TARGETING INFEASIBILITY QUESTIONS on OBUFSCATED CODES

Sébastien Bardin (CEA LIST)
Robin David (CEA LIST) & Jean-Yves Marion (LORIA)
IN A NUTSHELL

• Challenge: malware deobfuscation

• Infeasibility questions are a blind spot of current automated techniques

• We propose an efficient, robust and precise method for them

• Very promising case-studies
CONTEXTE: MALWARE COMPREHENSION

APT: highly sophisticated attacks
• **Targeted malware**
• **Written by experts**
• **Attack**: 0-days
• **Defense**: stealth, **obfuscation**
• **Sponsored by states or mafia**

The day after: **malware comprehension**
• understand what has been going on
• mitigate, fix and clean
• improve defense

USA elections: DNC Hack

Goal: help malware comprehension
• Reverse of heavily obfuscated code
• Identify and simplify protections
CHALLENGE: CORRECT DISASSEMBLY

Basic reverse problem
• aka model recovery
• aka CFG recovery
CAN BE TRICKY!

- code – data
- dynamic jumps (jmp eax)

### Sections

**.text**

8D 4C 24 04 83 E4 F0 FF 71 FC 55 89 E5 53 51 83
EC 10 89 CB 83 EC 0C 6A 0A E8 A7 FF FF FF 83 C4
10 89 45 F0 8B 43 04 83 C0 04 8B 00 83 EC 0C 50
E8 C0 FE FF FF 83 C4 10 89 45 F4 83 7D F4 04 77
3B 88 45 F4 C1 E0 02 05 98 85 04 08 8B 00 FF E0
C7 45 F4 00 00 00 00 00 EB 23 C7 45 F4 01 00 00 00
EB 1A C7 45 F4 02 00 00 00 EB 11 C7 45 F4 03 00
00 00 EB 08 C7 45 F4 04 00 00 00 00 EB 83 EC 08 FF
75 F4 68 90 85 04 08 EB 29 FE FF FF 83 C4 10 8B
45 F4 8D 65 F8 59 5B 5D 8D 61 FC C3 66 90 66 90
66 90 66 90 90 55 57 31 FF FF FF FF FF FF FF FF FF
81 C3 89 12 00 08 83 EC 1C 8D 6C 24 30 8D B3 0C
C0 FF FF FF 9B 0E B1 FD FF FF 8D 83 08 FF FF FF 29 C6
C1 0E 02 85 F6 74 27 8D B6 06 00 00 00 EB 44 24
38 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
FF FF FF FF FF FF FF FF EE C7 01 39 FF 75 DF 83 C4
C0 5B 5E 5F 5D C3 EB 0D 90 90 90 90 90 90 90 90

**.fini**

.code – data

**.rodata**

.code – data

### Code (Functions)

- **main**
- **unknown**
- **__libc_csu_init**
- **unknown**
- **__libc_csu_fini**
- **_term_proc**
- **_fp_hw, _IO_stdin_used**
- switch jump table

### Assembly

- push ebx
- sub esp, 8
- call get_pc[
- add ebx, 0x1217
- add esp, 8
- pop ebx
- ret
Obfuscation: make a code hard to reverse

- self-modification
- encryption
- virtualization
- code overlapping
- opaque predicates
- callstack tampering
- ...

eg: \(7y^2 - 1 \neq x^2\)

(for any value of \(x, y\) in modular arithmetic)

\[
\text{mov eax, ds:X} \\
\text{mov ecx, ds:Y} \\
\text{imul ecx, ecx} \\
\text{imul ecx, 7} \\
\text{sub ecx, 1} \\
\text{imul eax, eax} \\
\text{cmp ecx, eax} \\
\text{jz <dead_addr>} \\
\text{mov eax, ds:X} \\
\text{mov ecx, ds:Y} \\
\text{imul ecx, ecx} \\
\text{imul ecx, 7} \\
\text{sub ecx, 1} \\
\text{imul eax, eax} \\
\text{cmp ecx, eax} \\
\text{jz <dead_addr>} \\
\text{call +5} \\
\text{pop edx} \\
\text{add edx, 8} \\
\text{push edx} \\
\text{ret} \\
\text{byte{invalid}} \\
\text{[...]} \\
\\]

\(0x40104d\ \text{mov al, input_buffer[ecx]} \) \\
\(0x401053\ \text{xor al, 0x7d} \) \\
\(0x401055\ \text{cmp al, data_size[ecx]} \) \\
\(0x40105b\ \text{jne 0x40107b} \) \\
\(zf = 0 \) \\
\(0x40107b\ \text{... You are failure} \)

\(zf = 1 \) \\
\(0x40105d\ \text{inc ecx} \) \\
\(0x40105e\ \text{cmp ecx, 0x18} \) \\
\(0x401061\ \text{j1 0x40104d} \) \\
\(0x401063\ \text{... You are success} \)

\(ecx < 0x18\)
Constant-value predicates
(always true, always false)

- dead branch points to spurious code
- goal = waste reverser time & efforts

eg: \(7y^2 - 1 \neq x^2\)
(for any value of \(x, y\) in modular arithmetic)
EXAMPLE: STACK TAMPERING

Alter the standard compilation scheme:
ret do not go back to call

- hide the real target
- return site may be spurious code
STANDARD DISASSEMBLY TECHNIQUES ARE NOT ENOUGH

Static analysis
- too fragile vs obfuscation
- junk instr, missed instr.

Dynamic analysis
- robust vs obfuscation
- too incomplete
DYNAMIC SYMBOLIC EXECUTION CAN HELP

```c
int main () {  
    int x = input();  
    int y = input();  
    int z = 2 * y;  
    if (z == x) {  
        if (x > y + 10)  
            failure;  
    }  
    success;  
}
```

- Under-approximation
  - correct
  - relatively complete

- No false alarm

For deobfuscation
- find new real paths
- robust
- still incomplete

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths
Prove that something is always true (resp. false)

Many such issues in reverse
• is a branch dead?
• does the ret always return to the call?
• have I found all targets of a dynamic jump?

And more
• does this malicious ret always go there?
• does this expression always evaluate to 15?
• does this self-modification always write this opcode?
• does this self-modification always rewrite this instr.?
• ...

Not addressed by DSE
• Cannot enumerate all paths
OUR CHALLENGE

Check infeasibility questions in obfuscated codes
• scale to realistic malware sizes
• robust to obfuscation such as self-modification
• precise
• generic

Rest of the talk:
• opaque predicate
• stack tampering
OUR PROPOSAL: BACKWARD-BOUNDED SYMBOLIC EXECUTION

Insight 1: symbolic reasoning
- **precision**
- But: need finite #paths

Insight 2: backward-bounded
- \( \text{pre}_k(c)=0 \Rightarrow c \text{ is infeasible} \)
- finite #paths
- **efficient**, depends on \( k \)
- But: backward on jump eax?

Insight 3: dynamic partial CFG
- solve (partially) dyn. jumps
- **robustness**

Low FP/FN rates in practice
- ground truth xp

False negative (FN)
- can miss infeasibility
- why: \( k \) too small (miss \( \wedge \)-constraints)

False positive (FP)
- wrongly assert infeasibility
- why: CFG too partial (miss \( \vee \)-constraints)

Sébastien Bardin et al. -- S&P 2017 | 13
### EXPERIMENTAL EVALUATION

<table>
<thead>
<tr>
<th>Experiment Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controlled experiments (ground truth)</td>
<td>precision</td>
</tr>
<tr>
<td>Large-scale experiment: packers</td>
<td>scalability, robustness</td>
</tr>
<tr>
<td>Case-study: X-tunnel malware</td>
<td>usefulness</td>
</tr>
</tbody>
</table>
CONTROLLED EXPERIMENTS

- **Goal** = assess the precision of the technique
  - ground truth value

- **Experiment 1**: opaque predicates (o-llvm)
  - 100 core utils, 5x20 obfuscated codes
  - k=16: 3.46% error, no false negative
  - robust to k
  - efficient: 0.02s / query

- **Experiment 2**: stack tampering (tigress)
  - 5 obfuscated codes, 5 core utils
  - almost all genuine ret are proved (no false positive)
  - many malicious ret are proved « single-targets »

- **Very precise résultats**
- **Seems efficient**
CASE-STUDY: PACKERS

<table>
<thead>
<tr>
<th>packers</th>
<th>trace len.</th>
<th>#proc</th>
<th>#th</th>
<th>#SMC</th>
<th>opaque predicates</th>
<th>call stack tampering</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPProtect v2.0</td>
<td>1.8K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>159</td>
<td>0</td>
</tr>
<tr>
<td>ASPack v2.12</td>
<td>377K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>Crypter v1.12</td>
<td>1.1K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>125</td>
</tr>
<tr>
<td>Expressor</td>
<td>635K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>FSG v2.0</td>
<td>68K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Mew</td>
<td>59K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>PE Lock</td>
<td>2.3K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>95</td>
</tr>
<tr>
<td>RLPack</td>
<td>941K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>TELock v0.51</td>
<td>466K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Upack v0.39</td>
<td>711K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

- The technique scale on significant traces
- Many true positives. Some packers are using it intensively
- Packers using ret to perform the final tail transition to the entrypoint

Packers: legitimate software protection tools
(basic malware: the sole protection)
CASE-STUDY: THE XTUNNEL MALWARE (part of DNC hack)

Two heavily obfuscated samples
- Many opaque predicates

Goal: detect & remove protections
- Identify 50% of code as spurious
- Fully automatic, < 3h

<table>
<thead>
<tr>
<th>C637 Sample #1</th>
<th>99B4 Sample #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#total instruction</td>
<td>505,008</td>
</tr>
<tr>
<td>#alive</td>
<td>+279,483</td>
</tr>
</tbody>
</table>
SECURITY ANALYSIS: COUNTER-MEASURES (and mitigations)

• Long dependency chains (evading the bound k)
  • Not always requires the whole chain to conclude!
  • Can use a more flexible notion of bound (data-dependencies, formula size)

• Hard-to-solve predicates (causing timeouts)
  • A time-out is already a valuable information
  • Opportunity to find infeasible patterns (then matching), or signatures
  • Tradeoff between performance penalty vs protection focus
  • Note: must be input-dependent, otherwise removed by standard DSE optimizations

• Anti-dynamic tricks (fool initial dynamic recovery)
  • Can use the appropriate mitigations
  • Note: some tricks can be circumvent by symbolic reasoning

Current state-of-the-art
• push the cat-and-mouse game further
• raise the bar for malware designers
CONCLUSION & TAKE AWAY

• What has been done
  • Identify infeasibility questions as a blind spot of deobfuscation techniques
  • Propose an efficient, robust and precise method
  • Controlled experiments and large-scale studies

• Semantic analysis can change the game of deobfuscation
  • Complement existing approaches
  • Open the way to fruitful combinations [see the paper]

• Formal methods can be useful for malware, but must be adapted
  • Need robustness and scalability!
  • Accept to lose both correctness & completeness – in a controlled way

• BINSEC platform: looking for collaborations and users 😊
  • Open-source, still in its infancy