IT UNIVERSITY OF COPENHAGEN

Abstract

Department of Software Development and Technology (SDT)

Master's Thesis

Generic deobfuscator for Java

by Mikkel B. Nielsen

Obfuscation is a tool used to enhance the security for even the most security critical systems today. However, obfuscation will only make it more expensive, time wise, for a dedicated hacker to analyze a system. In contrast, it may lead one to question whether the system is actually secure beneath the surface. Obfuscation is a tool that provides security by obscurity. Security by obscurity is a controversial subject. Some experts do not recognize it as a security means, as it does not stop an attack vector but only conceals it. The goal of this project is to create a flexible deobfuscator, that is easy to extend. The deobfuscations will be constructed using partial evaluation techniques to specialize abstract syntax tree walkers. The deobfuscator is written in ANTLR for extra flexibility.

Acknowledgements

I would like to thank my supervisor, Dr. Joseph Kiniry. He has been extremely important in the process of making this thesis. His guidance and expertise has been vital for the outcome. I would also like to thank my dear friend, Jimmie Rindahl, for taking the time to listen to my ideas throughout the thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

DanID and the security of their system, NemId, the Danish authentication service, have recently received negative focus in the media and are criticised by security experts [7][23]. In 2011, a flaw in NemId was published in a paper, stating that the security could be circumvented by a simple man-in-the-middle attack (MITM). DanID denied the accusations and called it for purely theoretical. A short time after an example of such an attack was carried out in practice, resulting in a loss of 700.000 DKK. from eight Danske Bank customers [7]. The original idea for this thesis was to decompile and analyse the NemId java applet, but the lack of useful deobfuscators turned the attention to the area of deobfuscation instead.

1.2 Problem

Obfuscation is one method to introduce the security through obscurity principle. Security by obscurity has never achieved engineering acceptance as it contradicts security by design, open security, and the KISS principle [24]. The United States National Institute of Standards and Technology (NIST) even recommends against security through obscurity, quoting "System security should not depend on the secrecy of the implementation or its components" [26]. Using techniques that goes against recommendations and security principles, leads one to think that the creator may hide deeper software security flaws.

1.3 Contributions

A generic deobfuscator would not only be worthwhile in this particular case but may be a great help to developers around the world. The deobfuscator would help convince security critical software system companies to shift away from obfuscating their code and turn to more recognized security principles like open security.

Initially a syntax grammar for javap bytecode is written in the Extended BackusNaur Form (EBNF) syntax notation, and then used to generate a lexer and a parser using the ANTLR parser generator. It is discussed how tree walkers can be used to write specializations for the parser's output abstract syntax tree (AST). Each tree walker will be used to rewrite the AST, resulting in deobfuscations. Two deobfuscation examples are provided, and the implementation of more is described in section 5.

To evaluate the deobfuscator, the robustness and correctness are measured. This is done for bytecode in the Java Runtime Environment, as well as Groovy, JRuby, and Scala, to show robustness across different bytecode source languages. The result validates the approach as a prove of concept, though further work is needed before the deobfuscator is fully functional.

Chapter 2

Background

2.1 Obfuscation

Since the dawn of time, man have been stealing from others and, as a result, trying to protect his own valuables. A company developing software, given the enormous investment of time and money, is dependent on its software not to be stolen or copied. Piracy has long been, and continues to be, a major threat to the software industry. In its most recent study the Business Software Alliance reported that the global economic impact of pirated software totaled more than US$63 billion in 2011, up from US$58 billion in 2010, and no less than 57% of all computer users have piracy software on them [6].

Code obfuscation has attracted attention to improve software security by making it difficult for attackers to understand the inner workings of proprietary software systems. The idea is to make the task of reverse engineering programs too expensive in terms of resources or time required to do so. Even though obfuscation is against other security principles, such as Kerckhoffs principle [27], it is observed in security critical systems. Kerckhoffs principle holds that a system should be secure because of its design, and not because the design is unknown to an adversary. By breaking these principles, companies leave the question whether they actually have a secure system underneath the concealment, or if they are trying to hide security problems.

The Windows NT LAN manager authentication protocol is an example where security through obscurity failed. The protocol was held secret until the Samba team had to reverse engineer it. They needed to implement the Samba interoperability product for UNIX-based operating systems and produced a comprehensive documentation, revealing a series of bugs [18].

2.1.1 The dark side of obfuscators

There is no guarantee that only trusted software will be using obfuscation techniques, and as such, malicious software can be obfuscated to hide its signature from virus scanners. Vira use polymorphism and metamophism, among others, to change its signature. It will certainly be valuable to the anti-virus companies, if techniques are developed, that are able to counter the complex virus obfuscation techniques.

2.1.2 Obfuscation techniques

We distinguish between two broad classes of obfuscation transformations. The first, surface obfuscation, focuses on obfuscating the concrete syntax of the program. Surface obfuscation includes the removal of comments, format changing and variable name scrambling. Scrambling is done by renaming different variables in different scopes to the similar or identical identifiers. This is seen in many obfuscators, e.g., the Dotfuscator tool for obfuscating .NET code and Zelix for obfuscating Java bytecode [28][12]. The second class is called deep obfuscation which target is to obfuscate the actual structure of the program, e.g., by changing its control flow or data reference behavior.

While surface obfuscation makes it harder for humans to read and understand the code, it will not have any effect on decompilers and the effect can easily be reduced using a parser to resolve variable references and then rename accordingly. Deep obfuscation, on the other hand, affects the program analysis and reverse engineering tools by changing control flow.

2.1.2.1 Surface obfuscation

A series of obfuscators have been looked up to get a clear picture of which features they deliver. Some of the obfuscators include Zelix, ProGuard, yGuard, and JBCO [12][13][32][31]. Name obfuscation seems to be a typical feature in most java bytecode obfuscators available. Identifier scrambling is a one-way function that does not cost anything in terms of time or space penalty. It is a transformation with high potency, since a lot of the explicit knowledge is lost in the naming transformation.

Often programmers will place domain related code close together. Some obfuscators will try to move code around to separate related code.

Another obfuscation technique within the surface obfuscation area is string encryption. The obfuscator will encrypt all strings in the program and make the program decrypt

```
public static int foo = 2;

public helloWorld()
{
}

public void bar(int i){
  foo = 6 + i;
}

public int foo(){
  return 1;
}
```

```
public static int a = 2;

public a()
{
}

public void a(int  i0){
  a = 6 + i0;
}

public int a(){
  return 1;
}
```

FIGURE 2.1: The example shows how some obfuscators changes identifiers to similar names, making it difficult for humans to separate one from another.

the strings before usage. Not much harm is done as a good static analyser is able to simulate the decryption function, used by the program, to decrypt all strings.

2.1.2.2 Deep obfuscation

Too many techniques exist in the deep obfuscation area to be covered in this paper. Instead, the aim for the deobfuscator is to handle a couple techniques explained in the following section.

Opaque predicates

An opaque predicate is an expression that always evaluates to true (or false) independently of the state of the program. The outcome is known to the obfuscator when inserted, but will need evaluation at runtime. Giacobazzi, Jones and Mastroeni defines a program to contain opaque predicates if and only if an abstract interpretation is incomplete [1]. Figure 2.2 shows the different types of opaque predicates.

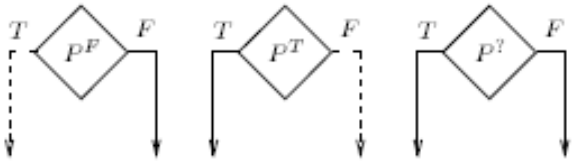

FIGURE 2.2: Different types of opaque predicates. Figure from [4].

Collberg, Thomborson, and Low divide opaque predicates into four resilience levels, ranging from trivial to full. An opaque predicate is trivial if a deobfuscator can deduce its value by a static local analysis. An analysis is local if its restricted to a single block of a control flow graph. They are also considered trivial if they are computed from calls to well-understood and simple library functions. An opaque predicate is weak if a deobfuscator can deduce its value by static global analysis. An analysis is global is it is restricted to a single control flow graph. An opaque predicate is strong if a deobfuscator

can deduce its value by static inter-procedural analysis. An analysis is inter-procedural if it can analyze the flow of information between procedures.

Collberg, Thomborson, and Low propose different kinds of counter measures against statical analysis. One counter measure is to design the opaque predicates such that several predicates have to be cracked at the same time. They do this by letting the opaque predicates have side effects, that will change the semantics of the program, if only one is removed.

Artificial blocks and dead code

Artificial blocks is one of the control flow obfuscations that uses opaque predicates as a basic building block. When an opaque predicate directs the control flow around a control flow block, the block itself is considered artificial. In Figure 2.3 an artificial block is illustrated. The predicate is dead code since it always evaluates to true.

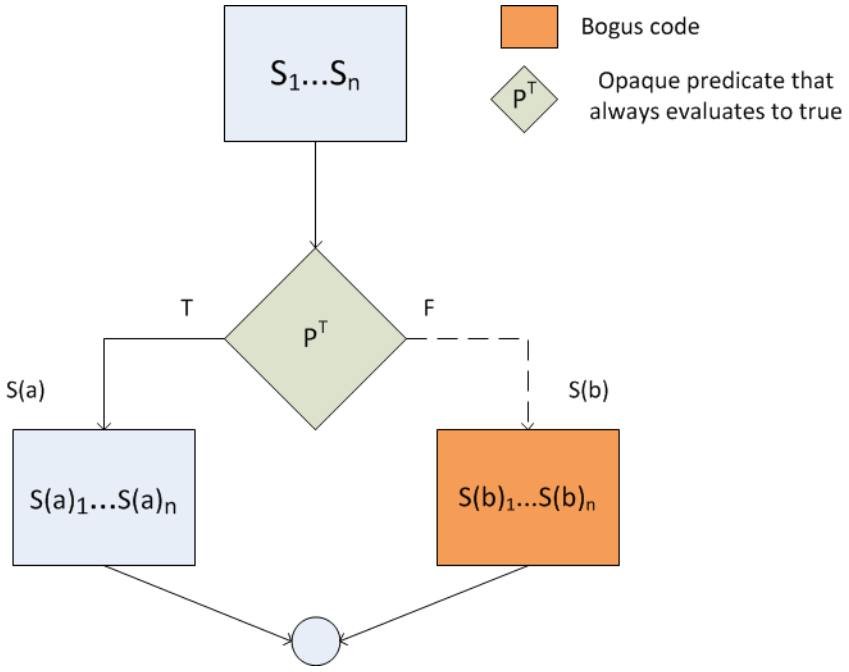

FIGURE 2.3: The example shows an opaque predicate P that always evaluates to true. The orange block is bogus code that will never be executed.

Reducible and non-reducible flow graphs

Most programming languages are compiled into native or virtual machine code which is more expressive than the language itself. When this is the case, it is possible to perform transformations to the virtual code that introduces sequence, which have no direct correspondence to the source language. Java bytecode has a goto instruction for which the Java language has no corresponding goto statement. The Java language can only express structured control flow, so by introducing unstructured gotos, the Java bytecode will express non-reducible flow graphs [4].

2.1.3 Using obfuscation

General obfuscators stumble into a couple of conflicts that makes obfuscation a subtle problem:

1. A general programming principle says: good programs are well-structured and have concise invariants. This is the exact opposite of what an obfuscator should aim for, as a good structure and concise invariants are used to understand and debug a program.
2. Program performance is often affected when obfuscated. Dead code will waste computation time as its results are never used. Jones, Gomard, and Sestoft call a specializer optimal with respect to interp if, for all programs, p, and data, d, one has for p'=[[spec]](interp,p): $\text{time}_{P'}(\text{d}) \leq \text{time}_{P}(\text{d})$ [3]. This is often not possible when obfuscating, as some obfuscation techniques work by adding bogus code as described in the deep obfuscation section. Instead, Giacobazzi, Jones, and Mastroeni define their own *less optimal* equation: $\text{time}_{P'}(\text{d}) \leq \text{c} * \text{time}_{P}(\text{d})$ [1]. This means that the obfuscation is acceptable for a reasonable small constant factor, c, dependent on the program, p.

2.2 Partial evaluation

Partial evaluation is a program transformation technique. The purpose of the techniques are most often to optimize programs by specialization. Partial evaluation will produce specialized programs that performs better than the original, while being guaranteed to behave the same way.

Jones, Gomard, and Sestoft describes the essence of partial evaluation, by letting a one-argument function be obtained from a two argument function by specialization, also written as: [[p]](s,d) = [[[[spec]] (p,s)]] (d) [3].

In Figure 2.4 a partial evaluator is given a subject program, P, and static input data, *InputS*. The partial evaluator will construct a new program *P'* that will yield the same results given the dynamic input, *InputD*, as the program P would yield given both *InputS* and *InputD*.

The static input, *InputS*, can be derived in many different ways. It can be input to a function or it could be a part of a function itself. A frequently occurring example is where one variable is changing more frequently than another [3]:

- a function f(x,y) is to be computed for many different pairs (x,y).

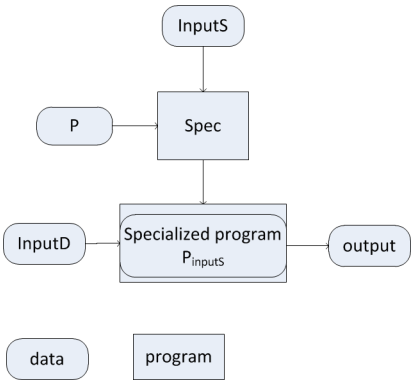

FIGURE 2.4: Show a partial evaluator, Figure 1.1 from [3]

- x is changed less frequently than y
- a significant part of fs computation depends only on x

The objective is to make program transformation by interpreter specialization, where the static input is a program. Giacobazzi, Jones, and Mastroen proves that semantics are preserved through interpreter specialization [1]:
[[p]](d)=[[interp]](p,d)= [[[[spec]](interp,p)]](d)= [[p']](d)
The algorithms are therefore inherited. The programming style however, is obtained through the interpreter, interp.

Partial evaluation have remarkable resemblance to deobfuscation. The static part of an obfuscated program corresponds to the code inserted by obfuscatiors. The dynamic part corresponds to the unobfuscated program. If the partial evaluator is able to identify the static part, this will be removed, leaving a deobfuscated program.

The usage of partial evaluation proves to cover a wide range of different goals, where computer graphics, database queries and neural networking are just some. In most cases, as well as all of these examples, the main motivation for partial evaluation is speed [3]. For this project, the first priority is for easy readability. However, many obfuscation techniques lower the efficiency of programs by inserting dead code or more branches. Since a deobfuscator will remove these, there is potential for speed optimization.

The three main program transformation techniques used for partial evaluation are symbolic computation, unfolding function calls, and program point specialization [3].

2.3 ANTLR

ANother Tool for Language Recognition, ANTLR, is a tool for constructing recognizers, interpreters and compilers. ANTLR will be the tool used to build the main components,

a lexer, a parser, and tree walkers, for the deobfuscator.

ANTLR provides support for tree construction, tree walking, translation, error reporting and error recovery [11]. ANTLR is generating parsers by taking a context-free grammar written in the Extended Backus-Naur Form (EBNF) [30]. One of the advantages of ANTLR, in contrast to other lexer and parser generators, is the consistent notation for specifying both lexers, parsers and tree parsers. Another reason to use ANTLR rather than other parser generators, is that ANTLR v3 has many ways to ease the grammar writing. One of them is its extension to LL(k) called LL(*) that allows arbitrarily far lookahead, that will dramatically increases the number of acceptable grammars.

```
methodDecl
        : modifier* returnType ID '(' args ')' body
        | modifier* returnType ID ';' // Abstract
        ;
```

FIGURE 2.5: No fixed maximum number of modifiers is specified. This leaves LL(k) recognizers unable to see past the modifiers.

The example in Figure 2.5 is non-LL(k) for any fixed k. No fixed amount of lookahead will be able to see past the modifiers as the grammar allows zero or more modifier symbols. Due to LL(*)'s cyclic implementation, it is able to look arbitrarily far ahead and will therefore allow such rules. LL(*) has a limit that makes it unable to handle some useful grammars. LL(*) uses Determanistic Finite Automation (DFA) to scan ahead, and as a result LL(*) is unable to make decitions on rules whose alternatives have recursive rule references. LL(*) would not be able to handle the rule in figure 2.5 if the return type rule is recursive as in Figure 2.6:

```
returnType
        : ID '<' returnType '>'
        | ID
        ;
```

FIGURE 2.6: The returnType rule is recursive making ANTLR's LL(*) recognizer unable to decite on the rule in figute 2.5

ANTLR allows other powerful methods for grammars where ANTLR is unable to build valid LL(*) recognizers. ANTLR's backtracking and syntactic predicate tell the parser to try an alternative. If the alternative fails, then ANTLR will rewind and try the next alternative until it finds a match. Such mechanisms are very powerful, but they come with an exponentially growing speed penalty in the worst case. Syntactic predicates have the extra advantage of being able to tell the grammar, that if it finds two alternatives, that both matches the same input phrase, then always take the first. If

an alternative depends on the context, ANTLR also provides semantic predicates. A semantic predicates are boolean expressions that turns alternatives on and off.

ANTLR mimics how the human mind recognizes text, by dividing the task into two separate phases. When reading, the brain does not read letter by letter, but instead aggregates character sequences into words and looks these up in a dictionary before recognizing the grammatical structure of the sentence. The first recognition phase is called a lexical analyser, scanner or simply a lexer. The lexer reads a stream of characters and produces vocabulary symbols, called tokens. Second phase is called parsing, and analyzes the output token stream from the lexer. The stream is turned into a parse tree, or a concrete syntax tree, that represents the syntactic structure of the input text. The third phase, that is often incorporated when working with lexers and parsers, is a tree translation phase. The parser syntax tree is transformed into an abstract syntax tree (AST), often omitting certain implicit elements. ANTLR combines the tree construction phase with the parsing phase. In addition, tree translations can be applied multiple times using tree walkers that performs computations on the AST. An emitter poses the last phase by outputting the final AST. For this purpose, ANTLR cooperates with the StringTemplate framework, a java template engine for generating formatted text output. This makes it possible for the emitter to have different templates, that each will emit the output in different ways. ANTLR uses this itself to generate lexers and parsers to different target languages [11].

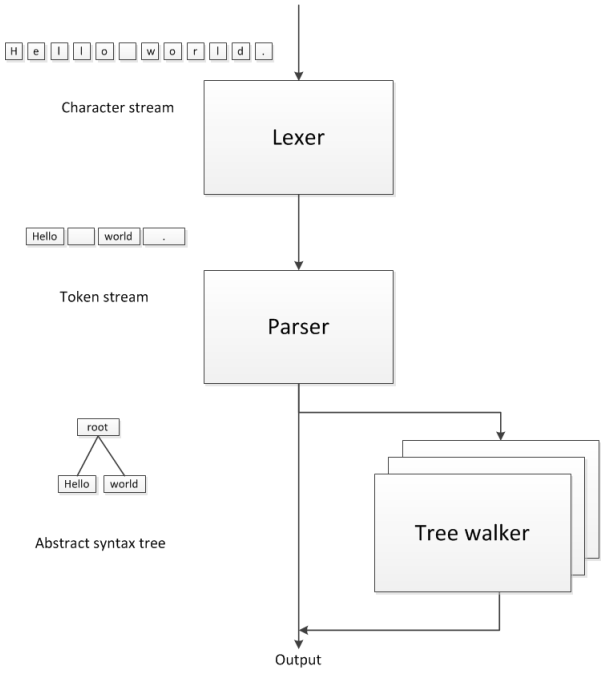

FIGURE 2.7: ANTLR control flow.

ANTLR supports a wide number of programming languages, but was originally written for Java. ANTLR has been tested for targeting other languages, but experience shows

that only Java can be trusted functional all the time. For that reason Java is the first choice when not affected by other constraints.

There are by now a substantial amount of grammars for different languages on the ANTLR website, but no grammar for the javap representation of java bytecode is available. This will have to be constructed.

2.4 The Java Virtual Machine

When Java source code is compiled, it is converted into an intermediate layer called Java Virtual Machine (JVM) bytecode. At runtime JVM will compile these intermediate instructions on the local environment by translating the generic bytecode into local architecture assembler code, e.g. x86 instruction. The JVM performs several different syntactic and structural checks on the bytecode to provide safety to the local environment. As well as verifying the bytecode, the JVM also manages hardware resources, like CPU consumption and memory usage. As the JVM only depends on the intermediate bytecode format, any language that can be expressed in terms of valid class files can be hosted by a JVM [14].

2.4.1 JVM Class Files

When a java program is compiled into JVM bytecode, each java class is compiled into a separate class file, with the distinct filename extension .class. The class file format precisely defines the representation of a class or interface, with data such as fields and methods. Class files can also contain various debug and meta data.

2.4.2 Javap

The javap executable that is shipped with Java Development Kit (JDK) is a Java bytecode disassembler. It is able to show information about class files in a detail level dependent on the user specified options. With no options specified, javap prints out the package, protected and public fields, and methods of the disassembled classes. Detailed description of javap is difficult to obtain, but the javap output is assumed to follow much of the JVM specification.

Chapter 3

Problem

3.1 The role of obfuscation

The word security in Security by obscurity implies that you are getting some protection. It also implies that you are not doing anything to remove attack vectors. This does not mean, in my opinion, that obfuscation and security by obscurity should never be used. All security is a question of risk management, and I will argue the same for security by obscurity. It is a question of how much value an attacker will get for his time. However, my personal opinion is that, when it comes to security critical systems, where the profit for an attacker is high, such as NemId, the focus should be on security principles that mitigate risk factors rather than hide them.

3.2 Problems with existing deobfuscators

Modern days obfuscators makes use of many different obfuscation techniques, and new techniques are continually developed to enhance the obfuscators [4] [8]. None of the tested deobfuscators are able to deobfuscate the very simple "HelloWorld" program provided in appendix B. The found deobfuscators will not do anything more than rename identifiers, leaving all control flow obfuscation be. Furthermore, none of the found deobfuscators provide an easy way to develop transformations for unsupported deobfuscations. To keep up with obfuscators, a deobfuscator should be open source and flexible, to allow further development when new techniques are encountered.

Chapter 4

Analysis

In the following sections, I will explain what I find to be criteria for good deobfuscation. And as a result of that, it will be explained why ANTLR was chosen as the prefered tool to make a deobfuscator as a partial evaluator.

4.1 Deobfuscation constraints

A deobfuscator is a program transformer, more specifically it must transform a program, P, into a new program, P', such that P' is easier to adapt and analyze. The criteria for good deobfuscation are:

1. *Semantics preservation:* Semantics preservation: $[[P']] = [[P]]$ is required.
2. *Automation:* P' is obtained from P without the need for hand work.
3. *Robustness:* All code valid to the JVM should be parsable by the deobfuscator.
4. *Readability:* P' is easy to adapt and analyze. For this obfuscator it means that the main decompilers, such as Dava [29], should be able to decompile P' back to source code while preserving semantics.
5. *Efficiency:* Program P' should not be much slower or larger than P.

Giacobazzi, Jones, and Mastroeni prove that many program obfuscations can be obtained by interpreter specialization, thus achieving the first two criteria [1]. In the same fashion, the aim will be to make deobfuscations by interpreter specialization, which will make the first two criteria achieved straight from correctness of the interpreter and specializer.

The deobfuscation itself does not need to be particularly fast. It does not matter if it takes a minute or a day as long as it is automated and can get the job done. Where exactly the threshold lies is up for discussion.

4.2 Design

The deobfuscation should be focused on Java bytecode. Two alternatives were considered in the initial design phase.

The first alternative involves using excisting bytecode analysis and manipulation frameworks such as BCEL, ASM or SOOT [16] [17]. The advantage with existing frameworks, is the immediate step past the development of a lexer and a parser. The frameworks constructs ASTs from the underlying bytecode and provide existing functionality for analysis, decompilation and optimization. Furthermore ASM supports both event driven and in memory processing, so the user can adapt the best way for the chosen task. The downside to incorporating a third party framework, is that it is difficult to find out if the tool is flexible enough. In this case, where the kind of deobfuscations can vary a lot, it is difficult to say, if either of the third party tools can provide the necessary flexibility.

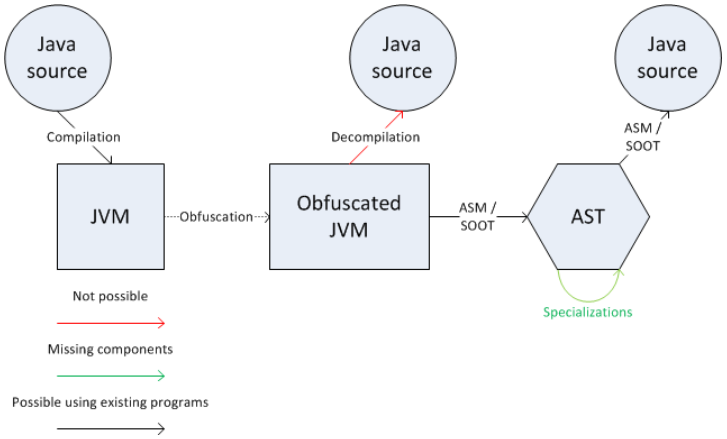

FIGURE 4.1: First alternative containing third party frameworks.

The second alternative is to use ANTLR to create the lexer and parser. The EBNF notation is very compressed compared to writing the actual code-behind, and the construction of the parser will give total control of every step of the generation. Furthermore, the deobfuscation specializations can be written as tree walkers. The decrease in size and the nature of EBNF syntax convince me, that it will be easier for developers to read and understand the structure of the code, as well as extend it with further deobfuscations in the future. There are a few disadvantages with using ANTLR. The lexer and parser will have to be acquired for a textual representation of the JVM bytecode, which require the syntax to be written. Furthermore, no exicisting frameworks that can transform javap

bytecode text to actual bytecode, have been found, thus making an AST to bytecode assembler a requirement for a fully functional deobfuscator.

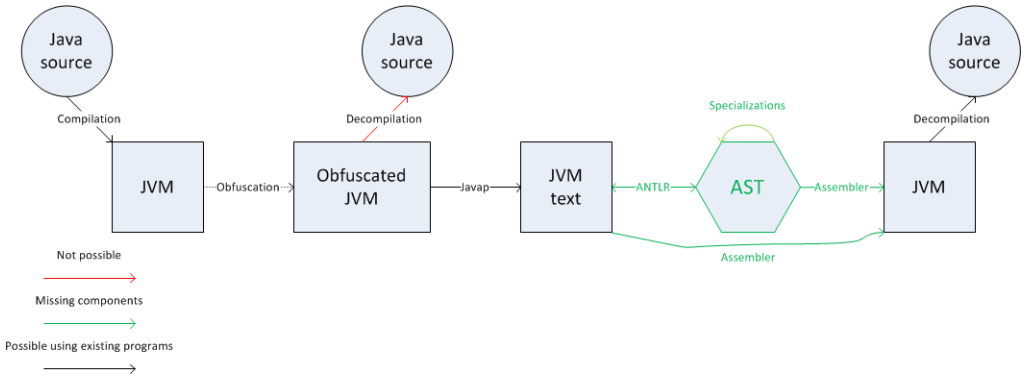

FIGURE 4.2: Second alternative with lexer and parser generated using ANTLR.

The choice for the final architecture fell upon using ANTLR. Especially the combination of extra control, flexibility and the ability for easy understanding, that allows further development in the future, weighs high. The extra work needed to create the lexer and parser seems reasonable and is a one time only job. It should also be noted, that it had my interest to get a deeper understanding of ANTLR. By this reasoning, the final architecture fell upon using ANTLR.

The components needed in the process, consist of a lexer and a parser for reading class files, a tree walker for printing out the textual representation of the bytecode, and a tree walker for constructing the JVM bytecode. For each distinct obfuscation technique that needs to be countered, a specialized tree walker needs to be constructed. The final deobfuscation process is pictured in figure 4.3.

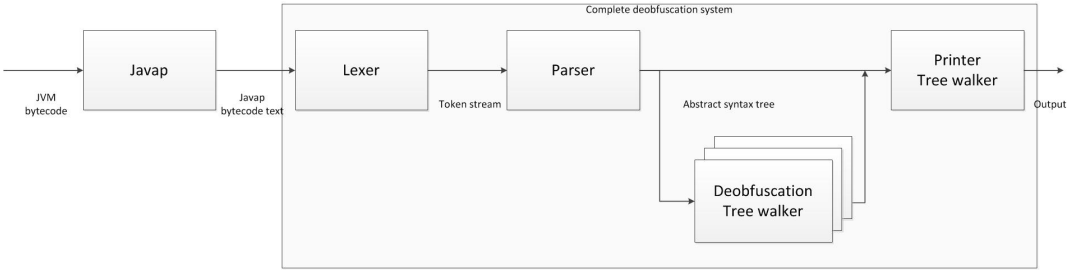

FIGURE 4.3: Data flow diagram for the architecture.

The system will take a javap bytecode text stream as input. The lexer and parser combined will create an AST. Tree walkers (or deobfuscators) will specialize the AST. The output AST is compared to the AST given as input, and as long as differences between the input tree and the output tree occur, another iteration will take place until the AST cannot be reduced or specialized any more. A final tree walker will then output a textual representation of the AST.

4.2.1 Specialization examples

4.2.1.1 Descrambling

In the process of descrambling java byte code, methods and fields will be treated different from constructors. Methods and fields are both referenced all over Java source code, but in bytecode, all references point to the constant pool, which contains the actual reference. The same is true for the constructor with the exception, that the name needs to follow the class and type name. The renaming of a constructor will require renaming all references to that type. The constant pool itself consists of constant pool types and values. All constant pool types, except base types and UTF8, are pointers to other constant pool types. A class, Foo, with a method, bar, with no return type, will contain at least six constant pool lines, consisting of:

1. Class	#4	//Foo
2. Method	#1.#3	//Foo.bar:()V
3. NameAndType	#5:#6	//bar:()V
4. UTF8	Foo	
5. UTF8	bar	
6. UTF8	V	

If the method, bar, is renamed, the corresponding constant pool line should change. When walking through the constant pool, all references will be constructed and stored with their tree tokens for fast access. The new names will all be constructed in a specialized tree walker. The tree walker will have to look up the references, and rename the constant pool tree tokens accordingly, for every renaming that takes place.

Renaming of class names and package names is more comprehensive, as these are references, not only from the constant pool, but everywhere types are used. It is possible only to register reference type names three places in addition to the constant pool. One for normal type names, a second for internal type names, and a third for generic descriptors. These three places should handle the map between reference type identifiers and their corresponding tree tokens.

```
Normal type: java.lang.Object
Internal type: Ljava/lang/Object;
Generic descriptor: <a extends java/lang/Object>
```

FIGURE 4.4: Bytecode reference types

The scrambling operation does two things:

1. It removes formal knowledge provided in namings.
2. It decreases the readability by letting several fields and methods look similar.

While renaming is a key component in descrambling code, the renaming itself will not necessarily make the identifiers meaningful, as it is not possible to restore the original names, that was carefully given by programmers in the first place, as scrambling is a one way operation.

It is not possible to reconstruct the lost domain knowledge, however, it is possible to add some information that will make it easier to read and understand. An acceptable descrambling should differentiate field names from method name etc., making people able to distinguish between types without having to look up further information than the name itself. Moreover, Java naming conventions should be followed to make the code overall easier to read. Following Java naming conventions will in itself add information in order to distinguish between constructs, e.g., classes and methods should follow different casings.

4.2.1.2 Control flow deobfuscation

Specializations can be constructed in many different ways dependent on the need. This example will take a Zelix obfuscation into consideration for a proposed way to deobfuscate it by specialization [12]. The complete bytecode for the obfuscation can be found in appendix B.

The analysis of the code obfuscated by Zelix, reveals the use of different obfuscation techniques. Figure 4.6 shows how Zelix uses two strong opaque predicates, b.a, and a.c, as marked with red. Zelix uses the bogus block, B, to point into the for-loop. Zelix has successfully transformed the control flow graph into a non-reducible control flow graph. To further enhance the obfuscation resilience, Zelix assigns values to the opaque predicate, a.c, with the bogus block. The result is that the decompiler, Dava, is unable to recognize the obfuscated structure of the bytecode. Both the before and after version of bytecode and source code is available in appendix B.

Zelix introduces public static opaque variables, by doing this, they assume that no reflection will occur, and that they are safe to use, as no threading is used. Zelix also require a complete program with one main method. If this is to be solved, one will have to make the same assumptions, that reflection and threading is not used to change field values, and that one main method will serve as the entrance point for the program. The next section proposes a way to solve the obfuscations using ANTLR.

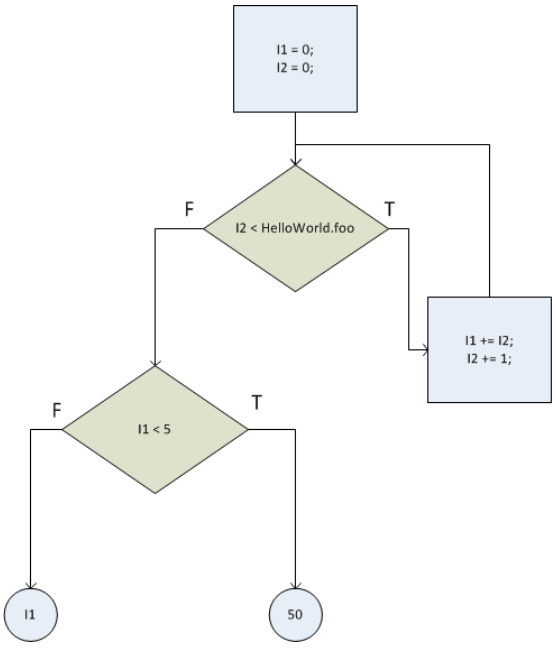

FIGURE 4.5: Method, foo, before obfuscation.

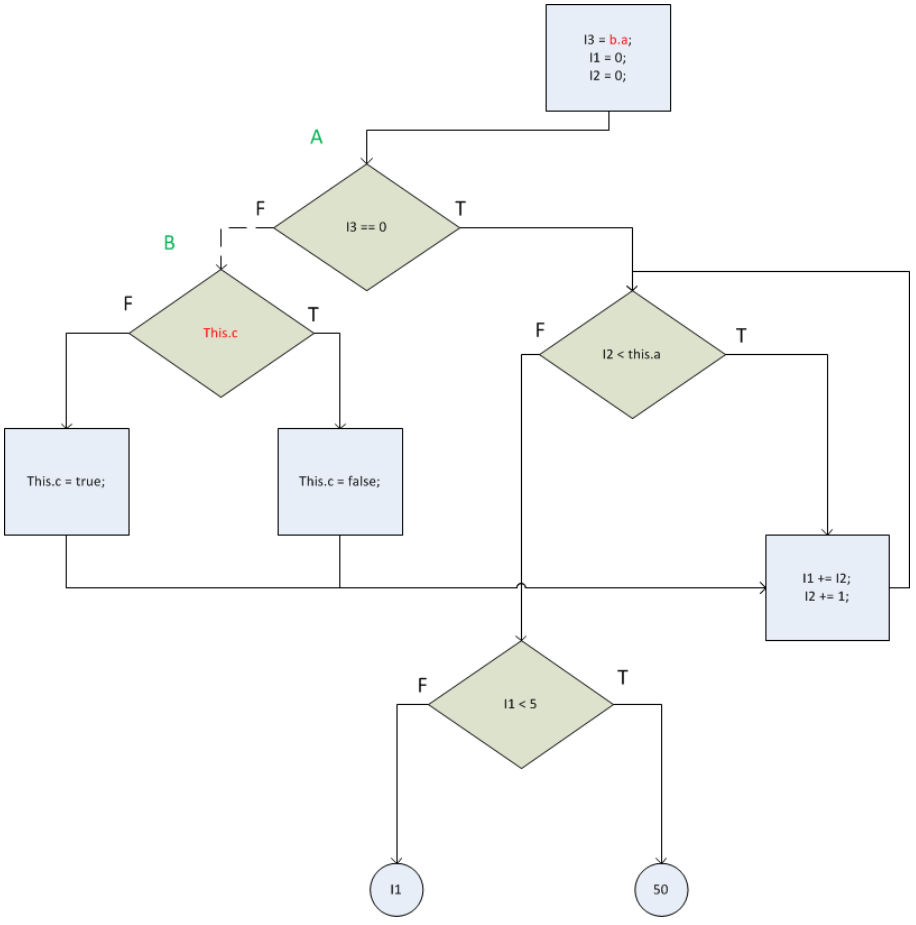

FIGURE 4.6: Method, foo, after obfuscation.

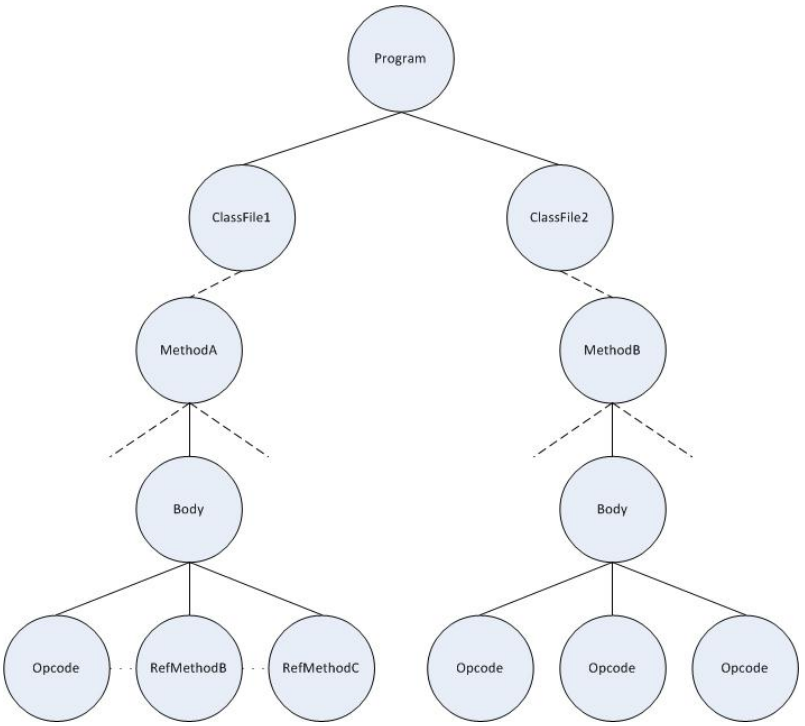

FIGURE 4.7: The figure shows a simplified version of how the ANTLR generated AST is thought to look like.

The use of strong opaque predicates, as described earlier, require inter-procedural interpretation. This will be solved by unfolding method calls, which as mentioned, is one of main program transformation techniques used for partial evaluation. Some logic will have to assure that recursive is not able to halt the deobfuscation.

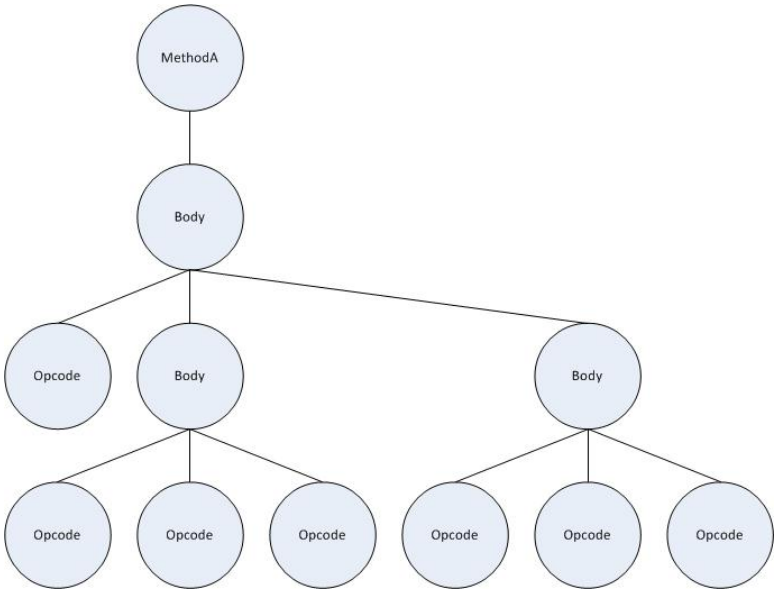

FIGURE 4.8: The figure shows an example of inlining methods in an AST.

Chapter 5

Implementation

5.1 Preconditions

The implementation of both the lexer and the parser is based on the output from javap, with the maximum amount of information. This can be retrieved by running javap with the following options specified: Javap -c -v -p -s -constants

ANTLRWork is the primary EDI when working with ANTLR, though the implementation of the lexer and parser have primarily been handled in Eclipse with the ANTLR plugin for easier cooperation with Java test code [10]. Shifting between EDIs disrupts the workflow, but using Eclipse on its own is unfortunately not an opportunity because of limitations. The ANTLR plugin currently does not support tree walker constructio,n and the documentation for getting the ANTLR plugin to work with Eclipse is not up to date. Further information about ANTLR setup can be found in appendix C.

5.2 Lexing

The lexer is primarily based upon the JVM and Java specifications [14]. This makes it easier to identify the meanings of tokens, and to some extent it is expected that the javap representation reflects the Java lexical structure specification [25]. To obtain all informations from the class files, javap has been provided with the following options: . Figure 5.1 shows how identifiers are constructed.

The Java and JVM specification describes how the actually Java and Java bytecode looks like. However, the javap representation of java bytecode is made for humans to read. This means that it does not necessarily conform to any specification, making the javap output less ideal for lexers and parsers [22]. For example, javap does not wrap

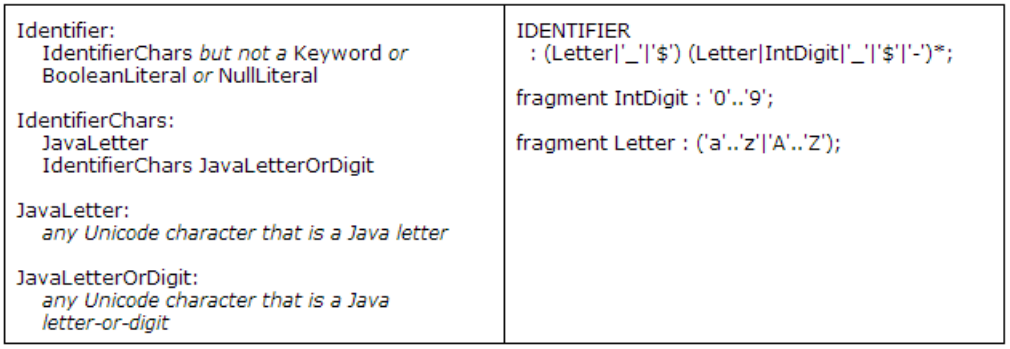

FIGURE 5.1: To the left, the lexical structure described in the Java specification. On the right side, the lexical structure described in the EBNF syntax.

string literals, placed in the UTF8 type in the constant pool, with quotes. This makes it very difficult for the lexer to determine when the string ends. The solution was to take all input characters from after the constant pool type, and treat those as one token. Figure 5.2 shows the mapping between a constant pool UTF8 type containing a string value and the grammar rule.

```
CONSTANT_TYPE_ASSIGNABLE
 : Constant_type (' ')+ (~('\n'|'\r'|' ')+ (' ')+)* ~('\n'|'\r'|' ')* '\r'? '\n'
 ;

#1 = UTF8       This is some made up text
```

FIGURE 5.2: The figure shows the mapping between a constant type lexer rule and an example of a constant type from javap output.

Javap does not escape backslashes. The backslash character in a javap text file is therefore written '\'. However, the escaped single quote is written '\" making it impossible for the lexer to distinguish between the two. At this point unescaped backslashed is not supported by the implemented lexer.

5.3 Parsing

The parser is also constructed as close to the JVM specification as possible. This will make it easier for one to compare the structure of the parser syntax to the JVM specification, and therefore make the whole understanding easier. As described, there are problems when using the javap representation. There is no definition for what can be expected in the javap output and because of that, the approach for constructing the parser grammar was mostly trial and error. As a result, the parser is less likely to be able to parse new languages compiled to Java bytecode.

As described, ANTLR provides different functionalities to help the user when creating the grammar. In some cases it is impossible to build a grammar for a given language,

without using either backtracking, semantic predicates or syntactic predicates. But the functionality comes with a cost. Beside making the parsing slower it also disables the interpreter that is shipped with Antlrworks, and debugging a syntax that contains backtracking or syntactic predicates is a tedious job for larger syntaxes. I have been able to limit the use of syntactic predicates to one place. The last occurance is due to the recursive nature of generic type names, which leaves the lookahead unusable, and the fact that the first guaranteed difference between methods and field are the parenthesis in methods, which appears after the return type:

referenceType field;
referenceType method();

In some cases there exists ambiguities in the parsed language itself. As with the english language, symbols can be added to remove ambiguities and reduce the need for backtracking. The javap bytecode required this technique one place. The DFAs in appendix D show that the fieldDefinition rule will possibly end with an identifier. An extra identifier is at the same time the only thing that is guarantied to distinguish methods from constructors. If the parser sees the valid text, as shown in Figure 5.3, in a class file, it will be unable to determine whether the identifier is a flag or a return type. This will end up with the parser always making one of two decisions and discarding the other option. To avoid this, it was decided to add a custom symbol,

```
private int field1;          private int field1;
flags: ID                    flags:

ConstructorName()            ID MethodName()
```

```
private int field1;
flags: ID
|
ID MethodName()
ConstructorName()
```

FIGURE 5.3: The figure top left and right example show the cases where an ID token is ambiguous. The bottom example shows the used solution.

5.4 Rewriting

Parr provides general guidelines for creating proper AST structures [11]. In addition, the rewrite rules have been build with the following guidelines in mind:

1. The amount of imaginary nodes should be held to a minimum. It is very easy to create a large number of imaginary nodes which does not add meaning to the AST structure. If a meaningful token can be used, there is no reason to create an imaginary node for the same purpose.
2. Tree walkers do not look ahead. To meet this constraint, any tree walker rule, where the cardinality is zero or more, should be placed as the last node or in a new subtree. Imaginary nodes will often serve as the root of subtrees generated for this purpose. The bottom rewrite rule in Figure 5.4 shows how imaginary nodes can be used to make a rewrite rule acceptable.

```
rule
  :  root subrule1? subrule2 ->                BAD!
      ^(root subrule1? subrule2)
  ;

rule
  :  root subrule1? subrule2 ->
      ^(root ^(SUBRULE1 subrule1?) subrule2)    GOOD
  ;
```

FIGURE 5.4: The figure shows an example of a bad rewrite on top and an acceptable in the bottom.

Implementing with these rules in mind will help the tree walker handle the AST better. A clean tree walker is placed within the project to make future specialization development faster.

5.5 Obfuscations

Two initial obfuscations have been developed. The first obfuscation is able to rename method names and field names.

The tree walker stores the class name in each class scope. The class names are used to construct complete identifier names, when renaming methods and fields. When the tree parser walks through the constant pool, it will store information about each of the lines. After the walker has scanned through all constant pool lines, it assembles information into the complete identifier names. For the example shown in the analysis, this means the assembly of class name, method name, return type and argument types.

The deobfuscator assumes that all code have been obfuscated and will rename every method upon tree walker recognition. The tree walker will replace the existing token, that represents the method name, with a new custom named token. The change will also

be reflected in the constant pool reference, by replacing the name token. The identifier generation logic is at this point limited to describe whether it is a field or a method, and give it an incremental number.

Due to time pressure, the scope of the second deobfuscation was reduced to focus on static local analysis. Figure 5.5 shows how the deobfuscation scans opcodes for If (false) blocks and removes dead code, if found. It then reassembles the opcodes before and after the removed block and changes the indexes accordingly. The example is unlikely, as a conditional branch like the one in figure 5.5 would not rely on a constant value, but rather on one or more opaque variables. The green lines in the figure indicates restructuring of indexes.

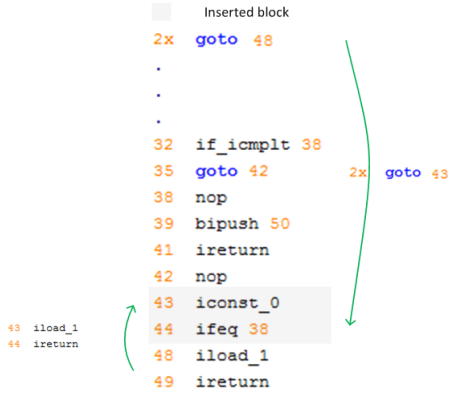

FIGURE 5.5: The green lines shows where indexes need reordering. The darkened block shows the inserted code that is to be removed.

Chapter 6

Validation

To evaluate the product, two parameters is measured, robustness and correctness. If a deobfuscator is not robust, or if an obfuscator knows the flaws of a deobfuscator, then the obfuscator can use targeted preventive transformations to make the deobfuscator unable to parse. The correctness should ensure semantic preserving, as it is very important that no changes occur without intention, in order to maintain the usefulness of the deobfuscator.

The product is validated as a whole as the components are dependent of each other. For example, it is not obvious whether there is a problem in the lexer or parser, when an unexpected lexer token appears. Furthermore, a test suite was established, that consists of four large java bytecode libraries: Scala, the Java Runtime Environment (JRE), Groovy and JRuby. Combined over 50.000 class files exist in the test suite.

6.1 Robustness

To validate the robustness, the test program runs the lexer, parser, and the clean tree walker on each file in the test suite. If no output or exception is recorded in the test flow, the test is assumed positive. Through the whole test suite, there were only nine files that failed to be parsed. Among those who failed, three different errors were identified:

Basic types used as identifiers.	*public string byte;*
Keywords used as identifiers.	*public string public;*
No escapes for backslash.	*public char escape = '\';*

The JVM does not have the same keywords as specified for the Java language. This makes it possible for Java bytecode compiled from other source languages to contain

these identifiers. The parser should not specify the Java language constraints if they are not reflected in the JVM specification.

There are other limitations that could potentially harm the parsing, but none have been found in the test suite so far. The limitations are further discussed in the known limitation section.

6.2 Correctness

Figure 6.1 shows the data flow when the correctness is measured. The correctness is measured by comparing lexer tokens from the original input, with the lexer tokens from a pretty printed text after parsing. This ensures that every token that contributes to the class is compared. All white spaces, new lines, and comments are filtered out. If the token streams from before and after parsing are unequal, the test fails. Through the iteration of all the successfully parsed files, no inconsistencies were observed.

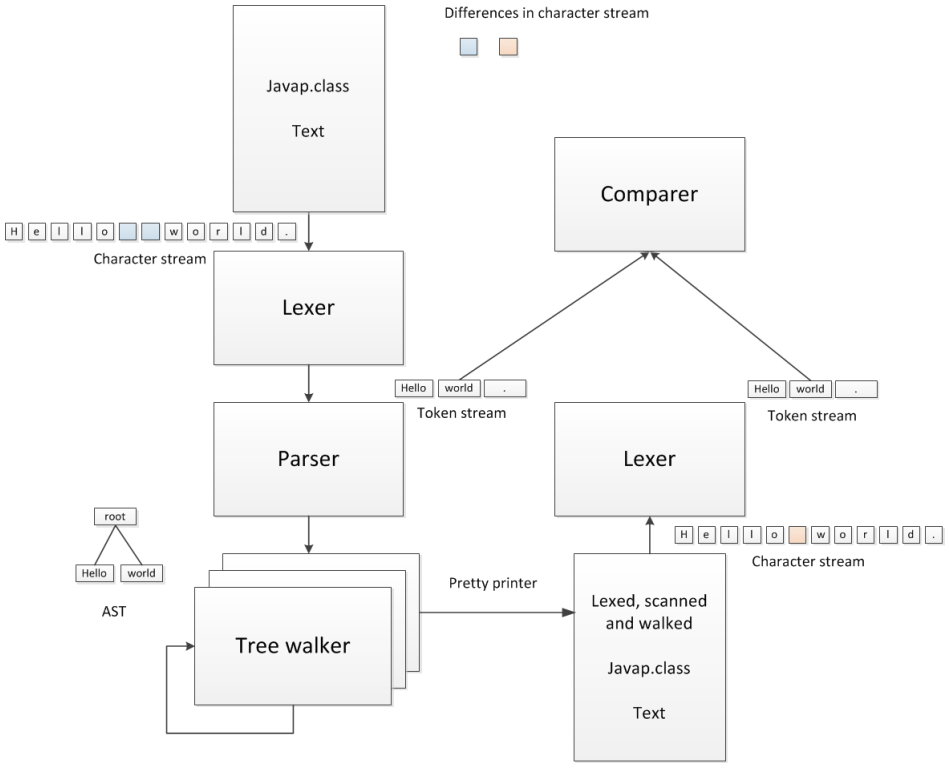

FIGURE 6.1: The figure shows what data are compared and from where it is taken.

The tests indicate that the parser and tree walker delivers high correctness, and as long as no filtered tokens contributes to the JVM interpretation, so does the lexer. A complete test with real bytecode will have to confirm this once the assembler has been made.

Chapter 7

Discussion

7.1 ANTLR vs third party parser

Throughout the creation of the deobfuscator, a substantial amount of time have been used on debugging the parsed javap output. If one was to pursue a similar approach, I would recommend thinking twice before deciding on using ANTLR to parse javap disassembled bytecode. Bytecode ASTs can be acquired by frameworks such as Soot [16], and they may prove to be sufficient. However, personal experience shows, that it can be difficult to predict the limitations of frameworks, until you get to learn them in various contexts.

Parsing the real Java bytecode have the advantage of detailed documentation that defines which structures can be expected. For example, Java virtual machines are required to ignore any unrecognized custom attributes. This is easy when they have to conform to a specific structure. The javaps representation may not necessarily conform to any specific structure which makes it more difficult to parse. At this point, only two custom attributes have been verified. The possible appearance of custom attributes should be examined to verify that the deobfuscator will not break when scanning a new custom attribute.

7.2 Future work

The highest priority for future work is to create a bytecode assembler. Without this it is difficult to verify that the generated bytecode is valid w.r.t. a JVM. The assembler can be made as a tree walker to replace the current pretty printer. For the purpose of writing an assembler, a mapping between text and bytecode can be found in the JVM

specification [14]. Once an assembler is constructed, the performance for the obfuscated and deobfuscated code should be verified, to see if it lives up to the fourth point in the deobfuscation constraints section.

Much information can be placed into the structure of abstract syntax trees. When parsing Java source code, the structure of the AST is able to specify precedence of control flow constructs. When parsing Java bytecode, however, there is possibility for unstructured gotos. In order to address the issue of unstructured gotos, articles regarding verification of unstructured control flow graphs, and articles about pattern matching should be further investigated. Pattern matching may also help with the naming of methods by revealing known structures in the control flow.

7.3 Other uses

Some obfuscation techniques e.g., inserting bogus code, behaves similar to errors that can be expected by a rookie programmer. An interesting idea is to let different specializations record all transformations when used to transform programs written by inexperienced programmers. The deobfuscator is then able to describe which specializations that were used the most, and the number of transformations done by each specializer. The deobfuscator will then proof suitable as a validation tool, for any source language that compiles into java bytecode.

7.4 Known limitations

The descrambling is not able to rename packages. An improvement will be to rename packages and place classes with high cohesion in distinct packages. The renaming should be configurable, as not all obfuscators may change package names, and renaming of existing names will make the code less readable instead.

The very first line in a javap class file contains the physical location on the local environment. As the grammar has been written on a machine running windows, the path syntax corresponds to a windows path. Support for other major operating systems should be added to the grammar. The limitations mentioned in the sections 5 and 6, should be refined, before this deobfuscator is considered ready. Right now te deobfuscator will only require an obfuscator to insert a string or char literal, containing a single escaped backslash, to break the parser. For existing obfuscations, it will very likely not mean anything, as these escapes so rarely appears in bytecode.

Chapter 8

Conclusion

This paper presents a lexer and a parser for javap output, as well as presenting an approach to implement different deobfuscation techniques. The product is robust enough to handle 99.9% of all files in three large bytecode libraries combined, without losing information from any file. Despite the limited product, I feel that I have found the recipe to further development of the deobfuscator.

I have reached my personal goals in the sense, that I have gotten a much better understanding of both ANTLR and Java bytecode. ANTLR is a powerful tool for constructing parsers and transforming abstract syntax trees. However, javap is made for the sake of human readability, and does not comply to any specification. To construct a syntax that is guarantied to work is not possible. The experience has also showed me, that every compiled source language contain their own small javap output diversities, that makes the construction of the syntax more difficult.

While there is room for improvements of the deobfuscator, as described in the future work section, I think that the product will provide strong fundamentals for a complete implementation of a flexible deobfuscator. I am in no doubt that this product can go a long way, and that it will benefit from the control and flexibility that comes along with being implemented from scratch. However, if others were to develop a similar product, I would recommend them to turn away from javap, and either construct a parser for Java bytecode, or to investigate the alternative of using an existing framework.

Appendix A

Appendix

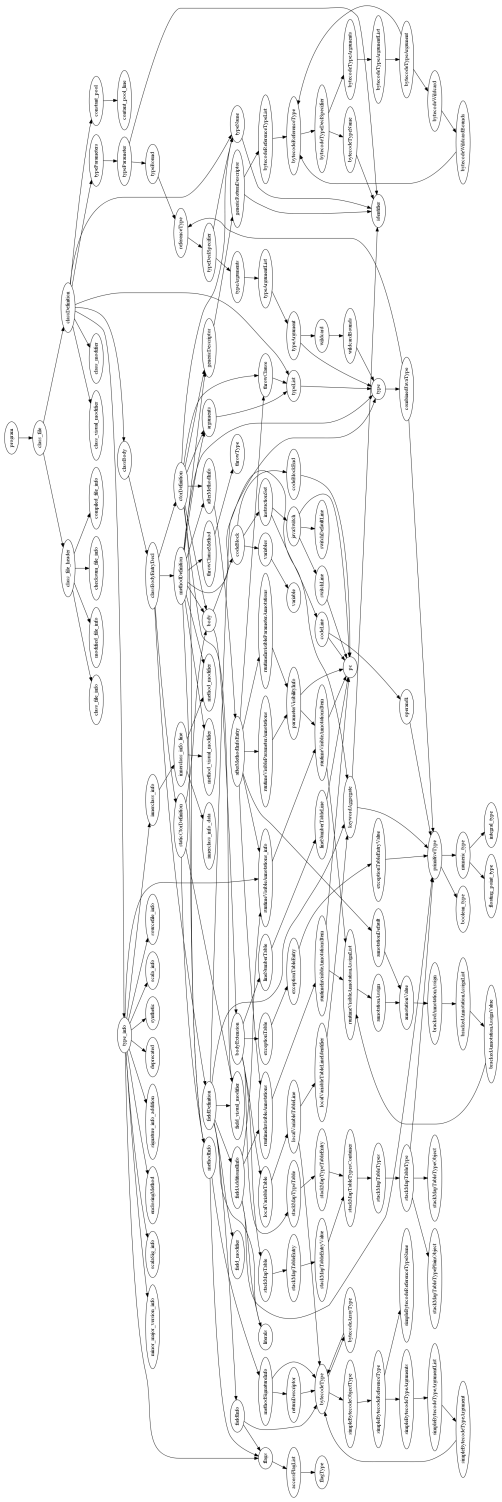

FIGURE A.1: The graph shows all parser rules and their connections currently in the grammar. A higher solution can be found on the provided dvd in the images and figures folder.

Appendix B

Appendix

Original main

```
package OriginalJava;
import java.io.IOException;
public class ObfuscatorTest {
        public static void main(String[] args) {
                HelloWorld world = new HelloWorld();
                try {
                        world.Bar(System.in.read());
                } catch (IOException e) {
                        e.printStackTrace();
                }
                System.out.println(world.Foo());
        }
}
```

Obfuscated (decompiled) main

```
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintStream;

public class b
{
    public static int a;

    public static void main(String[]  r0)
    {
        a r2;
        r2 = new a();
        try
        {
            r2.a(System.in.read());
        }
        catch (IOException $r5)
        {
            $r5.printStackTrace();
        }

        System.out.println(r2.a());
    }
}
```

FIGURE B.1

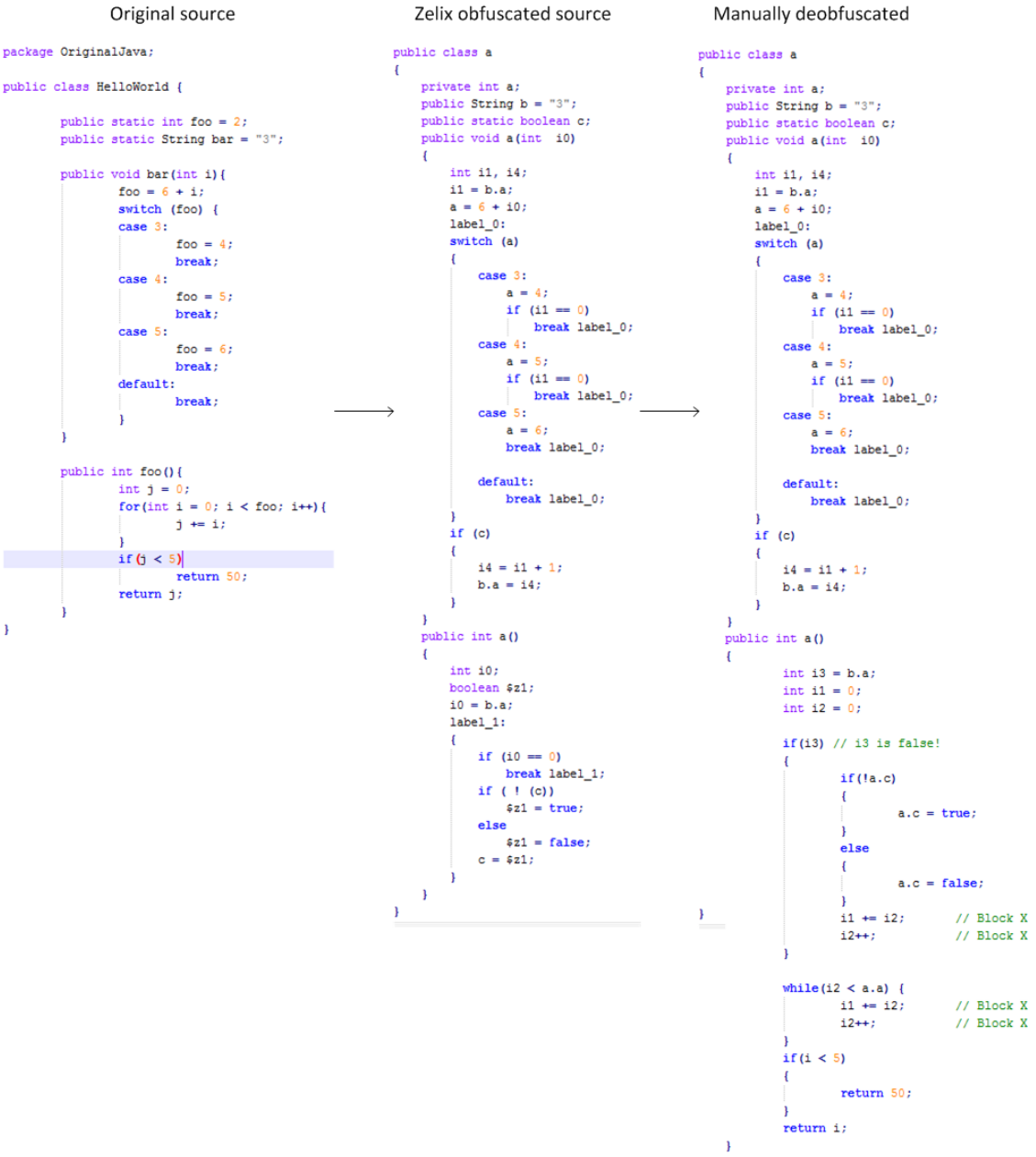
Original source
Zelix obfuscated source
Manually deobfuscated

FIGURE B.2

Original bytecode

```
public class HelloWorld {

  private int foo;
  public java.lang.String bar;

  public void bar(int arg0);
    0  aload_0 [this]
    1  bipush 6
    3  iload_1 [arg0]
    4  iadd
    5  putfield HelloWorld.foo : int [3]
    8  goto 46
   11  nop
   12  aload_0 [this]
   13  iconst_4
   14  putfield HelloWorld.foo : int [3]
   17  goto 80
   20  nop
   21  aload_0 [this]
   22  iconst_5
   23  putfield HelloWorld.foo : int [3]
   26  goto 80
   29  nop
   30  aload_0 [this]
   31  bipush 6
   33  putfield HelloWorld.foo : int [3]
   36  goto 80
   39  nop
   40  goto 80
   43  goto 80
   46  nop
   47  aload_0 [this]
   48  getfield HelloWorld.foo : int [3]
   51  nop
   52  tableswitch default: 39
         case 3: 11
         case 4: 20
         case 5: 29
   80  nop
   81  return

  public int foo();
    0  iconst_0
    1  istore_1
    2  iconst_0
    3  istore_2
    4  nop
    5  iload_2
    6  aload_0 [this]
    7  getfield HelloWorld.foo : int [3]
   10  if_icmplt 16
   13  goto 29
   16  nop
   17  iload_1
   18  iload_2
   19  iadd
   20  istore_1
   21  nop
   22  iload_2
   23  iconst_1
   24  iadd
   25  istore_2
   26  goto 4
   29  nop
   30  iload_1
   31  iconst_5
   32  if_icmplt 38
   35  goto 42
   38  nop
   39  bipush 50
   41  ireturn
   42  nop
   43  iload_1
   44  ireturn
}
```

Obfuscated bytecode

```
public class a {

  private int a;
  public java.lang.String b;
  public static boolean c;

  public void a(int arg0);
    0  getstatic b.a : int [28]
    3  istore_2
    4  aload_0 [this]
    5  bipush 6
    7  iload_1 [arg0]
    8  iadd
    9  putfield a.a : int [19]
   12  aload_0 [this]
   13  getfield a.a : int [19]
   16  tableswitch default: 71
         case 3: 44
         case 4: 53
         case 5: 62
   44  aload_0 [this]
   45  iconst_4
   46  putfield a.a : int [19]
   49  iload_2
   50  ifeq 71
   53  aload_0 [this]
   54  iconst_5
   55  putfield a.a : int [19]
   58  iload_2
   59  ifeq 71
   62  aload_0 [this]
   63  bipush 6
   65  putfield a.a : int [19]
   68  goto 71
   71  getstatic a.c : boolean [30]
   74  ifeq 84
   77  iinc 2 1
   80  iload_2
   81  putstatic b.a : int [28]
   84  return

  public int a();
    0  getstatic b.a : int [28]
    3  istore_3
    4  iconst_0
    5  istore_1
    6  iconst_0
    7  istore_2
    8  iload_3
    9  ifeq 33
   12  getstatic a.c : boolean [30]
   15  ifeq 22
   18  iconst_0
   19  goto 23
   22  iconst_1
   23  putstatic a.c : boolean [30]
   26  iload_1
   27  iload_2
   28  iadd
   29  istore_1
   30  iinc 2 1
   33  iload_2
   34  aload_0 [this]
   35  getfield a.a : int [19]
   38  if_icmplt 26
   41  iload_1
   42  iconst_5
   43  if_icmpge 49
   46  bipush 50
   48  ireturn
   49  iload_1
   50  ireturn
}
```

FIGURE B.3

Appendix C

Appendix

ANTLR 1.4.3 was used throughout the creation of this project.

For testing purposes, especially for testing tree walkers, I recommend setting ANTLR up in Eclipse. The following video provides the best guide I have been able to find: http://javadude.com/articles/antlr3xtut/, under the Prologue - Getting Eclipse set up for ANTLR 3.x development section. This should be sufficient a long way. However, I had problems with a few steps that with regards to the video, in what should happen automatically. In the project folder, one will manually have to configure the classpath and project configuration file. To show these files one must first click on the menu (downwards pointing triangle), select filters and then unselect .*resources. The files are now showing and should be configured as in the movie.

Appendix D

Appendix

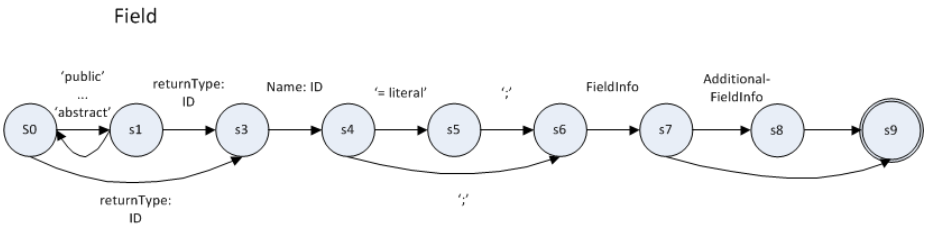

FIGURE D.1: Field DFA.

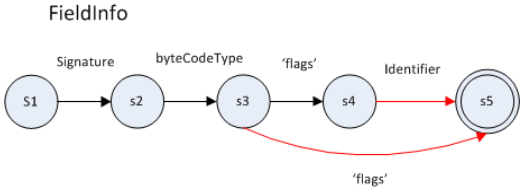

FIGURE D.2: The fieldInfo rule ends with an optional identifier.

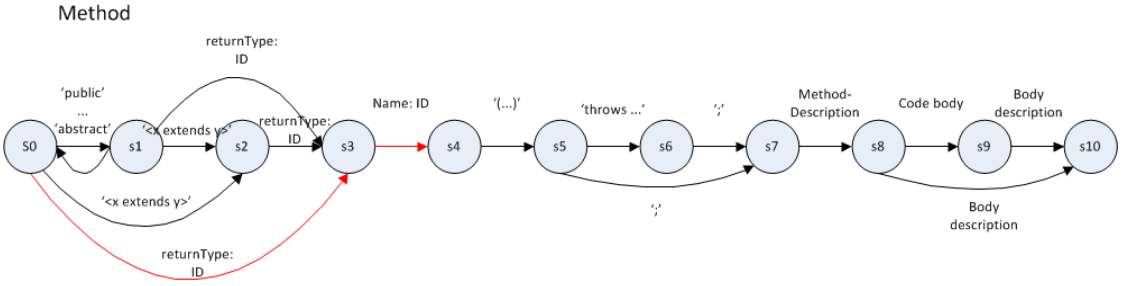

FIGURE D.3: Methods will possible begin with two identifiers.

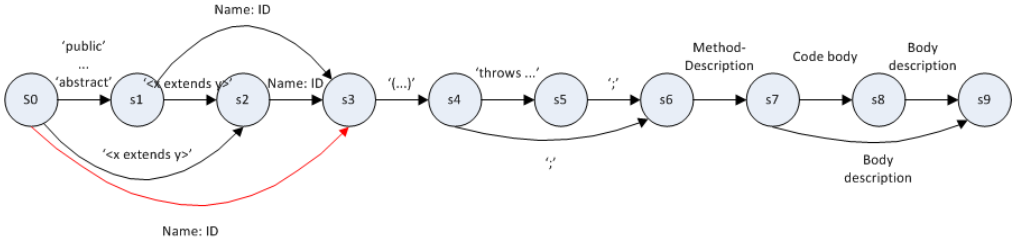

FIGURE D.4: Constructors will possible begin with one identifier.

Appendix E

Process

The goals for the thesis were higher than what have been achieved. The decision to construct a javap parser using ANTLR was made early on, together with Joseph Kiniry. The first two months were spent on digging into Java bytecode, doing manual deobfuscations, exploring the field of partial evaluation, and last, but certainly not least, learning Antlr. It took me almost two and a half months to create a lexer and a parser that were able to parse over 99% of the disassembled files. The remaining time included the writing of this report, and the construction of the two tree walker deobfuscations.

Most of the syntax had to be created using a trial-and-error approach, which resulted in a very tedious job. The goals for the thesis have had to be downgraded, as the process left not enough time to implement an interpreter, nor any deobfuscations that are of great help for the observed obfuscations.

Appendix F

The deobfuscator

The produced code is placed inside ThesisDeobfuscator.rar and ThesisDeobfuscator.zip, the contents of the two files are identical.

The grammar for the lexer and the parser is named JVM.g. The class used to test the grammar is called JVMRunner.java. The local reduction deobfuscation consists of both a tree walker and a java class, and is called OrFalseReduction. The descrambler is called JVMScramblingInformationGatherer and is limited to the tree walker alone.

All figures and images are places in the - Images and figures - folder.

The unparsable files are submitted for possible further inspection.

The three none-java source libraries are submitted in the BatFiles folder along with the bat-files used to unpack and disassemble them.

Bear in mind that both tests and most bat files use local paths to files, if they are to be used, then remember to change the destinations.

Bibliography

[1] R. Giacobazzi, N. Jones, I. Mastroeni. Obfuscation by Partial Evaluation of Distorted Interpreters. PEPM12, Philadelphia, PA, USA, 2012-1-2

[2] N. Jones. Obfuscation by Partial Evaluation of Distorted Interpreters (Slide), URL: http://dansas.sdu.dk/slides/Jones-SlidesObfuscation.pdf, accessed at 2012-2-1

[3] N. Jones, C. Gomard, P. Sestoft. Partial Evaluation and Automatic Program Generation. 1993.

[4] C. Collberg, C. Thomborson, D. Low, A Taxonomy of Obfuscating Transformations. Auckland New Zealand.

[5] S. Udupa, S. Debray and M. Madou. Deobfuscation: Reverse Engineering Obfuscated Code. WCRE05, Tucson, AZ 85721, USA (2005).

[6] Business Software Alliance, URL: http://portal.bsa.org/globalpiracy2011/, accessed at: 2013-1-4

[7] Version2, URL: http://www.version2.dk/artikel/nyt-netbank-indbrud-trods-nemid-hvordan-gjorde-tyvene-det-43477, accessed at: 2013-1-4

[8] Java ByteCode Obfuscator, URL: http://www.excelsior-usa.com/articles/java-obfuscators.html, accessed at 2013-1-5

[9] ANTLR, URL: http://antlrv3ide.sourceforge.net/, accessed at 2013-1-7

[10] ANTLRWorks, URL: http://www.antlr3.org/works/, accessed at 2013-1-7

[11] T. Parr. The Definitive ANTLR Reference. Version 2011-3-24

[12] Zelix, URL: http://www.zelix.com/, accessed at 2013-2-11

[13] Proguard, URL: http://proguard.sourceforge.net/, accessed at 2013-2-11

[14] The Java Virtual Machine Specification, URL: http://docs.oracle.com/javase/specs/jvms/se7/html/index.html, accessed at 2013-2-11

[15] M. Gmez-Zamalloa, E. Albert, G. Puebla. Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation. University of Madrid (2007)

[16] P. Lam, E. Boddeny, O. Lhotk and L. Hendren. The Soot framework for Java program analysis: a retrospective.

[17] E. Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns.

[18] Technet, URL: http://technet.microsoft.com/da-dk/magazine/2008.06.obscurity(en-us).aspx, accessed at 2013-2-12

[19] G. Naumovich, N. Memon. Preventing Piracy, Reverse Engineering, and Tampering. IEEE Computer Society (2003).

[20] N. Naeem, M. Batchelder, L. Hendren. Metrics for Measuring the Effectiveness of Decompilers and Obfuscators. (2006)

[21] C. Nachenberg. Understanding and Managing Polymorphic Viruses. 1996 Symantec Corporation.

[22] Jasmin, URL: http://jasmin.sourceforge.net/about.html, accessed at 2012-12-14

[23] Prosa, URL: http://www.prosa.dk/aktuelt/nyhed/artikel/it-sikkerhedsraad-kritiserer-nemid, last accessed at 2013-2-16

[24] Security through obscurity, URL: http://en.wikipedia.org/wiki/Security_through_obscurity, last accessed at 2013-2-16

[25] Java specification, URL: http://docs.oracle.com/javase/specs/jls/se7/html/index.html, last accessed at 2013-2-17

[26] NIST Computer Security Division (CSD), NIST SP 800-123, Guide to General Server Security, NIST Special Publication 800-123, July 2008

[27] Kerckhoffs' Principle, URL: http://en.wikipedia.org/wiki/Kerckhoffs's_principle, last accessed at 2013-1-20

[28] Dotfuscator, URL: http://msdn.microsoft.com/en-us/library/ms227240(v=vs.80).aspx, last accessed at 2013-2-17

[29] Dava, URL: http://www.sable.mcgill.ca/dava/, last accessed at 2013-2-26

[30] EBNF, URL: http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form, last accessed at 2013-2-12

[31] JBCO, URL: http://www.sable.mcgill.ca/JBCO/, last accessed at 2012-12-8

[32] yGuard, URL: http://www.yworks.com/en/products_yguard_about.htm, last accessed at 2013-2-20