Five shades of symbolic execution for vulnerability hunting

« Cyber in Sophia »

Summer School GDR Sécurité 2023

FROM RESEARCH TO INDUSTRY

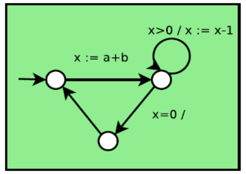
Sébastien Bardin

Senior Researcher, CEA Fellow

CEA LIST

The BINSEC Group:
ADAPT FORMAL METHODS TO BINARY-LEVEL SECURITY ANALYSIS

Model



Source code

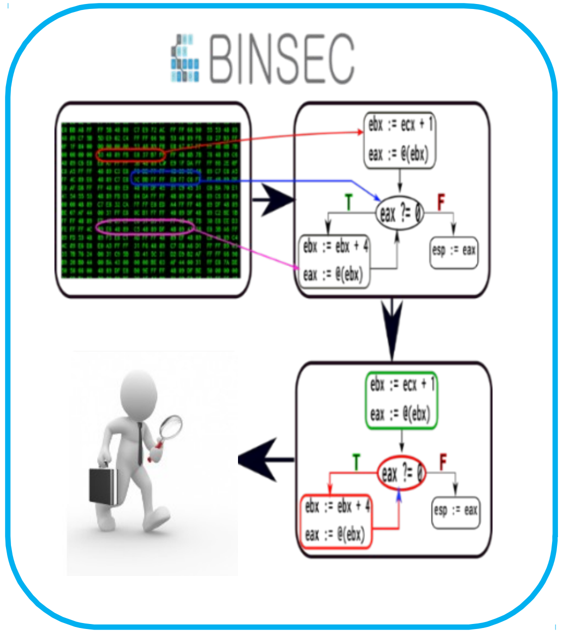
```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

Assembly

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

label:
 move @100 B
```

Executable

ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000



THEMIDA

https://binsec.github.io/

WHY THIS TALK?

- Focus on code-level security
- Implementation flaws / attacks

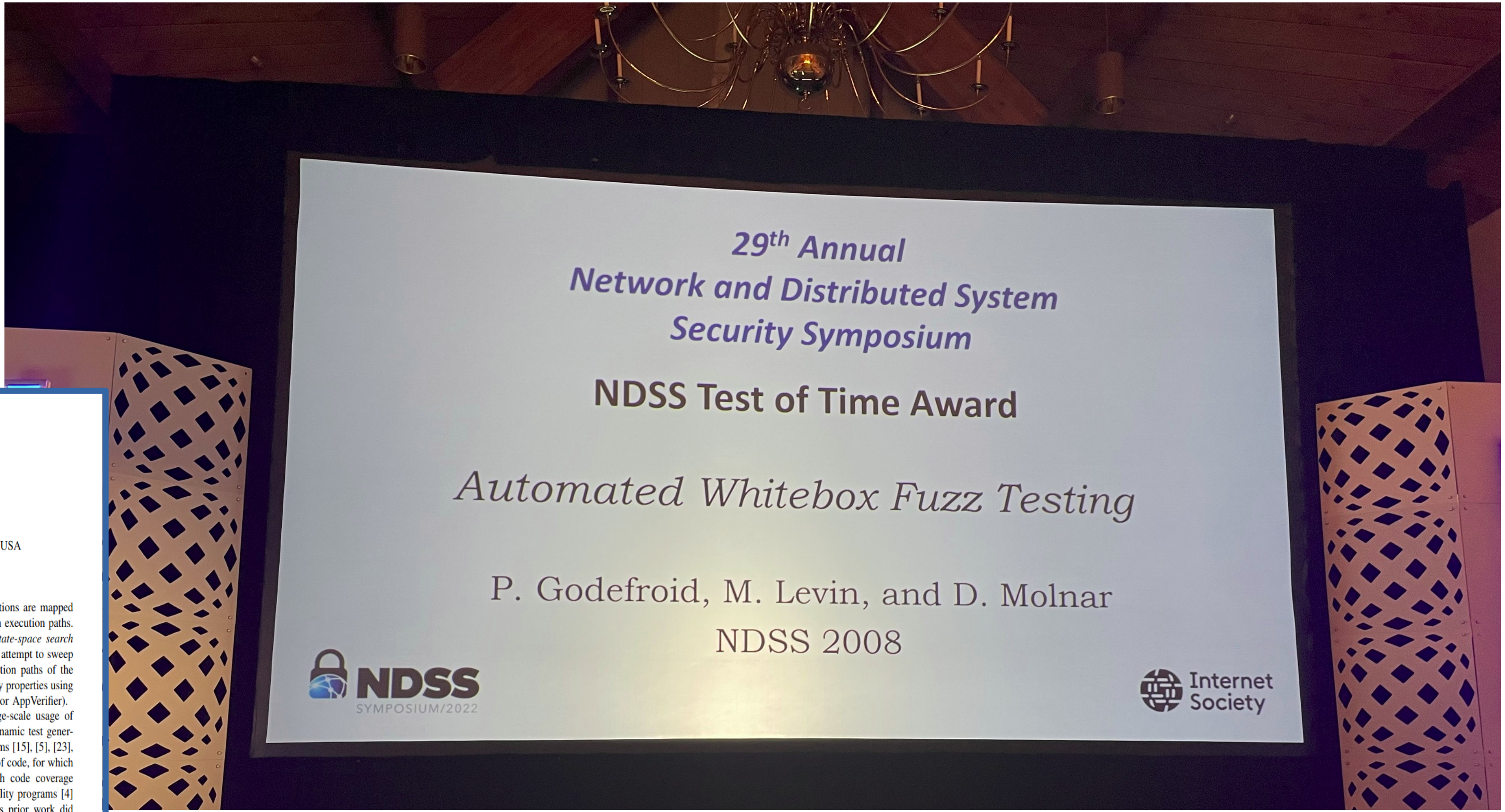
- I love Symbolic Execution : it is formal & it works :-)

- Originate from safety & testing, quickly adopted in security

- ***Questions***:
  - ***how can you use Symbolic Execution into a security context ?***
  - ***How does code-level security differ from code-level safety?***

- ***This lecture: our experience on adapting Symbolic Execution to several binary-level security contexts***

And in the end, it works !



Billions and Billions of Constraints: Whitebox Fuzz Testing in Production

Ella Bounimova, Microsoft Research, USA
Patrice Godefroid, Microsoft Research, USA
David Molnar, Microsoft Research, USA

Abstract—We report experiences with constraint-based whitebox fuzz testing in production across hundreds of large Windows applications and over 500 machine years of computation from 2007 to 2013. Whitebox fuzzing leverages symbolic execution on binary traces and constraint solving to construct new inputs to a program. These inputs execute previously uncovered paths or trigger security vulnerabilities. Whitebox fuzzing has found one-third of all file fuzzing bugs during the development of Windows 7, saving millions of dollars in potential security vulnerabilities. The technique is in use today across multiple products at Microsoft.

We describe key challenges with running whitebox fuzzing in production. We give principles for addressing these challenges and describe two new systems built from these principles: SAGAN , which collects data from every fuzzing run for further analysis, and JobCenter, which controls deployment of our whitebox fuzzing infrastructure across commodity virtual machines. Since June 2010, SAGAN has logged over 3.4 billion constraints solved, millions of symbolic executions, and tens of millions of test cases generated. Our work represents the largest scale deployment of whitebox fuzzing to date, including the largest usage ever for a Satisfiability Modulo Theories (SMT) solver. We present specific data analyses that improved our production use of whitebox fuzzing. Finally we report data on the performance of constraint solving and dynamic test generation that points toward future research problems.

solved with a *constraint solver*, whose solutions are mapped to new inputs that exercise different program execution paths. This process is repeated using systematic *state-space search techniques*, inspired by *model checking*, that attempt to sweep through as many as possible feasible execution paths of the program while checking simultaneously many properties using a *runtime checker* (such as Purify, Valgrind or AppVerifier).

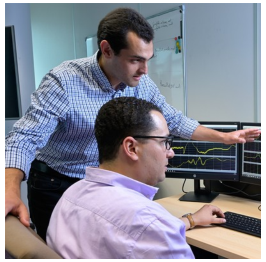
In this paper, we report on the first large-scale usage of whitebox fuzzing. Earlier applications of dynamic test generation focused on unit testing of small programs [15], [5], [23], typically consisting of a few thousand lines of code, for which these techniques were able to achieve high code coverage and find new bugs, for instance, in Unix utility programs [4] or device drivers [7]. While promising, this prior work did not report of any daily use of these techniques and tools. In contrast, we present here our experience running whitebox fuzzing on a much larger scale and in *production*.

We achieve this scale because the current *"killer app"* for dynamic test generation is *whitebox fuzzing of file parsers*. Many security vulnerabilities are due to programming errors in code for parsing files and packets that are transmitted over the internet. For instance, the Microsoft Windows operating

TEAM WORK SINCE 2012



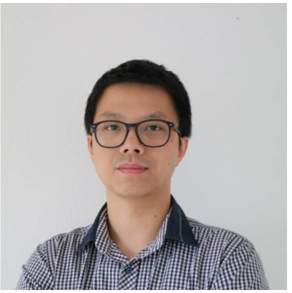






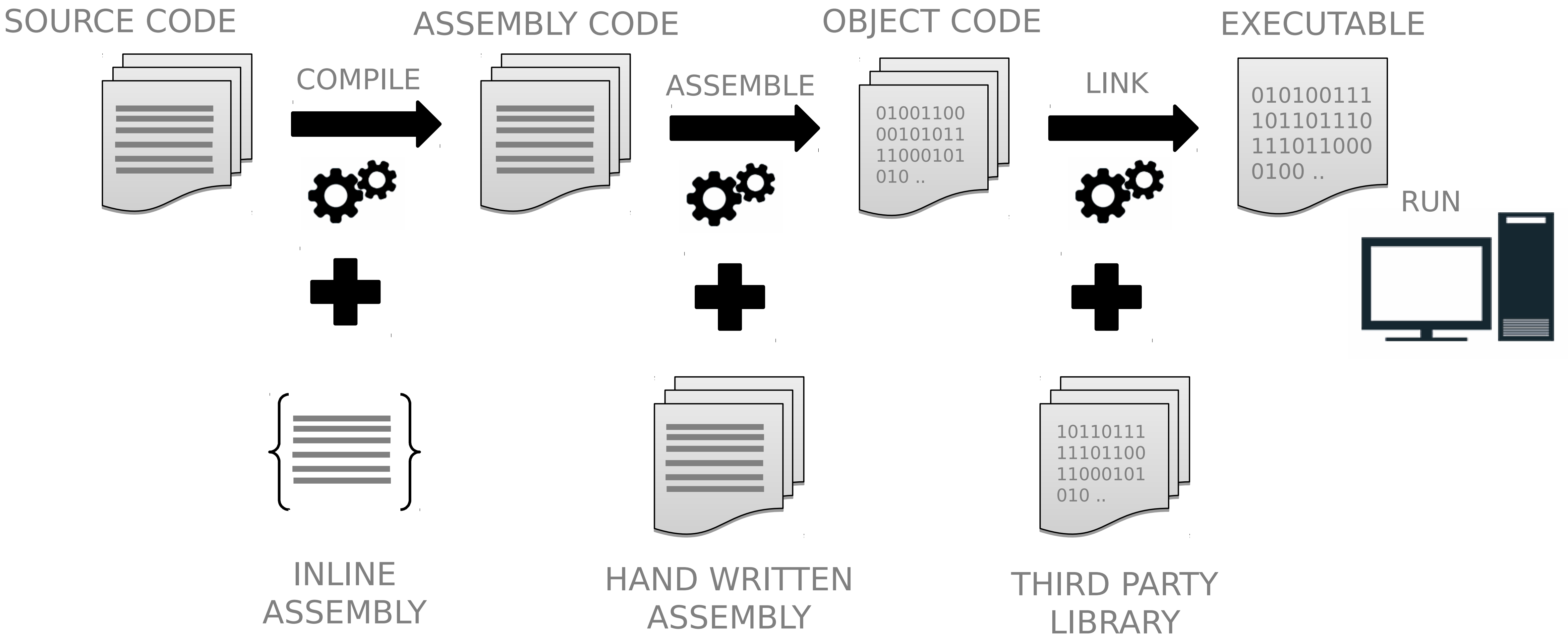








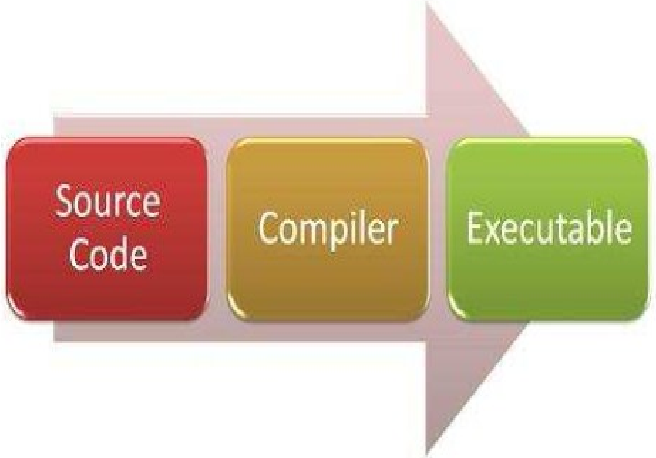

BACK TO BASICS



WHY GOING DOWN TO BINARY-LEVEL SECURITY ANALYSIS?

No source code

Post-compilation

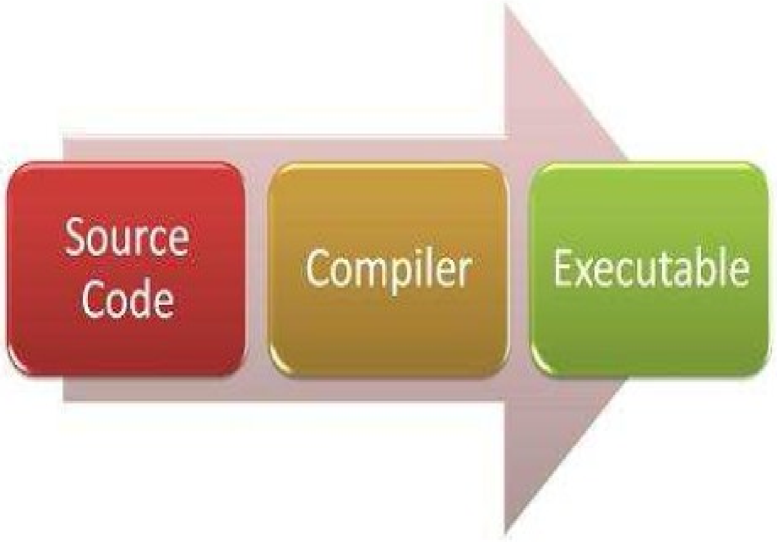


Malware comprehension

Protection evaluation

Very-low level reasoning

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

```
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**

EXAMPLE: third-party component analysis

Find a needle in the heap !

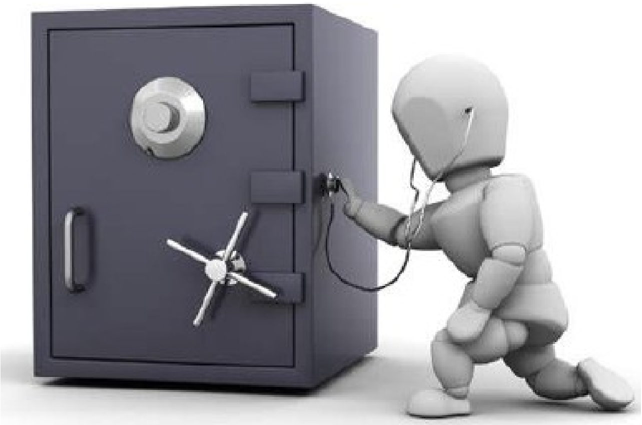
- Is it reasonably secure to use that ?

EXAMPLE: side channel attacks

```
private char[4] secret;

boolean CheckPassword (char[4] input) {

}
```

- Can you retrieve the secret with blackbox access?

EXAMPLE: side channel attacks



```
private char[4] secret;

boolean CheckPassword (char[4] input) {
 for (i=0 to 3) do
    if(input[i] != secret[i]) then
       return false;
    endif
 endfor
 return true;
}
```

- Can you retrieve the secret with blackbox access?

- Here, yes

OUTLINE

- Introduction
- What every honest person should know about Symbolic Execution
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

OUTLINE

- **Introduction**
- **What every honest person should know about Symbolic Execution**
- **Challenges of automated binary-level security analysis**
- **BINSEC & Symbolic Execution for Binary-level Security**
- **Shades of Symbolic Execution for Security**
- **Conclusion, Take away and Disgression**

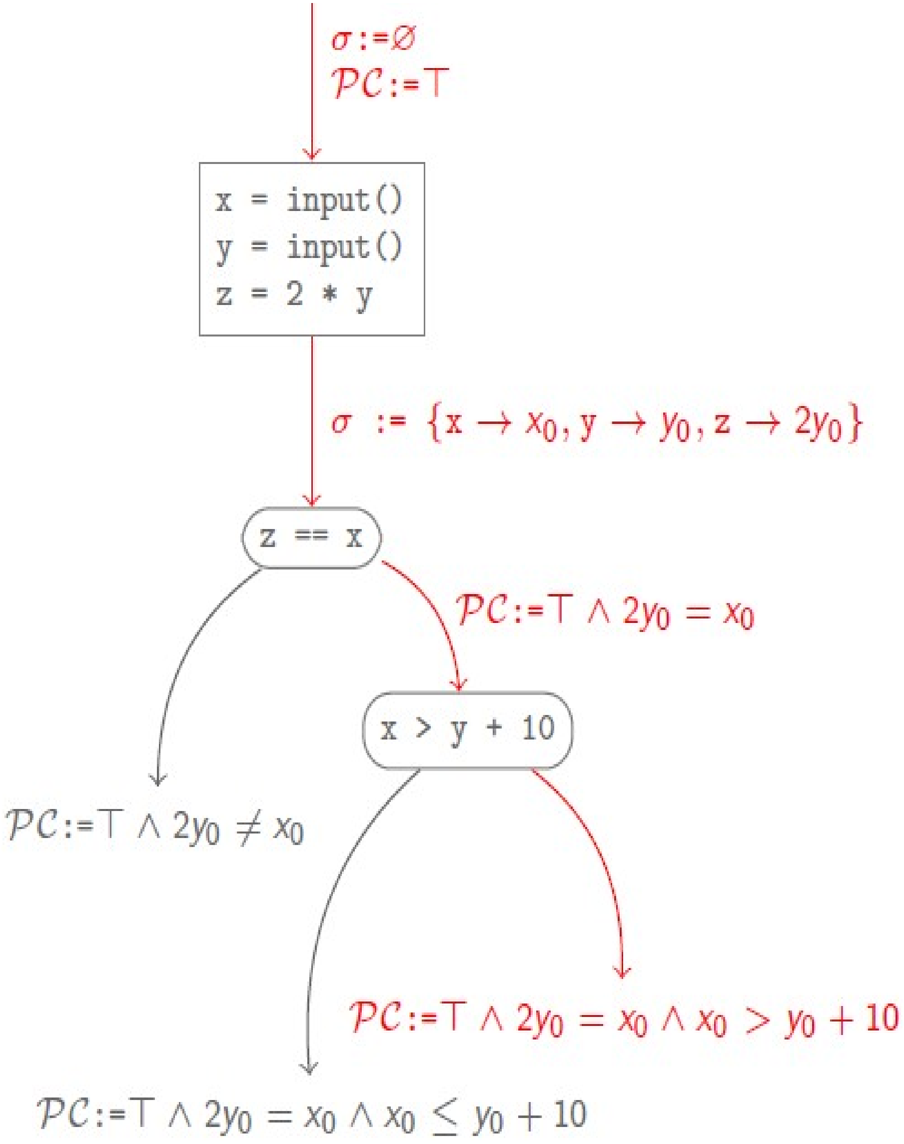
A TOOL OF CHOICE: SYMBOLIC EXECUTION (the fabulous 2005 year)

Find real bugs

Bounded verification

Flexible

```
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 a program

- Compute its « path predicate » f
- Solution of f = input following the path
- Solve it with powerful existing solvers

Détour : ABOUT FORMAL METHODS AND CODE ANALYSIS

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

Key concepts : $M \models \varphi$

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

Success in (regulated) safety-critical domains

Détour : 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
- φ : property to be checked
- $\models$: algorithmic check

- Reason about infinite sets of behaviours

- Typical ingredients: transition systems, automata, logic, …

Success in (regulated) safety-critical domains

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]
- ressource usage [Absint]

* : by CEA DILS/LSL

Détour : 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 : $M \models \varphi$

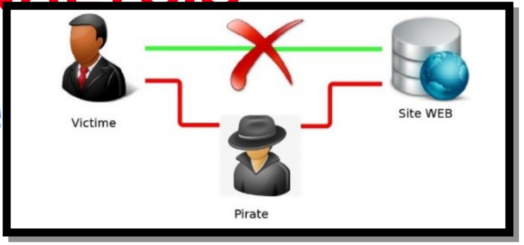
- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

- Reason about infinite sets of behaviours

- Typical ingredients: transition systems, automata, logic, …

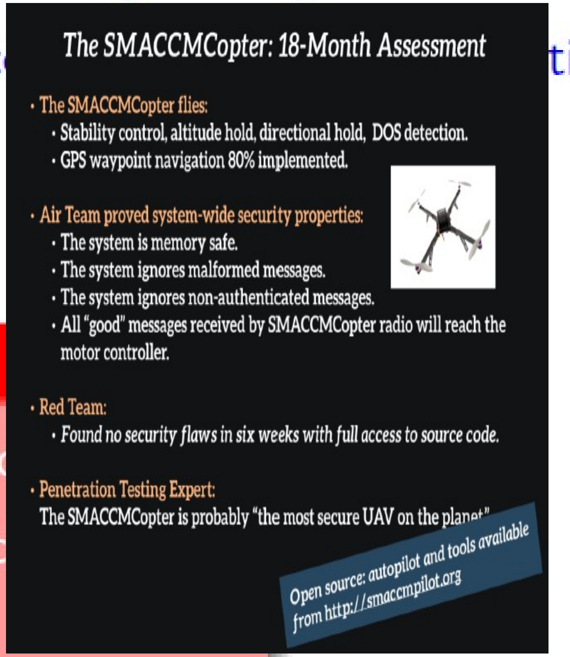
Success in (regulated) safety-critical domains

Détour : ABOUT FORMAL METHODS AND CODE ANALYSIS



- Between Software Engineering and Theoretical Computer Science
- Goal = proves c… …tical way

TLS 1.3


The SMACCMCopter: 18-Month Assessment

- The SMACCMCopter 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 SMACCMCopter 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 SMACCMCopter is probably "the most secure UAV on the planet"

Open source: autopilot and tools available from http://smaccmpilot.org

Key concepts : $M \models \varphi$

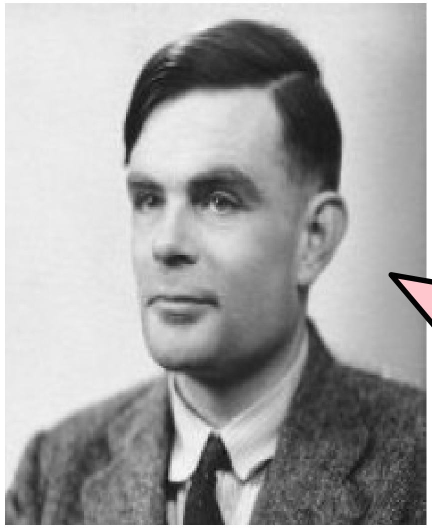
- M : semantic of the pr…
- φ : property to be chec…
- …ic check

Windows 10

CGC CYBER GRAND_CHALLENGE

A big success in many more domains!

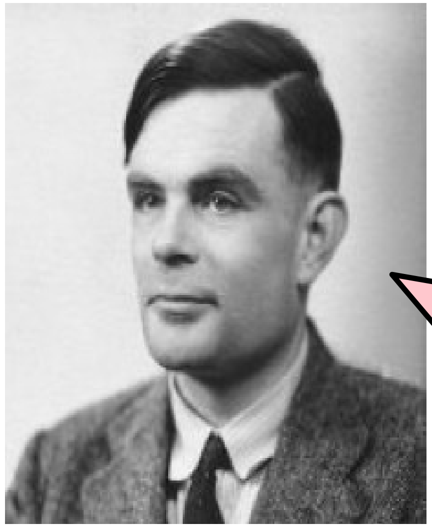
WAIT ??!!! Verification is undecidable



Cannot have analysis that
- Terminates
- Is perfectly precise

On all programs

They knew it was impossible, so they did it anyway



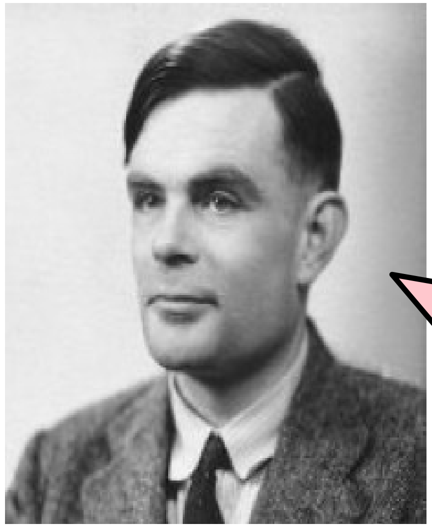
Cannot have analysis that
- Terminates
- Is perfectly precise

On all programs

- **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

They knew it was impossible, so they did it anyway



Cannot have analysis that
- Terminates
- Is perfectly precise

On all programs

- **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

Back in 2005 ...

Despite some successes, still several issues

- Lack of robustness
- False positive (centered on proving safety)
- May require (lots of) annotations

« Moving from a dream of automatic verification to a reality of automated debugging »
T. A. Henzinger

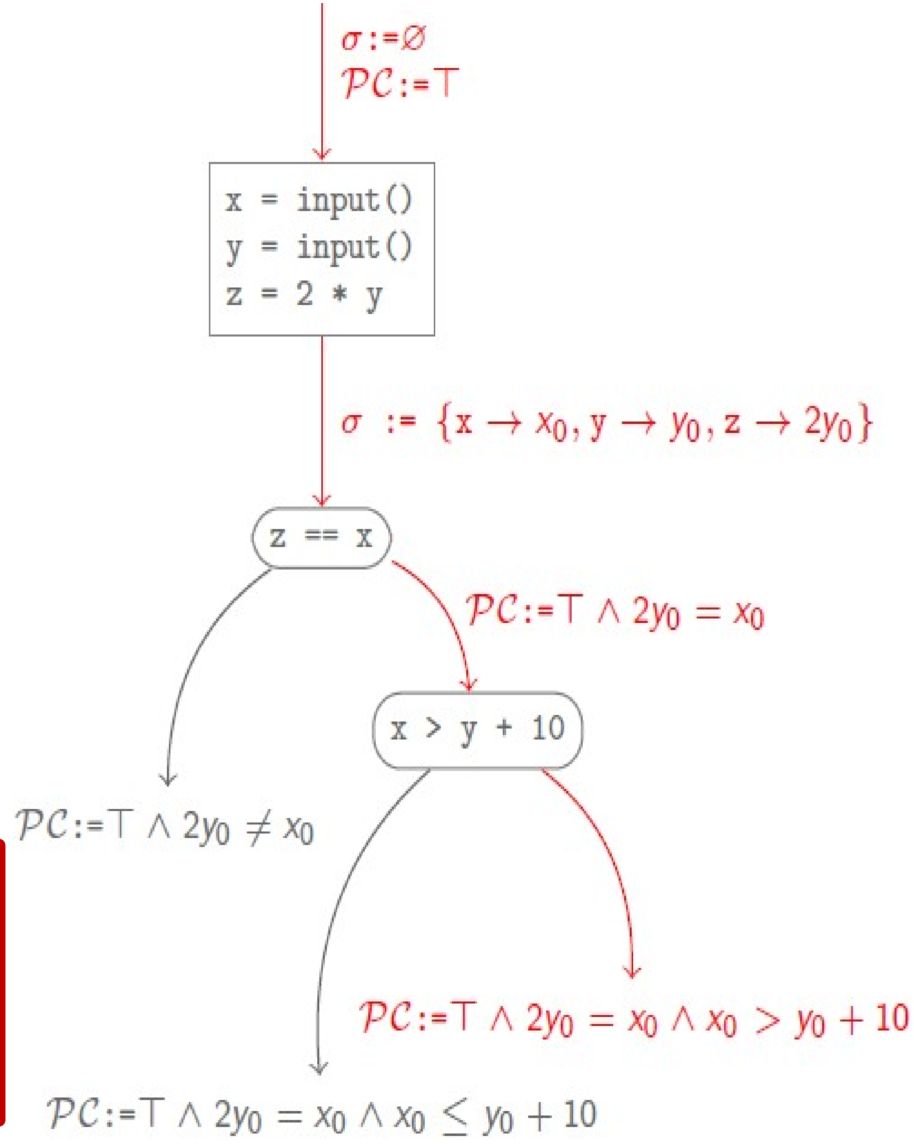
A TOOL OF CHOICE: SYMBOLIC EXECUTION (rebirth in 2005)

Find real bugs

Bounded verification

Flexible


```
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 a program

- Compute its « path predicate » f
- Solution of f = input following the path
- Solve it with powerful existing solvers

PATH PREDICATE COMPUTATION & SOLVING

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

$$\text{let } W_1 \triangleq Y_0 + 1 \text{ in}$$
$$\text{let } X_2 \triangleq W_1 + 3 \text{ in}$$
$$X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

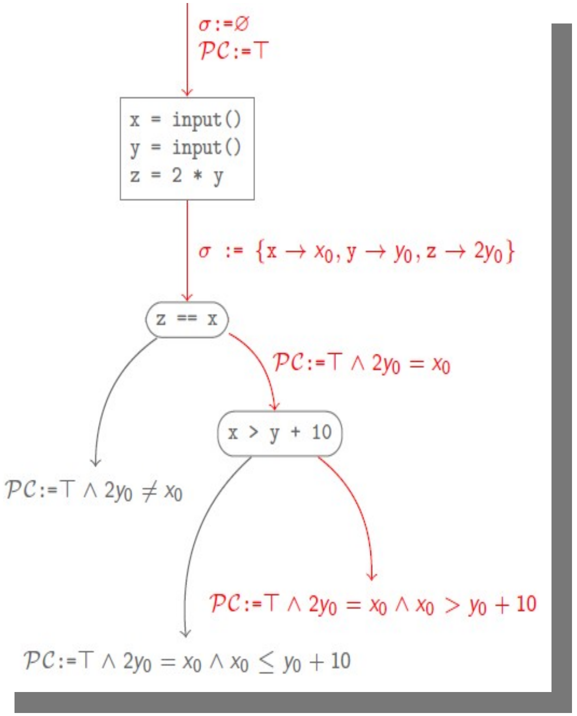
Blackbox solvers

Z3 Boolector CVC4

SMT Solver

my input!!

Y0 = 0 /\ Z0=3



PATH PREDICATE COMPUTATION & SOLVING

Loc	Instructi…
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (…
4	if (x < z) (bran…

Key ingredients
- Path search
- Constraint solving

Beware
- Path explosion
- Constraint solving cost

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

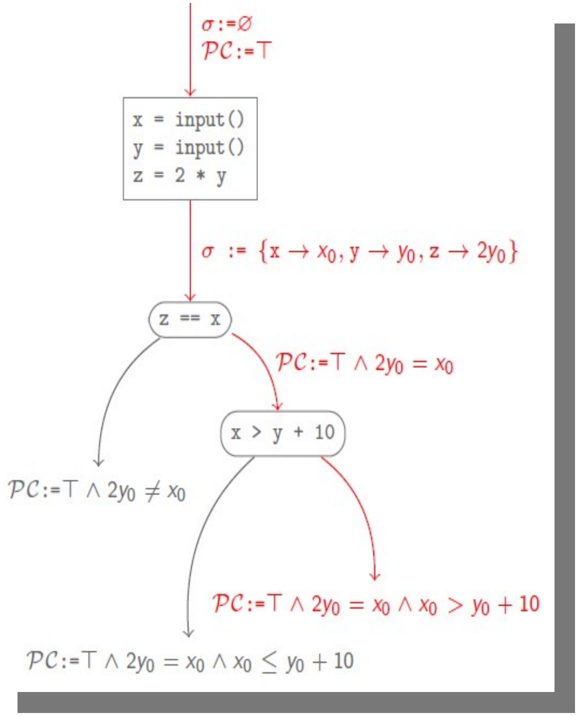
$$\text{let } W_1 \triangleq Y_0 + 1 \text{ in}$$
$$\text{let } X_2 \triangleq W_1 + 3 \text{ in}$$
$$X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Blackbox solvers

SMT Solver

my input!!

Y0 = 0 /\ Z0=3



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) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

« concretization »

- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

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) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

Symbolic Execution

- create a subformula $z = x * x$, out of theory [FAIL]

« concretization »

- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

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) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

Symbolic Execution

- create a subformula $z = x * x$, out of theory [FAIL]

Dynamic Symbolic Execution

- first concrete execution with x=3, y=5 [goto OK]

« concretization »

- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

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) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

Symbolic Execution

- create a subformula $z = x * x$, out of theory [FAIL]

Dynamic Symbolic Execution

- first concrete execution with x=3, y=5 [goto OK]
- during path predicate computation, $x * x$ not supported
  . x is concretized to 3 and z is forced to 9
- resulting path predicate : $x = 3 \wedge z = 9 \wedge y = z$

« concretization »

- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

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) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

Symbolic Execution

- create a subformula $z = x * x$, out of theory [FAIL]

Dynamic Symbolic Execution

- first concrete execution with x=3, y=5 [goto OK]
- during path predicate computation, $x * x$ not supported
  - x is concretized to 3 and z is forced to 9
- resulting path predicate : $x = 3 \wedge z = 9 \wedge y = z$
- a solution is found : x=3, y=9 [goto ERROR] [SUCCESS]

« concretization »

- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

ABOUT ROBUSTNESS (imo, the major advantage)

« concretization »

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

Very powerful

- Unsupported code
- Too costly reasoning
- Multi-thread
- Self-modification or packing
- …

Some optimizations

- formula simplifications
  - [memory, specific patterns]
- formula caching
- reuse of concrete models
- better modelling
- concretization
- ML-based (non-)solving
- …

- Search heuristics
  - Coverage, goal, novelty
  - ML-based search
- Path merging
- Path pruning (past, future)
- …

- parallelism
- pre-compilation
- ratio symbolic - concrete
- optimized implementations

Take away

Pros

- Find real bugs
- Robust (concretization)

- Pay as you go : bounded verification vs bug hunt
  -

- Flexible : properties, kind of analysis
  - local proofs, relational analysis, probabilistic, repair, synthesis, ...

- Rather natural to combine with dynamic analysis

Some issues & challenges

- Beware of #paths ! (loop, functions)
  - fully modular SE ?

- Beware of constraints (crypto mainly)

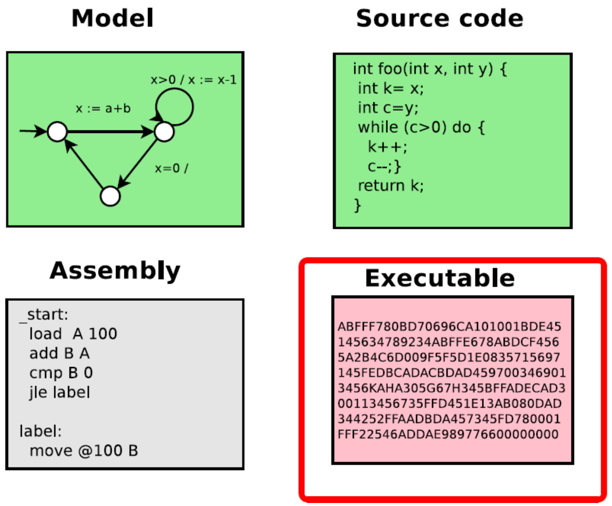
- End-to-end analysis : scale ?
- Local analysis : initialization ?

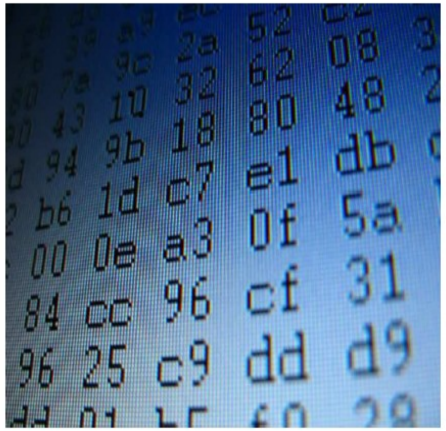
- Advanced langage features ?
  - OO, functional, dynamic code, etc.

OUTLINE

- **Introduction**
- **What every honest person should know about Symbolic Execution**
- **Challenges of automated binary-level security analysis**
- **BINSEC & Symbolic Execution for Binary-level Security**
- **Shades of Symbolic Execution for Security**
- **Conclusion, Take away and Disgression**

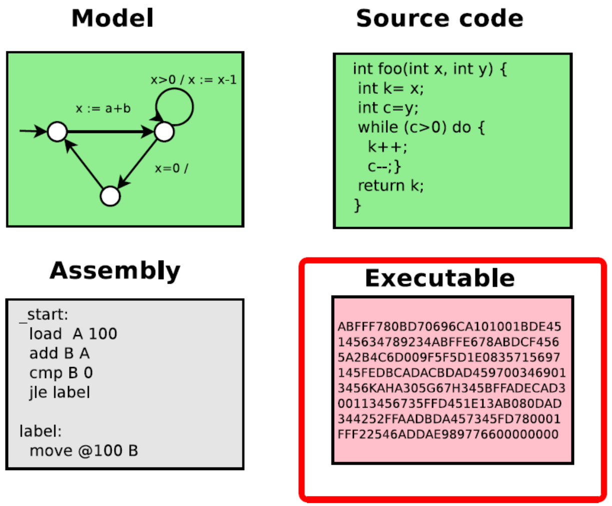
New challenges!

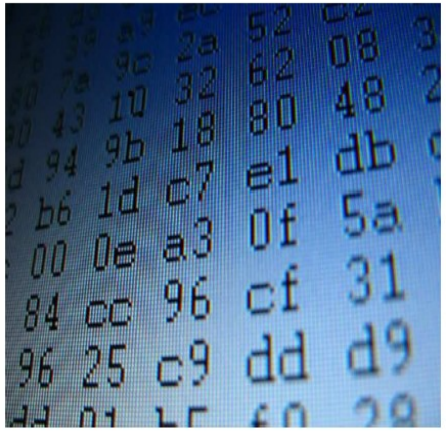




- Binary code
- Attacker
- Properties

New challenges!





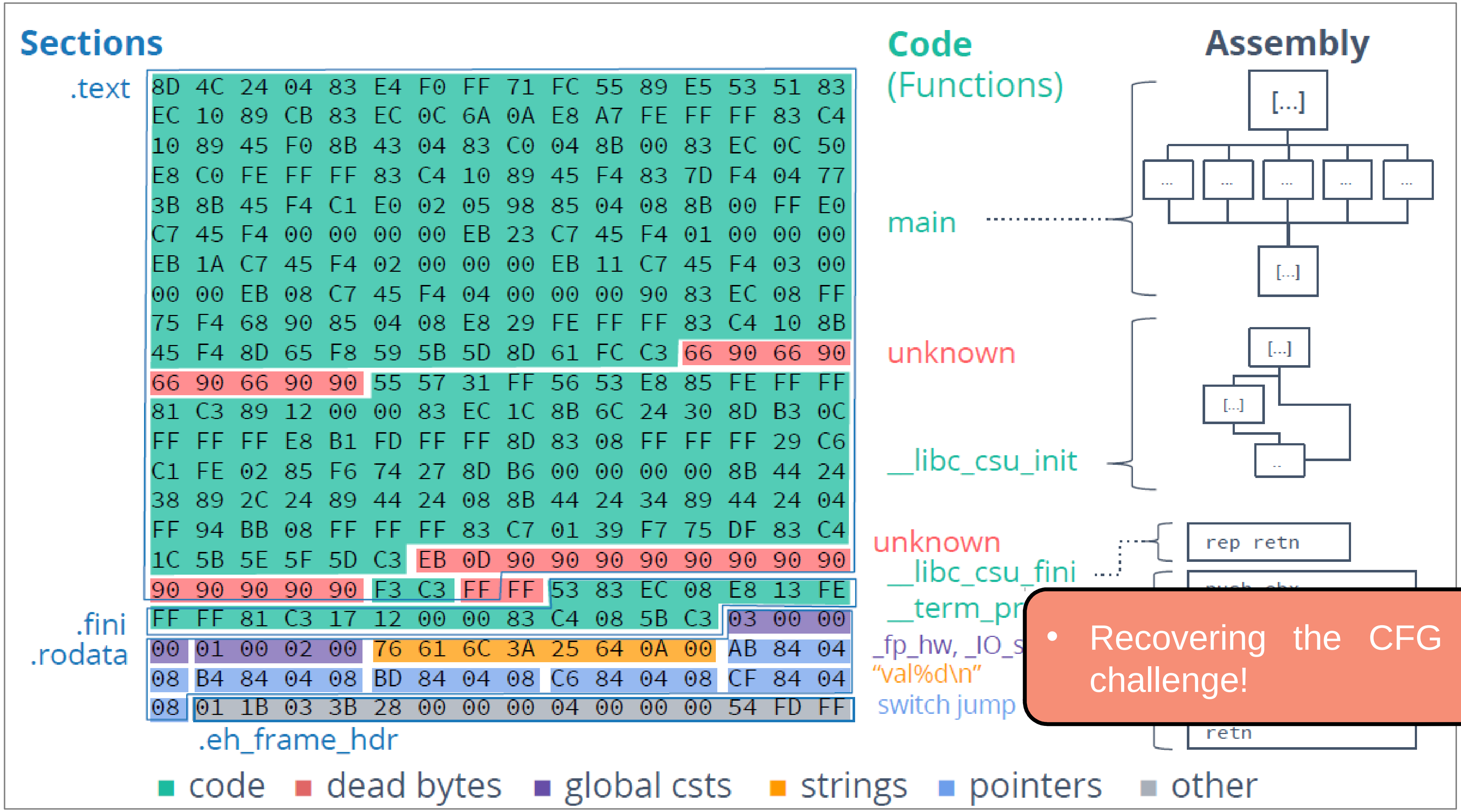
- Binary code
- Attacker
- Properties

CHALLENGE: BINARY CODE LACKS STRUCTURE

- Instructions?
- Control flow?
- Memory structure?

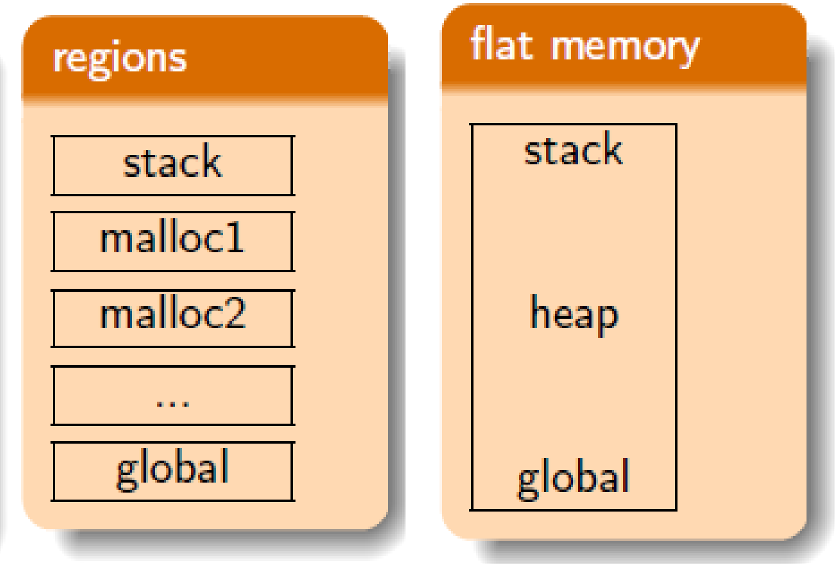
DISASSEMBLY IS ALREADY TRICKY!

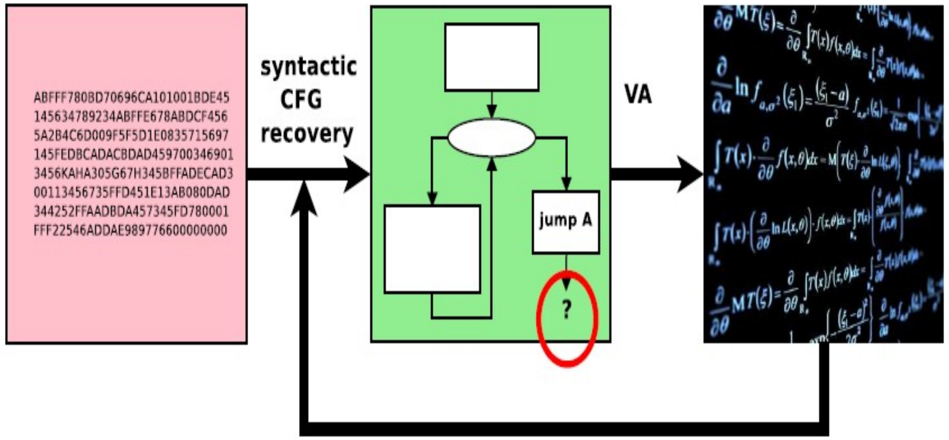
- code – data ??
- dynamic jumps (jmp eax)



- Recovering the CFG is already a challenge!

BINARY CODE SEMANTIC LACKS STRUCTURE





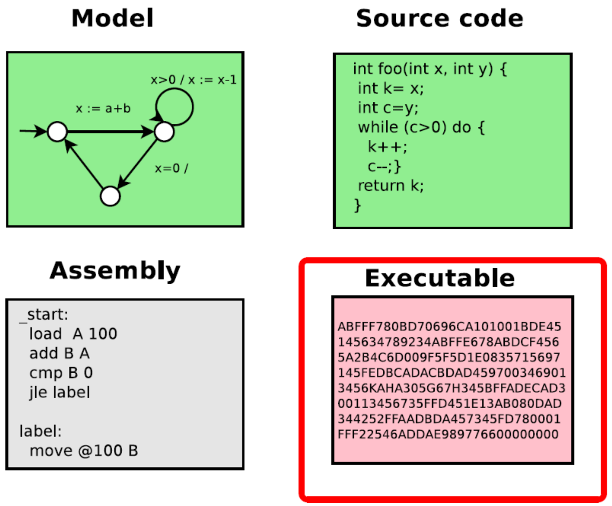

```
if (ax > bx) X = -1;
else X = 1;
```

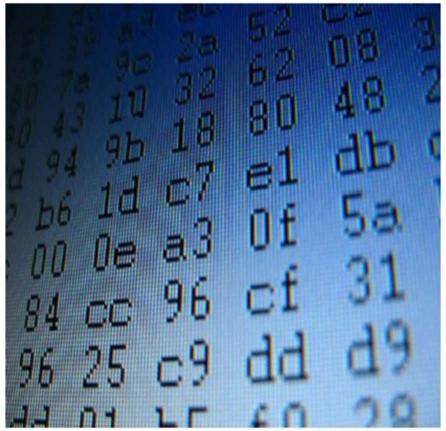
```
     OF := ((ax{31,31}≠bx{31,31}) &
                (ax{31,31}≠(ax-bx){31,31}));
     SF := (ax-bx) < 0;
     ZF := (ax-bx) = 0;
     if (¬ ZF ∧ (OF = SF)) goto l1
     X := 1
     goto l2
l1:  X := -1
l2:
```

Problems

- Jump eax
- Untyped memory
- Bit-level resoning

New challenges!



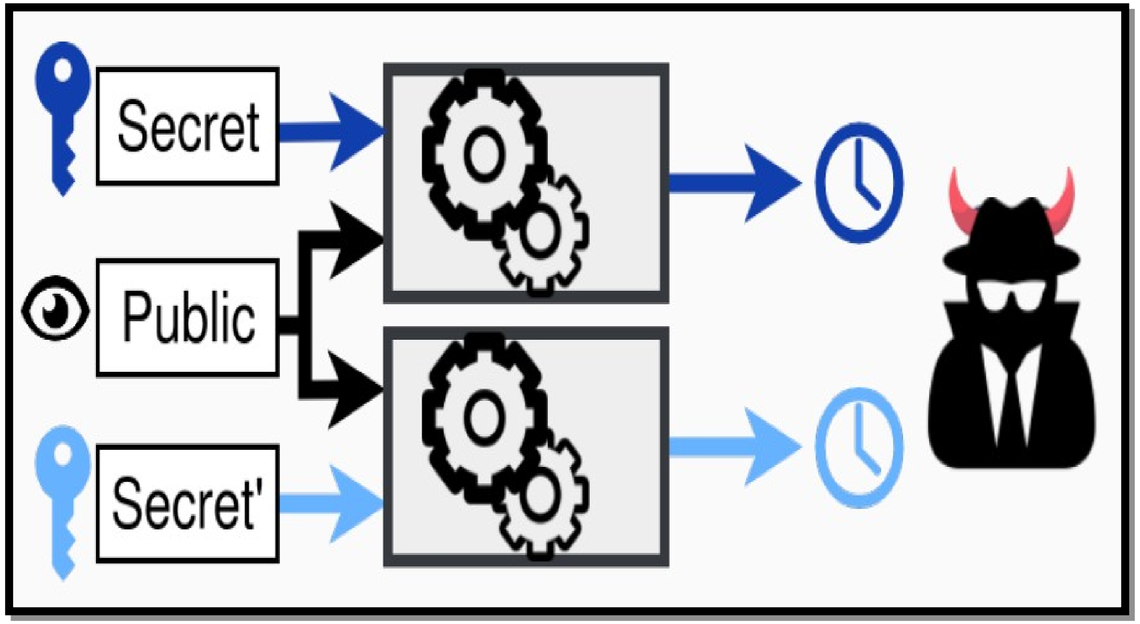


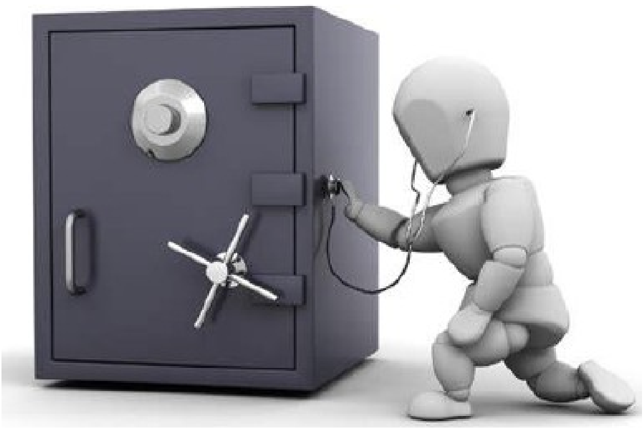
- Binary code
- Attacker
- Properties

New challenge : safety is not hyper-property :-)

Information leakage

Properties over pairs of executions

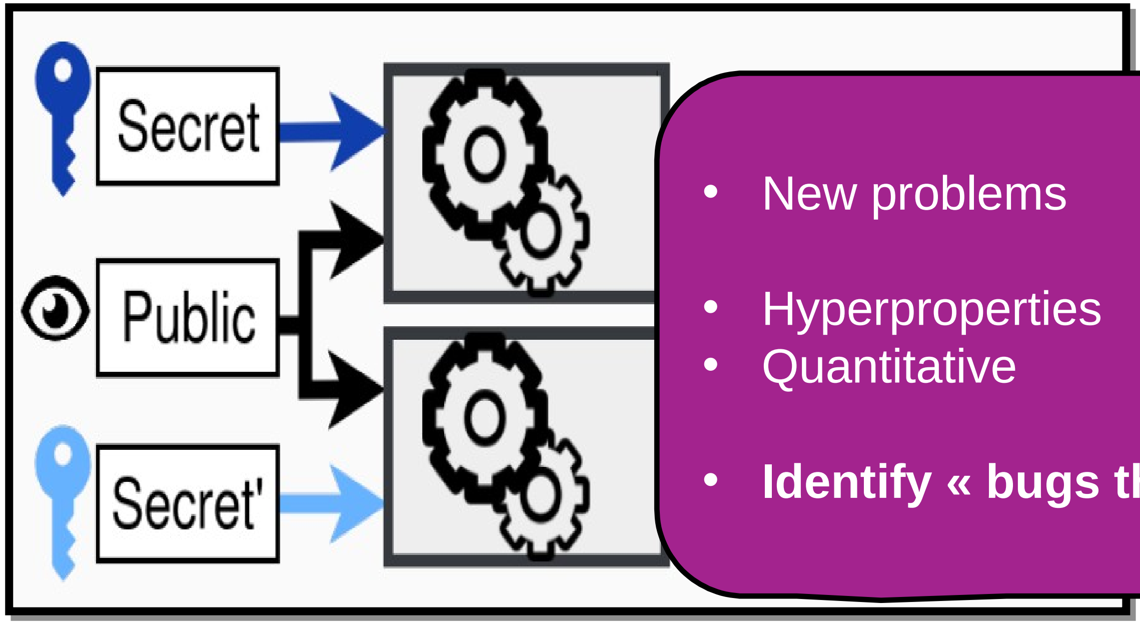




New challenge : safety is not hyper-property :-)

Information leakage

Properties over pairs of executions

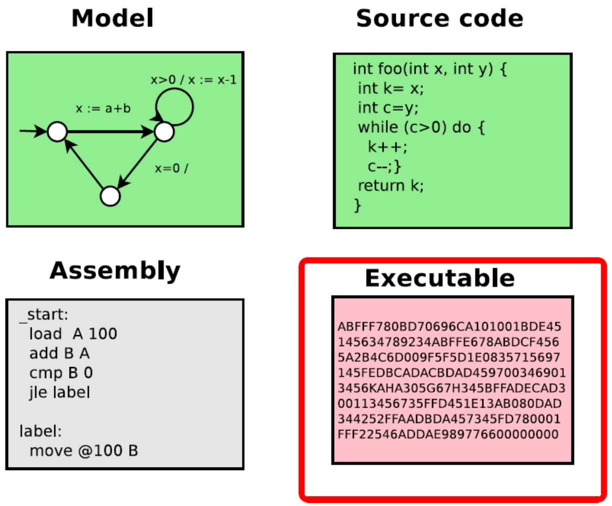


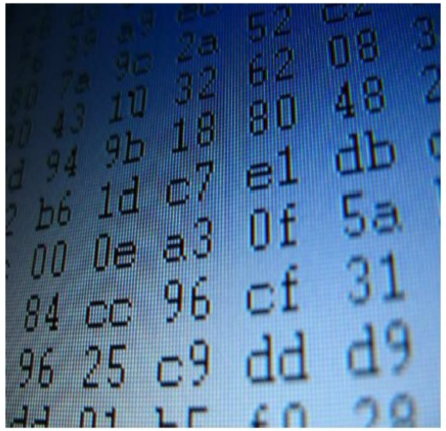
- New problems

- Hyperproperties
- Quantitative

- **Identify « bugs that matters »**

New challenges!





- Binary code

- Attacker

- Properties

CHALLENGE: ATTACKER

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!**

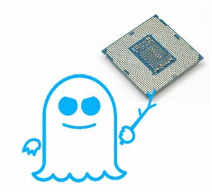
- **Still, our current attacker plays the rules: respects the program interface**
  - Can craft **very smart input**, but only through **expected input sources**

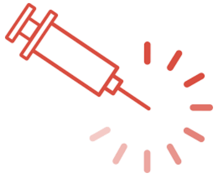
ATTACKER in Standard Program Ana

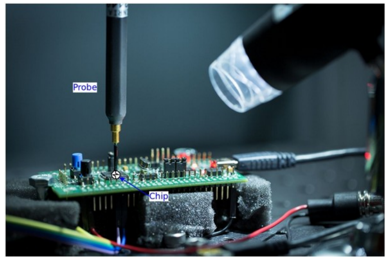
- **We are reasoning worst case: seems very**

- **Still, our attacker plays the rules: respects**
  - Can craft very smart input, but only through expecte

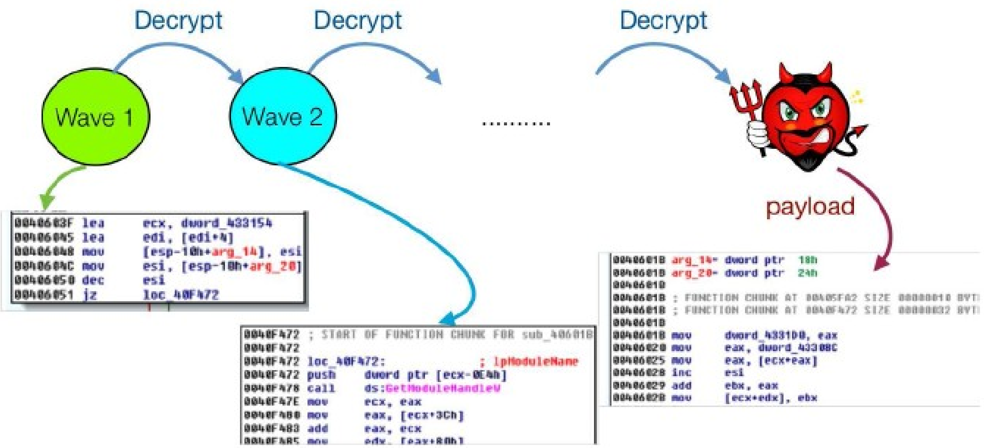
- **What about someone who really do not play the rules?**
  - Side channel attacks
  - Micro-architectural attacks
  - Fault injections







Another Line of attack : ADVERSARIAL BINARY CODE



address	instr
80483d1	call +5
80483d6	pop edx
80483d7	add edx, 8
80483da	push edx
80483db	ret
80483dc	.byte{invalid}
80483de	[...]

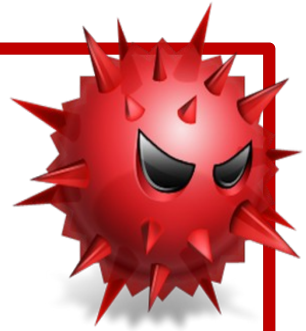
eg: $7y^2 - 1 \neq x^2$

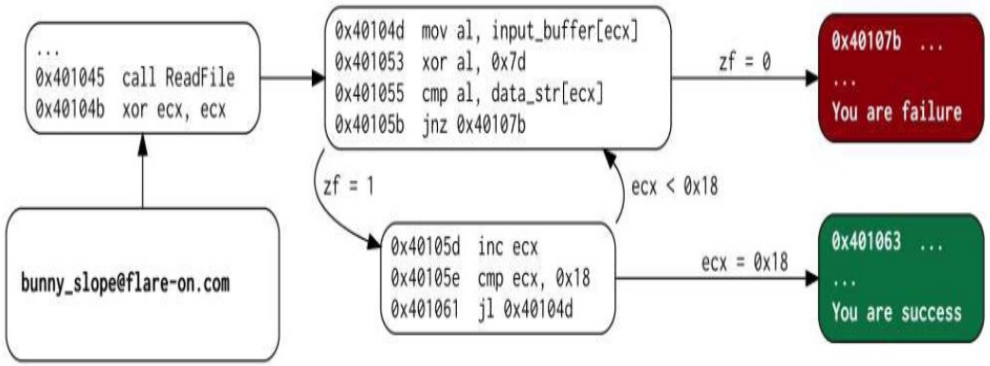
(for any value of x, y in modular arithmetic)

↧

```
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>
```

- **self-modification**
- **encryption**
- **virtualization**
- **code overlapping**
- **opaque predicates**
- **callstack tampering**
- **...**

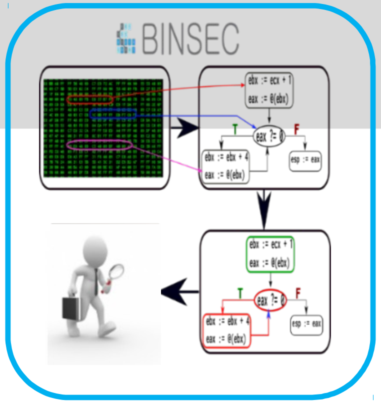




OUTLINE

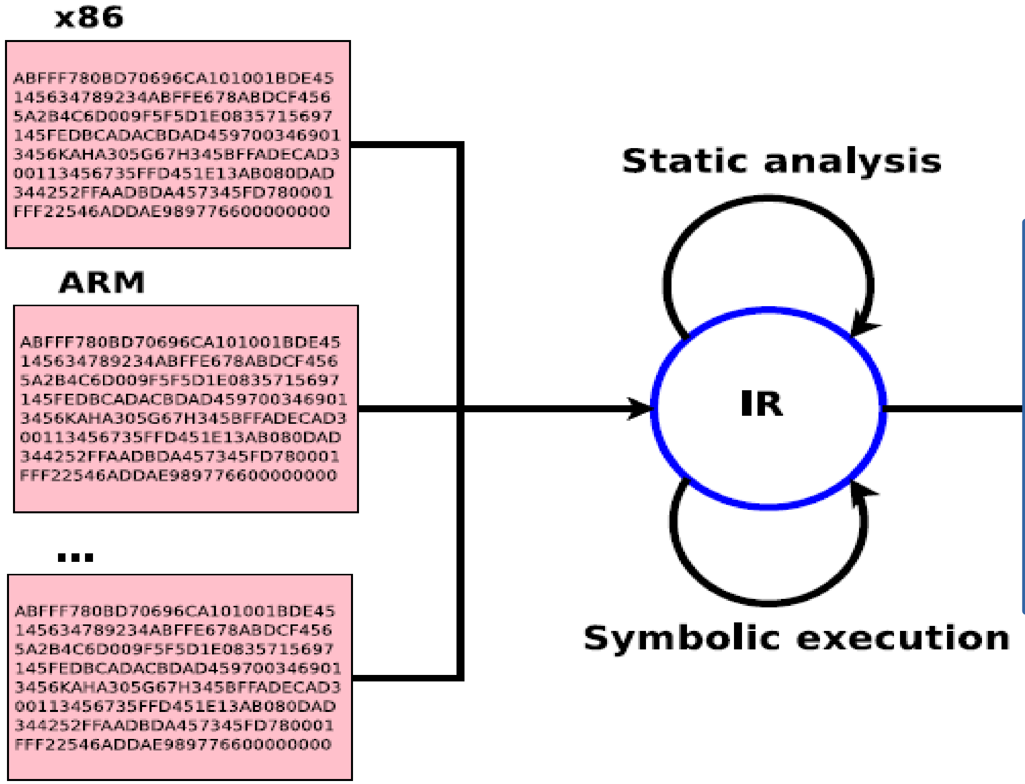
- Introduction
- What every honest person should know about Symbolic Execution
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

BINSEC: brings formal methods to binary-level security analysis



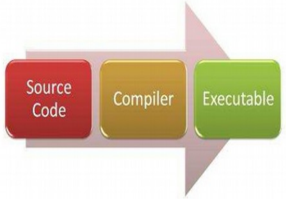
Break | Prove | Protect

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

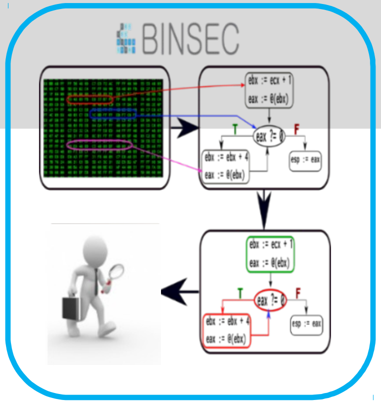


- **Advanced reverse**
- **Vulnerability analysis**
- **Binary-level security proofs**
- **Low-level mixt code (C + asm)**
- …

https://binsec.github.io/

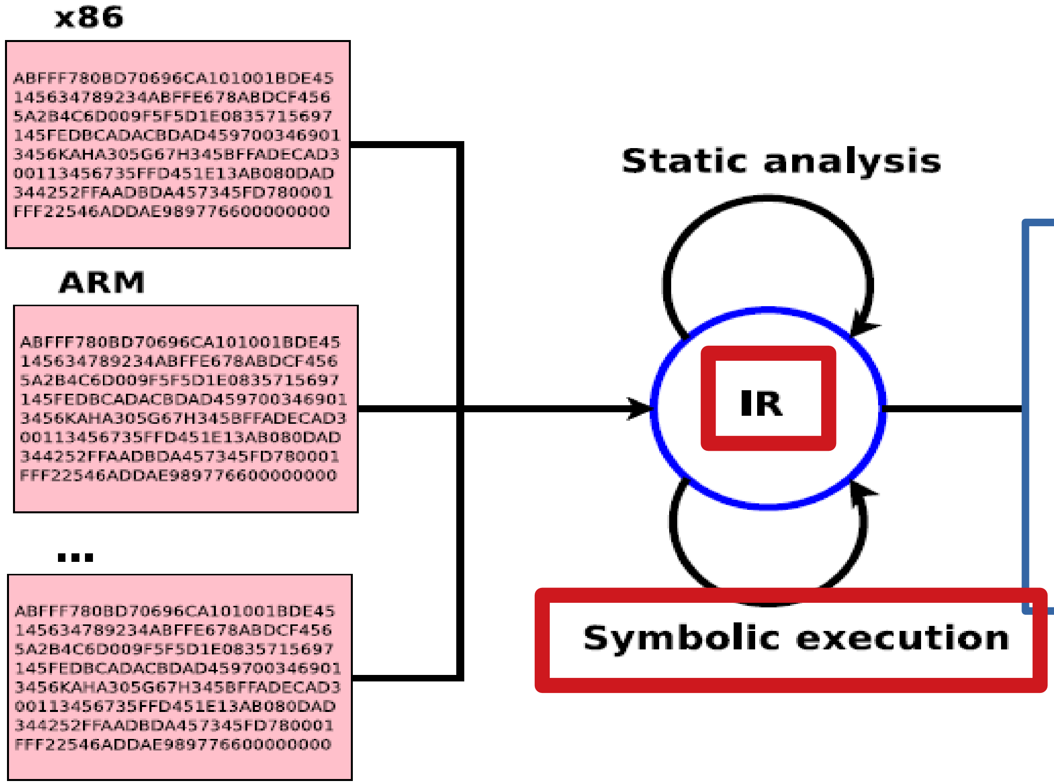


BINSEC: brings formal methods to binary-level security analysis



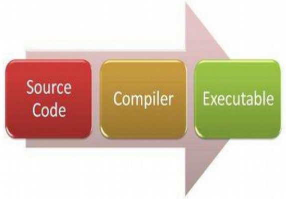
Break | Prove | Protect

- **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)
- ...

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]$_n$
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	:=	∧ | ∨ | ⊕ | shl | shr | sar
cmp	:=	= | ≠ | $>_u$ | $<_u$ | $>_s$ | $<_s$

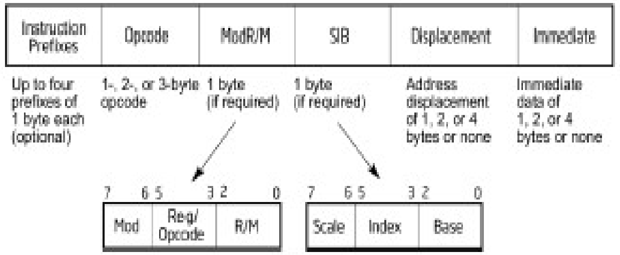
Multi-architecture

x86-32bit – ARMv7

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

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

INTERMEDIATE REPRESENTATION



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

81 c3 57 1d 00 00 $\overset{x86reference}{\Longrightarrow}$ ADD EBX 1d57

```
(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 2: SYMBOLIC EXECUTION

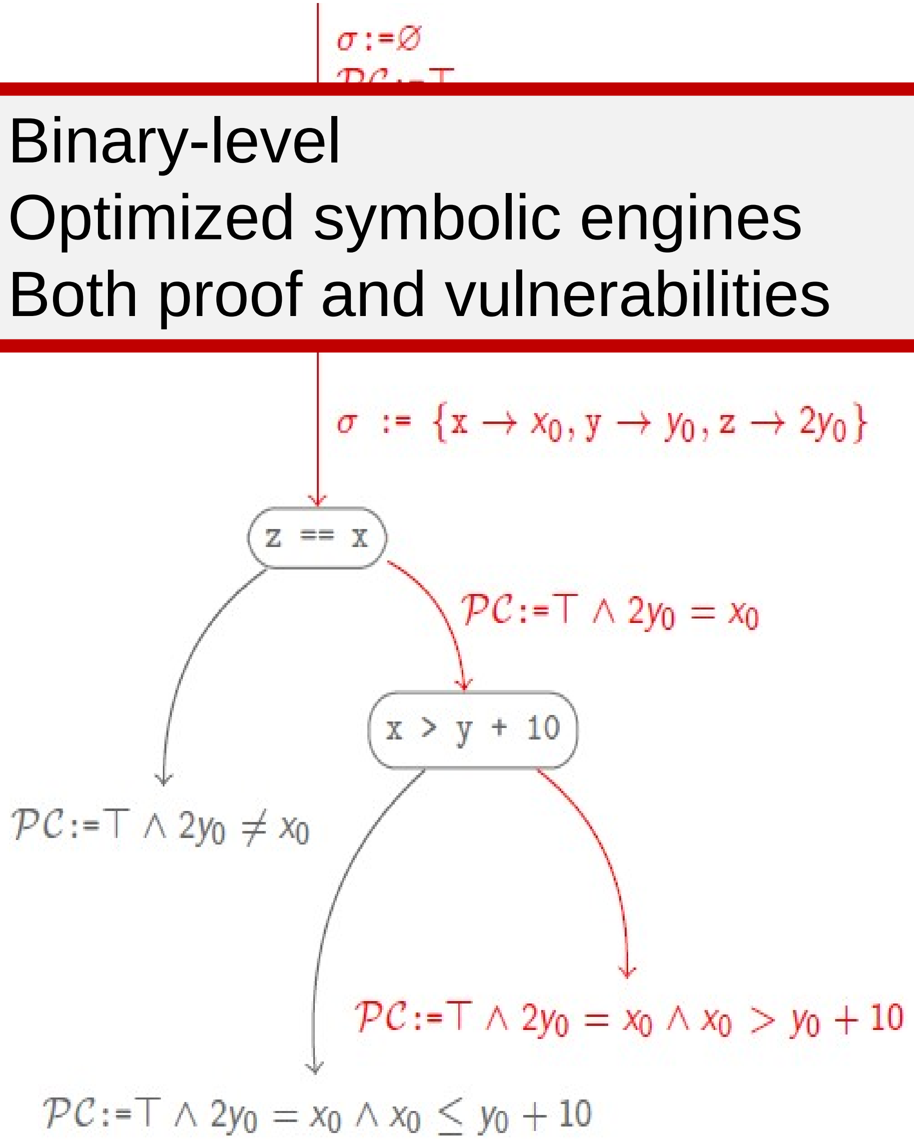
Find real bugs

Bounded verification

Flexible

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

- Binary-level
- Optimized symbolic engines
- Both proof and vulnerabilities

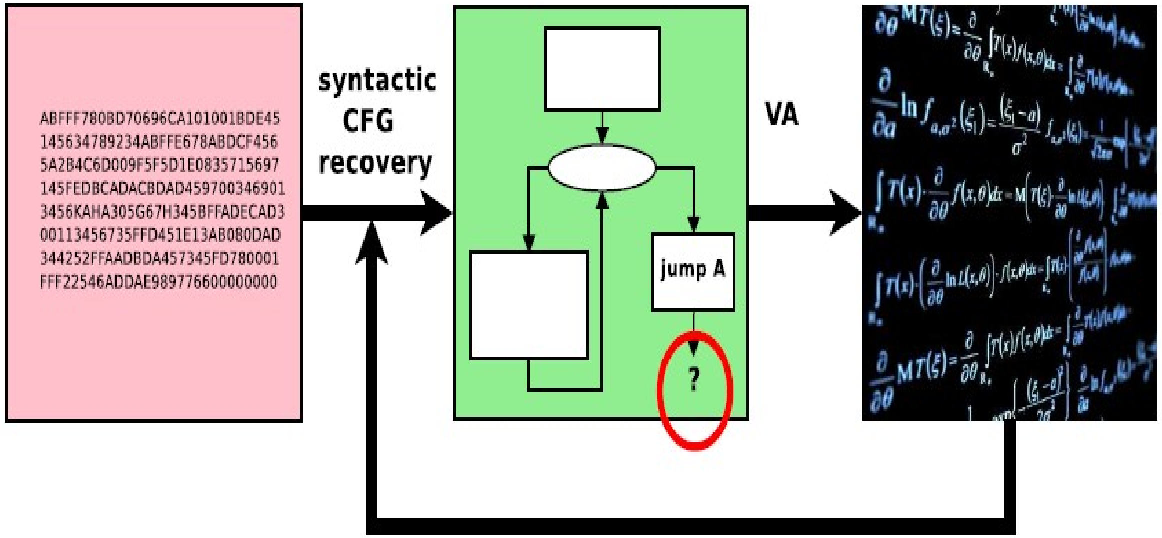


Given a path of a program

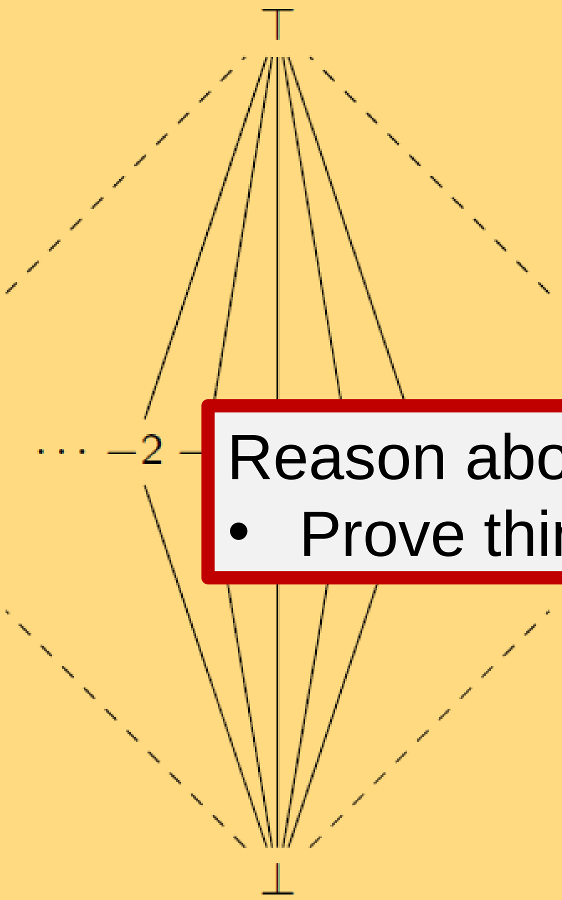
- Compute its « path predicate » f
- Solution of f = input following the path
- Solve it with powerful existing solvers

ALSO: STATIC SEMANTIC ANALYSIS
(harder, doable on *some* classes of programs)

Complete verification



Generalize constant propagation

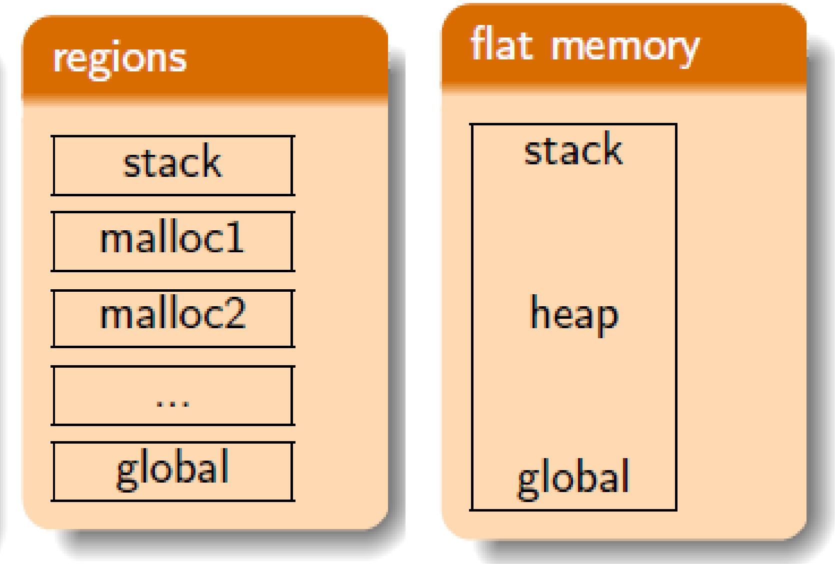


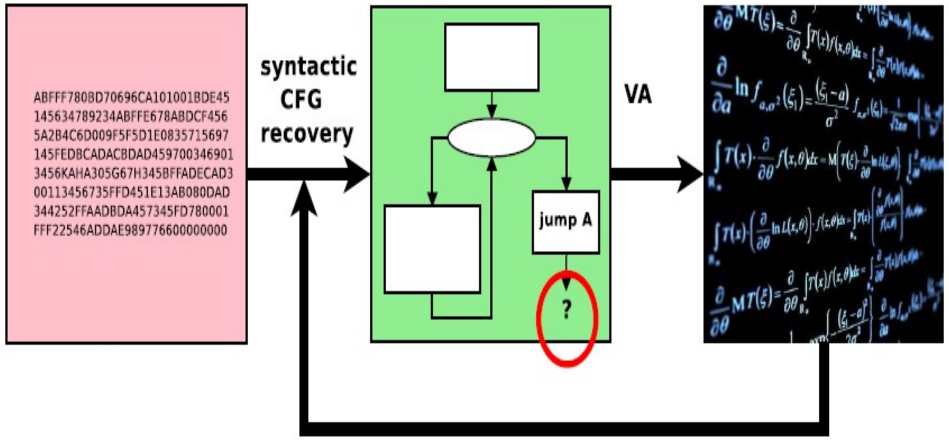
Reason about all paths
- Prove things

Framework : abstract interpretation

- notion of abstract domain
  - $\bot, \top, \sqcup, \sqcap, \sqsubseteq, \text{eval}^{\#}$
- more or less precise domains
  - . intervals, polyhedra, etc.
- fixpoint until stabilization

REMINDER: BINARY CODE SEMANTIC LACKS STRUCTURE





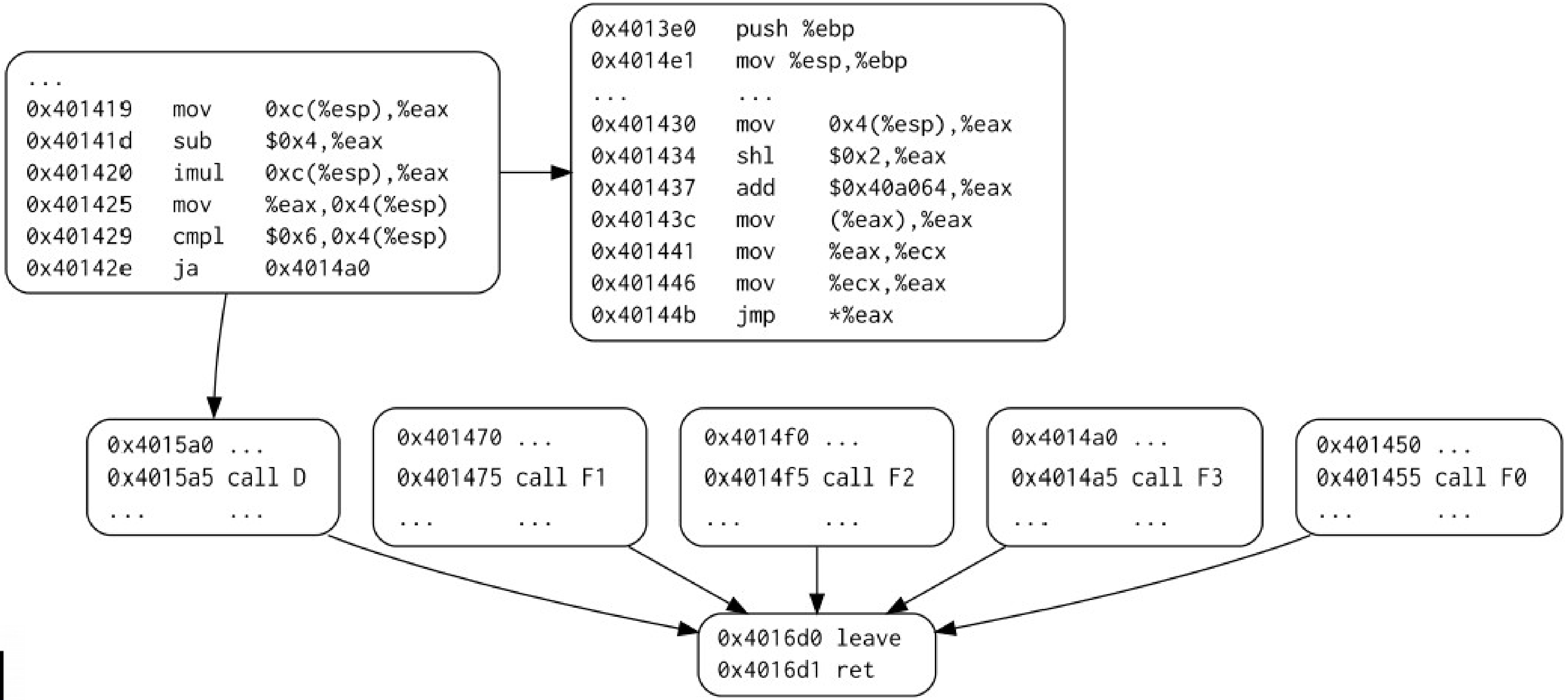

```
if (ax > bx) X = -1;
else X = 1;
```

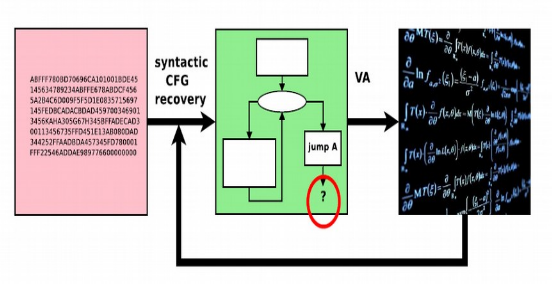
```
        OF := ((ax{31,31}≠bx{31,31}) &
                  (ax{31,31}≠(ax-bx){31,31}));
        SF := (ax-bx) < 0;
        ZF := (ax-bx) = 0;
        if (¬ ZF ∧ (OF = SF)) goto l1
        X := 1
        goto l2
l1:     X := -1
l2:
```

Problems

- Jump eax
- Untyped memory
- Bit-level resoning

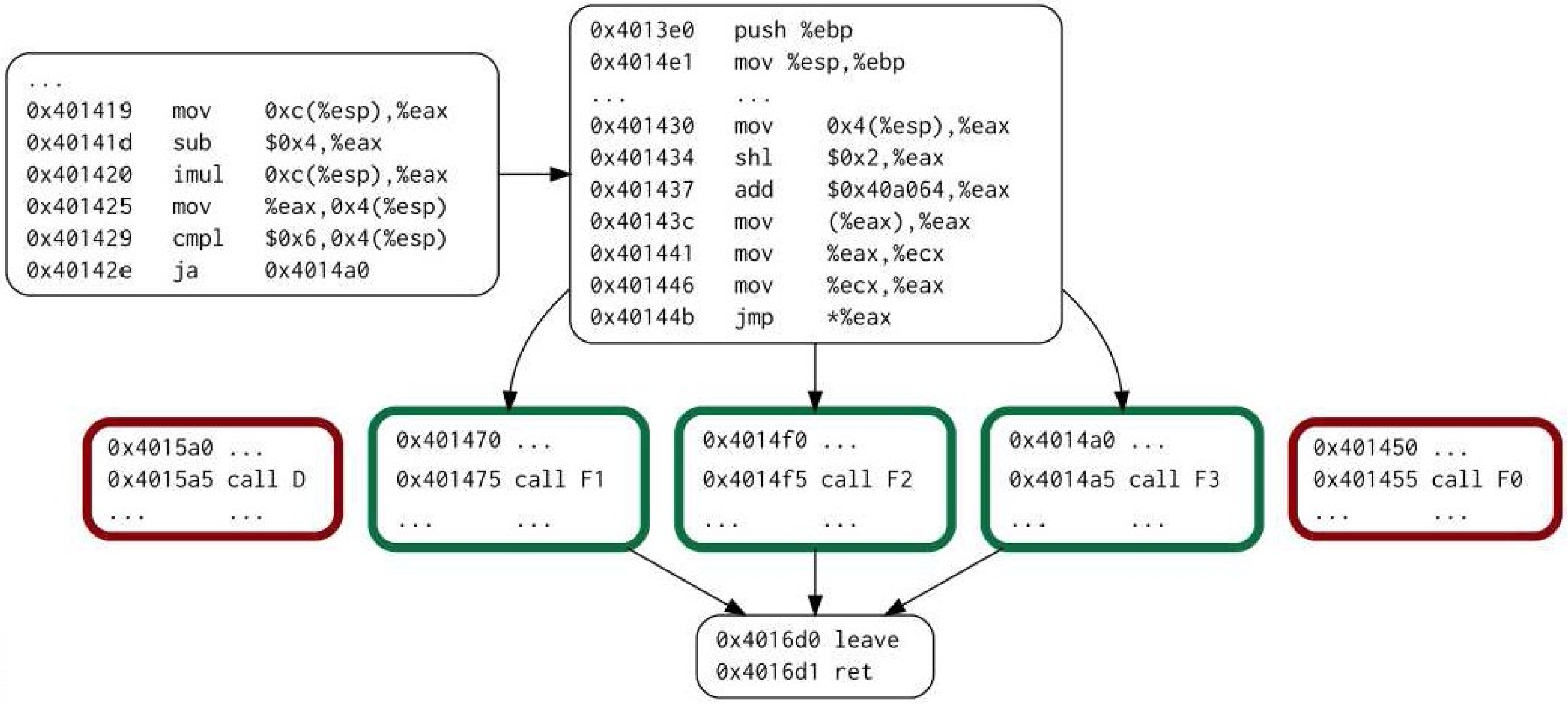
Dealing with dynamic jumps in SE is easy

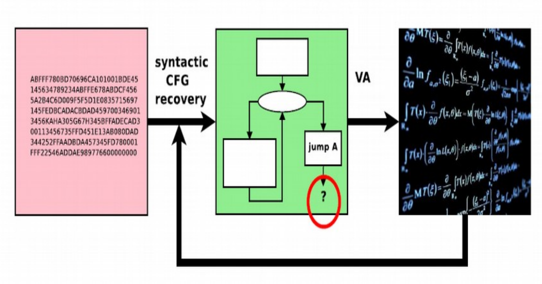




Dealing with dynamic jumps in SE is easy

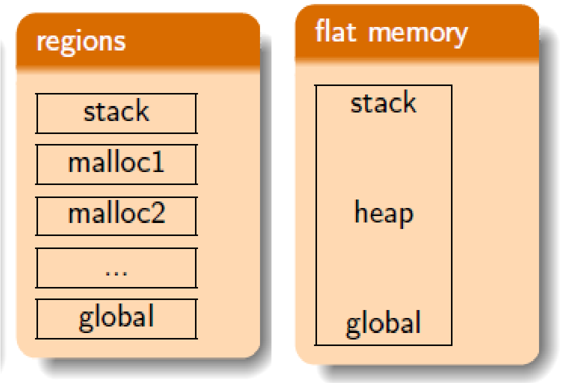
Get a first target
- **Then solve for a new one**
- **Get it, solve again, …**
- **Get them all!**





Dealing with memory is harder

- Bit-level resoning ⇒ theory of bitvectors (ok)
- Untyped memory ⇒ theory of arrays



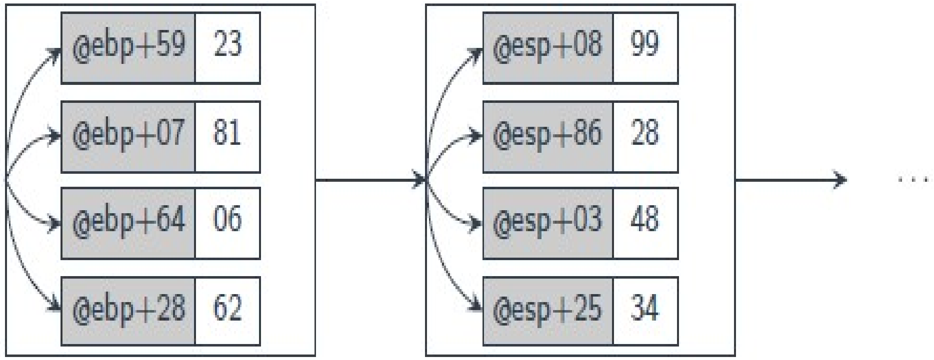
a single big array: solvers die

common solution: concretization

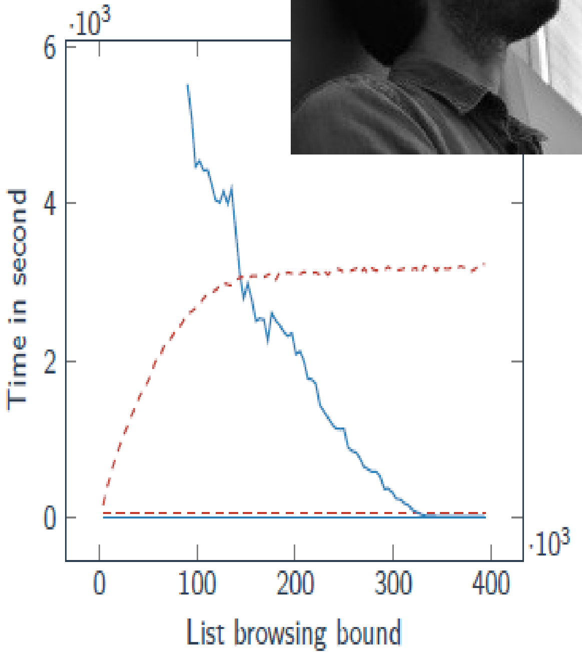
our solution: heavy simplification

Tuning the solver: intensive array formulas [LPAR 2018] (Benjamin Farinier)

- **Makes the difference!**



- 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

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

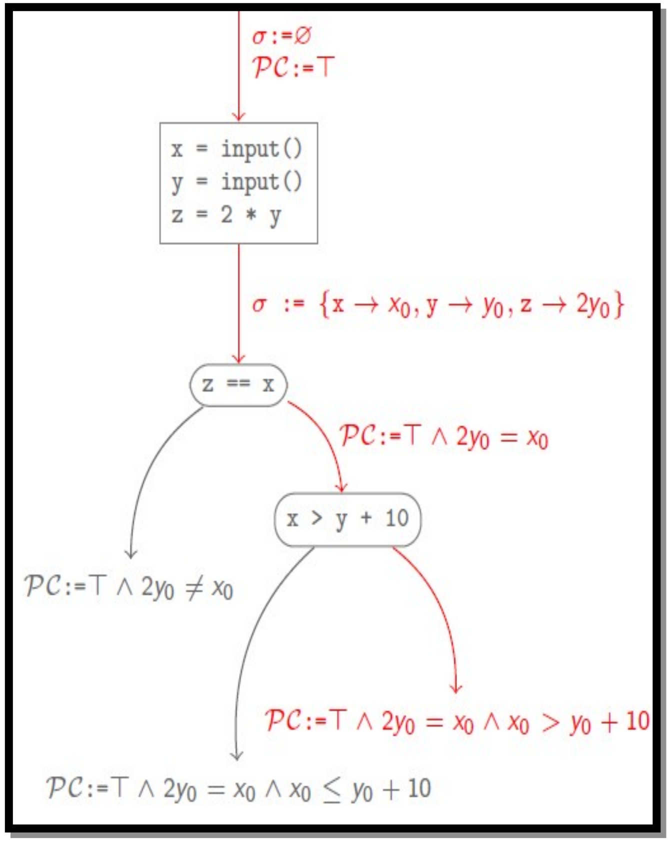
OUTLINE

- Introduction
- What every honest person should know about Symbolic Execution
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

OUTLINE

- Shades of Symbolic Execution for Security
  - Standard usage
  - Robust symbolic execution (CAV 2018, 2021)
  - Relational symbolic execution (S&P 2020)
  - Haunted symbolic execution (NDSS 2021)
  - Adversarial symbolic execution (ESOP 2023)

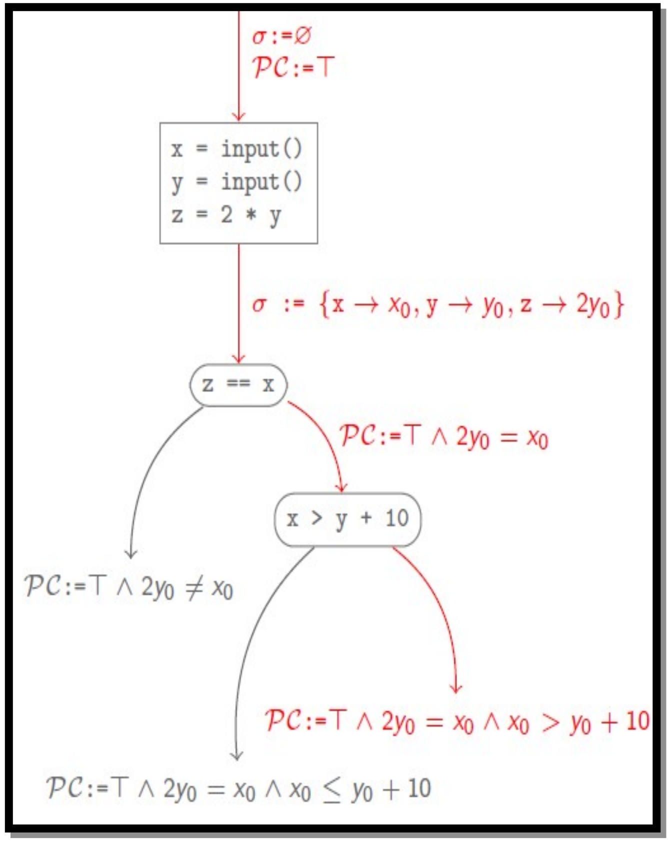
Vulnerability finding with symbolic execution (Godefroid et al., Cadar et al., Sen et al., etc.)



- Intensive path exploration

Challenge = path explosion

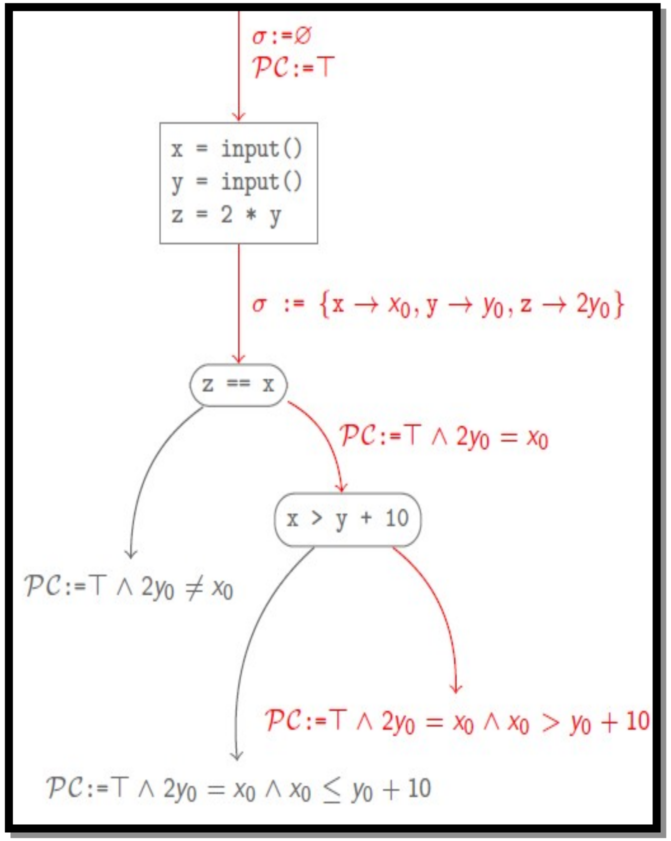
Vulnerability finding with symbolic execution
(Godefroid et al., Cadar et al., Sen et al., etc.)



- Intensive path exploration
- Target critical bugs

Challenge = path explosion

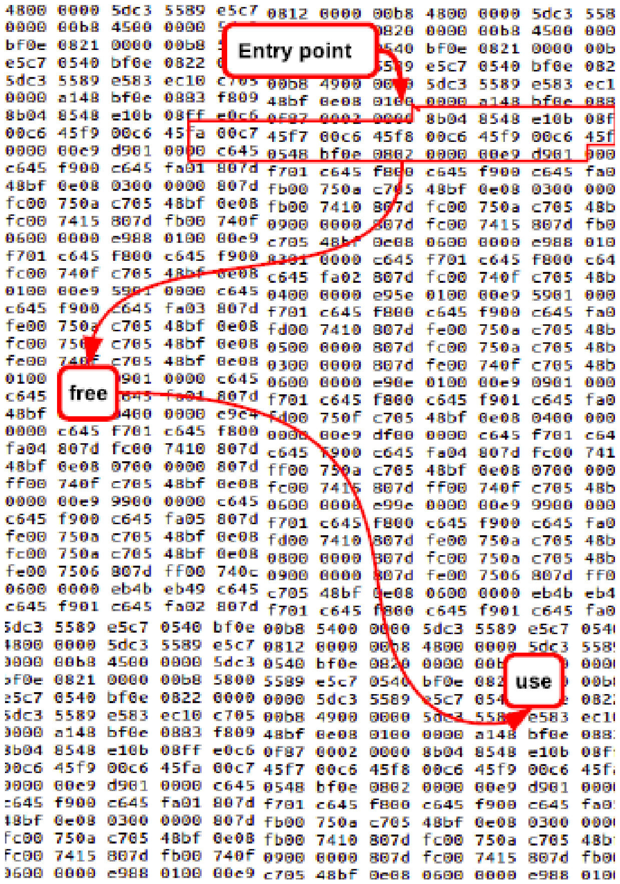
Vulnerability finding with symbolic execution (Heelan, Brumley et al.)



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

Challenge = path explosion

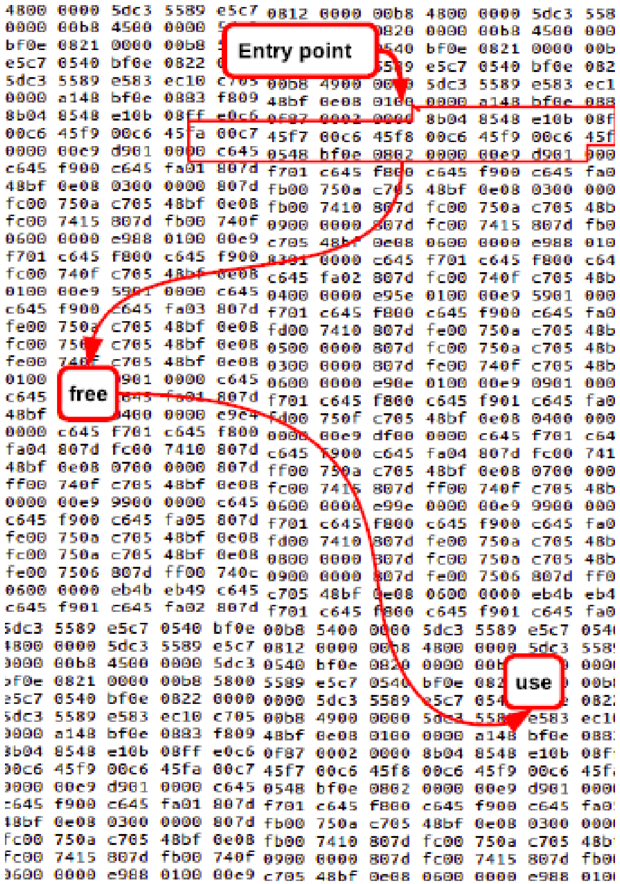
What about hard-to-find bugs ?
[SSPREW’16](with Josselin Feist et al.)

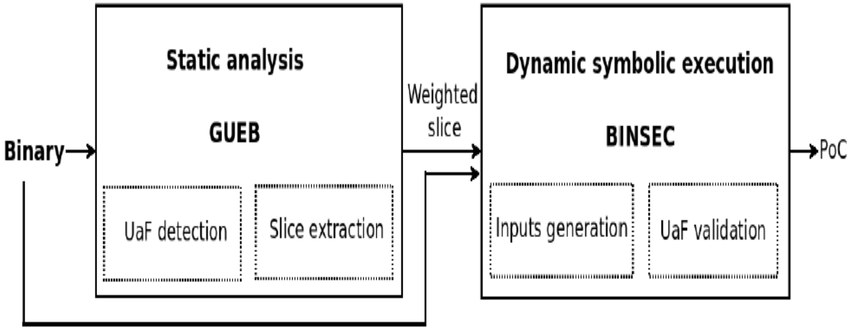


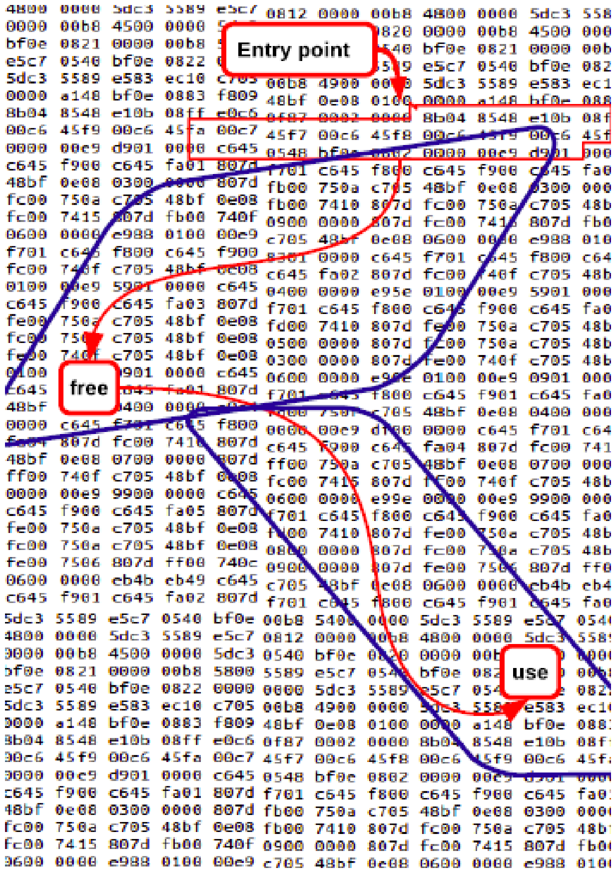
Use-after-free bugs
- Very hard to find
- Sequence of events
- DSE gets lost

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





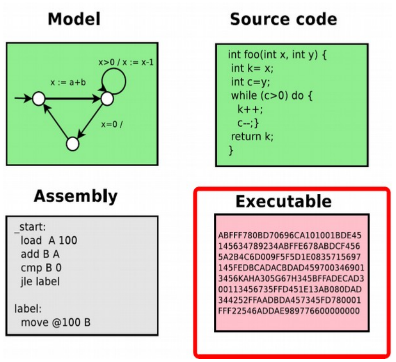


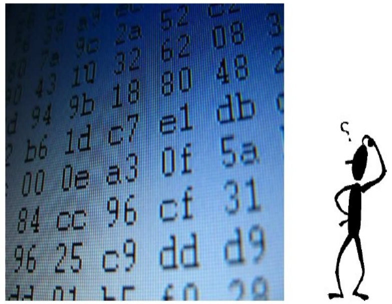
Guide SE with an unsound static analysis

OUTLINE

- **Shades of Symbolic Execution for Security**
  - **Standard usage**
  - **Robust symbolic execution (CAV 2018, 2021)**
  - **Relational symbolic execution (S&P 2020)**
  - **Haunted symbolic execution (NDSS 2021)**
  - **Adversarial symbolic execution (ESOP 2023)**

- **Problem : not all bugs are equal**





- Binary code
- Attacker
- Properties

Robust symbolic execution [CAV 2018, CAV 2021]

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

What?!!

Safety is not security …

```
int main () {
  int a = input ();
  int b = input ();

  int x = rand ();

  if (a * x + b > 0) {
    analyze_me();
  }
  else {
    ...
  }
}
```

- **for example here:**
  - SE will try to solve a * 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)

Robust symbolic execution [CAV 2018, CAV 2021]


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

What?!!

Safety is not security …

- **for example here:**
  - SE will try to solve a * x + b > 0
  - May return a = -100, b = 10, x = 0

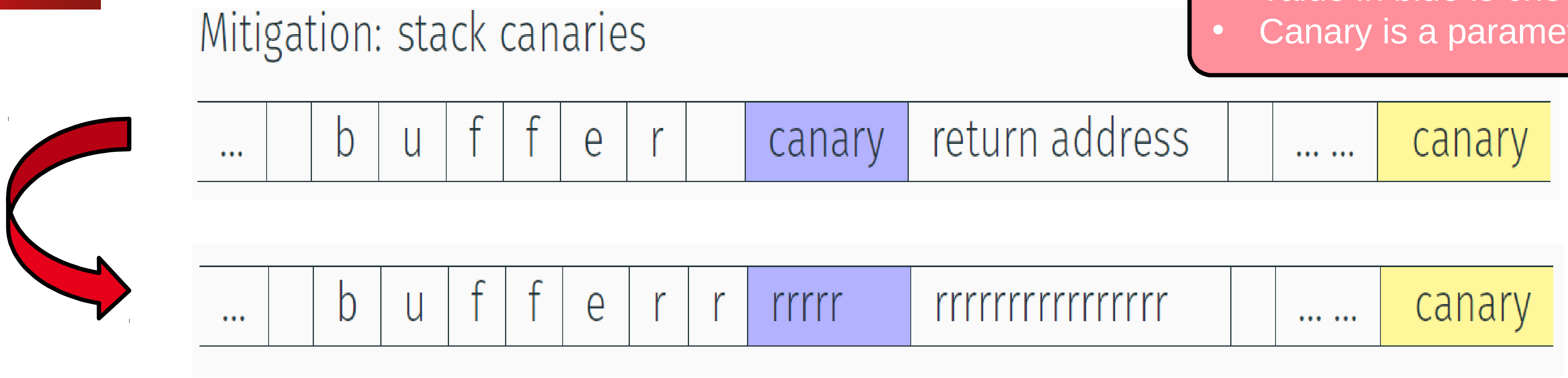
- **Problem: x is not controlled by the user**
  - If x change, possibly not a solution anymore
  - Example: (a = -100, b = 10, x = 1)

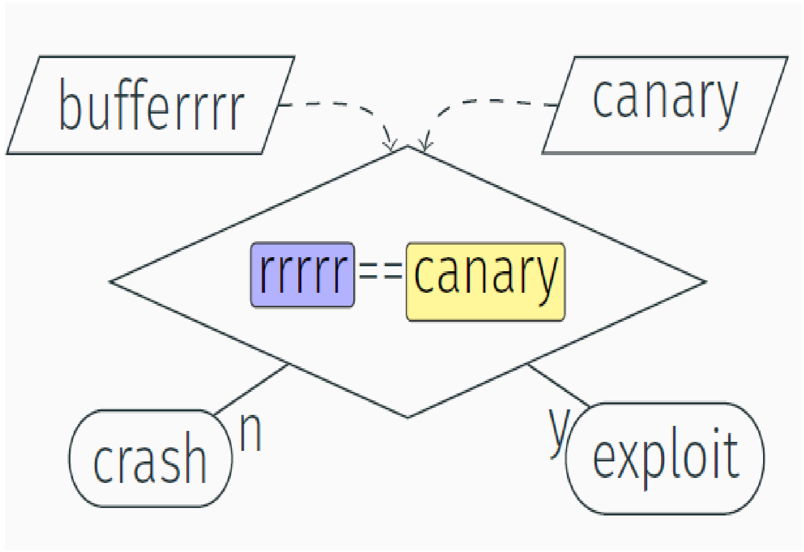
In practice: canaries, secret key in uninitialized memory, etc.

```
int main () {
  int a = input ();
  int b = input ();

  int x = rand ();

  if (a * x + b > 0) {
    analyze_me();
  }
  else {
    ...
  }
}
```

Problems with standard reachability?

- 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**

Real life false positives

Formally reachable, but
in reality, cannot be triggered **reliably**

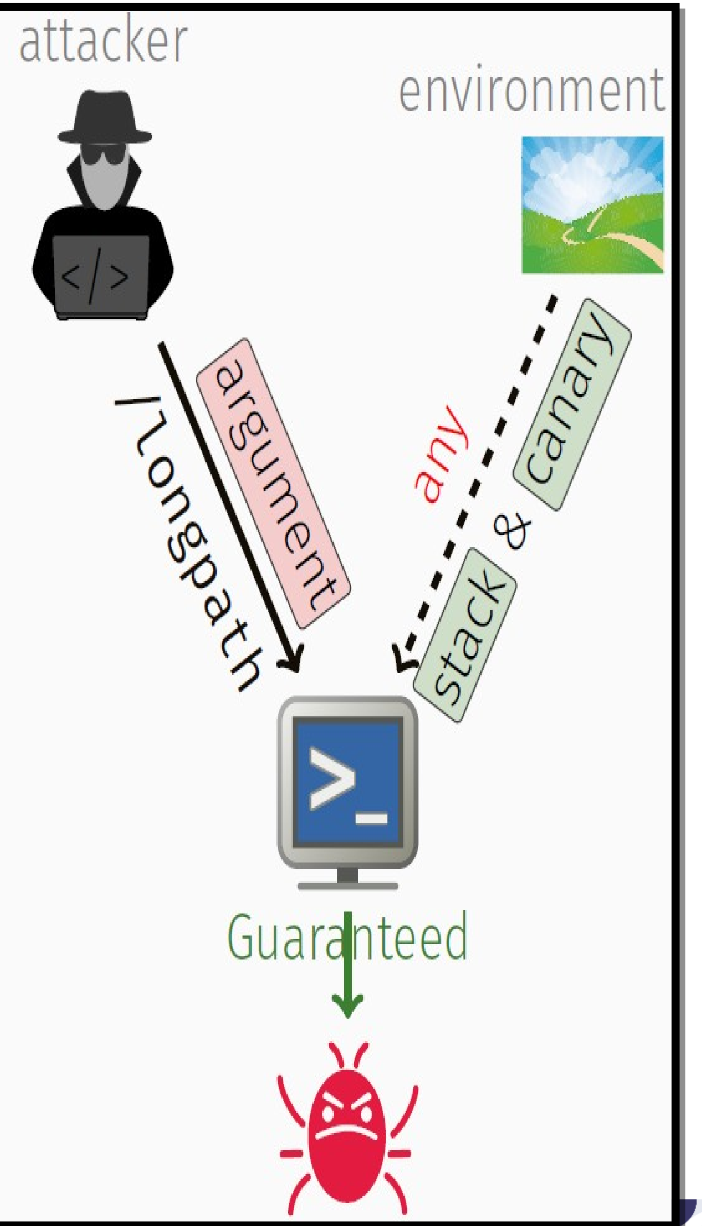
Our proposal [CAV 2018, CAV 2021, FMSD 2022]

Choose a threat Model

Partition input into controlled input a and uncontrolled input x

$(a, x) \vdash \ell$ means "with inputs a and x, the program executes code at ℓ"

Reachability of location ℓ	Robust Reachability of ℓ
$\exists a, x. (a, x) \vdash \ell$	$\exists a. \forall x. (a, x) \vdash \ell$



Adapting BMC and SE

Path merging

Optional in SE

Required for completeness in Robust SE

...and a few other differences

assume ψ: $\exists a.\forall x.\psi \Rightarrow \phi$ instead of $\exists a.\forall x.\psi \wedge \phi$

path pruning: no extra quantifier

concretization: only works on controlled values

$$\exists a.\forall X.\varphi \xrightarrow[X \text{ to } 90]{\text{concretize}} \exists a.\underbrace{\forall X.X = 90}_{\bot} \wedge \varphi$$

Proof-of-concept implementation

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

	BMC	SE	RBMC	RSE	RSE+path merging
Correct	22	30	32	37	44
False positive	**14**	**16**			
Inconclusive			1	7	
Resource exhaustion	10		13	2	2

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**

CVE-2019-19307 in Doas: beyond attacker-controlled input

Versatility of Robust Reachability

"Controlled inputs" are not limited to "controlled by the attacker"

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`"

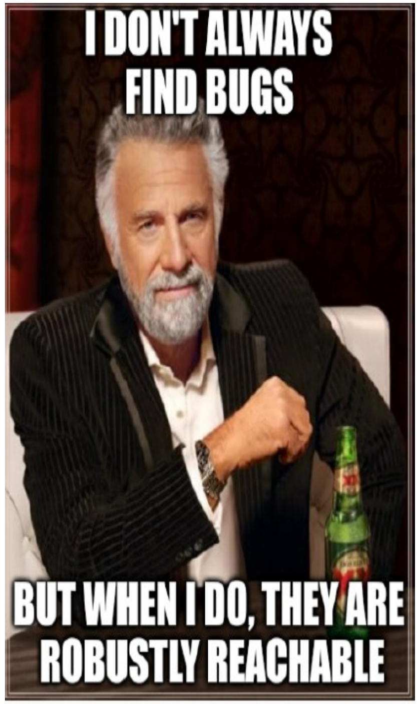
Stepping back

- **Robust reachability draws a line between some good bugs and bad bugs**
  - Based on replicability
  - Potential applications : better bug finding, bug priorization, test suite evaluation

- **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 reachbaility]
  - Fair model checking (Hart et al., 1983)

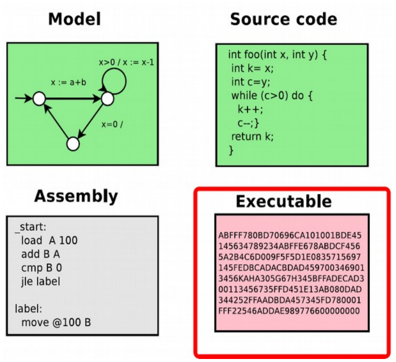
- **Qualitative « all or nothing » robust reachability may be too strong**
  - Mitigation : add user-defined constraints over the uncontrolled variables

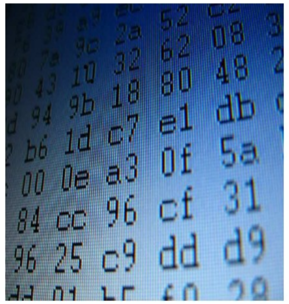


OUTLINE

- Shades of Symbolic Execution for Security
  - Standard usage
  - Robust symbolic execution (CAV 2018, 2021)
  - Relational symbolic execution (S&P 2020)
  - Haunted symbolic execution (NDSS 2021)
  - Adversarial symbolic execution (ESOP 2023)

- **Problem : some security properties are not mere safety**



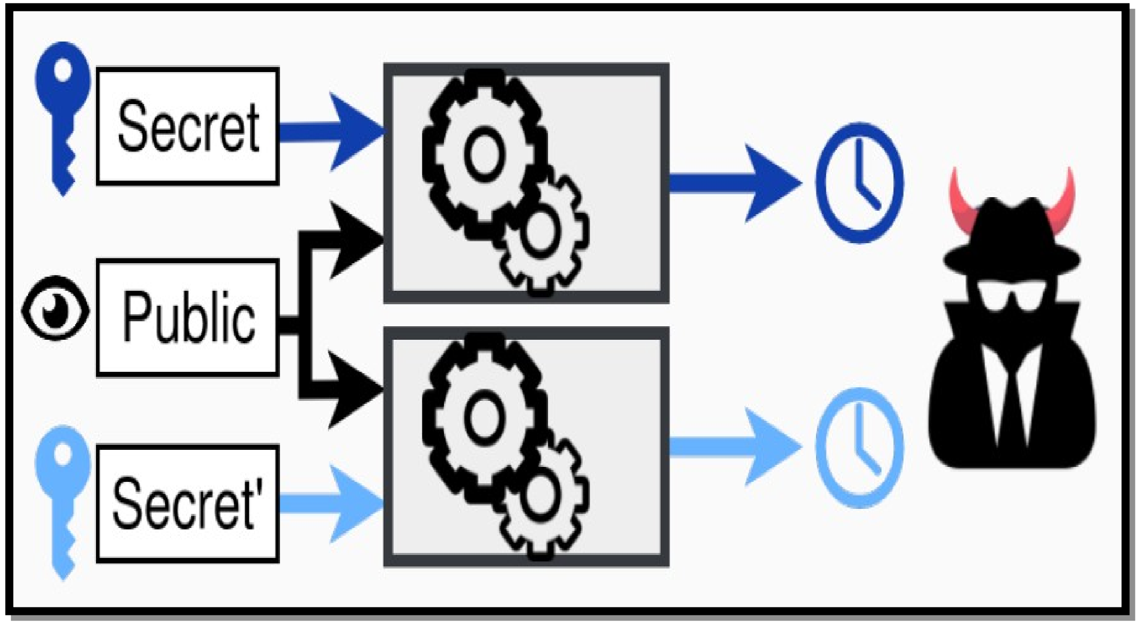


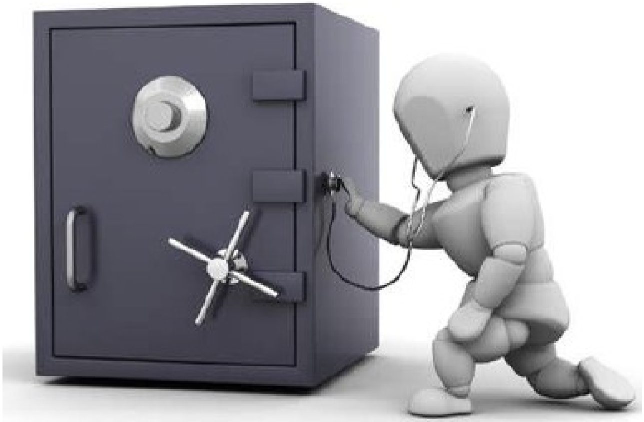
- Binary code
- Attacker
- Properties

« True » security properties (a.k.a. hyper-properties)

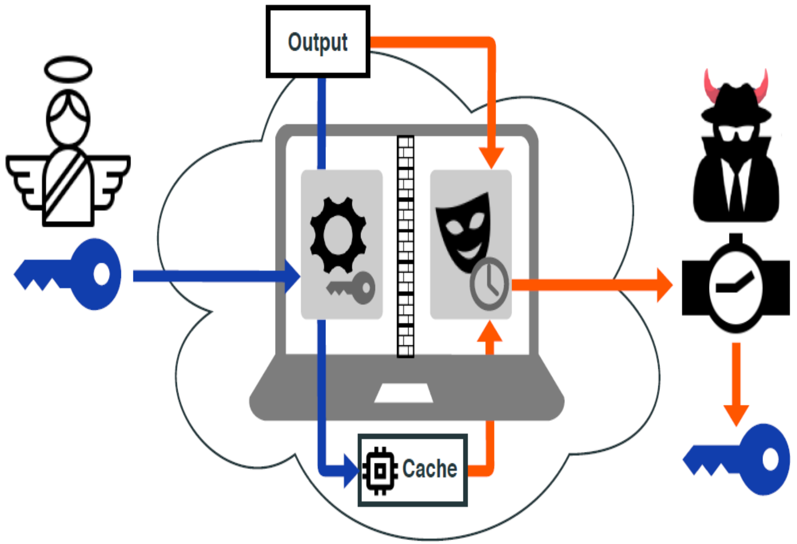
Information leakage

Properties over pairs of executions





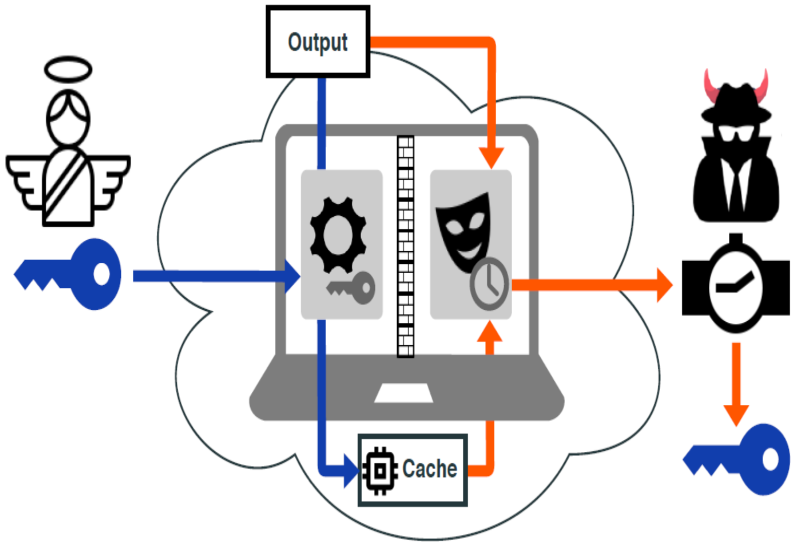
SECURING CRYPTO-PRIMITIVES -- [S&P 2020] (Lesly-Ann Daniel)



		#Instr static	#Instr unrol.	Time	CT source	Status		Comment
utility	ct-select	735	767	.29	Y	21×✗	21	1 new ✗
	ct-sort	3600	7513	13.3	Y	18×✗	44	2 new ✗
BearSSL	aes_big	375	873	1574	N	✗	32	-
	des_tab	365	10421	9.4	N	✗	8	-
OpenSSL	tls-remove-pad-lucky13	950	11372	2574	N	✗	5	-
Total		6025	30946	4172	-	**42** ×✗	110	-

- timing attacks
- cache attacks
- (secret-erasure)

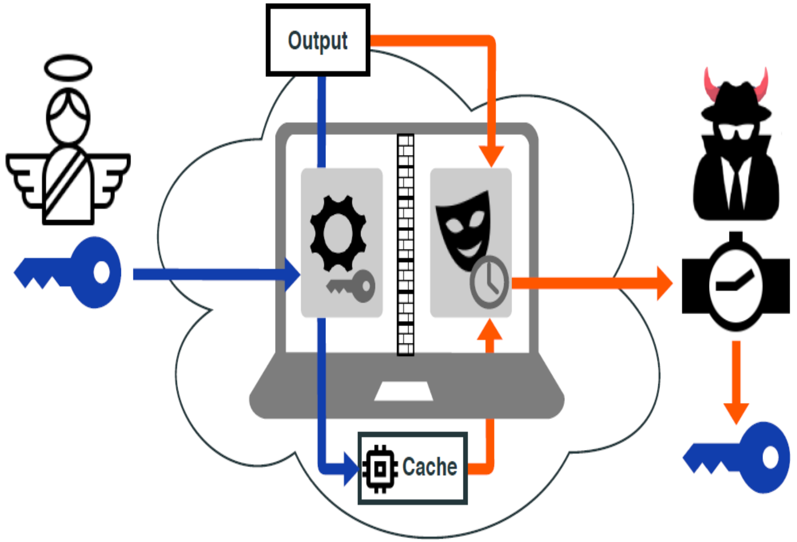
SECURING CRYPTO-PRIMITIVES -- [S&P 2020] (Lesly-Ann Daniel)



		#Instr static	#Instr unrol.	Time	CT source	Status	🐞	Comment
utility	ct-select	735	767	.29	**Y**	21×✗	21	1 new ✗
	ct-sort	3600	7513	13.3	**Y**	18×✗	44	2 new ✗
BearSSL	aes_big	375	873	1574	N	✗	32	-
	des_tab	365	10421	9.4	N	✗	8	-
OpenSSL	tls-remove-pad-lucky13	950	11372	2574	N	✗	5	-
Total		6025	30946	4172	-	**42** ×✗	110	-

- Relational symbolic execution
- Follows paires of execution
- Check for divergence

SECURING CRYPTO-PRIMITIVES -- [S&P 2020] (Lesly-Ann Daniel)



		#Instr static	#Instr unrol.	Time	CT source	Status		Comment
utility	ct-select	735	767	.29	Y	21×✗	21	1 new ✗
	ct-sort	3600	7513	13.3	Y	18×✗	44	2 new ✗
BearSSL	aes_big	375	873	1574	N	✗	32	-
	des_tab	365	10421	9.4	N	✗	8	-
OpenSSL	tls-remove-pad-lucky13	950	11372	2574	N	✗	5	-
Total		6025	30946	4172	-	**42** ×✗	110	-

- Relational symbolic execution
- Follows paires of execution
- Check for divergence
- Sharing, dedicated preprocessing

- 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 workl

Stepping back

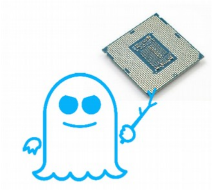
- **Symbolic execution efficient for simple but important relational problems**
  - constant time (different flavours)
  - secret erasure

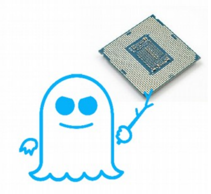
- **What about stronger relational properties ? [ex : non-interference, equivalence]**
  - The proposed method allows to find bugs
  - Main issue for generalization : quadratic number of pairs of paths

- **What about quantitative reasoning ? [QIF]**
  - Can try to use #SMT solvers, yet beware of scale / expressivity
  - Still the quadratic #pairs of paths problem

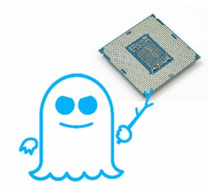
OUTLINE

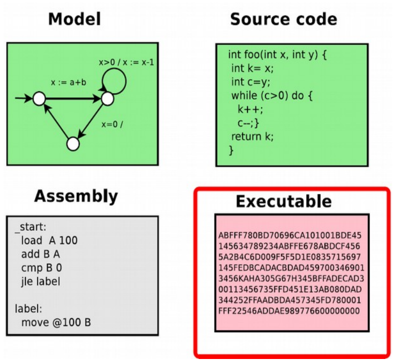
- **Shades of Symbolic Execution for Security**
  - **Standard usage**
  - **Robust symbolic execution (CAV 2018, 2021)**
  - **Relational symbolic execution (S&P 2020)**
  - **Haunted symbolic execution (NDSS 2021)**
  - **Adversarial symbolic execution (ESOP 2023)**

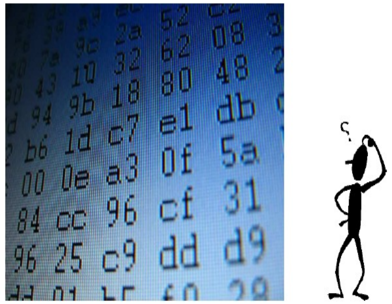


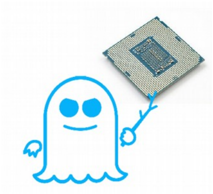


- **Problem : what if the attacker can observe more behaviours?**



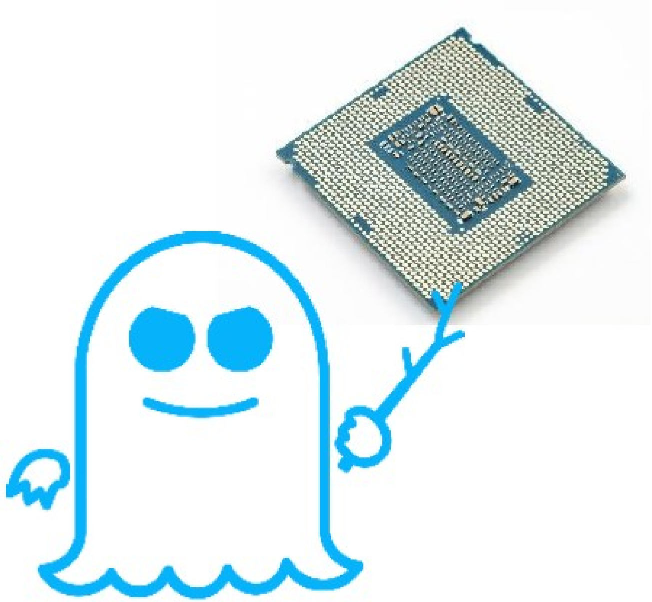






- Binary code
- Attacker
- Properties

Speculative executins and Spectre attacks



Spectre attacks (2018)

- Exploit speculative execution in processors
- Affect almost all processors
- Attackers can force mispeculations: transient executions
- Transient executions are reverted at architectural level
- But *not the microarchitectural state* (e.g. cache)

Challenge !

- Counter-intuitive semantics
- Path explosion:
  - **Spectre-STL**: all possible load/store interleavings !
- Needs to hold at binary-level

Path explosion for Spectre-STL on Litmus tests (**328** instr.)

Semantics	Paths
Sequential semantics	**14**
Speculative semantics (Spectre-STL)	**37M**

Challenge !

- Counter-intuitive semantics
- Path explosion:
  - **Spectre-STL**: all possible load/store interleavings !
- Needs to hold at binary-level

- Main idea :
- Smart encoding of speculation
- Can be seen as dedicated merge + targeted simplifications

Path explosion for Spectre-STL on Litmus tests (**328** instr.)

Semantics	Paths
Sequential semantics	**14**
Speculative semantics (Spectre-STL)	**37M**

Good first results, still some work :-)

	Target	Spectre-PHT	Spectre-STL
KLEESpectre [1]	LLVM	☺	-
SpecuSym [2]	LLVM	☺	-
FASS [3]	Binary	☹	-
Spectector [4]	Binary	☹	-
Pitchfork [5]	Binary	😐	☹
Binsec/Haunted	**Binary**	☺	😐

- Fun fact : spectre-pht protections may be vulnerable to spectre-stl

Stepping back

- **Some progress, but still a lot to do :-)**

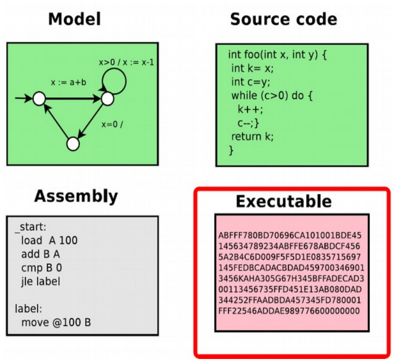
- **More and more sources of speculations**
  - Generic approach ? (cf Ponce de Leon et al.)
  - Link with micro-architecture people

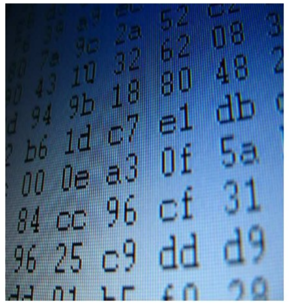
- **Criticity of the reported problems ?**

OUTLINE

- **Shades of Symbolic Execution for Security**
  - **Standard usage**
  - **Robust symbolic execution (CAV 2018, 2021)**
  - **Relational symbolic execution (S&P 2020)**
  - **Haunted symbolic execution (NDSS 2021)**
  - **Adversarial symbolic execution (ESOP 2023)**

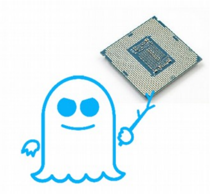
- **Problem : what about the attacker capabilities ?**

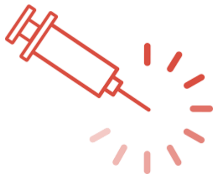


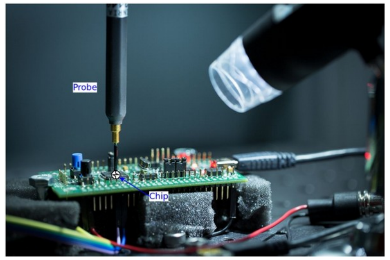


- Binary code
- Attacker
- Properties

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

Software-implemented hardware attacks

Electromagnetic pulses | Power glitch | Clock glitch | Laser beam | Faultline | DVFS

Race condition | Load Value Injection | Spectre | Rowhammer

Micro-architectural attacks

Man-At-The-End attacks

Context

- How to deal with that ?
- Principled ⇒ adversarial reachability
- Efficient ⇒ adversarial symbolic execution + optims

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

Hardware attacks

Software-implemented hardware attacks

Electromagnetic pulses	Power glitch	Clock glitch	Laser beam	Faultline	DVFS

Race condition	Load Value Injection	Spectre	Rowhammer

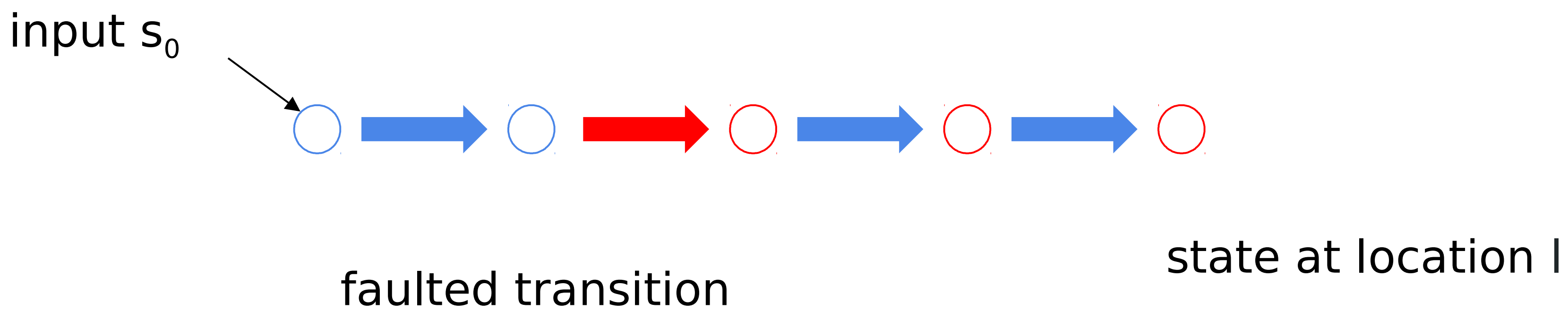
Micro-architectural attacks

Man-At-The-End attacks

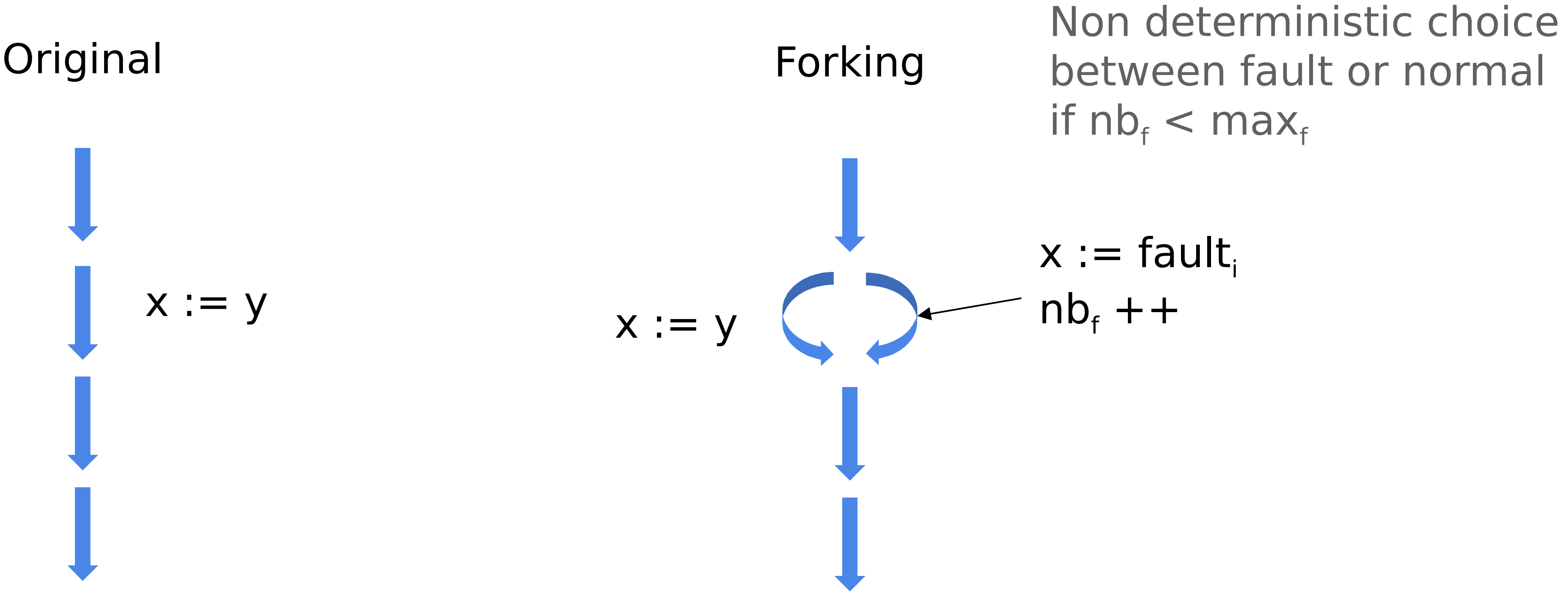
Adversarial reachability

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 adversarialy reachable in a program P for an attacker model A if $S_0 \mapsto^* l$, where $\mapsto^*$ is a succession of program instructions interleaved with faulty transitions.



Forking encodings



- ❑ Covers all adversarial behaviors
- ❑ Number of path exponential with # fault injection points

Forkless encodings and Adversarial Symbolic Execution

Original

x := y

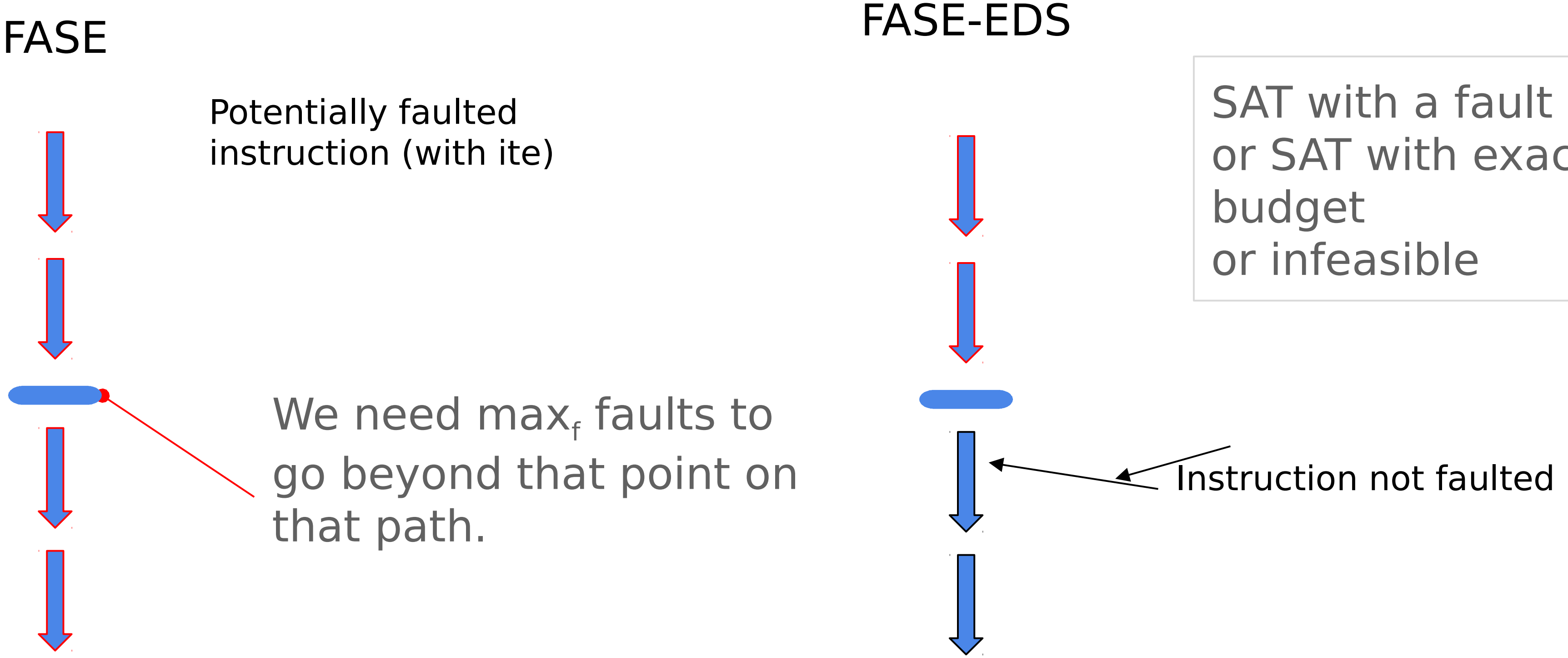
Forkless

x := ite $here_i$? $fault_i$: y

$here_i \in [0,1]$, $\Sigma\ here_i \leq max_f$

- ❑ Covers all adversarial behaviors
- ❑ **Only 1 path (cool!)**
- ❑ **More complex formulas (too many possible injection points)**

Early Detection of fault Saturation (EDS)



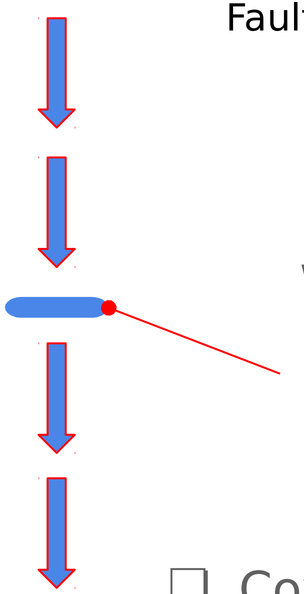
SAT with a fault margin
or SAT with exactly the fault budget
or infeasible

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

Injection On Demand (IOD)

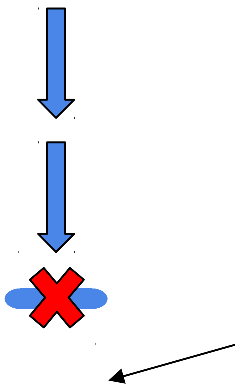
FASE

Faulted instruction



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

FASE-IOD

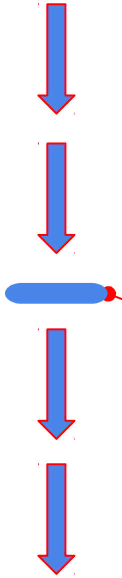


- ❑ Covers all adversarial behaviors, as complete as FASE
- ❑ Only 1 path
- ❑ Reduce number of fault injections
- ❑ Additional queries

Injection On Demand (IOD)

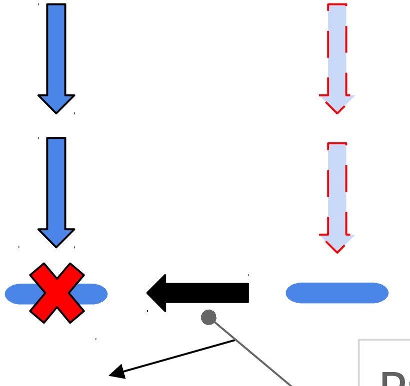
FASE

Faulted instruction



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

FASE-IOD



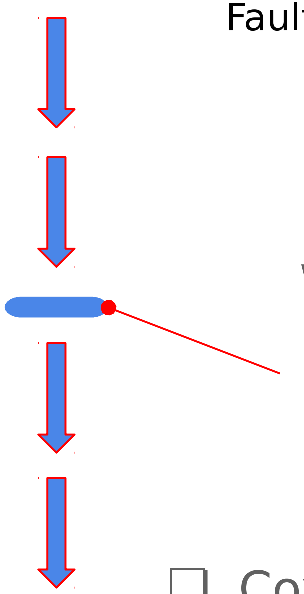
Path predicate switched for the faulted one

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

Injection On Demand (IOD)

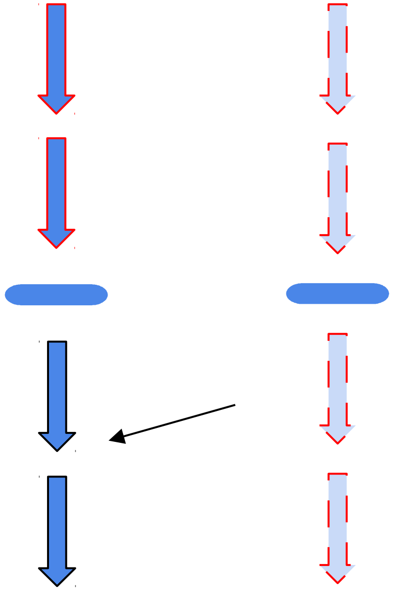
FASE

Faulted instruction



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

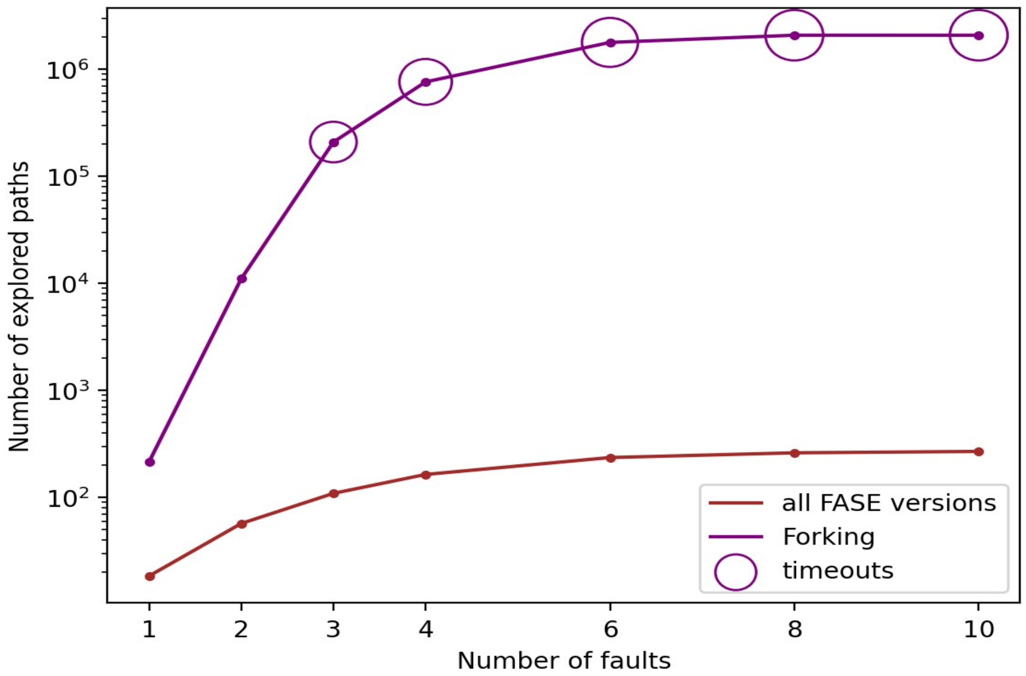
FASE-IOD

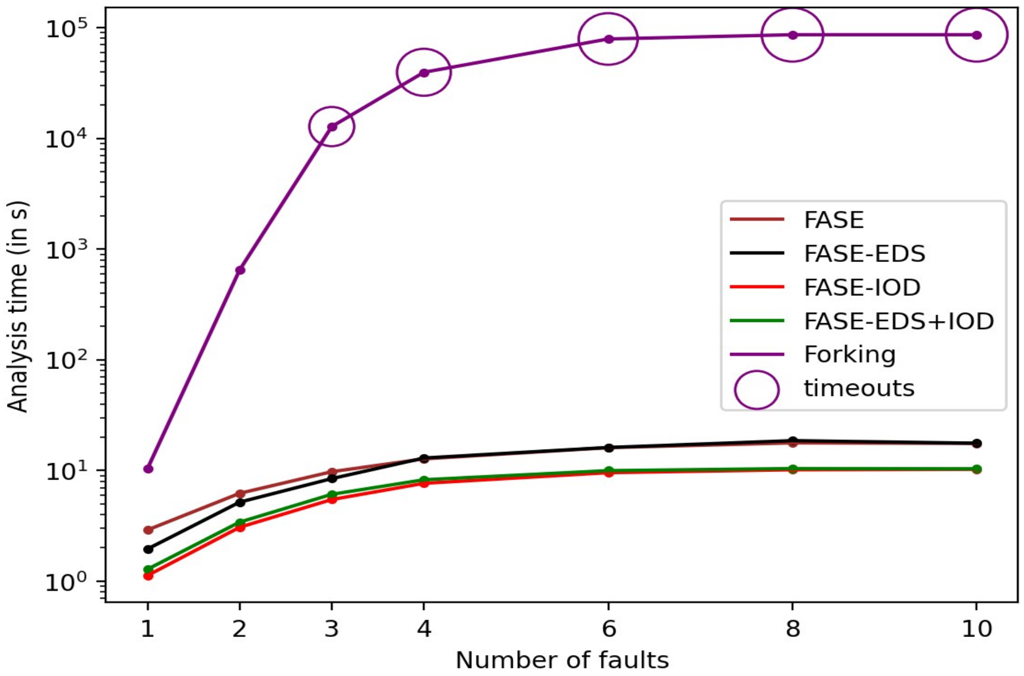


Bonus: under-approximation of nb_f

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

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 impementation [4] applied to VerifyPIN_0 → early loop exit attack with 1 arbitrary data fault or test inversion in valid CFG

[1] Puys, M., Riviere, L., Bringer, J., Le, T.h.: High-level simulation for multiple fault injection evaluation. In: Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance. Springer (2014)
[2] Dullien, T.: Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing (2017)
[3] de Ferrière, F.: Software countermeausres in the llvm risc-v compiler (2021), https://open-src-soc.org/2021-03/media/slides/3rd-RISC-V-Meeting-2021-03-30-15h00-Fran%C3%A7ois-de-Ferri%C3%A8re.pdf
[4] Lacombe, G., Feliot, D., Boespflug, E., Potet, M.L.: Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In: PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS) (2021)

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**

[1] Lacombe, G., Feliot, D., Boespflug, E., Potet, M.L.: Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In: PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS) (2021)
[2] Martin, T., Kosmatov, N., Prevosto, V.: Verifying redundant-check based countermeasures: a case study. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. (2022)

Stepping back

- Adversarial reachability takes an active attacker into account
- Well known in cryptographic protocol verification, not for code

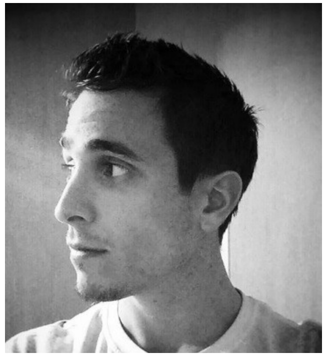
- generic: reachability, hyper-reachability, non termination

- Scalability ?
- Which capabilities for the attacker? [link with Hardware security community]
- Strong link with robust reachability

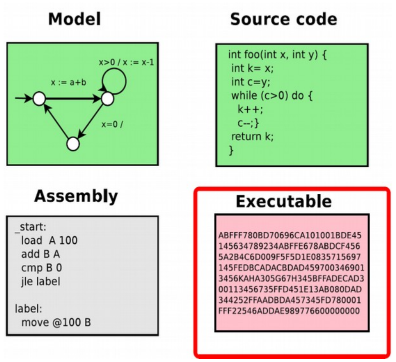
OUTLINE

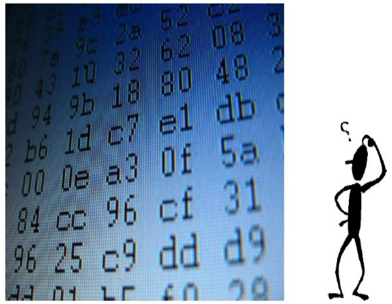
- **Shades of Symbolic Execution for Security**
  - **Standard usage**
  - **Robust symbolic execution (CAV 2018, 2021)**
  - **Relational symbolic execution (S&P 2020)**
  - **Haunted symbolic execution (NDSS 2021)**
  - **Adversarial symbolic execution (ESOP 2023)**

  - **Backward bounded symbolic execution (S&P 2017)**



- **Problem : sometimes the code itself is adversarial**

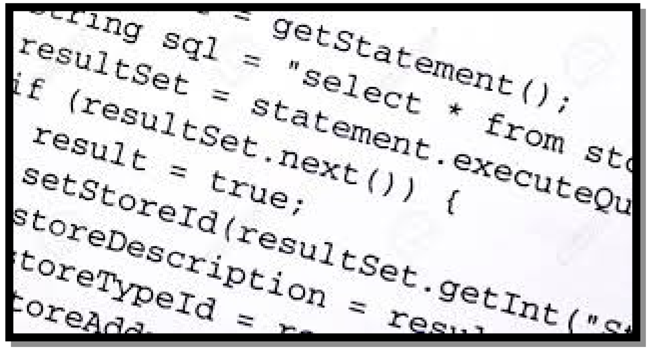


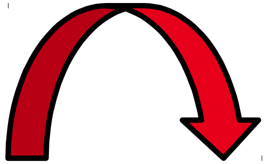


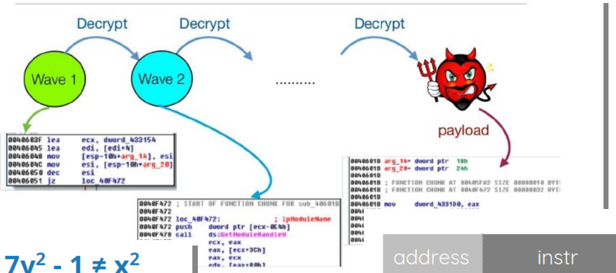
- Binary code
- Attacker
- Properties

CASE 2: code deobfuscation

- Adversarial code







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>
```

address	instr
80483d1	call +5
80483d6	pop edx
80483d7	add edx, 8
80483da	push edx
80483db	ret
80483dc	.byte{invalid}
80483de	[...]

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)

↧

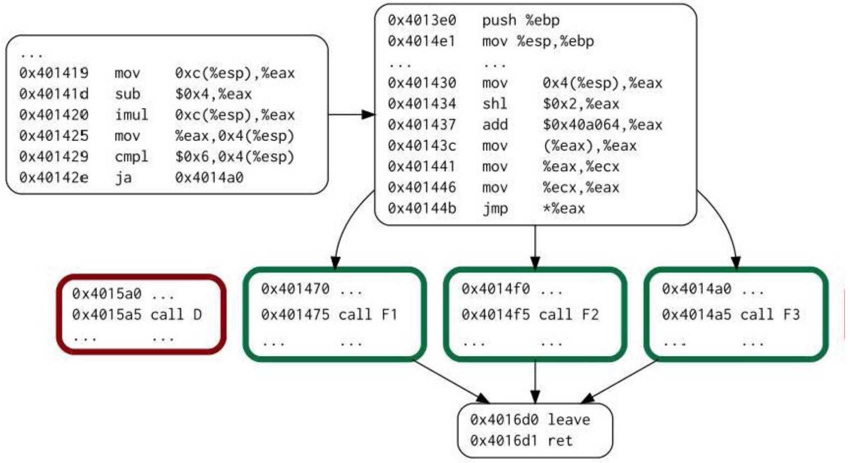
```
mov   eax, ds:X
mov   ecx, ds:Y
imul  ecx, ecx
imul  ecx, 7
sub   ecx, 1
imul  eax, eax
cmp   ecx, eax
jz    <dead_addr>
```

The predicate is always true

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

```
     OF := ((ax{31,31}≠bx{31,31}) &
              (ax{31,31}≠(ax-bx){31,31}));
     SF := (ax-bx) < 0;
     ZF := (ax-bx) = 0;
     if (¬ ZF ∧ (OF = SF)) goto l1
     X := 1
     goto l2
l1:  X := -1
l2:
```

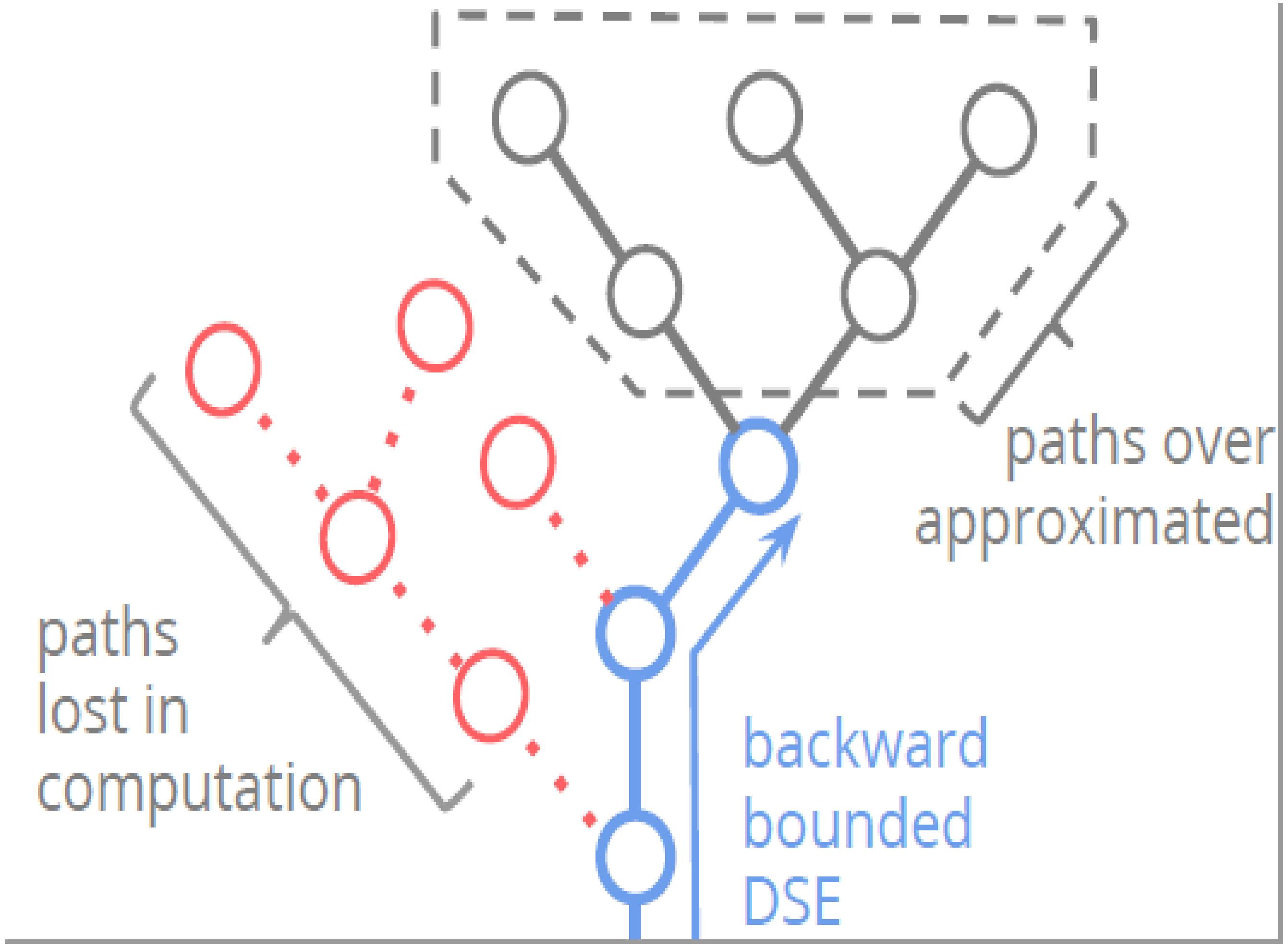
The two blocks are equivalent



With IDA + BINSEC

All jump targets are found

BACKWARD-BOUNDED DSE [S&P 2017] (with Robin David)



Backward bounded SE

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

- Prove things
- Local => scalable

Case : 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: 12min]

- Backward-bounded SE
- + dynamic analysis

	C637 Sample #1	99B4 Sample #2
#total instruction	**505,008**	**434,143**
#alive	+279,483	+241,177

Stepping back

- **Backward Bounded SE do allow proof and is scalable**

- **An attacker can try to evade it with delaying computation**
  - More advanced notions of bound

- **Can be used in other contexts than adversarial code analysis**
  - Local assertion proofs
  - Local finding of dynamic jumps

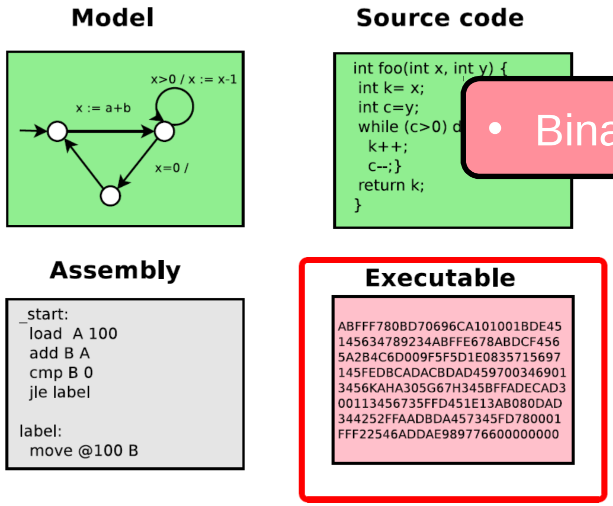
OUTLINE

- Introduction
- What every honest person should know about Symbolic Execution
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

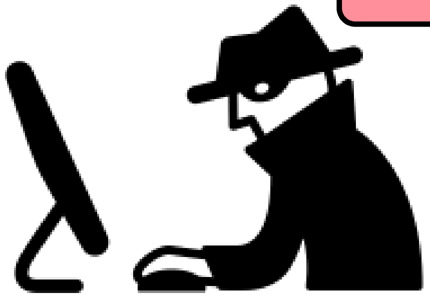
OUTLINE

- Introduction
- What every honest person should know about Symbolic Execution
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

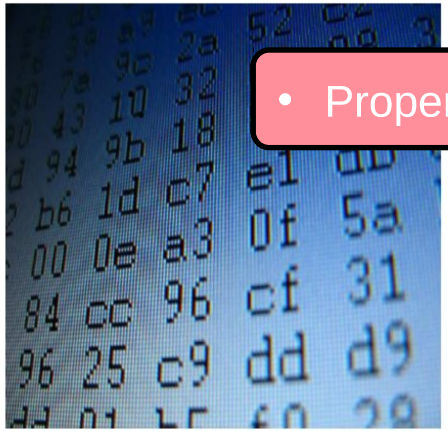
Safety is not security, fun new problems



- Binary code



- Attacker



- Properties

SOME KEY PRINCIPLES BEHIND OUR WORK?

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

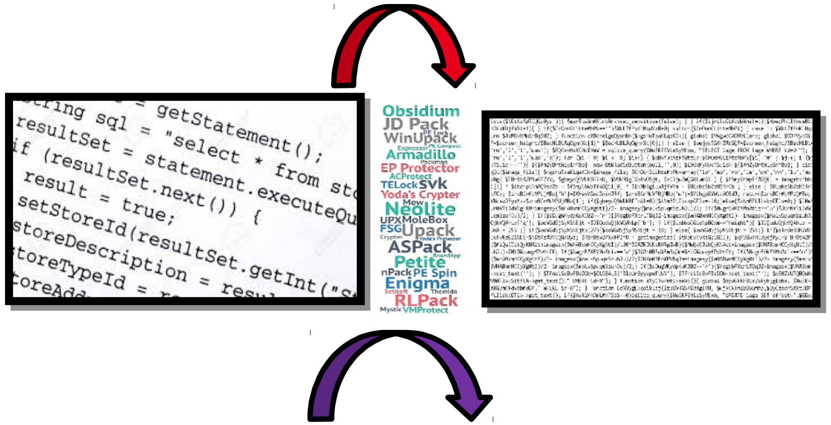
- **Can be adapted beyond the basic reachability case**
  - variants (backward, relational, robust, etc.)
  - combination with other techniques

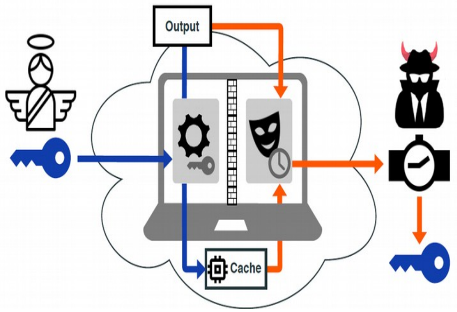
- **Finely tune the technology**
  - Tools for safety are not fully adequate for security
  - Dedicated preprocessing
  - Dedicated merging

Under the hood: finely tune the technology

- **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

Do it with style!

Malware deobfuscation Backward bounded SE	X-Tunnel 400k instrs → 40% junk	2017 1h30	2022 12 min
Constant time verification Semi-relational SE	13 well-known crypto primitives from OpenSSL, BearSSL, etc.	2020 3h + 2 TO	2022 3 min
ANSSI challenges Smart path merging, faster memory reasoning	souk : 2^{71} paths unicorn : 10^{9} instrs	TO 3h	30 s 30 min
Test suite extension Incremental concolic engine	Cyber Grand Challenge from 1 to 14 seeds Coverage : 437 → 2769	August 45 min	November 2 min

Conclusion

Fun for FM/PL researchers

Benefit system security

- I love Symbolic Execution : it is formal & it works :-)

- Security is not safety
  - Binary level, true security properties, important bugs, attacker model, etc.

- Still, Symbolic Execution is flexible enough to accomodate that
  - New exciting theoretical questions
  - Complicated algorithmic issues (push solvers to their edges)
  - Promising applications

- Some results in that direction, still many exciting challenges

BINSEC is available

https://binsec.github.io

- We are hiring !
- Many open postdoc / PhD positions

sebastien.bardin@cea.fr

THANK YOU

We hope you enjoyed the journey