Enhancing Accuracy of Android Malware Detection using Intent Instrumentation

Shahrooz Pooryousef and Morteza Amini
Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
pooryousef@ce.sharif.edu, amini@sharif.edu

Keywords: Android Security, Malware Detection, Code Instrumentation, Static and Dynamic Analysis.

Abstract: Event-driven actions in Android malwares and complexity of extracted profiles of applications' behaviors are two challenges in dynamic malware analysis tools to find malicious behaviors. Thanks to ability of event-driven actions in Android applications, malwares can trigger their malicious behaviors at specific conditions and evade from detection. In this paper, we propose a framework for instrumenting Intents in Android applications' source code in a way that different parts of the application be triggered automatically at runtime. Our instrumented codes force the application to exhibit its behaviors and so we can have a more complete profile of the application's behaviors. Our framework, which is implemented as a tool, first uses static analysis to extract an application's structure and components and then, instruments Intents inside the application's Smali codes. Experimental results show that applying our code instrumentation framework on applications help exhibiting more data leakage behaviors such as disclosing Android ID in 79 more applications in a data set containing 6,187 malwares in comparison to using traditional malware analysis tools.

1 INTRODUCTION

Smart Phones play an essential role in many concepts of our daily life such as entertainment, socialization and business (Corporation, 2014). Users of these devices can download diverse applications from third party markets and install them on their devices. As well as these devices be common, they have become the primary target of many attackers (Report, 2014). Android, as the most popular platform in the smart phone market share(IDC, 2014), has huge malware surface (Zhou et al., 2012a; Rastogi et al., 2014). According to the McAfee reports in 2013, Android has 2.47 million new malware samples, which indicate a 197% increase over 2012 (Report, 2014). Most of these malwares spread through the online markets such as Google Play (Zhou et al., 2012a) and these malwares perform activities such as collecting user information, sending premium-rate SMS messages, or stealing credentials (Felt et al., 2011b).

Many different static and dynamic methods have been proposed in the industry and academic world for detecting Android malwares. Static analysis methods (Bartel et al., 2014; Yang et al., 2015; Grace et al., 2012; Chin et al., 2011) search malicious signatures inside the application's source code. In contrast, dynamic malware analysis methods (Zheng et al., 2012a; Enck et al., 2014; Gilbert et al., 2011; Zhou et al., 2012a) run applications inside a sandbox environment and analyze applications' behaviors. Even though dynamic malware analysis tools exhibit many undesirable behaviors of malwares, they cannot capture all of the applications' suspicious behaviors completely. Android malwares use different tricks to evade from detection by malware analysis tools. Unfortunately, in Android, traditional dynamic malware analysis methods cannot yield a good accuracy in detecting suspicious behaviors due to the following reasons:

Existing Event-driven Actions. Due to the Android architecture, applications can register some of their functionalities to be triggered whenever an event is received or happened in the system. Android malwares use this feature to hide their suspicious behaviors in the presence of dynamic malware analysis methods. For example, a malicious action in the application can be triggered whenever the device is connected to the charger.

Complexity of Extracted Profiles of Applications' Behaviors. Second problem in Android malware analysis is the complexity of extracted profiles of applications' behaviors. In malware detection, getting a fine grained profiles of applications' internal behav-

Pooryousef, S. and Amini, M.
Enhancing Accuracy of Android Malware Detection using Intent Instrumentation.
DOI: 10.5220/0006195803800388
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 380-388
ISBN: 978-989-758-209-7
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

iors is critical for obtaining a good accuracy (Zhang et al., 2013). Some features of Android applications such as using Binder mechanism prevalently, or interacting with OS services (directly or indirectly) make extracting the profile of applications a very complex process. For example, using Binder mechanism leads generating so many system calls in the application so that extracting suspicious trace of system calls between these so many system calls is a challenging task (Zhang et al., 2013). In addition, after extracting application behaviors, we need to consider some other concepts such as the contexts of actions for differentiating them from the benign behaviors (Yang et al., 2015).

Evading from detection in Android malwares along with the complexity of generated profiles could impact on the accuracy of dynamic analysis methods. With our experiments on a data set containing 6,187 Android malwares, we found that many of the applications Security Sensitive APIs (SS-APIs) are invoking inside the specific components such as broadcast receivers. These components help malwares to trigger their malicious actions at the specific conditions. In this paper, in order to help dynamic malware analysis methods gathering more complete and exact profiles of applications' behaviors, we propose a framework for automatic code instrumentation in Android applications. With code instrumentation at an application's specific points, we can trigger the application's hidden behaviors and get a straight profile of the application's critical parts. After the instrumentation, our tool automatically builds and generates an executable application package and signs it.

In our framework, we enjoy Intents messages (Chin et al., 2011) to invoke and trigger different parts of Android applications automatically at runtime. In our proposed method, in contrast to fuzzing methods, for analyzing an application, by considering the application's structure and attributes, we do not need to generate too many input values (Fuzz, 2014). In fact, our proposed framework, first extracts the application's call graph and then, instruments Intents in the application's Smali codes and generates a new APK file of the application. As we directly work on the application's Smali codes and do not need to get the application's Java source code, neither decompilation of the Dalvik byte codes to Java source code nor obfuscation techniques have negative effect on our instrumentation process.

The rest of this paper is organized as follows. In Section 2, some background information about Android applications structure, Intent mechanism, and Smali format is presented. Section 3, presents the components of our Intent instrumentation framework, and describes these components' functionality, separately. After that, we present our evaluation results in Section 4. Section 5 presents the related work, and finally Section 6 concludes the paper and draws the future work.

2 BACKGROUND

Before describing the proposed method for code instrumentation in Android applications, we require to survey the components of Android applications, briefly describing the Android Intent mechanism, and introducing the Smali format of Android applications source code.

2.1 Components of Android Applications

Android is a mobile operating system that runs on the millions of devices in the world. One of the key success of the Android huge market share is existence of diverse applications for this platform; because of the open nature of its applications market place (Xu et al., 2012). Android applications usually are developed in Java language and packaged in APK files, which are ZIP compressed files. Android applications consist of four different components in order to perform their tasks. These components are activity, service, broadcast receiver, and content provider. Activities include applications' interfaces which users interact with them. Services are components that are used for background processing and do not have any interface for interacting with users. Anytime that an action needs to perform some long operations which do not need user interactions, developers use such components. Broadcast receivers are triggered with broadcast messages which are broadcasted by the operating system or other components of applications. The content providers can store and retrieve data in many types such as file system and SQL Lite data bases.

In addition to these components, Android applications have an XML file (Manifest.xml file) which includes some important information about an application such as the required permissions of the application, the names of its components, and the version of the OS compatible with the application. In Fig. 1, we have shown a manifest file where a broadcast receiver component and a service have been defined.

2.2 Android Intent Mechanism

Android uses Binder and Intent messages for providing Inter Process Communication. Binder mechanism

```
<receiver android:name="MyScheduleReceiver" >
        <intent-filter>
          <action
android:name="android.intent.action.BOOT_COMPLETED"
/>
        </intent-filter>
      </receiver>

   <service android:name=".BackgroundService" >
     </service>
```

Figure 1: An example of an Android Manifest.xml file that defines a service and a broadcast receiver.

is used when an application requires to interact with system privileged services (Chen et al., 2015). In order to provide a mechanism for interacting an application's components with each other and also with other applications' components, Android uses Intents (Chin et al., 2011). An Intent object, is a passive data structure that has an abstract description of an operation to be performed. In addition to the applications, the operating system can use Intents for triggering different components of applications. Intent invocation is done through some APIs such as *startActivityForResult*, and *startActivity*. In Android, Intents can be sent explicitly or implicitly. In explicit sending, sender specifies the receiver in the Intent definition. In implicit sending, sender does not specify the receiver and the operating system based on the Intent description chooses the target application or component for handling the received Intent.

2.3 Smali Format

Smali format is a representation of dex format (opcodes) used by Android Dalvik machine (Zheng et al., 2012a). Smali codes of Android applications can be obtained using tools such as Apktool (Google, 2014). To give a small example of the Smali format, in Fig. 2, the Smali code of a Java method is represented. This method writes custom log messages.

3 PROPOSED INSTRUMENTATION FRAMEWORK

The main idea of our Intent instrumentation framework is to place some specific Intents at the specific points of the application's source code. The goal is to trigger the components of the application automatically in order to activate all parts of the application at runtime. Furthermore, we can choose which com-

```
.method public log()V
   .locals 2

   .prologue
   .line 13
   const-string v0, "remote"

   const-string v1, "This is a Log method"

   invoke-static {v0, v1}, Landroid/util/Log;->v(Ljava/
lang/String;Ljava/lang/String;)I

   .line 14
   return-void
.end method
```

Figure 2: Smali format of a method.

ponents of the application be run respectively with regards to the application's function call graph. As we do not have access to the application's Java code, we use Smali codes for Intent instrumentation. The overall architecture of our framework has been depicted in Fig. 3. Our tool is based on static analysis, and thus we disassemble Android applications to the Smali files first. Then, we specify the structure of the application, for example the application's call graph and type of the components and their input values which are passed to them at runtime. After the instrumentation, we decompile the application's Smali files to generate a new APK file. In below, each of these steps are explained.

3.1 Disassembling Application

In the first phase, our tool takes an APK file, disassembles its dex file, and generates Smali files for all application's components, for example broadcast receivers and services. We disassemble applications' APK files using Apktool (Google, 2014). In this step, each Smali file is mapped to a single Java class, and its path corresponds to the path of the Java class in the application's source code. Working directly on the Smali codes helps us to overcome the obfuscation techniques which make most static analysis tools inefficient.

3.2 Extracting Application Structure

The tool then, parses the application manifest and Smali files for extracting the application's structure, i.e., the application's call graph. We used Androguard toolset (Desnos, 2014) to construct the application's function call graph in this step. Thereafter, we parse the application's manifest file to extract the names of

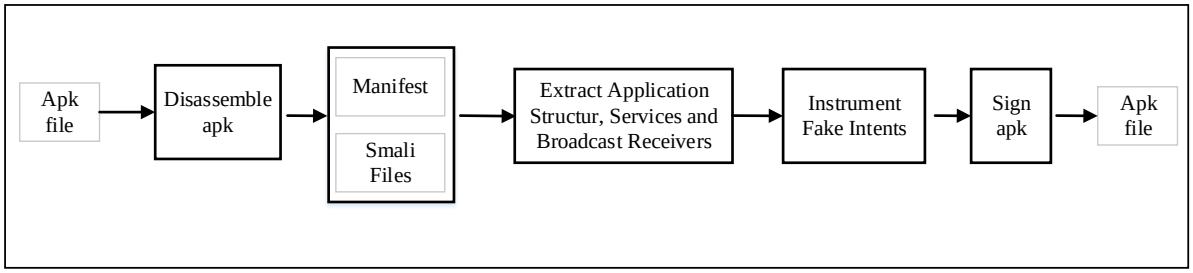

Figure 3: The steps of our Intent instrumentation framework.

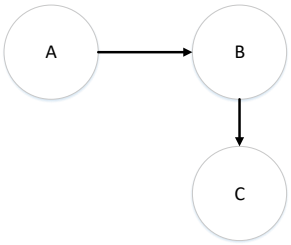

Figure 4: Dependency in component invocation.

its components. For generating Intents in the next step, we need some information about each component, such as components' expected values, whenever they are invoked from other components at runtime. In this step, these values are identified and extracted.

3.3 Instrumenting Intents

Manipulating an application's Smali codes and adding some new codes between Smali instructions can influence the application's main execution flow and so leads to change the application's original behaviors. In addition, we cannot guarantee that merging our Intent codes into the target application, does not affect the manipulated application to crash after installation. Also, we need to make sure that our instrumented Intents will invoke the application's different components at runtime. Specifically, for instrumenting Smali codes inside Android applications we need to consider the following issues.

Application Functionality: The order and dependency of invoking the application's components must be considered in triggering the application's components. For example, as depicted in Fig. 4, first, component *A* calls component *B* and then, component *C* is triggered by component *B*. So, we must consider this sequence and dependency in generating our Intents for the application. In addition, for some components, namely SMS receivers, if the target component expects some specific inputs, we must include these inputs in our Intent definition.

Executing Instrumented Codes: Another issue in instrumenting Intents, is assurance of executing the injected codes. If the application execution reaches with low probability to where our Intents have been instrumented, the place of instrumentation is not appropriate. This is a crucial issue; because our Intents must surely invoke the application different components.

In order to overcome the above constraints, we instrument our Intents at a specific point of the application. The best point is the application's main activity. Each application in Android, has an activity which is the application's main entry point. This activity is specified with "LUANCHER" tag in the manifest file. As *oncreate* method exists in all activities, we chose this method for placing our Smali codes. We found experimentally that the function's beginning is the most appropriate place for placing our codes and instrumenting in other places leads applications to crash at runtime.

Another challenge is the registers of the *oncreate* method. If they are manipulated carelessly, it can result in crashing the application at runtime. In Dalvik bytecode, registers are always 32 bits, and can hold any type of values; however there is a limitation on the number of registers for each method. As we require to use registers in defining our Intents and invoking them, changing the values of the registers that are used in another points of the application is risky. In our proposed framework, we need only three registers for Intent definition. However, for some components, if we require to pass some values to a target component, we need more registers.

In instrumentation, we consider components that have at least one SS-API. SS-APIs are APIs in Android system which handle a security sensitive action in the system (for example, connection to the Internet) (Yang et al., 2015). These APIs trigger the Android access control mechanism whenever they are invoked. We have used Felt et al. (Felt et al., 2011a) proposed method for identifying SS-APIs in applications.

Algorithm 1 depicts the steps of our Intent instrumentation approach. After the instrumentation, our tool automatically generates an executable APK and

signs it, which can be installed on the device. Our framework sign each instrumented APK using a randomly generated key.

Algorithm 1: Intent Instrumentation Flow chart.

Input: *Application's Smali Files, Application's Call Graph*
Output: *Instrumented APK file*
1: Find the Application's main activity and *oncreate* function
2: **for** Each application Component: **do**
3: **if** Component is an end point in the call graph and have **SS-API then**
4: Extract components' attributes
5: Generate Intents with considering components structure
6: Instrument Intents in *Oncreate*
7: **end if**
8: **end for**
9: **return** *Instrumented APK file*

4 IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented the proposed framework to show the applicability of the proposed approach and evaluate it. In our implementation, we used AndroGuard (Desnos, 2014) modules for constructing applications' function call graphs. For identifying the Android privileged APIs, we used the results of Felt et al. (Felt et al., 2011a) research. In order to disassemble APK files to Smali files, we employed Apktool (Google, 2014). Apktool allows us to take an Android APK file, convert it into a Smali source representation, and then recompile it back into a new APK file. We used Python scripts for static analysis and integrating the mentioned tool set and modules.

Using the implemented framework we can evaluate the feasibility and the beneficial of our code instrumentation approach on malware detection. First, we show the impact of profiling the specific parts (or components) of applications on analyzing application behaviors. Afterward, we evaluate the impact of our Intent instrumentation on illustrating more suspicious behaviors of Android malwares using traditional malware analysis frameworks such as AppsPlayground (Rastogi et al., 2013). We analyze how the accuracy of malware analysis tools can be enhanced after Intent instrumentation. The following subsections, demonstrate the practical advantages of our Intent instrumentation framework.

4.1 Malware Collection

To evaluate our framework, we collected 6,187 Android malwares, which are publicly available for researchers, in June 2016. Among these malwares, 4,367 Android malwares were downloaded from Virus share repository (VirusShare, 2014) and 1,820 malwares from ISBX center (ISBX, 2014). Our malware data set consists both old and new malware samples, and contains samples of the most known malware families.

4.2 Extracting Profiles of Applications

As explained in 1, the complexity of extracted profiles of applications can produce challenges for traditional proposed solutions in extracting and analyzing suspicious behaviors. In this section, we show that how separate analyzing the different components of an application can simplify the identification of the suspicious behaviors of the application. Note that our goal is not identifying the malicious behaviors and we just illustrate that extracting and separate analyzing an application's different components can show up clues about suspicious behaviors quickly. We consider a trace of function call graph which reach an SS-API as a sign of a suspicious behavior in our analysis process. Distinguishing malicious behaviors from the benign ones in the extracted behaviors requires further analysis using mechanism, such as machine learning techniques, which is out of the scope of this paper.

We analyzed the whole data set for characterizing how much Android components namely broadcast receivers and services are used in Android malwares and how many of applications' overall SS-APIs include in each component. As indicated in Fig. 5, 5,870 of the collected malwares (94.8%) contain at least one service and, 5,265 of the collected malwares (85%) include at least one broadcast receiver. Also, 3,633 of the malwares have at least 7 services and 2,400 of the malwares have at least 7 broadcast receivers. Almost 54 percent of applications' SS-APIs are invoked in broadcast receiver and service components as depicted in Fig. 6. This indicates that most of the malwares use SS-APIs in broadcast receivers and services permanently. In this way, malwares can trigger their malicious behaviors at the specific conditions and can evade from detection in malware analysis tools.

Considering behaviors of components separately can be helpful in finding malicious behaviors using traditional analysis methods, especially the ones analyzing applications' behaviors at system call level. We did a simple experiment on our data set for sup-

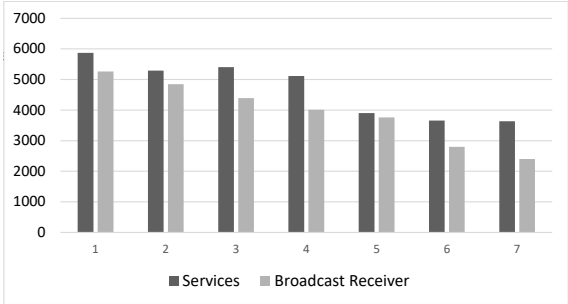

Figure 5: Number of components used in the malwares.

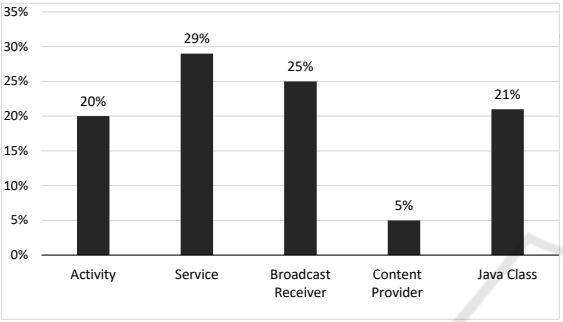

Figure 6: Percent of used SS-APIs in the components of malwares.

porting this claim. We disassembled some of collected malwares using Apktool and analyzed Smali files of these malwares. Precisely, we tracked applications' function call graph in order to find a trace of API calls which results in invoking an SS-API as a sign of suspicious behavior. We found that for finding a suspicious behavior in an application (a trace of APIs), we need to analyze in average six traces of function calls as depicted in Fig. 7. However, if we analyze different parts of applications' profiles separately, we need to analyze only five traces in average. Of course, for finding some malicious behaviors, we may need more comprehensive analysis on the whole application's profile.

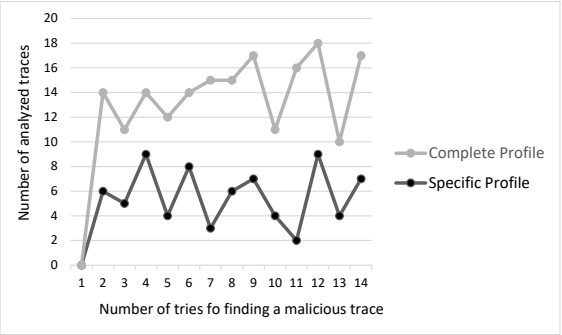

Figure 7: Extracting malicious behavior per trace.

Table 1: Overall detection of more leaked private information in our dataset.

Information type	Number of applications
Android ID	79
IMEI	67
Phone number	11
Contact number	9
Location data	10

4.3 Malware Analysis in Presence of Intent Instrumentation

Here, the advantages of our code instrumentation on exposing malware suspicious behaviors using one of the most comprehensive dynamic malware analysis tools, named AppsPlayground (Rastogi et al., 2013), is evaluated. Although our instrumentation approach can be employed in any dynamic malware analysis framework such as CopperDroid (Tam et al., 2015) or VetDroid (Zhang et al., 2013), we employed it in AppsPlayground, due to the fact that its source code is available online and has a very friendly interface. We compared the reports of AppsPlayground on our instrumented APKs with reports generated on the original APKs.

AppsPlayground is a framework for automated dynamic security analysis of Android applications. It integrates a number of detection, exploration, and disguise techniques to come up with an effective analysis environment. AppsPlayground can detect malicious activities inside malwares and also activities that are not malicious but can be annoying.

AppsPlayground tracks some data such as Android ID and IMSI in applications. Android ID is used to identify a user device and is a unique ID for each device. It is used usually in specific applications such as games or applications which need to identify the user's device. IMSI identifies a GSM subscriber on the cellular network. Other types of data that might be leaked include contact numbers and location data. In order to evaluate the AppsPlayground accuracy after Intent instrumentation, we compared the reported data leakage in two runs of it on our data set.

After applying our framework on the data set including 6,187 malwares, we identified exposures of private sensitive information in 79 instrumented applications which did not identified by AppsPlayground employing original APKs. Even though applying our framework on our data set did not make AppsPlayground to identify new uncaptured malwares, our instrumentation leads to exposing more suspicious behaviors in analyzed applications. The results of the detection of more leaked private infor-

mation are presented in Table 1. The found exposures include 79 leakage of Android ID and 67 leakage of IMEI. In the identified malwares, which leak user information, we found that related components need specific inputs. In fact, our instrumented codes force the components to be triggered and exhibit their behaviors. In addition, in 16 malwares, we saw that instrumented codes trigger some components of the applications which invoke native codes developed in C and C++ language. Although native codes may be used by benign Android applications for performance reasons, since the native code can have system call directly, malwares can use this capability to exploit vulnerabilities in OS kernel level.

5 RELATED WORK

There are several studies on the analysis of Android applications and characterizing the behaviors of Android malwares. In this section, we categorize the related studies into static code instrumentation, and dynamic analysis methods.

5.1 Code Instrumentation and Static Analysis

Code instrumentation in Android applications and generating new APK from modified Smali files can be used for benign or malicious purposes. Some work use code instrumentation at runtime or statically, as what we do for analyzing Android applications, for detecting application vulnerabilities. Karami et al. (Karami et al., 2013) proposed a framework for automatic code instrumentation in Android applications using behaviors of applications at runtime and also I/O system calls of applications. Aurasium (Xu et al., 2012) enforces user security policies in Android applications which enables dynamic and fine-grained policy enforcement. Aurasium, In contrast to work such as (Bugiel et al., 2013), (Jeon et al., 2012), and (Pooryousef and Amini, 2016), which modify Android source code for preparing a fine grained access control in applications, automatically repackages arbitrary applications to attach user-level sandboxing and policy enforcement code (Xu et al., 2012). IntelliDroid (Wong and Lie, 2016) is a generic Android input generator to produce inputs specific to dynamic analysis tools, in order to achieve higher code coverage in malware analysis process. ApkCombiner (Li et al., 2015) combines multiple Android applications using applications' Smali codes to analyze inter app vulnerabilities. ADAM (Zheng et al., 2012b) uses Smali code instrumentation for evaluating Android antivirus softwares. In comparison to the existing code instrumentation frameworks, our framework, by considering application structure and attributes of applications' components, can trigger applications' hidden behaviors and get a straight profile of the applications' critical parts. Using our framework, we can consider invoking some specific components. This is a good idea for analyzing the behaviors of some specific components separately. In addition, for malicious purposes, attackers can instrument malicious Smali codes into benign and popular Android applications (Zhou et al., 2012b; Zhou et al., 2013).

Analyzing source code of Android applications have been widely covered in many works which focus on static analysis methods. Appcontext (Yang et al., 2015) uses context of actions in Android applications to identify and differentiate malicious and benign actions. AppContext uses static analysis to extract the application function call graph and related actions and then correlates application contexts with the actions. SAAF (Hoffmann et al., 2013) analyzes application Smali codes and creates program slices for performing data-flow analysis. Bartel et al. (Bartel et al., 2014) use static analysis for extracting permissions in Android applications. ANDROGUARD (Desnos, 2014) is a tool set that decompiles and analyzes Android applications in order to detect malicious applications using their signatures. Stowaway (Felt et al., 2011a) statically analyzes that how an application uses Android APIs to detect whether an application's requests overpass its privileges or not. Woodpecker (Grace et al., 2012) analyzes Smali codes of applications statically to uncover capability leaks. Intent's vulnerabilities in components of Android applications have been analyzed with ComDroid framework proposed in (Chin et al., 2011).

5.2 Dynamic Analysis

Dynamic analysis of Android applications' behaviors is considered in many researches. We briefly introduce these works but do not elaborate them in details, because their overlap with our work is rather small. TaintDroid (Enck et al., 2014) uses data tainting for tracking data flows inside applications for preserving user privacy. AppInspector (Gilbert et al., 2011) can automatically generate input and log during program executions, and can detect privacy leakage behaviors by the analysis of log information. AppsPlayground (Rastogi et al., 2013) tries to drive applications along multiple paths in order to reveal their suspicious behaviors. SmartDroid (Zheng et al., 2012a) focuses on Android applications' behaviors which trigger with user interactions and uses Smali code

analysis for revealing application's structures. VetDroid (Zhang et al., 2013) identifies explicit and implicit permission use points inside applications and generates profiles of the applications based of these points. DROIDRANGER (Zhou et al., 2012a) implements a combination of a permission-based behavioral foot printing scheme to detect samples of already known malware families and a heuristic-based filtering scheme to detect unknown malicious applications. Some other methods such as (Peiravian and Zhu, 2013; Sahs and Khan, 2012) and (Burguera et al., 2011) use machine learning for analyzing different features of applications for malware detection.

6 CONCLUSION AND FUTURE WORK

Dynamic analysis tools usually cannot extract a complete and precise profile of applications' behaviors or cannot trigger or extract behaviors of specific components of applications separately. In this paper, we proposed a framework for instrumenting Intents in Android applications' Smali codes. Our instrumented Intents, trigger different components of applications automatically at runtime. We first showed that extracting profiles belong to some specific components of applications can simplify the analysis of applications' behaviors, and consequently facilitates finding suspicious behaviors. Then, further investigation using a dynamic malware analysis tool, named AppsPlayground, showed that code instrumentation (for forcing some components to be invoked) can facilitate detecting more malicious or suspicious behaviors in applications. We successfully applied our instrumentation framework to a data set of 6,187 malware applications. Experimental results showed that applying our code instrumentation framework on the applications, help exhibiting more data leakage such as Android ID in 79 more applications in comparison to using traditional malware analysis tools with no Intent instrumentation. However, our framework cannot trigger the native codes that are directly used in applications. In addition, some of the applications in our collected malwares do not produce manifest file in disassembling process of our framework.

REFERENCES

Bartel, A., Klein, J., Monperrus, M., and Le Traon, Y. (2014). Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. In *IEEE Transactions on Software Engineering*, volume 40, pages 617–632. IEEE.

Bugiel, S., Heuser, S., and Sadeghi, A.-R. (2013). Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 131–146.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM.

Chen, J., Chen, H., Bauman, E., Lin, Z., Zang, B., and Guan, H. (2015). You shouldn't collect my secrets: Thwarting sensitive keystroke leakage in mobile ime apps. In *Proceedings of 24th USENIX Security Symposium (USENIX Security 15)*, pages 657–690. USENIX Association.

Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM.

Corporation, I. D. (2014). Android market share reached 75% worldwide in q3 2012. http://techcrunch.com/2012/11/02/idc-androidmarket-share-reached-75-worldwidein-q3-2012 Access time: May 7,2013.

Desnos, A. (2014). Androguard-reverse engineering, malware and goodware analysis of android applications. https: code.google.com/p/androguard Access time: 2013, May.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *ACM Transactions on Computer Systems (TOCS)*, volume 32, pages 5:1–5:29. ACM.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011a). Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM.

Felt, A. P., Finifter, M., Chin, E., Hanna, S., and Wagner, D. (2011b). A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM.

Fuzz (2014). Fuzz. http://pages.cs.wisc.edu/bart/fuzz/ Access time: 2008, May.

Gilbert, P., Chun, B.-G., Cox, L. P., and Jung, J. (2011). Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM.

Google (2014). Apktool. https:code.google.compandroid-apktool Access time: 2016, May.

Grace, M. C., Zhou, Y., Wang, Z., and Jiang, X. (2012). Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, volume 14, page 19. Internet Society.

Hoffmann, J., Ussath, M., Holz, T., and Spreitzenbarth, M. (2013). Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM.

IDC (2014). Android and ios continue to dominate the worldwide smartphone market with android shipments just shy of 800 million in 2013. http://www.idc.com/getdoc.jsp Access time: February 2014.

ISBX (2014). An online data set of malwares. http://www.unb.ca/research/iscx/dataset/ Access time: 2015, May.

Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. (2012). 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*, pages 3–14. ACM.

Karami, M., Elsabagh, M., Najafiborazjani, P., and Stavrou, A. (2013). Behavioral analysis of android applications using automated instrumentation. In *Proceedings of the IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C)*, pages 182–187. IEEE.

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2015). Apkcombiner: Combining multiple android apps to support inter-app analysis. In *Proceedings of the IFIP International Information Security Conference*, pages 513–527. Springer.

Peiravian, N. and Zhu, X. (2013). Machine learning for android malware detection using permission and api calls. In *Proceedings of the IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 300–305. IEEE.

Pooryousef, S. and Amini, M. (2016). Fine-grained access control for hybrid mobile applications in android using restricted paths. In *Information Security and Cryptology (ISCISC), 2016 13th International Iranian Society of Cryptology Conference on*, pages 85–90. IEEE.

Rastogi, V., Chen, Y., and Enck, W. (2013). Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the 3rd ACM conference on Data and application security and privacy*, pages 209–220. ACM.

Rastogi, V., Chen, Y., and Jiang, X. (2014). Catch me if you can: Evaluating android anti-malware against transformation attacks. In *IEEE Transactions on Information Forensics and Security*, volume 9, pages 99–108. IEEE.

Report, M. T. (2014). Third quarter 2012. http://www.mcafee.com/ca/resources/reports/rp-quarterly-threat-q3-2012.pdf Access time: May 7, 2013.

Sahs, J. and Khan, L. (2012). A machine learning approach to android malware detection. In *Proceedings of the Intelligence and Security Informatics Conference (EISIC), 2012 European*, pages 141–147. IEEE.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society.

VirusShare (2014). An online data set of malwares. https:virusshare.com Access time: 2015, May.

Wong, M. Y. and Lie, D. (2016). Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, pages 1–15. The Internet Society.

Xu, R., Saïdi, H., and Anderson, R. (2012). Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552. USENIX Association.

Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., and Enck, W. (2015). Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313. IEEE.

Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. (2013). Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM.

Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. (2012a). Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the 2nd ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM.

Zheng, M., Lee, P. P., and Lui, J. C. (2012b). Adam: an automatic and extensible platform to stress test android anti-virus systems. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101. Springer.

Zhou, W., Zhang, X., and Jiang, X. (2013). Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12. ACM.

Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. (2012a). 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 (NDSS)*, volume 25, pages 50–52.

Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. (2012b). 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 (NDSS)*, volume 25, pages 50–52.