

Symbolic execution for binary-level security

~~\$0~~ ➔ A number of shades of symbolic execution

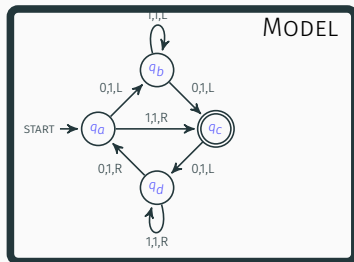
Sébastien Bardin & Richard Bonichon

20180409



CEA LIST

Map



SOURCE

```
int foo(int t) {
    int y = t * t - 4 * t;

    switch (y) {
        case 0: return 0;
        case 1: return 1;
        case 2: return 4;
        default: return 42;
    }
}
```

BINARY

```
00000000: 7f45 4c46 .ELF
00000004: 0201 0100 ....
00000008: 0000 0000 ....
0000000c: 0000 0000 ....
00000010: 0200 3e00 ...>
00000014: 0100 0000 ....
00000018: 2054 4100 TA.
0000001c: 0000 0000 ....
```

ASSEMBLY

```
addl $2, %eax
movl %eax, 12(%esp)
jmp L3
L2:
movl $5, 12(%esp)
L3:
movl 12(%esp), %eax
subl $4, %eax
```



March 2017

v 0.1

New release

Soon!

50 klocs

OCaml

LGPL

A sandbox for binary-level formal methods

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

Why is it hard ?

Code-data confusion

No specifications

Raw memory

Low-level operations

Code size

architectures

```
...  
080485ac  mov [ebp + 0xffffffff0], eax  
080485af  mov [ebp + 0xffffffff4], 0x8048708  
080485b6  cmp [ebp + 0xffffffff0], 0x9  
080485ba  ja 0x804861b  
080485bc  mov eax, [ebp + 0xffffffff0]  
080485bf  shl eax, 0x2  
080485c2  add eax, 0x8048730  
080485c7  mov eax, [eax]  
080485c9  djmp eax ; <dyn_jump>  
...
```

Automated binary-level formal methods

Abstract Interpretation

- 👍 all-paths
- 👍 scalability
- 👎 robust
- 👎 precise
- ⚙️ over-approximations

Symbolic Execution

- 👍 robust
- 👍 precise
- 👎 scalability
- 👎 single path
- ⚙️ under-approximations (DSE)

EXPLORE

PROVE

SIMPLIFY



Explore

Find bugs in your
binaries
(or play with them 😊)

What's the secret key ?

Manticore

```
int check(char *buf) {  
    check_char_0(buf[0]);  
    check_char_1(buf[1]);  
    check_char_2(buf[2]);  
    check_char_3(buf[3]);  
    check_char_4(buf[4]);  
    check_char_5(buf[5]);  
    check_char_6(buf[6]);  
    check_char_7(buf[7]);  
    check_char_8(buf[8]);  
    check_char_9(buf[9]);  
    check_char_10(buf[10]);  
    return 1;  
}
```



Bypass any kind of authentication

Impact

- Elevation of privilege
- Information disclosure
- Denial of service

Thanks to P. Biondi @



Code instrumentation

```
int main(int argc, char *argv[])
{
    struct {
        int canary;
        char buf[16];
    } state;
    my_strcpy(input, argv[1]);
    state.canary = 0;
    grub_username_get(state.buf, 16);
    if (state.canary != 0) {
        printf("This gets interesting!\n");
    }
    printf("%s", output);
    printf("canary=%08x\n", state.canary);
}
```

Can we reach "This gets interesting!" ?

Code snippet

```
static int grub_username_get (char buf[], unsigned buf_size) {
    unsigned cur_len = 0;
    int key;
    while (1) {
        key = grub_getkey ();
        if (key == '\n' || key == '\r') break;
        if (key == '\e') { cur_len = 0; break; }
        // Not checking for integer underflow
        if (key == '\b') { cur_len--; grub_printf("\b"); continue; }
        if (!grub_isprint(key)) continue;
        if (cur_len + 2 < buf_size) {
            buf[cur_len++] = key; // Off-by-two
            printf_char (key); }
    }
    // Out of bounds overwrite
    grub_memset( buf + cur_len, 0, buf_size - cur_len);
    grub_printf ("\n"); return (key != '\e');
}
```

Looking for Use-After-Free ? [SSPREW 16]



Key enabler: GUEB

```
00b8 5400 0000 5dc3 5589 e5c7 0540 bf0e
0812 0000 00b8 4800 0000 5dc3 5589 e5c7
0540 bf0e 0812 00b8 4800 0000 5dc3
5589 e5c7 0540 bf0e 0812 00b8 5800
0000 5dc3 5589 e5c7 0540 bf0e 0000
00b8 4900 0000 5dc3 5589 e583 ec10 c705
48bf 0e08 0100 0000 a148 bf0e 0083 f809
0f87 0002 0000 8b04 8548 e10b 03ff e0c6
45f7 00c6 45f8 00c6 45f9 00c6 45fa 00c7
0540 bf0e 0802 0000 00e9 d901 0000 c645
f701 c645 f800 c645 f900 c645 fa01 807d
f000 750a c705 48bf 0e08 0300 0000 807d
0500 7410 807d fc00 750a c705 48bf 0e08
0900 0000 807d fc00 7415 807d fb00 740f
c645f901 0e08 0000 0000 e988 0100 00e9
8301 0000 c645 f701 c645 f800 c645 f900
c645 fa02 807d fc00 740f c705 48bf 0e08
0400 0000 e95e 0100 00e9 5901 0000 c645
f701 c645 f800 c645 f900 c645 fa03 807d
f000 7410 807d fe00 750a c705 48bf 0e08
0500 0000 807d fc00 750a c705 48bf 0e08
0300 0000 807d fe00 740f c705 48bf 0e08
0600 0000 e90e 0100 00e9 0901 0000 c645
f701 c645 f800 c645 f901 c645fa01 807d
fd00 750f c705 48bf 0e08 0400 0000 e9e4
0000 00e9 df00 0000 c645 f701 c645 f800
c645 f900 c645 fa04 807d fc00 7410 807d
ff00 750a c705 48bf 0e08 0700 0000 807d
fc00 7415 807d ff00 740f c705 48bf 0e08
0600 0000 e99e 0000 00e9 9900 0000 c645
f701 c645 f800 c645 f900 c645 fa05 807d
fd00 7410 807d fe00 750a c705 48bf 0e08
0800 0000 807d fc00 750a c705 48bf 0e08
0900 0000 807d fe00 7506 807d ff00 740c
c705 48bf 0e08 0600 0000 eb4b eb49 c645
f701 c645 f800 c645 f901 c645 fa02 807d
```

Experimental evaluation

GUEB only

tiff2pdf

CVE-2013-4232

openjpeg

CVE-2015-8871

gifcolor

CVE-2016-3177

accel-ppp

GUEB + BINSEC/SE

libjasper

CVE-2015-5221

```
jas_tvparser_destroy(tvp);  
if (!cmpt->sampperx  !cmpt->samppery) goto error;  
if (mif_hdr_addcmpt(hdr, hdr->numcmpts, cmpt)) goto error;  
return 0;  
  
error:  
    if (cmpt) mif_cmpt_destroy(cmpt);  
    if (tvp) jas_tvparser_destroy(tvp);  
    return -1;
```

In a nutshell

GUEB + DSE is:

- ❶ better than DSE alone
- ❷ better than blackbox fuzzing
- ❸ better than greybox fuzzing without seed

Robustness

What if the instruction cannot be reasoned about ?

Program	Path predicate	Concretization	Symbolization
inputs a, b;			
x := a * b;	$x_1 = a \times b$	$a = 5$	$x_1 = \text{fresh}$
x := x + 1;	$\wedge \quad x_2 = x_1 + 1$	$\wedge \quad x_1 = 5 \times b$	$\wedge \quad x_2 = x_1 + 1$
assert(x > 10);	$\wedge \quad x_2 > 10$	$\wedge \quad x_2 = x_1 + 1$	$\wedge \quad x_2 > 10$
		$\wedge \quad x_2 > 10$	

Solutions

Concretize lose completeness

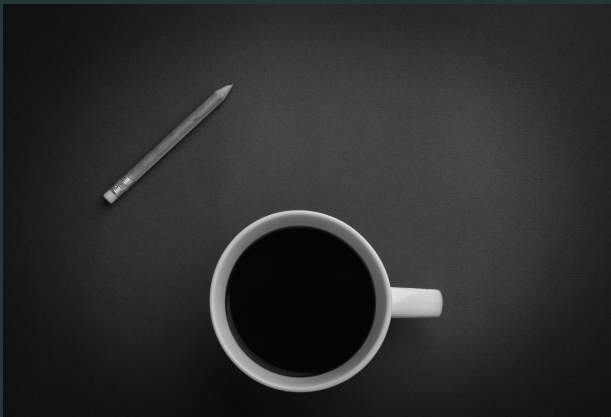
Symbolize lose correctness

A scenario

- $x := @[a * b]$
- Documentation says “Memory accesses are concretized”
- At runtime you get : $a = 7, b = 3$

What does the documentation really mean ?

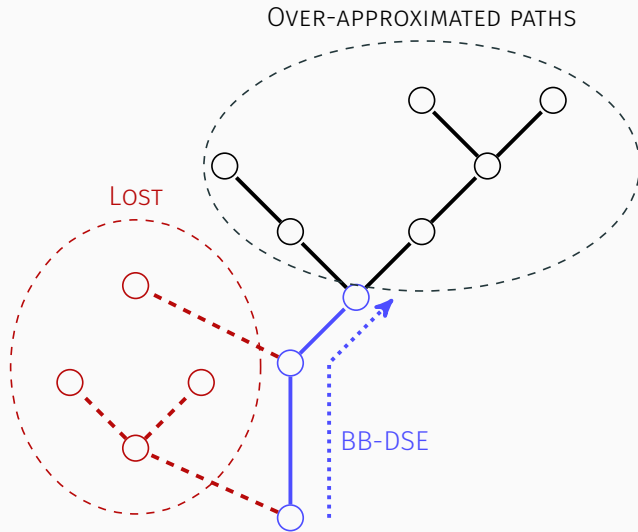
CS1	$x = \text{select}(M, 21)$	incorrect
CS2	$x = \text{select}(M, 21) \wedge a \times b = 21$	minimal
CS3	$x = \text{select}(M, 21) \wedge a = 7 \wedge b = 3$	atomic



Simplify

Remove unfeasible paths

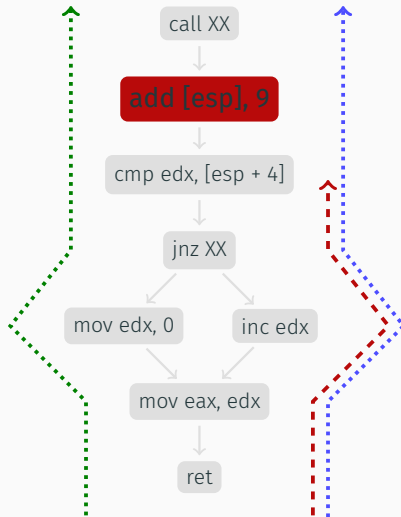
Key enabler: BB-DSE [SP 17]









BB-SE can help in reconstructing information:

- ⚙ Switch targets (indirect jumps)
- ⚙ Unfeasible branches
- ⚙ High-level predicates

Stack-tampering detection



Summarized view

	SE	BB-SE
feasibility queries		
infeasibility queries		
scaling		

Experimental evaluation

Ground truth experiments Precision

Packers Scalability, robustness

Case study Usefulness

Controlled experiments

Goal

Assess the precision

Opaque predicates — o-llvm

small k $k=16 \Rightarrow$ no false
negative, 3.5%
errors

efficient 0.02s / predicate

Stack tampering — tigress

- no false positive
genuine rets are proved
- malicious rets are single
targets

Goal

Assess the robustness and scalability

- 📖 Armadillo, ASPack, ACProtect, ...
- ⚙ Traces up several millions of instructions
- ❗ Some packers (PE Lock, ACProtect, Crypter) use these techniques a lot
- ❗ Others (Upack, Mew, ...) use a single stack tampering to the entrypoint

X-Tunnel analysis

	Sample 1	Sample 2
# instructions	$\approx 500\text{k}$	$\approx 434\text{k}$
# alive	$\approx 280\text{k}$	$\approx 230\text{k}$

> 40% of code is **spurious**

Protection relies only on opaque predicates

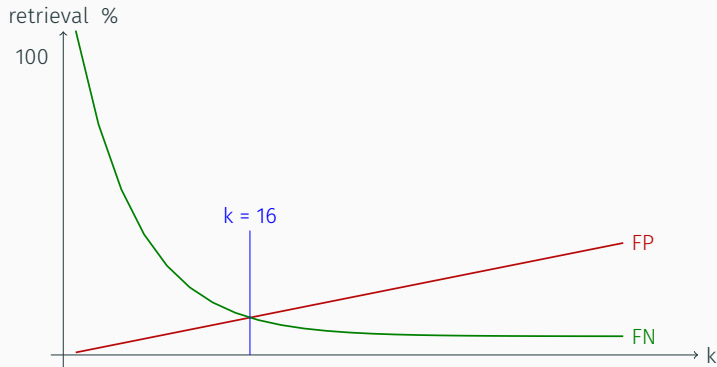
i Only 2 equations

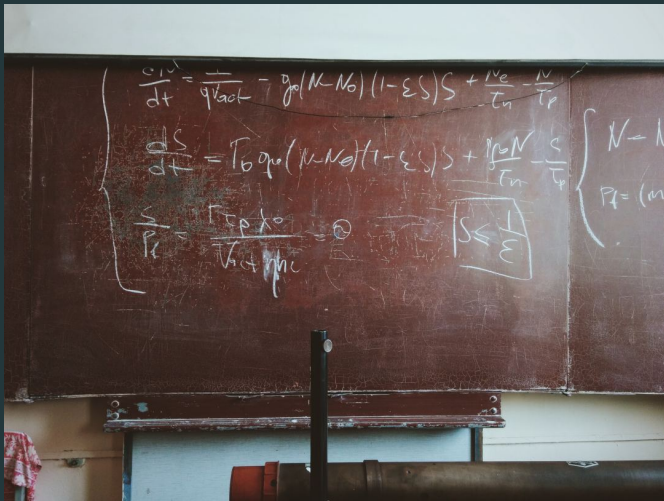
- $7y^2 - 1 \neq x^2$
- $\frac{2}{x^2+1} \neq y^2 + 3$

i Sophisticated

- original OPs
- interleaves payload and OP computations
- computation is shared
- some long dependency chains, up to 230 instructions

Experimental behavior









Prove

Low-level comparisons are not always what they seem to be ...

Some low-level conditions

Mnemonic	Flag	cmp x y	sub x y	test x y
ja	$\neg CF \wedge \neg ZF$	$x >_u y$	$x' \neq 0$	$x \& y \neq 0$
jnae	CF	$x <_u y$	$x' \neq 0$	\perp
je	ZF	$x = y$	$x' = 0$	$x \& y = 0$
jge	OF = SF	$x \geq y$	T	$x \geq 0 \vee y \geq 0$
jle	$ZF \vee OF \neq SF$	$x \leq y$	T	$x \& y = 0 \vee$ $(x < 0 \wedge y < 0)$
...				

code	high-level condition	patterns
or eax, 0 je ...	if $eax = 0$ then goto ...	
cmp eax, 0 jns ...	if $eax \geq 0$ then goto ...	
sar ebp, 1 je ...	if $ebp \leq 1$ then goto ...	
dec ecx jg ...	if $ecx > 1$ then goto ...	

Sometimes it gets even more interesting

```
cmp eax, ebx  
cmc  
jae ...
```



BINSEC

SE helps to

- 👍 Explore
- 👍 Prove
- 👍 Simplify

Semantics & SE
to the
Rescue



<https://rbonichon.github.io/posts/use-18>