BINARY-LEVEL SECURITY: SEMANTIC ANALYSIS TO THE RESCUE

Sébastien Bardin (CEA LIST)
Joint work with
Richard Bonichon, Robin David, Adel Djoudi & many other people
• Binary-level security analysis
  • Many applications, many challenges
  • Current syntactic and dynamic methods are not enough

• Formal methods can change the game … but must be strongly adapted
  • Complement existing methods
  • Need robustness and scalability!
  • Acceptable to lose both correctness & completeness – in a controlled way
  • New challenges and variations, many things to do!

• Achievements
  • Vulnerability detection [with Josselin Feist, Laurent Mounier and Marie-Laure Potet]
  • Malware deobfuscation [with Jean-Yves Marion]
  • BINSEC platform: Open-source, still in its infancy, on heavy refactoring
ABOUT MY LAB @CEA

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
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
NEXT BIG CHALLENGE

- Apply formal methods to less-critical software
- Very different context: no formal spec, less developer involvement, etc.

Some difficulties
- robustness [w.r.t. software constructs]
- no place for false alarms
- scale
- no model, sometimes no source

Guidelines & trends
- find sweet spots [drivers]
- manage abstractions
- reduction to logic

Sébastien Bardin et al. – pré-GDR Sécurité 2017
NOW: BINARY-LEVEL SECURITY

**Model**

```
x := a + b
```

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

**Source code**

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

**Assembly**

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

label:
    move @100 B
```

**Executable**

```
ABFF780BD70696CA101001BDE45
145634789234A8FEE678ABDCF456
5A284C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```
BENEFITS

No source code  More precise analysis  Malware

What for: vulnerabilities, reverse (malware, legacy), protection evaluation, etc.
EXAMPLE: COMPILER BUG

- Optimizing compilers may remove dead code
- pwd never accessed after memset
- Thus can be safely removed
- And allows the password to stay longer in memory

Security bug introduced by a non-buggy compiler

```c
void getPassword(void) {
    char pwd [64];
    if (GetPassword(pwd, sizeof(pwd))) {
        /* checkpassword */
    }
    memset(pwd, 0, sizeof(pwd));
}
```

OpenSSH CVE-2016-0777

Our goal here:
- Check the code after compilation
EXAMPLE: 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**

**Highly challenging [obfuscation]**
CHALLENGE: CORRECT DISASSEMBLY

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

- code – data
- dynamic jumps (jmp eax)

### Sections

<table>
<thead>
<tr>
<th>.text</th>
<th>.fini</th>
<th>.rodata</th>
</tr>
</thead>
<tbody>
<tr>
<td>8D 4C 24 04 83 E4 F0 FF 71 FC 55 89 E5 53 51 83</td>
<td>90 90 90 90 90 90</td>
<td>F3 C3 FF FF 53 83 EC 0E 4E 13 FF</td>
</tr>
<tr>
<td>EC 10 89 CB 83 EC 0C 6A 0A 1E A7 FE FF FF 83 C4</td>
<td>90 90 90 90 90 90</td>
<td>EB 0D 90 90 90 90 90</td>
</tr>
<tr>
<td>10 89 45 F0 8B 43 04 83 C0 04 8B 00 83 EC 0C 50</td>
<td>90 90 90 90 90 90</td>
<td>76 61 6C 3A 25 64 0A 00</td>
</tr>
<tr>
<td>E8 C0 FF FF FF 83 C4 10 89 45 F4 83 7D F4 04 77</td>
<td>90 90 90 90 90 90</td>
<td>EB 0D 90 90 90 90 90</td>
</tr>
<tr>
<td>3B 8B 45 F4 C1 E0 02 05 98 85 04 08 8B 0F FF E0</td>
<td>90 90 90 90 90 90</td>
<td>76 61 6C 3A 25 64 0A 00</td>
</tr>
<tr>
<td>C7 45 F4 00 00 00 00 00 EB 23 C7 45 F4 01 00 00 00</td>
<td>90 90 90 90 90 90</td>
<td>EB 0D 90 90 90 90 90</td>
</tr>
<tr>
<td>EB 1A C7 45 F4 02 00 00 00 EB 11 C7 45 F4 03 00</td>
<td>90 90 90 90 90 90</td>
<td>EB 0D 90 90 90 90 90</td>
</tr>
<tr>
<td>90 00 00 00 EB 0B C7 45 F4 04 04 00 00 00 90 83 EC 08 FF</td>
<td>90 90 90 90 90 90</td>
<td>EB 0D 90 90 90 90 90</td>
</tr>
<tr>
<td>75 F4 68 90 85 04 08 EB 29 FE FF FF 83 C4 10 8B</td>
<td>90 90 90 90 90 90</td>
<td>EB 0D 90 90 90 90 90</td>
</tr>
<tr>
<td>45 F4 8D 65 F8 59 5B 5D 8D 61 FC C3 66 90 66 90</td>
<td>90 90 90 90 90 90</td>
<td>EB 0D 90 90 90 90 90</td>
</tr>
</tbody>
</table>

### Code (Functions)

- main
- unknown
- _libc_csu_init
- unknown
- _libc_csu_fini
- _term_proc
- _fp_hw, _IO_stdin_used

```
“va\%d\n”
switch jump table
```

### Assembly

```
rep retn
push ebx
sub esp, 8
call get_pc[..]
add ebx, 0x1217
add esp, 8
pop ebx
retn
```
STATE-OF-THE-ART TOOLS ARE NOT ENOUGH

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

Just add

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

and break results

With IDA
[See later] CAN BECOME A NIGHTMARE

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

\[
\begin{align*}
\text{mov eax, ds:X} \\
\text{mov ecx, ds:Y} \\
\text{imul ecx, ecx} \\
\text{imul ecx, 7} \\
\text{sub ecx, 1} \\
\text{imul eax, eax} \\
\text{cmp ecx, eax} \\
\text{jz <dead_addr>}
\end{align*}
\]
EXAMPLE: VULNERABILITY DETECTION

Find vulnerabilities before the bad guys
• On the whole program
• At binary-level
• Know only the entry point and program input format
CHALLENGE: In-depth exploration (example: use after free)

- sequence of events, importance of aliasing
- strongly depend on implem of malloc and free

Dynamic analysis
• Too incomplete
### BONUS: (MULTI-)ARCHITECTURE SUPPORT

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four prefixes of 1 byte each (optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

Example of x86:
- more than 1,000 instructions
  - ≈ 400 basic
  - + float, interrupts, mmx
- many side-effects
- error-prone decoding
  - addressing modes, prefixes, ...

[Diagram of x86 instruction encoding]

---

Sébastien Bardin et al. – pré-GDR Sécurité 2017 | 16
THE SITUATION

- Binary-level security analysis is necessary
- Binary-level security analysis is highly challenging (*)
- Standard tools are not enough

Some challenges
- Correct disassembly (code identification)
- Multi-architecture support
- + All problems from source-level analysis

(*) i.e., more challenging than source code analysis
SOLUTION? BINARY-LEVEL SEMANTIC ANALYSIS

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

Advantages
- more robust than syntactic
- more thorough than dynamic

Challenges
- source-level ↔ binary-level
- safety ↔ security
- many (complex) architectures
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
• A simple Intermediate Representation
• A set of « standard » analysis
  • Adapted to binary
  • Robustness in mind
  • Keep correct/complete as much as possible
• Domain-dedicated analysis [combination]
Can recover useful semantic information
• More precise disassembly
• Exact semantic of instructions
• Input of interest
• …
KEY 1: INTERMEDIATE REPRESENTATION
[cav’11]

- Concise
- Well-defined
- Clear, side-effect free

(lhs := rhs)
(goto addr, goto expr)
(ite(cond)? goto addr)

```
(0x29e,0) tmp := EBX + 7511;
(0x29e,1) OF := (EBX[31,31]=7511[31,31]) && (EBX{31,31}<><tmp{31,31});
(0x29e,2) SF := tmp{31,31};
(0x29e,3) ZF := (tmp = 0);
(0x28e,4) AF := ((extu (EBX{0,7}) 9) + (extu 7511{0,7} 9)){8,8};
(0x29e,6) CF := ((extu EBX 33) + (extu 7511 33)){32,32};
(0x29e,7) EBX := tmp; goto (0x2a4,0)
```
KEY 1: INTERMEDIATE REPRESENTATION (2)  
-- simplifications [tacas’15 ]

<table>
<thead>
<tr>
<th>program</th>
<th>native loc</th>
<th>DBA loc</th>
<th>opt (DBA) loc</th>
<th>time</th>
<th>loc</th>
<th>red</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>166K</td>
<td>559K</td>
<td>673.61s</td>
<td>389K</td>
<td>30.45%</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>8K</td>
<td>23K</td>
<td>18.54s</td>
<td>18K</td>
<td>23.02%</td>
<td></td>
</tr>
<tr>
<td>echo</td>
<td>4K</td>
<td>10K</td>
<td>6.96s</td>
<td>8K</td>
<td>24.26%</td>
<td></td>
</tr>
<tr>
<td>less</td>
<td>23K</td>
<td>80K</td>
<td>69.99s</td>
<td>55K</td>
<td>30.96%</td>
<td></td>
</tr>
<tr>
<td>ls</td>
<td>19K</td>
<td>63K</td>
<td>65.69s</td>
<td>44K</td>
<td>30.58%</td>
<td></td>
</tr>
<tr>
<td>mkdir</td>
<td>8K</td>
<td>24K</td>
<td>19.74s</td>
<td>17K</td>
<td>29.50%</td>
<td></td>
</tr>
<tr>
<td>netstat</td>
<td>17K</td>
<td>50K</td>
<td>52.59s</td>
<td>40K</td>
<td>20.05%</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>12K</td>
<td>36K</td>
<td>36.99s</td>
<td>27K</td>
<td>23.98%</td>
<td></td>
</tr>
<tr>
<td>pwd</td>
<td>4K</td>
<td>11K</td>
<td>7.69s</td>
<td>9K</td>
<td>23.56%</td>
<td></td>
</tr>
<tr>
<td>rm</td>
<td>10K</td>
<td>30K</td>
<td>24.93s</td>
<td>22K</td>
<td>25.24%</td>
<td></td>
</tr>
<tr>
<td>sed</td>
<td>10K</td>
<td>32K</td>
<td>28.85s</td>
<td>23K</td>
<td>26.20%</td>
<td></td>
</tr>
<tr>
<td>tar</td>
<td>64K</td>
<td>213K</td>
<td>242.96s</td>
<td>154K</td>
<td>27.48%</td>
<td></td>
</tr>
<tr>
<td>touch</td>
<td>8K</td>
<td>26K</td>
<td>24.28s</td>
<td>18K</td>
<td>27.88%</td>
<td></td>
</tr>
<tr>
<td>uname</td>
<td>3K</td>
<td>10K</td>
<td>6.99s</td>
<td>8K</td>
<td>23.62%</td>
<td></td>
</tr>
</tbody>
</table>

Approach

- Inspired from standard compiler optim
- Targets: flags & temp
- Sound: w.r.t. incomplete CFG
- Inter-procedural (summaries)

- DBA level
- Instruction level
- Program level
KEY 2: PRECISE & ROBUST SYMBOLIC REASONING
(DSE, Godefroid 2005)

Perfect for intensive testing

• Correct, relatively complete
• No false alarm
• Robust
• Scale in some ways

// incomplete

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

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths
KEY 2: DSE, about ROBUSTNESS  (imo, the major advantage)

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ g(int x) \{ \text{return } x \times x; \} \]
\[ f(int x, int y) \{ z=g(x); \text{ if } (y == z) \text{ ERROR; else OK } \} \]

Symbolic Execution
- create a subformula \( z = x \times x \), out of theory [FAIL]

Dynamic Symbolic Execution
- first concrete execution with \( x=3, \ y=5 \) [goto OK]
- during path predicate computation, \( x \times x \) not supported
  - \( x \) is concretized to 3 and \( z \) is forced to 9
- resulting path predicate: \( x = 3 \land z = 9 \land y = z \)
- a solution is found: \( x=3, \ y=9 \) [goto ERROR] [SUCCESS]
And more … (more or less experimental)
APPLICATION: VULNERABILITY DETECTION
[SSPREW 2016, with VERIMAG]

Find vulnerabilities before the bad guys
• On the whole program
• At binary-level
• Know only the entry point and program input format
KEY IDEAS (Josselin Feist)

A Pragmatic 2-step approach
- Static: scale, not complete, not correct
- Symbolic: correct, directed by static
- Combination: scalable and correct
EXPERIMENTAL EVALUATION

- GUEB + manual analysis [j. comp. virology 14]
  - tiff2pdf : CVE-2013-4232
  - openjpeg : CVE-2015-8871
  - gifcolor : CVE-2016-3177
  - accel-ppp

- GUEB + BINSE/SE [sspew16]
  - Jasper JPEG-2000 : CVE-2015-5221

On these examples:
- Better than DSE alone
- Better than blackbox fuzzing
- Better than greybox fuzzing with no seed
APPLICATION: MALWARE DEOBfuscATION
[S&P 2017, with LORIA]

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

Goal: help malware comprehension
- Identify and simplify protections

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

```
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

<table>
<thead>
<tr>
<th>address</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>80483d1</td>
<td>call +5</td>
</tr>
<tr>
<td>80483d6</td>
<td>pop edx</td>
</tr>
<tr>
<td>80483d7</td>
<td>add edx, 8</td>
</tr>
<tr>
<td>80483da</td>
<td>push edx</td>
</tr>
<tr>
<td>80483db</td>
<td>ret</td>
</tr>
<tr>
<td>80483dc</td>
<td>.byte{invalid}</td>
</tr>
<tr>
<td>80483de</td>
<td>[...]</td>
</tr>
</tbody>
</table>
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 (Debray, Kruegel, …)

```
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
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
- pre_k(c)=0 => c 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 \-constraints)

False positive (FP)
- wrongly assert infeasibility
- why: CFG too partial (miss ∨-constraints)
### EXPERIMENTAL EVALUATION

<table>
<thead>
<tr>
<th>Controlled experiments (ground truth)</th>
<th>precision</th>
</tr>
</thead>
<tbody>
<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 util, 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 util
  • almost all genuine ret are proved (no false positive)
  • many malicious ret are proved « single-targets »

• Very precise results
• 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>ACP protect v2.0</td>
<td>1.8K</td>
<td>1</td>
<td>1</td>
<td>0</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>0</td>
</tr>
<tr>
<td>FSG v2.0</td>
<td>68k</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Mew</td>
<td>59K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>PE Lock</td>
<td>2.3M</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>0</td>
</tr>
<tr>
<td>TELock v0.51</td>
<td>466K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Unpack v0.39</td>
<td>711K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</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)
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)
CONCLUSION & TAKE AWAY

• Binary-level security analysis
  • Many applications, many challenges
  • Current syntactic and dynamic methods are not enough

• Formal methods can change the game … but must be strongly adapted
  • Complement existing approaches
  • Need robustness and scalability!
  • Acceptable to lose both correctness & completeness – in a controlled way
  • New challenges and variations, many things to do!

• First results
  • Advanced pocs and vulnerability detection and malware deobfuscation
  • BINSEC: open-source, still in its infancy, on heavy refactoring

// Advertisement: International Summer School on Software Protection @ Saclay