A Robust Dynamic Analysis System Preventing SandBox Detection by Android Malware

Jyoti Gajrani*
MNIT, Jaipur, India
2014rcp9542@mnit.ac.in

Jitendra Sarswat†
MNIT, Jaipur, India
2011uit1582@mnit.ac.in

Meenakshi Tripathi‡
MNIT, Jaipur, India
mtripathi.cse@mnit.ac.in

Vijay Laxmi§
MNIT, Jaipur, India
vlaxmi@mnit.ac.in

M.S. Gaur¶
MNIT, Jaipur, India
gaurms@mnit.ac.in

Mauro Conti
University of Padua, Italy
conti@math.unipd.it

ABSTRACT

Due to an increase in the number of Android malware applications and their diversity, it has become necessary for the security community to develop automated dynamic analysis systems. Static analysis has its limitations that can be overcome by dynamic analysis. Many tools based on dynamic analysis approach have been developed which employ emulated/virtualized environment for analysis. While it has been an effective technique for analysis, it can be espied and evaded by recent sophisticated malware. Malware families such as Pincer, AnserverBot, BgServ, Wroba have incorporated methods to check the presence of emulated or virtualized environment. Once the presence of the sandbox is detected, they do not execute any malicious behavior. In this paper, a robust emulated environment has been proposed and developed that is resilient against most of the detection techniques. We have compared our malware analysis tool DroidAnalyst against 12 publicly available dynamic analysis services and shown that our service is best when considering resilience against anti-emulation techniques. Incorporation of anti anti-detection techniques in the dynamic analysis that are purely based on emulation hinders the detection and evasion of emulated environment by malware.

Keywords

Dynamic Analysis, Anti-emulation, malware

*PhD. Scholar, CSE Dept.
†B.Tech. Scholar, CSE Dept.
‡Assistant Professor, CSE Dept
§Associate Professor, CSE Dept.
¶Professor, CSE Dept.

SIN '15, September 08 - 10, 2015, Sochi, Russian Federation

DOI: http://dx.doi.org/10.1145/2799979.2800004

1. INTRODUCTION

Smart-phones have become an essential part of our daily lives. It is because smart-phones are managing all our private data such as personal information, account details, photographs, contact information, etc. According to the NetMarketShare report[6], the largest share (52.47%) of mobile OS market was covered by Android phones till April 2015, followed by 38.81% market share of iOS. Being the most widely used mobile operating system, Android has attracted the attention of malware developers. Many static analysis techniques[15, 17, 24, 25] have been proposed for identifying Android malware, but malware families like BaseBridge, DroidKungFuUpdate, AnserverBot, and Plankton employ update attack[26], obfuscation, encryption and compression techniques to circumvent the static analysis.

Dynamic analysis techniques have also been proposed [20, 18, 4, 8, 16, 2, 10, 11, 22, 5, 1] to overcome the limitations of the static analysis. While in execution, normal (unobfuscated/ decrypted/ uncompressed) form of malware code has to be present in memory. This results in dynamic analysis techniques for malware analysis. Execution of malicious code may infect the underlying machine. Virtualization/emulation are techniques that isolate real hardware from infection when malicious code executes. Unfortunately, malware families like Pincer [7], BgServ, AnserverBot, FakeNetflix, Wroba etc. can determine that the application is running within an emulated/virtualized environment and dynamically change their behavior to evade the analysis. These techniques, employed by the malware, form anti detection techniques.

This paper presents the complete taxonomy of anti detection techniques that can be used by the malware to evade the analysis. This taxonomy characterizes anti detection techniques into 12 categories as shown in figure 2. Using this taxonomy, 13 sandboxes were evaluated including our framework. We have implemented techniques to prevent most of the anti-detection mechanisms in our dynamic analysis tool. The paper shows that DroidAnalyst is resilient against most of these detection mechanisms while other dynamic analysis tools are resilient against at most one or two.

The rest of the paper is organized as follows: Section 2 starts with motivating examples. Section 3 briefly summarizes all detection methods. Section 4 covers the approach used to handle these detection methods. Section 5 shows the evaluation and results of all dynamic analysis tools followed

by limitations in section 6 which covers methods which are still a challenge for security community. Paper covers related work in section 7 and conclusion in section 8.

2. MOTIVATING EXAMPLE

Through reverse engineering a variant of Android/Pincer.A [1]family, it was determined that it is checks various static properties before performing any malicious action. This malware sends SMS messages to its C&C server for stealing any data and communicating it through SMS. Figure 1 shows the code employed in detecting execution environment before further execution.

```
if ((str3.toLowerCase().equals("android")) || (str1.equals("000000000000000")) ||

    (str1.equals("012345678912345")) || (str2.equals("15555215554")) ||
      (Build.MODEL.toLowerCase().equals("sdk")) ||
      (Build.MODEL.toLowerCase().equals("generic")))
       com.security.cert.b.a.a.a(paramContext, true);

else
      {
        str8 = "false";
        continue;
      }
```

Figure 1: Pincer Sample Code Fragment that checks if execution environment is an emulator

The code snippet shows that it checks IMEI Number(str1), Phone Number(str2), Network Provider(str3), Build.Model before executing any malicious behavior. Therefore, changing these parameters before being checked by malware, is a way to handle anti-detection.

Before delving into the details of mitigation methods applied in the paper for handling emulator detection mechanisms employed by malware, next section presents the methods that can be used by malware to check the presence of emulated environment.

3. EMULATOR DETECTION METHODS

Figure 2 presents the complete taxonomy of detection methods that can be incorporated by malware to check the presence of emulated environment. These are described next in brief :

- **Phone ID-Based**: A smart-phone has a unique IMEI (International Mobile Equipment Identity), IMSI (International Mobile Subscriber Identity), phone number, network provider while all the emulators have some default values for these properties as shown in table 1. These values are used by malware to check the presence of dynamic analysis and further abandon its malicious activity.

- **Device Build**: Build properties give information regarding current device configuration. These system properties are present in the system image. Table 2 shows the values of various build properties which indicates the presence of emulator [23].

- **Network**: An emulated Android machine executes behind a virtual router within the 10.0.2/24 address space by default. Its network is isolated from the host machine's network. The emulator's network interface is having IP address 10.0.2.15 by default. The configured DNS servers and gateways also have particular values. These routing properties can be used as the detection method.

- **Sensors**: Sensors such as bluetooth is normally present in real mobile devices but missing in the emulator. The valid return value of `BluetoothAdapter.getDefaultAdapter()` is the indicator of Bluetooth capability and, therefore, shows that it is a physical device. It returns a NULL value for the emulator.

- **File Based Detection**: Table 3 shows the top 10 files as identified in [19]. It shows that some files are emulator specific while others are device specific. Therefore, some malicious application can perform malicious activity only after checking for the presence/absence of

[1]sha256sum: 032a095067442d60d0df9fadab07553152e5500a67fc95084441752eafd43f70

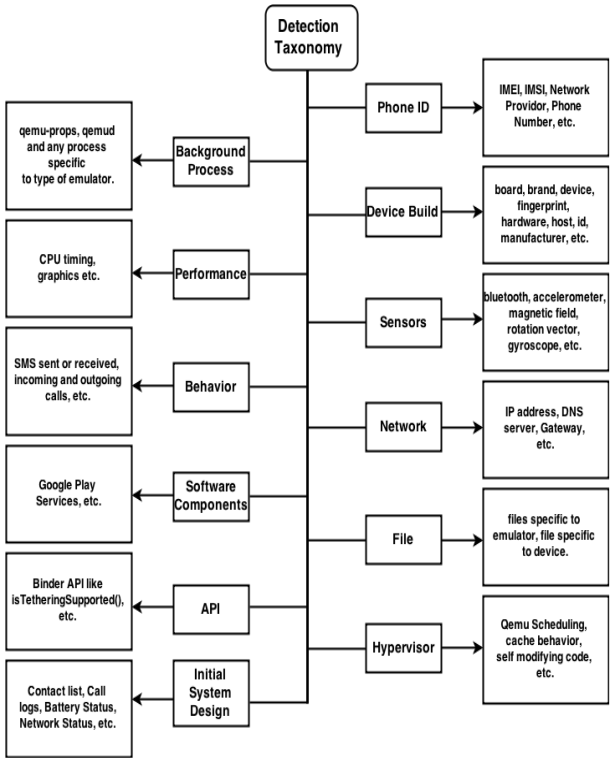

Figure 2: Emulator Detection methods taxonomy

Table 1: Phone ID

Property	Default Value	Modified Value
IMEI	000000000000000	xxxxxxxxxxxxxxx
IMSI	310260000000000*	xxxxxxxxxxxxxxx*
Phone Number	155552155ZZ	xxxxxxx55ZZ
Network Provider	Android	xxxxxxx

Table 2: Device Build

Property	Value	Implication
Build.BOARD	unknown	emulator
Build.BRAND	generic	emulator
Build.DEVICE	generic	emulator
Build.FINGERPRINT	generic*	emulator
Build.HARDWARE	goldfish	emulator
Build.HOST	android-test*	likely emulator
Build.ID	FRF91	emulator
Build.MANUFACTURER	unknown	emulator
Build.MODEL	sdk	emulator
Build.PRODUCT	sdk	emulator
Build.SERIAL	null	emulator
Build.TAGS	test-keys	emulator
Build.USER	android-build	emulator

specific files. In the table, first four files are specific to the emulator while other six are only present in the device.

Table 3: File Heuristics [19]

File Names	Specific To
"/proc/misc"	Emulator
"/proc/ioports"	Emulator
"/sys/devices/virtual/misc/cpu_dma_latency/uevent"	Emulator
"/proc/sys/net/ipv4/tcp_syncookies"	Emulator
"/proc/uid_stat"	Device
"/sys/devices/virtual/ppp"	Device
"/sys/devices/virtual/switch"	Device
"/sys/module/alarm/parameters"	Device
"/sys/devices/system/cpu/cpu0/cpufreq"	Device
"/sys/devices/virtual/misc/android_adb"	Device

- **Performance**: Performance can be measured by malware using some lengthy computation and measuring the time taken in that computation. An emulator takes longer time than a device. Graphical performance is another measure to distinguish between emulator and the device.

- **Hypervisor** : This detection is concerned with QEMU scheduling. Due to performance reasons, QEMU based emulators do not refresh the virtual program counter (PC) after execution of every instruction. As each executed instruction has to be translated according to the host machine and increasing the virtual PC results in one additional instruction execution. Therefore, looking at performance issues, QEMU based systems increase the virtual PC only when executing those instructions that break linear execution, such as branch instructions. But with this method of virtual PC updation, if scheduling is allowed to happen within the execution of a basic block, then it would be infeasible to restore the virtual PC. Due to this reason, scheduling in a QEMU environment takes place only after the execution of a basic block, not within its execution. While on real devices, scheduling can occur at any point [21].

4. APPROACH

Software Components like Google Play Store, Google Map and Initial System Design like contact list, battery status, call logs are necessary in an emulator to resemble a device. The pre-requisites that we incorporated in our framework are: geo-location provider emulation, device power characteristic, network status, adding sufficient number of contacts to the emulator, generating call and message logs. The implementation of techniques to handle other detection methods mentioned in taxonomy is as follows:

- **Emulator Binary Refinement [12]** : It is shown in the motivating example that IMEI number, IMSI number, network provider and phone number are commonly checked properties by the above-mentioned malware families. To handle Phone ID based detection, emulator binary images present in `sdk/tools/`, are refined using the hex editor [3]. We have created multiple such images, each with different values for these parameters. Modified values are some random values as shown in table 1. At run time, one of these random image is selected and the corresponding values are returned. These values can be modified in the following way:

  - IMEI & IMSI: The default value for IMEI can be found between the strings +CGSN and +CUSD, and for IMSI, value can be found in between the strings +CIMI and +CGSN as shown in the figure 3. So by replacing these default values to new random values as shown in table 1, the problem is resolved.
  - Network Provider: The first occurrence of the string `Android`, gives the required string to be modified. The string is replaced with the new random string as shown in table 1 .
  - Phone Number: It can be replaced by locating the string "%d%df%d". As the prefix of located string, we get "1555521%d" as our required string to be modified. The first 7 digits are entirely configurable, and the last four can be any even number in the range 5554 and 5584 inclusive.

- **System Image Refinement**: Device Build properties shown in the table 2 are all defined in build.prop file present in `/platforms/[target platform]/images`

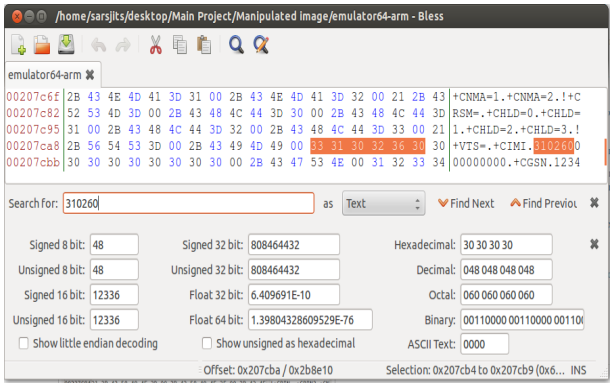

Figure 3: IMSI value refinement using Bless Hex editor

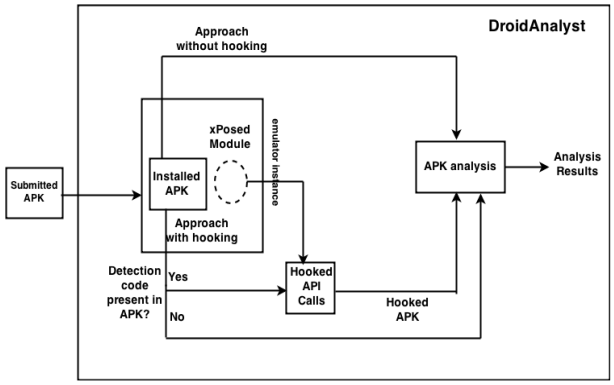

Figure 4: Illustrating use of hooking at run time

`/system.img`. For extraction of the image, yaffs2utils [14] is used. Apart from that, one more way of modifying the build properties is used with the help of setprop utility that comes with the SDK. But both the methods mentioned above were insufficient in modifying all the properties like ro.hardware. It has been observed that this hardware property has not been handled by any of the dynamic analysis tools and due to this the Build properties as a whole identify them as an emulator. So a utility called setpropex [9] was used to mitigate this problem. This utility is written in JNI and uses ptrace to change the way system build properties are accessed.

- **Android Run Time Hooking**: To make emulated environment resilient against various detection methods like File, Network and Sensors, the concept of Android Run Time Hooking is used. Using this, we dynamically modify APIs' behavior by hooking Java APIs. Android run time hooking framework Xposed [13], provides the facility to create modules that can change the behavior of the system and apps without touching any app. We used this module to hook all the incoming apps that check the presence of Files, Sensors, Network etc., as shown in figure 4, to return the values that resemble emulator as the device. The table 4 shows the API's which have been hooked, their original return values and modified return values. For e.g. the `getHostAddress()` method of `InetAddress` class can be used by malicious app to find the current IP address of machine and can be checked for the default IP address value of emulator. Here, in our solution, we hooked this method call and set the result to a value that is not the default value as shown in code snippet of figure 5. The method call is hooked in all apps that are installed and calling the method.

  Therefore, the malware that takes the decision of performing malicious actions based on these values is now tricked to execute its malicious behavior also on the emulator.

[2]for last 6 files mentioned in table 3, emulator returns false as these are only present in the device.

Table 4: Details of Methods Hooked

Class	Method	Return Value (Emulator)	Hooked Return Value
File	exists()	False/True [2]	True/False
InetAddress	getHostAddress()	10.0.2.15	192.168.1.2
Bluetooth Adapter	getDefaultAdapter()	NULL	TRUE
Sensor Manager	registerListener()	NULL	TRUE

5. EVALUATION AND RESULTS

A game (Tic Tac Toe) app is repackaged with the code to check the presence of emulated environment based on various methods covered in the section 3. This app is submitted to various sandboxes mentioned in table 5 and it sends a report to our web-server interface developed at http://www.droidanalyst.org/droidanalyst/androlog over an HTTP connection. This app detected these tools as emulator without being it detected as malicious. Next, a malicious activity is added to the app which deletes all user contacts only if the anti-emulation code identifies the environment as a device. This time the results given by these tools indicates the app as suspicious. This shows that these tools were checking for the presence of the malicious code but not anti-detection code. If the payload would have been encrypted/obfuscated then the probability of the application being identified as benign will be higher because encrypted/obfuscated payload will not be detected by static analysis. The same app is tested on our modified emulator, and it is verified that as it identified the emulator as device and it performed the malicious activity and deleted all the contacts of the emulator.

Table 5 shows the results of the experimentation done on various publicly available dynamic analysis tools. It can be seen that our framework is most resilient among all the 13 listed dynamic analysis tools (Here Y indicates that the specific system is resilient against specific method and N in-

Table 5: Comparison of Dynamic Analysis Tools based on resilience to Emulator Detection

Sandbox	Type	Year	Device Build	Network	Phone ID	Sensors	File	Hypervisor
Droidbox	offline	2011	As these are offline tools, they use system emulator					
DroidScope	offline	2012						
TaintDroid	offline	2010						
ForeSafe	online	2011	N	N	N	N	N	N
Tracedroid	online	2013	N	N	Y	N	N	N
Apk Analyzer	online	-	Paid Service					
Copperdroid	online	2012	N	N	Y	N	N	N
Andrubis	online	2012	N	N	Y	N	N	N
SandDroid	online	-	N	N	Y	N	N	N
ApkScan	online	-	N	N	Y	N	N	N
VisualThreat	online	2013	No response to the server					
Mobile Sandbox	online	2013	N	N	Y	N	N	N
DroidAnalyst	online	2014	**Y**	**Y**	**Y**	**Y**	**Y**	**N**

```
Method ipaddress =
java.net.InetAddress.class.getDeclaredMethod("getHostAddress");
  XposedBridge.hookMethod(ipaddress, new XC_MethodHook() {
      @Override
  protected void beforeHookedMethod(MethodHookParam param) throws Throwable
      {
          param.setResult("192.168.1.0");
      }
  });
```

Figure 5: getHostAddress() method hooked for returning values to resemble emulator as device

dicates that the specific system is not resilient against specific method). All these tools have been compared against the following properties: Device Build, Network, Phone ID, File, Sensors, Hypervisor. Our tool has been made resilient against detection by malware for Device Build, Network, Phone ID, Sensors, File, API, Software Components, Initial System Design properties. Performance based detection can be handled either using static analysis or it can be marked as suspicious if it takes longer time in responding.

6. LIMITATIONS

At a moment, our system is not able to handle some of the methods mentioned in the taxonomy. By checking presence of background processes which are specific to emulated environment, still malware can check emulated environment. By killing the background daemons like qemud and qemu-props, emulator will no longer able to work. Therefore, mitigation to this detection will be a challenge. Mitigation to Low-level cache and qemu scheduling detection methods, requires to redesign the emulator's scheduling policies, but it will lead to large reduction in performance. Due to this, it is not practical to make emulator's scheduling same as real device. As, a emulator can not be made as similar as to device in performance, detection based on performance can be still done.

7. RELATED WORK

Vidas [23] presented four broad classes to differentiate Android dynamic analysis systems using emulation and a real device based on their behavior, performance, hardware and software components, and some system design issues. Petsas [21] classified anti-analysis techniques used by Android malware in 3 categories namely static, dynamic and hypervisor-based heuristics. 12 sandboxes were evaluated based on these heuristics. It had been identified that all the evaluated dynamic analysis systems can be easily evaded using most of these heuristics. Morpheus [19] proposed a framework for automatically generating such detection heuristics. They identified 10,000 detection heuristics from which they selected 30 top ranked heuristics using the heuristic selector. They categorized these heuristics into 3 categories namely File, API and Property heuristics. In this paper, we presented a taxonomy considering all detection methods and evaluated 12 sandboxes along with our framework based on this taxonomy. The major contribution of our work is that we incorporated a solution to these techniques in our framework to make it robust against most of these detection heuristics.

8. CONCLUSIONS

This paper summarizes various techniques that can be used by malicious apps to evade detection in the Android dynamic analysis system. An application(apk) consisting of all the aforementioned evasion techniques was submitted to various publicly available dynamic analysis tools. It has been shown that all services or tools could be evaded by app against its detection by combining one or more tested techniques. Therefore, the strength of existing dynamic analysis systems in detecting Android malware reduces for malware having anti-detection mechanism. Moreover, methods for handling emulator detection by malware have also been developed and implemented in our framework for almost all detection methods. The low-level hypervisor detection, background processes, and behavior based detection are still a challenge. Mitigation against these, requires redesigning the emulator to resembles the real device. The mitigations em-

ployed make our framework a robust service against those malware that consists of anti-detection features.

9. REFERENCES

[1] APKAnalyser. https://github.com/sonyxperiadev/ApkAnalyser.

[2] ApkScan. http://apkscan.nviso.be/.

[3] Bless Hex Editor. http://home.gna.org/bless/.

[4] DroidBox: Android Application Sandbox. https://code.google.com/p/droidbox/.

[5] Foresafe: Automated static and dynamic analysis of Android apps. http://www.foresafe.com/.

[6] Mobile/Tablet Operating System Market Share. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1.

[7] Pincer.A: New Android Trojan. http://www.infosecurity-magazine.com/news/pincera-new-android-trojan-warning/.

[8] SandDroid: An automatic Android application analysis system. http://sanddroid.xjtu.edu.cn/.

[9] setpropex. https://goo.gl/nUSwTQ.

[10] TraceDroid. http://tracedroid.few.vu.nl/.

[11] VisualThreat. https://www.visualthreat.com/.

[12] VRT. http://vrt-blog.snort.org/2013/04/changing-imei-provider-model-and-phone.html.

[13] Xposed Module Repository. https://github.com/rovo89/Xposed.

[14] yaff2utils. https://code.google.com/p/yaffs2utils/.

[15] G. G. Anthony Desnos. Reverse engineering, Malware and goodware analysis of Android applications. https://code.google.com/p/androguard/, 2012.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. 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*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[17] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal. Androsimilar: Robust statistical feature signature for android malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 152–159, New York, NY, USA, 2013. ACM.

[18] A. Fattori, K. Tam, S. J. Khan, A. Reina, and L. Cavallaro. CopperDroid: On the Reconstruction of Android Malware Behaviors. Technical Report MA-2014-01, Royal Holloway University of London, February 2014.

[19] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 216–225, New York, NY, USA, 2014. ACM.

[20] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[21] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[22] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1808–1815, New York, NY, USA, 2013. ACM.

[23] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM.

[24] C. Zhen. APKInspector, a static analysis platform for android applications. https://code.google.com/p/apkinspector/, 2013.

[25] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[26] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.