Master research Internship

Master Thesis

Symbolic Evaluation and Disassembling x86 Low-Level Code

Domain : Binary Analysis - Symbolic Computation - Information Retrieval

Author:
Benjamin Rouxel

Supervisor:
Jean-Yves Supervisor

CARTE – LORIA

UNIVERSITÉ DE RENNES 1

Abstract

Malwares have been existing from the beginning of computer science. Nowadays malwares authors use more and more sophisticated obfuscations to hide their code. Analyzing malwares is not an easy task as their authors use of imagination to find obfuscations that defeat standard disassemblers. The major tool they use is a packer which warps the malicious code by series of self-modifying steps.

The key tools to study or detect malwares, are unpackers and disassemblers. Most of the time, those tools are studied independently and thus not compatible between each other. In addition, nowadays we have to process a huge amount of suspicious softwares, in this context existing solutions are not efficient.

This document focuses on x86 malwares and how to build a disassembler in conjunction of an unpacker which can handle self-modifying code and other obfuscation techniques. It uses dynamic introspection and static analysis to reconstruct the program in a high level of abstraction. The proposed solution implements the work of [1, 2, 3]. The method is evaluated as sound and efficient. Results are encouraging but further work will show if the accuracy of the disassembler is improvable.

Contents

1 Introduction

Context Malwares have been spread from the early 80's. They were first propagated through floppy disk then through the network attached to files like Microsoft Office Document (MyDoom[1] - 2004), then attached to emails (ILoveYou[2] - 2000) or propagated by infected USB drives (Stuxnet[4] - 2010). Recent attacks we heard about combine different techniques such as phishing which consists in making a fake website or email with colors of a bank or a legitimate organizations. Malwares are then installed on behalf of the victim while navigating on the fake website. The method used to spread malwares evolved with time, and the goal of their authors also changed.

At the beginning malwares authors were more interesting about flattering their egos. As it was a challenging game, they were pride to explode the counter of infected machines. Then criminal organizations saw the possibility to make profit and some malwares appeared like CryptoLocker[3] (2013), which encrypts the victim hard drive who then must pay to get the decryption key. Then, the fashion of Distributed Denial Of Service (DDOS) emerged ; this is an attack against a service or a server to make it unavailable by sending a lot of requests in a short period of time. Hackers developed malwares to create botnets which are a network of infected machine called zombies and answering to a command-and-control server (Waledac[4] - 2010). The botnet is then a huge strike force to perform DDOS. In the recent news, Julian Assange and Edward Snowden revealed several secrets from NSA[5]. We now know that such kind of agency builds malwares to infect other governments or to watch people ensuring they are not terrorists.

Moreover, following statistics given during Shonan meeting on low-level code analysis (March 2015), Google receives more that 300 000 files per day and has a collection of 400 millions of malware. All these files should be analyzed and classified in order to build defenses against malware threats. Thus performance become a major concern when dealing with a huge amount of softwares: *big code* analysis. But performance comes with a cost of precision, hence I will have to find the good threshold between performance and precision.

Problematics To prevent the threats coming from malwares, security analysts need tools to analyze a posteriori the new discovered malwares, and a priori to prevent our devices from installing malicious softwares and so to protect our lives. The former requires a sound disassembler/decompiler to allow a human analyst to study the behavior of the malware ; obviously malwares are distributed in a compiled form. Analyzing compiled code is called *reverse-engineering* as it is the reverse process of the software engineering. Also to protect our devices we need robust and efficient anti-viruses able to match every known malwares even when they are hidden with new techniques.

Current anti-viruses detection technique are based on software signatures or behavior analysis [5]. However for mutating viruses, the signature will change at each mutation bringing the need to update the database signature. Also, malwares are mostly hidden behind obfuscations and self-modifying phases, and software signature based detection technique only match the code of the first phase which is not really the malware. Thus changing tool (called a packer) used to hide a malware also changes its signature.

[1] http://edition.cnn.com/2004/TECH/internet/01/28/mydoom.spreadwed/
[2] https://en.wikipedia.org/wiki/ILOVEYOU
[3] https://en.wikipedia.org/wiki/CryptoLocker
[4] https://en.wikipedia.org/wiki/Waledac_botnet
[5] National Security Agency – Intelligence Service from the United States of America

When analyzing malwares there is different steps. First is to unpack the self-modifying code, then disassembling each phases followed by the reconstruction of a representation in a high level of abstraction on which it is possible to perform some complex static analyses or transform it in an other intermediate language prior to those kind of analyses. Each step is dependent of the previous one. The success of a step has an impact on the next one ; i.e.: if the disassembling is wrong then the more complex analysis will also be wrong. It is crucial that the unpacker and the disassembler are robust and sound as they are the two first steps.

Current anti-viruses implement basic unpacking features [6] but they fall into the trap when malwares authors use complex packers.

Commercial and research-oriented tools are designed for reverse-engineering purposes and use complex computation and/or heuristics to resolve common problem. This result in a large run-time overhead, making them unable to handle huge amount of code. In addition, some of them are able to deobfuscate some common obfuscations or packers, but do not resolve this problem in a general way. Also most of the known disassembler can not deal with self-modifying code, and unpacker are not build to be used with those disassemblers.

Objectives My objectives are then in two parts, analyzing and detecting malwares with a concern of performance for the detection part. The team CARTE from LORIA[1]/INRIA[2] aims to build a sound and efficient malwares detector (an anti-virus). The decision procedure of the detector (the procedure that decides if a software is malicious or not) is based on the morphology of the virus introduced by Kaczmarek [7]. The morphology is the Control Flow Graph (CFG) extracted from the input program. Thus the decision procedure needs the CFG extracted from the malware [8]. The morphological analysis needed for the decision is out of the scope of this master thesis and is detailed by Thierry[3].

The purpose of this master thesis is then to create a sound and efficient disassembler and CFG extractor of obfuscated and self-modifying malwares which can be used either by an analyst either by an automatic process on huge amount of code. The unpacker part is provided by CARTE, and must be linked to this disassembler under the project name CoDisAsm.

The second objective of this master thesis is carried by the ANR[3] project BINSEC [4] which aims to build a powerful binary analysis platform. Thus the created disassembler must be flexible enough to easily interact with the rest of this project.

This master thesis is the prolongation of the PhD thesis of Kaczmarek [7], Reynaud [1], Calvet [2] and Thierry [3]. All of them studied virology. Kaczmarek used formal methods to define the behavior of malwares and gave a formal frame for the morphological detection of malwares. Reynaud well studied the behavior of self-modifying malwares and the dynamic instrumentation of them. Calvet continued the work of Reynaud and showed the veracity of using dynamic analysis tools in order to catch self-modifying malwares. Finally, Thierry improved Calvet's and Reynaud's studies and finalized the frame for the preparation of the implementation.

Thus my experimental research topic aims to mix the techniques detailed in those thesis and evaluates the whole method in order to determine if it is usable in a system handling a huge number of software in a short period of time.

[1] Laboratoire lOrrain de Recherche en Informatique et Automatique

[2] Institut National de Recherche en Informatique et Automatique

[3] Agence National pour la Recherche

[4] `http://binsec.gforge.inria.fr/`

Organization of the document Section 2 introduces the basic knowledges about hardware and software architecture by looking inside an executable file and the way it is handled by a machine. Section 3 presents what is a *reverse-engineering* process in our context and the formal frame used by the implementation. Section 4 discusses about the issues a disassembler is confronted to and thus justifying the chosen method to implement a robust and efficient dissembler. Then section 5 describes the implementation, followed by the test protocol and results in section 6. Finally section 7 will briefly depict a panel of related work before concluding in 8.

2 Introduction to software and hardware architecture

Prior to detail how to disassemble an executable file, the reader must be aware of some basic knowledges about hardware and software architecture. Most of the computer around the world are running Microsoft Operating System. According to NetMarketShare, more that 90% of computer are on Microsoft OS for the year 2014. Also the main processor architecture is based on x86 architecture family from Intel ; Microsoft OS is only compatible with x86 and x86_64[1] processor, this definitely places this architecture as the most used. One of the primary goal of malwares authors is to infect the maximum number of machines. Hence most of the existing malwares have been built to run on this hardware and software architecture. So, naturally we choose to focus on it.

Hardware Like most processors, the x86 architecture inherited from the von Neumann model [9]. He described a generic machine with four main parts:

- The **Processing unit** is able to compute arithmetic or logic operation, and implicitly it updates some flags to get more information about the last computation, i.e.: overflow flag indicating an arithmetic overflow
- The **Control unit** is in charge of controlling the execution flow by managing a program counter (PC) pointing to the next instruction in memory to execute
- The **memory** which contains any needed bytes (code and data) to execute a program
- The **Input/Output controller** to handle input/output communication to external devices

Note on the memory : In modern processor, a running program sees the memory as it is alone in it. This mechanism is handled by the OS which allocates virtual memory pages where the virtual addresses are translated to physical one by the hardware with a **Memory Management Unit (MMU)**. Also, there is several level of memories like caches and some specific registers used to speed up accesses to data. In our simplistic/idealistic view of a processor, I will only consider one level of a big flat memory as it is the way a running program (a fortiori a malware) sees the memory.

Machine language Each existing family of processors has its own machine language. Machine code must be seen as a sequence of bytes that the electronic hardware can understand. Machine language is linked to a textual representation which is much more human friendly and called *assembly language*. Assembly is described as an Instruction Set Architecture (ISA) and for the x86 family it

[1] They recently added the support for ARM, but this does not change statistics, x86 is still the most used architecture

contains more than 700 instructions [10]. Indeed each time a new processor is added to the family it adds its bunch of new features and also new assembly instructions.

The x86 ISA is built on the principle of Complex Instruction Set Computing (CISC), as opposed to Reduced Instruction Set Computing (RISC) which includes less features. The x86 assembly instructions have variable length with a maximum of 15 bytes, and do not need to be aligned on power of 2 in memory. This ISA is then more flexible than RISC architecture, at the cost of possible hijacks (cf section 4) and ambiguities.

The format of an instruction is presented by figure 1. The prefix add optional behavior to the instruction like *rep* making the processor to repeat the current opcode until the register *ecx* reaches 0. There exists two styles for representing assembly, one from AT&T and one from Intel, here this document arbitrary use the later.

Examples of opcode and operands are presented in table 1. In the example of the jump the operand is 5 while the syntax says to jump 7 bytes. For historical reason, developers have in mind that the PC is at the beginning of the instruction until the instruction is fully executed, while for the processor the PC is at the end of instruction that it has just consumed. So the remaining 2 corresponds to the size of the jump instruction: $5 + 2 = 7$.

Figure 1: Instruction format in x86 assembly language

Opcode	Operands	Syntax	Semantic
a3	00 00 00 00	mov eax, 0**x**0	Stores the 32 bits of value 0 into register *eax*
f4		hlt	Stops the execution of the program
eb	05	jmp 0**x**7	Modifies the PC as $PC = PC + 0\mathbf{x}7$
55		push ebp	Stores the value of the register *ebp* on top of the stack

Table 1: Example of x86 instructions

Executable file Writing software in machine code is very difficult and error prone. People would prefer assembly language, or even higher level languages such as C or C++. Then their code must be compiled into machine code to get an executable file.

The structure of the executable depends on the OS, for Microsoft OS the format is named PE (Portable Executable). It can be summarized as shown by table 2. Headers will contain a lot of useful information for the program loader inside the OS, i.e.: the entry point (first instruction to execute), the size of binary, the signature. Those information describe the executable file and despite of their values are common to every PE file. The *Section Table* will contain the list of every sections in the executable, but also their size and start address. A section is a portion of bytes which can be data or code interchangeably. Named section (i.e.:.data, .text) are only significant for humans.

PE executable
Headers
Sections table
Sections some code imports some data ...

Table 2: PE binary format

3 Reverse-engineering an executable binary file

As mentioned above, to get an executable, a software must be compiled into machine code. Thus the compilation process is to translate a program from a source language to a target language. Generally the source language is from a high level of abstraction such as C, C++ ; meaning that the program author does not have to care about the OS or architecture on which the final executable will run.

Most of modern compilers transform the source code into an Intermediate Representation (IR). Then the compiler performs some analyses and optimizations on this IR independently of the source language and structure. One of this IR can have the form of a Control Flow Graph (CFG). This is a graph where nodes are a sequence of instructions (called *basic-block*) and edges are possible executable paths through those *basic-blocks*. Then the compiler transforms the IR into assembly code and finally to machine code when building the executable binary file. An example of a C code, the corresponding x86 assembly and the CFG are presented in figures 2. To go further with compilation please see *The Dragon Book* from Aho [11].

Reverse-engineering is the opposite process of the compilation as shown by figure 3. And by disassembling and CFG recovering it aims to reverse the last action of the compiler. This problem has been a lot studied, however as Paleari et al [12] showed every disassembly are incorrect. To reach such conclusion, they compared the instructions' semantic decoded by the disassembler with the semantic cabled in the processor when executed the same instruction. And they observed some differences.

In a first glance, it appears that disassembling is easy because it is only a matter of translating an opcode into its textual representation. However the von Neumann architecture introduced in the previous section allows to mix data and code. Also to decode a perfect disassembly of a program P, every decoded bytes must be reachable by at least one execution path of P (definition 1).

Definition 1 *The perfect disassembly D of a non-self-modifying program P is for every addresses a in $[0; 2^{32}[$: $D = \bigcup\{(a, decode(a, P))|P()reachesa\}$*

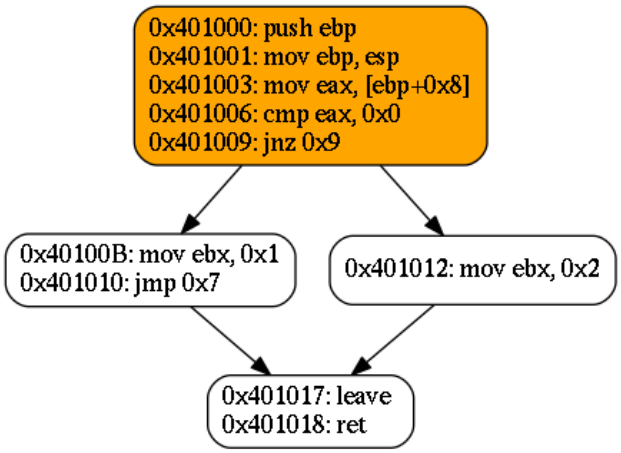
In general case, disassembling is undecidable, Calvet [2] gave the idea to reduce the halting problem to the disassembling one. Let's take a program CP verifying that a path exists to an address in an assembly code. For every non-self-modifying program P and address a, it returns $CP(P, a) = 1$ if and only if there exists a path to reach a, otherwise $CP(P, a) = 0$. If P is a program that do not need any input, and only contains the instruction *hlt* at address a. Then it is the same as answering if P halts, which is known to be undecidable.

```c
void main(int a) {
  int b;
  if(a == 0)
    b = 1
  else
    b = 2;
}
```

(a) Exemple of a C code

```
0x401000: 55              push   ebp
0x401001: 89 e5           mov    ebp,esp
0x401003: 8b 45 08        mov    eax, [ebp+0x8]
0x401006: 83 f8 00        cmp    eax,0x0
0x401009: 75 07           jne    0x09
0x40100b: bb 01 00 00 00  mov    ebx,0x1
0x401010: eb 05           jmp    0x07
0x401012: bb 02 00 00 00  mov    ebx,0x2
0x401017: c9              leave
0x401018: c3              ret
```

(b) Example a possible x86 assembly version of the above C code - intel syntax



(c) CFG corresponding of the code besides

Figure 2: Full example from C to the disassemble CFG through the assembly code

Disassembling code is undecidable for non-self-modifying code. Also, disassembling self-modifying code is a harder problem, it is undecidable too.

It is very difficult to reverse the compilation process, because when a binary file misses a lot of information, i.e.: there is no more variables. Also when dealing with malwares there is less information as usual because their author stripped the executable ; they removed every symbol from the binary just leaving the absolute necessary information to run the program. For example there is no access to the symbol table anymore, or to the jump table for indirect branch.

Following are 3 main different approaches to disassemble machine code. The two first are static methods ; they statically analyze the code like a compiler would do. The third approach is a dynamic analysis based on the execution of the program. The linear sweep disassembling method is given as reference, while the two others approaches prepare the implementation.

IDA pro[1] is the commercial reference for disassembling. It embeds great features and graphical interface but uses a lot of heuristics to detect some elements. It only uses static methods to disassemble and is not very robust with malwares.

3.1 Linear sweep disassembler

The most known *linear sweep disassembler* is GNU/objdump[2]. It takes a binary and extract the entry point from the header and then linearly disassembles the file by finding the $n+1$ instruction at the address of instruction n + the size of instruction n. An example is shown by figure 2b where

[1] https://www.hex-rays.com/products/ida/

[2] https://sourceware.org/binutils/docs/binutils/objdump.html

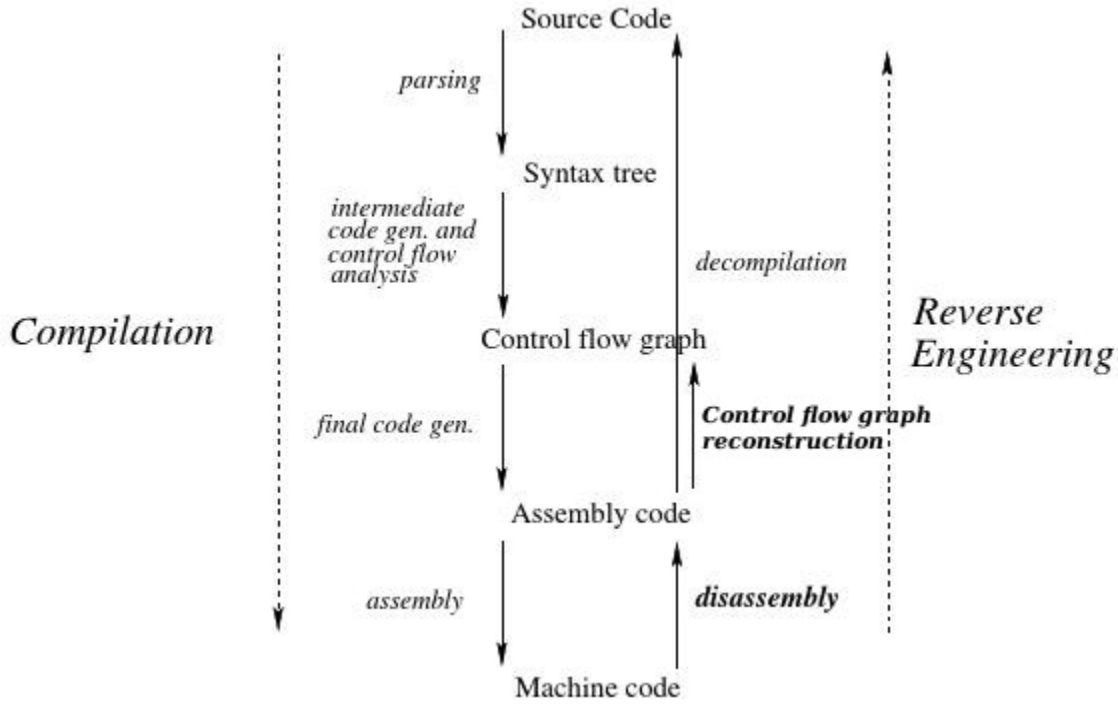


Figure 3: Compilation vs Reverse-engineering

instruction *leave* is found after the *mov ebx,* $0\mathbf{x}2$ at the address $0\mathbf{x}401012$ which is (address of the *mov*) $0\mathbf{x}401012 + 5$ (size of the *mov*).

This technique works well to disassemble well-aligned code, but ca not be used to distinguish data and code as it will disassemble any bytes after the entry-point. This method is presented as it is taken as a reference to compare the next disassembling method.

3.2 Recursive traversal

Linear sweep disassembler are not very robust, previous works use the *recursive traversal* of the binary and I will present a modified version in the implementation which is more efficient. This consists in disassembling the binary file while taking care of the possible execution flows. Depending on the instruction's type, the decoded next one will be found linearly after : if the type is a sequence flow instruction (i.e.: mov, jne) ; or at a target destination depending on the operand : if the instruction's semantic modifies the execution flow (i.e.: jmp, jne).

The algorithm 3.1 of a recursive traversal disassembler is given by Thierry [3]. The successor function is in charge of computing all the next possible instruction of I depending on the type of I.

This technique works well to disassemble any binary code, and because it takes care of the control flow, it allows to recover a CFG at the same time as disassembling (presented in figure 2c). With the assumption that any bytes not decoded during the recursive traversal is data, such disassembler are able to differentiate code and data.

However this algorithm can lost track of the control flow when dealing with *indirect branch* or some obfuscations (see section 4) . In practice, this algorithm is not very efficient when dealing

Algorithm 3.1: Recursive traversal of a program P

```
Input  : A program P composed and its entry-point
Output: D the perfect disassembly of P : a set of (address, instruction)
  function RecursiveDisassembler(P, ep)
    // Starting disassembling at the entry − point
    return RecursiveDisassemblerAux(ep, ∅, P)
  end function
  function RecursiveDisassemblerAux(addr, D, P)
    instr ← decode(addr, P)
    D ← D ∪ {(addr, instr)}
    // Recursively disassemble all unknown successors
    foreach addr' ∈ successor(I) do
      if (addr', ∗) ∉ D then
        D ← D ∪ RecursiveDisassemblerAux(addr', D, P)
      end
    end
    return D
  end function
```

with program containing a huge number of instructions : *big code* analysis. Indeed, it will make a recursive call at each instruction generating a lot of movements on the stack, but the logic is more easily understandable than in my modified version presented in the implementation section 5.2 which uses an iterative algorithm to perform the same task.

3.3 Dynamic reconstruction

The previous algorithm 3.1 statically disassemble the binary file. This is opposed to a dynamic analysis which executes the code. There is three possibilities to dynamically analyze the code :

- **Emulation** : the code is executed in an emulator such as QEMU/TEMU[1], this method is used by the BitBlaze platform [13].
- **Debugging** : the code is executed in debugging mode, the instrumenting tool will use the *Trap Flag* of the processor to make it stop after each execution, or insert trap instruction in the code to stop the execution (GNU/gdb[2])
- **Instrumentation** : the code is executed directly on the processor, but each instruction of the initial program is surrounded by added ones that allow the instrumentor to capture the evolution of the context (e.g. registers, memory, PC ...) Pintool[14]

Some of the mentioned references mix those techniques to have better results, making the frontier between them not so clean.

The goal of such techniques is to get an execution trace containing the sequence of the effectively executed instruction. Then depending on the tool, the analysis is done while executing the program

[1] http://wiki.qemu.org/Main_Page
[2] https://sourceware.org/gdb/

to avoid unwanted behaviors [15] or a posteriori to the execution [2] by analyzing the trace. The execution trace depends on the input arguments, or from the environment at the time it is executed. Hence one execution might not be enough to well represent the program and to fully analyze the it. For example the malware Jerusalem[1] (1987) only fire a malicious behavior upon every occurrence of Friday the 13th. Bringing the need to tweak the system clock to get a relevant execution trace. Some studies will execute the program several times while varying the input to ensure the discover of new execution paths [16].

Reynaud [1], Calvet [2] and Thierry [3] have a lot studied dynamic execution in order to analyze malwares, and Calvet gave a good explanation of the usefulness of such technique when dealing with malwares. Indeed they are protected by several levels of self-modifying code (cf: section 4.3), and the execution trace helps flattening them (cf: section 5.1) Moreover, Calvet showed that getting only one execution trace per running process is also enough for detecting malwares.

The actual tracer build from previous work records for each executed instruction the explicit and implicit impact on the system; i.e.: *push ebp* explicitly pushes the register *ebp* on top of the stack, and implicitly decrements *esp* by 4[2]. This has the advantage of being exhaustive and well represent the evolution of the system. However, this makes a huge trace file with a lot of information that might not be needed for an automatic CFG recovery tool, and so decrease the performance. In the implementation section 5.1, I will present a modified version I managed to use in coordination with the team engineer to improve the performance.

Besides the usefulness, malwares authors are aware of such analysis techniques and instrumenting malwares is a bit more complicated than usual softwares. Indeed their authors add some protections to allow the malware to detect if it is being executed in a controlled environment, i.e. : by measuring the execution time, by detecting breakpoints or by looking for specific register keys in the Microsoft OS. Also, malwares use techniques to loose the trace, i.e.: by creating a new process, or by injecting code in a running process. Thus, a good code instrumentation tool has to be able to tackle those anti-reversing techniques and ensure the malwares will not detect it.

Dinaburg et al [17] defined this property as transparency. A dynamic instrumentation tool must be invisible from the execution of the initial program. The implementation presented in section 5.1 is efficient and gives enough transparency to instrument malwares.

4 Pitfalls in disassembling

Disassembling is not an easy task as we have previously seen this problem is undecidable, and malware authors used of imagination to make the reverse-engineering process as much hard as possible. I try here to list common problems (non-exhaustive) and how the solution I implemented tackle them.

4.1 Common disassembling issues

Code and Data differentiation This issue has been well described in the literature [18], and it is known to be undecidable.

However I am not very concern about this issue. By using the recursive traversal algorithm previously detailed we follow the possible execution path. Thus ensuring that we decode only

[1] `https://en.wikipedia.org/wiki/Jerusalem_%28computer_virus%29`

[2] In x86 architecture the stack growths in reverse, starting at the last possible address.

instructions.

Indirect branch An *indirect branch* is a branch in the assembly code where the target is stored in a register, thus the value is only known at execution time and maybe not known during a classical analysis. It is worth noting that all targets from the example should be found in a *jump table* stored in the binary. However what is true for normal program compiled with a standard compiler, is not when dealing with malwares. Obfuscations tools may have removed or replaced *jump table* by something else in the code. An example of an indirect jump is given by figure 4.

Listing 4a presents a C code that when compiled create an indirect branch. At line 9 in 4b the instruction *jmp eax* will connect the execution flow to the address stored in the register *eax* during the run time.

Figure 4c shows the CFG reconstructed by a simple recursive traversal disassembler. The *jump eax* is the last instruction of the lefter block, and the disassembler is stuck at this point.

Figure 4d shows the same CFG but reconstructed by a recursive traversal disassembler with the help of an execution trace. The block color changed to pink exhibiting the fact that the contained instructions have been executed during a dynamic analysis. We observed here that there is a target jump for the *jmp eax* instruction (present in the second block from the top). Thus validating the minor improvement when an execution trace is used in conjunction of a static analysis. However, there are still missing possible targets for the indirect branch.

Figure 4e shows the same CFG reconstructed by a recursive traversal disassembler with the help of an execution trace and the help of the symbolic evaluation detailed in section 5.3.1. The green color exhibits the newly added block and at this point, showing the full CFG corresponding to the C/Assembly code from the example.

As mentioned earlier, indirect branch are a major problem when disassembling, this problem is partly solved by the use of dynamic analysis. And might be completely solved by much more complex computation discussed later.

System procedures System calls are used by software developer to interact with the environment such as interacting with input/output device, managing stored files ... . Hence, they are particular call to the API[1] provided by the OS on which the software is designed to be running. Thus they are OS dependent, and it is the main reason why an executable is not portable.

As their implementation are part of the OS, the code is not available in the studied binary file. Managing a *syscall*'s table is very time consuming, error prone, and must take care of the evolution of OS, thus it is not implemented by the disassembler in section 5. But it can be found in IDA pro.

When disassembling in a static way, and without such table, an assumption is made on the execution of the program. The function will return after the *call/syscall* instruction, which is the normal behavior.

By using an execution trace, a static disassembler is able to find the address of the effective return address.

4.2 Obfuscations

Previous disassembling issues were common to any disassembler. In addition, malwares use other techniques to make the reverse-engineering process harder. Collberg et al. wrote a book [19] which

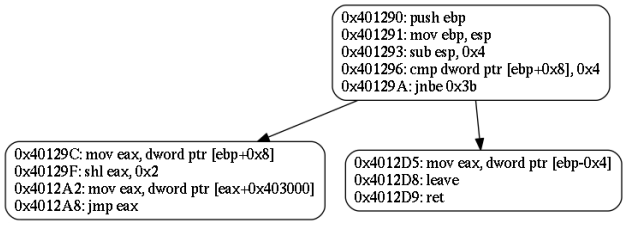
[1] Application Programming Interface

```
1  enum E { Zero, One,
2           Two, Three, Four
               };
3
4  int fun( E e ) {
5    int res;
6
7    switch( e ) {
8      case Zero  :
9        res = 0; break;
10     case One   :
11       res = 1; break;
12     case Two   :
13       res = 2; break;
14     case Three :
15       res = 3; break;
16     case Four  :
17       res = 4; break;
18   }
19   return res;
20 }
```

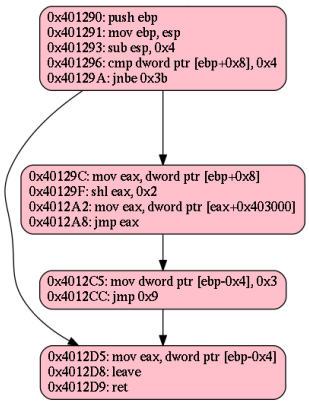
(a) *switch* statement do indirect jump using a jump table

```
1  0x401290: 55                    push ebp
2  0x401291: 89 e5                 mov ebp,esp
3  0x401293: 83 ec 04              sub esp,0x4
4  0x401296: 83 7d 08 04           cmp ebp+0x8],0x4
5  0x40129a: 77 39                 ja  0x4012d5
6  0x40129c: 8b 45 08              mov eax,[ebp+0x8]
7  0x40129f: c1 e0 02              shl eax,0x2
8  0x4012a2: 8b 80 00 30 40 00     mov eax,[eax+0x403000]
9  0x4012a8: ff e0                 jmp eax
10 0x4012aa: c7 45 fc 00 00 00 00  mov [ebp-0x4],0x0
11 0x4012b1: eb 22                 jmp 0x4012d5
12 0x4012b3: c7 45 fc 01 00 00 00  mov [ebp-0x4],0x1
13 0x4012ba: eb 19                 jmp 0x4012d5
14 0x4012bc: c7 45 fc 02 00 00 00  mov [ebp-0x4],0x2
15 0x4012c3: eb 10                 jmp 0x4012d5
16 0x4012c5: c7 45 fc 03 00 00 00  mov [ebp-0x4],0x3
17 0x4012cc: eb 07                 jmp 0x4012d5
18 0x4012ce: c7 45 fc 04 00 00 00  mov [ebp-0x4],0x4
19 0x4012d5: 8b 45 fc              mov eax,[ebp-0x4]
20 0x4012d8: c9                    leave
21 0x4012d9: c3                    ret
```

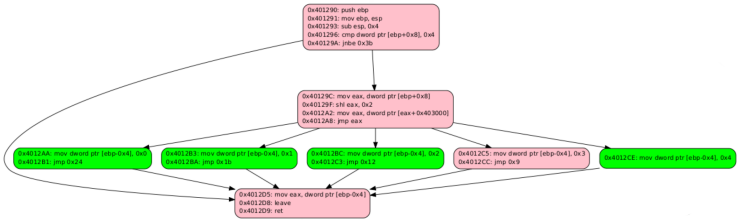
(b) Assembly of the beside C example with an indirect jump at line 9



(c) CFG extracted with a recursive transversal



(d) CFG extracted with a recursive transversal and an execution trace



(e) CFG extracted with a recursive transversal and an execution trace and using symbolic evaluation technique

Figure 4: Example of an indirect jump

well described a lot of them. As the list is quite long, I will here describe some of them and explain how to tackle them.

It must be noted that obfuscation's techniques are not only used for bad purposes. They are also used by legitimate companies to protect their intellectual property. Indeed, when the business product of an enterprise is based on the sale of a software, they don't want that hackers disassemble it to get a pro-version, or retrieve cryptographic keys, or make a similar software based on it.

Instruction overlapping This obfuscation occurs when at least one byte is used in more than one instruction. An example is shown in listing 5a for the disassembly of a linear sweep disassembler and 5b for the disassembly of a recursive traversal one. At line 2, there is the instruction *jmp +1* at address $0\mathbf{x}401027$, its effect will be to connect the execution flow at $0\mathbf{x}401027 + 1 = 0\mathbf{x}401028$. Thus the next executed instruction will be *inc ebx*. So, *jmp +1* and *inc ebx* shares one byte at address $0\mathbf{x}401028$, they are two overlapping instructions.

```
1 0x401022: bb 00 00 00 00 mov ebx,0x0
2 0x401027: eb ff          jmp +1
3 0x401029: c3             ret
4 0x40102a: 83 c3 42       add ebx,0x42
5 0x40102d: c3             ret
```

(a) Disassembled by a linear sweep disassembler

```
1 0x401022: bb 00 00 00 00 mov ebx, 0x0
2 0x401027: eb ff          jmp +1
3 0x401028: ff c3          inc ebx
4 0x40102a: 83 c3 42       add ebx,0x42
5 0x40102d: c3             ret
```

(b) Disassembled by a recursive traversal disassembler

Figure 5: Example of overlapping instruction

Comparing the two disassembly in figure 5, it is worth noticing that a linear sweep disassembler is wrong as it sees the instruction at line 4 as unreachable while the recursive traversal find the good execution path. Linear sweep disassembler are lost by this obfuscation.

To go further Jämthagen et al [20] shows the possibility to build a code with a complete hidden execution path. This is done by playing with the flexibility of the *nop*[1] instruction which do nothing but can have several bytes as operands. Meaning that a linear sweep disassembler will see a sequence of *nop* instructions while the processor or a recursive traversal disassembler will decode the right sequence of instructions.

By using a recursive traversal algorithm, a disassembler will follow jumping instructions thus following the overlapping instructions.

call* without *ret*, or *ret* without *call The *call* instruction is used to enter in a function, and the *ret* instruction corresponds to the return statement. Normal behavior is when a *call* instruction is found, the next one will be in a function and then until a *ret* instruction. However this is not always true, especially with malwares. Sometimes, *call* are used as a jump instruction, and the pushed program counter on top of the stack may be modified by further execution. Or a sequence like : *call* 5 ; *pop eax* will allow a program to access the value of the program counter and store it in *eax*. The inverse is also true, a *ret* can be used as a jump like in the sequence: *mov eax*, $0x42$; *push eax* ;*ret*. Those hijack the usage of call/ret by not entering in/leaving a function.

[1] no-operation instruction

To tackle this issue, a static disassembler must simulate the operation made on a stack. And obviously a dynamic analysis has no problem to follow the execution flow.

Dead Code This obfuscation refers to the addition of unreachable code or dead code. As shown by figure 6, a linear sweep disassembler will disassemble the listing of the figure 6a while the figure 6b shows what a recursive disassembler does. Thus the instruction at line 3 in 6a will never be executed and so is considered as dead code. In this example if the disassembler returns this dead code, an analysis acting at a higher level of abstraction for register *eax* (i.e.: by computing the possible values for *eax*) will return 0 (*sub* is the subtraction), but with dead code elimination this same analysis will return 42 which is the true value.

```
1 0x401022: b8 42 00 00 00 mov eax,0x42
2 0x401027: eb 03          jmp +5
3 0x401029: 83 e8 42       sub eax,0x42
4 0x40102c: c3             ret
```

(a) Disassembled by a linear sweep disassembler

```
1 0x401022: b8 42 00 00 00 mov eax, 0x42
2 0x401027: eb 03          jmp 5
3 0x40102c: c3             ret
```

(b) Disassembled by a recursive traversal disassembler

Figure 6: Example of dead code

This obfuscation is also inefficient against a recursive traversal disassembler which will see that dead code as data.

Opaque Predicate An opaque predicate is related to a test condition which always return the same value for any execution of the program. A very trivial example one is shown by the figure 7. To summarize this assembly code the register *eax* is equal to $2 + 2 = 4$ at line 4. And trivially at line 5, the test will always be *true* and only one branch of this jump can be taken (*je* is for jump if equal), and the *sub* will never be executed.

Same as dead code, opaque predicate will bring automatic analysis of the possible values for register *eax* to a wrong result.

The mentioned example is very simple but well illustrates the concept. Even if a basic static analysis (i.e.: constant propagation [11]) can tackle the example, there is much more complex opaque predicate. One can be defined using the little theorem of Fermat. It says that for every positive integer i: $i^{17} = i \mod 17$. This can easily be seen by a human analyst aware of such theorem, however this is complicated for a static automatic process.

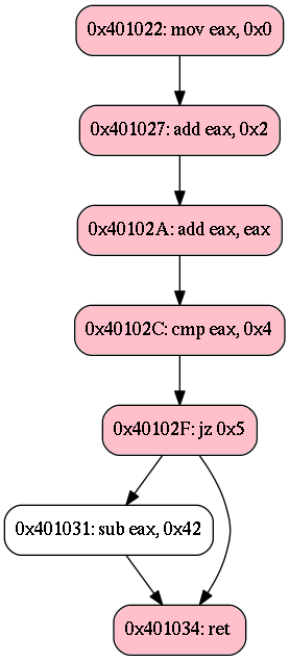
The trace gives us the good execution path (in pink), but the further static analysis will add the wrong one. In figure 7b, the static analysis added the *white* node to the *pink* ones extracted from an execution trace.

To gain efficiency I will leave my disassembler falling into this pitfalls. Thus the human analyst or the morphological analysis will need to challenge it. The loss of performances would come from a solver to which the static disassembler would need to interrogate for each conditional jump.

Control flow flattening This obfuscation aims to flatten the CFG. An example is shown by figure 8 where the two assembly from listing 8a8c are semantically equivalent. However one uses a *jmp eax* as a central point to redirect the flow to the next instruction whose make the CFGs on figure 8d the flatten version of figure 8b.

```
1  0x401022: b8 00 00 00 00 mov eax,0x0
2  0x401027: 83 c0 02       add eax,0x2
3  0x40102a: 01 c0          add eax,eax
4  0x40102c: 83 f8 04       cmp eax,0x4
5  0x40102f: 74 03          je  +5
6  0x401031: 83 e8 42       sub eax,0x42
7  0x401034: c3             ret
```

(a) disassembly with an opaque predicate at line 4-5



(b) CFG corresponding of the code besides

Figure 7: Example of opaque predicate

This obfuscation can be problematic for the morphological detection procedure if the reference is not flattened in the database and the same malware is newly propagated flattened, it will not be possible to detect the malware.

The flatten version contains an indirect branch which bring the need of an execution trace to properly recover the CFG (pink node).

4.3 Self-modifying code

As explained in previous section, the separation between code and data is not very clear in x86 PE executable. This allows a program to modify its own code in the memory at run-time.

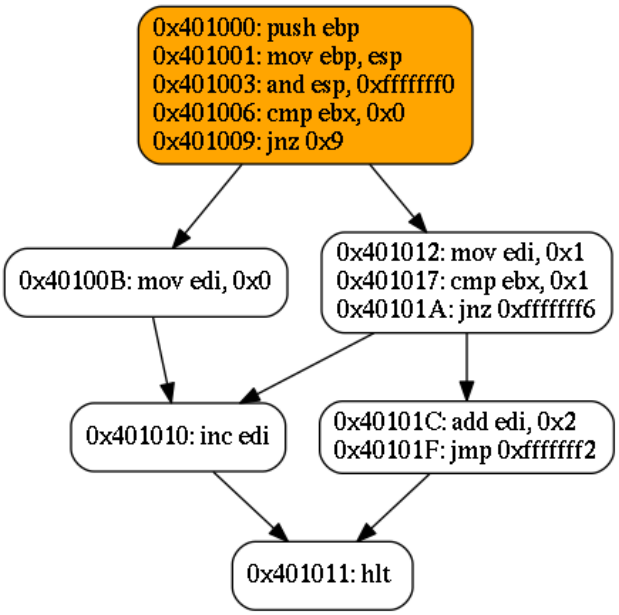
Then any program that creates new instruction above its own initial instruction or somewhere else in the memory and then branch the control flow to it, is considered to be self-modifying. Listing 9a shows a small example. The instruction at line 3 writes the content of the register *ebx* at the address pointed by *eax*, so the code become as presented by listing 9c.

First analysis of listing 9a would suggest an infinite loop with the *jmp* that connects the execution flow on it self. But the instruction at line 3 modifies the address pointed by the register *eax* with the content of register *ebx*, thus modifying the address $\mathbf{0x}401012$ with the value $\mathbf{0x}f4$ making the program terminates.

Self-modifying code can also be seen as an obfuscation technique. Also it is not only used for bad purposes. For example, the booloader of windows uses it to start the entire operating system.

```
1  0x401006: 83 fb 00        cmp ebx,0x0
2  0x401009: 75 07           jne 0x401012
3  0x40100b: bf 00 00 00 00  mov edi,0x0
4  0x401010: 47              inc edi
5  0x401011: f4              hlt
6  0x401012: bf 01 00 00 00  mov edi,0x1
7  0x401017: 83 fb 01        cmp ebx,0x1
8  0x40101a: 75 f4           jne 0x401010
9  0x40101c: 83 c7 02        add edi,0x2
10 0x40101f: eb f0           jmp 0x401011
```

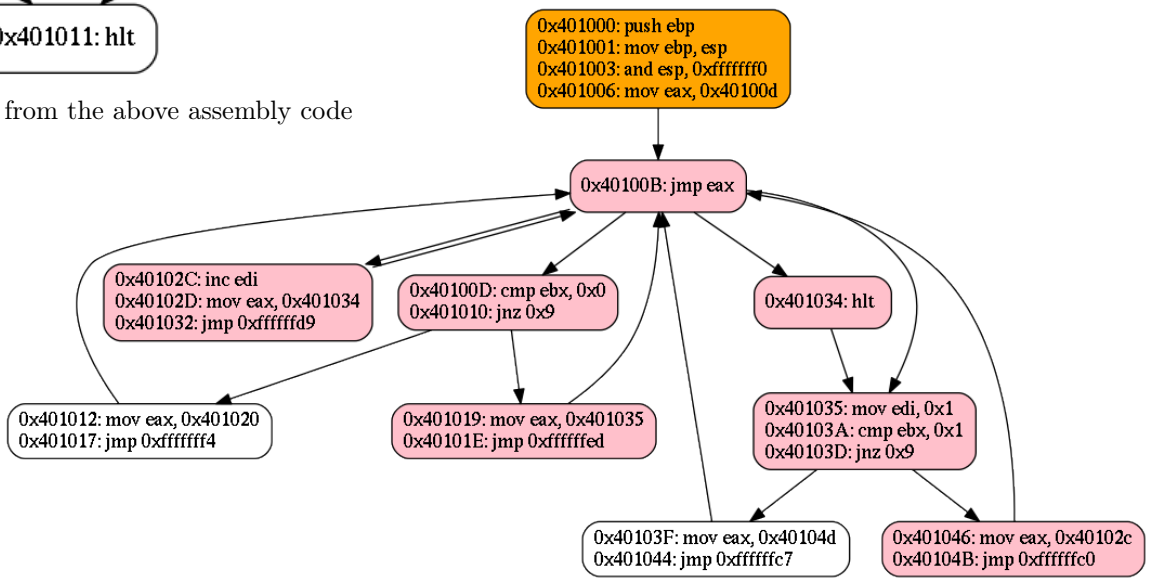
(a) Sample of an assembly that modify *edi* depending on *ebx*



(b) CFG from the above assembly code

```
1  0x40100b: ff e0           jmp eax
2  0x40100d: 83 fb 00        cmp ebx,0x0
3  0x401010: 75 07           jne 0x401019
4  0x401012: b8 20 10 40 00  mov eax,0x401020
5  0x401017: eb f2           jmp 0x40100b
6  0x401019: b8 35 10 40 00  mov eax,0x401035
7  0x40101e: eb eb           jmp 0x40100b
8  0x401020: bf 00 00 00 00  mov edi,0x0
9  0x401025: b8 2c 10 40 00  mov eax,0x40102c
10 0x40102a: eb df           jmp 0x40100b
11 0x40102c: 47              inc edi
12 0x40102d: b8 34 10 40 00  mov eax,0x401034
13 0x401032: eb d7           jmp 0x40100b
14 0x401034: f4              hlt
15 0x401035: bf 01 00 00 00  mov edi,0x1
16 0x40103a: 83 fb 01        cmp ebx,0x1
17 0x40103d: 75 07           jne 0x401046
18 0x40103f: b8 4d 10 40 00  mov eax,0x40104d
19 0x401044: eb c5           jmp 0x40100b
20 0x401046: b8 2c 10 40 00  mov eax,0x40102c
21 0x40104b: eb be           jmp 0x40100b
22 0x40104d: 83 c7 02        add edi,0x2
23 0x401050: b8 34 10 40 00  mov eax,0x401034
24 0x401055:eb b4            jmp 0x40100b
```

(c) Same sample as besides but after an CFG flattening phase

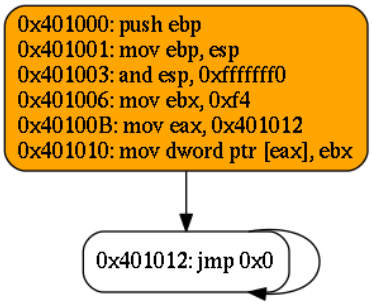


(d) Flatten version of the CFG in 8b and corresponding to assembly 8c

Figure 8: Flattening obfuscation example

```
1  0x401006: bb f4 00 00 00 mov ebx,0xf4
2  0x40100b: b8 12 10 40 00 mov eax,0x401012
3  0x401010: 89 18          mov [eax],ebx
4  0x401012: eb fe          jmp 0x401012
```

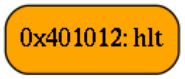
(a) the initial code



(b) CFG corresponding to the wave 0 (initial code)

```
1  0x401006: bb f4 00 00 00 mov ebx,0xf4
2  0x40100b: b8 12 10 40 00 mov eax,0x401012
3  0x401010: 89 18          mov [eax],ebx
4  0x401012: f4             hlt
```

(c) the code after self-modification



(d) CFG corresponding to the wave 1 (modified code)

Figure 9: Example of self-modifying code

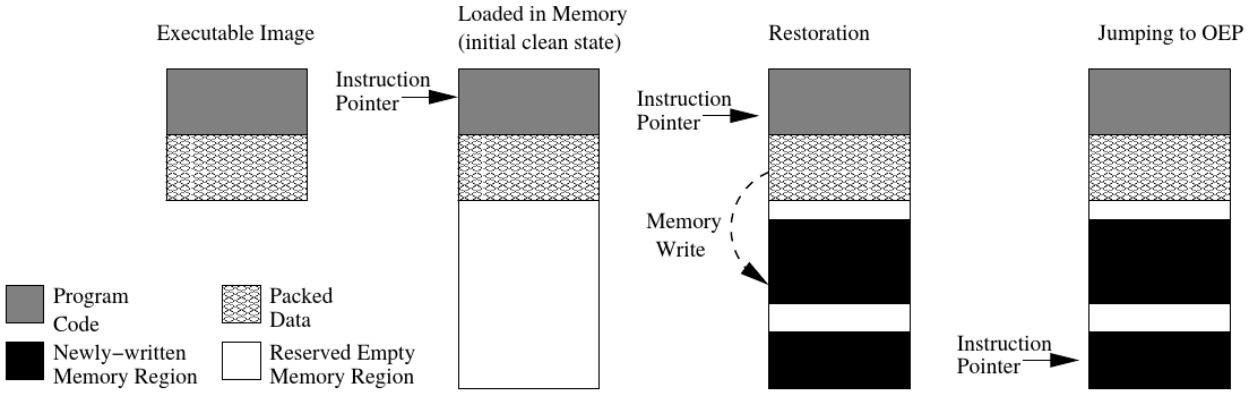


Figure 10: Overview of packed executable principle

And, this is used by JIT[1] compiler such as lit the JIT from LLVM[2].

To obfuscate their code with *self-modification*, malwares authors use an external tool called *packer*. Figure 10 illustrates the general process of a packed executable. The packed program is first loaded into memory like any other program, then the unpacking routine is executed. It allocates memory and copies the unpacked code in it and finally connects the execution to the entry-point of the unpacked code. The final unpacked code is referred as the payload which will contain the malicious code.

There is different techniques for packing, the more common are compression, or encryption. Hence the original binary contains a portion of data that is compressed or encrypted and the unpacking routine has in charge of decompressing or decrypting it in the allocated memory.

Some packers use several unpacking steps before actually extracting the payload. In an unpub-

[1] Just In Time compiler - compiling the code on the fly

[2] LLVM is a compiler suite coming to replace GNU/GCC `http://www.llvm.org`

lished article Bonfante et al.[21] show they are able to find packers with up to 635 unpacking steps before reaching the payload and that 53% of their testing corpus contains 2 unpacking steps.

Because of all of this steps and the fact that the payload is not in clear assembly in the initial binary, every static disassemblers are not able to retrieve the payload. Indeed they will only disassemble the first unpacking routine. Thus, *unpackers* use the previously mentioned dynamic analysis to trace packed code with the goal to retrieve the original code from this execution trace.

The notion of waves Guizani et al [22] first introduced the concept of waves to formalize all the unpacking steps. The idea is that each wave contains only non-self-modifying code ; and contains all the instructions present in memory when switching wave. It contains non-self-modifying code, because an instruction can not be written in memory and executed in the same wave. Hence each wave starts when the control flow is connected to a newly written instruction. The construction of the set of waves is a monotonic sequence, the control flow never go back to a previous wave even if it is linked to an already executed instruction in the wave $n - x$, the execution of the instruction is now considered to be in the wave n. The execution of an instruction is now defined by its address and its wave. Meaning if a disassembler would build the full CFG of the full execution of all wave, a same instruction at a same address could be present several times if it is executed in several waves.

As a wave is a non-self-modifying part of the packed program, it can be independently analyzed by a disassembler if the unpacker provides the list of bytes in memory (a snapshot) at some point of the execution of the wave, an entry-point (typically the first instruction of the wave) and maybe an execution trace to improve results.

On the example from figure 9, there is two waves, one is in 9a where the entry-point is **0x**401006 and second is in 9c where the entry-point is **0x**401012. Their corresponding CFG are presented by 9b and 9d.

Reynaud [1] showed that it is more pertinent for an unpacker to take a snapshot at the start point of the wave. Indeed by taking it at the very beginning he ensures all the instruction needed to execute the wave are present in memory. A packer author familiar with usual dynamic analyze technique could overwrite the executed code before leaving the current wave, making impossible the possibility to retrieve it.

Thierry [3] defined the notion of *perfect CFG* to schematize a wavy self-modifying program. This particular CFG built at the wave level illustrates the passage from a wave to another by taking care of the execution level and the written level.

For a malware detector, this formalization in waves is based on the assumption that a wave in the set contains the whole code of the payload. And as pointed by Calvet [2], for a majority of packers the malicious code is present in the last wave; or at least the payload is fully present in a wave. So executing the malware ones in a controlled environment to reach the payload seems enough. And the reconstruction of the CFG of that payload is done in the same way as any non-self-modifying program.

5 CoDisAsm: a concatic disassembler

Previous sections described the frame needed for the implementation and some challenging obfuscations. This master thesis topic aims to apply and experiments the presented theory here and more detailed by Reynaud [1], Calvet [2] and Thierry [3].

The figure 11 presents the global schema of the whole project.

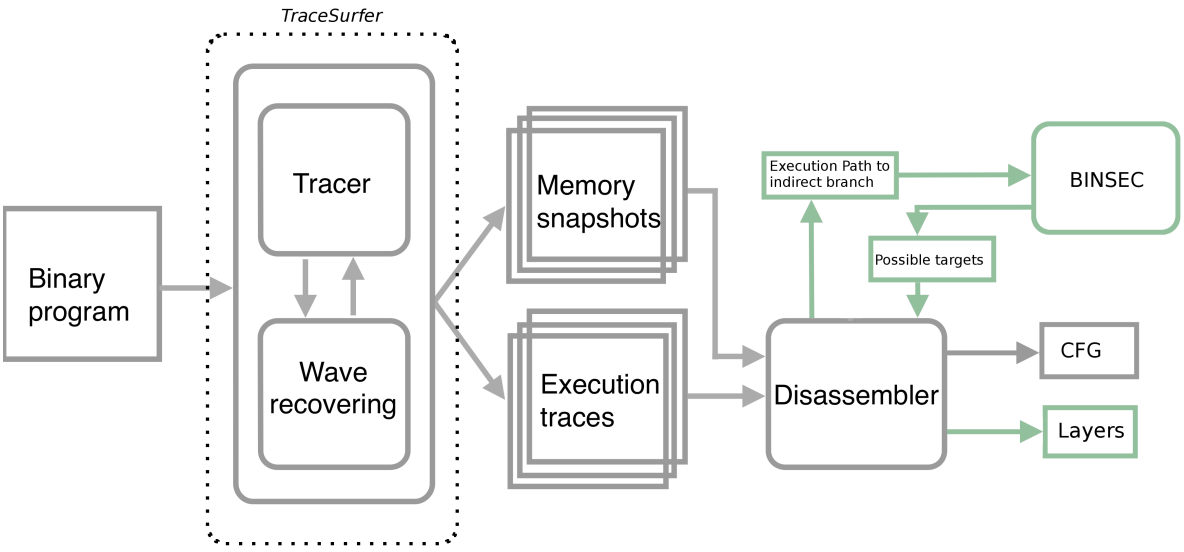


Figure 11: Schema of CoDisAsm

CoDisAsm stands for *Concatic Disassembler* for CONCrete path execution and stATIC disassembly. It joins the use of a dynamic instrumentation tool (TraceSurfer) for the concrete path execution and unpacking features with a static disassembler to recover the CFG. Hence this section presents my implementation of the static disassembler in conjunction with the improvements I managed to add in the tracer developed by previous work.

To complete the project I implemented two extensions presented in 5.3. First one is the interaction with the BINSEC[1] platform, as this master thesis is also part of the corresponding ANR[2] project. The second extension is the implementation of a concept previously formalize by Thierry[3]. Those two additions are left in extension at the moment as they require further investigations.

5.1 TraceSurfer

The project *TraceSurfer* has been started by Reynaud[1] and improved by its successor Calvet[2] and Thierry[3]. Its name is based on the concept of dynamic instrumentation (*trace*) and the self-modifying code representation called *wave* (*surfer*) introduced in section 4.3.

It aims to unpack malwares by executing it in a Virtual Machine (VM). For each analyzed malware, the VM is reseted to a fresh installation of the target Operating System ; here Microsoft Windows. It is important to use a clean install to ensure that malwares do not interfere with each other.

Then the machine code is instrumented by Pintool[14]. It exists several tools to instrument code (i.e.: DynamoRIO[3], Valgrind [4], DynInst [5]). However a study demonstrated Pintool is relatively

[1] http://binsec.gforge.inria.fr
[2] Agence National pour la Recherche
[3] http://www.dynamorio.org/
[4] http://valgrind.org/
[5] http://www.dyninst.org/

transparent from a semantical point of view [23] ; meaning malwares will not be able to detect it by looking on the respect of the instructions' semantic. Also it is easy to tune it, and its been designed by Intel itself as they use it to simulate new instructions when building chips. Moreover, Pintool handles multi-threads and can follow forked processes out-of-the-box.

Furthermore, malwares incorporate a lot of anti-reverse-engineering features. Those one are designed to detect if they are executed in a controlled environment such as a Virtual Machine. Thus Pintool has been previously tweaked to integrate countermeasure features to fool malwares, Ferrie [24] wrote a survey on such techniques. Just to illustrate some countermeasures that are implemented in TraceSurfer (non-exhaustive):

- Timing constraint :

  > Malwares check the execution time between two points in the program

  > Clock system calls are detecting by the tracer and a reduced value is returned to the malware depending on the tracing speed.

- Process injection :

  > Malwares create a new process, or use a already running process they do not own and inject code in it

  > Fork system call detection to follow the execution in the new/infected process

- Hardware breakpoint detection :

  > Malwares look at specific registers in the processor to see if the *TrapFlag*, or some *debug registers* are set to an address

  > Detection of access to those *specific registers* or *TrapFlag* and a fake value is returned to the malware

- Software breakpoint detection :

  > Identical but breakpoints are set on the OS

  > Not used by TraceSurfer

The initial output from TracerSurfer was an execution trace file in a homemade binary format representing the full execution of the malware. It contains for each instruction:

- Wave of execution
- Address in memory
- Opcode
- Eery explicit and implicit impact on the system when executing it. As a remember impact are accesses and modifications of registers/memory cells implied in the execution of the instruction (i.e.: *push ebp* explicitly pushes the register *ebp* on top of the stack, and implicitly decrements *esp* by 4).

Also for each new waves, TraceSurfer takes a snapshot of the memory and reconstruct a standalone executable binary file (PE format) starting at wave 1 ; wave 0 is the original executable.

Thus a static disassembler can, for each waves, take the binary corresponding to the desired wave, the execution trace file then disassemble the wave and reconstruct the CFG.

Improvements Experiments showed very poor performance ; the tracing time was very long and the disassembling of waves too. It was necessary to parse the whole file seeking for the good wave and skipping non-needed information.

Then I suggested to add minor changes. First modifying Pintool to stop recording the impact of instructions on the system ; leaving only the wave, addresses and opcodes. Those removed information were not necessary for our static disassembler and performance strongly increased. For example, in previous version the packer Armadillo took days to unpack and now it takes only few hours to finish in the new version (this packer was a commercial one and is well-known to be hard to reverse-engineer).

My second proposition was to split the execution trace file per wave. Indeed, the disassembler only disassemble one wave at a time, so it does not need the whole execution trace especially when a packer has a lot of waves.

In previous version, TraceSurfer output the whole set of instructions really executed. Meaning that when the code is a loop, the execution of the loop was flatten into the trace file, or when an instruction was prefixes by *rep* the instruction was in the file the number of time the processor executed it. My last suggestion to help the disassembler was to modify the tracer to make it return the partial CFG it actually executed. In examples from previous section 4, the *pink* nodes and the edges linking them was extracted by TraceSurfer ; the disassembler just had to fill the undisclosed paths.

With this three minor modifications implemented by the engineer from CARTE crew, TraceSurfer has less work to do and returns smaller files for the disassembler reducing the tracing time and the whole process time.

5.2 The disassembler

The second block of the project is the disassembler. It aims to statically disassemble the remaining control flow not taken during the dynamic analysis from the tracer. And obviously it is also able to disassemble a binary file without a trace file, but with less accuracy when dealing with malwares.

As input, CoDisAsm needs a binary corresponding to the wave which will be disassembled and an execution trace file. It first follows the trace as we are sure that the instruction contained in it are reachable, so are part of the final CFG.

It uses the recursive traversal approach describes before to look over the partial CFG extracted by TraceSurfer. Then each time a conditional jump is found and one branch is undisclosed it statically explores it by depth first until a stop point is found. A stop point can be of different nature:

- The instruction is already in the CFG
- The instruction is *hlt*, *ret* or *int*
- The instruction is a System Call
- The instruction is invalid

If the instruction is already in the CFG, it means the exploration of this control path is complete. If it reaches a *ret* or *int*, it can not statically know what would be the next instruction. Indeed this disassembler does not simulate the stack movements, so it is not able to tell what would be the popped address when a *ret* arises. Similarly for *int*, the disassembler does not simulate the

specific *interruption* registers. Those two choices are imposed by the constraint of efficiency. If the instruction is a system call, the target address will not be part of the binary and the disassembler can not continue to disassemble at an unknown address. However I choose to keep the assumption that the program will return at the next instruction linearly found. This is legitimate by the fact this is the *normal* behavior. Hence the disassembler will not take a wrong path in most cases, which is worthwhile for the morphological detection of malwares. Also if the instruction is not valid, it trivially stops. This could happen if, for example, the static disassembler follows the wrong branch of an *opaque predicate* (cf section 4.2) resulting in some places in memory with junk bytes.

The algorithm presented in section 3.2 is recursive. As a matter of performance, I modified it to make it iterative. Indeed, when dealing with large binary executable the recursive function call builds a stack frame to store arguments, return address, backup of the stack pointer and local variables which brought me a stack overhead usage.

Thus my new algorithm is presented in 5.1. To make it iterative and continue to do the recursive traversal, each new possible target address needs to be stored (see *delayed* in the algorithm) to discover conditional branch later while continuing until a stop point. Then when the latter is reached, it starts again from a new address found in the set of unexplored addresses.

Algorithm 5.1: Recursive traversal in iterative fashion

```
Input  : The program P to disassemble and an execution trace T
Output: D the perfect disassembly of P ; a set of (address, instruction)
  function DISASSEMBLE(P, T)
      D ← ∅
      foreach instruction I in the trace do
          foreach addr ∈ SUCCESSOR(I) do
              if (addr, *) ∉ D then
                  D ← D ∪ EXPLORE(P, addr, D)
      return D
  end function
  function EXPLORE(P, addr, D)
      delayed ← {addr}
      while ¬EMPTY(delayed) do
          addr ← POP(delayed)
          while addr is valid ∧(addr, ∗) ∉ D do
              I ← DECODE(addr, P)
              D ← D ∪ (addr, I)
              if ¬I is a stop point then
                  if I is a sequential instruction or is a conditional jump then
                      addr ← addr + SIZE(I)
                  if I is a conditional/unconditional jump or is a call then
                      delayed ← delayed ∪ {COMPUTETARGET(I)}
      return D
  end function
```

As a matter of simplicity, I removed wave information from the algorithm as the disassembler is statically disassembling only one wave at a time. Nevertheless to fulfill conditions of the methods, the *addr* are a pair composed of $(wave, addr)$, this is useful to disassemble many waves ; available as an extension of the disassembler. The multiple wave extension, just repeat the algorithm 5.1 from the wave 0 to the wished one. The connection between waves is done by looking for the correspondence between the last instruction of wave n from the corresponding execution trace file and the first instruction of wave $n+1$ also from the corresponding execution trace file.

5.3 Extensions of CoDisAsm

5.3.1 Interaction with BINSEC Platform

The BINSEC ANR project in which this master thesis is included aims to build a powerful binary analysis software. It takes its origin in the automatic test input generation where the main problematic is to compute the set of possible input values to pass as argument to a program in order to recover every possible execution paths. Even if malwares do not need arguments to run, our objectives are the same disassembling and reconstructing the complete CFG from a binary. Dynamic Symbolic Evaluation[25] (DSE, also called concolic execution) emerged from this research area. This describes a method to disassemble and reconstruct a binary file from multiple executions of the program with the help of Symbolic Evaluation (SE). The latter is a complex computation technique, and it helps computing the different input values.

Malwares may contain indirect jump not associated with a jump table (section 4). Determining the possible values and discovering new paths to add in the CFG would make the morphological analysis more accurate. In that way, SE can help to determine all the possible values for the register *eax* at the program point of the jump.

The present study is a conjoint work between Robin David currently in a PhD position at the CEA[1] working full time on BINSEC and myself.

First I will introduce the concept of Symbolic Evaluation before explaining my implementation in CoDisAsm and the problem we faced.

Symbolic evaluation The approach of the symbolic evaluation is to build a symbolic value for each interesting register/memory address at each point of an execution path [26], and other element are kept as concrete value.

A symbolic value is a complex formula representing the evolution of the symbolic value along an execution path. A concrete value is the *static* value of an element, similarly as a constant at a precise program point. The formula is built by keeping track of the control flow and the evolution of each symbolic values over this control flow. Then the formula is used to feed a SMT[2] solver which returns values that satisfy this formula. An example is given in the following table 3.

The example of table 3 is extracted from an inspired code similar to the switch in figure 4a. The goal here is to retrieve every possible values for register *eax* in the last instruction *jmp eax* in order to find the possible target of this unconditional jump. The *Input Symbolics* column represents the element we choose to keep as symbolic. Hence each symbolic values have their formula represented all the way in the execution path.

[1] Commissariat à l'Energie Atomique
[2] Satisfiability Modulo Theories

Instructions	Input Symbolics	Input Concretes	Formulas
push ebp	ebp		$\emptyset$
mov $ebp,\ esp$	$ebp,\ esp$		$ebp = esp$
mov $ecx,\ 42$	$ebp,\ esp$	$ecx = 42$	$ebp = esp$
cmp $[ebp+8],\ 4$	$ebp,\ esp,\ [ebp+8]$	$ecx = 42$	$ebp = esp$
ja $0\mathbf{x}8048494$	$ebp,\ esp,\ [ebp+8]$	$ecx = 42$	$ebp = esp$ (and path constraint $[ebp+8] < 8$)
sub $ecx,\ 40$	$ebp,\ esp,\ [ebp+8]$	$ecx = 2$	$ebp = esp$
mov $eax,\ [ebp+8]$	$ebp,\ esp,\ [ebp+8]$	$ecx = 2$	$ebp = esp,\ eax = [ebp+8]$
shl $eax,\ 2$	$ebp,\ esp,\ [ebp+8]$	$ecx = 2$	$ebp = esp,\ eax = ([ebp+8] << 2)$
add $eax,\ 0\mathbf{x}8048570$	$ebp,\ esp,\ [ebp+8]$	$ecx = 2$	$ebp = esp,$ $eax = ([ebp+8] << 2) + 0\mathbf{x}8048570$
sub $eax,\ ecx$	$ebp,\ esp,\ [ebp+8]$	$ecx = 2$	$ebp = esp,$ $eax = ([ebp+8] << 2) + 0\mathbf{x}8048570 - 2$
mov $eax,\ [eax]$	$ebp,\ esp,\ [ebp+8],$ $[[ebp+8] << 2 + 0\mathbf{x}8048570 - ecx]$	$ecx = 2$	$ebp = esp,$ $eax = [([ebp+8] << 2) + 0\mathbf{x}8048570 - 2]$
jmp eax			

Table 3: Trivial example of a symbolic evaluation

In the example 3, *ecx* introduced at line 3 receives the specific value 42. Then it is reduced by 40 and the concrete value is updated, and finally introduced in the formula for *eax*. Concrete values can be assimilated to value propagation from compiler [11]. Hence the concrete value will be replaced by its value in the formula, making it smaller and speeding up the resolution by the SMT solver.

When using symbolic evaluation on a program, a choice must be made on which element should be kept concrete or symbolic. In the example from 3, a reasonable strategy could have chosen to make *esp* and *ebp* concrete thus fixing their values. On each formula, a strategy must be constructed to decide what is concrete and what is symbolic. This question is still an active topic of research and a major topic in the PhD thesis of Robin David. I can just give the guesses we had with him.

- Obviously do not use a concrete value for your goal, *eax* in the previous example (table 3)
- It seems that using concrete values for addresses in load/store helps the solver to not returning stupid values, i.e.: returning 0**x**0000 for a jump
- In the example of figure 4d, using a concrete value for *ebp* and *esp* speed up the solver

Taking back the example of figure 4d, symbolic evaluation is able to retrieve the 4 missing targets as shown by figure 4e. I will not present the true symbolic value that allowed us to retrieve those values for register *eax* as it is quite huge (more than 700 lines).

Implementation In reverse-engineering, SE is usually used in conjunction of a dynamic analysis. The latter is run multiple times for each input values the SMT solver found, and the CFG is reconstructed from those multiple executions containing a feasible path. CoDisAsm will only use the capacity of the symbolic evaluator included in BINSEC platform.

The communication between CoDisAsm and BINSEC platform use a socket channel where the format of the message is defined using protobuf [1].

Once the static disassembler is done, it checks if there is indirect branch in the recovered CFG. For each one, it sends a request to the symbolic evaluator which sends back a set of possible values for the register in the indirect branch. For each of this possible values, CoDisAsm restarts the static disassembly by considering the value as an address of a successor of the indirect branch. A SMT solver used by the symbolic evaluator computes an over-approximation of the possible values. This potentially leads to wrong control flow path.

The symbolic evaluator then needs a trace. But to increase the accuracy of the solver, CoDisAsm must provide the maximum information of the state of the machine for each executed instruction. This includes the explicit and implicit impacts of an instruction (see section 3.3 or 5.1). This extra information helps the evaluator and the solver to reduce the formula or the frame of possible values. It also helps to find the best strategy to determine what element is concrete or symbolic.

Also the trace needed by this evaluator is a bit different. It corresponds to the sequence of instructions from the root node to the indirect jump and following an unique execution path. For node discovered by TraceSurfer CoDisAsm adds the extra information about the state of register-s/memory cells, and for node statically discovered it do not add any extra information.

This extension of CoDisAsm has 2 major drawbacks:

[1] Protocol Buffers, Google's data interchange format – https://github.com/google/protobuf/

- Performance are very poor ; due to the size of the execution trace and the resolution time by the SMT solver
- This has been tested only in the one case illustrated by figure 4e, and they are a lot to do left for future work

Discussion The presented extension uses some concept that need to be more develop. Only one small test has been made which after long hours of tweaking we managed to pass. This brought us to the need to define a strategy on what concretize and symbolize. This has to be formalize to be useful for generic analyses.

Also, some position needs to be clarify in the future. For example if an indirect jump is in a loop or after a loop, should CoDisAsm incorporates all iterations of the loop (if available), or only one and which one. If a possible value discovered by the solver creates a back edge in the CFG, suggesting a loop, should CoDisAsm request for new targets with the added edge in the path. At the present time, we arbitrary decided to only send the last iteration of the loop. Similarly for function call, if the indirect jump is in a function called several times, should CoDisAsm interrogates the solver the same amount of time with the different call paths.

Those remaining question I asked are still open and will certainly be developed in Robin David's thesis.

5.3.2 Overlapping instructions and layers

The section 4.2 presented the overlapping instruction as an obfuscation. Thierry [3] suggested that this obfuscation can be used to identify malwares. As an extension to CoDisAsm I implemented the formalization he proposed which can then be used by the malware detector to identify the malicious code.

To formalize this obfuscation, he places instructions into layers. The property of a layer is that it does not contain overlapping instructions. When two instructions overlap, the second (ordered by the control flow) is placed in an upper layer. Table 4 presents what a recursive traversal is able to build.

Addresses	0x401027	0x401028	0x401029	0x40102*a*	0x40102*b*	0x40102*c*	0x40102*d*
Bytes	eb	ff	c3	83	c3	42	c3
Layer 1	jmp +1			add ebx,0x42			ret
Layer 2		inc ebx					

Table 4: Layers of overlapping instructions from a recursive traversal disassembler

From this formalization some metrics identifying malwares can be extracted:

- The number of layers – indicating the complexity
- The number of addresses used by overlapping instructions – indicating the complexity
- The number of edges in the CFG crossing layers – indicating the frequency of usage of this obfuscation technique

Unfortunately I could evaluate those metrics on a real corpus of malwares du to the lack of time. Also, this must be added to the morphological analysis tool what has not be done from that time. Both of them are left for future work.

Implementation The algorithm 5.2 presents the implementation of the coherent cutting from Thierry[3] and presented in the previous example in table 4.

At first every instructions are part of the first layer. Then they are iteratively moved from layer n to layer $n+1$ when overlapping an instruction of the layer n. Thus making a new layer which is then checked for overlapping instruction and so one.

Algorithm 5.2: Layers creation

```
Input  : The CFG recovered from the previous algorithm
Output: The instructions arranged in the set of layers L
  function EXPLORE(CFG)
      // First, all instruction are in the first layer
      currentLayer ← ∅
      foreach instruction I ∈ CFG do
          currentLayer ← I
      L ← {currentLayer}
      // Then we iterate over each new layer to put overlapping instructions into an upper layer
      // Note: instructions are ordered by control flow
      layerCreated ← True
      while layerCreated do
          layerCreated ← False
          nextLayer ← ∅
          foreach instruction I ∈ currentLayer do
              if I overlaps an other instruction in currentLayer then
                  nextLayer ← nextLayer ∪ I
                  currentLayer ← currentLayer \ I
                  layerCreated ← True
          L ← L ∪ {nextLayer}
          currentLayer ← nextLayer
  end function
```

Statistic At that time the only conclusion we can have is that on a corpus of 500 malwares, 70% of the sample use at least one instruction overlapping as depicted by figure 12

6 Evaluation

Building a disassembler is not trivial as Paleari et al [12] pointed out every x86 disassemblers are incorrect due to the complexity of the instruction set. However I will try in this section to validate the presented approach implemented in the previous section.

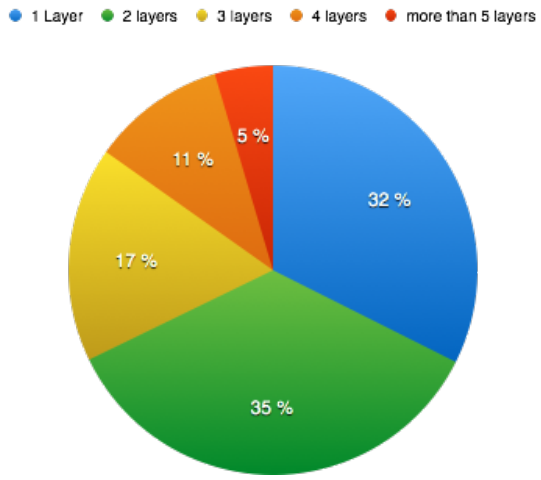


Figure 12: Number of layers extracted from 500 malwares

For the sake of simplicity I will drop the extensions and focus only on the main parts of the project (i) the disassembler (ii) the tracer (iii) their combination. And also because the two presented extensions (layers, BINSEC) are designed for future purposes.

6.1 Evaluation of the tracer

The tracer has been evaluated in every possible angle in the three thesis from Reynaud, Calvet and Thierry. Nonetheless we need to evaluate the improvements presented in previous section.

The goal of the tracer is to unpack the malicious code in many non-self-modifying code waves. To do so, it must be transparent enough to allow the execution flow to reach the payload hidden in waves. In order to detect the payload execution, we have to know what would be the result of the execution of the non-packed payload ; we have to know the initial program.

Nanda et al. [27] used the well-known *notepad.exe* to evaluate their unpacker. But we preferred to choose a smaller one without any graphical interface but with a print to the standard output in order to manually validate if the payload has been actually executed. Our choice went to *hostname.exe* which print the name of the machine to the standard output.

Thus, *hostname.exe* is packed with the corpus of packers available in our library and then traced. The metrics are :

- Execution of the payload
- Execution is complete
- Number of waves
- Execution time

The two first elements of metric indicate if the tracer was transparent enough and so if the payload has been actually executed. And the number of waves is part of the needed information representing the complexity of the packer.

Unfortunately we were not able to compare our tracer to some others. For example Renovo[13] is not maintained and is compilable with only a very old version of GNU/gcc. Polyunpack[28],

Omniunpack[29] are closed source. It appears that security analyst prefer build their own tools when a new threat come resulting in a lack of off-the-shelf softwares.

	payload executed	tracer crashed	#wave	tracing time
ACProtect v2.0	Yes	No	635	1h50min
Armadillo v9.64	Yes	No	165	12h30min
Aspack v2.12	Yes	No	3	5s
BoxedApp v3.2	Yes	No	6	19min
EP Protector v0.3	Yes	No	2	3s
Expressor	Yes	No	2	10s
FSG v2.0	Yes	No	2	2s
JD Pack v2.0	Yes	No	3	2min 16s
MoleBox	Yes	No	3	1min
Mystic	Yes	No	4	34s
Neolite v2.0	Yes	No	2	10s
nPack v1.1.300	Yes	No	2	3s
Packman v1.0	Yes	No	2	3s
PE Compact v2.20	Yes	No	4	3s
PE Lock	Yes	No	15	24s
PE Spin v1.1	Yes	Yes, wave 46	80	40s
Petite v2.2	Yes	No	3	3s
RLPack	Yes	No	2	4s
Setisoft v2.7.1	No	Yes, wave 32	32	-
TELock v0.99	Yes	No	18	7s
Themida v2.0.3.0	Yes	Yes, wave 14 & 80	106	still waiting
Upack v0.39	Yes	No	3	4s
Upx v2.90	Yes	No	2	2s
VM Protect v1.50	Yes	No	1	3s
WinUPack	Yes	No	3	4s
Yoda's Crypter v1.3	Yes	No	4	3s
Yoda's Protector v1.02	Yes	No	6	6s

Table 5: Tracer evaluation

Results Except for *Setisoft* we have seen the payload execution in every packers, so code of *hostname.exe* will be present in the extracted waves.

The execution of the old tracer version is not present in the study from Thierry[3], and it would take too much time to do it again. As I was told by the CARTE engineer who mastered the tracer, the tracing time seriously decreases from the old version to the new one. As an example it took several days to unpack the most complex one Armadillo and Themida, and now it takes only some hours. For the simpler one, it goes from minutes to seconds.

6.2 Evaluation of the disassembler

To our understanding the best way to evaluate a disassembler is to compare its result with the result from a compiler. However when the disassembler is supposed to deal with malicious softwares, it is not possible to compare the disassembly with the assembly or the CFG from the compiler, as the original code of the malware is not available. Nonetheless first test need to validate that the disassembler is sound. In order to achieve this task I will use a bunch of the benchmark suite, well-known in WCET research topic, from the Mälardalen WCET benchmark programs[30]. We choose this one as they provide a great corpus of small programs "collected from several different research groups and tools vendors around the world" (quoted from http://www.mrtc.mdh.se/projects/wcet/benchmarks.html).

In addition we are not only interesting on the comparison of the assembly/disassembly, we also want to check the CFG we are building as this will be the base of the signature for the morphological analysis.

As a compiler I choose GNU/gcc[1] because it is the most used, it is open-source and well documented. Also it implements dumping features to retrieve the assembly and the CFG.

The available option to dump a CFG is *-fdump-tree-cfg*. The resulted CFG is a text file containing the C code annotated with information about how to connect the labelized basic-blocks. This is not very useful because this option is run at a high level of abstraction during the compiler work ; there is a presence of code in C language. The resulting CFG can be very different to the one extracted at a lower level due to some optimizations from the compiler when assembling the code.

In order to retrieve the most low level intermediate representation from GNU/gcc I used the option *-fdump-rtl-dfinish* [2]. Thus GNU/gcc exports the CFG after the representation of the initial program in Register Transfer Language (RTL)[3] in which the instructions are written nearly one by one with some information on the resulting assembly instruction. Also it contains all the information of the control flow. At this point in the compiler we are at the lowest possible level because the choice of register to use for each instructions have been terminated, and the remaining part will just be pretty printing the RTL to assembly code.

However we prefer to have a graphical representation of this CFG. So I made a small script in order to compile the RTL dump to graphviz friendly format. This allow us to use the morphological analysis detailed by Thierry[3].

To illustrate this process of comparison, figure 13 presents a very small example where the C code is in listing 13a, the assembly in 13b and the disassembly in 13d. Also figure 13c shows what GNU/gcc dumped, and figure 13e what CoDisAsm trivially recovered.

As you can note, we do not have the same level of information in nodes between the two CFG. However this is not a big issue for our approach as we are interesting in the morphology of the graph.

Results Table 6 shows the results of the comparison between the output of GNU/gcc and CoDisAsm. First two columns compare the number of assembly instruction respectively generated by them.

[1] `https://gcc.gnu.org/`

[2] introduced in gcc-4.4.7, the documentation says it always produce an empty file, however we tested with gcc-4.8.4 and later, and it perfectly works

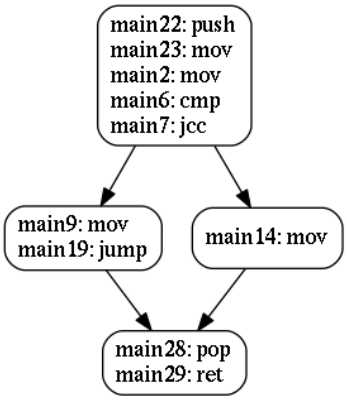
[3] https://gcc.gnu.org/onlinedocs/gccint/RTL.html

```
1  void main(int a) {
2    int b;
3    if(a == 0)
4      b = 1
5    else
6      b = 2;
7  }
```

(a) Small example in C

```
1          push    ebp
2          mov     ebp, esp
3          sub     esp, 16
4          cmp     [ebp+8], 0
5          jne     .L2
6          mov     [ebp-4], 1
7          jmp     .L1
8  .L2:mov [ebp-4], 2
9  .L1:leave
10         ret
```

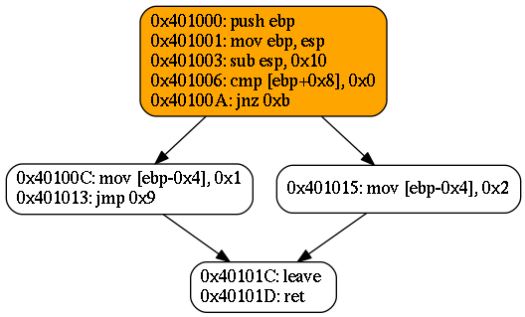
(b) Assembly generated by GNU/gcc from 13a



(c) CFG generated by GNU/gcc from 13a

```
1  0x401000: 55                    push ebp
2  0x401001: 89 e5                 mov ebp, esp
3  0x401003: 83 ec 10              sub esp, 0x10
4  0x401006: 83 7d 08 00           cmp [ebp+0x8], 0x0
5  0x40100a: 75 09                 jnz 0xb
6  0x40100c: c7 45 fc 01 00 00 00  mov [ebp-0x4], 0x1
7  0x401013: eb 07                 jmp 0x9
8  0x401015: c7 45 fc 02 00 00 00  mov [ebp-0x4], 0x2
9  0x40101c: c9                    leave
10 0x40101d: c3                    ret
```

(d) Assembly extracted by CoDisAsm corresponding to 13a



(e) CFG recovered by CoDisAsm corresponding to 13a

Figure 13: Differences between GNU/gcc and CoDisAsm

	#instr gcc	#instr cod	#diff	CFG similarity	cod time	IDA time
adpcm.exe	1191	1191	14	100%	50ms	2.6s
compress.exe	506	506	26	100%	27ms	2.8s
ns.exe	99	99	7	100%	4ms	2.4s
nsichneu.exe	5550	5550	374	100%	1.6s	3s
statemate.exe	1375	1375	186	88%	150ms	2.8s

#instr gcc : number of assembly instruction generated by GNU/gcc
#instr cod : number of assembly instruction extracted by CoDisAsm
#diff : number of different instruction between GNU/gcc and CoDisAsm
CFG similarity : percentage of similarity returned by the morphological analysis
cod time : disassembling time for CoDisAsm
IDA time : disassembling time for IDA pro

Table 6: Disassembler evaluation

The third column compares those instructions line by line and presents the number of mismatch. About this number, by looking closer we found only three interchanges: *sal* is used by gcc and *shl* is used by CoDisAsm, respectively for *jg* and *jnle*, *jne* and *jz*. When looking in the documentation, I found that those couple are actually aliases and are semantically equivalent. Thus depending on the decoder used you may have interchangeably both of us. So there is no semantical differences in assembly listings.

The fourth column presents the percentage of similarity between the two CFG returned by the morphological analysis tool describes by Bonfante et al. [8]. The 88% for *statemate.exe* comes from instructions featuring floating point unit. It seems that the assembler finaly generated by GNU/gcc is different than the level of abstraction at which I extracted the CFG, because they depend on the capacity of the target processor to deal with floating point arithmetic. Our disassembler well retrieve the final CFG, but the one extracted from the compiler is not enough low-level.

Finally the fifth column shows the time needed by the disassembler. Those test have been run on a laptop with a *Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz* with *4Gb* of RAM and running *Debian testing*. And it is worth noticing that the implementation from section 5.2 is far much efficient than IDA pro in the last column.

Those results show we have now a sound and efficient disassembler.

6.3 Evaluation the whole method

At this point we know if our disassembler is efficient, and we know if our tracer is also efficient. Thus we want to know if the combination of this 2 projects is efficient and usable as part of a malware detector. Also we need to clarify the usefulness of the trace.

To achieve this goal we will reuse the *hostname.exe* from the second evaluation step. By focusing on packers that really executed the payload and checking in which waves the original CFG is found. Thus it will validate if that the CFG of the malicious code could be recovered when dealing with malwares.

Also to validate the usefulness of the trace, we will try to disassemble the packed program without a trace and check if the payload is present. In case it is positive, the use of a trace would

be useless.

	tot T #instr	**tot D #instr**	**wave - payload**	**time**
hostname.exe	154	201	1 - 100%	4 ms
Aspack v2.12	1631	1030	3 - 100%	48ms
EP Protector v0.3	186	201	2 - 100%	5ms
Expressor	1217	487	2 - 100%	7ms
FSG v2.0	240	201	2 - 100%	6ms
JD Pack v2.0	7313	3468	3 - 100%	1.275s
MoleBox	9455	16230	2 - 2% 3 - 97%	6.677s
Mystic	3189	1910	4 - 100%	232ms
Neolite v2.0	1386	895	2 - 100%	72ms
nPack v1.1.300	818	474	2 - 100%	17ms
Packman v1.0	1337	533	2 - 100%	49ms
PE Compact v2.20	1139	223	4 - 100%	19ms
PE Lock	10968	5770	14 - 38%	117ms
PE Spin v1.1	14410	8255	???	309ms
Petite v2.2	1591	4694	3 - 68%	124ms
RLPack	464	341	2 - 100%	11ms
TELock v0.99	2274	1681	18 - 70%	56ms
Upack v0.39	494	303	3 - 100%	14ms
Upx v2.90	322	202	2 - 100%	8ms
VM Protect v1.50	264	859	1 - 60%	19ms
WinUPack	455	291	2 - 7% 3 - 99%	11ms
Yoda's Crypter v1.3	1038	1270	4 - 48%	44ms
Yoda's Protector v1.02	2482	2670	6 - 100%	22ms

tot T #instr: number of instructions found in different trace files
tot D #instr: number of instructions statically disassemble in every waves
wave - payload: the percentage of the payload found per wave, if not define it means 0% for the wave.
time: time to disassemble every waves

Table 7: Tracer evaluation

Results At first, the packed binary is statically disassembled. However the original CFG of *hostname.exe* was never present and the resulting CFG stops as the wave switching as expected.

Then the table 7 shows the disassembling details when using a trace in conjunction of CoDisAsm. The first column shows the accumulated number of instructions found in trace from each waves and is opposed to the number of instructions statically disassembled. The third column shows the wave that contain the payload and the percentage of it.

Finally, the time needed to recover the CFG of each waves.

There is some waves where we are not able to retrieve 100% of the payload. For example for the *Petite v2.2* packer, we only found 68% of similarity between the CFG of *hostname.exe* and the one extracted from the third wave. By looking closer on the graphs, it appears that the third wave start with a *call* instruction. As said in section 4.2 CoDisAsm make the assumption that the *call* will return to the next linearly found instruction. However in this case, it will never happen and it will actually disassemble some useless code. So the generated CFG will contain a lot of node that can not be matched with the graph of *hostname.exe*.

7 Related work

Malwares are an increasingly important threat for our computer and there is no ultimate solution to cure it. They are distributed as a binary file. In order to study old malwares or to build some tools to detect future one, a disassembler is required to recover the maximum information on the malicious code. However a simple disassembler like the well-known and widely used IDA pro is not enough because malwares authors use of their imagination to obfuscate their original code. Most of the time they use a packer which add several protection level depending on the complexity of the packer; those one warp the original code with several self-modifying level. Bayer et al. [31] proposed a short overview of the technique used by packers in 2008. Much more recently (May 2015) Ugarte-Pedrero et al. [32] studied the complexity of a corpus of packers to classify them.

The reference software in *reverse-engineering* is a commercial tool called IDA Pro [1]. It uses the recursive traversal method to disassemble in conjunction with a lot of complex heuristics in order to identify some elements in the assembly ; i.e.: function identification, indirect branch.

First disassembler is provided by the GNU binary utility library GNU/objdump. It uses the linear sweep technique to retrieve assembly code. Even if this tool is powerful when using by *linker time optimizer*[33], it has the major drawback to see each byte of the binary as part of an opcode, mixing potential code and data.

First improvement was to use static analysis and recursive traversal to recover the assembly and the CFG. Cifuentes et al [34], Kinder and Veith [35, 36] with Jackstab, both use data-flow analysis already mastered in compilation [11]. Even if Jackstab is able to deal with certain form of obfuscation, it can not handle self-modifying code.

Then with more accuracy and also more complexity, Reps et al. [37, 38, 39] with CodeSurfer created the Value Set Analysis algorithm based on abstract interpretation, also mastered in compilation [40]. Their method has been reused by Laporte et al. [41] to verify self-modifying property on software. They compute an over-approximation of the values present in registers and memory cells at each point of the program, resolving the problem of indirect jump[42]. However as every over-approximation techniques results can contain a lot of false-positive (wrong target for indirect jump). Thus Kinder et al. studied the possibility of alternating under-/over-approximation in [43].

And more recently Bardin et al. [44] combine advantages of data flow analysis and control flow reconstruction using k-set possibility for indirect branch.

Opposed to abstract interpretation, symbolic evaluation [45] uses a similar approach of the problem, but implemented in a fully different way. Instead of reporting the impact of instruction on an abstract machine, they create a formula describing the symbolic value of an element, then feed a SMT solver to compute the over-approximation. Bardin et al. [25, 46, 26] with OSMOSE use symbolic

[1] https://www.hex-rays.com/products/ida/

evaluation in conjunction of dynamic analysis to recover the CFG of a binary file. Nonetheless their primary goal is to build the set of possible input arguments in order to instrument all possible paths of a CFG. This method has been applied by Yadegari et al. [47] which are successful to deal with Return-Oriented Programing a complex obfuscation technique.

However static analyses create always an over-approximation which is often too imprecise and report a lot of false positives. That is why this document takes a more pragmatic path by combining dynamic and static analysis. Also none of them are able to retrieve the payload in a packed malware due to the self-modifying part. Some disassemblers use dynamic analysis such as BIRD from Nanda et al. [27] but it is still not design to deal with obfuscated code. The closer work to our is BAP from Brumley et al. [48] which use an hybrid method : the conjunction of an execution trace and the static analysis to recover the CFG. However this platform is not able to work with program containing system call.

The second utility software in the box of security analyst is an unpacker. This problem has been widely studied as new unpackers come with new discovered packing methods. And solutions are very different, however all of them aim to unpack without recovering a CFG.

Some unpackers are based on statistics or heuristics. PolyUnpack[28] compares the code in memory with the binary file to determine if it is self-modifying. Omniunpack[29] monitors memory pages and launch the anti-virus when a write is done on a *write-abord* page. Eureka[49] uses heuristics to decide when the code is unpacked based on system call usage. But most unpackers are based on dynamic analysis and execute the code. Renovo[13] and Cesare et al.[50] use an emulated environment based on QEMU/TEMU. Like Ehter[17], we prefer to use a virtual machine as it is easier to let the OS handling its own system calls. Coogan[51] tried to tackle the unpacking problem by using static analysis. They tried to identify unpacking routine from the original code using control flow and alias analysis. There exists an open-source framework called Titan Engine[52] to unpack code using dynamic analysis, however this is just a framework and would require more attention to test it.

To formalize self-modifying behavior, Debray et al. [53] first modeled this concept as phases. Then Guizani et al. [22] redefined it as waves where the sequence of waves is monotonic, and no possible back transition to previous wave.

8 Conclusion

This document first introduced a brief history of existing malwares before the context and complexity of analyzing this kind of softwares. It also contextualized the frame of the study by an overview of the chosen architecture particularly used by malwares: x86 processors and Microsoft Operating System.. Then the theoretical part presented the chosen way to disassemble and unpack malwares. This was followed by the presentation of common difficulties met when disassembling a normal program then when disassembling obfuscated one. Also, by answering to those pitfalls, a justification on the chosen disassembling/unpacking techniques has been expressed.

The full context of this master thesis exposed, the implementation realized during the internship is detailed and evaluated. The interpretation of the given results in section 6 are very encouraging. The built disassembler is far much efficient than the commercial reference IDA pro. Also, comparing the result with a compiler brought the implemented disassembler as closer as possible to the original code.

A last word on the evaluation, it would have been preferable to also compare the disassembly of

a malware. But the problem is in the lack of reference for malicious software to compare it with, indeed malwares are not distributed with their source code.

Nonetheless, the objectives presented in the introduction are reached. The disassembler coupled with the tracer is sound and efficient. Also its scalability has been showed by the implementation of extensions to improve accuracy of the disassembly and the malware detection.

However, there is still future work left, especially with extensions. Indeed, the more important one co-developed with BINSEC team member is still in early alpha and require a lot more work to be efficient. Also, the definition of a good strategy for the concretization/symbolization is needed and the way to build it in order to optimize the resolution of the formula which often returns the entire address space (2^{32} possibilities) as a result, which is unusable in practice. In addition, the improvement I suggested on the execution trace is incompatible with the BINSEC project. We had to revert to the previous version to gain information and improving the veracity of the solver in contrast of the loss of performance. In conclusion, the symbolic evaluation technique can be used for a malware analyst but not for a real time malware detector.

The unpacker can also be improved. A very recent study [32] on the complexity of packers revealed the possibility for malicious code to be split in several frames (their concept of frames is close to our waves). Thus fooling our assumption on the presence of the full code in a wave.

On the disassembler I was told to leave some future works and experiments. To improve the coverage of disassembled code, it would be useful to simulate movements on the stack and handling specific registers for interruptions. And I believe that implementing some abstract interpretation oriented feature to compare it with symbolic evaluation would be relevant. Also a generic disassembler must be open to other architecture such as ARM to detect and analyze malwares build for smartphones.

Acknowledgements

Thanks to Jean-Yves Marion for accepted me in internship, for all his useful advices and the time he consecrated on my supervision ; to Guillaume Bonfante and Fabrice Sabatier for their help and pertinent remarks ; to Hubert Godfroy my fellow workmate.

References

[1] D. Reynaud, Analyse de codes auto-modifiants pour la sécurité logicielle. PhD thesis, UThèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, 2010.

[2] J. Calvet, Analyse dynamique de logiciels malveillants. PhD thesis, Université de Lorraine, 2013.

[3] A. Thierry, Désassemblage et détection de logiciels malveillants auto-modifiants. PhD thesis, UThèse de Doctorat d'Université, Université de Lorraine, 2015.

[4] A. Thabet, "Stuxnet malware analysis paper," Freelancer Malware Researcher, pp. 3–28, 2010.

[5] P. Szor, "The art of computer virus defense and research," 2005.

[6] A. Swinnen and A. Meshbahi, "One packer to rule them all: Empirical identification, comparison and circumvention of current antivirus detection techniques.,"

[7] M. Kaczmarek, Des fondements de la virologie informatique vers une immunologie formelle». PhD thesis, Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, 2008.

[8] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "Architecture of a morphological malware detector," Journal in Computer Virology, vol. 5, no. 3, pp. 263–270, 2009.

[9] C. Timsit, "Du transistor à l'ordinateur," 2010.

[10] Intel, Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools (2Nd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[12] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, "N-version disassembly: differential testing of x86 disassemblers," in Proceedings of the 19th international symposium on Software testing and analysis, pp. 265–274, ACM, 2010.

[13] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in Proceedings of the 2007 ACM workshop on Recurring malcode, pp. 46–53, ACM, 2007.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," ACM Sigplan Notices, vol. 40, no. 6, pp. 190–200, 2005.

[15] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, "Behavior abstraction in malware analysis," in Runtime Verification, pp. 168–182, Springer, 2010.

[16] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in Security and Privacy, 2007. SP'07. IEEE Symposium on, pp. 231–245, IEEE, 2007.

[17] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in Proceedings of the 15th ACM conference on Computer and communications security, pp. 51–62, ACM, 2008.

[18] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in Machine Learning and Knowledge Discovery in Databases, pp. 522–536, Springer, 2011.

[19] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 184–196, ACM, 1998.

[20] C. Jamthagen, P. Lantz, and M. Hell, "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries," in Anti-malware Testing Research (WATeR), 2013 Workshop on, pp. 1–9, IEEE, 2013.

[21] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm :a disassembly of self-modifying binaries with overlapping instructions.".

[22] W. Guizani, J.-Y. Marion, and D. Reynaud-Plantey, "Server-side dynamic code analysis," in Malicious and unwanted software (MALWARE), 2009 4th international conference on, pp. 55–62, IEEE, 2009.

[23] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing cpu emulators," in Proceedings of the eighteenth international symposium on Software testing and analysis, pp. 261–272, ACM, 2009.

[24] P. Ferrie, "Anti-unpacker tricks," Proc. of the 2nd International CARO Workshop, 2008.

[25] S. Bardin and P. Herrmann, "Osmose: automatic structural testing of executables," Software Testing, Verification and Reliability, vol. 21, no. 1, pp. 29–54, 2011.

[26] S. Bardin, N. Kosmatov, and F. Cheynier, "Efficient leveraging of symbolic execution to advanced coverage criteria," in Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, pp. 173–182, IEEE, 2014.

[27] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh, "Bird: Binary interpretation using runtime disassembly," in Proceedings of the International Symposium on Code Generation and Optimization, pp. 358–370, IEEE Computer Society, 2006.

[28] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual, pp. 289–300, IEEE, 2006.

[29] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, pp. 431–441, IEEE, 2007.

[30] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," (Brussels, Belgium), pp. 137–147, OCG, July 2010.

[31] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors," in USENIX workshop on large-scale exploits and emergent threats (LEET), 2009.

[32] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," 2015.

[33] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 27, no. 5, pp. 882–945, 2005.

[34] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in Software Maintenance, 1998. Proceedings., International Conference on, pp. 228–237, IEEE, 1998.

[35] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in Computer Aided Verification, pp. 423–427, Springer, 2008.

[36] J. Kinder, "Static analysis of x86 executables," 2010.

[37] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86 - a platform for analyzing x86 executables," in Compiler Construction, pp. 250–254, Springer, 2005.

[38] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 32, no. 6, p. 23, 2010.

[39] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in Compiler Construction, pp. 5–23, Springer, 2004.

[40] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 238–252, ACM, 1977.

[41] S. Blazy, V. Laporte, and D. Pichardie, "Verified abstract interpretation techniques for disassembling low-level self-modifying code," Proc. of the 5th conference on Interactive Theorem Proving (ITP), 2014.

[42] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in Proceedings of the 10th ACM conference on Computer and communications security, pp. 290–299, ACM, 2003.

[43] J. Kinder and D. Kravchenko, "Alternating control flow reconstruction," in Verification, Model Checking, and Abstract Interpretation, pp. 267–282, Springer, 2012.

[44] S. Bardin, P. Herrmann, and F. Védrine, "Refinement-based cfg reconstruction from unstructured programs," in Verification, Model Checking, and Abstract Interpretation, pp. 54–69, Springer, 2011.

[45] J. C. King, "Symbolic execution and program testing," Communications of the ACM, vol. 19, no. 7, pp. 385–394, 1976.

[46] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in Computer Aided Verification, pp. 165–170, Springer, 2011.

[47] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," tech. rep., Technical report, Department of Computer Science, The University of Arizona, 2014.

[48] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in Computer Aided Verification, pp. 463–469, Springer, 2011.

[49] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in Computer security-ESORICS 2008, pp. 481–500, Springer, 2008.

[50] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," in Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107, pp. 61–70, Australian Computer Society, Inc., 2010.

[51] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic static unpacking of malware binaries," in Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, pp. 167–176, IEEE, 2009.

[52] M. Vuksan, T. Peričin, and V. Milunovic, "Fast & furious reverse engineering with titanengine. black hat usa 2009," 2009.

[53] S. Debray and J. Patel, "Reverse engineering self-modifying code: Unpacker extraction," in Reverse Engineering (WCRE), 2010 17th Working Conference on, pp. 131–140, IEEE, 2010.