Identifying and Evading Android Sandbox through Usage-Profile based Fingerprints

Valerio Costamagna
Universitá degli studi di Torino

Cong Zheng
Palo Alto Networks

Heqing Huang
IBM TJ Watson Research Center

ABSTRACT

Android sandbox is built either on the Android emulator or the real device with a hooking framework. Fingerprints of the Android sandbox could be used to evade the dynamic detection. So, in this paper, we first conduct a measurement on eight Android sandboxes and find their customized usage profile (e.g., contact, SMS) can be fingerprinted by attackers for evading the sandbox. From our measurement results, most Android sandboxes have empty usage profile fingerprints, or fixed fingerprints, or random artifact fingerprints. So, without protections on such user profiles, Android malware can identify these fingerprints that associate with different sandboxes and hide its malicious behaviors. At last, we propose several mitigation solutions trivial to implement, including generating and feeding random real usage profiles to the malware sample every time, as well as a hybrid approach, which combines both random and fixed usage profiles.

Keywords

Android; Mobile Security; AntiVirus; Sandbox; Fingerprinting

1. INTRODUCTION

Among the massive volume of Android apps used by Android users, there exists a lot of Android malware, which become the main threat for Android users currently. To mitigate the threat of the Android malware, static and dynamic analysis techniques are the main solutions to detect Android malware. Static analysis has the limitation on detecting the malware when using the code obfuscation, native code, Java reflection and packer. But, dynamic analysis can help detect such Android malware more precisely in its dynamic sandboxes. The traditional dynamic analysis sandbox is built either on the Android emulator or the real device to enable fast and effective malware detection.

To evade dynamic analysis, some anti-emulator techniques [19, 27, 14] were proposed, and they are commonly used by Android malware. In general, these techniques were designed to obtain fingerprints of the runtime environment of the Android emulator. Recently, BareDroid [16] was proposed to use real devices to build the Android sandbox. This method can mitigate the anti-emulator techniques , but we believe the arms race between dynamic analysis techniques and evasion techniques is endless.

No matter whether the Android sandbox is built on the Android emulator or the real device, the Android malware can still evade the detection of the sandbox through identifying the difference between the emulated phone and the real user phone. In this paper, we conduct a measurement on collecting fingerprints from public Android sandboxes, including AV sandboxes, online detection sandboxes and even sandboxes used by app markets. Through analyzing collected fingerprints, we propose an evading technique based on a new type of fingerprints of Android sandboxes: *usage-profile based fingerprints* (e.g., the contact list information, SMS, and installed apps). The measurement result shows that these Android sandboxes have no usage-profile based fingerprints, or only have a fixed usage-profile based fingerprint, or have a random artifact fingerprint. So, most current Android sandboxes have not protected their usage-profile fingerprints, and are potentially bypassed by malware samples.

Therefore, by analyzing the usage-profile based fingerprints of the Android sandboxes and real Android user's device, the Android malware can still potentially evade the dynamic sandbox because of two reasons: 1) extracting some fingerprints, such as installed apps, are not very sensitive towards the verdict making of dynamic analysis, since those behaviors are commonly existed in benign apps. For example, Android Ads SDK extracts the installed app list for accurately distributing ads; 2) even though extracting some fingerprints (e.g. the contact list and SMS) may be regarded by dynamic analysis as malicious, the Android malware can still evade the detection by mimicking or repackaging [30] as a contact app or SMS app. So, the advanced Android malware could firstly inspect these fingerprints and then launch other more powerful behaviors (e.g., rooting and sending SMS) if it does not identify the current environment as a sandbox environment.

To summarize, this paper makes the following contributions:

- 1) **New problem.** We propose a new Android sandbox fingerprinting technique, which is based on the careless design of usage-profiles in most current sandboxes. We observe that malware developers can col-

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

lect usage-profile based fingerprints from many Android sandboxes and then leverage these fingerprints to build a generic sandbox fingerprinting scheme for the sandbox analysis evasion.

- 2) **Implementation.** We conduct a measurement on collecting usage-profile based fingerprints on popular Android sandboxes. The results show that most Android sandboxes designers have not protected these fingerprints by generating the random fingerprints every time for running a different sample. Only few sandboxes generate the random fingerprints, but these random fingerprints are different from fingerprints in user's real phones.

- 3) **Mitigations.** We propose mitigations to further guide a proper design of these sandboxes against this hazard.

The remainder of the paper is structured as follows: in Section 2 we introduce background and motivations underlying our research, then in Section 3 we discuss our system design and in Section 4 we present collected results which effectively shows the effectiveness of our techniques. Proposed mitigations and related work are discussed in Section 5 and Section 6 respectively. Finally, Section 7 concludes the paper.

2. BACKGROUND AND MOTIVATION

In this section, we first describe the current status of the Android sandbox for detecting Android malware, and then present the existing evading techniques commonly used in Android malware. In this paper, we explore a new direction: distinguish the environment where the malware is running and focus on bypassing the sandbox though a new type of fingerprint: the usage-profile based fingerprint which basically exploits the content generated by real users. Though the measurement in section 4, we show that most sandboxes can be bypassed by using this kind of fingerprint.

2.1 Android sandbox

The Android sandbox is a running environment with a hooking framework, either on the real device or on the emulator. With the hooking framework, the Android sandbox can check the suspicious API calls or system calls, which are used to make a verdict for the app. Many hooking frameworks and approaches have been proposed in past years, including the static instrumentation, [8, 2, 11]. Different hooking frameworks can hook different levels of behaviors. Xposed can only hook the Dalvik instruction call by replacing the Zygote process. Adbi can hook system calls by the inline hooking. ArtDroid permits to intercept calls to Java virtual-methods. These hooking frameworks can also be implemented in the real device for detecting malware, even if malware applies the anti-emulator technique.

The automated UI interaction is the most important technique used in the Android sandbox to improve its coverage. One approach commonly used in most Android sandboxes is using the MonkeyRunner tool to generate random user interactions and system events to the sandbox. But, the random events are very limited on triggering all malicious behaviors of apps. To address this problem, SmartDroid [31], AppsPlayground [21], CuriousDroid [18] propose different approaches to improve the automatic capability of UI interaction.

2.2 Evading technique

To evade the dynamic analysis in the Android sandbox, Android malware adopt many evading techniques, which can be categorized as two aspects: 1) anti-emulator. 2) anti-interaction.

Anti-emulator techniques are used widely in most Android malware samples. The anti-emulator technique comes from identifying the difference of system and device information between the emulator and the real device. As the emulator is a default customized Android system, some system information is assigned with default values, e.g. the IMEI in the emulator is a string of zeros by default. The device information includes the information of sensors and sim card. Because of emulating these device information, attackers can find the difference from the emulated data. For anti-anti-emulator, the Android sandbox can send some artifacts to malware. However, there is an arm race when building the Android sandbox on the emulator, namely, the anti-emulator and anti-anti-emulator arm race. A countermeasure for anti-emulator techniques is building the Android sandbox on real devices.

2.3 Motivation

Even though the Android sandbox can be built on the real device, the Android malware can still evade its dynamic analysis through identifying the usage-profile based fingerprint, which reflects the real user information or the fake information, such as the contact list, SMS and the installed app list. Such usage-profile based fingerprints could be potentially leveraged to evade the Android sandbox by attackers. If the fingerprint contains fixed deterministic data or is empty or an artifact data which is not obviously created by users (e.g., "abcdefg" for the name in the contact list), Android malware can directly identify the sandbox environment and hide its following malicious behaviors. Even if all fingerprints look exact the same as for the fingerprints in a user's real device, there still be a way to identify the Android sandbox if the fingerprint is fixed or not random enough in every running time. The attackers can send a lot of probing apps to the sandbox of AVs, extract all pesudo-random fingerprints and build a general pattern for this sandbox. Later, other malware can use this pattern to evade the Android sandbox.

Therefore, in Section 4 we first conduct a measurement on fingerprints of some popular sandboxes, and then we conduct a study on whether these sandboxes protect their fingerprints for preventing themselves from evading.

3. SYSTEM IMPLEMENTATION

In this section, we present the design of the fingerprint collector. As depicted in Figure 1, it consists of two components: (I) scouting-app and (II) fingerprint extractor. The scouting-app is used to collect usage-profile fingerprints. It uses only public Android APIs to get information about the execution environment. The app does not use native code, neither Java reflection nor code obfuscation techniques to hide its sensitive behaviors. The reason is that the prior static analysis in most analyzers would highly regard the scouting app as a benign app and filter it out if none suspicious evidence is found. To this end, and to make the fin-

Figure 1: The design of fingerprint collector

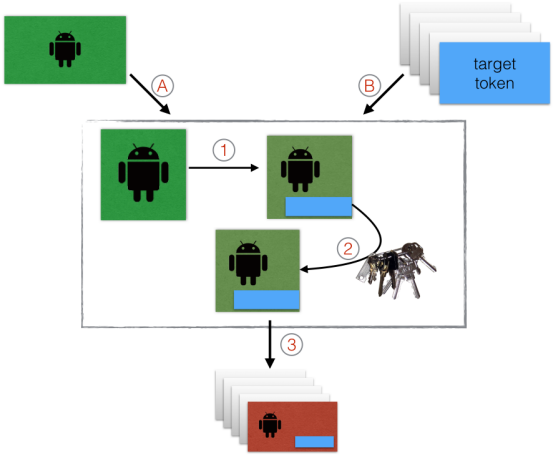

Figure 2: Scouting apps generator

Table 1: usage-profile Data

Type	Data
Contact	Name, numbers, email
Location	GPS, network, latitude, longitude
SMS	inbox, send, draft
WiFi	SSID, signal-strength
App	package-name, version, hash
Battery	battery-level, stat, chargePlug
Call	recent calls

gerprint process even more stealthy as well we implemented several scouting-app each testing for a subset of usage-profile we were interested in. By following this approach we avoid to create a single application that requires a lot of sensitive privileges which could be marked as suspicious and then manually analyzed later on, instead we use several scouting-app such that none of those would require a suspicious combination of sensitive privileges.
The scouting app uses different threads to execute the scouting logic. Each analysis sends its results back to the remote server in the JSON format. All types of collected fingerprints are shown in Table 1.

To track which sandbox analyzes the scouting app, we need to build a scouting app generator. In the scouting app generator, we use a testing app as the base app to generate different apps for each target sandbox. Each generated app is signed by a different certificate. We also make the signature of each app different to avoid the trivial caching mechanism used by the sandbox. Moreover, since we need to classify the results coming back from the testing app, we repackaged the testing app for inserting a *target token*. Then, at run-time the app sends back that token within its fingerprint data. Besides that, the app also sends back its certificate signature, so that we can check if the app has been repackaged by the sandbox for using the static instrumentation hooking framework. Advanced sandbox might employ mechanisms in order to fool the certificate signature check, to countermeasure those we included Java code which is responsible for calculating the DEX file hash value at runtime. With these checks in place, we can detect sandbox which eventually have modified the application being analyzed in order to disable signature checks.

Figure 2 shows the generator design: the system receives the testing app (A) and a list of *target tokens* (B) as inputs. Then, for each target token the generator performs repackaging of the testing app by inserting the target token (1) in app's assets directory. Then, a new digital certificate is created and the repackaged app is signed with. This procedure repeats for each target token. Finally, the output is a set of apps which contain a token for each different target sandboxes.

The fingerprint extractor component first does normalization on the results collected by our scouting applications. It then stores the unique data in a database and uses the token mechanism to create the mapping between the scouting app and the target sandbox. Finally, the collected data is analyzed to determine whether the produced data from the sandbox is dynamically generated or not.

4. RESULTS

In this section, we describe the fingerprints collected by the scouting app. Because of the difference in the Android app distribution channel, we choose different types of mobile sandboxes accordingly as the target of this work, which are shown in the Table 2. In fact, it is composed by both official and third-party stores, i.e. Google Play, aptoide, F-Droid, etc... and also by stock applications installed on real-world devices. First column in Table 2 shows the type of sandbox being analyzed, second and third columns represent sandbox's name and its availability at the time of writing respectively. We choose to target mobile anti-virus vendors because they use either their own customized sandboxes or an online sandbox service to dynamically analyze collected samples. Moreover, we collect the environment-related data from third-party stores, because it is one of the most popular malware spreading channel. Considering that online malware analysis services are used by both mobile anti-virus and third-party stores, we also collect environment-related data from available online sandboxes. Unfortunately, compared to other previous works [28, 20, 17, 22], we find that just few of these online services are available at the time of

Table 2: Mobile Sandboxes employed for evaluation. (✗ means not available at the time of writing)

Type	Sandbox	Available
Online	Andrubis [29]	✗
	SandDroid [24]	✓
	TraceDroid[26]	✗
	CopperDroid[23]	✗
	HackApp [4]	✗
	NVISO ApkScan [7]	✓
	Koodous [6]	✓
	VirusTotal [25]	✓
	Joe Sandbox mobile [9]	✓
	ForeSafe	✗
Antivirus	Bit Defender	✓
	360 mobile	✓
	TrendMicro	✓
	Kaspersky	✓
	Tencent mobile	✓
App store	Amazon [1]	✓

writing, as evicted by third column of Table 2.

We use the system described in Section 3 to generate 10 applications for each target sandbox. Then, depending of the target type, we manually upload each generated app by using the web interface provided by the online sandbox service, or install on a real device or send it to the third-party store. In the latter case, we immediately remove the app as soon as the scouting app has been analyzed to make sure nobody has ever downloaded it. Moreover, we do not disclose the mapping between the sandbox name and its representing label to allow to sandbox maintainers to fix this problem.

We receive the environment-related data from 80% of available sandbox in the Table 2. Table 3 and Table 4 show a summary of the most relevant environment-related data collected by our testing app. The former contains results of Contact data while the latter contains results of SMS data. Both Tables have the last two columns in common which report whether the specific sandbox does dynamically generates the environment-related data (Random data column) and count the number of collected results (num. of results column) respectively. As for Table 3 we reported name, number and email were collected by the analysis, instead in Table 4 we included sending number and body if any. We have not included the phone call data since all sandboxes return the empty result. Thus, the phone call data could be one of the best user-profile fingerprints.

As describe in Section 3, our system allows us to detect if a target sandbox returns a fixed data for a specific *environment artifact*. Unfortunately, as shown by "Random Data" column in Table 3 and 4, the results indicate that most Android sandboxes do not use dynamically generated environment data.

One interesting finding is that two mobile anti-virus sandboxes ran the application on a real-world device. In fact, they returned concrete
WiFi artifact data (i.e. "wifiscan":"DIRECTCnFireTV_8048", "wifiscan":"AppStore","wifiscan":"AVcontrol").
Moreover, the data collected from the *battery artifact*, presented in Table 5 is exactly the same for all the sandbox, except for the data returned by these two anti-virus sandboxes. In fact, an unmodified Android emulator returns a fixed battery stats, which consists of the following string: "chargePlug":"1","batteryStat":"2","batteryLevel":"50.0". Instead, results collected from a real-world device look like different:
"chargePlug":"2", "batteryStat":"3", "batteryLevel":"100.0".
Regarding *application artifact* data, we found that about 77% of targets the sandboxes contain the identical set of applications found on stock Android emulator. As Tables 6 shown, some anti-virus and all online sandboxes present only the default installed app list from the Android emulator, and they never return random data for the installed app list.

In the following list, we include some interesting apps' package name:

- com.amazon.geo.contextcards
- com.amazon.rialto.cordova.webapp.
  webapp50f95ccb054443059066310aefdf969b
- IAPV2AndroidSampleAPK
- com.amazon.otaverifier
- com.tencent.token

5. DEFENSE

As we discussed in previous sections, the environment-related data represents an interesting source of information, which could be exploited to identify the mobile sandbox. Building a database of fingerprints from all sandboxes, an attacker could take the advantage to easily detect an existing mobile sandbox by checking the presence of matching data in the running environment.

To avoid this trivial detection mechanism, it is important to generate environment-related data dynamically, so each application under analysis would see a different environment.

Note that the presence of each previous discussed *environment artifact* is also an important indicator of the goodness of the running environment. As discussed in Section 4, sandboxes have not included the Call artifact. Even though such type of environment data i.e. WiFi data, could not be generated, one can artificially inject it by using hooking frameworks introduced in previous works [11, 8, 5, 3].

In addition to set up the sandbox environment as close as a real user phone, the sandbox developer could take a hybrid approach to detect such fingerprint collection behavior. For example, some sandboxes are equipped with the same set of environmental data, and other sandboxes are equipped with complete different data. When performing the dynamic analysis, each suspicious sample should be run in these two different sandboxes with similar triggering events. If two types of sandboxes yield different malicious behaviors, it indicates that the malware performs the evasion attack by checking the usage-profile based fingerprints.

6. RELATED WORKS

A couple of previous research works focus on evading the Android sandbox or Android anti-virus scanners. Huang, et al. [13] discovered two generic evasions that can completely evade the signature based on-device Android AV scanners,

Table 3: Contacts usage-profile Results

Sandbox	Name	Phone	Email	Random data	num. of results
store_1	Mary Edwards Harry Grace	867-5309 867-5319	✗ ✗	✗	2
online_x	✗	✗	✗	✗	1
online_y	Firstname1 Firstname2 Firstname3	1 301-234-5678 1 381-234-5678 1 381-234-5678	✗ ✗ ✗	✗	3
av_1	Ion Gheo Txet4321	074-354-3219 072-345-6789 074-212-3456	✗ ✗ ✗	✗	4
av_2	Cynthia Alexander Alexandra Dolores	✗ ✗ ✗ ✗	✗ ✗ ✗ ✗	✗	2
av_3	MARS	1 566-666-6666	chengkai_tao@****.com.cn	✗	10
av_4	Boulder Hypnotherapy Ctr St. John Ambulance Maidstone Golf Centre	(303) 776 8100 061 412480 01622 863163	z**y@gmail.com l**k@stjohn.ie nick@totalgolfcoaching.co.uk	✓	36
av_5	Jian Li Jian Li Xuri Jin	1 3743888229 13606500401 13250324837	lijxev@admin.cn b0***54@admin.cn 55**43@yuepao.cn	✓	20

Table 4: SMS usage-profile Results

Sandbox	Num	Body	Random data	num. of results
store_1	2020845845 +18454119384 5618675309	Hey Who Important	✗	3
online	✗	✗	✗	1
av_1	12345 1234	smsmomealain smsmomealain	✗	2
av_2	1354-587-2365 1857-667-8565	Mum Jefferson	✗	2
av_3	1301-234-5678 1301-234-5678 1381-234-5678 1581-234-5678	Ggggggggg Testzzzzzz 123456789 Fffffffmmmmmm	✗	12
av_4	10668820 10086 13770837893	guchulaichonghuafei nihao,laikaitong4g-songliuliango nihao,nishi4staryonghu,keyihuantingji	✓	15
av_5	13540877911 15874984303 18660928896	u0oydyemvub4lu86kfcbwad46pvhmh6o 2fqjfkr629blowmxso4jh6dzqtk3f4j2 lhb0j8hxi48wdjiua1q0qvsleeffgt6g	✓	22

Table 5: Battery usage-profile Results

Sandbox	Battery stats	Random data
av_1	batteryStat'='3, 'chargePlug'='2', batteryLevel = '100'	✓
av_2	batteryStat'='2, 'chargePlug'='1', batteryLevel = '98'	✓
online, store	batteryStat'='2','chargePlug'='1','batteryLevel'= '50.0'	✗

Table 6: Installed Apps usage-profile Results

Sandbox	Emulator apps	uncommon apps	Random data
av_x	✓	✓	✗
av_y	✓	✗	✗
store	✓	✓	✗
online	✓	✗	✗

while we are focusing on the Android sandboxes used offline. Sand-Finger [15] collects fingerprints from 10 Android sandboxes and AV scanners to bypass current AV engines and all of fingerprints used by Sand-Finger are hardware-related or system-related, which are different from our user profile based fingerprints. Timothy [27] presented several sandbox evasions by analyzing the differences in behavior, performance, hardware and software components. He also revealed that dynamic analysis platforms for malware that purely rely on emulation or virtualization face fundamental limitations that may make evasion possible. The above approaches are mainly related to detect the Android emulator environment. Wenrui, et al. [12] proposed to evade the Android runtime analysis through by identifying automated UI explorations. Similarly, our work is to distinguish the difference between the sandbox environment and the real user device environment through usage-profile based fingerprints.

In [10] Blacktorne et al. proposed *AVLeak* a blackbox technique to extract emulator fingerprints. Although it address the similar problem about how to efficiently extract emulator fingerprints the implemented methodology is not suitable on Android. Moreover *AVLeak* was not focused on usage-profile based fingerprints which are quite relevant ones for the mobile ecosystem.

7. CONCLUSIONS

In this paper, we present a novel mobile sandbox fingerprinting for the sandbox evasion by checking the usage profiles. As demonstrated by the evidence of collected results, usage profiles data could be used by a malware to fingerprint current sandboxes. We demonstrate that most of our analyzed sandboxes are built with fixed usage profiles and they completely overlook this potential hazard. Mitigations are provided by us to prevent such evasion hazards, e.g., different app runtime should present dynamically generated usage profiles, or hybrid usage profiles. Our research raises the alert for the usage-profile based fingerprinting hazard when developing mobile sandboxes and sheds lights on how to mitigate similar hazards.

8. REFERENCES

[1] Amazon application store. https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011. Accessed: 2016-02-27.

[2] Android dynamic binary instrumentation. https://github.com/Samsung/ADBI. Accessed: 2016-02-27.

[3] Android hooker. https://github.com/AndroidHooker/hooker. Accessed: 2016-02-27.

[4] Apphack mobile scanner. https://apphack.com. Accessed: 2016-02-27.

[5] Cydia substrate for android. http://www.cydiasubstrate.com. Accessed: 2016-02-27.

[6] Koodus collaborative platform. https://koodous.com. Accessed: 2016-02-27.

[7] Nviso apk-scan. https://apkscan.nviso.be/. Accessed: 2016-02-27.

[8] Xposed framework. https://repo.xposed.info. Accessed: 2016-02-27.

[9] Joe mobile sandbox. http://www.joesecurity.org/joe-sandbox-mobile, 2016.

[10] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener. Avleak: Fingerprinting antivirus emulators through black-box testing. In *Proceedings of the 10th USENIX Conference on Offensive Technologies*, pages 91–105. USENIX Association, 2016.

[11] V. Costamagna and C. Zheng. Artdroid: A virtual-method hooking framework on android art runtime. *Innovation in Mobile Privacy & Security IMPS'16*.

[12] W. Diao, X. Liu, Z. Li, and K. Zhang. Evading android runtime analysis through detecting programmed interactions. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '16, pages 159–164, New York, NY, USA, 2016. ACM.

[13] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu. Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.

[14] 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.

[15] D. Maier, M. Protsenko, and T. Müller. A game of droid and mouse. *Comput. Secur.*, 54(C):2–15, Oct. 2015.

[16] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 71–80, New York, NY, USA, 2015. ACM.

[17] S. Neuner, V. Van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl. Enter sandbox: Android sandbox comparison. *arXiv preprint arXiv:1410.7749*, 2014.

[18] M. L. W. R. P. Carter, C. Mulliner and E. Kirda. Curiousdroid: Automated user interface interaction for android application analysis sandboxes. In *In Financial Cryptography and Data Security - 20th Internationa Conference (FC)*, 2016.

[19] 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.

[20] 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*, page 5. ACM, 2014.

[21] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and*

Privacy, CODASPY '13, pages 209–220, New York, NY, USA, 2013. 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*, pages 1808–1815. ACM, 2013.

[23] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[24] B. R. Team et al. Sanddroid: An apk analysis sandbox. xiâĂŹan jiaotong university, 2014.

[25] V. Total. Virustotal-free online virus, malware and url scanner, 2012.

[26] V. Van Der Veen, H. Bos, and C. Rossow. Dynamic analysis of android malware. *Internet & Web Technology Master thesis, VU University Amsterdam*, 2013.

[27] 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.

[28] 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*, pages 447–458. ACM, 2014.

[29] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414*, 1:5, 2014.

[30] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014.

[31] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM.