DEOBFUSCATION: SEMANTIC ANALYSIS TO THE RESCUE

Sébastien Bardin (CEA LIST)
Robin David (CEA LIST, QuarksLab)
Jean-Yves Marion (LORIA)
ABOUT MY LAB @CEA [Paris-Saclay, France]

CEA LIST, Software Safety & Security Lab

- rigorous tools for building high-level quality software
- second part of V-cycle
- automatic software analysis
- mostly source code
IN A NUTSHELL

• Challenge: malware **deobfuscation**

• Standard techniques (dynamic, syntactic) not enough

• **Semantic methods can help** [obfuscation preserves semantic]
  • Yet, need to be strongly adapted (robustness, precision, efficiency)

• **A tour on how symbolic methods can help**
  • *Explore and discover*
  • *Prove infeasibility*       [S&P 2017]
  • *Simplify* (not covered here)
OUTLINE

• Context
  • Malware comprehension
  • Semantic analysis

• The hard journey from source to binary
  • Explore & Discover
  • Prove infeasibility

• A few case-studies

• Conclusion
CONTEXT: 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
CANS 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 EA A7 FE FF FF 83 C4
10 89 45 F0 8B 43 04 83 C0 04 8B 00 83 EC 0C 50
E8 C0 FF FF 83 C4 10 89 45 F4 83 7D F4 04 77
3B 8B 45 F4 C1 E6 02 05 98 85 04 08 8B 00 FF E0
C7 45 F4 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
90 00 EB 08 C7 45 F4 04 00 00 00 90 83 EC 08 FF
75 F4 68 90 85 04 08 EB 29 FE FF FF 83 C4 10 88
45 F4 8D 65 F8 59 5B 5D 8D 61 EC C3 60 90 66 90
66 90 66 90 90 55 57 31 FF 56 53 E8 85 FE FF FF
81 C3 89 12 00 00 83 EC 1C 8B 6C 24 30 8D B3 0C
FF FF FF E8 B1 FD FF FF 8D 83 08 FF FF FF 29 C6
C1 FE 02 85 6F 74 27 8D B6 80 00 00 00 80 44 24
38 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
FF 94 BB 08 FF FF 83 C7 01 39 F7 75 DF 83 C4
1C 5B 5E FF FE 0D 0D 90 90 90 90 90 90 90 90
90 90 90 90 90 F3 C3 FF FF 53 83 EC 08 E8 13 FE
FF FF 81 C3 17 12 00 83 C4 08 5B C3 0D 0D 00
90 01 00 02 00 76 61 6C 3A 25 64 0A 00 AB 84 04
08 84 84 04 08 BB 84 04 08 C6 84 04 08 CF 84 04
08 01 1B 03 3B 28 00 00 00 00 00 00 00 54 FD FF
```

**Code (Functions)**

```
main

unknown

_libc_csu_init

unknown

_libc_csu_fini

_term_proc

_fp.hw._IO_stdin_used

"vaRndn"

switch jump table
```

**Assembly**

```
```

- code
- dead bytes
- global csts
- strings
- pointers
- other
REVERSE CAN BECOME A NIGHTMARE (OBFUSCATION)

Obfuscation: make a code hard to reverse
- self-modification
- encryption
- virtualization
- code overlapping
- opaque predicates
- callstack tampering
- ...

Goal: help malware comprehension
- Find real parts of the code
- Identify and simplify protections
- Ideal = revert protections

Obf: $7y^2 - 1 \neq x^2$
(for any value of x, y in modular arithmetic)
EXAMPLE: OPAQUE PREDICATE

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)

```assembly
  mov eax, ds:X
  mov ecx, ds:Y
  imul ecx, ecx
  imul ecx, 7
  sub  ecx, 1
  imul eax, eax
  cmp  ecx, eax
  jz <dead_addr>
```
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
STATE-OF-THE-ART TOOLS ARE NOT ENOUGH

- Static (syntactic): too fragile
- Dynamic: too incomplete

Just add

```
mov %eax,%ecx
mov %ecx,%eax
```

and break results

With IDA
THE SITUATION

- Malware deobfuscation is necessary
- Malware deobfuscation is highly challenging
- Standard tools are not enough – experts need better help!

- Static (syntactic): too fragile
- Dynamic: too incomplete
Semantic tools help make sense of binary

- Develop the next generation of binary-level tools!
- motto: leverage formal methods from safety critical systems

Semantic preserved by obfuscation

Can reason about sets of executions
- find rare events
- prove infeasibility

Advantages
- more robust than syntactic
- more thorough than dynamic

Challenges
- source-level $\leftrightarrow$ binary-level
- safety $\leftrightarrow$ security
- many (complex) architectures
<En aparté> ABOUT FORMAL METHODS

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way

Success in safety-critical

Key concepts: $M \models \varphi$
- $M$: semantic of the program
- $\varphi$: property to be checked
- $\models$: algorithmic check

Kind of properties
- absence of runtime error
- pre/post-conditions
- temporal properties
<En aparté> A DREAM COME TRUE … IN CERTAIN DOMAINS

Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [Airbus], railroad [Metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Ex: Airbus

Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C *]
- numerical precision [Fluctuat *]
- source-binary conformance [CompCert]
- resource usage [Absint]

* : by CEA DILS/LSL
NOW: BINARY-LEVEL ANALYSIS & OBFUSCATION

**Model**

```plaintext
x > 0 / x := x - 1

x := a + b

x = 0 /
```

**Source code**

```plaintext
int foo(int x, int y) {
    int k = x;
    int c = y;
    while (c > 0) do {
        k++;
        c--;
    }
    return k;
}
```

**Assembly**

```assembly
_start:
    load A 100
    add B A
    cmp B 0
    jle label

label:
    move @100 B
```

**Executable**

```
ABFFF780BD70696CA101001BDE45
145634789234A8FF678ABDCF436
5A284C6D009F5F5D1E0835715697
14FEDBCADBCABD459700346901
3456KAHA305G67H345BFF54E5D3
00113456735FFD451E13AB080DAD
344252FFAABDBA457345FD780001
FFF22546ADDAE989776600000000
```
THE HARD JOURNEY FROM SOURCE TO BINARY

Low-level semantics of data

- machine arithmetic, bit-level operations, untyped memory
  - difficult for any state-of-the-art formal technique

Low-level semantics of control

- no distinction data / instructions, dynamic jumps (jmp eax)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
  - violate an implicit prerequisite for most formal techniques

Diversity of architectures and instruction sets

- support for many instructions, modelling issues
  - tedious, time consuming and error prone

Wanted

- robustness
- precision
- scale
静态语义分析在二进制代码上非常非常难。

问题
- 跳转 eax
- 内存
- 位推理

框架：抽象解释
- 抽象域的概念
  - ⊥, ⊤, ∪, ∩, ⊆, eval#
- 更多或更精确的域
  - 区间，多面体等。
- 固点直到稳定化

问题
- 跳转 eax
- 内存
- 位推理
OUR APPROACH: BINSEC

**x86**

```
0x10000:
90 83 0c 05
```

**ARM**

```
0x10000:
E8 00 00 00 00
```

---

**Static analysis**

---

**Symbolic execution**

- \( \text{lhs} := \text{rhs} \)
- goto addr, goto expr
- \( \text{ite} \)(cond)? goto addr:
- assume, assert, nondet

---

**OUR APPROACH:**

BINSEC

---

**IR**

---

**malware analysis**

---

**vulnerabilities**

---

**dynamic disassembly**

---

**dynamic symbolic execution**

---

**static disassembly**

---

**partial safe CFG**

---

**execution trace**

---

**dynamic symbolic execution**

---

**obfuscation information**

---

**paths in computation**

---

**paths over approximate backward simulation**
KEY: DYNAMIC SYMBOLIC EXECUTION
(DSE, Godefroid 2005)

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

Perfect for intensive testing
• Correct
• No false alarm
• Robust
• Scale in some ways

// incomplete

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}

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

« dynamic analysis on steroids »

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths
IN PRACTICE

Can recover useful semantic information
• More precise disassembly
• Exact semantic of instructions
• Input of interest
• ...

cmp eax ebx
cmc
jae ...

CF := (eax<,ebx)
if (¬CF) goto ...

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
FORWARD & BACKWARD SYMBOLIC EXECUTION

\[ \sigma := \emptyset \]
\[ PC := \top \]
\[ x = \text{input}() \]
\[ y = \text{input}() \]
\[ z = 2 \times y \]
\[ \sigma := \{ x \rightarrow x_0, y \rightarrow y_0, z \rightarrow 2y_0 \} \]
\[ z := x \]
\[ PC := \top \land 2y_0 = x_0 \]
\[ x > y + 10 \]
\[ PC := \top \land 2y_0 \neq x_0 \]
\[ PC := \top \land 2y_0 = x_0 \land x_0 \leq y_0 + 10 \]

### (forward) DSE | bb-DSE

<table>
<thead>
<tr>
<th></th>
<th>(forward) DSE</th>
<th>bb-DSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>feasibility queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infeasibility queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scale</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**EXPERIMENTAL EVALUATION**

- **Controlled experiments** *(ground truth)* → **precision**
- **Large-scale experiment: packers** → **scalability, robustness**
- **Case-study: X-tunnel malware** → **usefulness**
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></td>
<td>159</td>
<td>0</td>
</tr>
<tr>
<td>ASPack v2.12</td>
<td>377K</td>
<td>1</td>
<td></td>
<td></td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>Crypted v1.12</td>
<td>1.1K</td>
<td>1</td>
<td></td>
<td></td>
<td>24</td>
<td>125</td>
</tr>
<tr>
<td>Expressor</td>
<td>635K</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>FSG v2.0</td>
<td>68k</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Mew</td>
<td>59K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>28</td>
<td>6</td>
</tr>
<tr>
<td>PE Lock</td>
<td>2.3K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>95</td>
<td>90</td>
</tr>
<tr>
<td>RLPack</td>
<td>941K</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>TELock v0.51</td>
<td>466K</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Upack v0.39</td>
<td>711K</td>
<td>1</td>
<td>1</td>
<td></td>
<td></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: PACKERS (fun facts)

Several of the tricks detected by the analysis

<table>
<thead>
<tr>
<th>OP in ACProtect</th>
<th>CST in ACProtect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1018f7a js 0x1018f92</td>
<td>10043a9 mov [ebp+0x3a8], eax</td>
</tr>
<tr>
<td>1018f7c jns 0x1018f92</td>
<td>10043af popa 0x10043bb at runtime</td>
</tr>
</tbody>
</table>

(and all possible variants ja/jbe, jp/jnp, jo/jno..)

<table>
<thead>
<tr>
<th>OP in Armadillo</th>
<th>CST in ACProtect</th>
</tr>
</thead>
<tbody>
<tr>
<td>10330ae xor ecx, ecx</td>
<td>1001000 push 16793600</td>
</tr>
<tr>
<td>10330b0 jnz 0x10330ca</td>
<td>1001005 push 16781323</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OP (decoy) in ASPack</th>
<th>CST in ASPack</th>
</tr>
</thead>
<tbody>
<tr>
<td>10040fe: mov bl, 0x0</td>
<td>10043ba push 0x10011d7</td>
</tr>
<tr>
<td>10041c0: cmp bl, 0x1</td>
<td>10043bf ret</td>
</tr>
<tr>
<td>1004103: jnz 0x1004163</td>
<td>Enter SMC Layer 1</td>
</tr>
</tbody>
</table>

ZF = 0

ZF = 1

1004163: jmp 0x100416d

1004105: inc [ebp+0xec]

Sébastien Bardin et al. – Virus Bulletin 2017 | 28
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
CASE-STUDY: THE XTUNNEL MALWARE (fun facts)

- Protection seems to rely only on opaque predicates
- Only two families of opaque predicates
- Yet, quite sophisticated
  - original OPs
  - interleaving between payload and OP computation
  - sharing among OP computations
  - possibly long dependencies chains (avg 8.7, upto 230)
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
<table>
<thead>
<tr>
<th></th>
<th>Feasibility</th>
<th>Infeasibility</th>
<th>Efficient</th>
<th>Robust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static syntactic</td>
<td>x</td>
<td>--</td>
<td>OK</td>
<td>x</td>
</tr>
<tr>
<td>Dynamic</td>
<td>--</td>
<td>x</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>DSE</td>
<td>OK</td>
<td>x</td>
<td>x</td>
<td>OK</td>
</tr>
<tr>
<td>BB-DSE</td>
<td>x</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>
FUTURE DIRECTION: SPARSE DISASSEMBLY

- **dynamic disassembly**
- **partial safe CFG**
- **static disassembly**
- **dynamic symbolic execution**
- **execution trace**
- **new input**
- **obfuscation information**
CONCLUSION & TAKE AWAY

• A tour on the advantages of symbolic methods for deobfuscation

• Semantic analysis complements existing approaches
  • Explore, prove infeasible, simplify
  • Open the way to fruitful combinations

• 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

• Next Step
  • Combines with user and learning!
  • Anti-anti-DSE