CS252 Project Report

Ujjawal Garg
San José State University
ujjawal.garg@sjsu.edu • (408)-752-6034

December 12, 2017

Abstract

The race for innovation in the software industry is getting more and more competitive. In this competitive environment, the protection of Intellectual Property (IP) is of utmost priority. According to a 2013 study done by International Data Corp., the software industry spends more than $114 billion each year to tackle Software Reverse Engineering (SRE) attacks. While legal tools like copyright and patents provide some measure of protection, most software companies lack the resources need for these legal battles. In such cases, it becomes important to protect the IP theft from happening in the first place. Code obfuscation is one of the most popular tools used to accomplish this task. This project implements a tool in Haskell that can obfuscate a Java source code.

1 Code Obfuscation

Code obfuscation is a process in which we deliberately create source code that is difficult for humans to understand. This is achieved by applying various obfuscation transformations that convert the source code from an easy to understand form into a form that is much difficult to understand. There are several obfuscation techniques [1]. Some of most commonly used techniques are described below:

1.1 Identifier Renaming

In this technique, the identifiers in a program are replaced with meaningless or confusing names. This makes the task of reverse engineer much more difficult as the information embedded in the original names are now not available to him.

1.2 Control Flow Transformation

In this technique, the original sequence of instructions in the program is jumbled by introducing jump sequences in the code. These jumps are determined by some opaque predicate values. Opaque predicate are expressions whose runtime value is known to the obfuscator, but is much difficult to guess for the reverse engineer.

1.3 Dead Code Injection

In dead code injection, junk code is inserted at various points of the program. If the hacker tries to tweak the parameters in a program, it could lead to the execution of this junk code, which usually results in a runtime error. This makes the job of hacker much more difficult, as he will now need to start over again.

1.4 Convert Static to Procedural Data

In this technique, information embedded in static data (e.g. strings) is removed by converting the string literal into method call to a program that gives back the same string literal.

2 Project description

The goal of this project was to get knowledge about the various obfuscation techniques, and how one of these techniques could be implemented in practice. For this project, the *identifier renaming* technique described in 1.1 was chosen because it performs quite well based on the metrics defined by Collberg [1]. He defined the following software metrics to assess the performance of an obfuscation:

- **potential** which measures the complexity of the code in human understanding;
- **resilience** which measures the difficulty of performing the de-obfuscation using automated tools;
- **cost** which measures the time and memory space required to perform the obfuscation.

The identifier renaming technique has high potential, one-way resilience, and zero cost. So, this is one of the most important obfuscation technique.

2.1 Why Java?

A Java source code is never really compiled to native code. Rather it is distributed in the form of bytecode. This bytecode can be easily converted back to a java program using popular IDEs like IntelliJ IDEA. So, obfuscating the logic of a program is much more important in Java than in other languages like say C++, where code is compiled to native binaries which are much harder to decompile. For this reason, Java was chosen as the choice of source code language that would be obfuscated.

2.2 Why Haskell?

The parsec library was already covered in class material. It was decided to use this existing knowledge instead of butting heads with a new tool.

3 Implementation Details

The following steps were used to perform the obfuscation:
Step 1: Construct the AST of the given source code
Step 2: Apply the transformations to the AST
Step 3: Change AST back to source code (obfuscated)

3.1 Generating the AST

For the first step, the structure of the Abstract Syntax Tree (AST) was needed to be defined. For this purpose, the syntax rules defined in the Java Language Specification [2] was used. Since the syntax of the complete Java language would be too large, only a small subset of features was implemented. This structure for the syntax tree can be found in Syntax module. The following types of statements are supported:

1. Variable declarations (both primitives and Objects)
2. Variable assignments
3. Arithmetic operations
4. Method calls
5. return statements

Apart from these the package declaration, import statements, and class declarations are also read into the AST. This reading is performed by the Parser module.

3.2 Performing the obfuscation

In order to perform the obfuscation, a Data.Map variable called *mapping* is maintained. This variable is initially empty. Whenever an identifier is encountered in the AST, it is checked whether this identifier is already present in the map. If yes, then this obfuscated value is what actually gets written to the new file instead of the original value. If not, then it is determined whether the identifier should be obfuscated or not. There are two types of identifiers that are excluded from obfuscation:

1. All the '.' separated identifiers. This is done to separate the identifiers that originated outside of this class. e.g. an identifier like *BigInteger.ONE* or *System.out.println* should not be obfuscated.

2. Reserved identifier like "main" method etc.

If it belongs to neither of these category, then a random 10-digit string is generated and stored as the new obfuscated replacement for the original identifier.

4 Challenges

1. One of the major challenges faced in the implementation was that the map variable needed to be passed around with each function call. This task got more complicated in situations where multiple obfuscations were needed to be performed simultaneously like in formal arguments of a method. These arguments were stored as a list of type *FormalParameterDecl*. To overcome this, a new function called *storeParams* was created that performed obfuscation on single argument, and this *storeParams* was passed as argument to a *foldl* call on the formal arguments list.

2. The other major challenge was while performing the actual obfuscation. i.e. getting the random string for a identifier. While generated the random string was not difficult, using the generated string was a difficult task. Since Haskell tries to enforce pure operations, using impure operations (like generating a random value) kept changing the signature of functions to impure (by add IO monad to the original return type). One way to tackle this was to convert each function to support IO monad, but this had the potential to turn into a very ugly looking code. So, instead of doing that the *unsafePerformIO* feature was used. As the name suggests, this operation is considered unsafe to use since impure values can lead to ambiguity in the results. But in this case, this ambiguity (or randomness) is actually the desired result, so it is justified to use this feature.

5 Future Work

- For this tool to be used in a practical environment, the complete set of Java language features would need to be supported.

- In this tool only a single obfuscation technique is performed. A practical implementation would use multiple techniques to make the code more difficult to understand. Moreover, metamorphism based obfuscation techniques could be considered to increase the resilience of the obfuscator.

- Empirical performance analysis can be done to better analyse the performance of the obfuscator.

References

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

[2] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.