Obfuscation techniques against signature-based detection: a case study

Gerardo Canfora*, Andrea Di Sorbo*, Francesco Mercaldo*, Corrado Aaron Visaggio*
*Department of Engineering, University of Sannio, Benevento, Italy
{canfora, disorbo, fmercaldo, visaggio}@unisannio.it

***Abstract*—Android malware is increasingly growing in terms of complexity. In order to evade signature-based detection, which represents the most adopted technique by current antimalware vendors, malware writers begin to deploy malware with the ability to change their code as they propagate.**

In this paper, our aim is to evaluate the robustness of Android antimalware tools when various evasion techniques are used to obfuscate malicious payloads. To support this assessment we realized a tool which applies a number of common transformations on the code of malware applications, and applied these transformations to about 5000 malware apps. Our results demonstrate that, after the code transformations, the malware is not detected by a large set of antimalware tools, even when, before applying the transformations, malware was correctly identified by most antimalware tools. Such outcomes suggest that malware detection methods must be quickly re-designed for protecting successfully smart devices.

I. Introduction

In recent years mobile phones have become among the favorite devices for running programs, browsing websites, and communication. With the increasing capabilities of these devices, they often represent the preferred gateways for accessing to sensitive assets, like private data, files and applications, and to connectivity services.

This has stimulated malware writers to heavily target mobile software.

In fact, G DATA security experts report that 440 267 new malware files were discovered in the first quarter of 2015. This trend represents an increase of 6.4 percent compared to the fourth quarter of 2014 (413 871) [1].

The mechanisms employed by attackers to diffuse malware can be grouped basically into three categories: (i) repackaging, (ii) attack upgrade and (iii) drive-by download [8].

Malware writers implement increasingly sophisticated techniques for obfuscating malicious behavior, in order to evade detection strategies employed by actual antimalware products [8]. During its propagation, malware code changes its structure, through a set of transformations, in order to elude signature-based detection strategies. Indeed, polymorphism and metamorphism are rapidly spreading among malware targeting mobile applications [5].

Paper contribution. This paper is aimed at evaluating the robustness of free and commercial antimalware solutions against a set of trivial and common transformations on mobile applications containing malicious payloads. We realized an engine that applies eight transformations (see Section II-B) to malware code which alter the code's shape, but not the behavior of the malware.

We used our engine on a dataset containing 5560 diffused malware and assessed the efficacy of 57 free and commercial antimalware products against the transformations we applied. To support this evaluation, for each analyzed antimalware, we compared the detection rates occurred before and after the application of the transformations. Outcomes demonstrate that the most antimalware is anymore able to recognize a large subset of malware after the transformations.

Several works in the literature evaluated the effectiveness of existing mobile malware detection mechanisms.

In [7] authors present ADAM, an automated system for evaluating the detection of Android malware. Using ADAM, researchers apply a set of trivial obfuscation techniques to a dataset containing 222 malware samples. For each antimalware product, results show how each of the obfuscations can significantly reduce the detection rate. ADAM implements only a few transformations, such as: (i) renaming methods, (ii) introducing defunct methods, (iii) code reordering, and (iv) string encoding, in addition to repacking and assembling/disassembling.

Authors in [2] test 11 antimalware solutions using 10 malware samples and 10 altered ones. In the evaluation they show that 7 on 10 antimalware were able to recognize all the sample belonging to malware dataset; while using the altered samples the malware identification decreases dramatically.

Researchers in [5] evaluate 10 antimalware tools using 6 original and transformed malware samples belonging to six different families. They conclude that all the antimalware products are susceptible to common evasion techniques.

Differently from these previous studies, we present a more exhaustive assessment of 57 among the most popular antimalware tools for Android, involving in our experiment a dataset containg above 5500 malware samples, belonging to 178 different malware families.

Paper structure. The paper proceeds as follows: Section II describes the case study; Section III illustrates the results of experiments and finally, conclusions are drawn in the Section IV.

II. Study and evaluation methodology

The main goal of our research is to evaluate the effectiveness of Android free and commercial antimalware products against a set of evasion techniques adopted for obfuscating malware's code.

Thus, the research questions we want to answer are:

Family	Inst.	Attack	Activation	Apps
FakeInstaller	s	t,b		925
DroidKungFu	r	t	boot,batt,sys	667
Plankton	s,u	t,b		625
Opfake	r	t		613
GinMaster	r	t	boot	339
BaseBridge	r,u	t	boot,sms,net,batt	330
Kmin	s	t	boot	147
Geinimi	r	t	boot,sms	92
Adrd	r	t	net,call	91
DroidDream	r	b	main	81

Table I
NUMBER OF SAMPLES FOR THE TOP 10 FAMILIES WITH INSTALLATION DETAILS (STANDALONE, REPACKAGING, UPDATE), KIND OF ATTACK (TROJAN, BOTNET) AND EVENTS THAT TRIGGER MALICIOUS PAYLOAD.

- **RQ1:** *To what extent are the detection algorithms adopted by free and commercial antimalware tools effective against well-known code obfuscation techniques?*
- **RQ2:** *What are the malware families able to pass the detection of antimalware tools after code transformations?*

This Section describes the research approach we used in order to answer the research questions and the techniques we employed to assess the robustness of antimalware products against evasion techniques.

A. The Dataset

The malware dataset used in this experiment was collected from Drebin project [1][3, 6]. It is partitioned according to the *malware family*: each family contains samples which have in common several characteristics, like payload installation, the kind of attack and events that trigger the malicious payload [8]. For brevity's sake, in Table I we list only the 10 malware families with the largest number of applications in our malware dataset with installation type, kind of attack and event activating malicious payload [4].

B. The Obfuscation Techniques

In this Section we describe the approach we used to transform the code of malware apps in our dataset (see Section II-A) and the obfuscation techniques we employed.

Android runs *Dalvik* executables stored in *.dex* files. In order to apply transformations to app code, we first obtained the *smali* (a human readable dalvik bytecode) representation of the code, through *apktool*[2], a tool for reverse engineering which allows to decompile and recompile Android applications. *Apktool* is able to decode resources to nearly original form and rebuild them after making some modifications. The *smali* representation was the target of our transformations.

We also designed and implemented a java tool able to apply a set of trivial and static code modifications to *smali* representation in an automated way. The transformation engine we developed has been released under open source license[3]. In the following we describe the transformations that the tool is able to perform:

1) **Disassembling & Reassembling.** The compiled Dalvik bytecode in *classes.dex* of the application package may be disassembled and reassembled through *apktool*. This allows various items in a *.dex* file to be re-arranged or represented in a different way. In this way signatures relying on the order of different items in the *.dex* file will likely be ineffective with this transformation.
2) **Repacking.** Every Android application has got a developer signature key that will be lost after disassembling the application and then reassembling it. To create a new key we used the *signapk*[4] tool to embed a new default signature key in the reassembled app to avoid detection signatures that match the developer keys.
3) **Changing package name.** Each application is identified by a unique package name. This transformation is aimed at renaming the application package name in both the Android Manifest and all the classes of the app, in order to elude detection by signatures based on package name.
4) **Identifier renaming.** To avoid detection signatures relying on identifier names, this transformation renames each package name and class name by using a random string generator, in both Android Manifest and *smali* classes, handling renamed classes' invocations.
5) **Data Encoding.** The *dex* files contain all the strings and arrays used in the code. Strings could be used to create detection signatures to identify malwares. To elude such signatures, this transformation encodes strings with a *Caesar cipher*. The original string will be restored during application run-time, with a call to a *smali* method that knows the *Caesar key*.
6) **Call indirections.** Some detection signatures could exploit the call graph of the application. To evade such signatures we designed a transformation which mutates the original call graph, by modifying every method invocation in the *smali* code with a call to a new method inserted by the transformation which simply invokes the original method.
7) **Code Reordering.** This transformation is aimed at modifying the instructions order in *smali* methods. A random reordering of instructions has been accomplished by inserting *goto* instructions with the aim of preserving the original runtime execution trace.
   The transformation was applied only to methods that don't contain any type of jumps (if, switch, recursive calls).
8) **Junk Code Insertion.** These transformations introduce those code sequences that have no effect on the function of the code. Detection algorithms relying on

[1]http://user.informatik.uni-goettingen.de/~darp/drebin/
[2]http://ibotpeaches.github.io/Apktool/
[3]https://github.com/faber03/AndroidMalwareEvaluatingTools
[4]https://code.google.com/p/signapk/

instructions (or opcodes) sequences may be defeated by this type of transformations.

This transformation provides three different junk code insertions: (i) insertion of *nop* instructions into each method (Figure 1 shows a code snippet resulting after applying type I junk code insertion), (ii) insertion of unconditional jumps into each method (Figure 2 shows a code snippet resulting after applying type II junk code insertion), and (iii) allocation of three additional registers on which garbage operations are performed.

```
# virtual methods
.method public final run()V
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
    .locals 2

    iget-object v0, p0, Lc$a;->c:Landroid/webkit/WebView;

    invoke-virtual {v0}, Landroid/webkit/WebView;->stopLoading()V

    iget-object v0, p0, Lc$a;->c:Landroid/webkit/WebView;

    invoke-virtual {v0}, Landroid/webkit/WebView;->destroy()V

    iget-object v0, p0, Lc$a;->d:Lb;

    invoke-virtual {v0}, Lb;->a()V

    iget-boolean v0, p0, Lc$a;->f:Z
```

Figure 1. An obfuscated smali code snippet with type I junk code insertion, i.e. adding nop (no operation) instructions.

```
.method static a(Ld;Ljava/util/Map;Landroid/net/Uri;Landroid/webkit/WebView;)V
   nop
   goto :saltojump_24
   :saltojump_24
   nop
   goto :saltojump_25
   :saltojump_25
   nop
   nop
   goto :saltojump_26
   :saltojump_26
   goto :saltojump_27
   :saltojump_27
   goto :saltojump_28
   :saltojump_28
   goto :saltojump_29
   :saltojump_29
   .locals 4
   .annotation system Ldalvik/annotation/Signature;
```

Figure 2. An obfuscated smali code snippet with type II junk code insertion, i.e. adding nop (no operation) and unconditional jump instructions.

We combine together all the transformations listed to generate stronger obfuscation, shown in Figure 3.

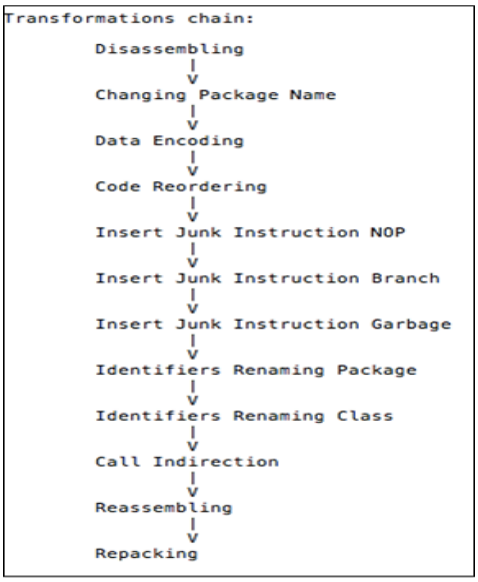

Figure 3. Transformation chain

C. The Experiment

We employed a dataset composed of 5560 malware apps belonging to 178 different malware families, as described in Section II-A.

For 794 elements of the dataset *apktool* failed in decompiling and recompiling the application. Thus we removed these apps from the original dataset because we would have not been able to apply code transformations. Hence, our final test set comprised 4766 malware apps.

We submitted each of the malware apps in our final test set to VirusTotal[5] through its web-service API. VirusTotal is a free online service that analyzes files and URLs enabling the identification of viruses, worms and other kinds of malicious contents, detected by 57 free and commercial antimalware engines and website scanners.

VirusTotal simply aggregates information about the tests performed by the antimalware tools it includes and reports into a human-readable form the results. Therefore, we collected all the scanning results for each app in a database for analysis.

In order to answer our research questions (see Section II), we applied all the transformations discussed in Section II-B combined together on the full malware dataset and then resubmitted them to VirusTotal and stored the results in the database.

To assess the robustness of the different antimalware employed by VirusTotal against simple code transformations, for each antimalware product we compared the results obtained before applying transformations with the results obtained after the transformation process.

III. Results

The 57 antimalware products we evaluted are listed in Table II.

[5]https://www.virustotal.com

Product:	**Web Site:**
Alibaba	http://www.alibabagroup.com/en/global/home
F-Secure	https://www.f-secure.com
AVG	http://www.avg.com/
ESET-NOD32	https://www.eset.com/
Avira	https://www.avira.com/
AhnLab	www.ahnlab.com/
Sophos	www.sophos.com/
GData	https://www.gdatasoftware.com/
BitDefender	www.bitdefender.com/
Ad-Aware	it.lavasoft.com/
Emsisoft	https://www.emsisoft.com/
MicroWorld-eScan	www.escanav.com/
NANO-Antivirus	www.nanoav.ru/
Kaspersky	www.kaspersky.com/
Avast	www.avast.com/
DrWeb	www.freedrweb.com/
Quick Heal	www.quickheal.com/
Ikarus	www.ikarussecurity.com/
VIPRE	www.vipreantivirus.com/
AVware	www.avware.com.br/
Microsoft	https://www.microsoft.com/
Fortinet	www.fortinet.com/
AegisLab	www.aegislab.com/
ClamAV	www.clamav.net/
Cyren	www.cyren.com/
Rising	http://rising-antivirus-free-edition.en.lo4d.com/
Symantec	www.norton.com/Symantec_Security
TrendMicro-HouseCall	housecall.trendmicro.com/
Comodo	https://www.comodo.com/
Qihoo-360	www.360safe.com/
TrendMicro	www.trendmicro.com/
Tencent	www.westcoastlabs.com/
McAfee	www.mcafee.com/
Zillya	zillya.com/
Jiangmin	jiangmin-antivirus.en.softonic.com/
VBA32	anti-virus.by/en/
F-Prot	www.f-prot.com/
Zoner	https://www.zonerantivirus.cz/
Kingsoft	http://www.ksoffice.net/
K7GW	https://www.k7computing.com/
Baidu-International	antivirus.baidu.com/
Norman	www.norman.com/
ALYac	asia.alyac.com/
TotalDefense	www.totaldefense.com/
McAfee-GW-Edition	www.mcafeeworks.com/Web-Gateway.asp
Agnitum	www.agnitum.com/
ViRobot	www.hauri.net/
Panda	www.pandasecurity.com/
Antiy-AVL	www.antiy.net/
nProtect	avs.nprotect.com/en/
K7AntiVirus	https://www.k7computing.com/
TheHacker	www.hacksoft.com.pe/
ByteHero	www.bytehero.com/
Bkav	https://www.bkav.com/
CMC	http://www3.cmcinfosec.com/
MalwareBytes	https://www.malwarebytes.org/
SUPERAntiSpyware	http://www.superantispyware.com/

Table II
THE 57 ANTIMALWARE EVALUATED IN THE CASE STUDY

antimalware	*#pre*	*#post*
Alibaba	565	578
F-Secure	4723	4767
AVG	4734	4432
ESET-NOD32	4550	4169
Avira	4749	4054
AhnLab	4615	3934
Sophos	4719	3875
GData	4747	3853
BitDefender	4735	3848
Ad-Aware	3726	3843
Emsisoft	4594	3725
MicroWorld-eScan	4742	3782
NANO-Antivirus	4716	3702
Kaspersky	4689	3543
Avast	4175	3338
DrWeb	4610	3240
Quick Heal	3697	2962
Ikarus	4467	2715
VIPRE	4767	2093
AVware	4636	2034
Microsoft	2437	1937
Fortinet	4458	1920
AegisLab	2988	1074
ClamAV	2122	1637
Cyren	4766	1629
Rising	2042	1544
Symantec	3255	1391
TrendMicro-HouseCall	3953	1292
Comodo	4711	1268
Qihoo-360	4486	1116
TrendMicro	3392	1080
Tencent	4522	728
McAfee	4784	600
Zillya	646	557
Jiangmin	4255	547
VBA32	2420	536
F-Prot	4692	505
Zoner	3933	389
Kingsoft	4267	367
K7GW	221	15
Baidu-International	4157	222
Norman	1058	218
ALYac	114	121
TotalDefense	1960	207
McAfee-GW-Edition	320	135
Agnitum	425	119
ViRobot	116	112
Panda	185	97
Antiy-AVL	83	82
nProtect	59	59
K7AntiVirus	150	36
TheHacker	8	2
ByteHero	0	0
Bkav	740	0
CMC	2	0
MalwareBytes	0	0
SUPERAntiSpyware	2	0

Table III
NUMBER OF SAMPLES PROPERLY RECOGNIZED BY ANTIMALWARE BEFORE AND AFTER TRANSFORMATIONS

A. RQ1 response

Table III shows the number of samples that are properly recognized by antimalware before and after transformations. Results are expressed for each antimalware product in terms of:

- number of samples properly recognized as malware before the transformation (*#pre*):
- number of transformed samples properly recognized as malware (*#post*):

In order to simplify the reading, we grouped the antimalware in the following equivalence classes according to the number of malware samples correctly detected:

- *C1*: it includes the antimalware that correctly recognized less than 1,000 malware instances;
- *C2*: it includes the antimalware that correctly recognized between 1,000 and 2,000 malware instances;
- *C3*: it includes the antimalware that correctly recognized between 2,000 and 3,000 malware instances;

- *C4*: it includes the antimalware that correctly recognized between 3,000 and 4,000 malware instances;
- *C5*: it includes the antimalware that correctly recognized between 4,000 and 4,500 malware instances;
- *C6*: it includes the antimalware that correctly recognized over 4,500 malware instances;

Obviously, antimalware tools falling in *Ci+1* class exhibit a better performance than antimalware falling into *Ci* class.

Table IV contains the number of antimalware tools that fall into each equivalence class with both (i) original samples (*#AM_pre*) and (ii) transformed ones (*#AM_post*).

class	*#AM_pre*	*#AM_post*
C1	17	26
C2	2	11
C3	5	12
C4	6	4
C5	10	3
C6	16	1

Table IV
NUMBER OF ANTIMALWARE ASSOCIATED WITH THE CORRESPONDENT EQUIVALENCE CLASS

Only one antimalware belongs to *C6* equivalence class with transformed samples, i.e. F-Secure is the only one that recognizes over 4,500 malware instances; while 26 antimalware tools appear to belong to *C1* equivalence class, i.e. recognize less than 1,000 malware, using transformed dataset (with original dataset, antimalware belonging to the same class were 17). Moreover after the transformations have been applied, we can notice a growth of the antimalware falling in the *C2* and *C3* classes of 550% and 250% respectively and, in the same time, a decrease of 150% and 333% for antimalware falling in the *C4* and *C5* classes respectively.

These results highlight that most of the evaluated antimalware tools are not able to correctly detect transformed known malware. This calls for free and commercial antimalware of adopting stronger detection algorithms able to identify malicious code obfuscations.

On a restricted group of antimalware (ALYac, Alibaba and F-Secure) we obtain a surprising result: these antimalware solutions show better performances in identifying transformed samples.

B. RQ2 response

Table V shows the results for each malware family.
Results are expressed in terms of:

- *pop*: family population;
- *#pre*: number of original samples (i.e. not obfuscated malware) recognized as *clean* by most of antimalware;
- *#post*: number of transformed samples (i.e. obfuscated malware) recognized as *clean* by most of antimalware;
- *%trusted:* percentage of transformed samples recognized as *clean* by most of antimalware with respect to the total family population;

For reasons of space we show in Table V only the families for which the value of *%trusted* is less than 100%. In fact, for the most of the considered malware families, the code transformations we applied made the majority of antimalware totally ineffective in detecting malicious apps.

Family	*pop*	*#pre*	*#post*	*%trusted*
FakeInstaller	919	1	918	99.89
Plankton	555	59	554	99.81
GinMaster	269	0	268	99.62
Geinimi	83	0	82	98.79
DroidDream	74	0	73	98.64
Adrd	72	0	70	97.22
Jifake	28	0	26	92.85
Stealer	14	0	13	92.85
Fidall	3	0	2	66.66
Kmin	95	7	59	62.1
JSmsHider	2	1	1	50
Dogowar	2	0	1	50
Gamex	6	1	3	50
EICAR	4	1	2	50
SMSZombie	10	0	2	20
DroidKungFu	561	0	102	18.18
Xsider	15	0	1	6.66
BaseBridge	310	1	16	5.16
ExploitLinuxLotoor	61	0	2	3.27
Exploit.RageCage	1	0	0	0

Table V
PERCENTAGE OF FAMILIES' MEMBERS NOT RECOGNIZED AFTER TRANSFORMATIONS

We briefly describe the malicious payload behavior for the most populous families in Table V:

- the *FakeInstaller* family is server-side polymorphic, i.e. the server provide different `.apk` files for the same URL request;
- the family *Plankton* represents the first example of polymorphic malware for Android;
- the *GinMaster* samples start malicious services as soon as it receives a BOOT_COMPLETED or USER_PRESENT intent;
- the *Geinimi* is the first Android malware in the wild that displays botnet-like capabilities;
- the *DroidDream* family gain root access to device to access unique identification information;
- the *Adrd* family is very close to Geinimi but with less server-side commands;
- the *Jifake* family sends SMS messages to premium-rate numbers;
- the *Stealer* family is able to steal sensitive information including wi-fi and browser passwords;
- the *Kmin* family send personal data to a remote server;
- the *DroidKungFu* installs a backdoor that allows attackers to access the smartphone;
- the *XSider* samples change the mobile device settings and gather information about the device;
- the *ExploitLinux-Lotoor* samples are rooting exploit that target Android devices up to 2.3 version in order to gain root privileges;
- *BaseBridge* malware send information to a remote server running more malicious services in back-

ground.

To answer RQ2, we define the following equivalence classes based on the percentage of transformed samples wrongly identified as trusted for family (*%trusted*):

- *C1*: percentage ranging from 0% to 50% (not included);
- *C2*: percentage ranging from 50% to 90% (not included);
- *C3*: percentage ranging from 90% to 95% (not included);
- *C4*: percentage ranging from 95% to 98% (not included);
- *C5*: percentage ranging from 98% to 99%;
- *C6*: percentage equal to 100%;

In table VI are reported the number of families (*#fam*) that fall into the equivalence classes we defined.

class	*#fam*
C1	6
C2	6
C3	2
C4	1
C5	5
C6	158

Table VI
NUMBER OF FAMILIES WITH THE CORRESPONDING EQUIVALENCE CLASS

Most of the families (158/178) belong to *C6* equivalence class: the samples of these families are considered trusted by most of evaluated antimalware. Another alarming result is that, after transformations, only 12 malware families have been properly recognized as malware at least in the 10% of cases by the majority of antimalware: SMSZombie, DroidKungFu, Xsider, BaseBridge, ExploitLinuxLotoor, EICAR, Gamex, Dogowar, JSmsHider, Kmin and Fidall.

Relying on malware families population we notice that the majority of antimalware are more robust against the obfuscation techniques for two families in particular: BaseBridge (for which only 16 samples on 310 have been incorrectly considered as trusted even after the transformations) and ExploitLinuxLotoor (for which only 2 samples on 61 have been incorrectly classified as trusted after the transformations).

On the other hand, for two malware families, among the most populous ones in our dataset, we found that in almost all the cases the detection has failed after the transformations: FakeInstaller (for which in 918/919 of the cases the transformations made the code able to fool the majority of antimalware, while, before transformations, only one sample was incorrectly classified as trusted), and Opfake (for which, after transformations, 608/608 samples evaded the majority of antimalware, while, before them, only 2 samples were incorrectly classified as trusted).

IV. Concluding remarks

Commercial antimalware tools make a large use of signature-based techniques, which allows to recognize known malware, but presents serious problems in identifying malware without knowing the signature and in general zero-day malware.

The main problem of signature-based detection techniques is the widespread introduction of malware before its inclusion in the database of antimalware signatures.

In this paper we evaluate the effectiveness and the robustness of 57 mobile antimalware using simple obfuscation techniques.

Results show that the current mechanism of signature-based detection is usually ineffective in detecting malware if the signatures are not present in the antimalware vendor database; in fact, trivial code transformations alter the signature of malware, preventing antimalware to detect transformed malware, that was correctly detected before transformations.

References

[1] Gdata mobile malware report. https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q1_2015_US.pdf, last visit 15 July 2015.

[2] On the effectiveness of malware protection on android. http://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/deutsch/042013-Technical-Report-Android-Virus-Test.pdf, last visit 20 April 2015.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[4] G. Canfora, A. De Lorenzo, F. Mercaldo, and C. A. Visaggio. Effectiveness of opcode ngrams for detection of multi family android malware. In *4th International Workshop on Security of Mobile Applications (ARES 2015)*, to appear, 2015.

[5] V. Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, Jan 2014. ISSN 1556-6013. doi: 10.1109/TIFS.2013.2290431.

[6] Michael Spreitzenbarth, Florian Echtler, Thomas Schreck, Felix C. Freling, and Johannes Hoffmann. Mobilesandbox: Looking deeper into android applications. In *28th International ACM Symposium on Applied Computing (SAC)*, 2013.

[7] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. Adam: An automatic and extensible platform to stress test android anti-virus systems. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 82–101, 2013.

[8] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*, 2012.