Symbolic execution for binary-level security

∄∅ ∃ A number of shades of symbolic execution

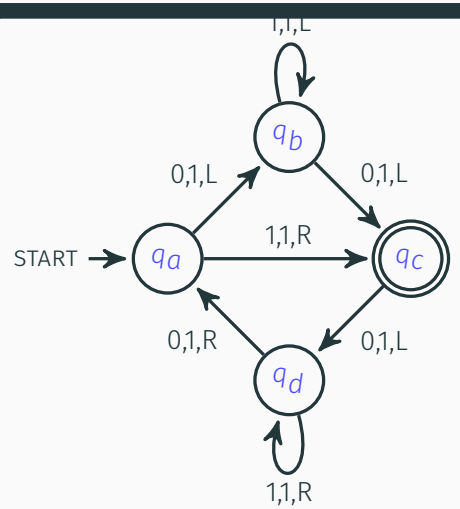
Sébastien Bardin & Richard Bonichon

20180409

CEA LIST

Map

Model



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;
  }
}
```

Assembly

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

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  ....
```

BINSEC : a sandbox for binary-level formal methods

March 2017

v 0.1

New release

Soon!

50 klocs

`OCaml`

LGPL

`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 + 0xfffffff0], eax
080485af   mov [ebp + 0xfffffff4], 0x8048708
080485b6   cmp [ebp + 0xfffffff0], 0x9
080485ba   ja 0x804861b
080485bc   mov eax, [ebp + 0xfffffff0]
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;
}
```

Bug finding : Grub2 CVE 2015-8370

Bypass any kind of authentication

Impact

- Elevation of privilege
- Information disclosure
- Denial of service

Thanks to P. Biondi @ AIRBUS GROUP

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);
}
```

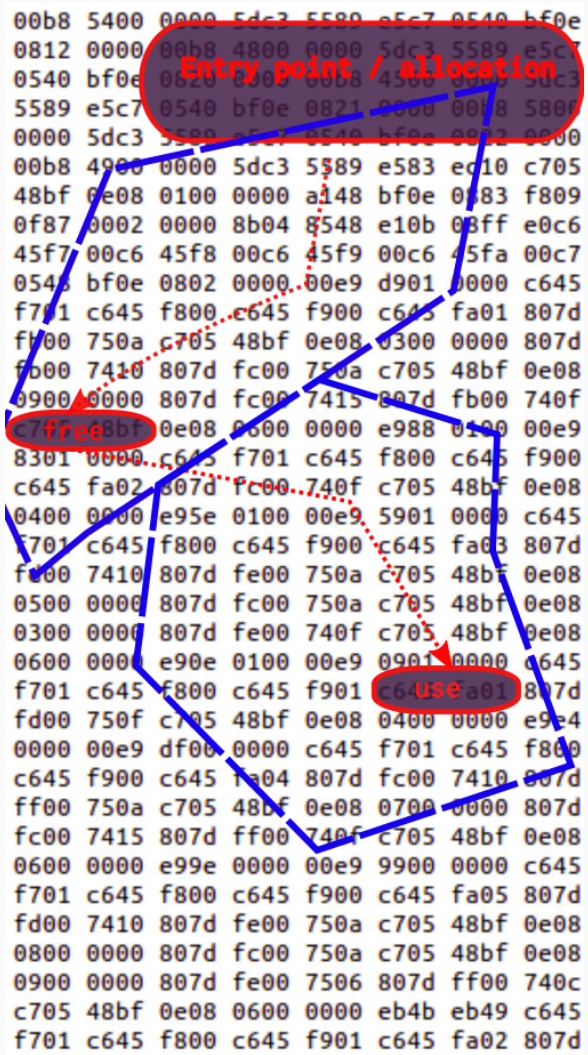
Can we reach `"This gets interesting!"` ?

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; }
    if (key == '\b') { cur_len--; grub_printf("\b"); continue; }
    if (!grub_isprint(key)) continu;e
    if (cur_len + 2 < buf_size) { buf[cur_len++] = key;
                                  printf_char (key); }
  }
  //  snip: Out of bounds overwrite
  grub_printf ("\n");  return (key != '\e');
}
```

Looking for Use-After-Free ?

Key enabler: GUEB



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

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;
```

Lessons learned

In a nutshell
GUEB + DSE is:

- better than DSE alone
- better than blackbox fuzzing
- better than greybox fuzzing without seed

C/S : robustness & tradeoffs

Robustness
What if the instruction cannot be reasoned about ?

Program

```
inputs a, b;
x := a * b;
x := x + 1;
assert(x > 10);
```

Path predicate

$$\begin{aligned} & x_1 = a \times b \\ \wedge\ & x_2 = x_1 + 1 \\ \wedge\ & x_2 > 10 \end{aligned}$$

Concretization

$$\begin{aligned} & a = 5 \\ \wedge\ & x_1 = 5 \times b \\ \wedge\ & x_2 = x_1 + 1 \\ \wedge\ & x_2 > 10 \end{aligned}$$

Symbolization

$$\begin{aligned} & x_1 = \text{fresh} \\ \wedge\ & x_2 = x_1 + 1 \\ \wedge\ & x_2 > 10 \end{aligned}$$

Solutions

Concretize lose completeness

Symbolize lose correctness

C/S Policies interpretation

A scenario

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

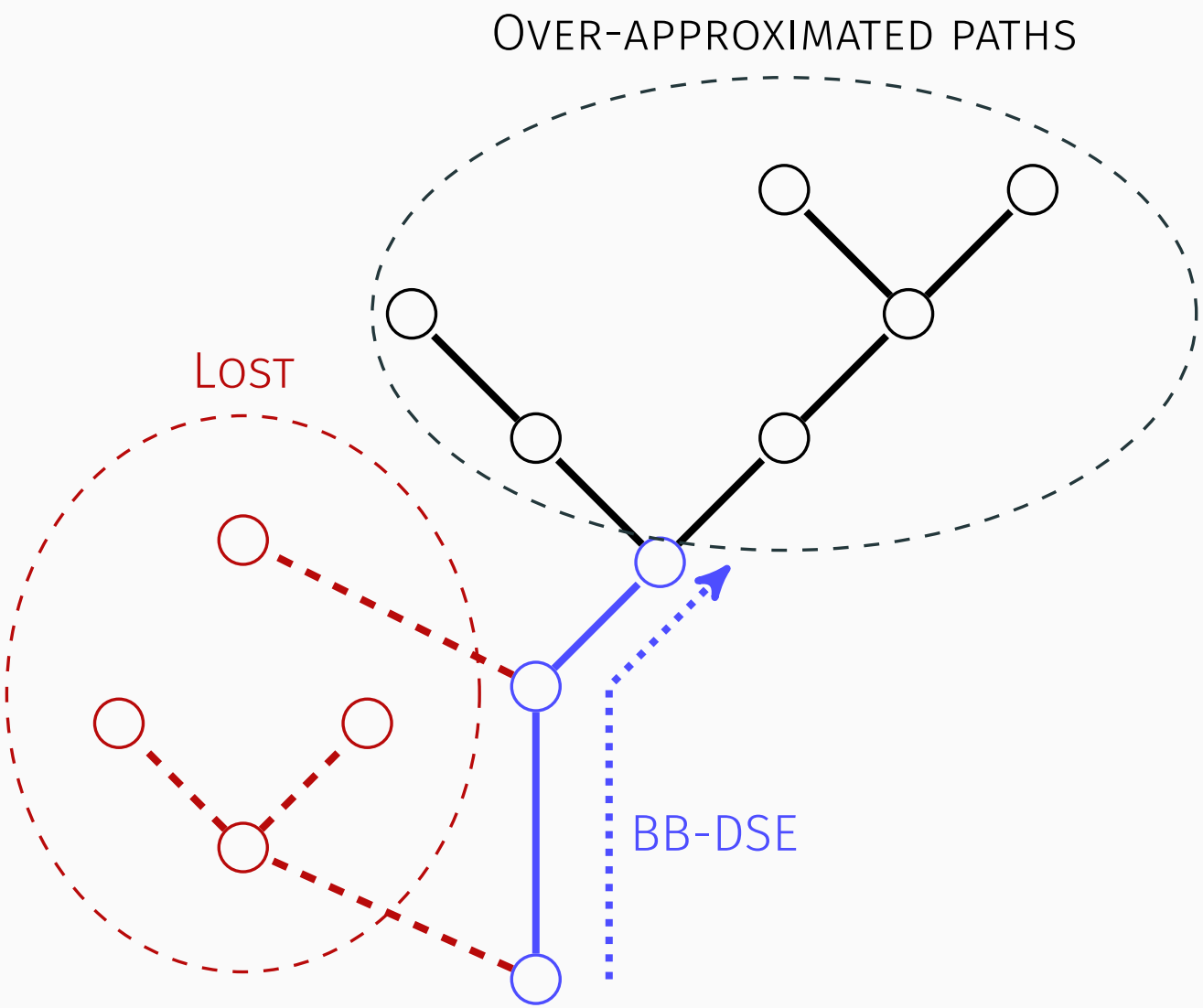
What does the documentation really mean ?

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

Simplify

Remove unfeasible paths

Key enabler: BB-DSE

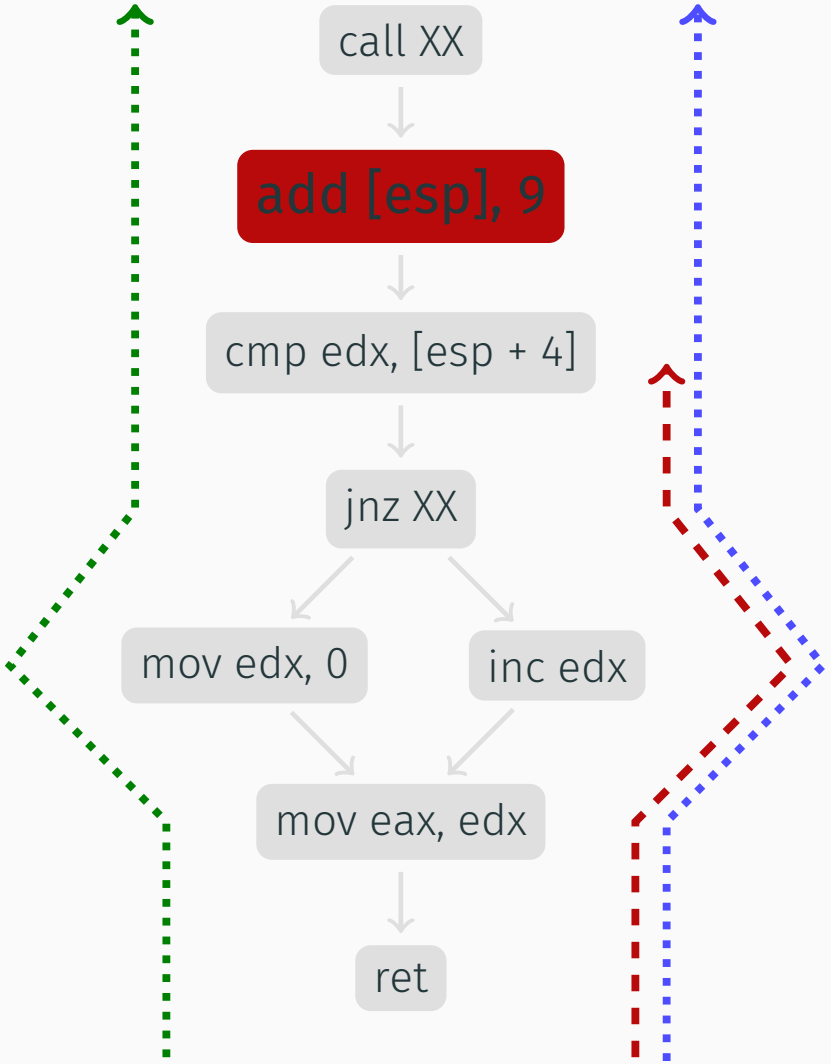


Playing with BB-SE

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

Packers

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	≈ 500k	≈ 434k
# alive	≈ 280k	≈ 230k

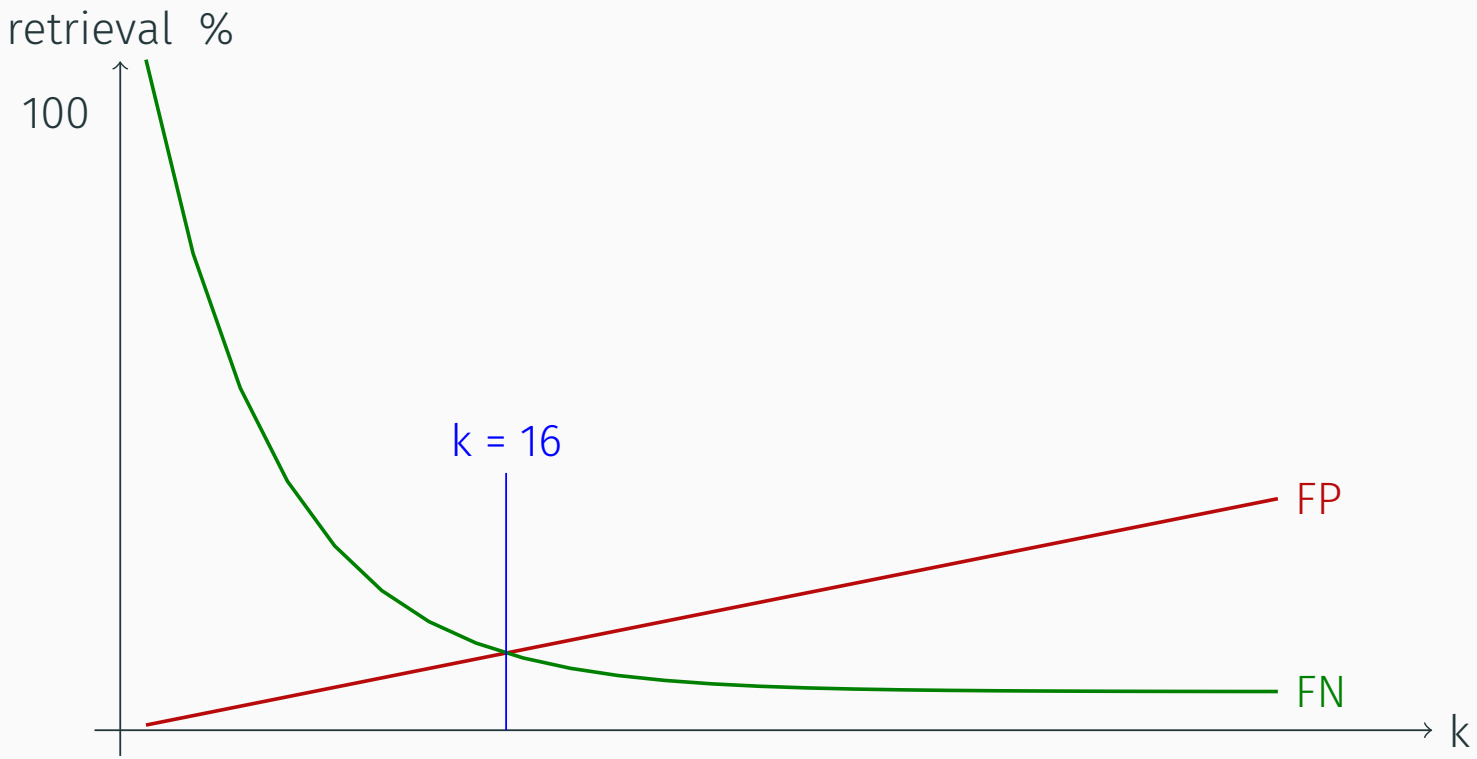
$> 40\%$ of code is spurious

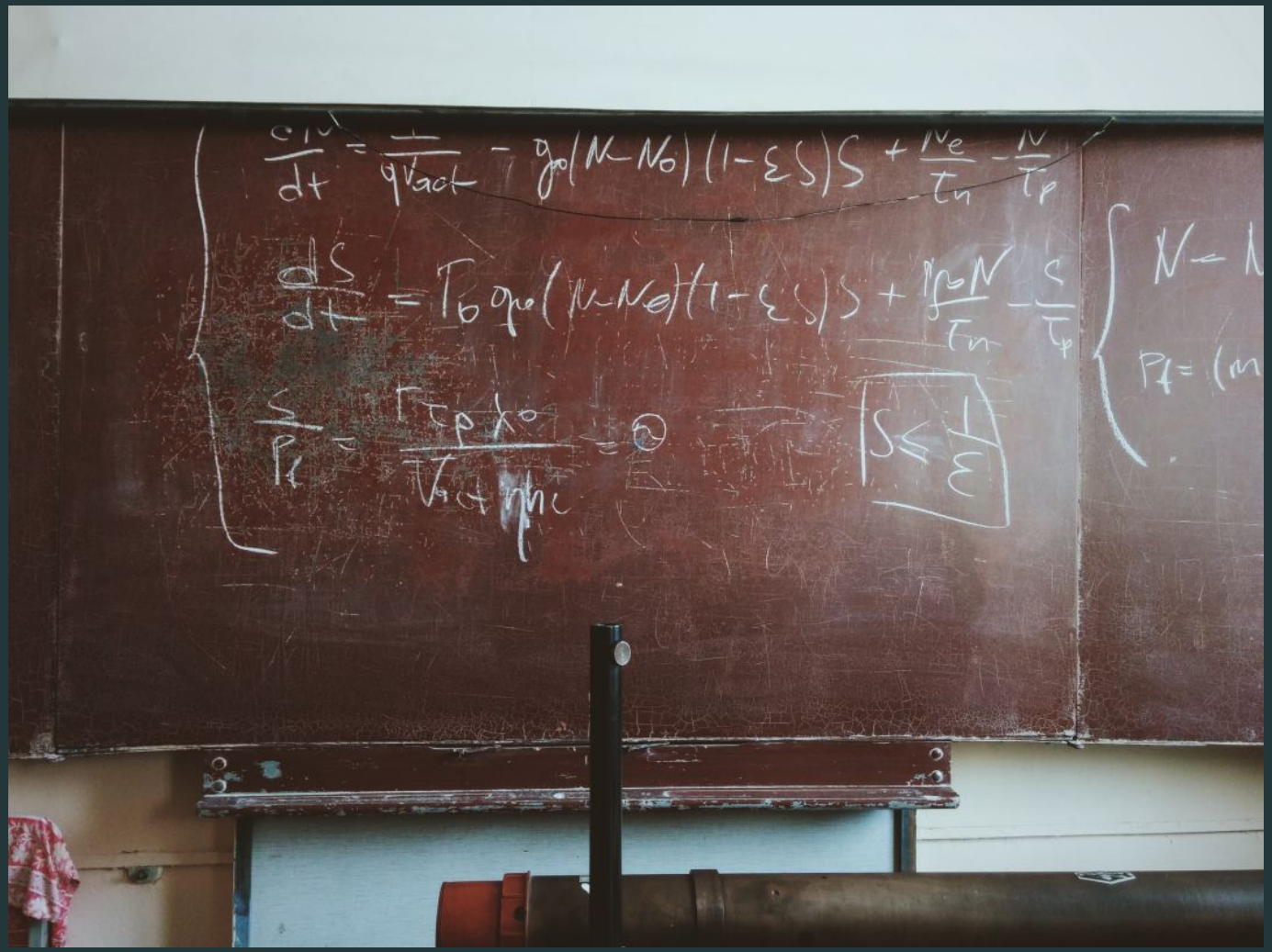
X-Tunnel: facts

Protection relies only on opaque predicates

- Only 2 equations
  - $7y^2 - 1 \neq x^2$
  - $\frac{2}{x^2+1} \neq y^2 + 3$
- Sophisticated
  - original OPs
  - interleaves payload and OP computations
  - compution 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	$x <_u y$	$x' \neq 0$	$\bot$
`je`	ZF	$x = y$	$x' = 0$	$x \& y = 0$
`jge`	OF = SF	$x \geq y$	$\top$	$x \geq 0 \vee y \geq 0$
`jle`	ZF $\vee$ OF $\neq$ SF	$x \leq y$	$\top$	$x \& y = 0 \vee$ $(x < 0 \wedge y < 0)$

...

Example zoo

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
❗