FROM SAFETY TO SECURITY: The Case of Binary-Level Code Analysis

Sébastien Bardin
(CEA LIST, France)

With Richard Bonichon, Matthieu Lemerre, Robin David, Josselin Feist, Adel Djoudi, Benjamin Farinier, etc.
ABOUT MY LAB @CEA

CEA LIST, Software Safety & Security Lab

- Located in Paris, France

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

Software Analyzers:

- frama
- GATel
- PathCrawler

- POLAR SSL
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

- Located in Paris, France

Sébastien Bardin -- 5th France-Japan Cybersecurity Workshop, 2019
Binary-level security analysis: many applications, many challenges

Standard techniques not enough

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

This talk: our experience on adapting source-level safety analysis for binary-level security
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
A DREAM COME TRUE … IN CERTAIN DOMAINS

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: MOVING TO BINARY-LEVEL SECURITY ANALYSIS

Sébastien Bardin -- 5th France-Japan Cybersecurity Workshop, 2019

Source code:

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

Model:

```
x > 0 / x := x - 1
x := a + b
x = 0 /
```

Assembly:

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

label:
    move @100 B
```

Executable:

```
ABFFF780BD70696CA1010018DE45
145634789234ABFEE678ABDCE856
5A284C6009F5D1E08357515897
345FEBDCA3CBDA459700346901
3456KAHA305G67H345BF8FADECD3
00113456735FFD451E13A0800DAD
344252F8A8D8A457345FD7800001
FFF22546ADDE98977600000000
```
OUTLINE

• The success of formal methods for safety

• Why binary-level security analysis?

• The hard journey from source-level safety to binary-level security

• Our approach

• Conclusion
WHY BINARY-LEVEL SECURITY ANALYSIS?

Malware comprehension

Protection-evaluation

Vulnerability analysis
BUT ... THIS IS HARD!!!
DISASSEMBLY IS ALREADY TRICKY!

- code – data ??
- dynamic jumps (jmp eax)

---

### Sections

```
.text
8D 4C 24 04 83 E4 F0 FF 71 FC 55 89 E5 53 83 C4
EC 10 89 CB 83 EC 0C 6A 0A E8 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 FF 83 C4 10 89 45 F4 83 7D F4 04 77
3B 8B 45 F4 C1 E0 02 05 98 85 04 08 8B 00 FF E0
75 F4 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 00 EB 11 C7 45 F4 03 00
00 00 EB 08 C7 45 F4 04 00 00 00 00 90 83 EC 08 FF
75 F4 68 90 85 04 08 EB 29 FE FF FF FF 83 C4 10 8B
45 F4 8D 65 F8 59 8B 5D 8D 61 FC C3 66 90 66 90
66 90 66 90 90 55 57 31 FF 56 53 E8 85 FE FF FF
81 C3 89 12 00 08 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 F6 74 27 8D B6 06 00 00 00 8B 44 24
3B 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
FF 94 BB 08 FF FF FF 83 C7 01 39 FF FF FF 83 C4
1C 5B 5E 5F 5D C3 EB 0D 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
`.fini

.rodata

.eh_frame_hdr
```
AND IT CAN GET WORST! (adversarial setting)

- 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)

```
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>
```
An old proverb comes true

Key concepts: $M \models \varphi$

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

We have a beautiful hammer

And it seems we found a nail 😊
OUTLINE

• The success of formal methods for safety

• Why binary-level security analysis?

• The hard journey from source-level safety to binary-level security

• Our approach

• Conclusion
BINARY-LEVEL ANALYSIS

- Low-level control (CFG?)
- Low-level data & memory

Break an implicit assumption in code analysis

At the edge of current methods
ATTACKER <not treated today>

Nature is not nice

Attacker is evil

From Florent Kirchner
NOT STRONGLY REGULATED ECOSYSTEM

- No coding guideline
- No annotation
- Require full automation
- Low tolerance to false positive

But absolute correctness is not required
  • « correct enough »
SECURITY IS NOT SAFETY

• **Source-level SAFETY**
  - Model: High-level language
  - Properties: safety
  - Algorithm: full correctness, possible help from user

• **Binary-level SECURITY**
  - Model: binary-level code, possibly adversarial, + attacker
  - Properties: safety, k-safety, « bugs vs vulnerabilities »
  - Algorithm: robust & precise enough, fully automated

- Strong incentive to human assistance
- Spec., parameter tuning, etc.
- no human assistance
- low tolerance to false positive
<apparté> STATIC SEMANTIC ANALYSIS IS VERY VERY HARD ON BINARY CODE

Framework: abstract interpretation
- notion of abstract domain
  \( \perp, \top, \sqcup, \sqcap, \subseteq, \text{eval}\#
- more or less precise domains
  . intervals, polyhedra, etc.
- fixpoint until stabilization

Problems
- Jump eax
- memory
- Bit reasoning
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 state-of-art formal methods*

Reasonable scale
OUTLINE

• The success of formal methods for safety

• Why binary-level security analysis?

• The hard journey from source-level safety to binary-level security

• Our approach

• Conclusion
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

```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)

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

Good points:
- Precise (no false positive)
- Robust (symb. + dynamic)
- Extend rather well to binary code

```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)

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

Good points:
- Precise (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 $\Leftrightarrow$ input following the path
- Solve it with powerful existing solvers

« concretization »
- Replace symbolic values by runtime values
- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost
ALLOWS TO EXPLORE A PROGRAM

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

Advantages
• Find new real paths
• Even rare paths

<< dynamic analysis on steroids >>
OUR KEY PRINCIPLES

• DSE is a good starting point for robustness & precision

• Can be adapted beyond the basic reachability case
  • variants
  • combination with other techniques

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

• Finely tune the technology
  • Tools for safety are not fully adequate
CASE 1: COMPLEX VULNERABILITY DETECTION
[SSPREW’16](with Josselin Feist et al.)

Use-after-free bugs
- Very hard to find
- Sequence of events
- DSE lost

Find a needle in the heap!
CASE 1: COMPLEX VULNERABILITY DETECTION
[SSPREW’16](with Josselin Feist et al.)

A Pragmatic 2-step approach
- Step 1: incorrect but scalable
- Steps 1+2: scalable and correct
CASE 1: COMPLEX VULNERABILITY DETECTION
[SSPREW’16](with Josselin Feist et al.)

**A Pragmatic 2-step approach**
- Step 1: incorrect but scalable
- Steps 1+2: scalable and correct

- Find a few new CVEs
- Much better than AFL here
CASE 2: BINARY-LEVEL PROOF

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>
```

The predicate is always true

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

The two blocks are equivalent

With IDA + BINSEC

All jump targets are found

Not addressed by DSE

- Cannot enumerate all paths

Sébastien Bardin -- 5th France-Japan Cybersecurity Workshop, 2019 | 30
Case 2: BINARY-LEVEL PROOF

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

- False Negative: $k$ too small
  - Missed proofs
- False Positive: CFG incomplete
  - Wrong proofs ?!
Wait …

- False Negative: $k$ too small
  - Missed proofs
- False Positive: CFG incomplete
  - Wrong proofs

- Low rate of wrong proofs
- Controlled XPs

Two heavily obfuscated samples
• Many opaque predicates

Goal: detect & remove protections
• Identify 45% of code as spurious
• Fully automatic, < 3h
CASE 3: finely tuning the technology

- SMT solvers are powerful weapons
- But (binary-level) security problems are terrific beasts
- Need to adapt them!

Two examples
- Scalability [LPAR 2018, Benjamin Farinier]
- Robustness [CAV 2018, Benjamin Farinier]
Example 1: scalability (reasoning about memory)

Array theory
- Necessary to model memory

ROW rule may introduce case-splits

Hard for solvers
- Case-splits

- Reading in \( a \) at index \( i \in \mathcal{I} \): select \( a_i \)
- Writing in \( a \) an element \( e \in \mathcal{E} \) at index \( i \in \mathcal{I} \): store \( a_{i\to e} \)

\[
\text{select} : \text{Array} \mathcal{I} \mathcal{E} \to \mathcal{I} \to \mathcal{E}
\]
\[
\text{store} : \text{Array} \mathcal{I} \mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array} \mathcal{I} \mathcal{E}
\]

\[
\forall a\ i\ e. \text{select} (\text{store} a_{i\to e}) i = e
\]
\[
\forall a\ i\ j\ e. (i \neq j) \Rightarrow \text{select} (\text{store} a_{i\to e}) j = \text{select} a_j
\]

Prevalent in software analysis
- Modelling memory
- Abstracting data structure
  (map, queue, stack...)

Hard to solve
- NP-complete
- ROW may require case-splits
Not pure theory!

Reverse of a ASPACK-protected code

Huge formula obtained by dynamic symbolic execution

293 000 select

24 hours of resolution!
Example 1: scalability (reasoning about memory)

- Our goal: dedicated formula preprocessing to remove « RoW »
  - Problem 1: standard « list »-representation for logical arrays induces a quadratic time preprocessing → prohibitive
  - Problem 2: need to cheaply but precisely reason about index equalities
Example 1: scalability (reasoning about memory)

- Dedicated data structure (list-map)
- Tuned for base+offset access
- Linear complexity

Propagate “variable+constant” terms
- If $y \triangleq z + 1$ then $x \triangleq y + 2 \Rightarrow x \triangleq z + 3$
- Together with associativity, commutativity...

- Reduce the number of bases

• Scale
• Good when only few bases

- Prove disequalities between different bases

Associate to every indices $i$ an abstract domain $i\#$
- If $i\# \cap j\# = \bot$ then $(a[i] \leftarrow e)[j] = a[j]$
- Integrated in the list-map representation
Tuning the solver: array formula simplification [LPAR 2018] with Benjamin Farinier

- Makes the difference!

### Table:

<table>
<thead>
<tr>
<th>no block cypher</th>
<th>Z3</th>
<th>#select</th>
</tr>
</thead>
<tbody>
<tr>
<td>no simplification</td>
<td>0</td>
<td>606.7</td>
</tr>
<tr>
<td>list-16</td>
<td>0</td>
<td>501.0</td>
</tr>
<tr>
<td>list-256</td>
<td>0</td>
<td>371.9</td>
</tr>
<tr>
<td>map</td>
<td>0</td>
<td>370.5</td>
</tr>
<tr>
<td>LMBN</td>
<td>0</td>
<td>46.0</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
Example 2: robust symbolic execution

- Standard symbolic reasoning may produce **false positive**
  - For example here:
    - SE will try to solve \( a \times x + b > 0 \)
    - May return \( a = -100, b = 10, x = 0 \)
  
- Problem: \( x \) is not controlled by the user
  - If \( x \) change, possibly not a solution anymore
  - Example: \((a = -100, b = 10, x = 1)\)
Example 2: robust symbolic execution

- Standard symbolic reasoning may produce **false positive**

- Actually, need to solve $\forall x. ax + b > 0$

- **Quantified formula**
- SMT solvers bad for that
Our solution: reduce quantified formula to the quantifier-free case

- Approximation
- But reuse the whole SMT machinery

Key insights:
- independence conditions
- formula strengthening

Example: robustness and quantification [CAV 2018]
Example: robustness and quantification

<table>
<thead>
<tr>
<th></th>
<th>SE classic</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAT correct</td>
<td>SAT incorrect</td>
<td>UNSAT or UNKNOWN</td>
</tr>
<tr>
<td>Booler</td>
<td>12</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>CVC4</td>
<td>7</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Yices</td>
<td>7</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>Z3</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>SE robust</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAT correct</td>
<td>SAT incorrect</td>
<td>UNSAT or UNKNOWN</td>
</tr>
<tr>
<td>Booler</td>
<td>N/A</td>
<td>N/A</td>
<td>24</td>
</tr>
<tr>
<td>CVC4</td>
<td>5</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>Yices</td>
<td>N/A</td>
<td>N/A</td>
<td>24</td>
</tr>
<tr>
<td>Z3</td>
<td>7</td>
<td>0</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>SE robust + elim.</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAT correct</td>
<td>SAT incorrect</td>
<td>UNSAT or UNKNOWN</td>
</tr>
<tr>
<td>Booler</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>CVC4</td>
<td>7</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Yices</td>
<td>7</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Z3</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

- set of crackme challenge
- compare true and false positive
Example: robustness and quantification

Back to 28: GRUB2 Authentication Bypass

- Original version: press Backspace 28 times to get a rescue shell
- Case study: same vulnerable code turned into a crackme challenge

- SE classic: incorrect solution
- SE robust: solvers TIMEOUT

- SE robust + elim.: correct solution in 80s
- SE robust + elim. + simpl.: correct solution in 30s
OUTLINE

• The success of formal methods for safety

• Why binary-level security analysis?

• The hard journey from source-level safety to binary-level security

• Our approach

• Conclusion
WHERE ARE WE?

Find a needle in the heap!

Find (new) CVEs

Revert protections

**Grub2 CVE 2015-8370**

- Elevation of privilege
- Information disclosure
- Denial of service

**Semantic approaches can work!**

- Explore
- Prove
- Simplify
KEY PRINCIPLES

• Robustness & precision are essential
  • DSE is a good starting point
  • dedicated robust and precise (but not sound) static analysis are feasible

• Can be adapted beyond the basic reachability case
  • variants
  • combination with other techniques

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

• Finely tune the technology
  • Tools for safety are not fully adequate for security
SECURITY IS NOT SAFETY

• Source-level SAFETY
  • Model: High-level language
  • Properties: safety
  • Algorithm: full correctness, possible help from user

• Binary-level SECURITY
  • Model: binary-level code, possibly adversarial, + attacker
  • Properties: safety, k-safety, « bugs vs vulnerabilities »
  • Algorithm: robust & precise enough, fully automated

• strong incentive to human assistance
  • spec., parameter tuning, etc.

• no human assistance
  • low tolerance to false positive

Sébastien Bardin -- 5th France-Japan Cybersecurity Workshop, 2019 | 49
SECURITY IS NOT SAFETY

• **Source-level SAFETY**
  • Model: High-level language
  • Properties: safety
  • Algorithm: full correctness, possible help from user

• **Binary-level SECURITY**
  • Model: *binary-level code, possibly adversarial, + attacker*
  • Properties: safety, k-safety, « bugs vs vulnerabilities » [robust solutions]
  • Algorithm: *robust & precise enough*, fully automated
CONCLUSION & TAKE AWAY

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

• Formal methods can help … 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!