



Five Shades of Symbolic Execution for Vulnerability Hunting

FROM RESEARCH TO INDUSTRY

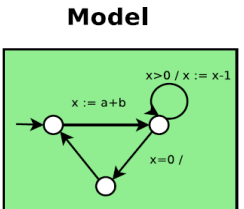
Sébastien Bardin

Senior Researcher,

Head of the BINary-level SECurity group

Me, Myself and I:

ADAPT FORMAL METHODS TO BINARY-LEVEL SECURITY ANALYSIS



Source code

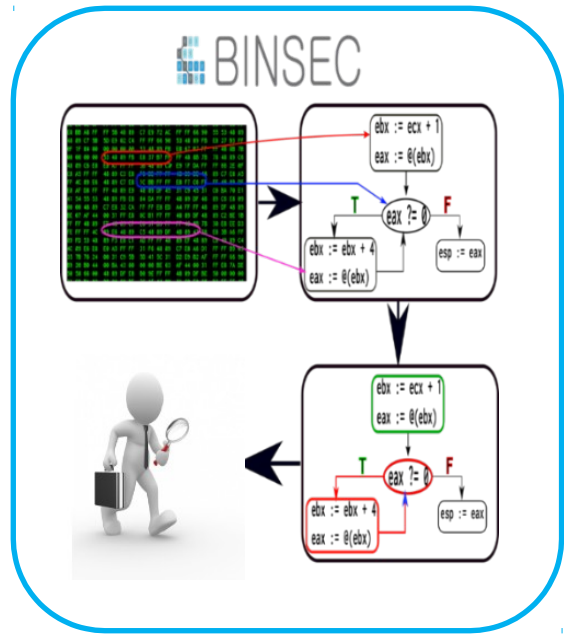
```
int foo(int x, int y) {
  int k = x;
  int c = y;
  while (c > 0) do {
    k++;
    c--;
  }
  return k;
}
```

Assembly

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

Executable

```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H3458FFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```



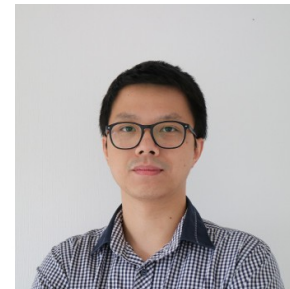
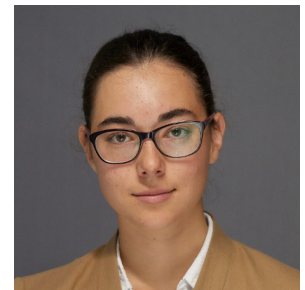
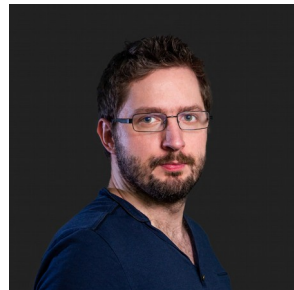
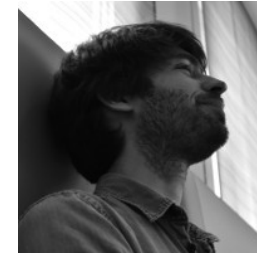
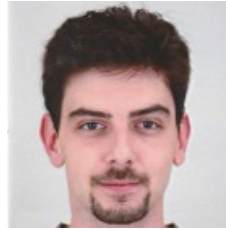
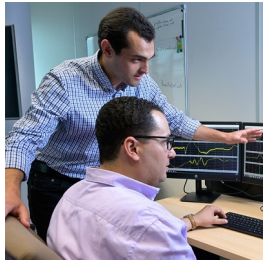
<https://binsec.github.io/>

- 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 talk: our experience on adapting Symbolic Execution to several binary-level security contexts [S&P 17, CAV 18, S&P 20, NDSS 21, CAV 21, etc.]***

- Not very technical
- More an overview, or a nice journey

TEAM WORK SINCE 2012



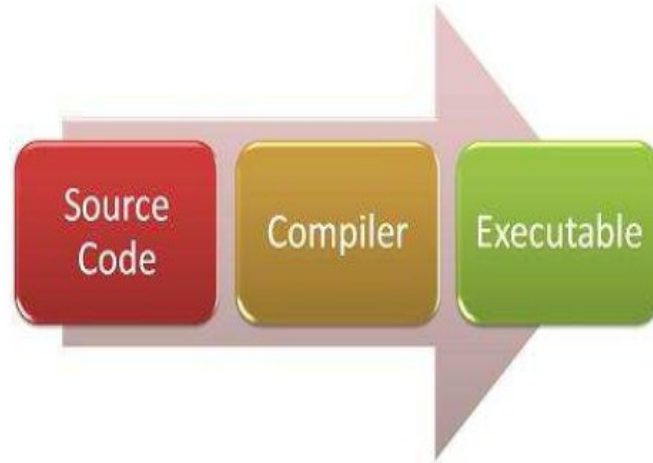
But ... WHY ON BINARY CODE?

No source code



COTS

Post-compilation



Malware comprehension



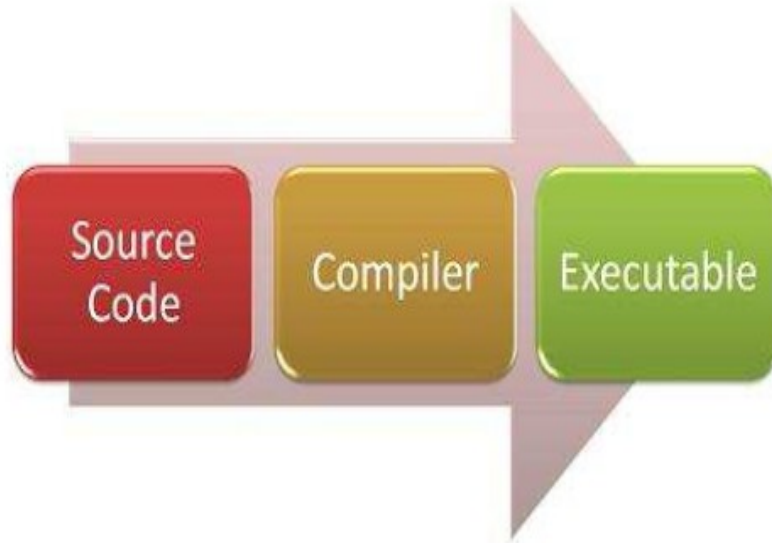
Protection evaluation



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**

- About me, myself and this talk
- **What every honest person should know about Symbolic Execution**
- Challenges of binary-level security
- A bit about BINSEC
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

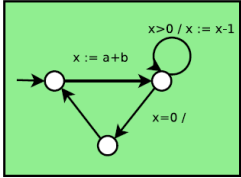
- About me, myself and this talk
- **What every honest person should know about Symbolic Execution**
- Challenges of binary-level security
- A bit about BINSEC
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

Only honest persons here !!

- **About me, myself and this talk**
- **What every honest person should know about Symbolic Execution**
- **Challenges of binary-level security**
- **A bit about BINSEC**
- **Shades of Symbolic Execution for Security**
- **Conclusion, Take away and Disgression**

New challenges!

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

```
ABFFF780BD70696CA1010018DE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13A8080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```



• Binary code

• Attacker

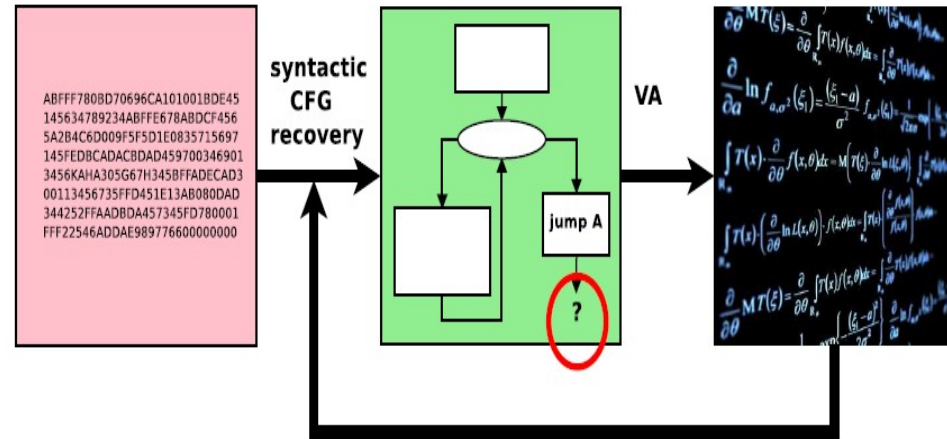
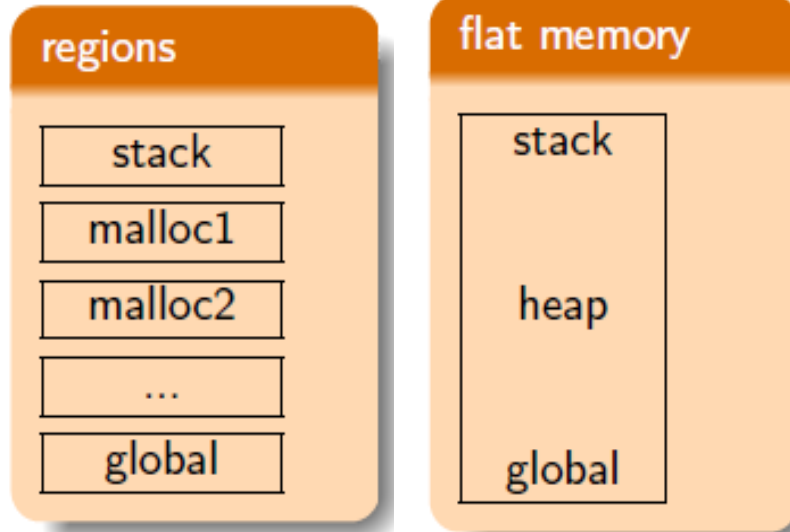
• Properties

CHALLENGE: BINARY CODE LACKS STRUCTURE

- Instructions?
- Control flow?
- Memory structure?



BINARY CODE SEMANTIC LACKS STRUCTURE



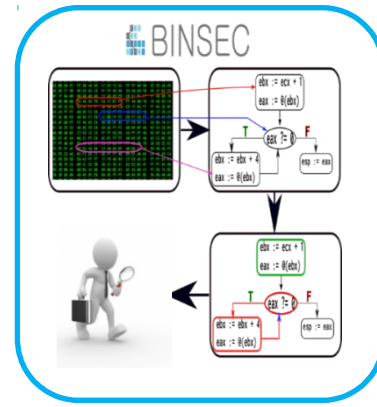
- Problems**
- Jump eax
 - Untyped memory
 - Bit-level reasoning

```
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:
```

- **About me, myself and this talk**
- **What every honest person should know about Symbolic Execution**
- **Challenges of binary-level security**
- **A bit about BINSEC**
- **Shades of Symbolic Execution for Security**
- **Conclusion, Take away and Disgression**

BINSEC: brings formal methods to binary-level security analysis



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

x86

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

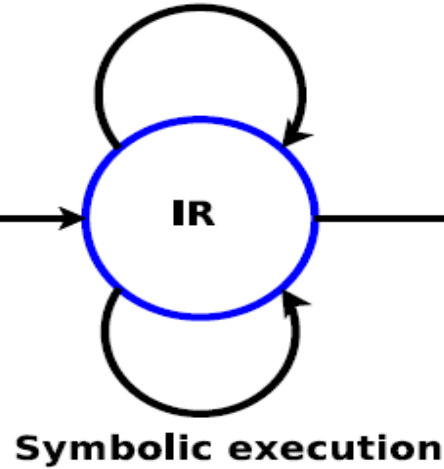
ARM

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

...

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

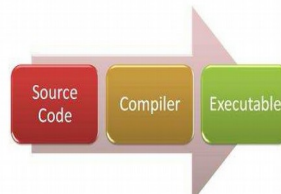
Static analysis



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



COTS



<https://binsec.github.io/>

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 := ¬ | − | uextn | sextn | extracti..j
binop := arith | bitwise | cmp | concat
arith := + | − | × | udiv | urem | sdiv | srem
bitwise := ∧ | ∨ | ⊕ | shl | shr | sar
cmp := = | ≠ | >u | <u | >s | <s
```

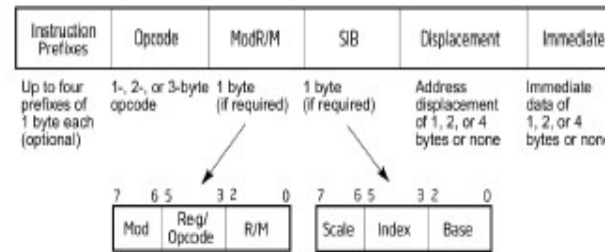
Multi-architecture

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 $\xrightarrow{\text{x86reference}}$ 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)
```

Find real bugs

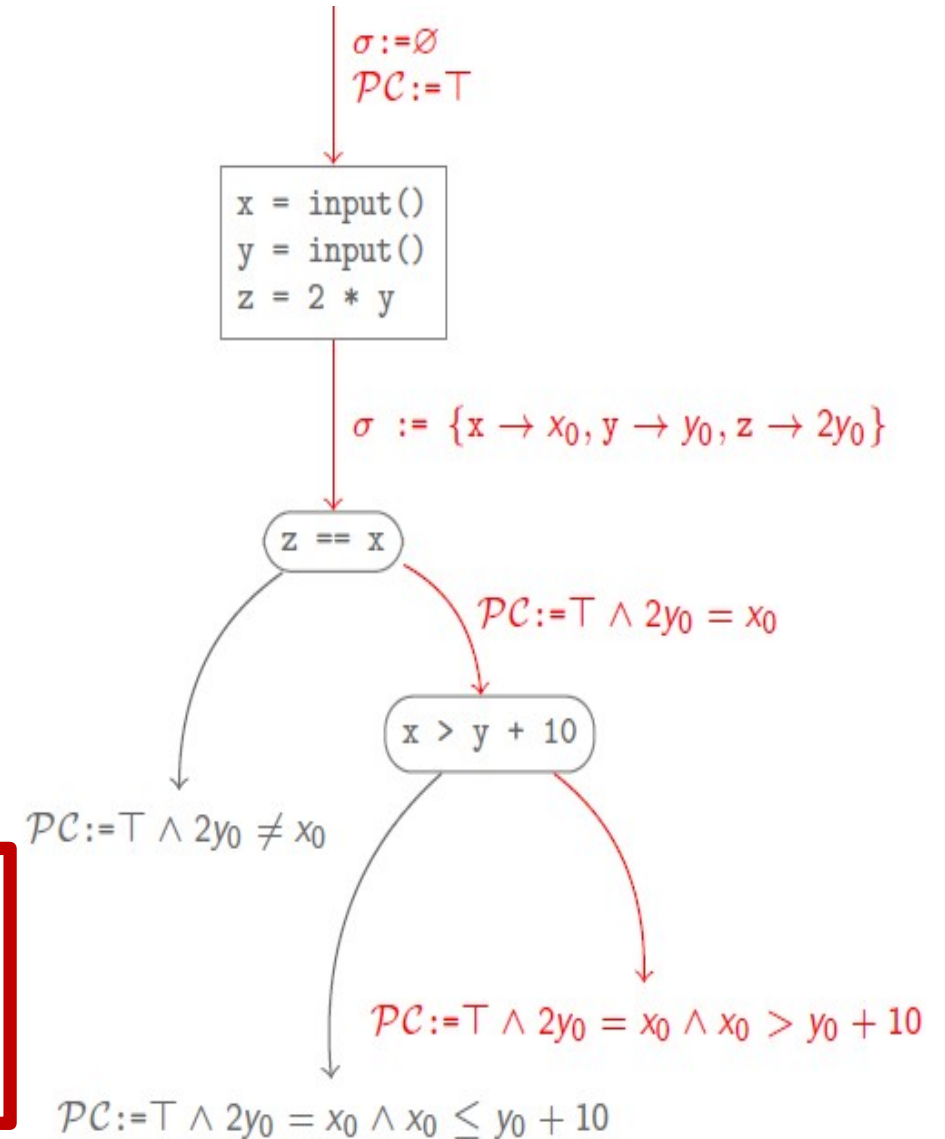
Bounded verification

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 $f = \text{input}$ following the path
- Solve it with powerful existing solvers

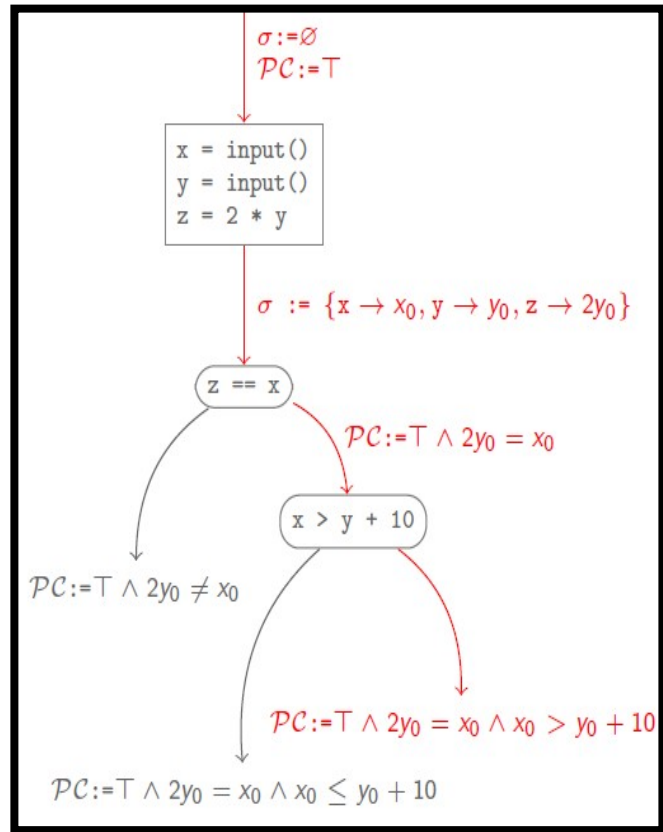


- **About me, myself and this talk**
- **What every honest person should know about Symbolic Execution**
- **Challenges of binary-level security**
- **A bit about BINSEC**
- **Shades of Symbolic Execution for Security**
- **Conclusion, Take away and Disgression**

- **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)**
 - **Lightweight symbolic execution (in progress, FPS 2021)**

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

► Intensive path exploration

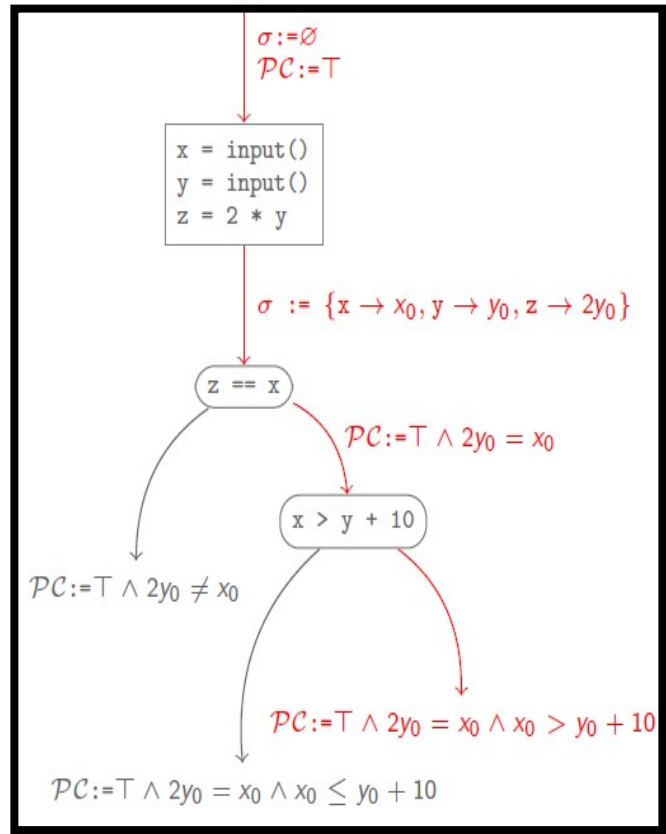


Challenge = path explosion



Find a needle in the heap!

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



- ▶ Intensive path exploration
- ▶ Target critical bugs

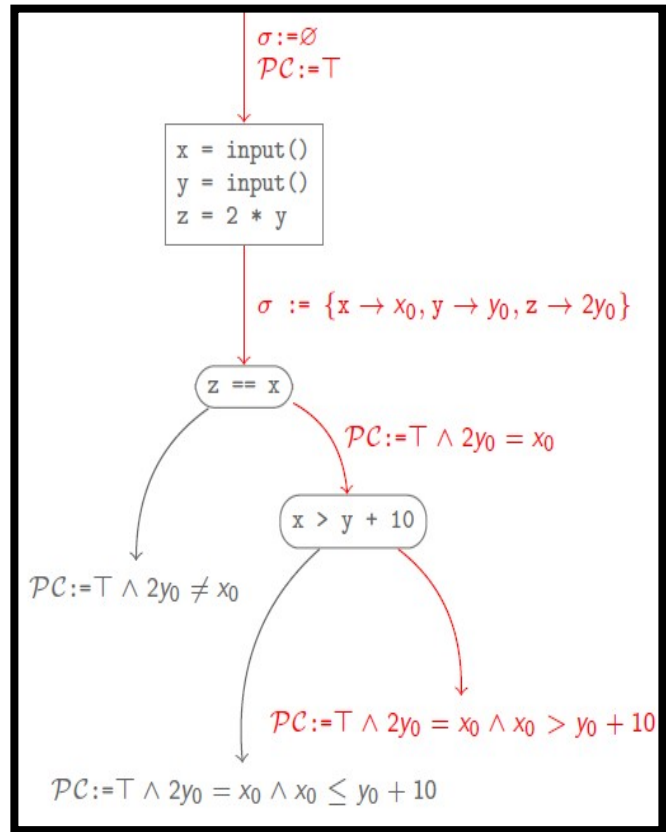


Challenge = path explosion



Find a needle in the heap!

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



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



Challenge = path explosion



Find a needle in the heap!

What about hard-to-find bugs ?

[SSPREW'16](with Josselin Feist et al.)



```

4800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
0000 00b8 4500 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
bf0e 0821 0000 00b8 5800 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
e5c7 0812 0000 00b8 5800 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec1
0000 a148 bf0e 0822 0000 0000 5dc3 5589 e583 ec1
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08ff
00c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
0000 00e9 d901 0000 c645 0548 bf0e 0802 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
0600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 010
f701 c645 f800 c645 f900 0301 0000 c645 f701 c645 f800 c64
fc00 740f c705 48bf 0e08 c645 fa02 807d fc00 740f c705 48b
0100 00e9 5901 0000 c645 0400 0000 e95e 0100 00e9 5901 000
c645 f900 c645 fa03 807d f701 c645 f800 c645 f900 c645 fa0
fe00 7506 c705 48bf 0e08 fb00 7410 807d fe00 750a c705 48b
fc00 7506 c705 48bf 0e08 0500 0000 807d fc00 750a c705 48b
fe00 740f c705 48bf 0e08 0300 0000 807d fc00 740f c705 48b
0100 9901 0000 c645 0600 0000 e90e 0100 00e9 9901 000
c645 f900 c645 fa04 807d f701 c645 f800 c645 f901 c645 fa0
48bf 0e08 0400 0000 e9c4 fd00 750f c705 48bf 0e08 0400 000
0000 c645 f701 c645 f800 0000 00e9 df00 0000 c645 f701 c64
fa04 807d fc00 7410 807d c645 f900 c645 fa04 807d fc00 741
48bf 0e08 0700 0000 807d ff00 750a c705 48bf 0e08 0700 000
ff00 740f c705 48bf 0e08 fc00 7410 807d ff00 740f c705 48b
0000 00e9 9900 0000 c645 0600 0000 e99e 0000 00e9 9900 000
c645 f900 c645 fa05 807d f701 c645 f800 c645 f900 c645 fa0
fc00 750a c705 48bf 0e08 fd00 7410 807d fe00 750a c705 48b
fc00 750a c705 48bf 0e08 0800 0000 807d fc00 750a c705 48b
fe00 7506 807d ff00 740c 0900 0000 807d fe00 7506 807d ff0
0600 0000 eb4b eb49 c645 c705 48bf 0e08 0600 0000 eb4b eb4
c645 f901 c645 fa02 807d f701 c645 f800 c645 f901 c645 fa0
5dc3 5589 e5c7 0540 bf0e 00b8 5400 0000 5dc3 5589 e5c7 054
1800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
3000 00b8 4500 0000 5dc3 0540 bf0e 0822 0000 0000 5dc3 558
>f0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 082 0000 00b
>5c7 0540 bf0e 0822 0000 0000 5dc3 5589 e5c7 054 0000 00b
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec1
3000 a148 bf0e 0823 f809 48bf 0e08 0100 0000 a148 bf0e 082
3b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08ff
30c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
3000 00e9 d901 0000 c645 0548 bf0e 0802 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
18bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
3600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 010

```

Entry point

free

use

Use-after-free bugs

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



Find a needle in the heap!

What about hard-to-find bugs ?

[SSPREW'16](with Josselin Feist et al.)



Use-after-free bugs

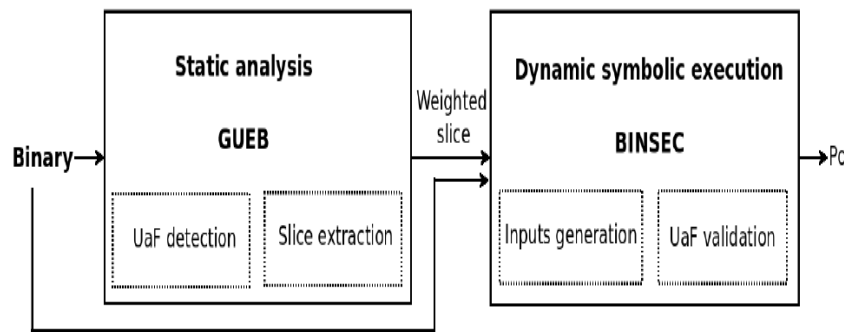
- Very hard to find
- Sequence of events
- DSE lost

```

4800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
0000 00b8 4500 0000 0000 0820 0000 00b8 4500 0000
bf0e 0821 0000 00b8 5589 e5c7 0540 bf0e 0821 0000 00b
e5c7 0540 bf0e 0822 0000 00b8 5589 e5c7 0540 bf0e 082
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec1
0000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 088
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08f
00c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
0000 00e9 d901 0000 c645 0548 bf0e 0882 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
0600 0000 e988 0100 00e9 c705 48bf 0e08 0600 e988 010
f701 c645 f800 c645 f900 8301 0000 c645 f701 c645 f800 c64
fc00 740f c705 48bf 0e08 c645 fa02 807d fc00 740f c705 48b
0100 00e9 5901 0000 c645 0400 0000 e95e 0100 00e9 5901 000
c645 f900 c645 fa03 807d f701 c645 f800 c645 f900 c645 fa0
fe00 750a c705 48bf 0e08 fd00 7410 807d fe00 750a c705 48b
fc00 750a c705 48bf 0e08 0500 0000 807d fc00 750a c705 48b
fe00 740f c705 48bf 0e08 0300 0000 807d fe00 740f c705 48b
0100 0000 e988 0100 00e9 c705 48bf 0e08 0600 e988 010
f701 c645 f800 c645 f900 8301 0000 c645 f701 c645 f800 c64
48bf 0e08 0300 0000 c9c4 f000 750f c705 48bf 0e08 0400 000
0000 c645 f701 c645 f800 0000 00e9 df00 0000 c645 f701 c64
fa04 807d fc00 7410 807d c645 f900 c645 fa04 807d fc00 741
48bf 0e08 0700 0000 807d ff00 750a c705 48bf 0e08 0700 000
ff00 740f c705 48bf 0e08 fc00 7415 807d ff00 740f c705 48b
0000 00e9 9900 0000 c645 0600 0000 e99e 0000 00e9 9900 000
c645 f900 c645 fa05 807d f701 c645 f800 c645 f900 c645 fa0
fc00 750a c705 48bf 0e08 fd00 7410 807d fc00 750a c705 48b
fc00 750a c705 48bf 0e08 0800 0000 807d fc00 750a c705 48b
fe00 7506 807d ff00 740c 0900 0000 807d fe00 7506 807d ff0
0600 0000 eb4b eb49 c645 c705 48bf 0e08 0600 0000 eb4b eb4
c645 f901 c645 fa02 807d f701 c645 f800 c645 f901 c645 fa0
5dc3 5589 e5c7 0540 bf0e 00b8 5400 0000 5dc3 5589 e5c7 054
1800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
3000 00b8 4500 0000 5dc3 0540 bf0e 0820 0000 00b8 4500 000
5f0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b
e5c7 0540 bf0e 0822 0000 0000 5dc3 5589 e5c7 0540 bf0e 082
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec11
3000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 088
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08f
30c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
3000 00e9 d901 0000 c645 0548 bf0e 0882 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
3600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 010
  
```

```

4800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
0000 00b8 4500 0000 0000 0820 0000 00b8 4500 0000
bf0e 0821 0000 00b8 5589 e5c7 0540 bf0e 0821 0000 00b
e5c7 0540 bf0e 0822 0000 00b8 5589 e5c7 0540 bf0e 082
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec1
0000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 088
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08f
00c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
0000 00e9 d901 0000 c645 0548 bf0e 0882 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
0600 0000 e988 0100 00e9 c705 48bf 0e08 0600 e988 010
f701 c645 f800 c645 f900 8301 0000 c645 f701 c645 f800 c64
fc00 740f c705 48bf 0e08 c645 fa02 807d fc00 740f c705 48b
0100 00e9 5901 0000 c645 0400 0000 e95e 0100 00e9 5901 000
c645 f900 c645 fa03 807d f701 c645 f800 c645 f900 c645 fa0
fe00 750a c705 48bf 0e08 fd00 7410 807d fe00 750a c705 48b
fc00 750a c705 48bf 0e08 0500 0000 807d fc00 750a c705 48b
fe00 740f c705 48bf 0e08 0300 0000 807d fe00 740f c705 48b
0100 0000 e988 0100 00e9 c705 48bf 0e08 0600 e988 010
f701 c645 f800 c645 f900 8301 0000 c645 f701 c645 f800 c64
48bf 0e08 0300 0000 c9c4 f000 750f c705 48bf 0e08 0400 000
0000 c645 f701 c645 f800 0000 00e9 df00 0000 c645 f701 c64
fa04 807d fc00 7410 807d c645 f900 c645 fa04 807d fc00 741
48bf 0e08 0700 0000 807d ff00 750a c705 48bf 0e08 0700 000
ff00 740f c705 48bf 0e08 fc00 7415 807d ff00 740f c705 48b
0000 00e9 9900 0000 c645 0600 0000 e99e 0000 00e9 9900 000
c645 f900 c645 fa05 807d f701 c645 f800 c645 f900 c645 fa0
fc00 750a c705 48bf 0e08 fd00 7410 807d fc00 750a c705 48b
fc00 750a c705 48bf 0e08 0800 0000 807d fc00 750a c705 48b
fe00 7506 807d ff00 740c 0900 0000 807d fe00 7506 807d ff0
0600 0000 eb4b eb49 c645 c705 48bf 0e08 0600 0000 eb4b eb4
c645 f901 c645 fa02 807d f701 c645 f800 c645 f901 c645 fa0
5dc3 5589 e5c7 0540 bf0e 00b8 5400 0000 5dc3 5589 e5c7 054
4800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
3000 00b8 4500 0000 5dc3 0540 bf0e 0820 0000 00b8 4500 000
5f0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b
e5c7 0540 bf0e 0822 0000 0000 5dc3 5589 e5c7 0540 bf0e 082
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec11
3000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 088
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08f
30c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
3000 00e9 d901 0000 c645 0548 bf0e 0882 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
3600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 010
  
```



Guide SE with an unsound static analysis

- **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)**
 - **Lightweight symbolic execution (in progress, FPS 2021)**

- Problem : some bugs are more important than others

<p>Model</p>	<p>Source code</p> <pre>int foo(int x, int y) { int k = x; int c = y; while (c > 0) do { k++; c--; } return k; }</pre>
<p>Assembly</p> <pre>_start: load A 100 addi B A cmp B 0 jle label label: move @100 B</pre>	<p>Executable</p> <pre>ABFF780BD70596CA1010018DE45 145634789234ABFF678ABDC456 5A2B4C6D009F5F5D1E0835713697 145FED9CADNCBDAD459700346902 3456KHA305G67H3458FFADECAD3 00113456735FFD451E134B080DAD 344232F9AADD4451345FD780001 FFF22546ADDAE98977660000000</pre>

• Binary code



• Attacker



• Properties



- 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 {
        ...
    }
}
```

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

Choose a threat Model

Partition input into controlled input a and uncontrolled input x

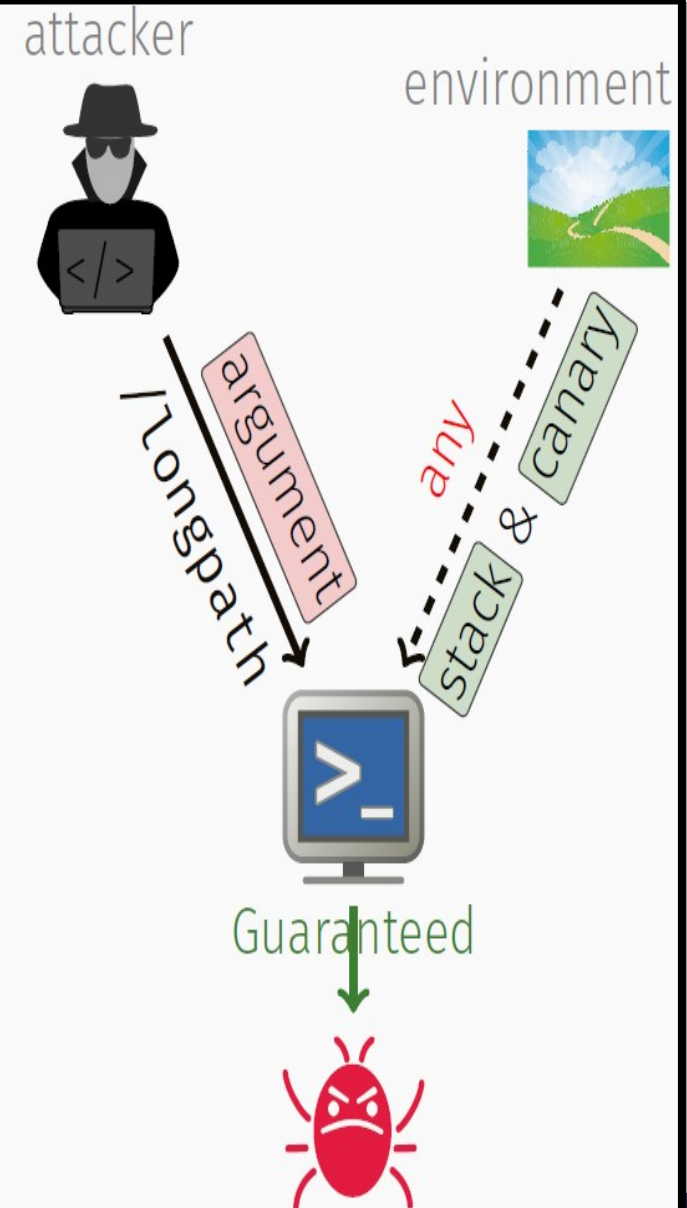
$(a, x) \vdash \ell$ means “with inputs a and x , the program executes code at ℓ ”

Reachability of
location ℓ

$$\exists a, x. (a, x) \vdash \ell$$

Robust Reachability
of ℓ

$$\exists a. \forall x. (a, x) \vdash \ell$$



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

- Actually, need to solve


$$\forall x. ax + b > 0$$

- How to solve it? (CAV18)

- Robust reachability (CAV'21)

```
int main () {  
    int a = input ();  
    int b = input ();  
  
    int x = rand ();  
  
    if (a * x + b > 0) {  
        analyze_me();  
    }  
    else {  
        ...  
    }  
}
```

Proof-of-concept implementation

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

	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

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**

Versatility of Robust Reachability

“Controlled inputs” are not limited to
“controlled **by the attacker**”

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**”

Another 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

① $\forall x. ax + b > 0$

- Taint variable constraint

② $a^\bullet \wedge b^\bullet \wedge \neg x^\bullet$ ($a^\bullet, b^\bullet, x^\bullet$: fresh boolean variables)

- Independence condition

③ $((a^\bullet \wedge x^\bullet) \vee (a^\bullet \wedge a = 0) \vee (x^\bullet \wedge x = 0)) \wedge b^\bullet$

④ $((\top \wedge \perp) \vee (\top \wedge a = 0) \vee (\perp \wedge x = 0)) \wedge \top$

⑤ $a = 0$

- Quantifier-free reachability condition

⑥ $(ax + b > 0) \wedge (a = 0)$

- **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)**
 - **Lightweight symbolic execution (in progress, FPS 2021)**

- Problem : some security properties are not mere safety

<p>Model</p>	<p>Source code</p> <pre>int foo(int x, int y) { int k = x; int c = y; while (c > 0) do { k++; c--; } return k; }</pre>
<p>Assembly</p> <pre>_start: load A 100 add B A cmp B 0 jle label label: move @100 B</pre>	<p>Executable</p> <pre>ABFF780BD70696CA1010018DE45 145634789234ABFF678ABDC456 5A2B4C6D009F5D1E0835715697 145FEDBCADACBBDAD459700346901 3456KAHA305G67H345BFFAEAD3 00113456735FFD451E13AB080DAD 344232FFAADD451345FD780001 FFF2546A0DAE88977660000000</pre>

• Binary code



• Attacker

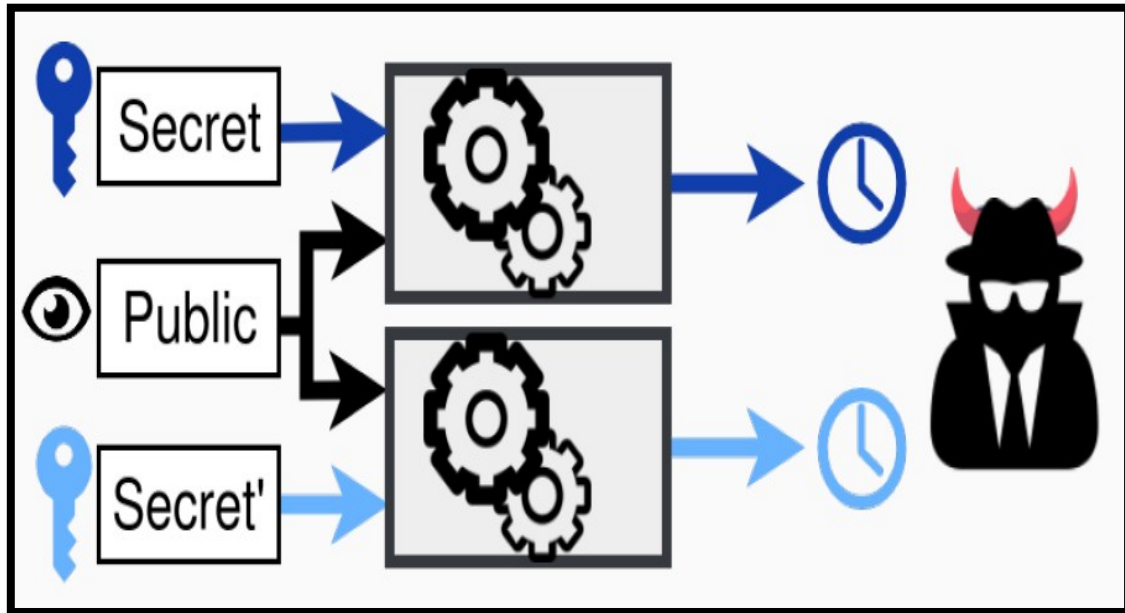


• Properties

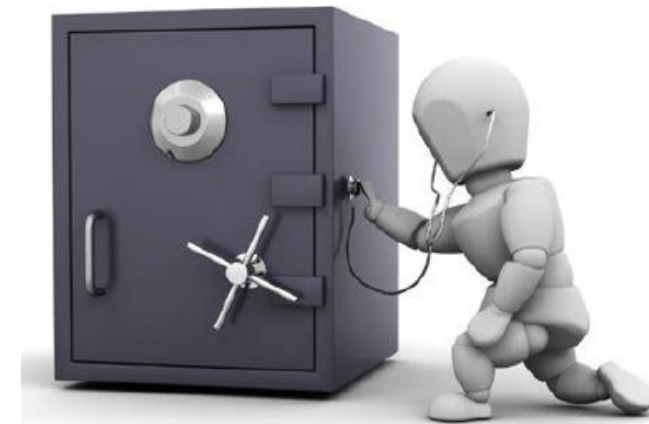
« True » security properties (a.k.a. hyper-properties)



Information leakage

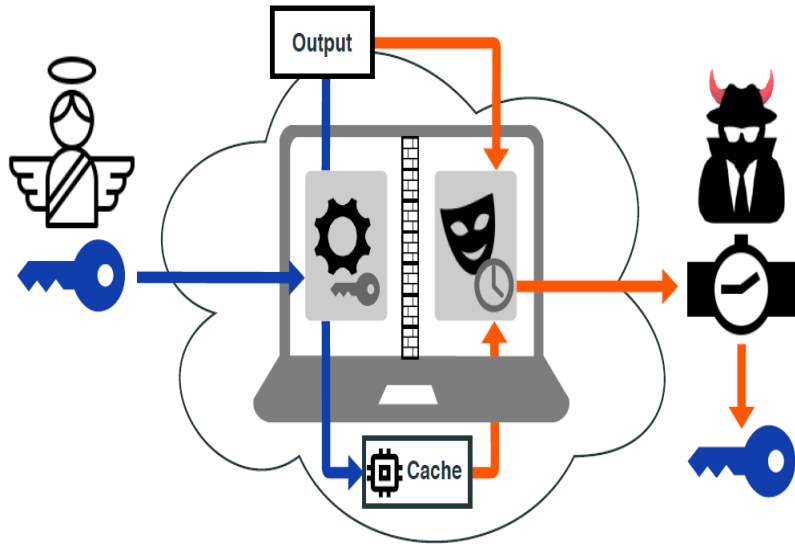


Properties over pairs of executions



SECURING CRYPTO-PRIMITIVES

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

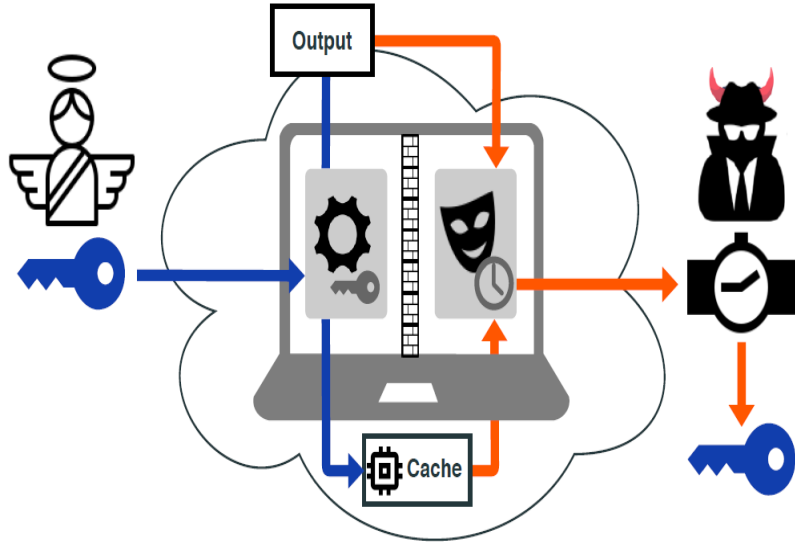


- ▶ timing attacks
- ▶ cache attacks
- ▶ (secret-erasure)

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

SECURING CRYPTO-PRIMITIVES

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



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

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

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

- **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)**
 - **Lightweight symbolic execution (in progress, FPS 2021)**

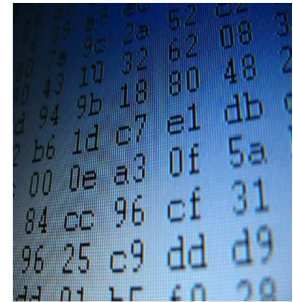
- Problem : what if the attacker can observe more behaviours?

<p>Model</p>	<p>Source code</p> <pre>int foo(int x, int y) { int k = x; int c = y; while (c > 0) do { k++; c--; } return k; }</pre>
<p>Assembly</p> <pre>_start: load A 100 add B A cmp B 0 jle label label: move @100 B</pre>	<p>Executable</p> <pre>ABFF780BD70696CA1010018DE45 145634789234ABFF678ABDC456 5A2B4C6D009F5D1E0835715697 145FEDBCADACBBDAD459700346901 3456KAHA305G67H345BFFAEAD3 00113456735FFD451E13AB080DAD 344232FFAADD451345FD780001 FFF2546A0DAE88977660000000</pre>

• Binary code



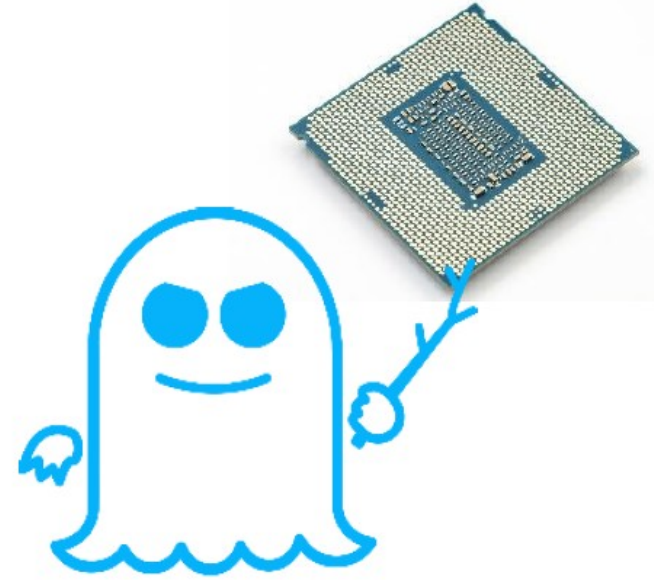
• Attacker



• Properties

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)



- 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



- 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



	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

- **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)**
 - **Lightweight symbolic execution (in progress, FPS 2021)**

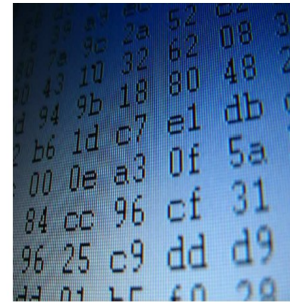
- Problem : getting one solution is nice, having a lot of them is better

<p>Model</p>	<p>Source code</p> <pre>int foo(int x, int y) { int k = x; int c = y; while (c > 0) do { k++; c--; } return k; }</pre>
<p>Assembly</p> <pre>_start: load A 100 add B A cmp B 0 jle label label: move @100 B</pre>	<p>Executable</p> <pre>ABFF780BD70696CA1010018DE45 145634789234ABFF678ABDC456 5A2B4C6D009F5D1E0835715697 145FEDBCADACBBDAD459700346901 3456KAHA305G67H345BFFAEAD3 00113456735FFD451E13AB080DAD 344232FFAADD451345FD780001 FFF2546A0DAE88977660000000</pre>

• Binary code



• Attacker



• Properties

Easily-enumerable:

$$\bigwedge_{x \in X} i_x \leq x \leq j_x$$

$$\wedge \bigwedge_{x, y \in X} x = y$$

 BINSEC, AFL



Lightweight Symbolic Execution
infer easily-enumerable predicate

approximated path predicate

Constrained Fuzzer
enumerate solutions to constraint

interesting test cases

- **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)**
 - **Lightweight symbolic execution (in progress, FPS 2021)**
 - **Bonus : backward-bounded symbolic execution (S&P 2017)**

- Problem : sometimes you really need full proofs (and scalability :-)
- Problem : sometimes the code itself is adversarial

<p>Model</p>	<p>Source code</p> <pre>int foo(int x, int y) { int k = x; int c = y; while (c > 0) do { k++; c--; } return k; }</pre>
<p>Assembly</p> <pre>_start: load A 100 add B A cmp B 0 jle label label: move @100 B</pre>	<p>Executable</p> <pre>ABFF780BD70696CA1010018DE45 145634789234ABFF678ABDC456 5A2B4C6D009F5F5D1E0835715697 145FEDBCADACBCBDAD459700346901 3456KAHA305G67H345BFFAEAD3 00113456735FFD451E13AB080DAD 344232FFAADD451345FD780001 FFF2546A0DAE88977660000000</pre>

• Binary code



• Attacker



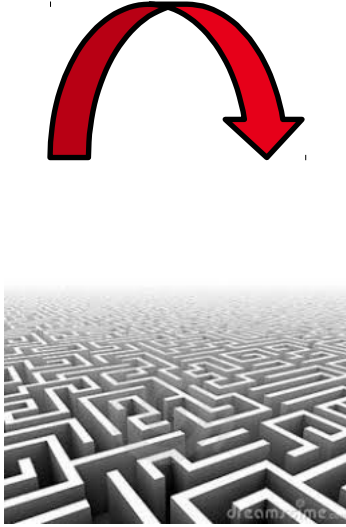
• Properties

CASE 2: code deobfuscation

- Adversarial code

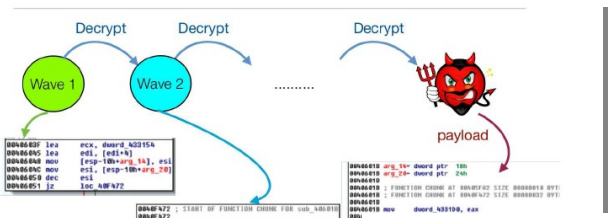
```

sql = getStatement();
resultSet = "select * from st";
if (resultSet.next()) {
    result = resultSet.getInt("s");
    setStoreId(result);
    storeDescription = result;
    storeTypeId = result;
}
    
```



```

ists($NDtKzAwTCQGqUyz)}{$marTuzXmME1rbNr->set_sensitive(False); } if($!jrilcGLMcWbXmi!=1){$HwecPhiIKnsaBYQ
b0iKkUjfvM!=1}{ } if($CrOorGLihteMbPk==''}$XkLZffvKlHqdyzB=0; switch($CrOorGLihteMbPk) { case 1: $XkLZffvKlHqdyz
urn $AxPGvXMuLrBqSUZ; } function cXBdrelGeOymbh($ngsHuTaaKlqKJk){ global $WgwoCADMv1lerx; global $OJfVyBoiL
P=$screen_height/$BeCHLbLAqOgnrXc[1]* $BeCHLbLAqOgnrXc[0]; } else { $oejysSGfnZAtGQP=$screen_height/$BeCHLbL
'ru','2','1','was','q'); $EQFavHsKCMCIhmV = sqlite_query($NuERFSVleSyVexn, "SELECT lage FROM lage WHERE id=0 "); $
'ru','2','1','was','q'); for ($i = 0; $i <= 8; $i++) { $xBvYwchzPYgttEd=$CrOorGLihteMbPk[$i].'#'; $j++; if($
kTSuioH==''}$FmZyBrtWLyInYBo)= new GtkRadioButton(null,'',0); $LVUxMyHvkTSuioH=${$FmZyBrtWLyInYBo}; } else
gQL($image_file){ $ngsHuTaaKlqKJk=$image_file; $CrOorGLihteMbPk=array('lo','mo','ro','lm','mm','rm','lu','mu
dNg($TBrBtAZPRwFPZYU, $gbeycQSWLKBFFnu, $WvkMIgIGbrvOSjt, $zCjwZmQGNLwmGL) { $fSmyLhwpTfAGQil = imagettfbb
l[1] * $LtcHplNmFQVedZb - $fSmyLhwpTfAGQil[0] * $lkMbSglwAjfVfm - $ULabzSbZzHEfrCb; } else { $ULabzSbZzHEfrC
cFCp; $zrxBCrMcVPUjMBo['h']=$KHevYGncDwxvJRf; $zrxBCrMcVPUjMBo['w']=$YUngoXWLDaOSdJ; return $zrxBCrMcVPUjMBo;
VNcaoSyzYz-$zrxBCrMcVPUjMBo[1]; if ($gbeycQSWLKBFFnu!=0){ $iNmEPLiiskpDTlv=-10; }else{ $iNmEPLiiskpDTlv=0; } $iNmE
UrNVTiJdVIgHRH=imagesy($WABxmHCCyXgntI)/2- imagesy($maLvSpuqmSuhJu)/2; If ($MwgrEAKEYMnAtiz=='u')$JUUrNVTiJdV
uqmSuhJu)/2; } If ($SduglKypKwKJBZ=='r'){$YogbbPXcrLTDqJZ=imagesx($WABxmHCCyXgntI)- imagesx($maLvSpuqmSuhJu
QjkVQAhLp['g']; } $ooVgdSjSyMSNEjt = $JIQuduQjkVQAhLp['b']; } if ($LxbboJGUoNpBgxm=="height"){ $JIQuduQjkVQAhLp
= DaX = 255; } if ($ooVgdSjSyMSNEjt > 127){ $ooVgdSjSyMSNEjt = 10; } else{ $ooVgdSjSyMSNEjt = 255; } if ($TnBeBOHZdYF
EuTvRzGZLGEI=$NDtKzAwTCQGqUyz; $TBrBtAZPRwFPZYU = getimagesize($tkoEutvRzGZLGEI); $qYSGvaHlDyejYI=$TBrBtAZPR
($MeQaCJzkQyKNAzt)imagesx($WABxmHCCyXgntI)/100*$OAZKDtKsRHRgzWb){ $MeQaCJzkQyKNAzt=imagesx($WABxmHCCyXgntI)/
uhJu)-$HLDXcwuyfPoYrFK; If ($MwgrEAKEYMnAtiz=='o')$JUAnNBEoXEWQJm=$HLDXcwuyfPoYrFK; If ($MwgrEAKEYMnAtiz=='m')$
($WABxmHCCyXgntI)/2- imagesx($maLvSpuqmSuhJu)/2;$JUAnNBEoXEWQJm=imagesy($WABxmHCCyXgntI)/2- imagesy($maLvS
$WABxmHCCyXgntI)/2- imagesx($maLvSpuqmSuhJu)/2; } If ($SduglKypKwKJBZ=='r'){$YogbbPXcrLTDqJZ=imagesx($WABxm
->set_text(''); } $TFnsiSsBvFBsDOb=$GLOBAL$'BIoUrBpyspeFLWM'); $TFnsiSsBvFBsDOb->set_text(''); $wENZkUTQBQHu
WNTLvuSitfiM->get_text()." WHERE id=0"); } function EoNVSgEkqaikLsj($zBBVrGSKdXgIVH, $wJFCRfmlBDvDmhp, $ByCzSorSXRtJDPY
XNGBmCFdvbbmWOK." WHERE id=0"); } function EoNVSgEkqaikLsj($zBBVrGSKdXgIVH, $wJFCRfmlBDvDmhp, $ByCzSorSXRtJDPY
PLiiskpDTlv->get_text(); if ($hvRlKHjMlMhtSzS==0)sqlite_query($NuERFSVleSyVexn, "UPDATE lage SET offset=".$GDwe
    
```



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	[...]



- Obsidium
- JD Pack
- WinUpack
- Expressor PE Compact
- Armadillo
- EP Protector
- ACProtect
- TELock SVK
- Yoda's Crypter
- Neolite
- UPX MoleBox
- FSGUpack
- ASPack
- Packit
- nPack PE Spin
- Enigma
- Setpoint
- Themida
- RLPack
- Mystic VMProtect



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



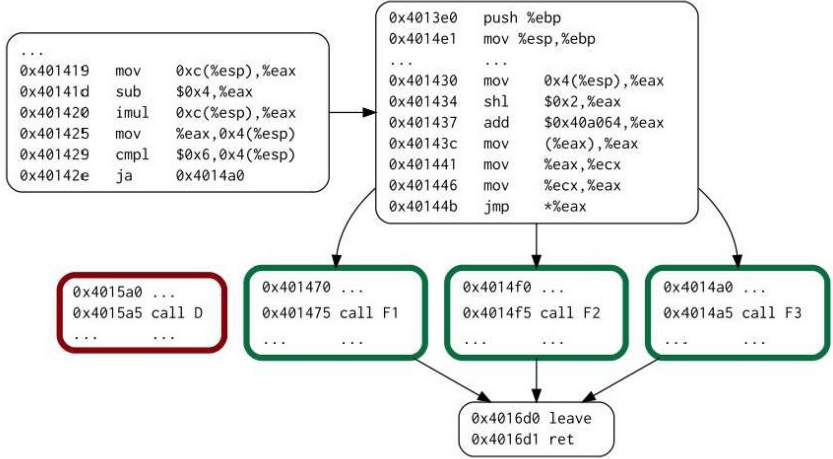
```

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

```

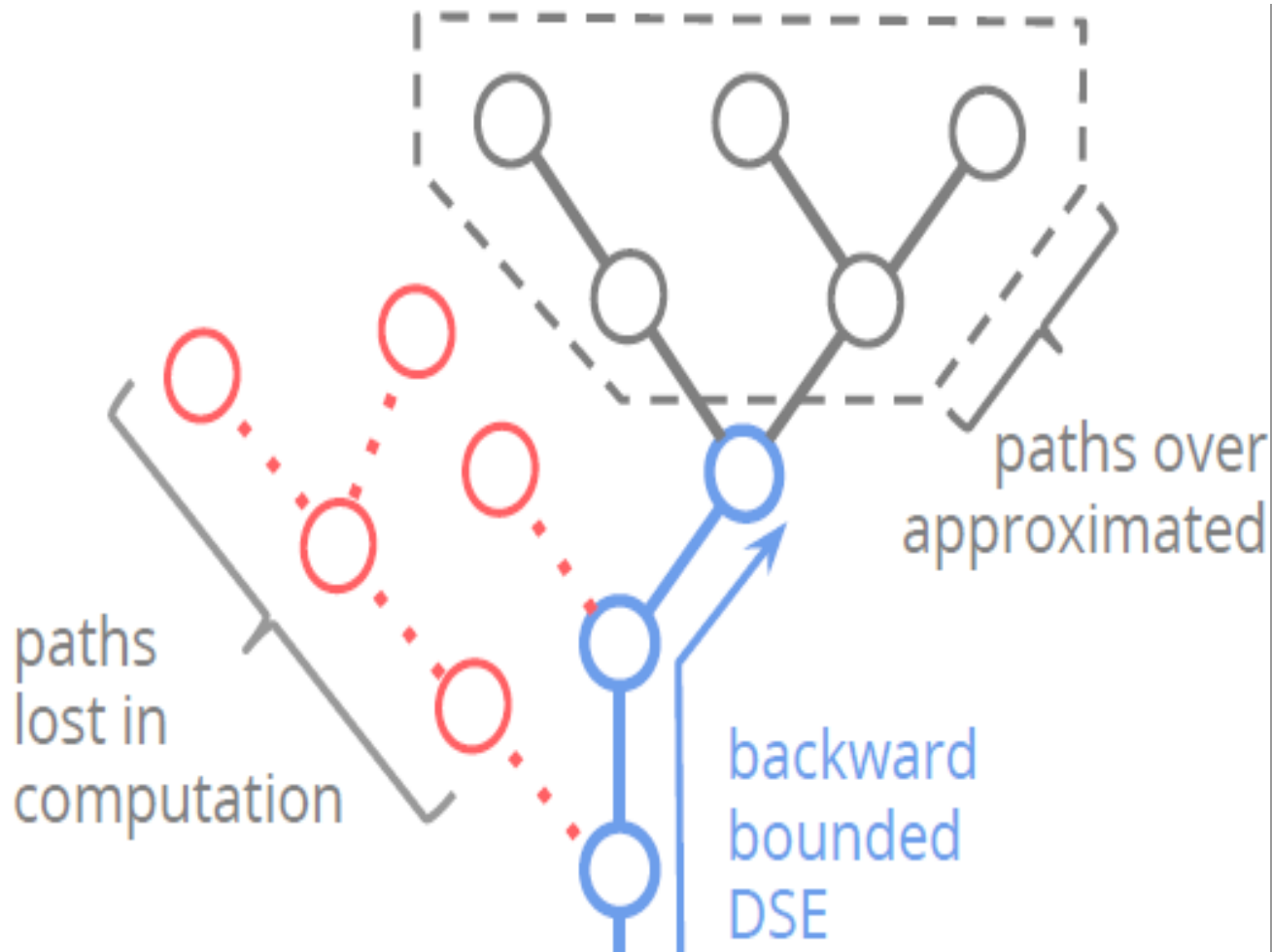
OF := ((ax{31,31}#bx{31,31}) &
      (ax{31,31}#(ax-bx){31,31}));
SF := (ax-bx) < 0;
ZF := (ax-bx) = 0;
if (~ZF ^ (OF = SF)) goto 11
X := 1
goto 12
11: X := -1
12:
    
```

The two blocks are equivalent



With IDA + BINSEC

All jump targets are found



Backward bounded SE

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

- Prove things
- Local \Rightarrow scalable

Case 2: THE XTUNNEL MALWARE

-- [BlackHat EU 2016, S&P 2017] (Robin David)



X-Agent Spyware

Now Targeting Apple's MacOS Users



Two heavily obfuscated samples

- Many opaque predicates

Goal: detect & remove protections

- Identify 40% of code as spurious
- Fully automatic, < 3h [now: 12min]

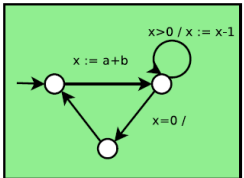
- ▶ Backward-bounded SE
- ▶ + dynamic analysis

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

- **About me, myself and this talk**
- **What every honest person should know about Symbolic Execution**
- **Challenges of binary-level security**
- **A bit about BINSEC**
- **Shades of Symbolic Execution for Security**
- **Conclusion, Take away and Disgression**

Safety is not security, fun new problems

Model



Assembly

```

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

label:
move @100 B
  
```

Source code

```

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

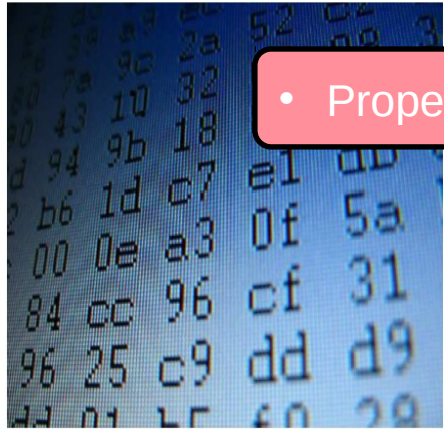
• Binary code

Executable

```

ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBD457345FD780001
FFF22546ADDAE989776600000000
  
```

• Attacker



• Properties



Under the hood: finely tune the technology



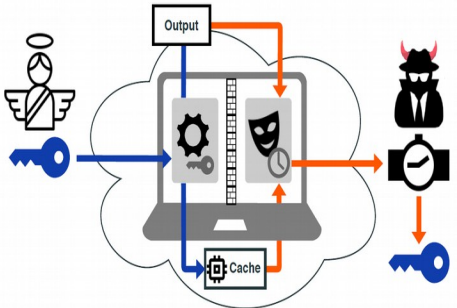
- SMT solvers are powerful weapons
- But (binary-level) security problems are terrific beasts

- Finely tuning the technology can make a huge difference

```

    ...ing sql = getStatement();
    resultSet = "select * from st...
    if (resultSet = statement.executeQu...
    resultSet.next()) {
    setStoreId = true;
    storeId(resultSet.getInt("s...
    storeDescription = resu...
    storeTypeId = resu...
    coreAdd...
  
```

- Obsidium
- JD Pack
- WinUpack
- Armadillo
- AdProtect
- EP Protector
- TELock SvK
- Yoda's Crypter
- NeoLite
- UpoHobox
- ASPack
- Petite
- Pack PE Spin
- Enigma
- RI Pack
- MySVMProtect



• Some queries: 24h => 1min

• 600x faster than prior approach

- 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