Software Obfuscation

Ashwin Naren

August 2024

Software Obfuscation is a set of methods used to make it harder to determine the function of code. Obfuscation is never a secure method of encrypting, it is simply an attempt to secure by making it harder to understand the function of code, which is why it is used in many systems as a first layer of defense. This is why it is often used in use cases where the environment is hostile, that is, the environment is actively trying to reverse engineer what the code does, and that cannot be changed. For this reason is it commonly used in malware and in game anticheat. Obfuscation is never fully effective because the premise assumes the user has some level of control over the system, whether it be the browser or the OS or the motherboard or emulator.

There are three main types of obstification. The first is automated obstification, which uses automated techniques to scramble the output the user runs. The next kind attempts to obfuscates the code itself, usually by making the code roundabout. The final kind uses cryptography secure methods to hide the code.

1 Automated

1.1 A brief overview of programming language internals

Obfuscation primarily is used in a hostile computer environment. Automated obfuscation is simply an extra step in a build process obfuscates the functionality of the code. Before we see how obstification works, we should probably take a look at the environment that we are in.

1.1.1 Instructions Sets & Machine Code

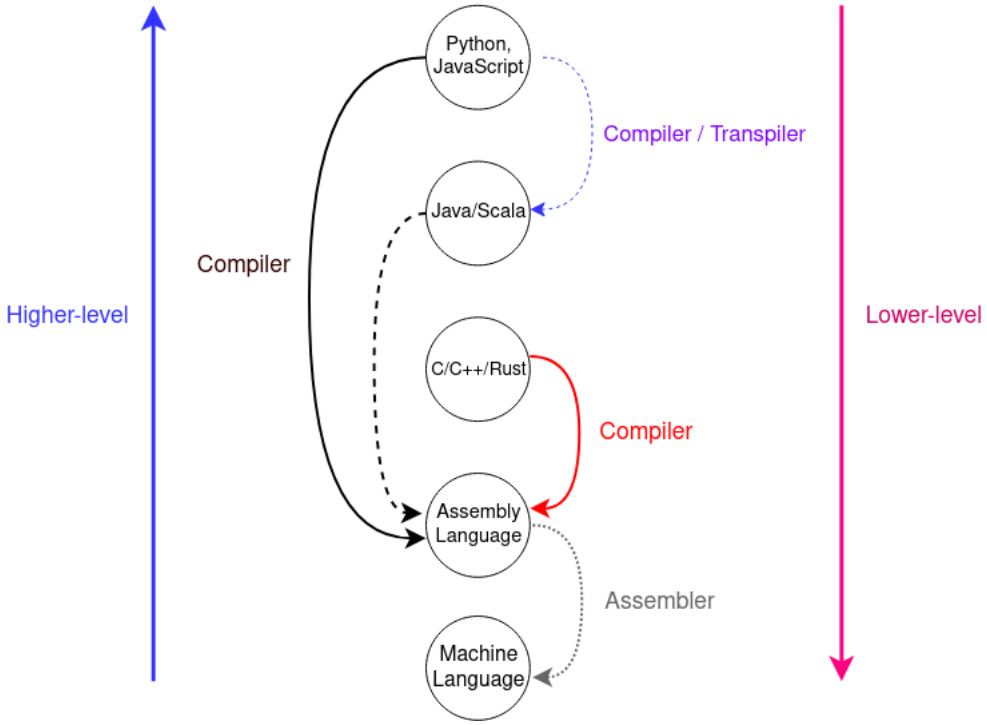
Every computer has an abstract model, or an instruction set, which CPUs implement. Instruction sets consist of opcodes, binary numbers, each opcode tells the computer to perform an singular action, which are all trivial, like adding two registers or loading a variable to a register. Opcode are usually presented in tables

1.1.2 Assembly Language

Remembers numbers can get tricky, so assembly maps these opcodes to symbols, like MOV and ADD. This makes it easier for humans to read. Usually the program that translates assembly to machine code is referred to as an assembler.

1.1.3 Compiler

As used in this paper, a compiler translates code from one language into another. Almost all languages have compilers, which the most obvious exception being machine language. Usually compilers either compile the code into assembly or into bytecode, a instruction set agnostic language that can be interpreted with more ease than the language it came from, usually via a virtual machine. A common examples is the Java Virtual Machine.



Most of the obstification occurs at the bytecode (for C# or Java and other JVM lanaguages) or compiler level for (C or C++ or any other language that compiles to assembly), but there is some obstification that can be applied to machine or assembly code. None of these methods apply to vanilla python or javascript, they require the source code to be present during execution and thus need to be prepackaged in order for these methods to work effectively.

1.2 Limitations

Compiler obstification is inherently limited due to the fact it is constrained by the compiler model we just defined. Compilers must execute either bytecode via a virtual machine or machine code. Both of which are not impossible to reverse engineer. However, we can make it difficult for potential reverse engineers.

1.3 Compiler obstification

Compilers do a lot of unintentional obstification by themselves. Compilers are heavily optimized and do not directly convert code to assembly without optimizing it first (although this can usually be disabled). Obfuscation by machine code can be pretty effective as it is difficult to understand exactly what the code does. However there are many tools that allow for reverse engineering of the code. Furthermore compilers often include symbols and function names that allow debuggers and other software to easily reverse engineer code functionality. Luckily, most of these features are easy to disable.

Some compilers are designed to intentionally Obfuscated, for example the movfuscator compiler for C intentionally only uses the MOV instruction when compiling to assembly. These are probably the best automated obstificaters but come at a performance cost.

1.4 Machine Code obstification

To go a step further simply store encrypted assembly code on the disk. Probably encrypted with a secure method like AES, or by using an obscure method that is hard to find. This method, however, needs to store the key somewhere. In a hostile environment, this is a vulnerability, because a sufficiently determined programmer can find the key.

A more common approach that is harder to combat is by inserting redundant and misleading machine code instructions that confuse anyone looking at the machine code. By having a sufficiently high ratio of valid machine code instructions to redundant or invalid machine code, it makes it incredibly difficult to understand the code. This commonly is used in viruses, because it prevents antivirus software from simply scanning the virus executables for malicious code. This can be defeated by dumping the processes memory during execution with a tool like Process Dump, which dumps the memory of a process during execution. A program that is obstification in this way decodes the machine code in memory, which makes it susceptible to this attack.

2 Code

While automated methods are used more frequently due to simplicity and maintainability, manually obfuscating code is harder for an attack to decode or understand. For example nesting lambdas in python yields highly obstificated one

liners. Using assembly alongside regular code and using short named variables is also a popular approach.

3 Cryptographic

Cryptographic obstification attempts to mathematically prove the validity of obstification methods.

There are two methods of cryptographic obstification that commonly examined, indistinguishably obfuscation and black-box obfuscation.

3.1 Black-Box Obfuscation

Definition 3.1 (black-box obfuscation)**.** Black-box obfuscation obfuscates a program in such a way that it is impossible to determine anything about it but it's input and output.

This is impossible for a certain class of problems:

Proof. Let $C_{a,b}(x)$ return b if $x = a$, and 0 otherwise.

Let $D_{a,b}(f)$ return 1 if $f(a) = b$ and f runs in at most polynomial time, and 0 otherwise.

Let C' and D' represent obfuscated versions of C and D that are claimed to be black boxes.

Now let us examine $C'_{a,b}(x)$ and $D'_{a,b}(f)$. The attacker, Eve, can not conceivably find a and b with just C', there is basically a probability of 0 that C' will return 1. However $D'_{a,b}(C'_{a,b}) = 1$, so given this pair of programs, Eve can distinguish them from $D'_{a,b}(Z) = 1$, where Z is a program that always returns 0. □

This makes Black-Box obfuscation impossible to fully implement, even theorectically.

3.2 Indistinguishability Obfuscation

Indistinguishability obfuscation is a type of obfuscation such that two programs the do the same thing being identical. This allows the implementation of the program to remain hidden. This is considered the "best" practical obfuscation because any secret that can be hidden by any other method can be hidden by this method.

Definition 3.2 (Non-Deterministic Turing Machine)**.** A Non-Deterministic Turing Machine can perform more than one action due to a given symbol, in contrast to a Deterministic Turing Machine.

Definition 3.3 (Probabilistic Polynomial Time)**.** Probabilistic Polynomial Time, or PP for short, is a class of algorithms that have a polynomial time constraint, that is $O(n^{\text{const}})$, by a non-deterministic Turing Machine.

Definition 3.4. An uniform probabilistic polynomial-time algorithm a is considered to be an indistinguishability obfuscator if it satisfies two properties.

For any boolean circuit B with n inputs and $x \in 0, 1^n$, $Pr[B'(x) = B(x) : B' \Longleftarrow a(B)] = 1$.

For any pair of boolean circuits n and m with the same size k that implement the same thing the distributions $a(n)$ and $b(n)$ are indistinguishable.