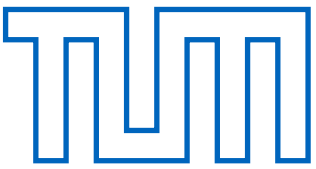
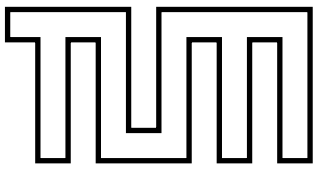


TECHNISCHE UNIVERSITÄT MÜNCHEN
DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Machine Code Obfuscation via Instruction Set Reduction and Control Flow Graph Linearization: Analysis and Countermeasures

Clemens Jonischkeit



TECHNISCHE UNIVERSITÄT MÜNCHEN
DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Obfuskierung von Maschinencode mittels Reduktion des Instruktionssatzes und Linearisierung des Kontrollflusses: Analyse und Gegenmaßnahmen

Machine Code Obfuscation via Instruction Set Reduction and Control Flow Linearization: Analysis and Countermeasures

Author: Clemens Jonischkeit
Supervisor: Prof. Dr. Claudia Eckert
Advisor: M.S. Julian Kirsch
Date: 15. March 2016

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt und nur die angegebenen Quellen verwendet habe.

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ort, Datum

Clemens Jonischkeit

Contents

Acknowledgements

This thesis would not have been possible without the help of several people I wholeheartedly wish to thank:
First, I want to thank my family – for your constant support during my studies, and for offering me the opportunity to go to university to study the subject I like.
Second, I also want to thank my caring girlfriend, Heidi, for giving me a good time when I am stressed out, and spending so much time with me, and carrying me through rough times.
Third, I want to thank Thomas Kittel for constructive feedback on an earlier version of this thesis.
Moreover, I want to thank my supervisor, Prof. Eckert, for giving me the opportunity to write this thesis, and for raising my interest in IT security in general.
Finally, without the constant support from my advisor Julian Kirsch this thesis could not exist in its current state. Thank you for discussions and helpful input during the past months.

Abstract

Obfuscation is widely used to hide sensitive data of software systems in scenarios where an analyst has full access to a software system. It transforms a program to become harder to understand and reverse engineer while preserving the semantics of the original program.
Current obfuscation methods try to achieve this by adding complexity to the control flow. Recently, a new approach to obfuscation has been proposed working in the exact opposite way: The control flow of a protected program is linearized, leaving only one continuous stream of machine code instructions containing the entire program. To further strengthen the code against analysis, all instructions of the original program are replaced by a single instruction.
In this Bachelor's Thesis, I will analyze the functionality of the proposed techniques and its applicability as an obfuscation method.
The thesis also presents an approach that allows for a complete recovery of the control flow of the original program utilizing a modified version of taint analysis in combination with a satisfiable modulo theory (SMT) solver.
We evaluate both, obfuscation and deobfuscation in context of applicability to real-world scenarios. We find that the time and space penalty introduced by this obfuscation method in its current form, add significant overhead to the program. Finally we evaluate our algorithm on several artificial programs to recover the original control flow and show that recovery is possible in most cases.

Abstrakt

Obfuskierung ist eine weit verbreitete Technik um in Szenarios, in denen ein Analyst Zugriff auf ein System hat, sensible Daten zu schützen. Es transformiert ein Programm in eine neu, schwerer zu analysierende, Form. Heutige Obfuskierungstechniken versuchen das zu bewirken, indem sie die Komplexität des Kontrollflusses erhöhen. Es wurde eine neue Technik vorgeschlagen, die genau in die Gegenrichtung arbeitet: Der Kontrollfluss wird linearisiert, sodass es nur noch einen Basic Block gibt, welcher das gesamte Programm beinhaltet. Dies wird in Verbindung mit der Reduzierung des Maschinenbefehlssatzes auf einen einzigen Befehl bewerkstelligt. In dieser Bachelorarbeit werde ich die Funktionsweise dieser Obfuskierungstechnik und ihre Anwendbarkeit untersuchen.
Diese Arbeit präsentiert außerdem eine Herangehensweise die es erlaubt den originalen Kontrollfluss, unter Zuhilfenahme einer modifizierten Taint-Analyse, komplett wieder her zu stellen.
Durch Messung der Einbußen in der Programm Größe und seiner Ausführungszeit bei Nutzung dieser Obfuskierungstechnik, werde ich zeigen, dass in der aktuellen Form, eine Anwendung kaum praktikabel ist.
Zuletzt werde ich noch die Analysemethoden zur Rückgewinnung des Kontrollflusses evaluieren und zeigen, das es in den meisten Fällen möglich ist, diesen zurück zu gewinnen.

1 Introduction

The section motivates our work by relating the concept of control flow linearization to several applications in the real world, establishes connections to similar work that has been done on the topic of obfuscation and outlines our contributions.

1.1 Motivation

In some scenarios a party wants to ship a program but does not want to disclose certain properties of an algorithm it uses. This could be a vendor trying to prevent theft of intellectual property or a company implementing a cryptographic algorithm denying people to know which crypto system and what keys have been used. This approach is bad from a security perspective, because if people manage to reverse engineer such code access to personal assets, such as chat logs or documentation on company infrastructure could be illegitimately obtained. This discouraged principle is usually referred to as *Security by obscurity* [20].

A special application of obfuscation in this context poses the protection of media content: In order to maintain the integrity of intellectual property, so-called *white box cryptography* is employed. In such a scenario, the defender assumes that the attacker is the end user of a product being theoretically in possession of all needed knowledge to recover a secret key and algorithm from a given system. The challenge to the defender is to provide an implementation that is complex enough that the attacker is deterred from performing a complete analysis of the system. Since the term *white box cryptography* was first introduced by Wyseur et al. [22], implementations of the Data Encryption Standard as well as the Advanced Encryption Standard such as [3] and [4], among others have been proposed.

A second party extensively using obfuscation are malware authors. In order to increase the resources needed for analysis, obfuscation is frequently employed: Increasing complexity in recent malware samples such as Rombertik[1] underlines this fact. While traditional malware has always been obfuscated by increasing the complexity of the inner mechanics of the malicious payload and thus blowing up the control flow to confuse an analyst, this work takes a different approach to the topic of obfuscation: Instead of *increasing* the complexity of the control flow in a given program, one can also *minimize* the control flow

[1] `https://blogs.cisco.com/security/talos/rombertik`

which, in an extreme corner case can degenerate to a linear block without any explicit control flow changes.

As malware is still a prevalent threat of our times, this thesis examines the strength and resilience recently proposed implementations [9] of such control flow linearization algorithms against an attacker and suggests analysis techniques that can be used to counter this new kind of obfuscation.

1.2 Related Work

The earliest original work mentioning the term *obfuscation* was written by Collberg et al. in 1997 [5].

During the last decades, many obfuscation tools emerged. Recent academic obfuscators include *Obfuscator LLVM* [15] and *Matryoshka* [11], theoretical background on obfuscation techniques can be found in the "Platform Independent Code Obfuscation" [1].

On the other hand one can find a great variety of software companies selling commercial obfuscators such as *VMProtect*[2], *ASPack*[3] or the now discontinued but still heavily used *Armadillo*[4].

Even patents proposing methods of software obfuscation exist: [17].

Generally an Obfuscation is the transformation of a program to a program of identical semantics such that it is more difficult to understand by an analyst. To give an overview about obfuscation techniques, we will split them into two classes: Data based obfuscation and Control flow based obfuscation. In Data based obfuscation, techniques are employed that aim to hide data or values by transforming them in different ways. Control flow based obfuscations apply a transformation on the control flow changing its appearance to confuse and distract the analyst.

First I will elaborate some Data based obfuscation techniques:

A popular way to hide data is to encode it. A function is applied to the data at compile time, changing the appearance of the static content. Before this data can be used, it has to be decoded. A way to implement this could be to undo compiler optimization, and instead of saving a number as it is, saving the number as factors, later to be multiplied. Analysis of these basic encoding schemes is not too difficult, since these expressions evaluate to constant values. Similar to that are function expecting encoded parameters. Here the encoding is done at run time rather than at compile time, but the idea is the same. Working on homomorphic mappings is more sophisticated to do this transformation. By using homomorphisms encoded data does not have to be decoded before it is manipulated [6].

Array restructuring also is a common technique, here the elements of an array are scrambled and a second array is introduced that indexes the elements of the original value. To access data of the array, the index array has to be used as a indirection of the access.

The second class of obfuscation techniques we discuss are control flow based obfuscations.

Instead of transforming the representation of data, they transfer the control flow of the program. This is done by changing and adding basic blocks and modifying transitions between them. For example, function in- and outlining is such a transformation. A function is said to be inlined, when instead of calling the function, the code of the callee is copied into the caller. This also is a common compiler optimization that can improve the run time

[2] `http://vmpsoft.com/`
[3] `http://www.aspack.com/`
[4] `http://www.siliconrealms.com/`

performance of the code. The opposite can be done as well, called outlining. Here code that is part of a function is moved into a new function, which is then called by the original function [18].

A program can be further obfuscated by introducing pseudo cycles. In this approach, a loop that is only executed once is added to the program. This will fool static analyzers as they don't recognize that loop is only run once, and so they will produce a more complex control flow graph.

Opaque predicates are another way to make the control flow graph more complex. The idea behind this is that code is executed under a condition that always evaluates to either true or false. This adds dead code to the program that is unreachable. Those opaque predicates are hard to find during static analysis since an arbitrarily complex condition has to be evaluated.

To hide the control flow the technique called "Control flow flattening" does not necessary introduce more basic blocks, but rather hides the connection between them. Like the name suggests it transforms the control flow to reduce its hight. To do so it introduces a proxy block, and instead of directly jumping to a different block, the proxy is used as a trampoline. In a extreme case the control flow can be reduced to two layers, all the original basic blocks and the proxy block [1].

Most of the techniques transforming the control flow add complexity to it, like control flow flattening hiding the connection between basic blocks or opaque predicates adding dead ones. Stephen Dolan showed in his paper *Mov is Turing complete* [8] a way to build a Turing machine with only using the MOV instruction of the x86 machine model. Based on the idea in this paper a new way to transform the control flow has been created. It creates a branch free program by emulating jumps and linearises the control flow, resulting in only one basic block containing the entire program. Christopher Domas implemented a compiler to translate a program into machine code utilizing this obfuscation technique[5] [9].

1.3 Contributions

This thesis aims to make the following contributions:

- We provide an in-depth description of the obfuscation methods employed by the MOVFUSCATOR by Christopher Domas [9].
- We give detailed theoretical background on the topic of recovering the original control flow graph from a program that had been obfuscated using the MOVFUSCATOR.
- We release the (to our knowledge) first generic deobfuscator for programs that have been compiled for the GNU/`x86_mov` target.
- We propose a generic way of recovering the control flow of the original program from the MOVFUSCATOR machine. (This process is also addressed as a broader topic referred to as *devirtualization* [16, 19].)

The remainder of this document is split up into five parts. Section 1 introduces the theoretical background explaining the original concepts behind the concept of a one-instruction machine and connecting it to the classical idea of Turing completeness. Section 2 gives a structured explanation of the implementation of two control flow linearization

[5] `https://github.com/xoreaxeaxeax/movfuscator/`

techniques and describe the details on how the MOVFUSCATORB as well as the MOVFUSCATORC model achieve semantic preserving obfuscation. Section 4 constitutes the core of our research and explains a complete generic way to recover the control flow of such an obfuscated program. We evaluate our proposed algorithm in section 5 on speed and correctness, showing that in case of the MOVFUSCATORB model the complete source code of the obfuscated application can be recovered. We conclude in section 6 and give ideas for future work on the topic.

2 Background

This section covers the theoretical background of our work. We briefly introduce the formal model of Turing machines and *WHILE* programs and recapitulate the axioms that are needed to achieve `WHILE` completeness. These axioms are then applied to show that the MOV instruction of the x86 architecture is `WHILE` and therefore Turing complete. We explain a way of organizing a machine that uses only one instruction that is employed by the implementation of Stephen Dolan [8].

2.1 Turing Machines

To show that The machines introduced later can compute any computable function i first want to introduce the model of the Turing machine. It is an abstract machine model commonly used in computability theory. It is a state machine operating on an infinite band of cells with each cell holding a symbol. It consists of a head capable of reading from and writing to the band, and a motor moving the position of the head by one in any direction or staying over the cell it was. It has been shown that with this simple model all computable functions can be computed. A function is said to be computable if an algorithm exists that returns the result of the function if it terminates [14] [12].

2.1.1 Introduction of the Formal Model

I will use a formal notation similar to state machines

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

using the following notation:

- Q: The finite set of states q_i of the Turing machine
- Σ: The finite set of input symbols
- Γ: The finite set of band symbols, $\Sigma \subset \Gamma$
- δ: A partial function $\delta : Q \times X = Q \times X \times \{\mathrm{L}, \mathrm{R}\}$, where $\{\mathrm{L}, \mathrm{R}\}$ describes the movement of the head
  - L: The head moves one cell to the left

  - R: The head moves one cell to the right

- q_0: The entry state
- B: The blank symbol, $B \in \Gamma, B \notin \Sigma$ (All cells not containing input symbols are initialized to this symbol.)
- F: The set of accepting states, $F \subseteq Q$

In the beginning the band of the Turing machine is initialized with the input. All other cells are set to the blank symbol. The head is placed over the leftmost symbol of the input and the machine is set to be in state q_0.

In each step the machine reads the symbol x from the cell under its head and makes a transition depending on the current state q, $\delta(q, x) = (q', x', d)$. If δ is not defined for given state and symbol the machine halts. In any other case the machine will write symbol x' into the cell, replacing x, then move the head according to d and switch to state q'. If the machine holds in some state $q_i \in F$, the input is accepted by this machine, otherwise ($q \notin F$) it is rejected [14].

2.1.2 Turing Completeness

The concept of Turing completeness is a concept of computability theory. It is said that a language or set of instructions is Turing complete, if it is possible to emulate a Turing machine with them. That entails that such language or instruction set can compute any function a Turing machine can compute.

2.2 WHILE programs

`WHILE` programs are a part of computability theory. They have been shown to be Turing complete, thus a language being `WHILE` complete is Turing complete as well. This is important since showing `WHILE` completeness can be easier than showing Turing completeness. The language of `WHILE` programs is a formal language defined as follows:

Algorithm	::= **start** Statements **end**
Statements	::= Statement | Statements ';' Statement
Statement	::= ϵ | Assignment | Loop
Assignment	::= Variable ':=' Expression
Expression	::= 0 | Variable | **succ** Variable | **pred** Variable
Variable	::= *any identifier*
Loop	::= **while** Variable $\neq 0$ **do** Statements **end**

The succ and pred operations are the successor and predecessor operation on the Natural numbers, so that succ$(i) = i + 1$. **pred** is defined likewise, but **pred**(0) is defined to be zero. To allow empty programs, ϵ is the empty statement. The input for these `WHILE` programs is expected to be already read in and saved to input states and output is generated in output states. So the input is part of the starting state and the output is a part of the terminal state of the program. This semantic is called an offline algorithm, since during it's execution no interaction with the environment is performed. Only countable many Variables are allowed but any identifier may be chosen. A statement can be an assignment, multiple statements in series, the empty statement or a *WHILE* loop. The while loop is defined to repeatedly execute the statements in the body of the loop, as long as a Varaible is not equal to zero. If this Variable is zero when the loop condition is first met, the loop body is not executed. [12]

a_0	0	a_1	0	···	a_{n-1}	0	0	1

Figure 1: *Grouping multiple cells to extend the set of symbols*

2.3 Brainfuck

The first version of the MOVFUSCATOR was written to compile any code written in the Brainfuck language to only MOV instructions. One reason to use Brainfuck as a base language is that it features a small set of operations which are not to hard to implement, also it is close to the syntactic of `WHILE` programs. Here I will give a short introduction to the language Brainfuck and explain why it is `WHILE` complete. First *Brainfuck* works on an infinite array of memory cells, each holding a Byte. These memory cells can be used to hold the variables of the `WHILE` program. It has one register R holding an index into the array of cells and only 8 instructions:

- $+$ increments the value of the cell pointed to by R
- $-$ decrements the value of the cell pointed to by R
- $>$ increments the pointer R.
- $<$ decrements the pointer R.
- *[* jumps forward to the matching] if $[R] \neq 0$
- *]* jumps backwards to the matching [if $[R] \neq 0$
- . writes the Byte pointed to by R to the output
- , reads one Byte of input into the cell pointed to by R

Without the size limitation the similarities between a `WHILE` and a Brainfuck program are easy to see. The cells hold unsigned integer values and the while loop directly corresponds to '[' and ']'. Assignment can be done by first moving to the target cell, clearing it and then using a loop to copy over the value, constantly running between the cells. With the memory of each cell being limited, $+$ and $-$ behave different in a sense that they over- or underflow, meaning that the successor of the highest value is the lowest and vice versa. This limitation can be overcome by grouping consecutive cells to represent a wider range of symbols, as seen in figure 1. After each of the cells holding values a zero cell is inserted and the group is terminated by two cells holding the values zero and one.

The cells a_i hold the value of the group in little endian notation. If the group consists of two cells and a_0 holds 0x37 and a_1 holds 0x13 the the value of the group would be 0x1337. The basic idea behind the group and all following operations is to work on the cells a_n, until a condition is met, like a successfull incrementation without overflow, then work is continued on the zeros. It is made sure that after the execution zells defined to hold zero still hold zero. In the end the pointer will be forced to point to a predictable location, so that the state is always certain after the operation. To comply to the definition of *WHILE* programs form Subsection 2.2, we need to show that a group of cells can be decremented without underflowing. The following operations can all be split into three basic components. The first does the desired operation on the group, leaving R to point to either of the terminating cells. The second one forces R to point the first of there two

```
1 [<]
```

Figure 2: *forcing R to point to a distinct cell*

```
1 [−(<< −)^i(>>)^i >] >>
```

Figure 3: *The predecessor operation*

cells, and the third part goes back to a_0. First i will show the implementation of the last two components, since they are used by all the operations.

We force R to point to the first terminal cell, by using the code from Listing 2. At the beginning R points to either of the terminal cells of a group. If it points to zero, this code will be skiped and R will not change. If it points to the last cell, holding one the loop will be executed exactly once. It will shift R to point to the first of the terminal cells. This cell holds zero and so execution of the loop stops. Either way R points to the first terminal cell now.

The other component simply walks back the pointer R to the start of the group. This is done by consecutive $<$ instructions.

Now i will show how the predeccessor, successor and the comparison with zero can be done using these groups. To calculate the predecessor, without underflowing we write for any a_i in the group the code from Listing 3 in sequence.

Let us assume that R does not point to a cell a_i but rather to one of the zero zells in between. By definition the loop will not be executed and R will be advanced by two, and either point to another zero, or the terminal one. If R points to a_i twp cases can occure. In the first one, the cell is zero, so that the loop is not executed. R will then point to a_{i+1}. In the second case a_i is greater than zero and is then decreased by one. If this happen all $a_j, j < i$ are decremented as well. They could not be decremented earlier, so they must hold the value zero and will underflow. In the end R will be adjusted to point right behind a_i, so that no further decrementations can occure.

The counter part to the predecessor operation is the successor operation. The challange here is that an overflow can only be detected after incrementing the value. So instead of In case R points to a cell after a_i its value is zero by definition, therefore R will be advanced by two, either pointing directly after a_{i+1} or to the second trailing zero. In the other case R points to a_i. If the value of a_i equals zero R is advanced to point to a_{i+1} or the first of the trailing zeros. If a_i is not equal zero, all cells a_i, a_{i-1}, ..., a_0 are decremented by one. Since $a_{i-1}..a_0$ hold zero the decrement of *Brainfuck* will underflow, assigning the largest possible value they can hold to them.

Incrementing a group is slightly more complex since [tests for zero, so after detecting that no overflow has occurred further increments have to be negated. The code in Listing 4 shows how this can be done. First it increments an a_i and if $+$ does not overflow it will set all zero fields after a_j with $i > j$ to the highest value, so that the addition by one will overflow and yield zero, to prevent the code inside the loop to be executed again. R will

```
1 +[> (>> −)^{n−i−1}(<<)^{n−i−1}] >>
```

Figure 4: *successor operation*

```
1 [<^{x+2i} + >^{x+2i}>] >>
```

Figure 5: *check if a group is not zero*

now point to the cells in between the a or the final one. If it does overflow the result in a_i will be zero and the loop will not be executed, R is set to point to a_{i+1} or the last zero. To be able to loop, we must be able to check whether a group equals zero or not. To do this I need one extra cell e. This can be located anywhere but I will assume that it lies at a lower address. x is the difference in address between e and a_0. For example if the cell is directly below a_0, then $x = 1$.

To check if the calue of the group is zero, the code from Listing 5 can be used. For a group to be equal to zero every member a_i of this group has to be zero. The check works very similar to the predecessor operation but when it encouters a value different from zero, one will be written to e. The R is set to point to cells in between of cells holding values.

e can now be used as the result of the check in the *WHILE* loop. It has to be manually computed at the end of the loop again. The last operation needed to achieve WHILE completness is the assignment. Here all a_i of the target have to be cleared using $[-]$ first and then all values a_i can be copied over one at a time. The assignment in conjunction with the predecessor operation can be done by first assiging one variable to the other and then using the successor or predecessor operation on the target.

Now that I have shown that any cell size can be emulated using Brainfuck, and that a behaviour that prevents an underflow from happening can be enforced, i have shown that Brainfuck is **WHILE** and thereby Turing complete.

2.4 The MOV instruction on x86

Before showing that the MOV instruction is Turing complete I will first give an overview about what the MOV instruction is, and what it does.

The MOV instruction is part of the x86 instruction set. It is one of the most used instructions as it copies data between machine registers and memory. It supports a few different addressing modes to address memory, making the mov instruction a very versatile instruction. The modes to address memory can be generally split into three ways. First in the direct addressing mode, the address of the memory cell is stored as an immediate value behind the operation, so that the address is hard-coded into the instruction. The second way is called “register indirect“. Here the target address is supplied by a register. Which register will supply the target information is encoded into the instruction. The last addressing mode calculates the address from two registers and an immediate value. The address is calculated using the formula, base + index $*\, n$ + offset, $n \in \{1, 2, 4, 8\}$. Base and index are registers, encoded in the instruction and the offset is an immediate following the instruction, n is usually refereed to as scale. This addressing mode is commonly used to access an array, as the index and scale allow easy addressing of individual elements. Only one operand, either source or target, of a given MOV instruction can target a memory location. The other operand has to be a register, for the source operand an immediate value is allowed as well. In following listings I will omit these restrictions in favor of readability. When an instruction in a later listing targets memory twice, a intermediate step has to be added, first copying the value to an internal register.

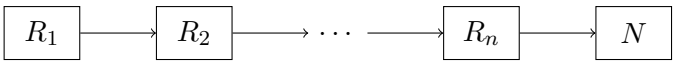


Figure 6: *The list of Symbols*

```
1 mov [R_i], 0
2 mov [R_j], 1
3 mov R_k, [R_i]
```

```
1 R_k = (R_i == R_j)
```

Figure 7: *equality check*

Stephen Dolan has shown the Turing completeness of the MOV instruction by introducing a machine model and then using this model to simulate a Turing machine. Christopher Domas provides an implementation of Dolan's model, making it the (to our knowledge) first public one-instruction compiler that we base our research on.

2.4.1 Machine Specification

To show while completeness of the MOV instruction I will first define a machine model, that can be emulated by a *x86* machine. In this section I now describe the architecture of the Turing machine that uses only the MOV instruction of the x86 platform as introduced by Stephen Dolan.

Memory is organized in 32bit wide cells (in the following we refer to this as *machine words* or simply *words*. Each of these cells can either hold an offset or a memory address. The offset is limited to values in the set $0, 1$, and both, 0 and 1 must not be valid addresses. The machine also uses a set of registers $R_1, ...R_n$ all holding exactly one machine word. The machine only uses the x86 MOV instruction described earlier. The machine does not use accepting or rejecting states, however this does not reduce the power since it can be built to write 'accept' or 'reject' to the band.

The model also comprises lists that can be considered lisp like, as seen in Figure 6: A non-empty list consists of a cell of two words, the first word represents the first element of the list, the second word pointing to the rest of the list, defined likewise. The empty list is represented by a cell at a specified address N. The content of N is undefined. Based on the described list implementation, the band, states and transition tables are defined to be an infinite lists. The Symbols are represented by cells at addresses S_1, ..., $S_{|\Gamma|}$, each corresponding to their respective symbol in Γ. Without loss of generality it is assumed that the blank symbol corresponds to S_1. Its content is unspecified.

2.4.2 Proof of Turing Completeness

To show Turing completeness we either have to show a way to emulate the Turing machine, or to emulate a language that is know to be Turing complete. I will show Turing completeness by giving a generic way to emulate any WHILE program. For this we need need several components, like while loops, assignments, the successor and predecessor operation. First need a primitive for equality checking. This is necessary for Turing completeness because it later allows the code to have effect only under certain conditions. To check whether two pointers R_i, R_j point to the same symbol the series of instructions from Figure 7 can be used.

```
1 mov [N], R_i
2 mov [N + 1], R_j
3 mov R_l, [N + R_k]
```

```
2 if (R_k)
3         R_l = R_j
4 else
5         R_l = R_i
```

Figure 8: *switching between data*

The basic idea behind this is to write values to the symbols and if both registers address the same symbol, the value written the second time will be read. This check is done in three steps. First zero is written to the address R_i points to, then one is written to the address R_j points to. Afterwards the register R_k is assigned the value of the cell R_i points to. If R_i equals R_j then the 0 that is written earlier will be overwritten by the 1 and thus the result in R_k will be one. If they are not equal then R_i and R_j point to different memory cells and $[R_i]$ will not be overwritten, resulting in a zero in R_k. This changes the content of the symbols that are addressed, but since the value of the symbol is defined by its address, and its content is unspecified, this has no effect later. No that we can check if two values are equal, we can now also switch between two sets of data. We use N, the termination of any list to temporary hold the two values. Since the contents of N are unspecified as well, those changes in the content have no effect later.

In Figure 8 we can see the instructions used to achieve switching between data. Here R_i and R_j are arbitrary data and R_k is an offset (either zero or one) which may stem from the result of an equality check. The result will be stored in R_l which, depending on the value of R_k will either be the original contents of register R_i or R_j [8].

To show `WHILE` completeness we now adjust the definitions made earlier a little, to allow the increment and decrement operations needed. Currently it is not possible to decrement or increment a symbol as the exist only as specified memory locations. I now define the symbols to be a double linked list, where the backward links of the first element points to it self. The pointer to the next element is stored right behind the content of S_i and the backwards pointer thereafter.

The empty statement of the while program is replaced by no instruction in our machine. The symbols are the Natural numbers, and are ordered so that $S_i = i - 1$, as counting of the symbols starts at one. The fundamental problem any approach that aims to implement a Turing complete machine is that the MOV instruction is not capable of branching, as it can not target the instruction pointer. I will show a technique to emulate these jumps, using a loop around the whole program. Looping can be done by using the segmentation fault signal handler or page aliasing depending on what level the program operates. This entails, that for every run of the loop, every instruction inside it will be executed. To mitigate this problem, we will assign a unique label to every basic block. In the beginning the first basic block is marked for execution by writing it's label to the register R_1. Every the machine state is potentially changed, a check is introduced, to see if the instruction is part of the basic block marked by R_1. Only if this check passes, the change is applied to the state of the machine. Jumps are performed by updating the label in R_1, what als may only occur when the current block is the block marked by R_1.

We will also use R_2 and R_3 to hold temporary values, these to registers are excluded from the machines state, as well as the contents of the Symbols and the content of N are not part of the state.

R_i accesses the contents of the register R_i, $[R_i]$ accesses the content the register points to, and $\&R_i$ is the address of the register in memory. Addressing a register like that is

```
1 mov [D], 0
2 mov [R1], 1
3 mov R2, [D]
4 mov [N], N
5 mov [N+1], &Ri
6 mov R2, [N+R2]
7 mov [R2], Rj
```

```
6 if (D == R1)
7         Ri = Rj
```

Figure 9: *Assignment*

```
1 mov [D], 0
2 mov [R1], 1
3 mov R2, [D]
4 mov [N], N
5 mov [N+1] , &Ri
6 mov R2, [N+R2]
7 mov R3, [Rj+n]
8 mov [R2], R3
```

```
8 if (D == R1)
9         Ri = succ(Rj); // if n = 1,
                  else pred(Rj)
```

Figure 10: *Incrementing / Decrementing Data*

possible, because they are defined to be a list, so they lie in memory. From now on we will use D as the identifier of the current basic block. It is a pointer to a constant S_n.

With this we now can implement a way to assign one register to another, see Figure 9. We first check if the basic block is the one marked by R_1 for execution and then switch between the target register and N, if the current block is the marked one, then R_j will be written to R_i. If not, it will be written to N, and have no effect on the machine status.

Now we build the **succ** and **pred** statement. The concept seen in Figure 10 is very similar to the concept we used defining our assignment. Here n is either one or two, depending on what link direction should be accessed. If $n = 1$, then target of the forward list is copied, so that R_i points to the successor afterwards. If $n = 2$, target address of the backwards link is copied, effectively executing the **pred** operation.

To implementation of the while loop has to be split into two parts. The first part opens the loop and handles the check of the variable. The second part closes the loop behind its loop body and always jumps back to the first, where the loop condition will then be checked. The opening of the loop is implemented, as seen in Figure 11. After making sure the current block is the one marked for execution, The value of R_i is checked. If it equals

```
1  mov [D], 0
2  mov [R1], 1
3  mov R2, [D]
4  mov [N], N
5  mov [N+1], &R1
6  mov R2, [N+R2]
7  mov [S1], 0
8  mov [Ri], 1
9  mov R3, [Si]
10 mov [N], Ri
11 mov [N+1], Sj
12 mov R3, [N + R3]
13 mov [R2], R3
```

```
10 if (D == R1) {
11         if (Ri != S1)
12                 Ri = Ri;
13         else
14                 Ri = Sj;
15 }
```

Figure 11: *opening of a while loop*

```
1  mov [D], 0
2  mov [R1], 1
3  mov R2, [D]
4  mov [N], N
5  mov [N+1], &R1
6  mov R2, [N+R2]
7  mov [R2], Sj
```

```
16 if (D == R1)
17         R1 = Sj;
```

Figure 12: *jump to label S_j*

```
1  mov [D], 0
2  mov [R1], 1
3  mov R2, [D]
4  mov [N], N
5  mov [N+1], 0
6  mov R2, [N+R2]
7  mov [R2], 0
```

```
18 if (D == R1)
19         halt;
```

Figure 13: *Assignment*

zero, then R_1 is updated to the label of the basic block following the while loop. If it does not equal zero, R_i is preserved and the loop body is executed. Now we need to find a way to transfer the control at the end of a basic block to its successor. The is done much like a normal assignment, but instead of copying the value from a different variable, a constant symbol is assigned, see Figure 12. In the case a loop is closed, this symbol is equivalent to the id of the opening block, else the next consecutive block is targeted. To finally end the while program we use a mechanic build into the computer. Accessing a page that is not mapped results in a segmentation fault. A signal is sent to the program and if no handler is registered to catch such signal the program is terminated. To trigger this segmentation fault we first add a basic block at the end of the while program. then we try to access the memory address zero if the current block is the one marked by R_1. The virtual address zero is normally not mapped to any physical address, and on the Linux operating system a user land program can not map this virtual address.

Now that we have found an implementation for every terminal of the `WHILE` language, our implementation is `WHILE` and therefore Turing complete.

2.5 SMT Solvers

Satisfiability Modulo Theories (SMT) solvers check the satisfiability of formulas written in a language containing interpreted predicates and functions. In their implementation they can be seen as extensions of propositional satisfiability (SAT) solvers to more expressive languages. They are used to lift the efficiency of SAT solvers to richer logic. Very schematically, an SMT solver abstracts its input to propositional logic by replacing every atom with a new proposition. Underlying, a SAT solver is used to provide Boolean models for this abstraction. The theory reasoner repeatedly refutes these models and refines the abstraction by adding new clauses, until either the theory reasoner agrees with the model found by the SAT solver, or the abstraction is refined to an unsatisfiable formula. [2] An established implementation of such an SMT solver is z3[6], developed by Microsoft Research. [10]

[6] `https://github.com/Z3Prover/z3/wiki`

Our approach for deobfuscation leverages the built-in constraint-solving techniques of z3 in order to determine possible jump targets of any given basic block, as we will see later.

2.6 Taint Analysis

The taint analysis is a very popular dynamic information flow analysis method. It can be used in several contexts like testing, debugging, policy enforcement and vulnerability detection. Taint analysis is performed by marking data associated to a memory location. This data is then tracked as it propagates through the program. A taint analysis typically has three characteristics:

- taint sources: specify the data to be traced
- taint propagation policy: specifies how taint is propagated during program operation
- taint sanitizers: indicates when data can be unmarked and taint can be removed

In context of vulnerability detection a taint source could be untrusted user input or network traffic. Propagation of this taint can occur via a covert channel, like the control flow. If tainted input reaches functions that don't properly check the tainted values against internal assumptions, a vulnerability is detected. Taint sanitizing happens when a static value is assigned to a tainted variable, effectively removing the taint. [21]

In context of deobfuscation I use the taint analysis a bit differently. I use it statically in a small context. I only obtain how the data is propagated through the program locally. Application of taint analysis is straight forward in a MOVFUSCATORC obfuscated binary. No jumps happen and MOV instructions modify data in an explicit way. I use the function principle also to scan the binary backwards to find the source of a value. Performing a backwards trace the source of the taint is a given instruction or memory location. The analysis iterates backwards over an buffer of previously disassembled instructions. If a MOV into a tainted register or memory location occurs the taint of this register is sanitized and the source is tainted, if it is not a constant or a memory location known to be a variable. Temporary storage locations are still tainted. When doing such backwards taint analysis the result will be a binary tree of memory access in a MOVFUSCATORC executable. This is due to the fact that the source operand of a MOV operation can only be affected by two register. Such accesses are recorded. Tainting forwards the taint source is the same as tainting backwards but propagation is restricted to copies into register only. When a not tainted value is assigned to a tainted register, taint is removed. This finds all memory locations the value is written to or when the value is used as an address.

```
1 mov [C], 0
2 mov [C + R_i], 1
3 mov R_k, [C]
```

```
20 R_k = (R_i == 0);
```

Figure 14: *check if R_i equals zero*

3 Obfuscation

The MOVFUSCATOR comes in two different flavours: The MOVFUSCATORB version that accepts a program written in *BrainFuck* and transforms it to MOV instructions. We suffix this flavour with a **B**, standing for Brainfuck. As Brainfuck code is not easily written, Domas created a second version of the single instruction set compiler: MOVFUSCATORC serves as a back end for the lcc retargetable ANSI C compiler [7] [13] allowing the full C standard being flawlessly compileable to MOV instructions.

3.1 MOVFUSCATORB

MOVFUSCATORB is a simple implementation based on the concepts of the original paper by Dolan and the ideas I showed in Section 2.4.2. It accepts a given program written in the *Brainfuck* language and returns assembler code consisting of only MOV instructions. Since Brainfuck is Turing complete any computable function can be computed. The main difference in implementation to the Machine defined in Section 2.4.1 is the use of arrays instead of lists. Symbols are now represented as positive natural numbers including zero. With the symbols not being memory locations any more, we can not write to them, so we need a different way to check if a symbol is zero or not. For this comparison 256 bytes of memory are reserved at address C. One Byte for each value.

The implementation, seen in Figure 14 is very similar to the comparison we saw earlier. All addresses $[C, .., C + 255]$ are included in the memory range reserved for C and since every cell can only hold a value $v \in [0, .., 255]$, the instruction *mov* $[C + r], 1$ always stays within the bounds of the array. The second thing we loose by defining the symbols to be integers, is the easy way to traverse the list of symbols get the successor and predecessor. For this operation we introduce read only lookup tables, one to increment numbers (I)

[7] `https://sites.google.com/site/lccretargetablecompiler/`

and one to decrement values (D), both with a range from 0 to 255.

$$I_i = \begin{cases} i+1 & \text{if} i \in [0, ..., 254] \\ 0 & \text{else} \end{cases}$$

$$D_i = \begin{cases} i-1 & \text{if} i \in [1, ..., 255] \\ 255 & \text{else} \end{cases}$$

Both operations wrap around 0. To calculate the increment or decrement of i, the array is accessed at index i. By the way those arrays are designed the returned value will be one larger or smaller than i or under or overflow.

The register R_1 from subsection 2.4.1 is named id here and implemented as a cell in memory. We also introduce the variable *ON* for convenience. It is set to 4 if the current block should take effect and to zero otherwise. The value four has been chosen to switch between two sets of data. Brainfuck operates on an array in memory and to have the program not change the state if a condition is false a second array is introduced. If *ON* is zero, this second array is accessed instead. Changes to this second array are discarded, as only the first array contains the real values that the program operates on. *ON* reduces the overhead from the redundant checks whether an operation can write to memory or not. Different from subsection 2.4.1 is that the basic block before and after a while loop share the same id. This is no problem here since MOVFUSCATORB has full control over the translation process and a jumps skipping loops are always forward jumps. This is based on the way *Brainfuck* is written, namely as a sequence of instructions that will be executed in that sequence. Only *]* allows to jump back. But since the loop end shares the label with the loop beginning it does not need to update the label in this case. Execution will continue with *ON* being off, until the end is reached and the program loops from the beginning, hitting the opening *[* first. Therefore execution is continued at the right place. To escape from the loop zero is dereferenced leading to a segmentation fault and effectively stopping the program.

3.2 MOVFUSCATORC

Christopher P. Domas implemented the so called "M/o/Vfuscator2". It is a C compiler capable of translating any C program into an executable binary for the x86 architecture that only uses MOV instructions. We relate his implementation to the theoretical model introduced earlier and point out similarities and differences.

In a first step an intermediate virtual machine is defined that is closely related to *x86*, but with less instructions. He then translates the intermediate instructions of the machine into only MOV instructions. He also wrote a library to extend the basic functionality of this machine to be able to handle IEEE floating point operations. As the IEEE model introduces a significant amount of overhead they are not always linked into the binary.

3.2.1 Overview

To make this possible he created a new back end for lcc, the x86/mov target. This target introduces a virtual machine as an intermediate step and then implements this machine with only `mov` instructions.

The machine consists of 11 registers:

operation	c style syntax	description
add x, y, z	$x = y + z$	addition
sub x, y, z	$x = y - z$	subtraction
band x, y, z	$x = y$ & z	bit wise and
bor x, y, z	$x = y \vert z$	bit wise or
bxor x, y, z	$x = y \hat{} z$	bit wise exclusive or
bcom x, y	$x = \ y$	bit wise inversion
neg x, y	$x = -y$	negation
lshu x, y, z	$x = y <<_{\text{unsigned}} z$	unsigned left shift
rshu x, y, z	$x = y >>_{\text{unsigned}} z$	unsigned right shift
rshi x, y, z	$x = y >>_{\text{signed}} z$	signed right shift
mul x, y, z	$x = y * z$	multiplication
idiv x, y, z	$x = y/z$	integer division
udiv x, y, z	$x = y/_{\text{unsigned}} z$	unsigned division
imod x, y, z	$x = y\%z$	integer modulo
umod x, y, z	$x = y\%z$	unsigned modul
cmp x, y	$x - y$	compares x and y
eq x, y, z	$x = y == z$	x is 1 if y = z else x = 0
not x, y	$x = \neg y$	x is not y
jmp_cc l, x, y	-	jump to label l if condtion is met
jmp l	goto l	always jump to label l

Table 1: *supported operations of the mov-machine*

- 4 Byte accessible 32 Bit general purpose integer registers
- 2 single precision floating point registers
- 2 double precision floating point registers
- 1 stack pointer pointing at the topmost element of the stack (full ascending stack)
- 1 instruction pointer indicating the label of the next-to-be executed basic block
- 1 status register containing the results of performed comparisons (as per Table 2).

It utilizes an arithmetical logical unit (ALU) capable of basic integer arithmetic and Boolean logic as depicted in table 1. Additionally there are 3 scratch registers that are used internally by the ALU.

The only instruction manipulating the status register is the `cmp` instruction, see table 2. Contidional and unconditional jumps are supported as well, see tables 1 and 3. Calls are performed by first pushing the label of the basic block behind the call instruction onto the stack and then jumping unconditionally. Likewise the implementation of "return" pops a label first and then jumps to it.

It is easy to see how this architecture can implement any `WHILE` program defined earlier. The registers can be emulated by memory cells. The MOV instruction itself is an assignment and constants can be represented as they are. It has addition and subtraction, also accepting an constant as one y or z. A while loop can be emulated by putting a

short	name
zf	zero flag
sf	signed flag
cf	carry flag
of	overflow

Table 2: *the flags of the status register*

condition code	condition (x, y)
eqi, equ	$x = y$
gei, geu	$x \geq y$
gti, gtu	$x > y$
lei, leu	$x \leq y$
lti, ltu	$x < y$
nei, neu	$x \neq y$

Table 3: *conditon codes for the jmp_cc instruction, postfix u and i destinglish between signed and unsigned*

compare $x = 0$ and a jmp_eq afterwards with the destination right behind the loop. Also at the end of the loop a jump instruction to the beginning is needed to construct a while loop.

The possible conditions for the `jmp_cc` operations are much like the ones known from the x86 architecture.

This immediate machine now has to be implemented to run on a *x86* computer. This can be done with only *x86* `mov` instructions. The main problem again is that the jump has to be emulated since the MOV instruction is not capable of copying data to or from the instruction pointer (a circumstance that has changed on the x86-64 platform). The same technique as described in subsection 2.4.1 is employed: Each basic block gets assigned a different label block. Special locations such as the return points of function calls (that is, in the usual case the block following the call instruction) also need to be assigned labels.

The designated jump target register referencing the label of the next executing basic block is called *target*. As mentioned earlier there is a designated register to show whether changes should take place or be discarded. We call this *ON* and possible values for it are either 0, representing “false“ or “off“ and 1, “true“ or “on“. Most operations writing to memory now simply check if *ON* is set to 1. For this there is a the *sel_data* helper. It consist of 2 adjacent cells both capable of holding addresses. The first cell has a fixed address pointing to a designated *discard* cell. Writing to this cell has no impact on the machine by definition. The second value can be set to an arbitrary address. For writing the jump target and to *ON* itself two specializations of this helper exist. The second cell of *sel_target* points to *target*, and the second cell of *sel_on* points to *ON* This allows writing c to address A in four steps.

```
1 mov [sel_data + 4], A
2 mov r, [ON]
3 mov r, [sel_data + r*4]
4 mov [r], c
```

Listing 1: *”copying c to A”*

The next difference is the way the arithmetic is implemented. This is much like MOVFUSCATORB, just that it has a lookup table for every operation. The machine internally works with 32 Bit wide registers. As the address space on the x86 platform is limited to 2^{32} different addresses, there is not enough space to build look up tables for all 32bit operations. Even if each value was mapped to only one result, the corresponding look up table would occupy the whole address space. (One would need $2^{32} * (32/8)$ Bytes of memory for all operations, way more than the amount of memory addressable with 32bit

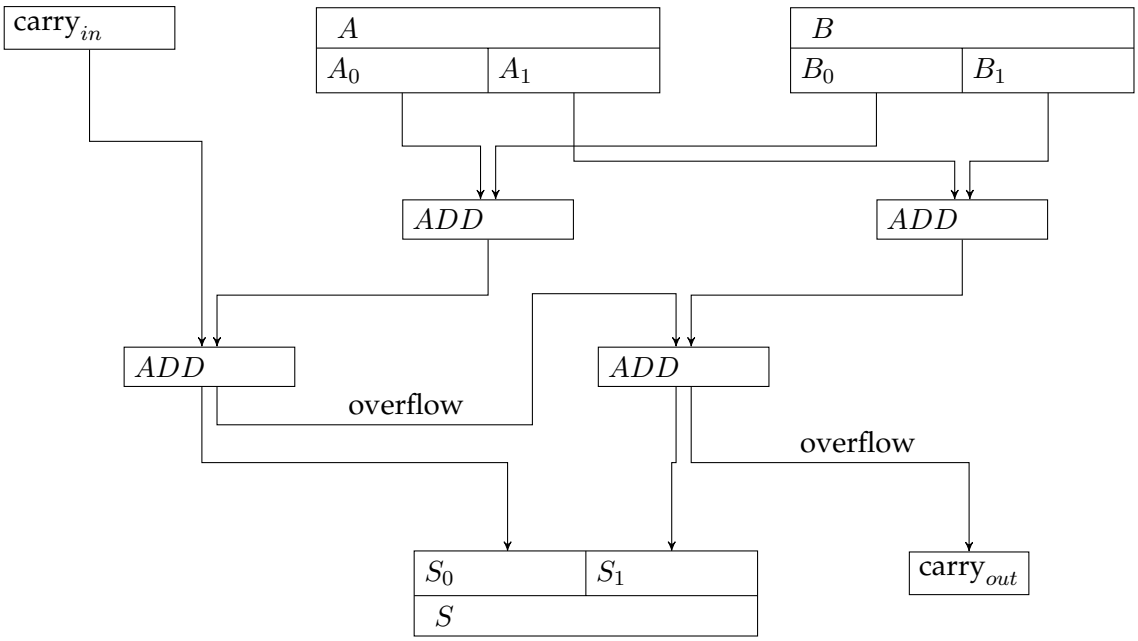


Figure 15: *Information flow during addition of two 16 bit numbers A and B with the result stored in S*

wide addresses.)

To overcome this limitation, look up tables are for the most part 8bit operations, deriving the correct result of the 32bit operation using multiple look ups and carries. One exception to this is the addition which uses a 16 bit look up table so the operation has to only be split in two parts: Adding the lower halfs, and adding the upper halfs together with the carry from the lower half.

3.2.2 Static Content

Since all arithmetic operations are carried out using look up tables the generated executable files contain lots of static data significantly increasing the file size (see Section 5). Based on the provided mathematical functionality those tables vary in size. They can be divided in two different groups depending of the *arity* of the operator: Tables for operations with only one operand as input and tables for operations with two input operands. Apart from these look up tables, there also exists static setup code in every MOV program that is responsible for setting up the execution environment.

Look Up Tables One dimensional look up tables δ are the key data structure being used for operations of arity one. Basically all results of such an operation are written as an one dimensional array such that

$$e_i = \delta(i) \ .$$

There exist three classes of look up tables: Apart from the mentioned one-dimensional tables (Table 4) there exist also two-dimensional tables (Table 5, explanation see below) and eight special tables:

Four of the latter ones help with multiplication and the other four help with division. They are mostly specializations of two dimensional lookup tables. The advantage here

label	size	description
alu_bn	256 Long words	8 tables each selecting the n'th bit
alu_inv8	256 Bytes	inverts every bit
alu_inv16	65536 Long words	inverting 16 Bit at once
alu_clamp32	512 Long words	return 32 if the operand is greater than 32
alu_sex8	512 Long words	sign extends a value
pushpop	stack size	stack pointer adjustment

Table 4: *One Dimensional lookup tables*

is the need for less instructions to compute the result gaining a performance increase. To invert the 16 Bit value a one simply accesses the special table *alu_inv16* at index a:

```
1 mov r, [alu_inv16 + a * 2]
```

Look Up Tables 2D Adding the two 16 bit values a and b can be done by fist getting the address of a table specific to the operation and one operand and then accessing index b in this table:

```
2 mov r, [alu_add16 + a * 4]
3 mov r, [r + b * 4]
```

Addresses are 32 bit wide, so a is multiplied by 4 in the first instruction. As the result is also 4 byte wide, a potential overflow has to be mitigated by also multiplying b by 4 in the second instruction.

A noteworthy table is the one being used for addition: Even though addition is an operation with two operands it is implemented using only one dimensional look up tables. One table holds all possible results where every cell contains its index i as value at location e_i with $i \in [0, \ldots, 2^{17}]$:

$$e_i = i$$

The second table provides pointers into the first table

$$b_i = \&e_i$$

realizing the addition of two 16 bit numbers a and b by a double dereference accessing b_i at position a and the result at position b. This primitive for 16 bit addition can be applied according to Figure 15 to provide a full 32-bit wide addition.

The last look up table is a very important and very big one. It is used to increase or decrease the stack pointer by four and contains all possible addresses the stack pointer can attain. It is placed at a fixed offset relative to the stack of the virtual machine. Because of this restriction the stack pointer is adjusted during the prologue: As this look up table consists of all possible stack addresses it is quiet large and also imposes a limit on how big the stack can get. This table is not exactly necessary since its functionality can be done via a subtract. It just reduces the instruction count needed to perform the push and pop operation and therefore aims to improve the performance of obfuscated programs.

An example of a push instruction can look like the following:

label	size	description
and	$2 * 2$ entries	boolean and
or	$2 * 2$ entries	boolean or
xor	$2 * 2$ entries	boolean exclusive or
xnor	$2 * 2$ entries	boolean equality
alu_b_s	$8 * 256$ entries	sets the xth bit in y
alu_b_c	$8 * 256$ entries	clears the xth bit in y
alu_eq	$256 * 256$ entries	$x == y$
alu_band8	$256 * 256$ entries	logical and
alu_bor8	$256 * 256$ entries	logical or
alu_bxor8	$256 * 256$ entries	logical exclusive or
alu_lshu8	$33 * 256$ entries	unsigned left shift
alu_rshu8	$33 * 256$ entries	unsigned right shift
alu_rshi8	$33 * 256$ entries	sign bits for right shift
alu_mull	$256 * 256$ entries	low byte of multiplication
alu_mulh	$256 * 256$ entries	high byte of multiplication

Table 5: *Two Dimensional lookup tables*

```
4 mov r, [sp]
5 mov r, [pushpop-stack-4+r]
```

with `pushpop` being the base address of the look up table and `stack` being the initial value of the stack pointer.

Prologue The prologue sets up the execution environment of the virtual machine. It adjusts the position of the stack, and registers a signal handler for the segmentation fault and the illegal instructions signals.

A common cause of receiving the segmentation fault is accessing a memory region in an illegal way. For example trying to access memory that is not mapped, or trying to execute memory that is marked read and write only.

An illegal instruction signal is received when the program tries to execute an instruction that is not viable on the chip. In context of the MOVFUSCATORC, the illegal instruction signal is abused to re-start the execution of the program by setting the address of the signal handler to the MOV program itself. As signal handlers can be nested, this technique fulfills one important property that, while handling a signal the cause of the signal may occur again and the handling function will be called again.

The segmentation fault signal handler is used to dispatch calls into external libraries, like the *libc*. After registration of the signal handler the main loop begins. The prologue adds some code to it. It adds static code that adjusts the stack pointer and pushed the arguments of the main function onto it. Then it calls the main function and finally calls the libc function *exit*.

Apart from the look up tables there are several data structures being set up during the prologue: To first start execution, *toggle_execution* is initialized to 1. This value is then written to *ON* and *toggle_execution* is set to zero. This prevents the adjustment of the stack and calling the main function to execute again. The data setup also contains the registers,

initialized to zero. And the stack pointer pointing to the new top of the stack.

3.2.3 Performance Estimation

Due to the way the jumps are emulated the performance of the program takes a big hit. For every iteration of a for loop the complete program will be executed. Therefore This performance penalty increases with the size of the program. While to difference is not severe in the test cases where the code size is very small the difference becomes obvious. The performance hit is even worse for programs obfuscated with MOVFUSCATORB, since Brainfuck relies on a lot of nested loops. We will perform a proper performance evaluation in the evaluation chapter of this thesis.

Control Flow Graph Linearization Emulating jump instructions labeling each basic block and return target generates a huge overhead in terms of number of instructions. But it also eliminates traditional jump instructions. The program will be executed linearly and every instruction will be executed exactly once per run of the loop. In the end an illegal instruction will create a call to its handler and the execution of the program starts from the beginning with a different state of the registers. This property of the program linearizes the program flow and gets rid of all the branches. The whole program appears to be one giant basic block that covers the semantics of the original program. This property is also the most desired one from the obfuscating defender's perspective.

Instruction Set Reduction The *x86* architecture has a lot of instructions to perform all sorts of operations. Among the most common ones are instructions to perform addition, subtraction and multiplication. By the instruction set reduction none of those instructions is present in the code anymore. MOVFUSCATORC uses look up tables to perform arithmetic and logical operations. But due to size limitations most operations on 32 Bit values have to be split into multiple parts. Each part computes the result of one or two bytes and the result has to be assembled afterwards. This accounts for another massive overhead in terms of the size of the obfuscated program.

Virtualization based Obfuscation Traditional virtualization based obfuscators embed an interpreter into an executable and transform the protected payload into a semantically equivalent byte-code representation that is executed by the interpreter at run-time. Classic obfuscators of such kind leverage the usual fetch-execute-writeback loop any CPU has to undergo which can be detected during the execution. The MOVFUSCATORC represents another instance of the kind of obfuscation, with the difference that the mentioned loop of the embedded interpreter CPU is not explicit anymore and therefore difficult to detect.

Additional Hardening Together with MOVFUSCATORC there come some python scripts that can post process the assembly output of the compiler. These scripts can harden the generated code against pattern-matching based deobfuscation attacks by using two techniques: instruction reordering and register renaming.

Both techniques are possible because for the MOV instruction the 4 basic *x86* general purpose registers `eax`, `ebx`, `ecx` and `edx` can be used as operands interchangeably. Splitting one higher level instruction into many *mov* instructions allows reordering of instructions

by allowing the following instruction to already use the registers that are not needed any more.

As an isomorphism of the MOVFUSCATORC there exist further hardening techniques that generate a 1:1 translation of the MOV instructions to an equivalent sequence of arithmetic instructions: This way, programs consisting only of arithmetic instructions, like exclusive or, addition or subtraction, can be created. Due to the trivial 1:1 mapping between the arithmetic program and the MOV program, this thesis only focuses on the latter kind of programs.

3.3 Summary

In this Section the working principle of the MOVFUSCATOR has been elaborated. It is based on the machine from Section 2.4.1 and uses a target register to emulated jumps. Arithmetical operations are performed using look up tables, splitting every operation into many instructions. Using these techniques a intermediate machine could be implemented, only using MOV instructions, linearising the control flow.

4 Deobfuscation

I will try to recover the original control flow in multiple steps. First static data like the prologue setting up the machine are examined. Then multiple passes over the instructions are performed. First the labels of the basic blocks are recovered. Then jump targets are restored. Finally the acquired data is put together to reconstruct the control flow graph of the underlying MOV machine.

4.1 MOVFUSCATORB

The MOVFUSCATORB translates a given program written in the Brainfuck language step by step. It first sets up the environment and then replaces every symbol of the original program by a static series of instructions. Only [and] have different code, since they have to manage the labels. MOVFUSCATORB features four switches affecting the program in a different ways.

- nojump: replaces the jump in the end with a segmentation fault for strict MOV compliance. The signal handler for the segmentation fault signal is set to be the program itself, without initialization of the environment.
- mmio: memory mapped input and output. Instead of using the kernel interface to read and print character, the input and output is mapped into memory. This assumes file backed input and output, or it will not work.
- O: Optimizes the program, instead of writing back the result after every '+', '−', '>' and '<' it writes back the accumulated result of a series of these instructions.
- cell16: Instead of having a cell width of one byte the cell width is increased to two bytes.

However most of the code stays the same. The total lack of any randomization gives the opportunity re-translate sequences of MOV instructions to Brainfuck program statements therefore not only breaking the obfuscation but fully recovering the original source code.

4.2 MOVFUSCATORC

MOVFUSCATORC is much more sophisticated than MOVFUSCATORB. It is implemented as a back end for lcc, a retargetable C compiler and is able to compile source code written in ANSI-C. The stream of instructions is much more complex. It also comes with python scripts randomizing the output in different ways, like reordering instructions and swapping the x86 base register. With the additional hardening introduced it is not possible any more to simply build a finite automaton and retrieve the original source code. Much more advanced techniques have to be used. Which we describe in the following.

4.2.1 Analysis of Static Setup

A very important part of MOVFUSCATORC is setting up the environment. The program must have the ability to loop without any explicit instruction. To do so, a illegal instruction at the end of the executable program is used in conjunction with a signal handler for the illegal instructions. This handler is set to be the program itself. It is also marked to not mask the signal, so it can occur during the signal handling itself, effectively restarting the program over and over again. From this handler we can get the effective start of the program, as it is the address of the signal handling function. Also a signal handler for segmentation faults is registered. It is used to dispatch calls to external libraries. To call the functions to set up the signal handler, arguments have to be passed on the stack. From this we get to know the address of the emulated stack pointer sp, later used in patching the binary executable. Then some more code is introduced that sets *ON* to one, once, calls main and then exit. To toggle on the execution it uses a fixed series of instructions, as seen in Listing 2. The *toggle_execution* switch is set up to be one, and after executing these instructions it is zero for the rest of the program's lifetime.

Apart from the prologue MOVFUSCATORC also adds static data to the data section. The data added are mostly look up tables. To identify those look up tables, we do a linear sweep over the data segments. If we find a pointer into the data segment we try to identify the look up table, and then skip an amount of data, according to the size of this table. We only find two dimensional look up tables, since one dimensional ones don't need pointer into the memory. First we consider the size of the pointer array. If it holds only two pointers, it is assumed to be a boolean operation. The result of boolean operations are either zero or one, so we shift all possible results, so every result is at a unique bit position. Then we use logical or to combine them into one value. We shift the value at index (0,0) by 0, the one at (0,1) by one, the one at (1,0) by two and the one at (1,1) by three. Zero represents false and one true. We check the result of this against the values from table 16a. The and operation equals the value 0x8 because both values have to be one so that "and" is one. (1,1) is shifted by three so a one there has value 8. or has the number 14, because only the lowest bit is not set, because it only results in zero in case of (0,0). "XOR" is true in cases (0,1) and (1,0), which results in a six and equality holds true for (1,1) and (0,0).

```
1 mov eax, [toggle_execution]
2 mov eax, [sel_on + eax * 4]
3 mov [eax], 1
4 mov [toggle_execution], 0
```

Listing 2: *Toggleing on the execution for the first time*

If it is not a boolean table, the pointer and their results are checked. If adjacent pointer points to adjacent cells in the look up table, it simply indexes the same table. This is done

operation	value
and	0x8
or	0xE
exclusive or (xor)	0x6
equality (xnor)	0x9

(a) *values to check for boolean operations*

name	value
bit set	0xCB
bit clear	0x4B
and	0x3
or	0xCF
xor	0xCC
mul_l	0x8D
mul_h	0x5

(b) *result of accessing lookup tables at (7,0xCB)*

Figure 16: *Signature values for different lookup tables*

in case of the addition operation. Also the second element of the array is checked to be one. After covering the corner cases for two dimensional look up tables we access a specific element of the table to get the type of the table. The first index is capped to be 7, because the tables to clear and set bits are that small, the second index can be as large as 255, the highest number in one Byte. The result of the access will be one Byte. I set the first index to seven, to have the highest number of bits set in it. Now we have to find a cell where every table has a different value using the second index. To generate an overflow in the multiplication so that the high word differs from zero, the two highest bits of the index are set. This also covers the bit set and bit clear operations. The four lowest bits are set to "1011", to cover the logical operations. The lowest bit is only set to achieve a result other than one for the and operation. This results in the number 0xCB. The result of accessing the tables at index (0x7, 0xCB) can be found in table 16b. Mul_l is the lower byte of the multiplication and mul_h is the high byte. They are two distinct tables.

After finding the base addresses of these tables, later accesses to these tables can be substituted by their respective arithmetic operations.

4.2.2 Reconstruction of the Original Control Flow Graph

Reverse engineering can be a tedious and exhausting task, depending on the targeted executable. There are many tools that assist the dynamic or static analysis of binaries, like *angr* or Hex-Rays *Interactive Disassembler* (*IDA*). Those tools provide a substantial amount of information to the person analyzing the executable. Based on the knowledge acquired by those tools control flow graphs can be created. Those graphs show the connection between different parts of code and give a hint on the importance of the basic blocks. The person reverse engineering the executable can decide to elaborate a function more closely or in a specific order based on this information. Most obfuscation tools affect this graph in different ways. MOVFUSCATORC removes branches and makes it appear like a linear stream of MOV instructions. It linearizes the graph by hiding the branch information in the *target* and *ON* registers. Analysis of code that accesses those registers will reveal information about the underlying structure and the original basic blocks. First we have to find the addresses of the register controling the execution emulated in memory and their location is not fixed. This data is revealed during the analysis of the static setup. The reconstruction of the original control flow can be split into three phases:

- The first phase scans the executable and builds a map of all labels and their corre-

sponding virtual address in the binary.

- The second phase performs another scan to retrieve all jump targets.
- In the third phase this data will be joined during a final sweep over the executable.

Finding the Labels In normal *x86* assembly labels are used as jump markers. They do not introduce code or overhead and merely act as jump targets. They mostly are a convenience feature so that the programmer does not have to calculate the relative jump distance such that the code stays relocatable. MOVFUSCATORC emulates jumps and execution has to be toggled on at the jump target. Every label that is jumped to introduces code into executable that checks if the current basic block's label is the label targeted in the *target* register. If it is targeted it writes one to *ON*. Returning from calls works similarly. Usually the return instruction `ret` pops one address from the stack and jumps to it. Here, it does not pop an address but rather a label. This label is automatically generated after every call at compile time, so the functionality stays the same. The code introduced at a label always performs three operations:

- Checking if the following basic block is the one with the targeted label
- Restoring of the register values (prevent spilled values)
- If the block is targeted writing one to *ON*, else writing a zero to this location

Labels are the only places where one is written to *ON*. This is done in three instructions where b_0 is the value of the equality check between label and target.

```
1 mov eax, [b0]
2 mov eax, [sel_on + eax * 4]
3 mov [eax], 1
```

After finding an instruction sequence like the above one, we perform a forward taint analysis initially tainting eax (or any other register that is being used as *index* value in the second instruction in case randomization is enabled). If this results in instruction three reading tainted memory, we have found a label.

Next we examine the condition under which the toggle occurs to recover the basic block's label. We perform backwards taint analysis tainting the indexing register of the second instruction. In this case it would be eax. When toggling on b_0 is a temporary register and the taint analysis will find the condition under which it is set to one. For labels the syntax tree of the instructions retrieved from the taint analysis has a very specific form. After replacing accesses to lookup tables with their corresponding function the tree looks like figure 17. Here the complexity introduced by MOVFUSCATORC can be seen. The simple comparison $target == label$ is split into four parts. Each of these parts tests one Byte of the equation and only if all bytes are equal execution is toggled.

This tree then is transformed into a set of axioms and constraints. The label and the memory cell are modeled as 32 bit bit-vectors. While the label is a constant, the memory cell is a variable. We feed this information, together with the axioms from the tree to the SMT solver *z3*. After a check if the problem is satisfiable, z3 finds a function for the target that satisfies the equation. If this function is a constant value, it is a label.

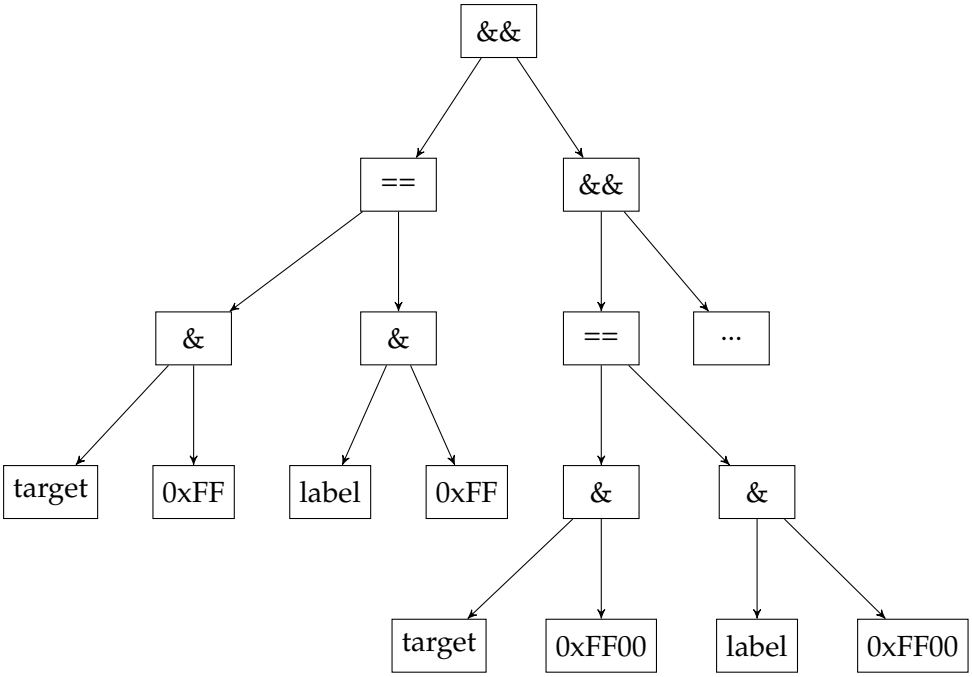


Figure 17: *Syntax tree obtained from backwards taint analysis to find a label*

Finding the Jump Targets In the second phase we focus on getting all the jump targets, so we can later decide what is a *jump*, *return* or *call* statement. The executable accesses targets in a similar way as it accesses *ON*. It has a memory location *sel_target* statically prepared so that *sel_target* points to the discard location and *sel_target* $+4$ points to *target*. The difference is, the position of *sel_on* is easily obtained. In the second phase we do a linear sweep over the executable. Until we find *sel_target*, we analyze the second operand of instructions of the form

```
1 mov r0, [off + r1 * 4]
```

if off $+4$ points to *target*, it is assumed to be *sel_target*. From now on, accesses to *sel_target* are found like accesses to *sel_on*. Upon an access to *sel_target* r_0 is tainted and followed forward, resulting in two possible instructions.

```
1 mov [r0], label
2 mov [r0], r1
```

In the first case, we found a regular or a conditional jump or a call. In the second case we found either a return or an indirect jump. Indirect jumps are most commonly found in implementations of switch case statements, where the jump target is fetched from a jump table. We remember the current jump target and move forward. On finding a *sel_on* that results in toggling off, *ON*, we mark the target label as a jump target.

Joining the Data Joining the data is done during a third pass over the binary. In this phase, we know all labels and whether a label is the targeted by at least on jump. The target of this phase is to build a ordered map M of virtual addresses and corresponding actions, like jumps, calls, returns and label. We assume that a unconditional jump, followed by a label that is not targeted by any jump, is a call. During this phase we also generate a patched version of the binary executable using *x86* `call`, `jmp`, `jnz`, `test` `pop` instructions.

In Section 5 this patched version is compared against the original version in terms of semantic equivalence.

During the third pass we continuously update the current jump target like when we searched for jump targets. But when we find an access to *ON* we elaborate more closely. If execution is toggled on, we look up the corresponding label to the virtual address, acquired during the first phase. This label is then added to our map M. If execution is toggled off, we first taint the index register of the instruction assessing *sel_on*. This will reveal the circumstances under which execution is toggled of. If the analysis results in not just access to *ON*, it is an indirect jump. We add this indirect jump together with the current target to M. If toggling only depends on *ON*, and the target is simply a label, it is a `jmp` or a `call`. To decide whether it is a jump or call we look at the next label. If it is not targeted by a jump we assume that we found a call. Either way we add the `jmp` or `call` to our map M. The last possibility is the indirect jump. This indirect jump is assumed to be a return statement, and also added to M.

Building the Graph Now that we have a the map M containing all the information about our control flow we can start to build a graph of it. MOVFUSCATORC is implemented to first call *main* after setting up the environment. So we know which label corresponds to the *main* function and start from there. To build the graph we use the M and make a copy of it C. We also introduce two lists F and J, each holding virtual addresses. F is a list of function starts that have to be processed, and J is a list of jump targets of the function currently processed. Initially J is empty and F contains the address of the label of the *main* function. If J is empty and F is not the first element is taken from F and put into J. J now only contains the beginning label of a function. An empty directed graph is created. While there are elements in J, the first element e is retrieved from C based on it's virtual address. If e does not exist in C because it may has already been deleted, we continue with the next element from J. It is attempted to create a new node with the virtual address as a key and if such node already exists that node will be modified instead. Then e is deleted from C. If e is a jump an edge with value two to the node corresponding to the target address is created, and if such node does not already exist, it is created. If it is a *call*, the target is added to F. If e is a conditional jump, a call or a label, an edge with value one to the next higher address corresponding to an element in M is created. This is equivalent to continuing execution in the program. If e is not a *jump* or *return*, then the next element with a higher address is retrieved from C which is processed like e. Removing the elements from C prevents the implementation from examining the same jump targets and elements again, so that the run time is bound by the number of elements in M. When J is empty after processing a function the graph can be simplified to represent basic blocks. For that we look at every node and if it only has one successor and the edge has a value of one, both nodes are part of the same basic block. This is due to the fact that a value of one on the edge is equivalent to continueing execution and a conditional jump has two successors. Those two nodes are then joined, adding edges from the first node to the successors of the second node with their respective value. Then the second node is deleted. The first node looked at again if it can be joined with further nodes.

4.3 Patching the Binary

While joining the data from the labels and the jump targets another pass over the executable is performed. During this pass, a patched version of the binary is created. This is done

by replacing instructions in place with other instructions like *call* or *jmp*, that make the control flow explicit again. In this section I will show what instructions will be patched and explain why the semantics of the program stays the same during the patching process. On finding an unconditional *jump*, not a *call*, the instruction that sets *ON* to zero is replaced by a relative jump to the targeted label. More precisely, the end of the check, where *ON* is set to one is targeted. The remaining bytes of the original instructions are overwritten with `nop` instructions. This skips all the code that may not effect the machine, since *ON* is zero, so execution is "off". Calls work very similar, instead of replacing the instruction with a `jmp` a series of instruction is introduced.

```
1 mov esp, [sp]       ; synchronize stack
2 pop eax             ; pop return label
3 call label          ; call function
4 jmp return_label    ; skip the label check
```

First the real stack and the emulated stack are synchronized, then the return label is popped. The the call is executed. This makes sure that on a return, an address, instead of a label is written to the jump target, *target*. At the end a jump to the return label is included. This is necessary, since these instructions overwrite more than one instruction which could have unpredictable side effects. So instead of executing this code, we jump to the dedicated return label for that call. Conditional jumps are a little more complex. The result of the test whether to jump or not can be found in the instruction accessing *sel_on*. If the index register holds one, the jump is executed, else it is not. The index register may only hold either one or zero. We replace this instruction with the instruction "`test R, R`" to update the x86 status register and to set the zero flag if `R` is zero. Then we replace the toggle off instruction with a `jnz label`, like with the unconditional jump. This jump is only executed if the zero flag in the machine status register is set. `Mov` instruction do not affect the machine status register, only the status register of the emulated machine. So this code works independent of the `mov` instructions in between. Last we have indirect jumps that we assume to be return statements.

4.3.1 Recovering of the Original Function Hierarchy

Concurrent to building the control flow graphs of the functions a call graph is generated. To find a call during processing the address of a function is not only added to F, but also an edge is created in the call graph. This edge starts at the function being processed and targets the called function. Differentiation between calls an jumps is done like mentioned earlier. If a label follows a jump that is not targeted by any jump it is assumed to be a call. This assumption is based on the way the MOVFUSCATORC works, namely by labeling basic blocks. To jump to a basic block the label is loaded into *target*. In the beginning of most basic blocks, the label of that basic block is checked against *target*. This entails that for every call there must be a label directly after the call that is targeted by the return.

4.3.2 Instruction Re-Substitution

A big problem working with code obfuscated by MOVFUSCATORC is splitting higher level instructions like addition into many `mov` instructions and multiple look ups into the look up tables. An equality check of two 32 bit registers for example consists of four look ups into the equality table as explained in context of reconstructing jump labels. Combining the results of these look ups takes another three look ups into the boolean and

table, resulting in seven look ups in total in case of a simple comparison. To recover the underlying structure of the higher level instructions I propose two different ways. The first possibility is using symbolic execution to execute one basic block at a time. The resulting mathematical function have to be compiled into *x86* instructions afterwards, a task that may not be trivial, depending on the size of the translated basic block. For the second possibility we start at the end of a basic block. From there we iterate backwards over the instructions. Once we find a writing memory access we taint the source of it. The data flow tree resulting from the taint analysis is then processed. If a part of the tree matches the signature of an operation (i.e. accesses an array known to perform a certain arithmetic operation), the part of the tree can then be replaced by the operation.

4.3.3 Defeating the Randomization

MOVFUSCATORC introduces additional hardening mainly in form of instruction reordering and register renaming. Other methods are substitution of the `mov` instruction by different instructions. Here we will focus on the first two methods. First, register renaming: The flow of information stays the same while the operands of *mov* instructions are renamed, if possible. Clearly, the taint analysis we use to recover the labels and jump targets is fully independent of this particular information and therefore resistant to register renaming. The taint sources and propagation is chosen so that they are independent of the name of the register being tainted.

Reordering instructions can become problematic in an edge case: The instruction writing back to memory can be put after toggling *ON* off. This can be done since the decision whether the change is written back or discarded is made in an earlier instruction, and the result is determined by the point *ON* is toggled off. This may result in a patched version that performs different from the original version and can also effect our current implementation of the taint analysis.

5 Evaluation

This section focuses on the effectiveness of the MOVFUSCATOR obfuscation as well as our approach of deobfuscation. First, we give a rough estimator on the cost of the obfuscation in terms of run time overhead and size of the generated code. In the second part we evaluate the methods as introduced by section 4 using different metrics:

- To measure *control flow similarity*, we compare the number of nodes and edges of an original sample program and its obfuscated and deobfuscated counterparts, respectively.
- In order to show that the presented algorithm leaves the program's original functionality intact, we apply it to an obfuscated version of an implementation of the sha2-256 [7] hash function.
- To measure the *degree* of simplification we also consult a simple instruction count metric
- As a non-academic addition we show that the proposed algorithm significantly decreases the complexity of an analysis of two selected programs whose proper source code we didn't control before: One that had been presented as *challenge* in the 2015 edition of the *Hackover*[8] capture the flag contest and an implementation of the *Advanced Encryption Standard* that had been provided by the author of the MOVFUSCATOR as a proof of concept.

5.1 Obfuscation

The main reason for evaluating the performance of the obfuscated code is to estimate the attractiveness of the MOVFUSCATOR for obfuscating real-world code such as malware.

5.1.1 Size and Time Penalty

First we compare the execution time of a program obfuscated with MOVFUSCATORB against a static translation from Brainfuck to machine code. We do not use an interpreter to execute the code since MOVFUSCATORB creates a file in assembly language which

[8] `https://hackover.de/`

Name	MOVFUSCATORB time	MOVFUSCATORB size	Static Translation time	Static Translation size	Description
sum	26 ms	285 KiB	0.7 ms	11.2 KiB	calculates the sum of digits
prime (20)	1398 ms	352 KiB	2.87 ms	28.9 KiB	calculates the primes up to 20
factor (84)	292 ms	486 KiB	36 ms	43.6 KiB	splits 84 into its prime factors

Table 6: *Comparison between* MOVFUSCATORB *and a static translation of the code, averaged over five runs*

is assembled into machine code. To reduce the overhead introduced by an interpreter we translate each brainfuck instruction into an equivalent c instruction and compile that program. For translation we use a slightly modified version of Steve Johnsons[9] implementation. The only thing I changed was allocating more memory as the array of cells and also setting the pointer to roughly one third into it. This should prevent segmentation faults by walking left or right out of the array. The resulting C file is compiled using the gnu c compiler without optimization. Table 6 shows the overhead in terms of size and run time introduced by obfuscation of Brainfuck code using MOVFUSCATORB. As can be seen from the table, employing the MOVFUSCATORB increases program size by an factor of about 10 to 20. Run times vary drastically from about a factor of 4 up to a factor of about 500 depending on the application's logic. An odd thing to mention is that for MOVFUSCATORB, prime has a longer run time than factor while the run time of the un-obfuscated versions is exactly the other way round. One possible explanation for this is that factor internally works using trial division, an algorithm that is very costly when implemented in brainfuck.

For the MOVFUSCATORC we created a small test bench of dummy programs with an easy control flow, so we can do proper evaluation. These programs are *call*, *for_loop*, *if_then_else* and *switch*. All of them represent a basic control flow concepts like loops, conditional jumps and calls. They are very small in code size, so most of the program's size is static overhead introduced by the executable format they're stored in. The run time of their un-obfuscated versions also is pretty similar, because initialization takes a significant amount of time for such small executable programs. To properly compare the overhead we created *primes*, an implementation of the sieve of Eratosthenes. Also an implementation of the advanced encryption standard, tiny-aes128[10], is tested. A third example is the hash function sha2-256 as implemented by Brad Conte[11]. Some minor adjustments were needed to get lcc to compile the SHA implementation correctly. As seen from Table 7 the size overhead introduced by the look up tables significatly amounting around 5.5 MiB. While the run time overhead for the small programs is negligible, it becomes quite large when adding loops and complexity to the original, un-obfuscated version of the program.

It is difficult to make a general estimation of the run time penalty due to the fact that for every backwards jump, the whole program will be executed once more, effectively adding a term depending on the total size and the layout of the program to its complexity. As seen in the table the program will become slower by factors of thousand, but it is hard

[9] `http://awk.info/?doc/bfc.html`
[10] `https://github.com/kokke/tiny-AES128-C/`
[11] `https://github.com/B-Con/crypto-algorithms`

name	MOVFUSCATORC time	MOVFUSCATORC size	lcc + as + gcc time	lcc + as + gcc size
call	0.98 ms	5.67 MiB	0.76 ms	3.11 KiB
for_loop	0.99 ms	5.67 MiB	0.77 ms	3.05 KiB
if_then_else	0.95 ms	5.67 MiB	0.76 ms	3.13 KiB
primes (100)	3.50 ms	5.71 MiB	0.78 ms	3.58 KiB
switch	1.01 ms	5.67 MiB	0.77 ms	3.20 KiB
tiny-aes128	2591 ms	6.28 MiB	1.18 ms	12.7 KiB
sha2-256[12]	81425 ms	5.87 MiB	28.8 ms	6.13 KiB

Table 7: *Runtime comparison between the* MOVFUSCATORC *version and the x86/linux version averaged over 1000 runs*

to estimate the exact factor, how much slower it will become. Measurements have been done on an Intel Core I5 M 460 clocking at 2.53 Ghz. Clearly, obfuscating complicated algorithms with the MOVFUSCATORC is not a good idea if performance is an issue.

5.2 Deobfuscation

In this subsection I will evaluate the proposed methods to de-linearize the control flow graph and retrieve an semantic equivalent binary that uses explicit control transfers, like jumps and calls.

5.2.1 Control Flow Flattening

To show the flattening of the control flow we compiled several small programs using MOVFUSCATORC. We then used the described methods to retrieve the underlying control flow graph of the simulated machine. We also compiled the same program with the same compiler, only using the x86/linux target instead of the mov target. We generated a control flow graph from the second binary as well and compared them with each other.

First I want to examine a basic program only containing one if-then-else statement. It outputs the number of parameter passed to the program, its code can be found in listing 3. As additional hardening the register renaming script has been applied to the assembly code. The reference graph and the generated graphs are straight forward. Both show the entry into the function first, then two blocks which are executed under a certain condition and in the end the function epilogue, as seen in Figure 18. As additional hardening instruction reordering has been applied. It only affected the names of nodes, as the start end end position of basic blocks changed.

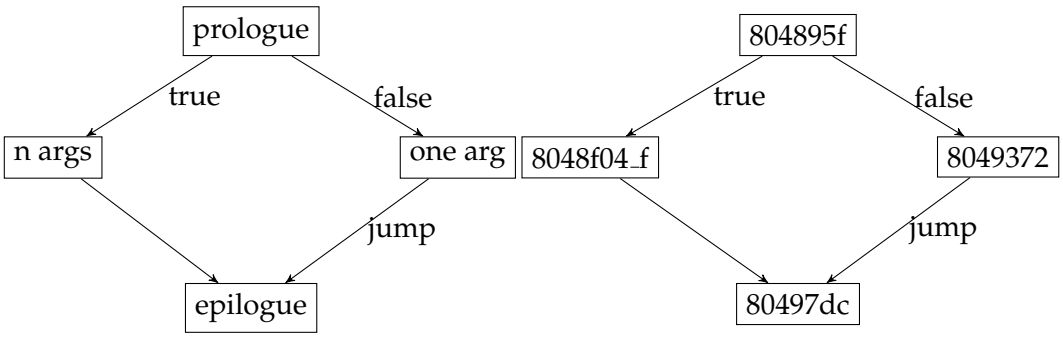
The second example is an implementation of a for loop. The for loop is iterated over ten times. On every of these iterations it prints out the number of iterations it already did, see listing 4.

The control flow graph of the reference executable shows the flow of the program very clearly. The first basic block includes the function prologue and initialization of i. The second basic block is the body of the for loop, with the check on the condition in the end. The condition is not checked on first entering the loop, because the range of the loop is fixed, and it will always be executed at least once. The last basic block contains the

```c
#include <stdio.h>

int main(int argc, char argv) {
        if (argc == 1)
                puts("There is one argument");
        else
                printf("There are %d arguments\n", argc);
        return 0;
}
```

Listing 3: *if-then-else program*



Reference graph of if-then-else *Graph of if-then-else by analysis*

Figure 18: *Control flow graphs for the if-then-else program*

function epilogue and returns zero. Figure 19 is the graph retrieved from the interactive disassembler (IDA)[13], while Figure 20 shows the graph generated by the analysis. In this simple example isomorphism can easily be seen. The analysis outputs the virtual address of a block as its name, while IDA outputs either a corresponding label or assembly code. The labels of the reference graph have been changed to simplify it. The additional hardening applied to this program was register renaming. It had no effect on the analysis and the resulting graph, as expected.

The third control transfer operation I want to show in a small example is the call instruction. To do so I will evaluate the call graph as well as the, control flow graphs of the functions. The program takes the first command line argument which is its name by convention. Then it calls a function that prints all text following the last slash.

In the call program, all the main function does is calling *print_name*. After a short inspection of the call graph, and relabeling *fun_804a6f7* with *main*, and *fun_804891e* with

[13] `https://www.hex-rays.com/products/ida/index.shtml`

```c
#include <stdio.h>

int main(int argc, char argv) {
        int i;
        for (i = 0; i < 10; i++)
                printf("loop run #%d\n", i);
        return 0;
}
```

Listing 4: *for_loop*

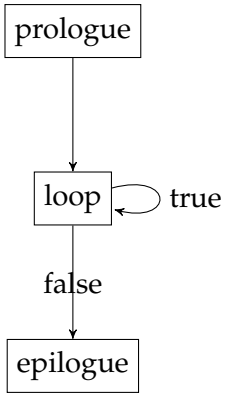


Figure 19: *Reference graph*

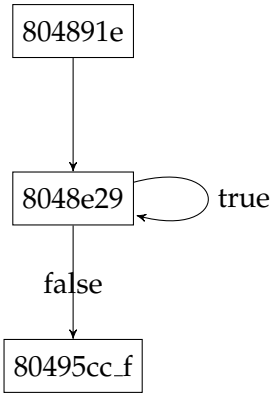


Figure 20: *graph from analysis*

```c
#include <stdio.h>

void print_name(char *path) {
        int i;
        int p = 0;
        for (i = 0; path[i] != 0; i++) {
                if (path[i] == '/')
                        p = i;
        }
        puts(path + p + 1);
}

int main(int argc, char** argv) {
        print_name(argv[0]);
        return 0;
}
```

Listing 5: *the call program*

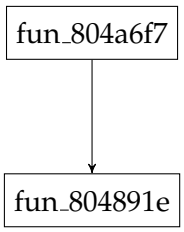


Figure 21: *The generated call graph*

print_name, the resulting graph is as expected.

At last I will evaluate code generated for a switch case statement, as seen in Listing 6. Switch case statements usually differ strongly in the way they are implemented from an if then else statements. Instead of checking every case, a jump table is used. This is done in two steps, first the program checks that the argument is within the bounds of the cases, if it is not the default case is jumped to. Second, an offset is calculated based on the switch value. A table containing the jump targets for all cases is then accessed at the calculated offset, effectively jumping to the case handling that value. The MOVFUSCATORC uses labels instead of addresses, but the way the switch works is the same. Since the label originates from memory an indirect jump is used to dispatch the jump. One assumption that I made during the analysis was, that indirect jumps are return statements. The graph in Figure 22, clearly shows a lack of basic blocks compared to what was expected. Going back to how the algorithm works, we can understand the wrong behavior. On encountering an indirect jump, it is assumed that this jump is part of a return statement, so a return element is added to the graph and the path is not further investigated. The actual target is not retrieved, so the algorithm has no knowledge about the labels targeted by the switch and instead of adding branches, the algorithm finishes.

Apart from the above mentioned examples I also evaluated a more complex program, an implementation of the secure hash algorithm 2. The number of functions that have

```c
#include <stdio.h>

char data[7][7] = {"never ", "gonna ", "
    give ", "you ", "up ", "let ", "
    down "};

int main(int argc, char **argv) {
        char *txt;
        switch (argc) {
                case 1:
                case 6:
                        txt = data[0];
                        break;
                case 2:
                case 7:
                        txt = data[1];
                        break;
                case 3:
                        txt = data[2];
                        break;
                case 4:
                case 9:
                        txt = data[3];
                        break;
                case 5:
                        txt = data[4];
                        break;
                case 8:
                        txt = data[5];
                        break;
                default:
                        txt = data[6];
                        break;
        }
        printf("%s\n", txt);
        return 0;
}
```

Listing 6: *switch case program*

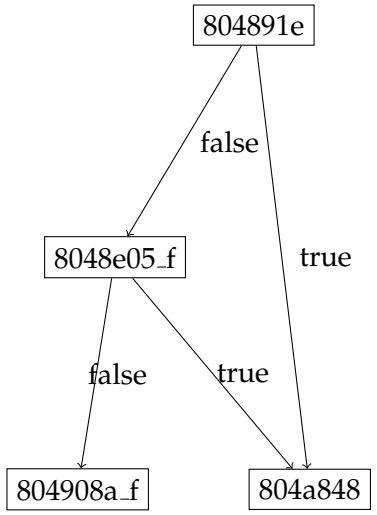


Figure 22: *graph generated for switch-case*

been found is equal to the number of functions in the binary, also the call graph generated is isomorph to the call graph for the reference executable, generated by IDA. For every control flow graph of a function, a isomorph control flow graph of the reference could be found, giving us a set of isomorph graphs. This suggests that the implementation of our algorithm for deobfuscation behaves within reasonable boundaries.

5.2.2 Empirical Correctness

We did not only compare the generated graphs against references, but we also tried to execute the patched binary and find any differences in the program's behaviour. First we tested all of the above programs. All acted as their non modified counter part with the exception of the switch program that did not terminate for ten seconds and was then stopped.

Exemplary for a complex real-world program we also tested tiny-aes128-C, an small implementation of AES, where the output of the modified version matched exactly the output of the original version.

Last we also applied our algorithm on an obfuscated version of the SHA2 function, more specifically SHA2-256. The only change I did to the original implementation was the

use of only 32 bit wide integers, instead of 64 bit wide ones. This is due to the fact, that lcc produced an incorrect executable that generated a wrong hash value. Using 32 Bit wide integers fixed this issue. With this change, all three versions passed the test and returned the expected result.

6 Summary

6.1 The MOVFUSCATOR in the Real World

One of the goals of this thesis is the evaluation of control flow linearization and instruction set reduction in terms of real world attractiveness. To do so, it is a reasonable choice to compare its benefits against the penalties a program suffers when being obfuscated using the MOVFUSCATOR from an defender's (obfuscator's) point of view. This is done to estimate the relevance of these techniques in obfuscating different programs.

Like mentioned in the beginning, the potential subjects employing control flow linearization could be in *whitebox cryptography* as used in digital rights management. Malware developers also use obfuscation to hide the functionality of their programs, to stay undetected or make an analysis very time consuming.

First I want to point out supporting factors of the MOVFUSCATOR. It is a new technique that instead of flattening the control flow or adding code that can not be reached under normal circumstances, linearizes it. Jumps are emulated causing the whole program is executed as a single basic block in a loop. This effectively reduces the program to a single basic block spanning all instructions. Instead of looking at the functions one by one, an analyst now has to consider the program as one block and has to find function primitives first. Instruction set reduction further increases the complexity of this one basic block by replacing arithmetic instructions by many MOV instructions.

The downside to these techniques is the added overhead in terms of generated code size and run time, as seen in Subsection 5.1. Most of that size overhead is due to the fact that MOVFUSCATORC uses many look up tables to perform the different operations trading in memory for improved run-time performance. MOVFUSCATORB does not suffer from a size penalty that severe, since it only needs a few look up tables. The downside to MOVFUSCATORB is that because it only has a few look up tables other functionality has to be emulated, leading to a enormous time overhead. While the time overhead of a program obfuscated with MOVFUSCATORC is not that high, it is still much slower than its not obfuscated counter part. Due to that time penalty it is practically not usable in systems that need a high throughput of information, like streaming high definition video over the internet or stealing personal data by encrypting them on disk. MOVFUSCATOR could be attractive for malware but the increase in size makes it harder for the malware to hide and may be a hindrance when it tries to spread over the internet. Additionally, a program consisting of only one type of instructions can easily be flagged by anti-virus scanners.

6.2 Towards Practically Feasible Deobfuscation

One of the core features of the MOVFUSCATOR is its ability to hide the effective control flow. The core contribution of our work is a generic approach that recovers the control flow of a protected binary, as shown in Section 4. Using the techniques described there the control flow consisting off the position and the size of the basic blocks could be reconstructed in most cases most cases. Due to time constraints, my approach can currently not handle indirect jumps, so it doesn't recover the control flow of all possible programs, but at least a good portion of it (including sophisticated hashing and encryption algorithms). The presented approach is sound against arbitrary register renaming and to a certain extent to instruction re-ordering.

6.3 Future Work

With knowledge of the control flow of the program and the virtual addresses of the basic blocks an analyst can split the executable into multiple parts and look at the basic blocks one at a time. Symbolic execution can then be employed to execute such basic blocks and retrieve the original instruction semantics. A way around the lack of processing indirect jumps is to use binary instrumentation. Instead of executing the indirect jump in the patched version, the jump target, obfuscated as a label, can be looked up in the list of labels retrieved during analysis. A relative jump to this label is then performed in place of the indirect jump. In case of a switch case statement this will only give one target corresponding to the statement.

As seen in the evaluation reconstruction of the control flow works in most general cases, with only indirect jumps posing a problem. With indirect jumps commonly employed in the implementation of switch case statements this incapability is a serious lack in usefulness. I will try to address this issue in future work by improving my analysis methods and correcting the wrong assumption that indirect jumps are return statements. A second issue I would like to address is retrieving the higher level instructions. The assembly code is not easy to read and with simple operations spread out over multiple instructions analysis can become very difficult. This may be done by finding accesses to the look up tables and analyzing the binary locally around to find a pattern that can be matched to an operation. If such pattern is found the involved instructions could be replaced by fewer, more explicit instructions.

References

[1] Oskar Arvidsson. Platform Independent Code Obfuscation, 2014.

[2] Clark Barrett, Leonardo de Moura, and Pascal Fontaine. *Proofs in Satisfiability Modulo Theories*. Mathematical Logic and Foundations. College Publications, London, UK, 2015.

[3] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. Oorschot. *Digital Rights Management: ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002. Revised Papers*, chapter A White-Box DES Implementation for DRM Applications, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[4] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Oorschot. *Selected Areas in Cryptography: 9th Annual International Workshop, SAC 2002 St. John's, Newfoundland, Canada, August 15–16, 2002 Revised Papers*, chapter White-Box Cryptography and an AES Implementation, pages 250–270. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[6] Bruce Dang, Alexandre Gazet, Elias Bachaalany, and Sébastien Josse. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. John Wiley & Sons, 2014.

[7] Quynh H. Dang. *Secure Hash Standard (SHS)*, 2015.

[8] Stephen Dolan. mov is Turing-complete.

[9] Christopher Domas. The Movfuscator, 2015.

[10] Vijay Ganesh. Decision Procedures for Bit-Vecotrs, Arrays and Integers, 2007.

[11] S. Ghosh, J.D. Hiser, and J.W. Davidson. Matryoshka: Strengthening Software Protection via Nested Virtual Machines. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 10–16, May 2015.

[12] Gerhard Goos. *Vorlesung über Informatik*. Springer-Lehrbuch, 1997.

[13] David Hanson and Christopher Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[14] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, 2nd edition, 2002.

[15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM: Software Protection for the Masses. In *Proceedings of the 1st International Workshop on Software Protection*, SPRO '15, pages 3–9, Piscataway, NJ, USA, 2015. IEEE Press.

[16] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 61–70, Oct 2012.

[17] C. Liem. System and method for aggressive self-modification in dynamic function call systems, November 24 2015. US Patent 9,195,476.

[18] Jasvir Nagra and Christian Collberg. *Surreptitious Software*. Addison-Wesley Professional, 2009.

[19] Rolf Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[20] Karen Scarfone, Wayne Jansen, and Miles Tracy. Guide to General Server Security. 2008.

[21] Emmanuel Sifakis and Laurent Mounier. Predictive Taint Analysis for Extended Testing of Perallel Executions. In *Hardware and Software: Verification and Testing*. Springer International Publishing, 2013.

[22] Brecht Wyseur. *Encyclopedia of Cryptography and Security*, chapter White-Box Cryptography, pages 1386–1387. Springer US, Boston, MA, 2011.