Master research Internship

Bibliographic report

Disassembling x86 Low-Level Code and CFG reconstruction

Domain : Binary Analysis - Symbolic Computation - Information Retrieval

Author:
Benjamin Rouxel

Supervisor:
Jean-Yves Marion
LORIA - CARTE

Abstract

Cyberterrorists do not only use Denial Of Service attacks. Past years have seen emerged a lot of very evolved viruses such as Duqu's driver [1] or Stuxnet [2]. In order to facilitate the reverse-engineering process of such programs, researchers need powerful binary analysis platforms as the source code is not available. The first step of such platform is : disassembling the binary file. This consists in translating low-level instruction code to a higher level of abstraction. However viruses are not easy to disassemble. Indeed developers use techniques to obfuscate their code in order to make the reverse-engineering process much harder. They also use self-modifying code which makes most of the binary analysis tools inefficient. This document presents existing approaches and tools to reconstruct program structure called Control Flow Graph from binary code, and gives a hint on their usefulness when dealing with code obfuscation and self-modifying code. This work is part of the ANR BINSEC[1] project which aims to build an efficient binary analysis platform with such programming techniques. At first this platform will support x86 assembly code. However the final goal is to be architecture independent in order to support all kind of binary code with the help of an Intermediate Language.

Contents

[1] http://binsec.gforge.inria.fr

1 Introduction

Reverse engineering is the opposite of compilation. It starts from a binary level to a higher level of abstraction. Disassembling is the first part of this process and consists in recovering the assembly code from the binary code. The second part is the Control Flow Graph (CFG) reconstruction. This program representation is well-known in compilation area as it represents every possible execution path of a program. To summarize the definition of a CFG: a node is a sequence of instructions (block of instructions or basic block) and an edge represents a conditional or unconditional jump from one node to another one. An other good practice in compilation theory is to use Intermediate Representation and in our case an Intermediate Language (IL). This helps gaining abstraction from the source code and from the Instruction Set Architecture (ISA) which depends on the configured target machine on the compiler when executable has been compiled. So most of the binary level analyzers first disassemble the binary code to an IL or a CFG in order to perform some useful analysis. Figure 1 illustrates the whole process of the reverse engineering compared to the compilation process.

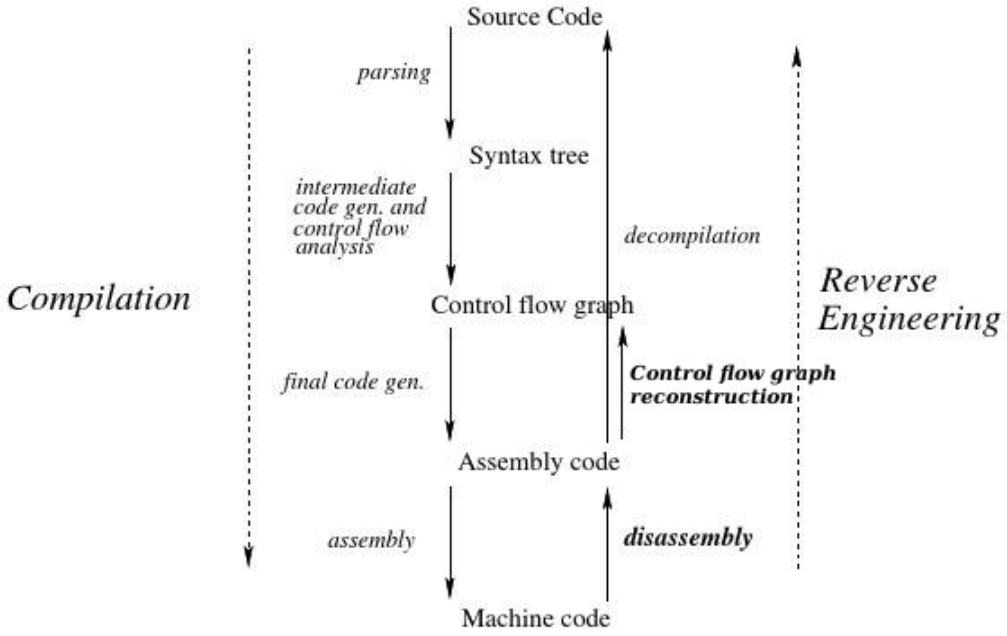


Figure 1: Compilation vs Reverse Engineering

However disassembling or reconstructing the CFG is not an easy task. Indeed the input data is just a bunch of bytes with potentially any structure: no separation between code instructions and data, no more textual variables, no easily recognizable function entry points... Also one of the first problem a disassembler will face is *indirect jump*, this is due to a branching instruction where the target address will be known only at run time, like calls to a shared library or function pointers.

Other difficulties come from *Obfuscated code*. This refers to techniques where the goal is to make the disassembling process more difficult [3]. Also, *Self-modifying code* are programs that modifying their code in memory while they are executing, and so the disassemble code may not be the one that will really run the feature to what the program has been created [1]. Those two techniques

are mostly used by computer viruses developers in order to make the analysis of their malwares or viruses more difficult to security experts. Nevertheless code obfuscation or self-modifying code are not only used for bad purposes. Some companies protect their intellectual property with obfuscation.

Binary analysis are used in a lot of different manner. Security researchers need it to study the behavior of computer viruses or malwares [1, 4]. Indeed the source code is not available for this kind of program, and so source code analysis is not possible. Real-time application developers need safe and robust binary analyzers. They need to compute an exact Worst Case Execution Time (WCET) for their application in order to schedule them well and being sure every functionality will be complete on time [5]. WCET must be computed at the binary level in order to increase its precision and safety. Automatic Test data Generators (ATG) use the binary level to estimate a set of values as input data to test a program and being sure to cover the whole code [6]. As some critical area needs more than just a bunch of tests like aeronautic, medicine, ... software verification tools now perform their validation on the binary level in order to validate that compilers do not introduce bugs [7]. Also most companies include Components Off the Shelf (COTS) in their applications. And They might wonder to check if they are not buggy nor contains exploitable vulnerabilities [8, 9, 10].

Binary analysis has been a lot studied and there exists plenty of tools. Some of them are commercials as the well-known IDA Pro[2], while most of them are research-oriented products as we will see in the rest of this report. However there exist no complete tool which disassemble every architecture assembly code nor handle every kind of code obfuscation or self-modifying machine.

This internship topic is part of the ANR[3] project BINSEC[4] which aims to build an architecture independent binary analysis platform. Available platforms have good results but only under some specific conditions, this project aims to extract the better part of each of them. Nevertheless none of them properly handle *Instruction Overlapping* or *Packed Executable* as a specific technique respectively from code obfuscation or self-modifying code.

Section 2 will present the main difficulties we will face. Section 3 will survey the disassembling and CFG reconstruction tools and techniques from the research area which are closed to the BINSEC project. Section 4 will list type of IL available with such tools and compare their usage. To conclude section 5 will summarize this document and shortly present my vision in accordance to my internship work.

2 Difficulties of our concern

Disassembling and reconstructing the CFG aim to retrieve the control flow of the program to be analyzed. However due to some x86 architecture specificities this can be painful like detecting the bytes of data and bytes of instructions because instruction length is not fixed, or like indirect jump because target address are a priori unknown. We will not focus on the instructions/data separation issue because studied disassemblers look for instructions at specific addresses into binary file depending of the current analysis flow meaning this kind of information is not a prerequisite. So we will focus only on recovering possible indirect jump target addresses.

In addition, the BINSEC platform aims to help security researchers to study computer viruses. A well-known expression in this particular field has been introduced by Balakrishnan and Reps [11] : *WYSINWYX*. This acronym stands for *What You See Is Not What You eXecute*. It means *pretty-*

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

[3] Agence National pour la Recherche

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

printing assembly code from binary file will not result to the set of instructions the processor will effectively execute. This problem is related to *Code Obfuscation* and/or *Self-modifying Code*. Both correspond to *program transformation* techniques whose are deliberately constructed to resist against reverse engineering [12]. The former categorizes transformations that are performed at compile time while the latter represents programs that modify themselves at run time. It is important to note that *Self-modifying* is meant to programs which rewrite themselves on purpose, as said it is not due to programming error such as *Buffer Overflow*[5].

Those transformation techniques are not initially met to be harmful, but have some application field where it is a protection against hackers. Companies commonly protect their intellectual property by incorporating such features in their product, i.e.: the part of a software which unlock a *pro*-version from the *free*-version. However *Code Obfuscation* and *Self-modifying Code* are also used by malwares and viruses designers to avoid the scrutiny of both anti-virus software and reverse engineers.

As mentioned before I will focus here only on the *Instruction Overlapping* obfuscation and on *Code Unpacking* modification, those one have not so much been studied in the disassembling process. More information can be found in [12] about code obfuscation and [13] about self-modifying code.

```
1  0000: B8 00 03 C1 BB          1  mov eax, 0xBBC10300
2  0005: B9 00 00 00 05          2  mov ecx, 0x05000000
3  000A: 03 C1                   3  add eax, ecx
4  000C: EB F4                   4  jmp $-10
5  000E: 03 C3                   5  add eax, ebx
6  0010: FF E0                   6  jmp eax
7  ...                           7  ...
```

(a) Pretty-printing disassembling

```
1  0000: B8 00 03 C1 BB          1  mov eax, 0xBBC10300
2  0005: B9 00 00 00 05          2  mov ecx, 0x05000000
3  000A: 03 C1                   3  add eax, ecx
4  000C: EB F4                   4  jmp $-10
5  0002: 03 C1                   5  add eax, ecx
6  0004: BB B9 00 00 00          6  mov ebx, 0xB9
7  0009: 05 03 C1 EB F4          7  add eax, 0xF4EBC103
8  000E: 03 C3                   8  add eax, ebx
9  0010: FF E0                   9  jmp eax
10 ...                           10 ...
```

(b) Overlapping care in disassembling process

Figure 2: Example of overlapping instructions and indirect jump in x86 assembly

[5]Buffer overflow correspond to a memory access violation. When writing more data that the receiving buffer can support.

2.1 Indirect Jump

Binary code is sequentially executed by the processor starting at a specific address inside the binary file. To modify the sequence flow, all ISA provide *jump* instructions, some are unconditional meaning the jump will be always executed by the processor, some are conditional meaning the jump will be taken if a condition is satisfied. We then must distinguish two kind of jump *direct* and *indirect*. The listing 2a shows at line 4 a direct jump: the address of the next instruction is statically known. In this case we know the next instruction will be at the current address ($0\mathbf{x}C$) minus an offset (10) resulting to the address $0\mathbf{x}2$. Then line 6 shows an indirect jump: to know the next instruction address we must know the value of register *eax*. In this example the value is stored in a processor registry, however it is also possible to store it at a memory location that can be read by a jump instruction.

Indirect jumps are used by compiler to enable higher language features such as C or C++ [14]. They are used to allow *function pointers* also referred as *indirect call*, the address of a function is stored at a location in memory when calling the function the address must before be loaded into a register or directly accessed from the memory. Also indirect jump may be used when doing *multiway branch* which consists in replacing *switch* statement or sequence of *if* statement by a smarter structure [15].

So indirect jumps are jump where the target address is not statically known but only at run time. This is one of the main difficulties when disassembling code or reconstructing a CFG from a binary file. When the disassembler find such kind of instruction it looses the flow of execution.

2.2 Instruction Overlapping

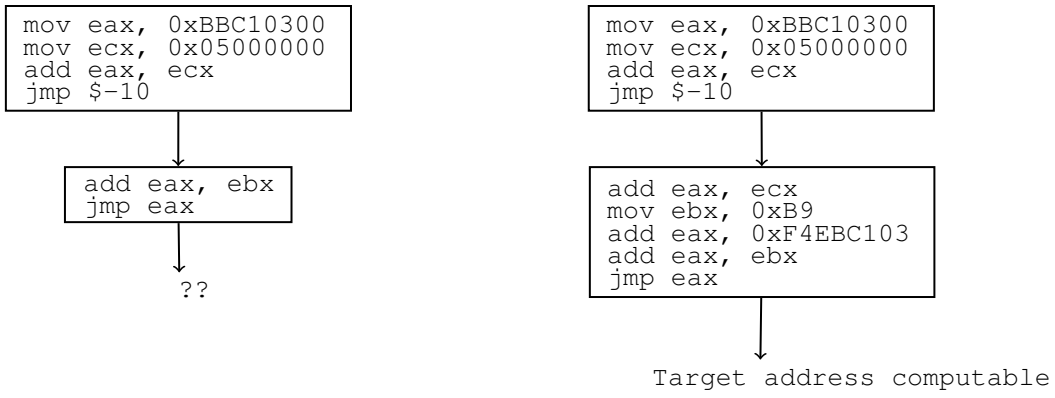
Another difficulty when disassembling code comes from code obfuscation techniques. Indeed those have been created to make disassembling and CFG reconstruction process much harder. We focus here on instruction overlapping because it hasn't been so much studied for that matter. This technique is related to a particularity from x86 assembly : the possibility to have not well-aligned instructions in memory (in listing 2a, all instruction addresses are not a power of 2).

This specificity allows to jump at any address inside the code section in memory. This is illustrated by figure 2. The initial representation of an executable in memory is shown by the left part of 2a, and the right part shows the result from a *pretty-printer* disassembler. The interesting part is on the direct *jmp* decoded from address $0\mathbf{x}C$ (line 4). This is a relative jump that will transfer control flow ($0\mathbf{x}C - 10 = 0\mathbf{x}2$) inside the first decoded instruction resulting in a new flow of instruction illustrated by lines 5,6,7 in listing 2b.

So instruction overlapping is the possibility for an instruction to be part of several instructions while disassembling always result to a valid assembly code. A disassembler not aware of such technique will output the CFG in figure 3a while an aware one will output the CFG in figure 3b. One interesting thing must be noted, the target address of the indirect jump *jmp eax* become computable to instruction overlapping aware disassembler with a simple data-flow analysis (presented in 3.1.1) as it was impossible before to know the value of register *ebx* and a fortiori *eax*.

2.3 Code Unpacking

In addition to code obfuscation, virus developers use Self-modifying code pattern. This refers to any code that alters its execution at run time. Usually the program rewrite itself in memory with new instructions. Code unpacking technique relies on program composed of two specific parts:



(a) CFG reconstructs from code in figure 2a

(b) CFG reconstructs from code in figure 2b

Figure 3: CFG reconstruction from previous example

- a payload : contains the real program to be executed
- an unpacker : piece of code to transform the payload

When just having a glance at the payload, it may appear to be harmless and maybe not representing executable instructions. The unpacker will take the payload and decodes it in memory before branching the control flow to the decoded payload. As seen by Marion in [16] this process can be repeated many times with different unpackers before having the real program in the memory.

Their exists different techniques for the unpacker. An example is encryption: the payload is encrypted, and then the packer will be in charge to decrypt it. The unpacker can also be a small compiler or interpreter and the payload just appears as a string which will become executable after compilation.

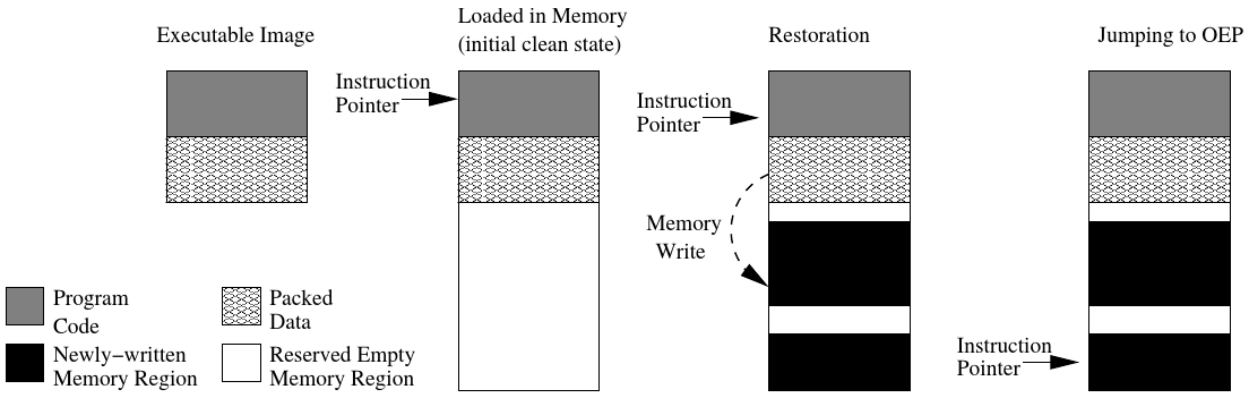


Figure 4: Overview of packed executable principle

Figure 4 presents the idea of how a packed executable works. From left to right, we have an executable file loaded in memory, then instructions start to be executed. Some of them will manage to copy the packed data to a new memory area after applying a transformation on it to make it executable. Finally, a jump will occur to branch to the newly available code. This example shows

the modification of an other area in the memory, but we can easily imagine to alter the already executed code part of the memory.

Obviously the main problem when dealing with packed code is the ability for disassemblers to recover the last payload value. So disassemblers must interpret or execute the unpacker in order to retrieve the original code. Binary analysis tools aware of code packing run the program to get an execution trace (see section 3.2).

3 Disassembling and CFG Reconstruction

Disassemblers allow to recover assembly code from binary code. First of them sequentially read the binary file and dump the instructions like GNU objdump[6]. Those one are categorized as *Linear sweep disassembler* and are useful for some applications such as link-time optimization [17]. However du to problematics pointed early (section 2), those disassemblers are not enough efficient to analyze malwares and computer viruses.

The CFG reconstruction is the process to recover control flow information. Control flow refers to the order each program instruction will be executed, this include to determine every next instruction of a conditional branch independently of the effective evaluation of the condition (e.g.: knowing the the start of instruction block of either the *then* either the *else* statement of an *if-then-else* statement). Thus every jump target address must be know.

Thus those two processes are related and most of binary analyzers covered by the literature provide description on how to disassemble and how to reconstruct the CFG with more or less constraints on the input binary code. Because it is not efficient to sequentially read the binary file, almost every binary analyzer presented in the literature perform an iterative reconstruction either of the assembly code, either of the CFG at the first pass. They all perform a recursive traversal by following direct jumps and present a technique to resolve the *indirect jump problem*.

For readers not familiar with some compilation/decompilation concepts, it is important to note that disassembling and reconstructing the CFG results are different. Disassembling results to an assembly instruction listing, while a CFG represents the different possible sequence of instructions called execution paths. Figure 5 shows those differences, on the left side a sample of a binary file, in the middle the corresponding instructions listing resulting from the disassembler, and on the right side the corresponding CFG. For convenience I also placed under those figures the corresponding source code in C language.

Disassembling has been a lot studied in different research field and every known approaches can be placed in the next two subsections. First, static analysis methods try to recover control flow information without executing the code, so only by analyzing the binary file. Second, dynamic introspection effectively runs the program to get one or more execution traces before statically analyzing it.

3.1 Static Analysis

Static analysis refer to techniques which don't need to effectively run the program in order to perform an analysis. Most of static disassembling methods are derived from compilation research such as data-flow analysis or abstract interpretation. Some other comes from automatic test data

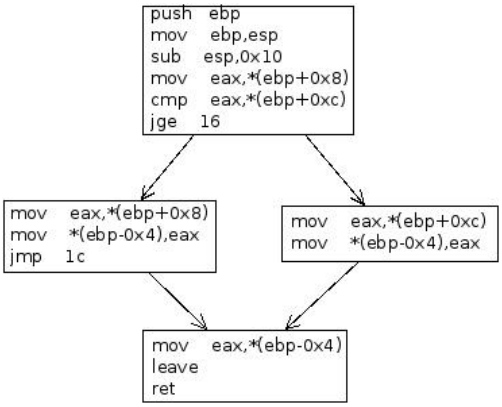
[6] `https://www.gnu.org/software/binutils/`

```
00000000 <min>:
 00: 55
 01: 89 e5
 03: 83 ec 10
 06: 8b 45 08
 09: 3b 45 0c
 0c: 7d 08
 0e: 8b 45 08
 11: 89 45 fc
 14: eb 06
 16: 8b 45 0c
 19: 89 45 fc
 1c: 8b 45 fc
 1f: c9
 20: c3
```

(a) Sample of a binary file for a *min* function

```
1  00000000 <min>:
2   push   ebp
3   mov    ebp,esp
4   sub    esp,0x10
5   mov    eax,*(ebp+0x8)
6   cmp    eax,*(ebp+0xc)
7   jge    16
8   mov    eax,*(ebp+0x8)
9   mov    *(ebp-0x4),eax
10  jmp    1c
11  mov    eax,*(ebp+0xc)
12  mov    *(ebp-0x4),eax
13  mov    eax,*(ebp-0x4)
14  leave
15  ret
```

(b) Assembly code corresponding to 5a



(c) CFG corresponding to 5a

```
1  int min(int a, int b) {
2      int c;
3      if(a < b) {
4          c = a;
5      }
6      else {
7          c = b;
8      }
9      return c;
10 }
```

(d) Source code in C language corresponding to the above example

Figure 5: Example showing differences between the results of a disassembler and a CFG recoverer

generator researches such as symbolic execution. As we will see later symbolic execution and abstract interpretation are related.

3.1.1 Data-Flow Analysis

Before any deeper analysis ever disassembler perform a data-flow one. As it sounds the principle is to compute the flow of data. In other words, what information is carried by each program point (typically instruction) depending on the performing analysis. This idea has been introduced by compiler optimizations [18]. However the main difference with compiler comes from the CFG. When compiling the whole CFG is known, when disassembling it is discovered on the fly. So, data-flow analysis are performed on a partial control flow graph and iteratively re-run when new edges are discovered and thus until no new edges are discovered.

Cifuentes et al. [19] developed a decompiler to recover C source file from SPARC[7] assembly in

[7]SPARC (from "Scalable Processor ARChitecture") is a RISC instruction set architecture (ISA) developed by Sun

which they only perform data-flow analysis. However their algorithm doesn't deal well with value that are not stored in registers (e.g. in memory). Jackstab created by Kinder et al. [20] well handles registers and memory cells. But if the data-flow analysis doesn't return a proper value the result will be an incomplete CFG.

Extracted from those previous presented work, most common data flow analysis for this task are:

- Constant Propagation : propagate constants between registers/memory
- Value Analysis : evaluate arithmetic expressions
- Available expression : retrieve arithmetic expression for a particular register/memory

With those analysis, disassemblers are able to statically compute some registers values.

```
1  mov eax, 0x42
2  mov ebx, 0x1
3  mov *ebx, eax     ; store 0x42 at memory location 0x1
4  mov eax, *ebx     ; load value located at address 0x1 into eax
5  jmp eax
```

Figure 6: Dummy example of x86 code where data-flow analysis fails

So, static analysis partially solves the problem of indirect jump but is not powerful enough. Indeed if a value is placed into a memory cell from a register, then if this value is loaded again into a register, data-flow analysis will not be able to retrieve value for the indirect branch. As it is illustrated by figure 6, no data-flow analysis will be able to determine the value of register *eax* (0**x**42). Also data-flow analysis consider the memory to be static, so obfuscated code and self-modifying code can not be handle with such techniques.

However data-flow analysis must be completed prior to perform other kind of analysis. As this might be very common, most of paper does not mention data oriented analysis, but they obviously use at least *Constant Propagation*.

3.1.2 Concrete/Abstract Interpretation

Dealing with data-flow analysis is not enough to recover relevant information on registers and memory cells. Further disassembling techniques introduced the concept of symbolic execution or interpretation. Those are still considered as static analysis as they still don't run the program. The idea is to study the semantic of each decoded instruction in order to virtually interpret the program.

Symbolic execution Coming from automatic test data generators area, King and Wegbreit [21] introduced a technique called *Symbolic Execution*, sometimes referred as *Symbolic Interpretation* or *Symbolic Evaluation*. This implies to interpret each instruction of the input program in order to retrieve the symbolic value of each register or memory cell. Symbolic values are expressed in term of their arithmetic expression. Thus when an indirect jump appears they are able to compute a set of values which are an approximation of the target addresses. It must be noted that this set is finite as the allocated memory to a process is bounded.

Microsystems

As the set of values can be huge, Bardin et al. [22] improved the previous method with a detection of precision loss with a method called Value Analysis with Precision Requirements (VAPR). The key to their algorithm is a backward introspection in order to locate the location of this loss of precision and how to regain it. They developed OSMOSE [6, 23] a tool to automatically generate test data in order to cover the whole code of binary executable. And to do so they must recover the CFG of the initial binary program.

Abstract Interpretation Cousot and Cousot [24] introduced the concept of abstract interpretation. This is based on the same concept as symbolic evaluation, interpreting each instruction of the program. However instructions are not concretely evaluated, their method builds an abstract representation of it which is then interpreted. The goal is to provide a general method to statically analyze a program in order to prove a specific property on it. It must be noted that the evaluation in the abstract domain is an *over-* or *under-* approximation of the concrete value, depending on the analysis.

This abstract concept has been ported to binary analysis by Balakrishnan and Reps [25, 11] by creating the Value Set Analysis (VSA) method, which compute a range of values for registers and memory cells. VSA is based on an abstraction of the machine leading to an over-approximation of the memory representation at the end of the analysis. Their abstract domain is based on *memory-regions* which represent a group of locations that have similar run-time properties and *abstract locations* which represent the location in the binary file where *data* can be found. The implementation of VSA is available in CodeSurfer/x86 [26] which is a platform build around the commercial tool IDA Pro[8].

The VSA method is the foundation of all disassemblers built on an abstract interpretation model such as [26, 27, 28, 8].

Keeping the concept of abstraction in mind, Marion [29] presented the possibility to reduce a self-modifying machine as a turing machine.

Laporte et al. [28] proved the possibility to disassemble self-modifying code using VSA. Their results are promising but their interpreter handles only some x86 instructions and tests have been done on handmade assembly code.

In some cases the result of an abstract interpretation can return the whole address space for a value. This is due to the over-approximation of the real value, and of course result in a lot of wrong target address. Kinder et al. [30] noted that alternating over- and under- approximation improves the results but can still remain too big in some real cases.

Symbolic evaluation and Abstract Interpretation are strongly related as both of them interpret the concrete instruction while keeping track of the possible values in an abstract memory/register representation. However both of them can return a huge set or interval of values for a target address, which even if it is bounded will result with a lot of more computation to determine if the included values are pertinent or not.

Abstract interpretation is well adapted to prove a property on a program rather than finding a specific value for a variable, e.g.: determining if the program has self-modifying behavior [13], or proved the compiled code is correct [7]. Symbolic evaluation seems more efficient when recovering CFG from binary file especially with the VAPR method. With such evolved static analysis instruction overlapping or indirect jump are less difficult to resolve however there is no ultimate solution.

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

Indeed some future viruses may wait for an external command coming from the network to get a value which could then be use in a jump. In this case it is not possible to statically find this value.

3.2 Dynamic Introspection

Static analysis shows its limit when dealing with indirect jump or self-modifying code. Those properties are strongly related to the effective execution of the program. The idea of Dynamic Introspection is to actually run the program in order to get an execution trace from it. Then this trace can be analyzed with already known static methods. This mixed between static methods and dynamic introspection is referred as Dynamic Symbolic Evaluation (DSE) [31] or Concolic Execution [6].

When dealing with self-modifying code especially packed code, it's useful to get an execution trace in order to get all the unpacking phases of the code [32]. Kand et al. [33] implement a shadow memory on which the executing code can write. When it does they mark bytes as dirty, and when a branch to a dirty bit is made then a snapshot of the memory is taken to keep track of the different phases. In CoDisAsm, Marion et al. [16] called this phases *waves*.

An other approach has been introduced by Chipounov and Candea [34, 35]. It aims to disassemble binary code to LLVM[9] assembly language in order to reuse analysis available with this compiler. Their work is based on the Klee[10] project and Qemu[11] which allow them to execute the binary code inside a symbolic virtual machine. However self-modifying was out of their scope.

In its Phd thesis, Calvet [36] showed the efficiency of dynamic analysis on malwares. However dynamic introspection has also its drawbacks. Usually the trace extraction is performed on a virtual machine, or on a machine in debug mode. As hackers are aware of such techniques they incorporate mechanisms to detect such situation, which are them cheated by researcher team [1]. This example is related to the main issue of dynamic introspection : the code coverage. Indeed there is no guarantee that the captured trace in one or several execution is very representative of the program behavior or that the execution trace will retrieve the whole binary code of the program.

4 Intermediate Language

Previous section explored the available methods to disassemble and reconstruct the CFG from a binary program. Most of the tools mentioned deal with an Intermediate Language (IL). This IL is used to abstract the Instruction Set Architecture (ISA) analyzed. It helps to build analysis that do not depend on the binary code but on the IL. Then adding support for a new ISA to the disassembler or the binary analysis platform is easy as only some kind of *driver* need to be developed to properly handle the architecture.

Dullien et al. [37] created an IL called REIL. It looks like the form of pseudo-assembly and can represent a small subset of x86 assembly. The advantage of their approach is the simplicity of expression types as they use only basic operands. However the lack of expressiveness requires more statement to translate complex instructions. Also they do not support floating arithmetic which can be problematic nowadays.

[9] `http://llvm.org/` LLVM project is a collection of modular and reusable compiler and toolchain technologies
[10] `klee.llvm.org`KLEE is a symbolic virtual machine on top of LLVM compiler infrastructure.
[11] `http://wiki.qemu.org/Main_Page` QEMU is a generic and open source machine emulator and virtualizer.

Brumley et al. [38] proposed a binary analysis platform with a powerful IL called BIL. It is designed to model all kind of processor behaviors. They convert assembly instruction not just in one other instruction but in a sequence of statement which represent all kind of flag modifications performed by a processor. Thus their IL is based on an operational semantic of ISA. However the platform they built around the IL uses a Linear sweep disassembler which we saw in previous section is not efficient with obfuscated code.

Cifuentes et al. [19] and Kinder et al. [27] used an IL based on Semantics Specification Language (SSL)[39] inspired on the grammar format description used by some compilers.

A much more graphical representation is presented by Bardin et al. [40] called Dynamic Bytes-vector Automate (DBA). They choose to use automate which is the closest representation from the processor behavior.

As intermediate representation comes from compiler, Chipounov and Candea [34, 35] used the LLVM assembly language[12]. The most challenge they faced was to retrieve variables type and structure from the binary as this information are needed by the LLVM IL.

In order to well represent instruction overlapping and packed executable code, the disassembler must not use its IL too early. Indeed if it first translate the binary code to the IL and then perform analysis from section 3 on the IL, it might be impossible to jump inside an already translated instruction. This issue might make REIL useless with our constraints. Also the IL must take care of the dynamic side imposed by self-modifying code, so DBA from Bardin et al or BIL from Brumley would be good choices.

Also the IL must be expressive enough to handle every particularities of every ISA. With that in mind, no one of the mentioned IL perfectly handle every thing. Most of them just handle a small set of an ISA, and some of them do not have floating unit. The LLVM IL might also be a good choice as it provides all features from high-level language. But some new instructions might be needed to handle self-modifying code, and some research must be done on type inference as required by the language.

5 Conclusion

Disassembling and reconstructing the CFG are two important parts in the process of decompilation. Both of them are related by the same issues *indirect branch*, *code obfuscation* and *self-modifying code*. Nevertheless some solutions exist and this document presented some methods and tools in order to accomplish this goal. However there is no unique working solution for every constraint and *Instructions Set Architecture*. About ISA, barely all tools try to be architecture independent by using an *Intermediate Language*. They transform the assembly code into an other representation in order to perform analysis on this one rather than on assembly code.

Available methods to disassemble binary code or reconstruct the CFG work but under a lot of constraints. Tools implementing them might not have been developed to study viruses and malwares as a primary goal. Indeed none of them are able to properly handle obfuscation or self-modifying code and security researchers prefer to use IDA Pro and perform hand-made analysis on real viruses [1, 2]. Current static methods help to resolve the issues mentioned above however they do not work on real virus cases.

Thus the conjunction of all the described methods seems to result in something more efficient. First running the viruses in a controlled environment to get an execution trace. Then using tech-

[12] `http://llvm.org/docs/LangRef.html`

niques from automatic test data generator in order to validate that there is no missing part in the trace (a full cover of the binary code). Then iteratively reconstruct the CFG from this full trace while using static analysis with parsimony just on small shadow zone to solve missing informations. In addition, the reconstructed CFG must not carry the initial assembly code, but an intermediate language. We have seen BAP [38], LLVM IL and DBA [23] are good candidates because of their expressiveness.

Acknowledgments

Thanks to my supervisor Jean-Yves Marion for his time and advices.

References

[1] G. Bonfante, J.-Y. Marion, F. Sabatier, and A. Thierry, "Analysis and diversion of duqu's driver," in Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on, pp. 109–115, IEEE, 2013.

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

[3] 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.

[4] G. Bonfante, J.-Y. Marion, and T. D. Ta, "Malware message classification by dynamic analysis," in The 7th International Symposium on Foundations and Practice of Security, vol. 8930, p. 16, Springer, 2014.

[5] E. R. Hanbing Li, Isabelle Puaut, "Traceability of flow information: reconsiling compiler optimizations and wcet estimation," 2014.

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

[7] X. Rival, "Abstract interpretation-based certification of assembly code," in Verification, Model Checking, and Abstract Interpretation, pp. 41–55, Springer, 2003.

[8] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," Journal of Computer Virology and Hacking Techniques, pp. 1–7, 2013.

[9] S. Rawat, L. Mounier, and M.-L. Potet, "Listt: An investigation into unsound-incomplete yet practical result yielding static taintflow analysis," in Availability, Reliability and Security (ARES), 2014 Ninth International Conference on, pp. 498–505, IEEE, 2014.

[10] G. Grieco, L. Mounier, M.-L. Potet, and S. Rawat, "A stack model for symbolic buffer overflow exploitability analysis," in Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pp. 216–217, IEEE, 2013.

[11] 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.

[12] J. Nagra and C. Collberg, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education, 2009.

[13] H. Cai, Z. Shao, and A. Vaynberg, "Certified self-modifying code," ACM SIGPLAN Notices, vol. 42, no. 6, pp. 66–77, 2007.

[14] C. Delannoy, Programmer en langage C++. Editions Eyrolles, 2011.

[15] G.-R. Uh and D. B. Whalley, "Effectively exploiting indirect jumps," Softw., Pract. Exper., vol. 29, no. 12, pp. 1061–1101, 1999.

[16] J.-Y. Marion, "Codisasm :a disassembly of self-modifying binaries with overlapping instructions," Challenges in Analysing Executables: Scalability, Self-Modifying Code and Synergy, p. 58.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labbé, "Binary-level testing of embedded programs," in Quality Software (QSIC), 2013 13th International Conference on, pp. 11–20, IEEE, 2013.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] J.-Y. Marion, "From turing machines to computer viruses," Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, vol. 370, no. 1971, pp. 3319–3339, 2012.

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

[31] 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.

[32] 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.

[33] 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.

[34] V. Chipounov and G. Candea, "Dynamically translating x86 to llvm using qemu," tech. rep.

[35] V. Chipounov and G. Candea, "Enabling sophisticated analyses of× 86 binaries with revgen," in Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on, pp. 211–216, IEEE, 2011.

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

[37] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," Proceeding of CanSecWest, 2009.

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

[39] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on, pp. 126–133, IEEE, 1998.

[40] 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.