Android Privacy Leak Detection via Dynamic Taint Analysis

Jedidiah McClurg Jonathan Friedman William Ng

{jrmcclurg, jonathanfriedman2013, willng}@u.northwestern.edu

Dept. of EECS
Northwestern University
Evanston, IL 60202

ABSTRACT

Android is a popular Linux-based smartphone operating system designed by Google. One of the primary adantages of Android is its relatively high level of security, centered on Unix processes and an explicit permissions system. Unfortunately, Android devices are still vulnerable to several types of attacks, a particularly concerning one being privacy leaks. Since devices store a large amount of sensitive information, it is important that this information not be leaked via internet connections or SMS messaging. We propose an integrated system to detect such privacy leaks via dynamic taint analysis. We have built a PC-based Java application to instrument apps with taint propagation functionality, and a proof-of-concept Android app to demonstrate that this system could conceivably be run on the device. The results of running several instrumented apps show that the system is effective at detecting privacy leakage with relatively minimal overhead.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; D.4.6 [**Operating Systems**]: Security and Protection—*invasive software*; D.3.4 [**Programming Languages**]: Processors—*parsing, code generation*

Keywords

Android, Dalvik, Taint Propagation, Dynamic Analysis

1. INTRODUCTION

Smartphones are becoming more pervasive in our lives, with the global number of users increasing at a tremendous speed. Smartphone shipments have increased 42% between 3rd Quarter 2010 and 3rd Quarter 2011, according to research data from Gartner. Smartphones are powerful and versatile in their functions and smartphone users are integrating the devices into their daily lives. For example, one common use is to store and manage contact information. In this regard, it can store not only phone numbers, but also geographical addresses, email addresses, and profile pictures of friends and family members.

The functionality of smartphones is further expanded by enabling third-party developer applications or improved support of web applications. Users can now use their smartphones to manage bank accounts or provide navigational directions. It is slowly taking over some of the traditional roles of the personal computer by providing users with more convenience and immediacy. However, the expanding of functionality and increase in versatility has dramatically increased the risk of malicious device attacks, primarily because they contain more valuable resources, but also because there are an increasing number of attack methods.

Android is designed to have a robust set of built-in security mechanisms [14], but experience and research has shown that this system is often insufficient to protect against some types of attacks [15]. One of the more damaging of these is stealing personal sensitive information stored in the device. It is clear that smartphones can contain large amount of sensitive information, e.g. personal information of all contacts, content of all past messages and emails, or account credentials of car rental accounts or bank accounts. In addition to these direct sources of information, attackers may be interested in more subtle information available from the device, e.g. the current geographical location identified by GPS or base station triangulation, audio recording of phone conversation or the immediate surrounding, or internet traffic data of the browser or any application.

This problem is becoming increasingly serious as malicious apps continue to crop up, sometimes even appearing in official marketplaces [20]. Users who frequent third party app stores may encounter official apps that have been repackaged with malicious intent [18]. Unfortunately, commonly-available anti-malware apps typically have a low detection rate in these types of instances [19]. Furthermore, users are often unable or unwilling to manually keep track of how all the sensitive information is being used by third party applications. In some cases this requires a very high level of technical knowledge and a good amount of time for one to manage it. Our system seeks to fill this gap by using dynamic taint analysis [13] to keep track of how the sensitive information is being used inside an Android app. The system is written entirely in Java, and so could easily be compiled/deployed to run as an Android app. The system

decompiles a given questionable Android app, inserts taint tracking functionality into the code, and recompiles it into a behaviorally-equivalent app. When this new instrumented app is run, it produces alerts whenever sensitive information leaves the device, e.g. over the internet or through SMS.

The rest of this paper is structured as follows. In Section 2 we compare our work with related work. In Section 3 we present our system and taint-propagation/analysis techniques. In Section 4 we evaluate the performance of our system. In Section 5 we present the taint tracking results. Finally in Section 6-7 we discuss future work and conclude.

2. RELATED WORK

There are many different techniques for Android malware detection/defense. Straightforward approaches include utilizing Android permissions and encrypting sensitive content [12], but these approaches may have poor coverage. Other approaches seek to view the app as a box and barrage it with inputs in an attempt to determine its behavior [11]. More complex approaches use techniques such as symbolic simulation [1] to analyze application bytecode. In contrast to these, our system uses techniques from the area of *dynamic analysis*. This approach has the advantage of being very fast, since the analysis is performed *while* the program is executing.

Other applications of dynamic analysis have included dynamic monitoring of permissions usage using runtime-level systems [3], and real-time permission revocation [2]. As far as we know, there has been no work on the feasibility of dynamic taint analysis for privacy protection that is both user friendly and agnostic to special permissions. TaintDroid [6] is conceptually similar to our system in that it is a real-time dynamic taint tracker for Android with the explicit goal of protecting users from information leakage. However, TaintDroid requires the user to root or jailbreak their phone to run it. We believe that regardless of the capabilities of the TaintDroid system, it is unrealistic to ask for special permissions and rooting or jailbreaking of phones to achieve security. Our system prototype can achieve similar types of detection while it can still be feasibly run on an Android device with standard permissions. To install the TaintDroid system, the user is required to have an unlocked boot loader and a development computer that can build the android source code. In many cases, unlocking a boot loader will void the user's warranty, and compiling the android source code is infeasible for the average Android user. Furthermore, installing TaintDroid requires the user to complete eight different steps on the command line, some of which include compilation. This limits the applications of this system to computer programmers or people who are very computer-savvy. Although this segment of the population does need some security protection, these users are presumably the least likely to download suspicious applications containing information leakage or malware. Thus, our system focuses on users with only a basic level of computer skills, and provides them with a simple tool to protect themselves from information leaks.

In contrast to dynamic analysis, others have approached Android security problems from the standpoint of static analysis. These approaches typically involve developing a formal semantics [4] for Android programs and then using fixpoint operations (or over-approximations thereof) to show that the semantics imply a permissions violation [10] [9] or privacy leak [7], or to compute a helpful invariant regarding the code [16]. A disadvantage of this approach is that the analysis can be very slow, especially for programs with many execution paths.

Some of these static analysis tools also require implementation at the OS-level [8] or Android runtime level, again running into the usability problem we mentioned. For example, PiOS is a static analysis information leak tracker for the iOS platform [5] with similar requirements to TaintDroid. Although the PiOS system is not intended for the user to install as a defense against malware, it is similarly difficult to install if the source were to be released. Even if the source was not released, the system depends on the phone being jail broken, which as we have mentioned, is not always a feasible approach. This leaves applications unchecked when Apple presents update to defeat the current jail breaking scheme.

Because we do not modify the android runtime like some of these other systems, we are currently forced to treat android system calls as a black box in which we assume always propagate taint values. This means that we are over-approximating the actual program semantics, which could lead to false positives. But it is possible that in the future we could adopt a hybrid static/dynamic approach for this case. Since android is open source, we have the ability to decompile the android system libraries (which we have already tested), and statically analyze them to obtain the taint propagation semantics for all android library functions. We have designed our system so that in the future we can put these statically-determined semantics into our taint propagation library, resulting in an even more accurate taint propagation system.

3. DETECTING PRIVACY LEAKS

We have implemented a PC-based prototype of our system in Java. We have verified that the Java-powered disassembler/assembler used by our system is able to run via an Android app, but due to time constraints, we decided to do development/testing on the PC. Conceivably the entire system is fully Android-compatible, and could thus be deployed as a single Android app, but we leave this for future work.

3.1 System Overview

The overall goal of our system is to instrument the app under consideration with taint propagation functionality which will cause the taints to be set/propagated as the app runs on the Dalvik VM of the device. The core of this functionality is implemented in the highlighted boxes of Figure 1.

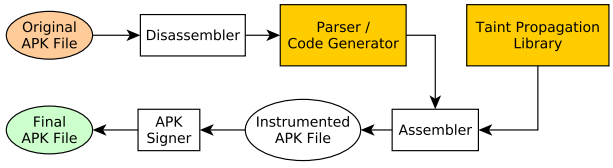

Figure 1: System Architecture

First, the app under consideration is scanned for its APK file. This file is then disassembled into its component Dalvik assembly files. After this, our system parses the assembly code, and inserts the appropriate taint propagation assignments or calls into the taint propagation library corresponding to each instruction. The resulting instrumented assembly code is assembled together with the taint propagation library into a new APK file. Finally, the instrumented APK file is signed using a user-specified public/private key pair.

This final APK file can then be executed as a normal app on the Android device. The taint propagation library will track the taints and inform us of privacy leaks via the device log.

3.2 App Decompilation/Recompilation

Apktool[1] is an APK file decoder/encoder which essentially functions as a frontend for the baksmali/smali[2] (dis)assembler. To generate assembly code from an Android app, we call `apktool decode` which extracts the manifest information and assembly code into a folder. The code in this folder can then be modified/instrumented, and then we call `apktool build` to reassemble everything into a new APK file.

During the decompilation, an important issue arises in regards to the parameter registers passed to methods, due to the fact that our code instrumentation must use new temporary local registers. In the Dalvik VM, a method's k parameters correspond to the last k registers in its set of locals. Normally, the baksmali decompiler uses the names $p_0, p_1, ..., p_{k-1}$ for these k registers, meaning that if the specified number n of local registers increases, p_i will now correspond to a different local register v_j. This means that we cannot simply increase the local register count for a method without causing the p_i to refer to incorrect locations (and typically causing the app to crash). We solve this problem by running `baksmali -no-parameter-registers` to force the assembly code to use the real v_j register names for all registers, including the parameters. Then, we can increase the local register count by m, if we also move the final k registers (parameters) to the location where the uninstrumented code was expecting them. That is, we must do move instructions `move` v_{j-m}`,` v_j where the v_j range over the last k local registers. After this, we are free to use the last m local registers however we want.

Recompilation takes a similar path to decompilation, this time using the command `apktool build` to package the manifest info and assemble the modified code back into a new APK file.

3.3 Code Instrumentation

To instrument the code with taint analysis information, we have developed a Java program which scans the decompiled APK folder for all files. It then iterates through these files and performs the following steps on each one:

1. If the file is not an assembly file, leave it unmodified
2. Otherwise if the file is an assembly file, iterate through each line (instruction) and append the assembly code which corresponds to the taint propagation semantics for that instruction.

To consolidate the taint propagation semantics into one place, each handled instruction is instrumented by a call into the taint propagation library which passes the instruction's parameters as parameters in the method call. For example, an `sget-object` v_j`, SomeField` instruction is instrumented by subsequently calling the Taint.updateTaintSgetObject(v_j, "SomeField") method. The taint propagation library then updates the taint values of the parameters accordingly.

Currently, we instrument two types of instructions in this way, specifically `sget` and `invoke`. Since this is only a subset of the full Dalvik instruction set[3], our system may give false positives. It is straightforward to instrument the remainder of the instructions, and we leave this as future work.

3.4 Taint Propagation

The taint propagation library has three functions:

1. Define the sensitive fields which correspond to taint sources
2. Keep track of taint information for live objects in the program
3. Generate an alert when tainted information leaves the device

The first function is implemented as a lookup table of field names. When field get instructions (such as `sget`) are received, the field name is looked up in the taint source table. If it exists, we will know the taint value for the recipient of the field's data.

The second function involves keeping a thread-safe global store of taint information. This is basically a hashtable which maps objects to integers, and thread safety is ensured by using `synchronized` methods for the table access functionality. This function also requires specification of the taint semantics for each instrumented instruction. The taint library handles this by providing an "update" method call for each type of handled instruction. One important exception is method calls into the Android runtime (i.e. code which has not been instrumented by our method). For this, the taint library uses the following (over-approximating) taint semantics:

$$\forall j, T(p_j) = \cup T(p_i)$$

where the p_i range over all method parameters. That is, each parameter acquires the union of the taints of all other parameters.

The third function is implemented by checking the method names on `invoke` instructions against a list of leak destinations. If the method is a leak destination, and any of the

[1]http://code.google.com/p/android-apktool/
[2]http://code.google.com/p/smali/
[3]http://source.android.com/tech/dalvik/dalvik-bytecode.html

parameters has a taint value, we consider this to be a privacy leak, and log the source and information regarding the leak destination.

3.5 Leak Detection

The leak detection is performed by the embedded taint propagation library as the instrumented code runs as a normal app on the device.

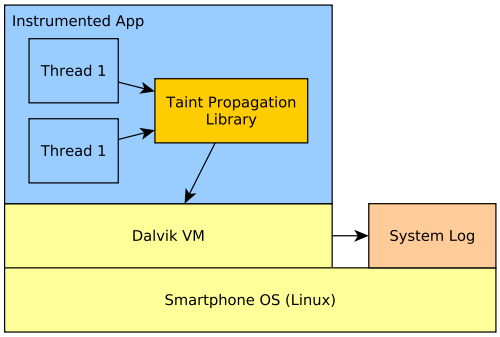

Figure 2: Leak Detection

The taint propagation library records flagged leaks to the system log, which we can view/record. In a production app, this information could be displayed as an alert to the user.

3.6 APK Signing

Android APK files must be signed before they can be loaded onto a device (or emulator). Recompilation of APK files via apktool destroys the original signature, so we must sign the generated APK file. SignAPK.jar is a simple third-party APK signer available in various places on the internet which comes with a key pair for easy signing of APK files[4]. There are other ways to do this which are endorsed by the Android documentation, but the advantage of the JAR file is that it is easy to use and can potentially be run on an Android device.

4. EVALUATION

We evaluated our system using a prototype Java implementation running on laptop. The machine specs were as follows: Ubuntu Linux (64bit), 3.8 GB RAM, 1.3 GHz Intel Core i3 processor (dual core, virtualizable to 4 cores). We installed the Android SDK, and created a new device in the Emulator based on the Google APIs, platform 4.0.3, level 15.

From colleagues, we obtained an unknown set of 110 free Android apps (APK format) from the Android marketplace. These apps were not expected to be malicious in any way.

4.1 File Processing Performance

First we ran our code instrumentation on each of the apps. We measured the time of each of the steps (decompilation, code instrumentation, recompilation, signing) in milliseconds using the Linux system `date` command. Our code instrumentation program printed information regarding the number of files/lines processed for each app. One app (`com.opera.mini.android`) failed to recompile correctly after our code instrumentation. Time did not permit us to investigate the source of the problem. The time results of this step for each of the remaining 109 apps are recorded in Figure 3. The overall processing time is shown clustered around the upper line, and the code-instrumentation time is shown by the lower cluster. It can be seen that the distribution is roughly linear with respect to the number of lines of processed Smali assembly code.

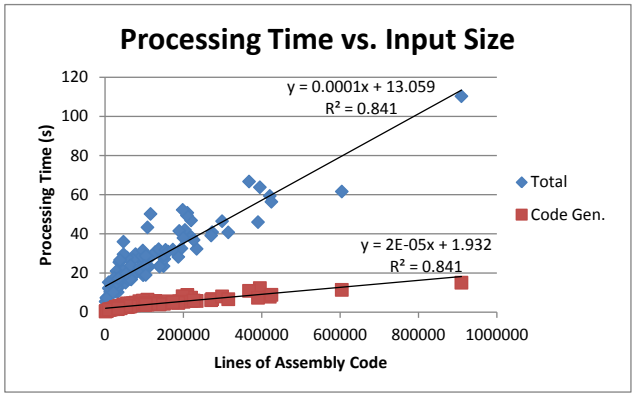

Figure 3: File Processing Performance

4.2 Taint Analysis Performance

Secondly, we selected 25 instrumented apps to run the taint analysis on. We were unable to analyze all 109 due to time constraints and the difficulty of automating interaction with the emulator/apps. For each of the selected apps, we installed it to the emulator using the `adb install` command, started output logging using the `adb logcat` command, and started the app via the `adb shell am start -W -S <package>/<activity>` command. This final command printed the number of milliseconds needed to start up the app, or timed out (presumably at 10s). After each instrumented app, we uninstalled it and did the same for the corresponding original app. We found that 2 of the 25 instrumented apps crashed upon startup. The relative times are shown in Figure 4, with points falling below the line showing slower instrumented code with respect to the original code. It can be seen that the results are clustered around the $y = x$ line, with roughly twice as many instances falling below the line as above. It is important to note that this is only a rough estimate of instrumented code performance, since it only examines a fraction of the program's execution, but still this hints that for the non-crashing examples, performance is reduced but acceptible.

4.3 Correct Operation

We checked that our code instrumentation preserves the proper app behavior by manually examining the `fm.last.android` app. Using an already-created Last.fm account, we were able to successfully log in using our credentials via the instrumented app, browse through the different settings, and listen to streaming music.

We are still investigating why one app failed to recompile correctly, and why two instrumented apps crashed in the emulator, but we suspect that this is due to a infrequent Dalvik construct being improperly used in instrumentation.

[4]http://android-dls.com/wiki/index.php?title=Generating_Keys

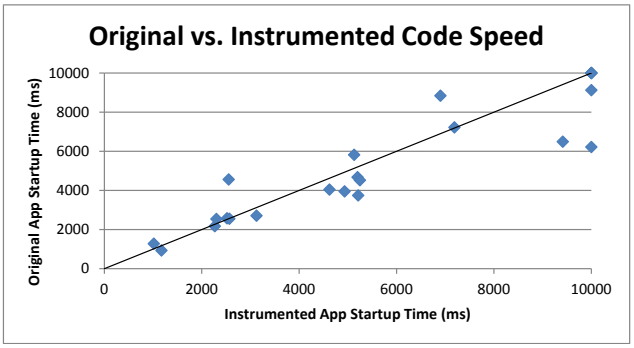

Figure 4: Taint Analysis Performance

5. RESULTS

Before running the taint analysis for the 25 aforementioned apps, we defined two static fields as tentative taint sources:

- `Landroid/os/Build;->MODEL:Ljava/lang/String;`
- `Landroid/os/Build$VERSION;->RELEASE:Ljava/lang/String;`

These correspond to the device model number and the Android version respectively. These fields are perhaps not the most sensitive information, but they do help identify the phone user in some way, and they do provide malicious apps with the ability to customize attacks [17]. Thus we use them as a basic example. We also define three leak destinations:

- `Ljava/net/URL;->openConnection()Ljava/net/URLConnection;` (if the connection `URL` is tainted, this is a leak)
- `Ljava/net/HttpURLConnection;->setRequestMethod(Ljava/lang/String;)V` (if the `String` is tainted, this is a leak)
- `Ljava/net/HttpURLConnection;->setRequestProperty(Ljava/lang/String;Ljava/lang/String;)V` (if the second `String` is tainted, this is a leak)

After running the analysis on the 25 apps, we examined the log files to find the following results. We found that 8 of the 25 apps contacted a website(s) during startup, and 6 of these involved a leak with respect to the above specifications.

In this instance, the majority of the communications flagged as leaks appear to be benign, since they involve a server owned by the maker of the app, but in two instances (com.cdroid.appinstaller and mobi.androidcloud.app.ptt.client), we can see that the device model number is being leaked to third-party ad services. This may be worrisome for some users, especially since the latter (TiKL Touch to Talk) describes their app as "Completely FREE. No hidden fee or ads."[5] Interestingly, the former (Cdroid App Installer) seems to have been removed from the Google marketplace since the time our set of apps was obtained.[6]

[5]https://play.google.com/store/apps/details?id=mobi.androidcloud.app.ptt.client&hl=en

Table 1: Detected Leaks of `MODEL`

App Name	MODEL
com.cdroid.appinstaller	http://mm.admob.com/
com.ebay.mobile	http://open.api.ebay.com/ http://mobilemotd.ebay.com/ http://rover.ebay.com/
com.microsoft.bing	http://t0.tiles.virtualearth.net/
com.tokasiki.android.voicerecorder	http://tokasiki.com
fm.last.android	http://cdn.last.fm/
mobi.androidcloud.app.ptt.client	http://data.mobclix.com/

Table 2: Detected Leaks of `RELEASE`

App Name	MODEL
com.cdroid.appinstaller	
com.ebay.mobile	http://open.api.ebay.com/
com.microsoft.bing	http://t0.tiles.virtualearth.net/
com.tokasiki.android.voicerecorder	
fm.last.android	http://cdn.last.fm/
mobi.androidcloud.app.ptt.client	

6. FUTURE WORK

In the future, we wish to develop a more detailed set of taint propagation semantics, covering each of the Dalvik instructions.

As we mentioned in the Related Work, we would also like to develop a tool to statically analyze Android runtime libraries to give us more accurate semantics for these methods.

7. CONCLUSION

Our system has demonstrated that it is possible to empower the user to keep track of the sensitive information without the need of special permissions or rooting. This will allow a broader user base to utilize this tool. It will increase the level of trust between the user and third party applications because the user will be provided with better information regarding the app. It will deter any malicious application trying to steal sensitive information from the user. Our techniques of code instrumentation and taint tracking are extensible, and the fact that analysis is done at the bytecode level enables us to track complex behavior even in large android applications.

8. ACKNOWLEDGMENTS

We would like to thank our mentor Viabhav Rastogi for his tireless assistance and insightful advice regarding this project. Secondly, we wish to thank the other EECS 450 students for the stimulating in-class discussions, and specifically comments regarding the project. Last but not least, we would like to thank Professor Yan Chen for teaching the class, which was very educational and afforded us a great opportunity in regards to working on this project.

[6]http://uk.androlib.com/android.application.com-cdroid-appinstaller-qqiCi.aspx

9. REFERENCES

[1] S. Anand, C. Păsăreanu, and W. Visser. Jpf–se: A symbolic execution extension to java pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, 2007.

[2] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. The android monitor-real-time policy enforcement for third-party applications.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Security*, 2011.

[4] A. Chaudhuri. Language-based security on android. In *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security*, pages 1–7. ACM, 2009.

[5] 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.

[6] 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. USENIX Association, 2010.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX security symposium*, number August, 2011.

[8] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[9] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[10] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou. Analysis of android applications' permissions.

[11] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of android applications on the cloud.

[12] P. Pocatilu. Android applications security. *Informatica Economică*, 15(3):163–171, 2011.

[13] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[14] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. Google android: A state-of-the-art review of security mechanisms. *Arxiv preprint arXiv:0912.5101*, 2009.

[15] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *Security & Privacy, IEEE*, 8(2):35–44, 2010.

[16] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *Proceedings of the Web*, 2011.

[17] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 417–432, Berkeley, CA, USA, 2009. USENIX Association.

[18] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of 2nd ACM Conference on Data and Application Security and Privacy (CODASPY 2012)*, 2012.

[19] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. 2012.

[20] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

APPENDIX

A. SOURCE CODE

We are maintaining a private Git repository containing the latest version of the source code. Please contact the authors if you wish to obtain access to this repository. The current code consists of the following files:

- Makefile
- Main.java
- Parser.java
- Code.java
- Type.java
- Utils.java
- Taint.java
- Time.java

If you have the Android SDK installed, with all utils on the search path, as well as `apktook`, `smali`, `baksmali`, `signapk`[7] on the search path, you can decompile an app `apps/myapp.apk` by typing `make APP=myapp decompile`, and generate the instrumented code by typing `make APP=myapp code`, generate a new recompiled/signed APK by typing `make APP=myapp`. You can do all three steps at once by typing `make APP=myapp complete`. Finally, you can install it to the emulator by typing `make APP=myapp install`, and uninstall using `make APP=myapp uninstall`.

[7]This should be a shell script which starts the SignApk.jar file using a key pair