FROM SOURCE-LEVEL SAFETY TO BINARY-LEVEL SECURITY

Sébastien Bardin (CEA LIST)

With Richard Bonichon, Matthieu Lemerre, Robin David, Josselin Feist, Adel Djoudi, Benjamin Farinier, etc.
<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
The SMACC Mcopter: 18-Month Assessment

- The SMACC Mcopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.

- Air Team proved system-wide security properties:
  - The system is memory safe.
  - The system ignores malformed messages.
  - The system ignores non-authenticated messages.
  - All "good" messages received by SMACC Mcopter radio will reach the motor controller.

- Red Team:
  - Found no security flaws in six weeks with full access to source code.

- Penetration Testing Expert:
  The SMACC Mcopter is probably "the most secure UAV on the planet."

Open source: autopilot and tools available from http://smaccmcopter.org
NOW: BINARY-LEVEL SECURITY

Model

\[
x > 0 \land x := x - 1
\]
\[
x := a + b
\]
\[
x = 0
\]

Source code

```c
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

```
ABFFF780BD70696CA101001BDE45
145634789234A8FFE678ABDCF436
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAABBDA457345FD780001
FFF22546ADDAE989776600000000
```
OUTLINE

• Why binary-level analysis?

• The hard journey from source to binary

• Our approach

• Lessons learned

• Conclusion
WHY?

Not all source code available

- vulnerabilities
- program analysis
- formal verification

Interested in low-level properties

- vulnerabilities
- side channels
- binary-level protection

No source code at all

- vulnerabilities
- reverse legacy
- malware
EXAMPLES

Vulnerability analysis

Malware comprehension

Find a needle in the heap!
BUT ... THIS IS HARD!!!
DISASSEMBLY IS ALREADY TRICKY!

- code – data
- dynamic jumps (jmp eax)

Sections

.text

8D 4C 24 01 34 02 83 E4 00 FF FF 71 FC 55 89 E5 53 51 83
EC 10 09 CB 10 EC 0C 6A 0A EB 08 0F 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 05 45 F4 83 7D F4 04 77
3B 8B 45 FF 41 05 02 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 4C 45 F4 02 00 00 00 EB 1C 4C 45 F4 03 00
00 00 EB 08 CB 45 F4 04 00 00 00 00 89 83 EC 0F
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 EC C3 66 90 66 90
66 90 66 90 SF 53 51 83 FF 56 53 EB 85 FE FF FF
81 C3 01 00 00 00 83 EC 1B 8B 6C 24 30 8D 83 0C
FF FF FF EE 08 FF FF FF 8D 83 08 FF FF FF 29 C6
C1 02 85 EF 67 27 8D B6 00 00 00 00 8B 44 2A
3B 89 2C 24 89 44 28 88 8B 44 2A 44 28 39 4A 24 04
FF 94 BB 00 FF FF FF FF 83 C7 01 39 FF 75 DF 83 C4
1C 5B 5F 5F 5F 5D 15 0B 0D 0D 00 0D 00 0D 00 0D
90 90 90 90 90 90 90 EC C3 90 FF FF 53 83 EC 08 EB 13 FE
FF FF FF 83 C1 12 00 00 83 EC 08 0B 05 FF FF
C1 12 00 00 9C 00 90 00 90 00 90 00 90 00 90 00 90
90 01 00 02 00 8E 16 0C 25 8D 0A 00 03 8B 04 04
08 0A 84 04 08 8D 84 04 08 8C 84 04 08 0B 84 04
08 0A 01 00 02 00 03 28 00 00 00 00 00 00 00 84 FF

.fini .rodata

.eh_frame_hdr

Code

(FUNCTIONS)

main

unknown

__libc_csu_init

unknown

__libc_csu_fini

__term_proc

switch jump table

Assembly

push ebx
sub esp, 8
call get_pc[..]
add ebx, 0x1217
add esp, 8
pop ebx
retn

rep retn

sub esp, 8
push ebx

institut cambridge
BUT … THIS IS HARD!!!

Static (syntactic)
• too fragile (program variations)

Dynamic
• too incomplete (rare events)
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 compilation or obfuscation

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

Advantages
- more robust than syntactic
- more thorough than dynamic

Challenges
- source-level $\mapsto$ binary-level
- safety $\mapsto$ security
- many (complex) architectures
OK but … WHICH APPROACH?  (Formal Method Zoo)
And HOW TO?

- Abstract interpretation
- Model Checking
- Symbolic model checking
- Bounded model checking
- Counter-example guided model checking
- Interpolation-based model checking
- k-induction
- ...

- Weakest precondition
- Property-directed reachability
- Symbolic execution
- Interactive theorem proving
- Type systems
- Correct by construction
- Method B
- .....
OUTLINE

• Why binary-level analysis?
• The hard journey from source to binary
• Our approach
• Lessons learned
• Conclusion
WANTED

Robustness
- able to survive dynamic jumps, self-modification, unpacking, etc
- *outside the scope of standard methods*

Precision
- Machine arithmetic (overflow) and bit-level operations
- Byte-level memory, possible overlaps
- *hard for soa formal methods*

Scale
THE GOOD CANDIDATE: SYMBOLIC EXECUTION (Godefroid, 2005)

Given a path of a program
- Compute its « path predicate » $f$
- Solution of $f$ $\Leftrightarrow$ input following the path
- Solve it with powerful existing solvers

```c
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```
THE GOOD CANDIDATE: SYMBOLIC EXECUTION
(Godefroid, 2005)

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

Good points:
- **Precise** (theory bitvectors + arrays)
- **No false positive**
- **Robust** (symb. + dynamic)
- Extend rather well to binary code

Given a path of a program
- Compute its « path predicate » f
- Solution of f ⇔ input following the path
- Solve it with powerful existing solvers
THE GOOD CANDIDATE: SYMBOLIC EXECUTION
(Godefroid, 2005)

Given a path of a program
• Compute its « path predicate » f
• Solution of f ⇔ input following the path
• Solve it with powerful existing solvers

Good points:
• No false positive = find real paths
• Robust (symb. + dynamic)
• Precise (theory bitvectors + arrays)
• Extend rather well to binary code

« concretization »
• Replace symbolic values by runtime values
• Keep going when symbolic reasoning fails
• Tune the tradeoff genericity - cost

\[ \sigma := \emptyset \]
\[ \mathcal{P} C := \top \]
\[ x = \text{input()} \]
\[ \sigma := \emptyset \]
\[ \mathcal{P} C := \top \]
\[ x > y + 10 \]
\[ \mathcal{P} C := \top \land 2y_0 = x_0 \land x_0 \leq y_0 + 10 \]

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

• Why binary-level analysis?
• The hard journey from source to binary
• Our approach
• Lessons learned
• Conclusion
BINSEC: SYMBOLIC ANALYSIS for BINARY

Rely on variants of Symbolic Execution

malware analysis

vulnerabilities

• Explore
• Prove
• Simplify

lhs := rhs

goto addr, goto expr

ite(cond)? goto addr:

assume, assert, nondet
PART I: EXPLORE (standard SE)

Forward reasoning
- Follows path
- Find new branch / jumps
- Standard DSE setting

Advantages
- Find new real paths
- Even rare paths

« dynamic analysis on steroids »
EXAMPLE: FIND THE GOOD PATH

Grub2 CVE 2015-8370
Elevation of privilege
Information disclosure
Denial of service

Crackme challenges
- input == secret → success
- input ≠ secret → failure
EXAMPLE: FIND THE GOOD PATH

Crackme challenges

- input == secret → success
- input ≠ secret → failure

Beware: scale?
CASE STUDY: VULNERABILITY DETECTION
[SSPREW’16](with Josselin Feist et al.)

Find a needle in the heap!

Static analysis
GUEB
Weighted slice
UaF detection
Sieve extraction
Dynamic symbolic execution
BINSEC
Inputs generation
UaF validation
PoC
Entry point
free
use
CASE STUDY: VULNERABILITY DETECTION
[SSPREW’16](with Josselin Feist et al.)

A Pragmatic 2-step approach
- Step 1: Incorrect but scalable
- All: scalable and correct

- Find a few new CVEs
- Much better than AFL here
PART II: PROVE

Backward bounded SE
- Compute k-predecessors
- If the set is empty, no pred.
- Allows to prove things
BACKWARD BOUNDED SE

- Compute k-predecessors
- If the set is empty, no pred.
- Allows to prove things

- False Negative: k too small
  - Missed proofs
- False Positive: CFG incomplete
  - Wrong proofs (low rate, controlled XPs)
IN PRACTICE

eq: $7y^2 - 1 \neq x^2$
(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>

if (ax > bx) X = -1;
else X = 1;

GF := ((ax{31,31}|bx{31,31}) &
(ax{31,31}|(ax-bx){31,31}));
SF := (ax-bx) < 0;
ZF := (ax-bx) = 0;
if (~ZF & (GF = SF)) goto 11
X := 1
goto 12
11: X := -1
12:

• Scalable switch target recovery
• Opaque predicate detection
• Call stack tampering
• High-level condition recovery

With IDA + BINSEC
CASE-STUDY: THE XTUNNEL MALWARE    [S&P’17]
-- part of DNC hack    (with Robin David)

Two heavily obfuscated samples
• Many opaque predicates

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

<table>
<thead>
<tr>
<th></th>
<th>C637 Sample #1</th>
<th>99B4 Sample #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#total instruction</td>
<td>505,008</td>
<td>434,143</td>
</tr>
<tr>
<td>#alive</td>
<td>+279,483</td>
<td>+241,177</td>
</tr>
</tbody>
</table>
OUTLINE

- Why binary-level analysis?
- The hard journey from source to binary
- Our approach
- Lessons learned
- Conclusion
KEY PRINCIPLES

• Robustness & precision:
  • dynamic symbolic execution

• Loss of guarantees
  • Accept … But control!
  • Look for « correct enough » solutions

• Finely tune the technology
SEVERAL WAYS TO CONTROL THE LOSS

• Use incorrect-incomplete approach, combined with a correct one

• Relative correctness/completeness in some restricted cases
  • proof: « Relative correctness » // if CFG ok then over-approx
  • explore: « Relative completeness » // find all bugs in k steps

• Degraded mode with a clear understanding where loss occurs

• Controlled experiments with ground truth
  • Assert %FP and %FN on (hopefully) representative benchmarks
• SMT solvers are powerful weapons

• But (binary-level) security problems are terrific beasts

• Mastery can make the difference!

An example: scalability

Array theory
• Necessary
• Hard for solvers!
Example: array formula simplification [LPAR 2018] with Benjamin Farinier

- Makes the difference!

<table>
<thead>
<tr>
<th>no block cypher</th>
<th>Z3</th>
<th>all arrays</th>
<th>non initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>no simplification</td>
<td>0 606.7</td>
<td>1448301</td>
<td>1448001</td>
</tr>
<tr>
<td>list-16</td>
<td>0 501.0</td>
<td>1075358</td>
<td>1052786</td>
</tr>
<tr>
<td>list-256</td>
<td>0 371.9</td>
<td>807778</td>
<td>762673</td>
</tr>
<tr>
<td>map</td>
<td>0 370.5</td>
<td>807778</td>
<td>762673</td>
</tr>
<tr>
<td>LMBN</td>
<td>0 46.0</td>
<td>65788</td>
<td>5044</td>
</tr>
</tbody>
</table>

- Huge formula obtained by dynamic symbolic execution
- 293 000 select
- 24 hours of resolution!

Using LMBN
- #select reduced to 2 467
- 14 sec for resolution
- 61 sec for preprocessing

Using list representation
- Same result with a bound of 385 024 and beyond...
- ...but 53 min preprocessing
OUTLINE

• Why binary-level analysis?

• The hard journey from source to binary

• Our approach

• Lessons learned

• Conclusion
• 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
  • Much better if specifically tuned for the problem at hand

• New challenges and variations, many things to do!

• Thanks for your attention!