FROM SAFETY TO SECURITY:
The Case of Binary-Level Code Analysis

Sébastien Bardin
(CEA LIST, France)

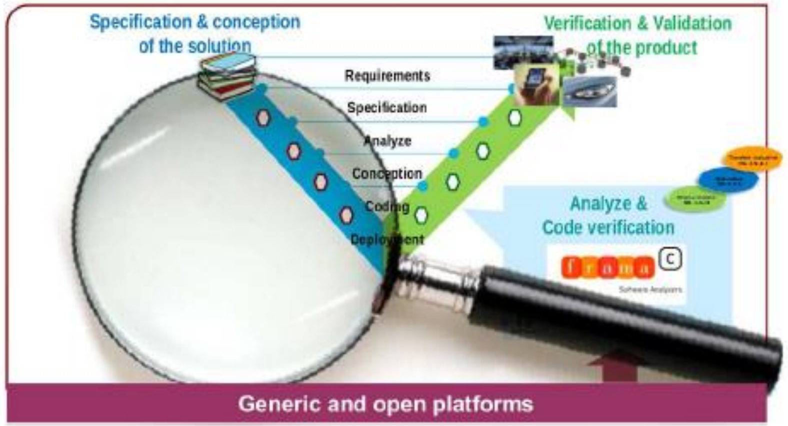
With Richard Bonichon, Matthieu Lemerre,
Robin David, Josselin Feist, Adel Djoudi, Benjamin Farinier, etc.


ABOUT MY LAB @CEA

- Located in Paris, France

CEA LIST, Software Safety & Security Lab

- rigorous tools for building high-level quality software
- second part of V-cycle
- automatic software analysis
- mostly source code



frama C Software Analyzers

GATeL

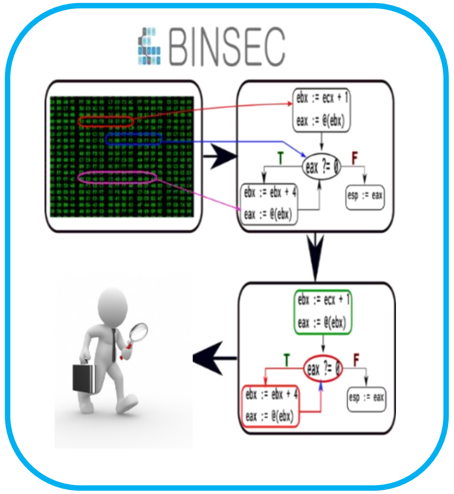
PathCrawler

POLAR SSL

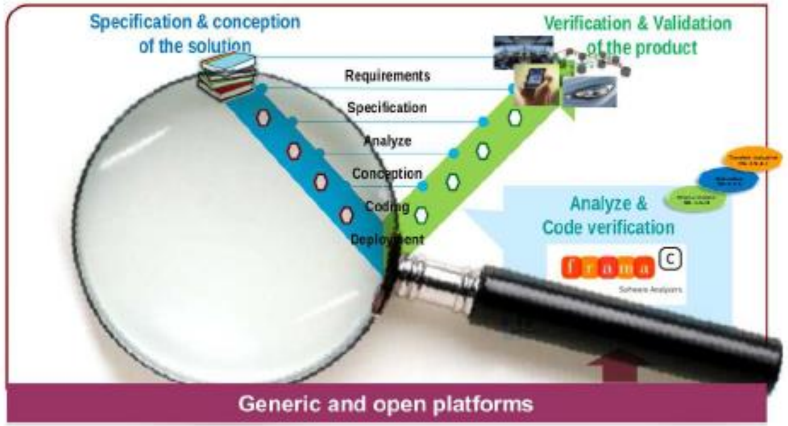
ABOUT MY LAB @CEA

- Located in Paris, France

CEA LIST, Software Safety & Security Lab



- rigorous tools for building high-level quality software
- second part of V-cycle
- automatic software analysis
- mostly source code



frama © Software Analyzers

GATeL

PathCrawler

POLAR SSL

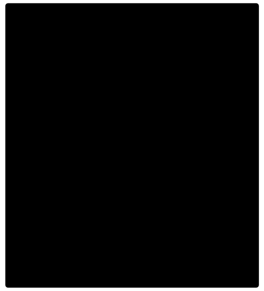
IN A NUTSHELL

- **Binary-level security analysis: many applications, many challenges**

- **Standard techniques not enough**

- **Formal methods can help … but must be strongly adapted**
  - [Complement existing methods]
  - Need robustness, precision and scalability!
  - Acceptable to lose both correctness & completeness – in a controlled way
  - **New challenges and variations, many things to do!**

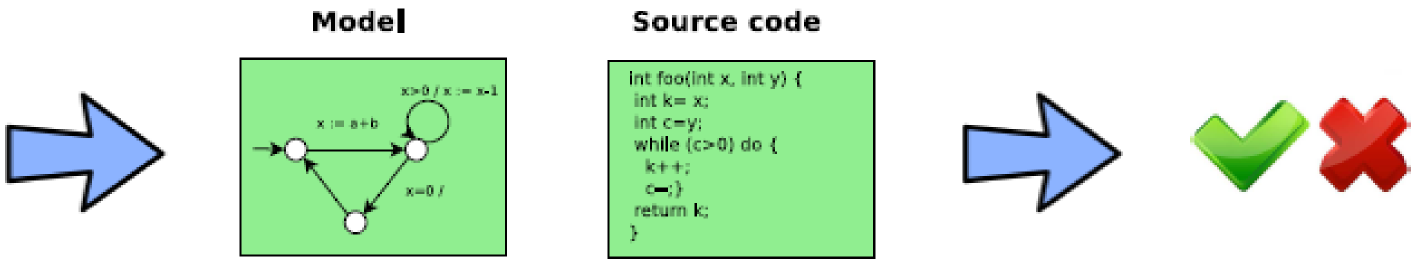
- ***This talk: our experience on adapting source-level safety analysis for binary-level security***



ABOUT FORMAL METHODS

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way

Success in safety-critical



Key concepts : $M \models \varphi$

- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

A DREAM COME TRUE … IN CERTAIN DOMAINS

Ex : Airbus

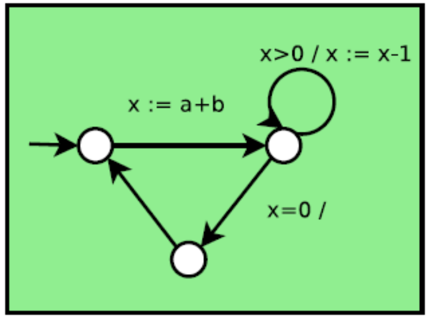
Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C *]
- numerical precision [Fluctuat *]
- source-binary conformance [CompCert]
- ressource usage [Absint]

* : by CEA DILS/LSL

NOW: MOVING TO BINARY-LEVEL SECURITY ANALYSIS

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
```

OUTLINE

- The success of formal methods for safety
- Why binary-level security analysis?
- The hard journey from source-level safety to binary-level security
- Our approach
- Conclusion

WHY BINARY-LEVEL SECURITY ANALYSIS?

Malware comprehension

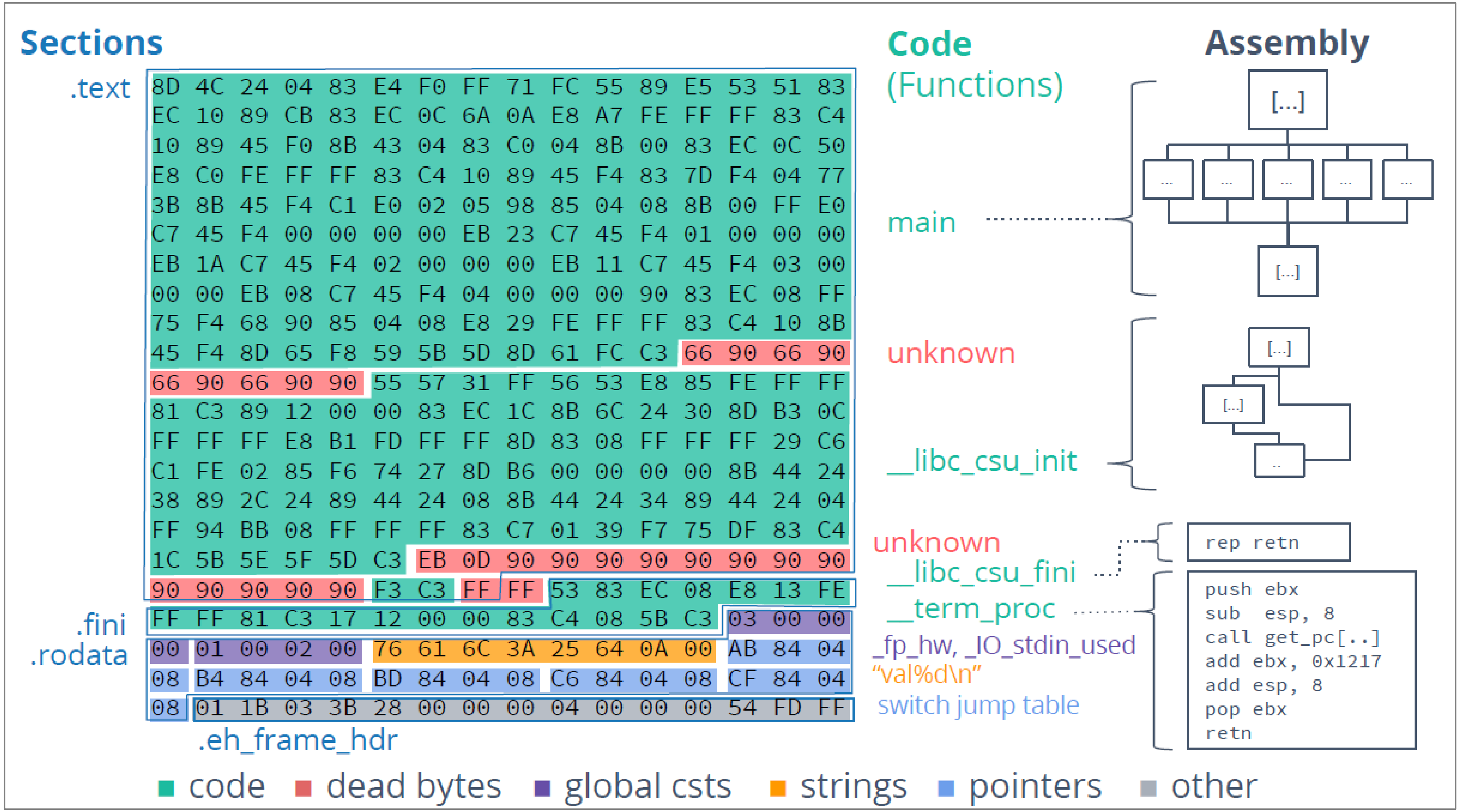
Protection evaluation

Vulnerability analysis

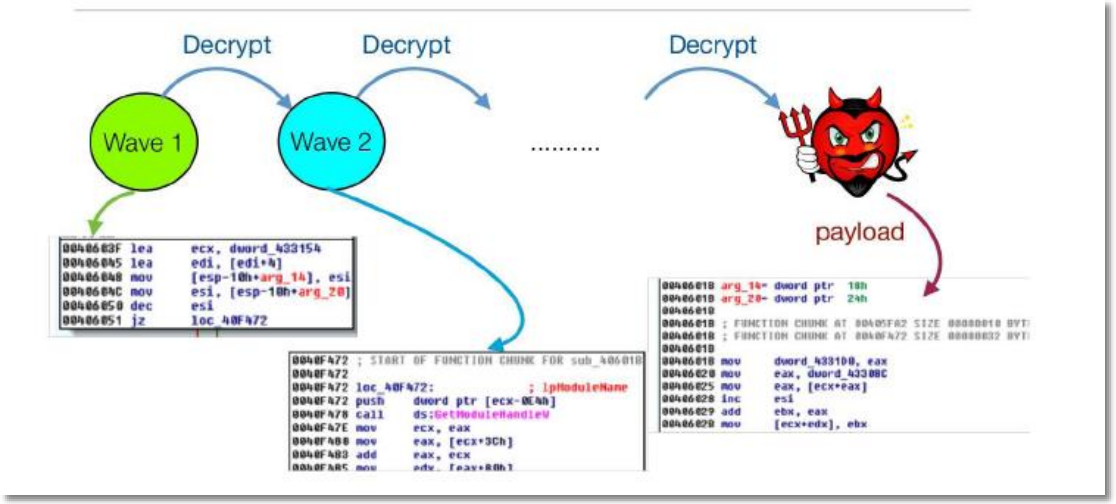
BUT … THIS IS HARD!!!

DISASSEMBLY IS ALREADY TRICKY!

- code – data ??
- dynamic jumps (jmp eax)



AND IT CAN GET WORST! (adversarial setting)



address	instr
80483d1	call +5
80483d6	pop edx
80483d7	add edx, 8
80483da	push edx
80483db	ret
80483dc	.byte{invalid}
80483de	[...]

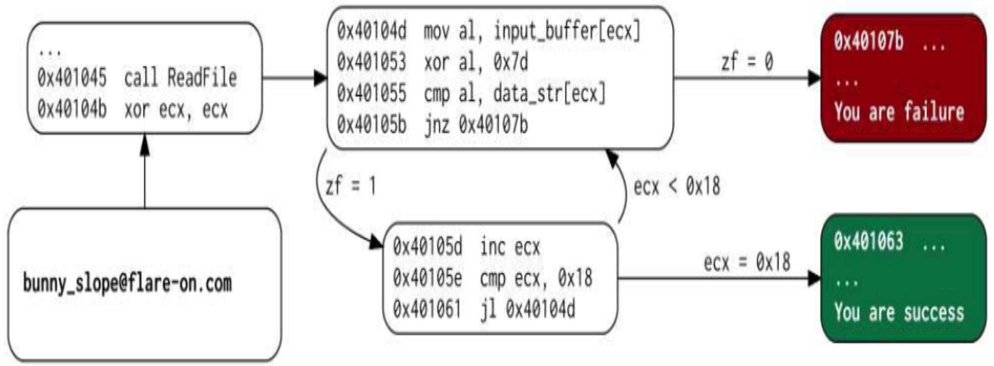
eg: $7y^2 - 1 \neq x^2$

(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>
```

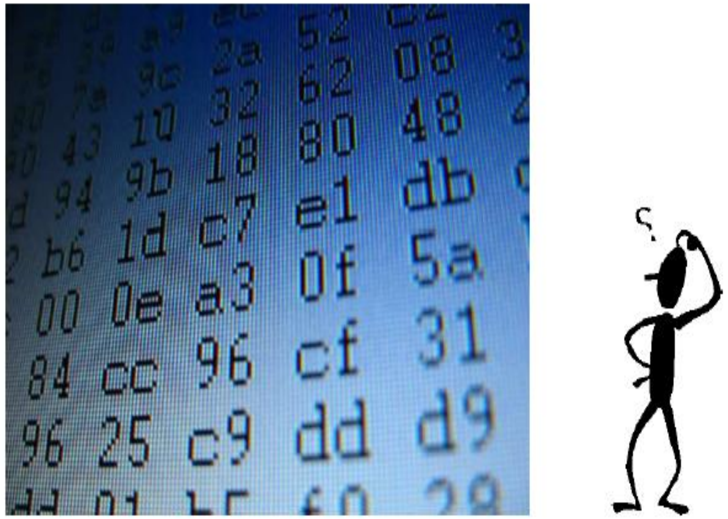
- **self-modification**
- **encryption**
- **virtualization**
- **code overlapping**
- **opaque predicates**
- **callstack tampering**
- **…**



An old proverb comes true

Key concepts : $M \models \varphi$

- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check



We have a beautiful hammer

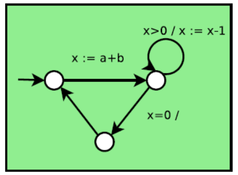
And it seems we found a nail ☺

OUTLINE

- The success of formal methods for safety
- Why binary-level security analysis?
- **The hard journey from source-level safety to binary-level security**
- Our approach
- Conclusion

BINARY-LEVEL ANALYSIS

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
```

Break an implicit assumption in code analysis

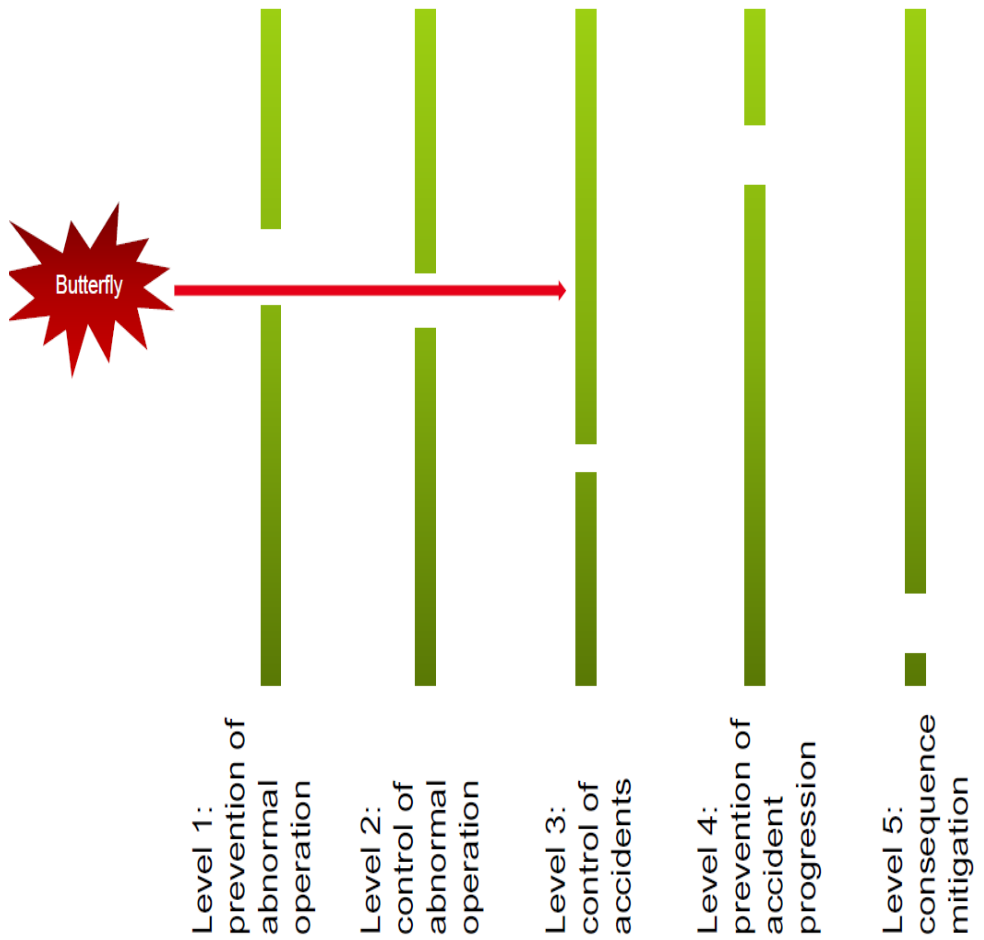
- **Low-level control (CFG?)**
- **Low-level data & memory**

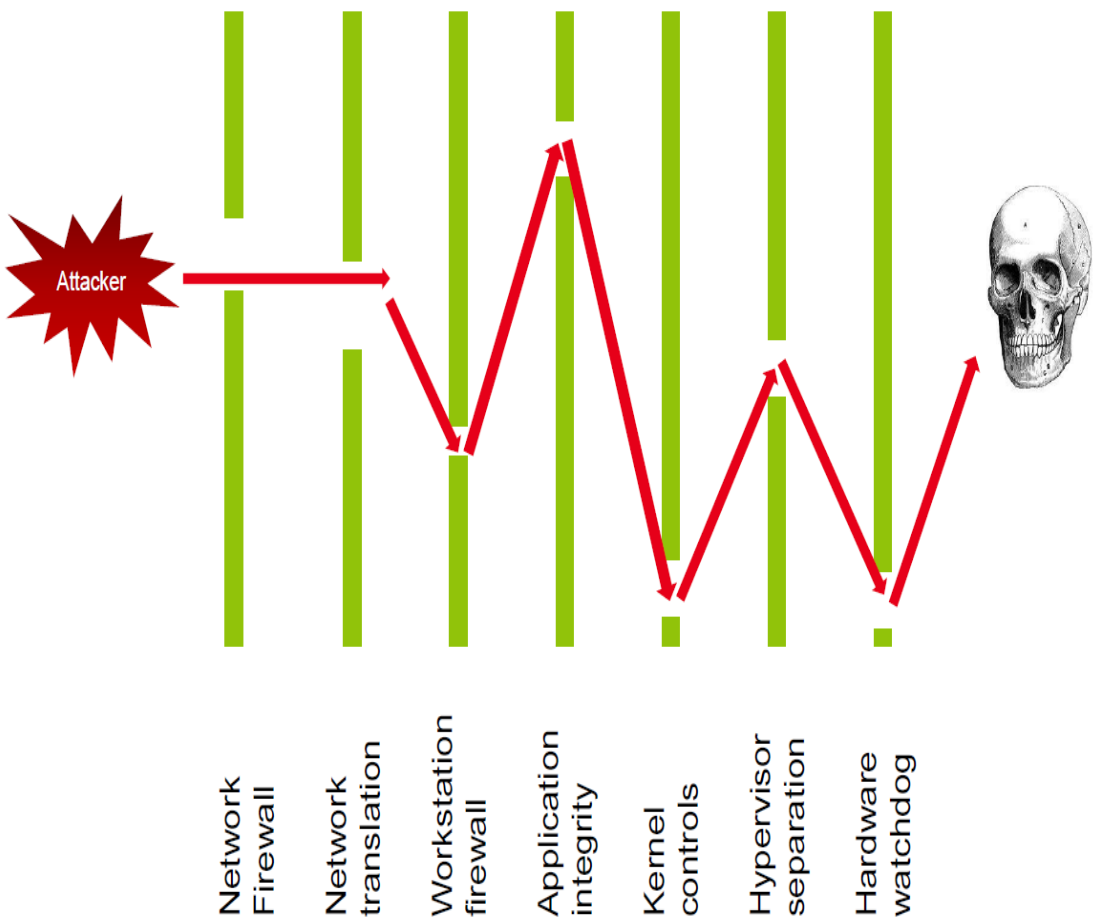
At the edge of current methods

ATTACKER <not treated today>

Nature is not nice

Attacker is evil





From Florent Kirchner

NOT STRONGLY REGULATED ECOSYSTEM

- **No coding guideline**
- **No annotation**
- **Require full automation**
- **Low tolerance to false positive**

But absolute correctness is not required

- « correct enough »

SECURITY IS NOT SAFETY

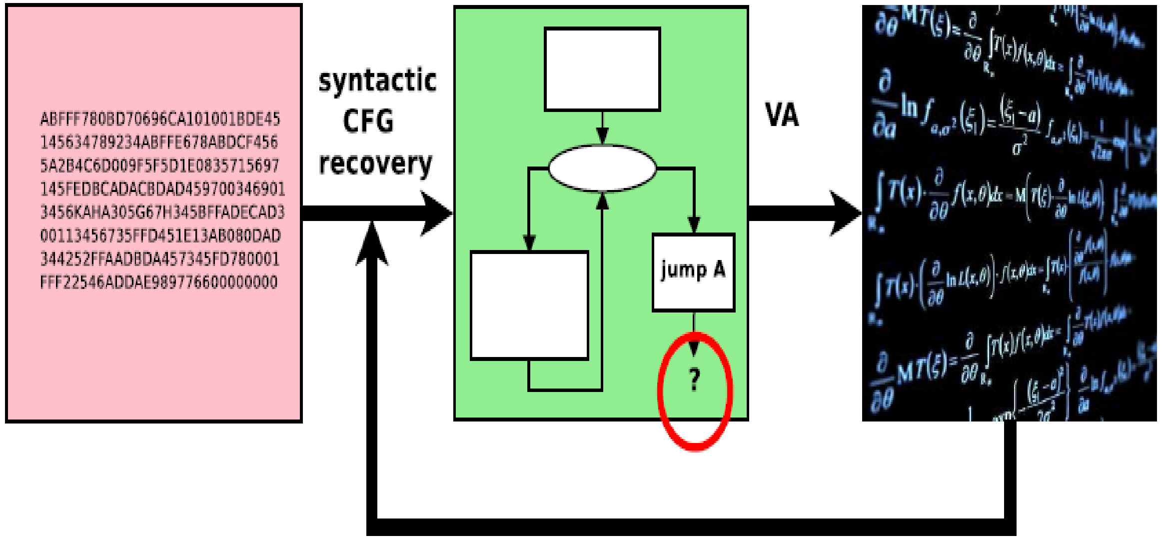
- Strong incentive to human assistance
- Spec., parameter tuning, etc.

- **Source-level SAFETY**
  - Model: High-level language
  - Properties: safety
  - Algorithm: full correctness, possible help from user

- no human assistance
- low tolerance to false positive

- **Binary-level SECURITY**
  - Model: binary-level code, possibly adversarial, + attacker
  - Properties: safety, k-safety, « bugs vs vulnerabilities »
  - Algorithm: robust & precise enough, fully automated

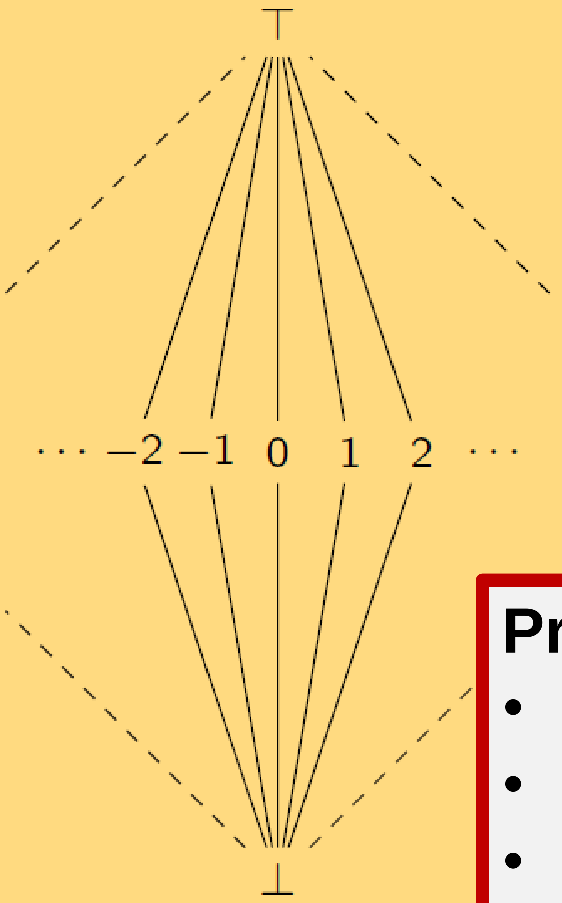
<aparté> STATIC SEMANTIC ANALYSIS IS VERY VERY HARD ON BINARY CODE



Framework : abstract interpretation

- notion of abstract domain
  $\bot, \top, \sqcup, \sqcap, \sqsubseteq$, eval$^{\#}$
- more or less precise domains
  . intervals, polyhedra, etc.
- fixpoint until stabilization

Generalize constant propagation



Problems

- Jump eax
- memory
- Bit resoning

WANTED

Robustness

- able to survive dynamic jumps, self-modification, unpacking, etc
- *outside the scope of standard methods*

Precision

- Machine arithmetic (overflow) and bit-level operations
- Byte-level memory, possible overlaps
- *hard for state-of-art formal methods*

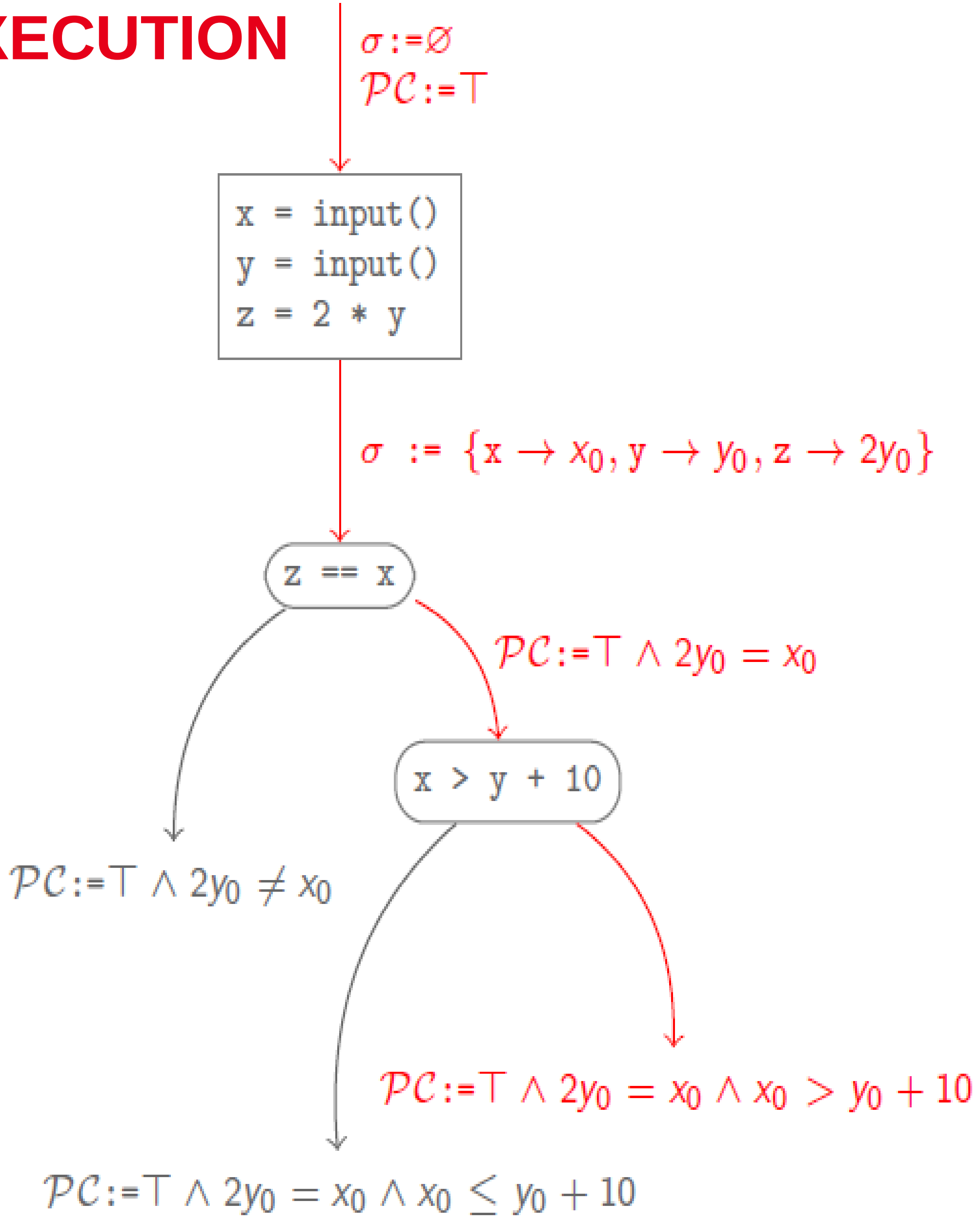
Reasonable scale

OUTLINE

- The success of formal methods for safety
- Why binary-level security analysis?
- The hard journey from source-level safety to binary-level security
- **Our approach**
- Conclusion

THE GOOD CANDIDATE: SYMBOLIC EXECUTION (Godefroid, 2005)

```
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

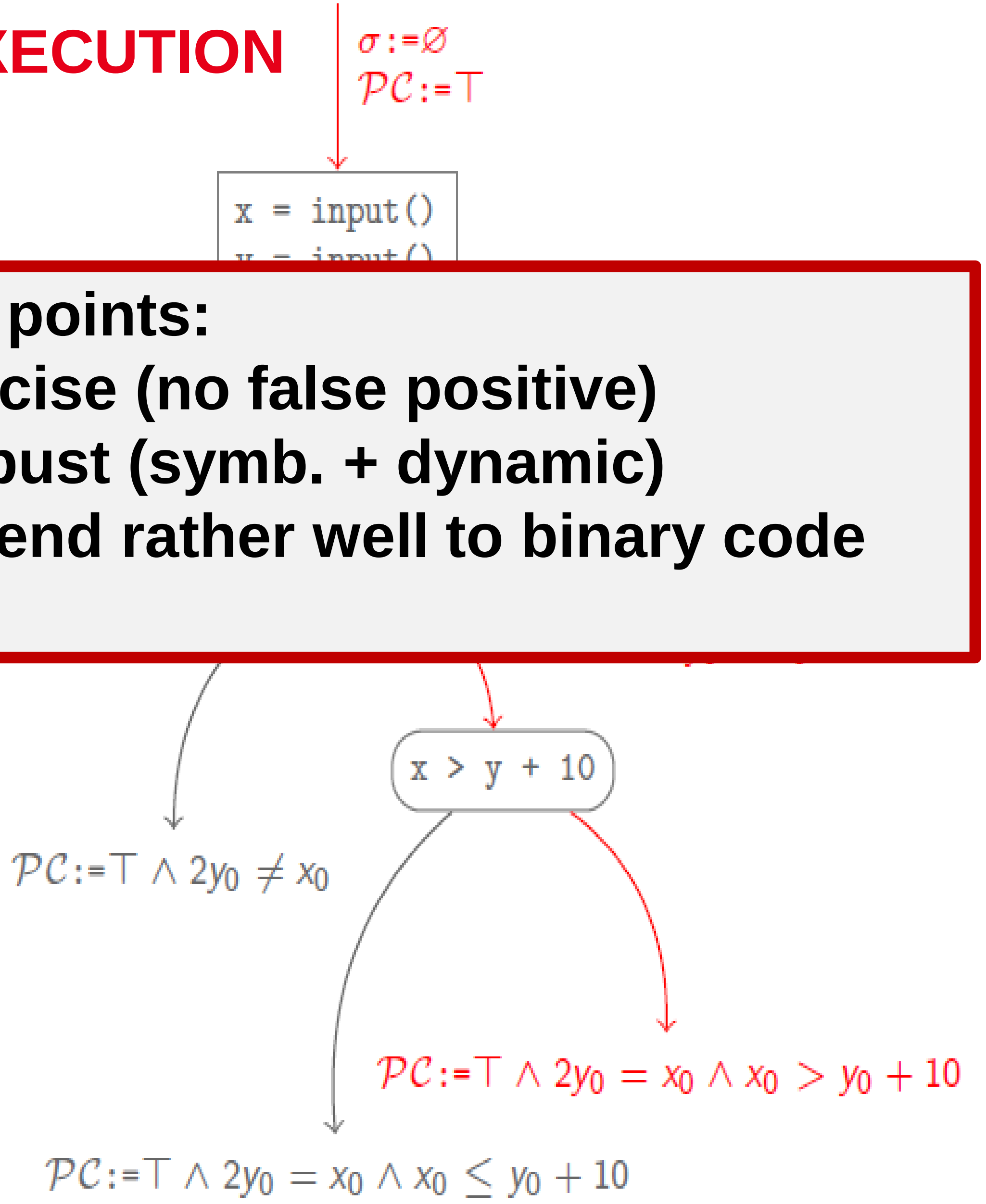
- Compute its « path predicate » f
- Solution of f ⇔ input following the path
- Solve it with powerful existing solvers

THE GOOD CANDIDATE: SYMBOLIC EXECUTION (Godefroid, 2005)

```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```

Good points:

- **Precise (no false positive)**
- **Robust (symb. + dynamic)**
- **Extend rather well to binary code**

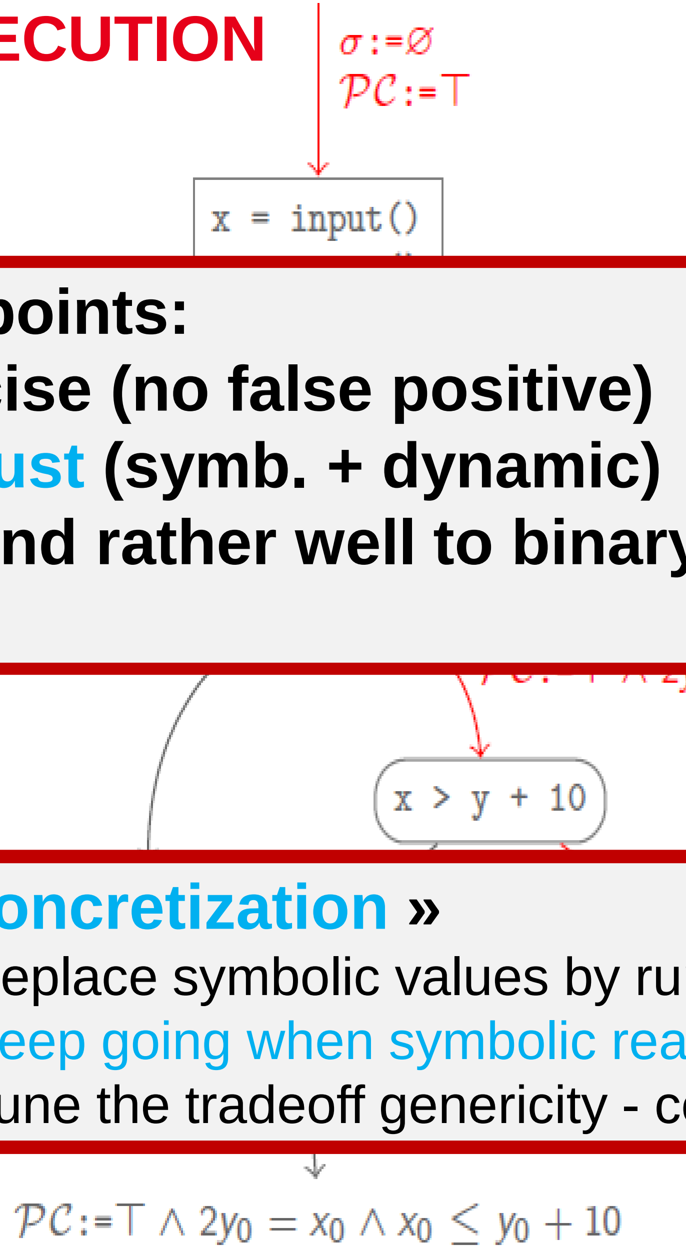


Given a path of a program

- Compute its « path predicate » f
- Solution of f ⇔ input following the path
- Solve it with powerful existing solvers

THE GOOD CANDIDATE: SYMBOLIC EXECUTION (Godefroid, 2005)

```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```



Good points:

- **Precise (no false positive)**
- **Robust (symb. + dynamic)**
- **Extend rather well to binary code**

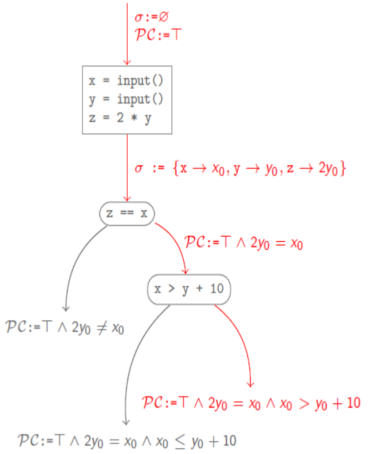
Given a path of a program

- Compute its « path predicate » f
- Solution of f ⇔ input following the path
- Solve it with powerful existing solvers

« concretization »

- Replace symbolic values by runtime values
- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

ALLOWS TO EXPLORE A PROGRAM



Forward reasoning

- Follows path
- Find new branch / jumps
- Standard DSE setting

GRUB2 CVE 2015-8370

Elevation of privilege

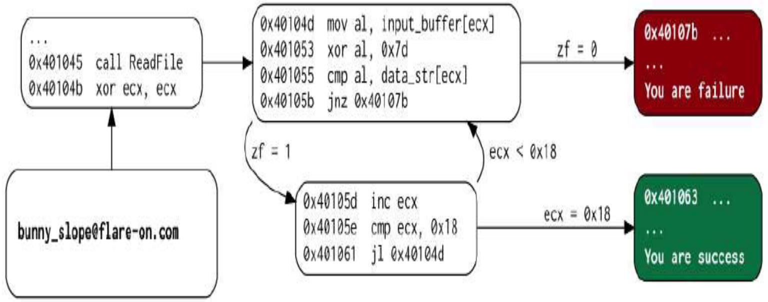
Information disclosure

Denial of service

Advantages

- Find new real paths
- Even rare paths

« dynamic analysis on steroids »



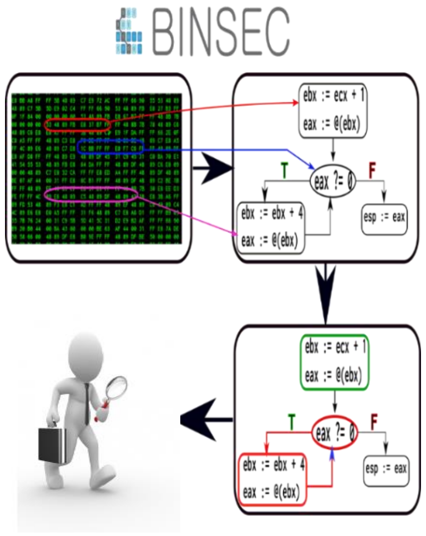
OUR KEY PRINCIPLES

- **DSE is a good starting point for robustness & precision**

- **Can be adapted beyond the basic reachability case**
  - variants
  - combination with other techniques

- **Loss of guarantees**
  - Accept … But control!
  - Look for « correct enough » solutions

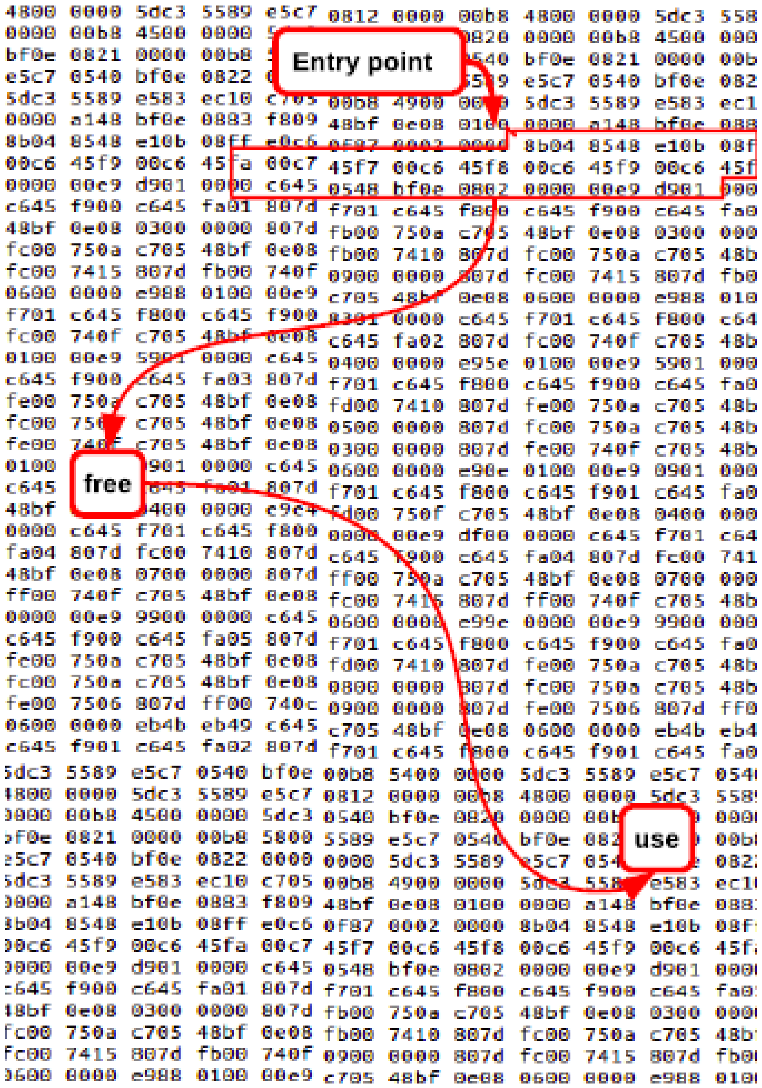
- **Finely tune the technology**
  - Tools for safety are not fully adequate



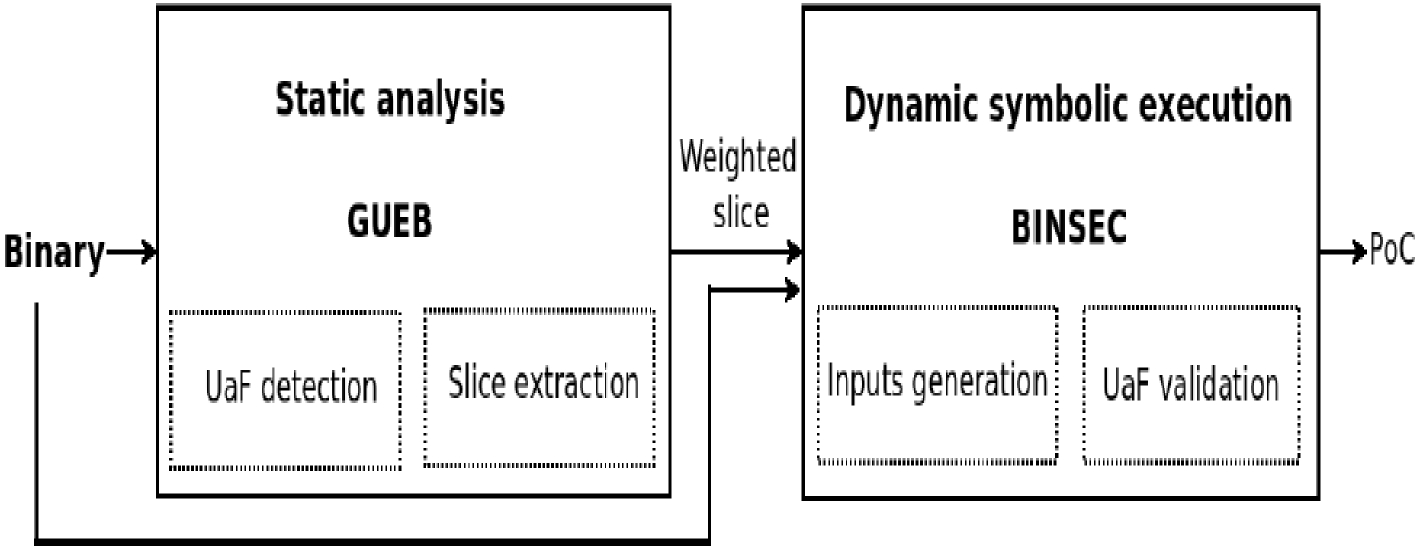
CASE 1: COMPLEX VULNERABILITY DETECTION [SSPREW'16](with Josselin Feist et al.)

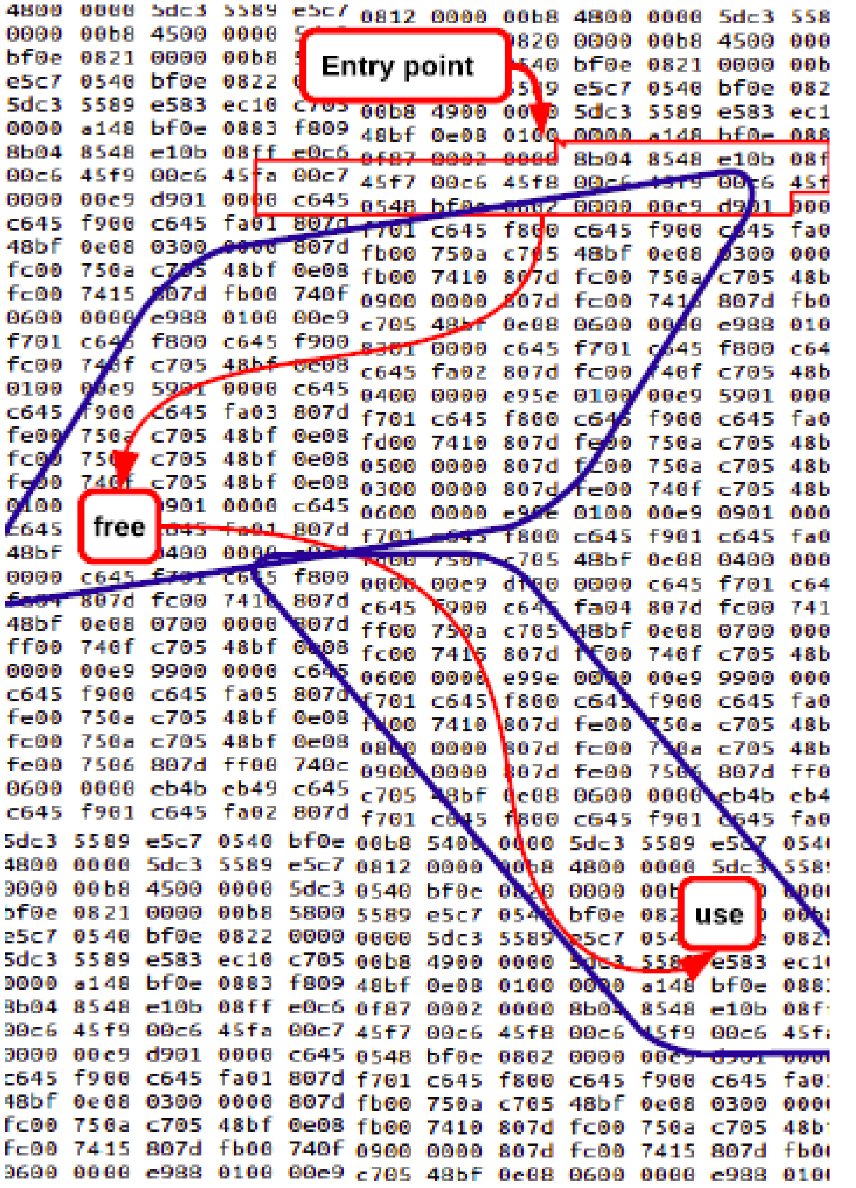
Use-after-free bugs

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



CASE 1: COMPLEX VULNERABILITY DETECTION [SSPREW'16](with Josselin Feist et al.)



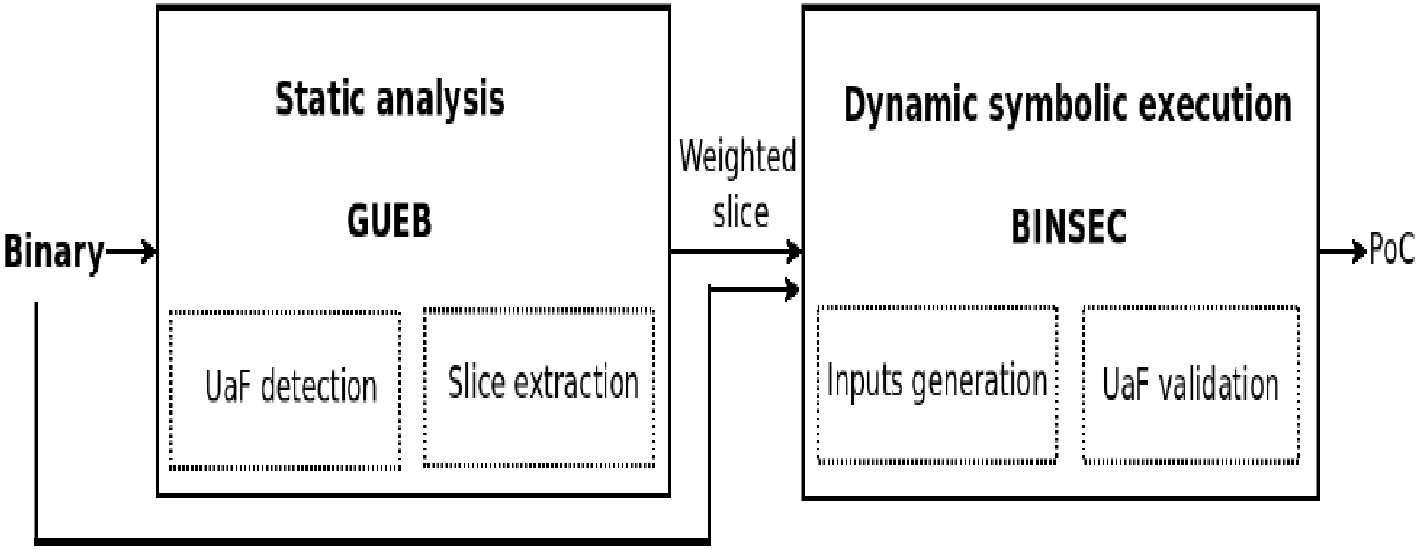


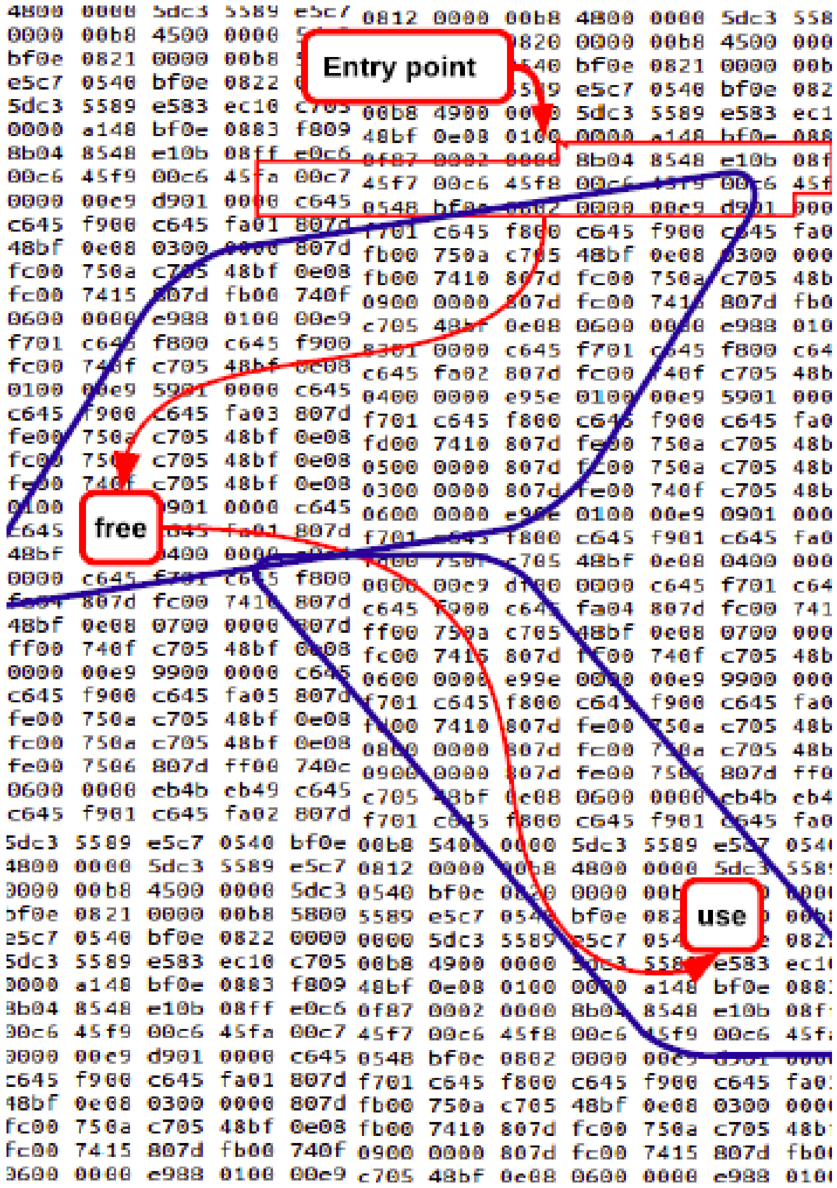
A Pragmatic 2-step approach

- **Step 1: incorrect but scalable**
- **Steps 1+2: scalable and correct**

Find a needle in the heap!

CASE 1: COMPLEX VULNERABILITY DETECTION [SSPREW'16](with Josselin Feist et al.)





A Pragmatic 2-step approach

- **Step 1: incorrect but scalable**
- **Steps 1+2: scalable and correct**

Find a needle in the heap !

- **Find a few new CVEs**
- **Much better than AFL here**

CASE 2: BINARY-LEVEL PROOF

Not addressed by DSE

- **Cannot enumerate all paths**

eg: $7y^2 - 1 \neq x^2$

(for any value of x, y in modular arithmetic)

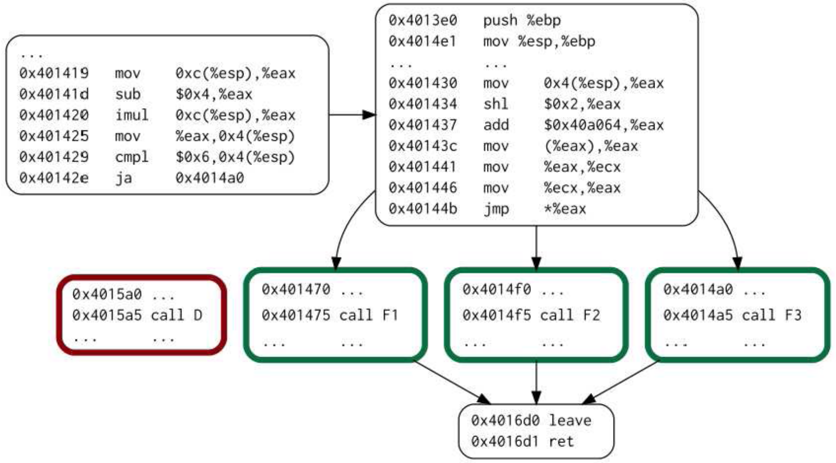
```
mov   eax, ds:X
mov   ecx, ds:Y
imul  ecx, ecx
imul  ecx, 7
sub   ecx, 1
imul  eax, eax
cmp   ecx, eax
jz    <dead_addr>
```

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:
```

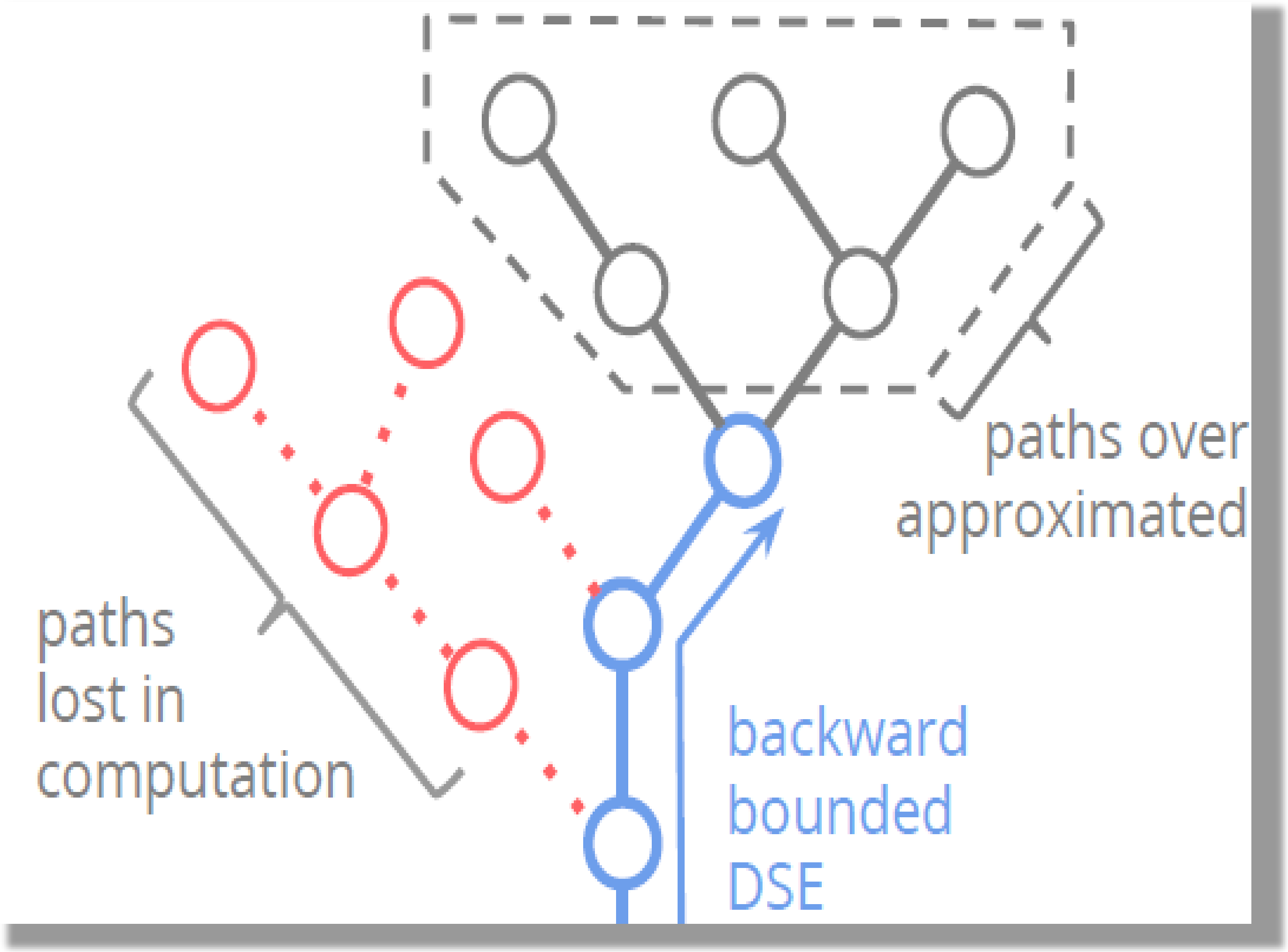
The two blocks are equivalent



With IDA + BINSEC

All jump targets are found

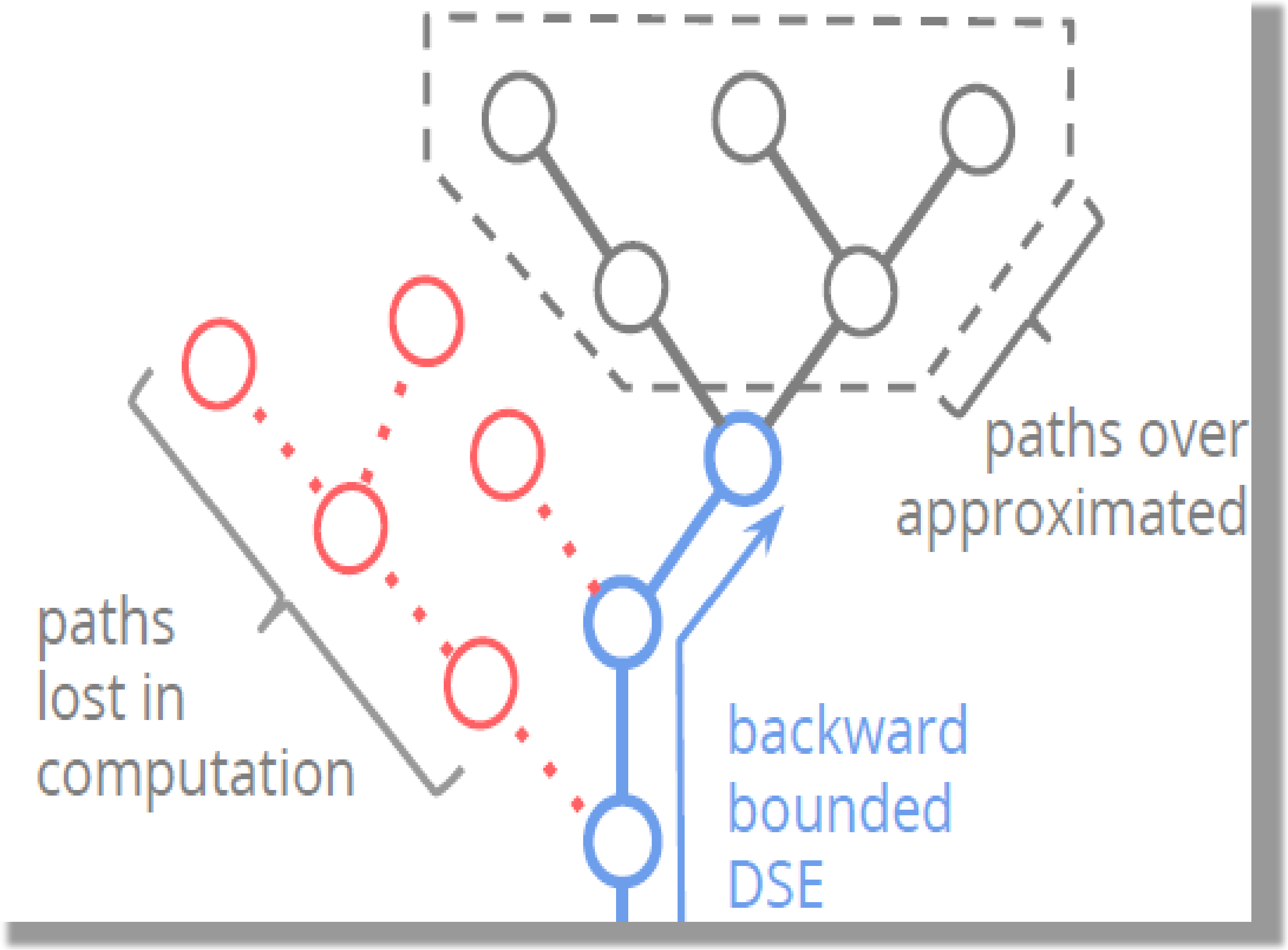
Case 2: BINARY-LEVEL PROOF



Backward bounded SE

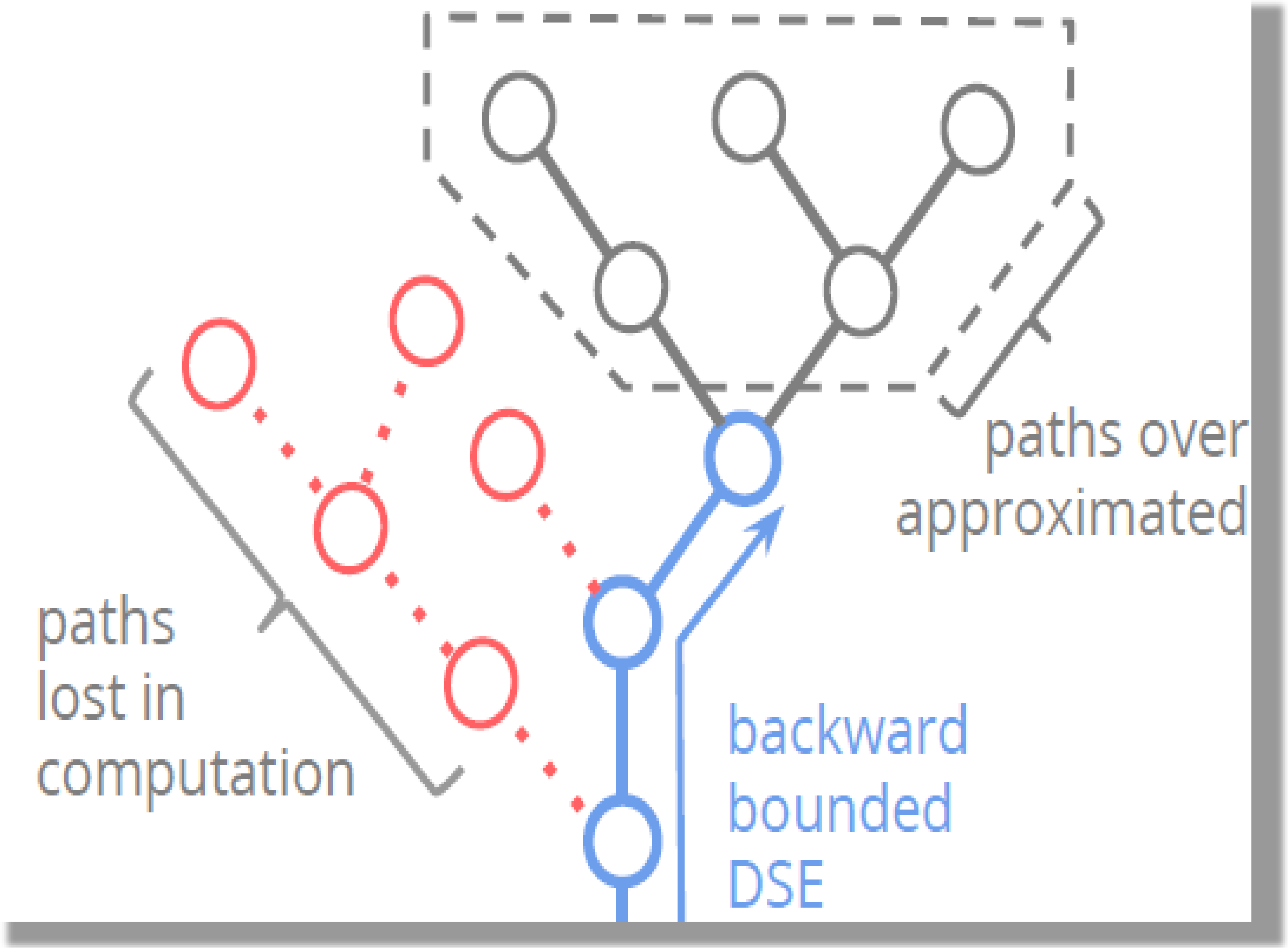
- **Compute k-predecessors**
- **If the set is empty, no pred.**
- **Allows to prove things**

Wait …



- **False Negative: k too small**
  - ***Missed proofs***
- **False Positive: CFG incomplete**
  - ***Wrong proofs ?!***

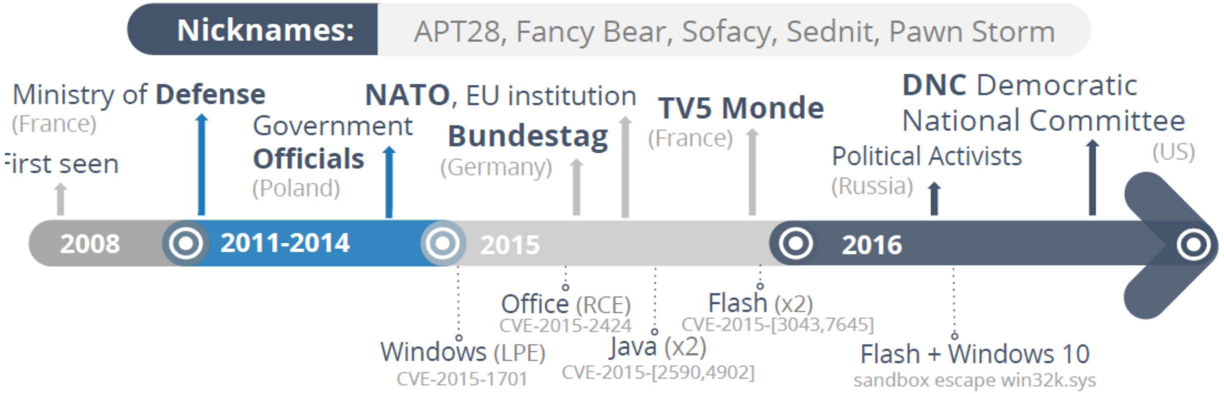
Wait …



- **False Negative: k too small**
  - ***Missed proofs***
- **False Positive: CFG incomplete**
  - ***Wrong proofs***

- Low rate of wrong proofs
- Controlled XPs

CASE-STUDY: THE XTUNNEL MALWARE [BH EU 16, S&P'17] -- part of DNC hack (with Robin David)



Two heavily obfuscated samples

- **Many opaque predicates**

Goal: detect & remove protections

- Identify 45% of code as spurious
- Fully automatic, < 3h

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

CASE 3: finely tuning the technology

- **SMT solvers are powerful weapons**

- **But (binary-level) security problems are terrific beasts**

- **Need to adapt them!**

Two examples
- Scalability [LPAR 2018, Benjamin Farinier]
- Robustness [CAV 2018, Benjamin Farinier]

Example 1: scalability (reasoning about memory)

Array theory

- **Necessary to model memory**

- Reading in a at index $i \in \mathcal{I}$: select a i
- Writing in a an element $e \in \mathcal{E}$ at index $i \in \mathcal{I}$: store a i e

$$\text{select} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E}$$
$$\text{store} : \text{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array}\,\mathcal{I}\,\mathcal{E}$$

ROW rule may introduce case-splits

$$\forall a\, i\, e.\ \text{select}\,(\text{store}\ a\ i\ e)\, i = e$$
$$\forall a\, i\, j\, e.\ (i \neq j) \Rightarrow \text{select}\,(\text{store}\ a\ i\ e)\, j = \text{select}\ a\ j$$

Hard for solvers

- **Case-splits**

Prevalent in software analysis

- Modelling memory
- Abstracting data structure (map, queue, stack...)

Hard to solve

- NP-complete
- ROW may require case-splits

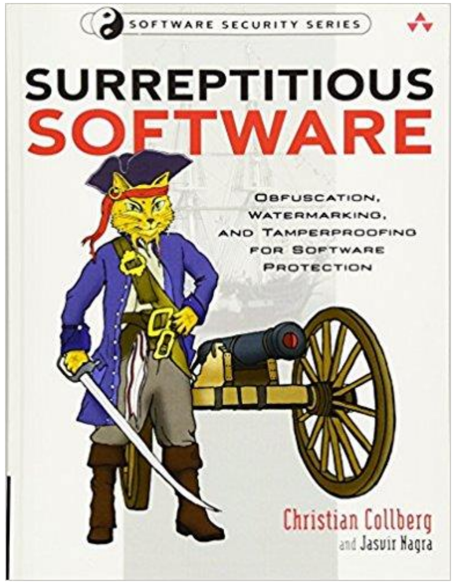
Not pure theory!

Reverse of a ASPACK-protected code

Huge formula obtained by dynamic symbolic execution

293 000 select

24 hours of resolution !



Example 1: scalability (reasoning about memory)

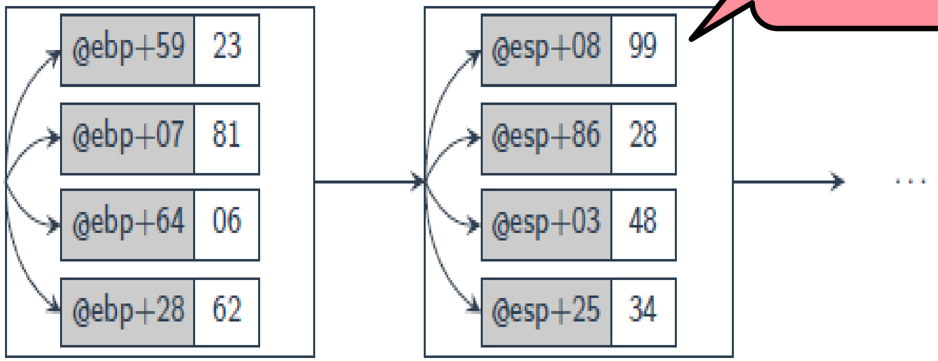
- **Our goal: dedicated formula preprocessing to remove « RoW »**

  - Problem 1: standard « list »-representation for logical arrays induces a quadratic time preprocessing ➔ prohibitive

  - Problem 2: need to cheaply but precisely reason about index equalities

Example 1: scalability (reasoning about memory)

- Dedicated data structure (list-map)
- Tuned for base+offset access
- Linear complexity



- Scale
- Good when only few bases

Propagate "variable+constant" terms

- If $y \triangleq z+1$ then $x \triangleq y+2 \rightsquigarrow x \triangleq z+3$
- Together with associativity, commutativity...

- Reduce the number of bases

- Prove disequalities between different bases

Associate to every indices i an abstract domain $i^\sharp$

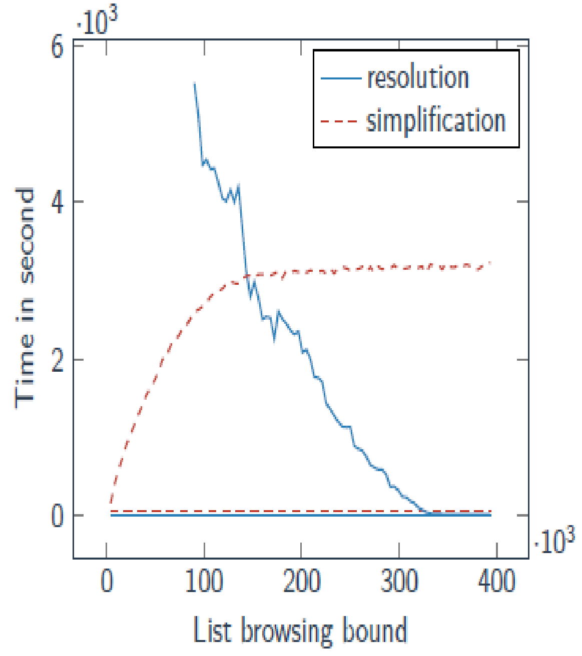
- If $i^\sharp \sqcap j^\sharp = \bot$ then $(a[i] \leftarrow e)[j] = a[j]$
- Integrated in the list-map representation

Tuning the solver: array formula simplification [LPAR 2018] with Benjamin Farinier

- **Makes the difference!**

no block cypher	Z3		#select all arrays	#select non initial
no simplification	0	606.7	1 448 301	1 448 001
list-16	0	501.0	1 075 358	1 052 786
list-256	0	371.9	807 778	762 673
map	0	370.5	807 778	762 673
LMBN	0	46.0	65 788	5 044

- Huge formula obtained by dynamic symbolic execution
- 293 000 select
- 24 hours of resolution !



Using LMBN

- #select reduced to 2 467
- 14 sec for resolution
- 61 sec for preprocessing

Using list representation

- Same result with a bound of 385 024 and beyond...
- ...but 53 min preprocessing

Example 2: robust symbolic execution

- **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)

```
int main () {
  int a = input ();
  int b = input ();

  int x = rand ();

  if (a * x + b > 0) {
    analyze_me();
  }
  else {
    ...
  }
}
```

Example 2: robust symbolic execution

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

- **Actually, need to solve** $\boxed{\forall x. ax + b > 0}$

- **Quantified formula**
- **SMT solvers bad for that**

```
int main () {
  int a = input ();
  int b = input ();

  int x = rand ();

  if (a * x + b > 0) {
    analyze_me();
  }
  else {
    ...
  }
}
```

Example: robustness and quantification [CAV 2018]

Our 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

  ❶ $\forall x. ax + b > 0$

- Taint variable constraint

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

- Independence condition

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

  ❹ $((\top \wedge \bot) \vee (\top \wedge a = 0) \vee (\bot \wedge x = 0)) \wedge \top$

  ❺ $a = 0$

- Quantifier-free reachability condition

  ❻ $(ax + b > 0) \wedge (a = 0)$

Example: robustness and quantification

	SE classic		
	SAT correct	SAT incorrect	UNSAT or UNKNOWN
Boolector	**12**	11	1
CVC4	7	9	8
Yices	7	11	6
Z3	**12**	12	0

	SE robust		
	SAT correct	SAT incorrect	UNSAT or UNKNOWN
Boolector	N/A	N/A	24
CVC4	5	**0**	19
Yices	N/A	N/A	24
Z3	7	**0**	17

	SE robust + elim.		
	SAT correct	SAT incorrect	UNSAT or UNKNOWN
Boolector	**12**	**0**	12
CVC4	7	**0**	17
Yices	7	**0**	17
Z3	**12**	**0**	12

- set of crackme challenge
- compare true and false positive

Example: robustness and quantification

Back to 28: GRUB2 Authentication Bypass

- Original version: press Backspace 28 times to get a rescue shell
- Case study: same vulnerable code turned into a `crackme` challenge

- SE classic:
  incorrect solution
- SE robust:
  solvers TIMEOUT

- SE robust + elim.:
  correct solution in 80s
- SE robust + elim. + simpl.:
  correct solution in 30s

OUTLINE

- The success of formal methods for safety
- Why binary-level security analysis?
- The hard journey from source-level safety to binary-level security
- Our approach
- Conclusion

WHERE ARE WE?

- Explore
- Prove
- Simplify

Find a needle in the heap !

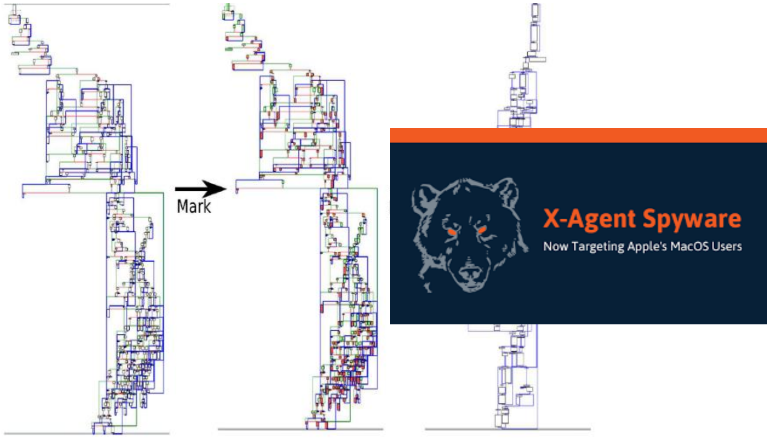
packers	trace len.	#proc	#th	#SMC	opaque predicates OK	opaque predicates OP	call stack tampering OK	call stack tampering tamper
ACProtect v2.0	1.8M	1	[illegible]	[illegible]	[illegible]	159	0	48
ASPack v2.12	377K	1	[illegible]	[illegible]	[illegible]	24	11	6
Crypter v1.12	1.1M	1	[illegible]	[illegible]	[illegible]	24	125	78
Expressor	635K	1	1	[illegible]	[illegible]	[illegible]	14	0
FSG v2.0	68k	1	1	[illegible]	[illegible]	[illegible]	6	0
Mew	59K	1	1	1	28	1	6	1
PE Lock	2.3M	1	1	6	95	90	4	3
RLPack	941K	1	1	[illegible]	[illegible]	[illegible]	14	0
TELock v0.51	406K	1	1	[illegible]	[illegible]	[illegible]	3	1
Upack v0.39	711K	1	1	[illegible]	[illegible]	[illegible]	7	1

The technique scale on significant traces

Many true positives. Some packers are using it intensively

Packers using ret to perform the final tail transition to the entrypoint

Find (new) CVEs



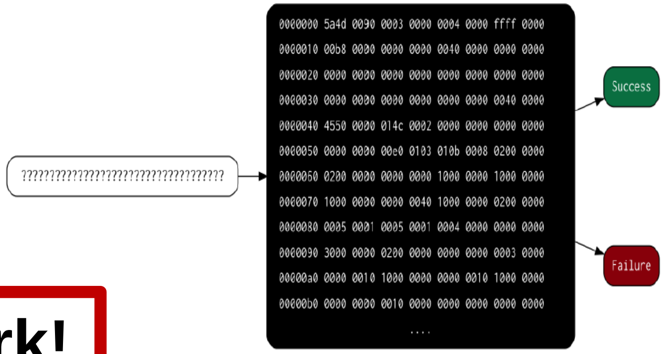
Revert protections

GRUB2 CVE 2015-8370

Elevation of privilege

Information disclosure

Denial of service



- **Semantic approaches can work!**

KEY PRINCIPLES

- **Robustness & precision are essential**
  - DSE is a good starting point
  - dedicated robust and precise (but not sound) static analysis are feasible

- **Can be adapted beyond the basic reachability case**
  - variants
  - combination with other techniques

- **Loss of guarantees**
  - Accept … But control!
  - Look for « correct enough » solutions

- **Finely tune the technology**
  - Tools for safety are not fully adequate for security

SECURITY IS NOT SAFETY

- strong incentive to human assistance
- spec., parameter tuning, etc.

- **Source-level SAFETY**
  - Model: High-level language
  - Properties: safety
  - Algorithm: full correctness, possible help from user

- no human assistance
- low tolerance to false positive

- **Binary-level SECURITY**
  - Model: binary-level code, possibly adversarial, + attacker
  - Properties: safety, k-safety, « bugs vs vulnerabilities »
  - Algorithm: robust & precise enough, fully automated

SECURITY IS NOT SAFETY

- strong incentive to human assistance
- spec., parameter tuning, etc.

- **Source-level SAFETY**
  - Model: High-level language
  - Properties: safety
  - Algorithm: full correctness, possible help from user

- no human assistance
- low tolerance to false positive

- **Binary-level SECURITY**
  - Model: **binary-level code**, **possibly adversarial**, + attacker
  - Properties: safety, k-safety, « bugs vs vulnerabilities » **[robust solutions]**
  - Algorithm: **robust & precise enough**, fully automated

CONCLUSION & TAKE AWAY

- **Binary-level security analysis**
  - Many applications, many challenges
  - Current syntactic and dynamic methods are not enough

- **Formal methods can help … but must be strongly adapted**
  - [Complement existing approaches]

  - Need **robustness** and scalability!
  - Acceptable to lose both correctness & completeness – in a **controlled** way
  - Much better if **specifically tuned** for the problem at hand

  - **New challenges and variations, many things to do!**

- **Thanks for your attention!**

Commissariat à l’énergie atomique et aux énergies alternatives
Institut List | CEA SACLAY NANO-INNOV | BAT. 861 – PC142
91191 Gif-sur-Yvette Cedex - FRANCE
www-list.cea.fr

Établissement public à caractère industriel et commercial | RCS Paris B 775 685 019