Five Shades of Symbolic Execution for Vulnerability Hunting

FROM RESEARCH TO INDUSTRY

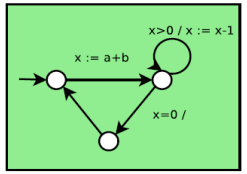
Sébastien Bardin

Senior Researcher,

Head of the BINary-level SECurity group

Me, Myself and I:
ADAPT FORMAL METHODS TO BINARY-LEVEL SECURITY ANALYSIS

Model



Assembly

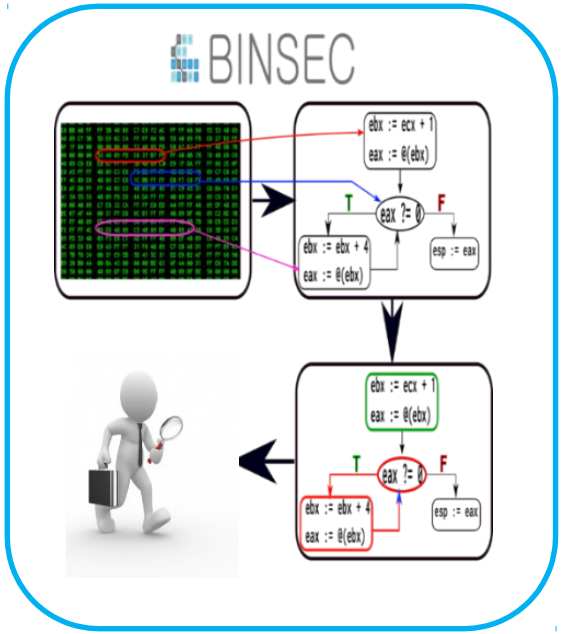
```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Source code

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

Executable

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



https://binsec.github.io/

IN A NUTSHELL

- 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

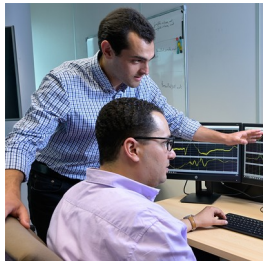
- *Questions:*
  - *how can you use Symbolic Execution into a security context ?*
  - *How does code-level security differ from code-level safety?*

- *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



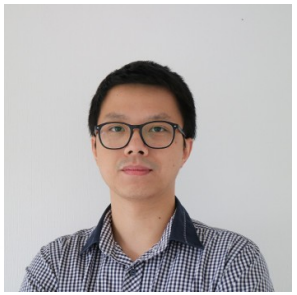










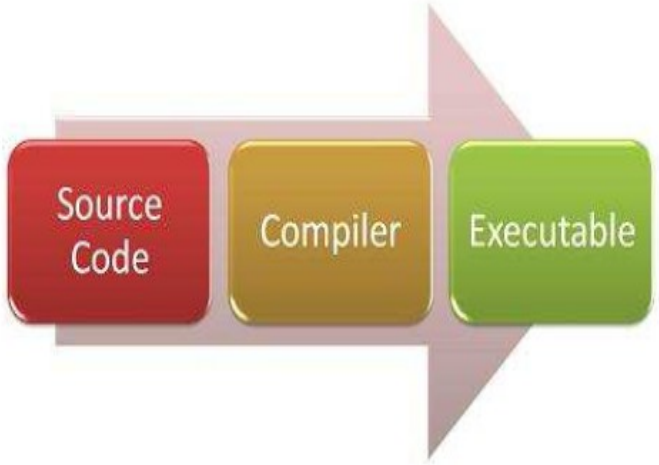





But … WHY ON BINARY CODE?

No source code

Post-compilation

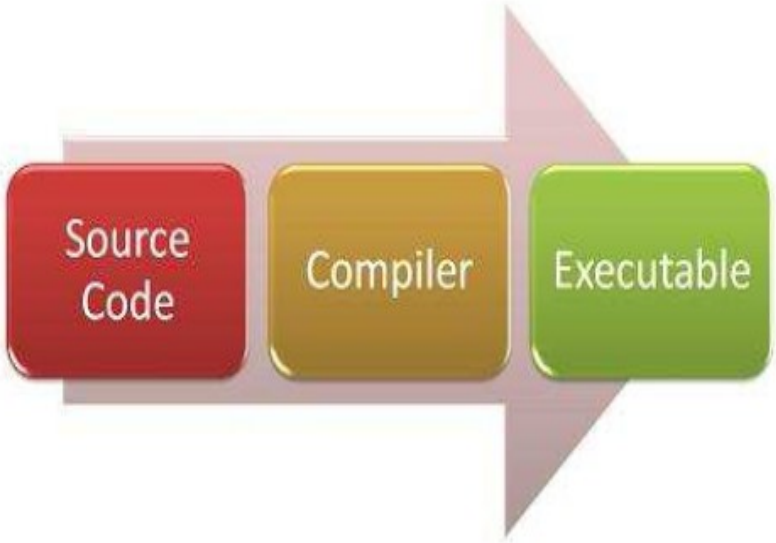


Malware comprehension

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**

OUTLINE

- 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

OUTLINE

- 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

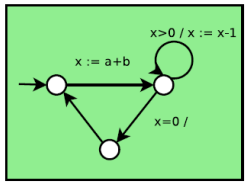
Only honest persons here !!

OUTLINE

- 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

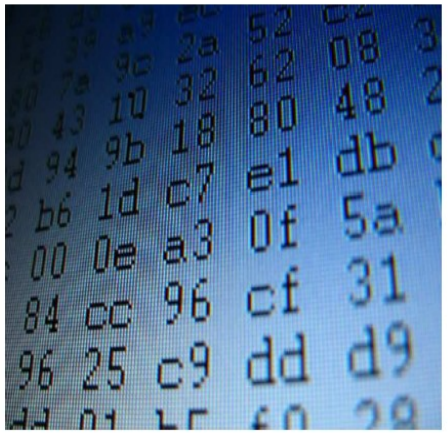
```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

Assembly

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

label:
  move @100 B
```

Executable

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

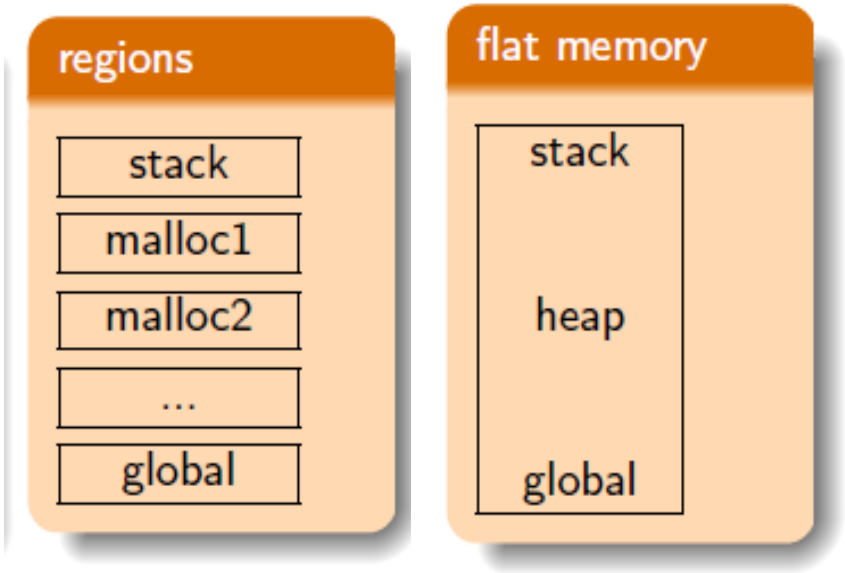


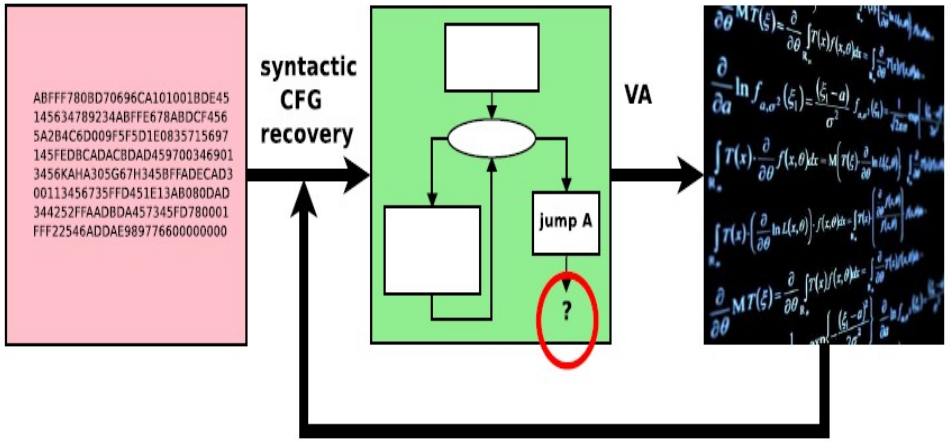
- Binary code
- Attacker
- Properties

CHALLENGE: BINARY CODE LACKS STRUCTURE

- Instructions?
- Control flow?
- Memory structure?

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:
```

Problems

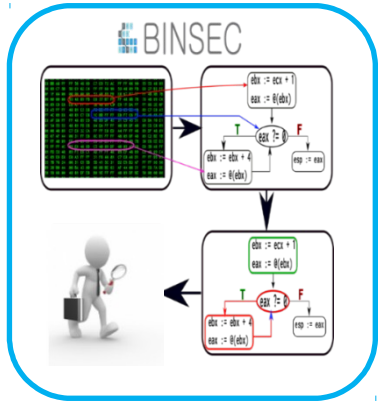
- Jump eax
- Untyped memory
- Bit-level resoning

OUTLINE

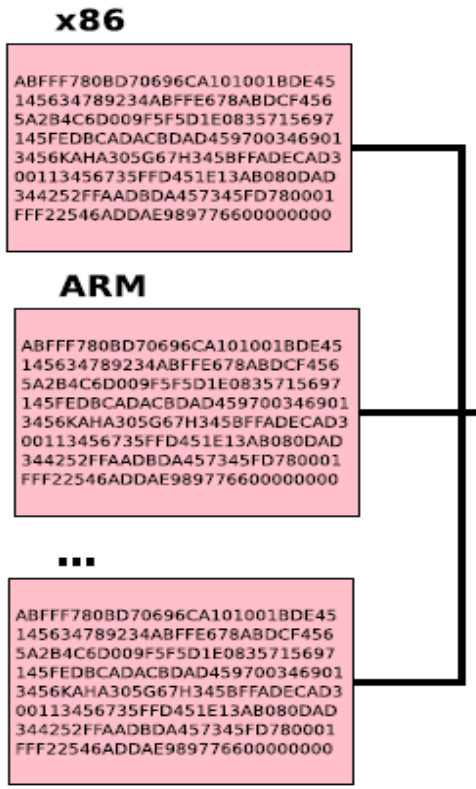
- About me, myself and this talk
- What every honest person should know about Symbolic Execution
- Challenges of binary-level security
- A bit about BINSEC
- Shades of Symbolic Execution for Security
- Conclusion, Take away and Disgression

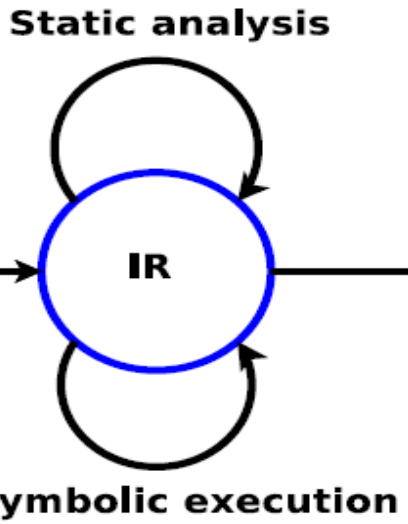
BINSEC: brings formal methods to binary-level security analysis

Break | Prove | Protect

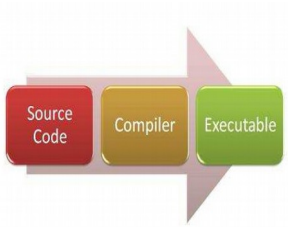


- **Explore many input at once**
  - Find bugs
  - Prove security
- **Multi-architecture support**
  - x86, ARM, RISC-V





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



https://binsec.github.io/

Key 1: INTERMEDIATE REPRESENTATION [CAV'11]

Binsec intermediate representation

$$
\begin{array}{rcl}
\text{inst} & := & \text{lv} \leftarrow \text{e} \mid \text{goto e} \mid \text{if e then goto e} \\
\text{lv} & := & \text{var} \mid @[\text{e}]_n \\
\text{e} & := & \text{cst} \mid \text{lv} \mid \text{unop e} \mid \text{binop e e} \mid \text{e ? e : e} \\
\\
\text{unop} & := & \neg \mid - \mid \text{uext}_n \mid \text{sext}_n \mid \text{extract}_{i..j} \\
\text{binop} & := & \text{arith} \mid \text{bitwise} \mid \text{cmp} \mid \text{concat} \\
\text{arith} & := & + \mid - \mid \times \mid \text{udiv} \mid \text{urem} \mid \text{sdiv} \mid \text{srem} \\
\text{bitwise} & := & \wedge \mid \vee \mid \oplus \mid \text{shl} \mid \text{shr} \mid \text{sar} \\
\text{cmp} & := & = \mid \neq \mid >_u \mid <_u \mid >_s \mid <_s
\end{array}
$$

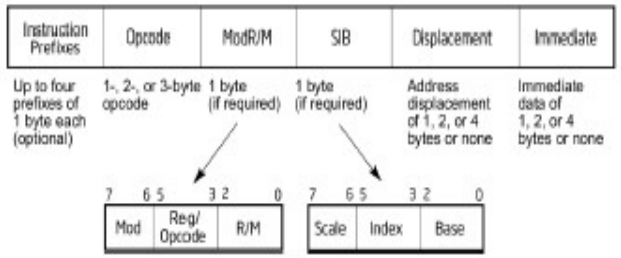
Multi-architecture

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**

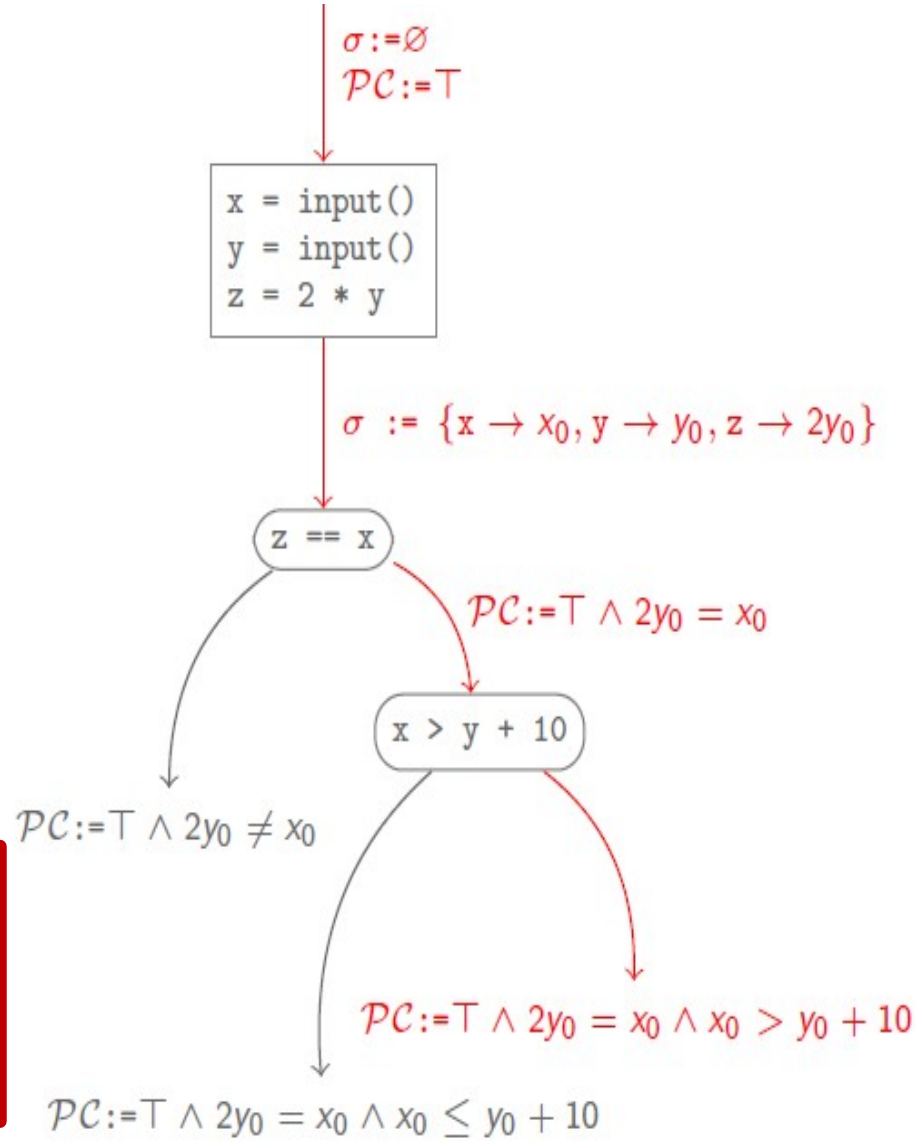
81 c3 57 1d 00 00 $\overset{x86 reference}{\Rightarrow}$ ADD EBX 1d57

```
(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)
```

A TOOL OF CHOICE: SYMBOLIC EXECUTION

- 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 = input following the path
- Solve it with powerful existing solvers

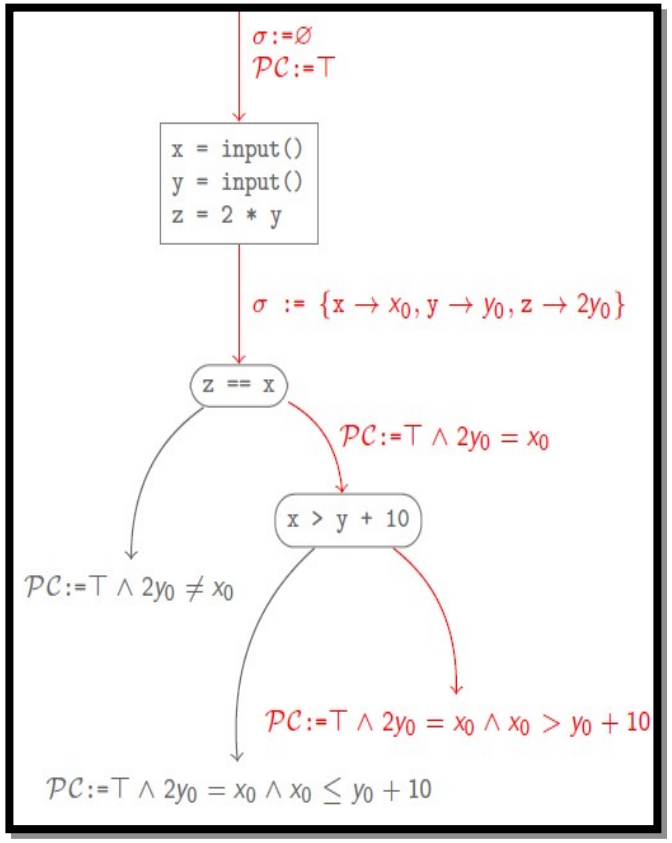
OUTLINE

- 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

OUTLINE

- **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)**

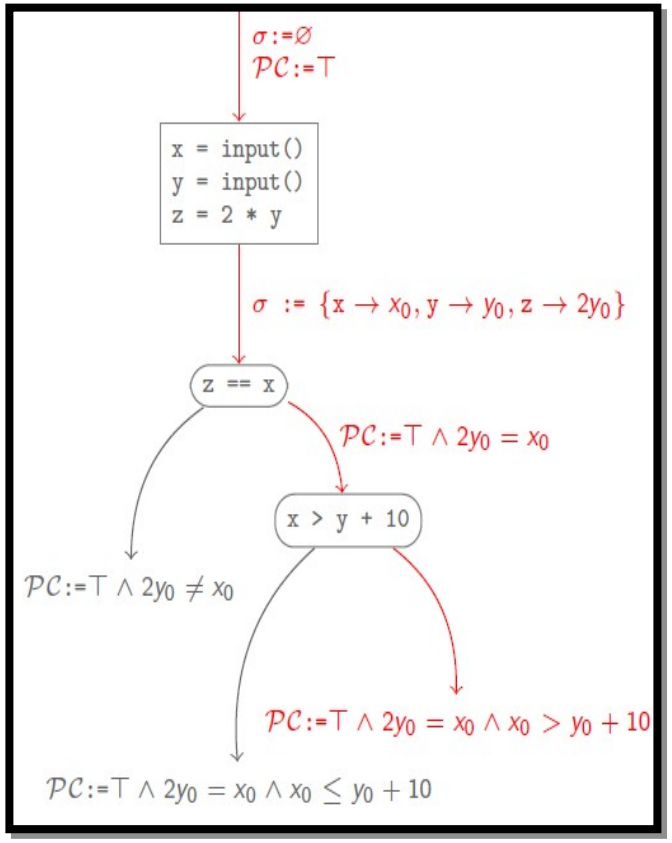
Vulnerability finding with symbolic execution (Godefroid et al., Cadar et al., Sen et al., etc.)



- Intensive path exploration

Challenge = path explosion

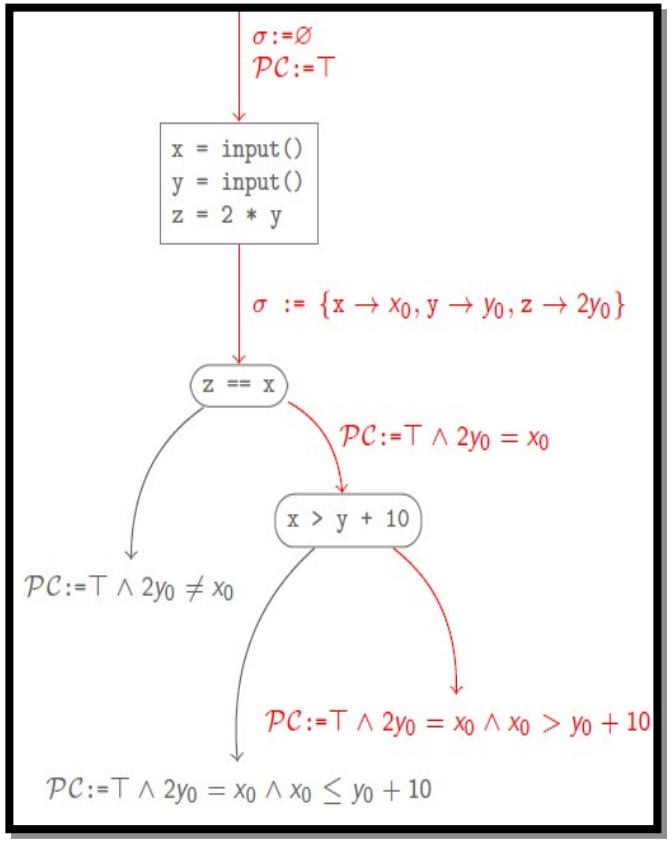
Vulnerability finding with symbolic execution (Godefroid et al., Cadar et al., Sen et al., etc.)



- Intensive path exploration
- Target critical bugs

Challenge = path explosion

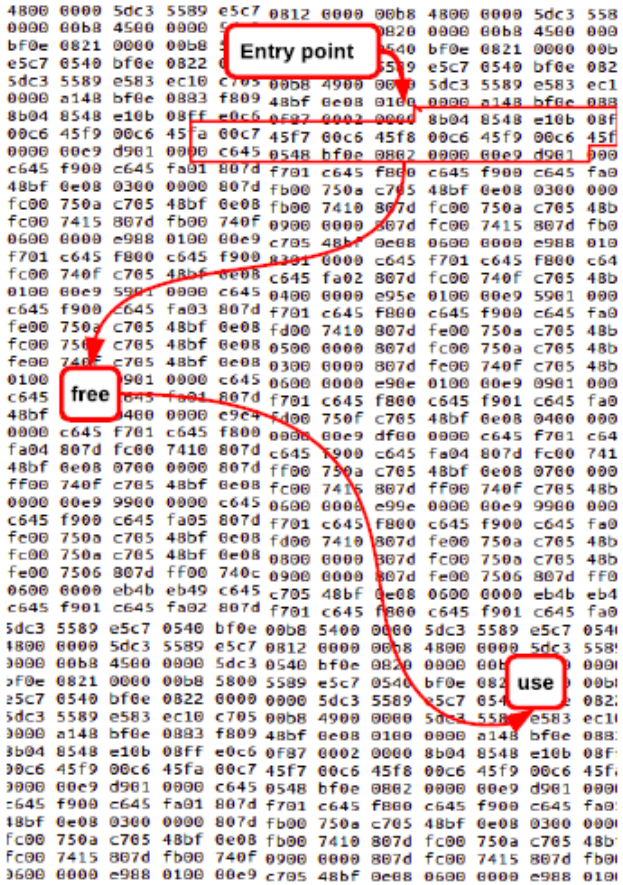
Vulnerability finding with symbolic execution (Heelan, Brumley et al.)



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

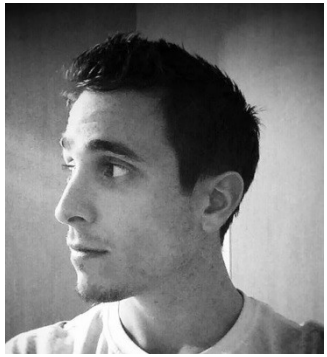
Challenge = path explosion

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

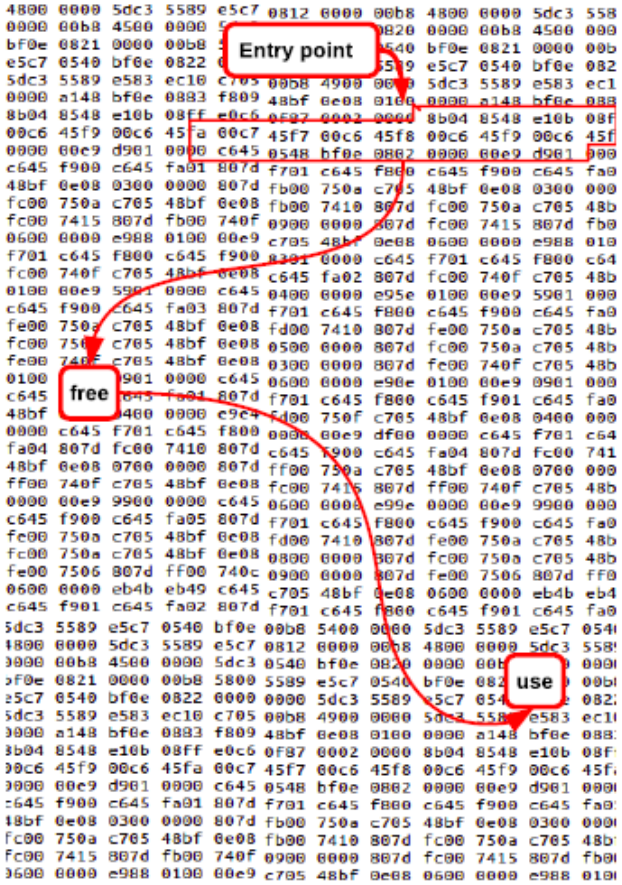


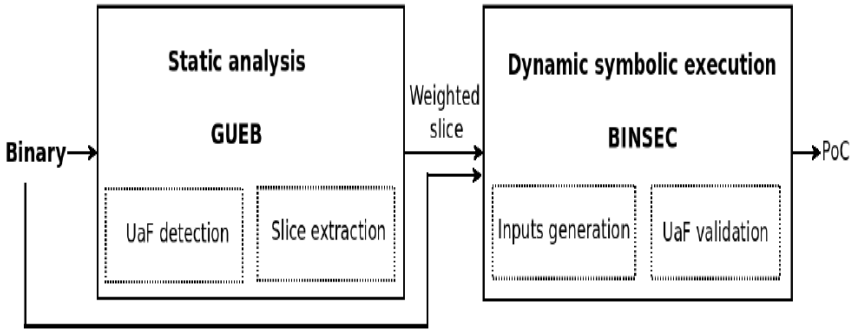
Use-after-free bugs
- Very hard to find
- Sequence of events
- DSE gets lost

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

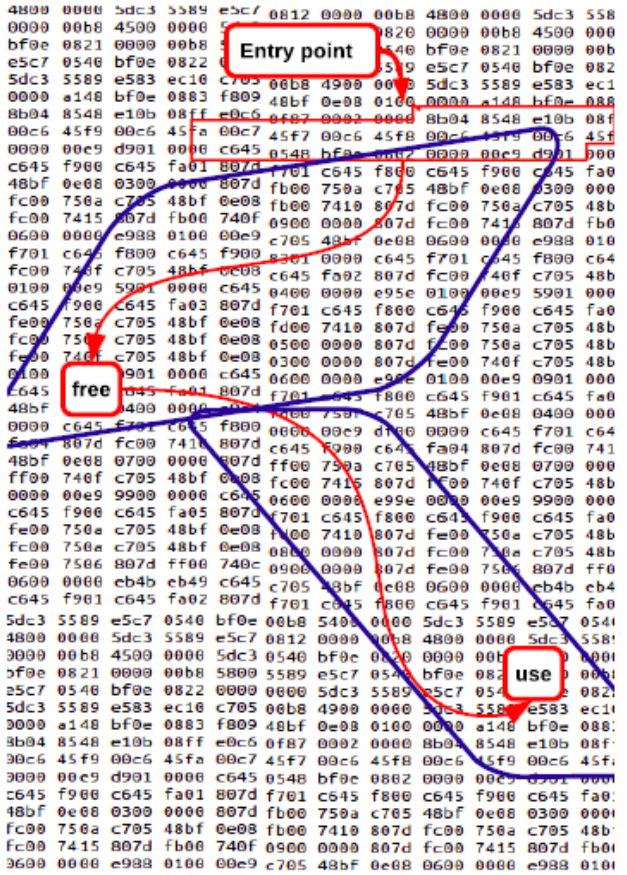


Use-after-free bugs
- Very hard to find
- Sequence of events
- DSE lost





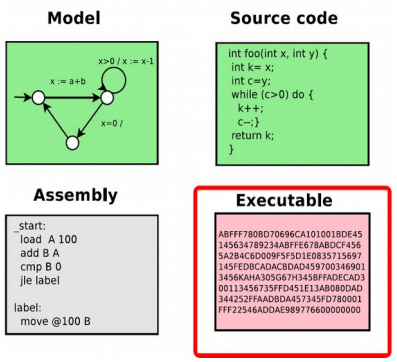
Guide SE with an unsound static analysis

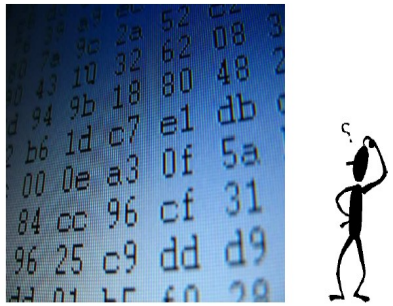


OUTLINE

- **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**





- Binary code
- Attacker
- Properties

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

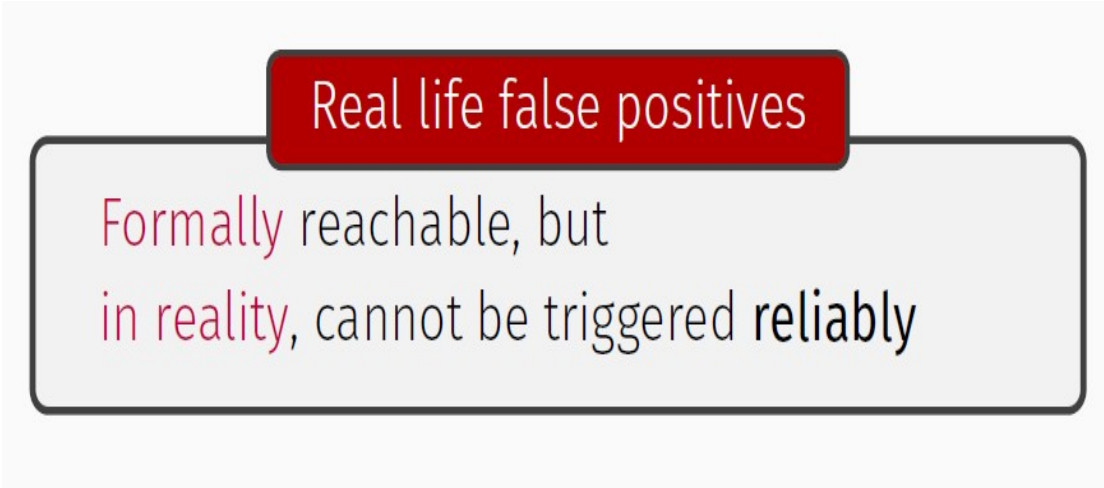
- **Problem: x is not controlled by the user**
  - If x change, possibly not a solution anymore
  - Example: (a = -100, b = 10, x = 1)

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 {
    ...
  }
}
```

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**



Our proposal

Choose a threat Model

Partition input into controlled input a and uncontrolled input x

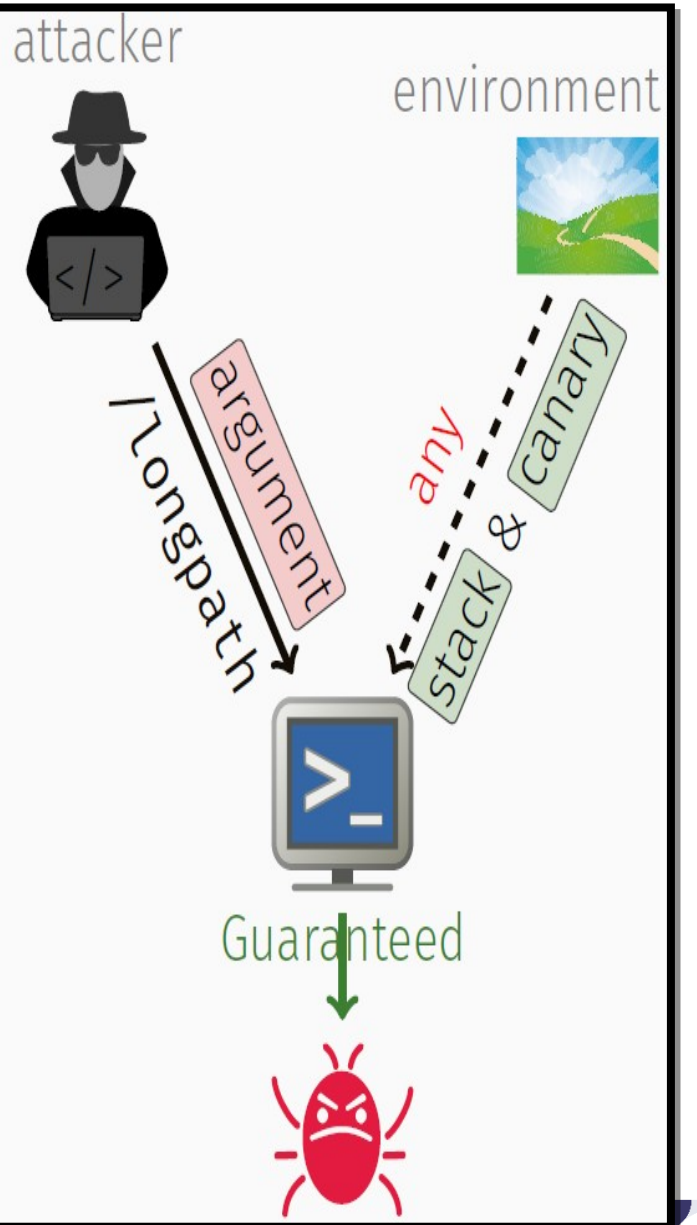
$(a, x) \vdash \ell$ means "with inputs a and x, the program executes code at ℓ"

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 **R**obust **SE** and **R**obust **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**

CVE-2019-19307 in Doas: beyond attacker-controlled input

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

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`"

Versatility of Robust Reachability

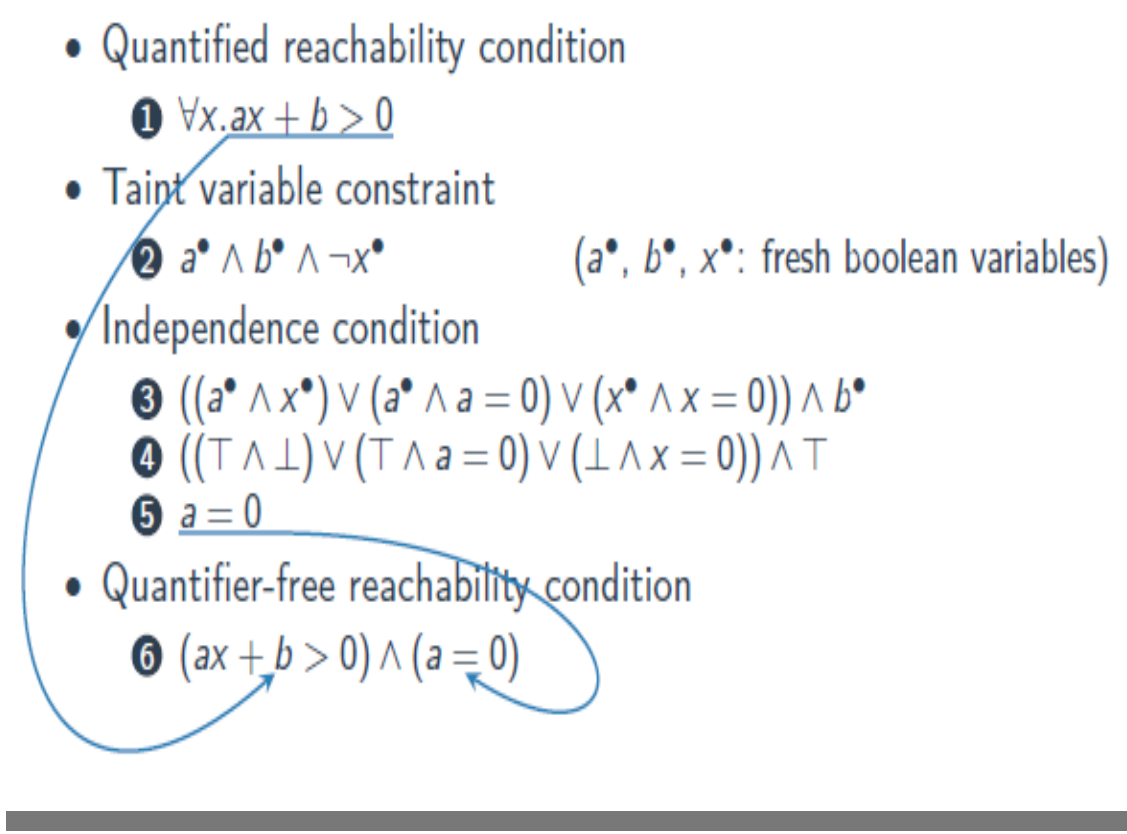
"Controlled inputs" are not limited to "controlled by the attacker"

Example: robustness and quantification [CAV 2018]

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

- Approximation
- But reuse the whole SMT machinery

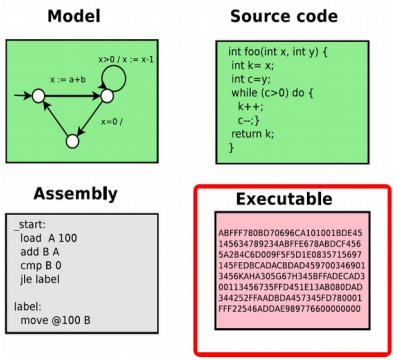
Key insights:
- independence conditions
- formula strengthening

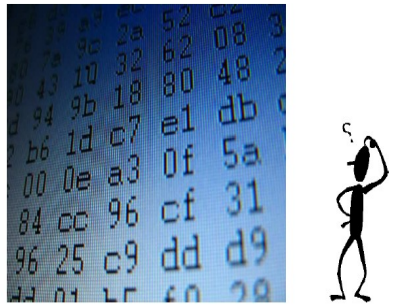


OUTLINE

- **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**



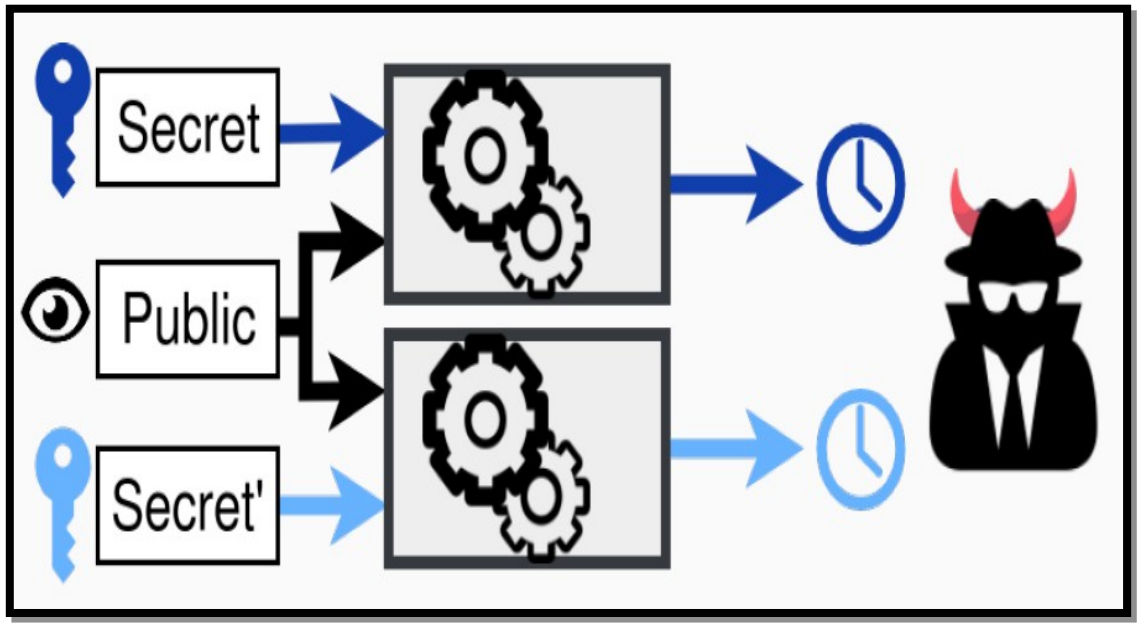


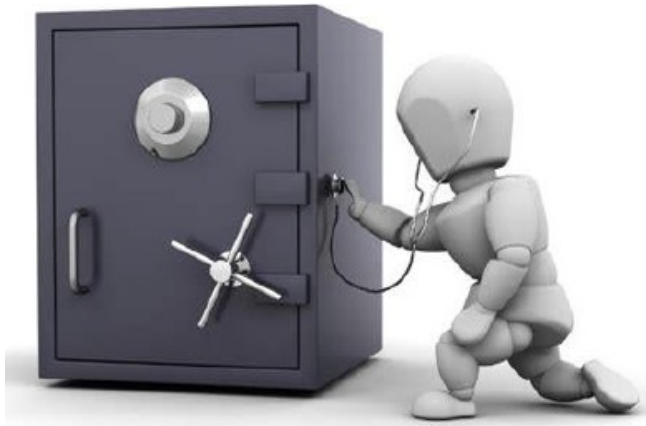
- Binary code
- Attacker
- Properties

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

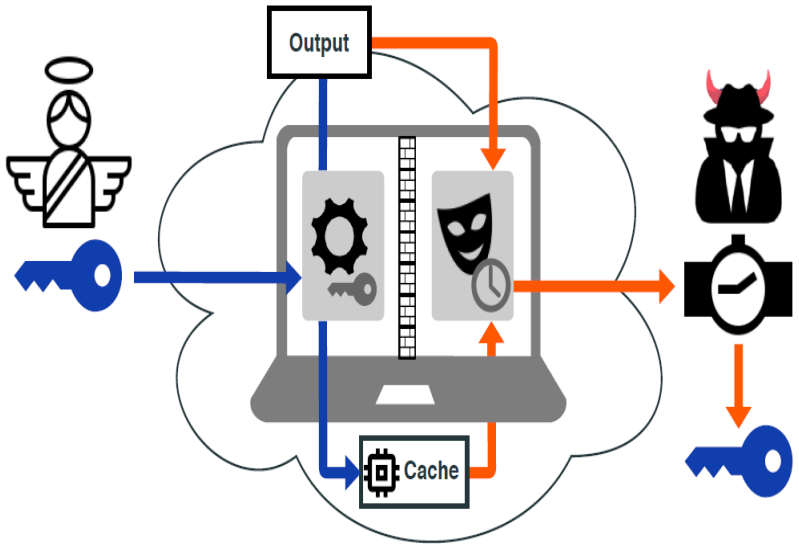
Information leakage

Properties over pairs of executions





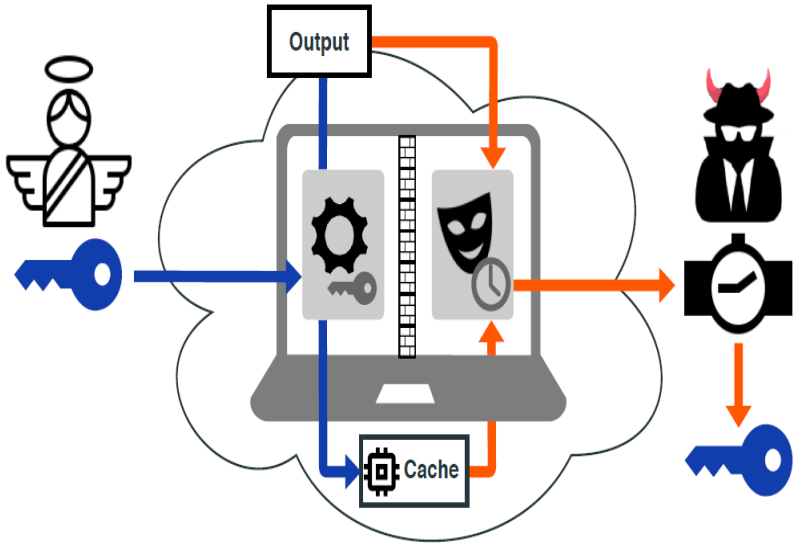
SECURING CRYPTO-PRIMITIVES -- [S&P 2020] (Lesly-Ann Daniel)



		#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	-

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

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



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

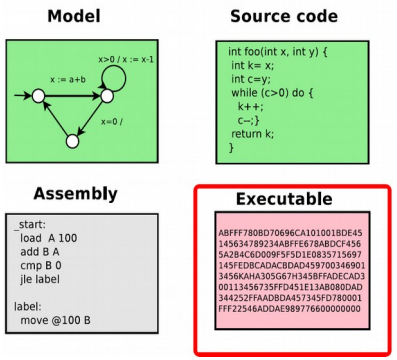
		#Instr static	#Instr unrol.	Time	CT source	Status	🐞	Comment
utility	ct-select	735	767	.29	Y	21×✗	21	1 new ✗
	ct-sort	3600	7513	13.3	Y	18×✗	44	2 new ✗
BearSSL	aes_big	375	873	1574	N	✗	32	-
	des_tab	365	10421	9.4	N	✗	8	-
OpenSSL tls-remove-pad-lucky13		950	11372	2574	N	✗	5	-
Total		6025	30946	4172	-	**42**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

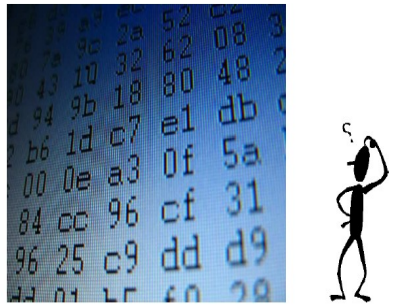
- 397 crypto code samples, x86 and ARM
- New proofs, 3 new bugs (of verified codes)
- Potential issues in some protection schemes
- 600x faster than prior workl

OUTLINE

- **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)**

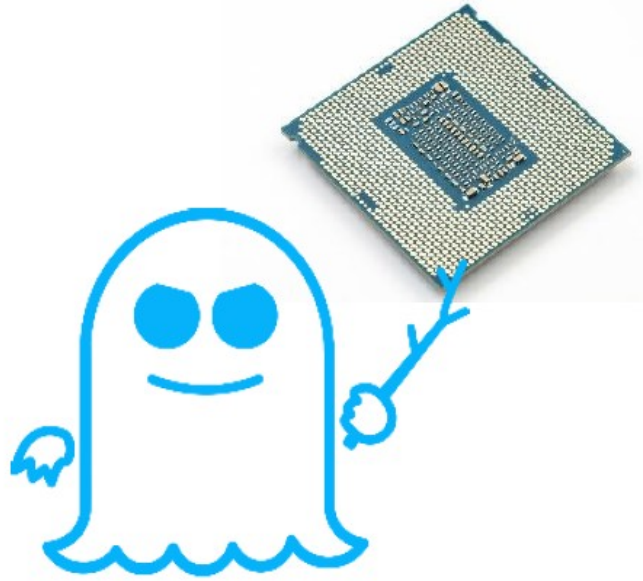
- **Problem : what if the attacker can observe more behaviours?**





- Binary code
- Attacker
- Properties

Speculative executins and Spectre attacks



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)

Challenge !

- 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**

Challenge !

- 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**

- Main idea :
- Smart encoding of speculation
- Can be seen as dedicated merge + targeted simplifications

CEA LIST, Software Safety & Security Lab

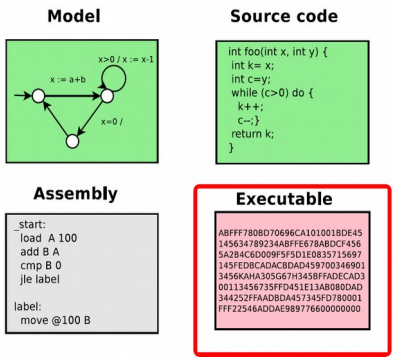
	Target	Spectre-PHT	Spectre-STL
KLEESpectre [1]	LLVM	☺	-
SpecuSym [2]	LLVM	☺	-
FASS [3]	Binary	☹	-
Spectector [4]	Binary	☹	-
Pitchfork [5]	Binary	😐	☹
Binsec/Haunted	Binary	☺	😐

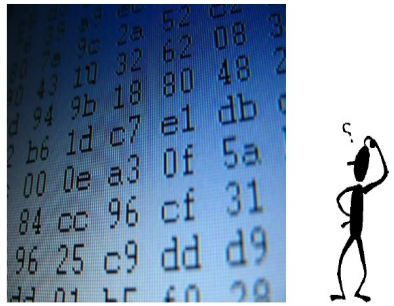
- Fun fact : spectre-pht protections may be vulnerable to spectre-stl

OUTLINE

- **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)**

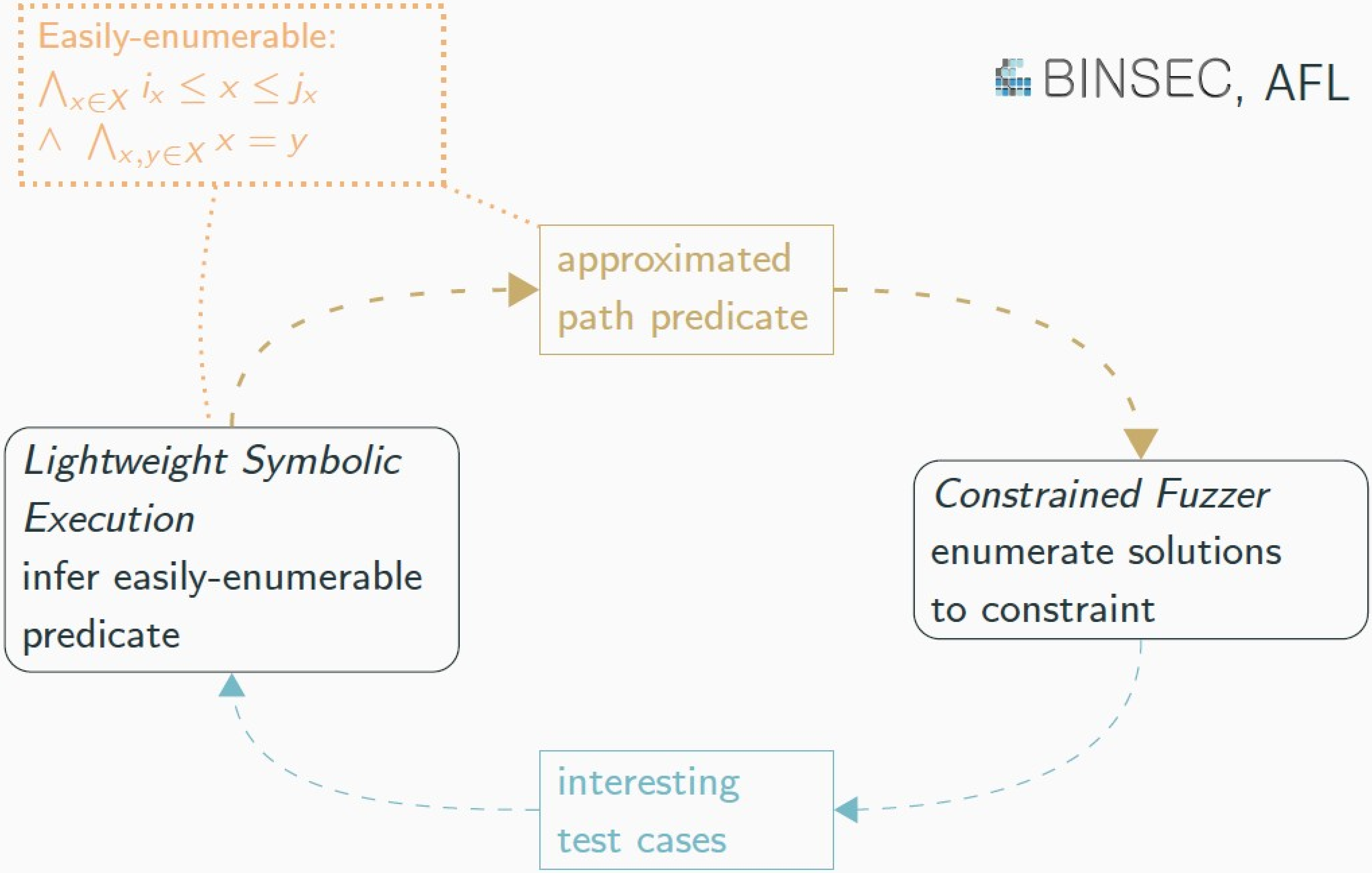
- **Problem : getting one solution is nice, having a lot of them is better**





- Binary code
- Attacker
- Properties

BINSEC, AFL

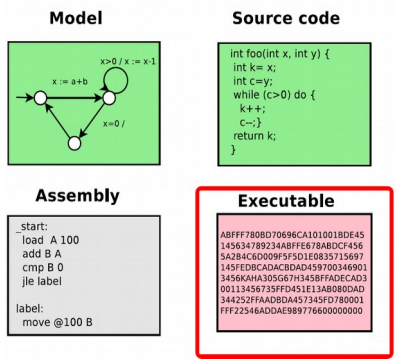


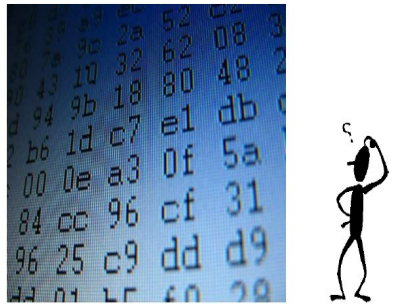
Easily-enumerable:
$\bigwedge_{x \in X} i_x \leq x \leq j_x$
$\wedge \bigwedge_{x,y \in X} x = y$
approximated path predicate
Lightweight Symbolic Execution
infer easily-enumerable predicate
Constrained Fuzzer
enumerate solutions to constraint
interesting test cases

OUTLINE

- **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**

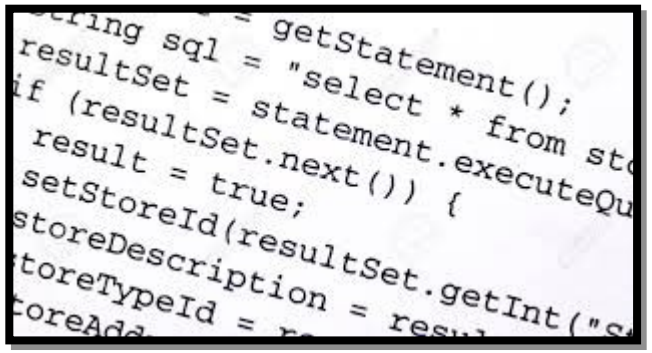


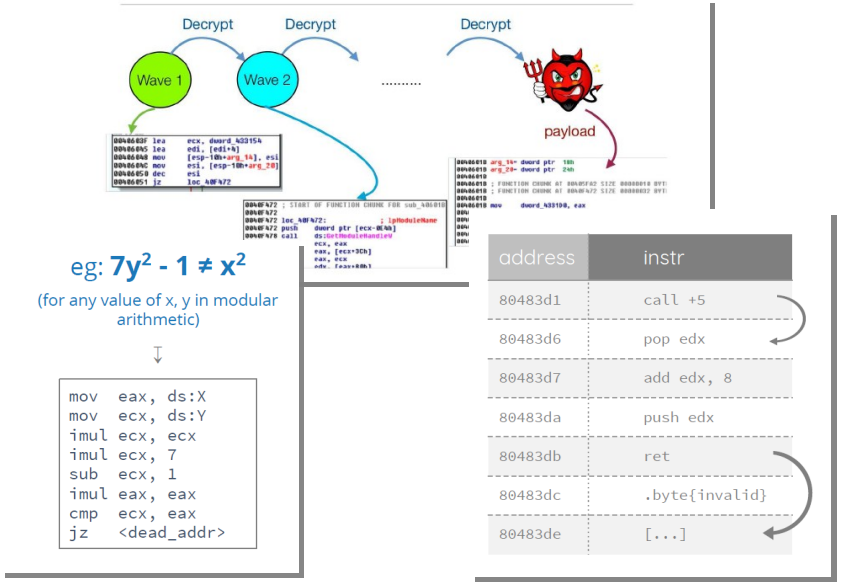


- Binary code
- Attacker
- Properties

CASE 2: code deobfuscation

- Adversarial code





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)

↧

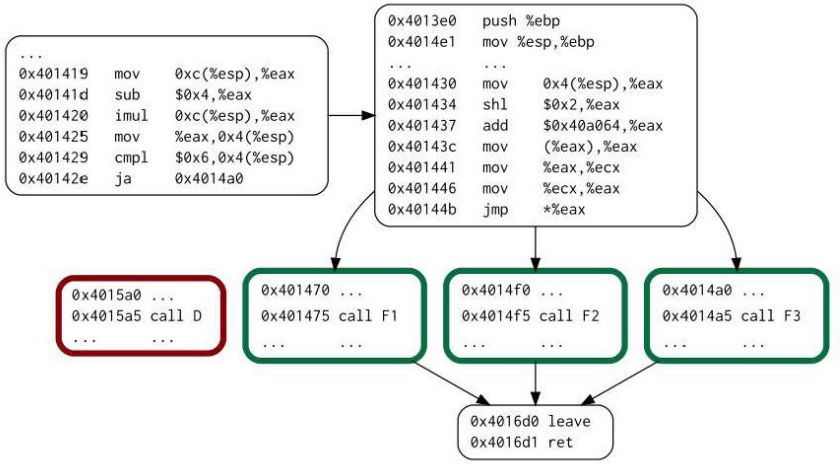
```
mov  eax, ds:X
mov  ecx, ds:Y
imul ecx, ecx
imul ecx, 7
sub  ecx, 1
imul eax, eax
cmp  ecx, eax
jz   <dead_addr>
```

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:
```

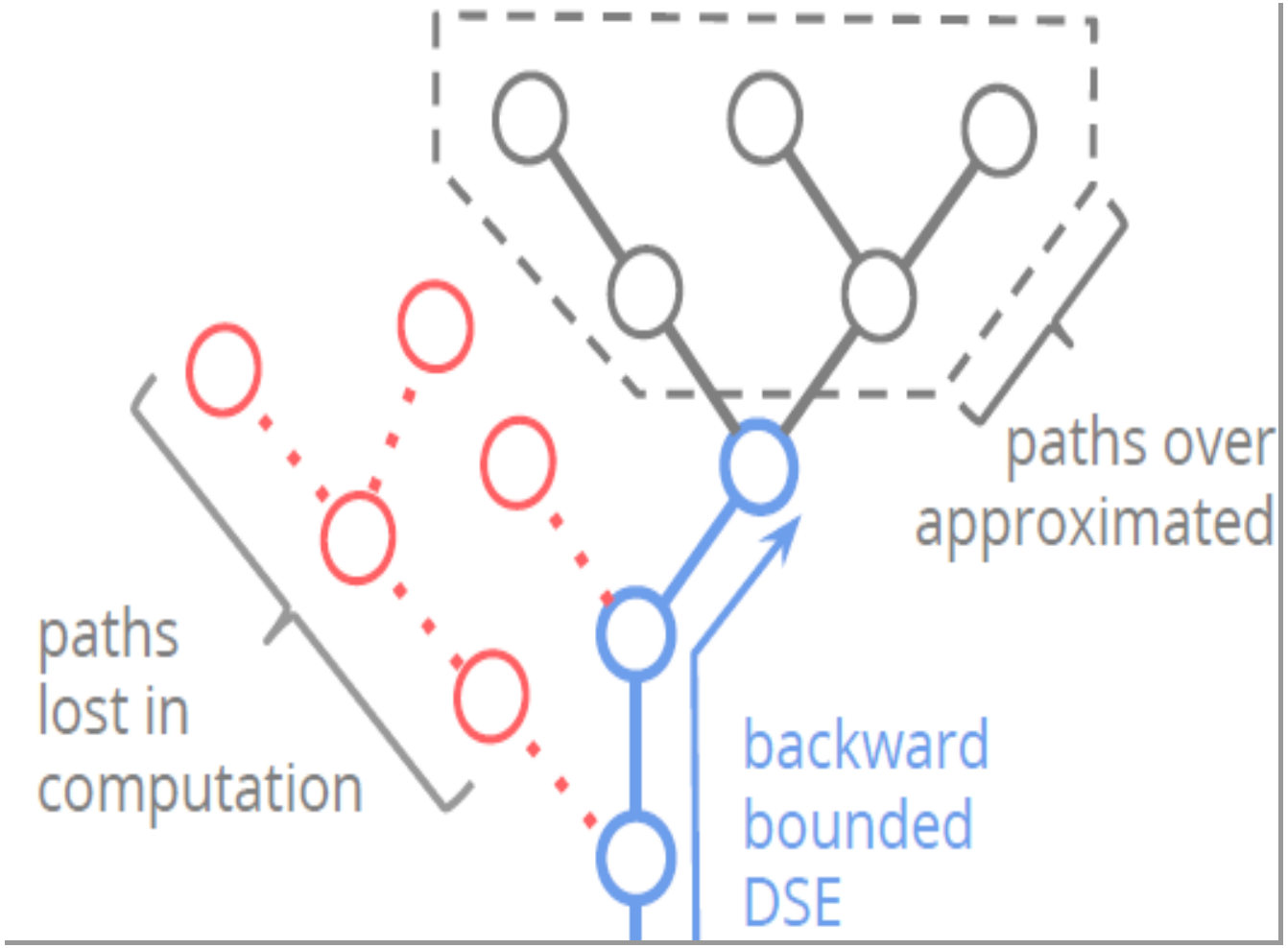
The two blocks are equivalent



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

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: 12min]

- Backward-bounded SE
- + dynamic analysis

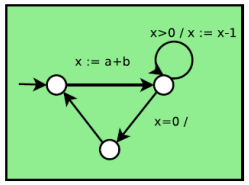
	C637 Sample #1	99B4 Sample #2
#total instruction	**505,008**	**434,143**
#alive	+279,483	+241,177

OUTLINE

- 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



Source code

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

- Binary code

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
```

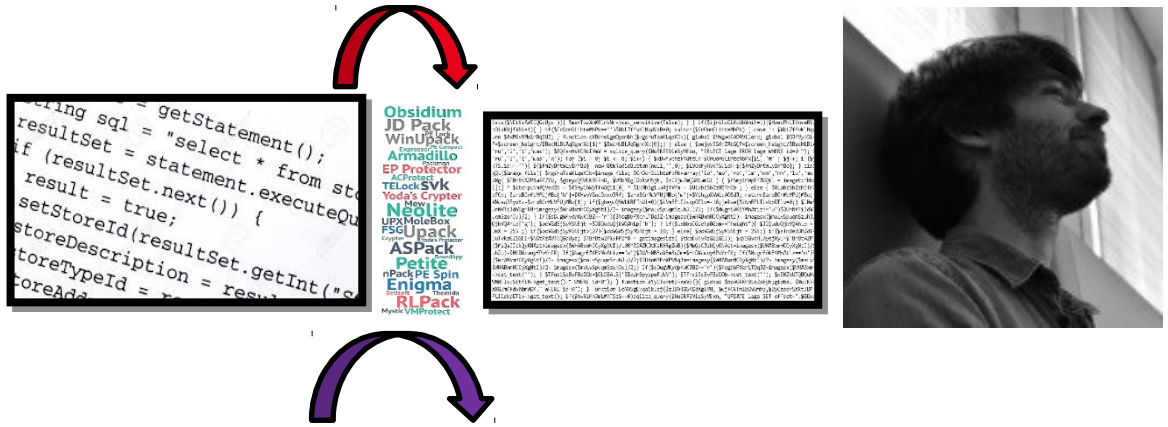
- Attacker

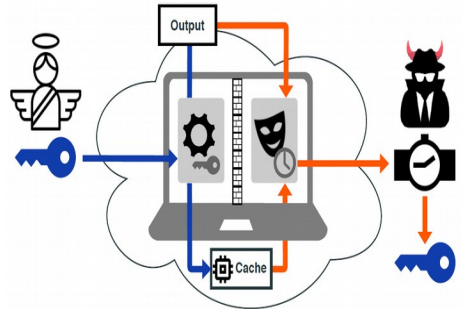
- Properties

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**





- Some queries: 24h => 1min

- 600x faster than prior approach

Conclusion

Fun for FM/PL researchers

Benefit system security

- **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