Toward
Security-Oriented Program Analysis

Sébastien Bardin
(CEA LIST, University Paris Saclay)

Joint work with the BINSEC group @ CEA
and many other collaborators

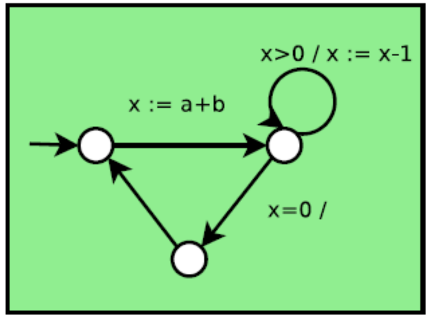
CODE-LEVEL SECURITY ANALYSIS

~~crypto~~ ~~protocols~~

Most attacks come from implementation bugs

A (binary-level) program analysis issue!

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
```

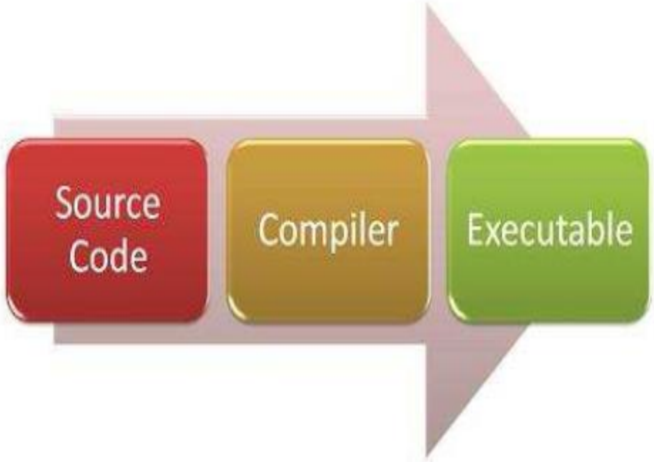
Executable

```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE98977660000000
```

WHY ON BINARY CODE?

No source code

Post-compilation

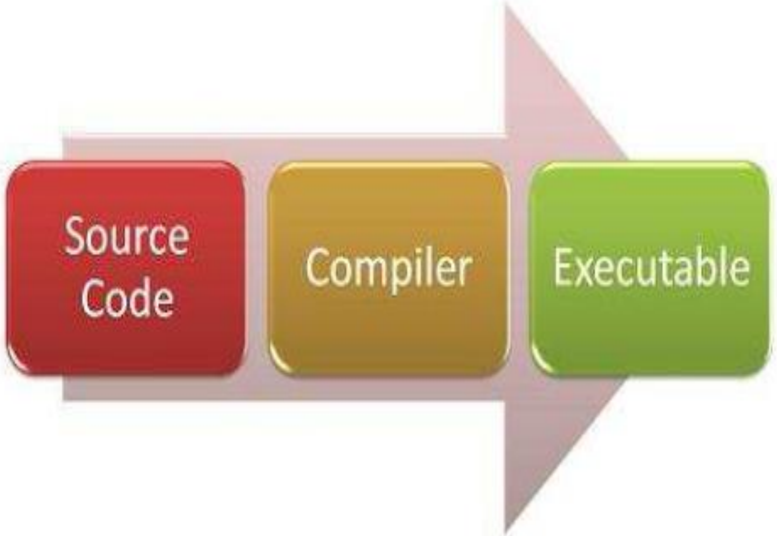


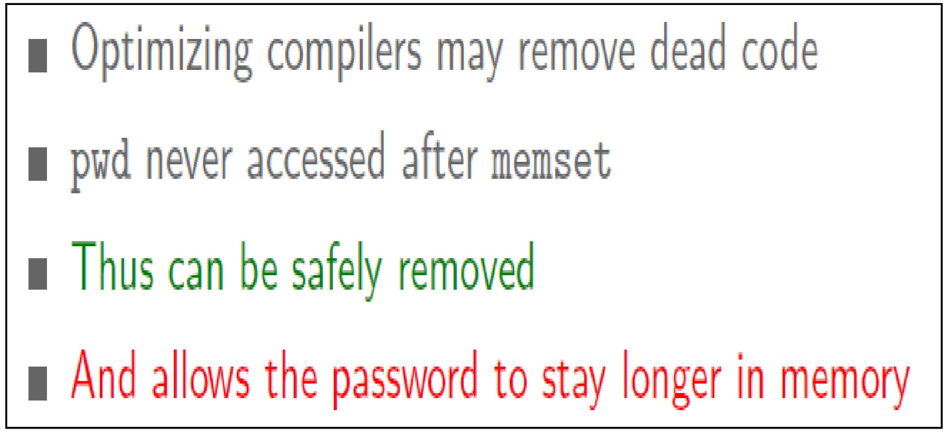
Malware comprehension

Protection evaluation

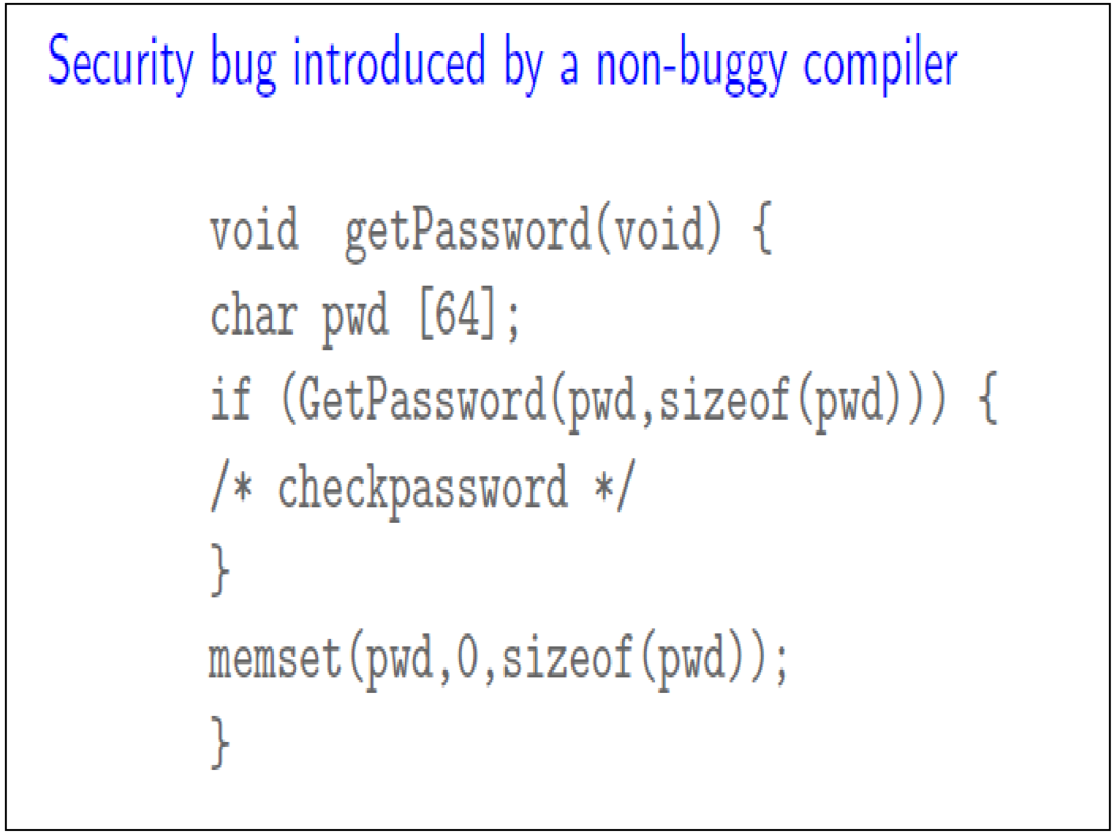
Very-low level reasoning

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



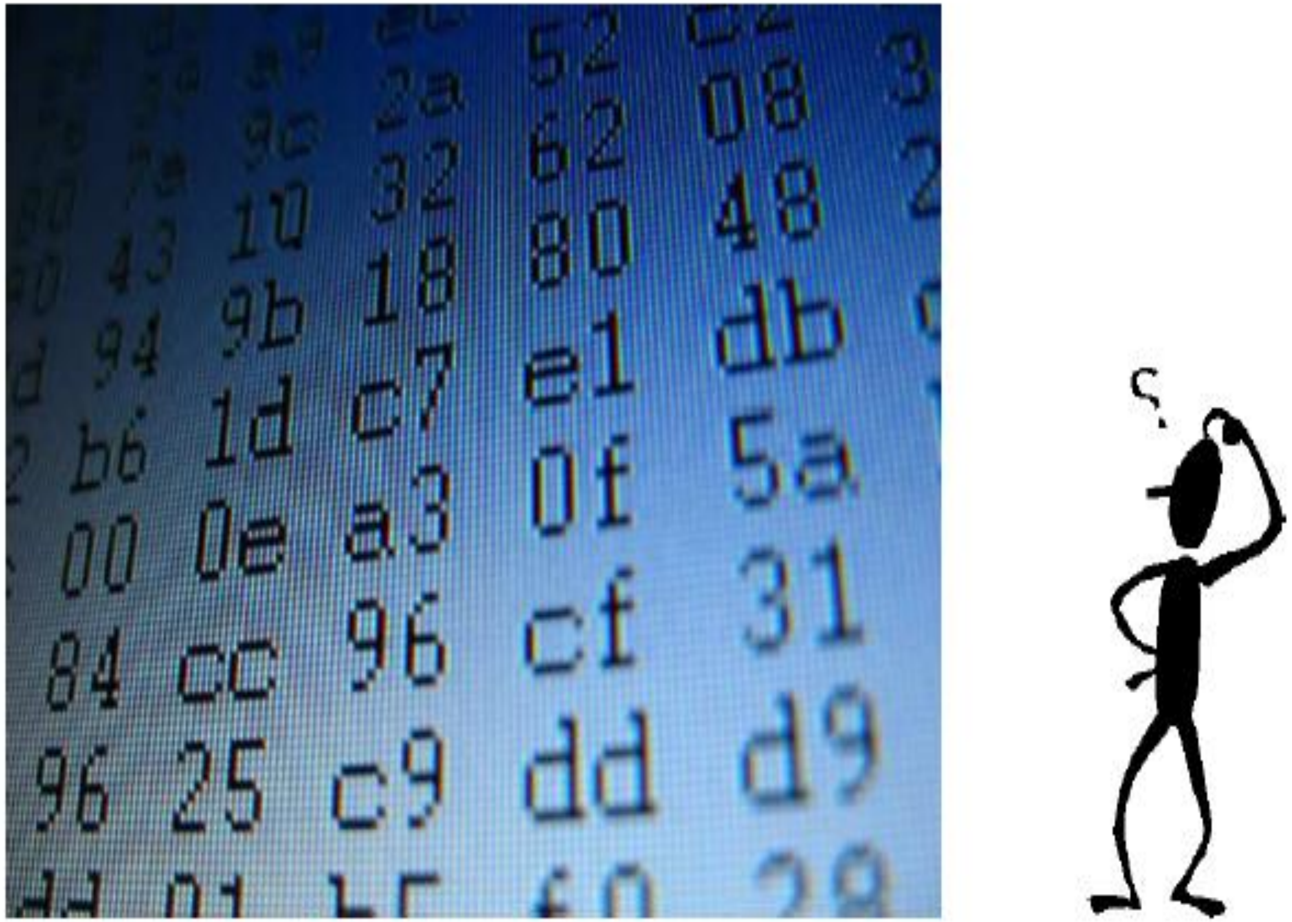
Security bug introduced by a non-buggy compiler

```
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

BINARY-LEVEL CODE ANALYSIS HAS MANY ADVANTAGES, BUT …



IN A NUTSHELL

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

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

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

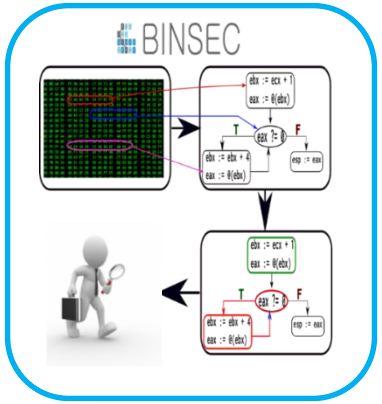
- ***Question**: how does code-level security differ from code-level safety?*

BINSEC: brings formal methods to binary-level security analysis

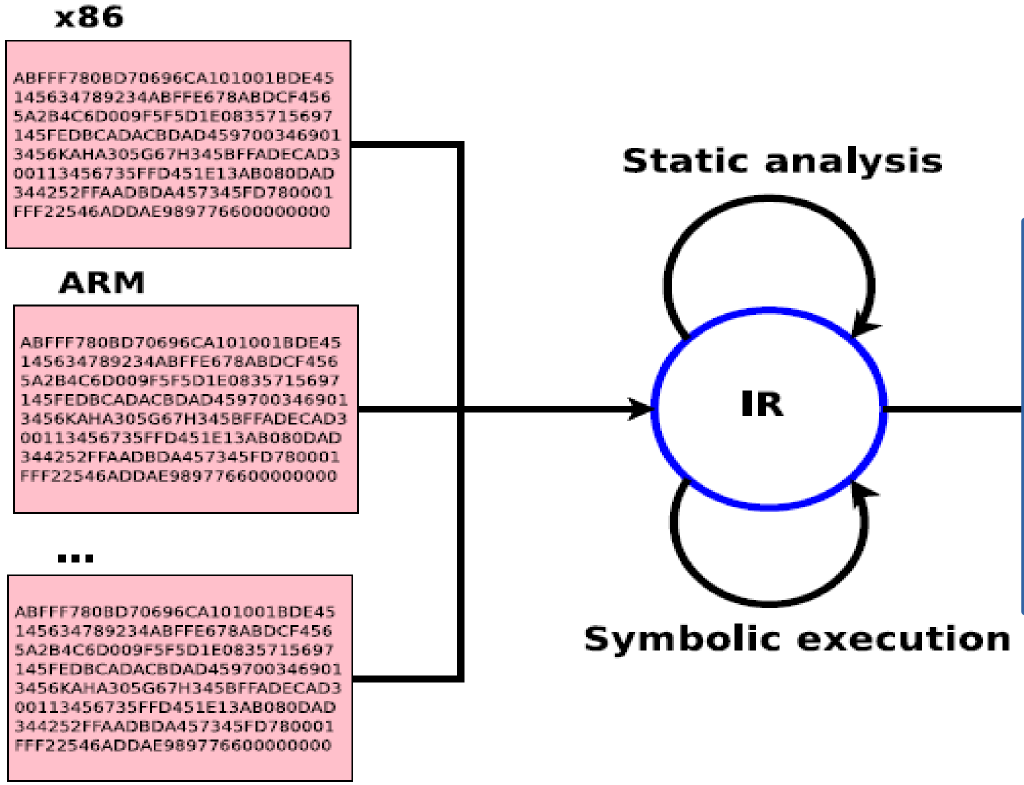
Break

Prove

Protect



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



x86

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

ARM

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

...

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

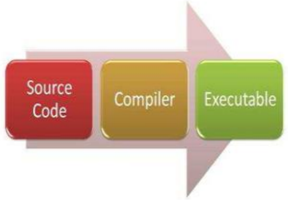
Static analysis

IR

Symbolic execution

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

COTS



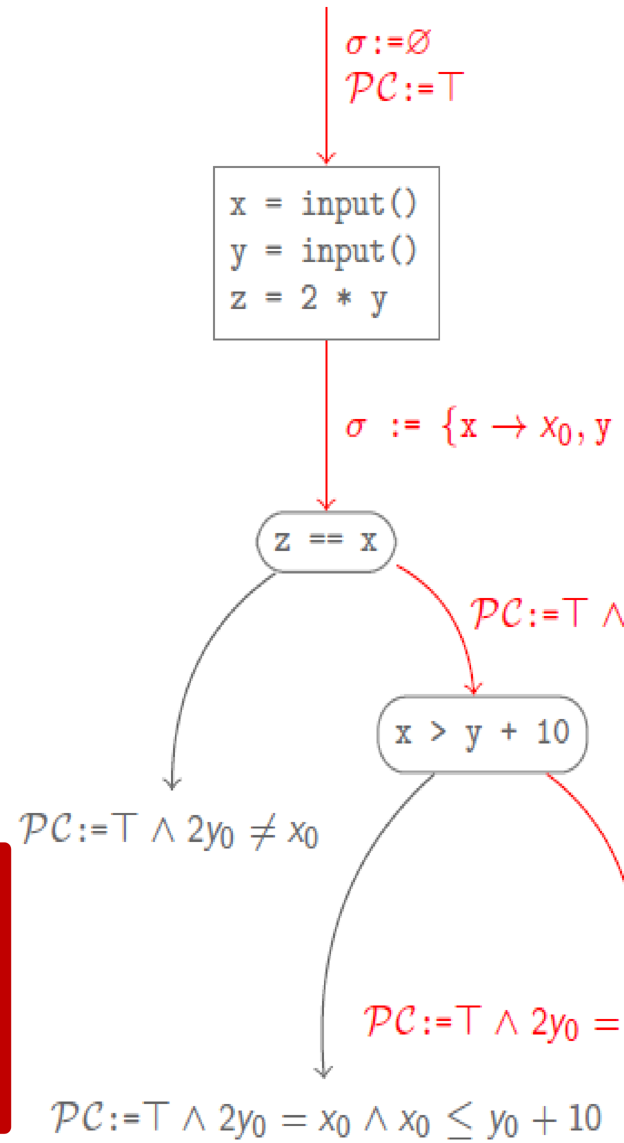
A TOOL OF CHOICE: SYMBOLIC EXECUTION (Godefroid 2005) [variants, optimizations]

Find real bugs

Bounded verification

Robust

```
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

OUTLINE

- Prologue: a little bit of formal methods for safety
- Binary-level security analysis: benefits & challenges
- The BINSEC platform
- From source-level safety to binary-level security: some examples
- Conclusion

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

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

- Reason about infinite sets of behaviours

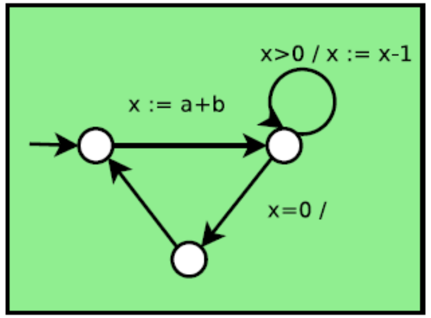
Success in (regulated) safety-critical domains

OUTLINE

- Prologue: a little bit of formal methods for safety
- Binary-level security analysis: benefits & challenges
- The BINSEC platform
- From source-level safety to binary-level security: some examples
- Conclusion

NOW: MOVING TO BINARY-LEVEL SECURITY ANALYSIS

Model



Source code

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

Assembly

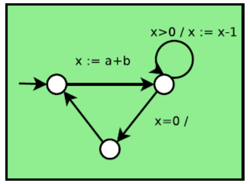
```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

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

NOW: MOVING 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
```

- Binary code

- Attacker

- Properties

OUTLINE

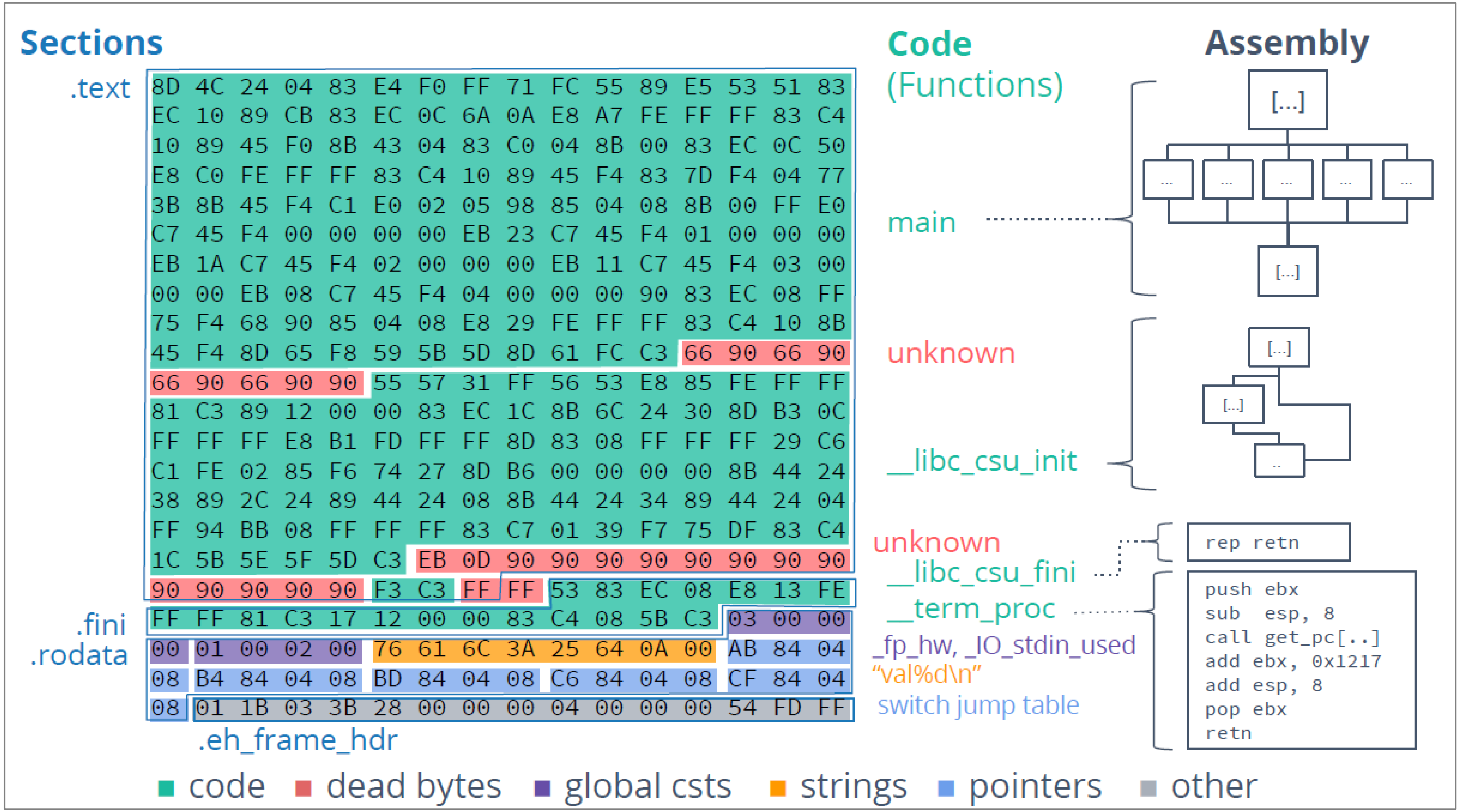
- **Prologue: a little bit of formal methods for safety**
- **Binary-level security analysis: benefits & challenges**
  - Going down to binary
  - Adversarial setting
  - « True security » properties
- **The BINSEC platform**
- **From source-level safety to binary-level security: some examples**
- **Conclusion**

CHALLENGE: BINARY CODE LACKS STRUCTURE

- Instructions?
- Control flow?
- Memory structure?

DISASSEMBLY IS ALREADY TRICKY!

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

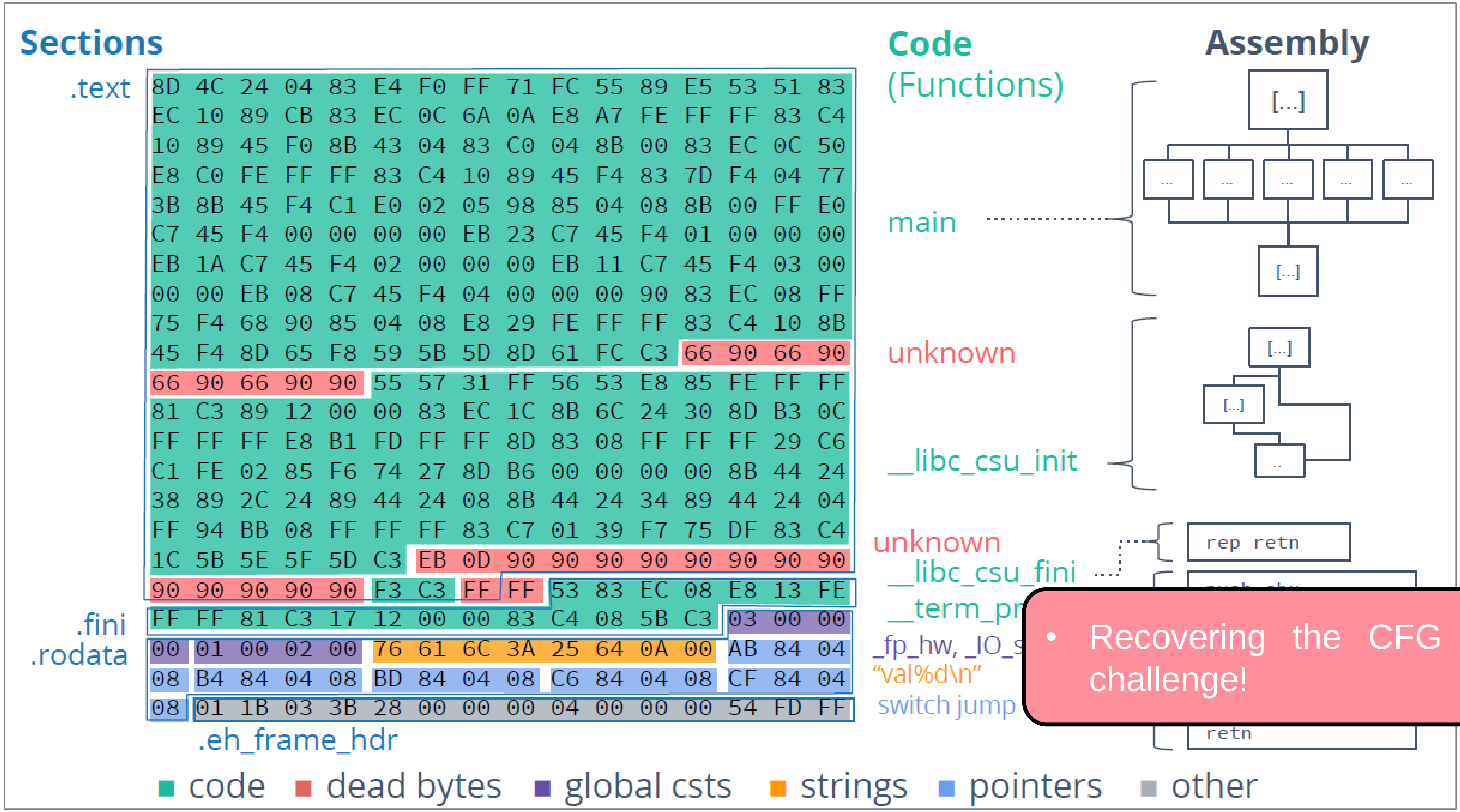


```
8D 4C 24 04 83 E4 F0 FF 71 FC 55 89 E5 53 51 83
EC 10 89 CB 83 EC 0C 6A 0A E8 A7 FE FF FF 83 C4
10 89 45 F0 8B 43 04 83 C0 04 8B 00 83 EC 0C 50
E8 C0 FE FF FF 83 C4 10 89 45 F4 83 7D F4 04 77
3B 8B 45 F4 C1 E0 02 05 98 85 04 08 8B 00 FF E0
C7 45 F4 00 00 00 00 EB 23 C7 45 F4 01 00 00 00
EB 1A C7 45 F4 02 00 00 00 EB 11 C7 45 F4 03 00
00 00 EB 08 C7 45 F4 04 00 00 00 90 83 EC 08 FF
75 F4 68 90 85 04 08 E8 29 FE FF FF 83 C4 10 8B
45 F4 8D 65 F8 59 5B 5D 8D 61 FC C3 66 90 66 90
66 90 66 90 90 55 57 31 FF 56 53 E8 85 FE FF FF
81 C3 89 12 00 00 83 EC 1C 8B 6C 24 30 8D B3 0C
FF FF FF E8 B1 FD FF FF 8D 83 08 FF FF FF 29 C6
C1 FE 02 85 F6 74 27 8D B6 00 00 00 00 8B 44 24
38 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
FF 94 BB 08 FF FF FF 83 C7 01 39 F7 75 DF 83 C4
1C 5B 5E 5F 5D C3 EB 0D 90 90 90 90 90 90 90 90
90 90 90 90 90 F3 C3 FF FF 53 83 EC 08 E8 13 FE
FF FF 81 C3 17 12 00 00 83 C4 08 5B C3 03 00 00
00 01 00 02 00 76 61 6C 3A 25 64 0A 00 AB 84 04
08 B4 84 04 08 BD 84 04 08 C6 84 04 08 CF 84 04
08 01 1B 03 3B 28 00 00 00 04 00 00 00 54 FD FF
```

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

DISASSEMBLY IS ALREADY TRICKY!

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

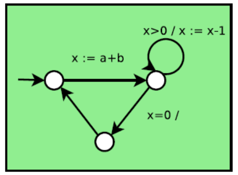


```
8D 4C 24 04 83 E4 F0 FF 71 FC 55 89 E5 53 51 83
EC 10 89 CB 83 EC 0C 6A 0A E8 A7 FE FF FF 83 C4
10 89 45 F0 8B 43 04 83 C0 04 8B 00 83 EC 0C 50
E8 C0 FE FF FF 83 C4 10 89 45 F4 83 7D F4 04 77
3B 8B 45 F4 C1 E0 02 05 98 85 04 08 8B 00 FF E0
C7 45 F4 00 00 00 00 EB 23 C7 45 F4 01 00 00 00
EB 1A C7 45 F4 02 00 00 00 EB 11 C7 45 F4 03 00
00 00 EB 08 C7 45 F4 04 00 00 00 90 83 EC 08 FF
75 F4 68 90 85 04 08 E8 29 FE FF FF 83 C4 10 8B
45 F4 8D 65 F8 59 5B 5D 8D 61 FC C3 66 90 66 90
66 90 66 90 90 55 57 31 FF 56 53 E8 85 FE FF FF
81 C3 89 12 00 00 83 EC 1C 8B 6C 24 30 8D B3 0C
FF FF FF E8 B1 FD FF FF 8D 83 08 FF FF FF 29 C6
C1 FE 02 85 F6 74 27 8D B6 00 00 00 00 8B 44 24
38 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
FF 94 BB 08 FF FF FF 83 C7 01 39 F7 75 DF 83 C4
1C 5B 5E 5F 5D C3 EB 0D 90 90 90 90 90 90 90 90
90 90 90 90 90 F3 C3 FF FF 53 83 EC 08 E8 13 FE
FF FF 81 C3 17 12 00 00 83 C4 08 5B C3 03 00 00
00 01 00 02 00 76 61 6C 3A 25 64 0A 00 AB 84 04
08 B4 84 04 08 BD 84 04 08 C6 84 04 08 CF 84 04
08 01 1B 03 3B 28 00 00 00 04 00 00 00 54 FD FF
```

- Recovering the CFG is already a challenge!

BINARY-LEVEL 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

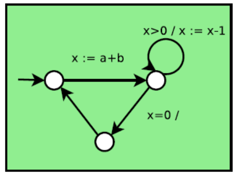
```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

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

Machine codes are complex

BINARY-LEVEL 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
```

Break an implicit assumption in code analysis

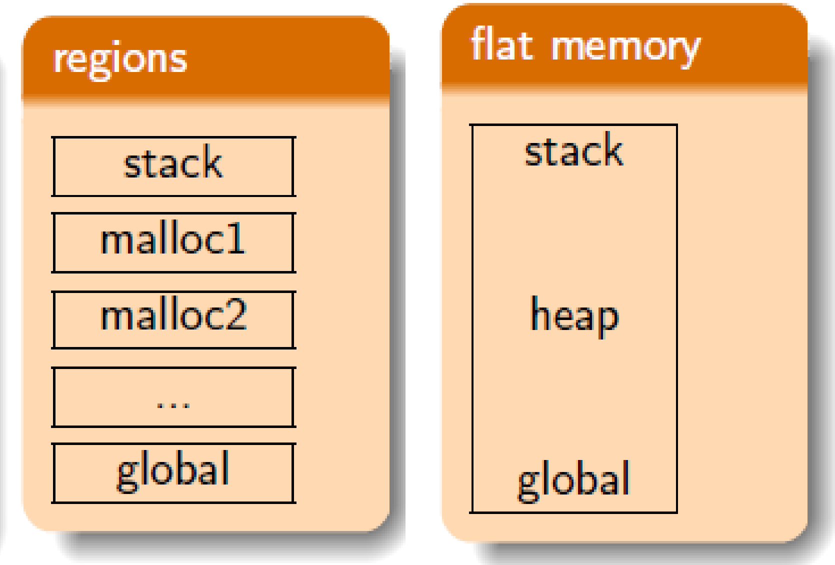
- **Low-level control (CFG?)**
- **Low-level data & memory**

At the edge of current methods

- Solved problem
- IR

Machine codes are complex

BINARY CODE SEMANTIC LACKS STRUCTURE

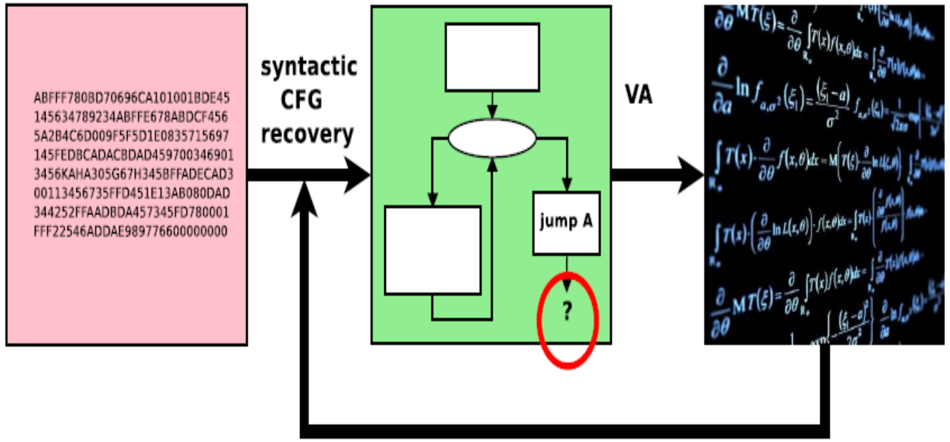


regions

- stack
- malloc1
- malloc2
- ...
- global

flat memory

- stack
- heap
- global



```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE98977660000000
```

```
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

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:
```

OUTLINE

- **Context: a little bit of formal methods for safety**

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

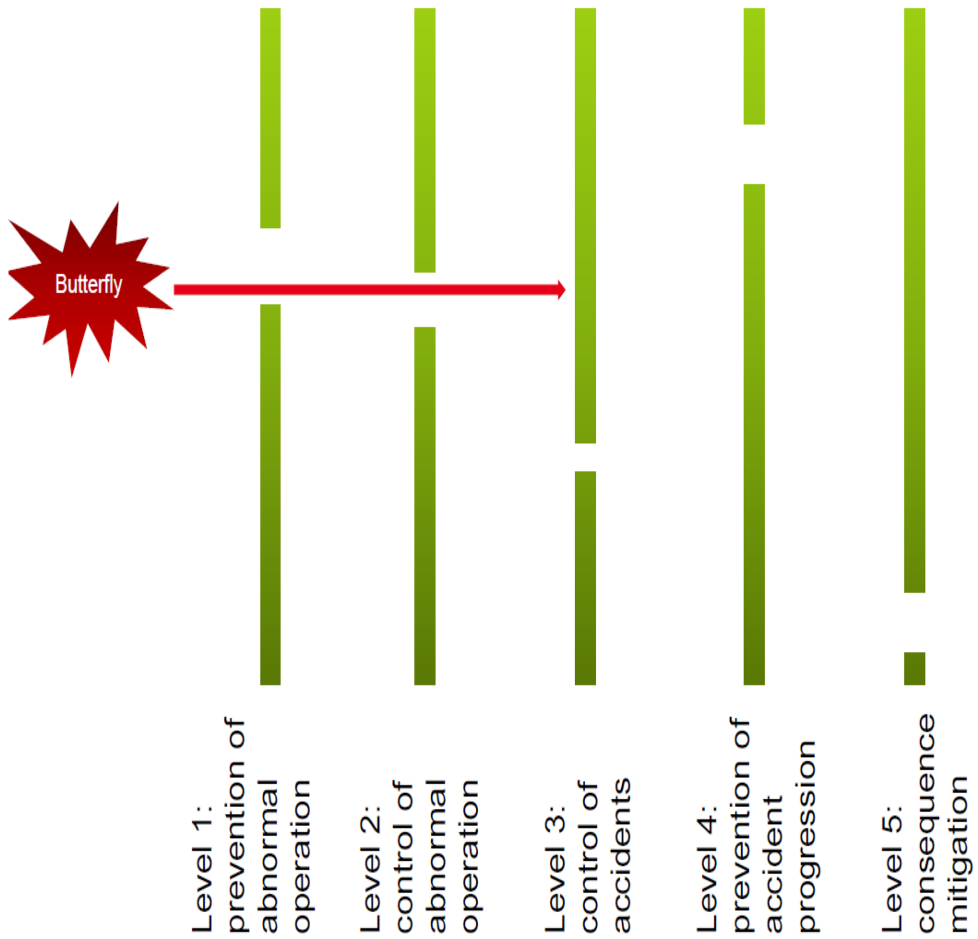
- **The BINSEC platform**

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

- **Conclusion**

CHALLENGE: ATTACKER

Nature is not nice



Attacker is evil

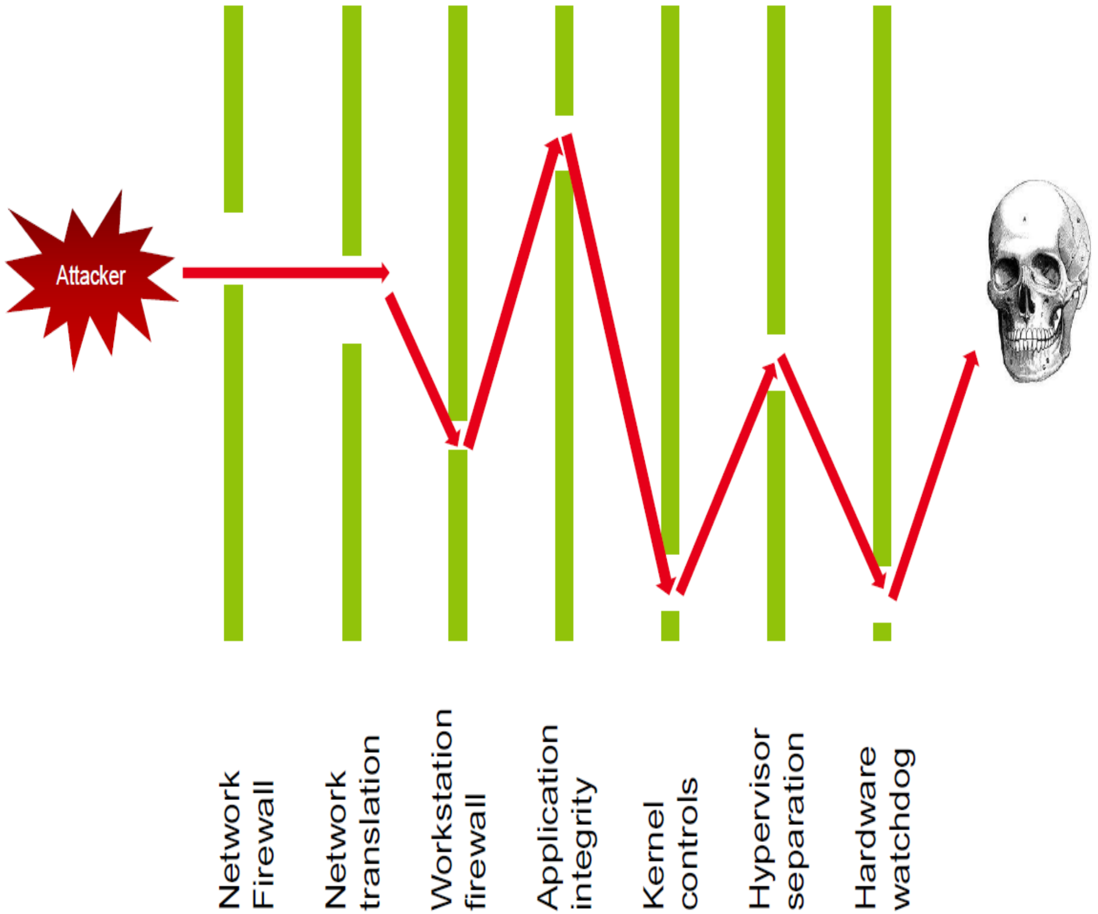


Image by Florent Kirchner

ATTACKER in Standard Program Analysis

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

ATTACKER in Standard Program Analysis

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

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

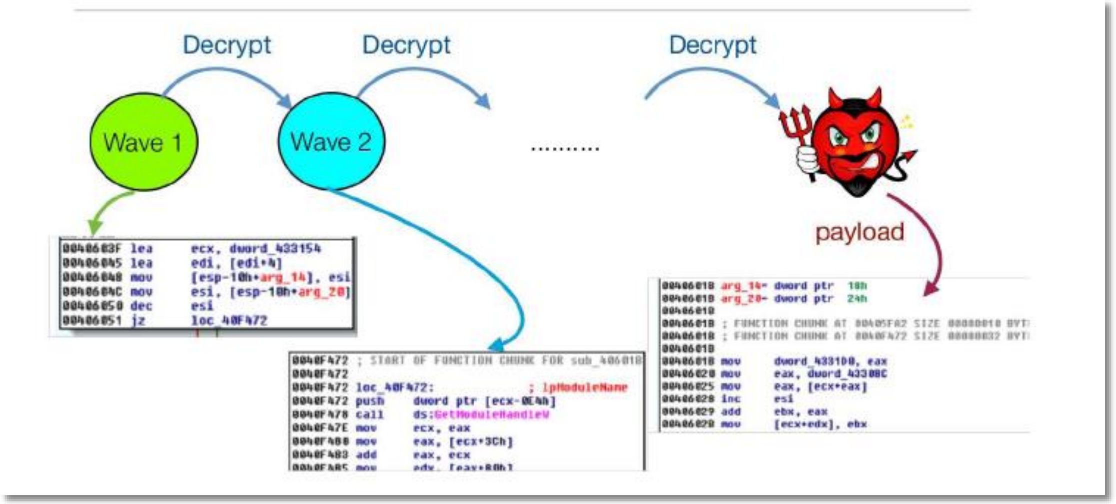
ATTACKER in Standard Program Analysis

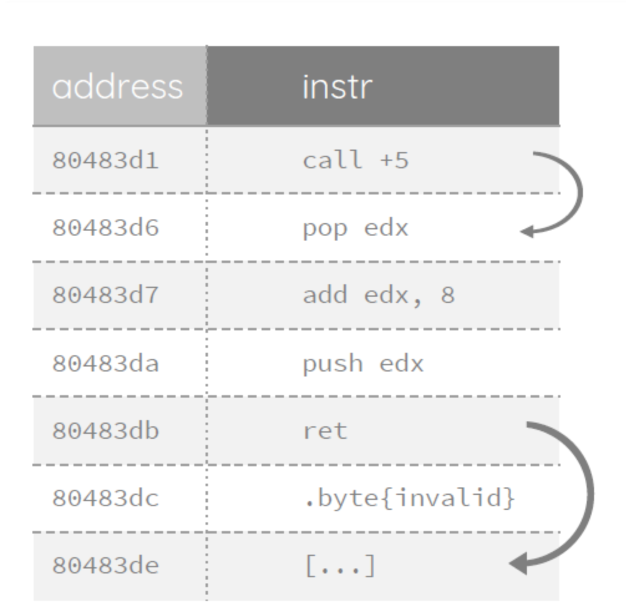
- **We are reasoning worst case: seems very powerful!**

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

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

ADVERSARIAL BINARY CODE





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

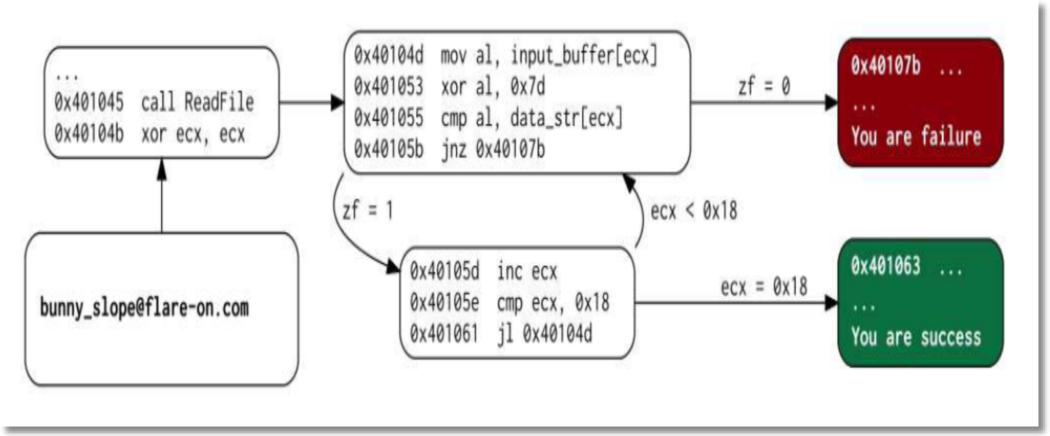
eg: $7y^2 - 1 \neq x^2$

(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**
- **…**



```
...
0x401045  call ReadFile
0x40104b  xor ecx, ecx
```

```
0x40104d  mov al, input_buffer[ecx]
0x401053  xor al, 0x7d
0x401055  cmp al, data_str[ecx]
0x40105b  jnz 0x40107b
```

```
0x40105d  inc ecx
0x40105e  cmp ecx, 0x18
0x401061  jl 0x40104d
```

```
0x40107b  ...
...
You are failure
```

```
0x401063  ...
...
You are success
```

bunny_slope@flare-on.com

OUTLINE

- **Context: a little bit of formal methods for safety**

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

- **The BINSEC platform**

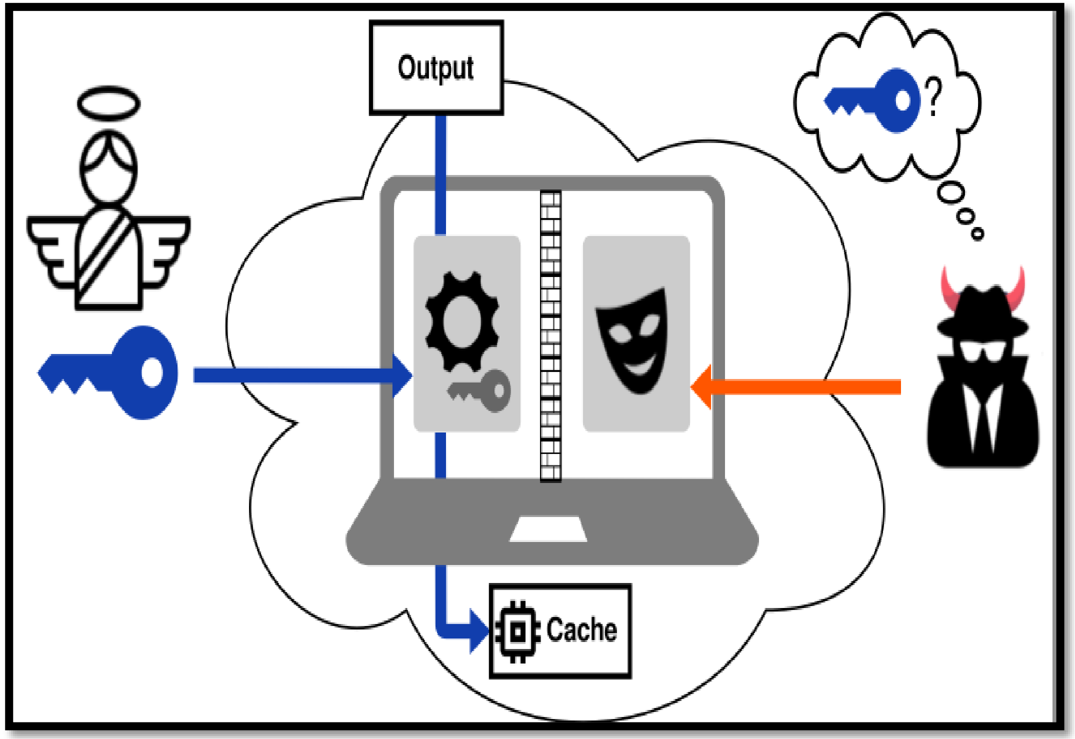
- **From source-level safety to binary-level security: some examples**

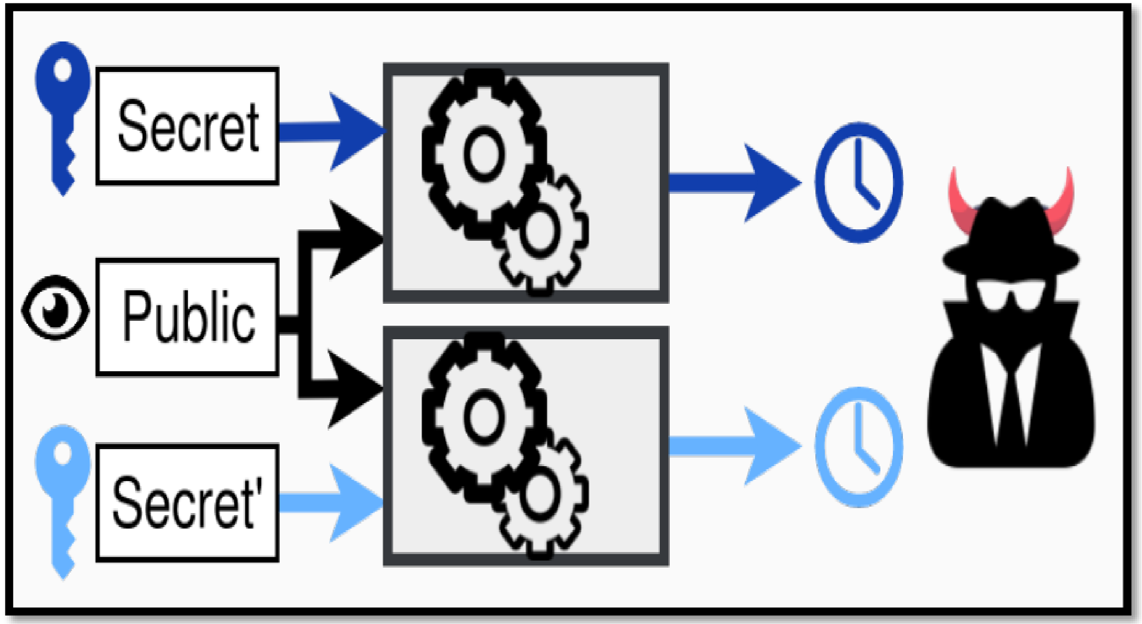
- **Conclusion**

EXAMPLE: TIMING ATTACKS

Information leakage

Properties over pairs of executions

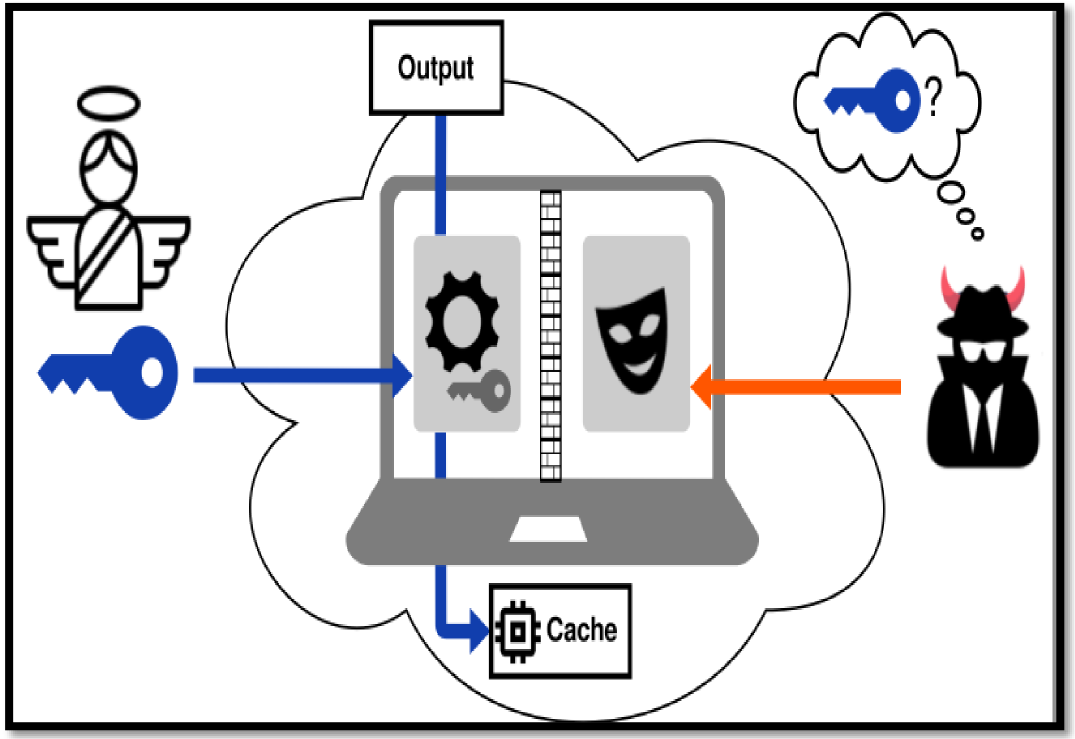




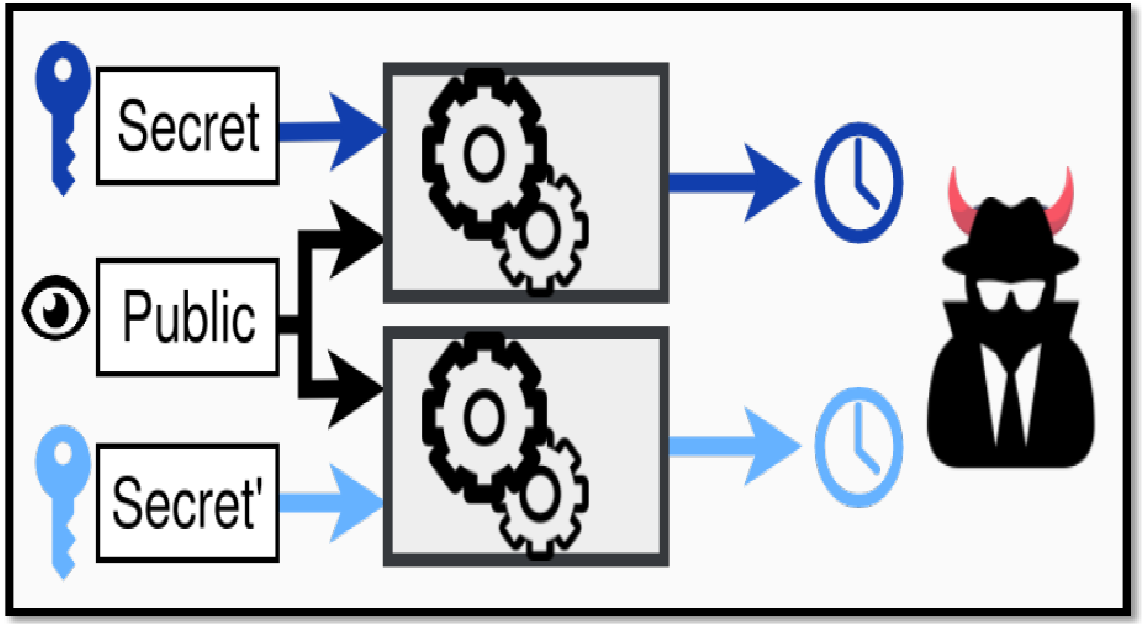
EXAMPLE: TIMING ATTACKS

- Hyperproperties
- Quantitative

Information leakage



Properties over pairs of executions



OUTLINE

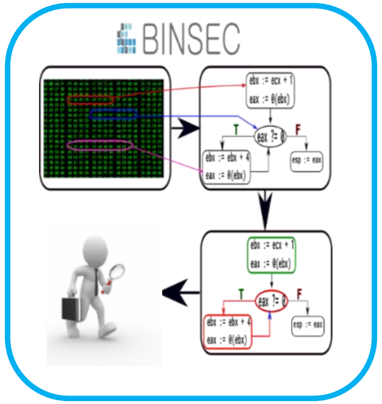
- Context: a little bit of formal methods for safety
- Binary-level security analysis: benefits & challenges
- **The BINSEC platform**
- From source-level safety to binary-level security: some examples
- Conclusion

BINSEC: brings formal methods to binary-level security analysis

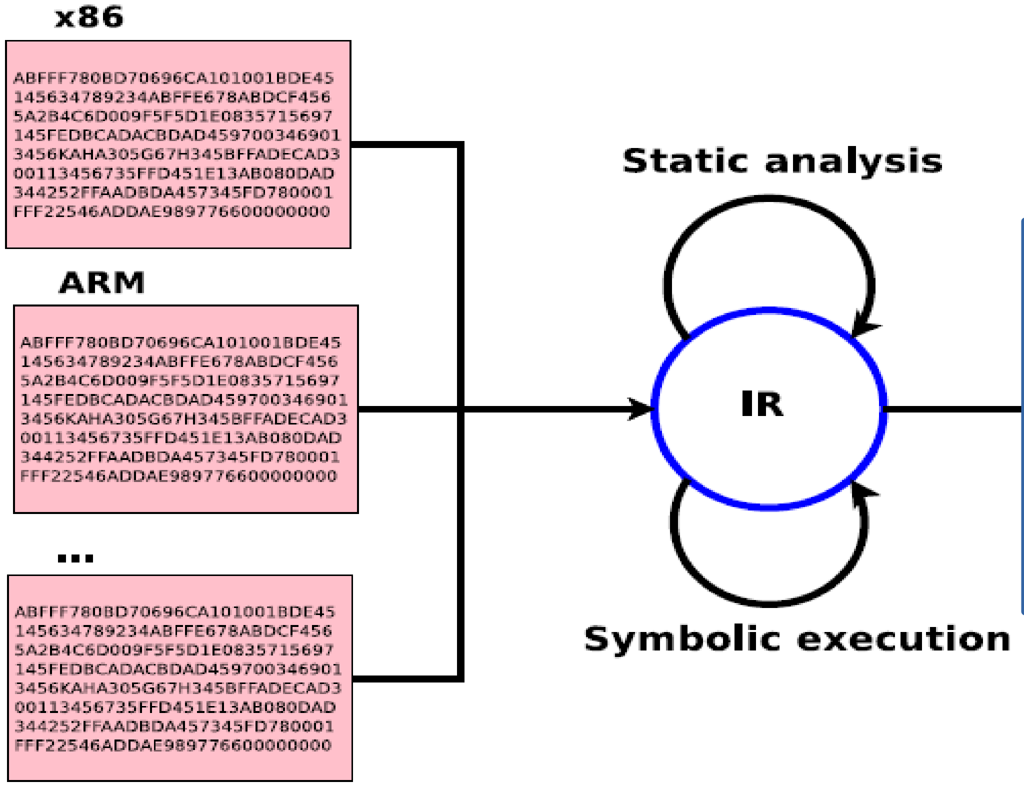
Break

Prove

Protect



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

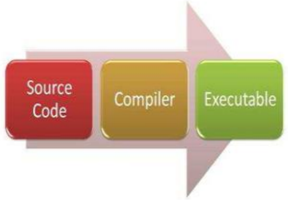


```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

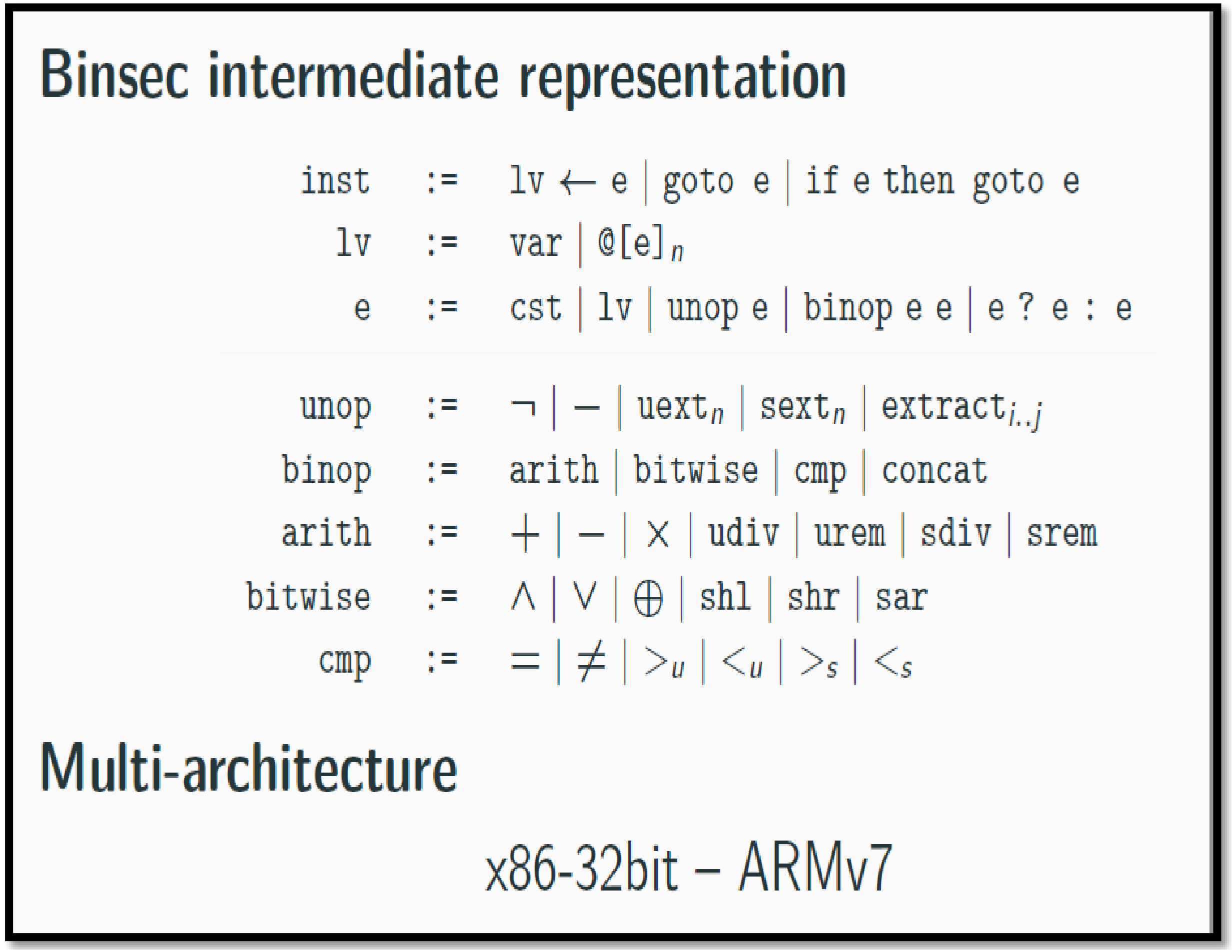
```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

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

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



Key 1: INTERMEDIATE REPRESENTATION [CAV’11]



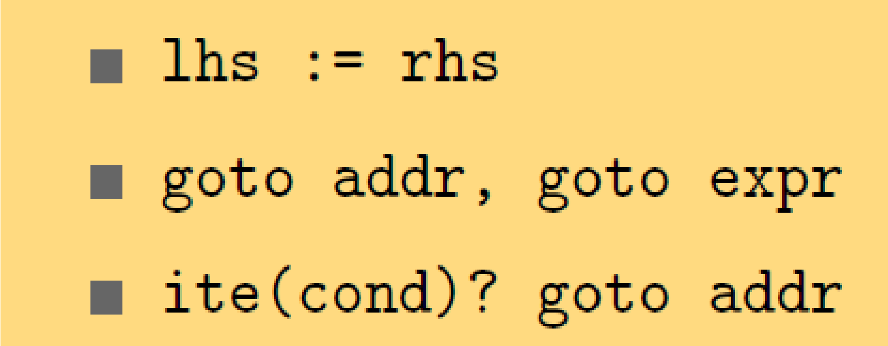
Binsec intermediate representation

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 := $\neg$ | $-$ | uext$_n$ | sext$_n$ | extract$_{i..j}$
binop := arith | bitwise | cmp | concat
arith := $+$ | $-$ | $\times$ | udiv | urem | sdiv | srem
bitwise := $\wedge$ | $\vee$ | $\oplus$ | shl | shr | sar
cmp := $=$ | $\neq$ | $>_u$ | $<_u$ | $>_s$ | $<_s$

Multi-architecture

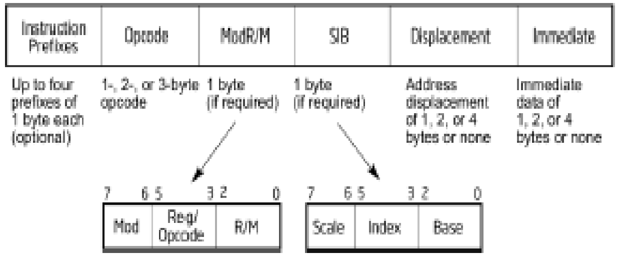
x86-32bit – ARMv7



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

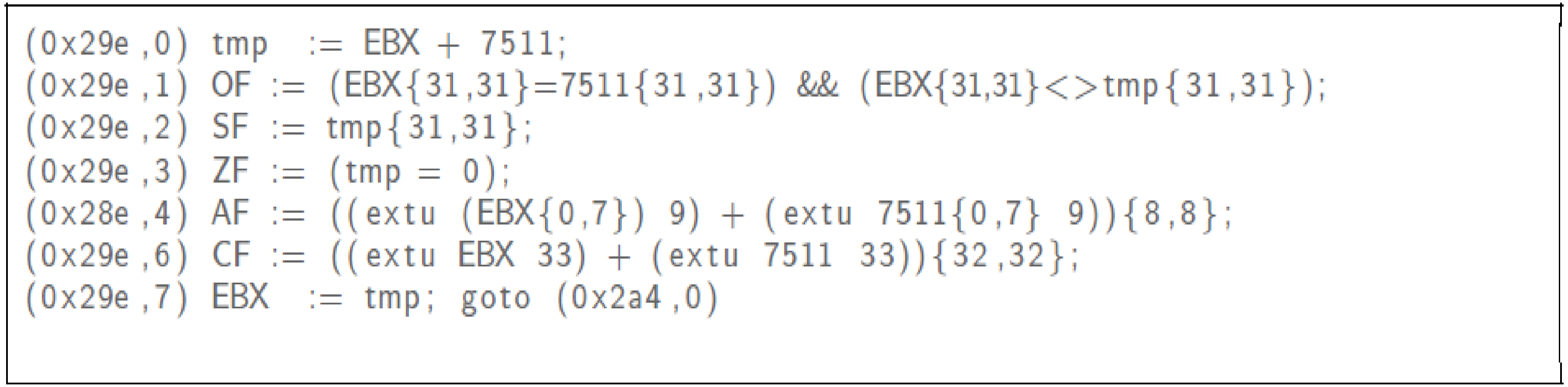
- **Concise**
- **Well-defined**
- **Clear, side-effect free**

INTERMEDIATE REPRESENTATION



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

$$\boxed{\text{81 c3 57 1d 00 00}} \overset{x86reference}{\Rightarrow} \boxed{\text{ADD EBX 1d57}}$$



```
(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 (Godefroid 2005) [variants, optimizations]

Find real bugs

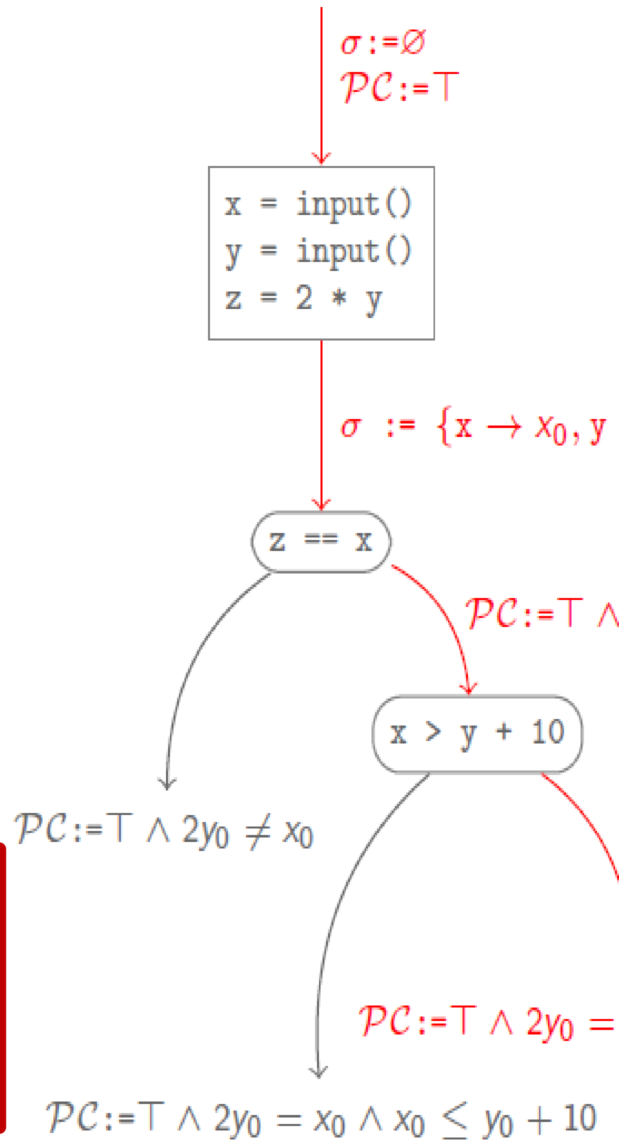
Bounded verification

Robust

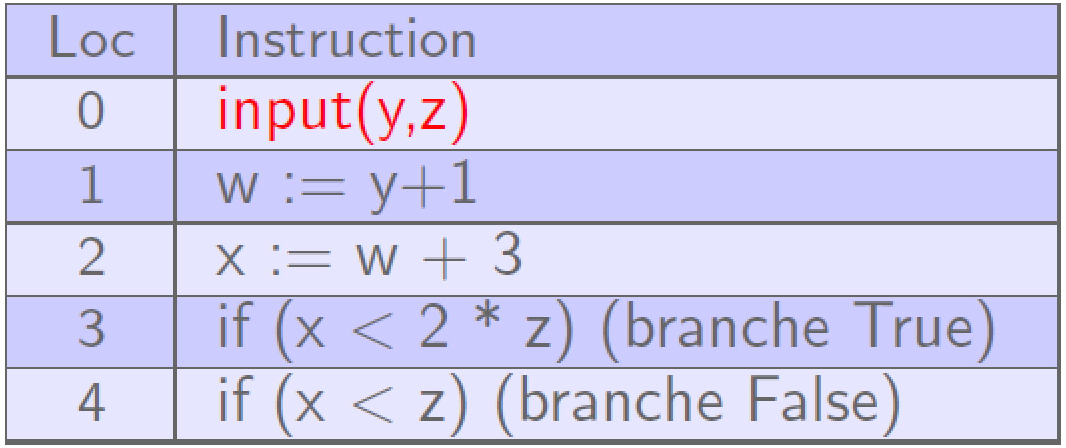
```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```

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)

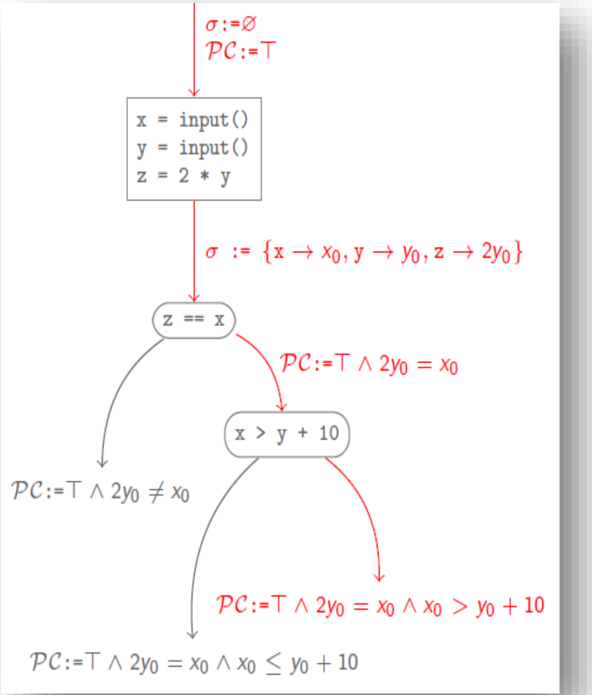
$$\text{let } W_1 \triangleq Y_0 + 1 \text{ in}$$
$$\text{let } X_2 \triangleq W_1 + 3 \text{ in}$$
$$X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Blackbox solvers

SMT Solver

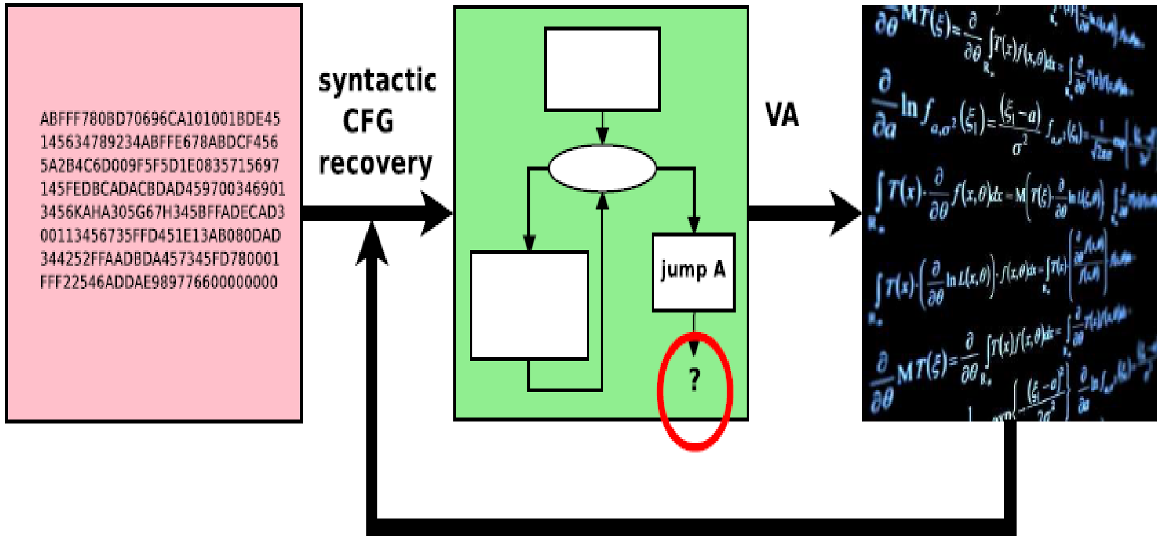
my input!!

Y0 = 0 /\ Z0=3



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

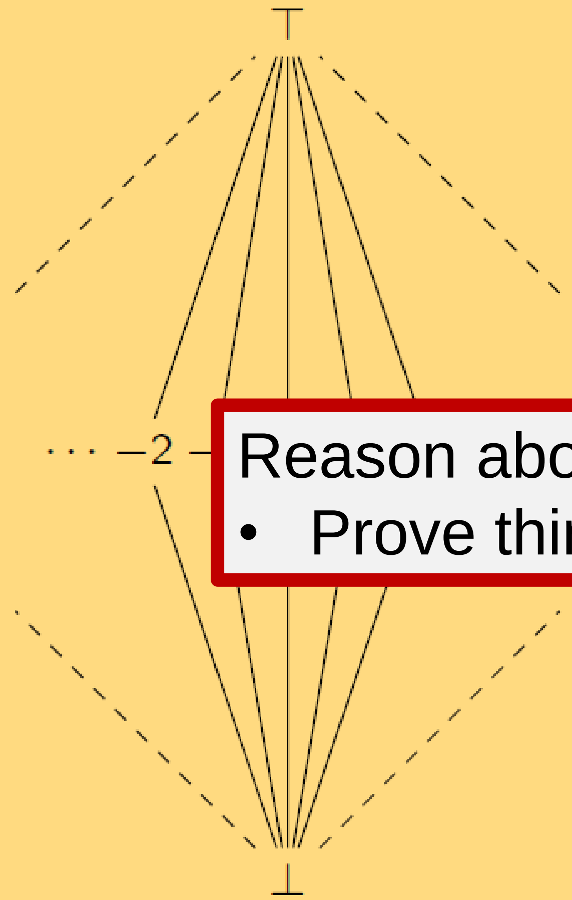
Complete verification



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

Generalize constant propagation



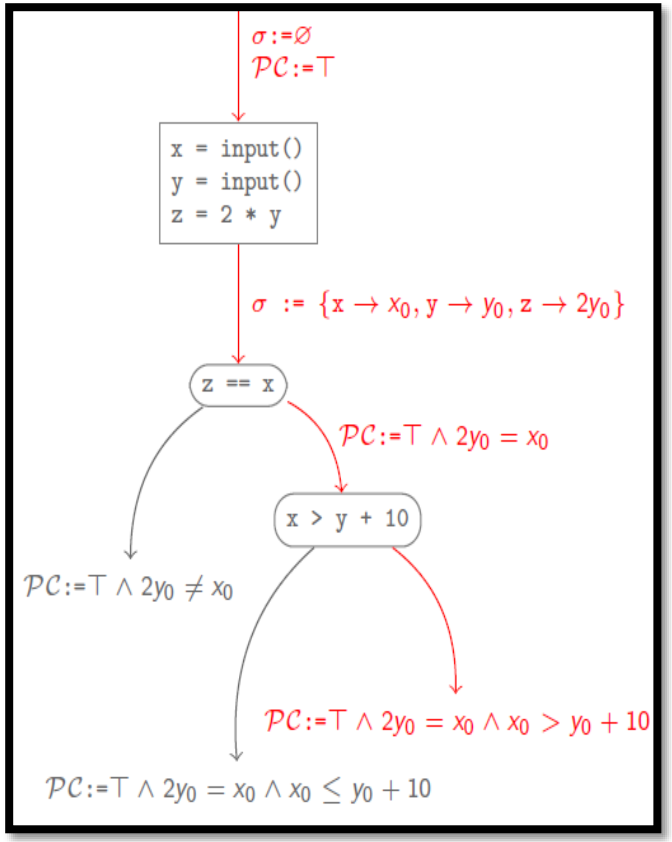
Reason about all paths

- Prove things

OUTLINE

- Prologue: a little bit of formal methods for safety
- Binary-level security analysis: benefits & challenges
- The BINSEC platform
- From source-level safety to binary-level security: some examples
- Conclusion

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

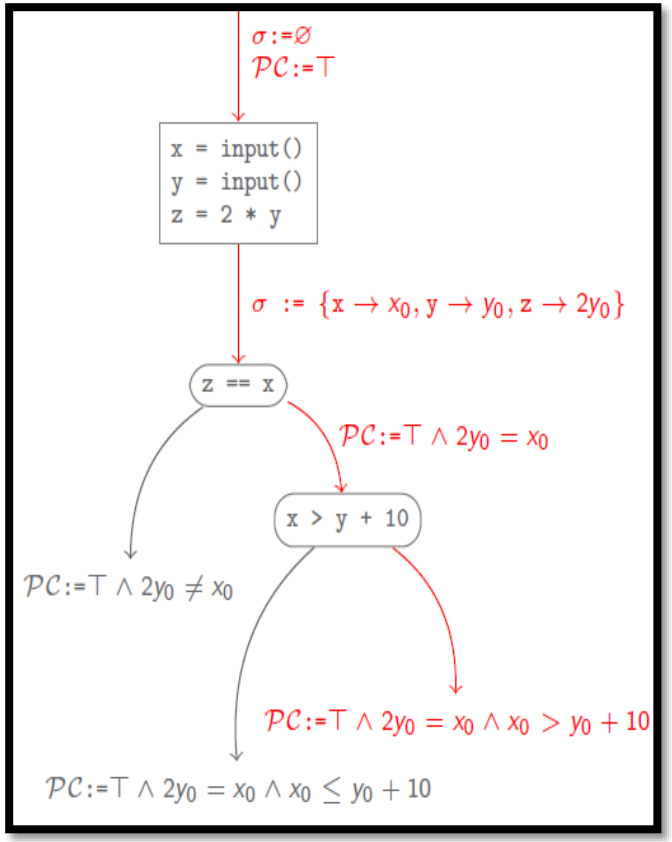


- Intensive path exploration
- Target critical bugs

Challenge = path explosion

Find a needle in the heap !

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

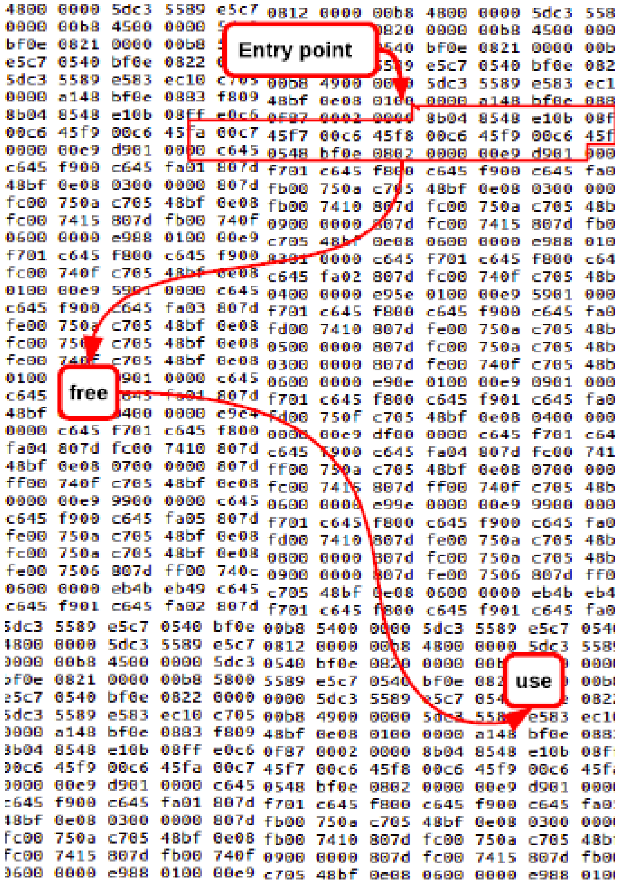


```
x = input()
y = input()
z = 2 * y
```

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

Challenge = path explosion

Case 1: What about hard-to-find bugs [SSPREW'16](with Josselin Feist et al.)

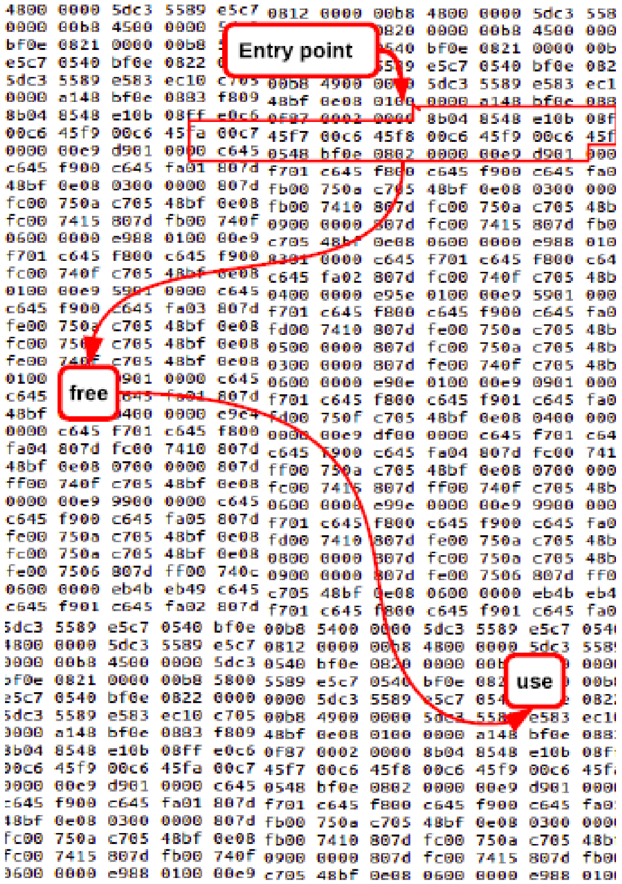


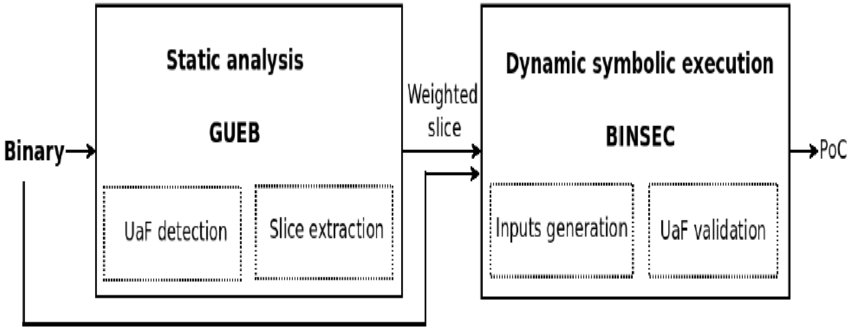
Use-after-free bugs

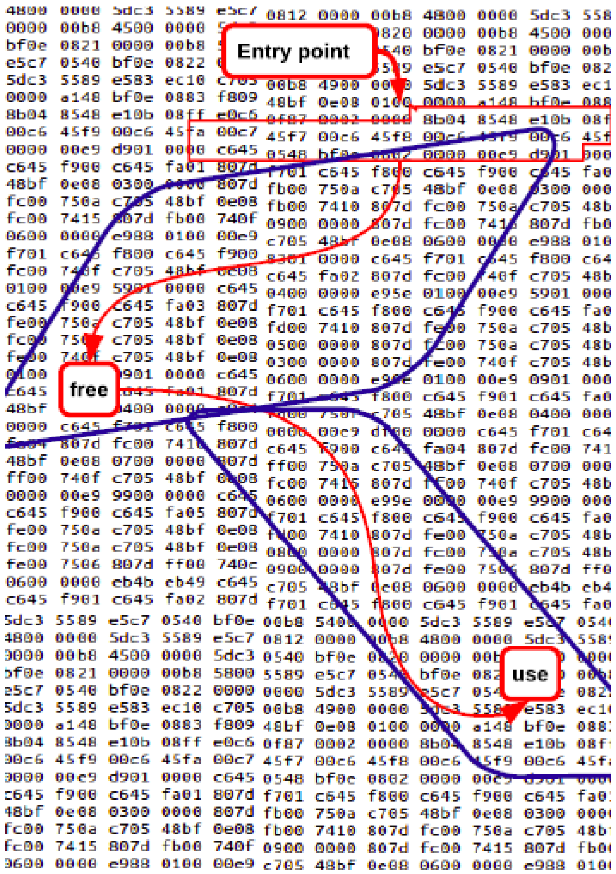
- Very hard to find
- Sequence of events
- DSE gets lost

Case 1: What about hard-to-find bugs [SSPREW’16](with Josselin Feist et al.)

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







Guide DSE with an unsound static analysis

CASE 2: reverse & deobfuscation

- Prove something infeasible
- SE cannot help here

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

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

↧

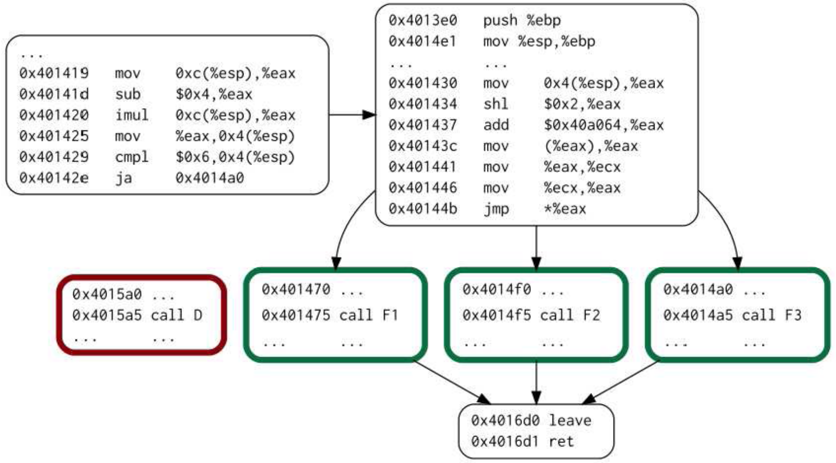
```
mov   eax, ds:X
mov   ecx, ds:Y
imul  ecx, ecx
imul  ecx, 7
sub   ecx, 1
imul  eax, eax
cmp   ecx, eax
jz    <dead_addr>
```

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



```
...
0x401419   mov    0xc(%esp),%eax
0x40141d   sub    $0x4,%eax
0x401420   imul   0xc(%esp),%eax
0x401425   mov    %eax,0x4(%esp)
0x401429   cmpl   $0x6,0x4(%esp)
0x40142e   ja     0x4014a0
```

```
0x4013e0   push %ebp
0x4014e1   mov %esp,%ebp
...        ...
0x401430   mov    0x4(%esp),%eax
0x401434   shl    $0x2,%eax
0x401437   add    $0x40a064,%eax
0x40143c   mov    (%eax),%eax
0x401441   mov    %eax,%ecx
0x401446   mov    %ecx,%eax
0x40144b   jmp    *%eax
```

```
0x4015a0 ...
0x4015a5 call D
...      ...
```

```
0x401470 ...
0x401475 call F1
...      ...
```

```
0x4014f0 ...
0x4014f5 call F2
...      ...
```

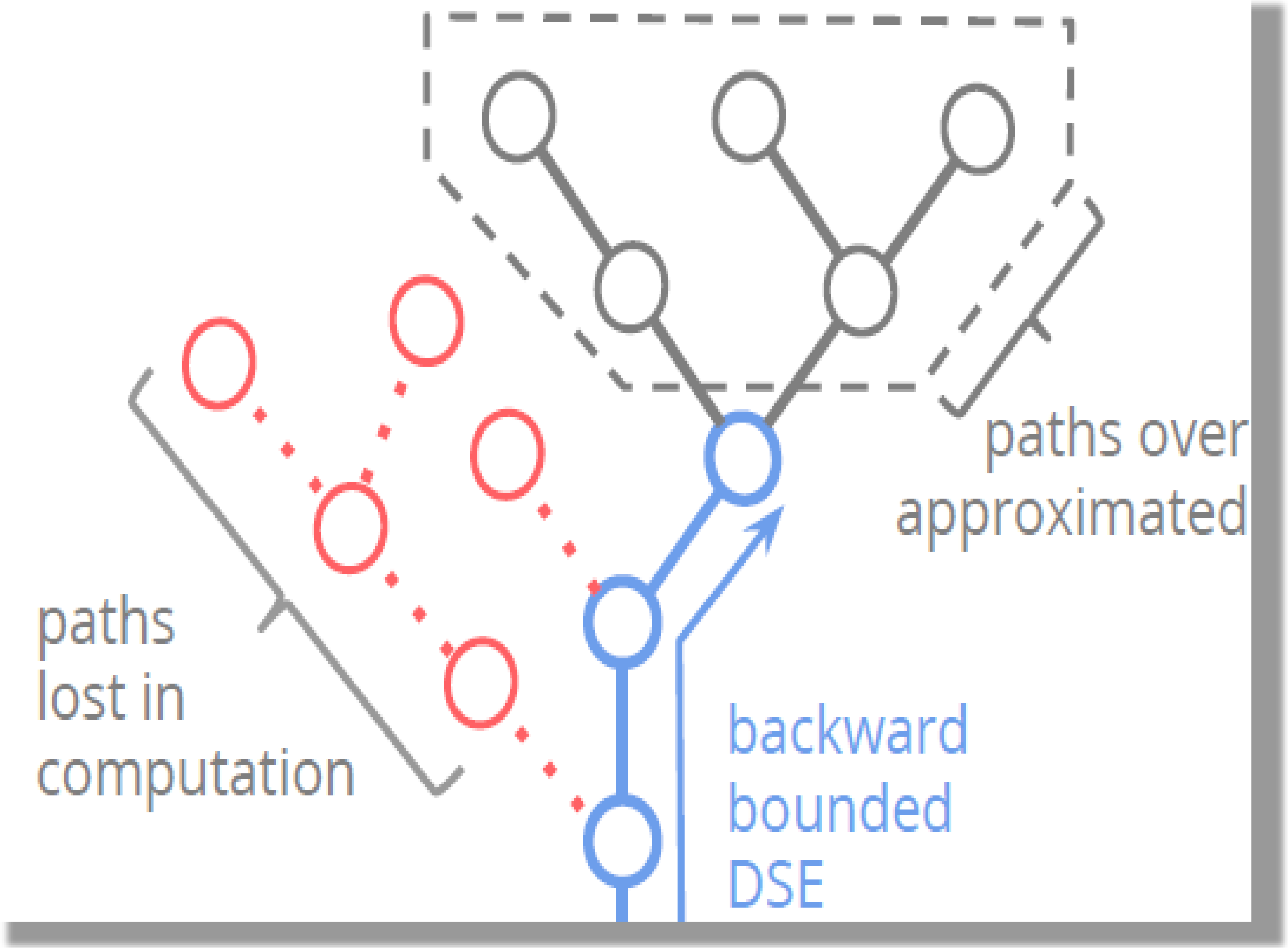
```
0x4014a0 ...
0x4014a5 call F3
...      ...
```

```
0x4016d0 leave
0x4016d1 ret
```

With IDA + BINSEC

All jump targets are found

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

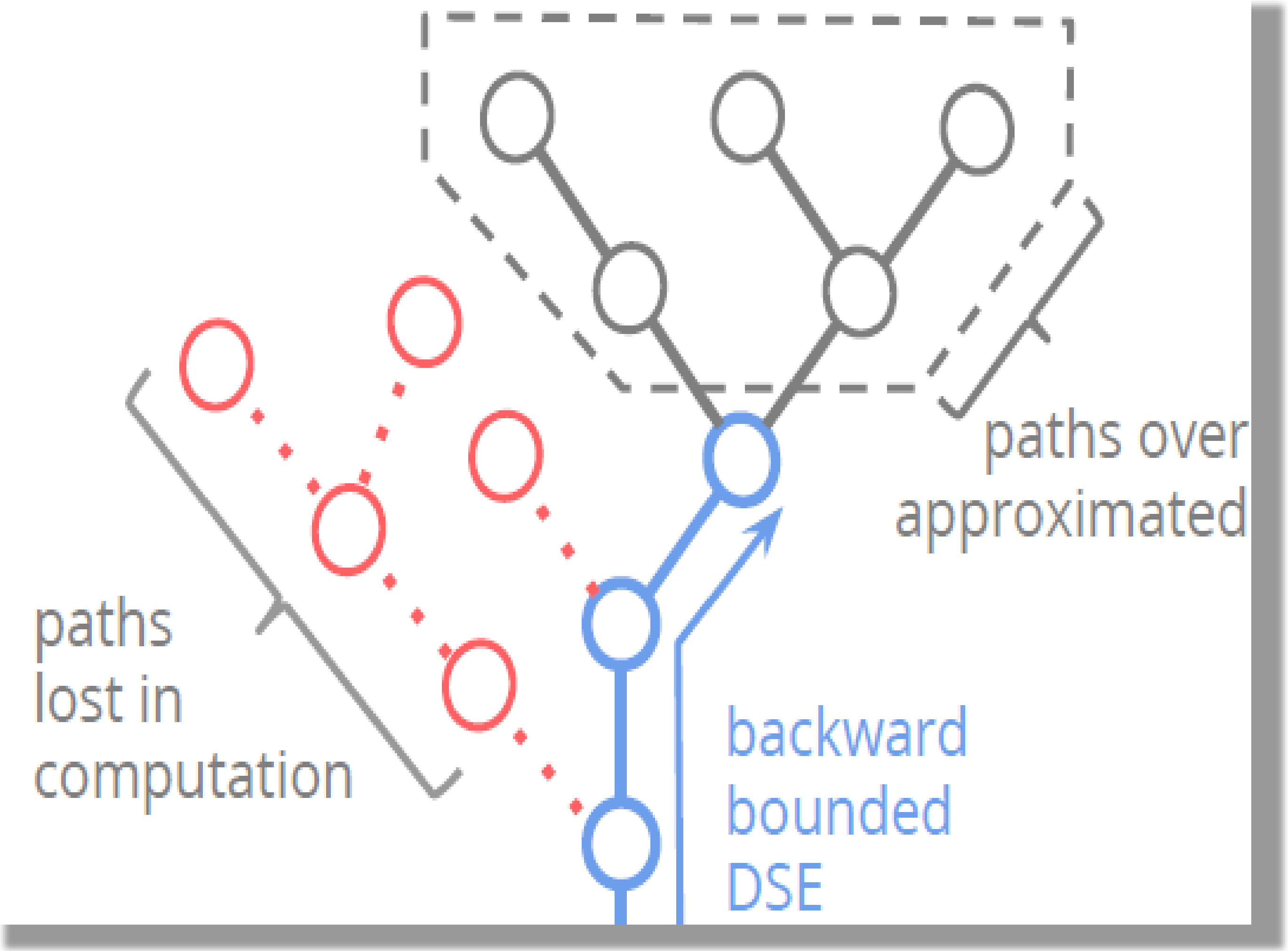


Backward bounded SE

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

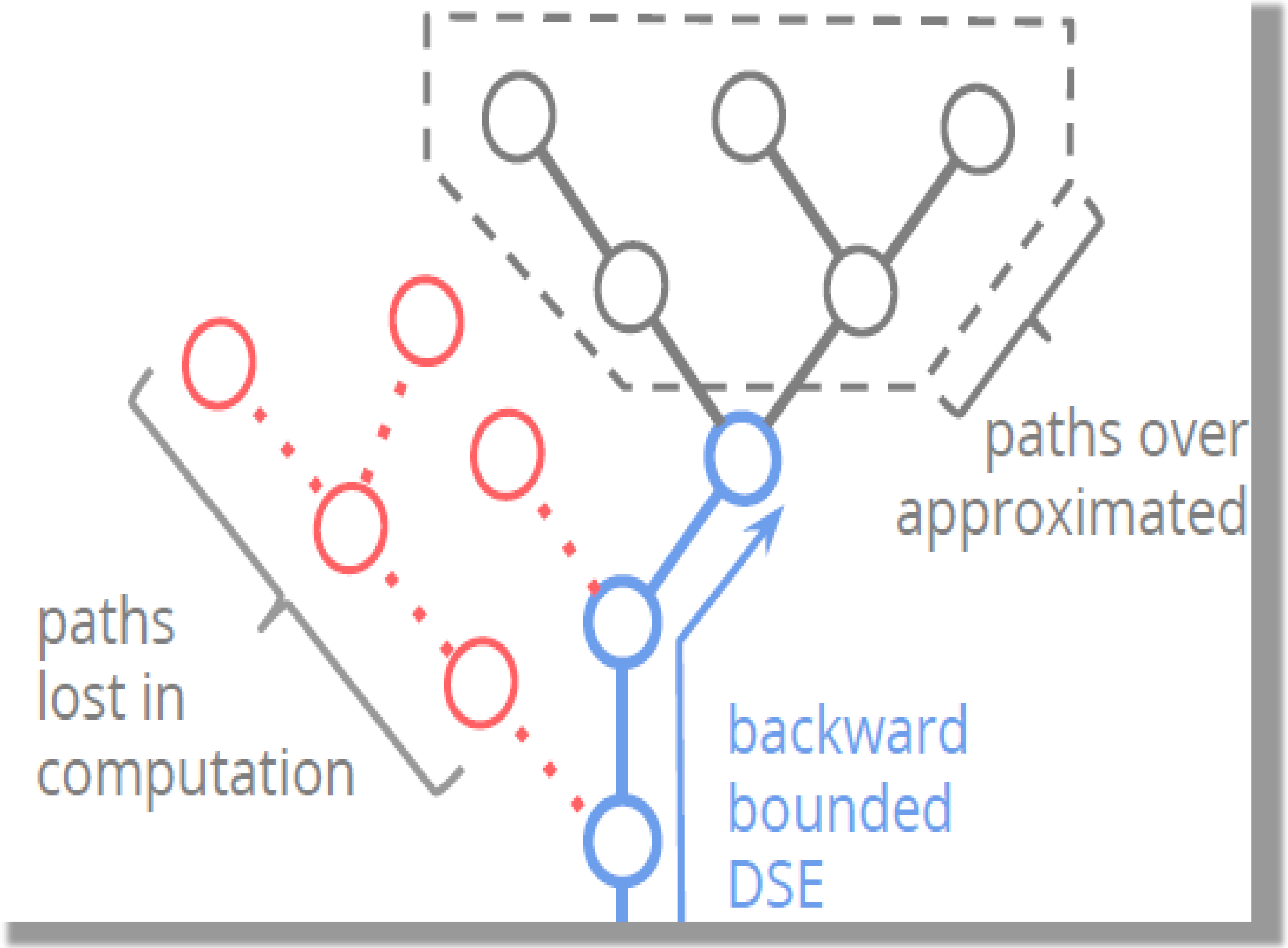
- Prove things
- Local → scalable

Wait …



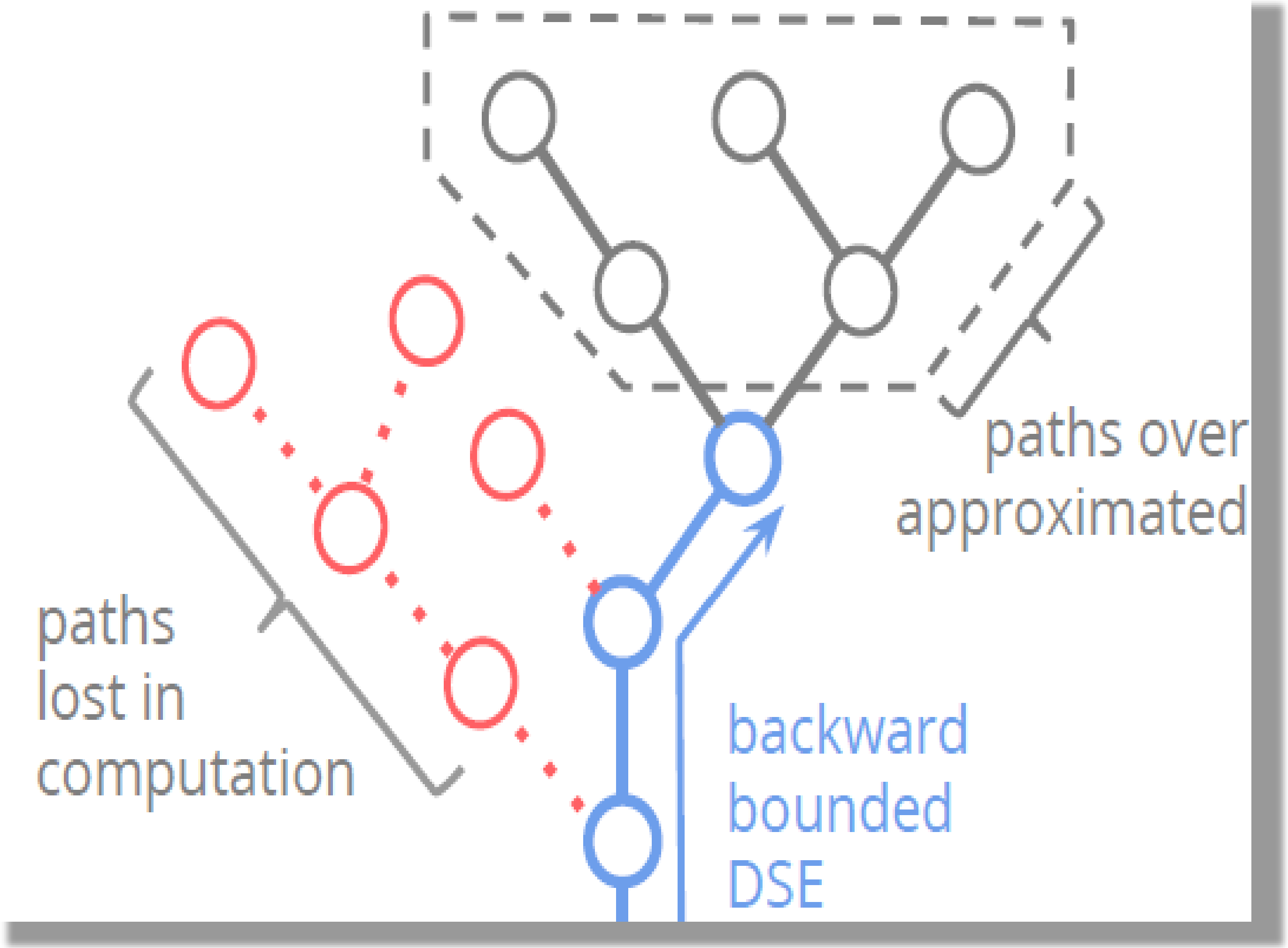
- **False Negative: k too small**
  - ***Missed proofs***

Wait …



- **False Negative: k too small**
  - ***Missed proofs***
- **False Positive: CFG incomplete**
  - ***Wrong proofs ?!***

Wait …



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

- Low rate of wrong proofs
- Controlled XPs

Case 2: THE XTUNNEL MALWARE
-- [BlackHat EU 2016, S&P 2017] (Robin David)

Two heavily obfuscated samples

- **Many opaque predicates**

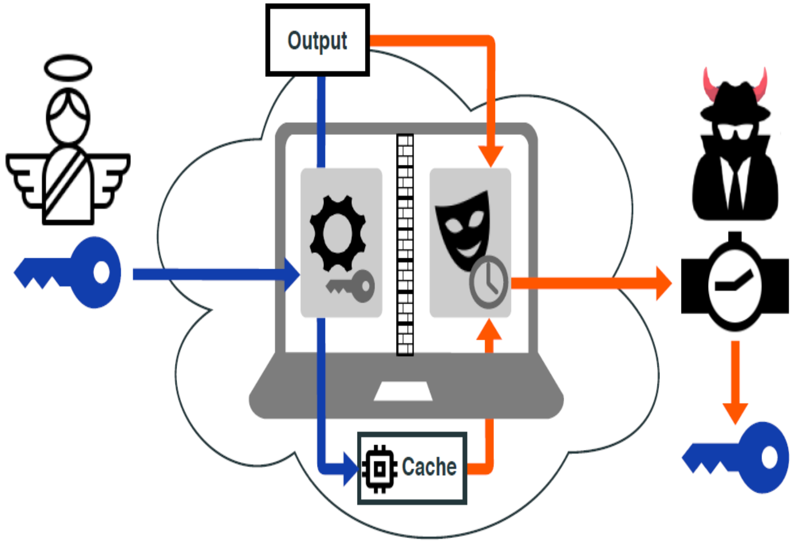
Goal: detect & remove protections

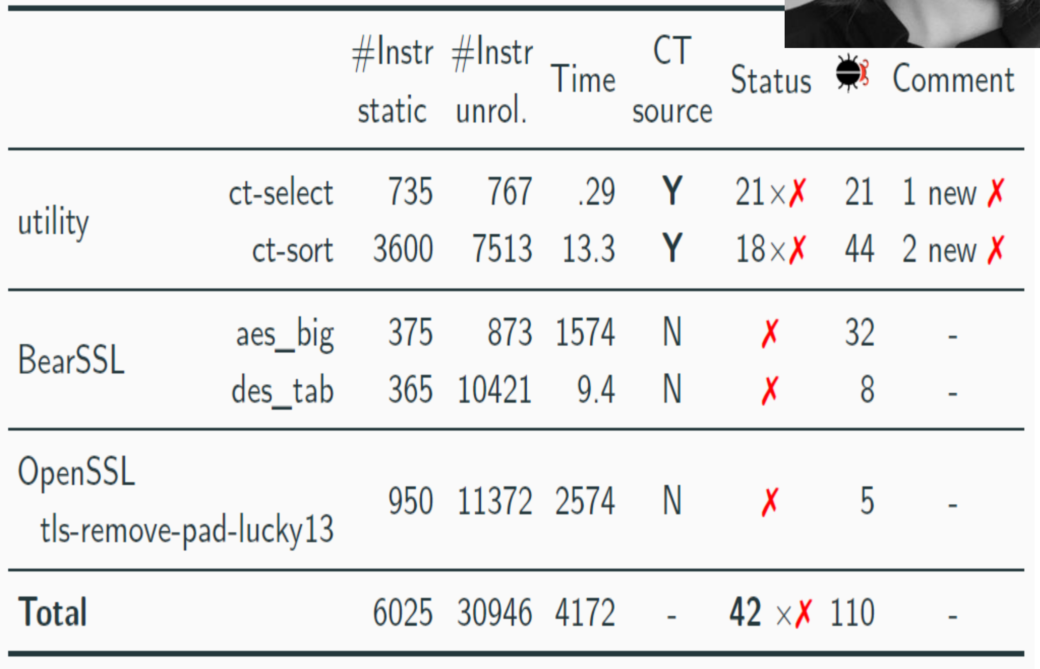
- Identify 40% of code as spurious
- Fully automatic, < 3h [now: 20min]

- Backward-bounded SE
- + dynamic analysis

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

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

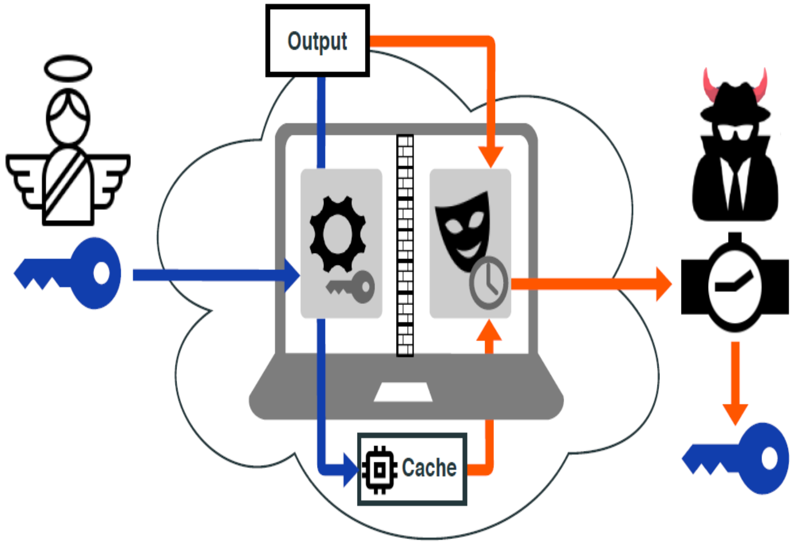




		#Instr static	#Instr unrol.	Time	CT source	Status	🐞	Comment
utility	ct-select	735	767	.29	Y	21×✗	21	1 new ✗
	ct-sort	3600	7513	13.3	Y	18×✗	44	2 new ✗
BearSSL	aes_big	375	873	1574	N	✗	32	-
	des_tab	365	10421	9.4	N	✗	8	-
OpenSSL	tls-remove-pad-lucky13	950	11372	2574	N	✗	5	-
Total		6025	30946	4172	-	**42**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

- Property: timing attacks
- Attacker: speculation

Case 3: SECURING CRYPTO-PRIMITIVES
-- [S&P 2020, NDSS 2021] (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	-

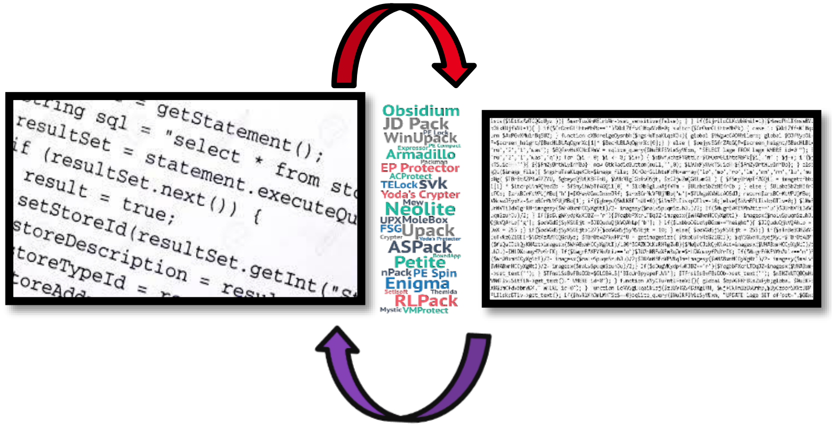
- Relational symbolic execution
- Follows paires of execution
- Check for divergence
- Sharing, merging, preprocess

- 397 crypto code samples, x86 and ARM
- New proofs, 3 new bugs (of verified codes)
- Potential issues in some protection schemes
- 600x faster than prior workl

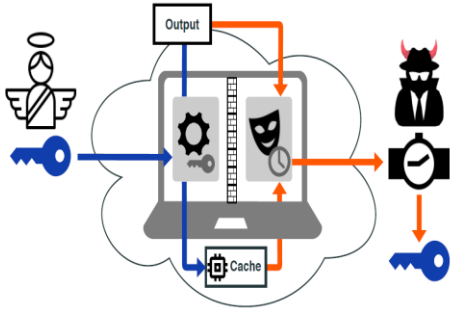
Under the hood: finely tune the technology

- **SMT solvers are powerful weapons**
- **But (binary-level) security problems are terrific beasts**

- **Finely tuning the technology can make a huge difference**



```
ring sql = getStatement();
resultSet = "select * from st
if (resultSet = statement.executeQu
result = true;
setStoreId(resultSet.getInt("s
storeDescription = resu
toreTypeId =
toreAdd
```

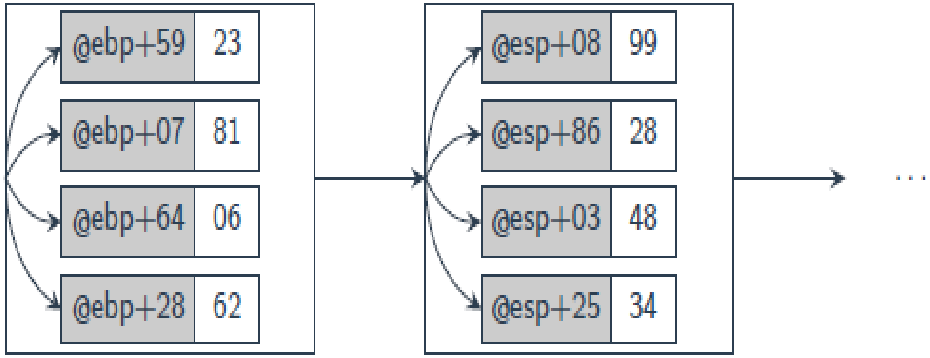


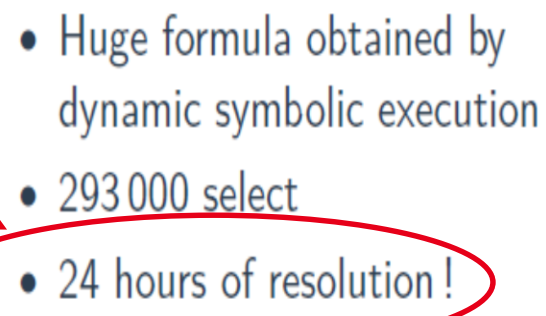
- Some queries: 24h → 1min

- 600x faster than prior approach

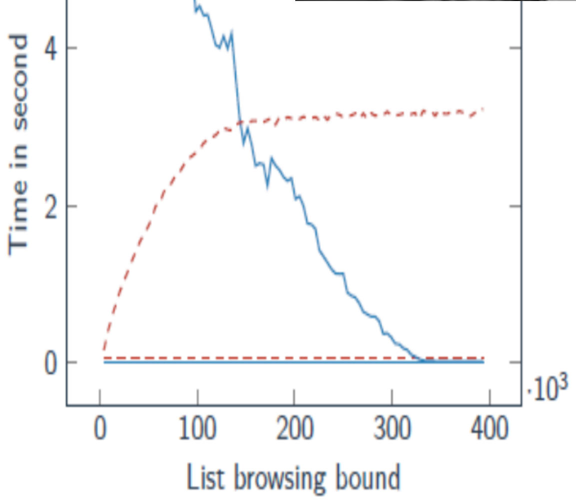
Tuning the solver: intensive array formulas [LPAR 2018] (Benjamin Farinier)

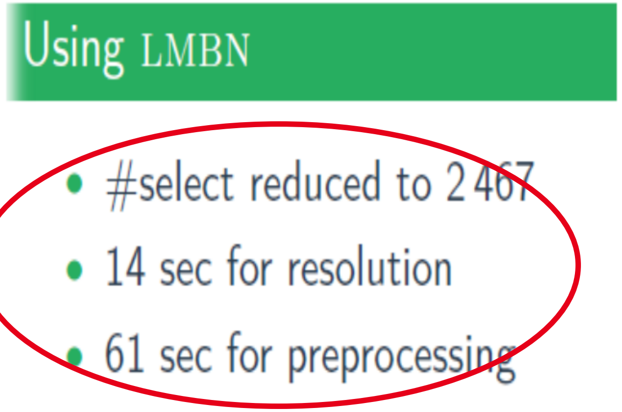
- **Makes the difference!**





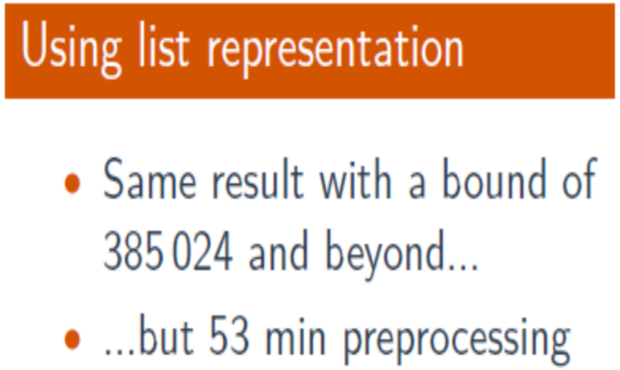
- Huge formula obtained by dynamic symbolic execution
- 293 000 select
- 24 hours of resolution !





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

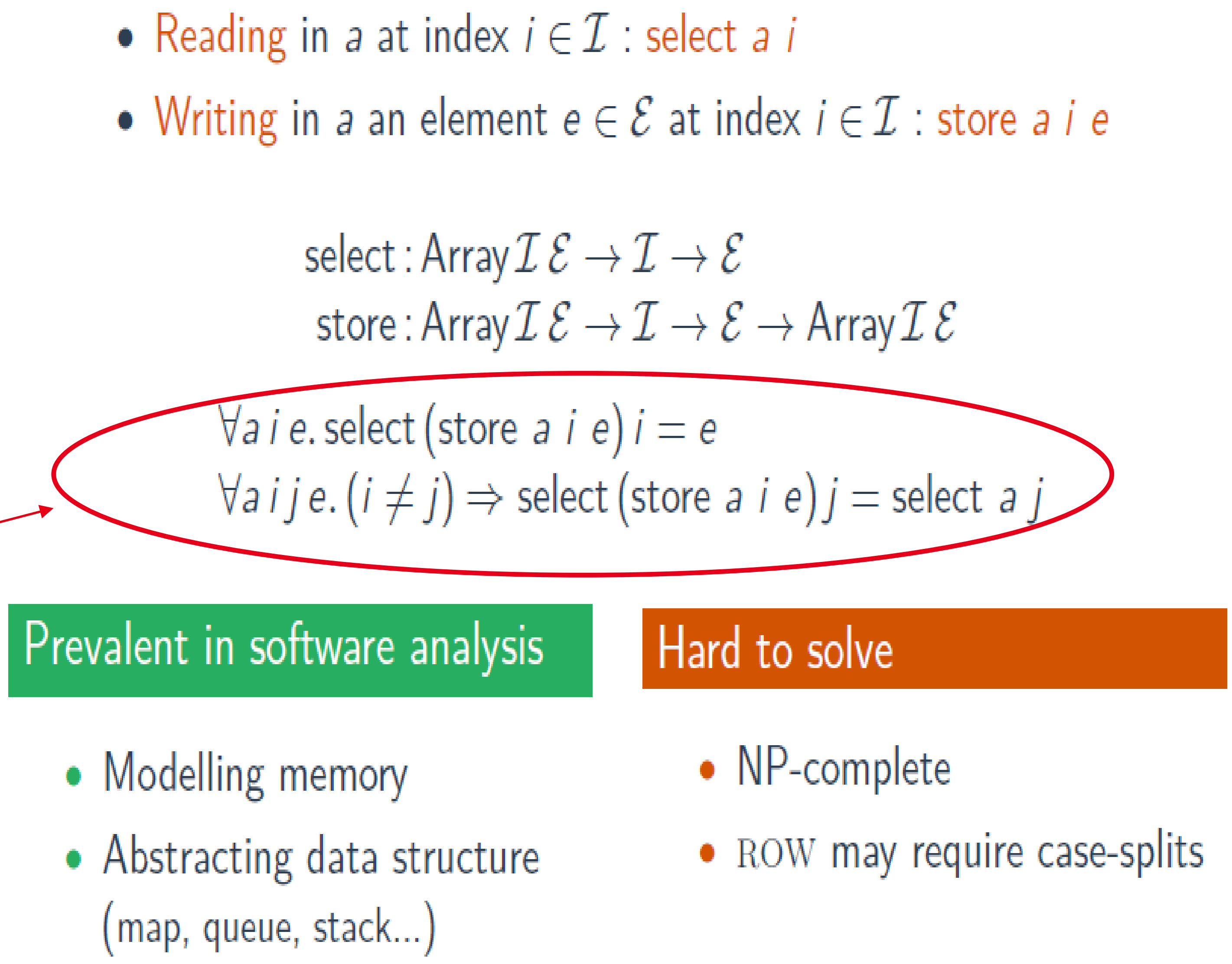
Zoom: efficient low-level memory reasoning [LPAR 2018]

Array theory

- **Necessary to model memory**
- **Hard for solvers (case splits)**

ROW rule may introduce case-splits

Goal: make it tractable



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

$$\text{select} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E}$$
$$\text{store} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array}\,\mathcal{I}\,\mathcal{E}$$

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

Prevalent in software analysis

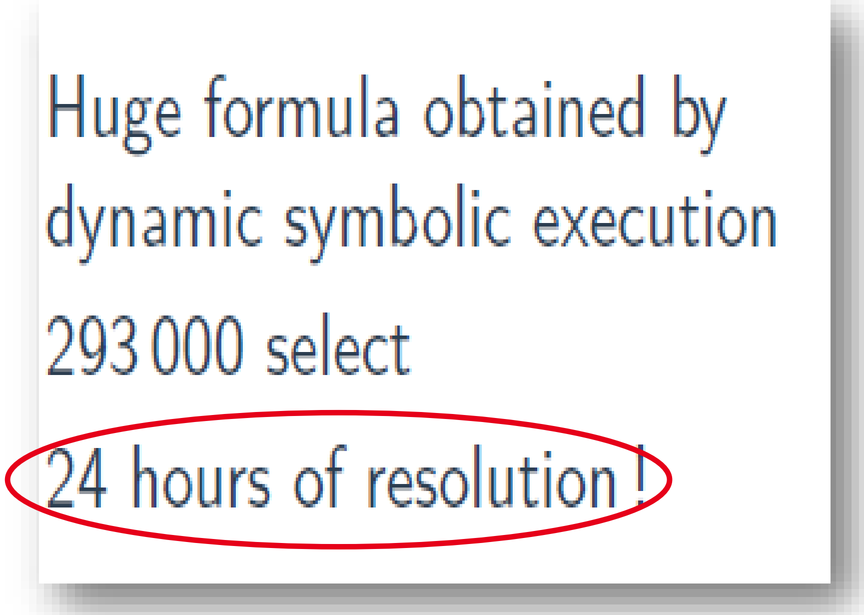
- Modelling memory
- Abstracting data structure (map, queue, stack...)

Hard to solve

- NP-complete
- ROW may require case-splits

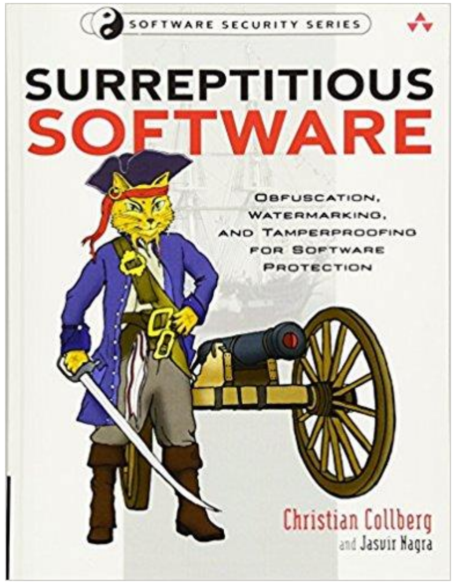
Not pure theory!

Reverse of a ASPACK-protected code


Huge formula obtained by dynamic symbolic execution

293 000 select

24 hours of resolution !

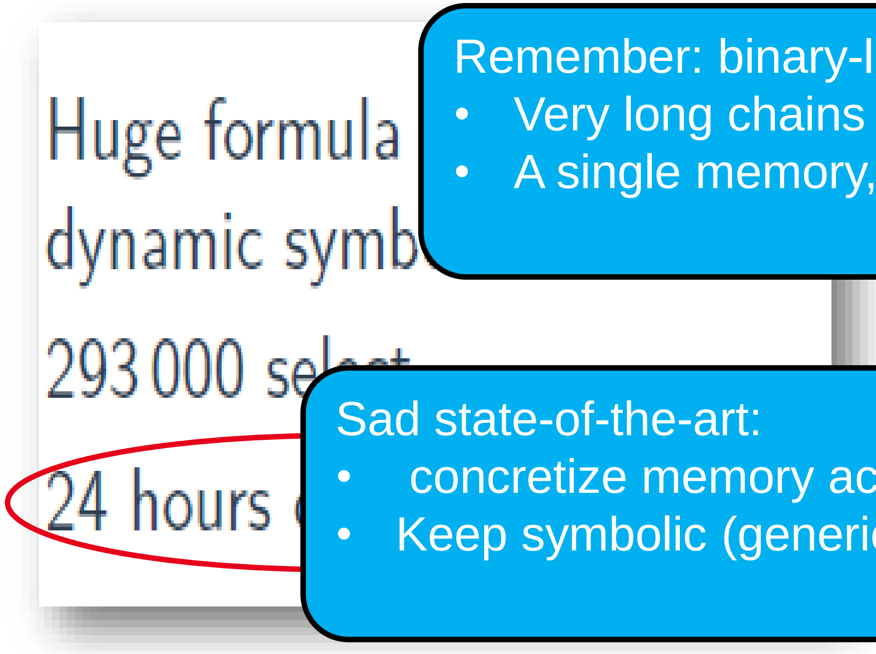


Obsidium
JD Pack
PE Lock
WinUpack
Expressor
PE Compact
Armadillo
Packman
EP Protector
ACProtect
TELock
SVK
Yoda's Crypter
Mew
Neolite
UPX
MoleBox
FSG
Upack
Crypter
Yoda's Protector
ASPack
BoxedApp
Petite
nPack
PE Spin
Enigma
Setisoft
Themida
RLPack
Mystic
VMProtect

Not pure theory!

Reverse of a ASPACK-protected code

```
...ring sql = ... getStatement();
resultSet = "select * from st
if (resultSet = statement.executeQu
  result = true;
  setStoreId(resultSet.getInt("St
  storeDescription = resu
  storeTypeId = r
  storeAdd
```



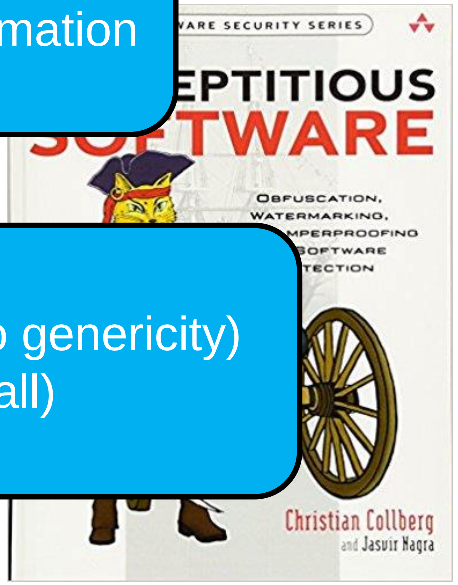
Huge formula

dynamic symb

293 000 select

24 hours

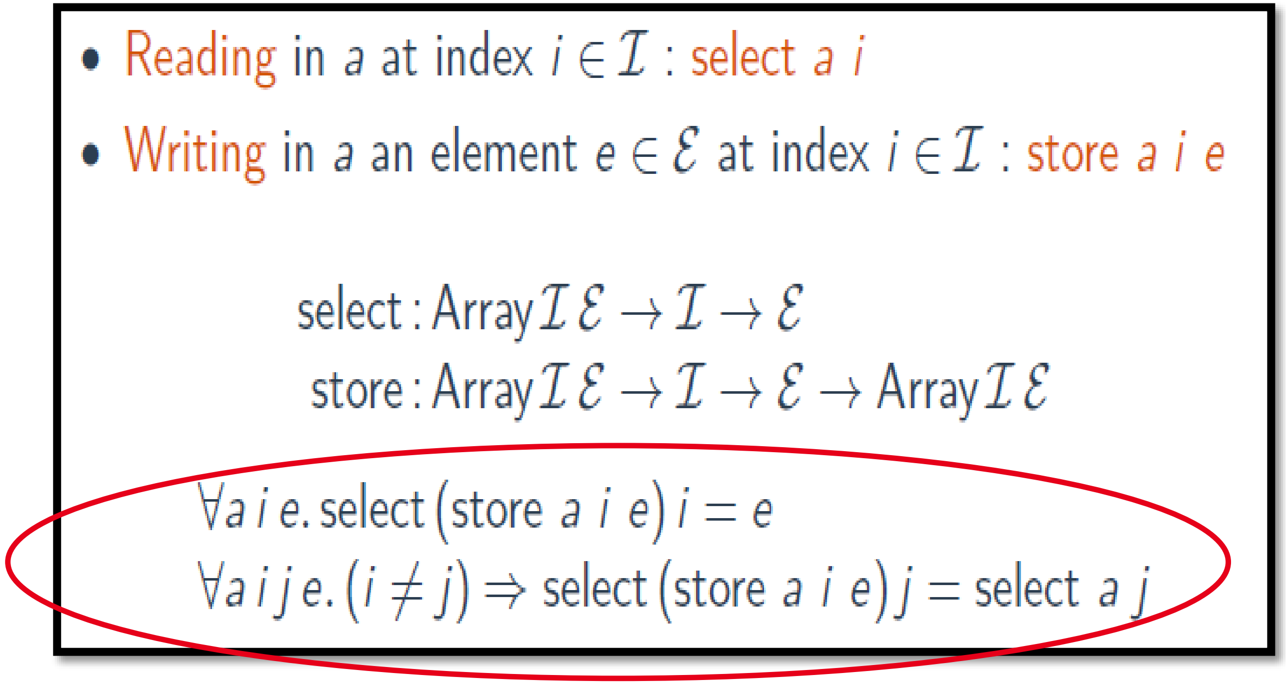
Remember: binary-level
- Very long chains of write
- A single memory, no partition information



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

Inner-working of array theory

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



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

$$\text{select} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E}$$
$$\text{store} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array}\,\mathcal{I}\,\mathcal{E}$$

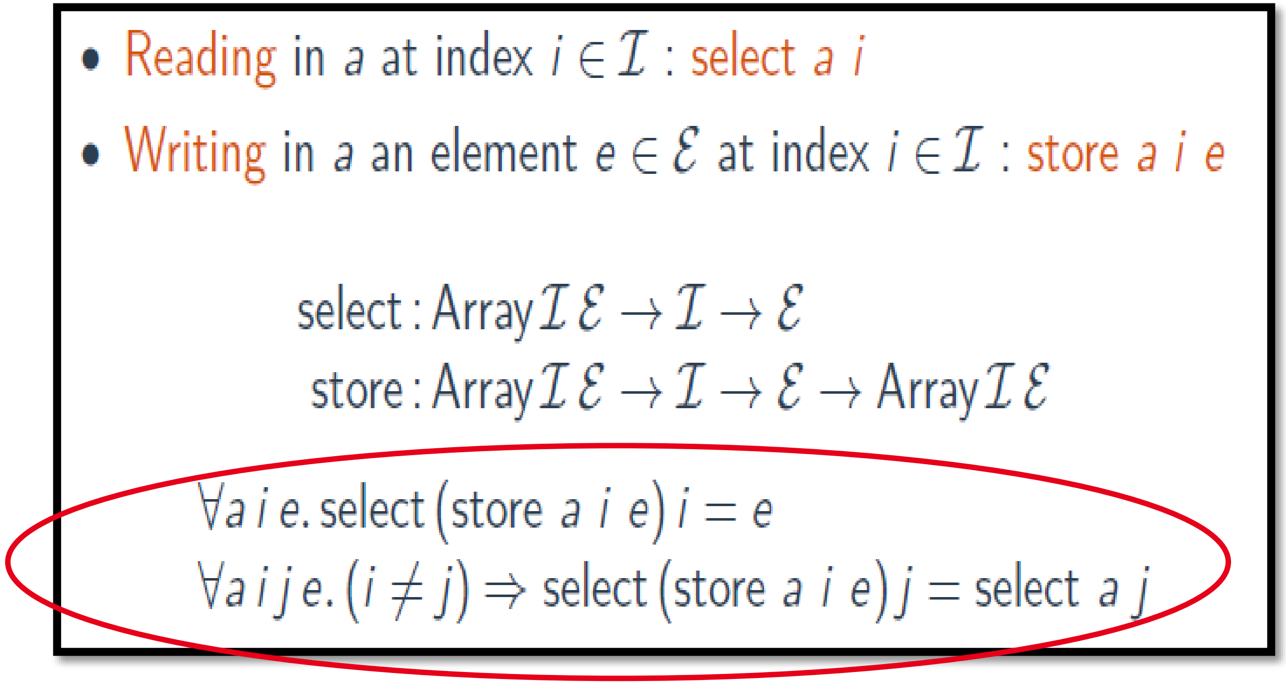
$$\forall a\, i\, e.\ \text{select}\,(\text{store}\ a\ i\ e)\,i = e$$
$$\forall a\, i\, j\, e.\ (i \neq j) \Rightarrow \text{select}\,(\text{store}\ a\ i\ e)\,j = \text{select}\ a\ j$$

ROW reasoning may introduce case-splits

Eliminate ROW

Inner-working of array theory

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



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

$$\text{select} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E}$$
$$\text{store} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array}\,\mathcal{I}\,\mathcal{E}$$

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

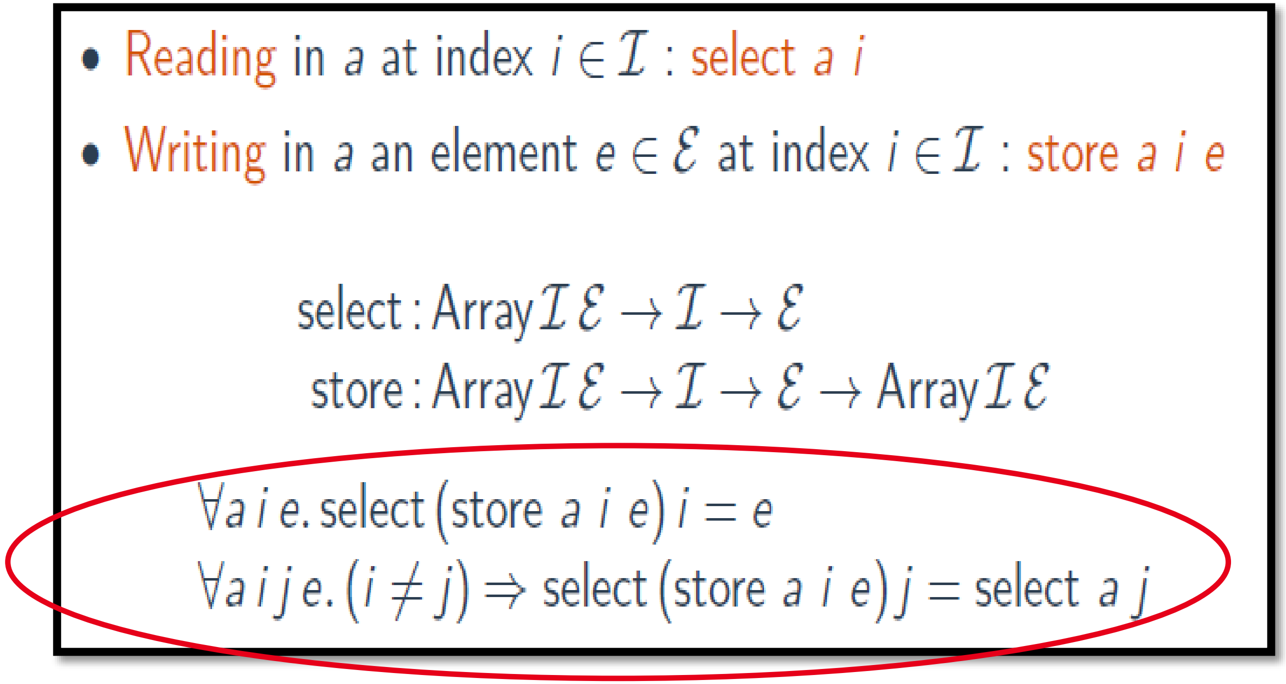
ROW reasoning may introduce case-splits

Eliminate ROW

ROW rules could be used as a preprocessing??

Inner-working of array theory

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



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

$$\text{select} : \text{Array}\,\mathcal{I}\,\mathcal{E} \rightarrow \mathcal{I} \rightarrow \mathcal{E}$$
$$\text{store} : \text{Array}\,\mathcal{I}\,\mathcal{E} \rightarrow \mathcal{I} \rightarrow \mathcal{E} \rightarrow \text{Array}\,\mathcal{I}\,\mathcal{E}$$

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

ROW reasoning may introduce case-splits

Eliminate ROW

ROW rules could be used as a preprocessing??

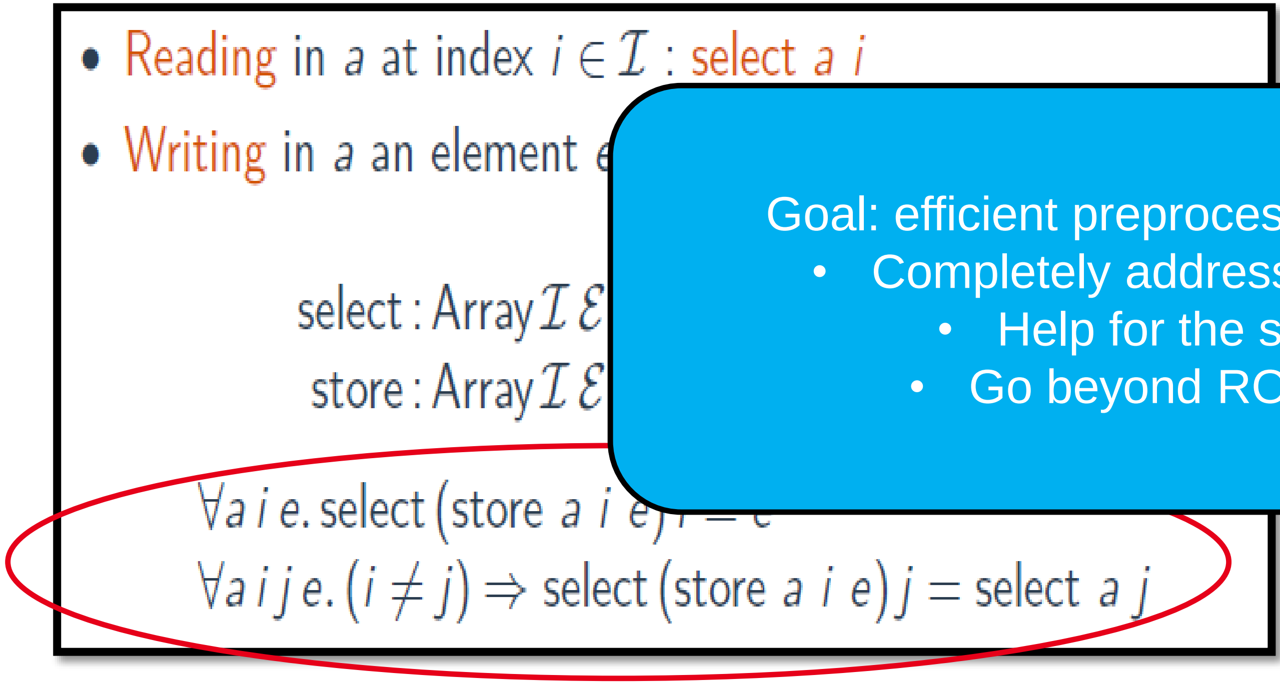
Quadratic reasoning

- Term-based equality
- Disequality??

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

Inner-working of array theory

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



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

select : Array $\mathcal{I}$ $\mathcal{E}$

store : Array $\mathcal{I}$ $\mathcal{E}$

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

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

Goal: efficient preprocessing to remove ROW

- Completely address the constant case
- Help for the symbolic case
- Go beyond ROW (eg, WOW)

ing may
se-splits

Eliminate ROW

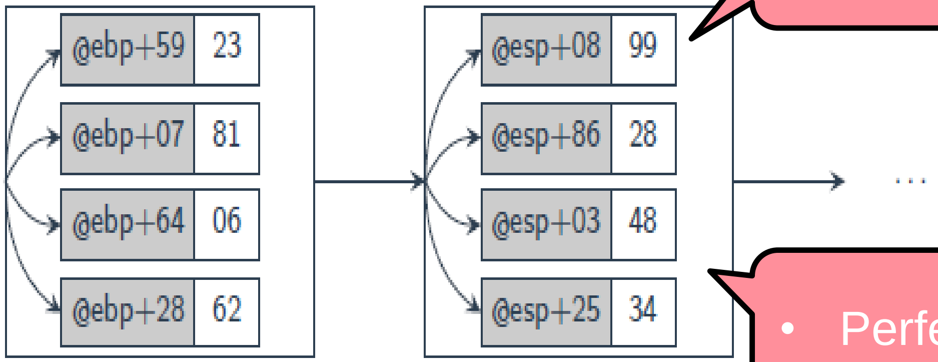
ROW rules could be used as a preprocessing??

Quadratic reasoning

- Term-based equality
- Disequality??

Fast array simplification (1)

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



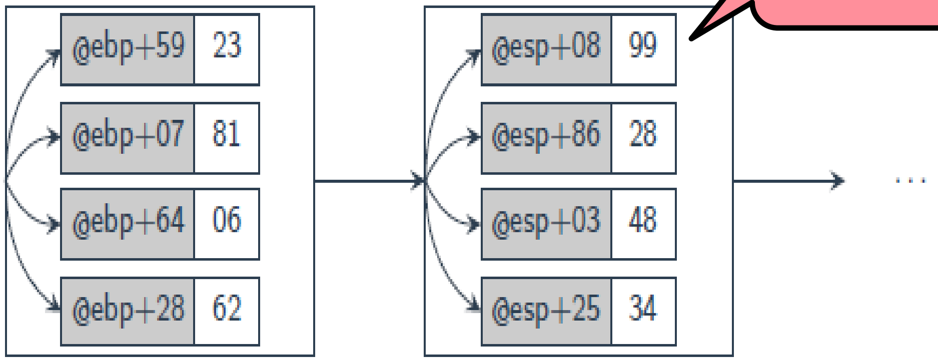
- Scale
- Good when only few bases

- Perfect for constant case

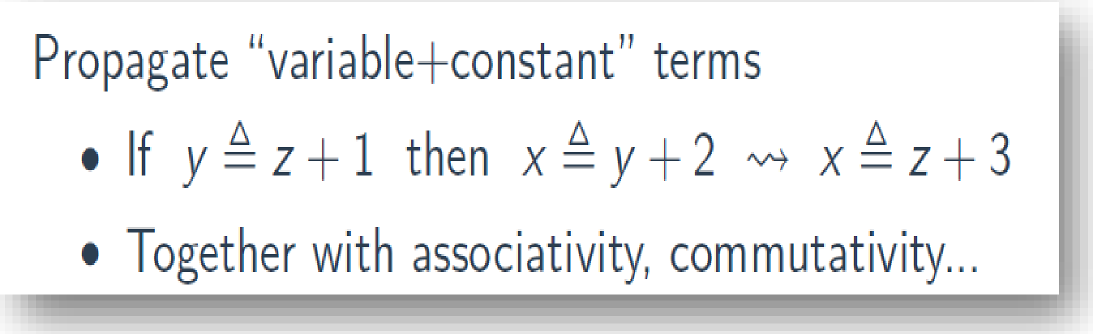
Still limited by term-equality reasoning

Fast array simplification (2)

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



- Scale
- Good when only few bases



Propagate “variable+constant” terms

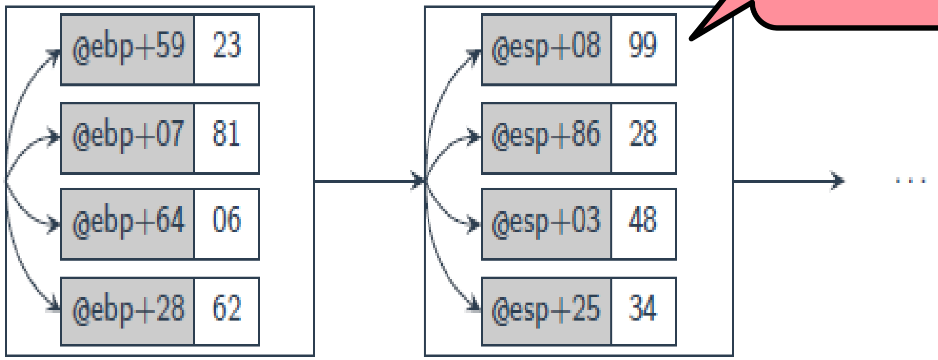
- If $y \triangleq z+1$ then $x \triangleq y+2 \rightsquigarrow x \triangleq z+3$
- Together with associativity, commutativity…

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

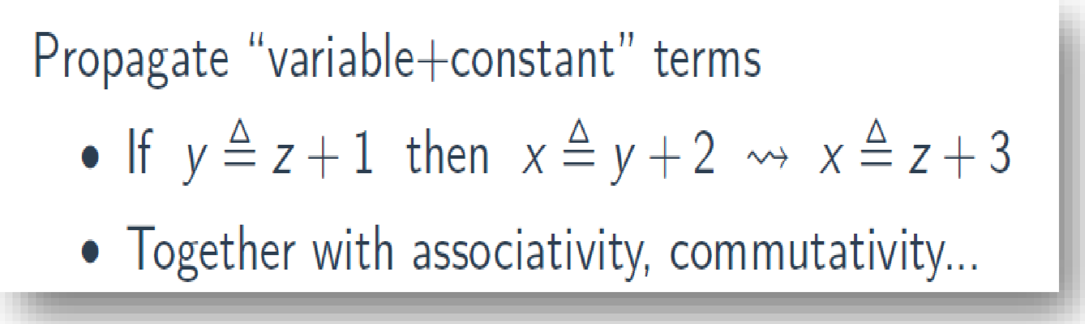
Still limited by disequality reasoning

Fast Array Simplification

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



- Scale
- Good when only few bases

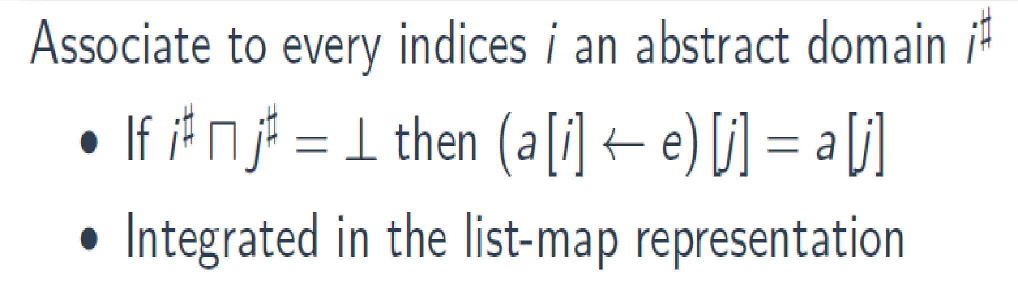


Propagate "variable+constant" terms

- If $y \triangleq z+1$ then $x \triangleq y+2 \rightsquigarrow x \triangleq z+3$
- Together with associativity, commutativity...

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

- Prove disequalities between different bases

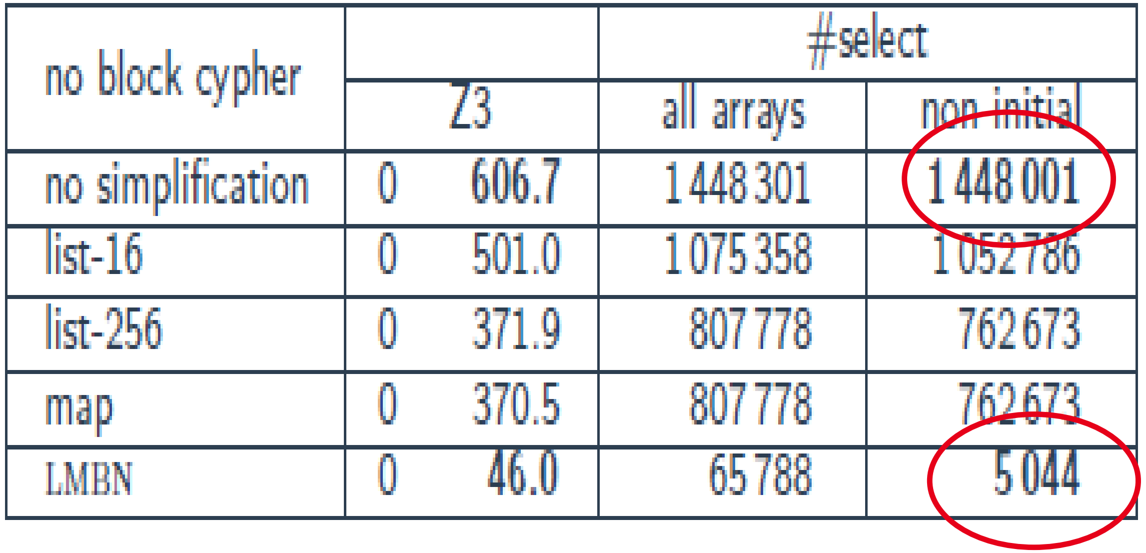


Associate to every indices i an abstract domain $i^\sharp$

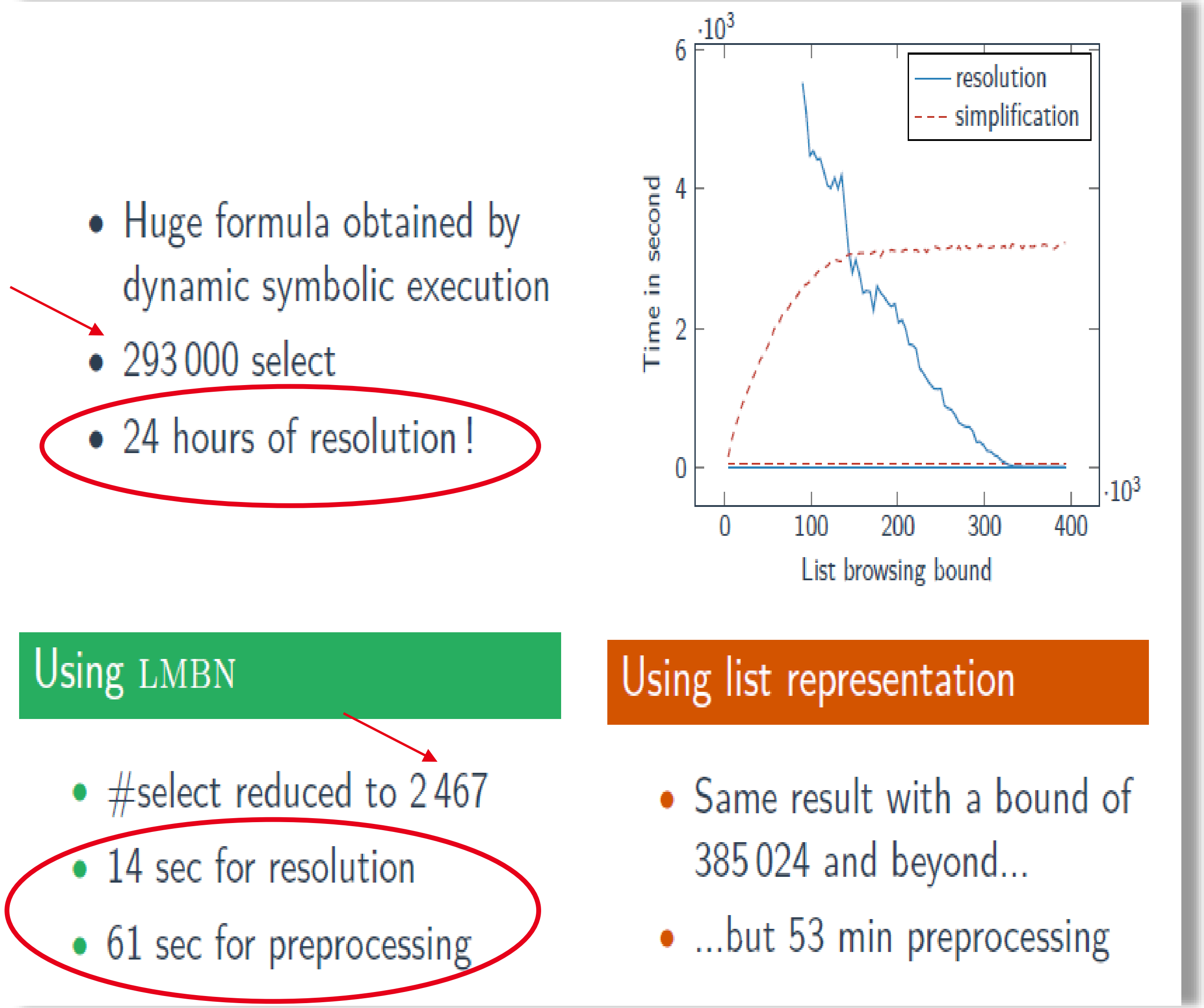
- If $i^\sharp \sqcap j^\sharp = \bot$ then $(a[i] \leftarrow e)[j] = a[j]$
- Integrated in the list-map representation

IT WORKS!

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



no block cypher			#select	
	Z3		all arrays	non initial
no simplification	0	606.7	1 448 301	1 448 001
list-16	0	501.0	1 075 358	1 052 786
list-256	0	371.9	807 778	762 673
map	0	370.5	807 778	762 673
LMBN	0	46.0	65 788	5 044



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

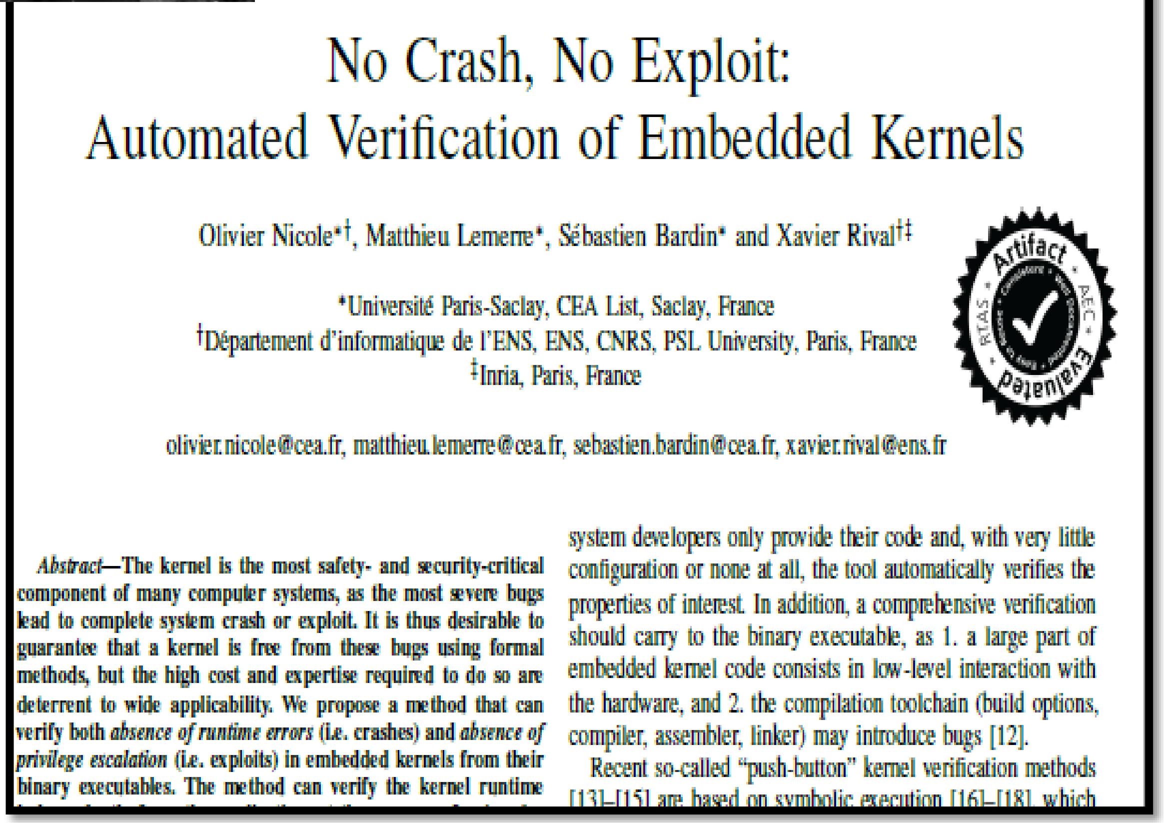
Using LMBN

- #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

Fresh results



No Crash, No Exploit: Automated Verification of Embedded Kernels

Olivier Nicole*†, Matthieu Lemerre*, Sébastien Bardin* and Xavier Rival†‡

*Université Paris-Saclay, CEA List, Saclay, France
†Département d'informatique de l'ENS, ENS, CNRS, PSL University, Paris, France
‡Inria, Paris, France

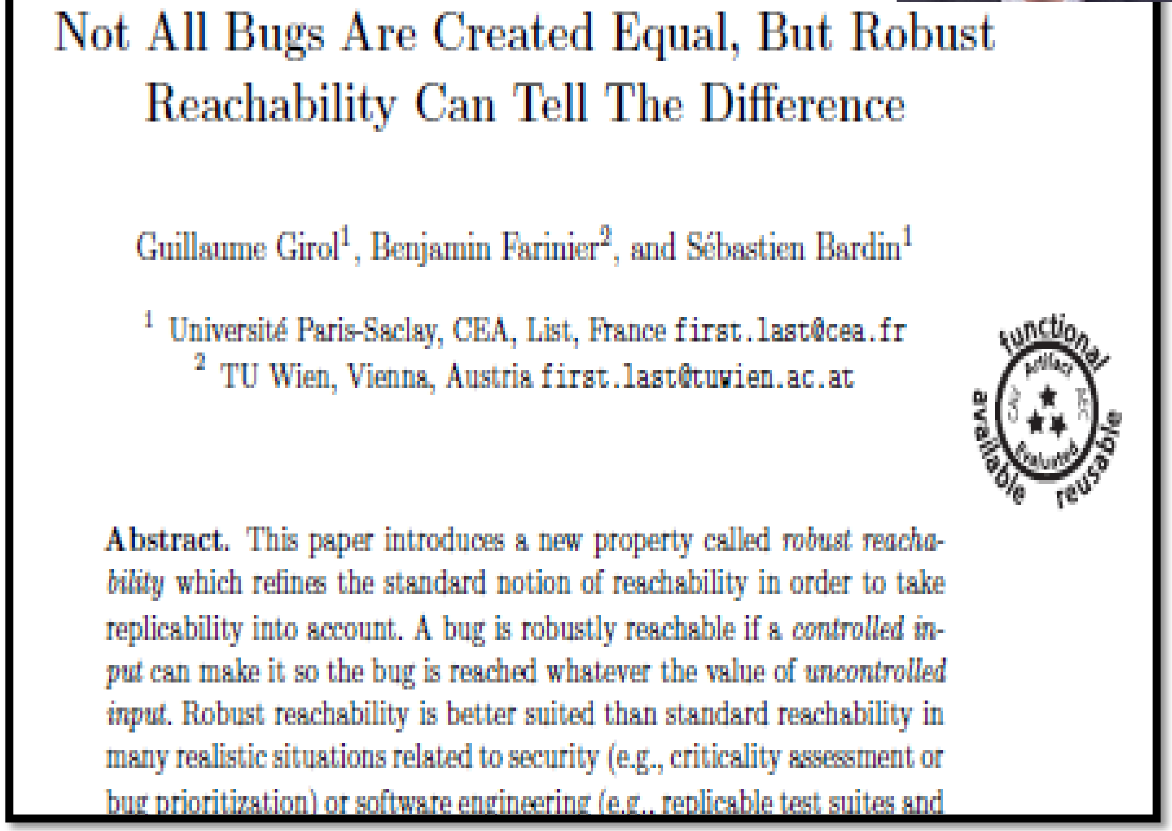
olivier.nicole@cea.fr, matthieu.lemerre@cea.fr, sebastien.bardin@cea.fr, xavier.rival@ens.fr

Abstract—The kernel is the most safety- and security-critical component of many computer systems, as the most severe bugs lead to complete system crash or exploit. It is thus desirable to guarantee that a kernel is free from these bugs using formal methods, but the high cost and expertise required to do so are deterrent to wide applicability. We propose a method that can verify both *absence of runtime errors* (i.e. crashes) and *absence of privilege escalation* (i.e. exploits) in embedded kernels from their binary executables. The method can verify the kernel runtime

system developers only provide their code and, with very little configuration or none at all, the tool automatically verifies the properties of interest. In addition, a comprehensive verification should carry to the binary executable, as 1. a large part of embedded kernel code consists in low-level interaction with the hardware, and 2. the compilation toolchain (build options, compiler, assembler, linker) may introduce bugs [12].

Recent so-called "push-button" kernel verification methods [13]–[15] are based on symbolic execution [16]–[18], which

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



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

Guillaume Girol[1], Benjamin Farinier[2], and Sébastien Bardin[1]

[1] Université Paris-Saclay, CEA, List, France first.last@cea.fr
[2] TU Wien, Vienna, Austria first.last@tuwien.ac.at

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

- Focus on robust bugs
- CAV 2021

Example 2: robust symbolic execution [CAV 2018, CAV 2021]

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

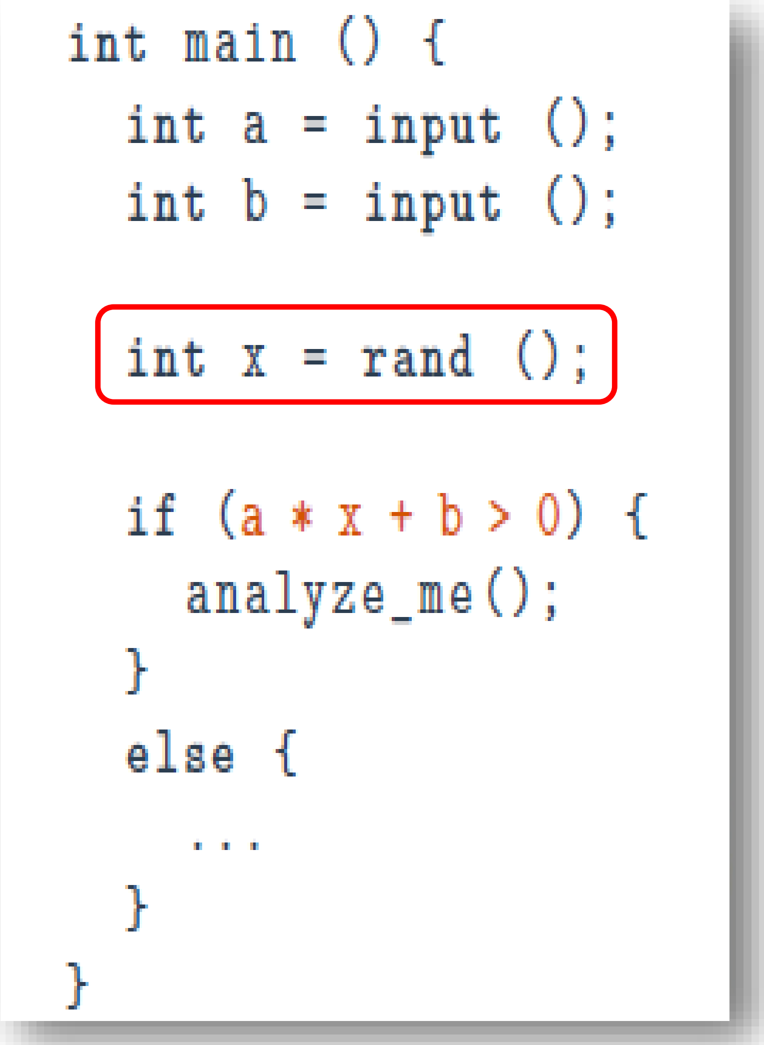
What?!!

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 {
    ...
  }
}
```

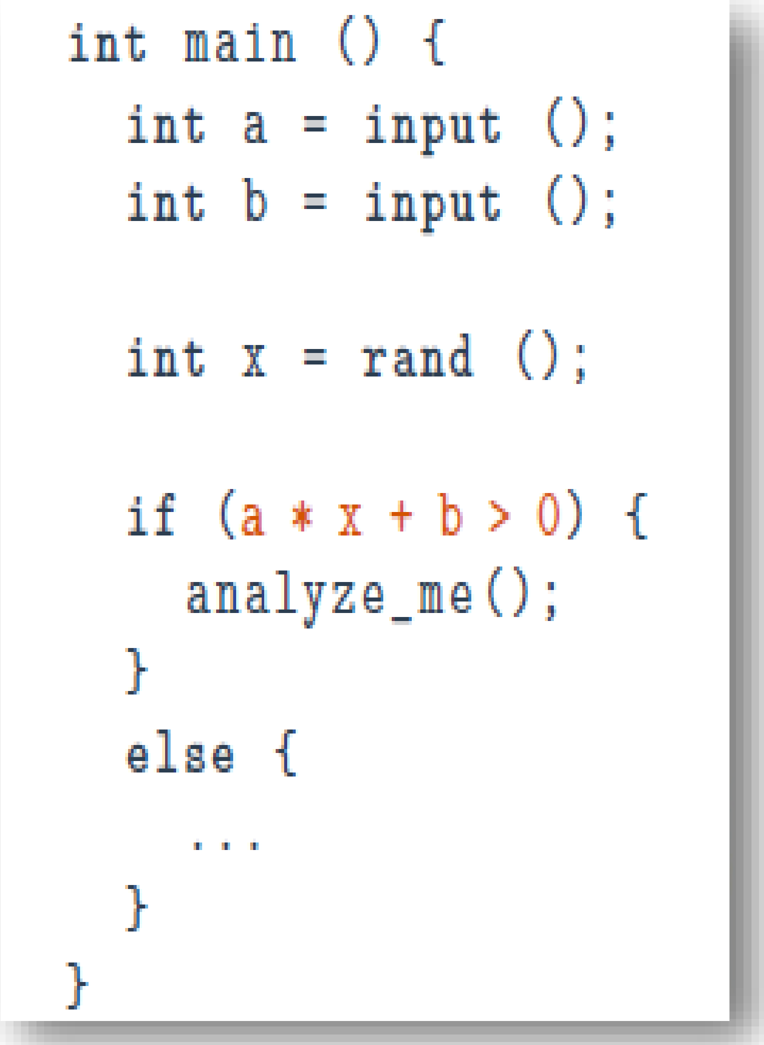
Example 2: robust symbolic execution

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

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

- **How to solve it? (CAV18)**

- **Robust reachability (CAV'21)**



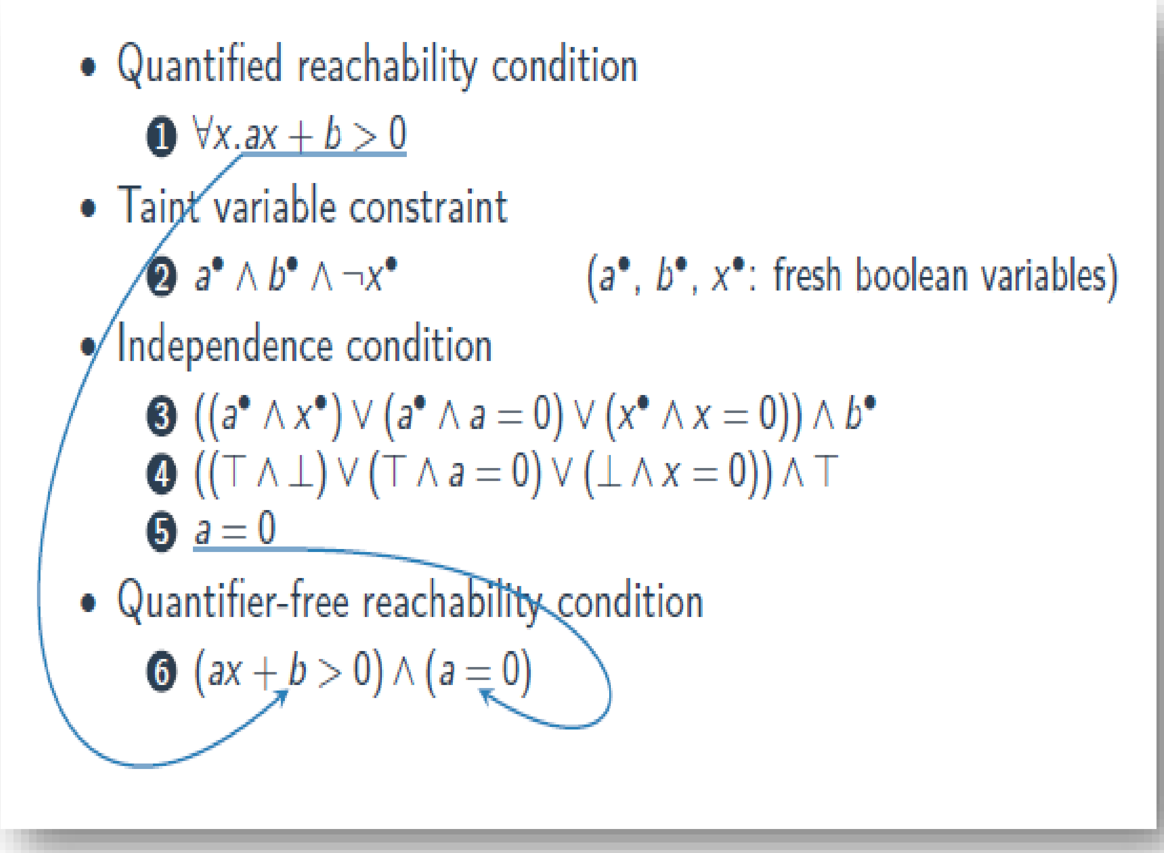
```
int main () {
  int a = input ();
  int b = input ();

  int x = rand ();

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

Example: robustness and quantification [CAV 2018]

Our solution: reduce quantified formula to the quantifier-free case

- Approximation
- But reuse the whole SMT machinery

Key insights:
- independence conditions
- formula strengthening



- Quantified reachability condition

  ❶ $\underline{\forall x. ax + b > 0}$

- Taint variable constraint

  ❷ $a^\bullet \wedge b^\bullet \wedge \neg x^\bullet$ $\qquad$ ($a^\bullet$, $b^\bullet$, $x^\bullet$: fresh boolean variables)

- Independence condition

  ❸ $((a^\bullet \wedge x^\bullet) \vee (a^\bullet \wedge a = 0) \vee (x^\bullet \wedge x = 0)) \wedge b^\bullet$

  ❹ $((\top \wedge \bot) \vee (\top \wedge a = 0) \vee (\bot \wedge x = 0)) \wedge \top$

  ❺ $\underline{a = 0}$

- Quantifier-free reachability condition

  ❻ $(ax + b > 0) \wedge (a = 0)$

OUTLINE

- Context: a little bit of formal methods for safety
- Binary-level security analysis: benefits & challenges
- The BINSEC platform
- From source-level safety to binary-level security: some examples
- Conclusions

SOME KEY PRINCIPLES BEHIND OUR WORK?

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

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

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

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

Conclusion

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

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

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

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

BINSEC is available (new release)

https://binsec.github.io

ANR Project TAVA

Looking for PhD & postdoc

sebastien.bardin@cea.fr

Software Safety and Security Laboratory
Software & Systems Engineering Department
List, CEA Tech

Florent Kirchner
florent.kirchner@cea.fr