Implementing a tool for better obfuscation using overlapping instructions

Erik Nylander
ada09eny@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisor: Christopher Jämthagen

May 5, 2014

Printed in Sweden
E-huset, Lund, 2014

Abstract

The abstract. The abstract.

Table of Contents

Chapter 1
Introduction

Chapter 2

Background

2.1 CISC

CISC (Complex Instruction Set Computer) denotes a series of microprocessor architectures mainly used in personal computers today. When the first CISC processors were launched, memory was expensive so in order to conserve memory space, instructions were designed to take up as little space as possible when used in programs. At the same time, engineers struggled to bridge the semantic gap, meaning the gap between high-level code and assembler code which would allow for better compilers as the assembler code approached high-level functionality. This resulted in a large set of instructions, many of them able to carry out several low level operations such as loading of data from memory, arithmetic instructions and storing of data to memory. In the end, this approach proved to be lacking because it made compilers difficult to implement since there were so many instructions to choose from. Another big problem was that the processors were hard to implement in hardware because the instructions became so complicated. The most important aspects of CISC architectures are as follows: [Bro99]

- An extensive set of instructions to provide a lot of possibilities when coding.
- Powerful instructions performing several low level operations meant to bridge the gap between assembler code and high-level code.
- Instructions of variable length.

2.2 RISC

As the limitations of CISC became more apparent in the 1980s, researchers at University of California at Berkeley and Stanford University started looking for less troublesome, more simple and overall more flexible alternatives. This research resulted in a new type of computer architecture called RISC (Reduced Instruction Set Computer). RISC aimed to make microprocessors easy to implement in hardware and easier to code for by using a much smaller but more straight forward instruction set. The developers of RISC always had the words "make the common case fast" in mind while developing. This largely contrasts the way of thinking while developing the older CISC-architecture where the developers aimed to make very powerful and specific instructions in order to make the assembler code more similar to high-level code while using as little memory as possible. The most important aspects of the RISC architecture are the following:

- All instructions are of the same length. This makes the instructions easier to decode and it also makes the architecture much simpler to implement in hardware.
- To provide many registers without any special purposes, allowing for variables and temporary calculations to be stored in registers as opposed to being stored in memory resulting in much faster calculations.
- Only certain load and store instructions may access memory. The purpose of this is to reduce the amount of low level operations an instruction can perform which makes more effective use of the ALU while at the same time reducing the complexity of the instructions.

Even though the RISC architecture had many advantages over CISC, it took a long time for RISC to take its place, mainly because of the lack of software support and the fact that memory was still expensive. Today however, RISC is the most commonly used microprocessor-architecture used overall while CISC still dominates the personal computer-market. Some well known manufacturers of RISC-chips are MIPS, Sun Microsystems(SPARC) and ARM. [Bro99]

2.3 IA-32 architecture

IA-32 is the third generation of x86 microprocessor architectures made by Intel. When it was launched in 1985 together with the Intel 80386 microprocessor it was the first 32-bit architecture from Intel, fully backwards compatible with programs made for previous generations of 16- and 8-bit processors. x86 in itself is a CISC architecture which together with the backwards compatibility of IA-32 to its predecessors made for a very complicated platform with an extensive set of instructions with different levels of complexity. Another consequence of this is that the instructions can vary in length from one byte up to fifteen bytes which unintentionally open some interesting possibilities for code obfuscation..

With the new 32-bit architecture and Protected Mode, the Intel 80386 could address up to 4 gigabytes of memory. However, when using Protected Mode, all programs are assigned a certain amount of memory to use in order to prevent programs from interfering with each other and any attempt to access memory outside of the allotted amount results in a segmentation fault.

2.3.1 The IA-32 instruction set

Being a CISC architechture, the instruction set for IA-32 contains a very large amount of instructions which can be roughly divided into the following categories: [AF14] [Hyd96]

Data movement instructions Instructions to move data around between memory or registers.

Arithmetic instructions These instructions perform arithmetic operations like addition and subtraction.

Logical instructions Logical instruction perform logical operations such as AND, OR and other bit-operations.

Control flow instructions These instructions change the control flow of the program.

I/O instructions I/O instructions gives access to the I/O ports on the processor in order to read data from or write data to peripheral devices.

String instructions String instructions are used to operate on data in the form of strings of bytes.

Interrupts Interrupt instructions invoke different kinds of software interrupts in the processor.

2.3.2 Looking at an x86 instruction

The IA-32 instruction format is shown in Table 2.1:

Prefix	Opcode	Mod-Reg-R/M	SIB	Displacement	Immediate
1-4 bytes	1-2 bytes	1 byte	1 byte	1-4 bytes	1-4 bytes

Table 2.1: The IA-32 instruction format

Prefix This is an optional field used to add prefixes to certain instructions in order to override segment sizes, operand sizes and so on.

Opcode The opcode consists of 1-2 bytes and is the instruction code that determines the type of instruction. Some instructions contains only of this field.

Mod-Reg-R/M The Mod-Reg-R/M byte is used in a couple of different ways, mainly to determine if an instruction will access memory or registers. This field will be explained in more detail later.

SIB SIB stands for *Scale-Index-Base* and is used to enable index-based addressing of memory. This field will be explained in more detail later.

Displacement The displacement field is used to access memory with a displacement. If the mod-reg-r/m-byte gives the register that the memory address will be collected from, the displacement field will determine a value to add to the address before accessing the memory.

Immediate Instructions that uses some kind of immediate value gathers this value from this field instead of picking one from a register or memory.

The Mod-Reg-R/M byte

The Mod-Reg-R/M byte consist of three groups of bits that together determines what kind of data that the instruction is accessing. These groups are explained below:

Mod	Reg	R/M
2 bits	3 bits	3 bits

Mod
The two *Mod* bits determines if the instruction shall access memory at an address stored in a register and also if a displacement shall be added to the address.

00 The processor will use indirect addressing mode. The address will be collected from the register specified by the r/m bits. Two exceptions are when the r/m field is *100* or *101*.

01 Indirect addressing mode but an 8-bit displacement will be added to the address before it is dereferenced.

10 Indirect addressing mode but a 32-bit diplacement will be added to the address before it is dereferenced.

11 Direct addressing mode, no memory will be acccessed. Both the reg and r/m fields will denote registers.

Reg
The *Reg* field will denote a register:
000 - eax, 001 - ecx, 010 - edx, 011 - ebx, 100 - esp, 101 - ebp, 110 - esi, 111 - edi.
R/M
The *R/M* field will denote a register just like the reg field. It also denotes the register from which an address will be dereferenced. There are two exceptions to this, namely if the mod bits are 00 and if the r/m bits are 100, then the SIB byte will be used when accessing memory or when the r/m bits are 101, then a displacement value will be dereferenced instead.

The SIB byte

The SIB byte like the Mod-Reg-R/M byte also contains three fields of bits called *Scale*, *Index* and *Base*. As mentioned before, this byte enables index-based memory access. When the SIB byte is used, the address being accessed will be calculated as follows: $Base + Scale * Index$. The fields in the SIB byte will be explained below:

Scale	Index	Base
2 bits	3 bits	3 bits

Scale
The two *Scale* bits determines what value that the *Index* value will be multiplied

with.

00 *Index* will be multiplied by 1.

01 *Index* will be multiplied by 2.

10 *Index* will be multiplied by 4.

11 *Index* will be multiplied by 8.

Index and Base
The *Index* and *Base* fields both denotes registers encoded in the same way as the *Reg* and *R/M* fields in the Mod-Reg-R/M byte. [Art13] [Irv10]

2.4 Code obfuscation

Code obfuscation is the action of deliberately making code harder to analyze, interpret and recreate. In the case of machine code, the idea is to make the code hard to disassemble correctly and hard for humans to read and understand once it has been disassembled. The creator of some program can choose to obfuscate their code for a variety of reasons, both benign and malevolent e.g a legitimate programmer may want to hide certain parts of a program, for example which ports the program listens to or the algorithm that determines if a serial code is correct while on the other hand a hacker might want to make his virus harder to analyze which could delay the analysis of the virus and the creation of a fix which in turn would let the virus infect a larger number of hosts. By achieving a high level of obfuscation, the risk of a program getting cracked, copied and spread on the internet is greatly reduced. It will also be less susceptible to hacker attacks since an adversary will have a difficult time finding security flaws to exploit.

2.4.1 Disassemblers

There are generally two different kinds of disassemblers namely static disassemblers and dynamic disassemblers. Static disassemblers use either a linear sweep algorithm or a recursive traversal algorithm to decode the instructions in a program. A linear sweep disassembler works in such a way that it reads the program byte after byte, decoding instructions as they come along. A clear advantage with this approach is that the entire program can be disassembled in a single sweep. It is also fairly quick since it only depends on the size of the program. The perhaps most critical disadvantage this approach has is that it does not take the control flow of the program in to account causing it to misinterpret things like data embedded in the stream of instructions.

Recursive traversal disassemblers works such that whenever they encounter a function call or a branch instruction it calculates the possible target address of the instruction, marks the current address as visited and then proceeds to recursively call its disassembler function with the potential target addresses of the current instruction. If it encounters an address that has already been visited, the function returns. This enables the disassembler to avoid interpreting embedded data as executable instructions.

A dynamic disassembler executes the code at the same time as it disassembles the instructions by using an attached debugger, allowing it to follow the control flow of the program by executing branch instructions and function calls.

Since an analyst can use different kinds of disassemblers, the obfuscator must take the weaknesses of each in account when obfuscating the code in order to achieve a high level of obfuscation to confuse the analyst.

2.4.2 Obfuscation techniques

Over the years, several obfuscation techniques have been invented and employed, ranging from simple tricks to confuse linear sweep disassemblers to sophisticated self modifying code. Some notable approaches will be presented below.

Junk byte insertion

This obfuscation technique aims to exploit the weakness of linear sweep disassemblers by embedding data among the executable instructions. Upon disassembly, the disassembler will interpret the junkbyte as the opcode to the start of an instruction, which could cause the following bytes to be interpreted as the rest of the instruction. This would make the program look completely different from the program written without the junkbyte. Note that a recursive traversal disassembler follow the control flow of the program so that it never encounters the junk byte. An example of junk byte insertion is displayed below. [CJ13]

Consider the following code:

```
jmp label               eb 01
db 0x8d                 8d    # junk byte

label:
xor eax, 0x12345678     35 78 56 34 12
int 0x80                cd 80
```

After assembling the code above and then using the GNU Assembler *objdump* [obj14] on it we see that the linear sweep algorithm produces the following code:

```
$ objdump -D -b binary -mi386 -M intel example.o
   0: eb 01                jmp    0x3
   2: 8d 35 78 56 34 12    lea    esi,ds:0x12345678
   8: cd 80                int    0x80
```

Here it is very clear that the linear sweep algorithm in *objdump* interpreted the data byte 0x8d as the the opcode for the x86 *Load Effective Address* instruction making the disassembled code look different from the code that was actually written. An important thing to note is that the junk byte must be placed so that it will never be reached, potentially causing the program to crash. Another important thing to notice in this example is the fact that the instruction *int 0x80* appears in both code snippets. This is due to the fact that the disassembler tends to synchronize with the correct code in a few instructions (one in this case) because of the variable length of x86 instructions.

Opaque predicates

Opaque predicates is a technique used to fool recursive traversal disassemblers and can be used together with junk byte insertion. The idea is to convert unconditional jump instructions in to conditional jump instructions but make it so that the jump is always taken at runtime. This forces the recursive traversal algorithm to consider both the target as well as the next instruction after the jump. By placing junk bytes directly after the jump instruction, the disassembler can be tricked into producing erroneous code. [CJ13]

Modifying this technique could also fool a dynamic disassembler. Since a dynamic disassembler will disassemble the code while it is being run, it will only encounter the legitimate code and never reach the junk byte. However, this could be overcome by making the seemingly unconditional jump conditional. This might seem confusing but if one were to make the conditional jump depend on the fact that a debugger was present or not, one could make the jump be not taken if that was the case. Then the dynamic disassembler would be forced to disassemble the junk byte, making it produce faulty code.

`example`

Instruction embedding

Instruction embedding [LSPM12] is an interesting obfuscation technique because it effectively hides one or more instructions inside another instruction. An instruction a embeds an instruction b if the last x bytes of a consist of all the bytes in b. The most important part of any obfuscation technique is to not change the behaviour of the program, luckily, the x86 architecture allows jumps into the middle of an instruction. This means that a jump can be made into the embedding instruction so that the next instruction to be run is actually the embedded instruction.

Since the embedding instruction must be able to contain the entirety of the embedded instruction there are constraints on what instruction can be embedded and which instructions they can be embedded in. The best instructions to use as embedding instructions are those that utilize both a 32-bit displacement value and a 32-bit immediate value, then it will contain 8 bytes that are fully customizable which means that an instruction with the length of 8 bytes can be embedded.

`example`

Using cryptographic hash functions or pseudo-random number generators

These two approaches both stem from the same idea, namely to generate the code at runtime so that it is never visible to an analyst using a static disassembler. Because the code is generated first when the program is run, it is only present at runtime and only in memory, which means that there is no code to analyze when the program is not being run.

In the case of using pseudo-random number generators, the basic idea is to find a seed s to a built-in pseudo random number generator G on the host such that a number of consecutive calls to G will generate the desired bytes. This is often very hard to do since the probability of finding such a seed is very small, especially as the number of desired bytes grow in size. Because of this the method can be altered by using different kinds of mappings between the generated numbers and the desired bytes. [JA09]

When using cryptographic hash functions, the idea is similar. By using some key combined with a salt value together with some cryptographic hash function, a sequence of bytes will be generated. The goal is to find the key, a salt and a hash function such that they together generate the bytes that make up a program. Like the case when using pseudo-random number generators this is very hard to achieve which is why this form of anti-disassembly is rarely used. [AdJ06]

Both of these approaches depend on finding certain values to make them work. The only way to do this is by doing an exhaustive search for suitable values which in turn requires a lot of computing power. If this search space could be divided among a large amount of hosts, time it would take to obtain the desired values would be greatly reduced. An interesting thing to note is that this kind of distributed computing mainly exist in botnets and that a programmer with a botnet at his disposal usually has malevolent purposes in mind.

Chapter 3

Instruction overlapping

3.1 Overview

Instruction overlapping is the obfuscation technique of which the main focus of this thesis lies on. It is similar to instruction embedding in that it hides instructions inside others but it is also more advanced and a more elegant solution. In the case of instruction embedding, an instruction hides another and a jump is needed to execute the hidden instruction. This requires a large amount of jump instructions when a large amount of instructions are hidden which could seem suspisious to an analyser. At the same time, it would not stand a chance against a dynamic disassembler since the control flow of the program is followed and only the instructions executed are disassembled. Instruction overlapping in its simplest form attempts to overcome the former of this problem by making the code look normal. With a bit of tweaking it can also overcome the second problem by making the disassembler execute the non-hidden code.

The main difference between instruction overlapping and instruction embedding is that instruction embedding hides one instruction inside another while in instruction overlapping, several instructions can be hidden inside several others. The main idea is that when looking at a program as a stream of bytes, if the program were to be run starting with the first byte in the stream, one execution path would be executed while if the program were to be run starting at an offset from the first byte, another execution path would be executed. The perfect scenario for this technique would be to have two completely executable paths (non of the executions paths crasches the program) so that the execution path starting at the first byte is the visible code (to the analyst) and the executions path starting at an offset is the hidden code. If this result was achieved, only one jump instruction into the hidded execution path would be needed to run the program as opposed to the case with instruction embedding where serveral jumps were needed. This would make the program look natural and not as suspisious which solves the problem with strange looking code when using instruction embedding. [CJ13]

3.2 Overlapping instructions

3.3 Main Execution Path and Hidden Execution Path

As mentioned before, instruction overlapping relies on having two executable paths in one program, the *Main-Executaion-Path*(from here on called MEP) and the *Hidden-Execution-Path*(from here on called HEP). The MEP is the path visible to an analyst and the HEP will consist of the program that is hidden.

Chapter 4 Implementation

4.1 Programming Language

The tool was implemented using C++ with some additional C-libraries. The choice of programming language largely depended on three main factors; speed, flexibility and convenience. C++ was the natural choice because it offers the convenience of the C++ standard libraries which offer good and well implemented container classes and iterators while still maintaning C-like performance.

4.2 Main features

The main features considered for the tool were:

- Being able to parsing and represent a program
- Being able to insert one or more bytes before the instructions in the HEP to optain a MEP
- Being able to determine which of the inserted bytes generates the longest and best MEP
- Swapping instructions in the HEP if illegal or unsuitable instructions are found in the correspondig place in the MEP
- Compensating for memory access in the MEP
- Emulate the instructions in the MEP and determine if a jump instruction is legal or not if one is found in the MEP

All of these features were fully implemented or implemented to an extent in the final product.

4.3 Input and Output

The tool was implemented so that a specific input generates an output in the form of a template for the obfuscated program read from the input.

4.3.1 Input

The intended input for the tool is a file containing the bytes representing an assembled program. The input file is assumed to be assembled in a way such that only the representation of the instructions are present in the input file, i.e. no code to make the program runable are added when the instructions are assembled.

4.3.2 Ouput

The output of the tools is presented as a template for the obfuscated program that was generated from the input file. It contains aside from the instructions a data section where data is allocated, a preface part and a memory adjusment part, former two will be explained further on.

4.4 Parsing and representing a program

In order to perform operations on a program it needs to be parsed and represented in some way. This part is done in two steps in the tool, first the input file is read and the bytes are stored in a *vector*, then the bytes are parsed and stored in *Instruction* objects.

Since the parsing and representation of x86 instructions would be very time consuming to implement, two different third party C libraries were used, Udis86 and Libemu.

4.4.1 Udis86

Udis86 is a disassembler library for x86 written in C. It is used to disassemble a stream of bytes and presenting them result both in hexadecimal byte form aswell as in assembly language. It can disassemble bytes coming from either a byte array or directly from a file. In the tool it is used to read a file and return the bytes and to present the output. [udi14]

4.4.2 Libemu

Libemu is a C library used for emulation. It provides the tools to build code analysis programs tailored to specific needs. The library contains an advanced emulator capable of executing a stream of instructions and present the resulting outcome. Alongside the emulator, Libemu offers debugging capabilities.

The tool uses Libemu to parse a stream of bytes and store them in Libemu’s internal instruction object. The reason for using an instruction object as opposed to only using the bytes and derive the data necessary from them is that this offers an extra layer of abstraction so that all the instructions can be handled the same way instead of having to utilize ad-hoc style handling for each different type of instruction. The tool wraps the internal instruction object to add some capabilities. [lib14]

4.5 "Preface"

Some instructions are very long, therefore they are difficult to hide. At the same time they often seem to make the MEP sync with the HEP. In order to avoid this behaviour the tool uses a few tricks to substitute these kinds of instructions by moving them out of the HEP and replace them with other instructions that makes the program behave the same but makes the HEP easier to hide.

After the input file has been read and parsed, a function called `doPreface()` is called. This functions loops through the instructions and looks for certain kinds of instructions. If one is encountered, it is substituted for one instruction in the Preface part of the program and one instruction in the HEP. How this is done is different for different kinds of instructions.

4.5.1 Workaround for LEA instructions

The *Load Effective Address* instruction loads an address and places it in a register. For example: `lea eax, [somedata]` computes the address of the source operand `somedata` and stores it in the destination register, `eax` without actually accessing the memory.

If the address of `somedata` in the example above was `0x12345678`, the instruction would be encoded as follows: `8d 05 78 56 34 12`. This instruction is the 6 bytes long which can be hard to hide.

What happens in `doPreface()` when it encounters a similar LEA instructions is that it removes the instructions from the HEP. It then proceeds to add a LEA instruction with the same destination register but with the label of the data minus 1 as the source operand. After that, an ADD instruction adding 1 to the destination register of the LEA instruction is added to the HEP where the original LEA instruction was located. For example:

```
lea eax, [somedata]
```

becomes:

```
preface:
lea eax, [somedata-1]

hep:
add eax, 1
```

The result is that the program behaves like it did prior to the change but the HEP will be easier to hide while the LEA instruction and the data which address is loaded is somewhat hidden.

The use of a label yields a problem that also affects other parts of the tool. Since the input file is derived from a file written in assembly language, the labels are translated into memory addresses when the file is assembled and because the assembler file is assumed to be assembled without any headers to make the program runable, the address that the label is translated into will not be correct when the output from the tool is run. Because of this, all use of labels has to be done manually after the output has been generated before the obfuscated program can be run.

4.6 Generating a MEP

The first step when generating a MEP from the HEP is to insert one or more bytes before the HEP (from here on called starting bytes) so that a new execution path with overlapping instructions is created. Since the x86 architecture can handle instructions that are up to 15 bytes long, the maximum number of bytes to insert before the HEP is 14. In the tool however, the maximum number of bytes inserted is three. This is mainly because the search space would be too great and the tool would take a very long time to run.

The choice of starting bytes depends on the instructions in the HEP, therefore all combinations of starting bytes (with some exceptions) are inserted before the HEP and the result is evaluated. Because the x86 provides some instructions with a length in the range of one to three bytes, these starting bytes are discarded since they only generate one instruction and no overlapping occurs.

4.6.1 Input starting bytes

Since the maximum number of bytes inserted before the HEP is three, all the combinations of one to three bytes are considered which results in a large search space. Before making exceptions all the bytes in the ranges `[0x00 - 0xFF]`, `[0x0000 - 0xFFFF]`, `[0x000000 - 0xFFFFFF]` are generated. This brings the number of bytes generated up to $256 + 256^2 + 256^3 = 16843008$ starting bytes. By removing the unsuitable starting bytes the number is reduced to 3316228 starting bytes.

These different starting bytes are stored in a special object called `Opcode` and then placed in a `vector<pair<Opcode, OpcodeMetaData> startingBytes` using the `pair` class offered by the C++ Standard Library.

After storage, the different starting bytes are inserted before the HEP and evaluated. The different results of the evaluation is stored in the `OpcodeMetaData` object coupled with the `Opcode` object.

4.6.2 Looking for the longest MEP

The first step in the evaluation of the starting bytes is checking the number of instructions before the MEP syncrhonizes with the HEP. Since the goal is to hide as many instructions as possible, a higher number means a better HEP. The number is calculated for each of the starting bytes by comparing the last instruction in the MEP in the HEP, if they are equal, the last instruction in both execution paths is popped and the process is repeated until the instructions differ. The number of hidden instructions is calculated as follows:

```
    Sync number = HEP size - number of equal instructions
```

The number is then saved in the `OpcodeMetaData` object for the starting bytes being checked.

After the sync number has been calculated for every `Opcode` object, the highest sync number is calculated and all starting bytes with a sync number less than the highest are removed.

4.6.3 Ranking the starting bytes

After the starting bytes generating the longest MEP have been found, the evaluation of the remaining starting bytes continues. This part of the evaluation looks more at the actual instructions in the generated MEPS and their characteristics. Starting bytes with MEPs containing undesireable instructions are removed. The different kinds of undesireable instructions are described below.

Instructions using SIB bytes

The SIB byte is used for index-based memory access. Firstly, all kinds of memory access in the MEP is undesireable but in some instances they can be remedied by adjusting for memory access which will be explained later. The problem with the use of SIB bytes is that they involve more than one register when calculating the memory address being accessed by the instruction. This means that both of the registers must contain values such that the calculated address is one that the program is allowed to access. This makes further usage of said registers in other registers difficult.

Interrupts

The explanation for why interrupting instructions are undesireable in the MEP is simple, they interrup the execution of the program.

Since memory access is a tricky problem to handle it is desired to have as few instructions accessing memory as possible. As a part of the ranking of starting bytes, the number of memory accessing instructions for each of the starting bytes is calculated and saved. The `OpcodeMetaData` object contains a `char usedRegs[]` array with eight elements, one for each register. To calculate the number of memory accessing instruction, the tool inserts each of the starting bytes before the HEP again an then loops through the generated MEP. When a memory accessing instruction is found, the register with the memory address being accessed is determined and the corresponding element in the `usedRegs` array is added by one. The number is then calculated by adding all the numbers in `usedRegs`. This could have been done without using the `usedRegs` array by just using an integer to save the number of memory accessing instructions. The reason to why the registers are recorded is that the best possible outcome would be if the registers were only used once or never, then it would not be enough if only the number of memory accessing instructions was used. For example a MEP containing 4 memory accessing instructions where all use different registers is better than a MEP containing only two memory accessing instructions where the same register is used twice.

4.7 Memory access adjusment

Describe how this is done

4.8 Swapping instructions

Describe swapping instructions

4.8.1 Swapping 32-bit mov instructions

How to swap 32-bit mov and why

Chapter 5
Result

Chapter 6

Discussion and further development

Chapter 7
Conclusion

References

[AdJ06] John Aycock, Rennie deGraaf, and Michael J. Jacobson Jr. Anti-disassembly using cryptographic hash functions. *Journal in Computer Virology*, 2, 2006.

[AF14] Mike Lack Adam Ferrari, Alan Batson. x86 assembly guide. `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`, 2014.

[Art13] Nitay Artenstein. x86 instruction encoding revealed: Bit twiddling for fun and profit. `http://www.codeproject.com/Articles/662301/x86-Instruction-Encoding-Revealed-Bit-Twiddling-fo`, 2013.

[Bro99] Mats Brorsson. *Datorsystem Program- och Maskinvara.* Studentlitteratur, 1999.

[CJ13] Martin Hell Christopher Jämthagen, Patrik Lantz. A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries. 2013.

[Hyd96] Randall Hyde. *The Art of Assembly 16-bit DOS version.* 1996.

[Irv10] Kip R. Irvine. *Assembly Language for x86 Processors 6th edition.* Pearson Education Inc., 2010.

[JA09] Daniel Medeiros Nunes de Castro John Aycock, Juan Manuel Gutiérrez Cárdenas. Code obfuscation using pseudo-random number generators. *Computational Science and Engineering*, 3, 2009.

[lib14] Libemu. `http://libemu.carnivore.it/`, 2014.

[LSPM12] Charles LeDoux, Michael Sharkey, Brandon Primeaux, and Craig Miles. Instruction embedding for improved obfuscation. In *ACM Southeast Regional Conference*, 2012.

[obj14] Gnu disassembler objdump. `https://sourceware.org/binutils/docs/binutils/objdump.html`, 2014.

[udi14] Udis86. `http://udis86.sourceforge.net/`, 2014.