Noname manuscript No.
(will be inserted by the editor)

Testing Android Malware Detectors Against Code Obfuscation: A Systematization of Knowledge and Unified Methodology

the date of receipt and acceptance should be inserted later

Abstract The authors of mobile-malware have started to leverage program protection techniques to circumvent anti-viruses, or simply hinder reverse engineering. In response to the diffusion of anti-virus applications, several researches have proposed a plethora of analyses and approaches to highlight their limitations when malware authors employ program-protection techniques. An important contribution of this work is a systematization of the state of the art of anti-virus apps, comparing the existing approaches and providing a detailed analysis of their pros and cons. As a result of our systematization, we notice the lack of openness and reproducibility that, in our opinion, are crucial for any analysis methodology. Following this observation, the second contribution of this work is an open, reproducible, rigorous methodology to assess the effectiveness of mobile anti-virus tools against code-transformation attacks. Our unified workflow, released in the form of an open-source prototype, comprises a comprehensive set of obfuscation operators. It is intended to be used by anti-virus developers and vendors to test the resilience of their products against a large dataset of malware samples and obfuscations, and to obtain insights on how to improve their products with respect to particular classes of code-transformation attacks.

Keywords Android malware detection, code obfuscation

1 Introduction

Android is the most popular mobile platform with a market share of 82.8% [6]. The application-distribution workflow is such that Android developers can sign their applications using self-signed certificates and publish them through the Google Play Store or alternative marketplaces. No central, trusted certificate authority is required and, as a matter of fact, no public key infrastructure is implemented. This makes it relatively easy to develop and distribute Android applications, including malicious ones. Thus, it is not surprising that, according to the leading security vendors and researchers, Android is not only the most popular mobile platform, but also the most targeted one. Indeed, threats against Android account for 97% of the new mobile malware discovered in the second half of 2013 [14]. This trend was confirmed by the Symantec's report where the authors observed that 46 new families of Android malware were discovered in 2014 [26]. Moreover, in this report, researchers observe that malware developers are continuously increasing the number of variants per family, for example, by repackaging well-known applications with malware.

The proliferation of malicious applications has led to the development of a plethora of commercial anti-malware (or anti-virus, or AV) products, distributed as free and paid applications in the various marketplaces. These detection tools typically rely on signatures matching for the automatic identification of malware. A signature is a program feature extracted from a malicious application that is meant to characterize the malicious nature of the application. Thus, any application that exhibits a malware signature is classified as malicious. Given the sandboxed runtime and restrictive security model of Android, anti-malware have very limited auditing capabilities. In the best case, without breaking the security model (i.e., without requiring root privileges), anti-malware products can scan applications using signatures that encode syntactic features.

Problem Statement. Malware authors have started to leverage several techniques, such as code obfuscation, to bypass detection. Indeed, given the aforementioned limitations of Android AVs, signatures can be easily evaded. With the term *code obfuscation* we refer to program transformations designed specifically to make static analysis difficult while preserving the program's original behavior [11]. Code obfuscation changes the code of a program, namely how the program is written, but not the semantics of a program, namely what the program computes. Thus, an obfuscating transformation transforms a program into a semantically equivalent one that is more difficult to analyze manually or with automatic tools. Malware writers have widely employed offensive code-obfuscation techniques in desktop scenarios and it is not surprising that such techniques are nowadays ported to the mobile world. Thus, it is of paramount importance to have rigorous methodologies to assess the effectiveness of mobile anti-malware tools against obfuscation. In the past, Christodorescu and Jha [9] in their seminal paper proposed a methodology to be applied to test the robustness of generic malware detectors against obfuscation transformations.

Goals of this Work. In this work, we systematize the state of the art in testing mobile malware detectors and we propose a unified workflow that anti-malware developers and vendors can use for (1) testing the resilience of their products against a large dataset of malware samples and obfuscations and (2) obtain insights on how to improve their products with respect to particular classes of obfuscations. More precisely, our approach, called AAMO (Automatic Android Malware Obfuscator), provides an automatic framework for the application of existing and potentially novel obfuscating transformations to large-scale datasets of Android malware. One goal of our testing methodology is to find the minimal sequence of obfuscation operators, applied according to a predefined order, that an attacker can create to evade an anti-malware product. In particular, we fix an order of obfuscation operators and we apply them one after the other in order to identify the shortest sequence of obfuscation operators that causes a false negative. Although this latter objective is similar to that of previous work (most notably [24, 20]), we notice a lack of well-established experimental protocols in this area, which instead is crucial for precise reproducibility. Although we validate our approach on a representative dataset (against the top 6 anti-malware products), we believe that the research community demands something beyond a comparative analysis or the mere demonstration that code-obfuscation techniques are effective. Therefore, rather than just providing a "snapshot" of the current anti-malware situation, which would be immediately obsolete (e.g., due to changes in the anti-malware products or dataset, and to the development of new obfuscation techniques), the focal point of AAMO is to provide a future-proof methodology and prototype that future researchers will be able to use.

The specifics of the AAMO framework for testing malware detectors for Android are as follows.

- **Comprehensiveness.** AAMO considers a large and comprehensive pool of obfuscation operators that includes all the obfuscations proposed in the literature for testing Android malware detectors.
- **Correctness.** As noted in [20], the use of certain obfuscation operators can make an application non functional. We consider an obfuscation operation as "successful" only if it produces a working application, which means that the evasion technique can be used by a malware developer.
- **Reproducibility and Fidelity.** Key to any anti-malware evaluation experiment is its reproducibility. Differently from previous work, we do not resort to closed source or commercial products to implement our obfuscation operators. We implement all the operators used in our experiments, and release the source code of the prototype.
- **Flexibility and Scalability.** Our lightweight code base allows new obfuscations to be implemented on top of AAMO and plugged in the framework. Moreover, AAMO allows to combine the obfuscations supported by the framework. AAMO can automatically obfuscate a large dataset of malware samples.
- **Insightfulness.** The analysis of the testing results of AAMO provides insights on how to improve the signature used by the AV with respect to the minimal combination of obfuscation operations that triggers a considerable amount of false negatives.

Summary of Contributions. This paper makes the following original contributions:

- We provide a systematization of the current state of the art in obfuscation for mobile applications, which can be used for benign purposes (e.g., software protection) as well as for malicious purposes (e.g., anti-malware evasion).
- We describe the design and implementation details of a new, general-purpose, advanced, mobile-tailored obfuscation tool for the Android platform.
- We presents the implementation of a framework for easily combining obfuscation techniques in order to obtain arbitrarily complex obfuscation transformations. The framework is fully automated, modular and it allows us to apply obfuscation techniques to a large-scale dataset of malware samples.

– We provide a validation of our framework on top of the AndroTotal research platform [19]. The main advantage in the use of AndroTotal is that this ensures that the tests are performed on the unmodified mobile anti-malware tools and reproduces the same conditions of a malware-detection task on the end user's device.
– The proposed framework for testing Andorid anti-malware tools provides insights to counteract the specific classes of obfuscation in order to implement future-generation anti-malware products. Indeed, we show that AAMO can be used to identify the class of obfuscation that breaks a given detection tool.

In the spirit of open science and for the benefit of the security of the Android ecosystem, we released a proof of concept implementation of AAMO as fully open source at `https://github.com/necst/aamo`.

Structure of the paper

In Section 2 we recall the basic background concepts and definitions regarding the Android platform. In Section 3 we recall the major researches published in the area of testing Android malware detectors, together with their limitation and current needs that motivated our work. Next, in Section 4 we provide an overview of our testing approach and the workflow that implements it in AAMO. Section 5 provides the details of the obfuscating techniques that we have implemented in AAMO, grouped into classes according to the code features that they target. Section 6 provides an instance of how AAMO is expected to work in a real-world use case. We present the obtained results and discuss them in order to provide suggestions for improving the considered AV products.

2 Android Platform

In this section we summarize the fundamental background concepts and terminology used in the paper.

2.1 Security and Runtime

The Android security model is based on application isolation. More precisely, it leverages the user-based resource separation mechanism offered by the underlying Linux kernel, in which users (along with their resources, processes, etc.) are kept isolated from one another. Apps are mapped one-to-one to users, and each of them runs in an isolated sandbox. This sandbox is implemented as an instance (i.e., process) of the Dalvik VM, a register-based machine with a high-level instruction set. For optimized performance, Dalvik executables (also called DEX files) can be just-in-time or ahead-of-time compiled into the native instruction set. The latter case have evolved into the current runtime called Android RunTime (ART), which executes native instructions. To allow sandboxed applications to interact with each other and the system, applications must declare the permissions that they want to use in order to access the resources (e.g., phone, network connectivity, Bluetooth, contacts).

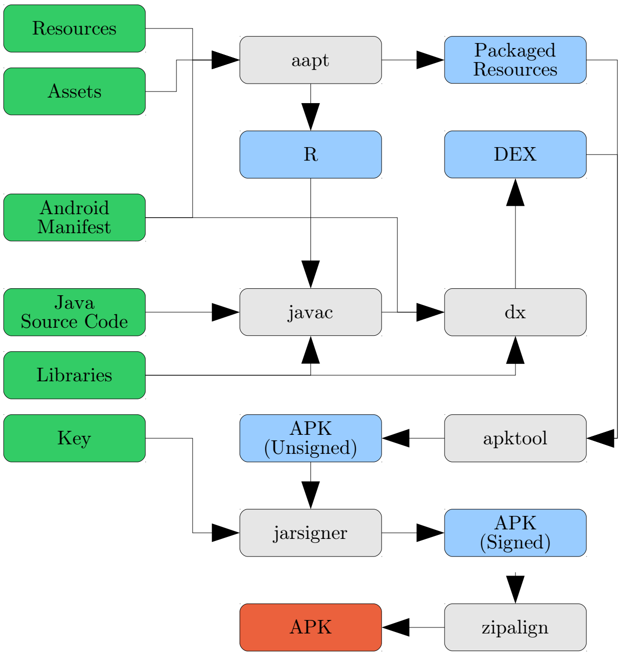


Figure 1 Android application build process.

2.2 Development and Deployment

Android is an operating system and development platform specifically designed for mobile devices such as smartphones, tablets, infotainment systems, or similar embedded devices. Android is build on top of Linux, which provides an hardware abstraction layer, ensuring portability to a wide range of architectures. It also provides memory management, process management, a security model and networking. On top of Linux, Android provides a set of low-level libraries and APIs in order that abstracts common tasks used by userspace applications. Android applications take advantages of an application framework, which includes Java-compatible libraries.

Android applications are mainly developed in Java, with native code extensions through the Java Native Interface. Applications are distributed as Android package (APK) files, which are compressed archives that contain the compiled Dalvik bytecode (see Section 2.3), resources (e.g., images, XML files, and other assets), certificates, and a manifest file. The manifest file contains a list of the permissions requested by the application, which must be approved by the user upon installation.

As in any Java application, each component is encapsulated in a class, which can be referenced from the manifest file. Referenced components are the entry points, and are used in order to interact with the Android environment (e.g., execute a function at boot, or upon receiving a call or text message).

In a typical Android context, only digitally signed applications can be installed. However, the lack of a public key infrastructure allows developers to use self-signed certificates to sign their applications and publish them through the Google Play Store. We consider the idea of submitting obfuscated samples to the Google Play Store interesting. However, since the Google Play Store uses the AVs from VirusTotal, the state of art work (based on VirusTotal) has already indirectly measured this aspect. There is a whole separate area dedicated to measuring the resilience of Google Play Store to the various forms of threats, and we believe our paper is not in that scope. Applications can be also retrieved from alternative marketplaces or directly downloaded into devices, making the problem of applying security checks even more complex.

The build process is summarized in Figure 1.

2.3 Bytecode and Representation

In this work we rely on an assembler-level representation of DEX instructions. To this end, we use the Smali language, which is compact yet easy to read, and well supported by program-analysis tools. In Smali, classes and files are mapped one to one, and each class is contained into a single Smali file named as the contained class. The directory structure reflects the packages hierarchy.

We hereby summarize the essential concepts of the Smali syntax required to define our obfuscation techniques in Section 5. Primitive types are represented by a single letter (e.g., `V` for void, `Z` for Boolean, `B` for byte, `S` for short). Reference types are objects and arrays, everything else is a primitive. Objects take the form `Lpackage/name/ObjectName`, where a leading `L` indicates that it is an object type. Arrays take the form `[Type`, where `Type` could be any type, including reference types. For arrays with multiple dimensions, multiple `[` character are added. The maximum number of dimensions is 255.

Methods are associated to the owner object using the `Owner;->MName(TypeType)Type` syntax, where parameters are simply listed, with no separators.

Fields are likewise always defined using a similar form, which refers to the owner, the field name and its type: `Owner;->Name:Ljava/lang/String`. Directive are declared with the `.Directive` token, and lines starting with `#` are interpreted as comments.

The Dalvik VM has 32-bit registers that can hold any type of value, using two adjacent registers for 64-bit types. There are two ways to specify how many registers are available in a method. The `.registers` directive specifies the total number of registers in the method. The `.locals` directive specifies the number of non-parameter registers in the method. The total number of registers includes the registers needed to hold the method parameters.

When a method is invoked, its parameters are placed into the last registers. The first parameter to a non-static method is always the object that the method is being invoked on (`this`), except for static methods. There are two naming schemes for registers: the normal `v` naming scheme, and the `P` naming scheme for parameter registers. The first register in the `P` naming scheme is the first parameter register in the method. Parameter registers can be equivalently referenced by either name.

2.4 Anti-malware Approaches and Limitations

There exist several commercial anti-malware products in response to the proliferation of malicious applications. In a traditional desktop environment, AVs applications run with administrative privileges, whereas on Android they must run with the same privilege level of a regular application. This restriction makes it difficult for AV applications to perform behavioral analysis and runtime scanning.

For this reasons Android applications are analyzed by using the provided APIs in order to scan the content of original APK files using static signature matching. Signatures are based on blocks of code, data- or control-flow graph fragments, identifiers, APIs calls sequences, strings, manifest, resources, and assets. In some (limited) cases, AV applications opt for cloud-based scanning. However, this is often reduced to pre-processing the APK offline and sending the result (e.g., hash, byte sequences extracted) to the AV engine's back end. Indeed, uploading the entire APK is not always applicable for obvious reasons (e.g., licensing issues, bandwidth, dynamic code loaded by the application).

2.5 Android Application Obfuscation

As part of the program-protection arsenal, code obfuscation is a powerful technique to render manual and automated reverse-engineering harder. Code obfuscation has been introduced for the first time in [11] as a promising technique to prevent malicious reverse engineering of programs that aims at violating the intellectual property of proprietary programs. Since then, code obfuscation has received the attention of both software developers for the protection of the intellectual property of their code (see [10] for a survey), and malware writers for the evasion of automatic detection tools (e.g., [18, 21]). Code obfuscation has also been studied from a theoretical point of view in order to prove the potentiality and limitations of this technique (e.g., [8, 12, 13, 16]).

The Android development tool chain includes a simple, open-source program-obfuscation tool called ProGuard, which removes debug symbols, normalizes identical code blocks, remove unused code, and "minifies" the identifiers' names. There exist a plethora of alternative commercial and open-source tools, which mainly differ from one another for their scope and obfuscation strength. For example, DexGuard is a commercial code optimizer and obfuscator derived from ProGuard. It provides reflection, identifier obfuscation for packages, classes, methods, and fields, and performs strings, resources, and asset and library encryption. Allatori, another commercial obfuscator, provides methods to modify the control flow graph in addition to string encryption. Similarly, Klassmaster provides flow obfuscation and string encryption. Dalvik-obfuscator and APKfuscator are open-source proof-of-concept obfuscators that leverage the junk- or dead-code insertion.

Generally, the goal of these (and other) obfuscation tools is to evade manual reverse engineering. As a consequence, however, code-obfuscation tools can be used to evade signature-based AVs because they produce a semantically equivalent program that differs from the original one. Indeed, according to [27], malware authors have been using obfuscation for evading AV detection, with both custom and commercial tools. For example variants of the Opfak.bo and Obad.a families were obfuscated with a commercial tool, and samples of the SmsSend.ND!tr family (March 2014) were obfuscated with the aforementioned APKfuscator. According to Fortiguard Labs [7], 17 to 27 percent of the malicious samples analyzed used ProGuard or some form of encryption, even if sometimes encryption is used in legitimate portions. For instance, samples of the Android/SmsSpy.HW!tr family (February 2014) had their XML configuration files encrypted and stored as an asset.

Since the focus of our work is not on code-obfuscation alone, but on its use for AV evasion and testing purposes, we refer the reader interested in this subject to [15, 23, 25].

3 State of the Art and Motivation

Christodorescu and Jha [9] in their seminal paper proposed a methodology to be applied to test the robustness of desktop malware detectors against obfuscating code transformations. The authors concluded that anti-malware tools have low resilience against offensive obfuscation. In this section, we systematize the state of the art in testing the detection rate of mobile malware in presence of obfuscation. J. Schette and Kulicke [17] have conducted a detailed investigation in order to better understand how Android malware can spread. At the same time, [17] was the first work to highlight the technical limitations of Android AV apps.

To the best of our knowledge, the main contributions in the area of Android AV testing are by Zheng et al. [29], Rastogi et al. [24], and more recently the work of Maiorca et al. [20]. The common aspect of all these contributions is that they focus primarily on highlighting the destructive effects of offensive code obfuscation against existing anti-malware apps. Rather, we believe that the value of such comparative studies lies in the methodology, not in the results. Indeed, the results inherently suffer from quick obsolescence due to the frequent signature updates and the invention of potentially novel code-obfuscation techniques. Maiorca et al. [20] strive to address this issue by repeating the experiments further times to quantify the changes. Although this has value, the authors do not tackle the issue of repeatably, which in our opinion is essential for evaluation methodologies.

In the remainder of this section, we provide a thorough review and comparison of the state of the art.

ADAM. The approach presented by Zheng et al. [29] is to generate the obfuscated variants either by directly modifying the APK files through repackaging transformations, or by disassembling the APK files, which are then obfuscated and re-assembled into bytecode and re-packaged into a new, obfuscated APK file. The evaluation presented in Zheng et al. [29] considers 222 malware samples, from which the authors generated 1,484 variants by applying their code-transformation techniques. Each variant is obtained by applying one of the above mentioned transformations. The final testing phase of ADAM considers two kinds of detection tools:

	ADAM	DroidChameleon		**AAMO**
Approach Characteristics	Zheng et al. [29]	Rastogi et al. [24]	Maiorca et al. [20]	(this work)
Reproducibility & Fidelity				
Use of Proprietary Obfuscators			✓	
Based on Research Tools				[19]
Thorough Implementation Details				✓
Source Code Availability	Upon Request [28]			[22]
Obfuscated Apps Validated	Can compile		✓	✓
Comprehensiveness				
Dataset Size (# Apps)	222	12	1,497	1,260
# Obfuscation Techniques (more details below)	12	8	10	17
# Anti-malware Products	11	10	13	6
Combined Transformations		✓	✓	✓
Obfuscation Techniques Implemented				
Android Specific				(Section 5.1)
Repackaging	✓	✓	✓	✓
Reassembly	✓	✓	✓	✓
Re-alignment		✓		✓
Simple Control-flow Modifications				(Section 5.2)
Junk Code Insertion	✓			✓
Debug Symbols Stripping	✓			✓
Defunct Code Insertion		✓		✓
Unconditional Jump Insertion		✓		✓
Advanced Control-flow Modifications				(Section 5.3)
Call Indirection	✓			✓
Code Reordering	✓			✓
Reflection	✓		✓	✓
Opaque Predicate Insertion				✓
Renaming				(Section 5.4)
Non-code files and resource renaming	✓		✓	✓
Fields and methods renaming	✓	✓	✓	✓
Package Renaming	✓	✓	✓	✓
Encryption				(Section 5.5)
Resource Encryption (asset files)			✓	✓
Native Code Encryption	✓		✓	✓
Data Encryption (strings)	✓	✓	✓	✓
Class Encryption			✓	

Table 1 Systematization of the state of the art in comparison to this work.

the top 10 VirusTotal [5] engines and one local engine (Antiy [1]). The results of their experiments show that the most effective repacking transformation lower the detection rate from $93,78\%$ to $82,29\%$. While the obfuscating transformations cause a bigger degradation of the average detection rate that reaches in the worst case the $50,95\%$. The authors tested the top 10 detection engines on VirusTotal in October and then in November 2011, then in 2012 they tested the local engine obtained from Antiy.

The first limitation of this work is that it does not consider combined code transformations. As a result, it provides an overview of the effects on the detection rate of each proposed code-transformation technique alone but not of their combination. The second limitation is given by the dimension of the dataset of applications considered in the experiments that is smaller that the one used in subsequent works. Moreover, this work makes extensive use of the AV engines through the popular VirusTotal platform, which is known to the research community for being inaccurate. Indeed, the VirusTotal authors themselves recommend not to use their tool to perform comparative studies. The reason is twofold. First, VirusTotal uses the command-line versions of the desktop AV engines, which are not always aligned. Secondly, when applied to mobile AV engines, VT does not use the AV applications, but only their signatures. This does not ensure that the experiment reflects the same detection rate that would be obtained by using the actual AV apps.

DroidChameleon provides an implementation of different obfuscating transformations and classifies them in three main groups with respect to their degree of complexity: trivial obfuscations, obfuscations which results can be detected by tools that use data- and control-flow analysis, and obfuscations which results cannot be detected even when using such analysis. Rastogi et al. have considered 8 of the most popular AV applications (as of February 2013) and a dataset of 6 malware samples from well-known families dated back to 2011 (to ensure that the signatures of the non-obfuscated samples are already present in the AV engines). The authors observe that combinations of at most two transformations are sufficient to make the detection fail. However, the lack of implementation details or source code make this approach very hard to reproduce on a more significant dataset and therefore it is difficult to prove the reliability of this interesting results and the scalability of the approach. Interestingly, the DroidChameleon was the first approach proposing to leverage the outcome of the tests to obtain details about the kind of signatures used by the AV engines. This idea, which is not new as it has been proposed by Christodorescu and Jha [9] in 2004, was never applied to mobile AVs before.

Recent Work. Maiorca et al. [20] evaluated the effectiveness of existing malware detectors for Android to code obfuscation. The authors considered the labeled Android Malware Genome Project dataset [30], which contains 1,260 samples, and a portion of the community-driven Contagio Minidump dataset [3], comprising 237 samples. In order to obfuscate the considered samples the proposed approach resorts to DexGuard [4], a commercial, closed-source code-obfuscation tool widely used for software protection. Out of the box, DexGuard performs trivial obfuscations such as the renaming of packages, methods, classes, fields and source files, as well as more complex transformations such as reflection, and string and class encryption. We note that the authors use a naif encryption algorithm (a simple XOR), whereas we consider DES, which is a more realistic choice for encryption-based obfuscation. However, it does not perform common control code obfuscations such as junk code insertion, code reordering and opaque predicates insertion, which we consider in our work. Whenever DexGuard failed to produce a working executable (which questions the reproducibility of the experimental protocol), the authors resorted to manually implementing the specific obfuscation strategies that caused errors. Unfortunately, no details nor source code is available to document the modifications they performed to obtain a working, obfuscated application. As a side effect, our analysis in Table 1 may be imprecise. What is clear is that Maiorca et al. [20] do not consider control-flow obfuscations. The authors then report the average detection rate of the different detection tools with respect to the obfuscating transformations used, considering multiple transformations as suggested by Rastogi et al. [24]. The results of this recent work prove that attackers might attain a good evasion rate, with minimum size increment, by employing trivial obfuscations and string encryption. However, the reproducibility of the proposed approach is limited by the lack of implementation details and the user of commercial, closed-source tools.

4 Principles and Approach Overview

Having surveyed the state of the art, our goal is to propose a comprehensive, unified, generic, practical, and correct methodology to test the effectiveness of Android AV applications, in the hope that researchers and AV developers can use it as a reference. We do not leverage any proprietary or closed-source tool or service. Last, to the best of our knowledge, we are the first to publish the details of the implementation of each obfuscation operator for Android that we have developed, along with the source code made available to the research community.

At a high level, our approach takes in input a set of malware samples, obfuscates it and tests whether each AV application detects the obfuscated version. This is the general approach to testing anti-malware products. In addition, our approach finds the shortest sequence of obfuscation operators (applied in a predefined order), that an attacker can create to evade an AV product. Indeed, we apply the implemented obfuscation operators in a predefined order until the detection rate of a product drops. Each code obfuscation performs only one transformation and works only on the intermediate representation in `.smali`. Details on the implementation of the obfuscating transformations are provided in Section 5.

In the remainder of this section we describe the principles at the root of our methodology and the testing workflow.

4.1 Comprehensiveness

As summarized in Table 1, we cover the most comprehensive set of obfuscating techniques for Android applications proposed by far, together with all the feasible combinations, thus obtaining the largest set of code-obfuscation techniques in the literature. In addition, we are the first to propose and implement the insertion of opaque predicates for obfuscating Android applications.

4.2 Correctness

Our approach ensures the correctness of the obfuscation operations with respect to the original malware sample. We consider an obfuscation operation as “successful” only if it produces a working application, which means that the evasion technique can be used by a malware developer. In other words, we ensure that the *semantics* of the original application is preserved as much as possible and that no syntax errors are introduced. In the Android context, this also means that the naming scheme and the directory structure must be valid after obfuscation, including XML and other resource files. Some of these aspects are automatically tested by build chain tools, others are tested by the Android Dalvik VM bytecode verifier [2] when the application is installed onto the device. Clearly, these static analyzers do not ensure that the behavior of the application is preserved. To this end, we select a subset of malware samples (from the dataset described in Section 6) and a set of benign applications for which we have extensively, manually tested that their core behaviors are not affected by our transformations.

4.3 Reproducibility and Fidelity

Key to any (anti-malware evaluation) experiment is its reproducibility. To this end, we do not rely on any closed source or proprietary tool, and release the technical details and source code of our prototype. Moreover, we ensure that our testing results are as close as possible to real conditions, that is, a malware author that uses obfuscation to evade a real anti-malware application installed on a user’s device. To meet this requirement, we run the unmodified versions of each anti-malware product that we test. Previous work have leveraged platforms such as VirusTotal, which consist of a black-box interface that the analyst can use to test whether a certain vendor has a signature for detecting a given binary. However, there is no precise information on the internals of such tools. In other words, we do not know how they match the signatures and how the target sample is treated. Several mobile anti-malware applications must leverage specific functionality in order to obtain a correct detection. For instance, some malware samples are detected upon installation, others are detected statically by inspecting the file. Therefore, to ensure fidelity, detection must be performed as if the anti-malware applications were running on a device. Our implementation is independent from the device, and thus can run perfectly on both physical and virtual devices. Technically, we rely on AndroTotal, a research tool available to the scientific community to perform AV tests, as further described in Section 6.

4.4 Flexibility and Scalability

To be future proof with respect to potentially novel obfuscation techniques that may be used by malware authors, we keep our code base as lightweight as possible, making it easy to implement new obfuscations on top of AAMO, and plug them in the framework. Moreover, AAMO allows to combine the operators in any possible order (although we advise for our proposed order to minimize the amount of obfuscation operators applied). Scalability is essential when dealing with potentially large datasets of malware samples. To this end, instead of relying on physical devices, we rely on virtual devices. Note that emulator evasion is not a problem in this case, because the analysis techniques applied by the Android AV engines (as well as the obfuscation operators) are fully static.

4.5 Insightfulness

In line with previous work, we believe that comparative results are useful only if insightful. The detection rate alone is not sufficient for the AV developer to understand what made the detection engine fail. Therefore, we apply the obfuscation operators in an order that not only minimizes the chances of breaking the target application, but that minimizes the length and simplicity of the chain of operators to apply. This, in turns, minimizes the runtime and, most importantly, provides a precise answer to the AV developer, about which specific obfuscation operator caused the AV to fail. In particular, from the experiments that we have conducted we are able to identify the weaknesses of each AV that we have considered in terms of the class of obfuscations that make the AV fail. This allows us to provide insights in how to improve the considered AV tool (see Section 6 for details).

4.6 Workflow

Following our design principles, our methodology can be summarized in the workflow depicted in Figure 2:

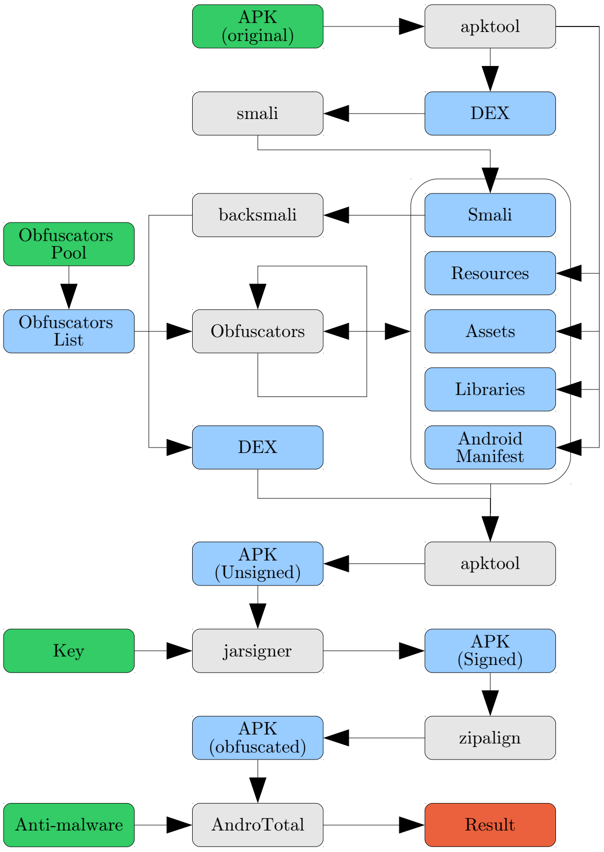


Figure 2 High-level workflow overview of AAMO.

(1) Disassembling: The first phase is to disassemble the sample file. The output of this phase is an intermediate representation of the assembler-level code. In practice, this is implemented via `apktool` (or equivalent utility), which extracts resources, assets, libraries, manifest and the DEX bytecode. The latter is then converted into its Smali representation.

(2) Obfuscating: A list of obfuscators is executed sequentially, and the output of each obfuscator is piped into the next one. Each obfuscator performs only one transformation and works only on the intermediate representation. The sample is obfuscated using the next obfuscators class retrieved from the set defined in Table 1, following this order:

1. Android specific obfuscations (Section 5.1)
2. Simple control-flow obfuscations (Section 5.2)
3. Advanced control flow obfuscations (Section 5.3)
4. Renaming obfuscations (Section 5.4)
5. Resource renaming obfuscations (Section 5.4.1)
6. Resource encryption obfuscations (Section 5.5.1)

(3) Correctness Tests and Reassembling: The resulting obfuscated sample in its intermediate form is built back into a valid Android application package using `apktool` (in “build” mode), which is then signed as normally.

(4) Testing: The obfuscated APK file obtained from the previous phases is scanned with the set of AVs. This phase is part of the workflow, but is implemented outside AAMO for simplicity and flexibility. In our prototype implementation, as detailed in Section 6, we rely on the AndroTotal research platform.

The next section focuses on the set of obfuscating transformations that we designed and implemented, which form the core of AAMO.

5 Obfuscation Operators Implementation

In this section we provide details on the obfuscating techniques that we implemented. For ease of presentation, we distinguish between five classes of obfuscators, as summarized in Table 1 (in comparison with the systematized state of the art).

5.1 Android Specific

This class of obfuscators focuses on simple, Android-specific code transformations that can be used to modify a file without altering its semantics. In other words, the resulting file will have a different hash while retaining the same exact functionality of the original file.

5.1.1 Repackaging

Since Android applications are packaged as compressed archive files, and since AVs rely on their digest as a first, nave signature, a simple obfuscation operator consists in un-compressing the APK files, adding junk resources (e.g., “empty” or unused files), and re-compressing the resulting content in a new archive. This procedure creates a functionally identical APK with a different hash.

Malware authors take repackaging techniques one step further, by embedding their malicious payload into various benign host applications. This has the effect of creating unique malware samples and, at the same time, luring the victim with benign-looking “container” applications.

5.1.2 Reassembly

Similarly to repackaging, disassembling and assembling the DEX code (i.e., `classes.dex`) contained in an APK archive has the result of creating a functionally equivalent application yet with a different digest. This is due to the artifacts left by the disassembling process itself, which does not produce the best optimized assembly code, or by the representation language used (Smali, in our case).

5.1.3 Re-aligning

Aligning an APK file means optimizing its structure so that the resulting archive can be mapped in memory efficiently (since Android uses memory-mapped files to access and load APK resources). More precisely, the convention is to align APK files to 4-byte boundaries.

As a side effect, aligning an archive produces a slightly different file, yet with the same functionality as the original one.

5.2 Simple Control-flow Modifications

This class of obfuscators modify the control flow by adding new code. The goal is to evade signatures that are based on simple fingerprints such as code size, digest, symbols table content.

5.2.1 Junk Code Insertion

This classic obfuscation operator inserts no-operation instructions (also known as "NOP"), opcode `0x00` in Dalvik, which does nothing for one machine clock cycle. The goal is to alter basic code signatures such as size or digest, as well as n-gram-based ones.

Our implementation inserts a random amount of NOPs to each method (excluding, for example, abstract methods, which contain no instructions by definition).

5.2.2 Debug Symbols Stripping

Debug symbols are sometimes found in production releases, mostly by left enabled by mistake. Debugging statements such as `.line`, `.source` or `.parameter` are used to ease debugging and help the developer by obtaining more verbose stack traces. This obfuscation operator removes any debug directives found in the Smali representation of the DEX bytecode.

5.2.3 Defunct Code Insertion

As part of the detection signatures, some products rely on the names of the methods and classes declared in the DEX file. The goal of this obfuscation operator is to evade such signatures by altering method and class names.

To this end, we randomly generate method or class names. Once we find a non-existing one, we insert it right after the `#direct methods` annotation. We generate the inserted code so as not to alter any part of the existing symbols and code.

```
.method public static FogLow(Ljava/lang/String;
    L java/lang/String;)V
  .registers 2
  .prologue
  invoke-static {p0, p1},
    Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
  return-void
.end method
```

Listing 1 Defunct method example.

```
.class public final L<ClassName>;
.super Ljava/lang/Object;
.source "<ClassName>.java"

# direct methods
.method public static FogLow(Ljava/lang/String;L ↙
      ↳ java/lang/String;)V
  .registers 2
  .prologue
  invoke-static {p0, p1}, ↙
        ↳ Landroid/util/Log;->d(Ljava/lang/String; ↙
        ↳ Ljava/lang/String;)I
  return-void
.end method

.method public static <MethodName >()V v0, "<String1>"
  .locals 2
  const-string
  const-string
  .prologue
  invoke-static {v0, v1}, L<ClassName>;->FogLow(v1, ↙
        ↳ "<String2>" Ljava/lang/String;Ljava/lang/String;)V
  return-void
.end method
```

Listing 2 Defunct class example.

5.2.4 Unconditional Jump Insertion

To further alter the control-flow structure, we insert forward and backward unconditional jumps, so as not to alter the code semantics. In the JVM or Dalvik VM, unconditional jumps are implemented with a go-to instruction (which has no higher-level equivalent in Java). We rely on the `goto/32` instruction, which gives the widest address range. More precisely, we add a first `goto/32` instruction at the very beginning of the body of each, right after the `.prologue` annotation. This first instruction jumps at the end of the method's body, after the return instruction. Right after this location we insert another `goto/32` that jumps back at the beginning of the method's body, such that to skip the first jump. The methods affected by this operator are randomly selected.

```
.method public FogLow(Ljava/lang/String;Ljava/lang/String;)V
  # ...preamble...

  .prologue
  goto/32 :GoToFogEnd
  :GoToFogBeg

  # ...body...

  return-void
  :GoToFogEnd
  goto/32 :GoToFogBeg
.end method
```

Listing 3 Unconditional jump insertion example.

5.3 Advanced Control-flow Modifications

This class of obfuscators alters the control flow significantly by adding new branches or modifying existing ones, and by leveraging reflection. Note that we are the first to port opaque predicates to the DEX architecture.

5.3.1 Call Indirection

This obfuscator aims to evade signatures based on the application's call graph. In practice, we redirect each method call to proxy methods that call the original method. These proxies share the same prototype of the original method, including parameters order and type, return type, invocation type, and registers. Return values, if any, are returned by the proxy methods. Each proxy method is a public static method, added right after the `#direct methods` annotation, with a unique randomized identifier. To increase randomness, we detour multiple calls to the same method to distinct proxies. We apply this obfuscator to framework-library and intra-application calls (excluding constructors and static initializers).

```
.class public final L<ClassName>;
  .super Ljava/lang/Object;
  .source "<ClassName>.java"

  # direct methods
  .method public static <Identifier>(Ljava/lang/String;L ↙
        ↳ java/lang/String;)I
    .registers 3
    .prologue

    invoke-static {p0, p1}, ↙
          ↳ Landroid/util/Log;->d(Ljava/lang/String; ↙
          ↳ Ljava/lang/String;)I
    move-result v0
    return v0
  .end method

  .method public static FogLow(Ljava/lang/String;L ↙
        ↳ java/lang/String;)V
    .registers 2
    .prologue

    invoke-static {p0, p1}, ↙
          ↳ L<ClassName>;-><Identifier>(Ljava/lang/String; ↙
          ↳ Ljava/lang/String;)I
    return-void
.end method
```

Listing 4 Call indirection.

5.3.2 Code Reordering

This obfuscator changes the static order of some instructions without changing the runtime execution flow of the original program. The goal is to evade signatures based on the order of instructions or DEX opcodes (e.g., n-grams). In each method body, we group instructions into uniquely labeled basic blocks. We then shuffle the resulting basic blocks and insert unconditional jumps between them to preserve the original execution sequence.

5.3.3 Reflection

We enhance the power of the code-reordering operator by implementing reflection-based obfuscation (which we also apply alone). Essentially, reflection means executing code which location in memory is determined at runtime (e.g., C function pointers, Java reflect API, Android dynamic code loading API). This obfuscation operator replaces static method calls to reflection calls, by loading the target method's name into a string on which the `java.lang.reflect.Method.invoked()` is then called.

```
.class public final L<ClassName>;
  .super Ljava/lang/Object;
  .source "<ClassName >.java"

  # direct methods
  const v0, 0x2
  new-array v1, v0, [Ljava/lang/Class;
  new-array v2, v0, [Ljava/lang/Object;

  const v0, 0x0
  const-class v3, Ljava/lang/String;
  aput-object v3, v1, v0
  aput-object p0, v2, v0
  const v0, 0x1
  const-class v3, Ljava/lang/String;
  aput-object v3, v1, v0
  aput-object p1, v2, v0

  const-string v0, "FogLow"
  const-class v3, L<ClassName>;

  invoke-virtual {v3, v0, v1}, ↙
        ↳ Ljava/lang/Class;->getMethod(Ljava/lang/String; ↙
        ↳ [Ljava/lang/Class;)L java/lang/reflect/Method;
  move-result-object v0

  invoke-virtual {v0, v3, v2}, ↙
        ↳ Ljava/lang/reflect/Method;->invoke( ↙
        ↳ Ljava/lang/Object; [Ljava/lang/Object;)
  move-result-object v0

  return-void

.end method

.method public static FogLow(Ljava/lang/String; ↙
      ↳ Ljava/lang/String;)V
  .registers 2
  .prologue

  invoke-static {p0, p1}, ↙
        ↳ Landroid/util/Log;->d(Ljava/lang/String; ↙
        ↳ Ljava/lang/String;)I
  return-void
.end method

.method public static <MethodName>()V
  .locals 2
  const-string v0, "<String1>"
  const-string v1, "<String2>"

  .prologue
  invoke-static {v0, v1}, ↙
        ↳ L<ClassName>;-><RndMethodName>(Ljava/lang/String; ↙
        ↳ Ljava/lang/String;)V
  return-void
.end method
```

Listing 5 Reflection example.

5.3.4 Opaque Predicate Insertion

Opaque predicates are conditional expressions whose constant value is known by the obfuscator while it is difficult for a compiler or static analyzer to deduce. Opaque predicates insertion aims at confusing static analysis tools that, not being aware of the constant value of the inserted opaque predicate, erroneously see both branches as possible (even if one is never executed at run time). For example, the developer could insert a condition on a function's return value. In practice, we implement this operator by setting two constant integer values in the first two local registries `v0` and `v1` of the method. We set these values randomly (above zero). Then, we append simple arithmetic instructions (e.g., `add-int`, `rem-int`) on these values, such that the outcome is always greater than zero. Using the `if-gtz` instruction (if greater than zero) we instruct the machine to execute a chain of unconditional jumps (implemented with the `goto/32` instruction), ending up to the function return instruction. The "else" branch is never executed. We apply this obfuscation only to methods with two or more local parameters.

5.4 Renaming

This class of obfuscators aims at evading signatures based on the presence of specific strings. This applies to non-code files (e.g., resources, assets) and code files (e.g., fields, classes, or methods names).

5.4.1 Non-code Files and Resource Renaming

Although we place this sub-class of obfuscators under the "renaming" umbrella, we discovered that it is effective enough even if employed alone. Therefore, in our experiments we treat it separately from the other renaming obfuscators.

This obfuscators parses the resource names from the XML files extracted from the APK, and replaces user-defined resource identifiers with the first eight characters of the MD5 of the identifier string, while updating references accordingly (including filename-based identifiers, for which it renames the respective files). In order to evade signatures based on the entire resource table, this obfusction operator adds a random number of useless resource IDs (generated by following the platform constraints).

If the developer of the targeted APK relies on the unique integer ID through a subclass of the `R` class to access resources—as suggested by the Android developers guide—no further change is required. However, if resources are referenced, inside the executable code, by their identifier, inconsistency issues arises. In these cases, we inject also a call to the Android framework method that is used to obtain a resource unique integer ID from its identifier.

5.4.2 Identifier Renaming

This obfuscator replaces each field, method or class name (called "identifier" from hereon) with the first eight characters of the MD5 of the identifier string itself. We rename the references accordingly, by changing the arguments passed to the `(i|s)get-*`, `(i|s)put-*`, and `invoke-*` instructions. In order to evade signatures based on the symbol table, we add to each class a random number of useless copies of the last field, with randomized unique identifiers. We use UTF-8 encoding in order to cope with Unicode identifiers, and pre pend an alphabetical character so as to generate a valid identifier. We propagate the renaming to the XML resource files and Android manifest, which may refer to class names.

5.4.3 Package Renaming

In the Java language, package names are a mechanism for organizing classes into name spaces. Developers typically use package names to organize classes belonging to the same category or providing similar functionality. Classes in the same package can access each other package-restricted members.

Packages are usually defined using a dot-separated hierarchy. In the smali representation, each class is referenced by pre pending the complete path of its package, with the forward slash instead of the dot. For example the `java.lang.String` class is referenced as `Ljava/lang/String` in Smali, reflecting the directory structure.

Each Android application declares, within its manifest file, the application's package name. This global package name serves as a unique identifier for the application (and as the default name for the application process). Certain applications marketplaces enforce that apps should have distinct package names. Given the central role of package names in identifying Android applications, many AVs use the package name string in their signatures.

This operator obfuscates changes the package structure of the application, including the Android package name definition, by renaming package identifiers, as well as the metadata (i.e., labels of application, activities, services, providers, receivers, intent filters, permissions, and actions) declared in the manifest file. In other words, this obfuscator performs a complete refactoring of the application code's name spaces. More precisely, it replaces the package name identifier with the first eight characters of the MD5 digest of the original identifier, taking care of propagating the change to any referring code.

5.5 Encryption

AV signatures are heavily based on sequences of bytes. Therefore, encrypting files or code in an application changes its bytes entirely. Encryption can thus potentially evade all the aforementioned signatures.

All the obfuscation operators described in this section follow a simple schema: They encrypt the target object with a symmetric yet non trivial encryption algorithm (in our case, DES), and then overrides the Android framework methods required to access the encrypted target object. The overridden methods begin with a proper decryption routine that is invoked every time an object is accessed by the application.

The decryption occurs always at runtime and so this obfuscation evades detection signatures that relies on resources content in order to build their signatures.

5.5.1 Resource Encryption (asset files)

File and path names are renamed to the MD5 of their string values and the raw objects are encrypted. Since the `AssetManager` class cannot be overridden, we locate each call to the `AssetManager.open` framework method and redirect them to a proper proxy method that we inject, which takes care of the decryption.

5.5.2 Native Code Encryption

Similarly to the previous obfuscator, we rename native code files and encrypt them. Also in this case we inject a proper proxy method to which we forward the calls originally directed to `System.load` (this way, we also intercept calls originating from its wrapper `loadLibrary`).

5.5.3 Data Encryption (strings)

This obfuscator provides string resource value encryption. The value of each string resource, plural string resource and string array resource, is replaced with its DES encrypted value.

6 Experimental Validation

In this section we describe how we have validated our unified methodology on a real-world use case, that is for testing the resilience of some of the major existing anti-malware products for Android against the comprehensive obfuscation operators that we implemented. In the remainder of this section we describe and justify our choice of dataset, present the obtained results, and provide a thorough discussion.

6.1 Dataset

Despite the availability of sample data sources such as VirusTotal, AndroTotal and Contagio Minidump, the Android Malware Genome Project [30] is the only curated dataset produced by the academic community. This dataset contains 1,260 well-organized malware samples that, comprising 49 malware families, cover the majority of existing Android malware families with heterogeneous characteristics and malicious behaviors. The dataset was collected between August 2010 and October 2011. Therefore, we are positive that AV vendors had enough time to update their signatures. All samples are proved to be malware by both automated and manual analysis.

Since our focus is not of comparing a large number of AV engines, but rather to show how methodology can be applied in practice, we selected the top 6 free AV applications available on the Google Play Store, as summarized in Table 6.4. We updated the malware signatures before the execution of our tests, on the same day, 01/27/2014.

6.2 Test Automation

To ensure reproducibility and fidelity of the results, (main characteristics of our methodology), we leverage AndroTotal, a research platform to automate Android anti-malware scans, in which tests are performed by simulating a real user's behavior when using an anti-malware app. AndroTotal [19] supports various AV applications and multiple Android platform versions. By providing an APK as input, AndroTotal produces the labels of the detected malware variant (if any), and the scanning time. Using AndroTotal allowed us to focus our efforts on the overall methodology, while being confident that scanning results are obtained with a rigorous and robust scientific method, already vetted by the research community.

We have performed our experiments on a specific target version of the Android Jelly Bean v4.1.x API level 16, which is among the top five most used versions.

6.3 Resource Requirements and Speed

For the obfuscation phase we used modest resources (a 4-cores machine with 8GB of memory). Our AAMO prototype took about 2 seconds to run the Android specific and the simple control-flow obfuscators, about 50 seconds when adding the advanced control-flow operators, and approximately 100 seconds with resource renaming and encryption activated. To run the full chain

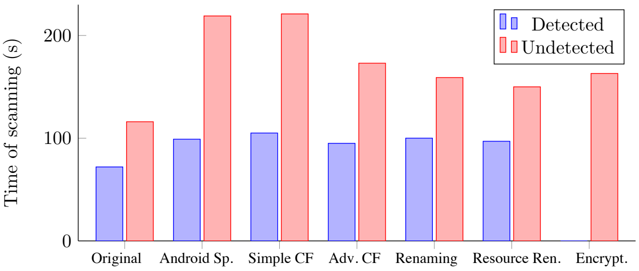


Figure 3 Average detection time by the outcome of the scan.

of obfuscators, AAMO never took more than 200 seconds.

We allocated slightly more resources to AndroTotal (an 8-cores machine with 16GB of memory). On average, the time required for scanning an obfuscated sample was 100 to 300 seconds, and increasing with the number of obufscations applied. This is because a non-matched signature is more likely than a matched signature if the sample is obfuscated, as summarized in Figure 3.

6.4 Discussion of Results

In the following we present our experimental results. To show how our methodology is insightful, we discuss how the obfuscation operator that caused the failure of an AV is an indicator of the weaknesses of the signatures behind the AV. For instance, this information could be used by the vendors as regression tests.

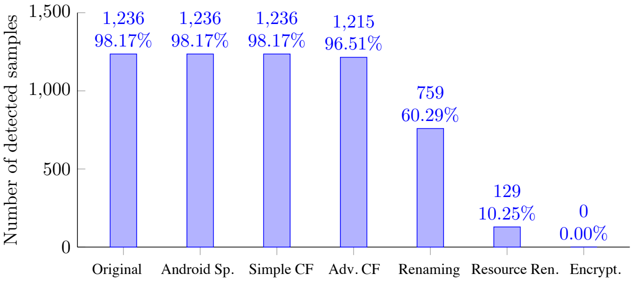
Table 3 shows the detection rate profile of the anti-malware tools that we tested, along the obfuscation chain. In particular, the table reports for each anti-malware tool its average detection rate with respect to the different class of obfuscation considered in the testing methodology. This value is computed as the average of the corresponding detection rates of the anti-malware for each malware family in the data-set. If the average detection rate is 0.00%, we denote it with "✗." For example, the average detection rate of the Dr. Web AV is 96.74% when applied to non-obfuscated malware, 85.78% when Android specific obfuscations are applied, 81.25% when the samples are further obfuscated with basic control flow obfuscations, and drops to 33.53% when advanced control flow obfuscations are applied. Table 4, in the appendix, describes in more details the results of our experiments by showing the profile of the average detection rate of the each anti-malware on the malware families considered in the dataset. When applicable, we re-run the experiment by enabling only the obfuscator that has caused the drop, and add the preceding ones, one at a time, until a comparable drop occurs again. In this way we ensure that the shortest and most effective sequence of operators is found. Observe that during our experiments we distinguish between renaming and resource renaming. This is because we have observed that these obfuscating transformations may influence in different ways the detection rate of an anti-malware tool (e.g., Avast is able to handle renaming but not resource renaming). Moreover, none of the considered anti-malware tools is able to detect a malware sample that employs resource encryption.

Vendor	Package Name	Version	Downloads
Avast	com.avast.android.mobilesecurity	2.0.3917	100k
Norton	com.symantec.mobilesecurity	3.3.4.970	50k
Dr. Web	com.drweb	7.00.3	50k
Kaspersky	com.kms.free	10.4.41	10k
Trend Micro	com.trendmicro.tmmspersonal	3.1	5k
Zoner	com.zoner.android.antivirus	1.8.0	5k

Table 2 AV applications used for our experimental evaluation.

The last line of Table 3 summarizes our results as the average detection rate. In the following we discuss the detection profile of each detection tool, to provide an example of how the AAMO methodology can be applied in practice.

6.4.1 Avast

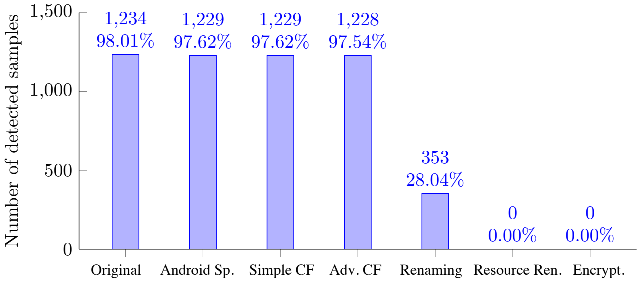


Android specific transformations and control flow obfuscations have no effect on the detection rate of Avast, that after applying these obfuscation is still of 96.51%. We argue that the signature used by this AV uses static analysis techniques in order to handle modifications of the control flow graph. Renaming causes a significant drop in the detection rate that reaches the 60.29%. From this fact, under a black-box assumption, we can conclude that Avast employs some sort of name-based matching that, in our experiments, succeeds in detecting only 60.29% of the samples. On the one hand, since the detection rate drops to 10.25% when *resource* renaming is applied, we conclude that Avast heavily relies on asset names. On the other hand, further experiments showed that when code renaming and encryption are applied together (without resource renaming), Avast reaches 36.22% detection, showing that code name-based features are also given importance.

		Cumulative Obfuscation Transformation					
Vendor	**Original**	**+Android Specific**	**+Simple CF**	**+Advanced CF**	**+Renaming**	**+Resource Ren.**	**+Encryption**
Avast	98.17%	98.17%	98.17%	96.51%	60.29%	10.25%	✗
Norton	98.01%	97.62%	97.62%	97.54%	20.04%	✗	✗
Dr. Web	96.74%	85.78%	81.25%	33.52%	32.49%	33.12%	✗
Kaspersky	97.70%	97.22%	96.82%	92.77%	45.51%	33.60%	✗
Trend Micro	96.98%	73.95%	71.64%	49.96%	49.72%	49.72%	✗
Zoner	98.01%	20.02%	20.02%	20.02%	20.02%	✗	✗
Overall	97.60%	78.79%	77.59%	65.05%	39.34%	21.11%	✗

Table 3 Detection rate of obfuscated malware samples after applying the list of obfuscators described in Section 5. Note that we differentiate between resource renaming and (generic) renaming.

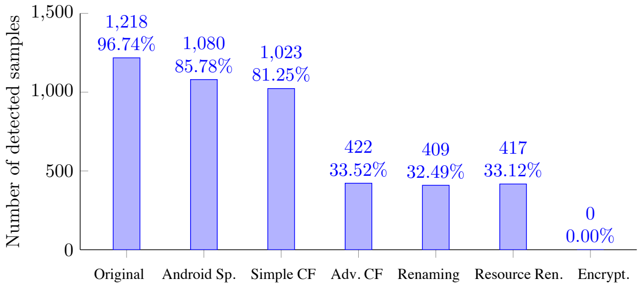
6.4.2 Norton



Android specific transformation and control flow obfuscations (both basic and advanced) have little effect on the detection rate of Norton, that after applying these obfuscation is still of 97.54%. Thus, we argue that the signatures are based on static analysis techniques that make it resilient to modifications of the control flow graph. Renaming obfuscations have an important impact on the detection rate that drops to 20.04%. In our experiments we also observed that the 69.50% of malware samples we have tested can evade detection by applying only code identifiers renaming and string encryption. Moreover, by manually dissecting these samples we have confirmed that the detection is based on code identifiers. The use of symbolic names (e.g., variable names) as a detection criterion, as opposed to the actual names, would reduce these false negatives.

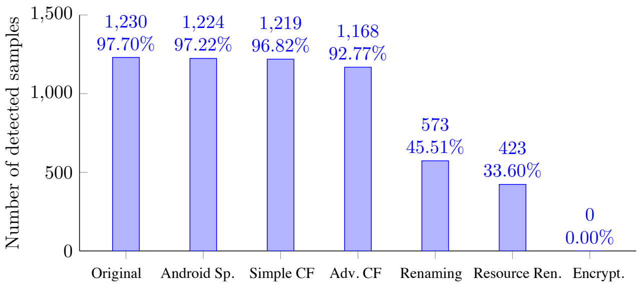
The contribution of renaming is generally substantial, but *resource* renaming obfuscation alone is not sufficient. For example, we applied only resource renaming and found out, by manual inspection, that 28.04% of the samples were still detected because of their package name.

6.4.3 Dr. Web



The application of Android specific transformations has a significant impact (from 96.74% to 85.78%). Indeed, 10.96% of samples in our dataset can evade detection by applying Android specific transformations. This suggests that Dr. Web relies on syntactic features. Basic control flow obfuscations have little effect (-4.53%), while advanced control flow obfuscations cause a 47.73% detection rate drop. So we argue that Dr. Web relies on some static analysis of the bytecode, which makes it resilient to basic control-flow obfuscation. However, more sophisticated analysis techniques should be employed by this product in order to identify opaque predicates and similar artifacts introduced by our advanced obfuscator.

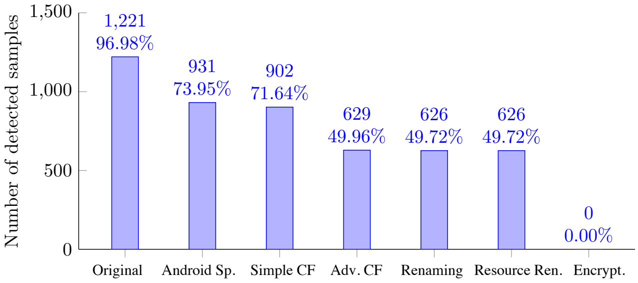
6.4.4 Kaspersky



Android specific transformation and control flow obfuscations (both basic and advanced) have little effect on the detection rate, which is still 92.77% after obfuscation. Instead, renaming obfuscations cause a significant drop in the detection rate (from 92.77% to

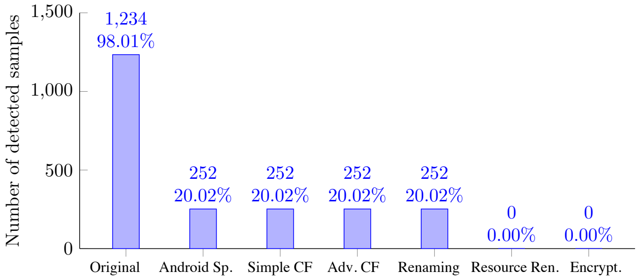
45.51%). Then, when applying resource renaming, we reach 33.60%, which becomes zero when encryption is used. By analyzing the contribution of each specific obfuscator, we found out that 47.26% of the samples can evade detection by applying only code-identifier renaming and string encryption.

6.4.5 Trend Micro



Android specific transformations have significant impact (from 96.98% to 73.95%). Basic control flow obfuscations have little effect (-2.31%), where advanced control flow obfuscations cause a 21.68% detection rate drop. Thus, we can assert that Trend Micro employs some sort of signatures based on static analysis of the bytecode that makes it resilient to some of the changes of the control flow graph. However, given the success of our advanced control-flow obfuscators, more sophisticated analysis techniques should be employed by this product in order to identify opaque predicates and similar artifacts. Renaming has no effect at all. This could be explained by the fact that the detection engine may leverage signatures on the *content* of assets and resource files.

6.4.6 Zoner



We observe that the average detection rate of Zoner drops from 98.01% to 20.02% when applying Android specific transformations. This suggests that Zoner heavily relies on syntactic features. Indeed, by manual analysis we have discovered that the 77.99% drop is due to naive detection based on a digest of the APK file or on its cryptographic signature. It is clear that this makes Zoner very sensitive to any trivial transformation (e.g., repacking, reassembly, re-aligning). Basic and advanced control flow obfuscations seem to have to effect at all, whereas the detection rate breaks down to zero when the renaming of resources is applied.

7 Conclusions

In the seminal paper by Christodorescu and Jha [9], the authors tested the resilience of desktop anti-malware tools against obfuscation. Their experiments showed that desktop malware detectors were not able to recognize obfuscated malware variants. In a similar vein the results obtained by Zheng et al. [29], Rastogi et al. [24] and Maiorca et al. [20], focus on showing that mobile malware detectors are not able to recognize obfuscated Android malware variants.

Our viewpoint is that, nowadays, these results are not surprising. They are clearly valuable to raise awareness among the vendors of anti-malware products. However, with this work we underline the importance of reproducibility in this research line. Reproducibility is essential to all research areas, and in particular to applied malware analysis.

Our response to the problem is toward ensuring reproducible AV testing. Rather than focusing our work on the results, we propose and *release* the implementation of a unified methodology (that takes into account the outcome of our systematization work) so that researchers and vendors can repeat the experiments autonomously, and keep on improving their prototypes and products. The main goal of this work was to propose and develop a reproducible and flexible framework for the analysis of Android AVs against obfuscation. As such, our focus was on the methodology, rather than on the single obfuscators, on which there is ample literature. Clearly, the class of obfuscating transformations offered by our current implementation of AAMO can be extended. For example, by implementing more sophisticated obfuscations that perform code and more advanced opaque predicates insertion. We have also validate our methodology in practice, to show that it can be used to produce results along the line of previous work, but we make a step further, following an open approach, and providing details and source code of our prototype. Observe that our validation has been done with respect to a specific order of the classes of obfuscations implemented in AAMO. Researches can decide to preform the validation of the AV product of interest considering also different orderings of the obfuscations and see how this affects detection.

References

1. Antiy. URL `http://www.antiy.net`.
2. Dalvik bytecode verifier notes. URL `http://www.netmite.com/android/mydroid/dalvik/docs/verifier.html`.
3. Contagio mobile e mobile malware mini dump. URL `http://contagiominidump.blogspot.com/`.
4. Dexguard. URL `http://www.saikoa.com/dexguard`.
5. Virustotal. URL `https://www.virustotal.com`.
6. Smartphone os market share, q2 2015, 2015. URL `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.
7. Axelle Apvrille and Ruchna Nigam. Obfuscation in android malware, and how to fight back. In *Virus Bulletin*, pages 1–10, 2014.
8. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001. ISBN 3-540-42456-3.
9. M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 34–44, 2004.
10. C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 2009. ISBN 0321549252.
11. C. Collberg, C. D. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. of Conf. Record of the 25st ACM Symp. on Principles of Programming Languages (POPL '98)*, pages 184–196. ACM Press, 1998.
12. Mila Dalla Preda and Roberto Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6): 855–908, 2009.
13. Mila Dalla Preda, Isabella Mastroeni, and Roberto Giacobazzi. A formal framework for property-driven obfuscation strategies. In *Fundamentals of Computation Theory - 19th International Symposium, FCT 2013, Liverpool, UK, August 19-21, 2013. Proceedings*, volume 8070 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2013.
14. F-Secure. H2 2013 threat report. Technical report, 2014.
15. Felix C. Freiling, Mykola Protsenko, and Yan Zhuang. An empirical evaluation of software obfuscation techniques applied to android apks. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part II*, volume 153 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 315–328. Springer, 2014.
16. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.
17. R. Fedler J. Schette and M. Kulicke. On the effectiveness of malware protection on android: an evaluation of android antivirus app. Technical report, 2013.
18. Sudarshan Madenur Sridhara and Mark Stamp. Metamorphic worm that carries its own morphing engine. *J. Comput. Virol.*, 9(2), May 2013.
19. Federico Maggi, Andrea Valdi, and Stefano Zanero. AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, pages 49–54. ACM. ISBN 978-1-4503-2491-5. doi: 10.1145/2516760.2516768. URL `http://doi.acm.org/10.1145/2516760.2516768`.
20. Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.
21. Mangesh Musale, Thomas H. Austin, and Mark Stamp. Hunting for metamorphic javascript malware. *J. Computer Virology and Hacking Techniques*, 11(2):89–102, 2015. doi: 10.1007/s11416-014-0225-8. URL `http://dx.doi.org/10.1007/s11416-014-0225-8`.
22. Federico Pellegatta, Federico Maggi, and Mila Dalla Preda. Aamo: Another android malware obfuscator (source code). `https://github.com/necst/aamo`.
23. Mykola Protsenko and Tilo Müller. PANDORA applies non-deterministic obfuscation randomly to android. In *8th International Conference on Malicious and Unwanted Software: "The Americas", MALWARE 2013, Fajardo, PR, USA, October 22-24, 2013*, pages 59–67. IEEE Computer Society, 2013.

24. Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 329–334. ACM, 2013.
25. Tim Strazzere and Jon Sawyer. Android hacker protection level 0. Defcon 22, Las Vegas, Aug 2014.
26. Symantec Corporation. Internet security threat report: April 2015. 20, 2015.
27. Roman Unuchek and Victor Chebyshev. Mobile malware evolution: 2013. `https://securelist.com/analysis/kaspersky-security-bulletin/58335/mobile-malware-evolution-2013/`, February 2014.
28. Min Zheng, Patrick P. C. Lee, and John C. S. Lui. Adam: An automatic and extensible platform to stress test android anti-virus systems (source code). `http://ansrlab.cse.cuhk.edu.hk/software/adam/`.
29. Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: an automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, volume 7591 of *Lecture Notes in Computer Science*, pages 82–101. Springer, 2012.
30. Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy.* URL `http://www.malgenomeproject.org/`.

Table 4: Detection rates of obfuscated malware samples.

Family	Product	Android Sp.	Simple CF	Adv. CF	Renaming	Resource Ren.	Encrypt.
ADRD (2011-02)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	81.82%	86.36%	✗	✗	✗	✗
	Kaspersky	✓	✓	95.45%	95.45%	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	4.55%	4.55%	4.55%	4.55%	✗	✗
AnserverBot (2011-09)	Avast	✓	✓	99.47%	98.93%	✗	✗
	Norton	✓	✓	✓	88.24%	✗	✗
	Dr. Web	90.37%	88.77%	96.26%	94.65%	97.33%	✗
	Kaspersky	✓	99.47%	✓	98.93%	99.47%	✗
	Trend Micro	99.47%	99.47%	99.47%	99.47%	99.47%	✗
	Zoner	1.60%	1.60%	1.60%	1.60%	✗	✗
Asroot (2011-09)	Avast	✓	✓	✓	✓	✓	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	62.50%	✓	✓	✓	✓	✗
	Kaspersky	87.50%	✓	✓	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	✗	✗	✗	✗	✗	✗
BaseBridge (2011-06)	Avast	99.18%	99.18%	99.18%	97.54%	49.18%	✗
	Norton	99.18%	99.18%	99.18%	36.89%	✗	✗
	Dr. Web	82.79%	7.05%	87.70%	85.25%	90.98%	✗
	Kaspersky	99.18%	99.18%	98.36%	96.72%	95.08%	✗
	Trend Micro	95.08%	95.08%	95.08%	95.08%	95.08%	✗
	Zoner	4.10%	4.10%	4.10%	4.10%	✗	✗
BeanBot (2011-10)	Avast	87.50%	87.50%	87.50%	✗	✗	✗
	Norton	87.50%	87.50%	87.50%	87.50%	✗	✗
	Dr. Web	62.50%	62.50%	✗	✗	✗	✗
	Kaspersky	5.00%	87.50%	5.00%	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
BgServ (2011-03)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	7.78%	88.89%	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	88.89%	88.89%	88.89%	88.89%	✗	✗
CoinPirate (2011-08)	Avast	✓	✓	✓	✗	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✗	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
CruseWin (2011-07)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	50.00%	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
DogWars (2011-08)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
DroidCoupon (2011-09)	Avast	✓	✓	✓	✓	✓	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✓	✓	✓	✗
	Kaspersky	✓	✓	✓	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	✗	✗	✗	✗	✗	✗
DroidDeluxe (2011-09)	Avast	✓	✓	✓	✓	✓	✗
	Norton	✓	✓	✓	✗	✗	✗

Family	Product	Android Sp.	Simple CF	Adv. CF	Renaming	Resource Ren.	Encrypt.
	Dr. Web	✓	✓	✗	✓	✓	✗
	Kaspersky	✓	✓	✓	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	✗	✗	✗	✗	✗	✗
DroidDream (2011-03)	Avast	87.50%	87.50%	87.50%	81.25%	81.25%	✗
	Norton	87.50%	87.50%	87.50%	81.25%	✗	✗
	Dr. Web	81.25%	81.25%	81.25%	5.00%	5.00%	✗
	Kaspersky	87.50%	87.50%	87.50%	87.50%	81.25%	✗
	Trend Micro	81.25%	81.25%	81.25%	81.25%	81.25%	✗
	Zoner	6.25%	6.25%	6.25%	6.25%	✗	✗
DroidDreamLight (2011-05)	Avast	97.83%	97.83%	95.65%	32.61%	✗	✗
	Norton	97.83%	97.83%	97.83%	✗	✗	✗
	Dr. Web	93.48%	89.13%	✗	✗	✗	✗
	Kaspersky	97.83%	97.83%	91.30%	28.26%	✗	✗
	Trend Micro	97.83%	86.96%	2.17%	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
DroidKungFu1 (2011-06)	Avast	✓	✓	97.06%	97.06%	✗	✗
	Norton	97.06%	97.06%	97.06%	2.94%	✗	✗
	Dr. Web	3.53%	82.35%	97.06%	✓	94.12%	✗
	Kaspersky	✓	94.12%	97.06%	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	44.12%	44.12%	44.12%	44.12%	✗	✗
DroidKungFu2 (2011-07)	Avast	✓	✓	✓	✓	✓	✗
	Norton	✓	✓	✓	43.33%	✗	✗
	Dr. Web	90.00%	86.67%	✓	✓	96.67%	✗
	Kaspersky	✓	✓	✓	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	33.33%	33.33%	33.33%	33.33%	✗	✗
DroidKungFu3 (2011-08)	Avast	99.35%	99.35%	99.03%	4.85%	✗	✗
	Norton	99.35%	99.35%	99.03%	✗	✗	✗
	Dr. Web	89.64%	83.17%	1.94%	1.94%	1.94%	✗
	Kaspersky	98.38%	97.41%	87.38%	✗	✗	✗
	Trend Micro	65.05%	62.14%	36.57%	35.92%	35.92%	✗
	Zoner	32.36%	32.36%	32.36%	32.36%	✗	✗
DroidKungFu4 (2011-10)	Avast	93.75%	93.75%	93.75%	91.67%	✗	✗
	Norton	93.75%	93.75%	93.75%	✗	✗	✗
	Dr. Web	82.29%	67.71%	4.17%	4.17%	4.17%	✗
	Kaspersky	93.75%	92.71%	88.54%	✗	✗	✗
	Trend Micro	93.75%	93.75%	93.75%	93.75%	93.75%	✗
	Zoner	18.75%	18.75%	18.75%	18.75%	✗	✗
DroidKungFuSapp (2011-10)	Avast	✓	✓	✓	✗	✗	✗
	Norton	66.67%	66.67%	66.67%	✗	✗	✗
	Dr. Web	✗	✗	✗	✗	✗	✗
	Kaspersky	66.67%	66.67%	66.67%	✗	✗	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	✗	✗	✗	✗	✗	✗
DroidKungFuUpdate (2011-10)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✗	✗	✗	✗	✗	✗
	Dr. Web	✗	✗	✗	✗	✗	✗
	Kaspersky	✗	✗	✗	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
Endofday (2011-05)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
FakeNetflix (2011-10)	Avast	✓	✓	✓	✗	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗

Family	Product	Android Sp.	Simple CF	Adv. CF	Renaming	Resource Ren.	Encrypt.
	Zoner	✗	✗	✗	✗	✗	✗
FakePlayer (2010-08)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	83.33%	✓	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
GamblerSMS (2011-07)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
Geinimi (2010-12)	Avast	91.18%	91.18%	91.18%	66.18%	1.47%	✗
	Norton	91.18%	91.18%	91.18%	47.06%	✗	✗
	Dr. Web	83.82%	6.47%	1.47%	1.47%	1.47%	✗
	Kaspersky	88.24%	91.18%	85.29%	57.35%	1.47%	✗
	Trend Micro	83.82%	3.53%	1.47%	1.47%	1.47%	✗
	Zoner	66.18%	66.18%	66.18%	66.18%	✗	✗
GGTracker (2011-06)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✓	✓	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
GingerMaster (2011-08)	Avast	✓	✓	✓	✓	✓	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	50.00%	✓	✓	✓	✓	✗
	Kaspersky	✓	✓	✓	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	✗	✗	✗	✗	✗	✗
GoldDream (2011-07)	Avast	95.74%	95.74%	59.57%	59.57%	✗	✗
	Norton	95.74%	95.74%	95.74%	✗	✗	✗
	Dr. Web	6.60%	6.60%	✗	✗	✗	✗
	Kaspersky	95.74%	93.62%	89.36%	59.57%	✗	✗
	Trend Micro	80.85%	61.70%	✗	✗	✗	✗
	Zoner	29.79%	29.79%	29.79%	29.79%	✗	✗
Gone60 (2011-09)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	66.67%	7.78%	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
GPSSMSSpy (2010-08)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
HippoSMS (2011-07)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
Jifake (2011-10)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✗	✗	✗	✗	✗	✗
	Kaspersky	✓	✓	✗	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
jSMSHider (2011-06)	Avast	✓	✓	✓	✗	✗	✗
	Norton	✓	✓	✓	✗	✗	✗

Family	**Product**	**Android Sp.**	**Simple CF**	**Adv. CF**	**Renaming**	**Resource Ren.**	**Encrypt.**
	Dr. Web	✓	93.75%	✓	87.50%	93.75%	✗
	Kaspersky	✓	✓	✓	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	✗	✗	✗	✗	✗	✗
KMin (2011-10)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	88.46%	✗	✗
	Dr. Web	84.62%	69.23%	1.92%	1.92%	1.92%	✗
	Kaspersky	✓	✓	✓	1.92%	1.92%	✗
	Trend Micro	✓	✓	1.92%	1.92%	1.92%	✗
	Zoner	✗	✗	✗	✗	✗	✗
LoveTrap (2011-07)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	✓	✗	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
NickyBot (2011-08)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
NickySpy (2011-07)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	✓	50.00%	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
Pjapps (2011-02)	Avast	96.55%	96.55%	96.55%	✗	✗	✗
	Norton	96.55%	96.55%	96.55%	✗	✗	✗
	Dr. Web	9.31%	82.76%	✗	✗	✗	✗
	Kaspersky	91.38%	89.66%	89.66%	✗	1.72%	✗
	Trend Micro	43.10%	44.83%	✗	✗	✗	✗
	Zoner	25.86%	25.86%	25.86%	25.86%	✗	✗
Plankton (2011-06)	Avast	✓	✓	✓	✓	✗	✗
	Norton	2.73%	2.73%	2.73%	✗	✗	✗
	Dr. Web	✓	2.73%	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✓	✓	✓	✓	✗	✗
RogueLemon (2011-10)	Avast	✓	✓	✓	50.00%	✗	✗
	Norton	50.00%	50.00%	50.00%	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
RogueSPPush (2011-08)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	7.78%	55.56%	✗	✗	✗	✗
	Kaspersky	✓	✓	88.89%	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	66.67%	66.67%	66.67%	66.67%	✗	✗
SMSReplicator (2010-11)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✓	✗	✗
	Dr. Web	✗	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✓	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
SndApps (2011-07)	Avast	✓	✓	✓	✗	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗

Family	Product	Android Sp.	Simple CF	Adv. CF	Renaming	Resource Ren.	Encrypt.
	Zoner	✗	✗	✗	✗	✗	✗
Spitmo (2011-09)	Avast	✓	✓	✓	✗	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✗	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
TapSnake (2010-08)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	50.00%	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	50.00%	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
Walkinwat (2011-03)	Avast	✓	✓	✓	✗	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✗	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
YZHC (2011-06)	Avast	✓	✓	✓	45.45%	✗	✗
	Norton	✓	✓	✓	4.55%	✗	✗
	Dr. Web	90.91%	81.82%	✗	✗	✗	✗
	Kaspersky	✓	✓	95.45%	4.55%	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
zHash (2011-03)	Avast	✓	✓	✓	✓	✓	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	81.82%	81.82%	✓	✓	90.91%	✗
	Kaspersky	✓	✓	✓	✓	✓	✗
	Trend Micro	✓	✓	✓	✓	✓	✗
	Zoner	✗	✗	✗	✗	✗	✗
Zitmo (2011-07)	Avast	✓	✓	✓	✓	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	✓	✗	✗	✗	✗
	Kaspersky	✓	✓	✗	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗
Zsone (2011-05)	Avast	✓	✓	✓	66.67%	✗	✗
	Norton	✓	✓	✓	✗	✗	✗
	Dr. Web	✓	83.33%	✗	✗	✗	✗
	Kaspersky	✓	✓	✓	✗	✗	✗
	Trend Micro	✗	✗	✗	✗	✗	✗
	Zoner	✗	✗	✗	✗	✗	✗