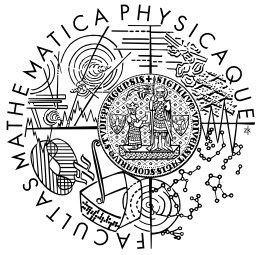


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Adolf Středa

Analysis of Virtual Machine based obfuscation

Department of Algebra

Supervisor of the master thesis:	Mgr. Milan Boháček
Study programme:	Mathematics
Study branch:	Mathematics for Information Technologies

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date July 18, 2018 Adolf Středa

Title: Analysis of Virtual Machine based obfuscation

Author: Adolf Středa

Department: Department of Algebra

Supervisor: Mgr. Milan Boháček, Department of Algebra

Abstract: Software systems may contain sensitive data that should be protected. In a scenario, where an analyst has full access to the system, it may be desirable to transform the program to become harder to understand and reverse-engineer, while preserving the original functionality of the program.

Machine code obfuscation tackles this problem by adding complexity to the program's control flow, a programming idiom removal, and various abstractions. Specifically, WProtect is an obfuscation engine that utilises a stack virtual machine and its own instruction set to achieve these properties.

In this thesis, I will analyse WProtect obfuscation engine, its obfuscation algorithms and present a generic approach to an extraction of a code protected by WProtect. Furthermore, I will design a generic framework for a static code extraction that is tweakable in order to support different WProtect configurations.

Several improvements to WProtect, both in terms of configuration and design, will also be proposed. These proposals mostly intend to mitigate vulnerabilities that are exploited in the code extraction, however, several proposals shall also include improvements specifically targeting static analysis prevention.

Keywords: WProtect, obfuscation, deobfuscation, virtual machine

Contents

1 Preliminaries **4**
- 1.1 Machine code . . . 4
- 1.2 The format of executable files (Windows) . . . 5
- 1.3 Complexity . . . 6
- 1.4 Code obfuscation . . . 8
- 1.5 Code analysis . . . 13
- 1.6 YARA . . . 15

2 Virtual machine **17**
- 2.1 WProtect Virtual Machine . . . 18
  - 2.1.1 Instructions . . . 18
  - 2.1.2 Code conversion . . . 20

3 Extraction of a code protected by WProtect **22**
- 3.1 Virtual machine extraction . . . 22
- 3.2 Instruction sequence recovery . . . 23

4 Mitigating WProtect's vulnerabilities **25**
- 4.1 Default configuration file . . . 25
- 4.2 Rolling key updates . . . 26
- 4.3 Instruction selector and WProtect initialization . . . 26
- 4.4 x86/x86_64 to WProtect translation . . . 27

5 Conclusion **29**
- 5.1 Future work . . . 29

Bibliography **30**

A Attachments **32**
- A.1 Source code . . . 32
- A.2 Instruction Table . . . 32

Introduction

Motivation

The distribution of software entails some risks, especially if the software utilises properties or algorithms that are to be secret. As these secrets could be recovered through a technique called reverse engineering, some parties resorted to a code obfuscation in order to deter such attempts. In general a code obfuscation can be employed to deter tampering or reverse engineering. This is especially desirable if the software itself is accessible by an analyst because, up to some partial exceptions, e.g. Intel SGX Enclave[1], it is possible to reverse engineer the system.

In legitimate cases, an obfuscation is mostly used to protect an intellectual property, be it an algorithm, configuration or redistribution protection. An example of a legitimate use is Denuvo[Den18] DRM and anti-tamper scheme that is incorporated into numerous games such as Far Cry 5, Assassin's Creed Origins.

Illegitimate uses often involve malware and while the motivations are similar they carry some notable differences. The first one is that malware often tries to hinder a code signature generation, since signatures may be subsequently used to detect new samples. Another reason lies in the malware's attempt to stay under the radar, i.e. it has to hide its purpose or even poise as a legitimate code.

The standard obfuscation approach lies in either making the code more complex or reducing the code structure. In extreme cases it can end up with a program as one big code block, for example, Movfuscator[2] utilises MOV instruction to completely flatten any code structure. However [Dom15a] also suggests that psychological means could be used – as he demonstrated in his project REpsych [Dom15b].

Contributions

This thesis aims to make the following contributions:

- We provide a theoretical background on the topic of the machine code obfuscation, mostly focusing on methods that are incorporated in the WProtect obfuscation engine.
- We provide a thorough description of obfuscation methods utilised by the WProtect obfuscation engine[Xia]. An analysis of WProtect's parts that are directly responsible for the obfuscation process is also provided.
- We propose a generic way of a control flow reconstruction for a machine code protected by WProtect.
- We release, to our knowledge, the first generic framework for the extraction of a code protected by WProtect. The WProtect component detection is based on YARA rules that are *de facto* standard in the field of malware analysis. This should make the framework flexible enough to support

[1] An Intel CPU feature that allows execution of a code in an encrypted RAM region with a limited hardware access. Intel claims it should protect the code itself from any analysis.

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

similar obfuscators or WProtect modifications through simple YARA rule modifications.

- Due to the exploitability of the WProtect obfuscation engine we offer several proposals that mitigate recognized vulnerabilities – both ones exploited in the control flow recovery and a few theoretical ones that were not utilised. These proposals range from simple configuration file modifications to virtual machine design changes.

The following text consists of several parts. Chapter 1 introduces the theoretical background required for the analysis of virtual machine based obfuscation. This background is further extended to incorporate methods used by the WProtect obfuscation engine. Chapter 2 introduces WProtect itself – both design and transformations procedures. This is further extended in chapter 3, where an algorithm for control flow extraction is proposed. Vulnerability mitigations are explored in chapter 4. Conclusions and ideas for future work on the topic are provided in chapter 5.

1. Preliminaries

The following section covers the theoretical background of this work. Basic definitions required for code analysis are introduced along with a few formal definitions and claims covering the topic of a code obfuscation. Since the complexity theory will be also used throughout some chapters, this chapter will also contain a short introduction to the complexity theory.

1.1 Machine code

Definition 1 (Machine code)**.** *Machine code is a set of instructions that are executed directly on the device's central processing unit (CPU).*

This definition can be extended to a notion of a *code*, that is a set of instructions forming a computer program which is executed by a computer. Many notions and claims presented in this work still hold, nevertheless, we shall focus specifically on machine code. Also, while there are many CPU architectures and instruction sets, this work targets only *x86* (32-bit) and *x86_64* (64-bit) instruction sets.

Every instruction has its code and a mnemonic, an abbreviation for instruction's operation, e.g. *ADD* represents the instruction that performs addition. Instructions are mostly followed by operands. Every operand has its size, almost always[1] aligned to powers of 2. Standard operand sizes are byte (8 bits), word (16 bits), double-word (32 bits), and quad-word (64 bits). Double-word and quad-word is sometimes shortened to *dword* and *qword*.

Currently three basic types of operands are available. The first type is a *memory address* that refers to a position in the memory. As every memory page[2] has access privileges (R – read, W – write, E – execute), these references can cause the instruction to fail due to insufficient privileges.

The second type is called an *immediate value*. Immediate values are hard-coded sequences of bytes that are directly interpreted as numbers (mostly integers).

The last type is a *register*. General purpose registers have their usage set mostly by a convention (e.g. EAX/RAX is traditionally used as an accumulator, ECX/RCX as a counter). On the other hand, there are some registers that have their purpose unchangeably set, e.g. EIP/RIP is always an instruction pointer. There is one register that is modified almost exclusively[3] as a side effect, it is so-called EFLAGS register that contains flags. These flags are sometimes modified by other instructions as their side-effect, e.g. there is *ZF* (zero flag) that is set to 1 if the result of an operation is zero (e.g. SUB EAX, EAX – subtraction of

[1] While other operand alignments are very uncommon, due to the nature of modern processors, they exist. A relatively new example is *cLEMENCy* architecture[D+17] that was developed for DEF CON 25 CTF. This architecture has bytes with 9 bits and thus operands use multiples of 9. Also many older computer designs used different word sizes, for instance, Soviet computer Setun[BMZ01] which used trits (ternary digits) and tryte (ternary "byte") that was composed of 6 trits.

[2] Page is the smallest amount of memory that can be allocated, its size is mostly 4 kB.

[3] Although, there are instructions that modify this register, they are rarely used.

Table 1.1: Examples of instructions with operands.

instruction	description
ADD EAX, EBX	Add EBX to EAX, the result is in EAX.
MOV EAX, EBX	Move EBX to EAX
SHLD EAX, EBX, CL	Shift EAX‖EBX by CL positions

EAX from EAX), SF that does the same if the result of an operation is negative. EFLAGS register is used in conditional jumps (e.g. JZ – jump if ZF=1) or special algebraic operations (e.g. ADC – *add with carry* that is used for addition of numbers that cannot fit into one register). Operands have also an ordering convention: first operand describes a destination, second operand a source and third being a parameter; some examples are listed in Table 1.1. Moreover, some registers offer an access to lower bytes through a different register name, e.g. we can access a lowest byte, word, double-word of RAX by names AL, AX, and EAX respectively. Since 64-bit registers are not available on the 32-bit architecture, double-word "alias" [4] accesses the whole register on the 32-bit architecture.

In contrast to machine code, we shall denote a code intended to be translated on the fly into machine code as *byte-code*.

1.2 The format of executable files (Windows)

Executable files for Microsoft Windows operating systems are called Microsoft PE (Portable Executable), further we shall call them PE files or, more generally, binary files. The format's specification is available at [PE].

There is a header at the beginning of a PE. The header contains MS-DOS executable (mostly a small stub that outputs *"This program cannot be run in DOS mode"*) with a signature *MZ\x00\x00*, PE file signature *PE\x00\x00*, COFF File Header (structure identifying the target platform and PE file attributes), optional headers, and a section table. These structures are located at the fixed offset.

Sections are the basic unit of code or data in PE files. Every section has to have a name and offset defined in the section table in the header. The section's name consists of at most 8 arbitrary bytes, if the name's size is 7 bytes or smaller it has to be followed by NULL terminator and padded to 8 bytes, if necessary. For longer names, this field contains a slash that is followed by an ASCII representation of a decimal number that is an offset into the string table. There is a convention that sets names for the most common cases (c.f. Table 1.2), nevertheless, arbitrary names can be used.

Every PE file (or even DLL library) assumes that it will be loaded to a specific memory address. The default addresses[Pri94] are 0x10000 (Windows NT), 0x400000 (DLL), 0x400000 (Windows 32-bit PE files), 0x140000000 (Windows 64-bit PE files); if the PE files has to be loaded to different address (e.g. due to

[4] And the same would hold for 16-bit architecture that would address the whole register by its 16-bit "alias". This ensures a backward compatibility, i.e. x86_64 CPU can easily interpret 16-bit, 32-bit or 64-bit machine code. Actually, the instruction set is made to be backward compatible to original Intel 8086[Ric15].

Table 1.2: Reserved section names (common examples)

section name	purpose
.data	initialized data
.edata	export table
.idata	export table
.pdata	exception information
.reloc	image relocations
.rsrc	resource directory
.text	executable code

Address Space Layout Randomization – ASLR), then these relocation are usually described in *.reloc* section.

1.3 Complexity

The basic building block of the complexity theory is a computational model. The most famous one was proposed by Alan Turing[Tur37]. Today his model is called Turing machine.

The intuitive description is based on an *infinite tape*, divided into discrete *cells*, and a *reading head*. In the beginning, almost all cells of the tape are empty. Only a finite continuous sequence of cells may be non-empty, this sequence is called *input*. The head starts at the beginning of the input.

The machine's behaviour is prescribed by its instructions. It executes these instructions until it reaches a final state (note that this may not happen). The choice of the instruction in every its step is determined by its state and the value that is read by the reading head.

Definition 2 (Turing machine)**.** *Let* Σ, Q *be non-empty finite sets, let* δ *be a function*

$$\delta : \Sigma \times Q \to \Sigma \times Q \times \to, \leftarrow, \_.$$

Moreover, $Q \supseteq \{q_s,\ q_a,\ q_r\}$*, where* q_s *is a starting state,* q_a *an accepting state and* q_r *is a rejecting state.*

Then Turing machine is a triplet $(\Sigma,\ Q,\ \delta)$*,* Σ *is called an alphabet,* Q *is a set of the machine's states and* δ *is its transition function.*

This notion can be expanded to so-called *non-deterministic Turing machine* by a simple extension of the transition function:

$$\delta \subseteq (\Sigma \times Q) \times (\Sigma \times Q \times \to, \leftarrow, \_).$$

This extension means that the transition function becomes a multivalued function, that is $(x,\ q) \in \Sigma \times Q$ may have more possible outcomes after the application of the transition function. $M(x) = q$ denotes a result, a state q, of an execution of Turing machine M with x as input.

Furthermore, we can define a set Σ^* as a set of all finite strings over Σ, leading us to a notion of language:

Definition 3. *Let M be Turing Machine. Language is an arbitrary subset $S \subseteq \Sigma^*$. There are two notions connected to Turing Machines and languages:*

M decides language $S \subseteq \Sigma^*$ if and only if

$$\forall x \in \Sigma^* : (M(x) = q_a \Leftrightarrow x \in S) \wedge (M(x) = q_r \Leftrightarrow x \notin S)$$

M accepts language $S \subseteq \Sigma^*$ if and only if

$$\forall x \in S : M(x) = q_a$$

Definition 4 (Polynomial time Karp-reducibility)**.** *Let $S,\ S' \in \Sigma^*$. Then language S is polynomial time Karp-reducible to S' if and only if*

$$\exists \text{ polynomial time computable } f : \Sigma^* \to \Sigma^* \forall x \in \Sigma^* : f(x) \in S' \Leftrightarrow x \in S.$$

We shall denote this relation by $S \leq_p S'$.

Definition 5 (NP)**.** *A decision problem $\mathcal{C}$ $\in$NP if it can be solved by a non-deterministic Turing machine in polynomial time.*

Definition 6 (NP-completeness)**.** *A decision problem $\mathcal{C} \in$ NP-complete if and only if $\mathcal{C} \in$ NP and every problem in NP is polynomial time Karp-reducible to it.*

One of the more interesting problems, both practically and theoretically, is called SAT problem (boolean satisfiability problem). It is a problem of determining whether there exists an interpretation that satisfies a given Boolean formula in CNF[5] – i.e. whether we can set literals to TRUE/FALSE so that the logical value of the whole formula is TRUE.

It was the first problem to be proven to be NP-complete. For brevity reasons, as the proof itself is rather technical and tedious, only the main idea of the proof shall be introduced.

Theorem 1 (Cook-Levin)**.** *SAT problem is NP-complete.*

Proof idea: It is easy to see that SAT problem is in NP. Once we have a solution then calculating the formula's logical value can be done in polynomial time, e.g. recursively according to the definitions of the Boolean operators in use.

Suppose we have $S \in$ NP, then there exists a Turing machine M running in polynomial time $p(n)$ such that $\forall x \exists w \in \Sigma^{p(|x|)} : x \in S \Leftrightarrow M(x,\ w) = 1$.

In every step of the Turing machine the data on the tape is limited by $|x| + p(|x|) + constant$ in size. Similarly the amount of steps required by M to accept input can be bounded by the same number. The conditions for these transitions to occur can be expressed as Boolean formulas with input as its literals. This formula (φ_x) can be expressed in CNF in polynomial time and thus following implications hold:

$$x \in S \Rightarrow \varphi_x \in \text{SAT}$$

$$x \notin S \Rightarrow \varphi_x \notin \text{SAT},\ \varphi_x \text{ cannot be satisfied}$$

□

[5]Conjunctive normal form – the formula is expressed as a conjunction of clauses, which are disjunctions of literals.

Unfortunately, these notions will not suffice. We shall require an object that can translate these theoretical results into a real-world implementations. This leads us to so-called Turing completeness:

Definition 7 (Turing completeness)**.** *A computational system that can compute every Turing-computable function is called Turing-complete.*

If the notion of a universal Turing machine is introduced then the condition for Turing-completeness could be simplified to being able to simulate a universal Turing machine.

In our case, the computational system is either an instruction set (x86/x86_64) or a programming language. In reality, many systems are, either intentionally or not, Turing-complete. For example conventional programming languages like C, Java, C#, Common Lisp, Python etc. are all Turing-complete. Similarly, R. Kaye proved that even an infinite extension of a game called Minesweeper is Turing-complete[Kay00]. x86/x86_64 instruction set is also theoretically Turing-complete; actually, S. Dolan[Dol13] proved that only *MOV* instruction suffices for its Turing-completeness.

Yet in reality, we are limited by available physical memory. In some cases, it might be more useful to use objects that reflect this fact. Unfortunately, setting a limit to a Turing machine tape size does allow arbitrarily long inputs, and thus a new object has to be introduced:

Definition 8 (Deterministic finite automata (DFA))**.** *Let* Σ, Q *be non-empty finite sets, let* δ *be a function*

$$\delta : \Sigma \times Q \to Q.$$

Moreover, $Q \supseteq q_s, F$*, where* q_s *is a starting state and* F *is a set of all accepting states.*

Then deterministic finite automaton is a quintuple $(\Sigma,\ Q,\ \delta, q_s, F)$*,* Σ *is called an alphabet,* Q *is a of the automaton's states and* δ *is called a transition function.*

While theoretically, as programming languages are mostly Turing-complete, Turing machines are a more general description, deterministic finite automata may provide a less abstracted description of a problem.

1.4 Code obfuscation

Code obfuscation is a collection of methods that can be used to protect software against reverse engineering[6][Adr13].

Let us call the inverse process, that is a process of a transformation of a source or machine code into a form that is easier for humans to understand, a deobfuscation.

Unfortunately, it is immediately obvious that both obfuscation and deobfuscation are partially subjective terms and are not equal to code structure simpli-

[6]A process that transforms the output of another process into an input of said process in part or completely.

fication; a counter-example to such equivalence is a code optimization done by a compiler or syntactic sugar[7].

In the following chapters, we will use these notions to analyse an obfuscation engine called WProtect. Fortunately, its construction will allow us to stay clear of these border-cases.

However, to limit acceptable range of modifications we shall utilise a definition proposed by Collberg et al.[CTL97]:

Definition 9 (Obfuscating transformation)**.** *Let $\tau : \mathcal{P} \mapsto \mathcal{P}$ be a transformation of a source program $\mathcal{P}$ into a target program $\mathcal{P}'$. Then τ is obfuscating transformation if $\mathcal{P}$ and $\mathcal{P}'$ have the same observable behaviour, i.e. following conditions must hold:*

- *If $\mathcal{P}$ fails to terminate or terminates with an error condition, then $\mathcal{P}'$ may or may not terminate*
- *Otherwise, $\mathcal{P}'$ must terminate and produce the same output as $\mathcal{P}$.*

The process of applying an obfuscating transformation is called obfuscation.

Observable behaviour is loosely defined as a *behaviour experienced by the user.* That means $\mathcal{P}'$ may have side-effects such as file creation, sending messages as long as these side effects are not experienced by the user. Moreover, we do not require $\mathcal{P}'$ and $\mathcal{P}$ to be equally effective.

Collberg et al.[CTL97] also present a basic catalogue of such transformations. We shall focus on so-called *control transformations*, i.e. transformations that affect the control flow.

Opaque predicates

Definition 10 (Boolean formula)**.** *Let us define an algebra $\mathcal{A} = (\{0, 1\}, \wedge, \vee, \neg)$. Let us denote $\wedge$ be the disjunction operator, $\vee$ the conjunction operator, and $\neg$ the negation operator. These operators are defined by the following truth tables:*

x	y	$x \wedge y$	$x \vee y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

x	$\neg x$
0	1
1	0

Then a Boolean formula is a polynomial over $\mathcal{A}$ in $n \in \mathbb{N}$ variables. A Boolean formula f in $n \in \mathbb{N}$ variables $(x_0, \ldots, x_{n-1})$ is called satisfiable if there exists an assignment, i.e. a homomorphism τ s.t. $\forall i \in \mathbb{Z}_n : \tau(x_i) \in \{0, 1\}$, for which $\tau \circ f = 1$ holds.

We shall need one more operator, which can be defined through the aforementioned ones. This operator is called XOR (exclusive OR) and it corresponds to the addition modulo 2.

[7]Syntax that is designed to make the code easier to read or to express. For example a ternary operator *"a ? b : c"* is shorter than its equivalent to *"if (a) {b} else {c}"*, nevertheless its usage can severely hinder the code readability, especially when it is used in nested expressions.

Definition 11 (Opaque predicate)**.** *Let X be a Boolean formula (predicate). Then we say that X is called opaque if its outcome is known at the obfuscation time.*

We will need to transpose these notions into a form that is more tangible in terms of programming languages. In the following example, we shall assume that we are working with numbers of a limited bit-length (often a power of 2 – e.g. 32 bit). Since these bits may also encode a sign, we need to set up a convention, for exemplary reasons we shall utilise a convention used by C++ programming language:

Suppose we have a sequence of bits

$$x_{n-1}x_{n-2}\dots x_0,\ x_i \in \{0,\ 1\} \text{ for } i \in \{0,\ 2, \dots,\ n-1\}.$$

If $x_n = 0$, then $x_{n-1}x_{n-2}\dots x_0$ represents a number corresponding to a number N whose binary representation is $x_{n-2}x_{n-3}\dots x_0$. Otherwise, it represents a number whose value is $N - 2^{n-1}$. We shall define several operators on these representations.

We can extend $\wedge$ to "&" (bitwise AND), i.e. conjunction operator applied on corresponding bits of representatives. Addition "+" is defined as in $\mathbb{Z}_{2^n}$, treating $x_{n-1}x_{n-2}\dots x_0$ as a binary representation of an element in this field. The simplest operator is "$-$" (negation) which only negates the bit x_n that corresponds to the number's sign. A binary equivalence operator "==" outputs 1 if and only if both sides correspond to the same sequence of bits, otherwise it outputs 0. As n-bit number can be represented by n bits, we are able to translate these expressions as Boolean formulas.

Example. [xor14] An expression

$$a\&b == b\&((b\&((b\&a) - b)) - 1),$$

where a, b are n-bit numbers, is an example of an opaque predicate that always evaluates to 1 (*True*).

Proof. We shall prove that this holds by induction.

Suppose a, b are only 1-bit numbers, then bitwise-and is equivalent to multiplication over $\mathbb{Z}_2$ and subtraction is equivalent to addition over $\mathbb{Z}_2$. This observation transforms the expression's right side into

$$b * (b * ((b * a) - b) - 1) = b * a\ + b + b = b * a.$$

Now we shall perform the inductive step. Suppose we have $n+1$-bit numbers and the claim holds for the first n bits. Bitwise-and corresponds to bitwise-multiplication, subtraction is more complex:

$$a = a_n a_{n-1}\dots a_1 a_0,\ b = b_n b_{n-1}\dots b_1 b_0;$$

$$(a\&b)_i = a_i * b_i;$$

$$(a-b)_i = a_i \oplus b_i \oplus c_{i-1},\ c_{i-1} \text{ is called } \textit{borrow};$$

$$c_i = b_i \oplus c_{i-1} \oplus a_i b_i \oplus a_i c_{i-1} \oplus b_i c_{i-1},\ i \in \{0, \dots,\ n-1\},\ c_{-1} = 0.$$

The aforementioned expression thus unpacks to:

$$b_i * (1 \oplus a_i \oplus c_{i-1} \oplus 1_i \oplus c'_{i-1}).$$

We require two borrows – each subtraction generates one. However, we can use induction hypothesis to get a relation

$$b_i * (1 \oplus a_i \oplus c_{i-1} \oplus 1_i \oplus c'_{i-1}) = a_i * b_i,\ i < n.$$

That means either $b_i = 0$ or $c_{i-1} \oplus c'_{i-1} = 1 \oplus 1_i$. If $b_n = 0$, then the claim obviously holds (both sides vanish), so let us suppose $b_n = 1$. Let $n > i > 0$, then the induction hypothesis reduces to:

$$1 \oplus c_{i-1} \oplus c'_{i-1} = 0 \Rightarrow c_{i-1} * c'_{i-1} = 0.$$

These borrows can also be expressed in the following way:

$$c_i = a_i b_i \oplus b_i \oplus c_{i-1}(b_i \oplus a_i b_i \oplus 1),$$

$$c'_i = (a_i b_i \oplus b_i \oplus c_{i-1} b_i)1_i \oplus 1_i c'_{i-1} \oplus c'_{i-1}(a_i b_i \oplus b_i \oplus c_{i-1} b_i) \oplus c'_{i-1} \oplus 1_i.$$

Without loss of generality, we may assume that $b_{n-1} \neq 0$, otherwise

$$c_{n-1} \oplus c'_{n-1} = c_{n-1} \oplus c'_{n-1}$$

and we iterate until either $b_i \neq 0$ or $b = 0$ (then this claim holds trivially). At first, suppose $n = 1$:

$$c_0 = a_0 b_0 \oplus b_0 \oplus c_{-1}(b_0 \oplus a_0 b_0 \oplus 1) = a_0 b_0 \oplus b_0,$$

$$c'_0 = (a_0 b_0 \oplus b_0 \oplus c_{-1} b_0) \oplus c'_{-1} \oplus c'_{-1}(a_0 b_0 \oplus b_0 \oplus c_{-1} b_0) + c'_{-1} + 1 = a_0 b_0 \oplus b_0 + 1.$$

$$c_0 + c'_0 = a_0 b_0 \oplus b_0 + a_0 b_0 \oplus b_0 + 1 = 1$$

Now we shall examine $n > 1$:

$$c_{n-1} = a_{n-1} b_{n-1} \oplus b_{n-1} \oplus c_{n-2}(b_{n-1} \oplus a_{n-1} b_{n-1} \oplus 1),$$

$$c'_{n-1} = c'_{n-2}(a_{n-1} b_{n-1} \oplus b_{n-1} \oplus c_{n-2} b_{n-1}) \oplus c'_{n-2},$$

$$c_{n-1} \oplus c'_{n-1} = (c_{n-2} \oplus c'_{n-2})(b_{n-1} \oplus a_{n-1} b_{n-1} \oplus 1) \oplus c_{n-2} c'_{n-2} b_{n-1} \oplus a_{n-1} b_{n-1} \oplus b_{n-1}$$

$$c_{n-1} \oplus c'_{n-1} = (b_{n-1} \oplus a_{n-1} b_{n-1} \oplus 1) \oplus a_{n-1} b_{n-1} + b_{n-1} = 1.$$

In both cases the following holds:

$$1 \oplus c_{n-1} \oplus c'_{n-1} = 0$$

and thus

$$b_n * (1 \oplus a_n \oplus c_{n-1} \oplus 1_n \oplus c'_{n-1}) = b_n * (1 \oplus a_n \oplus 1) = a_n b_n,\ n > 0.$$

□

As is shown by this example, it is rather easy to create predicates that are very hard to eliminate through automated processes without resorting to brute-forcing or probabilistic heuristics.

Informally we can interpret this definition in the following way: we have some information (a property of a predicate) available at the obfuscation time that is hard to deduce after the obfuscation. SAT problem is NP-complete (Theorem 1) and it's extension to so-called satisfiability modulo theories[8] (SMT) which even contains undecidable problems. Theoretically opaque predicates can significantly increase the complexity of the obfuscated program's analysis as deciding whether both branches of the code are valid generally takes exponential time. Similarly, there are constructions, exploiting such branching, that cause an exponential slowdown to any potential analysis.

Fortunately, many instances of SAT/SMT problem are relatively easy to solve and thus SAT-solvers or SMT-solvers such as Z3[Res] are sometimes successfully employed[Zub16] to decide such predicates.

Dead code insertion

Definition 12 (Dead code)**.** *A dead code is a subset of a source code or a machine code of a program that satisfies one of the following conditions:*

- *The code in this subset is never executed.*
- *If the code in the subset is executed then the program's state before and after its execution is the same or the changes in the state are never used in any other part of the program.*

While both cases might be easily detected either through control flow analysis or by tracking the program's state, this technique gets powerful once combined with other methods. For instance, we could insert a section of a code that will never be executed and mask the fact by an opaque predicate or mix the dead code with a regular code through the code reordering technique.

Hiding library calls and programming idioms

Library calls can provide useful clues to a reverse engineer and thus it might be worthwhile to hide these clues, e.g. we can load libraries at the runtime through Windows Kernel function *LoadLibrary*, similarly we can get the library's exports at runtime through function *GetProcAddress*. This method is quite common in malware[9] obfuscation, for example, Dridex family loads most libraries and their exported functions at runtime.

Similarly, there are many common programming *idioms* (patterns) that occur frequently in many applications. An experienced reverse engineer can utilise these patterns to heuristically determine the purpose of the whole code sections without resolving to a thorough analysis. An example of such an idiom can found

[8] SAT problem generalized by adding functions and predicate symbols that have additional interpretations, e.g. theory of integers.

[9] A shorthand for malicious software

in the common implementation of a CRC32 checksum algorithm. If a reverse engineer spots a table that starts with values

$$0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, \ldots$$

he may guess that this is a CRC32 table and the code that refers to such table computes a CRC32 checksum. Thus it might be reasonable to mask such idioms so a potential reverse engineer will have to analyse the section more thoroughly.

Table interpetation

One of the most effective and expensive transformations if called *table interpretation*. The idea is to convert a part of a code into a virtual machine code – *byte-code*. At runtime, this part is interpreted by a bundled interpreter. Naturally, the code may contain multiple parts protected in this way, every part may even utilise a different virtual machine. Moreover, this approach may be iterated.

Interestingly the underlying idea to transform machine code into an alternative byte-code is rather old and originally was used for a completely benign purpose: to avoid compiled program's platform dependency. A common example is the JVM (Java Virtual Machine) or Pascal's *p-code*, where the main motivation was cross-platform portability.

As WProtect is based mostly on this transformation, the entire chapter 2 is dedicated to this transformation.

Code reordering

Source code, and to some extent even machine code, exhibits a property called *locality*. In the case of source code, the idea is that its easier to read and understand items that are logically related if they are also physically close in the source code. This property is also extensively preserved during its compilation into machine code and it provides useful clues to a reverse engineer. While reordering itself does not have to add much obscurity to the program, it is very resilient (it is necessary to resolve possible dependencies) and thus hard to revert. Furthermore, there is a great potential if combined with other methods, e.g. in conjunction with a dead code insertion, it forces the deobfuscator to utilise more powerful methods than simple state tracking as now the dead code is interleaved with a working code.

1.5 Code analysis

Following definitions assume that a program to be analysed is an *imperative program*, i.e. a program that uses statements to change its state. In contrast, there is another paradigm called declarative that instead expresses the logic of a computation (e.g. regular expressions) instead of program's state transitions.

Definition 13 (Control flow)**.** *Control flow of a program is the order in which individual units of the program are executed.*

The notion of a *unit* can be specified according to the required analysis resolution, that is for example instructions, statements, or functions. Similarly, we can track changes in data to introduce a similar data-oriented notion *data flow*.

Static code analysis

Static code analysis studies only the code itself without executing it. Its advantages are intuitively clear – the code is not executed, i.e. there is no risk of malicious side effects caused by the execution of the code, moreover, we can always perform static code analysis as the only prerequisite is an ability to read the code.

More specifically static code analysis is based on structural information of the code. Control flow, data flow, and code dependencies are utilised to identify various features. Mostly these features are used to build a model of the program's states and then determine possible routes at each step. As full state analysis may be computationally infeasible, a trade-off between precision and granularity has to be considered.

Currently, there are two basic approaches that are used. The first one is called *disassembling*. Disassembling transforms a binary code into its mnemonic representation, which makes the code more understandable to humans. Unfortunately, the distinction between code and data is not clear, even for finite machines a generic polynomial-time algorithm is not available (c.f. Theorem 2). Thus the disassembler has to use heuristics to approximate the solution. x86 or x86_64 architecture does not help us to alleviate this problem as code and data can be freely interleaved.

To fix the definition of code we shall utilise a notion proposed by Wartell et al.[WZH+11]: bytes are *code* if they are executed during the runtime. Otherwise, we shall denote those bytes as *data*.

This notion may seem strange for high-level languages, where even a part intentionally written as a code may be classified as data. Nevertheless, on the low-level, this notion is more natural as it separates the, possibly wrong, intention from a real implementation. This is very useful when considering disassembly because generally the only information available is the compiled binary[10].

Theorem 2. *The problem of deciding whether bytes are code on a finite machine is not generally decidable in polynomial time.*

Proof. Suppose we have a boolean function $\mathcal{F} : \mathbb{F}_2^n \to \mathbb{F}_2$ representing a boolean formula. Then consider a program described by a following pseudo-code:

```
1  input: x ∈ {0, 1}^n
2  output: a ∈ {0, 1}
3
4  if F(x) equals 1:
5          return 1
6  else
7          return 0
```

Algorithm 1.1: SAT simulator

Suppose we have a polynomial-time machine that decides whether the fifth line is a code, i.e. there exists an input x such that the *if* statement is evaluated

[10] Again, this may be ambiguous in case of languages compiled to an intermediate byte-code (Java, C#) where the original "wrong" intention may be preserved. This is a part of the reason of a narrow focus on machine code which removes these numerous border cases.

as *True*. However, the existence of such an input equals to the existence of an interpretation of the boolean formula represented by $\mathcal{F}$. This means that our machine can solve a SAT problem in a polynomial time, which is a contradiction, SAT problem is in the *NP* complexity class. □

Dynamic code analysis

A basic notion used in the dynamic code analysis is *trace*. *Trace* is a path in the program's state space. Dynamic code analysis utilises trace to locate and follow features by following the running code. A code can be executed either on a processor or on an emulator.

Both approaches have their advantages and disadvantages, mostly in terms of implementation complexity and security. While it is straightforward to execute a given code on a real processor, it does not protect, or is hard to do so, the rest of the machine from the execution side-effects. These side-effects can be partially mitigated by the use of a virtual machine, nevertheless, Spectre[KGG+18] and Meltdown[LSG+18] vulnerabilities showed that even a virtual environment separation does not protect from all side-effects. On the other hand, we could try to emulate the code. While it gives us a total control over the side-effects, we need to more or less define all the behaviour beforehand, e.g. all the used system functions have to have a pre-set behaviour, otherwise, the emulation may fail.

The output of the dynamic code analysis is highly dependent on the input and the program's environment, which means that dynamic analysis may output different results for the same inputs.

1.6 YARA

YARA[11] is a tool originally developed by Victor M. Alvarez of VirusTotal with the aim to help malware researchers identify and classify malware samples.

```
1   rule PE_with_default_stub
2   {
3       strings:
4       $hex_0 = { 4D 5A 00 00 } // MS-DOS
5       $str_0 = "This program cannot " // stub
6       $str_1 = "be run in DOS mode"
7       $hex_1 = { 50 45 00 00 } //PE
8
9       condition:
10      $hex_0 at 0 and
11      $hex_1 at 60 and
12      $str_0 and
13      $str_1
14  }
```

Algorithm 1.2: YARA rule example

[11] https://github.com/VirusTotal/yara

YARA can be used to describe patterns of an arbitrary file. These descriptions are called *rules*. These rules can contain either static descriptions or can be extended through modules to include dynamic properties (such as mutex names used by a PE file). More specifically, they consist of a set of strings and boolean expressions that determine their logic. There is an example of a YARA rule that naively (it does not enforce a position of the stub) detects PE files with a default MS-DOS stub in Code 1.2.

YARA rules will be used to describe some parts of the analyzed virtual machine as they are *de facto* standard in malware analysis. However, only a small subset of their functionality will be used; the primary aim is a static analysis, only a regular expression based pattern matching will be of our interest.

2. Virtual machine

Virtual machines, or more generally a table interpretation based obfuscations, are common means of a code obfuscation. Although they require more complex design than e.g. dead code insertion they provide a way to hinder static code analysis and they even complicate dynamic code analysis as byte-code is interpreted on the fly (as opposed to packers – in their case the payload can be found in the memory).

The standard approach is to translate e.g. x86 or x86_64 instructions into virtual machine byte-code. If we want to execute such byte-code, a corresponding virtual machine interpreter has to be used. There are currently three approaches that are used in the interpreter construction.

The first one is called *switch dispatch*. It is the simplest and the most widely used method. The main loop of the interpreter consists of two parts: an instruction loader and a large *switch* statement. The instruction loader loads an instruction code which is then passed to the *switch* statement. The *switch* statement contains all the virtual machine's instructions (code blocks corresponding to these instructions). While it is very easy to implement, it has several downsides – it is rather inefficient as we have to check whether the opcode is in the virtual machine's range. Furthermore, every *case* statement has to end with *break*, i.e. every instruction is followed by a jump. Another inefficiency stems from there being the only a single indirect branch for predicting instructions[GBC+05]. As [DBC+03] observes, this causes 0.95 misprediction rate on current processors with branch prediction.

An alternative to switch dispatch is called *threaded dispatch*. Threaded dispatch is based on making an explicit sequence of steps generated by a compiler to implement a *switch* statement. Moreover, the dispatch code is appended at the end of every virtual machine instruction. This eliminates the range check from *switch dispatch* and it also greatly increases the rate of successful branch prediction [SCEG08]. Unfortunately, this construction is not reversible into ANSI C as there is no facility for *goto* statements that can jump to multiple different locations. To implement *threded dispatch* language that treats labels as first-class citizens[1] is required.

Even more complicated is *direct threaded dispatch*[SCEG08]. It removes the cost of table lookup by translating byte-code to a pointer to the corresponding instruction code. This, however, requires a pre-translation process that significantly increases the size of the virtual machine's code.

A virtual machine can be also classified by its computational model. There are two basic computational models – *register-based* and *stack-based*.

Register-based machines use registers to pass data to instructions. There are special, mostly consisting of small amounts of a fast storage, accessible locations available to a central processing unit. They may be restricted to a write-only or read-only access (e.g. EIP in x86/x86_64).

Stack-based machines utilise a stack to pass data to instructions. The stack

[1] An entity is considered a first class citizen if it supports all operations generally available to other entities. This mostly includes being passed as an argument, being returned from a function, being modified, or being assigned to a variable

is defined as a data structure with two operations: *push* and *pop*. A value can be stored by pushing it onto a stack, popping a value from a stack retrieves the value from a stack. These operations happen in the LIFO (last in, first out) order. In order to carry out an instruction, its parameters have to be pushed onto the stack, similarly, the instruction pops these parameters and pushes its back the result onto the stack.

2.1 WProtect Virtual Machine

WProtect[Xia] is an obfuscation engine that was published on `github.com` at the beginning of September 2015 by a user *Xiaoweime*. The project is no longer actively developed since 2. 11. 2016, nevertheless, the obfuscation engine is still actively used in some malware samples.

WProtect utilises a virtual machine to protect a code. WProtect itself is written in the C++ programming language and the obfuscation process is configurable through files written in the Lua[2] programming language. These files specify the translation of x86/x86_64 instructions to virtual machine's instructions. Currently, the WProtect virtual machine has 56 basic (x86), 15 extended (x64) instructions and it is a stack-based virtual machine, i.e. instruction's parameters are passed through the stack and similarly the results are returned onto the stack. WProtect's byte-code and immediate values are masked by a key. Because this key is updated at the beginning of every instruction and also at the beginning of the instruction selector, we shall call it a *rolling key*.

Still some registers or memory regions are reserved. The function containing the virtual machine itself has a pointer to its stack, which is allocated before the virtual machine is launched. Moreover, the following registers are reserved: ESI (instruction pointer), EBX (key), EBP (stack), EDI (stack base). Registers EAX (accumulator), EDX (extend EAX or hold a value from the stack), and ECX (shift index) are used for general calculations; their values are always relevant only in the context of an instruction. Sub-registers (*X, *L – e.g. AX, AL) are used when the size of an operand is word and byte respectively.

The machine's opcodes are defined in the header file located at the path *WProtect/src/WProtect/Protect/VirtualMachine/PCode.hpp*. The order of instructions in the virtual machine is randomized during its insertion into an existing machine code. The basic instruction set is consisted of opcodes described in tables A.1, A.2.

2.1.1 Instructions

There are two types of instructions – executive instructions and auxiliary instructions. The former type includes instructions that directly handle data and perform operations that are defined by the original code. These instructions also include data obfuscation – there is a rolling key update at the start of every instruction and immediate value deobfuscation if the instruction reads a value from the code. These updates and deobfuscations are randomized at the obfuscation time. If such an instruction affects EFLAGS register then all flags are pushed

[2] `https://www.lua.org`

Table 2.1: Generic executive instruction structure

Rolling key update
Load data (stack/immediate) Update stack/instruction pointer (Immediate value deobfuscation) ⋮
Effective code
Push results onto stack Update stack pointer ⋮
Jump to *check_stack*

onto the stack at the end of the instruction. The latter kind does not change a rolling key or perform any user-defined operation; these instructions rather handle the virtual machine's consistency, for instance, they check for machine's stack overflow or handle transitions from the virtual machine to a code that is outside the virtual machine.

Every instruction handles its side effects, i.e. stack and instruction pointers are changed by instructions themselves. A generic instruction layout is described in the table 2.1.

As the only instruction's variable code is the code handling the rolling key update and the immediate value deobfuscation it is straightforward to find a regular expression that matches these instructions and use them to derive YARA rules describing WProtect's instructions. Unfortunately, these variable parts are not of a fixed length and thus some rules will inevitably match multiple instructions. On the other hand, from experimental results, it seems that these ambiguities are one-sided, i.e. weaker rule describes also an instruction that has its own stronger rule and not vice versa.

Since it can be assumed that every instance of the virtual machine contains every instruction exactly once, these ambiguities may be resolved iteratively by propagating constraints imposed by this assumption. For instance, *in* instruction contains only the rolling key update and thus a YARA rule describing *in* also matches every other instruction, yet every other rule will not match *in* instruction (due to limited subset of instructions present in the rolling key update) and thus we can conclude that *in* instruction will be matched only by its derived YARA rule.

Interestingly, there is one instruction that has a misleading name – *nand*. This instruction does not implement NAND (negated AND) operation, it does calculate NOR (negated OR) instead. This seems to be a misnomer only – logical operations, such as OR, are implemented correctly. E.g. OR applied to numbers a, b (bytes, words, or double-words) is calculated in the following way

$$or(a, b) = nor(nor(a, b), nor(a, b)).$$

If *nand* instruction corresponded to logical NAND, then this construction would yield AND.

2.1.2 Code conversion

To use WProtect to obfuscate a part of a code, there has to be a small change at the source code level. By default only *C++* and *C* languages are supported, although support can be easily extended to more programming languages. At the start of a code to be protected a function *WProtectBegin* has to be called, similarly a function *WProtectEnd* has to be called at the end of this code. These functions are defined in the header file *WProtectSDK.h* which has to be included. These functions serve the only purpose – they surround the code by ASCII strings *WProtect Begin* and *WProtect End*, these strings are prefixed by two bytes (0xEB and 0x0F – these are interpreted as a jump to an instruction that follows the respective ASCII string) and terminated by a NULL byte. Naturally, we can use other ways to insert these strings into the machine code at their respective places.

To convert a tagged machine code into WProtect virtual machine byte-code a configuration file called *build_vm_handle.lua* is required. This file defines how x86/x86_64 instructions should be translated into sequences of WProtect instructions. Naturally, these translation recipes can be used handle to other kinds of obfuscations (i.e. dead code insertion and code reordering). For example, there is a preset configuration file called *build_vm_handle_kido.lua*[3] that shows us how these obfuscations can be done, its dead code insertion function is shown in Code 2.1. This code uses *GetJunkFunc1* to push a random register and conversely, *GetJunkFunc2* pops a random register. Both steps are repeated in a loop (*LoopDoFunc0*) several times, the upper bound for the number of iterations in the loop is defined in the same configuration file.

```
1  function DoSomeJunk(  )
2          local index = math.random(5)
3          if (index == 1) then
4                  local d = math.random(max_junc_count)
5                  local j = math.random(jfunc_count)
6                  LoopDoFunc0(GetJunkFunc1(j), d)
7                  LoopDoFunc0(GetJunkFunc2(j), d)
8          elseif(index == 2) then
9          elseif(index == 3) then
10         elseif(index == 4) then
11         elseif(index == 5) then
12         elseif(index == 6) then
13         elseif(index == 7) then
14         elseif(index == 8) then
15         elseif(index == 9) then
16         elseif(index == 10) then
17         end
18 end
```

Algorithm 2.1: *build_vm_handle_kido.lua* dead code insertion function

The configuration file handles several types of obfuscation. At first, every instruction has its equivalent described by WProtect's instructions. Even in the

[3]This configuration file is probably provided for demonstration purposes.

default configuration file the sequence describing an original instruction is not intuitive as it combines pattern obfuscation (even though WProtect has *add* instruction, the pre-configured instruction sequence for addition is more complex) and also, as the Lua Listing 2.1 suggests, dead code insertion.

The register states are saved to the stack's bottom at the virtual machine's initialization, similarly these states are recovered from the same location during *ret* instruction execution. These locations are normally accessed through special instructions (*pop_reg* and *push_reg*) only.

The byte-code is generated instruction-by-instruction almost linearly, i.e. code blocks also form continuous sequences of bytes in the byte-code. As the rolling key has to be updated after every instruction, the virtual machine resets the rolling key to 0x12345678 during every "jump" instruction. Currently, two types of jumps are implemented – there are unconditional jumps and conditional jumps. Unconditional jumps are realized by pushing the address onto the stack just before calling *set_key* instruction that realizes the jump itself. Unconditional jumps are more complex as it is necessary to push two addresses onto the stack and also verify the jump condition. Both these jumps use addresses provided by the byte-code (immediate value) as their destinations.

While technically *ret* instruction can be used as a jump in x86/x86_64[4], in WProtect it causes virtual machine's termination and subsequent return to a x86/x86_64 code that follows the virtual machine's code.

The generated byte-code is inserted into a newly created section *.WProtec*, along with the virtual machine itself. It is worth noting, that bytes of the byte-code are written in reversed order, that is first instruction to be executed is defined by the last byte of the byte-code.

[4]So called return-oriented programming (ROP)

3. Extraction of a code protected by WProtect

As the virtual machine's structure is rather complex, even for today's decompilers, it is necessary to strip down this layer of obfuscation. Fortunately, WProtect has a plethora of clues and a rather simple internal structure, thus this can be done mostly automatically. We shall break down the extraction process into several logical steps. The main reasoning for such division is the observed behaviour of malware that uses WProtect – authors adjusted some parts in order to hinder an automatic analysis. While these adjustments may prevent fully automatic analysis, the algorithm needs only partial adjustments to support these changes. We shall name two stages that enable adjustments without the need to rewrite the engine itself.

The first one is called virtual machine insertion. During this stage, a virtual machine's code (not its byte-code) is inserted into the original binary files. While the construction is more-or-less standard, specific implementation variabilities may complicate the analysis. On the other hand, there seem to be several invariants that are preserved even among binary files that utilise modified WProtect.

The second one is called instruction translation. The translation is configurable through aforementioned *build_vm_handle.lua* file. It allows us to redefine how each x86/x86_64 instruction shall be interpreted and what obfuscations shall be applied to it. By default, the translation only breaks patterns in the code, inserts a dead code and performs a code reordering on a small scale. However, this could be easily extended to other methods, several options will be elaborated in Chapter 4.

This work also contains several Python scripts[1] that can be used to automatically extract a code protected by WProtect, or at least its equivalent (in terms of inputs, outputs, and side-effects). All the described stages of the code extractions are implemented in one of these scripts. The scripts utilise two important libraries and one external service. The first library is called *Miasm*[2], it is used to disassemble a given library. *Keystone*[3] assembler is used to assemble extracted instructions, represented in an assembly language, into a x86/x86_64 machine code.

3.1 Virtual machine extraction

The first problem, that has to be tackled, is the detection of the virtual machine itself. While it seems that the samples mostly preserve WProtect's section name (*.WProtec*), there seems to be variability in the initialization and the instruction selection. By default, the virtual machine initialization sets the key to 0x12345678 by a *MOV* instruction, where the key is passed as an immediate value. Similarly,

[1] Also available on `https://github.com/stredaa/vm-translator`.

[2] A reverse engineering framework in Python sourced at `https://github.com/cea-sec/miasm`.

[3] A lightweight multi-platform and multi-architecture assembler framework. It is accessed through its Python bindings. The project's website is `http://www.keystone-engine.org/`.

it is possible to deduce the location of the instruction table by looking for a *PUSH instruction_table[4 * EAX]* followed by *RET* in the *.WProtec* section. Each of these locations is contained in one code block, where code block is a sequence of lines of code where only the last instruction is either a jump or *RET* and the first instruction is a destination of some jump, *RET*, or it is at the base address.

To detect these blocks, a YARA rule is employed. This choice was made to provide flexibility should the initialization or the instruction selection change. At first, a binary file is provided. Then a Python library *Miasm* is utilised to split the machine code contained in the binary file into code blocks, a first block is recovered thanks to the base address. Other blocks are recovered by recursive exploration, i.e. every block may have references to other blocks (e.g. jump) – these references are then followed in order to add new blocks. Note that this approach does assume that anti-disassembly methods are not used in the binary file, this kind of protection is out of this work's scope. Similarly, we assume that the initialization block of the virtual machine is consisting of a reachable code, i.e. its block will be discovered during the exploration phase.

Once these blocks are detected, the initial key value, the byte-code's location, the instruction table location, and the instructions themselves are recovered. Moreover, the instruction selector's rolling key update and instruction deobfuscator are extracted from the blocks. Then they are assembled through *Miasm* into small machine code snippets. These snippets are symbolically executed through *Miasm's* symbolic execution engine[4]. Symbolic expressions describing changes in the key or instruction byte are recovered from the symbolic engine and translated into Python source code. This source code is evaluated in order to define lambda expressions describing the key update and byte-code deobfuscation. This process is implemented in *vm.analysis.WProtectEmulator* class.

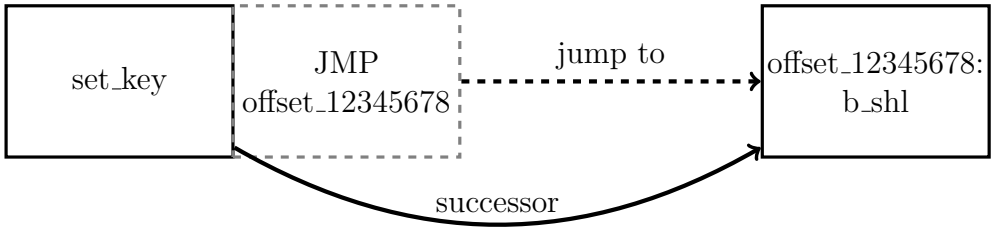
Moreover, by providing a suitable class describing WProtect's instructions (e.g. *vm.mnemonic.WProtectMnemonic*), it is possible to extract virtual machine's instruction set for further processing. Again, YARA rules are employed in the process of instruction recognition and a file *mnemonics.yara* contains YARA rules describing WProtect's default instruction set. As aforementioned in the previous chapter, these YARA rules cannot identify an instruction without any context *per se*. However, comparison of all instructions (or more correctly their corresponding positive detections) provides enough constraints to resolve these ambiguities. Generally, every virtual machine's instruction also performs a rolling key update and therefore the rolling key update also has to be extracted – this is done automatically in the same manner instruction selector's rolling key update was recovered.

3.2 Instruction sequence recovery

In the previous step, we have recovered the byte-code's location (coinciding with the initial instruction pointer value), instructions themselves, the rolling key, the rolling key update and deobfuscation routines. Now we can walk through

[4]Instructions are evaluated symbolically. For instance, it allows computing of a symbolic expression describing the state of a register at the end of the symbolic evaluation. If we symbolically executed *INC EAX* then the symbolic representation of *EAX* would be $EAX = EAX_{inititial} + 1$.

Figure 3.1: Code block chaining diagram



all possible[5] paths step-by-step with *vm.analysis.WProtectTracer*. As the class' name signifies, it also allows an insertion of a function that is executed with every encountered instruction, effectively creating a callback that may be used for further processing. This callback can be used to insert nodes, an intermediate representations of virtual machine's instructions, into an instance of *semantic.flow.WProtectControlFlow*. Such data may be used in two ways – it might be desirable to render a control flow graph or, more importantly, to reconstruct code blocks. Due to the fact that jump addresses are provided as immediate values, we are able to perform this translation to an intermediate representation statically. The only parts, that are evaluated (although symbolically), are rolling key updates and immediate value deobfuscations.

At this point, the code extraction is almost done. Code blocks are a crucial ingredient in the transformation of the byte-code's intermediate representation into machine code. Code blocks can be translated into an assembly language instruction by instruction without any further contextual analysis, so we can perform such translation on each code block. Once all the blocks are translated, they can be easily connected together by means of *labels*[6] and (conditional) jump instructions. Jump instructions are inserted after a code block's last instruction, the destination of each jump is a location represented by a label of its preceding instruction's successor. The diagram of this process is shown in Figure 3.1.

Translated code blocks, with inserted jumps, are assembled with *Keystone* and put together.

[5]We are still under the assumption that no anti-disassembling methods were used, so the control flow should be recoverable.

[6]A sequence of ASCII characters followed by semicolon that represents a specific location.

4. Mitigating WProtect's vulnerabilities

Due to the nature of WProtect, it is possible to recover a complete control flow graph under the given assumptions. To mitigate exploited vulnerabilities changes in either configuration or design have to be incorporated so that even code unprotected by anti-disassembling techniques gains some anti-disassembling properties – for instance polynomial time irrecoverable control flow graph.

Some of these modifications, such as self-modifying code, shall be presented as concepts only, as their full evaluation is out of this work's scope. Modifications covered by the theory presented in chapter 1 shall be evaluated more thoroughly.

4.1 Default configuration file

The default configuration file has several deficiencies. By itself, it employs only straightforward obfuscations such as dead code insertion and programming idiom hiding. Even worse – it creates new idioms as all jumps follow very similar structure. Furthermore, the control flow is almost identical to the original code's control flow.

The first proposal is based on superfluous conditional jumps combined with opaque predicates. Additional conditional jumps break the original control flow. The problem of destination selection might be solved by a destination caching for further reuse. As we are restricted to unobservable side-effects, opaque predicates are also introduced in order to mask these invalid jump destinations.

The only real hindrances are the key reset, that occurs during *set_key* invocation, and the necessity of getting the next instruction's address. A rather straightforward approach requires a modification of *src/WProtect/Protect/VirtualMachine/BuildVMByteCode.cpp* in order to provide next instruction's address and a manual jump handling either through *set_key*, or directly by using *set_pc* instead. Or we may even directly exploit access to x86/x86_x64 registers and recreate these effects by a register manipulation.

Unfortunately, both of these approaches do not follow the WProtect's idioms and thus, unless the key reset is also obfuscated or additional pre-processing and restrictions are added, they are distinguishable from regular jumps.

This modification is particularly effective against the algorithm used in this work which exploited the correspondence between jumps in the byte-code and the original code. Furthermore, the code's granularity is increased. High code granularity may become a problem because code blocks may be far too short to optimise without any further context. Thus it may render block-wise optimisation inefficient, forcing the analyst partition the code into bigger units that may contain jump instructions. In addition to a risk of an over-optimisation, analysis will have to deal with code redundancy caused by opaque predicates. Also, we can no longer guarantee strong correspondence between the original and recovered control flow.

The second proposal is the introduction of randomness into instruction templates. It is straightforward to assign multiple templates to one instruction in

order to complicate pattern discovery. This method is partially used in the exemplary configuration file *build_vm_handle_kido.lua* where superfluous pushes and pops are inserted, nevertheless these can be easily removed by simple heuristics tracking the stack's size; the proposed extension is generally vulnerable to these heuristics as alternative templates do not depend on dead code insertion only. While this modification does not affect the algorithm devised in this work in any way (in terms of complexity), it may be efficient against machine learning algorithms that could be deployed to recover patterns corresponding to original x86/x86_64 instructions.

4.2 Rolling key updates

Rolling key updates are generated randomly and may contain trivial updates that allow more efficient key state backtracking. Moreover, every unmasked instruction byte has to be in a certain range (a size of the instruction table). This fact can be utilised to narrow down the space of possible preceding rolling key states.

To mitigate these problems two solutions will be proposed. In order to make every key "valid" in terms of correct instruction it should be enough to take the instruction byte value modulo the size of the virtual machine instruction set. While this improvement may allow for adding some randomness, it only makes backtracking optimizations less useful, as we can still exploit stack contents to discard invalid key states (e.g. stack underflow or addresses required for *set_key*).

However, since the byte-code's entry point is hard-coded, it is rather straightforward to observe that this presented vulnerability has a low severity. Furthermore, this modification does not generally introduce significant slowdown to the complexity of said backtracking. It only causes a removal of several properties that may be exploited to create heuristics solving some special cases.

Nevertheless, there is one upgrade that may be worthwhile, especially in conjunction with jump insertions. A rolling key reset, that occurs during jumps, always uses a hard-coded key that is pushed as an immediate value. This could be mitigated by an application of several operations on the immediate value in order to force the analyst to recognize and evaluate the corresponding instruction sequence. Similarly, the key's value may be pseudo-randomly generated, for instance, from the destination's address.

4.3 Instruction selector and WProtect initialization

Currently, the framework relies on *Miasm* disassembler to discover code blocks containing the virtual machine itself. Using anti-disassembly techniques such as a self-modifying code or a control flow obfuscation[1] may cause the disassembler to fail before recovering WProtect's code blocks.

While this modification can be partially defeated by YARA rule adjustments and disassembling heuristics (such as limited virtual machine location brute-

[1] E.g. a jump to a destination that has to be decoded first. Note that these obfuscations often already thwart disassembler's heuristics.

forcing), self-modifying code could be used to enforce the employment of dynamic analysis or extensive static analysis during the code's analysis.

All initial values, that is the initial rolling key and the instruction table location, are also provided as immediate values. Similarly, we may utilise self-modifying code or hide these values in more complex computations to heighten the requirements on the automated analysis.

This modification may also be particularly restricting to potential analysis, as some parts of the code will have to be either dynamically analysed or symbolically executed.

These adjustments require a modification of *src/WProtect/Protect/VirtualMachine/BuildVMByteCode.cpp*.

4.4 x86/x86_64 to WProtect translation

This section covers mitigations that require extensive WProtect's source mode modifications, often on various levels. These modifications mostly target the structure of virtual machine itself and thus may impose new constraints on the translation process.

Code reuse

Every byte in WProtect's byte-code has at most one intended interpretation. This may be circumvented by rolling key updates and jumps – e.g. using different initial rolling keys may result in different runs of the virtual machine. To ensure the validity of the code, several restrictions have to be imposed on the WProtect itself; all intended runs must result in correct instruction unmasking and should not underflow or overflow the stack. These restrictions mostly limit randomness that is introduced into the instruction ordering and rolling key updates. This modification is illustrated in the following toy example:

Example. Suppose that every instruction has the identity as the rolling key update function. The same holds for the instruction selector rolling key update. The instruction unmasking is defined as bitwise-and. Let us define two initial keys $k_0 = 0$ and $k_1 = 1$ and assign a value 0 to *ret* and a value 1 to *set_key*. Then one run may end immediately, while the second may have its key changed to -1 (there are no zeroes in its binary representation) and then proceed in arbitrary computations.

This modification requires rather complex code changes. At first, it is necessary to find several keys and calculate rolling key update function, so that runs, they induce, do not exhibit any anomalous artefacts (e.g. stack underflow). These requirements may significantly limit the length of induced runs. However, if the initial key is determined by e.g. an opaque predicate then, unless the analyst can determine the result of the predicate, it is necessary to examine all possible runs.

A cheaper, while unfortunately also more easily detectable, variant is a direct byte-code modification. An instruction *b_write_mem* can write a byte to an arbitrary writeable memory region. If permissions of the WProtect's section are set to writeable then the code may modify itself at the runtime. Since this operation

requires a pointer to the memory location, it may be detected by a parameter-instruction pairing. Again, this may be hindered by its obfuscation.

Instruction grouping

WProtect uses a deterministic mapping of instructions onto a sequences of WProtect instructions. This property may be exploitable by pattern recognition algorithms to probabilistically recognize these sequences and find their pre-images. The aforementioned idea of utilising multiple templates for x86/x86_64 instructions may be extended to multiple instructions. We could introduce new instructions into WProtect's instruction set that are equivalent to specific x86/x86_64 or WProtect instruction sequences. For instance, we could create a special instruction for ASCII string comparison.

Again, as in the case of multiple instruction templates, this modification affects only pattern finding and pattern matching algorithms.

Polymorphic virtual machine

Similarly to the idea of a polymorphic byte-code, it may be possible to make the virtual machine itself polymorphic. This may be implemented in numerous ways. We shall propose two variants.

The weaker variant utilises encryption (or encoding) to obscure instructions. This may be useful to prevent YARA rule matching as now every original instruction has to be decrypted (or decoded) before its execution. Moreover, as this modification is easy to randomize, it significantly increases, in conjunction with WProtect initialization obfuscation, computational resources that are required for its analysis. Even though the overhead also affects a "user" , the "user" is not often limited by hardware and performance requirements[2].

We could upgrade this idea and use memory access to rewrite the virtual machine itself, e.g. reorder its instructions or change the whole virtual machine. If the source code of the "new" virtual machine is not embedded in a straightforward way (such as immediate values) then this modification enforces either a dynamic analysis or an extensive static analysis because we need to extract both instructions and a new byte-code from the first layer. In this case, WProtect's randomization (e.g. rolling key updates) proves to be especially daunting because it creates high amount (tenths to hundreds of unknown bytes if the original design is used) of unknowns that have to be determined.

[2] An automated analysis is often performed either in high volumes or on customer's hardware. In the first case, it is desirable to preserve a certain throughput. The latter case presents an ethical problem instead.

5. Conclusion

The WProtect obfuscation engine in its default configuration contains vulnerabilities that can be leveraged in the code extraction. The intended design preserves the original control flow structure and thus the code optimization may be done on the level of code blocks. This property is very desirable, the code optimization problem may be partitioned into small instances and the probability of recovered code's similarity to the original code is higher as the control flow preservation is enforced.

5.1 Future work

We have managed to recover a representative x86/x86_64 code. However, there is still much information that is not used. For example, the uniqueness of every instruction's translation template. Therefore, it may be worthwhile to design an algorithm that tries to identify patterns created by those templates – in an ideal case, this might lead to one-to-one deobfuscation (we recover the original machine code).

Similarly, we could transform the resulting code into LLVM IR (intermediate representation), e.g. by RetDec[1]. The resulting LLVM IR can be further optimized by LLVM leading to even more readable code.

Also, there are many possible engine's improvements, few of them were proposed in chapter 4. It may be worth exploring to find the least invasive methods of their implementation, ideally restricted to Lua configuration files only.

[1]Retargetabe Decompiler (RetDec) is an open-source machine-code decompiler based on LLVM. The project's website is `https://retdec.com/`.

Bibliography

[Adr13] Oskar Adrvidsson. Platform independent code obfuscation, 2013.

[BMZ01] NP Brousentsov, SP Maslov, and EA Zhogolev. Development of ternary computers at moscow state university. 2001.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[D+17] Deadwood et al. The clemency architecture, 2017. `https://2017.notmalware.ru/89dc90a0ffc5dd90ea68a7aece686544/clemency-201707271159.pdf`, Accessed 16. 4. 2018.

[DBC+03] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, pages 41–49, New York, NY, USA, 2003. ACM.

[Den18] Denuvo, 2018. `https://denuvo.com/`, Accessed 16. 4. 2018.

[Dol13] Stephen Dolan. mov is turing-complete. *Cl. Cam. Ac. Uk*, pages 1–4, 2013.

[Dom15a] Chris Domas. Def con 23: Repsych, 2015. `https://media.defcon.org/DEF%20CON%2023/DEF%20CON%2023%20presentations/DEFCON-23-Chris-Domas-REpsych.pdf`, Accessed 16. 4. 2018.

[Dom15b] Chris Domas. Repsych, 2015. `https://github.com/xoreaxeaxeax/REpsych`, Accessed 16. 4. 2018.

[GBC+05] David Gregg, Andrew Beatty, Kevin Casey, Brian Davis, and Andy Nisbet. The case for virtual register machines. *Science of Computer Programming*, 57(3):319–338, 2005.

[Kay00] Richard Kaye. Infinite versions of minesweeper are turing-complete. 2000.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[PE] Pe format. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx`, Accessed 16. 4. 2018.

[Pri94] Matt Priek. Peering inside the pe: A tour of the win32 portable executable file format, 1994. https://msdn.microsoft.com/en-us/library/ms809762.aspx, Accessed 16. 4. 2018.

[RA16] Kung-Kiu-Lau Rehman Arshad. *A Concise Classification of Reverse Engineering Approaches for Software Product Lines*. 4 2016.

[Res] Microsoft Research. Z3. https://github.com/Z3Prover/z3, Accessed 16. 4. 2018.

[Ric15] Brian Richardson. Why cheap systems run 32-bit uefi on x64 systems, 2015. https://software.intel.com/en-us/blogs/2015/07/22/why-cheap-systems-run-32-bit-uefi-on-x64-systems, Accessed 16. 4. 2018.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, January 2008.

[SGX] Overview of Intel® software guard extension enclaves. https://software.intel.com/sites/default/files/managed/14/0e/Overview%20of%20Intel%20SGX%20Enclave.pdf, Accessed 24. 4. 2018.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[WZH+11] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Differentiating code from data in x86 binaries. In *ECML/PKDD*, 2011.

[Xia] Wprotect. https://github.com/xiaoweime/WProtect, Accessed 16. 4. 2018.

[xor14] xorpd. *Xchg Rax, Rax*. CreateSpace Independent Publishing Platform, 2014.

[Zub16] Tomislav Zubcic. Experimenting with z3 – proving opaque predicates, 2016. http://zubcic.re/blog/experimenting-with-z3-proving-opaque-predicates, Accessed 16. 4. 2018.

A. Attachments

A.1 Source code

There is an archive with the framework's source code attached. It contains YARA rules for unmodified WProtect instructions, a YARA rule describing the initialization code block, toy examples protected by WProtect, and the source code itself.

The source code is released under MIT license, the newest version is available on Github: `https://github.com/stredaa/vm-translator`.

A.2 Instruction Table

Table A.1: Opcodes (part 1) of x86 WProtect's virtual machine instructions. x86 opcodes are used in comments where possible, otherwise a short description is provided.

mnemonic	**comment**
b_read_stack	read a byte in the stack (relative address)
w_read_stack	read a word in the stack (relative address)
d_read_stack	read a dword in the stack (relative address)
b_write_stack	write a byte in the stack (relative address)
w_write_stack	write a word in the stack (relative address)
d_write_stack	write a dword in the stack (relative address)
b_push_reg	push the saved register value (byte)
w_push_reg	push the saved register value (word)
d_push_reg	push the saved register value (dword)
b_pop_reg	rewrite the saved register value (byte)
w_pop_reg	rewrite the saved register value (word)
d_pop_reg	rewrite the saved register value (dword)
b_push_imm	push an immediate byte
w_push_imm	push an immediate word
d_push_imm	push an immediate dword
b_shl	SHL byte
w_shl	SHL word
d_shl	SHL dword
b_shr	SHR byte
w_shr	SHR word
d_shr	SHR dword
shld	SHLD
shrd	SHRD
b_nand	negated *or* on bytes
w_nand	negated *or* on words
d_nand	negated *or* on dwords
set_pc	set the value of the ESI register
ret	exit the virtual machine (return)

Table A.2: Opcodes (part 2) of x86 WProtect's virtual machine instructions. x86 opcodes are used in comments where possible, otherwise a short description is provided.

mnemonic	**comment**
in	NOP
rdtsc	RDTSC
cpuid	CPUID
check_stack	check if EBP > EDI + 0x64
dispatch	
push_stack_top_base	PUSH ESP
b_read_mem	read a byte from the memory
w_read_mem	read a word from the memory
d_read_mem	read a dword from the memory
b_write_mem	write a byte to the memory
w_write_mem	write a word to the memory
d_write_mem	write a dword to the memory
pop_stack_top_base	POP ESP
b_push_imm_sx	push an immediate byte as dword (SX)
w_push_imm_sx	push an immediate word as dword (SX)
b_push_imm_zx	push an immediate byte as dword (ZX)
w_push_imm_zx	push an immediate word as dword (ZX)
b_add	ADD byte
w_add	ADD word
d_add	ADD dword
b_rol	ROL byte
w_rol	ROL word
d_rol	ROL dword
b_ror	ROR byte
w_ror	ROR word
d_ror	ROR dword
set_key	change the rolling key and ESI
run_stack	call ebp + offset (byte on the stack)