Toward Security-Oriented Program Analysis

Sébastien Bardin
(CEA LIST, University Paris Saclay)

Joint work with the BINSEC group @ CEA
and many other collaborators
Most attacks come from implementation bugs

A (binary-level) program analysis issue!

Sébastien Bardin - Séminaire SoSySec, 2021

int foo(int x, int y) {
    int k = x;
    int c = y;
    while (c > 0) do {
        k++;
        c--;
    }
    return k;
}
WHY ON BINARY CODE?

- No source code
- Post-compilation
- Malware comprehension
- Protection evaluation
- Very-low level reasoning
EXAMPLE: COMPILER BUG (?)

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

- **secure source code**
- **insecure executable**

- 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
BINARY-LEVEL CODE ANALYSIS HAS MANY ADVANTAGES, BUT …
IN A NUTSHELL

• **This talk:** our experience on adapting source-level safety analysis to the case of binary-level security [S&P 17, CAV 18, S&P 20, NDSS 21, CAV 21, etc.]

• **Challenge:** how to move from safety-oriented code analysis to security-oriented code analysis

• **Question:** how does code-level security differ from code-level safety?
BINSEC: brings formal methods to binary-level security analysis

- Explore many inputs at once
- Find bugs
- Prove security
- Multi-architecture support
  - x86, ARM, RISC-V

- Advanced reverse
- Vulnerability analysis
- Binary-level security proofs
- Low-level mixt code (C + asm)
- ...
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;
    } else
        success;
}
```
• Prologue: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusion
ABOUT FORMAL METHODS AND CODE ANALYSIS

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

- Reason about the meaning of programs
- Key concepts: \( M \models \varphi \)
  - \( M \): semantic of the program
  - \( \varphi \): property to be checked
  - \( \models \): algorithmic check

- Typical ingredients: transition systems, automata, logic, ...
- Reason about infinite sets of behaviours

Success in (regulated) safety-critical domains
OUTLINE

• Prologue: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusion
NOW: MOVING TO BINARY-LEVEL SECURITY ANALYSIS

**Model**

```
x := a + b
x := x - 1
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
    jie label
label:
    move @100 B
```

**Executable**

```
ABFF780BD70696CA1010018DF45
145634789234ABFFE678ABDCF456
5A2846CDD09F55F9D6835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFYADACAD3
00113456735FFD451E13AB08000AD
344252FAADBD457345FD780001
FF22546ADDABE98977660000000
```
NOW: MOVING TO BINARY-LEVEL SECURITY ANALYSIS

Model

\[ x = x + y \]

Source code

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

Assembly

```assembly
start:  load A 100
add B A
jmp B 0
label:  move @100 B
```

Executable

```
4977B032706B080280310E8B34
5A894885E085B0283085B0285
5B99485E085B0283085B0285
5C89465E085B0283085B0285
5D89465E085B0283085B0285
5E89465E085B0283085B0285
```

• Binary code
• Attacker
• Properties
OUTLINE

• Prologue: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges
  • Going down to binary
  • Adversarial setting
  • « True security » properties

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusion
CHALLENGE: BINARY CODE LACKS STRUCTURE

• Instructions?
• Control flow?
• Memory structure?
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 51 83
EC 10 89 CB 83 EC 0C 6A 0A E8 A7 FE 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 8B 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 90 83 EC 08 FF
75 F4 68 09 85 04 08 E8 29 FE FF FF 83 C4 10 88
45 F4 8B 65 F8 59 55 5D 8D 51 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 06 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 00 00 00 00 EB 44 24
38 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
FF 94 BB 08 FF FF FF FF FF 83 C7 01 39 F7 75 DF 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 F3 C3 FF FF 53 83 EC 08 E8 13 FE
FF FF 81 C3 17 12 00 08 83 5C 05 8B C3 03 00 00
00 01 00 02 00 76 61 6C 3A 25 64 0A 00 AB 84 04
08 84 84 04 08 BD 84 04 08 C6 84 04 08 CF 84 04
08 01 1B 03 3B 28 00 00 00 00 04 00 00 00 54 FF

.rodata

.eh_frame_hdr

- code
- dead bytes
- global csts
- strings
- pointers
- other

Code (Functions)

main

unknown

__libc_csu_init

unknown

__libc_csu_fini

__term_proc

 fp_hw_IO_stdin_used

switch jump table

Assembly

[...]
DISASSEMBLY IS ALREADY TRICKY!

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

• Recovering the CFG is already a challenge!
BINARY-LEVEL ANALYSIS

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

Machine codes are complex
BINARY-LEVEL ANALYSIS

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

Machine codes are complex

At the edge of current methods

Break an implicit assumption in code analysis

- Solved problem
- IR

Sébastien Bardin - Séminaire SoSySec, 2021 | 19
**BINARY CODE SEMANTIC LACKS STRUCTURE**

### Problems
- Jump eax
- Untyped memory
- Bit-level reasoning

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

```plaintext
GF := ((ax[31,31] ≠ bx[31,31]) &
(ax[31,31] + (ax-bx)[31,31]));
SF := (ax-bx) < 0;
ZP := (ax-bx) = 0;
if (~ZF) & (GF == SF) goto 11
X := 1
goto 12
11: X := -1
12:
```
BINARY CODE SEMANTIC LACKS STRUCTURE

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

\[
\begin{align*}
\text{OF} & := ((ax \{31,31\} \# bx \{31,31\}) \& \\
& \quad (ax \{31,31\} \# (ax-bx) \{31,31\}));
\text{SF} & := (ax-bx) < 0; \\
\text{ZF} & := (ax-bx) = 0; \\
\text{if } & (\neg \text{ZF} \land (\text{OF} = \text{SF})) \text{ goto l1} \\
\text{X} & := 1 \\
\text{goto l2} \\
\text{l1:} & \text{ X := -1} \\
\text{l2:} &
\end{align*}
\]
• Context: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges
  • Going down to binary
  • Adversarial setting
  • « True security » properties

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusion
Nature is not nice

Attacker is evil

Butterfly

Attacker

Level 1: prevention of abnormal operation
Level 2: control of abnormal operation
Level 3: control of accidents
Level 4: prevention of accident progression
Level 5: consequence mitigation

Network
Firewall
Network translation
Workstation firewall
Application integrity
Kernel controls
Hypervisor separation
Hardware watchdog

Image by Florent Kirchner
ATTACKER in Standard Program Analysis

• We are reasoning worst case: seems very powerful!
ATTACKER in Standard Program Analysis

• We are reasoning worst case: seems very powerful!

• Still, our current attacker plays the rules: respects the program interface
  • Can craft very smart input, but only through expected input sources
ATTACKER in Standard Program Analysis

• We are reasoning worst case: seems very powerful!

• Still, our attacker plays the rules: respects the program interface
  • Can craft very smart input, but only through expected input sources

• What about someone who do not play the rules?
  • Side channel attacks
  • Micro-architectural attacks
ADVERSARIAL BINARY CODE

- 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)

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

• Context: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges
  • Going down to binary
  • Adversarial setting
  • True security properties

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusion
EXAMPLE: TIMING ATTACKS

Information leakage

Properties over pairs of executions
EXAMPLE: TIMING ATTACKS

Information leakage

Properties over pairs of executions

• Hyperproperties
• Quantitative
OUTLINE

• Context: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusion
BINSEC: brings formal methods to binary-level security analysis

- Advanced reverse
- Vulnerability analysis
- Binary-level security proofs
- Low-level mixt code (C + asm)

- Explore many input at once
- Find bugs
- Prove security
- Multi-architecture support
  - x86, ARM, RISC-V
Key 1: INTERMEDIATE REPRESENTATION [CAV’11]

Binsec intermediate representation

\[
\begin{align*}
\text{inst} & := \text{lv} \leftarrow e \mid \text{goto e} \mid \text{if e then goto e} \\
\text{lv} & := \text{var} \mid \@[e]_n \\
\text{e} & := \text{cst} \mid \text{lv} \mid \text{unop e} \mid \text{binop e e} \mid e \oplus e \\
\text{unop} & := \neg \mid \neg \mid \text{uext}_n \mid \text{sext}_n \mid \text{extract}_{i,j} \\
\text{binop} & := \text{arith} \mid \text{bitwise} \mid \text{cmp} \mid \text{concat} \\
\text{arith} & := + \mid - \mid \times \mid \text{udiv} \mid \text{urem} \mid \text{sdiv} \mid \text{srem} \\
\text{bitwise} & := \land \mid \lor \mid \oplus \mid \text{shl} \mid \text{shr} \mid \text{sar} \\
\text{cmp} & := = \mid \neq \mid >_u \mid <_u \mid >_s \mid <_s
\end{align*}
\]

Multi-architecture

x86-32bit – ARMv7

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

- \( \text{lhs} := \text{rhs} \)
- \( \text{goto addr}, \text{goto expr} \)
- \( \text{ite} \text{(cond)? goto addr} \)
**INTERMEDIATE REPRESENTATION**

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

![Intermediate Representation Diagram](image)

```
81 c3 57 1d 00 00 \rightarrow ADD EBX 1d57
```

```
(0x29e,0) tmp := EBX + 7511;
(0x29e,1) OF := (EBX\{31,31\} = 7511\{31,31\}) \&\& (EBX\{31,31\} \Leftrightarrow tmp \{31,31\});
(0x29e,2) SF := tmp\{31,31\};
(0x29e,3) ZF := (tmp = 0);
(0x29e,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)
```
Given a path of a program
• Compute its « path predicate » \( f \)
• Solution of \( f \) \iff \text{input following the path}
• Solve it with powerful existing solvers
### PATH PREDICATE COMPUTATION & SOLVING

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y + 1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) (branche True)</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) (branche False)</td>
</tr>
</tbody>
</table>

SMT Solver

Blackbox solvers

my input!!

Y0 = 0 ∧ Z0 = 3

\[
\text{let } W_1 \triangleq Y_0 + 1 \text{ in} \\
\text{let } X_2 \triangleq W_1 + 3 \text{ in} \\
X_2 < 2 \times Z_0 ∧ X_2 \geq Z_0
\]
ALSO: STATIC SEMANTIC ANALYSIS
(harder, doable on some classes of programs)

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

Complete verification

Reason about all paths
• Prove things

Generalize constant propagation

Complete verification
OUTLINE

• Prologue: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusion
Case 1: Vulnerability finding with symbolic execution (Godefroid et al., Cadar et al., Sen et al.)

- Intensive path exploration
- Target critical bugs

Find a needle in the heap!

Challenge = path explosion
Case 1: Vulnerability finding with symbolic execution (Heelan, Brumley et al.)

- Intensive path exploration
- Target critical bugs
- Directly create simple exploits

Challenge = path explosion

Find a needle in the heap!
Case 1: What about hard-to-find bugs [SSPREW’16](with Josselin Feist et al.)

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

Find a needle in the heap!
Case 1: What about hard-to-find bugs [SSPREW’16](with Josselin Feist et al.)

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

Guide DSE with an unsound static analysis
CASE 2: reverse & deobfuscation

- Prove something infeasible
- SE cannot help here

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

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

- The predicate is always true
- The two blocks are equivalent
- All jump targets are found

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

- Prove things
- Local $\xrightarrow{\text{scalable}}$
True Negative: k too small
• Missed proofs
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
Case 2: THE XTUNNEL MALWARE
-- [BlackHat EU 2016, S&P 2017] (Robin David)

Two heavily obfuscated samples
• Many opaque predicates

Goal: detect & remove protections
• Identify 40% of code as spurious
• Fully automatic, < 3h  [now: 20min]

Backward-bounded SE
• + dynamic analysis

<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>
Case 3: SECURING CRYPTO-PRIMITIVES
-- [S&P 2020, NDSS 2021] (Lesly-Ann Daniel)

- Property: timing attacks
- Attacker: speculation
Case 3: SECURING CRYPTO-PRIMITIVES
-- [S&P 2020, NDSS 2021] (Lesly-Ann Daniel)

- Relational symbolic execution
- Follows pairs of execution
- Check for divergence
- Sharing, merging, preprocess

- 397 crypto code samples, x86 and ARM
- New proofs, 3 new bugs (of verified codes)
- Potential issues in some protection schemes
- 600x faster than prior work
SMT solvers are powerful weapons
But (binary-level) security problems are terrific beasts

Finely tuning the technology can make a huge difference

Some queries: 24h → 1min
600x faster than prior approach
Tuning the solver: intensive array formulas
[LPAR 2018] (Benjamin Farinier)

- Makes the difference!

- Dedicated data structure (list-map)
- Tuned for base+offset access
- Linear complexity
Array theory
- Necessary to model memory
- Hard for solvers (case splits)

Goal: make it tractable

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

Hard to solve
- NP-complete
- ROW may require case-splits

- Reading in an element at index $i \in \mathcal{I}$: select $a_i$
- Writing in an element $e \in \mathcal{E}$ at index $i \in \mathcal{I}$: store $a_i e$

\[
\begin{align*}
\text{select} : \text{Array} \mathcal{I} \mathcal{E} &\rightarrow \mathcal{I} \rightarrow \mathcal{E} \\
\text{store} : \text{Array} \mathcal{I} \mathcal{E} &\rightarrow \mathcal{I} \rightarrow \mathcal{E} \rightarrow \text{Array} \mathcal{I} \mathcal{E}
\end{align*}
\]

\[
\forall a i e. \text{select} (\text{store} a i e) i = e
\]

\[
\forall a i j e. (i \neq j) \Rightarrow \text{select} (\text{store} a i e) j = \text{select} a j
\]
Not pure theory!

Reverse of a ASPACK-protected code

Huge formula obtained by dynamic symbolic execution
293,000 select
24 hours of resolution!
Not pure theory!

Reverse of a ASPACK-protected code

Remember: binary-level
- Very long chains of write
- A single memory, no partition information

Sad state-of-the-art:
- concretize memory accesses (scale, no genericity)
- Keep symbolic (generic but no scale at all)
Inner-working of array theory

- Reading in $a$ at index $i \in I$: select $a_i$
- Writing in $a$ an element $e \in E$ at index $i \in I$: store $a_{i+e}$

\[
\text{select} : \text{Array}(I, E) \rightarrow I \rightarrow E \\
\text{store} : \text{Array}(I, E) \rightarrow I \rightarrow E \rightarrow \text{Array}(I, E)
\]

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

« Logical arrays » as chains of store
(« list representation »)

ROW reasoning may introduce case-splits

Eliminate ROW
Inner-working of array theory

- Reading in `a` at index `i ∈ I`: select `a i`
- Writing in `a` an element `e ∈ E` at index `i ∈ I`: store `a i e`

```
select : Array I E → I → E
store : Array I E → I → E → Array I E

∀ a i e. select (store a i e) i = e
∀ a i j e. (i ≠ j) ⇒ select (store a i e) j = select a j
```

« Logical arrays » as chains of store
(« list representation »)

ROW reasoning may introduce case-splits

Eliminate ROW

ROW rules could be used as a preprocessing??
Inner-working of array theory

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

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

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

- Constant case: too slow
- Symbolic case: no simplif.

« Logical arrays » as chains of store
(« list representation »)

- ROW reasoning may introduce case splits
- Eliminate ROW

- ROW rules could be used as a preprocessing??

- Quadratic reasoning
- Term-based equality
- Disequality??
Inner-working of array theory

- **Reading in a at index \( i \in I \): select \( a_i \)**
- **Writing in \( a \) an element \( e \):**
  
  \[
  \begin{align*}
  & \text{select} : \text{Array} I \times I \\
  & \text{store} : \text{Array} I \\
  & \forall i, e. \text{select} (\text{store} a i e) j = e \\
  & \forall i, j e. (i \neq j) \Rightarrow \text{select} (\text{store} a i e) j = \text{select} a j
  \end{align*}
  \]

**Goal:** efficient preprocessing to remove ROW
- Completely address the constant case
- Help for the symbolic case
- Go beyond ROW (eg, WOW)

**Questions:**
- **ROW rules could be used as a preprocessing??**
- **Quadratic reasoning**
- **Term-based equality**
- **Disequality??**

« Logical arrays » as chains of store
(« list representation »)
Fast array simplification (1)

- Dedicated data structure (list-map)
- Tuned for \texttt{base+offset} access
- \texttt{Base} can be symbolic
- \( n \times \ln(n) \) complexity

\begin{itemize}
    \item Scale
    \item Good when only few bases
    \item Perfect for constant case
\end{itemize}

Still limited by term-equality reasoning
Fast array simplification (2)

- Dedicated data structure (list-map)
- Tuned for base+offset access
- Base can be symbolic
- $n \times \ln(n)$ 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
- Perfect for symb. stack over simple functions

Still limited by disequality reasoning

- Scale
- Good when only few bases
Fast Array Simplification

- Dedicated data structure (list-map)
- Tuned for base+offset access
- Base can be symbolic
- $n \times \ln(n)$ complexity

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

- Reduce the number of bases
- Perfect for symb. stack over simple functions

- 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
IT WORKS!

- Excellent for DSE-like formulas
- Slight overall improvement over SMTCOMP

**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

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

No Crash, No Exploit: Automated Verification of Embedded Kernels

Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival

Abstract—The kernel is the most safety- and security-critical component of many computer systems, and as most serious bugs lead to complete system crash or exploit. It is thus desirable to guarantee that a kernel is free from these bugs using formal methods, but the high cost and expert knowledge required to do so are deterrent to wide applicability. We propose a method that can verify both absence of runtime errors (i.e., crashes) and absence of privilege escalation (i.e., exploits) in embedded kernels from their binary executables. The method can verify the kernel runtime system developers only provide their code and, with very little configuration or none at all, the tool automatically verifies the properties of interest. In addition, a comprehensive verification should carry to the binary executable, as 1. a large part of embedded kernel code consists in low-level interaction with the hardware, and 2. the compilation toolchain (build options, compiler, assembler, linker) may introduce bugs [12]. Recent so-called “push-button” kernel verification methods [13-14] are based on symbolic execution [15-16] which

- Full verification of embedded kernels
- RTAS 2021 (best paper award)

Not All Bugs Are Created Equal, But Robust Reachability Can Tell The Difference

Guillaume Girol, Benjamin Farinier, and Sébastien Bardin

Abstract. This paper introduces a new property called robust reachability which refines the standard notion of reachability in order to take replicability into account. A bug is robustly reachable if a controlled input can make it so the bug is reached whatever the value of uncontrolled input. Robust reachability is better suited than standard reachability in many realistic situations related to security (e.g., criticality assessment or bug prioritization) or software engineering (e.g., replicable test suites and

- Focus on robust bugs
- CAV 2021
Example 2: robust symbolic execution [CAV 2018, CAV 2021]

- 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: $(a = -100, b = 10, x = 1)$

In practice: canaries, secret key in uninitialized memory, etc.
Example 2: robust symbolic execution

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

- Actually, need to solve $\forall x. ax + b > 0$
  
  - How to solve it? (CAV18)
  
  - Robust reachability (CAV’21)
Our solution: reduce quantified formula to the quantifier-free case

- Approximation
- But reuse the whole SMT machinery

Key insights:
- independence conditions
- formula strengthening
OUTLINE

• Context: a little bit of formal methods for safety

• Binary-level security analysis: benefits & challenges

• The BINSEC platform

• From source-level safety to binary-level security: some examples

• Conclusions
SOME KEY PRINCIPLES BEHIND OUR WORK?

• 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 (backward, relational, robust)
  • 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
Conclusion

• **Security is not safety**, and it’s great fun for FM/PL researchers
  • Binary level, attacker model, true security properties

• **Need to revisit (deeply?) standard methods**
  • Two different stories: Symbolic Execution vs. Static Analysis
  • Variants, combinations

• **Need a real « security-oriented » code analysis framework**

• **Some results in that direction, still many exciting challenges**

BINSEC is available (new release)
https://binsec.github.io

ANR Project TAVA

Looking for PhD & postdoc
sebastien.bardin@cea.fr