Get the VM

Local network

ssid	binsec@ssprew
password	binsec@ssprew

Access

ip	10.10.10.254
user	guest
password	☺ (leave the field empty)

Or through one of the USB flash drives or from

`https://rbonichon.github.io/posts/ssprew-17/`

This URL also includes details about use

BINSEC: a Tutorial

Sébastien Bardin & Richard Bonichon

with the help of

R. David, A. Djoudi, B. Farinier, J. Feist, G. Girol, M. Lemerre, Y. Lhuillier, F. Recoules & Y. Vinçont

20171204

CEA LIST

Outline

Context

Dynamic Bitvector Automata

Basic static disassembly

Symbolic execution

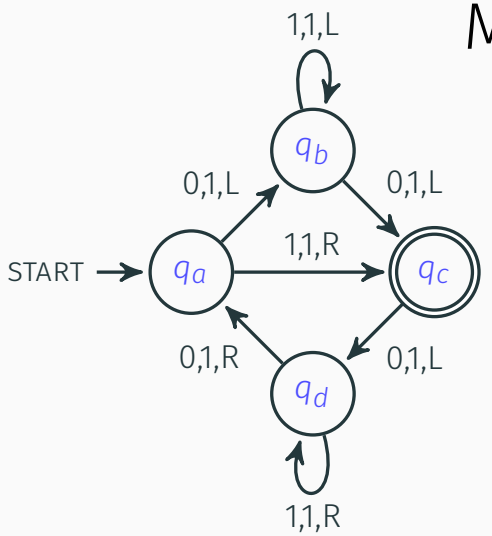
Backward Symbolic Execution

Advanced case studies

Conclusion

Context

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  ....
```


WHY
?


01001001
00100000
01001100
01001111
01010110
01000101
00100000
01011001
01001111
01010101
code

Should you blindly trust ...

Visual Studio

FFmpeg

OpenSSL
Cryptography and SSL/TLS Toolkit

SQLite

What is printed here ?

```c
#include "stdio.h"

long foo(int *x, long *y) {
  *x = 0;
  *y = 1;
  return *x;
}

int main(void) {
  long l;
  printf("%ld\n", foo((int *) &l, &l));
  return 0;
}
```

Binary code doesn't lie

	gcc 7.2.0	clang 5.0
-O0	1	1
-O1	1	0
-O2	0	0
-O3	0	0

COTS / Legacy

Why is it hard ?

Code-data confusion

No specifications

Raw memory, low-level operations

Code size

architectures

Binary-level security analysis is

necessary

very challenging

Standard (syntactic) tools are

not enough

Semantic Program Analysis

Used succesfully in safety-critical systems

Semantics

is preserved by compilation

is preserved by obfuscation

thus cannot be hidden

can reason about sets of execution

find rare events

prove and simplify

About

BINSEC is an analysis platform for binary code

Main goal

Adapt formal methods used for safety on source code to advance security on binary executable

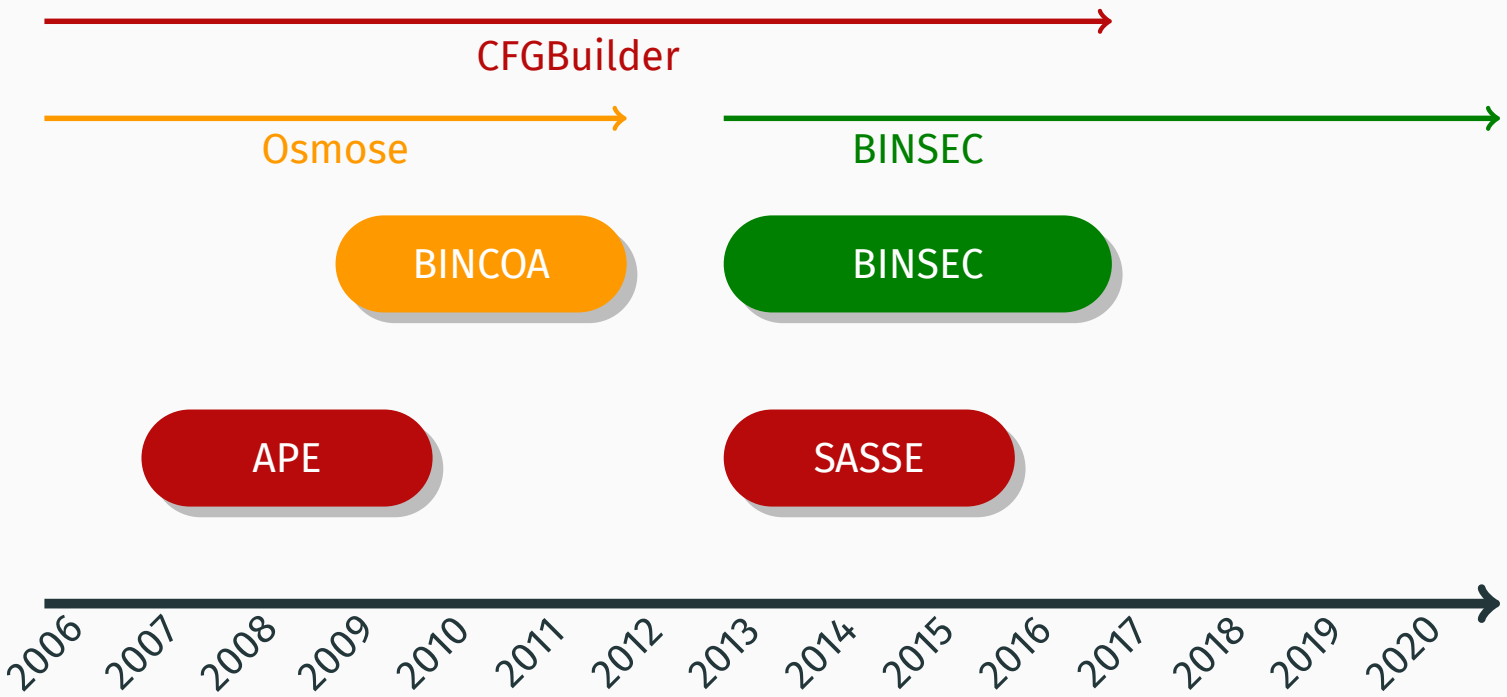
Application fields

VULNERABILITY DETECTION

MALWARE ANALYSIS

CODE VERIFICATION

Timeline


CFGBuilder
Osmose
BINSEC
BINCOA
BINSEC
APE
SASSE
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020

Software

March 2017

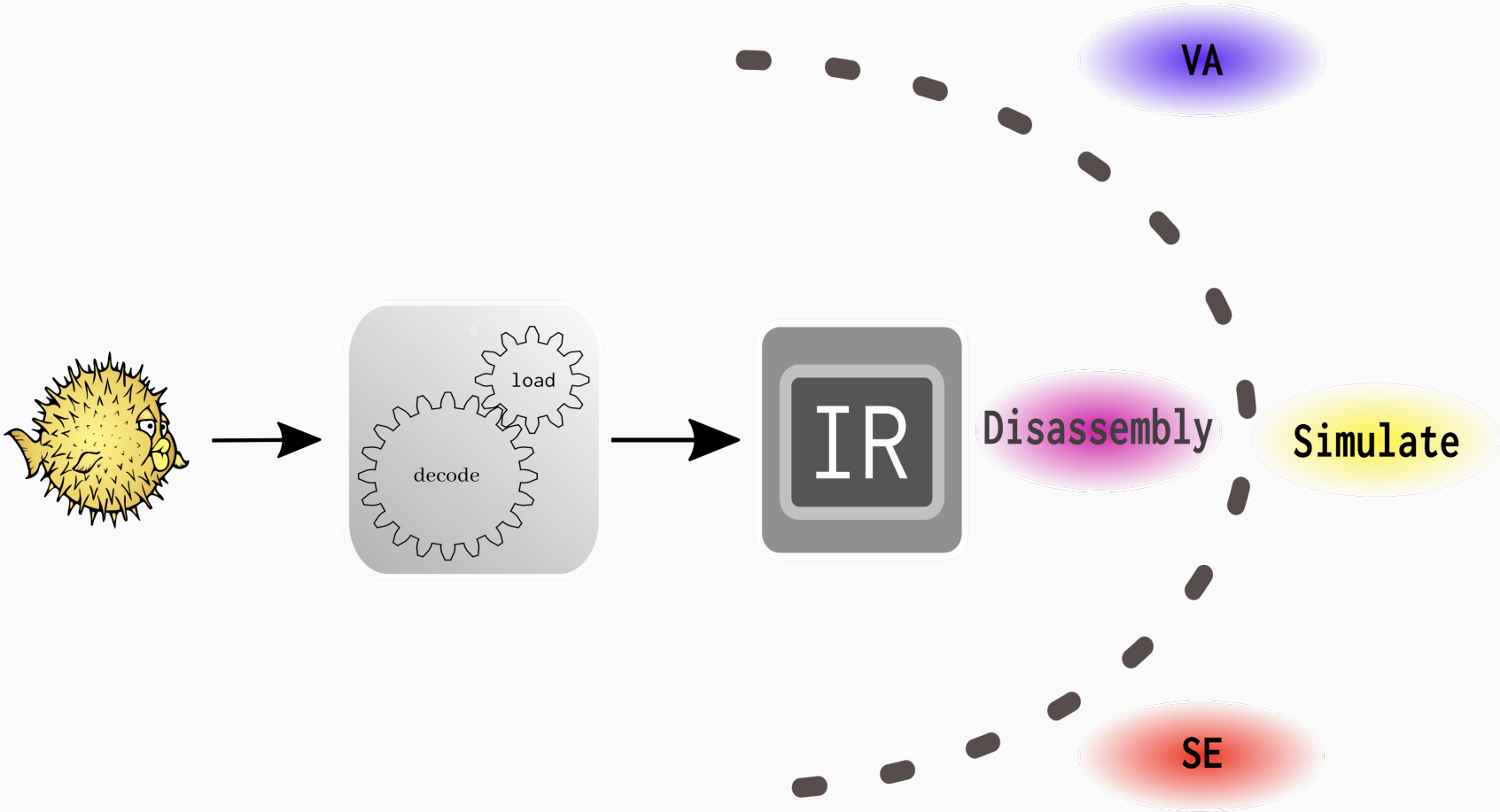
v 0.1

50 klocs

`OCaml`

LGPL

Overview of architecture



BINSEC

Explore

Prove

Simplify

Dynamic Bitvector Automata

Qualities of a good IR

Small

Well-behaved

(semantics, typing)

Extensible

Flexible

Syntax

Instructions

```
<i> :=
  | <lv> := <e>
  | goto <e>
  | if <e> then goto <addr>
           else goto <addr>
  | nondet <lv>
  | undefined <lv>
  | <logical>


<logical> :=
  | assert <e>
  | assume <e>
```

Expressions

```
<lv> :=
 | <var>
 | <var>{lo, hi}
 | @[<e>]


<e> :=
  | <e> <bop> <e>
  | <uop> <e>
  | @[<e>]
  | <var>
  | <cst>
```

Example

```
binsec disasm -machdep x86 -decode 0416
```

```
[result] 04 16 / add al, 0x16
          0: res8 := (eax(32){0,7} + 22(8))
          1: OF := ((eax(32){7} = 0(1)) & (eax(32){7} != res8(8){7}))
          2: SF := (res8(8) <s 0(8))
          3: ZF := (res8(8) = 0(8))
          4: AF := ((extu eax(32){0,7} 9) + 22(9)){8}
          5: PF := !
                   (((((((res8(8){0} ^ res8(8){1}) ^ res8(8){2}) ^
                         res8(8){3}) ^ res8(8){4}) ^ res8(8){5}) ^
                     res8(8){6}) ^ res8(8){7}))
          6: CF := ((extu eax(32){0,7} 9) + 22(9)){8}
          7: eax{0, 7} := res8(8)
          8: goto ({0x00000002; 32}, 0)
```

Semantics is not always easy

> *After executing* `shl ecx, 32`*, is the overflow flag* `OF` *defined? If so, what is its value? After executing* `shl cl, 1`*, is the overflow flag* `OF` *defined? If so, what is its value?*

– *R.Rolles, The Case for Semantics-Base Methods in Reverse Engineering*

What the doc says

The OF flag is affected **only on 1-bit shifts**. *For* **left shifts**, *the OF flag is set to 0 if the most- significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.*

– *IA-32 Intel Architecture Software Developer's Manual*, 3-703

Solution `shl ecx, 0x20`

```
binsec disasm -machdep x86 -decode c1e120
```

```
[result] c1 e1 20 / shl ecx, 0x20
          0: res32 := ecx(32)
          1: SF := (res32(32) <s 0(32))
          2: ZF := (res32(32) = 0(32))
          3: CF := (ecx(32) << - (1(32))){31}
          4: OF := \undef
          5: ecx := res32(32)
          6: goto ({0x00000003; 32}, 0)
```

Solution `shl cl, 0x01`

```
binsec disasm -machdep x86 -decode c0e101
```

```
[result] c0 e1 01 / shl cl, 0x1
          0: res8 := (ecx(32){0,7} << 1(8))
          1: SF := (res8(8) <s 0(8))
          2: ZF := (res8(8) = 0(8))
          3: CF := ecx(32){7}
          4: OF := (res8(8){7} ^ CF(1))
          5: ecx{0, 7} := res8(8)
          6: goto ({0x00000003; 32}, 0)
```

Arm example

```
binsec disasm -machdep arm -decode 060050e1
```

```
[result] e1 50 00 06 / cmp      r0, r6
          0: tmp32_0 := (r0(32) - r6(32))
          1: nxt_n := (tmp32_0(32) <s 0(32))
          2: nxt_z := (tmp32_0(32) = 0(32))
          3: nxt_c := (r0(32) >=u r6(32))
          4: nxt_v := (nxt_n(1) ^ (r0(32) <s r6(32)))
          5: n := nxt_n(1)
          6: z := nxt_z(1)
          7: c := nxt_c(1)
          8: v := nxt_v(1)
          9: goto ({0x00000004; 32}, 0)
```

Semantics is not always easy (part II)

Thanks to

for uncovering a number of bugs in BINSEC with MeanDiff

Your mission

Use only BINSEC to explore your binaries ....

Basic static disassembly

Goals

Uncover program instructions

Get a static control-flow graph (CFG)

Focus: linear disassembly

At address `a`

Read instruction `i`
Compute its DBA encoding
Add it to the CFG

Uncover instruction

Add address `a + sizeof(a)` to the worklist

Add flow information

if `i` is

a jump add an edge between `i` and its jump target(s)

a call add edges to its callee and its linear successor (call returns)

otherwise add an edge to its linear successor

Disassembly modes

- linear
- recursive
- extended linear (aka linear + recursive)
- bytewise

Limitations

Does it work ?

Is my CFG : correct ? complete ?

Unprotected (compiled) code

Yes (mostly)

Protected code

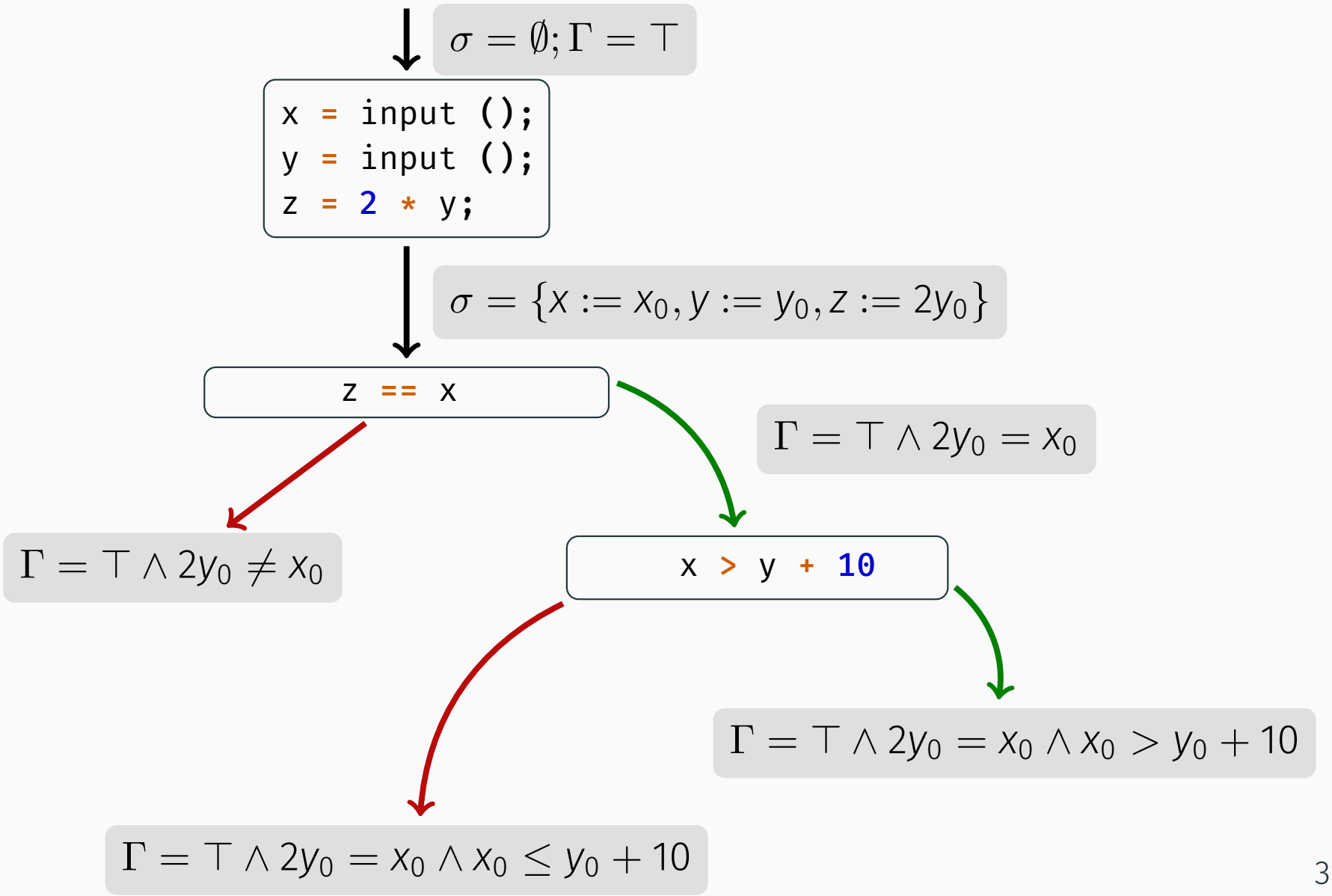
It depends ...

Symbolic execution

What is symbolic execution ?

Interpret paths of the program as logical formulas

Running SE


$\sigma = \emptyset; \Gamma = \top$
x = input ();
y = input ();
z = 2 * y;
$\sigma = \{x := x_0, y := y_0, z := 2y_0\}$
z == x
$\Gamma = \top \wedge 2y_0 = x_0$
$\Gamma = \top \wedge 2y_0 \neq x_0$
x > y + 10
$\Gamma = \top \wedge 2y_0 = x_0 \wedge x_0 > y_0 + 10$
$\Gamma = \top \wedge 2y_0 = x_0 \wedge x_0 \leq y_0 + 10$

Now what ?

Feasability of path

$\longleftrightarrow$ / $\longrightarrow$

Constraint solving

$\longrightarrow^{?}$

Test input

Checking feasability

```
x = input ();
y = input ();
z = 2 * y;
z == x;
! (x  > y + 10);
```

```
(declare-fun x0 () Int)
(declare-fun y0 () Int)

(define-fun z0 () Int (* 2 y0))

(assert (= z0 x0))
(assert (not (> x0 (+ y0 10))))
(check-sat) ;; sat

(get-model) ;; x0 := 0, y0 := 0
(get-value (z0)) ;; z0 := 0
```

DSE

DSE

Symbolic execution using a concrete (dynamic) execution trace

```
x = g (); // g() > 10
y = f (); // f() == 0
z = 2 * y;
z == x;
! (x  > y + 10);
```

```
(define-fun x0 () Int 11) ;; concrete
(define-fun y0 () Int 0)  ;; concrete

(define-fun z0 () Int (* 2 y0))

(assert (= z0 x0))
(assert (not (> x0 (+ y0 10))))
(check-sat) ;; unsat
```

Goal oriented vs exploration

SE in effect assesses the reachability of a path.

Repeated applications can be used if you need branch coverage (test, verification).

Let's play!

Manticore Challenge

A classic crackme example from
`https://blog.trailofbits.com/2017/05/15/magic-with-manticore/`

> *The first solution to the challenge that executes in under 5 minutes will receive a bounty from the Manticore team.*

Manticore check functions

Char 0

```
int check_char_0(char chr) {
  register uint8_t ch =
    (uint8_t) chr;
  ch ^= 97;

  if(ch != 92) exit(1);

  return 1;
}
```

Char 9

```
int check_char_9(char chr) {
  register uint8_t ch =
    (uint8_t) chr;
  ch ^= 61;
  ch += 41;
  ch += 11;
  if(ch != 172) exit(1);
  return 1;
}
```

Solution

```
# Char.chr (92 lxor 97);;
- : char = '='
```

Solution

```
# let v = 172 - 11 - 41 in
  Char.chr (v lxor 61);;
- : char = 'E'
```

Binary Manticore

Let's check it out with BINSEC!

Manticore

What about other architectures ?

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!"` ?

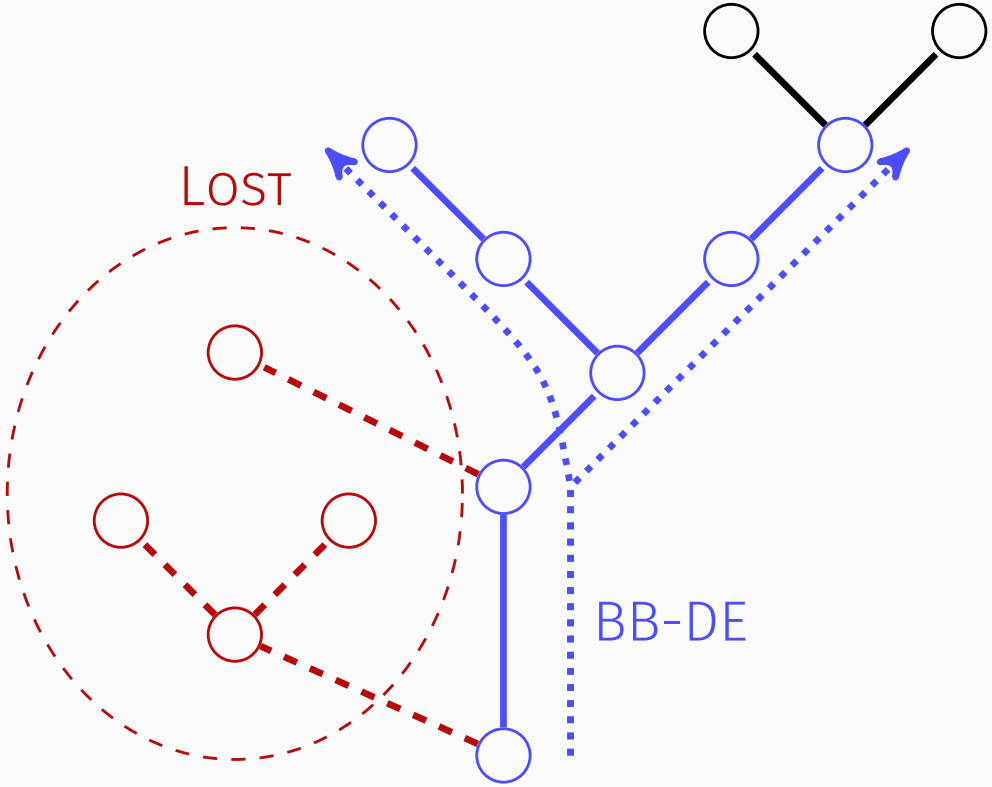
Code snippet

```
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');
}
```

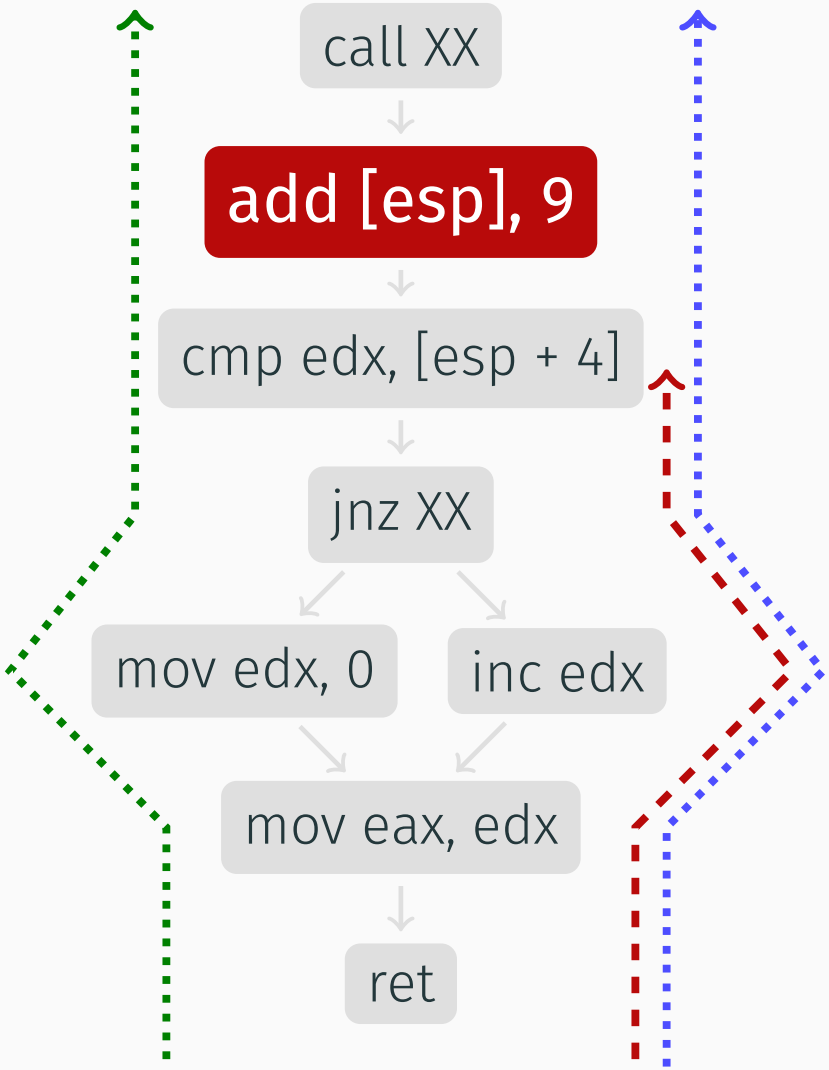

OKAY
LET'S
DO
THIS

Backward Symbolic Execution

Backward-bounded symbolic analysis



Illustration



Summarized view

	SE	BB-SE
feasibility queries	👍	👎
infeasibility queries	👎	👍
scaling	👍	👍

Playing with BB-SE

BB-SE can help in reconstructing information:

- Switch targets
- High-level predicates
- Unfeasible branches

The switch

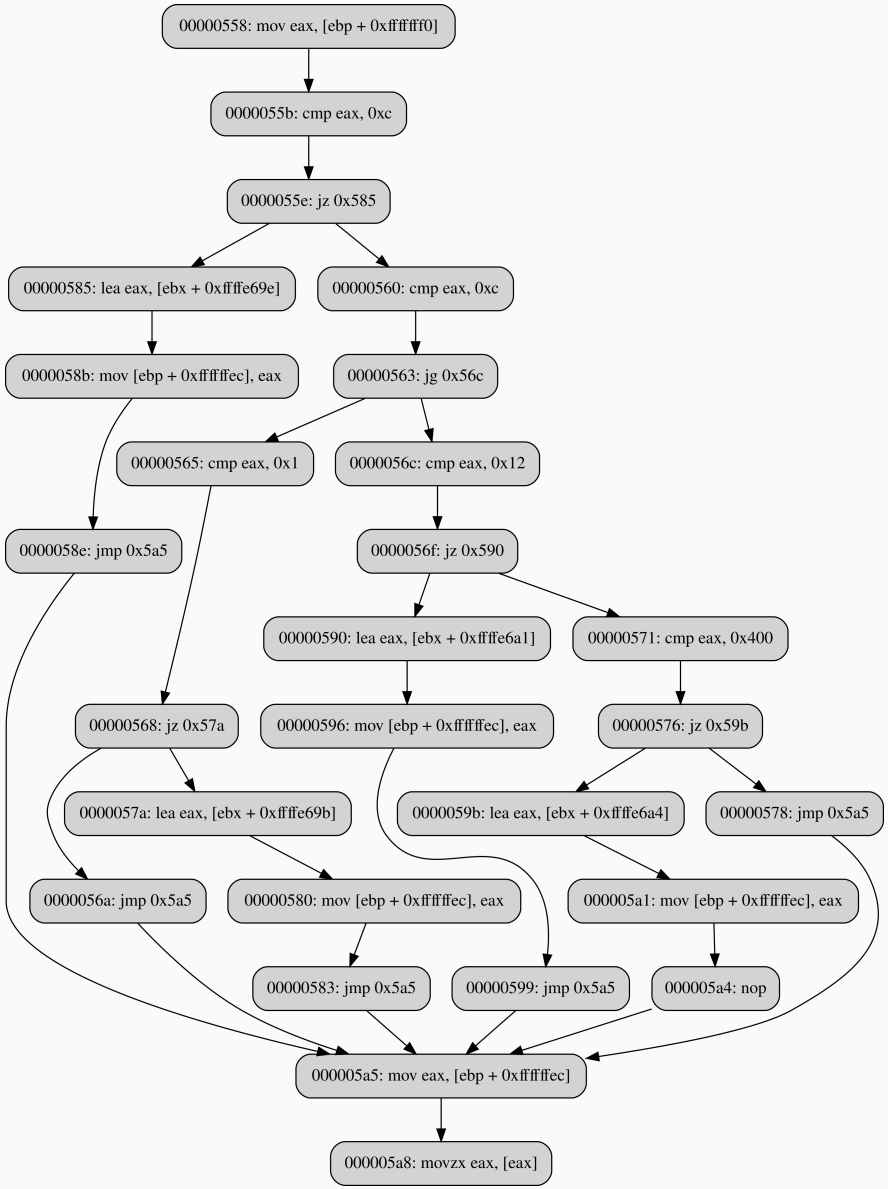
The tree switch

```
char *b = "01"
switch (a) {
  case 1: b = "10";
          break;
  case 12: b = "42";
          break;
  case 18: b = "93";
          break;
  case 1024: b = "16";
          break;
}
```

Compiled version

```
...
00000555 89 45 ec                   mov [ebp + 0xffffffec], eax
00000558 8b 45 f0                   mov eax, [ebp + 0xfffffff0]
0000055b 83 f8 0c                   cmp eax, 0xc
0000055e 74 25                      jz 0x585
00000560 83 f8 0c                   cmp eax, 0xc
00000563 7f 07                      jg 0x56c
00000565 83 f8 01                   cmp eax, 0x1
00000568 74 10                      jz 0x57a
0000056a eb 39                      jmp 0x5a5
0000056c 83 f8 12                   cmp eax, 0x12
0000056f 74 1f                      jz 0x590
00000571 3d 00 04 00 00             cmp eax, 0x400
00000576 74 23                      jz 0x59b
00000578 eb 2b                      jmp 0x5a5
...
```

Tree view


00000558: mov eax, [ebp + 0xfffffff0]
0000055b: cmp eax, 0xc
0000055e: jz 0x585
00000585: lea eax, [ebx + 0xffffe69e]
00000560: cmp eax, 0xc
0000058b: mov [ebp + 0xffffffec], eax
00000563: jg 0x56c
00000565: cmp eax, 0x1
0000056c: cmp eax, 0x12
0000058e: jmp 0x5a5
0000056f: jz 0x590
00000590: lea eax, [ebx + 0xffffe6a1]
00000571: cmp eax, 0x400
00000568: jz 0x57a
00000596: mov [ebp + 0xffffffec], eax
00000576: jz 0x59b
0000057a: lea eax, [ebx + 0xffffe69b]
0000059b: lea eax, [ebx + 0xffffe6a4]
00000578: jmp 0x5a5
0000056a: jmp 0x5a5
00000580: mov [ebp + 0xffffffec], eax
000005a1: mov [ebp + 0xffffffec], eax
00000583: jmp 0x5a5
00000599: jmp 0x5a5
000005a4: nop
000005a5: mov eax, [ebp + 0xffffffec]
000005a8: movzx eax, [eax]

The jumpy switch

```
switch (a) {
    case 1: b = "10";
            break;
    case 2: b = "42";
            break;
    case 3: b = "93";
            break;
    case 4: b = "16";
            break;
    case 5: b = "25";
            break;
}
```

Compiled version

```
binsec disasm ~/examples/switch/a.out
```

```
...
080485ac 89 45 f0                   mov [ebp + 0xfffffff0], eax
080485af c7 45 f4 08 87 04 08       mov [ebp + 0xfffffff4], 0x8048708
080485b6 83 7d f0 09                cmp [ebp + 0xfffffff0], 0x9
080485ba 77 5f                      ja 0x804861b
080485bc 8b 45 f0                   mov eax, [ebp + 0xfffffff0]
080485bf c1 e0 02                   shl eax, 0x2
080485c2 05 30 87 04 08             add eax, 0x8048730
080485c7 8b 00                      mov eax, [eax]
080485c9 ff e0                      djmp eax ; <dyn_jump>
...
```

Let's start

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

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 ...	👎

This can get even more interesting

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

Opaque predicates

Definition

A predicate whose branches cannot be both taken by any execution.

It is a predicate that is either always $\top$ or always $\bot$.

The simplest

```
if (x == x + 1) printf ("true\n");
else printf("false\n");
```


ENJOY

Advanced case studies

Combinations

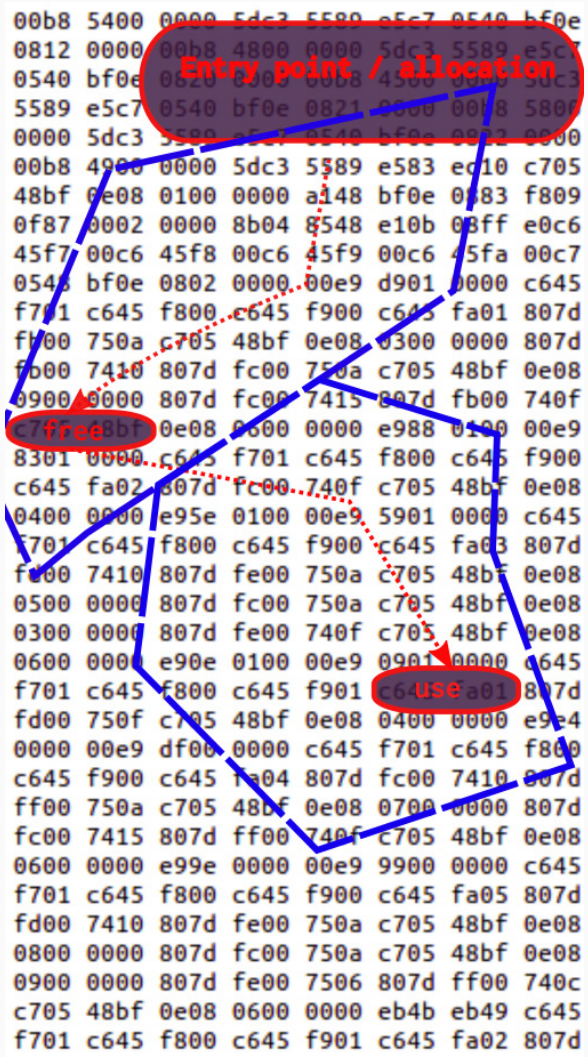
Static analysis + DSE for bug-finding

DSE for malware deobfuscation

Code verification with SE + VA

Looking for UAF ?

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:

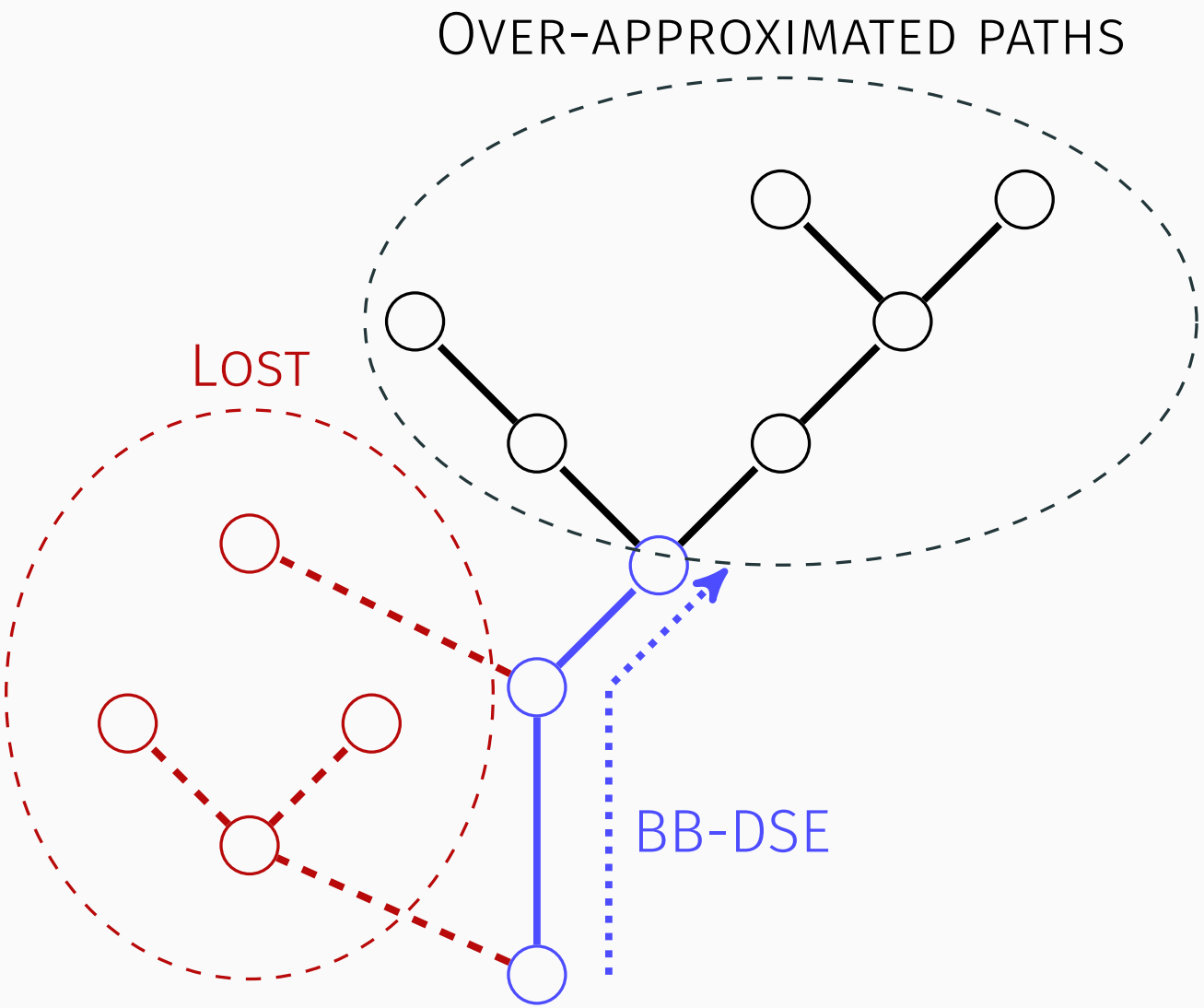
- ℹ better than DSE alone
- ℹ better than blackbox fuzzing
- ℹ better than greybox fuzzing without seed

Malware comprehension

I have multiple samples of the same malware, some obfuscated

Are there evolutions in functionalities between those samples ?

Key enabler: BB-DSE



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	$\approx$ 500k	$\approx$ 434k
# alive	$\approx$ 280k	$\approx$ 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

Verification of a RTOS (WIP)

```
while(1) {
  kernelcode();
  usercode();
}
```

4000 traces of $\approx$ 7k instructions

Results

A sound CFG provided memory safety is proven.

1 potential bug found

a set of invariants about values contained in registers and memory

Conclusion

What I did not talk about

PINSEC

Abstract interpretation

DBA-to-LLVM

C/S Policies

...

BINSEC

See you soon on for 0.2

- Enhanced CFG construction
- Better disassembly
- ARM v7 (with `http://unisim-vp.org/`)
- SSE
- Enhanced SMT support

Wait for 0.3 for 64 bits architectures

Challenges ahead

User-friendliness

Enhanced combinations of analyses

Address robustness, precision & scalability

Become 

for binary code ?

BINSEC

Semantics to the Rescue