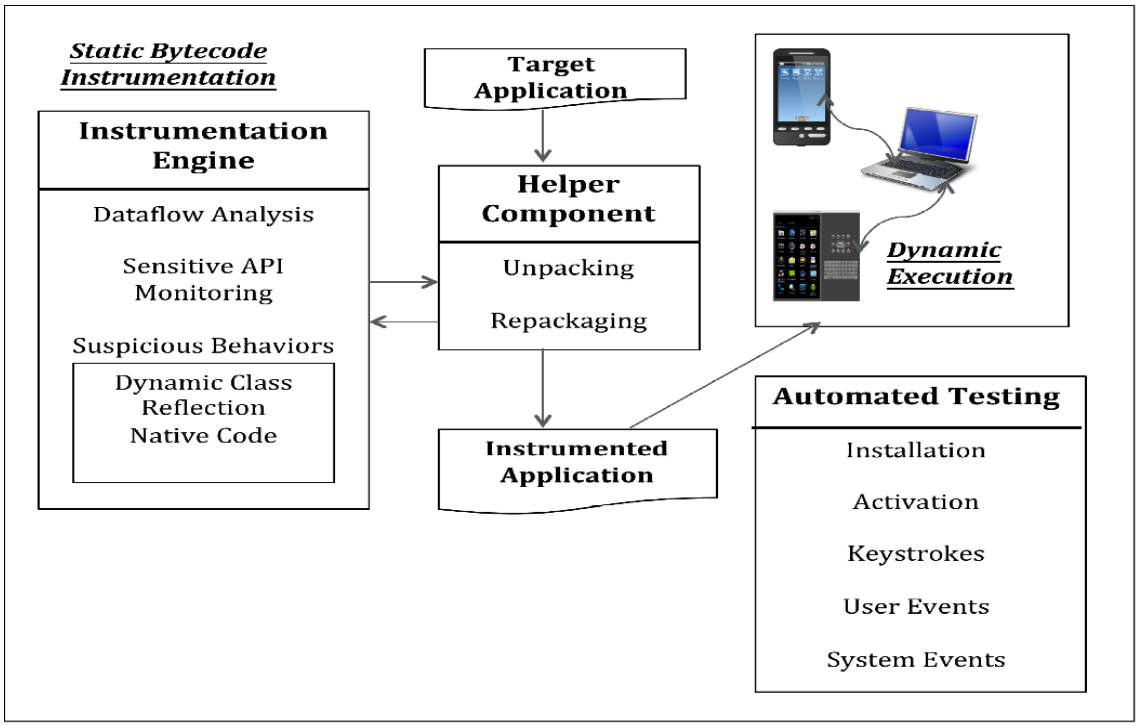

Fig. 2: *AspectDroid* Implementation architecture

signing. Android applications are written in Java and compiled into a compressed class called `classes.dex`. However, the AspectJ compiler does not understand the `dex` file format, thus the need for decompression before weaving. We use a popular open source tool called `dex2jar`, which takes an application file (.apk) or classes.dex as input and outputs a jar file containing individual .class files. When the target application is unpacked, it can be weaved together with desired aspects. After the instrumentation process, the class files are repackaged (dexed and zipped) and re-signed into an Android-compatible app using `jar2dex` and `versign` respectively.

4.1.2 Automated testing

Unlike many traditional applications, smartphone apps are mostly event-driven and exhibit their true functionalities based on user interactions and in response to system events. For example, forcing an SMS to be received so that a broadcast receiver can be activated is an important system event that needs to triggered for us to observe SMS abuse.

In the case of bulk analysis, manual execution of apps and triggering such events can be time-consuming. One of the drawbacks of dynamic analysis is code coverage and a single execution path corresponding to a single app execution, whereby information obtained may not necessarily represent the complete behavior of the target app. Our assumption is the more a tool can explore an app, the more information about the app's behavior can be obtained. For that reason, we build into *AspectDroid* an automated testing module as Python scripts which triggers a series of system and user events to more fully exercise an app's functionality. This module combines some open source tools together with custom-built instrumentation programs. These events are designed to mirror real-life events on a regular Android device. They include:

1. App installation and activation of its main activity, as specified in the manifest, using `adb`.
2. Random keystrokes that simulate user touch and gestures on the app using `monkey`.
3. A user input is simulated where necessary within the instrumentation framework. `EditText` user inputs can be associated with different input types. Most developers specify input types as provided by the Android API – email, password, etc. We make a best effort to generate data to match its possible input type. This program is attached to the body of the instrumentation code.
4. SMS, calls and device settings are generated and manipulated using `uiautomator` while GPS coordi-

nates are simulated and triggered on the emulator by `telnet`.

Independent testing frameworks like Android Monkey are limited to only random application touches and gestures. With our automated testing, the simulated user input built on the `EditText-SetText` method automatically creates the needed `textbox` data during analysis, which proves to be very important. For example, if an EditText is expecting an email, if the Ok button is hit using Monkey, the application may return an error and program execution may not proceed due to an empty text. But with our injected input text, execution will proceed without an error.

Other vital parts of this testing module built with uiautomator help with forcing various system's event like `Calls`, which would otherwise have to be done manually.

5 Testing and Evaluation

Our approach seeks to provide analysts with an easier to use and more flexible system for application analysis. It is capable of examining and monitoring Android applications without restriction based on version and/or platform while still maintaining a very high level of accuracy. The objectives of the evaluation were to quantify the following aspects of the system's performance:

1. *Accuracy.* We tested the accuracy of our data flow algorithm on 105 applications from the DroidBench corpus.
2. *App Analysis.* We further evaluate the effectiveness of our system by comparing the behavioral patterns in 100 real malware families from the Drebin dataset and a set of 100 apps downloaded from Google Play. We examine data exfiltration, Telephony abuse, reflective invocation, dynamic class loading, and native code execution.
3. *Execution overhead.* We measured the cost associated with dynamic execution of the target app post-instrumentation.

5.1 Accuracy of Data Flow Algorithm

DroidBench 2.0 [8] is an open source project consisting of 120 simulated Android applications used for testing analysis tools. These applications evaluate the accuracy of an algorithm in detecting data flow between a source and a sink. The authors employ different methods of data manipulation, such as callbacks, arrays, application lifecycle, inter-application communication, loops, reflection, threading and implicit flows to hide the flow of sensitive data. The apps are relatively small and they may not necessarily be representative of real life apps and/or malware in terms of size. However, they contain a wide spectrum of diverse, tricky data flow paths that can be employed by malicious and/or over-privileged applications, thus making them a corpus of interest to test *AspectDroid*.

Before executing the apps with *AspectDroid*, we execute the untampered dataset to determine if they are running correctly and producing expected results. Out of 120, 15 apps failed to execute correctly in our environment due to either permission errors or other bugs and were excluded from the analysis. The remaining 105 apps were instrumented using our *AspectDroid* prototype. All apps were tested on emulator version 4.4 using the automated testing module except for the three apps from the emulator detection group, which were retested on the physical phone.

Based on the original source code for the 105 apps, the ground truth indicates 86 apps have data leaks and 19 apps have no leaks. Our experiments show that *AspectDroid* yielded 81 true positive (TP) results, 16 true negative, 2 false positives, and 6 false negatives. Thus, the *AspectDroid*'s precision is 97.59%, recall is 93.10%, and the standard F-measure stands at 95.29%. Subsequent analysis showed that in the two false positive cases, tainted and untainted data were added to a data structure, then the app sinks only the untainted data. Our algorithm taints an object that contains a tainted field, entry or element and does not handle removal of that data/object from taint map once it is written. Since the untainted data is still part of a tainted object, we recorded a false positive. With respect to the six false negatives, four were apps with the following propagation paths: `Public API Field1`, `StartProcessWithSecret` and `Implicit Flows`. Our data flow algorithm taints by means of data comparison (possible taint with items on taint map), thus data exfiltration that is not explicit cannot be detected. The remaining two under tainting were a result of an optimization added to our propagation rule in order to reduce the effect of over-weaving (which results in too much additional code added to the application). This optimization is a tradeoff between the effect of over-weaving and a possible false negative; hence, these two false results are avoidable.

5.2 App Analysis

To test the effectiveness of *AspectDroid* for analyzing Android applications for violations of security and privacy concerns, we used malware samples from the Drebin [7] dataset, a corpus comprising 179 malware families.

In our experiments, we picked one sample per family from the top 100 families. For the non-malicious samples, we downloaded 100 Android apps from Google Play. All 200 samples are instrumented, recompiled and executed using our automated testing module.

In our prototype we tagged 27 important data sources, including phone related data (IMEI, IMSI, ICCID, line Number and location data), user data (database queries) and input data. We also created joinpoints on some sensitive APIs that perform telephony functions, native code execution, dynamic class loading and reflection invocation. The data flow and sensitive API traces created after each app execution are then parsed using a Python script to obtain the aggregated result. We categorize the analysis result into 4 groups: data exfiltration, telephony abuse, reflection and dynamic class loading and native code execution.

5.2.1 Data Exfiltration

Malware and to a large extent privacy-agnostic applications often target user and/or phone related data either with malicious intent, for advertisement or identification/records purposes. Most sensitive data are guarded by one-time permissions (for Android versions 1-5) that gives an app open access to quite a large group of data on a device e.g., Phone-State permission. We define exfiltration as unauthorized writes of sensitive data to a file (log, sharedprefs, user-defined files), Network and SMS that are not explicitly granted by the user at the point of transfer. Our analysis of the 100 malware samples showed 127 explicit data exfiltration paths of the 27 tainted sources carried out by 23 samples. Our results showed IMEI, IMSI, ICCID and LN are the most widely exfiltrated phone data. This is followed by contacts, call logs and SMS from user-related data. SharedPref and Network are the most common sink calls we noted while SMS seems to be the least. For the Google Play apps, we observed 25 exfiltration paths, most of which are location and phone IMEI. Network is the sink path for all these data leaks.

5.2.2 Telephony Abuse

SMS is one of the most widely abused resources on Android smartphones. Out of the 100 malware families we evaluated, 8 families were recorded to have some level of SMS abuse. The apps in the Pirater family send SMS to all contacts on the user's phone, posing as the user. The socially engineered, "friendly" SMS generated by Pirater contains a link that downloads the same malware to the receiver's phone if clicked. The MobileTX family, on the other hand, does not just abuse SMS functionality, but also transmits the phone's ICCID to a private number via SMS. The remaining 6 families send specially crafted SMS to premium numbers. We have not recorded any phone call interceptions, spoofing or recording in any of the analyzed malware. We observed the use of SMS in 2 apps and CALLs from 3 apps which belong to the communication category on Google Play. In all these instances, the SMS and CALLs were authorized by the user, based on user-supplied input.

5.2.3 Reflection and Dynamic Class Loading

The Reflection API is part of the standard Java environment and allows method calls to be resolved dynamically at runtime. It is a powerful tool that can be employed by malware to evade static detection. We have observed 5 malware families that use reflection in different ways. We then examine if such invocation exhibits some element of malicious intent. The Mobsquz and FakeDoc families reflectively check if the device has support for telephony-related services (phone calls and SMS). Although this may not necessarily constitute malicious behavior, given the functionality of the applications as an antivirus scanner and battery optimizer, it requires further analysis. The FaceNiff family uses reflection to invoke the methods of a background service that spoofs user accounts and passwords after it has successfully executed the super user command. The 2 other remaining families, BaseBridge and DroidDream, are not suspicious as they both invoke methods from GUI-related classes. We observed 34 instances of reflective call invocation on the Google Play apps. Surprisingly, this is higher than the malware. However, none of the API calls invoked are from our sensitive API call list. We also observed that the BaseBridge family dynamically creates 3 jar files (bootablemodule.jar, moduleconfig.jar, mainmodule.jar) and 2 Dex files (mainmodule.dex and bootablemodule.dex). Within the timeframe for our automated testing and even with an extended manual execution afterwards, the app did not load these new classes dynamically as expected. Thus, we rewrote the binary to force the app to load the new dex files. This enables our dynamic instrumentation to trace the loading joinpoint and the newer classes were instrumented using our dynamic instrumentation engine. For the Google Play apps, 7 dynamic classes were loaded in 5 apps within our testing time. We were able to successfully instrument and execute all the dynamic classes.

5.2.4 Native Processes

Android applications are commonly written in pure Java code, although quite a number of them include an embedded C/C++ binary. Over the years, Android malware has exploited this capability to embed mostly root exploits that trigger privilege escalation. In other instances, Linux commands that communicate with the underlying Android kernel are becoming increasingly common. In our data set, 9 out of the 100 malware families invoke native processes 72 times. Commands like `su, chmod, ps, mount` and Android's `logcat` are the most widely executed native processes. We have also noted the execution of an unknown binary (myicon) in DroidKungFu family. Since *AspectDroid* does not instrument native code, we log the code path and then manually extract the code using adb. An Md5Sum later verified that the native binary is a root exploit belonging to the family `RageAgainstTheCage`. We've noticed native code execution in 6 out of the 100 Google Play apps. In comparison with the malware apps, the Google Play apps all executed ".so" libraries vs. starting other processes like `chmod` or `su`. Beyond exploring that a particular native code has been called within the Java execution, *AspectDroid* does not monitor its content as the instrumentation engine works only on Java. Thus it is inconclusive what some unknown native libraries do.

5.3 Runtime Overhead

The most important costs of instrumentation occur at runtime, since both CPU and memory usage are vital on a resource constrained machine. It is especially important that apps limit their resource usage to avoid possible garbage collection. Though uncommon in foreground processes, this does occur when apps consume too many resources.

The CPU usage is the percentage of CPU time used by a process. We measured the value given the system uptime (uTime), processes start time (startTime) and the CPU time spent in both user and kernel code for the main process and any of its child processes (uTime, sTime, cuTime, csTime). The formula is given below:

$$\begin{aligned} seconds &= upTime - (startTime/Hertz) \\ tTime &= uTime + sTime + CuTime + csTime \\ cpu_usage &= ((tTime/Hertz)/seconds) * 100 \end{aligned} \quad (1)$$

We carried out this experiment by re-running the 100 malware families using automated testing on the same platform, keystroke seeds, and number/pattern of system and user events. Using the `procrank` utility, we obtained the process memory size from each app both before and after instrumentation as well as the CPU indices above. The experiment was executed 5 times and an average for each metric (Memory and CPU) was computed.

The dark portion of the stacked bar chart illustrated in Figure 3 shows the memory usage for each malware pre-instrumentation, while the lighter shade shows the overhead after instrumentation. The data illustrates that the `MemSize` difference is uniform and on average, 1MB of additional memory is required to execute the instrumented application. This translates to *approx* 16% more memory usage on average. In our tests, this overhead caused no issues with any of the apps.

Figure 4 on the other hand shows the percentage of CPU needed to render and execute each malware. The dark portion indicates the CPU-usage before instrumentation while the lighter portion stacked showed the CPU-usage overhead. Although the results are not uniform, the average CPU overhead is approximately 5.91%. Some apps tend to have significantly more overhead than others. We manually examined these apps and found two important factors: the number of data sources tagged and the propagation path can have varied and compounded impact on the CPU usage overhead. CPU intensive apps like games that requested a lot of tagged data, and especially if the request is along the path of an Activity, tend to require more CPU time to load the activity, thus increasing the percentage of CPU usage (e.g., the PJApps and Jifake families). Although the Fujacks malware family has the highest CPU usage pre-instrumentation, its overhead is negligible since it did not request any tagged data.

6 Challenges and Discussion

In the evaluation section we discussed the accuracy of the *AspectDroid* algorithm in detecting data leaks, the importance of tracing resource abuse and detection of suspicious behaviors like reflection, native code and dynamic class loading. Furthermore, we also highlighted the overhead associated with our system. Naturally, some challenges remain. In bytecode weaving, the compiler has to make a best effort adjustment to registers, fields, methods and instructions of the weaved class. Some applications can be sensitive to this kind of intrusion and as such can pose a setback in our recompilation process. We make a best effort to optimize the weaving process, especially in data flow aspects, while at the same time keeping false negatives as low as possible. Specifically, we ensure that:

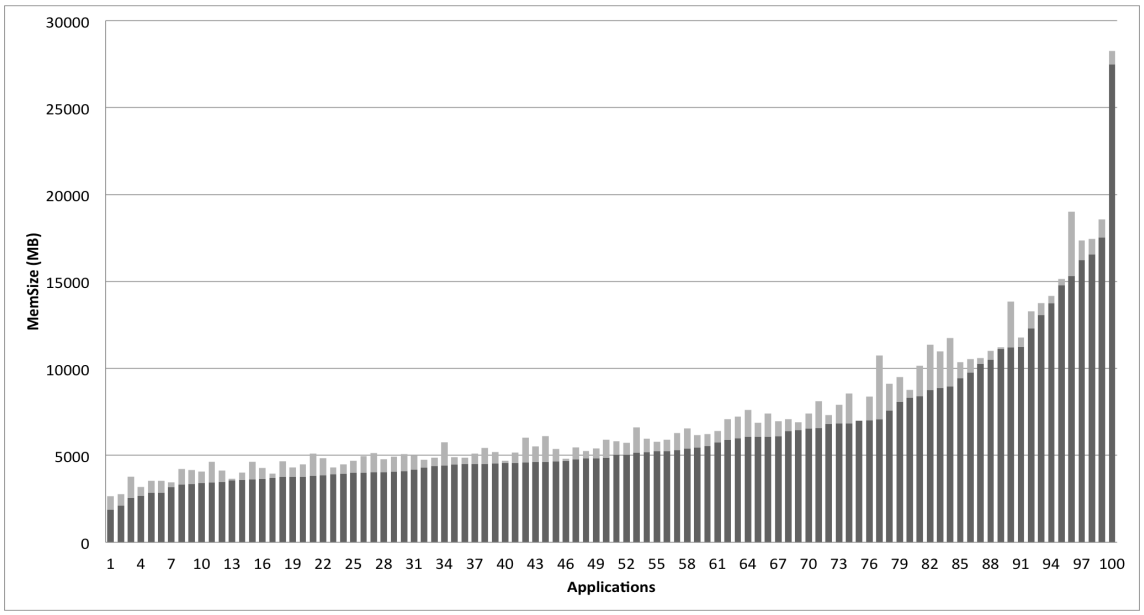

Fig. 3: MemSize Overhead (MB)

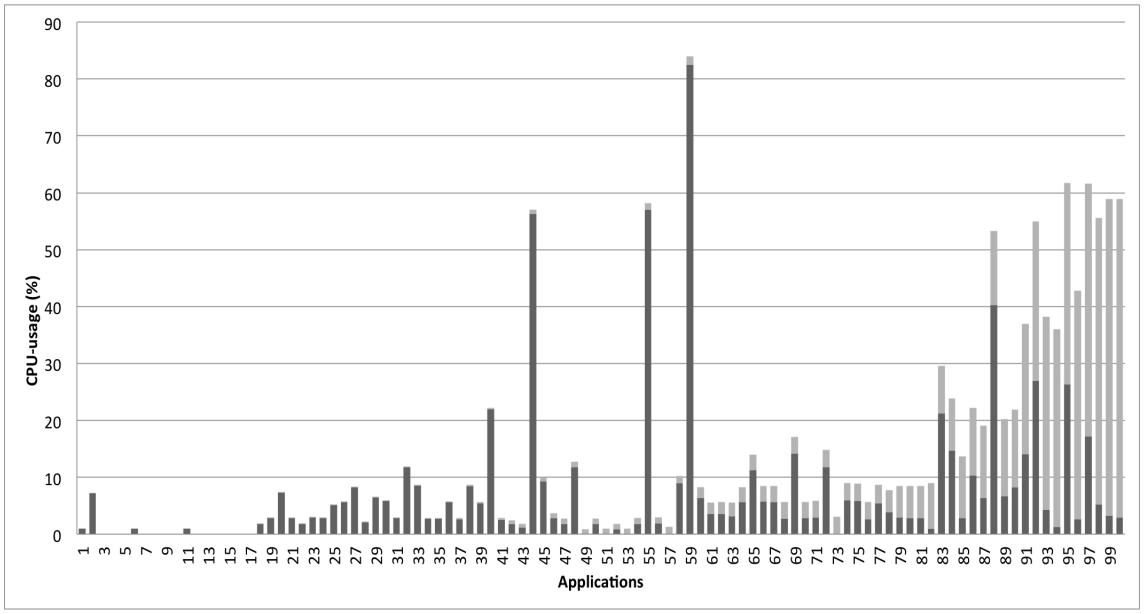

Fig. 4: CPU-usage Overhead (%)

1. Propagation rule 1, which handles primitive returns, excludes void and boolean values. Allowing boolean values in our taint map significantly increases false positives.
2. GUI-related classes that handle graphics, views, and activities are also excluded from propagation in propagation rule 7.
3. Well-known public libraries from Android, Amazon, Google, Samsung, and Apache are excluded from the scope of weaving, however their calls are included within the signatures, if necessary.
4. We use abstract methods within advices to reduce the number of instructions added directly to the weaved class.

Overall, our system effectively uses these optimization techniques to boost its accuracy and performance. In our testing, AspectDroid has proven to be effective in analyzing data flow paths, sensitive API monitoring and analysis of suspicious behaviors. It provides a flex-

ible and efficient system for assessing Android applications and does so with relatively low overhead.

6.1 Limitations

The main limitations of every static bytecode instrumentation are anti-unpacking and anti-repackaging obfuscation mechanisms. Developers can include `obfuscated bytecode` in their compiled dex files that decompilers cannot parse correctly. However, in most cases this obfuscation does not affect method invocation, which is what *AspectDroid* uses to create joinpoints. More so, malware can detect instrumentation code and/or change the package signatures, which can negatively affect analysis with *AspectDroid*.

The AspectJ instrumentation framework is limited to processing only method instructions. As opposed to variable level tainting, AspectDroid's data flow compares the hashes of raw data and as such cannot be affected by simple manipulations through variable reassignment. However, arithmetic instructions can have an adverse effect on our taint propagation (though we have not encountered such in our analysis). As mentioned in Section 3, joinpoints on conditional instructions cannot be created due to the limitation in the Aspectj's APIs. This limits our data flow analysis to explicit data exfiltration.

Another limitation of our approach is analysis of native code. At this point, AspectDroid can only trace to the point where a native class is loaded and executed and it can return the name and parameters for the execution. However, it cannot trace inside native code. Very few Android applications use native code and even with malware, the native code is typically used only for privilege escalation which is heavily dependent on system vulnerabilities.

6.2 Future Work

As part of future work, we are working on improving our automated testing module such that all control flow paths are forced to execute. Using code refactoring, we intend to inject simple methods after arithmetic and conditional instructions that analyze the preceding instructions' parameters. Then joinpoints will be created for the new method call at weaving time. This will take care of possible mis-propagation, thereby improving our data flow analysis. Furthermore, the ability to analyze native code will significantly improve the scope of *AspectDroid* and as such, we intend to include a debugging architecture in a future revision of *AspectDroid*. Within the instrumented advice during native code execution, a debugger can be started with the ID of the new process to collect lower level syscalls made by the native code.

7 Related Work

7.1 Application-level instrumentation

The first application-level dynamic taint tracking on Android was developed by [37,36]. Their system, called Capper, is designed to monitor exfiltration of sensitive data from source to sink. However, their work requires a large amount of static analysis to refactor the Java bytecode and compute taint slices, which are used at runtime as the taint propagation map. This system is prone to most of the inaccuracies of static taint tracking that can result from simple obfuscation techniques. More so, data sources, propagation and sinks that pass through reflective call invocation are not processed if the invoked class and method names cannot be statically resolved. And finally, like most app-level analysis systems, Capper does not handle dynamic class instrumentation. Our system, on the other hand, can perform better data flow analysis since it can handle Java reflection and runtime class instrumentation.

Other research efforts that target app-level instrumentation are [6,9,10,19,25]. The authors of DroidBox developed APImonitor [6] to counter its numerous porting issues. APImonitor like [9], [10], RV-Droid [19] and [25] all use static bytecode instrumentation to analyze method calls in target applications at runtime. Although they use different instrumentation frameworks, these systems are all limited to sensitive API monitoring during program execution. In contrast, *AspectDroid* is a complete analysis system that targets security concerns such as data flow analysis, sensitive API monitoring, as well as analytics of suspicious behaviors.

7.2 Low-Level Instrumentation

Most Android dynamic analysis tools are developed by instrumenting the operating system code and/or the underlying framework. TaintDroid [17] is a real time dynamic taint tracking system that monitors the flow of sensitive data. It uses some basic data flow rules to track the movement of tainted variables, method files and IPC messages from sources until they reach a specified Java library sink.

Several extensions to TaintDroid [16,29,33] were built with added functionalities. DroidBox [16], for instance, logs an app's activities related to starting services, broadcast receivers, SMS and calls made, cryptography oper-

ations performed using the Android API, and file read-/write operations, irrespective of taint marking. Andrubis [33] is an automated analysis system that combines both static and dynamic approaches to an app's analysis. Applications submitted via an online link are dynamically examined on a QEMU-based emulation environment for method tracking, system level analysis and data exfiltration using TaintDroid. Other systems like AppsPlayground [29] added more functionality, such as kernel level-monitoring and automated testing, to TaintDroid. The critical design rule for these approaches rely on low-level instrumentation, making them very OS version-dependent and in some cases platform-dependent. It is important to note that TaintDroid based systems depend fully on the Dalvik virtual machine and as such will require a complete makeover to port to the new Android runtime. Furthermore, stealthy malware can often detect emulation environments which may result in inaccurate analysis. Lastly, due to significant requirements for expert knowledge to port from version to version, the capacities of such system for long term analysis is very limited.

Other host-based dynamic analysis tools are DroidScope, AASandbox and Crowdroid. DroidScope [35] uses virtual machine introspection to monitor the activity of untrusted applications. This system performs API tracing, native instruction and Dalvik tracing, and taint tracking. AASanbox [11], on the other hand, evaluates system call logs by placing hooks between kernel space and user space. These hooks hijack the system calls made and log information such as process ID, syscall name and execution time. CrowDroid [15] analyzes system calls performed by an application based on logs collected using the strace debugging utility in a lightweight CrowdClient. This system is limited to extracting only Linux-specific information like open files, but cannot give broad information on IPC and Android-specific data.

Dynamic binary instrumentation (DBI) systems like DynamicRIO [13], PIN [28], Spike [32], and Dyninst [14] that perform runtime monitoring are mostly dependent on low level system operation. With the exception of PIN, these are not applicable to ARM systems. DBI techniques are also very dependent on the underlying hardware architecture and as such will require modification of the operating system to perform Android app analysis.

8 Conclusion

In this paper we have discussed *AspectDroid*, a hybrid system for Android app analysis, which provides an efficient and flexible alternative for detecting suspicious and illicit behavior independent of Android runtime and or system releases. Our goal is to ease analysis and avoid the numerous problems associated with porting between versions and building a customized device kernel.

The instrumentation engine which is at the heart of *AspectDroid* is designed to achieve three main objectives: data flow analysis, detection of resource abuse and analytics of suspicious behavior like native code and reflective call invocation. *AspectDroid* leverages the AspectJ instrumentation framework to inject monitoring code. The instrumented app is then executed dynamically to trace and log runtime activities at specific joinpoints. It can also instrument runtime classes for further analysis thus increasing code coverage. We have demonstrated that *AspectDroid* can achieve up to 95.29% F-score accuracy in detecting data leaks. Further analysis of 100 malware families for the Drebin dataset and 100 apps from Google Play showed our system can effectively analyze a diverse set of apps, including stealthy malware, with very minimal CPU and memory overhead.

References

1. Apache http server project. [Online; accessed 11-December-2015].
2. Apache commons - common lang, 2015. [Online; accessed 30-August-2015].
3. Eclipse- aspectj compiler, 2015. [Online; accessed 26-August-2015].
4. Ali-Gombe, A., Ahmed, I., Richard III, G. G., and Roussev, V. Aspectdroid: Android app analysis system. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy* (2016), ACM, pp. 145–147.
5. Android-Developers. Android studio, 2015. [Online; accessed 11-August-2015].
6. APIMonitor. Installation and usage of droidbox apimonitor, 2012. [Online; accessed 06-May-2015].
7. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., and Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2014).
8. at the European Center for Security, S. S. E., and by Design EC SPRIDE, P. Droidbench benchmarks, 2015. [Online; accessed 08-May-2015].
9. Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard–enforcing user requirements on android apps. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 543–548.
10. Bartel, A., Klein, J., Monperrus, M., Allix, K., and Le Traon, Y. Improving privacy on android smartphones through in-vivo bytecode instrumentation. Tech. Rep. 978-2-87971-111-9, uni. lu, 2012.
11. Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. An android application sandbox system for suspicious software detection. In *Malicious and*

unwanted software (MALWARE), 2010 5th international conference on (2010), IEEE, pp. 55–62.

12. Bob, P. Dex2jar, 2014. [Online; accessed 13-December-2014].
13. Bruening, Derek Zhao, Q., and Amarasinghe, S. Transparent dynamic instrumentation. In *International Conference on Virtual Execution Environments* (March 2012), VEE-12.
14. Buck, B., and Hollingsworth, J. K. An api for runtime code patching. *Int. J. High Perform. Comput. Appl. 14*, 4 (Nov. 2000), 317–329.
15. Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2011), SPSM '11, pp. 15–26.
16. DroidBox. Droidbox - android application sandbox, 2011. [Online; accessed 06-May-2015].
17. Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. 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* (2010), OSDI'10.
18. Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 235–245.
19. Falcone, Y., Currea, S., and Jaber, M. Runtime verification and enforcement for android applications with rv-droid. In *Runtime Verification*, vol. 7687 of *Lecture Notes in Computer Science*. 2013, pp. 88–95.
20. Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 627–638.
21. Feng, Y., Anand, S., Dillig, I., and Aiken, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), FSE 2014, pp. 576–587.
22. Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, vol. 7344 of *Lecture Notes in Computer Science*. 2012, pp. 291–307.
23. Incorporated, T.-M. Android malware: How worried should you be?, 2012. [Online; accessed 4-October-2016].
24. Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 3–14.
25. Karami, M., Elsabagh, M., Najafiborazjani, P., and Stavrou, A. Behavioral analysis of android applications using automated instrumentation. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on* (2013), IEEE, pp. 182–187.
26. Kiczales, G., Lamping, J., Lopes, C., Hugunin, J., Hilsdale, E., and Boyapati, C. Aspect-oriented programming, Oct. 15 2002. US Patent 6,467,086.
27. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP'97 ? Object-Oriented Programming*, vol. 1241 of *Lecture Notes in Computer Science*. 1997, pp. 220–242.
28. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), PLDI '05, pp. 190–200.
29. Rastogi, V., Chen, Y., and Enck, W. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (2013), CODASPY '13, pp. 209–220.
30. Software, G. Gdata mobile malware report: Q4/2015, 2016. [Online; accessed 3-October-2016].
31. Team, A. The aspectj tm programming guide, 2002-2003.
32. Vasudevan, A., and Yerraballi, R. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48* (2006), ACSC '06, pp. 311–320.
33. Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., and Platzer, C. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001* (2014).
34. Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of the 2012 Seventh Asia Joint Conference on Information Security* (2012), ASIAJCIS '12, pp. 62–69.
35. Yan, L.-K., and Yin, H. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium* (2012), pp. 569–584.
36. Zhang, M., and Yin, H. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium* (2014), NDSS 2014.
37. Zhang, M., and Yin, H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (2014), ASIA CCS '14, pp. 259–270.
38. Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (May 2012), pp. 95–109.
39. Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the Network and Distributed System Security Symposium* (2012), NDSS2012.