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

Jean-Yves Marion
Professor at l’Université de Lorraine
Institut Universitaire de France
This is a joint works with

Guillaume Bonfante
José Fernandez
Benjamin Rouxel
Aurélien Thierry
Fabrice Sabatier
Jean-Yves Marion
Problem
What really makes this program?

Input:
an x86 obfuscated binary code

What Happen After You Hit Return ?
Towards a high level semantics

Disassembly

Memory or PE file

Jean-Yves Marion
Towards a high level semantics

Jean-Yves Marion
Towards a high level semantics

```assembly
long int suite(long int x)
{
  long int u=0;
  long int i=0;
  for(i=0;i<x;i++)
  {
    if ((i % 2)==0) u=2*u+2;
    else u=2*u+1;
  }
  return u;
}
```

Decompile
Abstraction by reverse-refinement

High-level semantics

Disassembly is a crucial step

Low-level semantics

Jean-Yves Marion
Context

Practical issues related to binary analysis

Code Protections against analysis
Malware protections

1. Obfuscation
2. Cryptography
3. Self-modification
4. Code overlapping
5. Anti-analysis tricks

Malware analysis is very hard

Rational approach to help Felix the cat!
A common protection scheme for malware

This is a run of the **packer** Telock with 18 waves

A run is a sequence of waves

---

### Self-modifying program schema

**Wave 1**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>01005000</td>
<td>pushfd</td>
</tr>
<tr>
<td>01005001</td>
<td>push 0x3</td>
</tr>
<tr>
<td>01005003</td>
<td>jae 0x1005009</td>
</tr>
<tr>
<td>01005005</td>
<td>jmp 0x1005009</td>
</tr>
<tr>
<td>01005007</td>
<td>db 0x75 ; 'u'</td>
</tr>
<tr>
<td>01005008</td>
<td>db 0x75 ; 'u'</td>
</tr>
<tr>
<td>01005009</td>
<td>call 0x1005014</td>
</tr>
<tr>
<td>0100500e</td>
<td>xor ax, 0x773</td>
</tr>
<tr>
<td>01005012</td>
<td>jmp 0x1005031</td>
</tr>
<tr>
<td>01005014</td>
<td>add esp, 0x4</td>
</tr>
<tr>
<td>01005017</td>
<td>jmp 0x100501b</td>
</tr>
<tr>
<td>01005019</td>
<td>db 0x75 ; 'u'</td>
</tr>
<tr>
<td>0100501a</td>
<td>db 0x75 ; 'u'</td>
</tr>
<tr>
<td>0100501b</td>
<td>dec dword</td>
</tr>
</tbody>
</table>

**Wave 2**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>01007088</td>
<td>call 0x100708d</td>
</tr>
<tr>
<td>0100708d</td>
<td>sub dword [ss:esp], 0x23a</td>
</tr>
<tr>
<td>01007094</td>
<td>jmp dword [ss:esp+0x4]</td>
</tr>
</tbody>
</table>

**Wave 18**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1006ba7</td>
<td>mov ebx, dword ptr [ebp+0x403783]</td>
</tr>
<tr>
<td>1006bad</td>
<td>xor esi, esi</td>
</tr>
<tr>
<td>1006baf</td>
<td>not ebx</td>
</tr>
<tr>
<td>1006bb1</td>
<td>or esi, ebx</td>
</tr>
<tr>
<td>1006bb3</td>
<td>jnz 0xa</td>
</tr>
<tr>
<td>1006bbd</td>
<td>add ebx, dword ptr [ebp+0x403763]</td>
</tr>
</tbody>
</table>

---

Jean-Yves Marion
Almost all malware is self-modifying!

Based on 100 samples of Koobface, Conficker, Cutwail, Ramnit, Storm Worm, Waledac

See Virus Bulletin 2015
Problem
How to obtain a correct Control Flow Graph of a self-modifying code?

Disassembly of self-modifying codes
Where are we?

**The input:**
An obfuscated x86 binary file with no information except an entry point (e.g. a malware)

Can we recover the assembly code of the original program inside a binary file?
- **when the binary file is self-modifying**

**Goal**
The disassembly of each wave of code of a self-modifying code (e.g. packers).
Disassembly is a key task in software debugging and malware analysis. It involves the recovery of assembly instructions from binary machine code. It can be problematic in the case of malicious code, as malware writers often employ techniques to thwart correct disassembly by standard tools.

Nonetheless, disassembly is a crucial step in malware reverse engineering. Correct disassembly of binaries is necessary to determine the range of values in the register(register).

In this paper, we focus on the disassembly of x86 self-modifying binaries with overlapping instructions. Current state-of-the-art methods to thwart correct disassembly by standard tools.

Our approach substantially improves the success of disassembly when confronted with overlapping instructions. Our approach substantially improves the success of disassembly when confronted with overlapping instructions. Our approach substantially improves the success of disassembly when confronted with overlapping instructions.

We introduce a novel disassembly method, called CoDisasm: Concatatic disassembly of self-modifying binaries with overlapping instructions, that combines CONCATIC techniques to thwart correct disassembly by standard tools.

In order to perform further reverse engineering, we need to reconstruct the control flow graph (CFG) in order to perform further reverse engineering. To our knowledge, no other disassembler thwarts both of these obfuscations methods together.

Experimental results on about five hundred malware samples show that our approach correctly recovers large parts of the input. In this paper, we focus on the disassembly of x86 self-modifying binaries with overlapping instructions. Current state-of-the-art methods to thwart correct disassembly by standard tools.

Our approach substantially improves the success of disassembly when confronted with overlapping instructions. Our approach substantially improves the success of disassembly when confronted with overlapping instructions.

In this paper, we focus on the disassembly of x86 self-modifying binaries with overlapping instructions. Current state-of-the-art methods to thwart correct disassembly by standard tools.

We introduce a novel disassembly method, called CoDisasm: Concatatic disassembly of self-modifying binaries with overlapping instructions, that combines CONCATIC techniques to thwart correct disassembly by standard tools.

In order to perform further reverse engineering, we need to reconstruct the control flow graph (CFG) in order to perform further reverse engineering. To our knowledge, no other disassembler thwarts both of these obfuscations methods together.

Experimental results on about five hundred malware samples show that our approach correctly recovers large parts of the input.

We introduce a novel disassembly method, called CoDisasm: Concatatic disassembly of self-modifying binaries with overlapping instructions, that combines CONCATIC techniques to thwart correct disassembly by standard tools.

In order to perform further reverse engineering, we need to reconstruct the control flow graph (CFG) in order to perform further reverse engineering. To our knowledge, no other disassembler thwarts both of these obfuscations methods together.

Experimental results on about five hundred malware samples show that our approach correctly recovers large parts of the input. We introduce a novel disassembly method, called CoDisasm: Concatatic disassembly of self-modifying binaries with overlapping instructions, that combines CONCATIC techniques to thwart correct disassembly by standard tools.

In order to perform further reverse engineering, we need to reconstruct the control flow graph (CFG) in order to perform further reverse engineering. To our knowledge, no other disassembler thwarts both of these obfuscations methods together.

Experimental results on about five hundred malware samples show that our approach correctly recovers large parts of the input.

We introduce a novel disassembly method, called CoDisasm: Concatatic disassembly of self-modifying binaries with overlapping instructions, that combines CONCATIC techniques to thwart correct disassembly by standard tools.

In order to perform further reverse engineering, we need to reconstruct the control flow graph (CFG) in order to perform further reverse engineering. To our knowledge, no other disassembler thwarts both of these obfuscations methods together.

Experimental results on about five hundred malware samples show that our approach correctly recovers large parts of the input. We introduce a novel disassembly method, called CoDisasm: Concatatic disassembly of self-modifying binaries with overlapping instructions, that combines CONCATIC techniques to thwart correct disassembly by standard tools.

In order to perform further reverse engineering, we need to reconstruct the control flow graph (CFG) in order to perform further reverse engineering. To our knowledge, no other disassembler thwarts both of these obfuscations methods together.

Experimental results on about five hundred malware samples show that our approach correctly recovers large parts of the input.
Wave extraction

Secure sand boxing

Dynamic analysis

Next Goal: Disassemble each PE-wave separately determined by the trace
TeLock with 18 waves

Secure sand boxing

First instruction

An execution trace

Last instruction

A trace indicates a sequence of code waves

Wave 1    Wave 2    .........    Wave 17    Wave 18

PE-Wave 1  PE-Wave 2  .........  PE-Wave 16  PE-Wave 18

Dynamic analysis
Armadillo (V9) : 2 processes 11 threads

Secure sand boxing

An execution trace

A trace indicates a sequence of code waves

Wave 1  Wave 2  ..........  Wave 163  Wave 164

Dynamic analysis

PE-Wave 162 Contains the packed file
Wave disassembly

Disassemble each PE encoding a wave

Instruction overlapping

Separate code from data ??

A path in the

Solved partially indirect jumps
Instruction overlapping in UPX packer

010059f0  89 f9  mov ecx,edi
010059f2  79 07  jnz +9
010059f4  0f b7 07  movzx eax, word [edi]
010059f7  47  inc edi
010059f8  50  push eax
010059fa  b9 57 48 f2 ae  mov ecx, aef24857
010059ff  55  push ebp
Instruction overlapping in UPX packer

```
010059f0  89 f9  mov ecx,edi
010059f2  79 07  jnz +9
010059f4  0f b7 07  movzx eax, word [edi]
010059f7  47  inc edi
010059f8  50  push eax
010059fa  b9 57 48 f2 ae  mov ecx, aef24857
          57  push edi
010059fb  48  dec eax
          f2 ae  repne scasb
010059fd  55  push ebp
          55  push ebp
```
The overall architecture of the disassembler

We combine concrete path execution and static analysis

Workflow:

- x86 file
- Server
- compact traces + Memory dumps
- Disassemble

(available at http://marionjy.loria.fr/)
We are what we read!

- J. Caballero, N. M. Johnson, S. Mccamant, and D. Song: Binary code extraction and interface identification for security applications
- J. Kinder, F. Zuleger, and H. Veith: An abstract interpretation-based framework for control flow reconstruction from binaries
- T. W. Reps and G. Balakrishnan: Improved memory-access analysis for x86 executables
- S. Debray and J. Patel: Reverse engineering self-modifying code: Unpacker extraction
- N. Krishnamoorthy, S. Debray, and K. Fligg: Static detection of disassembly errors
- B. Schwarz, S. Debray, and G. Andrews: Disassembly of executable code revisited
- C. Kruegel, W. Robertson, F. Valeur, and G. Vigna: Static disassembly of obfuscated binaries
- J. Kinder. Static analysis of x86 executables
- G. Vigna. Static disassembly and code analysis
Pitfalls:

- Call/ret obfuscations
- Indirect jumps
- Opaque Predicates
Another key issue: Big code!

Answers to these questions should be automatic and should deal with vast amounts of samples

- About 6 million of samples inside our lab
- About 300,000 files by day for Google via VirusTotal!
- 400 millions of samples for Google

80 hours (=3 days) to analyze 300,000 files with a time put of 1 sec. by file
Summary: Codisasm

- A model of self-modifying computations based on Waves
- Deals with self-modifying codes
- Deals with overlapping instructions
- Follows processes and implements a sequence of anti-anti-debugging tricks

*The last wave of Mystic packer generates 3 processes*
Packer disassembly

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

1. We sent the backdoor hupigon.eyf to VirusTotal:
   - 45 AV detected hupigon.eyf over 57 AV

   SHA256: fdd726007bdec7ef8a22e79f2f9e7115a86f1271f0be583858e4d09546860d33
   Nom du fichier: Hupigon.eyf
   Ratio de détection: 45 / 57
   Date d’analyse: 2015-05-16 14:05:50 UTC (il y a 4 mois, 3 semaines)

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Résultat</th>
<th>Mise à jour</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALYac</td>
<td>Backdoor.Agent.YRI</td>
<td>20150516</td>
</tr>
<tr>
<td>AVG</td>
<td>BackDoor.Hupigon5.RUD</td>
<td>20150516</td>
</tr>
<tr>
<td>AVware</td>
<td>Trojan.Win32.Generic!BT</td>
<td>20150516</td>
</tr>
</tbody>
</table>

Jean-Yves Marion
Epilogue: a malware writer scenario

1. We sent the backdoor `hupigon.eyf` to VirusTotal:
   - 45 AV detected `hupigon.eyf` over 57 AV

2. We packed `hupigon.eyf` with Mystic packer: `Mystic[hupigon.eyf]`

3. **Only 22 AV** identified `Mystic[hupigon.eyf]` has malicious **without identifying** `hupigon.eyf`

![VirusTotal Image](https://www.virustotal.com/fr/file/601231c1b38053337822369de812b3e7f47811baa45192cfc59636c3a3289ed0/analysis/)

<table>
<thead>
<tr>
<th>SHA256</th>
<th>601231c1b38053337822369de812b3e7f47811baa45192cfc59636c3a3289ed0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nom du fichier</td>
<td>codisasm-mystic-Hupigon.eyf</td>
</tr>
<tr>
<td>Ratio de détection</td>
<td>22 / 57</td>
</tr>
<tr>
<td>Date d'analyse</td>
<td>2015-05-16 16:25:43 UTC (il y a 4 mois, 3 semaines)</td>
</tr>
</tbody>
</table>

**Antivirus**

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Résultat</th>
<th>Mise à jour</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALYac</td>
<td>Gen:Heur.Zygug.1</td>
<td>20150516</td>
</tr>
<tr>
<td>AVG</td>
<td>Win32/Cryptor</td>
<td>20150516</td>
</tr>
</tbody>
</table>
Epilogue: a malware writer scenario

1. We sent the backdoor hupigon.eyf to VirusTotal:
   - 45 AV detected hupigon.eyf over 57 AV

2. We packed hupigon.eyf with Mystic packer: Mystic[hupigon.eyf]

3. Only 22 AV identified Mystic[hupigon.eyf] as malicious without identifying hupigon.eyf

4. We uses Codisasm on Mystic[hupigon.eyf]. We obtained 4 waves

5. We checked that one of the processes in the last wave contains hupigon.eyf

6. We sent the PE file of the last wave to VirusTotal: 20 AV identified correctly hupigon.eyf
### SHA256:
42516a3a7cc6b33804c5a62c5e36e8e7ecb28a92c5416d5c772e601d7ad66003

### Nom du fichier:
mystic2-Hupigon.eyf_wave3-WithAllocSec

### Ratio de détection:
20 / 57

### Date d'analyse:
2015-05-16 16:38:34 UTC (il y a 4 mois, 3 semaines)

<table>
<thead>
<tr>
<th>Antivirus</th>
<th>Résultat</th>
<th>Mise à jour</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALYac</td>
<td>Backdoor.Agent.YRI</td>
<td>20150516</td>
</tr>
<tr>
<td>AVG</td>
<td>BackDoor.Hupigon5.RUD</td>
<td>20150516</td>
</tr>
<tr>
<td>Ad-Aware</td>
<td>Backdoor.Agent.YRI</td>
<td>20150516</td>
</tr>
<tr>
<td>AhnLab-V3</td>
<td>Win-Trojan/Hupigon.34304.AA</td>
<td>20150516</td>
</tr>
<tr>
<td>Antiy-AVL</td>
<td>Trojan[Backdoor]/Win32.Hupigon</td>
<td>20150516</td>
</tr>
<tr>
<td>Avast</td>
<td>Win32:Neptunia-WC [Trj]</td>
<td>20150516</td>
</tr>
</tbody>
</table>
Questions?

Code available at http://marionjy.loria.fr/

Thank you

Jean-Yves Marion