

CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions

Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, Aurélien Thierry

▶ **To cite this version:**

Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, et al.. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. 22nd ACM Conference on Computer and Communications Security, Oct 2015, Denver, United States. 10.1145/2810103.2813627 . hal-01257908

HAL Id: hal-01257908
https://inria.hal.science/hal-01257908

Submitted on 18 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions

Guillaume Bonfante
Université de Lorraine, CNRS and INRIA

Jose Fernandez
Ecole Polytechnique de Montréal

Jean-Yves Marion
Université de Lorraine, CNRS and INRIA
Jean-Yves.Marion@loria.fr

Benjamin Rouxel
Université de Lorraine, CNRS and INRIA

Fabrice Sabatier
Université de Lorraine, CNRS and INRIA

Aurélien Thierry
Université de Lorraine, CNRS and INRIA

ABSTRACT

Fighting malware involves analyzing large numbers of suspicious binary files. In this context, *disassembly* is a crucial task in malware analysis and reverse engineering. It involves the recovery of assembly instructions from binary machine code. Correct disassembly of binaries is necessary to produce a higher level representation of the code and thus allow the analysis to develop high-level understanding of its behavior and purpose. Nonetheless, it can be problematic in the case of malicious code, as malware writers often employ techniques to thwart correct disassembly by standard tools.

In this paper, we focus on the disassembly of x86 self-modifying binaries with overlapping instructions. Current state-of-the-art disassemblers fail to interpret these two common forms of obfuscation, causing an incorrect disassembly of large parts of the input. We introduce a novel disassembly method, called *concatic* disassembly, that combines CONCrete path execution with stATIC disassembly. We have developed a standalone disassembler called *CoDisasm* that implements this approach. Our approach substantially improves the success of disassembly when confronted with both self-modification and code overlap in analyzed binaries. To our knowledge, no other disassembler thwarts both of these obfuscations methods together.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Security

Keywords

Disassembler; Malware; Dynamic Analysis; Overlapping Instructions; Self-Modifying Codes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813627.

1. INTRODUCTION

This paper focuses on malicious binary code, and more specifically x86-binaries. Nowadays there are two opposite core problems that we have to face in order to fight malicious binary code. On the one hand, each day a high volume of (executable) files are observed and processed. Google receives more than 300 000 files per day and has a collection of 400 million malware samples. All these files must be analyzed and classified in order to build defenses against malware threats. It is a necessity to devise tools that are able to correctly handle very large collections of machine code. On the other hand, malware is quite often well-crafted software that is heavily protected against analysis. As a result, accurate and automatic malware analysis represents a true challenge. Moreover, current tools available are not necessarily well adapted to process large amounts of code, because most of them are designed for reverse engineering and often involve complex computations. The main objective of this paper is to develop methods to disassemble and to construct control flow graphs of binary codes, that are robust and able to process efficiently a quite large amount of binary code.

Disassembly and pitfalls. Disassembly is the first step in the analysis of malware binaries and it is an essential one as all subsequent steps crucially depend on the accuracy of the disassembly. Indeed, it is from the disassembly of a binary that we can reconstruct the control flow graph (CFG) in order to perform further reverse engineering analysis tasks. It is also from the disassembly that we develop decompilers in order to extract relevant high-level semantic information. However, there are several inherent difficulties in devising a disassembly process. It has been reported [21] that up to 65% of the code is typically incorrectly disassembled. One difficulty is that it is almost impossible, to separate machine code from data. Both are mixed in a long sequence of bytes. Instructions such as `jmp` may skip data, jumping from one piece of code to another one. Moreover, these jumps are not statically predictable. An illustration of this fact is an indirect jump like the instruction `jmp eax`. To pursue static analysis, it is then necessary to determine the range of values in the register `eax`, or at least a good approximation of the `eax` values. It is worth noting that determining the destination of an indirect jump is undecidable, which implies that separating code from data is also an uncomputable task. Most previous work [31, 22, 33, 20, 19] has tried to solve the problem of indirect jumps employing static analysis methods. That said, there are other signif-

icant issues. In this paper, we focus on two of them: (i) self-modifying code, and (ii) overlapping instructions. Both obfuscation techniques are designed to protect code against human and automated analysis, and are in fact widespread in malware.

```
; data size
01006e62              mov ecx, 0x1dc2
; ebx is the pointer on the block to decrypt
; ebx=0x1005090
01006e67              inc ebx
01006e6f   loop:      rol byte ptr [ebx+ecx], 0x5
01006e73              add byte ptr [ebx+ecx], cl
01006e76              xor byte ptr [ebx+ecx], 0x67
01006e7a              inc byte ptr [ebx+ecx]
01006e7d              dec ecx
01006e80              jnle loop
; jump to the decrypted data
 01006e82             jmp 0x01005090
```

Figure 1: Decryption loop of tELock of data from address 0x01005090 to 0x01006e52

Nowadays, malware is almost always self-modifying. Generally, this kind of code protection consists of a sequence of complex and intertwined unpacking/decryption and protection routines. For example, the packer tELock 0.99 uses 18 layers to unpack and to protect the original code. In Figure 1, we present a simple —but commonly seen in malware— example of a decryption loop based dynamic analysis on a one-time pad cipher inside a layer of the packer tELock. The encrypted code is run after decryption at address `0x01005090`. Packers and malware authors protect in a very effective manner the original code by mostly avoiding potential dynamic analysis that attempt to analyse malware behavior. Commonly found protection methods may be quickly classified in two categories. The first category combines anti-debbuging, anti-virtualization, and anti-disassembly mechanisms in various forms in order to evade system monitoring. For example, the packer tELock contains several anti-debugging routines. The second category employs obfuscations and "code slicing" methods in order to reveal the original code. A packer has the ability to show just slices of the original code and to hide the rest of the code. For example, the packer ACProctect interleaves the original code with its code and unpacks library calls only when it is required.

Generic unpackers were not designed to deal with all these protections. Most of them [25, 17, 30, 13] perform dynamic analysis and have heuristics to find the unpacking layer that contains the original code. One exception is that of the static analysis-based unpacker proposed by Coogan *et al.* [9]. According to Ugarte *et al.* [32], generic unpackers can be deceived, and therefore fail, because (i) they rely on specific packer families, (ii) malware authors use handmade packers, and (iii) they are based on assumptions that are no longer valid. Indeed, nowadays packers combine the original code with their own code. As a result, the original code is often not totally available in memory and we cannot take a single memory snapshot to capture it. Moreover, packers may use several processes or threads to run the original code. For these reasons, we develop a dynamic analysis system to trace processes and threads. Thanks to our model of self-modifying code, we take a sequence of memory snapshots containing at least all the instructions of the original code that are executed.

The second issue concerns overlapping instructions, which is a typical feature of x86 machine code and a common anti-disassembling mechanism. Consider for instance the following execution sequence of bytes extracted from the packer tELock0.99

```
fe 04 0b eb ff c9 7f e6 8b c1
```

occurring in the code snippet in Figure 2. The correct control flow graph is given in Figure 3. The instruction at the

```
01006e7a    fe 04 0b        inc byte [ebx+ecx]
01006e7d    eb ff           jmp +1
01006e7e       ff c9        dec ecx
01006e80    7f e6           jg 01006e68
01006e82    8b c1           mov eax, ecx
```

Figure 2: Overlapping assembly in tELock0.99

address `01006e7d` is `jmp +1`. This instruction is encoded by two bytes and it jumps to the second byte of its opcode at address `01006e7d+1`, which corresponds to an instruction `dec ecx`. The opcode of `dec ecx` is `ff c9` which shares the byte `ff` at address `01006e7d+1` with the `jmp +1` opcode. As a result, both instructions `jmp +1` and `dec ecx` overlap each other.

The overlap is just there to obfuscate the code. Figure 4 displays the disassembly result respectively output by IDA Pro (v6.3) [11] which is incorrect. The reason is that off-the-shelf disassemblers make the assumption that instructions do not overlap and so misinterpret the execution sequence above. There is one important exception: the Jakstab disassembler proposed by Kinder [18], which handles overlapping instructions but not self-modification.

It is worth mentioning that tELock combines self-modification and overlapping instruction obfuscation techniques. For explanatory reasons, we choose to separate and display them in two independent snippets in Figure 1 and 3.

Objectives. The first objective of our work is to devise a disassembler of x86-malware code. Inputs are stripped binary code, with no information of any kind, and that are usually heavily obfuscated. In particular, we have focused on (i) self-modifying binaries, and (ii) on binaries containing overlapping instructions. An important point is that we make the assumption that slices of the original code may be executed in any wave (i.e. unpacking layer) when the

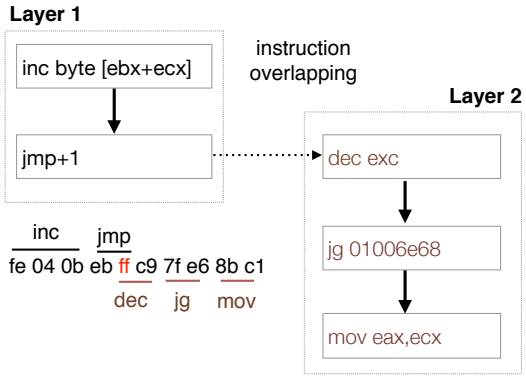


Figure 3: Control flow graph for the tELock sample

```
01006E7A inc byte ptr [ebx+ecx]
01006E7D jmp loc_1006E7D+1
01006E7D ; ---------
01006E7F db 0C9h ;
01006E80 db 7Fh ;
01006E81 db 0E6h ;
01006E82 db 8Bh ;
01006E83 db 0C1h ;
```

Figure 4: Disassembly of tELock example with IDA Pro (v6.3)

analyzed code is packed. In other words, we do not assume that the original code is entirely visible at some point of the unpacking process. The second objective is to develop an effective complete disassembly architecture that is able to automatically process each binary code file in a reasonable amount of time.

The concept of code waves. We consider that each input binary is self-modifying code. That is, the execution of a binary will usually deploy different waves of code. Thus, an execution might be viewed as a sequence of waves, where a wave is produced by previous waves. Most of the time, a wave is produced by unpacking or by decrypting some data. For example, tELock generates 18 waves and the misalignment given in Figure 3 occurs at Wave 3. Each wave is determined by an execution level. We begin with Wave 1 in which the starting code is run. Then, there is Wave 2 for which the executed code has been written by Wave 1. Next, the process repeats itself and switches from Wave k to $k+1$ each time we run data written during Wave k. Notice that the code run at Wave $k+1$ can be generated by several previous waves (not only by Wave k).We found such an example in the packer UPolyX, which we observed in a Hupigon sample: `hupigon.eyf`. The execution of UPolyX consists of a first wave that generates a second unpacking routine and part of the payload. The second wave starts with the execution of the second unpacking routine that calls the first unpacking routine and generates the remainder of the payload. Finally, the third wave is triggered and executes the payload. In this example, we see that the payload is written at Waves 1 and 2.

Our model is closely related to the one suggested by Guizani *et al.* [12], which is why we use the same terminology. The main difference is that we simplify the wave computation and we use a monotonic numbering, which allows us to take a memory snapshot at the right time to dump a wave. Debray and Patel [10] define the notion of phase. The definition of a phase is closely related to the notion of wave. Dalla Preda *et al.* [28] define a fixed-point semantics of self-modifying programs, which is also similar.

The method in a nutshell. The disassembly method proceeds in two steps. In the first step, we perform dynamic analysis. We instrument a binary (see Section 3.2), and run it in a sandbox. The code instrumentation is able to bypass some anti-analysis evasion mechanisms. We follow threads and processes created by the binary by instrumenting them on the fly. We collect execution traces of all threads and processes. Then, we determine a sequence of waves as explained in Section 3 and we take memory snapshots to disassemble the code of each wave in the second step.

In the second step, described in Section 4, we disassemble each wave. Each wave provides a memory snapshot and a (sub-)trace. This step consists in identifying and in disassembling the code in each wave with the executed trace as a hint. For this, we implement a recursive disassembler that follows the trace. The trace indicates a sequence of executed addresses but of course the memory snapshot of the wave also contains "dormant" instructions, that have not been executed but that will also be disassembled by CoDisasm Nonetheless, this trace will be our guide to perform a recursive traversal disassembly. In fact, the instruction addresses gathered in the trace are starting points for disassembly. For example, the packer PE Spin has 58 indirect jumps which are immediately solved by using the trace.

We now have to face the second issue: overlapping instructions. Our approach is to split the memory analyzed in layers. Each layer corresponds to an overlap. To illustrate this idea, let us go back to the tELock example. The memory is constituted of 10 bytes: `fe 04 0b eb ff c9 7f e6 8b c1`. As shown in Figure 6, this defines two layers `fe 04 0b eb ff` and `ff c9 7f e6 8b c1`. Our approach can also thwart obfuscations such as those shown by Jämthagen *et al.* [16].

Other obfuscation techniques may be also resolved from the trace. For example, a trace gives the return address of a call even if the return address has been modified. Notice that this information is easily available in dynamic analysis, unlike in static analysis.

The results. We propose a simple model of self-modifying program executions, dubbed *wave semantics*, that allows us to reconstruct the original code. We also generalize the notion of control flow graph to deal with self-modifying code with overlapping instructions.

From this model, we have developed a two step disassembler called CoDisasm. In the first step, CoDisasm collects an execution trace of a stripped binary. This trace is analyzed and split into code waves. At the end, CoDisasm outputs a set of layers for each wave, where each layer contains a set of non-overlapping instructions. From this set of layers, we reconstruct an enhanced control flow graph (see Section 4.3). Next, we can apply other techniques to discover new pieces of code thus obtaining a speculative disassembly of the code (see Section 4.4).

CoDisasm overview The overall architecture of our disassembler CoDisasm is shown in Figure 5. The CoDisasm disassembler performs a static disassembly along with a concrete execution with the aim of maximizing coverage. It includes two main components:

1. A dynamic analysis component that collects execution traces of the threads and processes of a binary run. For this, we developed *Pin_tracer* to instrument code, which is based on Pin [24]. We recover each code wave by taking a memory snapshot at the beginning of the wave, as explained in Section 3. A Portable Executable (PE) file is then built for each snapshot.

2. From the execution traces, each memory snapshot is disassembled following the algorithm described in Section 4, taking care of overlapping instructions. At the end, we have a sequence of disassembled code waves, which corresponds to the code discovered thanks to the set of collected traces.

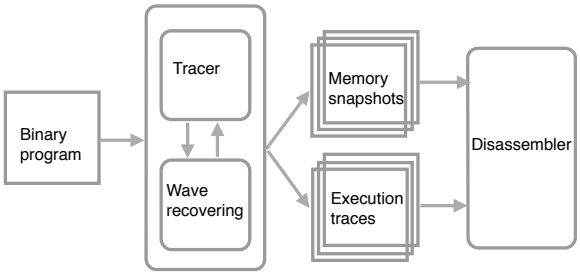


Figure 5: CoDisasmdisassembler architecture

CoDisasm is available at `http://www.lhs.loria.fr` in the research pane. In our lab, CoDisasm was deployed on 100 virtual machines and so 500 malware are analyzed in less than 20 minutes.

We do think that the combination of dynamic and static disassembly may be of particular use on malware. That is why, we have called this approach *Concatic* which stands for *Conc*rete executions and st*atic* disassembly. To support this claim we report on some tests and evaluations we have made of CoDisasm in Section 5. We discuss limitations of our approach 6 and then conclude in Section 7.

2. RELATED WORK

Most of the previous work on disassembly has focused on static disassembly. A static disassembler extracts the assembly code from a binary without running it. Traditionally, two methods of static disassembly have been employed. The first method is a linear sweep of the memory. It is used by several link-time optimizers and by the GNU utility tool `objdump`. This method consists in parsing the memory from an entry-point, opcode by opcode. The main drawbacks of this simple method are that (i) the data within the analyzed binary is interpreted as opcodes, potentially leading to an incorrect result, and (ii) it does not handle overlapping codes.

The second method is a recursive traversal of the code, which consists in examining and following each instruction's successor. When a conditional jump is encountered, like a `jcc`, both branches are parsed. Thus, a priori, data is not misinterpreted. However, in certain cases, and in particular in malware analysis, the control flow can be obfuscated and this might be done in several different ways. As a result, Moser, Kruegel and Kirda [26] showed that commercial virus scanners do not detect many known viruses that are obfuscated by these methods. Linn and Debray [23] suggest the insertion of opaque predicates [8] so that only one branch of a conditional jump is taken, since the other branch may contain junk code which might be taken as valid by a recursive disassembler. Indirect jumps [23] are another way to thwart disassembly. As a consequence, most of the disassemblers are hybrid, that is, they use both methods: linear sweep and recursive traversal. In the case of indirect jump obfuscations, Schwartz, Debray and Andrews [31] propose a method to recover jump tables in order to identify all possible jumps. However, symbol tables and relocation information are not always easily obtainable. Lastly, the insertion of junk code after a call, like explained in [23], can fool disassemblers. Kruegel *et al.* [22] developed a disassembly heuristic by parsing the memory byte by byte in order to construct all possible CFG with their relationships. Then, they state several principles in order to rule out certain CFG, like the fact that the parsed code never contains overlapped instructions.

Another direction is to identify the values of a register by static analysis. As a result, we may expect to find the range of an indirect jump or to identify the return address on the stack in order to determine where a `ret` return instruction will go. This kind of analysis is based on a combination of methods like program slicing, constant propagation and abstract interpretation, which are now quite mastered for high-level programming languages. The adaptation of static analysis to binaries turns out to be difficult because most of the assumptions on which formal methods lean, are violated in machine code. For low-level programming languages, Reps *et al.* [29] developed CodeSurfer, a tool that computes an approximation of the register values by using the value-set-analysis algorithm. Kinder and Veith [20] developed Jakstab based on data flow analysis together with control flow reconstruction. More recently, Bardin, Herrmann and Védrine [1] combine advantages of both of these methods. An interesting new direction has been proposed by Ogawa *et al.* [15], who suggest a pushdown model of assembly language to determine register values thanks to SMT solvers. In all cases, static analysis tools build an over-approximation which is often too imprecise and reports a lot of false positives. That is why we have taken a more pragmatic path by combining dynamic and static disassembly methods.

We are aware of a few other approaches that combine static and dynamic disassembly. Kinder and Kravchenko [19] propose to narrow the search space of static analysis by using traces. Their goal is to improve Dynamic Symbolic Evaluation (DSE). The dynamically collected information helps the symbolic step, for example by suggesting relevant approximations by concretization. They do not address the problem of self-modification. The platform BIRD of Nanda *et al.* [27] apply speculative disassembly by mixing static and dynamic techniques. The difference with our work is that BIRD is designed for non-obfuscated binaries. Caballero *et al.* [2] developed a similar disassembly process in order to extract input/output interface to reuse binary functions. Their approach uses a dynamic analysis that collects a trace and a memory dump and then they disassemble the memory dump using trace information. The difference with our work is that their disassembly process is based on a single memory snapshot and does not handle overlapping instruction.

3. SELF-MODIFYING CODE

3.1 Wave semantics

When analyzing binary code, we are faced with code that is, most of the time, self-modifying. This is why, we propose a model of execution of self-modifying code based on code *waves*. The idea is to associate, at any time, and to each memory address a write level and an execution level. At the beginning, for every address the execution level is set to 1 and the write level to 0. Every data written by an instruction of execution level 1 increases its write level to 1. This typical situation corresponds to an unpacker, which

Addresses 0x01006e	7a	7b	7c	7d	7e	7f	80	81
Bytes	fe	04	0b	eb	ff	c9	7f	e6
Layer 1 @0x01006e7a	inc [ebx+ecx]			jmp +1				
Layer 2 @0x01006e7e					dec ecx		jg 0x1006e68	

Figure 6: Layers of a subset of the TELock code segment

decompresses some piece of data. In our model, data are decompressed in a memory area and so each address of this area gets write level 1. Then, the unpacker transfers on-the-fly the control to the "decompressed" data. As a result, data at write level 1 is executed, thus triggering the second wave of execution, and we set the execution level to 2. In turn, Wave 2 may generate a third wave and this process may repeat.

We define *Wave k* as the whole set of instructions, executed or not, which are present when the execution level reaches k (See discussion in Section 6). As a result, we can see a run of a program as a sequence of waves. Notice that in this model, non-self-modifying code will only have one wave.

The rationale behind the model of a self-modifying code run as a sequence of waves is that we can extract a snapshot of the memory at the beginning of each wave from the execution of a binary. This snapshot contains all the instructions deployed by the binary to run this wave and possibly some silent code. Our objective is then to disassemble this memory snapshot in order to recover the assembly code contained in a wave.

The wave semantics that we propose is defined at the lowest possible level of abstraction in the sense that we see all computations inside the system through the eyes of the single core processor. Consequently, this model takes into consideration threads and processes.

3.2 Collecting execution traces

In practice, we focus exclusively on Windows/x86 binaries. To this end, we use Pin which is a dynamic instrumentation framework supported by Intel [24]. We developed and used a Pin tool, that we refer to as *Pin_tracer*, to collect execution traces of x86-code. *Pin_tracer* is able to trace newly created threads and processes. It also tries to detect code injection in a running process. If such an event occurs, it instruments the injected process. For example, the driver of Duqu illustrates this mechanisms by injected in memory within `service.exe`. In this case, *Pin_tracer* traces `service.exe`. Code injections are detected by monitoring calls to the `CreateRemoteThread` and `CreateRemoteThreadEx` functions from the Windows API. When *Pin_tracer* detects a new process, then a new pin tool is attached to this process. Thus, a new trace is generated. Finally, we collect all traces of the threads and processes detected.

Given the fact that many malware use anti-emulation, anti-debugging and anti-virtualization techniques, including on Pin, we built some anti-evasion functionality into *Pin_tracer*. In particular, we attempt to cover the following evasion techniques:

- Time check, to verify whether or not the malware code is monitored.
- EIP (instruction pointer) check, to verify whether or not the malware code has been instrumented.
- Checksum checks (CRC) on parts of the code, to check whether or not it has been altered.
- Use of interrupt table manipulating instructions such as SIDT, SLDT, etc., in order to check whether or not the code runs in a virtual machine.

In each of these cases, the counter-measure implemented is to return the expected value, which is not always possible to determine.

Regardless of the method of collection, execution traces are important tools in reverse engineering that we use as an enabler to thwart code protections. We therefore need to formalize the notion of execution traces as a basis for reasoning on self-modification behaviors. An execution trace is a sequence of operations performed by a program, where at each step, we gather a sequence of information such as process IDs, register values and read/write memory addresses that we collectively refer to as as *dynamic instruction*. A dynamic instruction $\mathbf{D}$ is a tuple composed of:

- a memory address $\mathcal{A}[\mathbf{D}]$,
- the machine instruction $\mathcal{I}[\mathbf{D}]$ run at address $\mathcal{A}[\mathbf{D}]$,
- the set $\mathcal{W}[\mathbf{D}]$ of memory addresses written by the instruction $\mathcal{I}[\mathbf{D}]$.

An *execution trace* is a finite sequence $\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$ of dynamic instructions. Figure 7 shows the dynamic trace of the program in Figure 1 after two iterations.

Algorithm 1: Computation of execution and write levels

```
Update(X,W,D)
    X ← max(X, W(A[D]) + 1) ;
    foreach m ∈ W[D] do
        W(m) ← X ;
    end
    return (X, W)
```

$\mathbf{W}(\mathcal{A}[\mathbf{D}])$ is a shortcut for $\max(\mathbf{W}(\mathcal{A}[\mathbf{D}]), \ldots, \mathbf{W}(\mathcal{A}[\mathbf{D}]+k))$ where k is the number of bytes encoding the instruction.

3.3 Execution and write levels

The goal of this section is to delineate waves inside an execution trace. A wave is determined from both (i) the execution level, and (ii) the write level of each memory address. The write level of each memory address is stored into a finite mapping $\mathbf{W}$ that we call the *write level table*.
Given an execution trace $\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$, we define a sequence of pairs composed of the execution level and the write level table $(\mathbf{X}_0, \mathbf{W}_0), (\mathbf{X}_1, \mathbf{W}_1), \ldots, (\mathbf{X}_n, \mathbf{W}_n)$ for each dynamic instruction that satisfies the following properties: (i)

Figure 7: Trace execution of the tELock snippet shown in Figure 1

$\mathcal{A}[\mathbf{D}]$	$\mathcal{I}[\mathbf{D}]$	$\mathcal{W}[\mathbf{D}]$	$\mathcal{A}[\mathbf{D}]$	$\mathcal{I}[\mathbf{D}]$	$\mathcal{W}[\mathbf{D}]$
01006e62	mov ecx, 0x1dc2		01006e7d	dec ecx	
01006e67	inc ebx		01006e80	jnle loop	
01006e6f loop:	rol byte ptr [ebx+ecx], 0x5	0x01006e52	01006e6f loop:	rol byte ptr [ebx+ecx], 0x5	0x01006e51
01006e73	add byte ptr [ebx+ecx], cl	0x01006e52	01006e73	add byte ptr [ebx+ecx], cl	0x01006e51
01006e76	xor byte ptr [ebx+ecx], 0x67	0x01006e52	01006e76	xor byte ptr [ebx+ecx], 0x67	0x01006e51
01006e7a	inc byte ptr [ebx+ecx]	0x01006e52	...	...	

Before executing the dynamic instruction $\mathbf{D}_{i+1}$, the the execution level is $\mathbf{X}_i$ and the write level table is $\mathbf{W}_i$ and (ii) after executing $\mathbf{D}_{i+1}$, the execution level is $\mathbf{X}_{i+1}$ and the write level table is $\mathbf{W}_{i+1}$. We shall say that the *execution level* of the dynamic instruction $\mathbf{D}_{i+1}$ is given by $\mathbf{X}_{i+1}$. The sequence of execution levels and the write level tables are obtained by iteratively applying the function `Update` shown in Algorithm 1.

We have defined the list of pairs (execution level, write level table) for explanatory reasons, but in fact, the execution level is shared by the entire memory. In fact, the execution level is shared by any memory address executed in a wave. Thus, it is sufficient to keep track of the current execution level and the write level table. Consequently, we begin by setting all write levels to 0 and the execution level to 1. That is, $\mathbf{W}(m) = 0$ for each memory address m and $\mathbf{X} = 1$. Then, we apply the function `Update` on arguments $(\mathbf{X}, \mathbf{W})$ and $\mathbf{D}$ in order to determine the next execution level and the next write level table: $(\mathbf{X}, \mathbf{W}) = \texttt{Update}(\mathbf{X}, \mathbf{W}, \mathbf{D})$.

Algorithm 2: Wave recovery for self-modifying codes

input : PE File
output: The number of waves $\mathbf{X}$ and for each wave, a snapshot and a trace in the lists traceList and waveList

```
Wave_ recovery()
    foreach address m do
    |  W(m) ← 0
    end
    X ← 0 ;
    trace ← ∅ ; list of dynamic instructions
    traceList ← ∅ ; list of traces
    waveList ← ∅ ; list of memory snapshots
    wave ← Snapshot() ;
    Add (waveList,wave) ;
    Computation of subtraces and memory snapshots
    while not at end do
        D ← Pin_Tracer() ;
        Add(trace, D) ;
        (X, W) ← Update(X, W, D) ;
        ip ← Pin_Next_Instruction() ;
        if W(ip) ≥ X then
            New wave
            Add (traceList,trace) ;
            trace ← ∅ ;
            wave ← Snapshot() ;
            Add (waveList,wave) ;
        end
    end
    return (X, waveList, traceList)
```

3.4 Reconstructing waves from a trace

We are now ready to split an execution trace into subtraces depending on their execution levels. From an execution trace $\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$, we have previously described how to compute the sequence of execution levels $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$. It is not difficult to see that this sequence is weakly monotonic, that is $\mathbf{X}_i \leq \mathbf{X}_{i+1}$ for all $i = 1, n$. The number of waves observed in this execution trace is $K = \max_i(\mathbf{X}_i) = \mathbf{X}_n$. In other words, in our model of self-modification, there are $K - 1$ successive code self-modification in this execution. As a result, we can extract K sub-traces of dynamic instructions, which are defined as follows:

$$\begin{aligned}
\texttt{trace}(1) &= \mathbf{D}_1, \ldots, \mathbf{D}_{\ell_1 - 1} \\
&\quad \text{where } \mathbf{X}_j = 1 \text{ for } j = 1, \ell_1 - 1 \\
\texttt{trace}(i) &= \mathbf{D}_{\ell_i}, \ldots, \mathbf{D}_{\ell_{i+1} - 1} \\
&\quad \text{where } \mathbf{X}_j = i \text{ for } j = \ell_i, \ell_{i+1} - 1 \\
\texttt{trace}(K) &= \mathbf{D}_{\ell_K}, \ldots, \mathbf{D}_n \\
&\quad \text{where } \mathbf{X}_j = K \text{ for } j = \ell_K, n
\end{aligned}$$

At the same time, we can also take a memory snapshot at the beginning of each wave. Thus, we have K memory snapshots. Let us call $\texttt{wave}(i)$ the memory snapshot at the beginning of the i-th wave, that is before executing instruction $\mathbf{D}_{\ell_i}$. Notice that the memory snapshot $\texttt{wave}(1)$ is the snapshot of the starting code. As a result, our model of self-modification ensures that the memory snapshot $\texttt{wave}(i)$ contains any instruction in the trace $\texttt{trace}(i)$ and probably also other dormant instructions that we have to identify.

3.5 Overview of the wave recovery algorithm

The wave recovery algorithm is presented in Algorithm 2. The input is a PE file that is loaded into memory. The first step of the algorithm initializes the write level table and takes an initial snapshot of the memory $\texttt{wave}(1)$. The first memory snapshot contains all the code and data that are inside the PE file sections.

In a second step, the *Pin_Tracer* executes code one statement at a time as explained in Section 3.2. *Pin_Tracer* runs one instruction and, at each step, gathers the corresponding dynamic instruction. Then, *Pin_Tracer* computes the write level table as described in Section 3.3. The index of the current wave is given by $\mathbf{X}$. We also gather all instructions executed during the current wave in the list $\texttt{trace}(\mathbf{X})$.

Finally, we determine the address of the next instruction to be executed thanks to a Pin tool called *Pin_Next_Instruction*. If the execution level of the next instruction increases, then we know that Wave $\mathbf{X}$ ends there and that Wave $\mathbf{X} + 1$ will start as soon as the next instruction will be executed. Therefore, we take a memory snapshot $\texttt{wave}(\mathbf{X} + 1)$ of the memory before the beginning of Wave $\mathbf{X} + 1$. Otherwise, we stay in the same wave and binary execution is resumed.

A memory snapshot combines (i) the code and data in a PE file, and (ii) all data stored in dynamically allocated memory areas (e.g. malloc). It is necessary to consider dynamic memory allocations because it is possible to jump into data that, for example, comes from a decryption loop.

3.6 Example

The introductory example (Figure 1) presents a decryption loop that generates two waves. The first wave mainly consists of the loop and `trace`(1) is composed of the nine dynamic instructions in the interval $[01006e62, 01006e82]$. The second wave is triggered when the condition at address `01006e80` is false and the control is transferred to the address `01005090`. `trace`(2) is composed of the dynamic instruction in the interval $[01005090, 01006e52]$. Figure 8 illustrates the execution of this example and provides the execution level $\mathbf{X}$ and the write level table $\mathbf{W}$. For example, take instruction `xor byte ptr [ebx+ecx], 0x67`. The execution level is 1. This instruction performs a memory write at the address pointed by the value of `ebx+ecx`. Since the value of `ebx+ecx` for that execution is $01006e82$, we set $\mathbf{W}(01006e82) = 1$.

3.7 Disassembly completeness

A discussion on disassembly completeness may seem quite theoretical at first glance. Nevertheless, it is a necessary digression in order to be able to discuss disassembler evaluation criteria in Section 5. We now put forth a definition of a semantics for self-modifying programs. In Section 3.4, an execution trace $D = \mathbf{D}_1, \ldots, \mathbf{D}_n$ defines $\texttt{trace}(i)$ corresponding to the instructions run in the i-th wave. We call each subtrace a code wave. The set of all code waves of a trace D is $\texttt{trace}(D) = \{\texttt{trace}(i) \mid i = 1..K \text{ where } K \text{ is the last wave}\}$.

We define the *wave semantics* of a given binary as a graph $G = (V, E)$ defined as follows. The set of vertices V is the set of all code waves for any execution trace, that is

$$V = \bigcup_{\text{for all traces } D} \texttt{trace}(D)$$

Two vertices W and W' are connected, that is $(W, W') \in E$ if W and W' are two consecutive subtraces of a trace. In other words, there is an execution where the successor of the last instruction of W is the first instruction of W', or yet if the wave denoted by W jumps to the wave denoted by W' in some execution.

As a result, the wave semantics is a graph G that represents all possible self-modifications of a binary and encodes all possible execution paths. The wave semantics of a binary provides the partially ordered list of all instructions that can be run. For that reason, the wave semantics G could be used to mesure the correctness of a disassembler (of self-modifying programs), because a perfect disassembly of a binary should be able to reconstruct the graph G. Of course, a perfect disassembler does not exist because the problem of disassembling is undecidable; and from this fact, the wave semantics is uncomputable. Any disassembler provides an approximation of the wave semantics. So at least from a theoretical point of view, the distance with the wave semantics may provide a metric to evaluate disassemblers.

4. OVERLAPPING INSTRUCTIONS

We now have all the necessary information to start the second phase, which consists in disassembling the code of a wave from a snapshot of the memory together with the trace of the wave. Recall that inside a wave, there is no code self-modification. However, other obfuscations may occur, in particular x86 overlapping instructions. In this section, we address this issue. We present a recursive algorithm that statically disassembles and correctly handles overlapping instructions.

Algorithm 3: Recursive disassembler, recovering overlapping instructions

```
input  : The memory snapshot of a wave, its execution
         trace and an empty set of layers
output: A set of layers resulting from the disassembled
         wave
disassembler(wave,trace)
    L← New();
    Set_layers ← {L};
    foreach addr ∈ trace do
        if addr ∉ Set_layers then
            if the addr has not been processed ;
            Set_layers ← recursive_traversal(wave, addr,
            Set_layers,L);
        end
    end
    return Set_layers

recursive_traversal(wave, addr, Set_layers,L)
    opcode ← disasm(wave, addr);
    if (addr,opcode) is aligned with L then
        Add the instruction to an aligned layer;
        Add( L,addr,opcode);
    else
        Create a new layer with the instruction;
        Lnew = New() ;
        Add(Lnew,addr,opcode);
        SetUnion(Set_layers,Lnew);
    end
    foreach successor of (addr,opcode) do
        Set_layers ← recursive_traversal(wave,successor,
        Set_layers,L)
    end
    return Set_layers
```

4.1 Layers

Two dynamic instructions overlap when they share at least one byte in memory. We will say that a set of dynamic instructions is mis-aligned if at least two instructions overlap. Otherwise, we will say that the instructions of this set are aligned. Take again the teLock snippet in Figure 2 and look at Figure 6. The instructions `jmp +1` and `dec ecx` have the byte at address `0x01006e7e` in common. So, they are overlapping instructions. Both overlapping instructions create two sequences of aligned dynamic instructions. Each sequence forms what we will call a layer.

Before we define layers, we have to introduce the notion of connected instruction set. A set L of instructions is *connected* if given two instructions $\mathbf{D}$ and $\mathbf{D}'$, there is a path between $\mathbf{D}$ and $\mathbf{D}'$ composed of instructions in L. That is, there is a sequence $\mathbf{D} = \mathbf{D}_1, \ldots, \mathbf{D}_n = \mathbf{D}'$ of instructions in L such that the instruction $\mathbf{D}_{i+1}$ is a successor of $\mathbf{D}_i$. The successors of the instruction $\mathbf{D}$ are all the reachable instructions from $\mathbf{D}$ that we can predict. For example, a sequential

After several iterations of the loop			
$\mathcal{A}[\mathbf{D}]$	$\mathcal{I}[\mathbf{D}]$	**W**	**X**
01006e62	mov ecx, 0x1dc2	0	1
01006e67	inc ebx	0	1
01006e6f loop:	rol byte ptr [ebx+ecx], 0x5	0	1
01006e73	add byte ptr [ebx+ecx], cl	0	1
01006e76	xor byte ptr [ebx+ecx], 0x67	0	1
01006e7a	inc byte ptr [ebx+ecx]	0	1
01006e7d	dec ecx	0	1
01006e80	jnle loop	0	1
01006e82	jmp 0x01005090	0	1
0x01005090	decrypted byte	1	_
0x01005091	decrypted byte	1	_

Wave 1: `trace`(1) instruction in $[01006e62, 01006e82]$

After transferring the control to 01005090			
$\mathcal{A}[\mathbf{D}]$	$\mathcal{I}[\mathbf{D}]$	**W**	**X**
01006e62	mov ecx, 0x1dc2	0	_
01006e67	inc ebx	0	_
01006e6f loop:	rol byte ptr [ebx+ecx], 0x5	0	_
01006e73	add byte ptr [ebx+ecx], cl	0	_
01006e76	xor byte ptr [ebx+ecx], 0x67	0	_
01006e7a	inc byte ptr [ebx+ecx]	0	_
01006e7d	dec ecx	0	_
01006e80	jnle loop	0	_
01006e82	jmp 0x01005090	0	_
0x01005090	decrypted byte	1	**2**
0x01005091	decrypted byte	1	**2**

Wave 2: `trace`(2) instructions in $[01005090, 01006e52]$

Figure 8: The two waves generated by the example in Figure 1

instruction like `mov eax,ebx` has one successor which is the next instruction, while `jnz 100` has two successors: the instruction at address 100 and the next one. On the other hand, we may not be able to determine the successor of an instruction like `jmp eax` if we have no certain value for the register `eax`.

We now come to the second key notion. A *layer* L is a set of dynamic instructions that satisfies the following two properties: (i) two instructions in L never overlap, and (ii) the set L is connected. Our objective is to construct a set of layers that approximates the code inside a wave.

4.2 Disassembling algorithms

Algorithm 3 defines the disassembly procedure. Its inputs are a memory snapshot `wave` of a given wave and its corresponding sub-trace `trace`. Both inputs come from the first phase that we have presented in the previous section. The algorithm inspects recursively the memory snapshot `wave` from each address in the trace. For this, we begin with a new empty layer. We disassemble recursively and we add instructions to a layer in a consistent way. That is, we guarantee that layers are always a sequence of aligned instructions. When an instruction cannot be added to a layer in a consistent way, that is, if the instruction overlaps at least one other instruction in one of the already computed layers, we create a new layer. We add the misaligned instruction to the new layer. The new layer is added to the current set of layers. As a result, we maintain during the disassembly a set of coherent layers, such that: (i) no instruction inside the layer overlaps another instruction in the same layer, and (ii) if we take two layers in this set, then there are at least two instructions from each layer which are mis-aligned. The output is a set of coherent layers that together form an under-approximation of the complete disassembled code inside a wave.

Notice that this algorithm follows all found execution paths. For example, when a conditional instruction like `jcc` is encountered, we follow both successors. Moreover, the trace gives us some valuable additional information. For example, Linn and Debray [23] propose to modify the return value on the stack of a call as an obfuscation technique. In this case, the trace immediately gives the correct return address and thus provides a correct answer to this common technique.

4.3 Recovering an enhanced CFG

From each layer, we reconstruct a control flow graph (CFG) that we call pre-CFG. Since each layer is a connected set of instructions, each pre-CFG is a connected graph. All pre-CFG are connected together. Indeed, there is at least one edge between a node of a pre-CFG and the root of another pre-CFG, which comes from the instruction that creates the overlap. Finally, a node can have multiple incoming edges which corresponds to a resynchronization of the code.

We illustrate and sum-up the construction by an example coming from the packer UPX. In Figure 9, we show the two layers created by the conditional jump `jnz +9`. Therefore, there are two pre-CFG that correspond to both layers generated by UPX. The dashed edge corresponds to the instruction overlap due to the `jnz +9` instruction. The code resynchronizes at the `push ebp` instruction.

4.4 Speculative disassembly

At the end, we perform a speculative disassembly by running a linear sweep on unexplored pieces of memory, byte by byte as Vigna [33] proposes. In order to identify valid layers, that is to separate code from data, we apply well-known heuristics employing pre-determined scoring [22] and statistical methods [21]. Finally, the trace is taken into account to evaluate the probability of correctness of each reconstructed disassembly.

5. EVALUATION

5.1 Methodology

In order to evaluate a disassembler, it is necessary to define what we expect to be a correct output of a disassembler. In the case of a regular binary code produced by a compiler, it is sufficient to compare the disassembler output with the compiler assembly output. But in the case of a heavily obfuscated binary code like a malware, the evaluation of a disassembler is a non-trivial problem that presents complex challenges.

Before going further, we need to discuss what we mean by a "correct disassembler". A correct disassembler should only output instructions which are in a possible execution path, that is an approximation of the wave semantics as already defined and discussed in Section 3.7. Recall that the wave semantics of a binary code provides the set of all instructions that can be run. Thus, it is important to measure the approximation obtained with respect to the wave semantics in order to determine the code coverage.

This response is not completely satisfactory. For example, a malware may be packed and in this case one might be in-

Addresses	0xf2	0xf3	...	0xf9	0xfa	0xfb	0xfc	0xfd	0xfe	0xff
Bytes	79	07	...	47	b9	57	48	f2	ae	55
Layer 1 @0xf2	jns +9 (0xfb)		...	inc edi	mov ecx, aef24857					push ebp
Layer 2 @0xfb						push edi	dec eax	repne scasb		

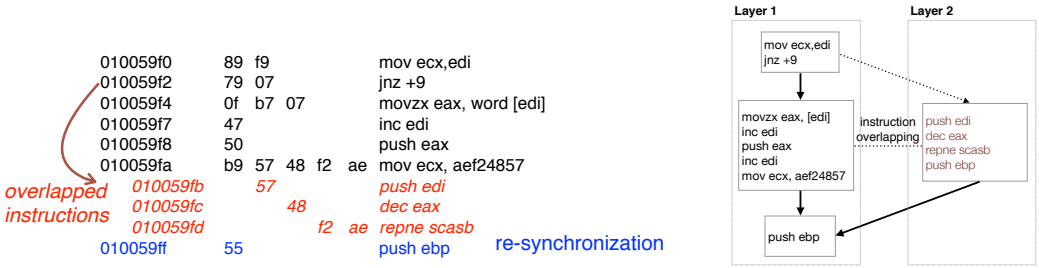


Figure 9: Overlapping : UPX case

terested in reconstructing the assembly code of the malware. The assembly code reconstruction of a packed malware is a different issue than the one studied in this work. Indeed we may for example develop a packer in which the malware functionalities and the protection functionalities are fully intertwined. In this case, malware functionality identification and reconstruction is a research subject *per se*. For example, the work of Yadegari *et al.* [34] developed a de-obfuscation method to extract a simplified code. For this, they combined a dynamic analysis with concolic executions in order to collect several traces, which are simplified in order to reconstruct a control flow graph.

Another approach is to compare disassemblers. The comparison between disassemblers is currently difficult because there is no benchmark based on obfuscated binary codes. In particular, it makes no sense to compare CoDisasm with off-the-shelf disassemblers because none deal with self-modifying code and overlapping instructions.

As such, it is difficult to assess a disassembler and we recognize that we have not been able to define a metric that allows us to adequately determine code coverage for CoDisasm, or for any other disassembly tool, for that matter. That is why, we propose a fourfold evaluation of CoDisasm focusing on testing functionality and usefulness of the tool and showing that there is no major operational problems with the tool or its approach. First in Section 5.2, we check that CoDisasm correctly retrieves the code of a regular binary produced by a compiler. Second in Section 5.3, while we cannot verify its correctness on malware for which source code is normally not available, we verified the relevance of our approach by running the tool on 500 malware families, and observing the number of waves and layers; more precisely, we were able to deduct that tools not handling self-modification and code overlap simultaneously would have failed to correctly disassemble the majority of those samples. Third, in Section 5.4, we successfully benchmarked CoDisasm by packing known applications with 28 different readily available packers and retrieving these known applications. Finally in Section 5.5, we illustrated CoDisasm's capacity with malware analysis by packing a known malware and showing that our approach may considerably help malware analysis.

Program	#Inst.	#Inst.CoDisasm	Time (ms)
adpcm.exe	1191	1191	120
compress.exe	506	506	34
ns.exe	99	99	6
nsichneu.exe	5550	5550	1700
statemate.exe	1375	1375	155

#Inst. = number of instructions
#Inst.CoDisasm= number of instructions disassembled

Table 1: Precision of disassembly

5.2 Experimental validation of correctness

We consider regular binaries coming from a compiler. We show the correctness of CoDisasm on regular programs in Table 1. These samples are taken from the Mälardalen WCET benchmark programs [14]. This correctness is simply established by comparing assembly outputs.

5.3 Relevance of our approach on malware

We demonstrate that our approach is relevant by taking 500 malicious software from the public repository `malware.lu`. All these malware are detected by at least three well-known anti-virus software. We verify our assumptions that (i) malware are self-modifying code by computing the number of waves, and that (ii) malware use overlapping instructions by computing the number of layers per sample.

Table 2 shows the number of waves generated by the samples. It can be seen that 93% are self-modifying code. Half of them have only 2 waves. In this case, most of them could be disassembled by using first a generic unpacker and then by running a disassembler on the unpacked code. However, the remaining 40% of samples are more difficult to analyze. Generic unpackers fail, while our approach works, thus confirming its usefulness with respect to discovery and analysis of waves.

Table 3 shows the number of layers obtained on the same samples. As can be seen, 70% of the samples use at least one instruction overlapping technique.

5.4 Relevance of our approach on packers

In this section, the goal is to show that we are able to retrieve the original code of a packed binary. For this, we take hostname.exe, which plays the role of a probe that we can easily detect. Notice that the same experiment with

# Waves	1	2	3	4	5–10	> 10
	8%	53%	12%	6%	13%	9%

Table 2: Number of waves from 500 malware

# Layers	1	2	3	4	≥ 5
	32%	35%	17%	11%	5%

Table 3: Number of layers from 500 malware

an unknown binary like a malware will not be conclusive because we do not know *a priori* its assembly code (probably generated at runtime). Therefore, we packed hostname.exe with 28 different packers. The results are shown in Table 4.

We display the number of processes, threads and waves of the 28 packers. The last column indicates whether instructions are run in dynamically allocated memory or not. What we immediately see is that packers massively use waves, some of them being dynamically allocated. The cascade of waves may be as deep as 635. We were dumbfounded to see that up to 20% of some of these waves were composed of overlapped instructions. The case of `armadillo` is astounding. We observed 132 overlapping instructions and the packer creates 11 threads and has 2 processes. (The father process creates a new process which contains the original code. Then the father process attaches to the son process like a debugger.)

For all packers but Setisoft, we observed that the packed code behaves like hostname.exe. In fact, Setisoft detects the presence of *Pin_tracer* and does not run hostname.exe. Thus, in all but one case, we can state that we escape anti-debugging techniques and that we correctly reach the "payload". In all but three cases (PE Lock, PE Spin and VM Protect), we have been able to manually find the original code of hostname.exe within the waves disassembly, sometimes sliced into small pieces. The packer VM Protect is a code virtualizer, and thus it is expected that we cannot see that original code, only its intermediate representation. PE Spin and PE Lock are based on code transformations, and again it is not surprising that the original code cannot be recovered. This sequence of test shows that *Pin_tracer* is able to correctly instrument many significant packers.

For completeness, we also repeated the experiment replacing hostname.exe with other software. No significant differences in results were observed.

Finally, we determine for each packer the number of instructions by wave and also the number of layers. To conduct these experiments, we packed again hostname.exe ,which has 335 instructions. Due to a lack of space, we just present the results for Aspack in Table 5.4 and TELock in Table 6.

5.5 A malware writer scenario

In this last experiment, we sent the backdoor `hupigon.eyf` to the Virus Total Web service. From a total of 57 antivirus products, 45 detected `hupigon.eyf` and correctly identified it. We then packed `hupigon.eyf` with the Mystic packer and sent it back to Virus Total. This time, only 22 antivirus products detected that the file was malicious, but none were able to identify it.

We analyzed the same Mystic-packed file with CoDisasm. The Mystic packer generates 4 waves. The last wave creates a new process, which in turn creates two new processes.

Packer name	#proc.	#thr.	#Wave	DM
ACProtect v2.0	1	1	635	N
Armadillo v9.64	2	11	165	Y
Aspack v2.12	1	1	3	N
BoxedApp v3.2	1	15	6	Y
EP Protector v0.3	1	1	2	N
Expressor	1	1	2	N
FSG v2.0	1	1	2	N
JD Pack v2.0	1	1	3	N
MoleBox	1	1	3	N
Mystic	1	1	4	Y
Neolite v2.0	1	1	2	N
nPack v1.1.300	1	1	2	N
Packman v1.0	1	1	2	N
PE Compact v2.20	1	1	4	Y
PECrypt V1.02	1	4	99	Y
PE Lock	1	1	15	Y
PE Spin v1.1	1	1	80	Y
Petite v2.2	1	1	3	N
RLPack	1	1	2	N
Setisoft v2.7.1	1	5	32	Y
TELock v0.99	1	1	18	Y
Themida v2.0.3.0	1	28	106	Y
Upack v0.39	1	1	3	N
Upx v2.90	1	1	2	N
VM Protect v1.50	1	1	1	N
WinUPack	1	1	3	N
Yoda's Crypter v1.3	1	1	4	Y
Yoda's Protector v1.02	1	1	6	N

#proc. = number of processes ; #thr. = number of threads
#Wave = number of waves ; DM = Code run in allocated memory

Table 4: Packer analysis

Wave	Time (ms)	#Instructions	#Layers
1	62	1189	3
2	47	1115	3
3	20	357	1

Table 5: Aspack v2.12

Wave	Time (ms)	#Instructions	#Layers
1	1	85	4
2	1	67	1
3	1	20	2
4	1	43	2
5	13	693	4
6	1	18	1
7	1	28	1
8	1	16	1
9	1	51	1
10	1	36	1
11	1	23	1
12	1	49	1
13	2	134	3
14	9	496	3
15	5	333	3
16	17	799	2
17	3	172	1
18	8	431	1

Table 6: TELock v0.99

We traced the 4 waves and the 3 processes of the last wave. We verified from the disassembly output of CoDisasm that Wave 4 contains the payload `hupigon.eyf`. Then, we sent to Virus Total the PE file reconstructed from the last wave. This time, 20 antivirus products correctly detected it as `hupigon.eyf`. We think that this relatively low rate of detection is due to the fact that the antivirus products available on Virus Total are not made to scan the sent PE file, which is just a memory dump at the last wave.

This experiment illustrates a typical scenario where a malware writer builds a new malware by concealing a malicious code with a packer. The key point here is that the dynamic analysis of CoDisasm correctly reconstructs the disassembly code generated by the packer, and so successfully found `hupigon.eyf`. This has lead us to think of the potential usefulness of a Virus Total extension where each suspect binary is first disassembled by CoDisasm, which then produces a set of waves that can then be parsed by each anti-virus. Exploring this idea will likely be the object of future research.

6. DISCUSSION

In this work, we have just considered a single execution trace. We may wonder whether or not a single trace is enough. From our experience, the sequence of waves generated by packers rarely depends on inputs to the program and is almost blind to its execution environment. Our assumption is comforted by some of our previous experimental studies described [4, 7]. To conduct this work, we used 600 malware divided in six well-known families. We showed that less than 2% of malware interact with the system environment in the middle waves. We found that, (i) most of the time payloads are in the low last waves, and (ii) the wave structure is relatively simple. That is why, we were able to extract payloads in our experiments. But, and as we have already observed it, we should quickly develop the ability to automatically generate a set of traces to cope with code slices of the payloads triggered only when they are used. It is even possible to think of an attack, where the malware writer generates a massive number of waves in order to block and frustrate analysis with CoDisasm.

The situation is quite different when we deal with binary code in general. Take for example a botnet. A botnet will try to connect in order to receive commands, but it may fail if it is run in an isolated testbed [5]. As a result the trace obtained will not be relevant. A solution is to extract message formats and then to forge messages to generate traces in order to cover the botnet code [3, 6]. In other cases, an interesting direction would be to determine values to run unexplored paths in a wave to see whether or not it will produce new waves. Take this toy example:

```
if (date() = "Friday␣the␣13th") {
 unpack(); executePayload();
 }
else print "Hello␣world";
```

If this code is found in a wave and if it is analyzed in any other day than Friday the 13th, it will not produce the unpacked code.

7. CONCLUSION

We have developed a disassembler, called CoDisasm, that targets obfuscated malware x86 binary files running on Windows. It comes with an IDA plug-in, called BinViz, to visualize code unpacking waves, which was not described in this paper, but is available at `www.lhs.loria.fr`. CoDisasm disassembles binaries that are both self-modifying and that employ overlapping instructions as obfuscation techniques, something that is increasingly common in modern malware.

To accomplish this, the disassembler combines dynamic analysis of the binary and a static recursive disassembly procedure. We have devised and implemented an array of novel techniques, that we have dubbed *concatic disassembly*, to address challenges like the discovery of code waves and code layers. From a technical point a view, the dynamic analysis of binaries relies on a robust tracer taking into account anti-analysis mechanisms and tracing threads and processes. From a theoretical and fundamental point of view, we provide an effective model of self-modifying programs with overlapping instructions. CoDisasm is probably one of the first tools to achieve these results.

CoDisasm was mainly designed as an automatic disassembler tool which outputs sequence of disassembled code waves. In turn, each wave may be analyzed by other tools. We illustrate this approach with the example of the Hupigon malware in Section 5.5. Nonetheless, while it is able to automatically and seamlessly process a moderate amount of binary code (i.e. 30 binaries per minute in our lab), it would be necessary to speed-up disassembly in order to face the large amount of binaries received and processed each day by anti-virus companies.

Acknowledgments

The authors would like to thank Juan Caballero, Saumya Debray and Tim Kornau with whom we discussed this work, and who provided invaluable feedback. Work partially funded by French ANR (project BINSEC, grant ANR-12-INSE-0002).

8. REFERENCES

[1] S. Bardin, P. Herrmann, and F. Védrine. Refinement-based cfg reconstruction from unstructured programs. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 54–69, 2011.

[2] J. Caballero, N. M. Johnson, S. Mccamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proc. ISOC Network and Distributed Systems Security Symp. (NDSS)*, 2010.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proc. ACM Comp. Communications Security Conf. (CCS)*, pages 523–529, 2007.

[4] J. Calvet. *Analyse dynamique de logiciels malveillants.* PhD thesis, École Polytechnique de Montréal and Université de Lorraine, 2013.

[5] J. Calvet, C. R. Davis, J. M. Fernandez, W. Guizani, M. Kaczmarek, J.-Y. Marion, and P.-L. St-Onge. Isolated virtualised clusters: Testbeds for high-risk security experimentation and training. In *Proc. Usenix Cyber Security Experimentation and Testing (CSET)*, 2010.

[6] J. Calvet, C. R. Davis, J. M. Fernandez, J.-Y. Marion, P.-L. St-Onge, W. Guizani, P.-M. Bureau, and A. Somayaji. The case for in-the-lab botnet

experimentation: creating and taking down a 3000-node botnet. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 141–150. ACM, 2010.

[7] J. Calvet, F. Lalonde Lévesque, J. M. Fernandez, J.-Y. Marion, E. Traourouder, and F. Menet. WaveAtlas: surfing through the landscape of current malware packers. In *Proc. Virus Bulletin Conf.*, 2015.

[8] C. Collberg and J. Nagra. *Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Software Security Series, 2009.

[9] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *Proc. IEEE Working Conf. on Reverse Engineering (WCRE)*, pages 167–176, 2009.

[10] S. Debray and J. Patel. Reverse engineering self-modifying code: Unpacker extraction. In *Proc. IEEE Working Conf. on Reverse Engineering (WCRE)*, pages 131–140, 2010.

[11] I. Guilfanov. The ida pro disassembler and debugger. `http://www.hex-rays.com/idapro/`.

[12] W. Guizani, J.-Y. Marion, and D. Reynaud-Plantey. Server-side dynamic code analysis. In *Proc. Int. Conf. Malicious and Unwanted Software (MALWARE)*, pages 55–62, 2009.

[13] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *Proc. Int. Symp. Recent Advances in Intrusion Detection (RAID)*, pages 98–115, 2008.

[14] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *Proc. Int. Work. on Worst-Case Execution Time Analysis (WCET)*, pages 137–147, 2010.

[15] N. M. Hai, O. Mizuhito, and Q. T. Tho. Pushdown model generation of malware. Technical report, Japan Advanced Institute of Science and Technology, Japan, 2014.

[16] C. Jämthagen, P. Lantz, and M. Hell. A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries. In *Proc. Workshop on Anti-malware Testing Research (WATeR)*, 2013.

[17] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proc. ACM Workshop on Recurring Malcode (WoRM)*, pages 46–53, 2007.

[18] J. Kinder. *Static analysis of x86 executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[19] J. Kinder and D. Kravchenko. Alternating control flow reconstruction. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 267–282, 2012.

[20] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 214–228, 2009.

[21] N. Krishnamoorthy, S. Debray, and K. Fligg. Static detection of disassembly errors. In *Proc. IEEE Working Conf. on Reverse Engineering (WCRE)*, pages 259–268, 2009.

[22] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. USENIX Security Symposium*, pages 255–270, Berkeley, CA, USA, 2004.

[23] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. ACM Conf. Comp. Communications Security (CCS)*, pages 290–299, 2003.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, K. Hazelwood, and V. J. Reddi. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2005.

[25] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2007.

[26] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2007.

[27] S. Nanda, W. Li, L.-C. Lam, and T. cker Chiueh. Bird: binary interpretation using runtime disassembly. In *Proc. Int. Symp. Code Generation and Optimization (CGO)*, 2006.

[28] M. D. Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. Townsend. Modelling metamorphism by abstract interpretation. In *Proc. Int. Static Analysis Symposium (SAS)*, pages 218–235, 2010.

[29] T. W. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2008.

[30] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, pages 289–300, 2006.

[31] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proc. IEEE Working Conference on Reverse Engineering (WCRE)*, pages 45–, 2002.

[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. In *Proc. IEEE Symp. Security and Privacy (S&P)*, 2015.

[33] G. Vigna. Static disassembly and code analysis. In M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 19–41. Springer US, 2007.

[34] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *Proc. IEEE Symp. Security and Privacy (S&P)*, 2015.