Efficient Code Obfuscation for Android

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author: Alexandrina Kovacheva
Supervisor: Prof. Alex Biryukov
Reviewer: Prof. Jean-Sébastien Coron
Advisor: Dr. Ralf-Philipp Weinmann

August 2013
Declaration

I, Alexandrina Kovacheva, declare that this thesis titled, “Efficient Code Obfuscation for Android” and the work presented in it are my own. I confirm that:

■ This work was done wholly while in candidature for a master degree at the University of Luxembourg.

■ Where I have consulted the published work of others, this is always clearly attributed.

■ Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

■ I have acknowledged all main sources of help.

■ Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

______________________________

Date:

______________________________
Acknowledgements

I would like to thank my two supervisors for trusting me to work on this topic without me having prior knowledge on the subject and for guiding me through the way. The last six months have been the most self-growing period of my master studies. I learned a lot and I had fun doing so.

I would also like to thank the brave hearted, adventurous and self-taught musicians in my life. Your music inspires me, it makes my days. Without you my life is a cappella.
Abstract

Recent years have witnessed a steady shift in technology from desktop computers to mobile devices. In the global picture of available platforms, Android stands out as a dominant participant on the market and its popularity continues rising. While beneficial for its users, this growth simultaneously creates a prolific environment for exploitation by vile developers which write malware or reuse software illegally obtained by reverse engineering. A class of programming techniques known as code obfuscation targets prevention of intellectual property theft by parsing an input application through a set of algorithms aiming to make its source code computationally harder and time consuming to recover. This work focuses on the development and application of such algorithms on the bytecode of Android, Dalvik. The main contributions are: (1) a study on samples obtained from the official Android market which shows how feasible it is to reverse a targeted application; (2) a proposed obfuscator implementation whose transformations defeat current popular static analysis tools while maintaining a low level of added time and memory overhead; (3) an attempt to initiate a discussion on what techniques known from the x86 architecture can(not) be applied on Dalvik bytecode and why.
Introduction

Ever since the early 1990s, devices combining telephony and computing have been offered for sale to the general public. In 1997, the term smartphone was introduced for the first time with the release of Ericsson’s GS88 “Penelope” [44]. Although one might deride that smartphones are merely in their sixteens, their rapid development and extensive usage nowadays are indisputable. A report from February 2013 estimated the total number of smartphone devices sold only in 2012 as surpassing 1.75 billion units with a record peak in the last quarter [21]. In addition to making and receiving calls, smartphones allow their users to generate, store and share multimedia by accessing the Internet through various applications. Similar functionalities have tablet computers, another class of mobile devices. Due to their wide ranging applicability and high mobility both smartphones and tablets have been preferred over stationary or laptop computers as access devices to personal information services such as e-mail, social network accounts or e-commerce websites. These services are easily made available to the end user via online mobile application markets. By the end of 2012, the market was dominated with a ratio of 70% by the Android platform [25]. This huge market share as well as the sensitivity of the user data processed by most applications raise an important security question regarding the source code visibility of the developed mobile software. Firstly, developers have an interest of protecting their intellectual property against piracy. Moreover, an alarming 99% of the mobile malware developed in 2012 has been reported to target Android platform users and inspections reveal both qualitative and quantitative growth [20]. In terms of quality, Android malware has evolved from applications sending SMS messages to premium-rate numbers without the user’s authorization to sophisticated code that is able to infect legitimate applications and propagate via Google Play (the official Android market) [7]. Hence, Android application code protection is crucial to maintaining a high level of trust between vendors and users which in turn reflects in a correct functioning of the Google Play market itself.

In general, there are two main approaches towards software protection: enforcing legal software usage policies or applying various forms of technical protection to the code. This work concentrates on the latter, more precisely on a technique called code obfuscation. In the context of information security the term obfuscation encompasses various deliberately made modifications on the control-flow and data-flow of programs such that they become computationally hard to reverse engineer by a third party. The applied changes should be semantic preserving with ideally negligible or minor memory-time penalty. Prior to elaborating on how to apply obfuscation on Android software, an introduction to the platform fundamentals is necessary.
## 1.1 Android architecture overview

Android is an open source Linux-based operating system running on a large set of touchscreen devices. Launched in 2007 by Google, it is designed to meet the limited computational capacity of a mobile device’s hardware. The principal processor of Android devices is the ARM platform for which the operating system is optimized. Following is an overview of the Android architecture with an insight to a limited set of essential components for the scope of this work. A full description is available at the Android Developers website [1].

![Android system architecture overview](image)

The underlying entity of the system is its kernel which bridges the hardware of the device and the remaining software components. Being a Linux-based kernel, it allows remote access to the device via a Linux shell as well as the execution of standard Unix commands.

Going up one level in the system stack abstraction is the Dalvik Virtual Machine (DVM). The DVM is highly tailored to work according to the specifications of the Android platform. It is optimized for a slower CPU in comparison with a stationary machine and works with relatively little RAM memory: 20MB after the high-level system services have started [5]. The DVM is register-based, differing from the standard Java Virtual Machine (JVM) which is stack-based. Such a solution is motivated by the fact that register-based architectures require fewer executed instructions than stack-based architectures. Although register-based code is approximately 25% larger than the stack-based, the increase in the instructions fetching time is negligible: 1.07% extra real machine loads [13]. Moreover, the Android OS has no swap space imposing that the virtual machine works without swap. Finally, mobile devices are powered by a battery thus the DVM is optimized to be as energy preserving as possible. Except being highly efficient, the DVM is also designed to be replicated quickly because each application runs within a “sandbox”: a context containing its own instance of the virtual machine assigned a unique Unix user ID.

At the same abstraction level as the virtual machine are the native libraries of the system. Written in C/C++, they permit low level interaction between the applications and the kernel through Java Native Interface (JNI). Although a limited set has been shown on
1.2. THE ANDROID PACKAGE FILE IN DETAILS

Fig 1.1 the functionalities provided by these libraries expand to cover features such as text rendering, application window management, drawing of 2D and 3D graphics etc. A noteworthy library of this layer is SQLite since mobile applications often store a user’s identifiable information in such a database which, if not protected adequately, might be accessed by a third party for malicious purposes.

The next layer is the application framework which provides generic functionality to mobile software through Android’s application programming interface (API). The following listed represent key structure concepts of Android applications:

**Activity.** The unitary concept which all applications are built upon. From a design perspective, an activity corresponds to a single screen with a user interaction interface. Each activity has standard defined methods for managing its lifecycle which is initiated with the `onCreate()` method. The control between activities is interchanged by an “intent” which can be either direct or indirect depending on whether the application invokes a concrete activity or calls external applications. It is exactly the Activity classes of the application which are usually infected by malicious software and thus must be properly protected.

**Service.** Services are application processes which most often run in background assuming no user interaction is needed to keep them alive. They can also serve as supply components from the current application to external ones. Malicious code can be packed into a legitimate application by exploiting weaknesses of services which are not managed adequately [7].

**Content provider.** Content providers are an interface for managing the access to a structured set of data of the current or external applications. Additionally to encapsulating data, these components define mechanisms for defining data security [16].

**Broadcast receiver.** Broadcast announcements are made upon events which affect the entire system such as an incoming phone call, a screen turn off or wireless availability. A broadcast receiver responds to such an announcement and is often used to trigger the execution of malicious code [7].

The top layer of the Android OS stack is where custom applications are compiled, installed and executed. The file format of the install ready application is called Android Package (APK) and all the mobile software is distributed over Google Play in this format. The APK format is a package management system based on the ZIP file archive format. Further details about the contents of Android applications are provided in the subsequent section.

To show that Android is targeting a wide range of devices, including resource constrained ones, the minimal device hardware requirements [13] are given on table 1.1. Currently, most smartphones and tablets largely exceed the listed.

1.2 The Android package file in details

Familiarizing with the components of Android’s architecture is the primary step towards building safe applications or alternatively reversing them efficiently. Having the former as base knowledge, the natural continuation is being acquaint with the APK file structure as well as an application’s lifecycle.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chipset</td>
<td>ARM-based</td>
</tr>
<tr>
<td>Memory</td>
<td>128 MB RAM; 256 MB Flash External</td>
</tr>
<tr>
<td>Storage</td>
<td>Mini or Micro SD</td>
</tr>
<tr>
<td>Primary Display</td>
<td>QVGA TFT LCD or larger, 16-bit color or better</td>
</tr>
<tr>
<td>Navigation Keys</td>
<td>5−way navigation with 5 application keys, power, camera and volume controls</td>
</tr>
<tr>
<td>Camera</td>
<td>2MP CMOS</td>
</tr>
<tr>
<td>USB</td>
<td>Standard mini-B USB interface</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>1.2 or 2.0</td>
</tr>
</tbody>
</table>

Table 1.1: Minimal hardware requirements to run Android.

1.2.1 APK structure

The contents of an APK archive clearly vary largely by the purpose an application is created for. However, the here presented file structure is one which all Android applications comply with. Directories are denoted in bold font, files have their extensions appended to the names.

**META-INF**

- **CERT.RSA** The certificate of the application. In order to be accepted for installation, an APK file *must* be digitally signed with a certificate whose private key is held by the application’s developer. Since the certificate is not required to be signed by a trusted certificate authority [1], it is typically not done so.
- **CERT.SF** A file listing the application resources and their SHA-1 digest.
- **MANIFEST.MF** The application manifest file.

**res** Contains the raw resources of the application such as images and audio files.

**AndroidManifest.xml** A binary file declaring all the components and permissions required by the application to be executed in the system.

**classes.dex** The container of the classes of the application in the Dalvik Executable bytecode format. This file is of key importance: if not protected, the application’s reversing is straightforward.

**resources.arsc** Contains the pre-compiled application resources.

Although not obligatory, it is common for applications to have a **lib** directory with the pre-compiled native code for a specific processor architecture.

1.2.2 APK build and installation processes

The applications for Android are written using the Java programming language. A standard Java environment compiles each separate class in the `.java` source code file into a corresponding Java bytecode `.class` file. For example: having a single `.java` file containing one public class, one static inner class and two anonymous classes processed by the `javac` compiler will result in the generation of four separate `.class` files. These are later packed together in a single `.jar` archive file. The JVM unpacks the `.class` files, parses and executes their code at runtime.
On the Android platform, the build process differs after the point when the `.class` files have been generated. Once having the latter, they are forwarded to the “dx” tool which is part of the standard Android SDK. This tool compresses all `.class` files into a single `classes.dex` file i.e. the `.dex` file is the sole bytecode container for all the application’s classes. After it has been created, the `classes.dex` is forwarded to the ApkBuilder tool altogether with the application resources and shared object (.so) files which, if present, contain native code. As a result, the APK archive is created and the final compulsory step is its signing. Figure 1.2 shows the APK build process and the possible obfuscation manipulations which are optional during the build stages. The next chapter provides more details on bytecode analysis and protection.

Upon installation, there are two notable steps performed: primary is the APK verification and secondary is the bytecode optimization. For security reasons applications whose legitimate signature as well as correct `classes.dex` structure cannot be verified are rejected for installation by the OS. Once verified, the `.dex` file is forwarded for optimization: a necessary step due to the high diversity of Android running hardware. Thus, Dalvik executable is a generic file format which needs additional processing to achieve best performance for the concrete device architecture. The command to manually invoke the optimizer is `dexopt` which outputs an `.odex` (optimized DEX) pre-processed version of the `classes.dex` file and stores it locally in /data/dalvik-cache. The optimization step removes the `classes.dex` from the original APK archive and loads in memory the `.odex` file upon execution. This step occurs only once, during the initial run of the application which explains the usually slower first application launch comparing to the subsequent ones.

### 1.2.3 DEX file format overview

The `classes.dex` file is a crucial component regarding the application’s code security because a reverse engineering attempt is considered successful when the targeted source code has been recovered from the bytecode analysis. Hence studying the DEX file format together with the Dalvik opcode structure is tightly related to both designing a powerful obfuscation technique or an efficient bytecode analysis tool.

In comparison to the standard Java bytecode, Dalvik bytecode is compact and its space optimization concept is based on data sharing. Memory is saved by assuring minimal
data repetition and applying implicit typing and labeling. Figure 1.3 shows the .dex file structure and compares a .jar archive composed of multiple .class files with an APK containing the same classes packed in a single .dex file. Also, the mappings from the sections of the .class file to the ones in the .dex file are shown. Although not depicted, the remaining .class files are mapped analogically.

![Figure 1.3: Structure and mapping of .class to .dex files.](image)

Each .class file has its own heterogeneous constant pool which may contain duplicating data. For example, multiple methods which return variables of the same type, say `String`, will result in a repeating `Ljava/lang/String;` in each of the method’s signatures. The memory efficiency of a .dex file comes primarily from the type-specific constant pools used to store the data. This means that in the previously given example, the constant `Ljava/lang/String;` will be present only once in the type_ids pool and will be referenced by each method using it. As a consequence, there are significantly more references within a .dex file compared to a .class file. This optimized .dex design ensures data granularity and allows compression as efficiently as up to 44% of the size of an equivalent .jar archive [13].

Regarding the Dalvik bytecode, some general remarks on the instructions format are a necessary prerequisite to the next chapters. As already mentioned, the DVM is register based. Registers are considered 32 bits wide to store values such as integers or floating point numbers. Adjacent register pairs are used to store 64-bit values. There is no alignment requirement for these register pairs [33]. If a method has N arguments, they land in order in the last N registers of the method’s invocation frame [33]. The corresponding instruction mnemonic of the method is formatted in a dest-then-source ordering for its arguments. During the install-time optimization process, some instructions may alter. In total, there are 218 used valid opcodes in Dalvik bytecode [33][34].
1.3 Android security overview

The last section of this chapter gives a brief overview of the OS security mechanisms. By default each application is limited within a sandbox. There are two possibilities for external applications communication: using permissions or the inter process communication (IPC) mechanisms provided by Android.

Permissions grant access to potentially sensitive data such as user personal information including messages or contacts, metrics provided by a phone sensor like GPS or information regarding the phone identity i.e. phone number, IMEI, IMSI. To request any such data an application needs to explicitly declare it with a corresponding permission (e.g. for precise location the permission would be `ACCESS_FINE_LOCATION`). Before an application is installed the user is faced with an ultimatum to either accept the list of its declared permissions, or revoke the installation. Permissions may not be altered after installation, but the application is allowed to query whether a permission has been granted to it. Ideally, applications are designed to comply with the least privilege principle: only requesting permissions needed for their correct functioning. However, a practical survey on apps obtained from Google Play shows that privacy invasion is common practice. In the examined set, a ratio of 30% contained overprivileged applications [14].

Indirect intents are the main mechanism which makes IPC possible in Android. This happens by having one application send an intent to a receiving auxiliary component of the other application such as a broadcast receiver or a content provider. The following figure gives a clarification of the possible internal and external interactions occurring in the system [17]. Green arrows indicate data access requests from the applications to the Android API. Red arrows follow the information IPC and non-IPC flow which might contain sensitive data.

![Android Application Framework](image)

Figure 1.4: Internal and external process communication in Android.

Further Android security analysis as well as work related to application permissions misappropriation can be found in [14, 17, 18, 46].
Dalvik Bytecode Analysis and Protection

Reverse engineering and code protection are processes which are opposing each other, yet none can be classified as neither good nor bad. It is the intentions of the agent performing either action which are biased. From a “good” developer’s viewpoint, code protection is a means towards intellectual property preservation and reverse engineering can be used to detect malware. Flipping the coin, an adversary would use code protection to make their malicious code analyst-resistant and perform reverse engineering to examine potential applications as attack targets.

Either way, to recover the original code of an application bytecode analysis is most often used. By applying both dynamic and static techniques, it is possible to detect an overprivileged application design, find patterns of malicious behavior or trace user data such as login credentials. Dynamic analysis is the process of extracting the desired information during runtime. This method requires simulation of the complete input domain of the examined application to reach high precision in the evaluation of the program behavior or to successfully track the desired data. By contrast, static analysis is executed on raw bytecode. Usually, an automatic tool is run through the targeted code and outputs an approximation of its control flow and data flow. The approximation accuracy depends on the used reverse engineering algorithms by the analysis tool as well as on what forms of technical protection the examined code has underwent. In the best (or worst) case despite the applied protection on the input, the entire source code is completely recovered.

2.1 Bytecode analysis tools

Due to its simplicity over bytecode for other architectures as well as the little protection applied in practice, Dalvik bytecode is currently an easy target for the reverse engineer. The here listed set of analysis tools and decompilers is a representative of the large available variety.

**dexdump** Included as a part of the standard Android SDK, this is the most easily accessible tool to a developer performing Dalvik bytecode disassembling [15]. The implemented analysis algorithm is linear sweep i.e. it traverses the bytecode and expects each next valid instruction to succeed the currently analyzed one. In the case of non-obfuscated code the disassembling will be successful, however a modification on control flow complexity can fail the recovery process.

**dedexer** A disassembler tool for dex files [27]. Outputs the recovered bytecode in a Jasmin-like syntax.
**baksmali** One of the most popular Dalvik bytecode decompilers [32]. Due to the more sophisticated underlying analysis algorithm, recursive traversal, the recovery rate of baksmali is greater than the previously presented tools. The algorithm improvement lies in the fact that the next instruction need not necessarily be immediately following the current one i.e. jumps are successfully processed. However, this approach only minimizes but does not eliminate the effects of some control flow manipulations as will be shown later. Due to its popularity, baksmali is used by multiple reverse engineering tools as a base disassembler, amongst which is the also well-known **apktool**.

**dex2jar** A binary file conversion tool which takes as its input a .dex file and generates its corresponding .jar archive containing the extracted .class files [28]. To view the source code, any Java decompiler such as JAD or JD-GUI can be used.

**radare2** An interactive console tool for both bytecode disassembling and analysis which allows very precise control from the user regarding the decompilation process [31]. For specific bytecode functions, decompilation is done with the integration of the open-source boomerang decompiler. Besides the usage of recursive traversal, the user may specify decompilation starting at a specific address. Because of this hybrid approach, some obfuscation techniques breaking other decompilers are reversible with radare2, however not automatically.

**androguard** An analysis and disassembling tool processing both Dalvik bytecode and optimized bytecode [26]. The tool has three different decompilers: DAD, DED and JAD. The one used by default is DAD which is also the fastest due to the fact it is a native decompiler. Its underlying algorithm is recursive traversal. Also, androguard has a large online open-source database with known malware patterns. Additional features such as measuring efficiency of obfuscators by comparing a program with its obfuscated version, visualizing the application as a graph and permissions examination are available as separate scripts.

**dexter** An online analysis tool [29] processing APK files and displaying a rich set of results amongst which: application’s defined and used permissions; ratio of obfuscated versus non-obfuscated code; ratio of internal versus external packages; broadcast receivers and content providers etc. This tool also allows graph visualization of the application and full list of strings used by the application. Although free to use, dexter has its code closed on the server-side and the only information about the underlying performed algorithms available is that currently it performs solely static analysis.

**dexguard** Introduced in June 2013, a set of scripts currently targeting mainly automated strings deobfuscation and recovery of the .dex file [6]. This tool has a hybrid approach of dynamic and static analysis and is comprised of: (a) .dex file reader, (b) Dalvik disassembler, (c) basic Dalvik emulator, (d) .dex file parser. At the moment of this work’s submission this tool is not publicly available. Also, for the future the developers plan to keep its code server-side closed.

**IDA Pro** A widely used commercial tool [12] for reverse engineering under multiple supported architectures. IDA Pro has multiple features such as program graph visualization and support of plug-ins which extend its standard functionality.
2.2. BYTECODE PROTECTION TOOLS

Evidently, there are numerous tools to the help of the reverse engineer which can be used either separately or to complement each other. The same diversity cannot be claimed for software regarding the code protection side which is presented in the following section.

2.2 Bytecode protection tools

Referring back to figure 1.2, two optional steps where obfuscation may be applied are available: (a) at source code and (b) bytecode level. Most existing open-source and commercial tools work on source code level. The reason is that effective protection techniques successfully applied on Java source code have been suggested in previous works [11]. Furthermore, Java code is architecture-independent giving freedom to design generic code transformations. Lowering the obfuscation level to bytecode requires the algorithms applied to be tuned accordingly to the underlying architecture. Researched techniques exist for x86, some of which can be mapped to the Android platform. The here listed tools are concentrated on bytecode modifications with the exception of ProGuard which is a Java obfuscator part of the Android SDK. The remaining examples introduce a set of obfuscation techniques, some of which resisted the majority of the formerly introduced reverse engineering tools at the time they were announced. However, certain analysis tools have updated their algorithms to circumnavigate these techniques. Details on the exact obfuscation algorithms implemented by open-source tools are given in the next section.

ProGuard A Java source code obfuscator [30]. ProGuard performs variable identifiers name scrambling for packages, classes, methods and fields. It shrinks the code size by automatically removing unused classes, detects and highlights dead code, but leaves the developer to remove it manually.

dalvik-obfuscator An open-source bytecode obfuscation tool [33]. Given a standard APK file as input, it outputs its corresponding obfuscated APK version. The underlying algorithm is the well known under the x86 architecture junk byte injection.

APKfuscator Another open-source bytecode obfuscation tool [41] which applies multiple variations of dead code injection.

DexGuard A commercial Android obfuscator [37] working both on bytecode and source code level (should not be mistaken with dexguard analysis tool). Performs various techniques including strings encryption, encrypting app resources, tamper detection, removing logging code.

The here described open-source bytecode obfuscation tools have the status of a proof-of-concept software rather than being used at regular practice by application developers. To show the ease with which source code can be retrieved from Android mobile software, a case study on applications including both legitimate and malware apps was performed and the results are presented in the upcoming chapter.

2.2.1 Dalvik bytecode obfuscation techniques

Obfuscation should prevent from extracting metadata about the program both on an abstract and concrete level: it should be computationally hard to determine the control flow or recover correct mnemonics from a bytecode sample.
A general requirement to all transformations is that given a program \( P \), the following two must hold for its obfuscated version \( O(P) \) [11]:

1. (functionality) The observable behaviour between \( P \) and \( O(P) \) should be identical i.e. they should produce the same result. The term “observable behaviour” concerns the program as experienced by the user. It is allowed that \( O(P) \) has side effects which \( P \) does not originally have as long as they are not perceived by the user.

2. (polynomial slowdown) The program size and running time of \( O(P) \) are at most polynomially larger than those of \( P \).

The following techniques are sorted in ascending order according to the computational difficulty for their reverse engineering. Whenever a technique is used by an obfuscation tool, this is explicitly noted with accompanying details on the concrete implementation.

**Identities name scrambling.** This technique affects the layout of the program and can be implemented both on source code and bytecode level. Its purpose is to obfuscate the program on an abstract level by replacing the meaningful names of variables, methods, classes, files with ones which provide no metadata information regarding the code. Identities name scrambling is implemented both in ProGuard and in APKfuscator with some major differences. ProGuard works on Java source code and uses replacement with minimal lexical-sorted strings \( \{a, b, c, \ldots, aa, ab, \ldots\} \) to have little space penalty cost which is essential on mobile devices [24]. APKfuscator works on bytecode level and exploits the Unix filesystem restriction that a class name should not exceed 255 characters [12]. This exploit is possible also on Dalvik bytecode due to the class definition item structure used in the .dex file format [34]. As shown on figure 2.2.1, one may replace the classname with a larger one stored in theubyte[] data type constant. A .dex format requirement is to have all strings sorted alphabetically without the occurrence of repeating string names [34]. Furthermore, any misplace of the entries in the .dex header tables requires a corresponding relevant offset change in all references pointing to that particular table entry. To avoid such a risky manipulation, APKfuscator implements name scrambling by simply appending data to the class name without modifying its position in the constant pool table.

**Encoding manipulations.** This transformation regards both the file layout and the data structures of the program. By specification, the byte ordering in the .dex format is little-endian. The ARM Architecture Reference Manual [2] states that ARM processors support mixed-endian access in hardware, meaning that they can operate in either little-endian or big-endian modes. Hence, the DVM verifier is supposed to be able to detect the encoding of the interpreted .dex file and convert big-endian to little-endian and vice versa. While changing the encoding is not hard to implement, it has been suggested as potentially efficient since the majority of the Dalvik bytecode analysis tools work only with little-endian encoded files [12].
Strings obfuscation. This technique is a well known data transformation applied often on source code level. Although it is not implemented by any of the examined open-source obfuscators, it is possible to adjust it to the level of Dalvik bytecode. String obfuscation prevents from metadata information extraction and is efficient against static analysis. Since many applications process personal data, it is rather common to store strings such as user credentials in a database. However, the consequence of keeping the latter in plaintext is making them an easy target for the reverse engineer. There is a significant difference between obfuscating the strings of a program and scrambling the variable names: changing the latter does not affect the semantics of the program. By contrast, strings need to be on one hand encrypted to prevent static extraction and on the other hand, they need to be available as plaintext during runtime such that a process like user verification is performed successfully. Depending on whether obfuscation is applied on source code or bytecode level the effort needed to obtain the plaintext string varies. What can be done on source code level is passing the string $s$ as an argument to an invertible transformation function $F$: it is $F(s)$ which is stored in the code. Whenever the plaintext string is needed during runtime, the program returns $F^{-1}(F(s)) = s$. Hence, performing string obfuscation requires the implementation of a custom encryption/decryption algorithm or preferably, the usage of a standardized algorithm. On Android, with this approach the encrypted strings will be stored in the string_ids constant pool, i.e. the cyphertext would be visible to the reverser and obtaining the plaintext relies on the hardness of breaking the encryption algorithm. As a remark to the latter, previous work reveals usages of deprecated algorithms \[18\] as well as implementations of custom XOR ciphers \[46\] which clearly are poor security practices. While theoretically possible, it is not feasible to perform obfuscation by storing encrypted strings in the constant pool on bytecode level. Having the entire string_ids table shuffled and later reassembled such that: (a) the ordering of the content is alphanumeric; (b) does not contain repeating entries and (c) fixing all table reference offsets across the bytecode is worth a huge programming effort simultaneously being highly error prone. An alternative improved approach is converting each string first into a byte array, encrypting the bytes and storing the encrypted bytes instead of the encrypted string. This makes it significantly harder for a third party to obtain the plaintext since the encrypted bytes will no longer appear in the string_ids constant pool forcing the reverse engineer to manually scan the bytecode to discover the encrypted string.

Dead code injection variants. Dead code injection is another transformation which is borrowed from x86. It affects the control flow of the application and is implemented on bytecode level by both dalvik-obfuscator and APKfuscator, each of the tools using its own variation of the technique. In essence, this algorithm modifies the control flow by inserting code which will never be executed, yet adds nodes and edges to the program graph which respectively increases the complexity. To guarantee that the execution will not go through the introduced bogus paths, a conditional branch is used for redirection. Thus, it is necessary that this condition is especially chosen as producing an a priori known to the programmer result, but which is computationally hard to estimate at runtime, i.e. it is either always true (directing to “good” paths) or it is always false (never directing to “bad” paths). Such conditional constructs are called opaque predicates and they have been used, among others, in Java source code obfuscation \[11\]. At bytecode level, the implemented in the two obfuscators dead code injection variants are using legitimately defined in the documentation but somewhat special instructions.
In Dalvik-obfuscator the dead code injection transformation cracked tools using both linear sweep and recursive traversal disassembling algorithms at the time of its submission [40]. To inject the code the variable length instruction fill-array-data-payload is used. Before the entry point of the method-to-be-obfuscated, two instructions are added: the fill-array-data-payload which overlaps the method’s code and a preceding opaque predicate which redirects the execution to the valid method contents. The figure gives an intuitive idea of the difference between (a) non-obfuscated and (b) obfuscated code using this technique [40].

Both linear sweep and recursive traversal algorithms fail to recover the correct bytecode sequence because of the preceding opaque predicate. Linear sweep cannot handle any “jumping” control flow manipulation. Recursive traversal will discover the presence of the fill-array-data-payload instruction because of the condition, but will consider it a legitimate branching leaving untouched the overlapped instructions. The result is displaying the method internals as a sequence of bytes instead of source code.

In APKfuscator three different variations of dead code injection are implemented [42]:
(a) inserting illegal opcodes in dead code; (b) using legitimately defined opcodes into “bad” objects; (c) injection of code in the .dex header by exploiting a discrepancy between the claims of the official .dex file format documentation and what the Dex Verifier does in reality.

(a) Since the injected code will contain illegal opcodes, a consideration using this technique must be made with regards to the Dex Verifier. To implement this variant successfully, the illegal opcodes should be injected into classes which are not used in the application i.e. the dead code itself contains the illegal opcodes. If bad opcodes were used in meaningful classes, the application would crash not being able to execute them. Furthermore, the dead code should not be removed by the optimizer, otherwise the transformation is meaningless.

(b) This injection variant exploits the fact that there exist multiple legitimate, but unused Dalvik opcodes e.g. 0xFC, 0xFD, 0xFE, 0xFF [33]. Let us have the following injected bytecode sequence:

```
1201  // load 0 in v1
3801 0300  // if v1 == 0 (always true), jump ahead
1A00 FF00  // load const-string at index 0xFF (not existing)
```

The verification of the upper sequence is successful since all opcodes are legitimate, but due to the fact that the opcode 0xFF does not correspond to any valid address, some disassembling tools fail recovering the entire application, others fail processing only the obfuscated file [42].

(c) The third injection variant performed by APKfuscator is based on the tool’s author observation that there is an inconsistency between the official .dex file format specification and what the Dex Verifier actually does. For the header_item it is claimed in the documentation that the header size has a fixed length of header_size = 0x70 [34]. Since Android is an open source platform, it is possible to review the code and observe the following for the Dex Verifier:
On line 3, a check is performed to see if the length of the header is less than 0x70 and if it is, an error is raised. On line 7, if the header size exceeds 0x70 a warning is raised, but the file is accepted as valid and execution continues. This mismatch is used as a precondition to increase deliberately the size of the header (no problem with file verification) and inject additional code in the header item after its last component data_off. Injection in the header requires fixing the alignment of all the succeeding sections and tables in the .dex file as well as each item linked to the modified tables. Such implemented, this injection causes the analysis tools to process the .dex file as a valid one, but to extract the code from the header manual intervention might be needed. Although a proper example of exploiting inaccuracy gaps between documentation and source code, this modification is trivial to detect: if the header size exceeds 0x70 the “red alert” is on.

**Executable compression.** A technique known under the x86 architecture which is often used by malware to hide its code. The aim of this method is constructing a single executable which contains the program’s compressed code packed with a decompressor stub. Compression, frequently combined with code encryption, is used both to decrease the size of the executable as well as to obfuscate the code. During runtime the decompressor stub firstly extracts the compressed code and then executes the original program. Reversing a program which has underwent such a transformation cannot be done with static analysis. The two principal methods to handle it are either manual examination of the decompression stub and then unpacking the program or by performing dynamic analysis.

In 2011, an Android spyware called Plankton was reported to be the first malware which exploits Dalvik class loading capability to stay stealthy and dynamically extend its own functionality [19]. In comparison to the upper described, this malware starts a service running in background upon the application launch. The service sends collected user data of the infected device to a remote server and receives back a URL to download a .jar file containing executable bytecode. Once downloaded, the executable is started through the standard DexClassLoader system class and its init() method is invoked using reflection.

**Self modifying code.** Self modifying code is a known code transformation applied successfully on the x86 architecture whose purpose is to hinder dynamic analysis. Used often by malware in combination with buffer overflow attacks, it has also found its application in obfuscation techniques for legitimate software. Having a program protected against static analysis results in a more complex yet identical upon every execution control flow. By contrast, dynamic code changes have an effect at runtime altering the execution path.
upon each program invocation.

The applicability of executable compression, self modifying code as well as other known dynamic obfuscation algorithms on Android bytecode is discussed in the final chapter of this work. It is not uncommon that an obfuscation technique needs to be designed with a balance between the added program complexity and the robustness of the modified code against analysis. Regarding this, dynamic obfuscation techniques increase resilience considerably, but it can be a challenge to apply them uniformly on an input APK file which is why a chapter is dedicated to that topic.

The next chapter presents a case study whose purpose is to justify the claim that current analysis tools are powerful enough to analyze free applications retrieved from Google Play. Also, we show that a very small proportion of the examined files are deliberately preprocessed to resist analysis.
A Case Study on Applications

There exist an extensive set of works examining applications from the viewpoint of privacy invasion, as was cited in the Introduction chapter. The current case study aims to show that bytecode undergoes few protection. If present, obfuscation is very limited with regards to the potential transformation techniques which could be applied, even for apps which were found to protect their code. The study was performed in two stages. Initially, automated static analysis scripts were run on bytecode for a coarse classification the purpose of which was profiling the apps according to a set of chosen criteria. A secondary, fine grinding examination, was to manually select a few “interesting” apps and looking through the code at hand. All applications studied were available through the official Google Play market as of March 2013.

3.1 Applications collecting

To be able to obtain applications from Google Play, a user must be registered and have their account associated with at least one Android running mobile device. Installation can be invoked either directly from the device, or by requesting an application from the website after which the installation process starts as soon as the mobile device goes online. It is exactly the second feature that was used to collect the applications. A web crawler was developed requesting the 50 most popular applications from each of the 34 categories available on the market and “catching” them before they are downloaded to the device. The downloaded apps set was initially 1700, however, there were applications in repeating categories making it a total of 1691 examined files. The download was executed on a machine with a running NOD32 antivirus software and 94 of the files raised a malware alert. Hence, although not primarily planned for the analysis, the entire set was divided into 1597 safe and 94 malware-alert apps with the latter subset undergoing additional processing.

3.2 Applications study

Disassembly of all the .dex files was performed with DAD, the default disassembler in the androguard analysis tool. The motivation behind this choice is that of all the previously presented freely available tools androguard had the largest successful disassembly ratio. Selecting DAD was due to the fact it is a native disassembler recovering each class on-the-fly and as such is faster than other disassemblers [26]. The lines of bytecode analyzed numbers approximately to 338,200,000 thus disassembly time efficiency was a crucial issue. Moreover, of the three available decompilers in androguard, DAD performed best in terms of reversing the bytecode with only 7 applications defeat-
ing it (left to be analyzed entirely manually) while the other two decompilers hindered significantly.

The here enumerated criteria were used for apps profiling:

1. **Obfuscated versus non-obfuscated classes.** A study on the usage of Pro-Guard which is the officially available in the Android SDK code obfuscator was an easy target. Since this tool applies variable name scrambling in a known pattern, the classes names and contained methods were processed with a pattern matching filter according to the naming convention i.e. looking for minimal lexical-sorted strings. A class whose name is not obfuscated, but contains obfuscated methods was counted as an obfuscated class.

2. **Strings encoded with Base64.** Several of the malware-alert applications were found to contain “hidden” from the resources files in the form of strings encoded with Base64. Manual examination of a limited number of these revealed nothing but .gif and flash multimedia files. However, this finding suggests that it might be common practice that binary data is hidden as a string instead of being stored as a separate file in the /res/ directory. It is also technically possible that code can be hidden for example with an encoded .so file. Thus, filtering the application string pool for Base64 encoding entries was considered relevant for the study.

3. **Dynamic loading.** Dynamic loading allows invocation of external code not installed as an official part of the application. It has been discovered as a technique applied in practice by applications executing malicious code [19]. For the initial automation phase its presence was only detected by pattern matching check of the classes for the packages:

   - Ldalvik/system/DexClassLoader
   - Ljava/security/ClassLoader
   - Ljava/security/SecureClassLoader
   - Ljava/net/URLClassLoader

4. **Native code.** Filter the class definition table for the usage of code accessing system-related information and resources or interfacing with the runtime environment. For the coarse run only detecting the presence of native code in the following packages was considered:

   - Ljava/lang/System
   - Ljava/lang/Runtime

5. **Reflection.** The classes definition table was filtered for the presence of the Java reflection packages for access to methods, fields and classes.

6. **Header size.** Referring to the bytecode injection possibility in the .dex header by exploiting the discrepancy between the format documentation and the file verification in reality, the header size was also checked.

7. **Encoding.** A simple flag check in the binary file for whether an application uses the support of mixed endianess of the ARM processor.

8. **Crypto code.** The Android SDK javax.crypto and java.security.spec packages provide various classes and interfaces for applications implementing algorithms for encryption, decryption, or key agreement. With regards to previous studies on inappropriate user private data handling as well as deliberate cryptography misuse, the classes were also initially filtered for the usage of the packages:
3.3. AUTOMATION RESULTS

All the 1691 applications were profiled according to the formerly listed criteria. For the malware-alert raising set of 94 apps, the initial automation also included the following:

9. **Permissions.** Although not directly related to the usage obfuscation, permissions review helps narrowing down the target data used by the application.

10. **Auxiliary.** To facilitate the second phase of the study which also included manual examination, information on the services, receivers, providers and main activity class of the application was gathered.

Once having been processed according to the former listed criteria, the malware-alert files were studied for similarity with over 200 available malware samples. Since file comparison is a time-costly operation, to improve efficiency the malware samples themselves were classified into clusters by comparing them with each other. This “clusterification” reduced the initial set to 153 malware files which in turn had a noticeably positive time-performance impact. To summarize, in total the malware-alert apps were processed in three stages: (a) general profiling; (b) coarse comparison to determine the belonging cluster; (c) fine comparison with each application in the cluster. For all similarity tests the `androsim.py` tool part of `androguard` was used. Merely giving a similarity score based on static analysis with known malware is not sufficient to classify an application as malicious, but because the primary topic of this work is not related to malware detection and analysis, no further processing was conducted. All 94 files were sent as report to Google with according accompanying information. As a result, 24 applications listed in the appendix were removed from the market.

3.3 Automation results

The distribution of applications according to the percentage of obfuscated code with ProGuard is shown on table 3.1. On table 3.2 are noted the absolute number of occurrences of each factor the apps were profiled for. The extended studies on the malware-alert files are shown on table 3.3. An observation to be made is that all malicious applications make use of reflection. This, however, is not a sign of malicious behavior. It simply indicates that these applications load classes in a non-standard manner. A typical example scenario of legitimate usage of reflection is having a database engine loaded from the firstly-found database driver. In a malicious context reflection could be used to load custom code from the application resources.

The automated study reveals that encoding strings in base64 is quite common practice: 840 applications containing a total of 2379 strings were found and examined, shown on table 3.4. To determine the file format from the decoded strings the `python magic` library\footnote{https://github.com/ahupp/python-magic} was used. Unfortunately, 1156 files which is 48.59\% of the total encoded files could not be identified by this approach and using the Unix `file` command lead to no better results. The remaining set of files was divided into multimedia, text and others. Some files might be archived data/code which is denoted as ERROR in the table. This supposition is based on the fact that the output error message was “unpack requires a string argument of length \(n\)” which could be a password (\(n\) was originally represented by an integer). As a final remark to table 3.4 is that the percentage marks the occurrences in the 1241 successfully identified files.
<table>
<thead>
<tr>
<th>OBF</th>
<th>100%</th>
<th>(100 − 80]</th>
<th>(80 − 60]</th>
<th>(60 − 40]</th>
<th>(40 − 20]</th>
<th>(20 − 0]</th>
<th>0%</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>82</td>
<td>291</td>
<td>196</td>
<td>166</td>
<td>283</td>
<td>423</td>
<td>250</td>
<td>1691</td>
</tr>
<tr>
<td>%</td>
<td>4.85</td>
<td>17.21</td>
<td>11.59</td>
<td>9.82</td>
<td>16.74</td>
<td>25.01</td>
<td>14.78</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 3.1: Obfuscation ratio. The row with # marks the absolute number of applications with obfuscated number of classes in the given range. The row with % marks the percentage this number represents in the set of the total applications.

<table>
<thead>
<tr>
<th>OBF</th>
<th>B64</th>
<th>NAT</th>
<th>DYN</th>
<th>REF</th>
<th>CRY</th>
<th>HEAD</th>
<th>LIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>41.839</td>
<td>840</td>
<td>629</td>
<td>224</td>
<td>1519</td>
<td>1236</td>
<td>1691</td>
</tr>
<tr>
<td>%</td>
<td>46.74</td>
<td>49.68</td>
<td>37.20</td>
<td>13.25</td>
<td>89.83</td>
<td>73.09</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 3.2: Profiling the set of applications according to the given criteria: OBF (total obfuscated classes), B64 (number of apps containing base64 strings), NAT (number of apps with native code), DYN (number of apps with dynamic code), REF (number of apps with reflection), CRY (number of apps with crypto code), HEAD (number of apps with header size of 0x70), LIT (number of apps with little endian byte ordering). The row with # marks the absolute numbers of occurrences, % marks the percentage this number represents in the set of the total applications.

<table>
<thead>
<tr>
<th>OBF</th>
<th>B64</th>
<th>NAT</th>
<th>DYN</th>
<th>REF</th>
<th>CRY</th>
<th>HEAD</th>
<th>LIT</th>
<th>REC</th>
<th>SER</th>
<th>PRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>1433</td>
<td>67</td>
<td>13</td>
<td>30</td>
<td>94</td>
<td>48</td>
<td>94</td>
<td>94</td>
<td>79</td>
<td>89</td>
</tr>
<tr>
<td>%</td>
<td>38.10</td>
<td>71.28</td>
<td>13.83</td>
<td>31.91</td>
<td>100</td>
<td>31.91</td>
<td>100</td>
<td>100</td>
<td>84.04</td>
<td>94.68</td>
</tr>
</tbody>
</table>

Table 3.3: Profiling the set of malicious applications according to the given criteria. The annotations are analogical to the ones on Table 3.2 with the addition of: REC (total number of applications having receivers), SER (total number of applications having services), PRO (total number of applications having providers).

<table>
<thead>
<tr>
<th>type</th>
<th>#</th>
<th>%</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII text</td>
<td>56</td>
<td>4.51</td>
<td>TXT</td>
</tr>
<tr>
<td>ERROR</td>
<td>3</td>
<td>0.24</td>
<td>OTH</td>
</tr>
<tr>
<td>GIF</td>
<td>48</td>
<td>3.87</td>
<td>MUL</td>
</tr>
<tr>
<td>HTML</td>
<td>3</td>
<td>0.24</td>
<td>OTH</td>
</tr>
<tr>
<td>ISO-8859 text</td>
<td>1</td>
<td>0.08</td>
<td>TXT</td>
</tr>
<tr>
<td>JPEG</td>
<td>33</td>
<td>2.66</td>
<td>MUL</td>
</tr>
<tr>
<td>Non-ISO extended-ASCII text</td>
<td>24</td>
<td>1.93</td>
<td>TXT</td>
</tr>
<tr>
<td>PNG</td>
<td>522</td>
<td>42.06</td>
<td>MUL</td>
</tr>
<tr>
<td>TrueType font text</td>
<td>548</td>
<td>44.17</td>
<td>MUL</td>
</tr>
<tr>
<td>UTF-8 Unicode text</td>
<td>1</td>
<td>0.08</td>
<td>TXT</td>
</tr>
<tr>
<td>XML document</td>
<td>2</td>
<td>0.16</td>
<td>OTH</td>
</tr>
</tbody>
</table>

Table 3.4: Classification of the base64 encoded strings. Categories are denoted as follows: TXT for text, MUL for multimedia, OTH for other.
3.4 Manual review

A set of several applications was selected for manual review, the selection criteria trying to encompass a wide range of possible scenarios. Among the files were: (1) the most highly obfuscated (89.7%) malware-alert application; (2) a highly popular social application with no obfuscation and a large number of packages; (3) a popular mobile Internet browser with 100% obfuscated packages; (4) an application which androguard (DAD) and dexter failed to process; (5) an application which is known to use strings encryption and is claimed to be obfuscated as well; (6) an application containing many base64 encoded strings; (7-10) four other applications both legitimate and malware-alert chosen at random. Additionally, the permissions usage of all malware-alert files was reviewed and analyzed.

With the exception of application (4) all files were successfully processed by androguard. The source code of all checked obfuscated methods was successfully recovered to a correct Java code with the androguard plugin for Sublime Text. The control-flow graphs of all analyzed files was recovered successfully with androgexf.py. However, in some applications the excessive number of packages created an inappropriate setting for adequate analysis thus the graphs were filtered by pattern-matching the labels of their nodes. Having the graphs of all applications simplified revealed practices such as implementation of custom strings encryption-decryption pair functions and having their source code implementation hidden in a native library (seen in two of the analyzed files). Reviewing the graph of application (4) was a key towards understanding why some tools break during analysis: they simply do not handle cases of Unicode method or field names (e.g. 文章:Ljava/util/ArrayList;)

A summary of interesting strings which some apps referenced to is given below:

http://media.admob.com/- mobile ads website, found in 2 of the reviewed files;
tel://6509313940 - the phone number of Admob Inc., found in 2 of the reviewed files, these apps also made use of the Landroid/telephony/TelephonyManager and Landroid/telephony/gsm/GsmCellLocation classes;
http://dl.dropbox.com/u/30899852/mraid/inmobi_mraid.js
http://dl.dropbox.com/u/30899852/mraid/inmobi_mraid_bridge.js - two publicly shared JavaScript files via Dropbox containing functionality for making calls, sending mails, and SMS messages. There was an application which had in its strings “...try connect to Loco”, most probably a services server related to the app, but curiously it also stored “locoforever” in plaintext. Yes, the password. Regarding the permissions used in the malware-alert applications, it is no surprise that 100% of the apps required the android.permission.INTERNET together with the android.permission.ACCESS_NETWORK_STATE. About 63% of the apps required location information with android.permission.ACCESS_COARSE_LOCATION and android.permission.ACCESS_FINE_LOCATION, some applications not having any functionality related to location services such as changing the phone’s wallpaper. In fact, some at-first-sight wallpaper applications had as much as 27 permissions including install priviledges, writing to the phone’s external storage, read and write in the browser bookmark history and others. These results only come as confirmation to what previous studies have already established as user privacy invasive practices [18].
3.5 Conclusions and remarks

The main conclusion of both automated and manual inspection is that even in cases where some tools hindered recovering the bytecode mnemonics or source code, there is a way round to obtain relevant information. Where a given tool is not useful, another can be used as complement. Reversing large applications may be slowed down due to the complexity of the program graph, but with appropriate node filtering a reasonable subgraph can be obtained for analysis. To prevent information extraction by static analysis some applications made use of Java reflection or embedding valuable code in a native library. Apart from using ProGuard to rename components and decrease program understandability, no other code obfuscation was found. Using Unicode names for classes and methods could be regarded as an analogical type of obfuscation to ProGuard: it affects merely program layout not the control flow.

Finally, a number of considerations need to be taken into account when reviewing the results of the performed study. (1) Only freely available applications were processed: the results will highly likely differ if identical examinations were performed on payed applications. (2) The set of popular applications in the Google Play market differs with the country of origin of the requesting IP address: the download for this study was executed on a machine located in Bulgaria. (3) To verify the correctness of the obfuscated versus non-obfuscated code ratio a comparison with the dexter analysis tool which also computes this proportion was done. Whenever obfuscation was found present, the here presented obfuscation percentage is slightly higher than the one outputted by dexter. The reason for this deviation is that the current study examines only internal packages while the dexter tool also considers external libraries which increases the overall number of counted packages. Furthermore, the current study was done on an obfuscation-per-class basis, while dexter uses the unit per-package. Results where no code obfuscation was present were identical. (4) The mobile malware samples for Android were downloaded from a freely available malware download source\footnote{http://contagiominidump.blogspot.co.il/} where they numbered 242 unique files for the Android platform as of March 2013.
Implementing a Dalvik Bytecode Obfuscator

The results in the previous chapter confirmed that little protection on Android applications is used in practice. This chapter describes a possible implementation of a Dalvik bytecode obfuscator including four transformations whose main implementation accents fall on fulfilling the generic and cheap properties.

In the context of this work the term “generic” denotes that the transformations are constructed in aspiration to encompass a large set of applications without preliminary assumptions which must hold for the processed file. On Android this can be a real challenge since an application has to run on a wide range of devices, OS versions and architectures. It can happen that applications which are not obfuscated at all have limited device support either because the developers intentionally decided so, or due to a limitation such as lack of testing devices hardware. Thus, it is crucial that any applied code protection would not decrease the set of application running devices. When a transformation is characterized as “cheap” this is in referral to previously published work by Collberg et. al. on classifying obfuscating transformations [10]. By definition, a technique is cheap if the obfuscated program $P'$ requires $O(n)$ more resources than executing the original $P$. Resources encompass processing time and memory usage: two essential performance considerations, especially for mobile devices.

Following is a description of the general structure of the Dalvik bytecode obfuscator as well as details on the four transformations applied.

4.1 Structure overview

The approach used by the here presented obfuscator is identical to the one used in dalvik-obfuscator [38]. The input is an APK file which can be either processed by ProGuard i.e. with renamed classes and methods, or not modified at all. Auxiliary tools used during the obfuscation are the pair smali assembly and baksmali disassembly. The application is initially disassembled with baksmali which results in having a directory of .smali files. The corresponding hierarchical file structure is as follows: one sub-directory per package with exactly one .smali file corresponding to each class. Internal classes are marked with a $ sign in the file name. These files contain mnemonics retrieved from the immediate bytecode interpretation. Three of the transformations parse, modify the mnemonics and assemble them back to a valid .dex file using smali. One transformation modifies the bytecode of the .dex file directly. After the modifications have been applied, the .dex file is packed together with the resource files, signed and is verified for data integrity. This last step yields a semantically equivalent obfuscated version of the APK file. Figure 4.1 summarizes the entire obfuscation process.

\footnote{https://github.com/alex-ko/innocent}
Adopting this workflow has the advantage of accelerating the development process by stepping on a .dex file assembler and disassembler pair. However, a disadvantage is that the implemented obfuscator is bound by the limitations of the used external tools. As will be shown in the next section this approach has its constraints regarding the range of the transformations’ applicability.

4.2 Bytecode transformations

The here suggested tool can apply four techniques designed such that all of them affect both the data and the control flow. The transformations targets are calls to native libraries, strings normally visible in the constant pool, 4-bit and 16-bit numeric constants used by the applications. Native calls are redirected through external classes in methods that we would call here “wrappers”. Strings are encrypted and numeric constants are packed in external class-containers, shuffled and modified. In other words the transformations aim to harden meta-information recovery by complimenting program data hiding with hardening control flow through additional external classes. The fourth modification injects dead code which has a minor effect on the control flow, but makes the input APK resistant to reverse engineering with current versions of some popular tools which is why we call it here “bad” bytecode injection.

Let us denote the four transformations as follows: adding native call wrappers with ‘w’, packing the numeric variables with ‘p’, obfuscating the strings with ‘o’ and adding bad code with ‘b’. Since the bytecode is modified after executing either of the transformations, a consideration about the order in which they should be applied is necessary. The simple automaton on the right accepts words representing the order of applying the transformations. The 5-tuple \((Q, \Sigma, \delta, q_0, F)\) is defined as:

\[
Q = \{St, Ob, Ba\}, \quad \Sigma = \{w, p, o, b\}, \quad \delta = \{(St, St, Ob, Ba), (Ob, Ob, 0, Ba), (0, 0, 0, 0)\},
\]

\[
q_0 = \{St\}, \quad F = \{St, Ob, Ba\}. \quad \text{The states are denoted as} \text{St for the starting state, Ob}\text{obfuscated strings state, Ba bad code added state. Adding native call wrappers and re-packing numeric constants can happen before or after encrypting strings as well as multiple times, each additional processing decreasing performance. Regarding the injected code, in this implementation our tool uses external (dis)assembly which breaks by the injected bytes sequence thus no further transformation is possible. In general, one can further process the file with a custom assembly resistant to the “traps” in the code.} 
\]
4.2. BYTECODE TRANSFORMATIONS

4.2.1 Adding native call wrappers

Native libraries are mostly used for self-contained, CPU-intensive operations which do not allocate much memory, such as signal processing or physics simulation. The majority of the files with native library calls collected from the case study are games and communication related apps. While native code itself is not visible through applying static analysis, calls to native libraries cannot be shortened by tools such as ProGuard. The reason is that method names in Dalvik bytecode must correspond exactly to the ones declared in the external library for them to be located and executed. One way to decrease understandability is to scramble the names of the native C/C++ functions in advance and to call the scrambled names. This was not seen anywhere in practice. Hence meta information about the functionality implemented by the native libraries can be extracted easily.

The proposed transformation here does not address the issue with comprehensive method names since this depends on the developer. However, another source of useful information is the locality of the native calls i.e. by tracking which classes call particular methods relevant conclusions can be made. Thus, to harden the usage tracking process one could place the native call in a supplementary function, what is referred here as a native call wrapper. The exact sequence of steps taken is on the following schematic figure:

The application is primarily scanned for the location of native calls by pattern matching the mnemonics in the method declarations. Let us have a class containing native calls which are highlighted in colors on (a). For each unique native method a corresponding wrapper with additional arguments is constructed redirecting the native call. To complicate the control flow, the wrappers are scattered randomly in external classes from those located originally. As a final step each native call is replaced with an invocation of its respective wrapper as shown in (b).

The overall impact of this transformation on the program graph can be seen as a transition from what is depicted in (c) to the final result in (d). Initially, the locality of the native method calls give a hint on what the containing class is doing. For example during the manual application review it was trivial to locate a class containing calls to a custom encryption implemented in a native library (Lcom/.../util/SimpleEncryption; encryptString(Ljava/lang/String; I) Ljava/lang/String;) i.e. knowing exactly which class to track accelerates reversing the custom encryption algorithm.

By contrast, after applying the here suggested transformation once, the reversing time and effort is increased by locating the wrapper, reviewing its code and concluding that there is no logical connection between the class containing the wrapper and the native invocation. If the transformation is applied more than once, the entire nesting of wrappers has to be resolved. Usually, a mobile application would have hundreds of classes to scatter the nested wrapping structures: a setting that definitely slows down the reversing process.
4.2.2 Packing numeric variables

The idea behind this transformation stems from what is known in previous works as opaque data structures [9]. The basic concept is to affect data flow in the program by encapsulating heterogeneous data types in a custom defined structure. The access to the actual variables is protected with an opaque predicate. During runtime the variables can be retrieved only if the opaque condition is fulfilled or the program has reached a specific state where the predicate evaluates to a desired value.

The target data of this particular implementation are the numeric constants in the application. Analogically to the previous transformation, the bytecode mnemonics are primarily scanned to locate the usages and values of all 4-bit and 16-bit constants. After gathering the latter, the obfuscator packs them in a 16-bit array (the 4-bit constants being shifted) in a newly-created external class as shown on (a) in the schematic figure below. Let us call this external class a “packer”. The numeric array in the packer is then processed according to the following steps. Firstly, to use as little additional memory as possible, all duplicated numeric values are removed. Next, the constants are shuffled randomly and are transformed in order to hide their actual values. Currently only three simple transformations are implemented: XOR-ing with one random value, XOR-ing twice with two different random values and a linear mapping. Then, a method stub to get the constant and reverse the applied transformation is implemented in the packer. Finally, each occurrence of a constant declaration is replaced with an invocation to the get-constant packer method.

![Diagram](a)

The transformation thus put represents not much of added complexity to the program. To further challenge the reverser, the packer class creates between 3 and 10 replicas of itself, each time applying anew the shuffling and the selection of the numeric transformation to the array. This means that even if the obfuscated application has several packer classes which apply the XOR-twice transformation, in each of them the two random numbers for the transformation will differ as well as the data array index of every unique numeric value. Designed like this, the transformation has the disadvantage of data duplication. However, an advantage that is possible due to this reduplication is removing the necessity that a single class containing constants is calling the get-constant method of the same packer which is shown on (b) in the figure above.

To summarize, control flow is complicated by multiple factors. Firstly, additional classes are introduced to the application i.e. more data processing paths in the program graph for the reverser to track. Then, in each packer class the array constant values will be seemingly different. Lastly, different packers are addressed to retrieve the numeric constants in a single class and the reverser would have to establish that the connection between each of the different packer calls is merely data duplication. Metadata information is hidden on an abstract level with the supplementary graph paths and the modified numeric values. Therefore by applying this transformation both static and dynamic analysis are hindered.
4.2.3 Strings obfuscation

Strings obfuscation is the only transformation which was found to be applied in some of the examined applications. Naming methods and classes with UTF-8 can be considered a form of strings obfuscation because in the .dex file format the latter are stored in the strings constant pool. Moreover, as was verified during manual analysis this naming convention breaks some of the analysis tools.

The decision to include this transformation in the tool is motivated by the fact that it could be a contribution since none of the here cited open-source tools implements strings encryption at the moment of submission. Moreover, the transformation is designed in such a way that it aspires to add more control flow complexity than what is currently found to be implemented [4] and instead of using a custom algorithm (usually simply XOR-ing with one value) the strings here are encrypted with the RC4 stream cipher [23].

A general reminder regarding the efficiency of this transformation is that hiding the key adequately can be more important than the strength of the used encryption algorithm.

The figure on the right gives an overview to how the transformation works. The classes containing strings are primarily filtered out. A unique key is generated for and stored inside each such class. All strings in a class are encrypted with the same class-assigned-key. Encryption yields a byte sequence corresponding to each unique string which is stored as a data array in a private static class field. This results in removing strings from the constant pool upon application re-assembly thus preventing from visibility with static analysis. A consideration to use static class fields for storing the encrypted strings is the relatively small performance impact. Decryption occurs during runtime, the strings being decoded once upon the first invocation of the containing class. Whenever a given string is needed, it is retrieved from the relevant class field.

Analogically to previous transformations, adding control flow complexity is at the cost of duplication. The obfuscator parses a decryption class template and creates between 3 and 10 semantically equivalent replicas of itself in the processed application as shown in the figure. Each class containing strings chooses randomly its corresponding decryption class. A simple trick applied with the aim to increase potency (i.e. confusing a human reader, not an automated tool [10]) is naming the replicas with logical strings which give no hint as to what is contained in the decryption class. Normally, a human reader would not expect decryption functionality in a class called InternalLoggerResponse.

To summarize, there are several minor improvements of our suggested implementation over what was found in related works. Encrypting the strings in each class with a unique key slows down automatic decryption because the keys are placed at different positions and need to be located separately for each class. Designing the transformation by using a decryptor-template approach allows in principal the developer to modify this template: they can either choose to strengthen potency and resilience or change easily the underlying encryption/decryption algorithm pair. Finally, the added control flow complexity is increased by the supplementary decryption classes.

4.2.4 Injecting “bad” code

Ideally, a highly resilient transformation would defeat the reverse engineering tool used by the adversary forcing them to either improve their custom deobfuscator or, hopefully for the source code defender, to give up. The proposed here transformation has as main
purpose to defy popular static analysis tools without claiming to be highly resilient. In fact, it is the contrary. We show that a simple combination of known exploits is enough to cause certain tools to crack and produce an output error. There are two defeat target tool types: decompilers and disassemblers performing static analysis. The used techniques are classified in previous works as “preventive” [10] for exploiting weaknesses of current analysis tools.

To thwart decompilers an advantage is taken from the discrepancy between what is representable as legitimate Java code and its translation into Dalvik bytecode. Similar techniques have been proposed for Java bytecode protection [4]. The Java programming language does not implement a goto statement, yet when loop or switch structures are converted into bytecode this is done with a goto Dalvik instruction. Thus by working directly on bytecode it is possible to inject verifiable sequences composed of goto statements which either cannot be processed by the decompilers or do not translate back to correct Java source code. In this particular implementation a bogus method is created containing goto statements which recursively call each other. Having this underlying idea in common, different variations are generated to harden automatic detection. Above is given the skeleton of an example recursive goto code sequence with an indirect recursion whose inner code is not detectable as dead code by the Dalvik optimizer.

To thwart disassemblers several “bad” instructions are injected directly in the bytecode. Execution of the bad code is avoided by a preceding opaque predicate which redirects the execution to the correct paths. This technique has already been shown to be successful [40]. However, since its publishing new tools have appeared and others have been fixed. The here suggested minor modifications are to include in the dead code branch: (1) an illegal invocation to the first entry in the application methods table; (2) a packed switch table with large indexes for its size; (3) a call to the bogus method we previously created such that it looks as if it is being used (not to be removed as dead code). The bytecode sequences corresponding to the first two items are given below with their mnemonics.

1. `7400 0000 0000 invoke-virtual/range {} method@0000`
2. `2b01 fdff ffff packed-switch v1, fdff ffff`

### 4.3 Transformation limitations

In order to take effect, all the here listed transformations had to comply with both the Dalvik verifier and optimizer. Although verification can be suppressed by changing a constant in the bytecode, this does not seem an eligible behavior for a goodware application. Moreover, the workflow used by our obfuscator relies on external tools which imply their own constraints. Hence, it is worth noting the limitations of the proposed transformations.

**Native Call Wrappers** is applied only to native methods which have no more than 15 registers. The reason is that smali has its own register implementation distinguishing between parameter and non-parameter registers and is working only by representing methods with no more than 15 non-parameter registers. In case more registers need to be allocated, the method is defined with a register range, not a register number.
4.4. PERFORMANCE RESULTS

Table 4.1: Profiles of the test applications. The label abbreviations are identical to those in the case study of applications. The black bullet marks a presence of the criteria. The label MISC stands for “miscellaneous” and indicates notable app features. In the facebook app, CCL stands for the custom class loader.

<table>
<thead>
<tr>
<th>APP</th>
<th>OBF</th>
<th>NAT</th>
<th>DYN</th>
<th>REF</th>
<th>CRY</th>
<th>MISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.adobe.reader.apk</td>
<td>0%</td>
<td>•</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>SD card</td>
</tr>
<tr>
<td>com.alensw.PicFolder.apk</td>
<td>100%</td>
<td>•</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>camera</td>
</tr>
<tr>
<td>com.disney.WMPLite.apk</td>
<td>5%</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>graphics</td>
</tr>
<tr>
<td>com.ebay.kr.gmarket.apk</td>
<td>0%</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>UTF-8 text</td>
</tr>
<tr>
<td>com.facebook.katana.apk</td>
<td>84%</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>CCL</td>
</tr>
<tr>
<td>com.microsoft.office.lync.apk</td>
<td>0%</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>phone calls</td>
</tr>
<tr>
<td>com.rebelvox.voxer.apk</td>
<td>0%</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>audio, SMS</td>
</tr>
<tr>
<td>com.skype.android.access.apk</td>
<td>0%</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>audio, video</td>
</tr>
<tr>
<td>com.teamlava.bubble.apk</td>
<td>0%</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>graphics</td>
</tr>
<tr>
<td>cz.aponia.bor3.czsk.apk</td>
<td>0%</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>GPS, maps</td>
</tr>
<tr>
<td>org.mozilla.firefox.apk</td>
<td>0%</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>internet</td>
</tr>
<tr>
<td>snt.luxtraffic.apk</td>
<td>0%</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>GPS, maps</td>
</tr>
</tbody>
</table>

Defined so to ease the editing of smali code, this has its restrictions on our transformation. Fortunately, on average an application has around 10% of methods using more than 15 registers which is not a severe limitation.

Packing Numeric Variables is applied only to the 4-bit and 16-bit registers, because there is a risk of overflowing due to the applied transformation when extended to larger registers. Clearly, a transformation shifts the range of the possible input values. Regarding the simple XOR-based modifications, the scope is preserved but a linear mapping shrinks the interval of possible values. Also, packing variables was restricted only to numeric constant types because in Dalvik registers have associated types i.e. packing heterogeneous data together might be a type-conversion dangerous operation. In the last chapter more details are given on this particular part of the DVM as well as the limitations it implies.

4.4 Performance results

To verify the efficiency of the developed tool a set of 12 test applications was selected among the huge variety. Nevertheless, this set tried to cover as many different features as possible. This includes games, social communication apps, location-related apps, apps containing UTF-8 encoded strings and apps manipulating the phone storage. The selected APK files and their profiling are shown on Table 4.1. Both obfuscated and non-obfuscated with ProGuard applications were selected, since none of the transformations has an impact on method names. As somewhat of a challenge, the facebook app was included to the benchmarks because it implements its own custom class loader to bypass the Dalvik maximum memory allocation restriction which is not a typical behavior for an application [36]. With the exception of one app, all others necessarily have native code. Otherwise testing the wrapper transformation is useless.

The performance tests of the modified applications were executed on two mobile devices: (1) HTC Desire smartphone with a customized Cyanogenmod v7.2 ROM, Android v2.3.7; (2) Sony Xperia tablet with the original vendor firmware, Android v4.1.1.

Detailed technical information about the test devices can be found in the appendix.
During the development process all transformations were tested and verified to work separately. On Table 4.2 are given the results of their combined application in accordance to the order specified by the automata on Figure 4.2. The plus sign should be interpreted as that the transformations have been applied consequently (e.g. \texttt{w+o+p} means applying adding wrappers then obfuscating strings then packing variables).

With the exception of the bad code injection on the facebook application, every application undergoing the possible combinations of transformations was installed successfully on both test devices. An observation on the error console logs for the facebook application suggests that it might implement its own bytecode verifier, or at least it passes the bytecode through a custom parser which conflicts with the injected bad code. The rest of the transformations did not make the app crash. For the Korean ebay app no crash occurred, but not all of the UTF-8 strings were decrypted successfully i.e. some messages which should have been in Korean appeared as their UTF-8 equivalent byte sequence. The most probable reason is that large alphabets are separated in different Unicode ranges and smali implements a custom UTF-8 encoding/decoding which might have a slight discrepancy with the encoding of python for some ranges. Finally, the voxer communication app did not initially run with the injected bad code. This lead to implementing the possibility to toggle the verification upon bytecode injection. By setting a constant in the method as verified its install-time verification can be suppressed. Enabling this feature let the voxer app run without problems. However, verifier suppression is disabled by default for security considerations.

Besides the upper mentioned, no other anomalies were noted on the tested applications. No noticeable runtime performance slowdown was detected while testing manually. The memory overhead added by each transformation separately is shown on Table 4.3. Because the applications differ significantly in size, for a better visual representation only the impact on the least significant megabyte is shown.

\begin{table}[h]
\centering
\begin{tabular}{l|c|c|c|c}
\textbf{APP} & \textbf{w} & \textbf{w+o} & \textbf{w+o+p} & \textbf{w+o+p+b} \\
\hline
\texttt{com.adobe.reader.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.alensw.PicFolder.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.disney.WMPLite.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.ebay.kr.gmarket.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.facebook.katana.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.microsoft.office.lync.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.rebelvox.voxer.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.skype.android.access.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{com.teamlava.bubble.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{cz.aponia.bor3.czsk.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{org.mozilla.firefox.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\texttt{snt.luxtraffic.apk} & \textbullet & \textbullet & \textbullet & \textbullet \\
\end{tabular}
\caption{Testing the obfuscated applications on HTC Desire and Sony Xperia tablet. The transformations abbreviations are as follows: \texttt{w} adding native wrappers, \texttt{o} obfuscating strings, \texttt{p} packing variables, \texttt{b} adding bad bytecode. The black bullet indicates successful install and run after applying the series of transformations.}
\end{table}

\footnote{https://code.google.com/p/smali/source/browse/dexlib/src/main/java/org/jf/dexlib/Util/Utf8Utils.java}
Testing analysis tools on modified bytecode

4.5.1 Adding native call wrappers

For this transformation the application was analyzed with androguard, presicely with the androlyze.py console tool and the Sublime text plugin. Initially, all native methods with their containing packages were found in the androlyze console using:

```python
> a, d, dx = AnalyzeAPK("com.rebelvox.voxer.apk")
> show_NativeMethods(dx)
```

When attempting to view the source code of the five found methods, all of them were empty. For example:

```python
method: get_frame_to_play ([B)V [public static native] size:0
```

This means that their actual code is located in a native library and cannot be seen with static analysis. However, here we look for their usage, not their implementation. By assigning to a unique variable each of the native methods, we can use the androguard function `show_Paths()` to track the usage. In this particular case, our wrapper was located in the class `com/android/support/v4/util/AtomicFile` and had the name `d(Lcom/rebelvox/voxer/System/NativeSystem; [B)V`. The next step is to locate
where `Landroid/support/v4/util/AtomicFile/d()` is called. The same approach was used and eventually the original call was found. Thus, this transformation alone does not represent a serious reversing slowdown. For the challenge, another reversing round when applied twice was done. While analysis time was indeed increased, this also had a slightly negative impact noticeable on the performance of the HTC smartphone. The Sony tablet ran smoothly.

4.5.2 Packing numeric variables

For this transformation the Unix `grep` command and `baksmali` tool were used. The latter were selected because we are looking for numeric constants packed in a separate class which can be done quite quickly with pattern matching. As a first step, the app was processed with `baksmali` which produced a directory with corresponding files for each class. A recursive `grep` search was done to locate the occurrences of all `const/16` because we know that all packed constants are shifted to 16 bits.

Regarding the previously discussed limitation of this transformation not all numeric constants are packed, only when this is type safe for the registers. Thus, a first challenge to the reverser is how to determine statically which of the classes contain the real constants and which contain the modified constants.

Let us suppose our obfuscator source code is available to anyone, as it actually is. Then, to filter out the injected by our obfuscator packer classes is no longer a time consuming task. In this particular case, the knowledge of the keywords forming the pseudo-random packer class name was used to distinguish them. The keywords can be referenced in the `utilsSmali.py` file, in the `generateClassName` method. Finally, any text editor can be used to view the mnemonics generated by `smali` and due to the simplicity of our transformations, no significant knowledge of Dalvik bytecode is necessary to obtain the initial constant values.

As a final remark, this transformation is a very good example of how relative it is to estimate which reversing tool is best. Knowing exactly what to look for, we used the right combination of tools and techniques too find it. Had we used the `androguard` DAD decompiler to review mnemonics and convert back to source code, all we would have gotten inside the packer class is the constant get method alone:

```java
public static short get(int p3)
{
    return ((org...BasicInternalImplementationProcessor.data[p3] ^ 244) ^ 24);
}
```

This is because we tricked the DAD decompiler by placing the data array after the return statement. This code parsed as legitimate without any problems by `baksmali` managed to fool `androguard` which implements a seemingly more sophisticated recursive traversal algorithm.

4.5.3 Strings obfuscation

For this transformation the application was analyzed with and the `androguard` Sublime text plugin. Since this transformation affects all hardcoded strings in the app, we are free to pick a random class for examination. According to the description, all strings are stored as byte arrays in private class fields and are decrypted once altogether upon class initiation. While there is no way to verify the decryption without runtime emulation, we could still make an attempt to statically obtain the strings. Let us look inside the class and its init:
4.5. TESTING ANALYSIS TOOLS ON MODIFIED BYTECODE

Lcom/rebelvox/voxer/System/LocalNotificationManager;-><clinit>()V
static LocalNotificationManager()
{
    v1 = new byte[150];
    v1 = {205, 159, 2, ......, 119, 127};
    v0 = new com.actionbarsherlock.BasicRandomEventHandler(v1);
    v1 = new byte[5];
    v1 = {136, 88, 68, 135, 21};
    com.rebelvox.voxer.System.LocalNotificationManager._p7890 = v0.up(v1);
    v1 = new byte[6];
    v1 = {12, 90, 93, 245, 185, 102};
    com.rebelvox.voxer.System.LocalNotificationManager._e1951 = v0.up(v1);
    ...
}

We can see that the initialized variables are static string class fields:
field: _e1951 Ljava/lang/String; [private static java.lang.String]
field: _p7890 Ljava/lang/String; [private static java.lang.String]

An instance of the class BasicRandomEventHandler is stored in the parameter register v0 and each string class field is assigned a value by calling the up method from this class. Although its name does not immediately suggest implementing a string decryption algorithm, let us suppose the reverser looks inside the BasicRandomEventHandler class (comments were added to clarify the functionality of each method to the reader). As a reminder, the encryption is done using RC4.

com/actionbarsherlock/BasicRandomEventHandler extends java/lang/Object
method: <init> ([B)V [public constructor] size:61 //initiate stream from seed
method: up ([B)Ljava/lang/String; [public] size:26 //actual decryptor

Looking at the recovered source code of the methods none of them appears to call any of the other methods, although a correlation between the constructor and RGB can be established due to the similarity of the performed actions. The reverser has to look at the mnemonics of the up method to see that it invokes the RGB method for decryption. An experienced reverser would recognize the RC4 algorithm, but to decrypt they need to re-write the disassembled code to recover the plaintext or emulate the execution.

A tool which claims to do this automatically is dexguard, however its is unavailable at submission time so we could not challenge our transformation [6]. Moreover, even if this process is automated, each time the stream needs to be re-initiated manually with the uniquely generated decryption class key. Another tool which does automatic strings decryption is part of dex2jar and is called dex-tool-0.0.9.12[3]. In this case it is useless against our encryption because it handles only methods with the signature Ljava/lang/String en(decrypt(Ljava/lang/String); but we represent the encrypted strings as byte data arrays.

In total our transformation encrypted 9725 strings which were distributed in more than 2000 of the 3539 classes i.e. more than 2000 unique keys to decrypt with. A rough estimation of the time and efforts needed to reverse all strings left to the reader.

4.5.4 Injecting “bad” code

**androguard**

*Executed command*

```bash
./androlyze.py -i com.rebelvox.voxer.apk -m exec
```

*Output*

```
23 (0000004a) packed-switch-payload 12b0000:
24 (00000052) AG:invalid_instruction (OP:fd)
25 (00000054) AG:invalid_instruction (OP:ff)
```

*Note:* The entire app was successfully processed by androguard, but the output produced the methods internal code as a packed switch data array. Some methods for which injection is not applicable were recovered successfully (see also dedexer).

**apktool and baksmali**

*Executed commands*

```bash
apktool d com.rebelvox.voxer.apk testApktool
java -jar baksmali-1.4.2-dev.jar -o testBaksmali com.rebelvox.voxer.apk
```

*Output*

```
UNEXPECTED TOP-LEVEL EXCEPTION:
org.jf.dexlib.Util.ExceptionWithContext: regCount does not match the number of arguments of the method
at org.jf.dexlib.Util.....withContext(ExceptionWithContext.java:54)
at org.jf.dexlib.Code.....IterateInstructions(InstructionIterator.java:91)
at org.jf.dexlib.CodeItem.readItem(CodeItem.java:154)
at org.jf.dexlib.Item.readFrom(Item.java:77)
at org.jf.dexlib.OffsettedSection.readItems(OffsettedSection.java:48)
at org.jf.dexlib.Section.readFrom(Section.java:143)
at org.jf.dexlib.DexFile.<init>(DexFile.java:431)
at org.jf.baksmali.main.main(main.java:280)
Caused by: java.lang.RuntimeException: regCount does not match the number of arguments of the method
at org.jf.dexlib.Code.Format.Instruction3rc.<init>(Instruction3rc.java:44)
at org.jf.dexlib.Code.Format.....$Factory.makeInstruction(Instruction3rc.java:145)
at org.jf.dexlib.Code.....IterateInstructions(InstructionIterator.java:82)
... 6 more
Error occurred at code address 152
code_item @0x91074
```

*Note:* Since apktool is based on baksmali their console outputs were identical.
4.5. TESTING ANALYSIS TOOLS ON MODIFIED BYTECODE

DARE decompiler

Executed command
dare -d testDare com.rebelvox.voxer.apk

Output
Processing class #2486: Lnet/hockeyapp/android/internal/ExpiryInfoView;
W/dalvikvm(11427): Error while translating ao opcode: type object - constant:103
W/dalvikvm(11427): Unknown instruction format
W/dalvikvm(11427): Error while translating ao opcode: type object - constant:103
W/dalvikvm(11427): Unknown instruction format
W/dalvikvm(11427): Error while translating ao opcode: type object - constant:103
W/dalvikvm(11427): Unknown instruction format

Note: According to the project website, DARE is the improved to target Android version of the DED decompiler [43]. When attempting to process the modified application with DARE, a large console log similar to the output above was produced. After some point, the decompiler looped endlessly: for the testing it was left to run 3 hours with no success. When keyboard-interrupted, the result was having a nested hierarchy of directories corresponding to the packages of the application as well as for its optimized version. Eventually, the application was not processed at all since those directories were empty.

dedexer

Executed command
java -jar ddx1.25.jar -d testDedexer classes.dex

Output without injecting junk code sequences after the opaque predicate
Processing android/...ServiceInfoCompat$AccessibilityServiceInfoStubImpl
Processing android/...ServiceInfoCompat$AccessibilityServiceInfoIcsImpl
Processing android/support/v4/accessibilityservice/AccessibilityServiceCompat
Processing android/...ServiceInfoCompat$AccessibilityServiceInfoVersionImpl
Processing android/support/v4/app/Fragment
Unknown instruction 0xFF at offset 000A4CBC

Note: Only a small part of the app (the upper listed 5 classes) was successfully processed by dedexer.

Output with injecting junk code sequences after the opaque predicate
192876: goto 19289a
192878: data-array
0x00, 0x32, 0x10, 0x03, 0x00, 0x28, 0x08, 0x28, 0xF5, 0x1A, 0x00, 0xF3, 0x1B, 0x71, 0x20, 0x16, 0x0F, 0x10, 0x00, 0x0A, 0x00
end data-array
19289a: goto 192876
19289c: data-array
0x71, 0x00, 0xFC, 0x4E, 0x00, 0x00, 0x13, 0x00, 0x2F, 0x6E, 0x20, 0x72, 0x49, 0x10
end data-array

See the files addBadCode.py, method buildOpaque and utilsOpaque.py, part 2: junk code.
Note: The entire app was processed, but when looking inside a .ddx file few parts of the code were translated back to legitimate mnemonics. The majority of the recovered code looked like the data array bytes given above. The recursively calling goto sequence can be seen between the addresses 192876 and 19289a. The method internal code is represented as a data array on address 19289c. It is not always applicable to inject the bad code sequences. For example methods which are static, native or abstract are not processed because they do not have the necessary registers to inject the opaque predicate. Hence, some methods were reversed successfully.

**dex2jar**

*Executed command*

```
./d2j-dex2jar.sh com.rebelvox.voxer.apk
```

**Output**

```
dex2jar touched-com.rebelvox.voxer.apk -> touched-com.rebelvox.voxer-dex2jar.jar
...DexException: while accept method:[Landroid/...ModernAsyncTask$3;.done()V]
at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReader.java:701)
at com.googlecode.dex2jar.reader.DexFileReader.acceptClass(DexFileReader.java:448)
at com.googlecode.dex2jar.reader.DexFileReader.accept(DexFileReader.java:330)
at com.googlecode.dex2jar.v3.Dex2jar.doTranslate(Dex2jar.java:84)
at com.googlecode.dex2jar.v3.Dex2jar.to(Dex2jar.java:239)
at com.googlecode.dex2jar.v3.Dex2jar.to(Dex2jar.java:230)
at com.googlecode.dex2jar.tools.Dex2jarCmd.doCommandLine(Dex2jarCmd.java:109)
at com.googlecode.dex2jar.tools.BaseCmd.doMain(BaseCmd.java:168)
at com.googlecode.dex2jar.tools.Dex2jarCmd.main(Dex2jarCmd.java:34)
Caused by:...DexException: while accept code in method:[...AsyncTask$3;.done()V]
at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReader.java:691)
... 8 more
Caused by: java.lang.ArrayIndexOutOfBoundsException: 0
at com.googlecode.dex2jar.reader.Dex OpcodeAdapter.xrc(Dex OpcodeAdapter.java:791)
... 8 more
```

**Note:** Having this output error `dex2jar` produces an empty `.jar` file.

**dexter**

To the benefit of the reverser or the disappointment of the code protector, `dexter` did not fall for any of our bytecode injection tricks. This result was expected since we use a similar approach to what was described by one of the tool’s authors for our bytecode injection [40]. Alongside the development process about 20 applications were analyzed with `dexter`, four of which produced an error. Since the code is server-side closed and no error log information was available on the website, only a supposition on what may have caused the error is suggested here for the sole purpose to give feedback for improving the tool. Three out of the four applications which crashed had UTF-8 names (e.g. `NotifierSettings$容`) which most likely is an indicator that `dexter` does not yet handle such cases. The fourth problematic app was successfully reversed with `androguard`. 
4.6 SUMMARY

JD-GUI

Output

```java
public void setUpdateThrottle(long paramLong)
{
    if (('å' % 2 == 0) || ((-1 + 'å' * 'å') % 8 != 0))
        while (true)
            new String[3];
    this.mUpdateThrottle = paramLong;
    if (paramLong != 0L)
        this.mHandler = new Handler();
}
```

Note: To test separately the effect of the recursive goto sequences on decompilers, bad code injection was removed. In JD-GUI some classes produced //INTERNAL ERROR/. The remaining classes were translated into not compilable, yet relatively easy to correct by manual examination Java code. Although not intentional, the transformation had an effect on the encoding of the variable names and represented them as strings instead of numeric variables. An obvious drawback of the currently used opaque predicates is the ease with which they can be detected and removed manually. This weakness is due to the fact that in order to comply with Dalvik’s requirement for the registers to have known types, they had to be initialized with a value before being used by the predicates. Trying to avoid this resulted in a verifier error.

4.6 Summary

This chapter proposes a possible implementation of a Dalvik bytecode obfuscator. The obfuscator is called half-jokingly “Innocent Dalvik Obfuscator” for two reasons. Firstly, none of the transformations applied alone is robust enough against an experienced reverse engineer armed with multiple analysis tools. Secondly, combined together our transformations have a very reasonable impact on the underlying application: no more than 1Mb of additional memory altogether and no noticeable CPU slowdown when tested with an old phone. It is often the case that a balance between resilience, potency, stealth and cost has to be found in an efficient obfuscator. This can lead to compromise either with performance, or with security. Moreover, one is not limited to mingling solely on bytecode level. In the current state of most freely available Android analysis tools, our four bytecode transformations combined with a source code level UTF-8 class and method naming can already provide a good protection level against all here tested tools.
Final Remarks

The final chapter focuses on topics which went naturally alongside the development of the Dalvik obfuscator. In the succeeding section an attempt to initiate a discussion on applying known x86 bytecode obfuscation techniques on Dalvik is proposed. Both static and dynamic techniques are reviewed. In the concluding discussion are given a summary of the contributions of this work and a possible future development.

5.1 Remarks on obfuscating Dalvik bytecode

Our suggested obfuscator aimed to be generic, meaning that the transformations do not reduce the input file set with preliminary requirements. Here we argue about the limitations of applying some obfuscation techniques on Dalvik bytecode. These limitations might be either that the transformation is not generic or it cannot be applied at all. The term “not generic” should be interpreted as that it can be applied in practice, but it has to be tailored to the particular application. Such restrictions emerge because the nature of some transformations is dependent on whether or not a program has certain features, which respectively implies constraints on the input file.

Part of the here written conclusions are based on first-hand attempts to evaluate some techniques. Others are result of looking through the Android source code files and reading related works.

5.1.1 Static obfuscation techniques

Encoding. Despite the fact that the ARM-based platform supports mixed endianess, the dex file verifier expects the input bytes to be little endian. As a reference the code fragment from the verifier which checks the endianess is presented below:

```
FILE: /dalvik/libdex/DexSwapVerify.cpp, LINE: 301, PLATFORM v4.2.2
1: if (pHeader->endianTag != kDexEndianConstant) {
2:   ALOGE( "Unexpected endian_tag:%#x", pHeader->endianTag);
3:   return false;
4: }
```

Checking the value for the endian constant shows that it is assigned to \(0x12345678\) which in the dex file reference stands for little endian [34]. The exact code fragment is given below:

```
FILE: /dalvik/libdex/DexFile.h, LINE: 75, PLATFORM v4.2.2
1: enum {
2:   kDexEndianConstant = 0x12345678,
3:   kDexNoIndex = 0xffffffff,
4: );
```

Given these circumstances, endianness manipulations are not feasible as was suggested in [32] since the file would not be verified.
Reordering code and data. Usually in a non-obfuscated program the locality of code and data play an important role as giving information to the reverse engineer. Therefore, a logical way to distribute important information is to apply code and data reordering. In C/C++ like languages where the programmer is himself responsible for memory management and could optimize certain operations with pointer arithmetics, a misplaced of parts in the code could have various consequences. When performed by taking data dependencies into account, reordering can be regarded as an obfuscation technique. This was suggested in 1993 by Fred Cohen as means to create semantically equivalent versions of the same program [8]. The very same technique applied regardless of data boundaries could result in either a non-working program or an appropriate setting for buffer overflow exploits. The latter is possible in an architecture like x86: there is no separation between data and instructions, both are written on the same memory block and instructions are executed consecutively [45]. There are two reasons why buffer overflow exploits are not directly applicable on Dalvik bytecode. Firstly, the DVM checks array access bounds for each architecture which is supported to run Android. This can be seen in the two samples (ARMv7 and x86) of source code below:

FILE: /dalvik/vm/mterp/out/InterpAsm-armv7-a.S, LINE: 1895, PLATFORM v4.2.2
1: cmp r1, r3 @ compare unsigned index, length
2: bcs common_errArrayIndex @ index >= length, bail

Note: In the assembly file for ARMv7 all opcodes containing AGET (array get) and APUT (array put) perform bound checks. Only the first is given as a proof above.

FILE: /dalvik/vm/mterp/out/InterpC-x86.cpp, LINE: 970, PLATFORM v4.2.2
1: if (GET_REGISTER(vsrc2) >= arrayObj->length) {
2:   dvmThrowArrayIndexOutOfBoundsException(arrayObj->length, GET_REGISTER(vsrc2));
3:   GOTO_exceptionThrown();
4: }
5: }

Secondly, as in other virtual machines and interpreters, in the DVM the instructions are separated in memory from the data because of data security and reliability issues. As a final remark, although on the level of DVM it is not possible to exploit buffer overflow, underneath the DVM, the native architecture still follows the principle of no separation between data and instructions. Thus, one could make use of this technique with a custom native module.

Jump exploits limitations. An obfuscation technique for thwarting recursive traversal proposed on x86 assembly is implementing a branch function which alternates the control flow [22]. The basic idea is to construct a finite map over jump locations in the program and replace direct jumps with a call to a special function which returns the mapped jump target address. A schematic illustration is given below:

11: goto a1
12: goto a2
13: goto a3
M = { l1 -> a1, l2 -> a2, l3 -> a3 }
l1: call M

The non-obfuscated code and its corresponding control flow are given on (a). The generated address mapping function M is shown on (b). The result of redirecting the control flow through M is shown on (c). To increase the potency by hiding the real address
values in the branch function $M$, one could store their hashed values and return the reversed hash value at runtime. This improvement is possible on x86 because this architecture allows direct manipulation of registers. Moreover, the instruction pointer itself is a register i. e. its value can be altered with load or store instructions. For Dalvik bytecode a verification function enforces constraints on the branch instructions targets. This can be seen in the following (only the most relevant parts of code are cited):

FILE: /dalvik/vm/analysis/DexVerify.cpp, LINE: 717, PLATFORM v4.2.2

```c
1: if (!selfOkay && offset == 0) {
2:     LOG_VFY_METH(meth, "VFY: branch offset of zero
3:         not allowed at %#x", curOffset);
4:     return false;
5: }
6: if (((s8)curOffset + (s8)offset) != (s8)(curOffset+offset))
7:     LOG_VFY_METH(meth, "VFY: branch target overflow %#x +%d",
8:         curOffset, offset);
9:     return false;
10: }
11: if (absOffset < 0 || absOffset >= insnCount ||
12:     !dvmInsnIsOpcode(insnFlags, absOffset))
13: {
14:     LOG_VFY_METH(meth,
15:         "VFY: invalid branch target %d (-> %#x) at %#x",
16:         offset, absOffset, curOffset);
17:     return false;
18: }
```

When the code is loaded, the DVM preliminarily scans and marks the beginning addresses of the instructions. Each instruction is then flagged by the space offset which it requires, leaving all unflagged bytes to be interpreted as data or parts of a long instructions. The main reason why unconditional address jumps are impossible is because the DVM expects each target to be constant i.e. its value must be known at compile time and cannot be altered during runtime. On line 1 the cited above code asserts that instructions do not branch into themselves with the exception of a few ones allowed to do so. On line 7 a check against 32-bit overflow is done. On line 11 the check prevents from unconditional memory jump, only valid opcodes can be jump targets. To summarize, the DVM expects valid instructions as jump destinations and manages them as constant offsets. Code containing violations of these requirements would cause a verifier (VFY) error.

**Merging or splitting code.** Popular transformations applied by obfuscators which add complexity to the program graph include control flow flattening and injecting dead code in a method. Although differing by their underlying ideas, in essence these modifications require to model the input program as a set of abstractions, parse it according to these abstractions and modify it by either merging or splitting code. There are considerable limitations when executing those techniques directly on Dalvik bytecode. The reason is that in Dalvik one cannot freely meddle with registers because they have associated types. This can be seen in the bytecode structural verifier, the summary of its most relevant parts given below. On the left are code starting line positions and on the right side is a short description of what is implemented.
Let us now look at how these imposed by the DVM register type restrictions influence the concrete merge and split techniques.

**Control flow flattening** is a code merge technique in which a nested control flow sequence is packed into a “flattening” structure. In Java and C/C++ this structure is most often a switch statement, in C/C++ and x86 assembly one could also use labels and goto statements instead. To clarify, a simple example of Euclid’s GCD algorithm with its corresponding control flow is given:

```java
1: int gcd(int a, int b) {
2:     while(a != b) {
3:         if(a > b) {
4:             a = a - b;
5:         } else {
6:             b = b - a;
7:         }
8:     }
9:     return a;
10: }
```

After flattening the same sequence of code and its graph would look like the following:

```java
1: int gcd(int a, int b) {
2:     int next = 0;
3:     switch(next) {
4:         case 0: if(a != b) next = 1; else next = 4; break;
5:         case 1: if(a > b) next = 2; else next = 3; break;
6:         case 2: a = a - b; next = 0; break;
7:         case 3: b = b - a; next = 0; break;
8:         case 4: return a;
9:     }
10: }
```

Constructing a flattened version of a given method on Dalvik bytecode requires complex preliminary analysis. After the code is divided in abstractions for each branch of the flattening structure, the union of the registers needed for all those branches has to be taken as the input register number of the newly created method. Then, unlike in Java,
5.1. REMARKS ON OBFUSCATING DALVIK BYTECODE

the code cannot simply be copied into the branch statement, an analysis is needed for the registers types usages and possible side effects. If a side effect of the code is for example modifying more than one value, these cannot be returned by the flattened function and a shared class fields must be used to maintain the semantics. Moreover, all entry and exiting points of the branches need to be asserted correct register types. While this is technically possible, it is hard to be implemented generically and might be quite an unsafe operation.

Injecting dead code in a method is a code splitting technique. In the previous chapter one possible variant of dead code injection with opaque predicates was shown. A straightforward “trick” to guarantee that this modification is type safe is to inject the dead code before any registers are used i.e. just after the method declaration. However, a thorough preliminary static analysis is needed if the bogus branch was to split the method in two. Here are proposed two implementation possibilities. The simpler is to trace which registers are free at the point of insertion and use only those freely available registers to construct the opaque predicate. Although relatively type safe, such an implementation will highly likely restrict the strength of the inserted opaque predicate: by default the bytecode is optimized for using as least as possible additional registers. Empirical testing showed that at most three registers were found to be freely available at suitable intersection points, a challenge to design a strong opaque predicate with. The alternative is to allocate as much registers as needed for a strong predicate which has the drawback of being risky. Firstly, the registers need to be checked for availability by tracing both before and after the insertion point because jumps are possible in either direction. Secondly, a register type-checking with regards to the possible jumps is required. Finally, after inserting the dead code all used registers need to be converted back to types that the succeeding code is expecting to receive. Another restriction is that not all register types can be freely converted into each other as can be seen in the merge table in the /dalvik/vm/analysis/CodeVerify.cpp file. Again, this is technically possible on bytecode, but much more feasible to apply code on source code level.

5.1.2 Dynamic obfuscation techniques

The following techniques can be successfully applied on Android, however with a limitation regarding generality. This limitation is imposed by the system class loader. There are two main “obstacles” when applying dynamic obfuscation: (1) publicly accessible methods work only on files which are saved to the file system before loading; (2) optimization, which is a compulsory step before execution, stores in memory the optimized files and secures them with system permissions. To circumvent those, a custom class loader needs to be implemented and previous work suggests that one possibility is to have it as native code loaded by the Java Native Interface provided in the DVM [39]. Such a custom loader could be used to implement either of the below listed transformations.

Dynamic code changes. To complicate dynamic analysis, it is possible to obfuscate a program such that its control flow differs upon each execution. Two essential steps need to take place for a program to be a self modifying one. Firstly, the code has to be converted into an “initial configuration” state after which the runtime code transformer should be added [9]. It is the second step which is not applicable generically on Android because the logic for dynamic changes should be inside a custom class loader. Since the DVM is based on the JVM, the instructions do not have direct memory access because
Java does not support pointer operations for data integrity reasons. Thus, the custom class loader would act as part of the DVM itself, having access to the virtual machine’s memory where the code is and alternating the program behavior. While possible, this is clearly not a generic transformation, it needs to be applied to the concrete target program. For example, in C/C++ programs, a possible dynamic change technique is to duplicate the semantics of a method in two syntactically different versions which interchange calls at runtime [9]. In the DVM, the JIT compilation requires that one tailors such techniques by adding means to locate the methods in interest during execution (e.g. with an a priori know value variable).

**Dynamic code loading.** Used both by malware and legitimate applications to load external code, this technique is shown to be successfully applied with the help of a custom class loader [39]. To answer the question whether it can be applied generically, a consideration has to be made. Let us suppose one would like to load some given classes externally. This means that all invocations to those classes, be it to access static class fields or to create a class objects, have to go through the custom class loader. This implies that the external class loader could induce noticeable performance slowdown if not implemented optimally. Moreover, the case study on market applications proves a major proportion of the apps use Java reflection. If one would like reflection to work with dynamic class loading, the entire application needs to be processed with the custom loader: a challenge regarding performance issues. To maintain a good performance, only selected classes should be loaded dynamically which imposes a constraint on the usage of reflection. Therefore genericity with dynamic code loading is restricted.

**Code encryption.** There are several considerations which need to be taken into account to adapt this technique for the Android platform. While it is clear that the encryption would be performed on the application .dex file, there are some subtleties regarding the decryption at runtime. During the unpacking process, after the successful decryption of the .dex file, it should be passed to the DVM for loading and execution. Dynamic loading is possible due to the support of reflection in Dalvik, but the contained public methods can only be executed if the file is stored in the file system. Thus, by saving the decrypted and decompressed .dex file on the device’s storage, the previously applied protection becomes impractical. Moreover, the bytecode is optimized upon its initial launch and the .odex file is stored in the cache secured by enforced system permissions. Implementing a custom file dex loader can bypass the restrictions of interfacing directly the libraries within the DVM. To summarize, encryption can be implemented analogically to dynamic code loading which brings up the mentioned performance and lack of genericity considerations. In this case, the performance is also highly dependent on the efficiency of the chosen encryption/decryption algorithms pair. Finally, the key must be stored in the decryption program stub i.e. is available to the reverser and if not hidden appropriately this technique is ineffective.

This subsection concludes with a remark regarding the stealth of the here listed dynamic transformations on Android. Applying either of them to the entire application is not performance efficient, yet selecting a subset of classes to load dynamically or encrypt gives an immediate hint to where the valuable code is. It can be the case that code which needs to be protected is also critical for the performance of the application. If so, obfuscation represents an additional layer of processing time and allocated memory. Therefore each application which makes use of some dynamic modifications can be seen as a special case which needs determining what technique to use and how.
5.2 Discussion

This work accented on several important aspects of code obfuscation for the Android mobile platform. To commence, we confirmed the statement that currently reverse engineering is a lightweight task regarding the invested time and computational resources. We studied more than 1600 applications for possible applied code transformations, but found no more sophisticated protection than variable name scrambling or its slightly more resilient variation of giving Unicode names to classes and methods. In some applications we also found encryption applied on strings generated during runtime. Yet, these applications themselves had hardcoded strings visible with analysis tools.

Having demonstrated the feasibility of examining randomly selected applications, we proposed a proof of concept open-source Dalvik obfuscator with the purpose of introducing a reasonable slowdown in the reversing process. Our obfuscator performs four transformations three of which target both data flow and control flow. The last transformation is a slight modification to a proven efficient technique from previous work. We challenged various analysis tools on our modified code, showed that the majority of them are defeated and proposed an already used in practice supplementary source-code transformation to target the remaining.

During the development process it was occasionally necessary to look through the source code of the DVM. Also, except several blog posts no previous comments were found on what known from the x86 architecture obfuscation techniques can be applied on Android.

This motivated the writing of the last chapter: our attempt to initiate such a discussion by summarizing how popular techniques can be adapted for Dalvik bytecode.

Android is merely since five years on the market, but because of its commercial growth much research is conducted around it. The evolution of the platform is a constantly ongoing process. It can be seen in the source code that some of the now unused bytecode instructions were former implemented test instructions. Possible future opcode changes may invalidate the effects our transformations. Moreover, analysis tools will keep on getting better and to defeat thems newer, craftier obfuscation techniques will need to be applied. This outwitting competition between code protectors and code reverse engineers exists ever since the topic of obfuscation has been established of practical importance. So far, evidence proves this game will be played continuously.
Bibliography


[16] , Content providers, URL: http://developer.android.com/guide/topics/providers/content-providers.html


[18] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall, These aren’t the droids you’re looking for: retrofitting android to protect data from imperious applications, Proceedings of the 18th ACM conference on Computer and communications security (New York, NY, USA), CCS ‘11, ACM, 2011, pp. 639–652.


[27] Dedexer Project Home Page, URL: http://dedexer.sourceforge.net/

[28] Dex2jar Project Home Page, URL: https://code.google.com/p/dex2jar/

[29] Dexter Project Home Page, URL: http://dexter.dexlabs.org/


[31] Radare2 Project Home Page, URL: http://radare.org/y/?p=download


Appendix
<table>
<thead>
<tr>
<th>App name</th>
<th>Old URL</th>
<th>SHA 256</th>
<th>Malware</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ck.screen.wallpapers.theme.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.lock.screen.wallpapers.theme</code></td>
<td><code>1d04c6f60a280e97ce8f2b913c9edbbcc34b53bdaa5f511bd418f60f292aba</code></td>
<td><code>BC2EE6F861843E6FE5A4A14CB44372.apk</code></td>
</tr>
<tr>
<td><code>com.app4xtreme.nfsdrifting.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.app4xtreme.nfsdrifting</code></td>
<td><code>173c15baf398e1bc27634b0ea2dd462f4e69527879fbb32e26154b2150d17548kim.apk</code></td>
<td></td>
</tr>
<tr>
<td><code>com.asphaltsevenfree.cheats.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.asphaltsevenfree.cheats</code></td>
<td><code>1d04c6f60a280e97ce8f2b913c9edbbcc34b53bdaa5f511bd418f60f292aba</code></td>
<td><code>BC2EE6F861843E6FE5A4A14CB44372.apk</code></td>
</tr>
<tr>
<td><code>com.blwp.s4lwp.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.blwp.s4lwp</code></td>
<td><code>1d04c6f60a280e97ce8f2b913c9edbbcc34b53bdaa5f511bd418f60f292aba</code></td>
<td><code>BC2EE6F861843E6FE5A4A14CB44372.apk</code></td>
</tr>
<tr>
<td><code>com.emoji.keyboard.emoticons.texting.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.emoji.keyboard.emoticons.texting</code></td>
<td><code>1d04c6f60a280e97ce8f2b913c9edbbcc34b53bdaa5f511bd418f60f292aba</code></td>
<td><code>BC2EE6F861843E6FE5A4A14CB44372.apk</code></td>
</tr>
<tr>
<td><code>com.neon.purple.keyboard.skin.free.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.neon.purple.keyboard.skin.free</code></td>
<td><code>1d04c6f60a280e97ce8f2b913c9edbbcc34b53bdaa5f511bd418f60f292aba</code></td>
<td><code>BC2EE6F861843E6FE5A4A14CB44372.apk</code></td>
</tr>
<tr>
<td><code>com.yanhong.banknote.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.yanhong.banknote</code></td>
<td><code>5b6402cc7e2e37271ee14e907e58c289c280cd71391b28807286f0393c124486</code></td>
<td><code>ThreatJapan_4C937667CB23E857D42B664334E1142A_NewsAndroidcode03.apk</code></td>
</tr>
<tr>
<td><code>com.yanhong.fashion.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=com.yanhong.fashion</code></td>
<td><code>5b6402cc7e2e37271ee14e907e58c289c280cd71391b28807286f0393c124486</code></td>
<td><code>ThreatJapan_4C937667CB23E857D42B664334E1142A_NewsAndroidcode03.apk</code></td>
</tr>
<tr>
<td><code>puzzle.droidapp.awesomesg.apk</code></td>
<td><code>https://play.google.com/store/apps/details?id=puzzle.droidapp.awesomesg</code></td>
<td><code>40be91b33429e0a22877a7a6f2204c5b95ed89b785e1b19149baa7acb20f6b</code></td>
<td><code>4A300481411AB1992467959491DF412C.apk</code></td>
</tr>
</tbody>
</table>

Table A.1: Malware apps removed from the market.
<table>
<thead>
<tr>
<th>App name:</th>
<th>com.blazemobile.SketchGuruArtistPicturePhoto.apk</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA 256:</td>
<td>213e042b3d5b489467c5a461ffdd2e38edaa0c74957f0b1a0708027e66080890</td>
</tr>
<tr>
<td>Malware:</td>
<td>56033daef6a020d8e64729acab103f818.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.geek.radio.Bulgaria.apk</td>
</tr>
<tr>
<td>SHA 256:</td>
<td>be90c12ea4a9dc40557a45387c7d631324ae396f7343c8</td>
</tr>
<tr>
<td>Malware:</td>
<td>zitmo.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.huashao.guns.apk</td>
</tr>
<tr>
<td>SHA 256:</td>
<td>fbf03f3d3c3066fa80b83d11fe29d3e6f9de685435e182ce12c64f3f71f0</td>
</tr>
<tr>
<td>Malware:</td>
<td>plankton.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.kennedy.cIphoneRingtones.apk</td>
</tr>
<tr>
<td>Old URL:</td>
<td><a href="https://play.google.com/store/apps/details?id=com.kennedy.cIphoneRingtones">https://play.google.com/store/apps/details?id=com.kennedy.cIphoneRingtones</a></td>
</tr>
<tr>
<td>SHA 256:</td>
<td>213e042b3d5b489467c5a461ffdd2e38edaa0c74957f0b1a0708027e66080890</td>
</tr>
<tr>
<td>Malware:</td>
<td>56033daef6a020d8e64729acab103f818.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.lwp.drift.racing.apk</td>
</tr>
<tr>
<td>Old URL:</td>
<td><a href="https://play.google.com/store/apps/details?id=com.lwp.drift.racing">https://play.google.com/store/apps/details?id=com.lwp.drift.racing</a></td>
</tr>
<tr>
<td>SHA 256:</td>
<td>5b6402cc7e2e37271ee14e907e58c289c280cd71391b28807286f0393c124486</td>
</tr>
<tr>
<td>Malware:</td>
<td>ThreatJapan_4C937667CB23E857D42B664334E1142A_NewsAndroidcode03.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.maribethmedia.archery.apk</td>
</tr>
<tr>
<td>SHA 256:</td>
<td>213e042b3d5b489467c5a461ffdd2e38edaa0c74957f0b1a0708027e66080890</td>
</tr>
<tr>
<td>Malware:</td>
<td>56033daef6a020d8e64729acab103f818.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.maribethmedia.killingtime.apk</td>
</tr>
<tr>
<td>SHA 256:</td>
<td>213e042b3d5b489467c5a461ffdd2e38edaa0c74957f0b1a0708027e66080890</td>
</tr>
<tr>
<td>Malware:</td>
<td>56033daef6a020d8e64729acab103f818.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.monapps.ark.three.apk</td>
</tr>
<tr>
<td>Old URL:</td>
<td><a href="https://play.google.com/store/apps/details?id=com.monapps.ark.three">https://play.google.com/store/apps/details?id=com.monapps.ark.three</a></td>
</tr>
<tr>
<td>SHA 256:</td>
<td>5b6402cc7e2e37271ee14e907e58c289c280cd71391b28807286f0393c124486</td>
</tr>
<tr>
<td>Malware:</td>
<td>ThreatJapan_4C937667CB23E857D42B664334E1142A_NewsAndroidcode03.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>com.sharamobi.h2d.fruits / lol / manga / tattoootive.apk</td>
</tr>
<tr>
<td>Old URL:</td>
<td><a href="https://play.google.com/store/apps/details?id=com.sharamobi.h2d.fruits">https://play.google.com/store/apps/details?id=com.sharamobi.h2d.fruits</a> / lol / manga / tattoootive</td>
</tr>
<tr>
<td>SHA 256:</td>
<td>5b6402cc7e2e37271ee14e907e58c289c280cd71391b28807286f0393c124486</td>
</tr>
<tr>
<td>Malware:</td>
<td>ThreatJapan_4C937667CB23E857D42B664334E1142A_NewsAndroidcode03.apk</td>
</tr>
<tr>
<td>App name:</td>
<td>far.msword.ui.apk</td>
</tr>
<tr>
<td>Old URL:</td>
<td><a href="https://play.google.com/store/apps/details?id=far.msword.ui">https://play.google.com/store/apps/details?id=far.msword.ui</a></td>
</tr>
<tr>
<td>SHA 256:</td>
<td>48f7edc18cadc12914bb89e91336b988513d4151a9ed1975f6456e795163583</td>
</tr>
<tr>
<td>Malware:</td>
<td>856CAB6FEO28C9B6229F5D5305D0E98B.apk</td>
</tr>
</tbody>
</table>

Table A.2: Malware apps removed from the market.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>HTC Desire</td>
</tr>
<tr>
<td>CPU</td>
<td>ARMv7 Processor rev 2(v7l), 1GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Adreno 200 (AMD Z430)</td>
</tr>
<tr>
<td>RAM</td>
<td>512 MB</td>
</tr>
<tr>
<td>Storage</td>
<td>405MB built-in</td>
</tr>
<tr>
<td>SD card</td>
<td>2GB Micro SD</td>
</tr>
<tr>
<td>OS</td>
<td>Android 2.3.7 Gingerbread</td>
</tr>
<tr>
<td>ROM</td>
<td>CyanogenMod-7.2.0.1-bravo</td>
</tr>
<tr>
<td>MISC</td>
<td>A-GPS, Micro USB, Camera, Bluetooth 2.1, Wifi 802.11</td>
</tr>
</tbody>
</table>

Table A.3: Technical specifications for HTC Desire test smartphone.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Sony Xperia Tablet SGPT121</td>
</tr>
<tr>
<td>CPU</td>
<td>Nvidia Tegra 3 Quad-core, 1.7 GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>OnBoard Graphic</td>
</tr>
<tr>
<td>RAM</td>
<td>1024 MB</td>
</tr>
<tr>
<td>Storage</td>
<td>16GB built-in</td>
</tr>
<tr>
<td>SD card</td>
<td>None present</td>
</tr>
<tr>
<td>OS</td>
<td>Android 4.1.1 Ice Cream Sandwich</td>
</tr>
<tr>
<td>ROM</td>
<td>Sony proprietary firmware</td>
</tr>
<tr>
<td>MISC</td>
<td>A-GPS, USB, Camera, Bluetooth 3.0, Wifi 802.11</td>
</tr>
</tbody>
</table>

Table A.4: Technical specifications for Sony Xperia test tablet.