DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks

Vaibhav Rastogi, Yan Chen, and Xuxian Jiang[†]
Northwestern University, [†]North Carolina State University
vrastogi@u.northwestern.edu, ychen@northwestern.edu, jiang@cs.ncsu.edu

ABSTRACT

Mobile malware threats (e.g., on Android) have recently become a real concern. In this paper, we evaluate the state-of-the-art commercial mobile anti-malware products for Android and test how resistant they are against various common obfuscation techniques (even with known malware). Such an evaluation is important for not only measuring the available defense against mobile malware threats but also proposing effective, next-generation solutions. We developed DroidChameleon, a systematic framework with various transformation techniques, and used it for our study. Our results on ten popular commercial anti-malware applications for Android are worrisome: none of these tools is resistant against common malware transformation techniques. Moreover, the transformations are simple in most cases and anti-malware tools make little effort to provide transformation-resilient detection. Finally, in the light of our results, we propose possible remedies for improving the current state of malware detection on mobile devices.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*

General Terms

Security

Keywords

Mobile; malware; anti-malware; Android

1. INTRODUCTION

Mobile computing devices such as smartphones and tablets are becoming increasingly popular. Unfortunately, this popularity attracts malware authors too. It has been reported that on Android, the most popular smartphone platform [11], malware has constantly been on the rise [13]. With the growth of malware, the platform has also seen an evolution of anti-malware tools, with a range of free and paid offerings now available in the official Android app market, Google Play.

In this paper, we aim to evaluate the efficacy of anti-malware tools on Android in the face of various evasion techniques. For example, polymorphism is used to evade detection tools by transforming a malware in different forms ("morphs") but with the same code. Metamorphism is another common technique that can mutate code so that it no longer remains the same but still has the same behavior. For ease of presentation, we use the term polymorphism in this paper to represent both obfuscation techniques. In addition, we use the term 'transformation' broadly, to refer to various polymorphic or metamorphic changes.

Polymorphic attacks have long been a plague for traditional desktop and server systems. While there exist earlier studies on the effectiveness of anti-malware tools on PCs [8], our domain of study is different in that we exclusively focus on mobile devices like smartphones, which require different ways for anti-malware design. Also, malware on mobile devices have recently escalated their evolution but the capabilities of existing anti-malware tools are largely not yet understood. In the meantime, simple forms of polymorphic attacks have already been seen in the wild [26].

To evaluate existing anti-malware software, we develop a systematic framework called *DroidChameleon* with several common transformation techniques that may be used to transform Android applications automatically. Some of these transformations are highly specific to the Android platform only. Based on the framework, we pass known malware samples (from different families) through these transformations to generate new variants of malware, which are verified to possess the originals' malicious functionality. We use these variants to evaluate the effectiveness and robustness of popular anti-malware tools.

Our results on ten popular anti-malware products, some of which even claim resistance against malware transformations, show that all the anti-malware products used in our study have little protection against common transformation techniques. Our results also give insights about detection models used in existing anti-malware and their capabilities, thus shedding light on possible ways for their improvements. We hope that our findings work as a wake-up call and motivation for the community to improve the current state of mobile malware detection.

To summarize, this paper makes the following contributions.

- We systematically evaluate anti-malware products for Android regarding their resistance against various transformation techniques in known malware. For this purpose, we developed DroidChameleon, a systematic framework with various transformation techniques to facilitate anti-malware evaluation.
- We have implemented a prototype of DroidChameleon and used it to evaluate ten popular anti-malware products for Android. Our findings show that all of them are vulnerable to common evasion techniques. Moreover, we find that 90% of the signatures studied do not require static analysis of bytecode.
- Based on our evaluation results, we also explore possible ways to improve current anti-malware solutions. Specifically, we point

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.
Copyright 2013 ACM 978-1-4503-1767-2/13/05 ...$15.00.

out that Android eases developing advanced detection techniques because much code is high-level bytecodes rather than native codes. Furthermore, certain platform support can be enlisted to cope with advanced transformations.

2. BACKGROUND

Android is an operating system for mobile devices such as smartphones and tablets. It is based on the Linux kernel and provides a middleware implementing subsystems such as telephony, window management, management of communication with and between applications, managing application lifecycle, and so on.

Applications are programmed primarily in Java though the programmers are allowed to do native programming via JNI (Java native interface). Instead of running Java bytecode, Android runs Dalvik bytecode, which is produced from Java bytecode. In Dalvik, instead of having multiple `.class` files as in the case of Java, all the classes are packed together in a single `.dex` file.

Android applications are made of four types of components, namely activities, services, broadcast receivers, and content providers. These application components are implemented as classes in application code and are declared in the AndroidManifest (see next paragraph). The Android middleware interacts with the application through these components.

Android application packages are jar files containing the application bytecode as a `classes.dex` file, any native code libraries, application resources such as images, config files and so on, and a manifest, called AndroidManifest. It is a binary XML file, which declares the application package name, a string that is supposed to be unique to an application, and the different components in the application. It also declares other things (such as application permissions) which are not so relevant to the present work. The AndroidManifest is written in human readable XML and is transformed to binary XML during application build.

Only digitally signed applications may be installed on an Android device. Signing keys are usually owned by individual developers and not by a central authority, and there is no chain of trust. All third party applications run unprivileged on Android.

3. FRAMEWORK DESIGN

In this work, we focus on the evaluation of anti-malware products for Android. Specifically, we attempt to deduce the kind of signatures that these products use to detect malware and how resistant these signatures are against changes in the malware binaries. In this paper, we generally use the term transformation to denote semantics preserving changes to a program. Since we are dealing with malware, we only care about the interested semantics such as sending SMS message to a premium number and not things like change of application name in the system logs.

In this work, we develop several different kinds of transformations that may be applied to malware samples while preserving their malicious behavior. Each malware sample undergoes one or more transformations and then passes through the anti-malware tools. The detection results are then collected and used to make deductions about the detection strengths of these anti-malware tools.

We classify our transformations as trivial (which do not require code level changes or changes to meta-data stored in AndroidManifest), those which result in variants that can still be detected by advanced static analyses involving data-flow and control-flow analysis (DSA), and those which can render malware undetectable by static analysis (NSA). In the rest of this section, we describe the different kinds of transformations that we have in the DroidChameleon framework. Where appropriate we give examples, using original and transformed code. Transformations for Dalvik bytecode are given in Smali (as in Listing 1), an intuitive assembly language for Dalvik bytecode.

```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
invoke-static {v10, v11}, Lcom/android/root/Setting;->
     runRootCommand(Ljava/lang/String;Ljava/lang/String;)
     Ljava/lang/String;
move-result-object v7
```

Listing 1: A code fragment from DroidDream malware

3.1 Trivial Transformations

Trivial transformations do not require code-level changes or changes to meta-data stored in AndroidManifest. These transformations are meant to defeat signatures based on whole files (or a part of file that changes simply by reorganizing file sections) or the key used to sign an application package. We have the following two transformations for this purpose.

Repacking. Recall that Android packages are signed jar files. These may be unzipped with the regular zip utilities and then repacked again with tools offered in the Android SDK. Once repacked, applications are signed with custom keys (the original developer keys are not available). Detection signatures that match the developer keys or a checksum of the entire application package are rendered ineffective by this transformation. Note that this transformation applies to Android applications only; there is no counterpart in general on traditional Desktop operating systems although the malware in the latter are known to use sophisticated packers for the purpose of evading anti-malware tools.

Disassembling and Reassembling. The compiled Dalvik bytecode in `classes.dex` of the application package may be disassembled and then reassembled back again. The various items in a dex file may be arranged or represented in different ways and thus a compiled program may be represented in more than one form. Signatures that match the whole `classes.dex` are beaten by this transformation. Signatures that depend on the order of different items in the dex file will also likely break with this transformation. Similar assembling/disassembling also applies to the resources in an Android package and to the conversion of AndroidManifest between binary and human readable formats.

3.2 Transformation Attacks Detectable by Static Analysis (DSA)

The application of DSA transformations does not break all types of static analysis. Specifically, forms of analysis that describe the semantics, such as data flows are still possible. Only simpler checks such as string matching or matching API calls may be thwarted.

Changing Package Name. Every application is identified by a package name unique to the application. This name is defined in the package's AndroidManifest. We change the package name in a given malicious application to another name.

Identifier Renaming. Most class, method, and field identifiers in bytecode can be renamed. We note that several free obfuscation tools such as ProGuard [5] provide identifier renaming. Listing 2 presents an example transformation for code in Listing 1.

```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
invoke-static {v10, v11}, Lcom/hxbvgH/IWNcZs/jFAbKo;->
     axDnBL(Ljava/lang/String;Ljava/lang/String;)Ljava/
     lang/String;
move-result-object v7
```

Listing 2: Code in Listing 1 after identifier renaming

Data Encryption. The dex files contain all the strings and array data that have been used in the code. These strings and arrays may be used to develop signatures against malware. To beat such signatures we can keep these in encrypted form. Listing 3 shows code in Listing 1, transformed by string encryption.

```
const-string v10, "qspgjmf"
invoke-static {v10}, Lcom/EncryptString;->applyCaesar(
     Ljava/lang/String;)Ljava/lang/String;
move-result-object v10
const-string v11, "npvou!.p!sfnpvou!sx!tztufn]ofyju]o"
invoke-static {v11}, Lcom/EncryptString;->applyCaesar(
     Ljava/lang/String;)Ljava/lang/String;
move-result-object v11
invoke-static {v10, v11}, Lcom/android/root/Setting;->
     runRootCommand(Ljava/lang/String;Ljava/lang/String;)
     Ljava/lang/String;
move-result-object v7
```

Listing 3: Code in Listing 1 after string encryption. Strings are encoded with a Caesar cipher of shift +1.

Call Indirections. This transformation can be seen as a simple way to manipulate call graph of the application to defeat automatic matching. Given a method call, the call is converted to a call to a previously non-existing method that then calls the method in the original call. This can be done for all calls, those going out into framework libraries as well as those within the application code. This transformation may be seen as trivial function outlining (see function outlining below).

Code Reordering. Code reordering reorders the instructions in the methods of a program. This transformation is accomplished by reordering the instructions and inserting `goto` instructions to preserve the runtime execution sequence of the instructions. Listing 4 shows an example reordering.

```
goto :i_1
:i_3
invoke-static {v10, v11}, Lcom/android/root/Setting;->
     runRootCommand(Ljava/lang/String;Ljava/lang/String;)
     Ljava/lang/String;
move-result-object v7
goto :i_4  # next instruction
:i_2
const-string v11, "mount -o remount rw system\nexit\n"
goto :i_3
:i_1
const-string v10, "profile"
goto :i_2
```

Listing 4: Code in Listing 1 reverse ordered

Junk Code Insertion. These transformations introduce code sequences that are executed but do not affect rest of the program. Detection based on analyzing instruction (or opcode) sequences may be defeated by junk code insertion. Junk code may constitute simple `nop` sequences or more sophisticated sequences and branches that actually have no effect on the semantics.

Encrypting Payloads and Native Exploits. In Android, native code is usually made available as libraries accessed via JNI. However, some malware such as DroidDream also pack native code exploits meant to run from a command line in non-standard locations in the application package. All such files may be stored encrypted in the application package and be decrypted at runtime. Certain malware such as DroidDream also carry payload applications that are installed once the system has been compromised. These payloads may also be stored encrypted. We categorize payload and exploit encryption as DSA because signature based static detection is still possible based on the main application's bytecode. These are easily implemented and have been seen in practice as well (e.g., DroidKungFu malware uses encrypted exploit).

Function Outlining and Inlining. In function outlining, a function is broken down into several smaller functions. Function inlining involves replacing a function call with the entire function body. These are typical compiler optimization techniques. However, outlining and inlining can be used for call graph obfuscation also.

Other Simple Transformations. There are a few other transformations as well, specific to Android. Debug information, such as source file names, local and parameter variable names, and source line numbers may be stripped off. Moreover, non-code files and resources contained in Android packages may be renamed or modified.

Composite Transformations. Any of the above transformations may be combined with one another to generate stronger obfuscations. While compositions are not commutative, anti-malware detection results should be agnostic to the order of application of transformations for the cases discussed above.

3.3 Transformation Attacks Non-Detectable by Static Analysis (NSA)

These transformations can break all kinds of static analysis. Some encoding or encryption is typically required so that no static analysis scheme can infer parts of the code. Parts of the encryption keys may even be fetched remotely. In this scenario, interpreting or emulating the code (i.e., dynamic analysis) is still possible but static analysis becomes infeasible.

Reflection. The Java reflection API allows a program to invoke a method by using the name of the methods. We can convert any method call into a call to that method via reflection. This makes it difficult to analyze statically which method is being called. A subsequent encryption of the method name can make it impossible for any static analysis to recover the call.

Bytecode encryption. Code encryption tries to make the code unavailable for static analysis. The relevant piece of the application code is stored in an encrypted form and is decrypted at runtime via a decryption routine. Code encryption has long been used in polymorphic viruses; the only code available to signature based antivirus applications remains the decryption routine, which is typically obfuscated. To accomplish this the majority of the malware code may be stored in an encrypted dex file that is decrypted and loaded dynamically through a user-defined class loader.

4. IMPLEMENTATION

Apart from function outlining and inlining, we applied all other DroidChameleon transformations to the malware samples. We have implemented most of the transformations so that they may be applied automatically to the application. Automation implies that the malware authors can generate polymorphic malware at a very fast pace.

We utilize the Smali/Baksmali [6] and its companion tool Apktool [1] for our implementation. Our code-level transformations are implemented over Smali. Moreover, disassembling and assembling transformation uses Apktool. This has the effect of repacking, changing the order and representation of items in the `classes.dex` file, and changing the AndroidManifest (while preserving the semantics of it). All other transformations in our implementation (apart from repacking) make use of Apktool to unpack/repack application packages.

5. RESULTS

We begin by describing our anti-malware and malware dataset, followed by our methodology, and then discuss our findings. We

Table 1: Anti-malware products evaluated. All tools collected in February 2013.

Vendor	Product	Package name	Version	# downloads
AVG	Antivirus Free	com.antivirus	3.1	50M-100M
Symantec	Norton Mobile Security	com.symantec.mobilesecurity	3.3.0.892	5M-10M
Lookout	Lookout Mobile Security	com.lookout	8.7.1-EDC6DFS	10M-50M
ESET	ESET Mobile Security	com.eset.ems	1.1.995.1221	500K-1M
Dr. Web	Dr. Web anti-virus Light	com.drweb	7.00.3	10M-50M
Kaspersky	Kaspersky Mobile Security	com.kms	9.36.28	1M-5M
Trend micro	Mobile Security Personal Ed.	com.trendmicro.tmmspersonal	2.6.2	100K-500K
ESTSoft	ALYac Android	com.estsoft.alyac	1.3.5.2	5M-10M
Zoner	Zoner Antivirus Free	com.zoner.android.antivirus	1.7.2	1M-5M
Webroot	Webroot Security & Antivirus	com.webroot.security	3.1.0.4547	500K-1M

Table 2: Malware samples used for testing anti-malware tools

Family	Package name	SHA-1 code	Date found	Remarks
DroidDream	com.droiddream. bowling-time	72adcf43e5f945ca9f72 064b81dc0062007f0fbf	03/2011	Root exploit
Geinimi	com.sgg.spp	1317d996682f4ae4cce6 0d90c43fe3e674f60c22	10/2011	Information exfiltration; bot-like capabilities
Fakeplayer	org.me.androidappli-cation1	1e993b0632d5bc6f0741 0ee31e41dd316435d997	08/2010	SMS trojan
Bgserv	com.android.vending. sec-tool.v1	bc2dedad0507a916604f 86167a9fa306939e2080	03/2011	Information exfiltration; bot-like capabilities; SMS trojan
BaseBridge	com.keji.unclear	508353d18cb9f5544b1e d1cf7ef8a0b6a5552414	05/2011	Root exploit; SMS trojan packed as payload
Plankton	com.crazyapps.angry. birds.rio.unlocker	bee2661a4e4b347b5cd2 a58f7c4b17bcc3efd550	06/2011	Dynamic code loading

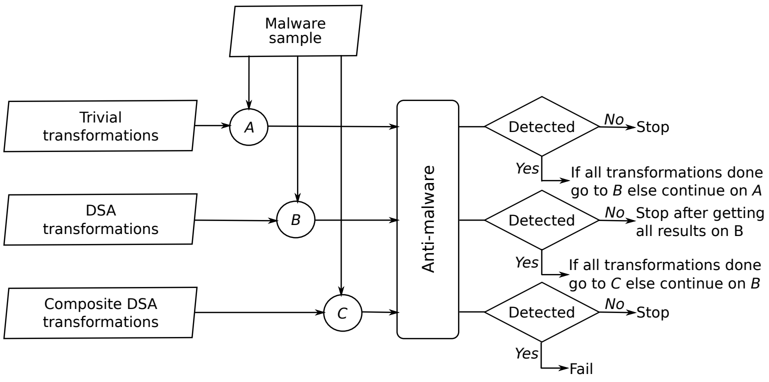
Malware sample
Trivial transformations
A
DSA transformations
B
Composite DSA transformations
C
Anti-malware
Detected
No
Stop
Yes
If all transformations done go to B else continue on A
Detected
No
Stop after getting all results on B
Yes
If all transformations done go to C else continue on B
Detected
No
Stop
Yes
Fail

Figure 1: Evaluating anti-malware

Table 3: Key to Table 4. Transformations coded with single letters are trivial transformations. All others are DSA. We did not need NSA transformations to thwart anti-malware tools.

Code	Technique
P	Repack
A	Dissassemble & assemble
RP	Rename package
EE	Encrypt native exploit or payload
RI	Rename identifiers
RF	Rename files
ED	Encrypt strings and array data
CR	Reorder code
CI	Call indirection
JN	Insert junk code
All transformations contain P	
All transformations except P contain A	

evaluated ten anti-malware tools, which are listed in Table 1. We selected the most popular products; in addition, we included Kaspersky, ESET, and Trend Micro, which were then not very popular but are well established vendors in the security industry. We had to omit a couple of products in the most popular list because they would fail to identify many original, unmodified malware samples we tested. All the products were downloaded directly from the official Android app market, Google Play, in February 2013.

Our malware set is summarized in Table 2. We used a few criteria for choosing malware samples. First, all the anti-malware tools being evaluated should detect the original samples. Second, the malware samples should be sufficiently old so that signatures against them are well stabilized. All the samples in our set were discovered in or before October 2011. All the samples are publicly available on Contagio Minidump [22]. Finally, as seen in the table, the set spans over multiple malware kinds, from root exploits to information stealing.

As has already been discussed, we transform malware samples using various techniques discussed in Section 3 and pass them through anti-malware tools we evaluate. Our methodology is depicted in Figure 1. For every malware-antimalware pair, we begin testing with trivial transformations and then proceed with transformations that are more complex. Each transformation is applied to a malware sample (of course, some like exploit encryption apply only in certain cases) and the transformed sample is passed through anti-malware. If detection breaks with trivial transformations, we stop (all DSA and NSA transformations also result in trivial transformations). Next, we apply all the DSA transformations. If detection still does not break, we apply combinations of DSA transformations. In general there is no well-defined order in which transformations should be applied (in some cases a heuristic works; for example, malware that include native exploits are likely to be detected based on those exploits). Fortunately, in our study, we did not need to apply combinations of more than two transformations to break detection. When applying combinations of transformations, we stopped when detection broke.

Our results with all the malware samples are summarized in Table 4. This table gives the minimal transformations necessary to evade detection for malware-anti-malware pairs. For example, DroidDream requires both exploit encryption and call indirection to evade Dr. Web's detection. These minimal transformations also give insight into what kind of detection signatures are being used. We next describe our key findings in the light of the detection results.

Table 4: Evaluation summary. Please see Table 3 for key. '+' indicates the composition of two transformations.

	DroidDream	Geinimi	Fakeplayer	Bgserv	BaseBridge	Plankton
AVG	RP	RI	RP + RI	RI	RI	RP + RI
Symantec	RI	RI	RP + RI	RI + ED	ED	P
Lookout	P	RI + ED	RP + RI	RI + ED	EE + ED	RI
ESET	RI + EE	ED	RI	RI	EE + ED	RI + ED
Dr. Web	EE + CI	CI	CI	CI	EE + CI	CI
Kaspersky	EE + ED	RI	RI	RI + ED	EE + ED	A
Trend M.	EE + RF	RI	A	A	EE + RF	A
ESTSoft	RP	RP	RP	RP	RP	RP
Zoner	A	RI	A	A	A	RI
Webroot	RI	RI	RP	RI	RP	RI

Finding 1 *All the studied anti-malware products are vulnerable to common transformations.* All the transformations appearing in Table 4 are easy to develop and apply, redefine only certain syntactic properties of the malware, and are common ways to transform malware. Transformations like identifier renaming and data encryption are easily available using free and commercial tools [4, 5]. Exploit and payload encryption is also easy to achieve. Such transformations may already be seen in the wild in current malware, e.g., Geinimi variants have encrypted strings [19] and DroidKungFu malware uses encrypted exploit code [3].

We found that only Dr. Web uses a somewhat more sophisticated algorithm for detection. Our findings indicate that the general detection scheme of Dr. Web is as follows. The set of method calls from every method is obtained. These sets are then used as signatures and the detection phase consists of matching these sets against sets obtained from the sample under test.

Finding 2 *At least 43% signatures are not based on code-level artifacts.* That is, these are based on file names, checksums (or binary sequences) or information easily obtained by the PackageManager API. We also found all AVG signatures to be derived from the content of AndroidManifest only (and hence that of the PackageManager API). Changing component names in AndroidManifest while keeping the code same was sufficient to break AVG's detection.

Finding 3 *90% of signatures do not require static analysis of bytecode. Only one of ten anti-malware tools was found to be using static analysis.* Names of classes, methods, and fields, and all the strings and array data are stored in the `classes.dex` file as they are and hence can be obtained by content matching. The only signatures requiring static analysis of bytecode are those of Dr. Web because it extracts API calls made in various methods.

6. DEFENSES

6.1 Semantics-based Malware Detection

We point out that owing to the use of bytecodes, which contain high-level structural information, analyses of Android applications becomes much simpler than those of native binaries. Hence, semantics-based detection schemes could prove especially helpful in the case of Android. For example, Christodorescu et al. [9] describe a technique for semantics based detection. Their algorithms are based on unifying nodes in a given program with nodes in a signature template (nodes may be understood as abstract instructions), while preserving def-use paths described in the template. Since this technique is based on data flows rather than a superficial property of the program such as certain strings or names of methods being defined or called, it is not vulnerable to any of the transformations (all of which are trivial or DSA) that show up in Table 4.

Semantics-based detection is quite challenging for native codes; their analyses frequently encounters issues such as missing information on function boundaries, pointer aliasing, and so on [16, 25]. Bytecodes, on the other hand, preserve much of the source-level information, thus easing analysis. We therefore believe that anti-malware tools have greater incentive to implement semantic analysis techniques on Android bytecodes than they had for developing these for native code.

6.2 Support from Platform

Note that the use of code encryption and reflection (NSA transformations) can still defeat the above scheme. Code encryption does not leave much visible code on which signatures can be developed. The use of reflection simply hides away the edges in the call graph. If the method names used for reflective invocations are encrypted, these edges are rendered completely opaque to static analysis. Furthermore, it is possible to use function outlining to thwart any forms of intra-procedural analysis as well. Owing to these limitations, the use of dynamic monitoring is essential.

Recall that anti-malware tools in Android are unprivileged third party applications. This impedes many different kinds of dynamic monitoring that may enhance malware detection. We believe special platform support for anti-malware applications is essential to detect malware amongst stock Android applications. This can help malware detection in several ways. For example, a common way to break evasion by code encryption is to scan the memory at runtime. The Android runtime could provide all the classes loaded using user-defined class loaders to the anti-malware application. Once the classes are loaded, they are already decrypted and anti-malware tools can analyze them easily.

Google Bouncer performs offline dynamic analysis for malware detection [18]. Such scanning however has its own problems, ranging from detection of the dynamic environment to the malicious activity not getting triggered in the limited time for which the analysis runs; Bouncer is no exception to this [21, 27]. We therefore believe offline emulation must be supplemented by strong static analysis or real-time dynamic monitoring.

7. RELATED WORK

Evaluating Anti-malware Tools. Zheng et al. [28] also studied the robustness of anti-malware against Android malware recently. They implement a subset of our transformations, use them to generate several malware variants, and test these on VirusTotal, a web-service that tests submitted samples against over 40 anti-virus products. Their results however only show the change in overall detection percentages as the transformations are applied. Our results are much stronger in that we can show that all anti-malware tools actually succumb for all malware samples tested. Moreover, we also deduce the weaknesses and strengths of some of the products.

Christodorescu and Jha [8] conducted a study similar to ours on desktop anti-malware applications nine years ago. They also arrived at the conclusion that these applications have low resilience against malware obfuscation. Our study is based on Android anti-malware, and we include several aspects in our study that are unique to Android. Furthermore, our study comes after much research on obfuscation resilient detection, and we would expect the proposed techniques to be readily integrated into new commercial products. Finally, our study is orthogonal to studies about completeness of detection of anti-malware tools such as those by AV-Test.org [2].

Obfuscation Techniques. Collberg et al. [12] review and propose different types of obfuscations. DroidChameleon provides only a few of the transformations proposed by them. Nonetheless, the set of transformations provided in DroidChameleon is comprehensive in the sense that they can break typical static detection techniques used by anti-malware. Off-the-shelf tools like Proguard [5] and Klassmaster [4] provide renaming of classes and class members, flow obfuscation, and string encryption. While the goal of these tools is to evade manual reverse engineering, we aim at thwarting analysis by automatic tools.

Obfuscated Malware Detection. Obfuscation resilient detection is based on semantics rather than syntac. As discussed earlier, Christodorescu et al. [9] present one such technique. Christodorescu et al. [10] and Fredrikson et al. [14] attempt to generate semantics based signatures by mining malicious behavior automatically. Kolbitsch et al. [17] also propose similar techniques. The last three works are for behavior-based detection and use different behavior representations such as data dependence graphs and information flows between system calls. Due to lower privileges for anti-malware tools on Android, these approaches cannot directly apply to these tools presently.

Smartphone Malware Research. Many works have been done towards discovery and characterization of smartphone malware [7, 15, 20, 23, 24, 29, 30]. Our work is distinct from these as we try to evaluate the efficacy of existing tools against transformed malware.

8. CONCLUSION

We evaluated ten anti-malware products on Android for their resilience against malware transformations. To facilitate this, we developed DroidChameleon, a systematic framework with various transformation techniques. Our findings show that all the anti-malware products evaluated are susceptible to common evasion techniques. Finally, we explored possible ways to improve the current situation and develop next-generation solutions.

We refer the interested readers to http://list.cs.northwestern.edu/mobile for further information about this work, such as more detailed technical reports.

References

[1] Android-apktool: A tool for reengineering Android apk files. http://code.google.com/p/android-apktool/.

[2] Test: Malware Protection for Android, March 2012. http://www.av-test.org/en/tests/android/.

[3] DroidKungFu. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html.

[4] Zelix Klassmaster. http://www.zelix.com/klassmaster/.

[5] ProGuard. http://proguard.sourceforge.net/.

[6] Smali: An assembler/disassembler for Android's dex format . http://code.google.com/p/smali/.

[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

[8] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[9] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, 2005.

[10] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6th ACM ESEC-FSE*, 2007.

[11] CNET, February 2013. http://news.cnet.com/8301-1035_3-57569402-94/android-ios-combine-for-91-percent-of-market/.

[12] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[13] F-Secure. Mobile Threat Report Q3 2012. http://www.f-secure.com/static/doc/labs_global/Research/Mobile%20Threat%20Report%20Q3%202012.pdf.

[14] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[15] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, 2012.

[16] L. Harris and B. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5): 63–68, 2005.

[17] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, 2009.

[18] H. Lockheimer. Android and security, February 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[19] Lookout. Geinimi Trojan Technical Analysis. http://blog.mylookout.com/blog/2011/01/07/geinimi-trojan-technical-analysis/.

[20] Y. Nadji, J. Giffin, and P. Traynor. Automated remote repair for mobile malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[21] J. Oberheide. Dissecting android's bouncer, June 2012. https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/.

[22] M. Parkour. Contagio Mobile. Mobile Malware Mini Dump. http://contagiominidump.blogspot.com/.

[23] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[24] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of ACM CODASPY 2013*, February 2013.

[25] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008.

[26] Symantec. Server-side Polymorphic Android Applications. http://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications.

[27] R. Whitwam. Circumventing Google's Bouncer, Android's anti-malware system, June 2012. http://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system.

[28] M. Zheng, P. Lee, and J. Lui. Adam: An automatic and extensible platform to stress test android anti-virus systems. *DIMVA*, July 2012.

[29] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. *Security and Privacy, IEEE Symposium on*, 2012.

[30] 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 Network and Distributed System Security Symposium*, 2012.