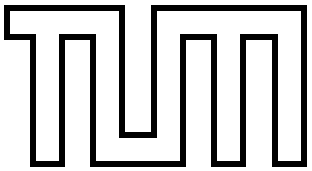


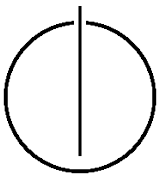
FAKULTÄT FÜR INFORMATIK

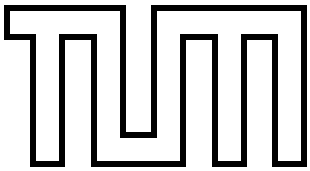
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Computer Science

Automated De-Obfuscation of Android Bytecode

Hannes Schulz





FAKULTÄT FÜR INFORMATIK

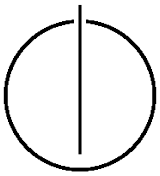
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Computer Science

Automated De-Obfuscation of Android Bytecode

Automatisierte Deobfuskierung von Android Bytecode

Author:	Hannes Schulz
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Dennis Titze
	Dr. Julian Schütte
	Thomas Kittel
Date:	July 15, 2014



I assure the single handed composition of this bachelor thesis only supported by declared resources.

Munich, July 15, 2014 Hannes Schulz

Abstract

This master's thesis examines the feasibility of automating bytecode deobfuscation for Android applications. It is motivated by the increasing obfuscation in Android applications and the inability to analyze or to recover their source code.
The terms obfuscation and deobfuscation are defined and a historic research is conducted. Currently existing obfuscation techniques as well as different analyzing methods are presented and evaluated. Derived from research and manual code analysis, approaches for their deobfuscation are discussed in theory.
Based on this knowledge a framework was built to automatically reverse the three core obfuscation techniques of the commercial obfuscator DexGuard. To avoid additional obfuscation mechanisms, the framework refrains from translating the Android applications into a high-level programming language.
The framework successfully reintroduces meaningful identifiers, is able to decrypt encrypted embedded data, and resolves important reflected calls. Malware scanners incapable of dealing with obfuscation could be used further in collaboration with this framework.

Contents

1 Introduction and Motivation

This chapter explains how Android applications are built. Obfuscation and the reasons for its stakeholders to use it are introduced. Followed by a short definition of deobfuscation and the motivation to build an automated deobfuscator. At the end of this chapter the outline will provide a summary of the chapters of this thesis.

1.1 Introduction

Android is a Linux based operating system, designed for mobile devices, released under open source licenses by Google.
According to its market share [27, 28] as well as in terms of "installed base" (the number and proportion of devices in use) [26], Android is the leading operating system in the mobile phone and tablet market.
To gain more features and functionality, third-party applications can be installed. They are available for free as well as commercial. They can be obtained from several markets or installed manually. Usually they are digitally distributed through Google's "Play Store" that contains over one million applications [6]. This popularity makes Android an interesting target for any developer.
The applications are primarily written in the programming language Java, extensions in C or C++ are also possible. A software development kit (SDK), using the eclipse integrated development environment (IDE) containing a debugger, software libraries and an emulator, is freely available.
The Dalvik virtual machine (DVM) is used to run those applications on Android. Before applications can be used with the DVM the Java code needs to be compiled into bytecode (the instruction set of the DVM). The idea of creating portable and platform independent applications in the Java language results in a well structured and easy analyzable bytecode. Because of Java's type safety and the architecture-neutral class files, bytecode compiled with the tools of the official SDK is very easy to decompile [53, 38].
In order to avoid the decompilation of their applications developers use obfuscators. An obfuscator renders software unintelligible to humans while to the DVM it is indifferent.
There are two groups of stakeholders. The first group the is using obfuscation for the protection of private property, since it is quite easy to reverse engineer compiled applications. Algorithms embedded in the application can be stolen and used to create cheap spin-offs of successful applications. Other reasons could be to prevent piracy through removal of license checks, to keep players from cheating by code manipulation, or to protect the application from the injection of malicious code.
The second group are the authors of malware. They use obfuscators to render their "evil code" (malicious, vulnerable, intentionally bad) unreadable in order to be harder to detect.
Obfuscators are available for free, commercial and of course it is possible to write a custom

one. The Android SDK for instance already includes the free software ProGuard [19]. It has only limited features like the renaming of identifiers and the removal of debug code but it can easily be activated. The commercial version of ProGuard – DexGuard – has advanced features like encryption and reflection.
The reversal of such an obfuscation process is called deobfuscation.
To automate the deobfuscation a deobfuscator was build as a part of this thesis.

1.2 Motivation

Several reasons that motivate and justify the existence and creation of such a deobfuscator are presented now.
A legitimate case to make use of a deobfuscator is the loss of source code of a valuable application where the developer is no longer available. Rewriting the application could take too much time for more complex applications or it could be too expensive. The recovery of the source code might be inevitable to update or to patch security flaws in the code of the program.
The obfuscator itself could have errors that lead to corrupt applications. Without a deobfuscator it is not verifiable if the obfuscated code still precisely does what it did before the obfuscation. Methods could have been altered in a defective way. For larger applications it takes time to manually verify all functions after each obfuscation.
In case of weaknesses in the obfuscator their users are given a false sense of security [35].
It is also hard to detect if somebody tampered with the obfuscation software on purpose. Somebody could send an email to a developer about an alleged update with a link to a modified version of an obfuscator. Each application obfuscated with this obfuscator could carry a malicious payload.
Authors of malware and malicious code use obfuscation as well to hide the true intention of their applications. There are two kinds of applications that contain malicious code. First, those which are written as dedicated malware applications to fulfill evil purposes. And second, existing (and mostly popular) applications injected with malicious payloads. Both are hard to detect in their obfuscated form.
Obfuscation could also be used to cover up piracy. If the whole application is copied and obfuscated it could still be possible to detect the fraud because of the equal designs. But a replication of the core algorithms, wrapped up in a new application that is obfuscated is much harder to detect.
Automated application analysis tools could fail because of obfuscation. Genuine applications could be detected as false positives [45] and marked as pirated. While malicious software could generate false negatives as the analysis software fails due to special obfuscation mechanisms.
Deobfuscators are also needed to repair altered code of obfuscation techniques that depend on flaws in existing tools. They exploit vulnerabilities to crash them if they assume a correct behavior of the application. Code conversion tools are a target for instance. They convert the applications bytecode to Java source code for easier inspection. Other frameworks might be based such tools and will therefore no longer work.
Obfuscators abuse language discrepancies between the Java programming language and bytecode instructions. This techniques could render applications unusable as unused com-

mands are getting deployed for instance.
And finally could deobfucators help to study obfuscation and to find ways to improve its concepts.
Manual inspection of obfuscated code is hard since it is mostly unreadable for the human analyst. And it might be impossible for any automated analyzing framework that is unable to deal with obfuscated code. While the rising number of available applications make an automated deobfuscation tool even more preferable.
Especially the advent of more and more complexity in the obfuscation of malicious applications are the driving force of this thesis.
Two examples of such obfuscated malware applications are the Android Trojans "Backdoor.AndroidOS.Obad.a" [25] (Obad) and "Android.iBanking" [14] (iBanking).
Both have implemented measures to evade the conversion to Java source code. Their methods and classes are renamed with meaningless names and most strings are encrypted in the source code. The iBanking Trojan even has measurements embedded in the code to avoid sandbox environments in which such applications are usually analyzed.
The obfuscation is not only used to cover up the intentions of the application (like data- or identity theft) but also to hide other exploits with the aim of gaining even more access to the device (like root privileges to avoid removal or file access). Because of the known language discrepancy and their exploits, the targeted language of the framework developed as part of this thesis is the Dalvik bytecode.
There are many tools and frameworks that may profit from this work. Analysis tools and frameworks that are unable to overcome obfuscated code can use this framework. This way they do not have to deal with deobfuscation but can still be used.

1.3 Outline

The next Chapter provides a more detailed definition of the terms obfuscation and deobfuscation. A taxonomy for obfuscation techniques is presented and the history of obfuscation and deobfuscation is reviewed. Other scientific approaches to automate the deobfuscation of bytecode and related work is discussed as well. At the end of the chapter further details about Java bytecode and the Android architecture are provided.
Static and dynamic analysis is explained in Chapter 3 along with the advantages of a combined approach. In addition different tools for manual inspection and automation of the framework are introduced.
Chapter 4 explains the different obfuscation and deobfuscation methods in theory. Each technique is classified according to the introduced taxonomy. The Chapter also discusses alternative code protection techniques as well as their applicability.
The thesis' framework is introduced in Chapter 5. It shows how some of the deobfuscation techniques of Chapter 4 are applied to the commercial obfuscator DexGuard.
The final Chapter a summarizes the thesis and presents the results of the implementation of the deobfuscation techniques in the framework. Chapter 6 also discloses limitations of the framework.

2 Background and Related Work

This chapter will provide the necessary background for a deeper insight into the techniques used in the history of obfuscation and their deobfuscation. It will start with a definition of the terms obfuscation and deobfuscation. A taxonomy will be presented to refine the techniques used in the field of obfuscation. Followed by the origins and early approaches of obfuscation as well as current related work. At the end of this chapter more details regarding Java bytecode and the Dalvik bytecode used for Android applications will be presented.
Derived from this knowledge this thesis derives methods used for the deobfuscation of Java bytecode in the subsequent chapters.

2.1 Definitions

Obfuscation is the technique where a supposedly one-way translation obscures source code. This way an easy readable and understandable program code is transformed into a complicated and for the human reader incomprehensible form.
Target for obfuscations are often proprietary algorithms or the program implementation and the details of the software components. Sometimes additional resources like pictures or layout declarations are obfuscated as well.
Obfuscated code is semantically identical to the original and has same functionality as the non-obfuscated code. For the executing machine that reads the code it makes no difference if for example an identifier is labeled `randomNumber` or just `a`.
There are also parts in software where obfuscation can not be applied. Certain fields need to be in its original form or the program is not readable to the executing machine. For instance the name that describes an Android application can not be obfuscated since it needs to be readable without executing the program.
Obfuscation can be achieved manually or it can be automated. Automation of the obfuscation of the whole application is savvy since the effort it takes to find the essential algorithms of a program is increased. If only the crucial parts are changed they might even stand out as they do not fit into the rest of the code.
The program to obfuscate an application or parts of it automatically is called ”obfuscator”. Its primary intention is to protect private property by preventing reverse engineering. It makes it less attractive cost wise and in terms of time and effort to recover the original code.
But not only regular developers have discovered obfuscation for their purposes. Cyber criminals like malware authors and hackers are using it to camouflage their malicious algorithms and to cover self incriminating information like their own server addresses. Obfuscators not only render identifiers meaningless or change the control flow (the sequence

of execution) but also utilize bugs and exploits in common analyzers to protect the code from inspection. Therefore not only humans are hindered from reading those programs but also automated analysis tools.
Deobfuscation is the process where parts or all of the original source code is restored from an obfuscated program.
For some obfuscation techniques it is not possible to reconstruct the obfuscated information. The already mentioned identifier renaming for example is not reversible. Once renamed, the original identifiers are lost.
During the research conducted for this thesis no obfuscation method has been found that was not based on the "security through obscurity" principle.
Security by obscurity attempts to establish security by keeping the design or the implementation to provide it secret [42]. This concept is usually disdained by security experts.
Collberg [38] confirms this statement: "As far as we know, there do not exist any techniques for preventing attacks by reverse engineering stronger than what is afforded by obscuring the purpose of the code".
Low [49] also claims: "[...] that code obfuscation is the most suitable technical protection technique that can be applied to portable languages like Java.".
Even if encryption is used, the key as well as the method for the encryption is always included within the application or it includes the instructions where and how to get them. Therefore deobfuscation should always be possible.

2.2 Taxonomy of Obfuscation Transformations

Definition 2.1. *Obfuscating Transformation is defined by Collberg [36] as the transformation τ of a source program P into the target program P':*

$$P \xrightarrow{\tau} P'$$

For the resulting program P' to be a legal obfuscationg transformation of P it needs to fulfill the following conditions:

- *If P fails to terminate or terminates with an error condition, then P' may or may not terminate.*
- *Otherwise, P' must terminate and produce the same output as P.*

For the classification of the different obfuscation transformations Collbergs [36] taxonomy is used. Figure 2.1 illustrates the four main targets identified for obfuscation transformations.

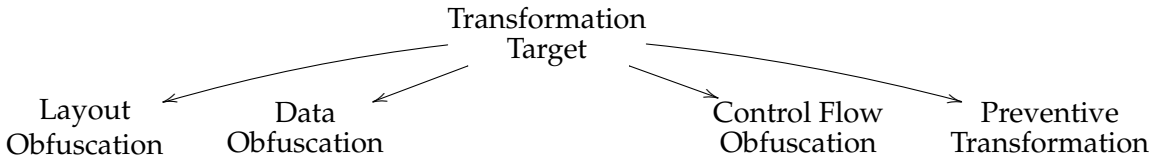


Figure 2.1: Colbergs Obfuscating Transformation Classification

Each one of them is introduced briefly in the next sections while the next chapters will

discuss concrete implementations and their reversals.
Debray [63] defines an alternative classification. He differentiates between two classes of obfuscation transformations. The first one is called "surface obfuscation". Surface obfuscations affect only the syntax of a program. They do not hide its semantic structures. When an identifier is renamed for instance, the rest of the structure of the program remains unchanged. Surface obfuscation is mainly meant to disguise the meaning of the program to the human reader. To the executing machine they are mostly transparent.
"Deep obfuscation", the second group of techniques obfuscates the structure of the program. In most cases this affects the control flow of a program. The structure gets significantly changed when for example if, for and while statements are rewritten.
It is much harder to deobfuscate this second transformations as they might also affect analysis tools.

2.2.1 Layout Obfuscation

Whenever the source and/or the binary structure of a program has been changed such that it is more difficult to understand a Layout Obfuscation was applied.
Examples for this obfuscation are:

- The scrambling of identifiers
- Changes in the formatting
- Removal of debug information
- Removal of comments

Non-alphabetical characters like mandarin characters for example are very hard to distinguish for people unaware of such characters, while for a machine it does not matter.
An implementation of identifier renaming as well as its reversal will be shown in Chapter 4.2.3.

2.2.2 Data Obfuscation

Applying data obfuscation to an application changes the structure of the data. The three subcategories of data obfuscation are depicted in Figure 2.2.

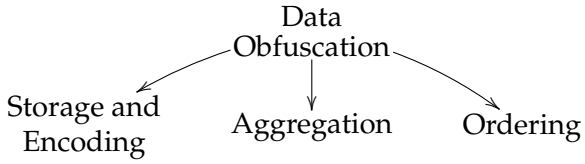


Figure 2.2: Subcategories of Data Obfuscation

Data storage and encoding is affected whenever the way objects are stored is changed. Variables like arrays or integers can be split, merged or even promoted to objects like in Listing 2.1.

Control flow flattening is one of the methods to obfuscate the control flow. It restructures the control flow logic of a program in a way that all ”basic blocks” seem to have the same predecessor and successor. A basic block is a number of code lines that only contain a single decision statement.

Listing 2.3: Sample Program

```
1  public int fill(int a, int b) {
2    int diff = 0;
3    if ( a > b ) {
4      diff = a-b;
5      do {
6        b++;
7      } while ( a != b );
8    } else {
9      diff = b-a;
10         do {
11       a++;
12     } while ( a != b )
13   }
14   return diff;
15 }
```

Figure 2.4 shows the control flow graph of Listing 2.3. In Figure 2.5 this program has been flattened and extended by a ”dispatcher variable”. The dispatcher variable is driving the execution of the different basic blocks. At the end of each basic block the dispatcher variable gets reassigned to the next block to be execution.

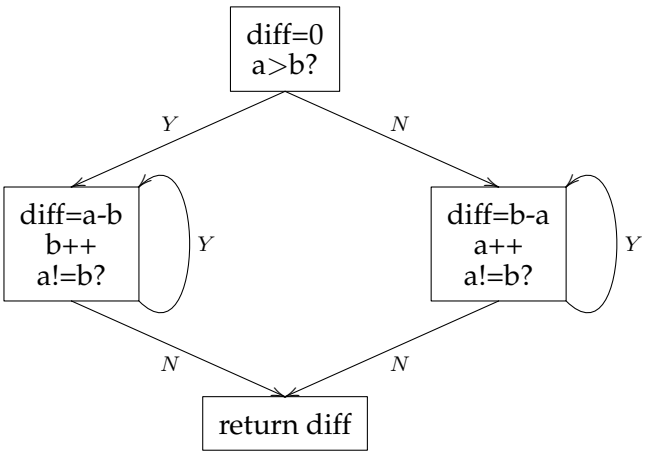


Figure 2.4: Control Flow Graph of the Sample Program 2.3

Manual inspection is harder for flattened programs since the dispatcher variable is assigned within the function. For most automated analysis tools it is most likely trivial to understand the structure and to reconstruct a compacter version of it. All the necessary information is included within the function. An intra-procedural data flow analysis (where the object of observation is a single function) is sufficient.
To advance this obfuscation several enhancements can be introduced. The complete dis-

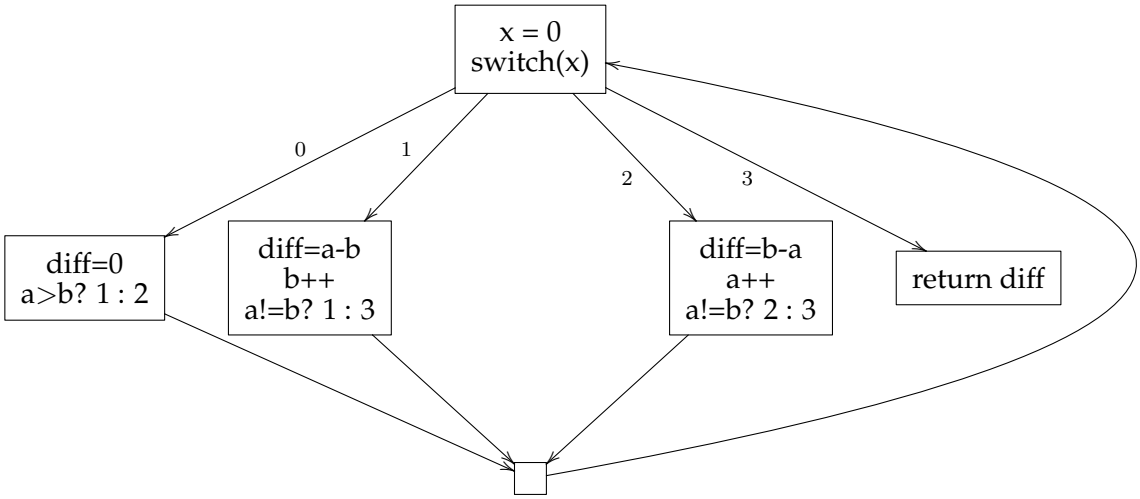


Figure 2.5: Flattened Control Flow Graph of Sample Program 2.3 with dispatcher variable

patcher variable assignment can be extracted and separated from the function. For instance they can be stored in a global array in the same class as the function and need to be passed before each call. Therefore the intra-procedural analysis is unsuitable. Now the calls and relationships among the procedures need to be analyzed. This is called an inter-procedural analysis.
To confuse even further, artificial basic blocks can be inserted to the function. The generated blocks are derivatives of existing ones but they have incorrect dispatcher variable targets. This is also known as cloning or redundant code. During the runtime of the program those blocks are never executed.
Additionally introduced invalid control flows are also called fake control flows. Opaque predicates [37] are constructs that are easy to compute for an obfuscator but very hard to predict for deobfuscators. Opaque predicates are boolean expressions which always evaluate to the same result, regardless of the used parameters. For example the code in the `if`-statement in Listing 2.4 is always executed.

Listing 2.4: Sample Program

```
1 Random rand = new Random();
2 int min = 1;
3 int max = 5;
4 int randomNumber = rand.nextInt((max - min) + 1) + min;
5 if ( randomNumber > 0 ) {
6         //code that is always executed
7 }
```

This construct can be extended arbitrarily. The complexity of the evaluated term can be increased by a mathematical function. For example:

$$e^{\sqrt{-1}*3,1415926535} - x \stackrel{?}{=} 0$$

The number of branches with decoy code blocks can be increased. And existing boolean expressions can be extended with opaque predicates.

2.2.4 Preventive Obfuscation

Preventive obfuscation is meant to stop decompilers and debuggers.
This is achieved by exploiting weaknesses and exceptions of those programs. Especially changes to the Java bytecode are eligible to let decompilers crash. There are a lot of bytecode constructs that are valid to the DVM. But they are either never generated in such a form by the compiler and therefore not recognized correctly by decompilers. Or they use constructs that are not applicable in Java byte-/sourcecode.

2.3 Early De-/obfuscation Attempts

Obfuscation of software is of interest since commercial software emerged.
Gosler [43] writes 1985 about how easily software is duplicable and how this affects profits and leads to stolen or manipulated data. In his paper he already addresses ”Software Analysis Denial” and topics like software piracy and modification.
Included is a proposal to protect software by adding a unique signature that is associated with the system. Therefore he modifies the magnetic material on the floppy disk. In the software he writes data to this area and then reads the data from this area. If the data read from the modified area matches the written data he concludes that it must be a copy.
In order to protect the signature check he proposed several technical measures to make the analysis more difficult.
Amongst them are the use of checksums that protect certain areas in the software. If those areas are changed the strored and the calculated checksums doesn't match any more. He also proposes the use of encryption by public key cryptography where the application only holds the public key. Any change in the application requires to first identify the decryption key and method. Then the code needs to be decrypted, modified and re-implemented without the decryption routines since it is not possible to encrypt the modified code without the private key.
In his ”Technological Denial Concepts” he goes one step further and proposes a penalty system that is activated each time a code penetration like a checksum violation is detected. As a penalty he proposes: ”[...] anything from destroying critical system components that would disallow further testing[...]” or ”[...]to subtly altering the system in such a way as to provide disinformation[...]”.
The Authors of malware are using obfuscation as well. By changing the appearance of the code of a program they try to elude detection.
The ”Brain” virus [22, 57, 58] was one of the first viruses that tried to hide its existence. When reading the infected sectors it reflected perfectly save data. It was written in 1986.
In the same year the virus ”Cascade” was detected. It was the first virus that made use of a ”decryptor” based on a XOR cipher. The decryptor is an encryption and decryption routine.
The only detectable constant of the virus was the decryptor. As a consequence of the simplicity of the XOR cipher it was very small. Since Anti-Virus software was only capable of simple pattern matching at that time it was difficult to detect the decryptor. The small pattern would have raised a lot of false positives. The later approach of ”signature matching”

was successful. Signature matching is the search for a fixed string of bytes within the data [40].
The next generation of malicious software used a technique called ”Oligomorphism” [68] and was first spotted in late 1990 . It was able to choose between several re-encryption methods. Every time it reproduced, it chose a different one randomly. Other Oligomorph malware programs were even able to generate decryption methods dynamically. This made signature based detection much harder but it was still possible.
In 1991 Solomon [62] defined the term ”Polymorphism”. Polymorphism is the ability of a programs to create many different versions of itself. They have almost no similarities in the crypto cipher signature as they spread. The ”Vienna” virus was the first polymorphic virus that has been found. It was written by Washburn [9] to prove signature detection is no longer sufficient. His code was able to generate an infinite number of different decryptors by inserting ”junk” instructions – machine instructions without any function – in order to create a version of a program with a different appearance [50]. Anti-Virus software was forced to use sandboxes approaches to find similarities and suspicious behavior. Detection was performed by matching the memory image of the program against known signatures.
The Regswap [65] virus, first found in 1998, no longer used polymorphic decryptors. Regswap used a technique called ”Register Reassignment” instead where the registers are changed for each generation of a program. Signature matching algorithms were therefore no longer sufficient. This type of malicious software is called ”Metamorphism”, meaning self shaping or self forming. It reprograms itself as it spreads. There are many more transformations known [54] including:

- Garbage Insertion - see above
- Register Swapping - see above
- Code Substitution - where instructions are exchanged with equivalent ones
- Subroutine Reordering - similar to call graph obfuscation – often used in current malicious code [33]
- Code Insertion - The code inserts itself into existing binary code

Obfuscation can also be some kind of sport.
The focus of the International Obfuscated C Code Contest (IOCCC) is creative obfuscation. The code in Listing 2.5, written by one of their winners, is an example for such a program. Its appearance was intentionally formed to look like pi – obfuscating its real purpose – the calculation of e.

Listing 2.5: Calculator for e Obfuscated and Formed as PI [23]

```
                                                                    char
                                                                  _3141592654[3141
          ],__3141[3141];_314159[31415],_3141[31415];main(){register char*
       _3_141,*_3_1415, *_3__1415; register int _314,_31415,__31415,*_31,
     _3_14159,__3_1415;*_3141592654=__31415=2,_3141592654[0][_3141592654
    -1]=1[__3141]=5;__3_1415=1;do{_3_14159=_314=0,__31415++;for(_31415
   =0;_31415<(3,14-4)*__31415;_31415++)_31415[_3141]=_314159[_31415]=-
 1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
__3_1415     +__3141;for                (_31415 = 3141-
             __3_1415   ;               _31415;_31415--
             ,_3_141 ++,                _3_1415++){_314
            +=_314<<2 ;                 _314<<=1;_314+=
           *_3_1415;_31                  =_314159+_314;
           if(!(*_31+1)                  )* _31 =_314 /
           __31415,_314                  [_3141]=_314 %
           __31415 ;* (                  _3__1415=_3_141
          )+= *_3_1415                    = *_31;while(*
          _3__1415 >=                     31415/3141 ) *
          _3__1415+= -                    10,(*--_3__1415
         )++;_314=_314                    [_3141]; if ( !
         _3_14159 && *                    _3_1415)_3_14159
         =1,__3_1415 =                    3141-_31415;}if(
         _314+(__31415                     >>1)>=__31415 )
         while ( ++ *                      _3_141==3141/314
        )*_3_141--=0                       ;}while(_3_14159
        ) ; { char *                       __3_14= "3.1415";
        write((3,1),                       (--*__3_14,__3_14
        ),(_3_14159                         ++,++_3_14159))+
       3.1415926; }                         for ( _31415 = 1;
      _31415<3141-                          1;_31415++)write(
     31415% 314-(                           3,14),_3141592654[
   _31415     ] +                           "0123456789","314"
   [ 3]+1)-_314;                            puts((*_3141592654=0
,_3141592654))                               ;_314= *"3.141592";}
```

2.4 Scientific Approaches and Related Work

Related scientific work in the field of Dalvik bytecode deobfuscation does, as far as this thesis could conclude, not exist.

However, in the field of obfuscation a lot of research has been done since the early 2000's . Wroblewski [67] said in 2002 about program code obfuscation: "Scientific approach to the problem of program code obfuscation is so young, that it is hard to say about its history. The first trials of systematic research connected with obfuscation are dated in the late 1990." He also pointed out Java as one of the driving factors for obfuscation because the triviality to transform compiled form to source code.

Collbergs [36] taxonomy of obfuscation techniques was helpful for classification.

Most work related to deobfuscation has either be done in general and only for single techniques or as proof of concept for an obfuscation technique.

In 2005 Chandrasekharan and Debray [35] presented a deobfuscation paper that showed how easily a combination of static and dynamic analysis can defeat obfuscation. In their study they targeted a few control flow obfuscation techniques and concluded their approach could be effective against other deobfuscation heuristics. Java was not the focus of their research though. They also stated in matters of related work: "We are not aware of any prior work on reverse engineering obfuscated code."

As reflection was recognized as a problem for any static analyzer, related work in this field is available.

In 2005 Livshits [48] et al. present an algorithm to resolve reflection in Java programs.

Bodden [31] et al. write 2011 about the problematic of static analyzers and how they can not overcome reflection and custom class loaders in Java programs. As a solution they

present a tool chain to partially solve the problem.
Wognsen's [66] thesis from 2012 about "Static Analysis of Dalvik Bytecode and Reflection in Android" uses static analysis to approximate program behavior to scan for malicious code in Android applications. Due to the heavy use of reflection in Android he also supports the static analysis of reflective calls. This is the most releated work since Wognsen also works with Dalvik bytecode to solve reflection.
There is a lot of other work that is affected by obfuscation. It is concerned with analyzing and/or protecting Java/Android applications and has to deal with obfuscation. Although they are not directly related to deobfuscation techniques they point out the urge of the availability of deobfuscators.
The analysis of Android applications is often researched. The bulk of them is concerned with the detection of malicious applications or malicious code within the applications. The bigger part though relies on techniques to decompile the Dalvik bytecode to Java bytecode that in some cases is even translated to Java source code for the inspection [31]. Again, sooner or later, most of them will face the problem of Control Flow Obfuscation techniques.
Yet others have already identified the problematic and are capable of dealing with dex bytecode [39].
Obfuscation has also been turned out to be problematic with repacked applications.
Repacked applications are in most cases popular applications that have been altered with a malicious payload or a backdoor. Obfuscation complicates the comparison of different versions of the same program as they might look different each time they are obfuscated. To prevent this Zhou [69] proposes a detection mechanism based on an embedded watermark on top of obfuscated applications. Although this could help to fight unauthorized repackaging of applications it can't guarantee the absence of malicious code.
Deobfuscation could make a comparison more easy and make a scan of the application for flaws possible.
By injecting a service into an arbitrary Android application Cauquil [32] presents a dynamic analysis that is based on detecting malicious behavior by observation. The upside of this is that the code does not need to be deobfuscated. The downside is that potential malicious code needs to be executed and that some of the malicious code might not be reached during the analysis and therefore will not be discovered. Additionally their tool seems to be in an early state and the paper does not state if and how any source code can be deobfuscated.

2.5 Java Bytecode

Java [24] is the programming language in which Android applications usually are written. To build an executable, Java source code is translated into Java class file by a compiler. For each class defined in the source code one single class file is generated. Class files contain bytecode as well as class related data like for example the constant pool that holds literal (fixed) constant values. Listing 2.6 shows the source code for a Java "Hello World!" program that has been compiled into bytecode as shown in Listing 2.7. The "Hello World!" string is stored in the constant pool.

Listing 2.6: "Hello World!" in Java Source Code

```
1 public class HelloWorld {
2         static String myString = "Hello World!";
3         public static void main(String[] args) {
4                 System.out.println(myString);
5         }
6 }
```

Listing 2.7: Java Bytecode of the "Hello World!" Program Disassembled with Javac (from the Java Development Kit).

```
1  public class HelloWorld {
2    static java.lang.String myString;
3
4    static {};
5      Code:
6         0: ldc           #10  // String Hello World!
7         2: putstatic     #12  // Field myString:Ljava/lang/String;
8         5: return
9
10   public HelloWorld();
11     Code:
12        0: aload_0
13        1: invokespecial #17  // Method java/lang/Object."<init>":()V
14        4: return
15
16   public static void main(java.lang.String[]);
17     Code:
18        0: getstatic     #23  // Field
             java/lang/System.out:Ljava/io/PrintStream;
19        3: getstatic     #12  // Field myString:Ljava/lang/String;
20        6: invokevirtual #29  // Method
             java/io/PrintStream.println:(Ljava/lang/String;)V
21        9: return
22 }
```

The class files are also the files that are distributed when the application is published. The virtual machine is the high level abstraction that emulates a physical machine and enables Java applications to be platform independent. The interpreter, usually the JVM, loads and executes those class files dynamically. Before the installation and execution they are loaded by the class loader and verified by the "bytecode verifier". The bytecode verification is validating if the compiler has produced valid bytecode. This step is necessary because the compiler could have been a custom one that produced errors. It could also have been intentionally tampered with or simple defective because of a broken file.
The "Class Linker" is responsible for adding the classes to the runtime system.
All necessary steps to execute a class file in the JVM are shown in Figure 2.6.
The "Hello World!" example already visualizes the high amount of source code information included within the class file. Since the `myString` variable was static even its variable name was left in the constant pool. In fact with the help of a decompiler the exact same source code as in Listing 2.6 can be recovered.

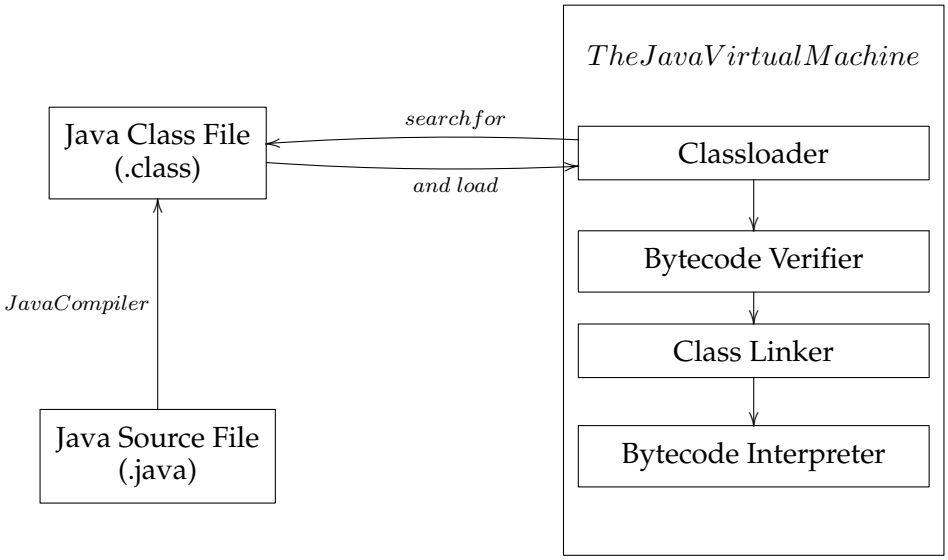


Figure 2.6: The Java Virtual Machine Loading a Class File

Because of the common used libraries the Java applications are relatively small which makes analyzing it even more easy [36].
Proebsting [53] even claims that for the recovery of source code from Java bytecode "fewer than a dozen simple code-rewriting rules reconstruct the high-level statements".

2.6 The Android Architecture

Instead of the Java Virtual Machine, Android is using the Dalvik Virtual Machine.
From an application level this makes only a small difference. In terms of architecture they

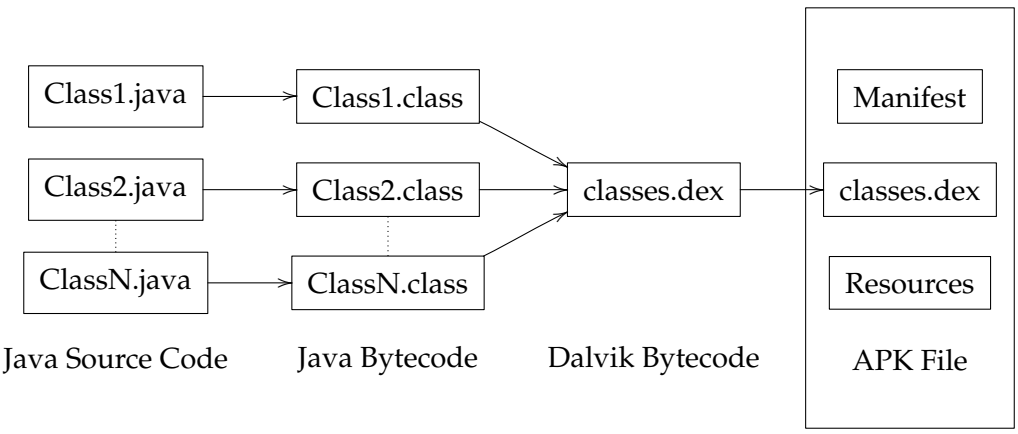


Figure 2.7: Creation of an APK File

use different approaches to store and retrieve operands and their results.
The JVM is a stack based and the DVM a register based machine [61]. Stack based machines use LIFO (last is first out) stacks where "PUSH" and "POP" operations are used to

carry out operations. Register based machines operands are stored in registers that need to be explicitly addressed.
Java source code files are translated into Java Bytecode with the help of the Java Compiler. In order to create DVM executable bytecode an additional intermediate step is necessary that converts Java bytecode into Dalvik bytecode. They are compiled into a single dex file, containing a single header, a single constant pool and all classes with their data. The program "dx" – the Dex compiler – that is part of the SDK usually takes care of this transformation. Figure 2.7 illustrates the procedure. Together with the manifest that contains information like name and version of the application, libraries, assets and other uncompiled resources, an Android application package file (APK) is created.
The "Hello World!" example from Listing 2.6 compiled with the Android Developer Tools (ADT) and disassembled with the tool Dexdump that is also part of the Android SDK looks like Listing 2.8.
The dexdump is not really pleasant to read and there is no tool to convert the changed output back into a `classes.dex` file.
Therefore this thesis will make use the smali assembler/disassembler [21]. Its syntax is based on Jasmin [15]. Jasmin was originally developed to write "simple assembler-like syntax" that can be converted to Java bytecode files.
Smali is a disassembler, no decompiler. This means the translated code consists of Dalvik opcodes [52] mixed with Jasmin syntax. The smali disassembly of the "Hello World!" application is shown in Listing 2.9.
Although there are many existing analysis programs for Java source code and it is very tempting to convert the Dalvik bytecode such that those programs can be used, this is not feasible because of preventive obfuscation. Therefore the framework developed as part of this thesis works with smali disassembled Dalvik bytecode.

Listing 2.8: Dalvik Bytecode of the "Hello World!" Program Disassembled with Dexdump

```
1  Class #253            -
2    Class descriptor  : 'Lcom/homework/messenger/HelloWorld;'
3    Access flags      : 0x0001 (PUBLIC)
4    Superclass        : 'Ljava/lang/Object;'
5    Interfaces        -
6    Static fields     -
7      #0              : (in Lcom/homework/messenger/HelloWorld;)
8        name          : 'myString'
9        type          : 'Ljava/lang/String;'
10       access        : 0x0008 (STATIC)
11   Instance fields   -
12   Direct methods    -
13     #0              : (in Lcom/homework/messenger/HelloWorld;)
14       name          : '<clinit>'
15       type          : '()V'
16       access        : 0x0008 (STATIC)
17       code          -
18       registers     : 1
19       ins           : 0
20       outs          : 0
21       insns size    : 5 16-bit code units
22       catches       : (none)
23       positions     :
24         0x0000 line=3
25       locals        :
26     #1              : (in Lcom/homework/messenger/HelloWorld;)
27       name          : '<init>'
28       type          : '()V'
29       access        : 0x10001 (PUBLIC CONSTRUCTOR)
30       code          -
31       registers     : 1
32       ins           : 1
33       outs          : 1
34       insns size    : 4 16-bit code units
35       catches       : (none)
36       positions     :
37         0x0000 line=2
38       locals        :
39         0x0000 - 0x0004 reg=0 this Lcom/homework/messenger/HelloWorld;
40     #2              : (in Lcom/homework/messenger/HelloWorld;)
41       name          : 'main'
42       type          : '([Ljava/lang/String;)V'
43       access        : 0x0009 (PUBLIC STATIC)
44       code          -
45       registers     : 3
46       ins           : 1
47       outs          : 2
48       insns size    : 8 16-bit code units
49       catches       : (none)
50       positions     :
51         0x0000 line=5
52         0x0007 line=6
53       locals        :
54         0x0000 - 0x0008 reg=2 args [Ljava/lang/String;
55   Virtual methods   -
56   source_file_idx   : 536 (HelloWorld.java)
```

Listing 2.9: Dalvik Bytecode of the "Hello World!" Program Disassembled with Backsmali

```
# static fields
.field static myString:Ljava/lang/String;

# direct methods
.method static <clinit>()V
    .registers 1

    .line 0
    const-string v0, "Hello World!"

    sput-object v0, Lcom/homework/messenger/HelloWorld;->myString:Ljava/lang/String;

    return-void
.end method

.method public constructor <init>()V
    .registers 1

    .line 0
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 3
    .param p0, "args"    # [Ljava/lang/String;

    .line 0
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;

    sget-object v1, Lcom/homework/messenger/HelloWorld;->myString:Ljava/lang/String;

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V

    .line 6
    return-void
.end method
```

3 Analysis Methods and Tools

To analyze a program there are two categories of tools [47, 66]. Both can be applied to disassembled machine code instructions.
The first one is static analysis where a program is analyzed without executing it.
The second method is the dynamic analysis where the program is executed and analyzed during runtime. This can be done with different inputs to trigger different behaviors. Based on the inputs the executed instructions are identified and their results are evaluated.
A combination of both methods is also possible and can have a complementary effect [41].
Since the algorithms of the static analysis are also part of disassemblers and decompilers they are presented at the end of this chapter. The tools are partially used for the deobfuscation framework of this thesis.

3.1 Static Analysis

The static analysis only reads the code once.
There are two different general approaches to the static analysis.
First, the "linear sweep". The algorithm of the linear sweep reads a program from its first instruction to the last. As it encounters them it processes them one by one. The linear sweep is also used by the DVM's bytecode verifier.
And second, "recursive traversal". The recursive traversal algorithm also takes into account the control flow for each instruction encountered. This means for branch instructions or function calls, the analysis continues with their possible successor instructions.
Each instruction visited gets marked and will not be analyzed again by the algorithm.
There are however some limitations to the static analysis. The linear sweep for example can not identify dead code instructions. It is also difficult to resolve reflection calls, especially if they are nested. And a combination [54] of obfuscation techniques for example can make it impossible to interpret the instructions correctly.

3.2 Dynamic Analysis

The dynamic analysis executes the code for the inspection of the called instructions.
This can be done in a lab environment on a dedicated physical device (possibly in a sandbox) or within a virtual environment. The behavior of the application as well as its interaction with system- and other components like the network interface is monitored.
Dynamic analysis only reaches the parts of the code that are actually executed during runtime. This can be used to analyze techniques that rely on control flow obfuscation since it will never reach branches with decoy code blocks for example. But it might be necessary to process the same code multiple times with different inputs to reach a high code coverage.

Other than with the static analysis, reflection calls are for instance simply executed and do not need any interpretation from the analyzer.

3.3 Combined Analysis

A combination of the two techniques is also possible and can result in a more efficient solution [41]. In a first step a static analysis could examine a program for certain methods or signatures and collect parameters. While in a second step a dynamic analysis runs the identified methods with those collected parameters and analyzes the results.

3.4 Tools

For the Framework developed as a part of this thesis as well as for the manual inspection of Android applications multiple tools have been used that are based on the analysis methods introduced in this chapter.
Dex2jar [10] converts Dalvik executables to Java class files. To disassemble the Dalvik bytecode dex2jar uses a static analyzing algorithm. It is not equipped with deobfuscation routines and therefore limited in dealing with obfuscated code.
JD-GUI [16], from the "Java Decompiler Project" is a decompiler and viewer for Java class files. It also uses a static analyzing algorithm to convert the Java bytecode.
Smali [21] is a assembler/dissasembler for dalvik bytecode. It is used in the framework of this thesis to disassemble applications before they are processed. For the disassembly it uses the static recursive traversal algorithm. It does not make any attempt to decompile nor does it deobfuscate any parts of the code.
The Android-apktool [2] is a reverse engineering tool. It uses smali to disassemble Dalvik bytecode but also has some decoding mechanisms to rebuild binary resource files like the manifest XML.

4 Obfuscation and Deobfuscation

This chapter will discuss different obfuscation methods and possible ways for deobfuscation in general.
Examples from research as well as actual used obfuscation will be given and a revision of their resilience against deobfuscation will be conducted for some of them.
Since it is crucial for the deobfuscation of an application to gain access to its code, the applicability of different approaches to either withhold the code or to provide it in different forms than Dalvik bytecode will be examined in the first section.
Encryption as an instrument of obfuscation will be discussed in a separate section.
Finally a few existing obfuscators are introduced and their capabilities are declared as far as this was possible from the vendors specifications.

4.1 Other Protection Techniques

Android applications are usually distributed in the form of APK files (see Chapter 2.6). They can be obtained and installed from different sources:

- The Google Play store
- Third party markets (for instance from different the phones manufacturers)
- Directly as APK files

APK files, for instance from a developer forum, can be processed immediately. After they have been downloaded and installed on the device, the APK file of applications from other origins can be gained physical access to by pulling them from a Android device using the Android Developer Tools (ADT).
The contents of the APK container file can easily be extracted to recover the Dalvik bytecode stored within the `classes.dex` file.
To hide the code or to avoid the inclusion of the most valuable parts within the application (like used algorithms) they can be outsourced. The possibilities to do so are subject of following discussion.

4.1.1 Dynamic Loading of Code

In the x86 architecture, viruses use a decryptor to hide its program code in an encrypted form within the code. If needed, the decryptor decrypts the whole program or parts of it at runtime.
A similar method can be used in Android by using a custom class loader [31, 55]. It needs to make use of the ”DexFile” [11] class that is part of the Android API. For the class loader to be able to load code, it needs to be in the form of a dex file.

The dex file needs to be inside a compressed archive that can be stored the following locations:

1. As an additional file within the APK.
2. It can be downloaded from an external source like the internet.
3. Or within a class, using a byte array

Nested archives are also possible with the custom class loader.
For static analysis, downloaded code is especially problematic since its code is completely invisible to the the analyzing tool [31].
Independent of the location where the code is received from it could additionally be encrypted to avoid detection and to make recovery more difficult. Therefore an additional step could be required to decrypt the data. Encryption is covered in Chapter 4.3.
As the DEX file is received and decrypted, its contents can be loaded by the class loader.
The first two approaches have the disadvantage that the DVM stores an optimized dex file on the devices file system once the code is executed. At this stage the file already needs to be decrypted and can easily be copied for further analysis.
The third approach is more complicated since it needs to access a private function of the DexFile class via the JNI to be able to directly work with the byte array. The exact procedure is explained in [60]. For this method neither the dex file needs to be stored to the file system nor is an optimizied dex file created. Instead the data is only existing in memory.
A possible procedure to deobfuscate this code could look like this:

1. Detection of code containing byte array
2. Decryption of the received byte array if needed
3. Saving of the code into a file
4. Extraction of the file to gain access to the dex file
5. Disassembling of the dex file for further analysis
6. Reintegration of the code if needed

In step one where the code containing byte array is detected, a number of factors could be considered to speed up the search:

- Is there any sign of a custom class loader within the application?
- Is the array of a size that could hold a class (i.e. not to small)?
- Is a method with a decryption signature included in the class?
- After decryption does the array start with a certain sequence?

4.1.2 Client-Server Model

In the client-server model an application installed on the device is only equipped with proprietary program logic. Therefore no significant source code is included within the code of the deployed application.
Instead it is equipped with a number of interfaces to communicate with a remote server. Each time new data is required, the application sends a query to the server equipped with the valuable algorithms or data. The server calculates the result for the query and sends it back to the client. At no time has the application any access to the algorithms used.
Situations where this is can even be necessary are those where the amount of data or the computing power exceeds the capabilities of a mobile device. Some navigation systems, Google earth [12] showing the world in 3D or weather forecast applications are some examples. Weather applications provide the current weather and a forecast to the user. To gain this information the application connects to a weather provider, downloads and presents the data to the user. The application itself has no forecast algorithm implemented.
But this model is only applicable for a limited number of applications. Applications relying on fast but simple calculations or even need to respond in real time can most of the time not rely on a client-server model. Neither can those which should or have to work without active network connection.
The main reason is the delay that is associated with each external query. The additionally produced communication overhead also results in bandwidth usage that might not be desirable by the end-user.
Besides, most developers depend on the computational capacity of the users devices. They are either not able or willing to finance those servers to support a client-server model as they might render their effort unprofitable.
In face of this restrictions it seems even harder to integrate a client-server model into an automated obfuscator. And there are even more open questions:

- How does the obfuscator identify the important algorithms?
- Where to outsource the identified code?
- Under whose authority is the outsourced location/data?
- Who is responsible in case of a breach?
- How to deal with different versions of the algorithm/application?

4.1.3 Native Code

With the Native Development Kit (NDK) [3] parts of an application can be written in the programming languages C and C++. This is possible because of Androids architecture that is based on a Linux kernel. Using the Java Native Interface (JNI), native code can be called from within the JVM.
This can be used to outsource parts of an application programmed in another language. Obfuscation can than be applied to this code as well to add an additional layer of obfuscation. Although the obfuscation techniques for other languages are similar they have different implementations as the ones described in this chapter.

It would be beyond the scope of this thesis to describe the deobfuscation techniques for those languages as well.

4.2 Bytecode Obfuscation and Deobfuscation

Different obfuscation techniques are introduced and classified in this section with respect to the obfuscation transformations of Chapter 2.2.

4.2.1 Anti Debug

Anti Debugging is a Preventive Obfuscation.
By inserting and executing a piece of code similar to the one in Listing 4.1 the application stops its execution if the presence of a debugger is detected. This can be checked at the start of the application or regularly for instance at the beginning of each important method.
Other than in the example would a real obfuscator of course not implement a clue like the debug message in line two.

Listing 4.1: Simple Debug Detection

```
1 if( android.os.Debug.isDebuggerConnected() ) {
2   Log.e(mTag, "Debugger detected, exit program!");
3   android.os.Process.killProcess( android.os.Process.myPid() );
4 }
```

Similar like newer viruses on the PC platform, malware applications also try to detect and evade [64] sandbox environments and emulators.
Therefore they explore their runtime environment before they start their actual application routines. Several values can be queried from the Android API and used to detect emulators since they are all initialized with default values:

- The International Mobile Station Equipment Identity (IMEI), a unique 15 digit number used to identify a mobile phone device
- The phone and voice mail number
- The SIM serial number
- The subscriber ID
- Software build information like: brand, device, manufacturer, model, product

Another characteristic of the Android emulator is the use of certain TCP ports for the Android Debug Bridge (ADB) as well as the conspicuous network address space that often used for virtual networks.
Since these values can partially be customized, Vidas [64] proposes to measure the performance of a device to detect the presence of an emulator. Emulators usually run on top

of other operating systems. The execution of applications without the phones specialized hardware plus the translation of the instructions to a different architecture makes the execution of the application much slower than on current mobile phones. Benchmarks for the CPU as well as for the graphical unit can be used to measure their performance. If they fail to meet a predefined threshold the application assumes it does not run on a real device.
The hardware components can also be detected or monitored to identify a virtual device. The hardware set of a physical device like sensors or CPU frequencies can be an indicator for a real device. The accelerometer or battery levels can also be monitored to detect emulators.
All this anti debug measures can only prevent a dynamic analysis where the application needs to be executed to trigger them. To detect them and to make deobfuscators aware, static analysis could always pre-check the code for known indicators. For the dynamic analysis either the detection or the counter measurements (like the part of the code, trying to close the application) can be removed. If this is not possible, the application can still be analyzed dynamically by the use of a physical device in a lab environment.

4.2.2 Anti Decompiling

Anti decompiling is another technology of the category preventive obfuscation.
To prevent the decompilation of Java/Dalvik bytecode to Java source code, language constructs [29, 51] only valid in bytecode can be used.
A statement that is for instance not available in Java source code is `goto` [17]. In order for microprocessors to be able to execute statements of higher level programming languages, they need to be translated by using the unconditional jump instruction `goto`.
This gap between the different languages can be exploited by injecing different `goto` statements into the bytecode.
A simple way of confusing decompilers is to use `goto` jumps in combination with opaque predicates.
Figure 4.1 shows a series of instructions that should never be executed. Because of the nested calls it is hard to detect the loop created with the `goto` statements. If the analyzer is not aware of such constructs it could either get stuck in the loop itself, crash or produce defective code.
Another language difference is given in the exception handling. The try/catch blocks usually generated by the Java compiler are well structured. They are strictly nested and do not overlap.

Listing 4.2: Overlaping Try/Catch Blocks

```
1 :label_1
2 // instructions
3 :label_2
4 // instructions
5 :label_3
6 // instructions
7 :label_4
8 .catch Ljava/lang/Exception; {:label_1 .. :label_3} :label_4
9 .catch Ljava/lang/Exception; {:label_2 .. :label_4} :label_4
```

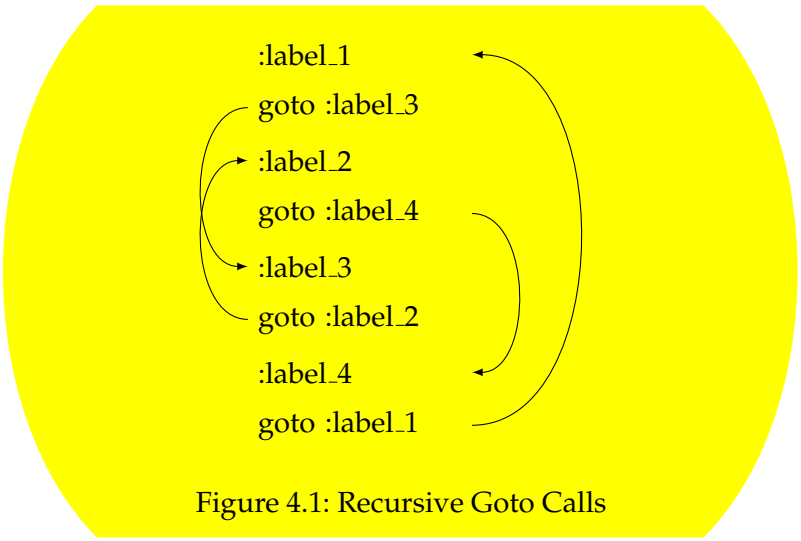


Figure 4.1: Recursive Goto Calls

Another valid bytecode construct is shown in Figure 4.2 where the exception handler is put on top of the construct. Listing 4.2 shows valid bytecode that makes use of overlapping try/catch blocks.
In the brackets of the `.catch` instruction are the labels defined that surrounded the code block this catch block covers. The last label defines the address to immediately jump to in case of an exception in the monitored code block.
While these constructs are difficult if not impossible for decompilers to handle, disassem-

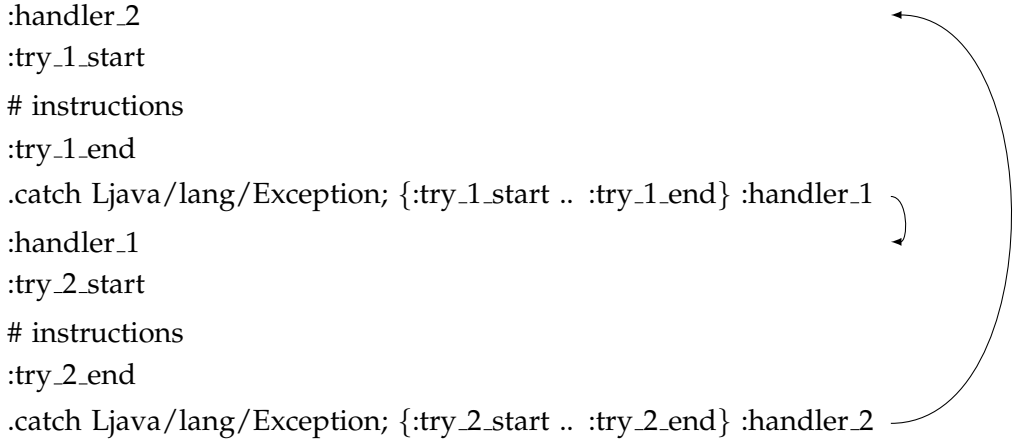


Figure 4.2: Nested Try/Catch Blocks

blers are not translating this statements into the higher level language constructs of the Java source code. Therefore they are not affected by this obfuscation technique.
Deobfuscators are also not affected as long as they work on a bytecode level.

4.2.3 Identifier Renaming

In Java names of packages, classes, methods, variables and parameters are called identifiers. In [46] Jones raises the hypothesis: "The expectation is that meaningful names will

provide information to subsequent readers of the code that will reduce the effort needed to comprehend that code." and concludes: "The impact of naming information [...] was significant.".
Naming conventions are unwritten laws that aim on increasing the readability even further by using certain standards for those identifiers.
Some examples are:

- Classes should be in "UpperCamelCase" – a notation using whole words and every word starts with an uppercase
- Methods and variables should use the "lowerCamelCase" notation where the first word starts with a lower chase and the following words use uppercase
- Constants are written in uppercase, separated by underscores

In the compiled Java bytecode the unchanged names of packages, top-level classes and interfaces, nested classes and interfaces, field and methods are used. Parameters of methods and constructors as well as local variable names are stripped from the bytecode.
Having all this information remaining in the code, a version of the Java source very close to the original code can be recovered. For the human analyst this information can be of great help to understand the algorithms of the application.
Listing 4.3 shows a piece of a structure that could have been recovered from an application. The names of the identifiers already reveal a great deal of information. Without any further knowledge it is already assumable that this is an application with a user interface, capable of Bluetooth connections, with some kind of artificial navigator for some kind of driving. Maybe a navigation system for an autonomous car or a remote control for some vehicle.

Listing 4.3: Example of Java Source Code without obfuscation

```
1  package artificialDriving {
2    public class Navigator{...}
3  }
4
5  package connectionHandler {
6    public class BluetoothHandler{...}
7  }
8
9  package userInterface {
10   public class MainActivity{...}
11 }
```

Different obfuscation approaches are trying to avoid leaving exactly those revealing identifiers in the code. They all have the renaming of those identifiers with meaningless names in common.
Listing 4.4 shows a simple yet powerful renaming mechanism. At first glance it is impossible to tell what kind of application this could be and what those classes and methods are meant to do.
Using the same identifier over and over again in all classes and for all parameters at least

once, makes differentiation very demanding for any human analyst. The help of sophisticated tools that link those identifiers to each others and jumps to- or highlights the references is almost essential. To increase the complexity, overloading is used to make the differentiation of the methods even harder.
Overloading is a concept in Java where several methods can have the same name as long as each of them takes a different set of input parameters.
Chan [34] shows how the over use of an identifier for types, fields and methods not only reduces the size of the bytecode but also claims to make recompiling impossible.

Listing 4.4: Example of Java Source Code with Simple Obfuscation

```
1  package a {
2          public class a {
3                  public void a() {...}
4                  public void a(int a) {...}
5          }
6          public class b {
7                  public void a() {...}
8                  public void a(int a) {...}
9          }
10 }
```

While the first example with its single characters might be hard to read, the OBad Android Trojan mentioned earlier makes use of a different approach that names classes and static variables by only using permutations of the characters `o`, `c`, `i` and `l`. In Listing 4.5 an excerpt is given that illustrates the difficulty to differentiating them.
Compared to single alphabetical characters they are much harder to remember. The combination of upper- and lowercase characters additionally confuses as they have to be differentiated as well.
This concept where characters that appear identical are used is called homoglyphic confusion. In the case of the OBad Trojan a visual inspection can be confused by the characters "I" and "l" as well as "c" and "o".

Listing 4.5: Identifier Obfuscation of the OBad Android Trojan

```
1  public final class CcoCIcI {
2      private static final byte[] COcocOlo;
3      private static boolean CcoCIcI;
4      private static BluetoothAdapter IoOoOIOI;
5      private static long OoCOocll;
6      private static String cOIcOOo;
7      private static final OoCOocll lOIIloc;
8      private static ArrayList occcclc;
9      private static final occcclc oclClII;
10     private static Thread ooCclcC;
11     ...
12 }
```

The illegibility of non-alphabetical characters in Listing 4.6 is enhancing this technique. But not only analysts are affected. Any software unable to deal with foreign characters will most likely produce errors or crash.

Listing 4.6: Identifier Obfuscation with Non-Alphabetical Characters

```
package 你 {
        public class 音 {
                public void 你() {...}
                public void 你(int ) {...}
        }
        public class 典 {
                public void 你() {...}
                public void 释() {...}
                public void 祢(int 你) {...}
        }
}
```

A simple solution would be to collect all unique identifiers and replace them with a variable followed by a unique number. For example every time the identifier "CcoCIcI" is found it is replaced by "identifier_1". Compared to non-alphabetical characters this is already an improvement and much easier on the eye.
The disadvantage of this solution is that the identifiers are still not unique. The identifier "identifier_1" could be a package name, a class name, a method name, a field name or most likely all of them at once. Additionally *each class* could have a field and at least one method called "identifier_1". Therefore code with non-alphabetical identifiers is merely transformed to a code with alphabetical identifiers. There is little to no advantage over the obfuscation in the first example.
A much better approach is to generate meaningful identifiers by utilizing all information available.
For the class name a very advanced renaming algorithm could guess what this particular class is doing and derive a name from that knowledge. Other than that there is only the possibility to generate a static name with a pattern like "class_sequential number". This would change the representation of a class as shown in Listing 4.7.

Listing 4.7: Class Renaming with Sequential Numbers

```
// the original class declaration
public final class 你
// the changed class name
public final class class1
```

This will contribute tremendously to the readability of the Java bytecode. File names have to be changed according to the renamed class name as well to match their class name.
There is much more information left in the code regarding to field names. Java's type safety enforces the declaration for constants, variables, and methods in advance to prevent type errors. In Listing 4.8 an example shows how this declaration can be used. To ensure the uniqueness of each field a sequential number is attached.
Since public fields could be accessed from other classes, attaching their class name is an other idea. But as the information about the calling object is always present in the source code as well, this will only bloat the code.
The return type can additionally be used for the renaming of methods as Listing 4.9 shows. Table 4.1 provides an overview of identifiers and information available to make their

names more meaningful.

type	replaced by
class	class as a string, sequential number
fiels	variable type, sequential number
methods	method, sequential number, returns, return type

Table 4.1: Identifier and Available Information for Renaming

Before an identifier is changed its name needs to be verified. Some identifiers might have not been obfuscated on purpose. For instance if other applications rely on their naming. To change an identifier with a meaningful name would mean to give away valuable information.

Listing 4.8: Use of Type Declaration for Field Renaming

```
1 // the original class declaration
2 double 你;
3 byte[] 释;
4 // the changed field names with more meaningful names
5 double double_1;
6 byte[] array_of_byte_1;
```

Simple pattern analysis could help to avoid such a case. If all identifiers are single alphabetical characters as in Listing 4.4 a check for the length could already be sufficient. In case of the example in Listing 4.5 the length of the identifier in combination with the number of different characters could be compared. In Listing 4.6, a check for non-alphabetical characters could be sufficient.

Listing 4.9: Use of Return Types for Method Renaming

```
1 // the original method declaration
2 public final String 释() {...}
3 // the changed method name, reflecting the return type
4 public final String method_1_returns_object_of_String() {...}
```

4.2.4 Junk Byte Insertion

Junk byte insertion is one of the techniques with focus on preventive obfuscation.
The concept of using non-functional artificial code blocks as well as opaque predicates has already been introduced in Chapter 2.2.3. Now a combination of the two methods for the further bytecode obfuscation is explained.
In a paper from 2012, Schulz [59] describes how junk byte insertion is used to thwart decompilers.
Using the pseudo instruction `fill-array-data-payload`, he tries to hide the original bytecode. The instruction is only meant to hold the data for a regular instruction and can not be executed itself. The key to hide bytecode is the variable length of the instruction. Placed at the beginning of a method, with the length of the size of the method it overlaps the original bytecode. A simple unconditional branch in front of the `fill-array-data-payload` instruction as shown in Listing 4.10 skips its execution.

Listing 4.10: Usage of the Fill-Array-Data-Payload Instruction

```
1 # the unconditional jump:
2 goto :label1
3 # fill-array-data-payload instruction
4 :label1
5 # actual instructions
```

The linear sweep algorithm might see the branch but it might fail to correctly evaluate the overlapping. Decompilers using the recursive traversal algorithm will evaluate the branch correctly and skip the instruction as well.
Schulz claims that with opaque predicates the recursive traversal will also evaluate the `fill-array-data-payload` instruction.
To the question if this could be fixed in his tool, the developer of smali said: "dalvik actually considers these methods to be invalid - the only reason it works is because of the class access flag trick that causes dalvik to skip class verification. The output of baksmali should match how dalvik "sees" the bytecode. In this case, the fact that dalvik "sees" actual instructions within the array payload instruction is a bug in dalvik (which was later fixed), and baksmali shouldn't produce output that matches the behavior of this bug in dalvik by default."
The same principle is used for different kinds of Junk Byte instruction injections that can be placed within a branch that is never supposed to be executed.
Illegal opcodes could be used. But this is not possible within a class that is checked by the bytecode verifier and would lead to a broken application that can not be executed at all. The illegal opcodes need to be included within an unused class. This classes can easily be removed by an optimizer though.
Another option is to insert unused Dalvik opcodes [52]. The bytecode validator will accept them since they are legitimate opcodes but analyzers might crash. Either because the opcode is not implemented or because it is implemented and pursues the bogus opcode.
A simple solution to avoid problems with unused, undocumented or new introduced opcodes could be to build in some kind of resilience that avoids the evaluation of those opcodes.
Another way to confuse decompilers is to use the size and/or the offset setting within the dex file header (where a description of class files is located) to exclude error containing code. Any decompiler ignoring this settings will most likely produce broken code.

4.2.5 String Obfuscation

Strings can easily be spotted and they give away a lot information about the purpose and the context of the surrounding source code. Therefore they are a preferred target of obfuscation. Changing the representation of strings is one of the techniques of the data obfuscation.
Instead of hardcoding the strings directly in the code they can be stored in a different representation and build up at runtime. This can be achieved in different ways.
The conversation of strings into an array form seems to be suited best for further obfuscation. The array type can for instance be char or byte. Listing 4.11 shows the necessary calls to encode and decode an arbitrary string into a byte array representation and back.

The byte array is particularly suitable. Upon manual inspection, the standard `toString` call in Java shows the ID of the object instead of the contents of the byte array. A sample conversion as well as the different representations of the outputs can be seen at the end of Listing 4.11.

Listing 4.11: Byte Array En-/Decoder and Conversion Example

```
1  /**
2   * Encoder
3   */
4  public static byte[] encodeString(String myString) {
5    return myString.getBytes();
6  }
7
8  /**
9   * Decoder
10  */
11 public static String decodeByteArray(byte[] byteEncodedString) {
12   return new String(byteEncodedString);
13 }
14
15 /**
16  * Byte array to string converter
17  */
18 public static String bytesToString(byte[] bytes) {
19   String result = "[";
20   for (int i = 0, len = bytes.length; i < len; i++) {
21     result += bytes[i];
22     if (i + 1 < len) result += ", ";
23   }
24   return result += "]";
25 }
26
27 public static void main(String[] args) {
28   String myString = "Hello World!";
29   System.out.println("The string to manupulate: " + myString);
30   byte[] myStringByteEnc = encodeString(myString);
31   System.out.println("The arrays object ID: " + myStringByteEnc);
32   String myStringByteEncToStr = bytesToString(myStringByteEnc);
33   System.out.println("The string representation of the byte array: " +
       myStringByteEncToStr);
34   String myStringDecoded = decodeByteArray(myStringByteEnc);
35   System.out.println("The decoded string: " + myString);
36 }
37
38 output:
39 The string to manipulate: Hello World!
40 The arrays object ID: [B@38f0b51d
41 The string representation of the byte array: [72, 101, 108, 108, 111,
    32, 87, 111, 114, 108, 100, 33]
42 The decoded string: Hello World!
```

Usually encoder and decoder methods does not look as simple as in the example. By adding mathematical calculations to change the stored values in the array, another layer of complexity is added. In this case, for this thesis the obfuscation technique is called "Bytecode Encryption" and further pursued in Chapter 4.3.
Similar to the signature detection of viruses (introduced in Chapter 2.3) the decoder method needs to be found to deobfuscate the encoded string. This can also be achieved by a search for the signature of the decoder method. For this thesis the signature of a Java method is specified in Definition 4.1.

Definition 4.1. *The Signature of a Method in Java comprises of the three components of the methods declaration:*

- *the return type*
- *the number of parameters*
- *the parameter types*

Other than in case of the oligomorph viruses the obfuscated applications inspected for this thesis contained only one method per class to deobfuscate strings. The signature for the deobfuscation method always looked the same within one application.
As the method for the decoding is found, it can be applied to the encoded strings. Therefore the decoding method can be rebuilt and embedded into a static analyzer. This only works under the condition that the obfuscater always uses the exact same encoding and decoding algorithms. Any marginal change to this algorithms will immediately cause the deobfuscator to fail.
Deobfucators are therefore equiped with a pool of small varieties for the decoder method. The better solution is to use a dynamic analysis approach that makes use of the given decoder methods.

4.2.6 Manifest Obfuscation

An obfuscation that is part of the data obfuscation is the manifest obfuscation.
In the process of the apk file creation, the manifest [7] is compiled into a binary XML file. To recover a human readable form several tools are available. One of them is the android-apktool [2].
Listing 4.12 shows the structure of the manifest file from an example application.
Although Android is very strict with the checks of the manifest file, some obfuscation techniques have been observed that aim to produce errors during the decoding of the binary. An example of such an error, produced by the android-apktool, can be seen in Listing 4.13.

Listing 4.12: Manifest Example

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.example.messenger"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="17"
        android:targetSdkVersion="19" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.homework.messenger.MessengerMain"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission
        android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

This is fixed [13] in newer versions of the android-apktool. Apparently this was achieved by hiding some of the named attributes.

Listing 4.13: Android-Apktool Failure During the Manifest Encoding

```
I: Regular manifest package...
[Fatal Error] AndroidManifest.xml:2:37: Element type "manifest" must be
    followed by either attribute specifications, ">" or "/>".
```

4.2.7 Tamper Detection

In order to detect any manipulation of the application a mechanism that verifies the integrity of the application can be implemented. The check can be executed at the start of the application or before certain methods are called.
A very basic approach could be to store and validate the size of certain files or the file modification time stamps. Another implementation could be based on hash values/checksums of code blocks, classes or the whole program.
Signing the application with a digital signature [56] using a public-key cryptosystem could be a more advanced approach.
The developer computes a signature by computing a hash value for the application (or

parts of it) that is then encrypted with his private key:

$$Signature = Encrypt_{private-key-developer}(Hash - function(application))$$

The signature along with a certificate is stored within the application.
To verify the signature the hash of the application is computed by the verifying entity. The signature that came with the application is then decrypted with the developers public key and the two are compared. If the hashes are equal, the signature is valid:

$$Hash - function(applicaiton) \stackrel{?}{=} Decrypt_{developer-public-key}(Signature)$$

If the signature matches with the one computed by the author, the application has not been tampered with. Additionally the author of the application is confirmed.
All these mechanisms are dependent on other techniques to further obfuscate the used algorithms. If they are applied as the only measure of protection the deobfuscator can simply remove the tamper detection calls. A search for certain code patterns used to calculate hashes for instance can easily be implemented to detect them.

4.2.8 Reflection

Reflection is an obfuscation techniqe of the data obfuscation.
It is a powerful feature, typically used to extend applications with external features, to browse classes, for debugging and in test tools.
But it can also be used to obfuscate applications. Java only supports introspective reflection [30] where the program can not change itself like in other languages capable of intercession. Reflection allows the inspection of classes, interfaces, fields and methods of applications at runtime [18]. Furthermore can classes be instantiated and their methods be invoked.
In Listing 4.14 the class `Crypto` is instantiated and its method `getPrivateKey()` is called as it is usually done in Java.

Listing 4.14: A Standard Call in Java

```
1 Crypto cryptModule = new Crypto();
2 privateKey = cryptModule.getPrivateKey();
```

In Listing 4.15 the same call is shown, this time with reflection. The most noticeable difference is the full classpath and the fact that the classpath and the method are given as strings.

Listing 4.15: Invokation of the Same Method with Reflection

```
1   Object reflectedClassInstance =
2         Class.forName("de.tum.secureApp.Crypto").newInstance();
3   Method methodToReflect =
4         reflectedClassInstance.getClass().getMethod("getPrivateKey");
5   Object invokeResult =
6         methodToReflect.invoke(reflectedClassInstance);
```

Especially static analysis is affected by this kind of reflection since it is usually unable to evaluate this calls.

Even in a standard situation where reflection is used can analyzing tools have problems with reflected code and therefore might ignore it [48]. Call graph generators can for example have trouble to detect the edges and knots correctly when reflection is used. Which leads to gaps in the graph and errors in the results of its evaluation.
Nested reflection and reflection in combination with Bytecode Encryption is even harder to handle.
An approach to statically solve some cases of reflection is shown in Chapter 5.3.

4.3 Bytecode Encryption

To encrypt bytecode means to apply obfuscation that changes the structure of the data. This is one of the techniques of the data obfuscation.
Low [49] suggests that encryption is the only way to defeat decompilation. But as long as there is no dedicated co-processor in form of a cryptochip it will always be possible to intercept and decrypt the source code.
In [44], Herzberg describes a hardware decryption systems to decrypt the code before it is passed to the main processor. At no time are the instructions stored in a user accessible memory and therefore protected against inspection.
Current Android devices offer no tamper proof environment that is accessible to developers.
Secure elements (SE) are present in the form of a smart card and in the NFC architecture. But the SE of the smart card is only accessible via the radio interface layer and only accepts very limited commands [1]. The addition of a standardized cryptochip to all devices could be a solution. For end users this would mean a higher price for the device which they are not likely willing to pay.
Since it is not guaranteed that all devices are equipped with the relatively new NFC technology or any kind of cryptochip, developers can not rely on it.
Obfuscators achieve encryption by introducing new methods to decrypt data represented only in encrypted form within the code. This data can represent everything from an integer to a full-blown file archive [1] containing one or more class files. Also dictionaries are possible that contain most of the strings used in the application.
The workflow to identify the use of encryption and to find a possible decryption routine could look like this:

1. Scan for classes containing suspicious data structures like byte arrays that seem to embody no useful data
2. Find where those data structures are used. There is probably a method within the class using it
3. Search the caller of this method and its return type
4. Should the return type be a string this is possibly a string decryption method

Listing 4.16 shows an example for an encrypted string and its decryption method. The data structure holding the encrypted data is the byte array 你 in Line 1. Its decryption is

[1] Zip for instance is a container format to archive files

performed only by the method 释 in Line 5. Although the method is parameterized there is no external data needed for the decryption, not even from an other class of the same application. Line 33 shows the call to the decryption method that only uses the variable i, that is given as a part of the encrypted data in Line 1.

Listing 4.16: Translated String Encryption Example in DexGuard (simple)

```
1  private static final byte[] 你 = { 81, -102, -37, -17, 24, 8, -1,
2      -80, 54, 19, 21, -2, -82, 74, -5, 20, -81, 68, 5, 2, 11, -76,
3      82, 1, 6, 3, -13, 1, -65 };
4
5  private static String 释(int paramInt1, int paramInt2, int paramInt3)
6  {
7    int i = 89 + paramInt3 * 2;
8    int j = 4 + paramInt1 * 2;
9    byte[] arrayOfByte1 = 你;
10   int k = 26 + paramInt2 * 2;
11   byte[] arrayOfByte2 = new byte[k];
12   int m = 0;
13   int i1;
14   if (arrayOfByte1 == null) {
15     i1 = k;
16   }
17   for (int i2 = j;; i2 = arrayOfByte1[j])
18   {
19     int i3 = i1 + i2;
20     j++;
21     i = i3 - 2;
22     int n = m;
23     m++;
24     arrayOfByte2[n] = ((byte)i);
25     if (m == k) {
26       return new String(arrayOfByte2, 0);
27     }
28     i1 = i;
29   }
30 }
31 ...
32   int i = -1 + 你[23];
33   Toast.makeText(localContext, 释(i, i, i), 1).show();
34 ...
```

Reversing this encryption can be complicated even tough all information for the decryption is available within the application. In general, possible ways to decrypt such data are:

- Use brute force or dictionary attacks on the data
- Get the source code, initialize the class containing the encrypted data, call the identified decryption method
- Manually inspect the source code and write a decrytion program

- Use dynamic analysis and fetch encrypted strings as they are decrypted
- Use reflection to get the application to decrypt the data during runtime

Brute force may be possible but since all the necessary data for the decryption is available this approach seems not to be necessary.
The easiest way to perform the decryption seems to run the decryption as the original application would. Listing 4.16 shows the code that includes the decryption method, translated to Java bytecode. The translation to Java source code was performed using the dex2jar converter. And even though no errors occurred, during the translation some instructions in the bytecode were not translated properly. Therefore this code is defective and does not decrypt the data.
By analyzing the Java bytecode it is also possible to write a static decryption program. This is not only time consuming, but on top of this the program is most likely not reusable. Even the slightest change in the encryption leads to useless results. And since obfuscators can easily vary their decryption algorithm every time the application is generated a little bit, a reliable automated decryption can not be guaranteed.
With access to the bytecode sources, an analogue method would be to implant the code into a dummy application. This has the advantage of calling the most original form of the code. As already mentioned, the encrypted data could hold several objects. Therefore an increasing number of calls to the decryption method leads to a growing complexity of this solution. Even automated it takes time to find the decrypted data as well as the decryption method and calls, to copy them into a dummy package, create and build the dummy application, transfer and install it, and finally run it and capture the decrypted output. Furthermore will this approach only be successful as long as the decryption method does not call any other methods.
Instead, injecting additional debug code could lead to a fast and valid encryption. The code could be injected directly into the applications bytecode. Automating the injection is easy. The debug code can be injected either directly to the decryption method. Matching which call caused the decryption could be complicated here. Or it could be injected after each decryption call where a sequential number could help to identify the position in the source code.

Listing 4.17: Injection of Bytecode to Output Decryption via Debug Messages

```
1 invoke-static {v1, v1, v1}, Lo释/;->(III)Ljava/lang/String;
2 move-result-object v1
3 const-string v2, "INJECTION_42"
4 invoke-static {v2, v1}, Landroid/util/Log;->
5                                 e(Ljava/lang/String;Ljava/lang/String;)I
```

In Listing 4.17 the bytecode to call the decryption method from Listing 4.16 is shown. Line 1 contains the actual call where the data is decrypted and received. The result is allocated then in register `v1` as shown in Line 2. The Lines 3 and 4 are injected lines. The label to identify the debug message is created in Line 3. Line 4 finally outputs the decrypted contents of register `v1` together with its identifier to the debugger. While this solution seems rather elegant there are several obstacles to overcome.
All decryption calls need to be reached at least once by a dynamic analysis to capture their decryption. Therefore all paths in the application need to be explored. Those are possibly

obfuscated as well by Control Flow Obfuscation. A full decryption can therefore not be guaranteed.
The last technique covered here for automating the decryption is by the use of reflection. Reflection is introduced in Chapter 4.2.8 where it is used to obfuscate the invocation of methods.
The fundamental advantage of reflection is that it is not necessary to know the name of classes, methods or fields of a program during runtime. Instead the program can be queried for this information using Java's reflection API.
If the name or the signature of the decryption method is known the application can be scanned to find it. Reflection allows to retrieve lists of the methods an their return and parameter types for each class. As the decryption method is found it can be used for the purpose of decryption.
The example in Listing 4.18 shows how reflection can be instrumented to detect the name of the decryption method.

Listing 4.18: Searching for a Class Name Using Reflection

```
1 // receive a list of all methods within a class
2 Method[] methodsInClass = getClassMethods(classToInvestigate);
3 // cycle through the list of methods
4 for (Method method : methodsInClass) {
5   if( method.getName().equals("getDecryptedText") )
6     return method;
7 }
```

As the method responsible for the decryption is found it can be utilized for the decryption. By the use of reflection it is possible to use the during runtime acquired method name and pass it to a method like the one in Listing 4.19.
It takes the name of an arbitrary class and method, creates an instance of the class, calls the method and returns the result.

Listing 4.19: Invokation of an Arbitrary Method by Reflection

```
1 public Object getDecryptedData(String className,
2                                String methodName) throws Exception {
3
4   Class reflectedClass = Class.forName(className);
5   Object reflectedClassInstance = reflectedClass.newInstance();
6   Method reflectedMethod = reflectedClass.getMethod(methodName);
7   Object invokeResult = reflectedMethod.invoke(reflectedClassInstance);
8   return invokeResult;
9 }
```

With the help of reflection, disassembling and reassembling is no longer needed for the investigation of an application. Tamper detection and anti debug techniques are also without effect.
For the deobfuscation of the encrypted strings in the applications, all calls to the decryption routine need to be found and replaced with decrypted data.

4.4 Existing Obfuscators

This section will briefly introduce different commercial obfuscators and their capabilities. Due to Androids security concept [5], each application has to declare the permissions it requests in the manifest. If an application for example needs to access the persistent storage of a device it needs to request it. The same applies for internet/network access. Applications with no need for accessing any network connection will not request the privileged to do so.
Requesting as few privileges as possible makes an application trustworthy. An application that has access to the device's storage (for example as a file manager) but no internet access, can not spy on users. With full network access on the other hand it could easily send the users data to a remote location.
This is most likely the reason for commercial obfuscators not to request additional permissions. And since they can not rely on each application already asking for full internet/network access, they have no possibility to use an external key validation- or decryption system.
Therefore techniques like Dynamic Loading of Code or the Client-Server Model will most likely not be part of any free or commercial obfuscator.
By comparing different obfuscators their similarities show how different obfuscators all use the same principles.

4.4.1 ProGuard Java Obfuscator

ProGuard [19] is a basic obfuscator that comes as a part of the Android SDK. It can be used to obfuscate Java and Android applications. Its features are:

- Shrinking (removal of unused code)
- Bytecode optimization (removal of unused instructions)
- Obfuscation (identifier renaming)
- Preverification (of the bytecode)

4.4.2 DexGuard Android Obfuscator

DexGuard is the commercial version of ProGuard. In addition to the features of ProGuard it offers the following features (according to the official webpage):

- Directly targets Dalvik bytecode
- Encryption of strings, classes, native libraries and assets
- Insertion of reflection
- Tamper detection
- Removal of logging code

4.4.3 Allatori Java Obfuscator

The Allatori obfuscator [4] for Java advertises the following features:

- Name obfuscation
- Flow obfuscation
- Debug info obfuscation
- String encryption
- Protection against popular decompilers

It is possible to integrate the obfuscator in the build process of the Android application. The Allatori homepage offers some samples of their obfuscated code. One of them can be seen in Listing 4.20.

Listing 4.20: Example of String Encryption in Allatori

```
1  // Original source:
2
3      private void checkLicense() throws Exception {
4          if (!isLicenseValid())
5              throw new Exception("Invalid License.");
6          else
7              return;
8      }
9
10 // Obfuscated with String Encryption then decompiled:
11     private void b() throws Exception {
12         if(!a())
13             throw new Exception(a.a("\\z't}}v5Q}}pwg\177{"));
14         else
15             return;
16     }
```

4.4.4 Arxan EnsureIT

Arcan [20] protect Android applications with their product "EnsureIT for Android/ARM". On their homepage they claim protection against reverse-engineering, disassembly and debug attempts. Unfortunately there are only limited information available to the used techniques.
In a product announcement [8] they write about: "multi-layered, end-to-end protection for Android applications" and "various security techniques (called Guards) such as obfuscation, checksum or anti-debug".

5 DexGuard Obfuscation Reversal

This chapter presents the framework created as a part of this thesis and its components to deobfuscate three of the main obfuscation techniques of DexGuard.
A combination of static- and dynamic analysis is used to decrypt strings. The reflection calls to obfuscate method invokations are then partially replaced with direct calls. And finally an algorithm is utilizing all available information to reintroduce meaningful identifiers.

5.1 The Framework

The framework's architecture is shown in Figure 5.1. It depicts the two components and details of its implementation.
The primary component is a Python program used for the static analysis, the secondary component is an Android application for the dynamic analysis. They are connected by a TCP/IP network.
The Python program can access all functions of the framework. A single wrapper class provides an abstraction from the implementation of the two components and the underlying classes.
The deobfuscator can be started from the command-line. The command-line arguments shown in Listing 5.1 are used to access the different functions and to supply additional parameters like the IP address the Android application is hosted.

Listing 5.1: The Command-Line Output of the Wrapper Class

```
> python wrapper.py --help
usage: wrapper.py [-h] -o OPTION [-a APKFILE] [-i HOSTNAME] [-p PORT] [-v]

The Deobfuscation Framework. Written 2014 by Hannes Schulz.

optional arguments:
  -h, --help                        show this help message and exit
  -o OPTION, --option OPTION        prepare, decrypt, unreflect, rename
  -a APKFILE, --apkFile APKFILE     The APK file of the application to analyze
  -i HOSTNAME, --hostname HOSTNAME The IP Reflector is bound to
  -p PORT, --port PORT              The Port Reflector is bound to
  -l LENGTH, --length LENGTH        Shorter Identifiers than [length] are
      considered obfuscated
  -v, --verbose                     Verbose mode
```

To unpack the `classes.dex` file from the APK file under investigation and to disassemble it with smali, the option *prepare* is provided. The option *rename* deals with identifier renaming, *decrypt* reverses bytecode encryption, and *unreflect* resolves reflection.
The Android application is called "Reflector". With the help of reflection it invokes the

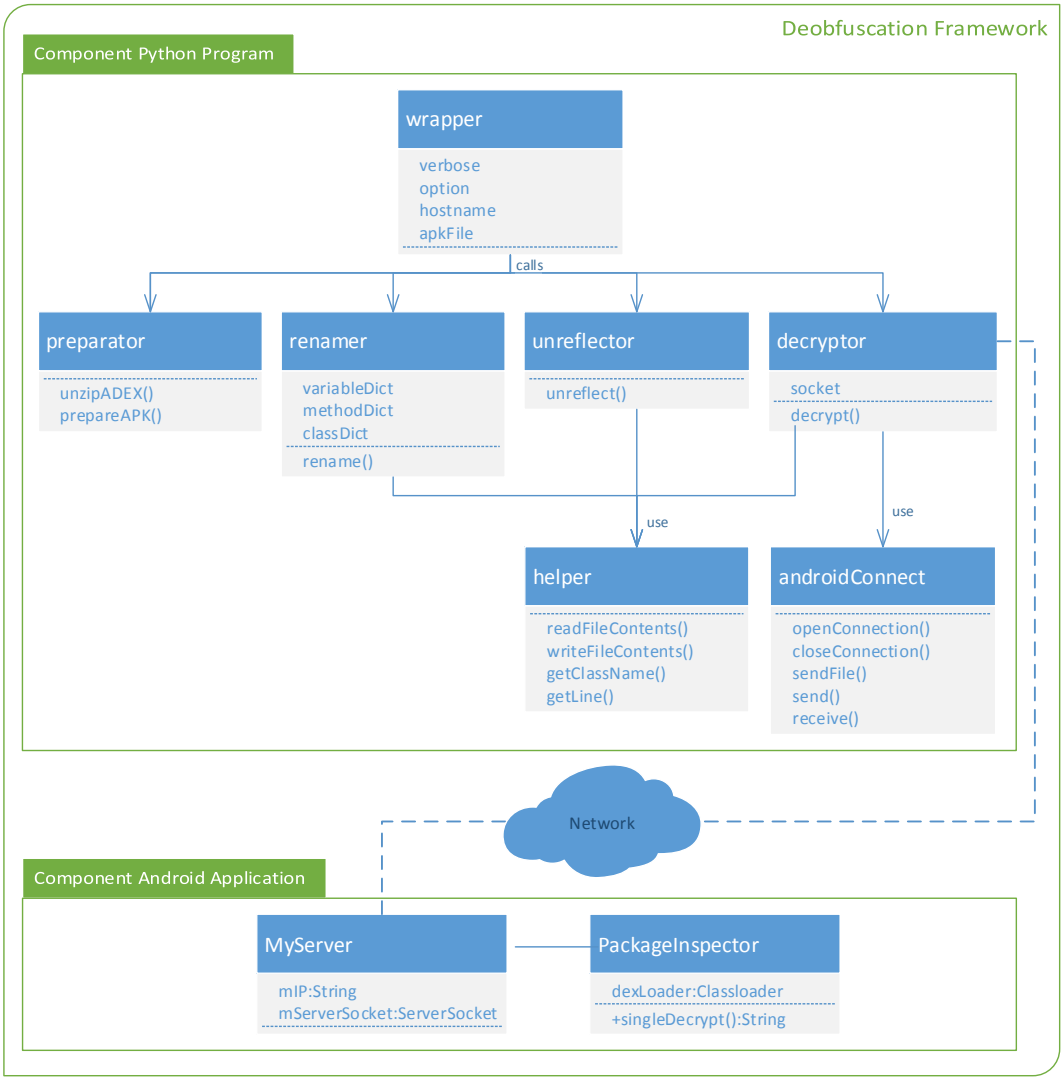
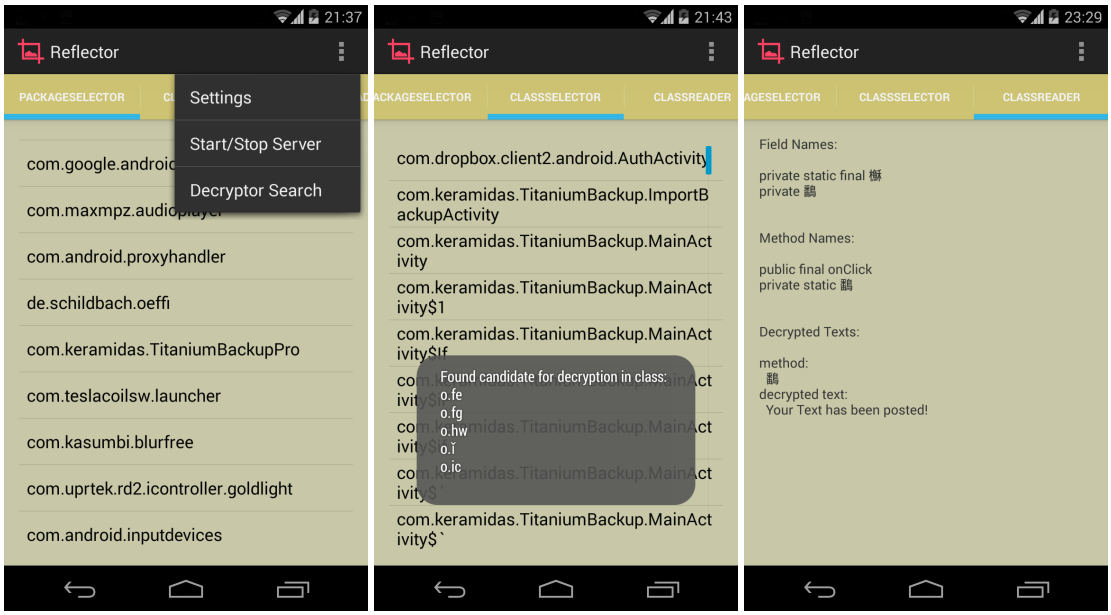


Figure 5.1: The Framework Architecture

decryption method of the analyzed application. Hence the name Reflector.
It can act in server mode to interact with the Python program to provide dynamic analysis to the framework. But it can also act as a standalone application.
In standalone mode the capabilities of Reflector are not limited to browsing applications and their classes. It can also read classes, search applications for potential decryptors and invoke them directly.
Screenshots of Reflector and its tabbed design layout that allows to switch between packages, contained classes and their contents are shown in Figure 5.2.
The advantage of having the dynamic component of the framework written as Android application is the possibility to run it on several environments. With a custom emulator for instance it is not possible to switch between emulated and dedicated Android hardware. This can be useful if anti debug obfuscation techniques prevent the usage of emulators.



(a) Package Selector (b) Class Selector (c) Class Reader

Figure 5.2: The Android Application Reflector

5.2 String Decryption

The first presented implementation of deobfuscation is the reversal of bytecode encryption. It is also recommended to apply this technique first since other obfuscation techniques like reflection are based on it.
Figure 5.3 shows the recommended order to apply the options of the framework.

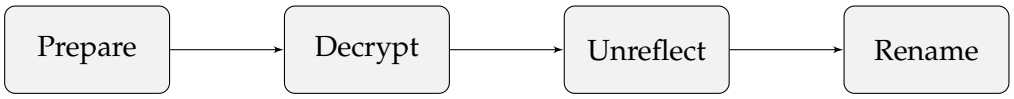


Figure 5.3: Recommended Sequence of Applying the Options of the Framework

The inspected DexGuard versions 5.3 and 5.5 encrypt strings by representing them in a byte array. These byte arrays are transformed by a certain set of instructions. If the string is required during runtime it is decrypted by a method containg the instructions to reverse the applied transformation.
Listing 5.2 shows a method to decrypt a string encrypted by DexGuard. The method was developed by manually analyzing and reversing the instructions from Dalvik bytecode.
The arguments of the `decryptString` method are also calculated within the original method. They define the position to start the encryption in the byte array, the length of the string as well as an initialization vector.

To simplify the method the instructions to reverse those arguments are not shown.
In an attempt to apply this static decryption method to encrypted byte arrays of other applications, or even the same application after recompilation, the method failed to provide a valid result. After recompilation, the algorithm in Line 25 in Listing 5.2 needed to be changed to the algorithm in Line 26. DexGuard seems to be equipped with a pool of variations to crate different versions of the encryption/decryption. Thus a static solution for the decryption of strings is not sufficient.

Listing 5.2: Java Method to Decrypt Obfuscated Strings

```
1  /**
2   * @param encryptedSourceArray
3   *            the encrypted source array
4   * @param encryptedArrayPos
5   *            defines the start of the encrypted string
6   *            in the byte array
7   * @param decryptedArrayLength
8   *            defines the length of the decrypted string
9   * @param currentValue
10  *            defines a start value
11  */
12 static String decryptString(byte[] cryptedArray, int encryptedArrayPos, int
       decryptedArrayLength, int currentValue) {
13
14   // create a temporary array
15   byte[] decryptedStringArray = new byte[decryptedArrayLength];
16
17   // encryption routine
18   for (int decryptedArrayPos = 0; decryptedArrayPos < (decryptedArrayLength -
         1); decryptedArrayPos++) {
19
20     // put the current value in the array
21     decryptedStringArray[decryptedArrayPos] = ((byte) currentValue);
22
23     // subtract the current value of the crypted
24     // array from the current value
25     //currentValue = currentValue - cryptedArray[encryptedArrayPos];
26     currentValue = currentValue + cryptedArray[encryptedArrayPos] - 2;
27
28     // set the pointer of the crypted array to the next position
29     encryptedArrayPos++;
30   }
31
32   // return the decoded string
33   return new String(decryptedStringArray, 0);
34 }
```

In a second attempt to decrypt strings the Android application Reflector was written. It uses Android's classloader and reflection to iterate through the classes and methods of an application and searches for decryption methods.
The key for the detection of the decryption method is to search for the decryption signature. Listing 5.3 shows the signature of the decryption method used in DexGuard. It always takes three integer arguments and returns a string. Its name is arbitrary in each class and therefore not suitable as indicator for a decryption method.

Listing 5.3: The Signature of DexGuards Decryption Method

```
// the signature in Java
private static String 释(int paramInt1, int paramInt2, int paramInt3)
// and in smali code
.method private static 释(III)Ljava/lang/String;
```

The code used in Reflector for the signature search in standalone mode is shown in Listing 5.4. All classes containing a potential decryption methods are stored into a list.

Listing 5.4: The Signature search in Reflector

```
// go through all methods of each class
Method[] mClassMethods = getClassMethods(classToInvestigate);
for (Method method : mClassMethods) {
  // condition one: return type needs to be string
  if (!method.getGenericReturnType().toString().equals("class␣
      java.lang.String"))
    continue;
  // condition two: the method takes three parameters
  if (method.getGenericParameterTypes().length != 3)
    continue;
  // condition four: all three parameters are of type int
  if (method.getGenericParameterTypes()[0] != (Integer.TYPE) ||
      method.getGenericParameterTypes()[1] != (Integer.TYPE) ||
      method.getGenericParameterTypes()[2] != (Integer.TYPE))
    continue;
  // write the method name to the log
  Log.d(TAG, "—>␣Possible␣Decryption␣found␣in␣class:␣" + classToInvestigate +
      ";␣method:␣" + method);
  // store the method in the list of detected decryptors
  mDecryptorMethods.add(method);
}
```

For the decryption, the classloader loads the classes in the list. And with the help of reflection the decryption methods are invoked as shown in Listing 5.5.

Listing 5.5: Invoking a Method to receive the Decrypted String

```
/**
 * A caller for potential decryption methods
 *
 * @param mMethod, the method to call
 * @return the decrypted string
 */
public String callMethod(Method mMethod) {
  mMethod.setAccessible(true);
  String result = "";
  try {
    result = (String) mMethod.invoke(classToInvestigate, 0, 0, 0);
  } catch (Exception e) {...}
  return result;
}
```

But this technique is only sufficient for a single encrypted string where the three integer parameter are defaulted to "0". In cases where the byte array contains more than one

string, the decryption method needs to be invoked with custom parameters. These are the same parameters used for the manual decryption in Listing 5.2.
In Listing 5.6 an excerpt of class with multiple calls (in Lines 4 and 11) to the decryption method is shown. The arguments for the first call of the decryption method in Line 4 are declared in the registers of the lines above the call. This repeats for the call in Line 11.

Listing 5.6: Excerpt of a Class with Multiple Encrypted Strings

```
1  const/16 v0, 0x17
2  const/4 v1, -0x1
3  const/16 v2, -0x9
4  invoke-static {v0, v1, v2}, Lo/xy释;->(III)Ljava/lang/String;
5  move-result-object v0
6  invoke-static {v0},
       Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
7  move-result-object v0
8  const/4 v1, 0x6
9  const/4 v2, -0x6
10 const/4 v3, -0x3
11 invoke-static {v1, v2, v3}, Lo/xy释;->(III)Ljava/lang/String;
12 move-result-object v1
13 const/4 v2, 0x0
14 invoke-virtual {v0, v1, v2}, Ljava/lang/Class;->
       getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;
15 move-result-object v0
16 const/4 v1, 0x0
17 invoke-virtual {v0, p0, v1}, Ljava/lang/reflect/Method;->
       invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
18 move-result-object v0
```

The solution to decrypt such strings is to use a combination of static- and dynamic analysis. The decryption workflow of the framework is shown in Figure 5.4.
The Python program starts with a search for a decryption signature in all classes of the application under investigation. To search for the signature of the decryption method the pattern "`(III)Ljava/lang/String;`" is used. It also determines if and where the decryption methods are invoked. For the names of the classes containing the decryption methods as well as for the classes containing a call to a decryption method a dictionary is created. Only if one or more potential decryption methods are found and at least of them is invoked, the program continues.
The framework opens a TCP/IP network connection to the Android application. Therefore Reflector needs to be running and the server mode needs to be activated. Additionally both components need to be located within the same network. Figure 5.2(a) shows the menu in the action bar where the server mode can be started.
Upon start of the server mode the application shows the network address Reflector is hosted. This IP address needs to be supplied to the Python program as a parameter.
As the connection is established the Python program sends the application to investigate to the Reflector application. The Reflector uses its classloader to load the dex file.
The Python framework iterates through the dictionary of classes containing a call to a decryption method. Additionally it searches for the arguments the decryption method is invoked with. In Listing 5.6 they are `0x17`, `-0x1`, and `-0x9` for the first call in the Line 4.
In a dialogue between the Python program and the Reflector application the byte array

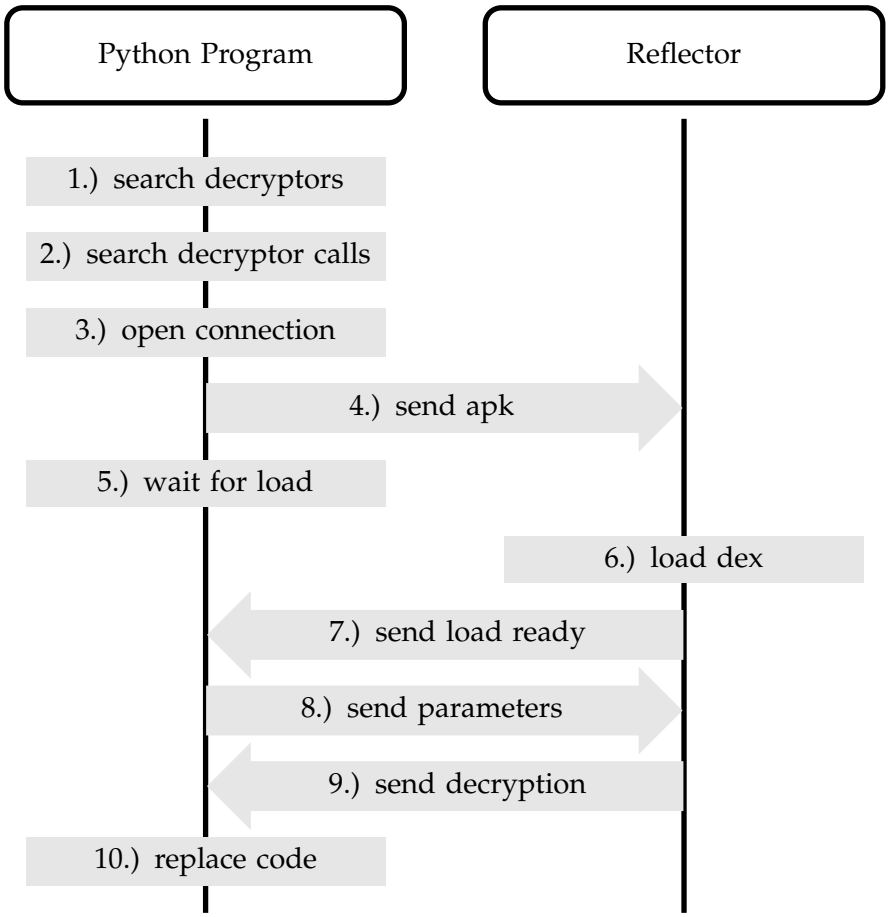


Figure 5.4: Decryption Workflow of the Deobfuscation Framework

is now decrypted. Each decryption call and its parameters is sent to Reflector. Reflector invokes the applications decryption method with the received parameter and sends back the result. Finally the Python program recreate the class file with decrypted strings.
The log of the decryption is shown in Listing 5.7.

Listing 5.7: The Log in the Pyton Program during Decryption

```
1 Found call in class: o/xy arguments: 23, -1, -9
2 sending: o/xy释;;23;-1;-9
3 received: android.content.Context
4
5 Found call in class: o/xy arguments: 6, -6, -3
6 sending: o/xy释;;6;-6;-3
7 received: getPackageCodePath
```

The decrypted version of the excerpt in Listing 5.6 is shown in Listing 5.8.
The invoke instructions to the decryption method were replaced with the decrypted strings. Currently the registers that hold the parameters for the decryption are not used elsewhere. To avoid any complications they are left withing the class after the decryption.

Listing 5.8: Listing 5.6 with Decrypted Strings

```
1  const/16 v0, 0x17
2  const/4 v1, -0x1
3  const/16 v2, -0x9
4  const-string v0, "android.content.Context"
5  invoke-static {v0},
       Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
6  move-result-object v0
7  const/4 v1, 0x6
8  const/4 v2, -0x6
9  const/4 v3, -0x3
10 const-string v1, "getPackageCodePath"
11 const/4 v2, 0x0
12 invoke-virtual {v0, v1, v2}, Ljava/lang/Class;->
       getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;
13 move-result-object v0
14 const/4 v1, 0x0
15 invoke-virtual {v0, p0, v1}, Ljava/lang/reflect/Method;->
       invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
16 move-result-object v0
```

The code excerpt also shows the invocation of reflection instructions. The next section shows how this reflection is resolved.
To be able to unreflect the calls, the decrypted strings are needed. Therefore the recommended order is to apply the decryption before resolving reflections.

5.3 Resolving Reflection

The Python program recognizes reflection within the bytecode by a static analysis. It searches for specific patterns, collects parameters and replaces the reflection calls with direct calls.
Chapter 4.2.8 already points out the complexity of different reflection constructs. To be able to reverse reflection, this program requires the instantiation of the reflection class- and method objects as well as the invocation to be in the same method. Additionally all parameters are expected to be within the method.
In case any external register is addressed the reflection can not be solved by the Python program. To include an investigation for external calls/parameters is beyond the scope of this thesis. This requirements seem to be met for the reflection introduced by DexGuard.
The previously decrypted code of Listing 5.8 was obfuscated with the help of reflections and is once more used as example.
For the reversal of reflections, the Python program starts with a search for classes that contain the `invoke` instruction. If found, the availability of the reflection class- and method object is verified.
The first argument of the `invoke` instruction contains the register of the reflected method object. In the example the `invoke` instruction is located in Line 15 where the register of the reflected method object is `v0`.
The program searches backwards in the bytecode to find the instruction that invoked the

`getMethod` function by searching for the assigned register. This is done in Line 12 in the example since the result of the instruction is stored in the register `v0` in Line 13.
The first parameter (`v0`) of the instruction that invokes the `getMethod` call holds the register of the reflected class object. The second parameter (`v1`) is the register where the name of the method to invoke is stored as string.
The name of the method (`getPackageCodePath`) is then recovered and the instruction to instantiate the reflection class object is located. In the example this is done in Line 5 where the instruction invokes the `getClass` method. The parameter of the instruction (`v0`) holds the register the name of the reflected class is stored in.
After locating and storing of the class name (`android.content.Context`) all necessary information to create a direct call are available. The result of the reconstructed direct call is put into the same register as the result of the reflected call.
If the reflection can be resolved completely, the code from the original file is replaced. Listing 5.9 shows the bytecode of the example after resolving the reflection.

Listing 5.9: Code of Listing 5.6 with Resolved Reflection

```
1 new-instance v0, Landroid/content/Context;
2 invoke-direct {v0}, Landroid/content/Context;-><init>()V
3 invoke-virtual {v0},
      Landroid/content/Context;->getPackageCodePath()Ljava/lang/Object;
4 move-result-object v0
```

The Python program also shows useful output as it deals with the reflection calls. Listing 5.10 shows the output for the resolved reflection call from the example in Listing 5.9 in Line 3. The Lines 1 and Line 2 show reflection calls that could not be resolved due to the use of external parameters.

Listing 5.10: Output of the Reflection Resolver

```
1 Class: o/a - methodNameRegister is an external parameter
2 Class: o/a - classInstanceObjectRegister is an external parameter
3 Class: o/b - unreflect call -> classname: "android.content.Context",
      methodname: "getPackageCodePath"
4 Class: o/b - unreflect call -> classname: "java.io.RandomAccessFile",
      methodname: "length"
```

5.4 Introduction of Meaningful Identifiers

Parts of the bytecode are obfuscated by substituting the original identifiers with meaningless identifiers. To make the interpretation as hard as possible, DexGuard uses Chinese characters, special characters and short combinations of letters. Figure 5.5 shows a screenshot of smali files as an example of the characters used in renamed classes. Smali uses the name of the class as filename in the disassembling process.
The identifiers of variables and methods within the files are replaced as well. Figure 5.6 partially shows the contents of an obfuscated class in smali disassembled Dalvik byte-

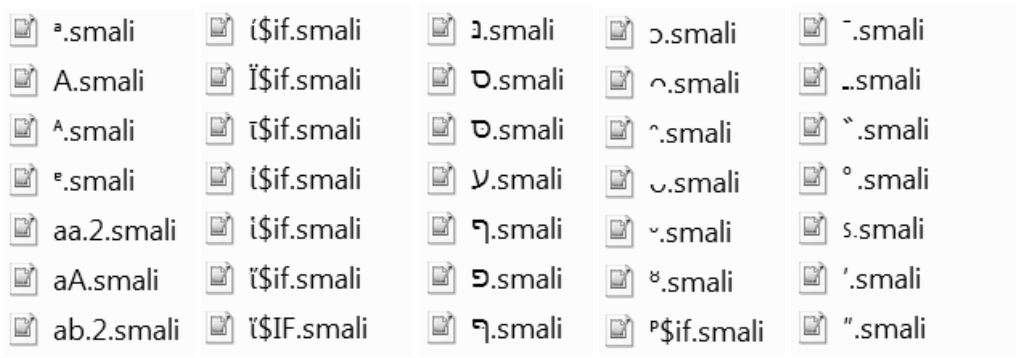


Figure 5.5: Class Name Substitution in DexGuard

code.
The Python program is able to avoid renaming of non obfuscated identifiers which still have meaningful identifiers. Therefore the length of the identifiers is analyzed. Is the length greater than the default or a custom value and only consists of alphabetic characters, it is considered not to be obfuscated. The default length is set to five characters, a custom value can be supplied as a parameter.

```
# instance fields
.field Ą:D
.field private ą:Z
.field private Ć:Z
.field private ć:Ljava/lang/String;
.field â:D
.field "¸:Landroid/location/Location;
.field private "¸:Landroid/location/LocationManager;
.field public 櫯:Z
.field final 鷂:Lde/tum/messenger/MessengerMain;

# direct methods
.method public constructor <init>(Lde/tum/messenger/MessengerMain;)V
    .registers 3
    invoke-direct {p0}, Landroid/app/Service;-><init>()V
    const/4 v0, 0x0
    iput-boolean v0, p0, Lo/鷂;->ą:Z
    const/4 v0, 0x0
    iput-boolean v0, p0, Lo/鷂;->Ć:Z
    const/4 v0, 0x0
    iput-boolean v0, p0, Lo/鷂;->櫯:Z
    iput-object p1, p0, Lo/鷂;->鷂:Lde/tum/messenger/MessengerMain;
    invoke-direct {p0}, Lo/鷂;->櫯()Landroid/location/Location;
    return-void
.end method
```

Figure 5.6: Excerpt from an Obfuscated Bytecode Class

The decompiled method in Figure 5.7 shows Java source code where the overuse of the `if` expression makes the method very hard to understand for the human reader.

```
private if 櫯(int paramInt1, int paramInt2)
{
  if localif = new if();
  localif.櫯 = paramInt1;
  localif.鶻 = this."_.鶻(this, paramInt1);
  localif.â = 1.0F;
  if ((paramInt2 < 0) || (paramInt2 >= this.ą.size()))
  {
    this.ą.add(localif);
    return localif;
  }
  this.ą.add(paramInt2, localif);
  return localif;
}
```

Figure 5.7: Meaningless Names within an obfuscated Method

The framework renames fields, methods and classes with meaningful- or at least easy distinguishable names. The workflow of the renaming process is shown in Figure 5.8.

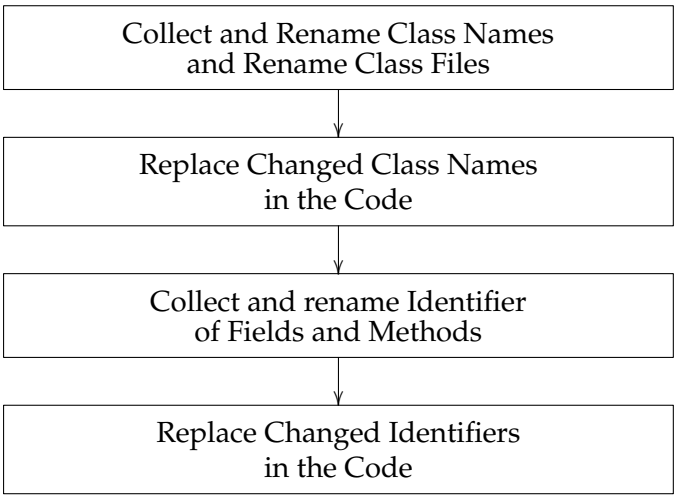


Figure 5.8: The Workflow to Reintroduce Meaningful Identifiers

To keep track of the changes, three dictionaries are generated during the renaming process. The old- and new names of the classes are in the first one. The other two dictionaries hold the old- and new names of the fields and methods that have been replaced, along with the classes they belong to.
The renaming procedure starts with the classes. Class names are replaced within the code and the corresponding filenames are adjusted.
Therefore all files are read once to collect their class names. As the class name is collected, a new name is generated and stored along with the old name in the corresponding dictionary. In the same run the name of each class is replaced and the files are renamed according to their new class names. Listing 5.11 shows an example of the verbose output.

Listing 5.11: Output during Collection of Class Names and Renaming of Class Files

```
1 Collected classname: o/ci  Changing to : o/class40
2 Collected classname: o/Oo  Changing to : o/class41
3 Collected classname: o/cC  Changing to : o/class42
4 Collected classname: o/CC  Changing to : o/class43
5 Collected classname: o/gL  Changing to : o/class44
```

After all class names are collected and replaced a second cycle replaces the references to the old class names within the code of all affected files. An example of the verbose output of the Python program is shown in Listing 5.12.

Listing 5.12: Output during the Renaming of Classes

```
1 Replacing changed classnames in file:
    \deobfuscator\smali\o\class40.smali
2 Replacing changed classnames in file:
    \deobfuscator\smali\o\class41.smali
3 Replacing changed classnames in file:
    \deobfuscator\smali\o\class42.smali
4 Replacing changed classnames in file:
    \deobfuscator\smali\o\class43.smali
5 Replacing changed classnames in file:
    \deobfuscator\smali\o\class44.smali
```

All field and method names are then replaced by identifiers that reflect their environment and their context. For field names the type of the variable is utilized to create new names. Table 5.1 shows the translation table used to identify the types in the bytecode.

declaration in smali	translation
`V`	void
`Z`	boolean
`B`	byte
`S`	short
`C`	char
`I`	int
`J`	long
`F`	float
`D`	double
`[`	array_of_
`L`	object_of_

Table 5.1: Translation Table for Field Types

More complex instances like arrays and objects are recursively translated. A multidimensional array of integers for instance results in `array_of_array_of_int_42`. For objects

the name of the corresponding object is attached to the identifier. This creates identifiers like `object_of_RadioButton_42`.
To rename methods the return type is utilized for a more meaningful name. To translate the return types, the same technique as with the fields is used.
Methods can have the same name but different parameters because of overloading. The algorithm labels every method name that it encounters more than once with the string `overloaded` instead of the return type. Otherwise all methods with the same name would bare the name of the return type of the first method. This would still result in valid identifiers but is most disturbing during the manual inspection of such code.

Listing 5.13: The Result Output of the Identifier Renaming shows a Class Outline

```
1  Collecting variable/method instances
2  class is: o/class42
3    v: int int_5615
4    v: boolean boolean_5683
5    v: object_of_String object_of_String_5113
6    v: array_of_byte array_of_byte_5684
7    v: object_of_ListView object_of_ListView_5082
8    v: object_of_class361 object_of_class361_5687
9    v: array_of_object_of_class1129 array_of_object_of_class1129_5675
10   m: int getNumber ()
11   m: void method_4462_overloaded (object_of_class980, object_of_class826)
12   m: object_of_String method_4462_overloaded ()
13   m: object_of_String method_4463_returns_object_of_String ()
14   m: object_of_Object method_4445_overloaded (object_of_InputStream)
15   m: object_of_Object method_4445_overloaded (object_of_String, object_of_Class)
16   m: object_of_String method_4445_overloaded (object_of_Object)
```

The verbose output in Listing 5.13 shows some examples for identifier renaming. It also gives an outline for each class. The structure is the same as in Java source code. Each line starts with a label that indicated if a variable or a method is described.
Finally the calls to the fields and methods are replaced. The output of the Identifier replacement is shown in Listing 5.14.

Listing 5.14: The Output of the Identifier Replacement

```
1  Replacing variable/method instances
2  parsing: com/example/class1
3  parsing: com/example/class2
4  parsing: com/example/class3
5  parsing: com/example/MainActivity
6  parsing: com/example/MainApplication
```

The code from the example in Figure 5.7 was translated with the process of this framework. Figure 5.9 shows the result.
The frameworks renaming module is programmed in a very efficient way. With the help of dictionaries and regular expressions, several thousand classes can be renamed within a second.

```
private class30 method_149_returns_array_of_int(int paramInt1, int paramInt2)
{
  class30 localclass30 = new class30();
  localclass30.int_84 = paramInt1;
  localclass30.object_of_Fragment_85 = this.object_of_class2_51.method_128_returns_boolean(this, paramInt1);
  localclass30.float_82 = 1.0F;
  if ((paramInt2 < 0) || (paramInt2 >= this.object_of_ArrayList_25.size()))
  {
    this.object_of_ArrayList_25.add(localclass30);
    return localclass30;
  }
  this.object_of_ArrayList_25.add(paramInt2, localclass30);
  return localclass30;
}
```

Figure 5.9: The Method of Figure 5.7 after Renaming

5.5 Limitations

The framework produces valid results for the investigated versions 5.3 and 5.5 of DexGuard.
For successful deobfuscation certain preconditions reflecting the obfuscations encountered during this study need to be fulfilled. The framework might also work for obfuscations of different obfuscators but it probably needs to be adapted.
The inspected obfuscated applications neither had the information about the used obfuscator nor the obfuscators' version embedded. This applies for applications from the Play Store as well as for self compiled and with DexGuard obfuscated applications. Automatically applying the deobfuscator might require a previous check for certain characteristics to determine the obfuscator and its version.
It is also possible that DexGuard's pool of variations was not completely exhausted by the limited number of compiled and obfuscated applications. Therefore unknown additional signatures of decryption methods or differently arranged arguments for the decryption method might not be detected by the framework. Differently generated reflection calls that appear in a different order for instance are also not resolvable.
The findings of the analysis of applications from external sources with several characteristics congruent to the inspected DexGuard versions were also incorporated in the framework. There is a possibility that they originated from an additional or similar obfuscator and led to wrong assumptions.
If the techniques deobfuscated by the framework were further obfuscated they might be undetected or deobfuscation could fail.

6 Conclusion

This thesis introduced a taxonomy for different obfuscation techniques. It examined the history of obfuscation and deobfuscation and their usage in malware. Derived from this knowledge, from scientific articles as well as from various internet sources, and manual code inspections, current state of the art obfuscation techniques have been presented.
Some of them have been analyzed in greater detail and theoretical possibilities for the deobfuscation were discussed.
As proof of concept a framework using static and dynamic analysis methods was written. It automatically deobfuscates three of the core techniques of the commercial obfuscator DexGuard.
Although the removal of meaningful identifiers is not reversible, the framework creates new meaningful identifiers that reflect their context. In contrast to special or Chinese characters they are an improvement for manual code inspections.
Because of the security by obscurity approach for the encryption in DexGuard, all necessary information for the decryption is available within the code. Using reflection, the same technique DexGuard uses to obfuscate the control flow, the framework simply utilizes the available decryption methods. Encryption is one of the most essential obfuscations since other techniques – otherwise are easily spotted – rely on it.
The framework partially deobfuscates reflection as well using a static approach. All parameters for the reflection are encrypted in the code but can be decrypted by the framework.
This framework only works for a certain set of obfuscations. If the signatures of the decryption method changes for instance the deobfuscator will fail.
Further work on this framework could realize a technique to identify different obfuscators and their versions. A signature database could be implemented with the content not limited to signatures of the decryption methods. It could also maintain information about additional reflection patterns. Characteristics of other obfuscators and their versions as well as signatures for new deobfuscation techniques could also be a part of the database.
As new techniques emerge the framework needs to be upgraded in order to detect them. New modules could therefore be attached and integrated into the framework.
The signature database as well as the framework's modules need to be updated regularly to detect new obfuscations, exactly like anti virus software. Therefore the framework could support the exchange of signatures by a community.

Bibliography

[1] Accessing the embedded secure element in Android 4.x. http://nelenkov.blogspot.de/2012/08/accessing-embedded-secure-element-in.html. (accessed on 9.7.2014).

[2] android-apktool - A tool for reverse engineering Android apk files. https://code.google.com/p/android-apktool/. (accessed on 9.7.2014).

[3] Android Native Development Kit (NDK). https://developer.android.com/tools/sdk/ndk/index.html. (accessed on 9.7.2014).

[4] Android Obfuscation. http://www.allatori.com/features/android-obfuscation.html. (accessed on 9.7.2014).

[5] Android Security Overview. https://source.android.com/devices/tech/security/#android-platform-security-architecture. (accessed on 9.7.2014).

[6] Android's Google Play beats App Store with over 1 million apps, now officially largest. http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest_id45680. (accessed on 9.7.2014).

[7] App Manifest. http://developer.android.com/guide/topics/manifest/manifest-intro.html. (accessed on 9.7.2014).

[8] Arxan Introduces a New Era in Android Security with Complete End-to-End Protection of Java and Native Apps. http://www.arxan.com/arxan-introduces-a-new-era-in-android-security-with-complete-end-to-end-protection-of-java-and-native-apps/?CategoryId=5. (accessed on 9.7.2014).

[9] Details and results of our analysis on the malware V2PX. http://www.mcafee.com/threat-intelligence/malware/default.aspx?id=98074. (accessed on 9.7.2014).

[10] dex2jar – Tools to work with android .dex and java .class files. https://code.google.com/p/dex2jar/. (accessed on 9.7.2014).

[11] DexFile - A DEX file Manipulator for class loaders. http://developer.android.com/reference/dalvik/system/DexFile.html. (accessed on 9.7.2014).

[12] Google Earth. http://www.google.com/earth/index.html. (accessed on 9.7.2014).

[13] Handles Manifests with Missing Named Attributes. https://github.com/iBotPeaches/Apktool/commit/e126a51b4bb8991042b48ec5bf916f396e75f6f0. (accessed on 9.7.2014).

[14] iBanking: Exploiting the Full Potential of Android Malware. http://www.symantec.com/connect/blogs/ibanking-exploiting-full-potential-android-malware. (accessed on 9.7.2014).

[15] Jasmin assembler for the Java Virtual Machine. http://jasmin.sourceforge.net/. (accessed on 9.7.2014).

[16] Java Decompiler – Yet another fast Java decompiler. http://jd.benow.ca/. (accessed on 9.7.2014).

[17] Java Language Keywords. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html. (accessed on 9.7.2014).

[18] Java Reflection. http://docs.oracle.com/javase/tutorial/reflect/index.html. (accessed on 9.7.2014).

[19] ProGuard. http://proguard.sourceforge.net/. (accessed on 9.7.2014).

[20] Security for Android Mobile Applications. http://www.arxan.com/products/mobile/ensureit-for-android-on-arm/. (accessed on 9.7.2014).

[21] Smali - An assembler/disassembler for Android's dex format. https://code.google.com/p/smali/. (accessed on 9.7.2014).

[22] The history of computer viruses. http://www.virus-scan-software.com/virus-scan-help/answers/the-history-of-computer-viruses.shtml. (accessed on 9.7.2014).

[23] The International Obfuscated C Code Contest (roemer.c). http://www0.us.ioccc.org/1989/roemer.c. (accessed on 9.7.2014).

[24] The Java® Virtual Machine Specification. http://docs.oracle.com/javase/specs/jvms/se7/html/index.html. (accessed on 9.7.2014).

[25] The most sophisticated Android Trojan. http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan. (accessed on 16.1.2014).

[26] Three graphs to stop smartphone fans fretting about 'market share'. http://www.theguardian.com/technology/2014/jan/09/market-share-smartphones-iphone-android-windows. (accessed on 9.7.2014).

[27] Worldwide Tablet Sales Grew 68 Percent in 2013, With Android Capturing 62 Percent of the Market. http://www.gartner.com/newsroom/id/2674215. (accessed on 9.7.2014).

[28] Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments Are On Pace to Grow 6.9 Percent in 2014. `http://www.gartner.com/newsroom/id/2692318`. (accessed on 9.7.2014).

[29] Michael Batchelder. MS thesis: Java Bytecode Obfuscation. 2007.

[30] Daniel G Bobrow, Richard P Gabriel, and Jon L White. Clos in context-the shape of the design space. *Object Oriented Programming: The CLOS Perspective*, pages 29–61, 1993.

[31] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[32] Damien Cauquil. Small footprint inspection techniques for Android. `http://events.ccc.de/congress/2012/Fahrplan/events/5123.en.html`, 2012. (accessed on 9.7.2014).

[33] Silvio Cesare and Yang Xiang. Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs. *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 181–189, November 2011.

[34] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for java bytecode. *Journal of Systems and Software*, 71(1):1–10, 2004.

[35] Srinivasan Chandrasekharan and Saumya Debray. Deobfuscation: Improving reverse engineering of obfuscated code. 2005.

[36] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. 1997.

[37] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[38] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *Software Engineering, IEEE Transactions on*, 28(8):735–746, 2002.

[39] Dr. Julian Schütte Dennis Titze, Philipp Stephanow. AppRay: Userdriven and fully Automated Android App security Assessment. 2013.

[40] Ozgun Erdogan and Pei Cao. Hash-av: fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks*, 2(1):50–59, 2007.

[41] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

[42] By Simson Garfinkel, Gene Spafford, and Second Edition. *Table of Contents Part IV : Network and Internet Security*. Number April. 1996.

[43] James R Gosler. Software protection: Myth or reality? In *Advances in Cryptology—CRYPTO'85 Proceedings*, pages 140–157. Springer, 1986.

[44] Amir Herzberg and Shlomit S. Pinter. Public protection of software. *ACM Transactions on Computer Systems*, 5(4):371–393, October 1987.

[45] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*, pages 169–186. Springer, 2013.

[46] DM Jones. Operand names influence operator precedence decisions. *C Vu*, 1:1–14, 2008.

[47] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. *Proceedings of the 10th ACM conference on Computer and communication security - CCS '03*, page 290, 2003.

[48] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for java. In *Programming Languages and Systems*, pages 139–160. Springer, 2005.

[49] Douglas Low. MS thesis: Java Control Flow Obfuscation. 1998.

[50] Egil Aspevik Martinsen. MS thesis: Detection of Junk Instructions in Computer Viruses. `https://www.duo.uio.no/bitstream/handle/10852/9943/Martinsen.pdf`, 2008.

[51] JesusFreke (Author of Smali). How does DalvikVM handle switch and try smali code. `http://stackoverflow.com/questions/14100992/how-does-dalvikvm-handle-switch-and-try-smali-code`, 2012. (accessed on 9.7.2014).

[52] Gabor Paller. Dalvik opcodes. `http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`. (accessed on 9.7.2014).

[53] Todd A Proebsting and Scott A Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *COOTS*, pages 185–198, 1997.

[54] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[55] SBA Research. Android Application Obfuscation. `http://www.usmile.at/sites/default/files/publications/201306_obf_report_0.pdf`, 2013. (accessed on 9.7.2014).

[56] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[57] Mike Schiffman. A Brief History of Malware Obfuscation. http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2/. (accessed on 9.7.2014).

[58] Mike Schiffman. A Brief History of Malware Obfuscation. http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2/. (accessed on 9.7.2014).

[59] Patrick Schulz. Dalvik Bytecode Obfuscation on Android. http://dexlabs.org/blog/bytecode-obfuscation. (accessed on 9.7.2014).

[60] Patrick Schulz. Code Protection in Android. *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universität Bonn*, 2012.

[61] Yunhe Shi, Kevin Casey, M Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2, 2008.

[62] Dr. Alan Solomon. Virus maker program. https://groups.google.com/forum/#!msg/alt.comp.virus/TSG5SooBBZw/LuIp_3zYLL0J. (accessed on 9.7.2014).

[63] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[64] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. *Proceedings of the 9th ACM symposium on Information, computer and communications security - ASIA CCS '14*, pages 447–458, 2014.

[65] M. Stamp W. Wong. Hunting for Metamorphic Engines. *Journal in Computer Virology, vol. 2, no. 3*, pages 211–229, 2006.

[66] Erik Ramsgaard Wognsen and Henrik Sonderberg Karlsen. MS thesis: Static Analysis of Dalvik Bytecode and Reflection in Android. 2012.

[67] Gregory Wroblewski. General Method of Program Code Obfuscation. 2002.

[68] Ilsun You and Kangbin Yim. Malware Obfuscation Techniques: A Brief Survey. *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, November 2010.

[69] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. AppInk : Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12. ACM, 2013.