AspectDroid: Android App Analysis System

Aisha Ali-Gombe
aaligomb@uno.edu

Irfan Ahmed
irfan@cs.uno.edu

Golden G. Richard III
golden@cs.uno.edu

Vassil Roussev
vassil@cs.uno.edu

Dept. of Computer Science
University of New Orleans
New Orleans LA 70148

ABSTRACT

The growing threat to user privacy related to Android applications (apps) has tremendously increased the need for more reliable and accessible app analysis systems. This paper presents AspectDroid, an application-level system designed to investigate Android applications for possible unwanted activities. AspectDroid is comprised of app instrumentation, automated testing and containment systems. By using static bytecode instrumentation, AspectDroid weaves monitoring code into an existing application and provides data flow and sensitive API usage as well as dynamic instrumentation capabilities. The newly repackaged app is then executed either manually or via an automated testing module. Finally, the flexible containment provided by AspectDroid adds a layer of protection so that malicious activities can be prevented from affecting other devices. The accuracy score of AspectDroid when tested on 105 DroidBench corpus shows it can detect tagged data with 95.29%. We further tested our system on 100 real malware families from the Drebin dataset [1]. The result of our analysis showed AspectDroid incurs approximately 1MB average total memory size overhead and 5.9% average increase in CPU-usage.

CCS Concepts

•**Security and privacy** → **Malware and its mitigation;**
•**Software and its engineering** → *Dynamic analysis;*

Keywords

Android, Instrumentation, AspectJ, Dynamic Analysis

1. INTRODUCTION

Many Android applications are well-known for privacy violations and data leakage [2]. For instance, they may transfer personal data outside the devices of end-users without their consent. Andrubis [3] performed an analysis on over a million malicious and benign apps, and found that 38.79% of the apps have various forms data leakage. The security and privacy concerns surrounding these revelations increases the need for reliable and accessible app analysis systems.

In this paper, we present `AspectDroid`, a dynamic 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 Dalvik virtual machine. It is a fully automated system that weaves in monitoring code at compile time, using a set of predefined security concerns such as data/resource usage and possible abuse, new code execution, and other non-traditional behaviors like reflective calls and native code execution.

AspectDroid has an automated testing engine that provides a means of executing the instrumented application while stimulating random user and system events without human interaction. Since apps can be monitored using real life scenarios, both with regards to the platform on which the application is executed as well as events involving real data, AspectDroid allows definition of a containment policy such allowing sink calls or and blocking or manipulating sink data content.

We use two well-known datasets of Android apps for thoroughly evaluating the effectiveness and efficiency of AspectDroid. The first dataset contains 105 Android apps from the DroidBench project and is used to detect data leaks (by apps), and effectiveness of containment policies. The results show that AspectDroid can accurately detect data leaks with 95.29% F-score accuracy, and can effectively allow, block or manipulate data at sinks without crashing applications. The second dataset contains 100 malware families from the Drebin dataset and is used to detect four important aspects of an Android application: data exfiltration, use of reflection and dynamic class loading, use of native code and SMS abuse. The results of our analysis show that Device ID, Subscriber ID and Sim serial number are the most widely exfiltrated phone data; five malware families use reflection for malicious purposes such as invoking the methods of a background service to spoof user accounts and passwords; eight families have some level of SMS abuse, such as sending SMS to all contacts on the user's phone posing as the user; and nine families invoke native processes.

We also used the second dataset to measure the instru-

CODASPY'16 March 09-11, 2016, New Orleans, LA, USA

ACM ISBN 978-1-4503-3935-3/16/03.

DOI: http://dx.doi.org/10.1145/2857705.2857739

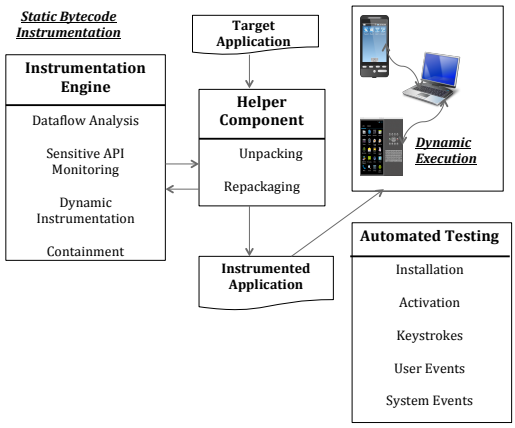

Figure 1: *AspectDroid* system architecture

mentation overhead on both static weaving and dynamic execution. The results show that AspectDroid has limited memory overhead of around 1MB, and a reasonable 5.9% CPU-Usage overhead.

2. SYSTEM DESIGN

AspectDroid is an Android app analysis system that consists of three modules: an *instrumentation engine*, an *automated tester*, and a *helper component* (see Figure 1).

The task of the *instrumentation engine* is to inject monitoring code into the target app statically based on some specific cross-cutting concerns. The injected code executes alongside the original code and performs custom logging and other analytical functions with the help of the *automated tester*. Finally, the *helper component* provides some utility/supporting functions needed before and after the instrumentation phase.

2.1 Instrumentation engine

The instrumentation engine (IE) forms the backbone of AspectDroid and is designed to address four objectives; *Dataflow analysis*, *Sensitive API monitoring Dynamic instrumentation Containment policy enforcement*

2.1.1 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, propagation and sink* joinpoints.

Taint sources represent the data of interest; in our case, we are interested in sources that are relevant to the privacy and security of the user and the data stored on the device. We define *vital* sources as phone-related data, component providers, file reads and user input APIs. Specialized pointcuts are created using signatures to intercept these API joinpoints. 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

Taint sinks are defined as points where our 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.

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. The following list illustrates our seven (7) point rules in picking a propagation's joinpoint:

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 other object return type that contains tainted arguments.
7. Rule 7: All joinpoints with other object return type that contains tainted target.

In order 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.

2.1.2 Sensitive API monitoring

Access to resources and sensitive data are requested through specialized API calls. Access to media, telephony (SMS and calls), reflection invocation and native code execution are important points of interest in performing in-depth analysis, therefore we define pointcuts that pick the joinpoints corresponding to such APIs. The advice at such joinpoints logs the corresponding target, arguments(s) and return value.

2.1.3 Dynamic instrumentation

AspectDroid implements *dynamic 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 automated testing engine, and waits for notification to proceed. On receipt, the component will pull the referenced dex file with `adb` and instrument it as necessary. The resulting dex file is pushed back to its original directory and

the normal program flow is resumed. This dynamic instrumentation feature considerably expands the coverage of the functionality of target applications.

2.1.4 Containment

AspectDroid targets the analysis of all kinds of applications, including malware. As such, we need a containment policy that will restrict malicious apps from going wild. Within the design of our instrumentation aspects, we built some flexible containment policies such that analyst can choose if sink calls are to be executed, blocked or manipulated. *Execution* containment policy allows the sink to proceed with its original target object and parameters. *Blocked* policy completely stops the joinpoint from executing by returning `null` to its around advice. This policy means if the program is on the verge of sending data over the network, that call will be skipped and program execution will continue. *Manipulated* policy modifies the parameters associated with the sink joinpoint if it contains tainted data. For example, if a `sendMessage` joinpoint is sending out location information, the data will be replaced with some random string, and the joinpoint will be allowed to continue.

2.2 Helper component

The helper component contains modules that automate key utility actions and ease the flow of *AspectDroid*. In particular, it implements unpacking, re-packaging and application signing.

2.3 Automated testing

For bulk testing, these events are designed to mirror real-life events on a regular Android device:

1. Installation of the repackaged app on a device/emulator.
2. Activation of the main activity as specified in the manifest.
3. Random keystrokes that simulate user touch and gesture on the app using *monkey*.
4. User input is simulated within the instrumentation framework using *EditText* user input types.
5. Incoming and outgoing SMS and calls are generated using *uiautomator*.
6. GPS coordinates are simulated and triggered on the emulator via telnet.
7. Device settings related to the network (wifi and cellular network), Bluetooth, and location access are set on and off on the emulator.
8. Information obtained from various joinpoints is logged in trace files.

3. TESTING AND EVALUATION

The objectives of the evaluation were to quantify the system's accuracy, ability to thwart malicious actions, and execution overhead. We tested the accuracy of our data flow algorithm and our containment policies on 105 applications from the DroidBench corpus. Our results indicate *AspectDroid* can detect data leak with F-score accuracy of 95.29%.

Table 1: Malware Analysis Result

	Malicious Apps
Data Exfiltration	*AckPosts, Aks, Ancsa, jSmsHider, Saiva, Vidro, Gonca, RootSmart, RATC, JSExploit-DynSrc, Xsider, Ssmsp, Mobsquz,FakeTimer, DroidKungFu, Spy.GoneSixty, Kmin, GGTrack, MobileTx Dougalek, FakeDoc, Loozfon, Placms*
Telephony Abuse	*MobileTx, Iconosys, UpdtKiller Pirater, Mania, FakeInstaller FakePlayer, Foncy*
Reflection and Dynamic Class Loading	*Mobsquz, FakeDoc FaceNiff, BaseBridge DroidDream*
Native Code	*Ancsa, Qicsom, RATC DroidKungFu, Xsider DroidSheep, Gmuse FakeDoc, FaceNiff*

Using 100 real malware families from the Drebin dataset, *AspectDroid* examines them for data exfiltration, reflective invocation and dynamic class loading, SMS abuse and native code as shown in Table 1. Finally, the dynamic execution overhead for *AspectDroid* has an average of 1MB memory overhead and 5.9% CPU usage overhead.

4. CONCLUSION AND FUTURE WORK

In this paper we've discussed *AspectDroid*, our hybrid system for Android application analysis, which provides a comprehensive, efficient and flexible alternative for analysis of Android applications to detect illicit activity. We have shown that *AspectDroid* can detect data leaks with acceptable accuracy while keeping its resource overhead minimal. As part of our future work, we intend to improve *AspectDroid*'s manipulation containment policy as well as provide a more generic automated testing module.

Acknowledgment

This work was partially funded by the NSF grant, CNS #1409534.

5. REFERENCES

[1] 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).

[2] 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.

[3] 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).