DroidTest: Testing Android Applications for Leakage of Private Information

Sarker T. Ahmed Rumee and Donggang Liu

Department of Computer Science and Engineering,
University of Texas at Arlington,
Arlington, TX 76019, USA
sarker.ahmedrumee@mavs.uta.edu,dliu@uta.edu

Abstract. Smartphones have become a basic necessity in recent years, and a large portion of users are using them for storing private data such as personal contacts and performing sensitive operations such as financial transactions. As a result, there is a high incentive for attackers to compromise these devices. Researchers have also found that there are indeed many malicious applications on official or unofficial Android markets, and a large fraction of them steal private user data once they are installed on smartphones. In this paper, we propose a novel method to test Android applications for the leakage of private data. Our method reuses existing test cases, produced either manually or automatically, and converts each of them into a set of new *correlated* test cases. The property of these correlated test cases is such that- they will trigger the same result in our system if there is no leakage of private data. As a result, the leakage of information can be detected if we observe different outputs from executions under correlated inputs. We have evaluated our system on an Android malware dataset and the top 50 free applications on official Android market. The result shows that our tool can effectively and efficiently detect leakage of private data.

1 Introduction

The use of smartphones are increasing day by day. According to a recent survey [7], over 50 percent of the US mobile users own smartphones. Nowadays, a smartphone can be used for almost all the purposes a normal user can think of, such as web browsing, online chatting, email, social networking, gaming, audio and video conferences.

To meet the wide range of users' need, millions of applications have been developed and available in various online application stores, such as - Apple Appstore, Google Playstore for Android etc. While these online markets make it convenient for users to customize their smartphone systems for their needs, attackers have also become interested in them for compromising users' systems. Attackers may upload malicious applications and then compromise the systems that download and install those applications. For example, these malicious applications may steal private data or send SMS messages to premium numbers without user's knowledge. In addition, pirated versions of popular smartphone applications are commonly available at unofficial market places too. Attackers often embed malicious code in these pirated versions and can compromise a device when installed.

The above information leakage problem has already been studied in the literature. Here we restrict our study to Android platform. Existing solutions include static analysis [16] and dynamic monitoring [10]. Static analysis scans byte code or source code to find paths that may leak information. However, it does not execute the program and lacks program dynamic information. As a result, it suffers from high false positive rate. On the other hand, dynamic analysis monitors the execution of an application on a modified Android platform. However, the modification of Android system is not always practical from the end user perspective. In addition, dynamic monitoring often increases the overhead at the user side significantly.

In this paper, we propose to test Android applications before placing them on the market. The testing can be done by the market owner or a third party other than application developers by uploading the applications to a offline server running the proposed system. Conceptually, the first step of testing is to generate test cases to explore different program paths, and the second step is to examine each program path to see if it is possible to leak any sensitive information. The first step can be done by using GUI-based test case generation [19] or concolic testing like DART and CUTE [17, 21] as long as they are ported to Android platform. Here, we assume that there exists a set of test cases produced either manually or automatically by various existing methods, and our focus is on the second step, i.e., to test if the execution under a given test input is leaking sensitive information. One method is to monitor the outgoing packet of an application and check the content to see if it is leaking something similar to a phone number, a credit card number, etc. However, when the attacker obfuscates the packet, e.g., by encryption, then this method will fail. Another natural way to solve this problem is to use taint analysis, e.g., if the outgoing packet is tainted by private data, then we can say that this application is leaking information. For example, TaintDroid [10] is built on this simple idea. However, the problem with such idea is that, the application needs to be instrumented and monitored, which requires access to the source code. While the source code can be obtained by reverse engineering, a common trend right now is that many application developers obfuscate their code to make it difficult for source code-based program analysis.

To remove the need for program source code and relieve user side from expensive monitoring, in this paper we propose *DroidTest*, a server side black-box testing method; we only work on the input and the output of the application under test. In our system, each existing test case is used to produce a set of correlated test inputs. These correlated inputs are crafted in such a way that, the output will stay the same if there is no leakage of private information. In fact, the only difference between two correlated test cases is the input that includes sensitive information, e.g., current location, personal pictures, etc. As a result, if we observe different outputs generated by a set of correlated test cases, then we can say that the difference is due to the difference in the sensitive inputs. The benefit of DroidTest are as follows. First, it does not require access to the source code. As a result, obfuscating source code does not stop us from detecting the leakage of information. Second, it can detect both explicit and implicit data leakage.

We have evaluated our technique on two datasets. The first dataset includes applications that are known to contain malicious code leaking information; the second dataset includes the top 50 free applications from Android market. The result shows that we can successfully detect the information leakage of the applications in the first dataset. In addition, we have also found a good number of applications in the second dataset that are also leaking some private information without user's knowledge.

The remainder of the paper is organized as follows. Section 3 provides a necessary background on the Android security mechanism along with the assumptions made. Section 2 reviews related work on Android security. Section 4 describes the high level overview of the proposed approach. Section 5 provides detailed description of the proposed method. Section 6 contains the experimental results and findings. The last section concludes the paper and discusses some possible future directions.

2 Related Work

Security and privacy of smartphone applications is a field of active research in the recent years. Among them, stealing sensitive information from smartphones is one of the major threats [12].

Static and dynamic analysis techniques have been applied to track the suspicious behavior of applications. PiOS [9] applies static analysis on iOS apps to detect possible privacy leak. In particular, it first constructs the control flow graph of an iOS application, then checks whether there is an execution path from the nodes that access privacy source to the nodes corresponding to network operations. If such a path exists, it is considered a potential for information leakage. SCANDAL [20] also employs similar techniques to find the path between the sensitive sources of data and the network write operations in Android applications. However, as we mentioned, static analysis tools often suffer from high false positive rate.

TaintDroid [10] uses dynamic taint analysis to track data flow inside the Android operating system. It taints data from private information sources, and then checks the data leaving the system via network interface. It raises an alert when the outgoing data include tainted information. A major limitation of this system is it only looks for explicit information flow; thus cannot detect data leakage through implicit flow of information.

ScanDroid [15] extracts security specifications from the manifests that accompany Android applications, and then checks whether the information flow is consistent with those specifications. However, the analysis results in [14] and [11] show that such specification may not be always sufficient. Malicious applications can easily bypass them and still leak data.

TISSA [23] gives users detailed control over an application's access to a few selected private data (phone identity, location, contacts, and the call log) by letting the user decide whether the application can see the true data or some mock data. MockDroid [8] allows users to mock the access from an untrusted application to particular resources at runtime. AppFence [18] replaces sensitive information with shadow data. While these methods stop the leakage of data, they also impact legitimate applications that are allowed to access these data.

3 Background and Assumptions

3.1 Android Permission Scheme

Android governs the access to resources such as internet, SMS, Contacts etc. through its permission mechanism. At the install time, each application declares its required permissions such as - internet access, GPS data access, read phone state etc. There is no way to selectively accept or reject these permission requests. A user must accept all of those to install the application, otherwise abort the procedure. This was done with a view to informing users about which specific resources are accessed by each application.

But in practice, users pay little attention to these permission requests [13] and unnecessary permissions are often granted to applications. For example, an email application needs to send and receive emails. In Android, such an application must acquire the full internet access permission, which is more than necessary and can be exploited.

3.2 Sensitive Information Source and Sink

To detect the leakage of sensitive data, the first task is to define the set of private or sensitive information sources. Some commonly regarded source of private data are: Device id (IMEI), Subscriber Id (IMSI), Phone Number, Location and Contact information, User account information, SMS/MMS messages etc. However, the nature of application and context information are also important to decide whether a particular piece of data is sensitive or not. For example, location information sent by the popular Android application *Gasbuddy* cannot be termed as an instance of privacy violation, as it is the part of the its task and user is also aware of that.

Information sink means the ways data can leave the device, which includes - Network access, Outgoing SMS, MMS etc. Here, we check whether the information going through the sink contains some private data.

3.3 Adversary Model

In this paper, we assume that the attacker fully controls the development of the application under test. In other words, he can embed any code he wants. We assume that the application has the permission to read some of the sensitive information sources and access Internet. The question for us is to see if such sensitive data is leaked through Internet. During the test, we also assume that except the application under test, all other system components, e.g, the Android OS and the Dalvik VM, are secure. If they are compromised, then the adversary can easily bypass our monitoring. This assumption is reasonable since the testing system is built specifically for the purpose of detecting information leakage. On the other hand, we cannot make such assumption if the detection is done at the user end. Finally, we also assume that the malicious application does not compromise the Dalvik VM or Android OS during the execution.

4 DroidTest Overview

The objective of DroidTest is to detect malicious activity in Android smartphone applications that send out privacy-sensitive information out through network interfaces. A straightforward idea to solve this problem is to use static analysis techniques. Basically, we scan the program source code or binary to see if the data write operation is reachable from the read operation performed on sensitive data. However, as we mentioned, these techniques suffer from high false positive rate. Dynamic analysis does not suffer from the abundance of false positives. However, they involve significant modification of the Android operating system and also require user devices to do the expensive monitoring.

DroidTest is a black-box testing method; the system only works on the input and the output of the application under test. As mentioned, DroidTest uses existing test cases. Given a test case, it will examine the path triggered by such test case to see if there is any information leakage. The main idea of DroidTest is based on the following observation: given a deterministic function $f(x)$ ($x \in \mathcal{X}$), the result of this function does not leak any information about x if for all x_1 and x_2 we have $f(x_1) = f(x_2)$. In other words, given $f(x)$, the probability distribution of x is a uniform distribution over $\mathcal{X}$. As a result, if we test $f(\cdot)$ using two inputs $\{x_1, x_2\}$ and find that $f(x_1) \neq f(x_2)$, then it is for sure that function $f(x)$ is leaking some information about the input x. This basically means that if we fix all inputs to an application except those that are considered as sensitive, then the content of outgoing traffic should not change if there is no leakage of sensitive data.

In the high level, DroidTest works as follows. For each existing test case, we convert it into multiple *correlated* test cases. We consider these test cases as correlated since they only differ in the inputs from sensitive sources, e.g., current location, personal pictures, etc. In this paper, we only produce two correlated test cases from each of the existing test cases since we found that two is often enough for tracking the information leakage as long as the private data part is randomly mutated. Once we have the two correlated test cases, we run the application twice and monitor the outgoing traffic. If we see any difference in the outgoing traffic, then we can say that such difference is due to the difference in the sensitive data. This basically means, some private information is leaked by this application.

5 Implementation

Figure. 1 depicts a conceptual view of the DroidTest system. DroidTest has three major components: *test case generator*, *test case executor* and *kernel log collector*. The test case generator converts each existing test case into a pair of correlated test cases that only differ in the private data part; the test case executor runs the application once for each of the two correlated test cases and analyze if any private information is leaked through network interfaces; the kernel log collector basically monitors the network interface and collects the outgoing packet data. In the rest of this section, we first describe the necessary settings made in the Android operating system and then discuss the three major components of the proposed system in detail.

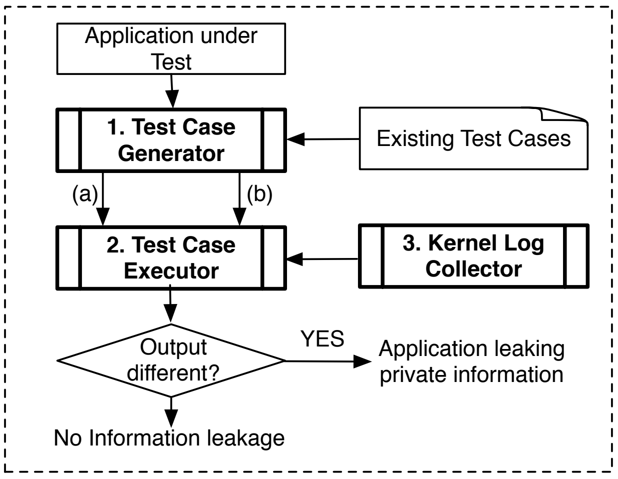

Fig. 1. High-level view of the proposed system. (a) and (b) are a pair of correlated test cases produced from an existing test case.

5.1 Test Environment Setup

DroidTest systm is a modified Android Operating system based on version 2.2, popularly known as Froyo. Applications are loaded and run in the emulator built from this modified Android system.

At first, We locate all APIs that are required to read phone specific information, find location data, contact lists and read incoming SMS contents etc. Then custom source code is inserted in these methods to change the return values with randomly generated values. As a result, different invocation of these methods will return different values. This setting helps us to create co-related test cases with variations in the sensitive data.

For example, to get the *Device ID* or *IMEI number*, one has to call the *getDeviceId()* method of the *Android.telephony.TelePhonyManager* class. It internally uses the *getDeviceId()* method of the class *PhoneSubInfo* under the package *com.Android.internal.telephony*. We modified the *getDeviceId()* of *PhoneSubInfo* class to read a mock IMEI number (randomly generated) and return it to the application.

In addition to the modification at the information sources, we also try to fix program inputs such as system time, random number generators, and environment variables to constant values. In the current implementation, we only modified the *java.util.Random* and *java.lang.System* classes. It aids to offset variation in program output for multiple runs due to the factors other than the user provided inputs.

5.2 Test Case Generator

As said earlier, we have manually generated test cases for the applications under test. Each application is run for some time (around 5 minutes) in the emulator. We manually exercise its basic features, e.g., try to navigate to all the screens, click the available buttons and links etc. For each such activity, we check whether the application has performed any sort of network activity or sent any text message. If so, the sequence of events that led up to this point from the application home screen(entry point) is taken as one candidate test case for this application. Test cases not resulting in any outgoing data are dropped.

5.3 Test Case Executor

This component will take a pair of correlated test cases and execute the application once for each of them. The executor takes two command line arguments to run - *the application name, the name of the file containing the test cases.* It then converts these test cases to Junit [3] test scripts. Within the Junit framework, individual test methods are written using the APIs of Robotium [6], which automatically passes the GUI events (actions in the test case) to the Android emulator.

Once the execution is done for each of the correlated test cases, we retrieve and compare the the monitoring results from the kernel logs. If they are different, then we report an alarm since some sensitive information is leaked through the internet by the application under test.

5.4 Kernel Log Collector

The last key component of our system is a simple Loadable Kernel Module, that intercepts the outgoing network packets. To do that, we hook the System Call Table of the Android operating system. This is done by placing a custom system call table in the same address as the original system call table.

Android operating system uses system call - *sys_sendto* to send data to network. We intercept this system call and log its parameters(packet data) to the kernel logs.

6 Experiment and Results

6.1 Scope of Sensitive Information Source and Sink

We have discussed about the sensitive information sources earlier. For this study, we only consider the ones listed in Table. 1. As the sink of sensitive information, we consider the *network interfaces.* However, Our system can be easily extended to work with additional sources and sinks of private data.

Table 1. Information sources considered private

IMEI number, IMSI number, Android OS Version, Phone Number, Phone Model, Contacts, Location information, Incoming SMS contents

6.2 Datasets

We consider two datasets. The first dataset [4] includes the Android applications that are known to contain malwares. The second data set includes applications from the Official Android Marketplace [5]. Whether these applications leak private data or not is unknown beforehand.

Data Set 1: Malicious Applications The malware database [4] has total 1260 samples under 45 different malware families. Among them, 28 were found to leak users private data by [2,22]. So, we also restrict our study to these malware samples.

From these samples, we further discarded any application having one or more of the following properties. Because, in that case an application do not have the capability to send data outside through internet. This further reduces our malware data set to *222 samples from 20 malware families.*

1. Perform no internet activity during its execution.
2. Conditions to activate or complete the malicious activity is no longer present. For example, the web server where the stolen information is sent is down, or the emulator has to send SMS to premium rate number, which is not possible in an emulated environment etc.

Data Set 2: Android Marketplace Applications From the Android market place, We select top 50 (as of March 2013) free applications to perfrom the testing. While collecting applications, we check whether they require certain permissions: *Full Internet access* and at least one of - *Read phone specific information, Contact data, and Location information.* Applications not having such permissions are discarded from further analysis. Table. 3 lists the name of these applications along with their leaked information.

6.3 Results

Experiment Results of Malware Dataset Table. 2 shows the result of experiment, that includes - the name of the malware family, number of samples in the test suite and the leaked information type. For each of these samples, we could successfully detect its known data leaking behavior. Hence, the detection rate for the proposed tool was found to be 100%.

Experiment Results of Android Market Place Applications Next, we apply the proposed method to test the *top 50 free applications* from the Android market place [5]. Table. 3 lists the name of these applications alongside the type of information leaked. The value *Nil* indicates that we didn't find any leakage of information during the testing.

From the experiment, we see that some highly rated applications can also leak private data without user's consent. The permission requests presented by these applications had no direct mention about their sending of private information to third party. We found that, *36*

Table 2. Android Malwares and Types of Leaked Information

Malware Samples		
Malware Family	**Number of Samples**	**Leaked Information**
BeanBot	4	IMEI, IMSI, Phone Number Android OS Version
BgServ	4	IMEI, Phone Number, Android OS Version
DroidDelux	1	IMSI, Phone Model, Android OS version
DroidDreamLight	30	IMEI, IMSI
DroidKungFu1	16	IMEI, Android OS version,Phone model
DroidKungFu2	8	IMEI, Android OS version,Phone model
DroidKungFuSapp	3	IMEI, Android OS version,Phone model
DroidKungFuUpdate	1	IMEI, Phone Number, Android OS Version
GoldFream	34	Incoming SMS, IMEI
Gone60	9	Contacts, SMS
LoveTrap	1	IMSI, GeoLocation
Plankton	11	IMEI
PjApps	47	IMEI, Phone Number
RougeSP Push	9	IMEI, Phone Number, Android OS version
SMSReplicator	1	Incoming SMS
SndApps	10	IMEI, Phone Number
WalkinWat	1	IMEI, Phone Number, Phone Model and Android OS Version
YZHC	22	IMEI, Incoming SMS
zHash	11	IMEI, IMSI
Zitmo	1	IMEI, Incoming SMS
Total	222	

out of 50 applications (72%) send one or more sensitive data to outside, mostly to advertising servers. This number is quite disturbing taking into the fact that, these applications are very much popular and widely used [5].

6.4 Discussions and Findings

We tested the applications by creating co-related pair of test cases for the program paths explored manually. So, it may miss some instances of true positives, as the set of test cases is not comprehensive. Detection rates of malicious applications reported above are also based on the generated test cases only. DroidTest can theoretically produce false positives, because we fixed only random numbers and system time to constant, and there may be other factors that can change program output without changing the sensitive input. But for the

Table 3. Top 50 Free Applications from Android Market and Leaked Information

Application Names	**Leaked Information**
TuneInRadio	Location
Crackle Movie	IMEI,IMSI, Android OS version
Brightest LED FlashLight,	Location Android OS version, Phone model, Build Version
Tiny Flashlight, Zillo, Instagram, JetPack Joyride, AskFM, FruitNinja, Lines, Simpson, Drag Racing, Vector, PicsArt, Ant Smasher, Google Translate	Nil
Scramble Free, iHeartRadio, HelloKitty	IMEI, Android OS version, Phone model, Build Version
Craiglist Mobile, Inkpad, MeetMe, Tagged, Shoot Bubble, AccuWeather	Android OS version, Phone model, Build version
Logo quiz, Cut the Rope, Amazon MP3,	Android OS version Phone model, Build version
Speed Test	Current Location, IMEI Build version
Sound Hound	IMSI
Sudoku	Current Location, Phone model Build version
Alarm Clock Extreme	Phone Model
DH Texas Poker, Words with Friends, ESPN score center, Restaurant Story, Angry Gran Run, Castle Defence, Zombie Frontier, Mind Game	IMEI, Android OS version, Phone model
Imdb, iFunny, GoWeatherEX, Texas HoldEm Poker, My Mixtapez Fashion Story, 4 Pics 1 Word, Tetris,	Android OS version, Phone model
Hulu plus	Android OS version

applications we tested, every instances of variation in output could be traced back to change in sensitive input sources, thus no false positives were produced.

From Table. 2 and Table. 3, we can see the type of information leaked by various applications. The most commonly leaked information is the unique device id or the IMEI number by both type of applications. If we closely observe the results of analysis of these applications, we can easily see that, applications from the malware data set leaks IMEI, IMSI, Phone number more than the other sensitive data. On the other hand, the mostly leaked information by the Android market place applications include Android operating system version, Phone model which are definitely not as sensitive as the device id (IMEI), subscriber id (IMSI) and the phone number. So, user must avoid downloading applications from the untrusted third party market places. The chance of leaking sensitive data are much higher in case of the unofficial market place applications, because unofficial market places apply hardly any security check of the applications.

Recently, with the release of latest Android 4.2 (JellyBean), google announced a new and exciting security feature called the *"Application Verification Service"*.Performance of this service in detecting malicious applications has been studied in [1]. Based on that, Table 4 shows the comparison of the results found by *DroidTest* and *Application Verfication Service.* It clearly shows that, the proposed system performed way better than App Verification Service.

7 Conclusion

The growing popularity of smartphones has led to the rising threats from malicious mobile applications. In this paper, we have demonstrated the need for testing Android applications before they are put into the market place. It comes from the fact that, many popular applications found in the official and unofficial Android market places leak private data to some third parties without proper user consent. To test mobile applications for data stealing behavior, we have proposed the DroidTest system and described its architecture, operation and evaluation through the testing of two datasets. The experiment results shows its effectiveness in detecting private information leakage.

Currently our system is an offline testing tool. In future, we plan to move the monitoring to the device to notify user about information leakage dynamically. We also plan to include more information sources in the sensitive input list. This will make DroidTest better equipped to detect zero day smartphone malwares. Currently, we manually generate the test cases for the applications under test. To

Table 4. DroidTest vs Google App Verfication Service

Malware Samples and Detection Results		
Malware Family	**Detected Samples**	
	DroidTest	**App Verification Service**
BeanBot	4	0
BgServ	4	0
DroidDelux	1	0
DroidDreamLight	30	18
DroidKungFu1	16	8
DroidKungFu2	8	9
DroidKungFuSapp	3	0
DroidKungFuUpdate	1	0
GoldFream	34	6
Gone60	9	0
LoveTrap	1	0
Plankton	11	2
PjApps	47	8
RougeSP Push	9	0
SMSReplicator	1	0
SndApps	10	0
WalkinWat	1	0
YZHC	22	3
zHash	9	1
Zitmo	1	0
Total	222	56

make our system more robust, we also plan to generate meaningful test cases for Android applications automatically in future.

8 Acknowledgments

The authors would like to thank Professor Xuxian Jiang and his research group from North Carolina State University for sharing us with the android malware data set.

References

1. Analysis of appverification tool from google. `http://www.csc.ncsu.edu/faculty/jiang/appverify/`.
2. Contagio mobile malware mini dump. `http://contagiominidump.blogspot.com/`.
3. Junit. `http://junit.sourceforge.net/`.
4. Malware data set. `http://www.malgenomeproject.org/policy.html`.

5. Official android marketplace: Google play. `https://play.google.com/`.
6. Robotium. `http://code.google.com/p/robotium/`.
7. Survey on smartphone users. `http://www.engadget.com/2012/05/07/nielsen-smartphone-share-march-2012/`.
8. A. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011.
9. M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.
10. W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.
11. A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.
12. A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. 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, 2011.
13. A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.
14. A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proc. of the USENIX Conference on Web Application Development*, 2011.
15. A. Fuchs, A. Chaudhuri, and J. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.
16. C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. *Trust and Trustworthy Computing*, pages 291–307, 2012.
17. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
18. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.
19. C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83. ACM, 2011.
20. J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. Scandal: Static analyzer for detecting privacy leaks in android applications.
21. K. Sen, D. Marinov, and G. Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.
22. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.
23. Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011.