Revisiting Program Analysis through the Security Lens

Journées Nationales du GDR GPL / AFADL 2023

FROM RESEARCH TO INDUSTRY

Sébastien Bardin
Senior Researcher, CEA Fellow
LSL/SABR
The BINSEC Group:
ADAPT FORMAL METHODS TO BINARY-LEVEL SECURITY ANALYSIS

https://binsec.github.io/

Looking for postdoc & PhD candidates
Program Analysis (PL) and Formal Methods come from critical safety needs
- Damn good there (in the hands of experts)

Now: a move from safety concerns to security concerns

Questions:
- how can we use standard PL/FM into a security context?
- how does code-level security differ from code-level safety?
- how does security differ from safety? [focus on the attacker]

This talk: share some insights from our biased experience [CAV 21, ESOP 2023]
TEAM WORK SINCE 2012

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

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

Success in (regulated) safety-critical domains
They knew it was impossible, so they did it anyway

Cannot have analysis that
• Terminates
• Is perfectly precise
On all programs

Answers
• Forget perfect precision: bugs xor proofs
• Or focus only on « interesting » programs
• Or put a human in the loop
• Or forget termination

• Weakest precondition calculi [1969, Hoare]
• Abstract Interpretation [1977, Cousot & Cousot]
• Model checking [1981, Clarke - Sifakis]
They knew it was impossible, so they did it anyway

• Weakest precondition calculi [1969, Hoare]
• Abstract Interpretation [1977, Cousot & Cousot]
• Model checking [1981, Clarke - Sifakis]

Answers
• Forget perfect precision: bugs xor proofs
• Or focus only on « interesting » programs
• Or put a human in the loop
• Or forget termination

Cannot have analysis that
• Terminates
• Is perfectly precise
On all programs
Given a path of a program
- Compute its « path predicate » f
- Solution of f = input following the path
- Solve it with powerful existing solvers
BACK TO BASICS

SOURCE CODE

ASSEMBLY CODE

OBJECT CODE

EXECUTABLE

COMPILE

ASSEMBLE

LINK

RUN

01001100
00101011
11000101
010 ..

010100111
101101110
111011000
0100 ..

010100111
101101110
111011000
0100 ..

THIRD PARTY
LIBRARY

HAND WRITTEN
ASSEMBLY

INLINE
ASSEMBLY

ASSEMBLY CODE

OBJECT CODE

EXECUTABLE

SOURCE CODE

inline
assembly
WHY GOING DOWN TO BINARY-LEVEL SECURITY ANALYSIS?

- 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
• Introduction

• Challenges of automated binary-level security analysis
• BINSEC & Symbolic Execution for Binary-level Security

• Robust reachability and bugs that matter
• Adversarial reachability

• Conclusion, Take away and Disgression
OUTLINE

• Introduction

• Challenges of automated binary-level security analysis
  • BINSEC & Symbolic Execution for Binary-level Security

• Robust reachability and bugs that matter
  • Adversarial reachability

• Conclusion, Take away and Disgression
New challenges!

- Binary code
- Attacker
- Properties
New challenges!

- Binary code
- Attacker
- Properties

Sébastien Bardin
CHALLENGE: BINARY CODE LACKS STRUCTURE

- Instructions?
- Control flow?
- Memory structure?
DISASSEMBLY IS ALREADY TRICKY!

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

Sébastien Bardin
BINARY CODE SEMANTIC LACKS STRUCTURE

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

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:
New challenges!

- **Binary code**
- **Attacker**
- **Properties**
New challenge: safety is not hyper-property :-)
New challenge: safety is not hyper-property :-)

Information leakage

Properties over pairs of executions

- New problems
- Hyperproperties
- Quantitative
- Identify « bugs that matters »
New challenges!

Main topic of the day:

- Binary code
- Attacker
- Properties

Sébastien Bardin
CHALLENGE: ATTACKER

Nature is not nice

Attacker is evil

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

Sébastien Bardin
• **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
• 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 really do not play the rules?
  • Side channel attacks
  • Micro-architectural attacks
  • Fault injections
Another Line of attack: 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)

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>

<table>
<thead>
<tr>
<th>address</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>80483d1</td>
<td>call 45</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>
OUTLINE

• Introduction

• Challenges of automated binary-level security analysis
  • BINSEC & Symbolic Execution for Binary-level Security

• Robust reachability and bugs that matter
• Adversarial reachability

• Conclusion, Take away and Disgression
**BINSEC: brings formal methods to binary-level security analysis**

**Explore many input at once**
- Find bugs
- Prove security

**Multi-architecture support**
- x86, ARM, RISC-V
- 32bit, 64bit

**Advanced reverse**
- Vulnerability analysis

**Binary-level security proofs**
- Low-level mixt code (C + asm)

**Multi-architecture support**
- x86, ARM, RISC-V
- 32bit, 64bit

**Static analysis**

**Symbolic execution**

**x86**

- ABFFFF780D76696CA101001BDE45
  - 4ABFFD34F923ABFEDF7EF6BDBC7F846
  - 5A2A44C02D35150305346031515697
  - 345FEDICACACDMD979F94190K1
  - 245FA04AD3E3AD003F33D0D3AD25
  - 300134567337FOFG313AB80R3AD0
  - 34425279N0R0445734F0D78B0K01
  - FFF22546AD5CA989776809800

**ARM**

- ABFFFF780D76696CA101001BDE45
  - 45156F79D3ABFEDF7EF6BDBC7F846
  - 5A2A44C02D35150305346031515697
  - 345FEDICACACDMD979F94190K1
  - 245FA04AD3E3AD003F33D0D3AD25
  - 300134567337FOFG313AB80R3AD0
  - 34425279N0R0445734F0D78B0K01
  - FFF22546AD5CA989776809800

**https://binsec.github.io/**
**BINSEC: brings formal methods to binary-level security analysis**

- **Break**
- **Prove**
- **Protect**

**x86**
- ABFF78B7D960AC1012661BE45
  - 435634789234ABFF7E80BDD769
  - 5A2345678901234567890123456
- ABFF78B7D960AC1012661BE45
  - 435634789234ABFF7E80BDD769
  - 5A2345678901234567890123456

**ARM**
- ABFF78B7D960AC1012661BE45
  - 435634789234ABFF7E80BDD769
  - 5A2345678901234567890123456
- ABFF78B7D960AC1012661BE45
  - 435634789234ABFF7E80BDD769
  - 5A2345678901234567890123456

**Explore many input at once**
- Find bugs
- Prove security
- Multi-architecture support
  - x86, ARM, RISC-V
  - 32bit, 64bit

**Static analysis**

**Symbolic execution**

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

**https://binsec.github.io/**
Key 1: INTERMEDIATE REPRESENTATION [CAV’11]

Binsec intermediate representation

```
inst := lv ← e | goto e | if e then goto e
lv := var | @e

e := cst | lv | unop e | binop e e | e ? e : e

unop := ¬ | ¬ | uext_n | sext_n | extract_i,j
binop := arith | bitwise | cmp | concat
arith := + | - | × | udiv | urem | sdiv | srem
bitwise := ∧ | ∨ | Θ | shl | shr | sar
cmp := = | ≠ | >u | <u | >s | <s
```

Multi-architecture

x86-32bit – ARMv7

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

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

- Concise
- Well-defined
- Clear, side-effect free
Given a path of a program
• Compute its « path predicate » f
• Solution of f = 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**

let $W_1 \triangleq Y_0 + 1$ in 
let $X_2 \triangleq W_1 + 3$ in 
$X_2 < 2 \times Z_0 \land X_2 \geq Z_0$

**Blackbox solvers**

- Boolector
- Y0 = 0 \land Z0 = 3

**my input!!**

Sébastien Bardin
PATH PREDICATE COMPUTATION & SOLVING

Key ingredients
- Path search
- Constraint solving

Many optimizations
- Preprocessing, caching, etc.
- Search heuristics, path pruning, merge, etc.
- Concretization

Beware
- Path explosion
- Constraint solving cost

Blackbox solvers

SMT Solver

Sébastien Bardin

```
let W₁ ≝ Y₀ + 1 in
let X₂ ≝ W₁ + 3 in
X₂ < 2 × Z₀ ∧ X₂ ≥ Z₀
```

```
Y₀ = 0 ∧ Z₀=3
```
Typical application: Vulnerability finding & automated testing

- Intensive path exploration
- Target critical bugs
- Or high coverage
- From scratch
- Or enhanced prior test suite

Symbolic execution – fuzzing – static analysis
OUTLINE

• Introduction

• Challenges of automated binary-level security analysis
• BINSEC & Symbolic Execution for Binary-level Security

• Robust reachability and bugs that matter
• Adversarial reachability

• Conclusion, Take away and Disgression
• Problem: not all bugs are equal
The problem of « false positive in practice »

- Reachability-based reasoning may produce false positive in practice

```c
int main () {
    int a = input ();
    int b = input ();
    int x = rand ();

    if (a * x + b > 0) {
        analyze_me();
    } else {
        ...
    }
}
```
The problem of « false positive in practice »

- Reachability-based reasoning may produce false positive in practice

Sébastien Bardin
Reachability-based reasoning may produce false positive in practice:

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

What?!! Safety is not security …
The problem of « false positive in practice »

• **Reachability-based reasoning** may produce false positive in practice

  - 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.
Problems with standard reachability?

Mitigation: stack canaries

- Value in blue is checked against canary
- Canary is a parameter

In practice, only $2^{-32}$ to bypass canary
- Not considered an attack

Still, Symbolic Execution reports a bug
- just need canary ==rrrr
- False positive
Problems with standard reachability? (2)

• Randomization-based protections
  • Guess the randomness

• Bugs involving uninitialized memory
  • Guess memory content

• Undefined behaviours
  • Exist also in hardware

• Stubbing functions (I/O, opaque, crypto, …)
  • Guess the hash result …

• Underspecified initial state
Our proposal [CAV 2018, CAV 2021, FMSD 2022]

Choose a threat Model
Partition input into controlled input \( a \) and uncontrolled input \( \overline{x} \)

\((a, \overline{x}) \vdash \ell\) means “with inputs \( a \) and \( \overline{x} \), the program executes code at \( \ell \)”

Reachability of location \( \ell \)

\[ \exists a, \overline{x}. (a, \overline{x}) \vdash \ell \]

Robust Reachability of \( \ell \)

\[ \exists a. \forall \overline{x}. (a, \overline{x}) \vdash \ell \]

Guaranteed
Adapting BMC and SE

Path merging
- Optional in SE
- Required for completeness in Robust SE

...and a few other differences
- Assume $\psi \land \phi$ instead of $\exists a . \forall x . (\psi \implies \phi)$
- Path pruning: no extra quantifier
- Concretization: only works on controlled values

\[
\exists a . \forall x . \phi \xrightarrow{\text{concretize}} \exists a . \forall x . x = 90 \land \phi
\]
Proof-of-concept implementation

- A binary-level Robust SE and Robust BMC engine based on BINSEC
- Discharges quantified SMT(arrays+bitvectors) formulas to Z3
- Evaluated against 46 reachability problems including CVE replays and CTFs

<table>
<thead>
<tr>
<th></th>
<th>BMC</th>
<th>SE</th>
<th>RBMC</th>
<th>RSE</th>
<th>RSE+path merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>22</td>
<td>30</td>
<td>32</td>
<td>37</td>
<td>44</td>
</tr>
<tr>
<td>False positive</td>
<td>14</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inconclusive</td>
<td>10</td>
<td></td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Resource exhaust</td>
<td>10</td>
<td>13</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Robust variants of SE and BMC

No false positives, more time-outs/memory-outs, 15% median slowdown
Case-studies: 4 CVE

CVE-2019-14192 in U-boot (remote DoS: unbounded memcpy) Robustly reachable
CVE-2019-19307 in Mongoose (remote DoS: infinite loop) Robustly reachable
CVE-2019-20839 in libvncserver (local exploit: stack buffer overflow)
  Without stack canaries: Robustly reachable
  With stack canaries: Timeout
CVE-2019-19307 in Doas (local privilege escalation: use of uninitialized memory)
  Doas = OpenBSD’s equivalent of sudo
  Depends on the configuration file /etc/doas.conf
  Use robust reachability in a more creative way
Reinterpret “controlled input” differently:

the **attacker** controls nothing, only executes
the **sysadmin** controls the configuration file: **controlled input**
the **environment** sets initial memory content etc: **uncontrolled inputs**

The meaning of robust reachability here

Are there configuration files which make the attacker win all the time?
**Yes:** for example typo “permit ww” instead of “permit www”
Alternative formalism: non-interference

<table>
<thead>
<tr>
<th>Non Interference</th>
<th>for all $a$</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robust reachability</td>
<td>for a single $a$</td>
<td>yes</td>
</tr>
</tbody>
</table>

Non-interference + Reachability $\not\Rightarrow$ Robust Reachability
As a hyperproperty, robust reachability is pure hyperliveness
  • not a trace property (most studied case)
  • not \((k\text{-})\)hypersafety \(\Rightarrow\) not solvable with self-composition

Temporal logics: Expressible in CTL, HyperLTL, but no provers for generic programming languages

Need a dedicated proof method!
Stepping back

- Robust reachability draws a line between some good bugs and bad bugs
  - Based on replicability

- Several formalisms can express robust reachability
  - [games, ATL, hyperLTL, CTL]
  - Yet no efficient software-level checkers

- A few prior attempts, on different dimensions
  - Quantitative or probabilistic approaches (model checking, non interference)
  - Automated Exploit Generation (Avgerinos et al., 2014)
  - Test Flakiness (O'Hearn, 2019) [a specific case of robust reachability]
  - Fair model checking (Hart et al., 1983)

- Qualitative « all or nothing » robust reachability may be too strong
  - Mitigation: add user-defined constraints over the uncontrolled variables
  - WIP: quantitative definitions, inference of robustness conditions
Potential applications

• **Better testing / bug finding tools**
  • Ex: find replicable bugs
  • Ex: generate non-flaky tests

• **Test suite evaluation**
  • Are the test case replicable?

• **Bug prioritisation**
  • Replicable bugs first
Idea: reduce quantified formula to the quantifier-free case

- Approximation
- But reuse the whole SMT machinery

Key insights:
- independence conditions
- formula strengthening
OUTLINE

• Introduction

• Challenges of automated binary-level security analysis
• BINSEC & Symbolic Execution for Binary-level Security

• Robust reachability and bugs that matter
• Adversarial reachability

• Conclusion, Take away and Disgression
Problem: what about the attacker capabilities?
Context

- Many techniques and tools for security evaluations.
- Usually consider a weak attacker, able to craft smart inputs.
- Real-world attackers are more powerful: various attack vectors + multiple actions in one attack.

### Hardware attacks
- Electromagnetic pulses
- Power glitch
- Clock glitch
- Laser beam
- Faultline
- DVFS

### Software-implemented hardware attacks
- Race condition
- Load Value Injection
- Spectre
- Rowhammer

### Micro-architectural attacks

### Man-At-The-End attacks

Sébastien Bardin
Context

- Many techniques and tools for security evaluations.
- Usually consider a weak attacker, able to craft smart inputs.
- Real-world attackers are more powerful: various attack vectors + multiple actions in one attack.

**Hardware attacks**
- Electromagnetic pulses
- Power glitch
- Clock glitch
- Laser beam
- Faultline
- DVFS
- Race condition
- Load Value Injection
- Spectre

**Software-implemented hardware attacks**
- Rowhammer

**Micro-architectural attacks**

**Man-At-The-End attacks**

Sébastien Bardin
State-of-the-Art: software-implemented fault injection

**Mutant generation:** create a new mutated program for each fault configuration.

- $k$ (faults) among $n$ (lines) mutant generated

**Forking technique:** fork the analysis with a fault at each possible fault location.

- $k$ (faults) among $n$ (lines) paths created

- Both faces scalability issues (path explosion) hindering multi-fault analysis.
- They don’t provide formalization of the underlying problem.
Contributions

- We formalize the **Adversarial Reachability** problem.
- We propose **Adversarial Symbolic Execution**, with dedicated **optimizations**.
- We propose an **implementation** and **evaluation** of our technique.
- We perform a security analysis of the **WooKey bootloader**.
Adversarial reachability

**Goal:** have a formalism extending standard reachability to reason about a program execution in presence of an advanced attacker.

**Adversarial reachability:** A location \( l \) is adversarially reachable in a program \( P \) for an attacker model \( A \) if \( S_0 \xrightarrow{\ast} l \), where \( \xrightarrow{\ast} \) is a succession of program instructions interleaved with faulty transitions.
Forking encodings

Original

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]

\[ x := y \]
Forkless encodings and FASE

- Covers all adversarial behaviors
- Only 1 path
- More complex formulas

Original:
- $x := y$

Forkless:
- $x := \text{ite } here_i \ ? \ fault_i : y$
  - $here_i \in [0,1]$, $\Sigma \, here_i \leq \max_f$
Early Detection of fault Saturation (EDS)

**FASE**

- Potentially faulted instruction (with ite)
- We need max_\tau faults to go beyond that point on that path.

- Covers all adversarial behaviors, as complete as FASE
- Only 1 path
- Reduce number of fault injections along a path

**FASE-EDS**

- SAT with a fault margin or SAT with exactly the fault budget or infeasible
- Instruction not faulted

Sébastien Bardin
Injection On Demand (IOD)

- Covers all adversarial behaviors, as complete as FASE
- Only 1 path
- Reduce number of fault injections
- Additional queries

We can’t go beyond that point on that path without more faults.
Injection On Demand (IOD)

- Covers all adversarial behaviors, as complete as FASE
- Only 1 path
- Reduce number of fault injections
- Additional queries

We can’t go beyond that point on that path without more faults.

Path predicate switched for the faulted one
Injection On Demand (IOD)

FASE

Faulted instruction

We can’t go beyond that point on that path without more faults.

- Covers all adversarial behaviors, as complete as FASE
- Only 1 path
- Reduce number of fault injections
- Additional queries

FASE-IOD

Bonus: under-approximation of $\text{nb}_f$
RQ2 - scaling without path explosion

➔ Forking explodes in explored paths while FASE doesn’t.
➔ Translates to improved analysis time overall.
Security scenarios using different fault models

**CRT-RSA:** [1]
- basic vulnerable to 1 reset → OK
- Shamir (vulnerable) and Aumuler (resistant) → TO

**Secret-keeping machine:** [2]
- Linked-list implementation vulnerable to 1 bit-flip in memory → OK
- Array implementation resistant to 1 bit-flip in memory → OK
- Array implementation vulnerable to 1 bit-flip in registers → OK

**Secswift countermeasure:** llvm-level CFI protection by STMicroelectronics [3]
- SecSwift implementation [4] applied to VerifyPIN_0 → early loop exit attack with 1 arbitrary data fault or test inversion in valid CFG

---

Case study

**WooKey bootloader**: secure data storage by ANSSI, 3.2k loc.

**Goals:**

1. Find known attacks (from source-level analysis)
   a. Boot on the old firmware instead for the newest one [1]
   b. A buffer overflow triggered by fault injection [1]
   c. An incorrectly implemented countermeasure protecting against one test inversion [2]

2. Evaluate countermeasures from [1]
   a. Evaluate original code → **We found an attack not mentioned before**
   b. Evaluate existing protection scheme [1] (**not enough**)
   c. **Propose and evaluate our own protection scheme**

---


Stepping back

- Adversarial reachability takes an active attacker into account
- Well known in cryptographic protocol verification, not for code
- Generic: reachability, hyper-reachability, non termination
- Scalability?
- Which capabilities for the attacker? [link with Hardware security community]
- Strong link with robust reachability
OUTLINE

• Introduction

• Challenges of automated binary-level security analysis
  • BINSEC & Symbolic Execution for Binary-level Security

• Robust reachability and bugs that matter
  • Adversarial reachability

• Conclusion, Take away and Disgression
TAKE AWAY: SECURITY IS NOT SAFETY

- Fun for FM/PL researchers
- Important applications

- Reachability is well suited for safety, yet security leads to many new interesting variations
- Still many things to do !!
- Symbolic Execution appears to be versatile enough
- BINSEC is open source, check it [with us]

https://binsec.github.io/

Looking for postdoc & PhD
THANK YOU