NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science

Evaluating Android Anti-malware against Transformation Attacks

March 2013

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

Technical Report NU-EECS-13-01

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, a majority of them can be trivially defeated by applying slight transformation over known malware with little effort for malware authors. Finally, in the light of our results, we propose possible remedies for improving the current state of malware detection on mobile devices.

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, a majority of them can be trivially defeated by applying slight transformation over known malware with little effort for malware authors. Finally, in the light of our results, we propose possible remedies for improving the current state of malware detection on mobile devices.**

I. Introduction

Mobile computing devices such as smartphones and tablets are becoming increasingly popular. Unfortunately, this popularity attracts malware authors too. In reality, mobile malware has already become a serious concern. It has been reported that on Android, one of the most popular smartphone platforms [1], malware has constantly been on the rise and the platform is seen as “clearly today’s target” [2], [3]. 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 a common obfuscation technique that has been widely used by malware 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 [4], our domain of study is different in that we exclusively focus on mobile devices like smartphones that 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, there are warnings that Android malware will become more sophisticated, we will soon see polymorphic malware, and they will be able to quickly propagate from device to device using poisoned SMS messages and social network postings to infected links [5]. In fact, simple forms of polymorphic attacks have already been seen in the wild [6]. It is thus imperative for mobile security systems to have good defenses against polymorphic strains.

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. The techniques themselves are simple. The fact that even without much technical difficulty, we can evade anti-malware tools, highlights the seriousness of the problem. Many of them succumb to even trivial transformations such as repacking or reassembling that do not involve any code-level transformation. This is in contrast to the general understanding, also substantiated by reports from the industry [7], [8], that mobile anti-malware tools work quite well. Our evaluation dataset includes products that these reports claim to be perfect or nearly perfect. 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.

We emphasize that making judgment which anti-malware product is the best is a non-goal for this paper. There are other important characteristics of anti-malware, such as the completeness of the signature database and resource consumption, that we do not evaluate. Additionally, security vendors typically package malware detection with other functionalities such as locating missing device or filtering spam SMS together in their offerings. Evaluating these functionalities remains beyond the scope of this paper.

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. Apart from general transformations, we also develop transformations that are specific to the Android platform.
- 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.
- We studied the evolution of anti-malware tools over a period of one year. Our findings show that some anti-malware tools have tried to strengthen their signatures with a trend towards content-based signatures while previously they were evaded by trivial transformations not involving code-level changes. The improved signatures are however still shown to be easily evaded.
- Based on our evaluation results, we also explore possible ways to improve current anti-malware solutions. Specifically, we point out that Android eases advanced static analyses because much of the Android application code is high-level bytecodes rather than native codes. Hence, anti-malware products could implement the already proposed semantics-based approaches for malware detection more easily for mobile platforms than for PCs where most applications are native binaries. Furthermore, certain platform support (in terms of offering higher privileges to anti-malware) can be enlisted to cope with advanced transformations.

In contrast with a closely related work [9], DroidChameleon is much more comprehensive with many more transformation techniques and with complete evasion of anti-malware tools, which is not even attempted in this work. Further details are offered in Section VIII.

The rest of this paper is organized as follows. We present in Section II the necessary background and detail in Section III the DroidChameleon design. We then provide implementation details in Section IV and summarize our malware and anti-malware data sets in Section V. After that, we present our findings in Section VI, followed by a brief discussion in Section VII on how to improve current anti-malware solutions. Finally, we examine related work in Section VIII and conclude in Section IX.

II. 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. Third party applications run unprivileged on Android. The rest of this section will cover some background on the Android middleware and application fundamentals, application distribution, Android anti-malware, and signatures for malware detection.

A. Android Fundamentals

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 by the application build toolchain from Java bytecode. Dalvik is a virtual machine designed to run in low-memory environments and is similar to the Java Virtual Machine (JVM) with the most notable difference being that it is register based (JVM is stack based). Most of the JVM concepts such as classes, class loaders, reflection, and so on are adopted as specified by the Java Language Specification in the Dalvik virtual machine. In Dalvik, instead of having multiple `.class` files as in the case of Java, all the classes are packed together in a single `.dex` (Dalvik Executable) file to minimize redundant strings and other constants. The dex file format keeps the Dalvik bytecode and specifies the organization of the various sections and items in the file. There are separate sections for keeping strings, class definitions, code items, and so on.

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. The reader is referred to the official Android Documentation for detail on these.

Android application packages are jar files[1] 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. Application packages are signed similar to the signing of a jar file. Signing is only for the purpose of enabling better sharing among applications from the same developer and recognizing packages that come from the device vendor (such packages may have more privileges) and not verifying trust in the application. Signing keys are thus owned by individual developers and not by a central authority, and there is no chain of trust.

[1] Java Archive format, which is really a zip file format

B. Android Anti-malware Solutions

With the proliferation of malware, there are now scores of both free and paid anti-malware products available in the official Android market. Many are from obscure developers while well-established, mainstream antivirus vendors offer others.

In order to get an insight on the workings of the anti-malware products, we briefly describe the necessary parts of the Android security model. Android achieves application sandboxing by means of Linux UIDs. Every application (with a few exceptions relating to how applications are signed) is given a separate UID and most of the application resources remain hidden from other UIDs.

Android anti-malware products are treated as ordinary third party applications and have no additional privileges over other applications. This is in contrast with the situation on traditional platforms such as Windows and Linux where antivirus applications run with administrator privileges. An important implication of this is that these anti-malware tools are mostly incapable of behavioral monitoring and do not have access to the private files of the application. The original application packages however remain intact and are readable by all applications. (Copy protected application packages are not readable by all applications but this feature is deprecated; paid applications are reportedly kept encrypted since Android 4.1. Note that malware have not been found in paid apps.) These application packages may thus be used for static, signature-based malware detection. Moreover, Android provides a broadcast when a new application is installed. All the anti-malware applications we study have the ability to scan applications automatically immediately following their installation, most likely by listening to this broadcast.

Android also provides a PackageManager API, which allows applications to retrieve all the installed packages. The API also allows getting the signing keys of these packages and the information stored in their AndroidManifest such as the package name, names of the components declared, the permissions declared and requested, and so on. Anti-malware applications have the opportunity to use information from this API as well for malware detection.

C. Malware Detection Signatures

While developing malware transformations, it is important to consider what kind of signatures anti-malware tools may use against malware. Signatures have traditionally been in the form of fixed strings and regular expressions. Anti-malware tools may also use chunks of code, an instruction sequence or API call sequence as signatures. Signatures that are more sophisticated require a deeper static analysis of the given sample. The fundamental techniques of such an analysis comprise data and control flow analysis. Analysis may be restricted within function boundaries (intra-procedural analysis) or may expand to cover multiple functions (inter-procedural analysis).

III. 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. We next define transformations more specifically.

Let $\mathbf{P}$ be the set of all programs. A transformation is a mapping $\tau : \mathbf{P} \rightarrow \mathbf{P}$ that preserves the relevant semantics of the program. Note that we do not require all semantic behaviors to be preserved; we instead look for preserving only an interesting subset of behaviors of a given program. In case of malware, this interesting subset is the malicious behavior. For example, when a transformation corresponds to changing the package name of an application, the system logs about that application may show a different package name, but this behavior is not relevant. On the other hand, sending out a text message to a premium rate number without user consent is a relevant behavior when studying malware. Clearly, if two transformations preserve the relevant semantics, so will their composition.

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.

The transformation set in the DroidChameleon framework is comprehensive in the sense that we can expect to beat any static program analysis technique with these transformations. We also provide some Android-specific transformations (repacking and package renaming) which would give us important insights about the workings of Android anti-malware. Moreover, some transformations such as renaming identifiers and reflection do not apply to native code files typical to PCs. We classify our transformations as trivial (which do not require code level changes), those which result in variants that can still be detected by static 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 and very similar to Jasmin assembly language for Java 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

A. Trivial Transformations

Trivial transformations do not require code-level changes. We have the following transformations in this category.

1) 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 for Windows applications although the malware in the latter operating systems are known to use sophisticated packers for the purpose of evading anti-malware tools.

2) 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 (classes, methods, strings, and so on) in a dex file may be arranged or represented in more than one way and thus a compiled program may be represented in different forms. 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) 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. Package names of apps are concepts unique to Android and hence similar transformations do not exist in other systems.

B. 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. Except for certain forms (depending on the accuracy and detail of information needed) of data flow analysis and control flow analysis, we can expect other forms of detection described in Section II-C to be vulnerable to transformations described in this section.

1) Identifier Renaming: Similar to Java bytecode, Dalvik bytecode stores the names of classes, methods, and fields. It is possible to rename most of these identifiers without changing the semantics of the code. Constructors and methods that override super-class methods can however not be renamed. In general, such transformations apply only to source code or bytecode (which preserve symbolic information) and not to native code. We note that several free obfuscation tools such as ProGuard [10] 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

2) Data Encoding: 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 transform the dex file as follows. All the strings are stored in an encoded form, such as by the application of a simple Caesar cipher. Any access to an encoded string is immediately followed by a call to a routine for decoding the string. As an illustration, Listing 3 shows code in Listing 1, transformed by string encoding.

```
const-string v10, "qspgjmf"
invoke-static {v10}, Lcom/EncodeString;->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/EncodeString;->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 encoding. Strings are encoded with a Caesar cipher of shift +1.

The initialization data for arrays of primitive types is stored as bytes in the dex file. We encode these bytes using simple XOR cipher. Any operation to fill arrays with data is immediately followed by a call to a routine to decode the newly filled array.

3) 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).

4) Code Reordering: Code reordering reorders the instructions in the methods of a program. This transformation targets detection schemes that rely on the order of the instructions, based on either the whole instructions, or part of the instructions such as opcodes. This transformation is accomplished by reordering the instructions and inserting `goto` instructions to preserve the runtime execution sequence of the instructions. We note that even though the Java language does not have a `goto` statement, the JVM and the Dalvik virtual machine both have the `goto` instruction. Since `goto` is not provided in the Java source language, a source level representation of the transformed program may not exist. Listing 4 shows an example reordering. Note that `move-result-*` must be the first instruction after a call to capture the return value.

```
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

5) 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. We propose two different kinds of transformations for this purpose: `nop` insertion, and arithmetic and branch insertion.

NOP insertion: This transformation simply inserts sequences of `nop` instructions in the code. It is easy to detect and undo.

Arithmetic and branch insertion: This transformation introduces junk arithmetic and branch instructions based on simple templates. The branch instructions have arbitrary branch offsets. The branch conditions are designed to be always false so that the branches are never actually taken. We assume that the value of these conditions (true or false) will be opaque to anti-malware tools being tested. Such obfuscation may create additional dependencies in control flow analysis. Listing 5 demonstrates some of the junk code that we generate. As in code reordering, we point out that there may not be a source level equivalent which compiles to the transformed program because branches are made to arbitrary offsets whereas control flow in Java is based on nested blocks (save the limited use of `break` and `continue`).

```
const/16 v0, 0x5
const/16 v1, 0x3
add-int v0, v0, v1
add-int v0, v0, v1
rem-int v0, v0, v1
if-lez v0, :junk_1
```

Listing 5: An example of a junk code fragment

6) 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).

7) 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. It is typically used by compilers for optimizing code related to short functions. The outlining refactoring has been proposed to eliminate duplicate code in programs [11]. However, outlining and inlining can be used for call graph obfuscation also. Outlining can also be used to impede all kinds of intra-procedural analyses. If a function is broken into sufficiently small chunks, intra-procedural analysis will not be able to give any useful information. Interprocedural analysis is still possible though.

8) Other Simple Transformations: There are a few other transformations as well, specific to Android. Bytecode typically contains a lot of debug information, such as source file names, local and parameter variable names, and source line numbers. All this information may be stripped off. Another possible transformation is due to the nature of Android packages, which are zip files. Files archived in these zip files may be renamed. Finally, Android packages contain various resources apart from the `classes.dex` and AndroidManifest. All these resources may be renamed or modified appropriately.

9) 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 in all cases discussed here.

C. 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.

1) Reflection: Reflection is an easy way to obfuscate method calls. Reflection is the ability provided by certain programming languages allowing a program to introspect itself and change its behavior at runtime. In Java, the reflection API allows a program, among other things, to invoke a method by using the name of the methods. In reflection transformation, we convert every 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. Listing 6 illustrates code in Listing 1 after reflection transformation.

```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
const/4 v13, 0x2
new-array v14, v13, [Ljava/lang/Class;
new-array v15, v13, [Ljava/lang/Object;
const/4 v13, 0x0
const-class v12, Ljava/lang/String;
aput-object v12, v14, v13
aput-object v10, v15, v13
const/4 v13, 0x1
const-class v12, Ljava/lang/String;
aput-object v12, v14, v13
aput-object v11, v15, v13
const-string v13, "runRootCommand"
const-class v12, Lcom/android/root/Setting;
invoke-virtual {v12, v13, v14}, Ljava/lang/Class;->
     getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/
     lang/reflect/Method;
move-result-object v13
const/4 v16, 0x0
invoke-virtual {v13, v12, v15}, Ljava/lang/reflect/Method
     ;->invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/
     lang/Object;
move-result-object v7
check-cast v7, Ljava/lang/String;
```

Listing 6: Listing 1 with method call by reflection

2) 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 in different ways at each replication of the virus to evade detection. We discuss here code encryption alone; obfuscation of the decryption routine may be possible by other methods discussed above.

We accomplish bytecode encryption by moving most of the application in a separate dex file (packed as a jar) and storing it in the application package in an encrypted form. When one of the application components (such as an activity or a service) is created, it first calls a decryption routine that decrypts the dex file and loads it via a user defined class loader. In Android, the `DexClassLoader` provides the functionality to load arbitrary dex files. Following this operation, calls can be made into the code in the newly loaded dex file. Alternatively, one could define a custom class loader that loads classes from a custom file format, possibly containing encrypted classes. We note that classes which have been defined as components need to be available in `classes.dex` (one that is loaded by default) so that they are available to the Android middleware in the default class loader. These classes then act as wrappers for component classes that have been moved to other dex files.

IV. 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. Certain transformations such as native code encryption are not possible to completely automate because one needs to know how native code files are being handled in the code.[2] Transformations that require modification of the AndroidManifest (rename packages and renaming components) have not been completely automated because we felt it was more convenient to modify manually the AndroidManifest for our study. Nevertheless, it is certainly possible to automate this as well. Finally, we did not automate bytecode encryption, although there are no technical barriers to doing that. However, we have implemented a proof-of-concept bytecode encryption transformation manually on existing malware.

We utilize the Smali/Baksmali [12] and its companion tool Apktool [13] for our implementation. Apktool is able to unpack an application package, disassemble `classes.dex` into smali code and convert AndroidManifest to human readable form among other things. It can also assemble and repack a package. Most of the code transformations are applied to the smali assembly code, which is assembled later into dex code. Only method and field renaming was implemented directly on the dex code, yet using the underlying library for smali/baksmali. The assembling and disassembling transformation is implemented simply by decoding and building with 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. Our overall implementation comprises about 1,100 lines of Python and Scala code.

We verified that our implementation of transformations do not modify the semantics of the programs. Specifically, we tested our transformations against several test cases and verified their correctness on two malware samples, DroidDream and Fakeplayer. In general, verifying correctness on actual malware is challenging because some of the original samples have turned non-functional owing to, for example, the remote server not responding, and because being able to detect all the malicious functionality requires a custom, appropriately monitored environment. Indeed, our original DroidDream sample would not work because it failed to get a reply from a remote server; we removed the functionality of contacting the remote server to confirm that the malicious functionality works as intended.

V. The Dataset

This section describes the anti-malware products and the malware samples we used for our study. We evaluated ten anti-malware tools, which are listed in Table I. There are dozens of free and paid anti-malware offerings for Android from various well-established anti-malware vendors as well as not-so-well-known developers. We selected the most popular products; in addition, we included Kaspersky 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. One of the tools, Dr. Web, actually claims that its detection algorithms are resilient to malware modifications.

Our malware set is summarized in Table II. We used a few criteria for choosing malware samples. First, all the anti-malware tools being evaluated should detect the original samples. We here have a question of completeness of the signature set, which is an important evaluation metric for antivirus applications. In this work however, we do not focus on this question. Based on this criterion, we rejected Tapsnake, jSMSHider and a variant of Plankton. 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 [14].

Our malware set spans over multiple malware kinds. DroidDream [15] and BaseBridge [16] are malware with root exploits packed into benign applications. DroidDream tries to get

[2] Native code stored in non standard locations is typically copied from the application package to the application directory by the application itself (possibly through an available Android API).

TABLE I: Anti-malware products evaluated.

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 II: Malware samples used for testing anti-malware tools

Family	Package name	SHA-1 code	Date found	Remarks
DroidDream	com.droiddream. bowlingtime	72adcf43e5f945ca9f72 064b81dc0062007f0fbf	03/2011	Root exploit
Geinimi	com.sgg.spp	1317d996682f4ae4cce6 0d90c43fe3e674f60c22	10/2011	Information exfiltration; bot-like capabilities
Fakeplayer	org.me.androidapplication1	1e993b0632d5bc6f0741 0ee31e41dd316435d997	08/2010	SMS trojan
Bgserv	com.android.vending. sectool.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

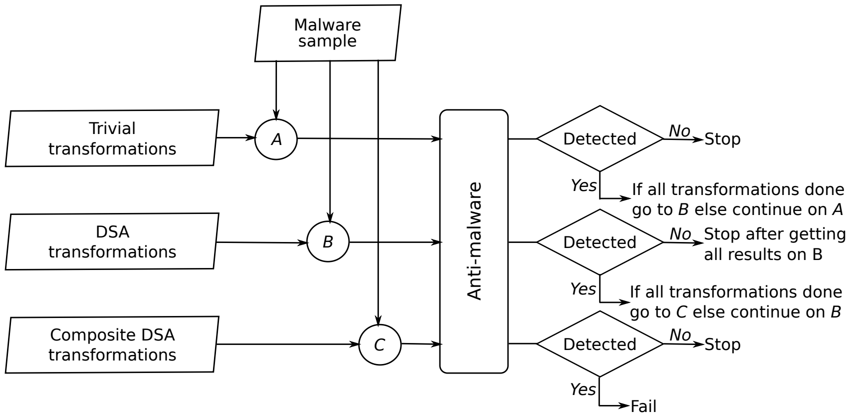
Malware sample
Trivial transformations
DSA transformations
Composite DSA transformations
A
B
C
Anti-malware
Detected
No
Stop
Yes
If all transformations done go to *B* else continue on *A*
Detected
No
Stop after getting all results on B
Yes
If all transformations done go to *C* else continue on *B*
Detected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

Figure 1: Evaluating anti-malware

TABLE III: Key to Tables IV, V and VI. 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	Encode strings and array data
CR	Reorder code
CI	Call indirection
JN	Insert junk code
All transformations contain P	
All transformations except P contain A	

root privileges using two different root exploits, rage against the cage, and exploid exploit. BaseBridge includes only one exploit, rage against the cage. If these exploits are successful, both DroidDream and BaseBridge install payload applications. Geinimi [17] is a trojan packed into benign applications. It communicates with remote C&C servers and exfiltrates user information. Fakeplayer [18], the first known malware on Android, sends SMS messages to premium numbers, thus costing money to the user. Bgserv [19] is a malware injected into Google's security tool to clean out DroidDream and distributed in third party application markets. It opens a backdoor on the device and exfiltrates user information. Plankton [20] is a malware family that loads classes from additional downloaded dex files to extend its capabilities dynamically.

VI. Results

As has already been discussed, we transform malware samples using various techniques discussed in Section III and pass them through anti-malware tools we evaluate. We will now briefly describe our methodology and then discuss the findings of our study.

We describe our methodology through Figure 1 and through Tables IV and V, which depict the series of transformations applied to DroidDream and Fakeplayer samples and the detection results on various anti-malware tools. Empty cells in the tables indicate positive detection while cells with 'x' indicate that the corresponding anti-malware tool failed to detect the malware sample after the given transformations were applied to the sample. The tables reflect a general approach of our study. 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.[3] 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. We do not show the redundant combinations in the tables for the sake of conciseness. The last rows do not form part of our methodology; we construct them manually to show the set of transformations with which all anti-malware tools yield.

Our results with all the malware samples are summarized in Table VI. 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. Our tool produces actual malware; we take special precaution to avoid spreading these samples and are careful with whom we share these samples. We next describe our key findings in the light of the detection results. These findings are not meant to be statistical conclusions; yet they give a general idea of the capabilities of anti-malware tools.

Finding 1 *All the studied anti-malware products are vulnerable to common transformations.* All the transformations appearing in Table VI 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 [10], [21]. Exploit and payload encryption is also easy to achieve. Although most of current Android malware uses simple techniques, without the use of sophisticated transformations, we point out that some of these transformations may already be seen in the wild in current malware. For example, Geinimi variants have encrypted strings [22]. Similarly, the DroidKungFu malware uses encrypted exploit code [23]; a similar transformation to DroidDream allows easy evasion across almost all the anti-malware tools we studied. Finally, there are reports of similar server-side polymorphism as well [6]. In future, it is likely that more and more malware will adopt sophisticated techniques for polymorphism. No transformations just discussed thwart static analysis.

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. We also tested Dr. Web against reflection transformation (not shown in the tables) and were able to evade it. This offers another confirmation that signatures are based on method calls. Furthermore, we also found (by limiting our transformations) that only framework API calls matter; calls within the application make no difference. It seems that the matching is somewhat fuzzy (requiring only a threshold percentage of matches) because we found on DroidDream and Fakeplayer that results are positive even when a few classes are removed from the dex file. For these two families, we could create multiple minimal sets of classes that would result in positive detection. As mentioned earlier, Dr. Web indeed claims it has signatures that are resilient to malware modifications. It is difficult to say if the polymorphic resistance of these signatures is any stronger than other signatures depending on identifier names and string and data values. In particular, such signatures do not capture semantic properties of malware such as data and control flow. Our results aptly demonstrate the low resistance.

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). In case of AVG, the signatures are based on application component classes or package names or both. Furthermore, this information is derived from AndroidManifest only. We confirmed this by placing a fake AndroidManifest in malware packages and assembling them with the rest of the package kept as it is. This AndroidManifest did not have any of the components or package names declared by the malware applications. The results were that detection was negative for all the malware samples.

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.

Finding 4 *Anti-malware tools have evolved towards content-based signatures over the past one year.* We studied compare our findings that we obtained in February 2012 (Table VII) to our present findings obtained in February 2013 (Table VI). Some of the anti-malware tools have changed considerably for the same malware samples. Last year, 45% of the signatures were evaded by trivial transformations, i.e., repacking and assembling/disassembling. Such signatures have virtually no resilience against polymorphism. Our present results show a marked decrease in this fraction to 16%. We find that in all such cases where we see changes, anti-malware authors have moved to content-based matching, such

[3] All DSA and NSA transformations also result in trivial transformations because of involving disassembling, assembling and repacking. Hence, there is no use in proceeding further.

TABLE IV: DroidDream transformations and anti-malware failure. Please see Table III for key. ‘x’ indicates failure in detection.

	AVG	Symantec	Lookout	ESET	Dr. Web	Kaspersky	Trend M.	ESTSoft	Zoner	Webroot
P										
A									x	
RP	x							x	x	
EE			x						x	
RI		x							x	x
ED									x	
CR									x	
CI									x	
JN									x	
RI+EE		x	x	x					x	x
EE+ED			x			x			x	
EE+RF			x				x		x	
EE+CI			x		x				x	
RP+RI+EE+ED+RF+CI	x	x	x	x	x	x	x	x	x	x

TABLE V: Fakeplayer transformations and anti-malware failure. Please see Table III for key. ‘x’ indicates failure in detection. EE transformation does not apply for lack of native exploit or payload in Fakeplayer.

	AVG	Symantec	Lookout	ESET	Dr. Web	Kaspersky	Trend Micro	ESTSoft	Zoner	Webroot
P										
A							x		x	
RP							x	x	x	x
RI				x		x	x		x	
ED							x		x	
CR							x		x	
CI					x		x		x	
JN							x		x	
RP+RI	x	x	x	x		x	x	x	x	x
RP+RI+CI	x	x	x	x	x	x	x	x	x	x

TABLE VI: Evaluation summary. Please see Table III 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	EE	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

TABLE VII: Summary of results from anti-malware tools downloaded in February 2012. Please see Table III for key. ‘+’ indicates the composition of two transformations. Results that have changed since then are depicted in bold (see Table VI for comparison).

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

as matching identifiers and strings.

Furthermore, for malware using native code exploits, many anti-malware tools previously matched on the native exploits and payloads alone. The situation has changed now as all of these additionally match on some content in the rest of the application as well. Although the changes in the signatures over the past one year may be seen as improvement, we point out that the new signatures still lack resilience against

```
<manifest ... package= "com.crazyapps.angry.birds.rio.unlocker" ... >
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <activity android:label="@string/app_name" android:name= ".AngryBirdsRioUnlocker" ... >
            :
        </activity>
        <service android:name= "com.plankton.device.android.AndroidMDKProvider" ... />
    </application>
```

```
<manifest ... package= "com.hDEWJu.oYlCvk.hFYkwc.FgDOHA.UPkmVF" ... >
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <activity android:label="@string/app_name" android:name= ".LncHMH" ... >
            :
        </activity>
        <service android:name= "com.rawJbA.DKPTQc.aaMYse.QUivSk" ... />
    </application>
```

Figure 2: An example evasion. Changes required in AndroidManifest of Plankton to evade AVG (original first and modified second; only relevant parts are shown with differences highlighted). No other changes are required. The application will not work though until the components are also renamed in the bytecode. We confirm that AVG's detection is based on the contents of AndroidManifest alone (see Finding 2).

polymorphic malware as our results aptly demonstrate.

VII. Defense against Transformation Attacks

In this section, we discuss how the current state of malware detection on Android may be improved. We identify how anti-malware tools should improve their detection techniques and that mobile platforms should provide special support to antimalware tools.

A. *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. [24] 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[4] described in the template. The signature template abstracts data flows and control flows, which are semantics properties of a program. 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 VI. These techniques further have a potential for a very low false positive rate as the authors demonstrate in their work. Such a detection scheme is arguably slower than current detection schemes but offers higher confidence in detection. This is just another instance of the traditional security-performance tradeoff. Christodorescu et al. had actually reported the running times to be in the order of a couple of minutes on their prototype and had suggested real performance is possible with an optimized implementation [24]. Developing signature templates itself may be challenging. Automatic signature generation has been discussed in the context of dynamic analysis [25], [26] but it may be possible to adapt similar techniques to static analysis as well.

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 [27], [28]. Even disassembly of native binaries can be error prone [29], [30]. Stripped binaries pose even greater problems, which are not fully solved yet and current solutions for accurate disassembly require combination of static and dynamic techniques [31]. 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.

B. *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 visible code on which signatures can be developed (of course, the decryption routing may still be used for generating signatures). The use of reflection simply hides away the edges in the call graph. A sophisticated data flow analysis can still uncover those edges; however, 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

[4] A def-use path for a variable signifies a definition of that variable in a program and all uses of that variable, reachable from that definition.

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.

We note that providing privileges for dynamic monitoring to anti-malware applications would promote opportunities for malware to trick users to grant high privileges. This is again a trade-off. Anti-malware tools on PCs typically require high privileges and do useful work even though there are issues of fake antiviruses [32].

We note that Google recently introduced on-phone app verification [33], which checks the app checksum against a malware database upon installation. This however is not sufficient against polymorphic attacks each instance of a malicious app is unique. Google also performs offline app analysis for malware detection using its Bouncer service [34]. This is based on emulation (using virtual machines) of real phone environments. Such scanning by emulation however has its own problems, ranging from detection of a virtualized environment to the malicious activity not getting triggered in the limited time for which the emulation runs; Bouncer is no exception to this [35], [36]. We therefore believe offline emulation must be supplemented by strong static analysis or real-time dynamic monitoring.

VIII. Related Work

A. Evaluating Anti-malware Tools

AV-Test.org, an antivirus evaluation lab, rated anti-malware products for Android for the completeness of their detection [7], [8]. Our study is orthogonal to their study in that we evaluate how anti-malware products perform in detecting polymorphic variants of known malware. Most of the tools (9/10) we studied are rated as "very good" by them. This provides us reason to believe that the tools we did not study will not have any better resistance to polymorphism.

Zheng et al. [9] also studied the robustness of anti-malware against Android malware recently using a tool called ADAM. ADAM implements only a few transformations, renaming methods, introducing junk methods, code reordering, and string encoding, in addition to repacking and assembling/disassembling. Our set of transformations is much more comprehensive and includes renaming packages, classes, encoding array data, inserting junk statements, encrypting payloads and native exploits, reflection, and bytecode encryption as well. Finally, we also have composite transformations. Many of the additional transformations, including the composite ones, were crucial for evading anti-malware tools. Based on the above, we point out that ADAM is not always able to evade an anti-malware tool. Rather than attempting complete evasion, it simply offers percentages depicting how many variants were detected by the anti-malware tools (and these percentages are also very high). In contrast, our framework is comprehensive, aimed towards complete evasion of all anti-malware tools. We believe our results make a clear statement – all anti-malware tools can be evaded using common obfuscation techniques. Unlike ADAM, our result is able to highlight the severity of the problem and is easily accessible.

Christodorescu and Jha [4] conducted a study similar to ours on desktop anti-malware applications eight 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 dates after many research works (see below) on obfuscation resilient detection, and we would expect the proposed techniques to be readily integrated into new commercial products.

Finally, there are many works in the industry about the evaluation of desktop antivirus tools on metrics such as signature completeness, usability and so on [37], [38].

B. Obfuscation Techniques

Collberg et al. [39] review different types of obfuscations and classify them based on reverse engineering by a human and by automated tools, and the overhead added to the application. They propose many different obfuscations possible on Java (or Dalvik) code. Collberg et al. further propose sophisticated transformations such as modifying inheritance graphs and method cloning and implementation of opaque predicates (predicates whose outcome is difficult to arrive at while reverse engineering but is known to the obfuscator) to insert junk code [40], [41]. DroidChameleon provides only a few of the transformations proposed by them. Nonetheless, the set of transformations provided in DroidChameleon is comprehensive (together with the advanced transformations) in the sense that they can break typical static detection techniques used by anti-malware. As for opaque predicates, we use such techniques in our transformation for inserting junk code with the assumption that anti-malware tools will not be able to resolve conditions we use therein.

There are many tools that provide obfuscation for Java bytecode. Proguard [10] provides renaming of classes and class members. Other tools like Klassmaster [21] additionally provide flow obfuscation and string encryption. We provide much of these functionalities. While the goal of these tools is to evade manual reverse engineering, we aim at thwarting analysis by automatic tools.

C. Obfuscated Malware Detection

As already discussed, to deal with malware obfuscation, the detection techniques must be based on semantics rather than the syntax of the code. These detection techniques should therefore be based on data flow and control flow analyses of the samples under test. Christodorescu et al. [24] present one such technique. Their algorithm is based on matching given samples against a template by unifying nodes in samples with nodes in the template while preserving def-use relationships. In subsequent work, Preda et al. [42] propose a semantics-based framework to prove properties about malware detectors. Kruegel et al. [43] tackle the problem of disassembling binaries that have been made hard to disassemble for malware analysis. Christodorescu et al. [44] and Fredrikson et al. [25] attempt to generate semantics based signatures by mining malicious behavior automatically. Kolbitsch et al. [26] 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. Sequence alignment from bioinformatics [45], [46] has also been applied to malware detection and related problems [47], [48]. Further work is also there to compute statistical significance of scores given by these classical sequence alignment algorithms [49], [50]. It may be possible to adapt such techniques to detect transformed malware with high performance.

D. Smartphone Malware Research

With the growth of malware on smartphones, several research works have been done in this direction. DroidRanger [51] and Riskranker [52] use (mostly) static analysis to detect unknown malware from both known and unknown malware families. They identified several new malicious applications in the official Android market as well as alternative application markets. Peng et al. [53] investigate probabilistic models to rank risks for Android apps. Anti-malware authors may explore their approaches, which may serve as heuristics to raise malware suspicions. Crowdroid [54] uses crowd sourcing to collect system calls from applications running on mobile devices then uses clustering to identify malicious behavior. Such techniques cannot be currently used by unprivileged third-party anti-malware applications on Android. Felt et al. [55] present a survey of smartphone malware. They present taxonomy of smartphone malware and explore the incentives to develop mobile device malware. Zhou and Jiang [56] provide another, more recent survey of Android malware. They study how well anti-malware tools detect malware samples found in the wild. The tools have good detection on some families, like Fakeplayer and Geinimi, but fail in our tests when the samples are transformed. Airmid [57] proposes new mobile infrastructure for malware mitigation. Apart from Android, they also explored malware on Symbian and iOS. Bose et al. [58] and Kim et al. [59] have used logical ordering of applications' actions and power consumption respectively to construct behavioral detection of Symbian malware. VirusMeter [60] also uses power consumption to catch misbehaving Symbian malware. It is still to be demonstrated if these techniques apply well to Android also. In a summary, none of the above works focuses on evaluating current mobile anti-malware solutions.

IX. 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 and may succumb to even trivial transformations not involving code-level changes. Finally, we explored possible ways in which the current situation may be improved and next-generation solutions may be developed.

References

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

[2] McAfee, "Mcafee threats report: Third quarter 2011," http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2011.pdf.

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

[4] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. ACM, 2004.

[5] Fortinet, "2012 threat predictions," http://blog.fortinet.com/2012-threat-predictions/.

[6] Symantec, "Server-side polymorphic android applications," http://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications.

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

[8] "Are free Android virus scanners any good?" http://www.av-test.org/fileadmin/pdf/avtest_2011-11_free_android_virus_scanner_english.pdf.

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

[10] "ProGuard," http://proguard.sourceforge.net/.

[11] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *In POPL*. ACM Press, 2000, pp. 155–169.

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

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

[14] M. Parkour, "Contagio Mobile. Mobile malware mini dump," http://contagiominidump.blogspot.com/.

[15] Lookout, "Update: Security alert: DroidDream malware found in official Android Market," http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/.

[16] "Android.Basebridge — Symantec," http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99.

[17] "Android.Geinimi — Symantec," http://www.symantec.com/security_response/writeup.jsp?docid=2011-010111-5403-99.

[18] "AndroidOS.FakePlayer — Symantec," http://www.symantec.com/security_response/writeup.jsp?docid=2010-081100-1646-99.

[19] "Android.Bgserv — Symantec," http://www.symantec.com/security_response/writeup.jsp?docid=2011-031005-2918-99.

[20] "Plankton," http://www.csc.ncsu.edu/faculty/jiang/Plankton/.

[21] "Zelix Klassmaster," http://www.zelix.com/klassmaster/.

[22] Lookout, "Geinimi trojan technical analysis," http://blog.mylookout.com/blog/2011/01/07/geinimi-trojan-technical-analysis/.

[23] "DroidKungFu," http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html.

[24] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*. IEEE, 2005, pp. 32–46.

[25] 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*. IEEE, 2010, pp. 45–60.

[26] 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*. USENIX Association, 2009, pp. 351–366.

[27] 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*. ACM, 2008, pp. 74–83.

[28] L. Harris and B. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.

[29] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 45–54.

[30] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 290–299.

[31] S. Nanda, W. Li, L. Lam, and T. Chiueh, "Bird: Binary interpretation using runtime disassembly," in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*. IEEE, 2006, pp. 12–pp.

[32] Microsoft, "Watch out for fake virus alerts," http://www.microsoft.com/security/pc-security/antivirus-rogue.aspx.

[33] J. Raphael, "Exclusive: Inside Android 4.2's powerful new security system," November 2012, http://blogs.computerworld.com/android/21259/android-42-security.

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

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

[36] 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.

[37] N. J. Rubenking, "PCMag. The best antivirus for 2012," http://www.pcmag.com/article2/0,2817,2372364,00.asp.

[38] "AV-Test," http://www.av-test.org/index.php?L=1.

[39] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[40] ——, "Breaking abstractions and unstructuring data structures," in *Computer Languages, 1998. Proceedings. 1998 International Conference on*. IEEE, 1998, pp. 28–38.

[41] ——, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Conference Record of the Acm Symposium on Principles of Programming Languages*, vol. 25. ACM, 1998, pp. 184–196.

[42] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. ACM, 2007.

[43] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 255–270.

[44] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. ACM, 2007.

[45] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, March 1970.

[46] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences." *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[47] X. Jiang and X. Zhu, "vEye: Behavioral footprinting for self-propagating worm detection and profiling," *Knowledge and Information Systems*, vol. 18, no. 2, pp. 231–262, 2009.

[48] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna, "Automatic network protocol analysis," in *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[49] A. Agrawal and X. Huang, "Pairwise statistical significance of local sequence alignment using multiple parameter sets and empirical justification of parameter set change penalty," *BMC Bioinformatics*, vol. 10, no. Suppl 3, p. S1, 2009.

[50] ——, "Pairwise statistical significance of local sequence alignment using sequence-specific and position-specific substitution matrices," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 8, no. 1, pp. 194–205, 2011.

[51] 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*, ser. NDSS '12, 2012.

[52] 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*, ser. MobiSys '12. ACM, 2012.

[53] 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.

[54] 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*. ACM, 2011, pp. 15–26.

[55] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.

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

[57] Y. Nadji, J. Giffin, and P. Traynor, "Automated remote repair for mobile malware," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 413–422.

[58] A. Bose, X. Hu, K. G. Shin, and T. Park, "Behavioral detection of malware on mobile handsets," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. ACM, 2008.

[59] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. ACM, 2008.

[60] L. Liu, G. Yan, X. Zhang, and S. Chen, "Virusmeter: Preventing your cellphone from spies," in *Recent Advances in Intrusion Detection*. Springer, 2009, pp. 244–264.