ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems

Min Zheng, Patrick P. C. Lee, and John C. S. Lui

Dept of Computer Science and Engineering, The Chinese University of Hong Kong
{mzheng,pclee,cslui}@cse.cuhk.edu.hk

Abstract. With the rising threat of smartphone malware, both academic community and commercial anti-virus companies proposed many methodologies and products to defend against smartphone malware. Thus, how to assess the effectiveness of these defense mechanisms against existing and unknown malware becomes important. We propose *ADAM*, an automated and extensible system that can evaluate, via large-scale stress tests, the effectiveness of anti-virus systems against a variety of malware samples for the Android platform. Specifically, ADAM can automatically transform an original malware sample to different variants via repackaging and obfuscation techniques in order to evaluate the robustness of different anti-virus systems against malware mutation. The transformation and evaluation processes of ADAM are *fully automatic*, *generic*, and *extensible* for different types of malware, anti-virus systems, and malware transformation techniques. We demonstrate the efficacy of ADAM using 222 Android malware samples that we collected in the wild. Using ADAM, we generate different variants based on our collected malware samples, and evaluate the detection of these variants against commercial anti-virus systems.

1 Introduction

Malware (e.g., worms, viruses, and trojans) has been a well-known threat in the computing and networking communities. With the proliferation of smartphones, the threat of smartphone malware becomes more formidable. TGDaily [49] reported that there was a 33% increase in smartphone malware over 2009. As of October 2011, Tencent Mobile Security Laboratory [48] identified around 13,000 and 6,000 mobile phone viruses in the Symbian and Android platforms respectively. Given the threat of smartphone malware, researchers (e.g., [10, 11, 13, 35, 42, 44, 46, 52]) have proposed various smartphone malware detection systems, and anti-virus software companies also develop commercial security solutions to detect smartphone malware. However, new pieces of smartphone malware keep evolving and attacking various distributions of smartphone platforms [36]. Thus, understanding the smartphone malware battle between the good and evil sides is critical for the community to improve the state of the art of the smartphone malware detection solutions. This motivates us to design a system that can *stress test* an anti-virus solution, so that one can systematically evaluate the effectiveness (or ineffectiveness) of existing smartphone malware detection systems against the emergence of smartphone malware.

Evaluating smartphone malware detection is a non-trivial issue, especially with the challenge that there are a wide variety of smartphone operating systems available nowadays. In this work, we focus on *Android*, a Linux-based operating system that runs Java-based applications. Android applications can be directly self-signed and published by application developers through the official Android Market [6] without being subject to any official security validations. This unmoderated nature of Android provides a fertile ground for the development of both benign and malicious applications. As reported by International Data Corporation [30], Android led all smartphone OSes with 38.9% of market share in 2011, and is expected to grow to more than 40% of the market through 2015. Meanwhile, Android also becomes the most targeted operating system for smartphone malware [36]. Note that even the Android Market applies stringent security checks to its hosted applications, it cannot entirely resolve the malware distribution among Android phones, since some countries may ban the access to the Android Market (see Section 6). Thus, by focusing on the Android platform, our evaluation study can provide representative insights into the robustness of existing smartphone malware detection systems.

There are number of studies (e.g., [14, 15, 37, 38]) that focus on evaluating the effectiveness of existing malware detection systems. Such evaluation studies employ different *obfuscation* techniques to transform a malware program into different variants (with the original malicious behavior preserved), and then check whether a malware detection system still treats the variants as malware. Note that most of these studies (e.g., [14, 15, 38]) focus on PC-based malware only, and limited studies (e.g., [37]) consider malware for smartphones. Given the growing popularity of smartphones, there is an urgent need to understand the effectiveness of anti-virus systems on smartphones, as well as their robustness against new and evolving malware. Furthermore, it remains challenging to scale up the evaluation to a large number of malware samples, as we need to ensure the correctness of various obfuscation techniques for each malware sample. Although we narrow down our focus on Android, the evaluation is still overwhelmed by numerous Android malware samples in the wild [48] as well as various malware detection solutions. Thus, the key motivation of this work is to develop an evaluation system that can *automatically* apply to general classes of smartphone malware and anti-virus solutions, and ultimately, support large-scale evaluation.

In this paper, we design and implement *ADAM*, an automated system for evaluating the detection of Android malware. ADAM applies different transformation techniques to generate different variants of each Android malware sample, and evaluates the effectiveness of different smartphone malware detection systems in identifying such malware variants. ADAM is designed to be *automated*, *generic*, and *extensible*. It automatically transforms an Android malware sample into different variants through various repackaging and obfuscation techniques, while preserving the original malicious behavior. ADAM then evaluates the detection of these variants against different smartphone malware detection systems. Such malware transformations and detection evaluations are generic enough to support heterogeneous malware samples and malware detection systems, respectively. Lastly, ADAM can be extensible to support new implementations of malware transformations and detection evaluations.

As a proof of concept, we demonstrate how ADAM can be used to assess the robustness of existing anti-virus systems in practice. We collected 222 malware samples in the wild. We use ADAM to generate different variants for each collected malware sample, and show that ADAM has a very high success rate in the automated generation of variants. We proceed to pass the variants to different commercial anti-virus engines hosted on the web portal VirusTotal [51]. We discuss the findings and implications based on the detection results returned from VirusTotal, but we emphasize that ADAM can also be integrated with other anti-virus systems.

The rest of the paper proceeds as follows. In Section 2, we provide a brief background on how to prepare and generate an Android application. In Section 3, we present the design of ADAM. In Section 4, we present various transformation techniques to generate different malware variants. In Section 5, we present our evaluation results against different anti-virus systems. In Section 6, we discuss several open issues. Section 7 surveys related work, and Section 8 concludes the paper.

2 Background

Let us describe the software life cycle of building an Android application from source code, as well as the reverse engineering process of an Android application. This lays the foundation of how our ADAM system transforms an Android malware application into another runnable Android malware variant while preserving the malicious behavior.

2.1 Building an Android Application

An Android application is mainly written in Java source code. The build process of an Android application is to compile and package a Java source code project into an `.apk` file that can run on a smartphone device or emulator. We now summarize the key steps of the build process [2] as follows.

1. **Preparation.** An Android project contains Java source code (and possibly some other native code), as well as metadata such as resources and programming interfaces. The build process first converts the metadata information into Java code or interfaces.
2. **Compilation.** All Java source code files as well as the converted metadata are compiled together into `.class` files, which contain Java bytecode.
3. **Bytecode conversion.** All Android applications run on the *Dalvik Virtual Machine (DVM)*, which is a runtime environment similar to the Java Virtual Machine (JVM) but is designed for mobile devices that generally have limited hardware resources. The build process converts all `.class` files into `.dex` files, which contain the Dalvik Executable bytecode.
4. **Building.** All resource files, including both non-compiled and compiled files, as well as the `.dex` files are then packaged (i.e., zipped) into a single `.apk` file.
5. **Signing.** The `.apk` file needs to be digitally signed before it can be published in well-known sites (e.g., Google Market). It is typical that the `.apk` file is signed with the application developer's private key, rather than by a centrally trusted authority [4]. As described in Section 1, this type of unmoderated mechanism leads to

proliferation of Android applications, but at the same time, allows easy penetration of malware programs.

6. **Alignment.** To optimize the performance of the Android program (e.g., reducing memory usage), the `.apk` file can be aligned along the byte boundaries with the zipalign tool [5]. Note that some integrated development environment (IDE), such as Eclipse with the ADT plugin, will automatically zipalign the `.apk` file after signing the file with the developer's private key.

2.2 Process of Reverse Engineering an Android Application

In order to stress test the effectiveness of an anti-virus system, we need to create a library of malware variants and from existing malware. In most cases, the source code of malware (or an Android application) is not readily available, but instead, we can only access its `.apk` file and its underlying `.dex` files. To generate various malware variants, one has to resort to *reverse engineering*. We review two approaches that can be used to reverse-engineer an Android application.

1. **Decompiling.** The goal of the decompiling process is to convert a `.dex` file (with the DVM bytecode) into the `.java` source code files. A typically approach is to first convert the `.dex` file to `.class` files (e.g., using the dex2jar utility [20]), which are then converted to `.java` files using Java decompiler (e.g., using the Java Decompiler utility [33]). It is important to note that, the decompiling process may generate a source code file that is significantly different from the original one.
2. **Disassembling.** The disassembling process is to convert a `.dex` file into `.smali` files (e.g., using utility like apktool [9]), which contain assembly-like code for the Android OS. The process takes the Dalvik opcodes of a `.dex` file and converts them into low-level instructions. Typically, the decoded `.smali` files can be rebuilt again back to a `.dex` file.

In this paper, we focus on using the disassembling approach to reverse-engineer an Android malware sample. Through the disassembling approach, we can systematically locate specific assembly-like instructions for different malware samples, and apply code obfuscation to generate malware variants. We elaborate this in Section 4.2.

3 Design Overview of ADAM

In this section, we present an overview on the design of ADAM, an automated system for evaluating the detection of existing Android-based malware detection systems.

3.1 Design Goals

ADAM aims for the following design goals:

– *Security analysis.* ADAM checks whether an Android-based malware sample in `.apk` format can be detected by an existing anti-virus system. For this analysis, we do not need the source code of the malware sample.

– *Automated transformation.* ADAM automatically transforms a malware sample into different malware variants, while preserving the original malicious behavior. No manual modification of a malware sample is required.
– *Generic application.* ADAM can be applied for general classes of Android-based malware samples and malware detection systems.
– *Extensibility.* ADAM provides an interface that can easily integrate new implementations of transformation techniques and detection methodologies.

3.2 Building Blocks

ADAM is composed of different building blocks. Figure 1 illustrates how different building blocks are involved in testing malware samples against anti-virus systems. Let us now describe how each building block works, and argue how the building blocks can be extended for different variants of implementation.

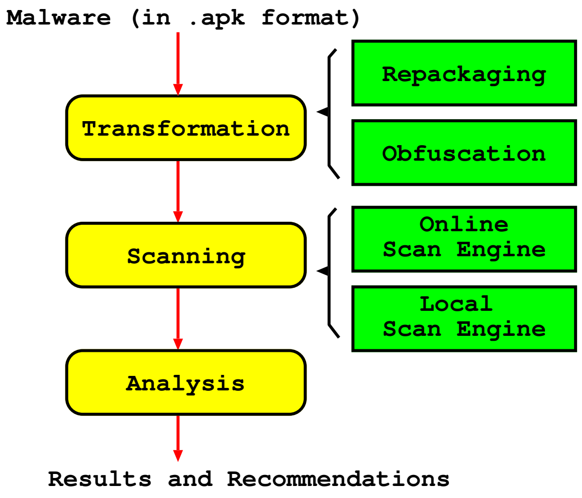

Fig. 1. Design flow of ADAM.

Transformation. Given an input `.apk` malware file, ADAM transforms it into different variants of `.apk` files based on various transformation techniques, such that each output `.apk` file preserves the original malicious behavior of the input `.apk` files. We implement two classes of transformation techniques: *repackaging* and *code obfuscation*. Details of these techniques are described in Section 4. We emphasize that ADAM is extensible in the sense that one can plug-in other transformation techniques to generate different `.apk` variants.

Scanning. For each malware variant that we create, we pass it to an anti-virus scan engine. Here, we focus on the scan engines of commercial vendors, while we can also plug-in other malware detection systems with the correct interface.

In ADAM, we support two types of scan engines: *online* and *local*. An online scan engine refers to a web service that provides a library of open APIs. Users can upload an `.apk` file to the web service and obtain the results via the APIs. Typically, the web service is free of charge, but *rate-limit* the number of samples that can be scanned. Also, the scanning performance varies depending on the current network conditions. In our implementation, we use the VirusTotal web portal [51], which is connected to various commercial anti-virus systems at its backend. On the other hand, a local (or desktop) scan engine uses the command-line interface provided by an anti-virus vendor. It simply specifies an `.apk` file as an input command-line argument and obtains the scanned results. In our implementation, we integrate the Linux desktop version of the anti-virus engine obtained from Antiy [8]. Our evaluation study covers both online and local scan engines (see Section 5).

Analysis. ADAM collects the results from the anti-virus scan engines of different commercial vendors. One can determine if a scanned `.apk` file is a malware sample based on the decisions of one or multiple anti-virus systems. Aggregating the results of multiple anti-virus systems can potentially increase the detection rate[40]. The analysis results can be summarized and presented, so as to provide recommendations for anti-virus vendors to evaluate the effectiveness of the state of the art of malware detection for Android.

4 Malware Transformation

In this section, we present techniques that we use to transform a given malware sample into different variants. The resulting variants will be used by ADAM as inputs to evaluate the effectiveness of different malware detection systems. Specifically, we consider two classes of transformation techniques: *repackaging* and *code obfuscation*, both of which take an `.apk` file as an input and generate a different `.apk` file as an output. Furthermore, we require that the output of an `.apk` file preserves the same logic and functionality as the original input `.apk` file.

It is important to note that *by no means do we claim our transformation techniques are new*, as they have also been studied in other evaluation systems for malware detection (e.g., [14, 15, 37, 38]). On the other hand, ensuring the applicability of existing transformation techniques in *general* Android applications remains a challenging issue. In the following, we consider a number of transformation techniques that can be *automated* (i.e., without manual intervention) for general `.apk` files. Hence, one can easily generate malware variants for a large number of malware samples.

4.1 Repackaging

In ADAM, we consider different repackaging methods that work directly on an input `.apk` file and regenerate a different `.apk` file *without modifying the source code of the input* `.apk` *file*. Thus, making the transformation easily deployable and preserving the functionality of the input `.apk` file. We consider three techniques that are currently supported by ADAM. One common key feature of all such techniques is that they are all built on the official Android or Java development utilities, which we expect are more robust and stable than other third-party tools.

Alignment. The alignment technique realigns the data of an `.apk` file, so as to generate different content but preserving the same logic for an `.apk` file. We use zipalign[5] to realign the uncompressed data within an `.apk` file (e.g., images or raw files) on 4-byte boundaries so that all portions can be accessed directly via `mmap()` function. The zipalign utility is available in the Android SDK, and is originally designed for providing optimization for `.apk` files. Since the alignment optimization changes the internal structure of the `.apk` file, it accordingly changes some of the signature patterns, such as the cryptographic hash of the `.apk` file. If an anti-virus system directly identifies malware based on the cryptographic hash signature (e.g., MD5), then the alignment technique can easily evade the detection of anti-virus system.

Our system applies alignment to an `.apk` file as follows. It is recommended [5] that all `.apk` files are aligned on 4-byte boundaries to achieve optimization. Thus, it is possible that the original input `.apk` file has already been 4-byte aligned. To ensure that the `.apk` file is actually transformed to a different output, our current implementation applies zipalign with the 8-byte alignment boundaries.

Re-sign. The re-sign technique is to generate a different signature for an `.apk` file. Android requires that every `.apk` file be digitally signed before the `.apk` file can be published and run on a smartphone. According to the official documentation [4], it is allowed and typical that an `.apk` file is self-signed by its application developer *without involving a trusted central authority*. One of our observations (which is not officially documented) is that an `.apk` file can be re-signed multiple times with different certificates and private keys, so that a different signature is generated and attached to the `.apk` file. This can evade the detection of anti-virus systems that simply identifies malware by its original `.apk` signature.

To (re-)sign an application, we use the Keytool and Jarsigner utilities, both of which are available in the Java SDK [41]. We first use Keytool to generate a self-signed key and put the key in a key store. We then use Jarsigner to sign an `.apk` file using the key store as the input.

Rebuild. The rebuild technique disassembles an `.apk` file and rebuilds the assembly code (without being modified) into another `.apk` file. We use apktool [9] to disassemble the Dalvik bytecode within an `.apk` file into Smali code [34] (see Section 2), and rebuild the Smali code back to Dalvik bytecode using apktool. After the disassemble process, the original `.apk` file and the repackaged `.apk` file are exactly the same, but repackaged `.apk` file will have different Dalvik bytecode order from the original one depend on the parser's analysis. This makes the resulting Dalvik bytecode (and hence the `.apk` file) different from the original one and at the same time, preseves the logic and functionality of the original `.apk` file.

We then sign the output `.apk` file with a randomly generated private key as described in the re-sign technique. This rebuild technique is effective to evade anti-virus systems that use cryptographic hash and/or `.apk` signature for malware identification.

4.2 Code Obfuscation

In code obfuscation, we modify the program code of an `.apk` file so as to make an anti-virus system more difficult to reverse engineer[17]. In particular, code obfuscation changes the size and content of the `.apk` file, but without modifying the logical behavior. Our code obfuscation techniques operate on Smali code [34], an assembly-like language based on the Dalvik executable code. The Smali syntax provides the debug information of how the variables and methods are invoked. This enables us to easily add obfuscated code to an `.apk` file.

To apply code obfuscation to an `.apk` file (of a malware sample), we first disassemble it using the apktool utility [9] into a `.smali` file. We modify the `.smali` file according to each obfuscation technique which we will describe shortly. We then rebuild the modified `.smali` file into an `.apk` file using apktool and sign the output `.apk` file, as in our previously proposed rebuild and re-sign techniques, respectively (see Section 4.1). One can use zipalign [5] for data alignment so to generate an optimized `.apk` file.

There are various code obfuscation techniques proposed in the literature, especially for the Java language (e.g., see [17]) on which Android is based. Note that some code obfuscation techniques depend on the underlying semantics of a program, and typically require *manual code modification* that cannot be easily automated. For example, substitution of code with different lines of code may need to be carefully carried out so as to preserve the original malicious behavior [37]. Also, our goal is to show that even with simple obfuscation techniques, one can generate new malware samples that can easily evade the detection of anti-virus systems. Thus, we consider several general code obfuscation techniques *that can be automated*, while being sufficient to subvert most of the anti-virus systems.

Inserting defunct methods (e.g., [14, 15]). We add new methods that perform defunct functions to Smali code, and these inserted methods do not change the logic of the original source code. The rationale of this obfuscation technique is to modify the *method table* in the Dalvik bytecode, and hence change the signature that is generated based on the method table.

There are many ways to add defunct methods. In our implementation, we implement a `Log.d` debug method [3] that prints a simple string in Android (obviously, other defunct methods can be added to ADAM). We first disassemble the method into Smali format. We then insert the `Log.d` method before the constructor method of each class in the disassembled `.smali` file. To locate a constructor method, we search for the string “`# direct methods`” in each `.smali` file, because the constructor method must follow this string. Figure 2 shows how we insert a defunct method.

Renaming methods (e.g., [37]). We obfuscate a method name with a different string, and hence change the signature that is generated by the method name. In our implementation, we first identify all the system library method names from `Android.jar` in Android SDK, so as to differentiate them from user-defined methods. Then we search for all user-defined methods (i.e., other than the library methods) in each `.smali` file, and append a randomly generated string (e.g., “`abc10`”) at the end of each user-defined method that we find. We modify the method name when the method is first defined, as

```
...
# direct methods
.method public OFLog(Ljava/lang/String;)V
...
.method public constructor <init>()V
...
```

Fig. 2. Inserting a defunct method (i.e., `OFLog`).

well as when it is called within the code. We point out that our system can also rename other types of identifiers, including packages, variables, and classes, so as to make the code more obfuscated. For example, Figure 3 shows how we rename a user-defined method "`foo`" into "`fooabc10`". To summarize, this type of obfuscation can evade anti-virus system that uses method names to generate virus signatures.

```
.method public static fooabc10(Ljava/lang/String;)V
...
invoke-static {v1}, Lcom/test;->fooabc10(Ljava/lang/String;)
```

Fig. 3. Renaming a method from `foo` to `fooabc10`.

Changing control flow graphs (CFGs). Some anti-virus systems (e.g., Androguard [1]) can use CFGs to generate signatures and detect the presence of malware. A CFG signature can be defined based on a grammar table [12]. Here, we modify the CFG of a `.smali` file and so as to change its CFG signature.

We consider one particular CFG obfuscation called the *Goto-obfuscation*. We insert `goto` statements to each method in a `.smali` file. At the beginning of a method, we insert a `goto` statement to jump to the end of the method; at the end of the method, we insert another `goto` statement to return to the beginning of the method. We insert a `return` statement before the second `goto` statement, so that the latter will not be called again when the method is finished. Figure 4 illustrates how we insert `goto` statements into a method `foo`.

```
.method public foo(Ljava/lang/String;)V
 .prologue
 goto :CFGGoto2
 :CFGGoto1
...
 return-void
 :CFGGoto2
 goto :CFGGoto1
.end method
```

Fig. 4. Goto-obfuscation.

Encrypting constant strings. We encrypt all constant strings that we find in a `.smali` file, and decrypt them when they are being processed. This modifies the signatures that are generated by these constant strings. Here, we consider a simple symmetric encryption method based on the Caesar cipher, in which we shift the character byte of each alphabet letter (i.e., `A-Z` or `a-z`) by a constant integer value. For example, we can encrypt a string "`DecryptString`" in a TextView control by subtracting all bytes by 10. The encrypted string will become "`:[YhofjIjh_d]`". We then add the decryption method `decrypt` (i.e., by adding all bytes by 10) before the TextView control is called. Figure 5 shows how the example works. In summary, this type of obfuscation can evade anti-virus system that uses constant string to generate virus signature.

```
#direct methods
.method public static DecryptString\
(Ljava/lang/String;)Ljava/lang/String;
...
const-string v1, ":[YhofjIjh_d]"
...
invoke-static { v1},\
Lcom/test;->DecryptString\
(Ljava/lang/String;)Ljava/lang/String;
move-result-object v1
invoke-virtual {v0, v1}, Landroid/\
widget/TextView;->setText\
(Ljava/lang/CharSequence;)V
```

Fig. 5. Encrypting a constant string.

5 Evaluation of Anti-Virus Systems

In this section, we use ADAM to evaluate the effectiveness of current commercial anti-virus systems in the detection of Android smartphone malware, and examine their robustness of dealing with malware variants as stated in our previous section. We conduct large-scale analysis using ADAM as follows. We collect a total of 222 Android malware samples in the wild, and generate different variants for each collected malware sample based on our transformation techniques (see Section 4). We feed these malware variants into different commercial anti-virus systems, and test if these variants are diagnosed as malware. Our analysis provides us a comprehensive picture of the effectiveness of current commercial anti-virus systems. Most notably, it enables us to validate the *automated* and *generic* properties of ADAM in experimenting with large-scale malware samples and anti-virus systems.

5.1 Malware Dataset

We collect a total of 222 *distinct* Android malware samples (with unique MD5 hashes) from three different sources, including:

- **Old public samples.** We download 57 Android malware samples from [18], a well-known blog website that maintains a collection of mobile malware. The blog author frequently updates the blog and shares the most recent malware samples to the public. The 57 samples are the public Android malware samples that are published from March 2011 to September 2011. Given the well-publicized blog, current anti-virus systems should have a very high detection rate on these malware samples.
- **New public samples.** On Oct. 22, 2011, the blog [18] published 96 new Android samples from an anonymous source. We believe it is interesting to study how fast the anti-virus systems add the signature of these new malware samples to their databases.
- **Private samples.** On Oct. 20, 2011, we obtained 69 Android malware samples from Antiy [8], an anti-virus company based in China. These samples are *unpublished*, so other anti-virus companies may not have enough signatures to detect these malware samples. Our hypothesis is that existing anti-virus systems will have a lower detection rate on these samples.

We carefully investigate the malicious logic of each of the 222 malware samples. We group the malware samples that have the same logic into a *family*. After our investigation, we group the 222 malware samples into 38 different families. Furthermore, we can classify the malware families into four categories, as we briefly describe below.

Repackaging malware. All malware samples in this category are transformed from legitimate applications via the disassembling approach (see Section 2.2). Briefly speaking, an attacker adds extra permissions and malicious services to a legitimate application, and repackages everything into a new (malicious) application. When a user installs the repackaged application, the application will perform malicious activities such as collecting the user's personal information and sending it to a remote server, or sending payment short messages to some premium SMS numbers. In our malware dataset, we have 138 malware samples from 12 families that fall into this category. For example, there are 32 samples of a family called Geinimi. All the Geinimi samples are transformed from different legitimate applications. Each of these malware samples has a common Java package called Geinimi, which contains a service called Adservice that performs malicious activities as listed above. This service modifies `AndroidManifest.xml` in the `.apk` file and starts automatically when the system boots up. In this category, there are also some well-known malware families such as DroidKungFu, BaseBridge, and Hongtoutou that have been studied in the literature [25].

Display-modification malware: This type of malware has a feature that all malware samples have the same application structure and same malicious behavior, but they have different icons, names, wallpapers, themes, or pictures. We have 46 malware samples of two families in our malware dataset that belong to this category. For example, the Kmin family is a wallpaper changer application, and contains 42 samples in our dataset.

Camouflage malware. This category of malware has a key feature that they imitate the same user interface of some original application in order to steal a user's account credentials. We have a total of 13 samples of six families in our malware database. For example, one family is called FakeNetflix, whose package name is "com.netflix.mediaclient" and is the same as that of the legitimate application Netflix. This family of malware displays a login screen to the user so as to steal the user's password and send it to a remote server.

Generic malware. This type of malware is just a plain malicious application without being camouflaged as any legitimate application. An attacker may simply physically access a smartphone and install the malware there. We have 25 malware samples of 18 families in our malware dataset under this category. For example, one family is called NickiSpy, whose package name is called "com.nicky.lyyws.xmall". It can steal users' credentials and wiretap users' phone calls in the background. It can also record any phone conversation and store it under the directory named "shangzhou/callrecord" in the SD card.

5.2 Anti-Virus Systems

We conduct our evaluation against the commercial anti-virus products hosted on the web portal VirusTotal [51] (see Section 3.2) in October and November 2011. Note that VirusTotal hosts over 40 anti-virus products, and our study only focuses on the *top 10* products that give the highest detection rates for our 222 original malware samples (i.e., without transformations) in November 2011.

In addition, in February 2012, we also evaluate a commercial anti-virus product that we obtained from Antiy [8], and the product is known to run the same engine as that being deployed in smartphone platforms.

We note that some anti-virus systems, such as Androguard [1], can detect malware based on control-flow-graph signatures (see Section 4.2). However, our evaluation does not consider Androguard, whose latest version is released in September 2011 at the time of this paper being written, while our malware samples are collected since October 2011. We think that it is unfair to evaluate Androguard using the malware samples collected after its latest release.

5.3 Analysis

For the 222 malware samples we collected, we apply our transformation techniques stated in Section 4, including three repackaging techniques and four code transformation techniques, to each malware sample. All samples can be successfully transformed by the Re-sign technique. However, two of the samples can be transformed by the Alignment technique, but cannot be transformed by the Rebuild and the four code transformation techniques because of the re-compilation errors. Also, 10 of the samples cannot be transformed by all techniques except Resign, mainly because they just contain `.dex` files that cannot be rebuilt into `.apk` files. Therefore, we can only generate a total of 1484 variants. Nevertheless, our transformation techniques have a success rate of 95.5%, showing the robustness of ADAM.

5.4 Results

We evaluate all malware samples and their transformation variants against different anti-virus systems.

AV Products	Original	Alignment	Re-sign	Rebuild
Kaspersky	95.95%	94.34%	94.59%	94.76%
F-Secure	95.50%	95.75%	95.05%	91.90%
Emsisoft	94.59%	93.87%	93.69%	75.24%
Ikarus	94.59%	94.34%	93.69%	75.24%
GData	94.14%	93.87%	93.69%	90.95%
TrendMicro	94.14%	91.98%	92.79%	77.62%
NOD32	92.79%	88.68%	88.29%	95.24%
Sophos	92.79%	94.81%	94.14%	78.10%
Antiy-AVL	92.34%	91.98%	89.19%	72.38%
Fortinet	90.99%	89.15%	88.74%	71.43%
Overall Average	93.78%	92.88%	92.39%	82.29%

(a) Detection of original malware samples and their variants generated by repackaging.

AV Products	Insert	Rename	Change CFG	Str. Encrypt
Kaspersky	93.81%	73.33%	94.76%	90.95%
F-Secure	90.00%	90.00%	90.48%	68.57%
Emsisoft	83.81%	26.67%	82.86%	25.24%
Ikarus	83.81%	26.67%	83.33%	25.24%
GData	90.95%	90.48%	91.43%	88.10%
TrendMicro	61.90%	61.90%	63.81%	35.71%
NOD32	95.24%	91.90%	95.24%	90.48%
Sophos	54.29%	54.29%	54.76%	49.05%
Antiy-AVL	70.00%	19.05%	67.14%	19.52%
Fortinet	48.57%	15.71%	42.86%	16.67%
Overall Average	77.24%	55.00%	76.67%	50.95%

(b) Detection of malware variants generated by code obfuscation.

Table 1. Detection rates for various anti-virus systems: Time: November 2011.

(1) Analysis of all transformation techniques. Table 1 shows the experimental results of the detection rates of each of the top 10 anti-virus systems that we choose, while the evaluation was conducted on November 21, 2011. We discuss our findings below.

• *Original malware samples.* We first test the original malware samples that we collected (without applying any transformation). The top 10 anti-virus systems we consider performed well in the original sample detection, they all have over 90% of detection rates. The average detection rate is 93.78%. This indicates that anti-virus companies have already begun to value the security of Android systems, and responded quickly to the emergence of new Android malware. In the following, we analyze the detection rates due to different transformation techniques when compared to the detection of the original malware samples.

• *Alignment.* As mentioned in Section 4, alignment via zipalign only changes the cryptographic hash signature of an `.apk` file. After alignment, there are only slight drops of the detection rates for all anti-virus systems (by at most 4%).

• *Re-sign.* We re-sign the `.apk` file of each malware with a random key. We observe that the detection rates of all the 10 anti-virus systems that we consider are not much different from the results for the original samples and the alignment transformation. We believe the reason is that most anti-virus products apply the unzipping process to deal with the `.apk` files before scanning. The re-signed `.apk` file will be no different from the original `.apk` file after the unzipping process because the signature process is based on the whole `.apk` file, and does not change the content of an `.apk` file. Note that after being re-signed, each `.apk` file has a different cryptographic hash. This may reduce the detection rate if an anti-virus system relies on cryptographic hashes as signatures, similar to the observations in the alignment transformation.

• *Rebuild.* After the rebuild process, the average detection rate of the 10 anti-virus products drops from 93.78% (in the original sample detection) to 82.29%. To understand this phenomenon, we used Dedexer [50] and UltraCompare [31] to analyze the original samples and the rebuilt variants. Dedexer is a disassembler tool for `.dex` files. Unlike apktool, the Dedexer tool can be used as a `.dex` parser to generate a detailed log file on the internal structure of a `.dex` file. UltraCompare is a comparison utility that can handle binary file comparison, text comparison and folder comparison.

After rebuilding a `.dex` file, the result shows that the `.dex` file has changed. We use UltraCompare to compare the detailed log files of these `.dex` files. We find out that the checksum, the signature, some offsets, and some size values have changed. In addition, although the strings or method names do not change, their index orders are different from the original `.dex` file. These changes imply that just using fragments of a `.dex` binary file as the malware signature may not be effective, and it may reduce the detection rate.

• *Insert defunct methods.* After the insert defunct methods transformation, the average detection rate of the 10 anti-virus systems has decreased from 93.78% down to 77.24%. Then we use UltraCompare to compare the detailed log files of the original samples and malware variants. The most distinctive difference between the rebuilt variant and the insert-defunct-methods variant is the method table. In the method table, the total number of methods has changed and the size of the method table becomes bigger because we insert additional defunct method implementations. Therefore, if an anti-virus system uses the hash value of all of the method names as the signature, then adding defunct methods will make the detection ineffective. For example, we insert defunct methods process to one of our malware samples called `snake`. Then we compare the variant with its original sample. We find that the total number of methods increases from 239 to 259, and the file size is only increased by 4%. Again, these changes are due to the insertions of defunct code.

• *Renaming methods.* After the renaming methods transformation, the average detection rate has decreased from 93.78% down to 55.00%. In particular, the detection rates of some anti-virus products drop significantly, for example, from 92.34% to 19.05% for Antiy-AVL. In addition to the changes in the rebuild process, the method table has also

AV Products	Original	Alignment	Re-sign	Rebuild
F-Secure	93.24%	95.28%	94.59%	89.05%
Kaspersky	93.24%	90.09%	89.64%	62.38%
Emsisoft	90.99%	90.09%	87.84%	61.90%
Ikarus	90.99%	90.09%	87.84%	61.43%
GData	88.74%	92.45%	91.44%	86.67%
Sophos	88.74%	86.32%	86.49%	68.10%
Antiy-AVL	86.04%	75.00%	73.42%	54.76%
TrendMicro	85.59%	75.94%	74.32%	53.81%
Fortinet	79.28%	68.87%	68.47%	43.33%
NOD32	77.93%	55.66%	52.25%	35.24%
Overall Average	87.48%	81.98%	80.63%	61.67%

Table 2. Detection rates for various anti-virus systems using original malware samples and their variants generated by the repackaging techniques: October 2011.

changed significantly. Also, the implementation of methods has also changed because the methods now invoke different method names. This indicates that if anti-virus systems use method names to generate signatures, then they may fail in the detection. We observe that the renaming method transformation is more effective in evading malware detection compared to inserting defunct methods.

• *Changing control flow graphs.* We used the Goto obfuscation technique so that every method implementation has been added with 4 lines of Goto statement while other changes are the same as the rebuild process. We find out that the result is similar to that of the insert-defunct-methods transformation. The average detection rate has decreased from 93.78% down to 76.67%.

• *String encryption.* After this transformation process, the string table and method table will change because of the string encryption and the insertion of decryption methods. The average detection rate of all anti-virus systems has decreased from 93.78% to 50.95%. This indicates that a lot of anti-virus systems that we consider use constant strings as the signature to detect the presence of malware.

(2) Evolution of malware detection. We used ADAM to carry out the *first* stress test on all anti-virus systems in October 2011 right after we collected all malware samples. Here, we only focus on the detection of original malware samples and their variants generated by the repackaging techniques. Table 2 shows the detection rates of the top 10 anti-virus systems that we consider in Table 1. Compared with the detection rates on Table 1, which we carried out the experiment in November 2011, we see that most of the anti-virus systems *improve* in the malware detection, in particular, on the original malware samples. This shows that anti-virus systems are *rigorously updating* their signature databases. However, there are still a number of anti-virus systems which are not robust against simple malware transformations based on repackaging.

In February 2012, we obtained from Antiy [8] an anti-virus engine that runs atop a desktop PC with the Linux operating system and is known to have the same detection

Original	Alignment	Re-sign	Rebuild
94.59%	96.69%	94.59%	95.23%
Insert	**Rename**	**Change CFG**	**Str. Encrypt**
94.28%	93.80%	98.57%	94.28%

Table 3. Detection rates of the Antiy's anti-virus product in February 2012.

logic as that being deployed in smartphone platforms. We conduct evaluation against it in February 2012 using the same set of variants. Table 3 shows the results. We observe that the detection rates for the original malware samples and all their variants can achieve over 90%. This shows that commercial anti-virus products evolve to become more robust against malware transformations.

6 Discussion

In this section, we describe several open issues that we have not addressed in this work, and suggest the future directions.

Signature coverage. While we show that our transformation techniques can make a malware application evade the detection of a number of commercial anti-virus systems, it is non-trivial to accurately infer the underlying signatures being used by such systems. Also, although we confirm that some companies use the same anti-virus engine for both desktop and mobile versions (see Section 5.2), we cannot verify if all anti-virus systems that we tested on VirusTotal apply the same detection logic as in their mobile versions, as the latter can be better. One future work is to apply ADAM to evaluate both desktop and mobile versions of an anti-virus product and compare their detection performance.

Distribution model. We point out that it is generally difficult to distribute malicious applications through the official Android Market because of strict application checking. However, we believe that hackers can upload any malware to the third-party markets, given that the Android Market may be banned by some countries such as China [53]. Thus, users may have to use third-party markets to access mobile applications. In addition, "rooted" smartphones can install any applications and bypass any strict checking imposed by the Android OS, thereby making the spread of malware more feasible. It is interesting to further study the impact of the distribution model of mobile applications on the spread of malware.

Defense solutions. We propose several solutions that can defend against obfuscation and repackaging techniques we discussed. First, one can use a `.dex` parser to extract signatures from a `.dex` file to counter common obfuscation methods, because the logic and functionality of the `.dex` file does not change. Second, a good optimizer can handle the insertion of defunct methods and changing control flow graph methods, because all redundant code can be eliminated after the code optimization method. Third, using fuzzy hashing [29] to detect unknown malware appears to be a promising approach, but how to find the optimized parameters to control the anchor points remains a challenging research problem.

7 Related Work

Smartphone malware (e.g., viruses, worms, and trojans) presents a critical security threat to smartphones. A piece of malware can reside in smartphones, perform malicious activities, and compromise the trusts of smartphones. As the first smartphone worm Cabir appeared in 2004 [47], the research community has been alerted about the severity of smartphone malware [19, 26, 32]. While smartphone malware first appeared in Symbian OS, Schmidt *et al.* [45] implemented the first malware for Android. Since then, there has been a rapid spread of malware in different mobile platforms including Android and iOS (see the survey of [25]). Recently, Schlegel *et al.* [43] demonstrated an Android malware called Soundcomber that can steal voice data with only limited permission privileges.

Existing commercial anti-virus solutions identify smartphone malware mainly based on static signature-based detection, which aims to identify any malicious patterns of the source code of an application without executing it. However, smartphones typically have scarce computational and bandwidth resources, and so it is ineffective to have smartphones deploy anti-virus solutions and update the latest signatures in a timely manner. SmartSiren [13] is a proxy-based, collaborative detection system that collects the activities from various smartphones in order to detect the existence of malware. Bose *et al.* [10] propose a machine-learning-based framework that detects the presence of smartphone malware by looking into malicious behavior signatures, and show that behavioral detection gives higher detection accuracy and is more resilient to code transformation than conventional signature-based detection. Schmidt *et al.* [44] consider a similar collaborative system as in [13] and use behavioral detection, with the emphasis on Android systems. Paranoid Android [42] uses remote servers to examine the replicas of Android phones and identify security threats. Crowdroid [11] is a behavioral detection malware system for Android, and collects the system-call traces of various real Android users to identify malware. In summary, the above approaches mainly use a network-based system that remotely runs the malware detection.

There are host-based malware detection systems that directly run on smartphones. Xie *et al.* [52] propose access-control defense to limit the accesses of malware to critical system resources. VirusMeter[35] identifies malware that causes excessive battery power consumption on mobile devices. Andromaly[46] is an Android-based malware detection system that applies machine learning to identify anomalous behavior.

A number of researchers (e.g., [27, 39]) motivate the needs and challenges of testing security software, and AMTSO [7] is one major organization that propose different standards for testing anti-virus systems. There have been research studies that focus on testing the resilience of existing malware detection systems. Christodorescu and Jha [14, 15] show that simple code obfuscation techniques suffice to evade the detection of commercial anti-virus systems, which are mainly built on static signature-based detection. Moser *et al.* [38] show that obfuscation techniques based on opaque constants can evade static detection systems that consider instruction semantics (e.g., [16]). Note that these studies mainly consider malware on PCs but not on mobile devices. Morales *et al.* [37] evaluate the resilience of commercial anti-virus systems for the Windows Mobile OS, and consider it only with two virus samples and four commercial anti-virus systems. Our work, on the other hand, targets the Android OS and covers significantly larger sets of virus samples and commercial anti-virus systems.

There are studies that investigate the security issues in Android smartphones, such as privacy leakage and permission usage. Enck *et al.* [23] analyze the security of existing Android applications, by decompiling and recovering the Java source code of Android applications in Google's Android Market. Taintdroid [22] uses dynamic taint tracking to identify any privacy leakage in Android applications (a similar privacy leakage detection system PiOS [21] is designed for Apple iOS). AppFence [28] extends Taintdroid by controlling how private data can enter or leave an Android application. Stowaway [24] uses static analysis to identify the permission usage of the API calls in Android applications. Our work mainly focus on generating malware threats and examine the effectiveness of malware detection in Android smartphones.

8 Conclusions

We present ADAM, an automated, generic, and extensible platform that evaluates the detection of Android malware detection systems. ADAM applies different transformation techniques, including repackaging and code obfuscation, to an Android malware sample to generate different variants. Then it applies these variants to stress test the robustness of a wide range of anti-virus systems. ADAM is designed to be automatic, generic, and extensible for assessing the state of the art of Android malware detection. We conduct large-scale studies based on 222 Android malware samples against various commercial anti-virus systems, so as to demonstrate how ADAM provides recommendations to improve current detection mechanisms. Our ADAM prototype is available for download at: **http://ansrlab.cse.cuhk.edu.hk/software/adam**.

Acknowledgments

We would like to thank our shepherd, Lorenzo Martignoni, and the anonymous reviewers for their valuable comments.

References

1. Androguard. `http://code.google.com/p/androguard/`, 2010.
2. Android. Android Developers - Building and Running. `http://developer.android.com/guide/developing/building/index.html`.
3. Android. Log. `http://developer.android.com/reference/android/util/Log.html`.
4. Android. Signing Your Applications. `http://developer.android.com/guide/publishing/app-signing.html`.
5. Android. zipalign. `http://developer.android.com/guide/developing/tools/zipalign.html`.
6. Android Market. `https://market.android.com/`.
7. Anti-Malware Testing Standards Organization. `http://www.amtso.org`.
8. Antiy. `http://www.antiy.net`.

9. Apktool. http://code.google.com/p/android-apktool/.
10. A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral Detection of Malware on Mobile Handsets. In *Proc. of ACM MobiSys*, 2008.
11. I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.
12. S. Cesare and Y. Xiang. Classification of Malware Using Structured Control Flow. In *Proc. of the Eighth Australasian Symposium on Parallel and Distributed Computing*, 2010.
13. J. Cheng, S. H. Wong, H. Yang, and S. Lu. SmartSiren: Virus Detection and Alert for Smartphones. In *Proc. of ACM MobiSys*, 2007.
14. M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proc. of USENIX Security Symposium*, 2003.
15. M. Christodorescu and S. Jha. Testing Malware Detectors. In *Proc. of ISSTA*, 2004.
16. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, 2005.
17. C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Dept of Computer Science, University of Auckland, New Zealand, Jul 1997.
18. Contagio Mobile. http://contagiominidump.blogspot.com/.
19. D. Dagon, T. Martin, and T. Starner. Mobile Phones as Computing Devices: The Viruses are Coming! *Pervasive Computing*, 3(4):11–15, Oct 2004.
20. dex2jar. http://code.google.com/p/dex2jar/.
21. M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proc. of NDSS*, 2011.
22. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of USENIX OSDI*, 2010.
23. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. of USENIX Security Symposium*, 2011.
24. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proc. of ACM CCS*, 2011.
25. A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *Proc. of ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.
26. C. Guo, H. J. Wang, and W. Zhu. Smart-Phone Attacks and Defenses. In *ACM SIGCOMM HotNets*, 2004.
27. D. Harley. Making Sense of Anti-Malware Comparative Testing. *Information Security Tech. Report*, 14(1), Feb 2009.
28. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proc. of ACM CCS*, 2011.
29. D. Hurlbut. Fuzzy Hashing for Digital Forensic Investigators. http://accessdata.com/downloads/media/Fuzzy_Hashing_for_Investigators.pdf, May 2011.
30. IDC. Worldwide Smartphone Market Expected to Grow 55% in 2011 and Approach Shipments of One Billion in 2015, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS22871611, Jun 2011.
31. IDM Computer Solutions, Inc. File Compare — UltraCompare Professional. http://www.ultraedit.com/products/ultracompare.html, 2011.
32. J. Jamaluddin, N. Zotou, R. Edwards, and P. Coulton. Mobile Phone Vulnerabilities: A New Generation of Malware. In *Proc. of IEEE Int. Symp. on Consumer Electronics*, 2004.

33. Java Decompiler. http://java.decompiler.free.fr/.
34. JesusFreke. smali. http://code.google.com/p/smali/, 2011.
35. L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing Your Cellphone from Spies. In *RAID Symposium*, 2009.
36. McAfee Labs. McAfee Threats Report: Second Quarter 2011, 2011.
37. J. A. Morales, P. J. Clarke, and Y. Deng. Testing and Evaluating Virus Detectors for Handheld Devices. *Journal in Computer Virology*, 2(2):135–147, Sep 2006.
38. A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *Proc. of ACSAC*, 2007.
39. I. Muttik and J. Vignoles. Rebuilding Anti-Malware Testing for the Future. In *Virus Bulletin Conference*, 2008.
40. J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proc. of USENIX Security*, 2008.
41. Oracle. JDK Tools and Utilities. http://download.oracle.com/javase/1.5.0/docs/tooldocs/\#security, 2010.
42. G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile Protection For Smartphones. In *Proc. of ACSAC*, 2010.
43. R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proc. of NDSS*, 2011.
44. A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. K. y, K. A. Yüksely, S. A. Camtepe, and S. Albayrak. Static Analysis of Executables for Collaborative Malware Detection on Android. In *Proc. of IEEE ICC*, 2009.
45. A.-D. Schmidt, H.-G. Schmidt, L. Batyuk, J. H. Clausen, S. A. Camtepe, S. Albayrak, and C. Yildizli. Smartphone Malware Evolution Revisited: Android Next Target? In *Proc. of MALWARE*, 2009.
46. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intell. Info. Syst.*, 37:1–30, 2011.
47. Symantec. SymbOS.Cabir. http://www.symantec.com/security_response/writeup.jsp?docid=2004-061419-4412-99, June 2004.
48. Tencent Mobile Security Lab. Disguised the explosive growth of the virus. http://www.tastecate.com/freepages353623, Oct 2011.
49. TGDaily. Smartphone Malware at an All-Time High. http://www.tgdaily.com/security-brief/53060-smartphone-malware-at-an-all-time-high, Dec 2010.
50. Vesselin. Dexid. http://dl.dropbox.com/u/34034939/dexid.zip, 2011.
51. VirusTotal. http://www.virustotal.com.
52. L. Xie, X. Zhang, A. Chaugule, T. Jaeger, and S. Zhu. Designing System-Level Defenses against Cellphone Malware. In *Proc. of IEEE SRDS*, 2009.
53. S. Ye. Android Market is Currently Blocked in China. Here are your Alternatives, Sep 2011. http://techrice.com/2011/10/09/android-market-is-currently-blocked-in-china-here-are-your-alternatives/.