Revisiting Program Analysis through the Security Lens

Journées Nationales du GDR GPL / AFADL 2023

FROM RESEARCH TO INDUSTRY

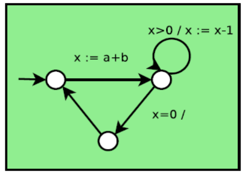
Sébastien Bardin

Senior Researcher, CEA Fellow

LSL/SABR

The BINSEC Group:
ADAPT FORMAL METHODS TO BINARY-LEVEL SECURITY ANALYSIS

Model



Source code

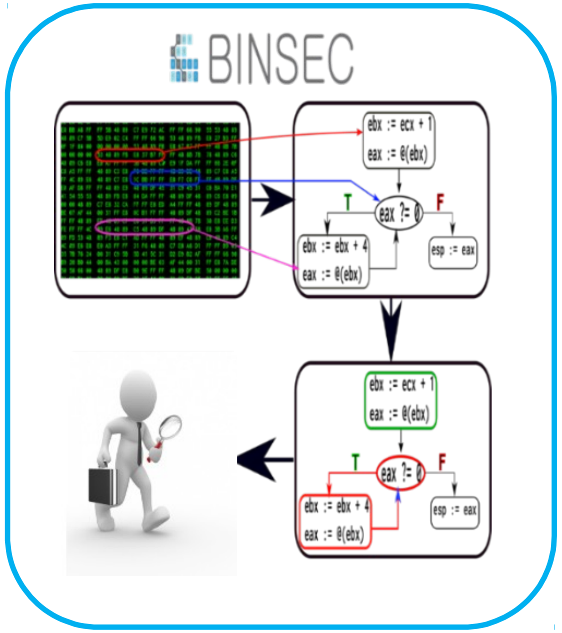
```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

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
```



https://binsec.github.io/

- Looking for postdoc & PhD candidates

WHY THIS TALK ?

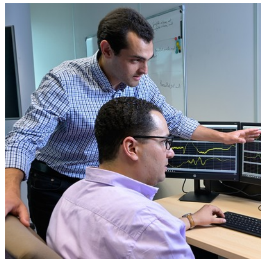
- **Program Analysis (PL) and Formal Methods come from critical safety needs**
  - Damn good there (in the hands of experts)

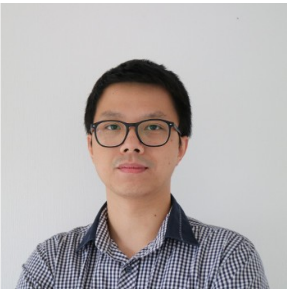
- **Now : a move from safety concerns to security concerns**

- ***Questions*:**
  - ***how can we use standard PL/FM into a security context ?***
  - ***how does code-level security differ from code-level safety?***
  - ***how does security differ from safety ? [focus on the attacker]***

- ***This talk: share some insights from our biased experience*** *[CAV 21, ESOP 2023]*

TEAM WORK SINCE 2012





Prologue : ABOUT FORMAL METHODS AND CODE ANALYSIS

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

- Reason about the meaning of programs

Key concepts : $M \models \varphi$

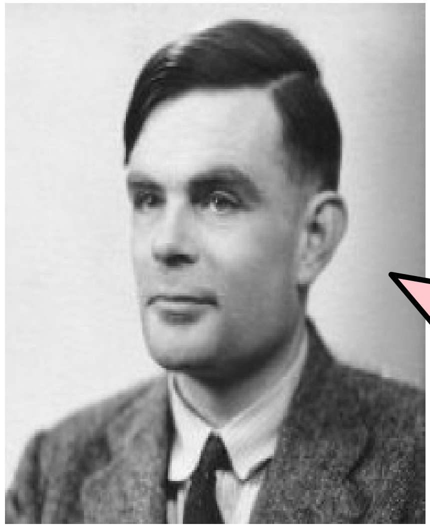
- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

- Reason about infinite sets of behaviours

- Typical ingredients: transition systems, automata, logic, …

Success in (regulated) safety-critical domains

They knew it was impossible, so they did it anyway



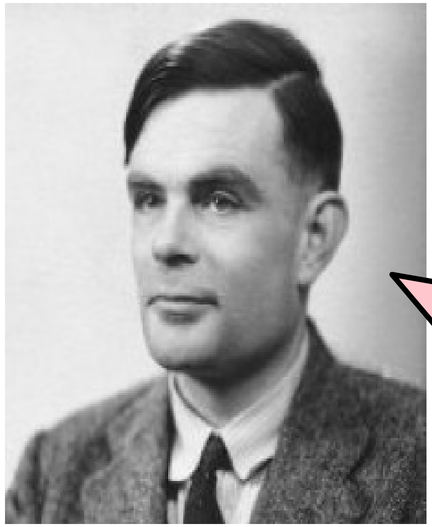
Cannot have analysis that
- Terminates
- Is perfectly precise

On all programs

- **Weakest precondition calculi** [1969, Hoare]
- **Abstract Interpretation** [1977, Cousot & Cousot]
- **Model checking** [1981, Clarke - Sifakis]

Answers
- Forget perfect precision: bugs xor proofs
- Or focus only on « interesting » programs
- Or put a human in the loop
- Or forget termination

They knew it was impossible, so they did it anyway



Cannot have analysis that
- Terminates
- Is perfectly precise

On all programs

- **Weakest precondition calculi** [1969, Hoare]
- **Abstract Interpretation** [1977, Cousot & Cousot]
- **Model checking** [1981, Clarke - Sifakis]

Answers
- **Forget perfect precision: bugs** xor proofs
- Or focus only on « interesting » programs
- Or put a human in the loop
- Or forget termination

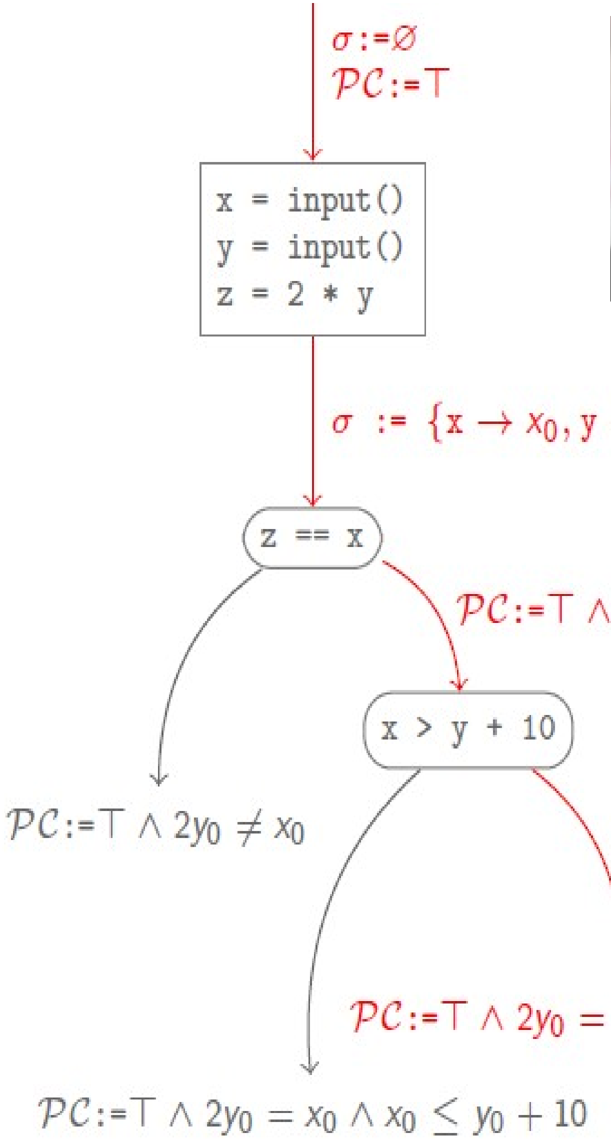
SYMBOLIC EXECUTION (Godefroid 2005)

Find real bugs

Bounded verification

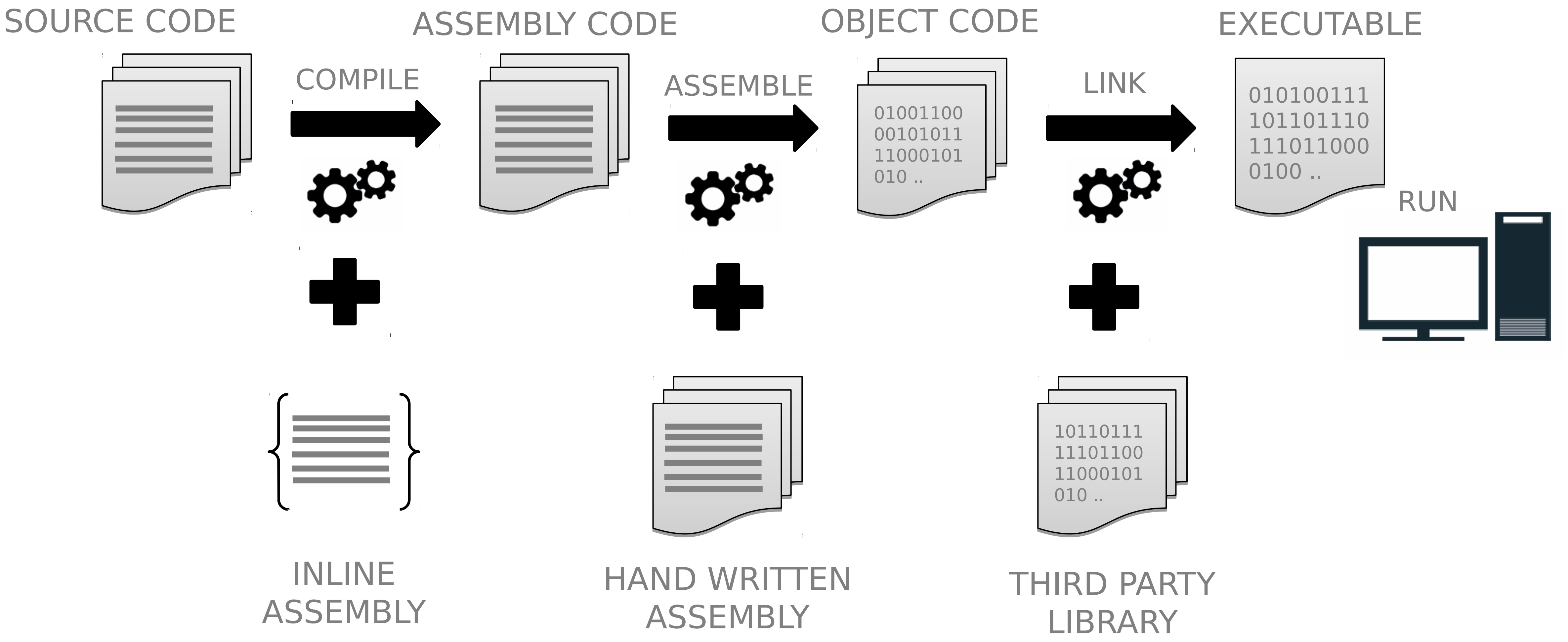
Flexible

```
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

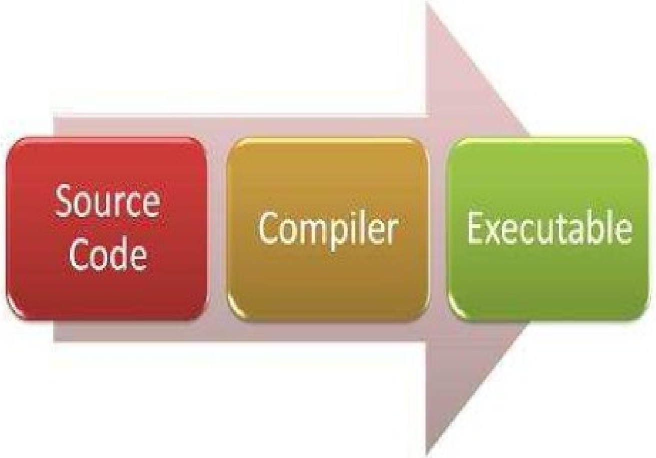
- Compute its « path predicate » f
- Solution of f = input following the path
- Solve it with powerful existing solvers


SOURCE CODE
ASSEMBLY CODE
OBJECT CODE
EXECUTABLE
COMPILE
ASSEMBLE
LINK
01001100
00101011
11000101
010 ..
010100111
101101110
111011000
0100 ..
RUN
10110111
11101100
11000101
010 ..
INLINE ASSEMBLY
HAND WRITTEN ASSEMBLY
THIRD PARTY LIBRARY

WHY GOING DOWN TO BINARY-LEVEL SECURITY ANALYSIS?

No source code

Post-compilation

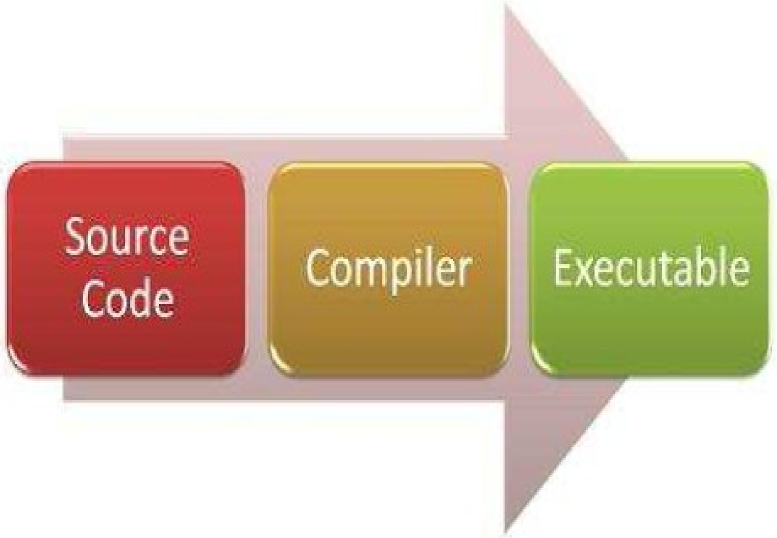


Malware comprehension

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

- **Introduction**

- **Challenges of automated binary-level security analysis**
- **BINSEC & Symbolic Execution for Binary-level Security**

- **Robust reachability and bugs that matter**
- **Adversarial reachability**

- **Conclusion, Take away and Disgression**

OUTLINE

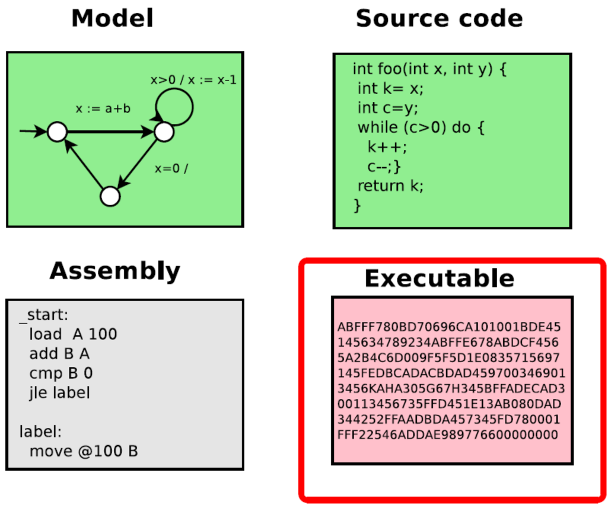
- Introduction

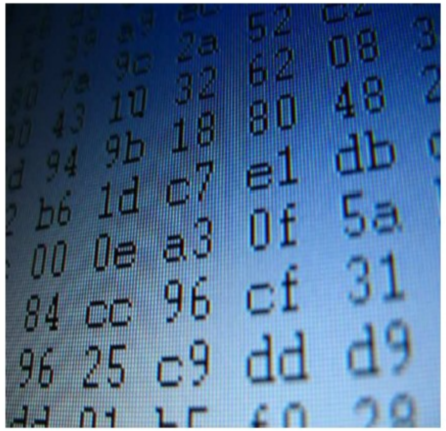
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security

- Robust reachability and bugs that matter
- Adversarial reachability

- Conclusion, Take away and Disgression

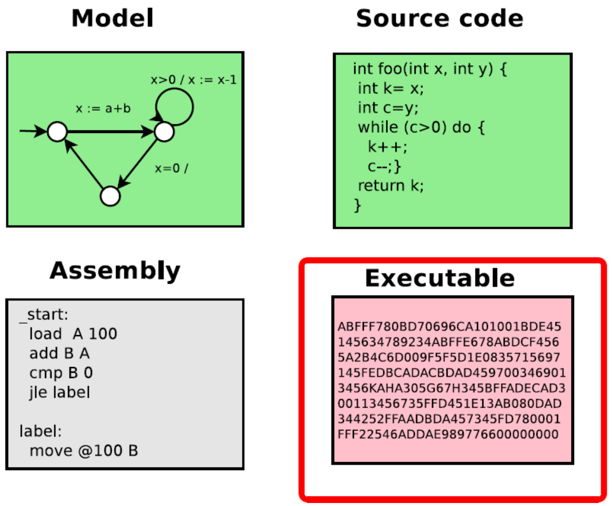
New challenges!

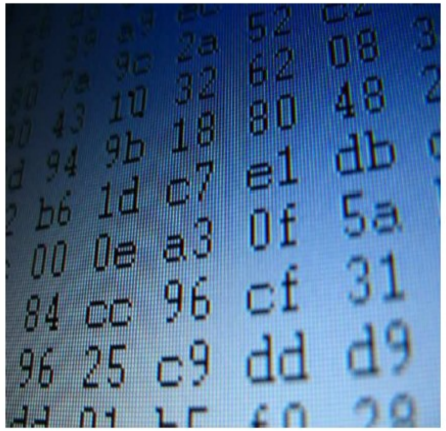




- Binary code
- Attacker
- Properties

New challenges!





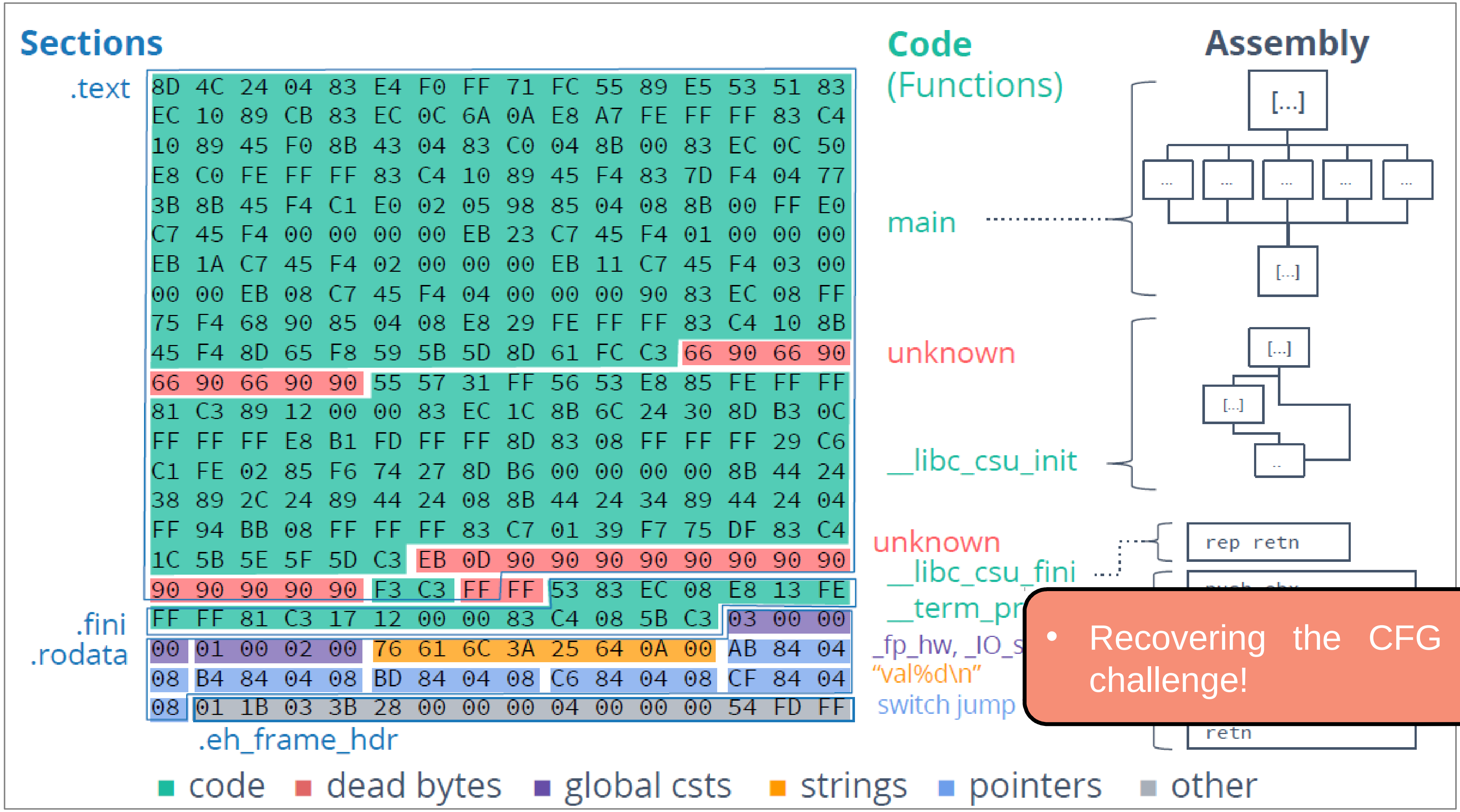
- Binary code
- Attacker
- Properties

CHALLENGE: BINARY CODE LACKS STRUCTURE

- Instructions?
- Control flow?
- Memory structure?

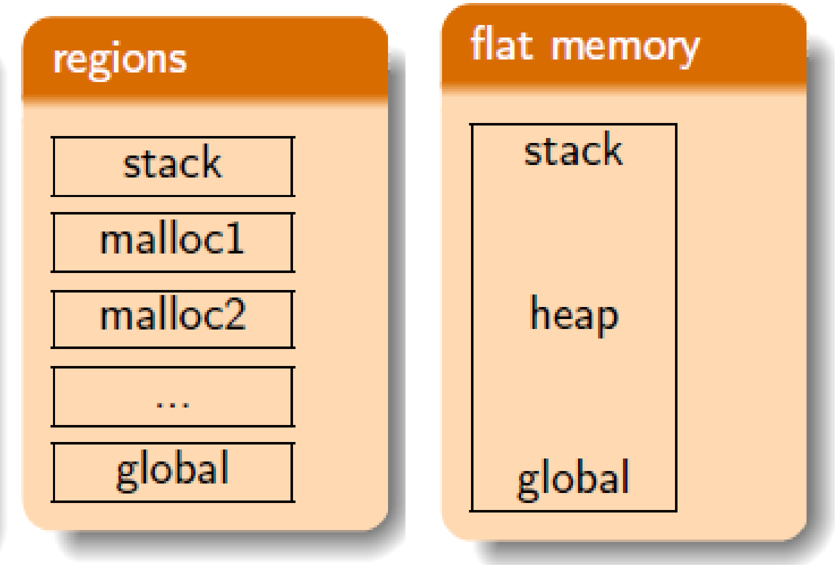
DISASSEMBLY IS ALREADY TRICKY!

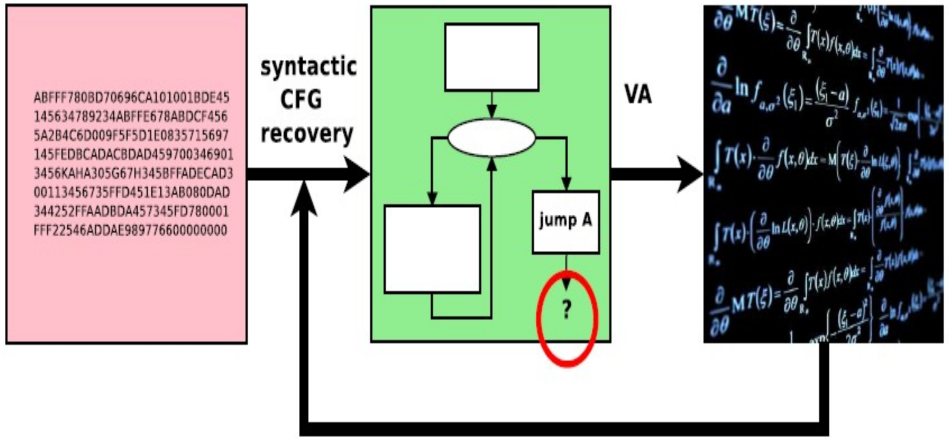
- code – data ??
- dynamic jumps (jmp eax)



- Recovering the CFG is already a challenge!

BINARY CODE SEMANTIC LACKS STRUCTURE





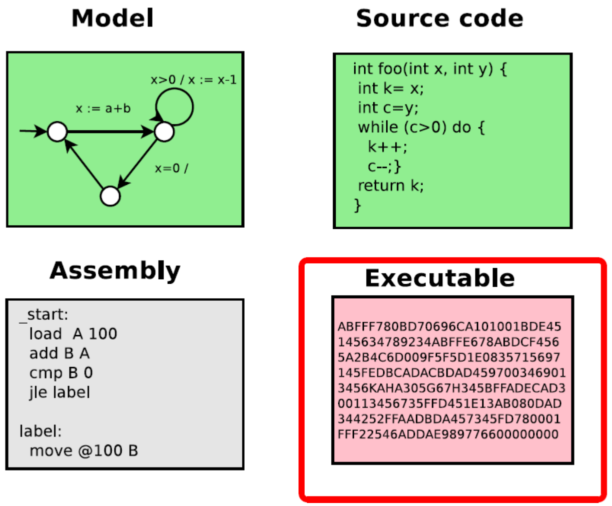
Problems

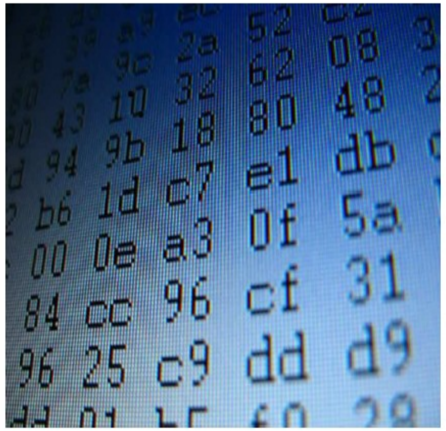
- Jump eax
- Untyped memory
- Bit-level resoning

```
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:
```

New challenges!



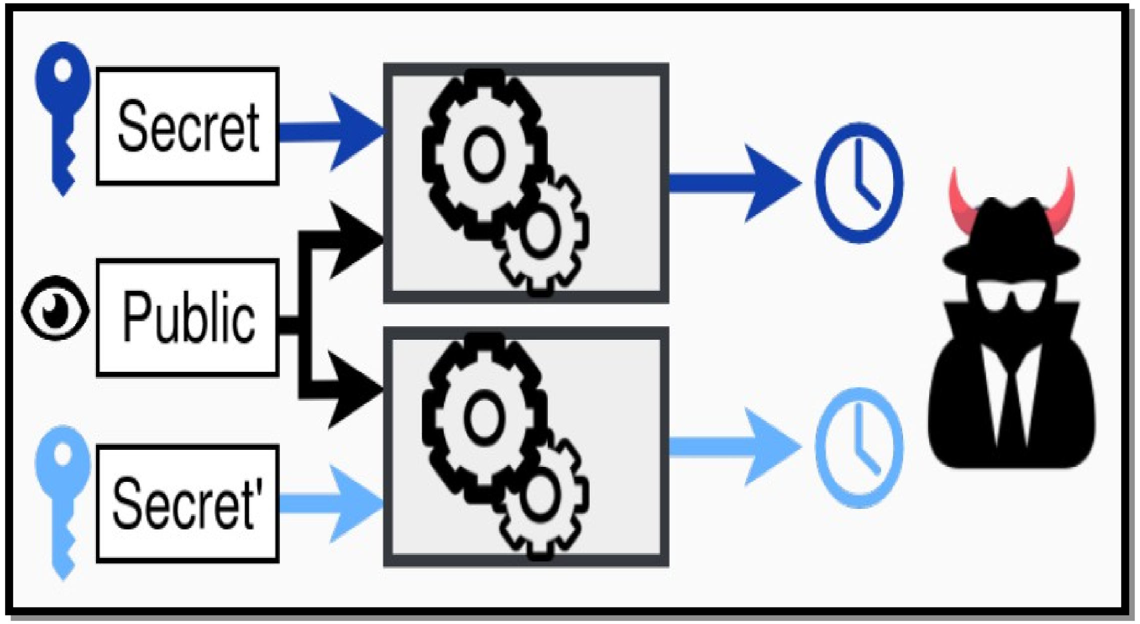


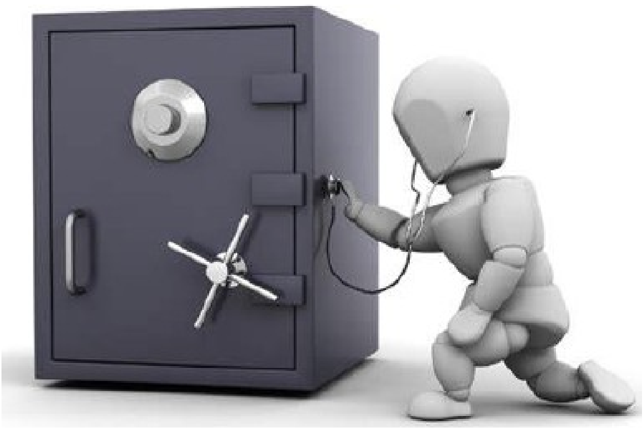
- Binary code
- Attacker
- Properties

New challenge : safety is not hyper-property :-)

Information leakage

Properties over pairs of executions

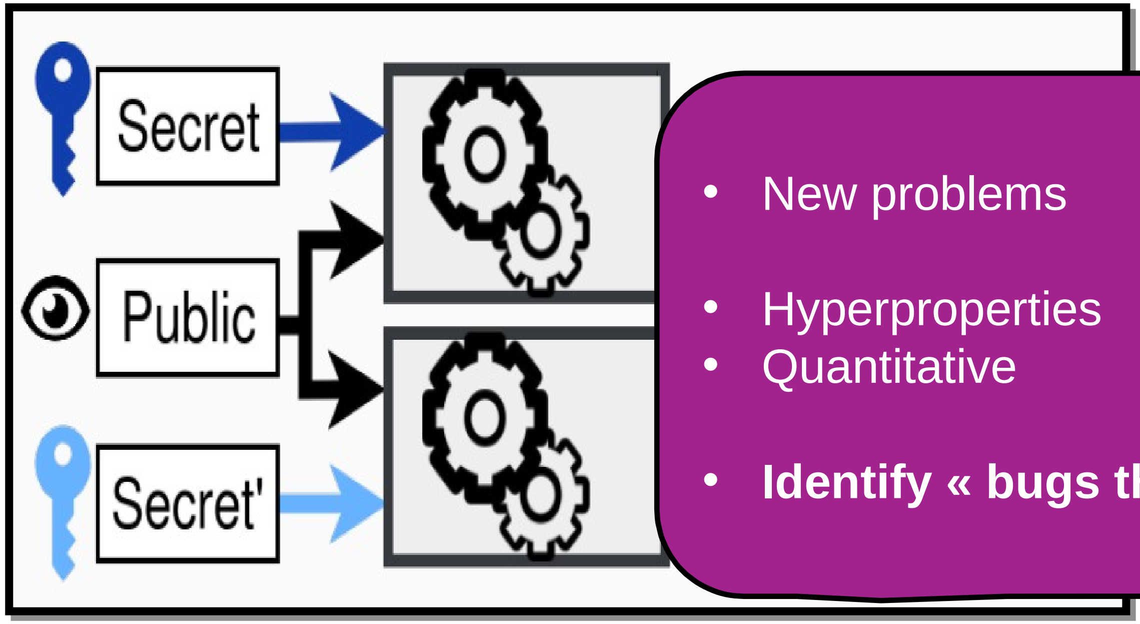




New challenge : safety is not hyper-property :-)

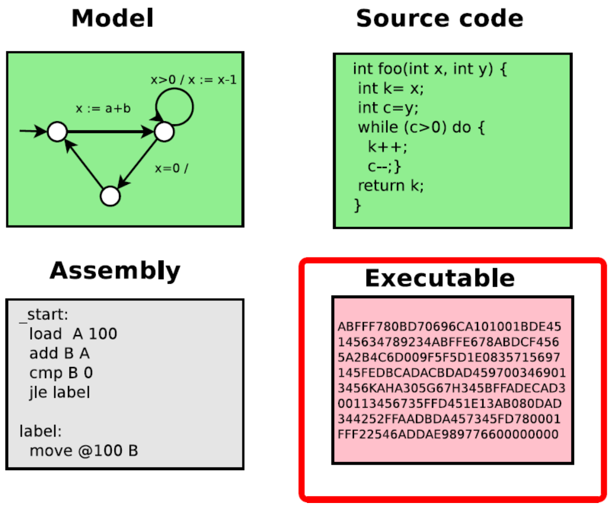
Information leakage

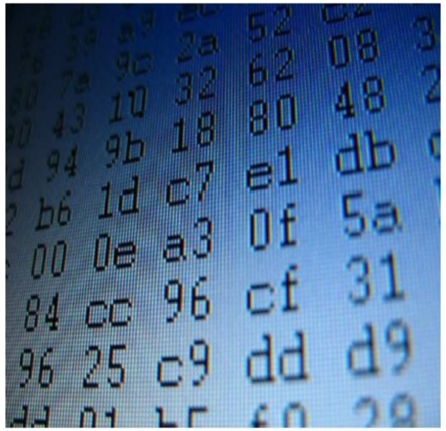
Properties over pairs of executions



- New problems
- Hyperproperties
- Quantitative
- **Identify « bugs that matters »**

New challenges!





- Binary code

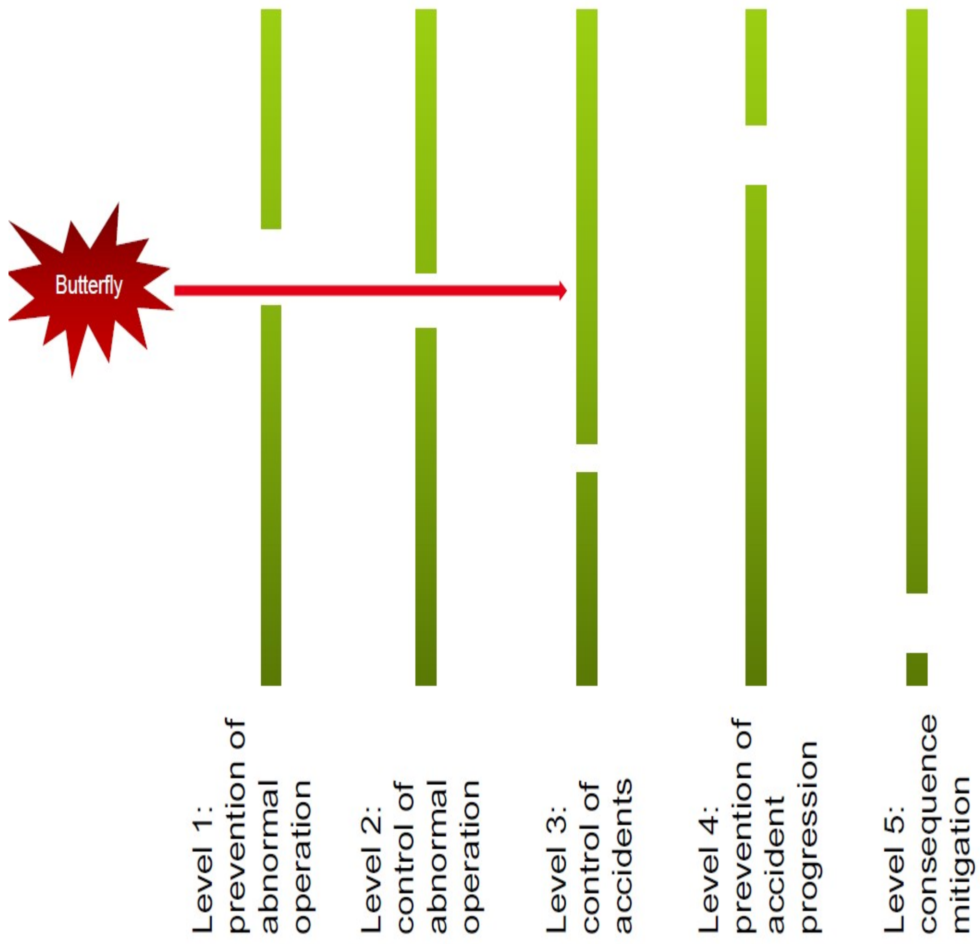
- Attacker

- Properties

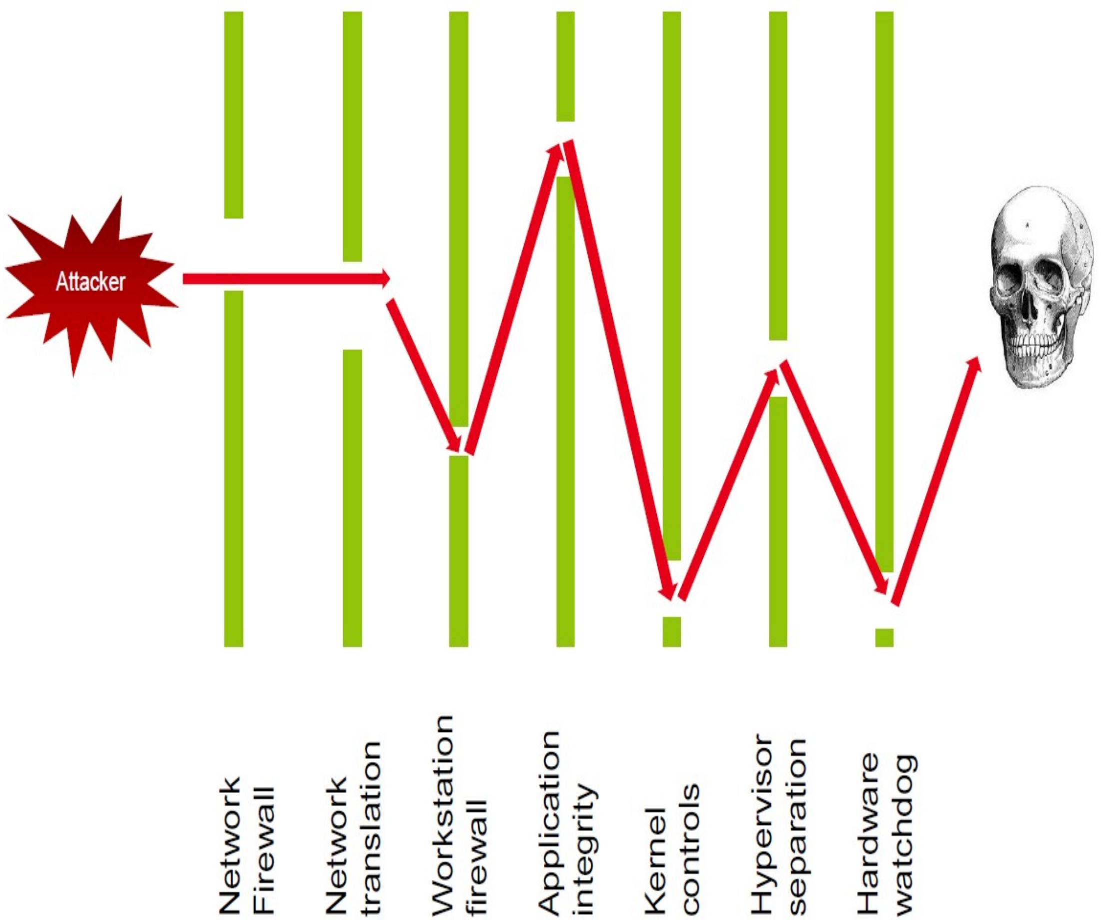
Main topic of the day

CHALLENGE: ATTACKER

Nature is not nice



Attacker is evil



ATTACKER in Standard Program Analysis

- **We are reasoning worst case: seems very powerful!**

ATTACKER in Standard Program Analysis

- **We are reasoning worst case: seems very powerful!**

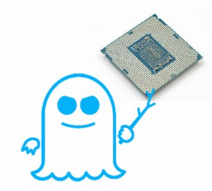
- **Still, our current attacker plays the rules: respects the program interface**
  - Can craft **very smart input**, but only through **expected input sources**

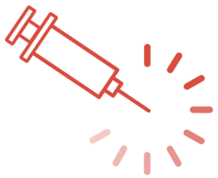
ATTACKER in Standard Program Ana

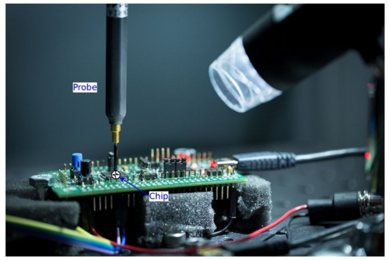
- **We are reasoning worst case: seems very**

- **Still, our attacker plays the rules: respects**
  - Can craft very smart input, but only through expecte

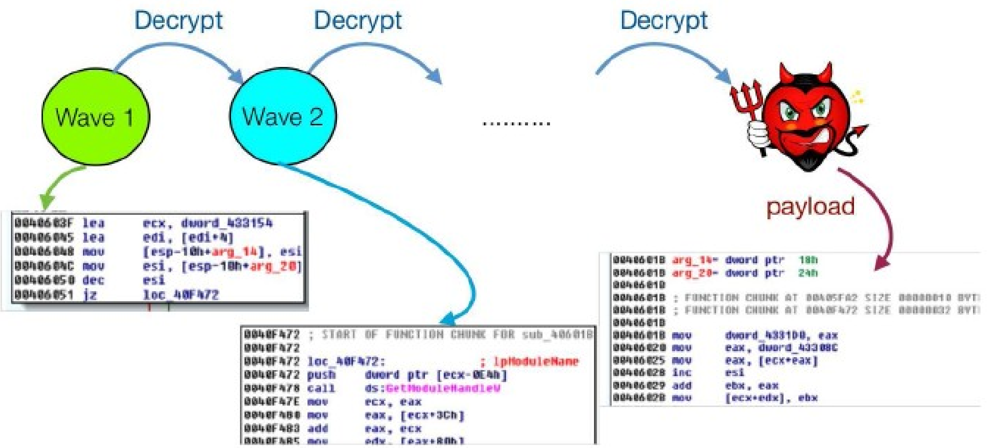
- **What about someone who really do not play the rules?**
  - Side channel attacks
  - Micro-architectural attacks
  - Fault injections







Another Line of attack : ADVERSARIAL BINARY CODE



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

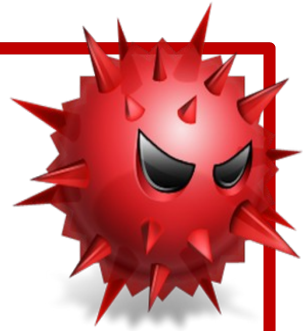
eg: $7y^2 - 1 \neq x^2$

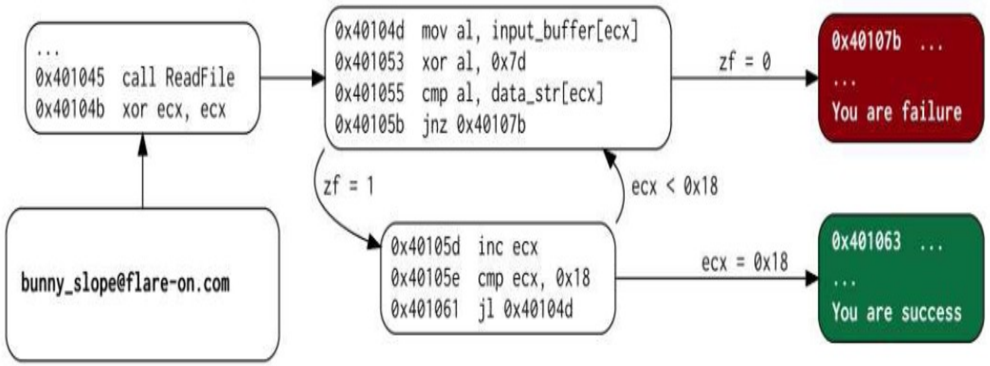
(for any value of x, y in modular arithmetic)

↓

```
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
- ...





OUTLINE

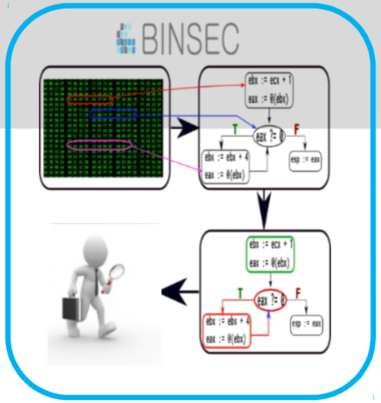
- Introduction

- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security

- Robust reachability and bugs that matter
- Adversarial reachability

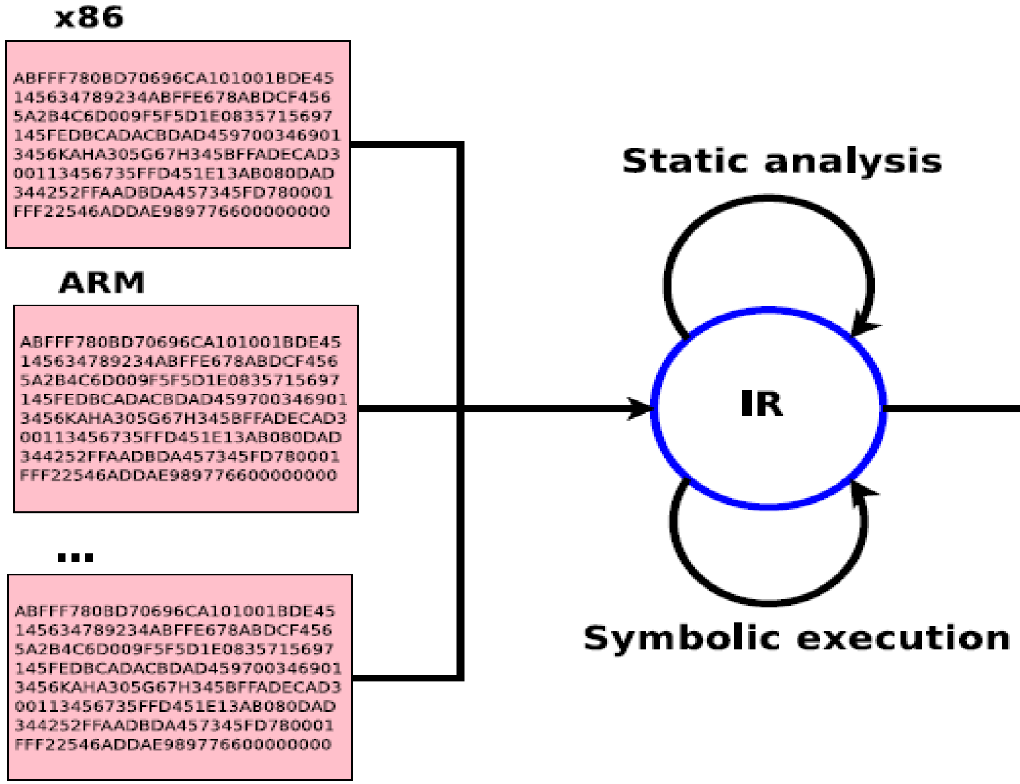
- Conclusion, Take away and Disgression

BINSEC: brings formal methods to binary-level security analysis



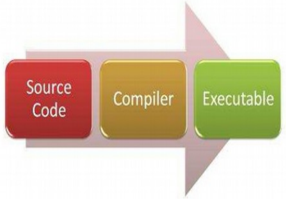
Break | Prove | Protect

- Explore many input at once
  - Find bugs
  - Prove security
- Multi-architecture support
  - x86, ARM, RISC-V
  - 32bit, 64bit

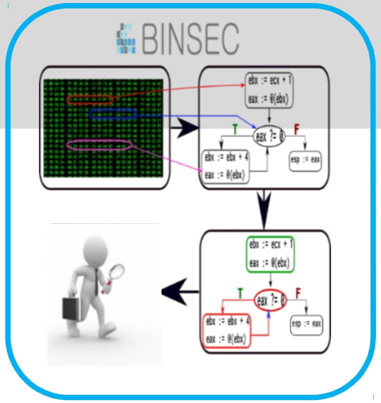


- Advanced reverse
- Vulnerability analysis
- Binary-level security proofs
- Low-level mixt code (C + asm)
- …

https://binsec.github.io/

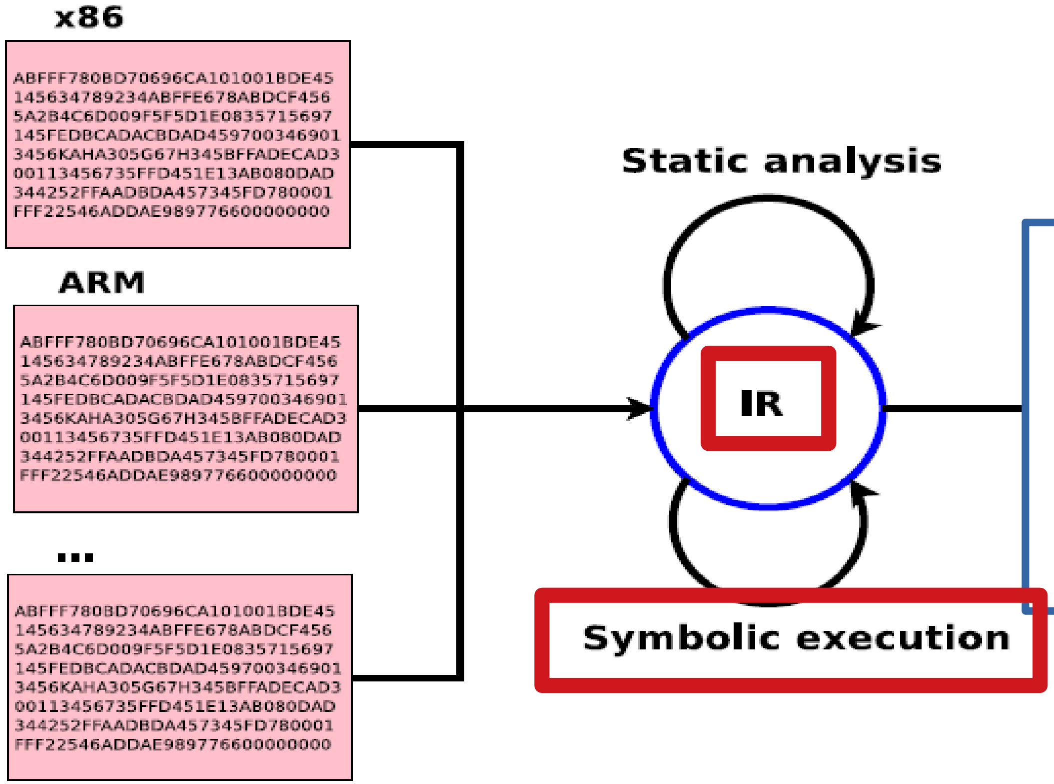


BINSEC: brings formal methods to binary-level security analysis



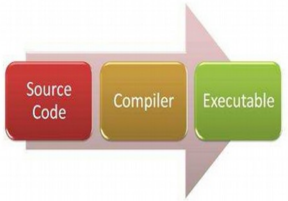
Break | Prove | Protect

- **Explore many input at once**
  - **Find bugs**
  - **Prove security**
- **Multi-architecture support**
  - x86, ARM, RISC-V
  - 32bit, 64bit



- **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

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

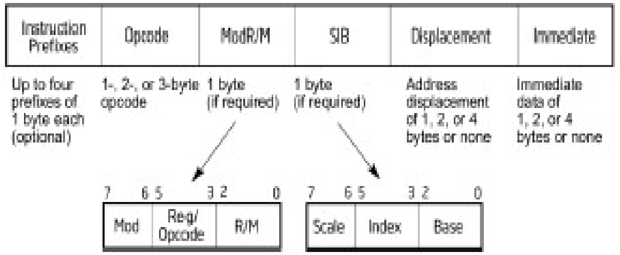
Multi-architecture

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 $\overset{x86reference}{\Rightarrow}$ 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)
```

Key 2: SYMBOLIC EXECUTION (Godefroid 2005)

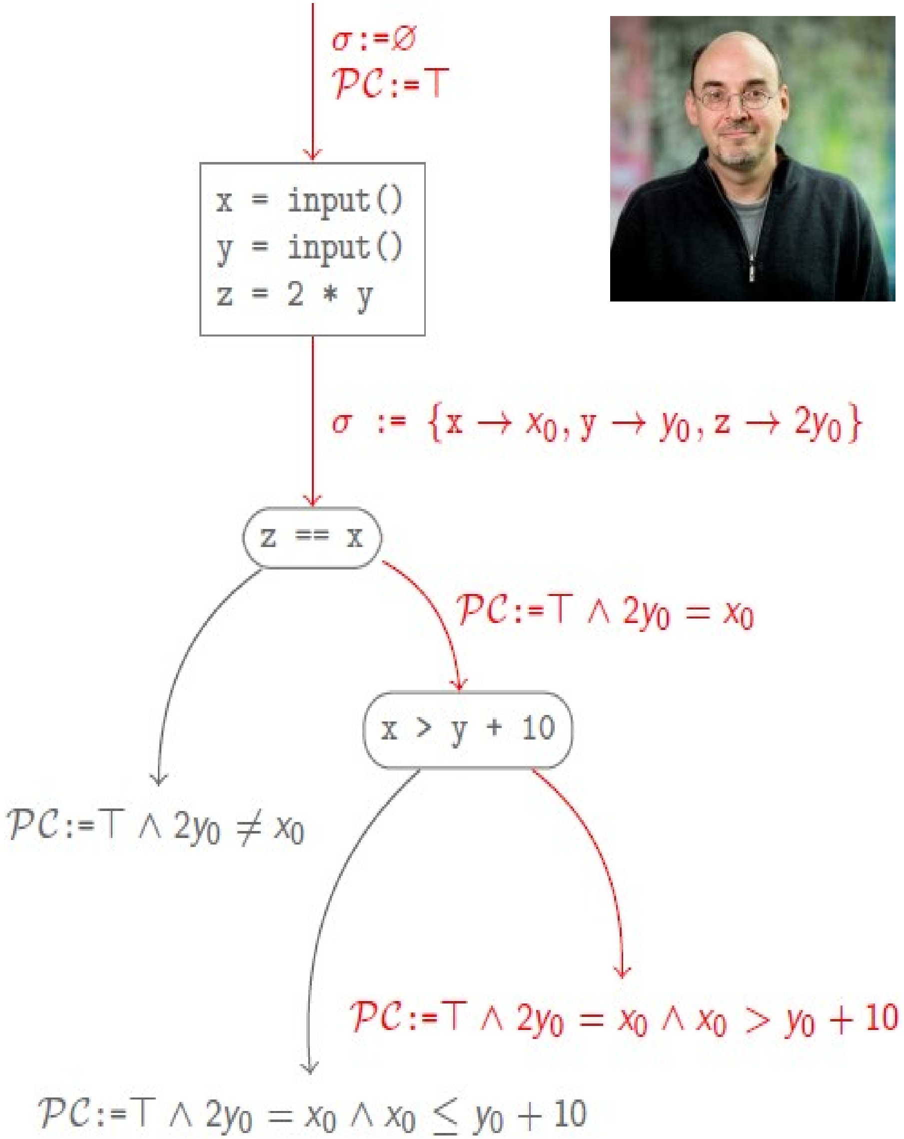
Find real bugs

Bounded verification

Flexible


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

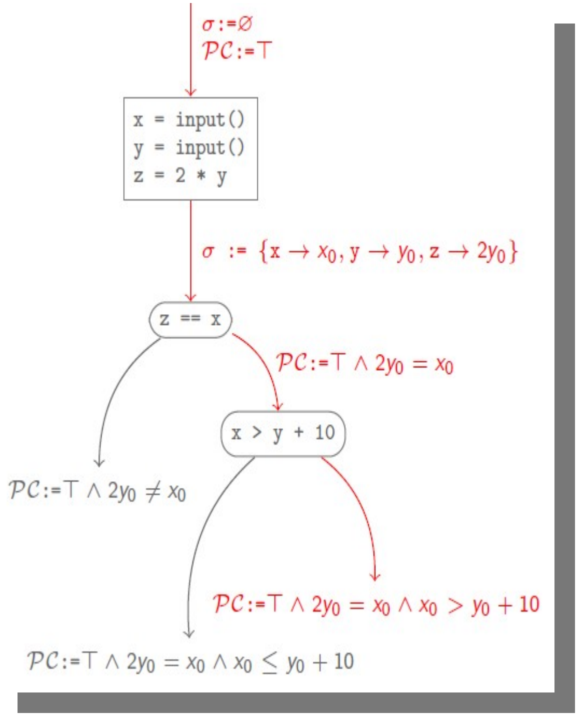
Given a path of a program
- Compute its « path predicate » f
- Solution of f = input following the path
- Solve it with powerful existing solvers



PATH PREDICATE COMPUTATION & SOLVING

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

$$\text{let } W_1 \triangleq Y_0 + 1 \text{ in}$$
$$\text{let } X_2 \triangleq W_1 + 3 \text{ in}$$
$$X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$



Boolector

Blackbox solvers

SMT Solver

my input!!

Y0 = 0 /\ Z0=3

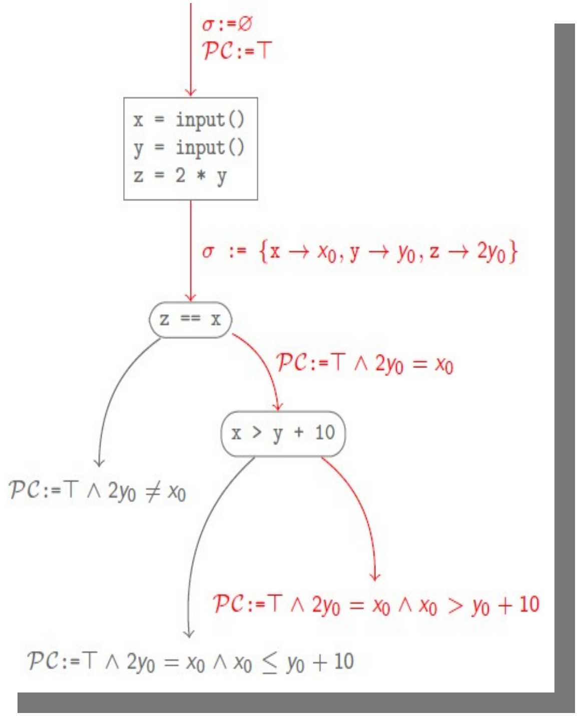
PATH PREDICATE COMPUTATION & SOLVING

Key ingredients
- Path search
- Constraint solving

Beware
- Path explosion
- Constraint solving cost

Loc	Instructi…
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (…
4	if (x < z) (bran…

Many optimizations
- Preprocessing, caching, etc.
- Search heuristics, path pruning, merge, etc.
- Concretization



let $W_1 \triangleq Y_0 + 1$ in
let $X_2 \triangleq W_1 + 3$ in
$X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$

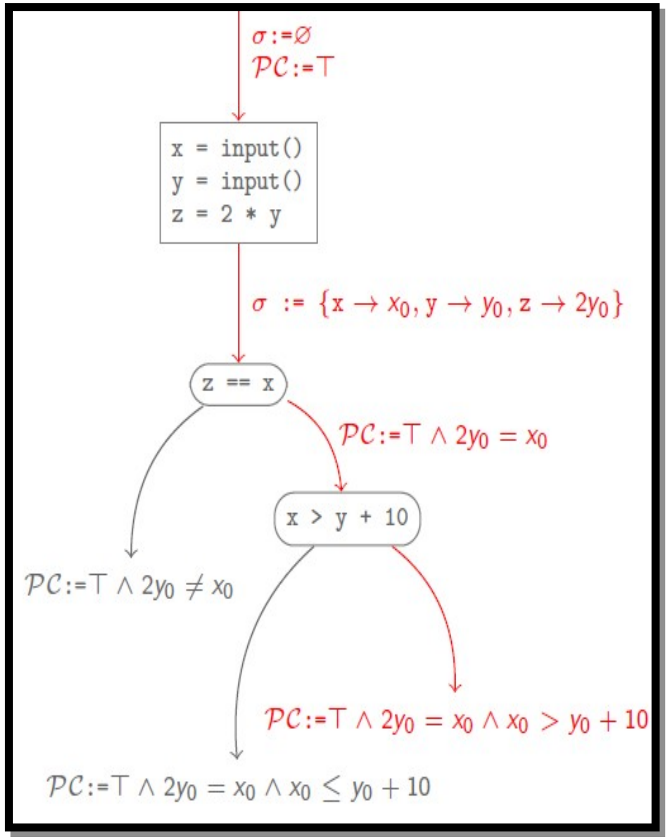
Blackbox solvers

SMT Solver

my input!!

Y0 = 0 /\ Z0=3

Typical application : Vulnerability finding & automated testing



- Intensive path exploration

- Target critical bugs
- or high coverage

- From scratch
- or enhanced prior test suite

Symbolic execution – fuzzing – static analysis

OUTLINE

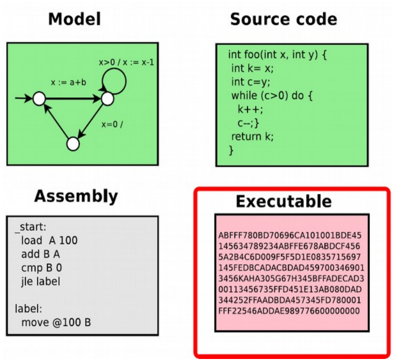
- Introduction

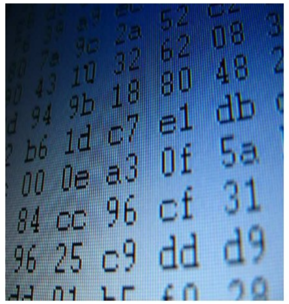
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security

- Robust reachability and bugs that matter
- Adversarial reachability

- Conclusion, Take away and Disgression

- **Problem : not all bugs are equal**





- Binary code
- Attacker
- Properties

The problem of « false positive in practice »

- **Reachability-based reasoning may produce** **false positive in practice**

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

  int x = rand ();

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

The problem of « false positive in practice »

- **Reachability-based reasoning may produce** **false positive in practice**

What?!!

Safety is not security …

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

  int x = rand ();

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

The problem of « false positive in practice »

- **Reachability-based reasoning may produce**
  false positive in practice

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 {
    ...
  }
}
```

The problem of « false positive in practice »

- **Reachability-based reasoning may produce** **false positive in practice**

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

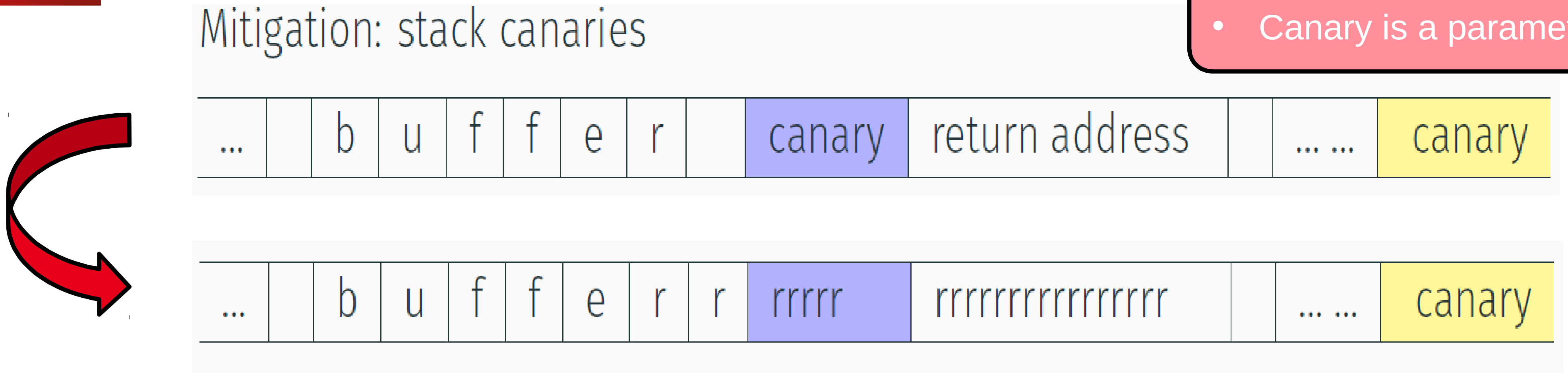
- **Problem: x is not controlled by the user**
  - If x change, possibly not a solution anymore
  - Example: (a = -100, b = 10, x = 1)

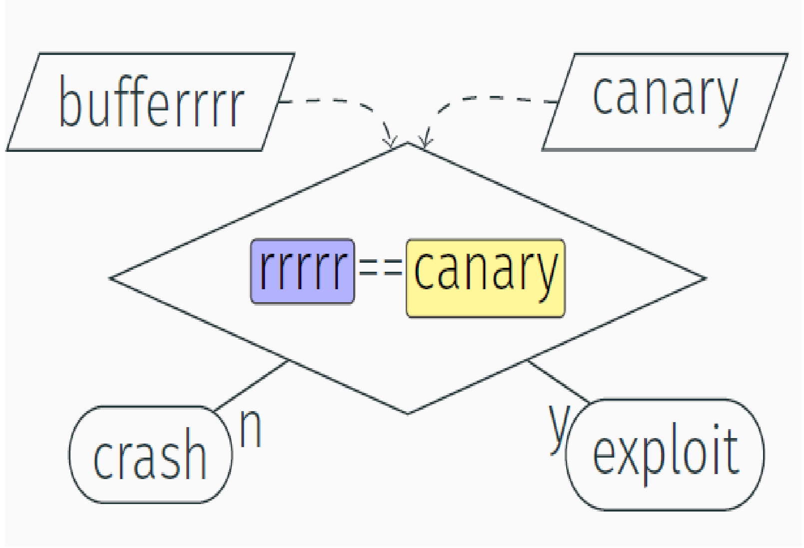
In practice: canaries, secret key in uninitialized memory, etc.

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

  int x = rand ();

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

Problems with standard reachability?

- Value in blue is checked against canary
- Canary is a parameter





- **In practice, only 2^-32 to bypass canary**
- **Not considered an attack**

Still, Symbolic Execution reports a bug

- **just need canary ==rrrr**
- **False positive**

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**

Real life false positives

Formally reachable, but
in reality, cannot be triggered **reliably**

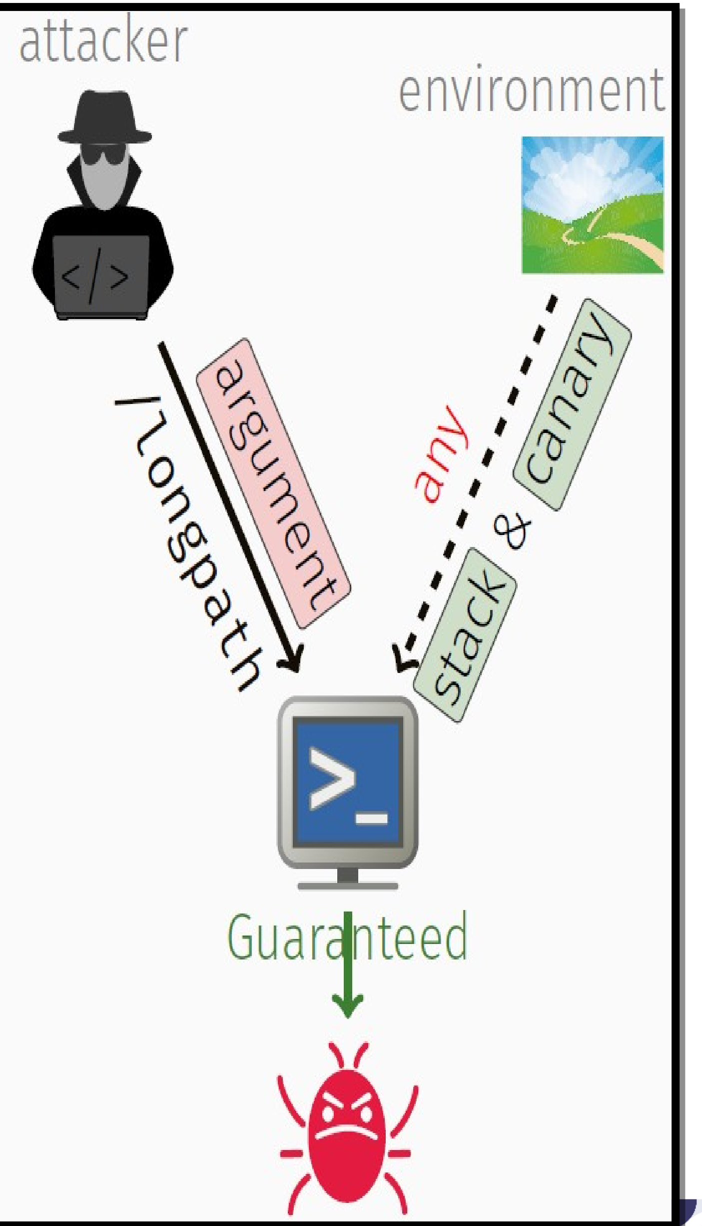
Our proposal [CAV 2018, CAV 2021, FMSD 2022]

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	Robust Reachability of ℓ
$\exists a, x. (a, x) \vdash \ell$	$\exists a. \forall x. (a, x) \vdash \ell$



Adapting BMC and SE

Path merging

Optional in SE

Required for completeness in Robust SE

...and a few other differences

assume ψ: $\exists a.\forall x.\psi \Rightarrow \phi$ instead of $\exists a.\forall x.\psi \wedge \phi$

path pruning: no extra quantifier

concretization: only works on controlled values

$$\exists a.\forall X.\varphi \xrightarrow[X \text{ to } 90]{\text{concretize}} \exists a.\forall X.\underbrace{X = 90}_{\bot} \wedge \varphi$$

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"

Alternative formalism: non-interference

	Behavior does not depend on x	Implies reachability
Non Interference	for all a	no
Robust reachability	for a single a	yes

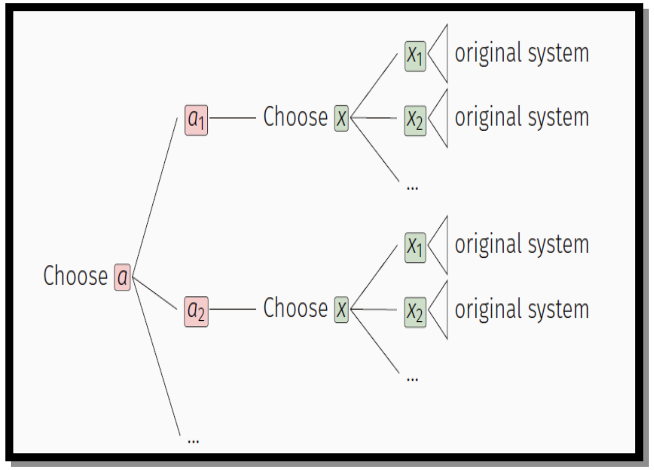
$$\text{Non-interference} + \text{Reachability} \underset{\not\Leftarrow}{\Rightarrow} \text{Robust Reachability}$$

Alternative formalisms (2)

As a hyperproperty, robust reachability is pure hyperliveness

- not a trace property (most studied case)
- not (k-)hypersafety ($\Rightarrow$ not solvable with self-composition)

Temporal logics: Expressible in CTL, HyperLTL, but no provers for generic programming languages



Need a dedicated proof method!

Stepping back

- **Robust reachability draws a line between some good bugs and bad bugs**
  - Based on replicability

- **Several formalisms can express robust reachability** [games, ATL, hyperLTL, CTL]
  - Yet no efficient software-level checkers

- **A few prior attempts, on different dimensions**
  - Quantitative or probabilistic approaches (model checking, non interference)
  - Automated Exploit Generation (Avgerinos et al., 2014)
  - Test Flakiness (O'Hearn, 2019) [a specific case of robust reachbaility]
  - Fair model checking (Hart et al., 1983)

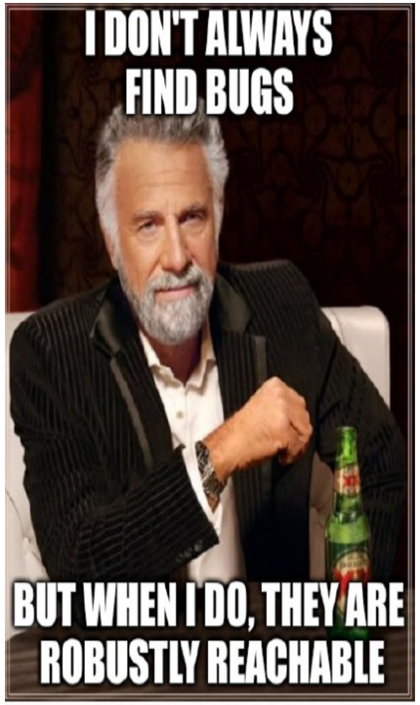
- **Qualitative « all or nothing » robust reachability may be too strong**
  - Mitigation : add user-defined constraints over the uncontrolled variables
  - WIP : quantitative definitions, inference of robustness conditions

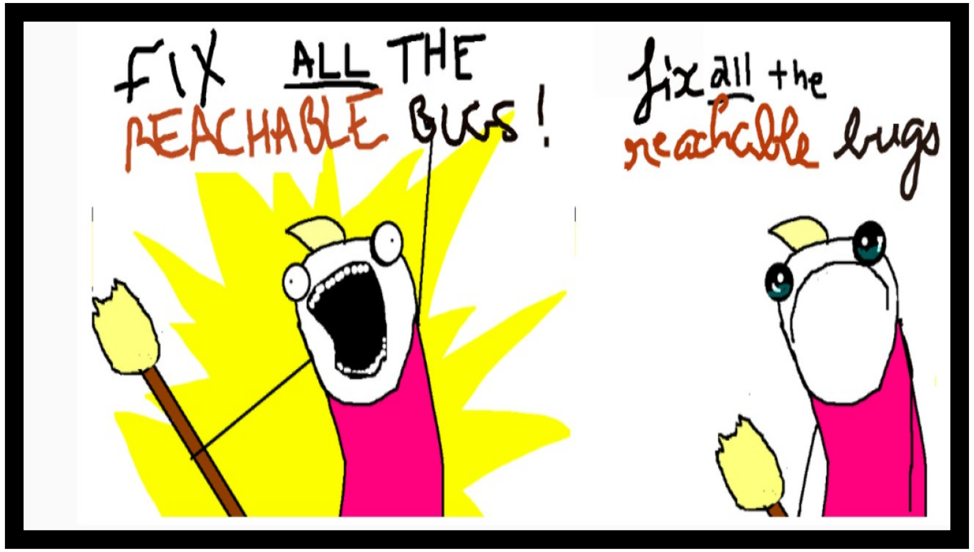
Potential applications

- **Better testing / bug finding tools**
  - Ex: find replicable bugs
  - Ex: generate non-flaky tests

- **Test suite evaluation**
  - Are the test case replicable?

- **Bug prioritisation**
  - Replicable bugs first





// aparté : solving robustness queries w/o quantifiers [CAV 2018]

Idea : 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 $a^\bullet \wedge b^\bullet \wedge \neg x^\bullet$ ($a^\bullet$, $b^\bullet$, $x^\bullet$: 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 $(ax + b > 0) \wedge (a = 0)$

OUTLINE

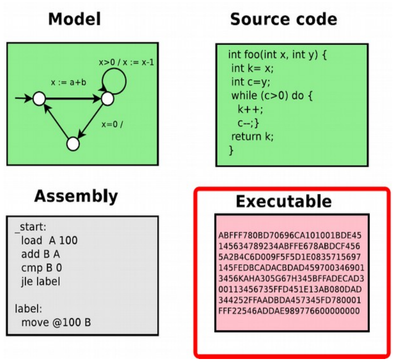
- **Introduction**

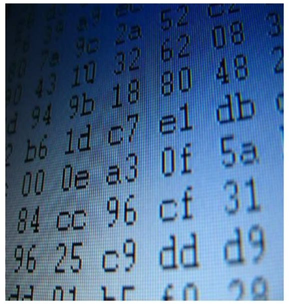
- **Challenges of automated binary-level security analysis**
- **BINSEC & Symbolic Execution for Binary-level Security**

- **Robust reachability and bugs that matter**
- **Adversarial reachability**

- **Conclusion, Take away and Disgression**

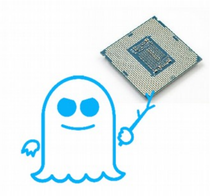
- **Problem : what about the attacker capabilities ?**

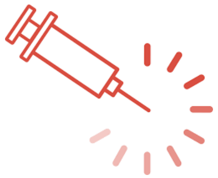


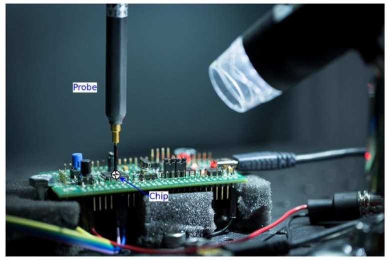


- Binary code
- Attacker
- Properties

Context







- Many techniques and tools for security evaluations.
- Usually consider a weak attacker, able de **craft smart inputs**.
- **Real-world attackers are more powerful: various attack vectors + multiple actions** in one attack.

Hardware attacks

Software-implemented hardware attacks

Electromagnetic pulses | Power glitch | Clock glitch | Laser beam | Faultline | DVFS

Race condition | Load Value Injection | Spectre | Rowhammer

Micro-architectural attacks

Man-At-The-End attacks

Context

- How to deal with that ?
- principled
- efficient

- ❑ Many techniques and tools for security evaluations.
- ❑ Usually consider a weak attacker, able de **craft smart inputs**.
- ❑ **Real-world attackers are more powerful: various attack vectors + multiple actions** in one attack.

Hardware attacks

Software-implemented hardware attacks

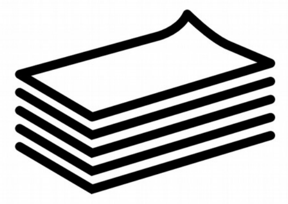
Electromagnetic pulses | Power glitch | Clock glitch | Laser beam | Faultline | DVFS

Race condition | Load Value Injection | Spectre | Rowhammer

Micro-architectural attacks

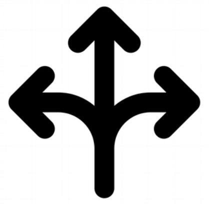
Man-At-The-End attacks

State-of-the-Art: software-implemented fault injection



Mutant generation: create a new mutated program for each fault configuration.

k (faults) among n (lines) mutant generated



Forking technique: fork the analysis with a fault at each possible fault location.

k (faults) among n (lines) paths created

- ❑ Both faces scalability issues (path explosion) hindering mutli-fault analysis.
- ❑ They don’t provide formalization of the underlying problem.

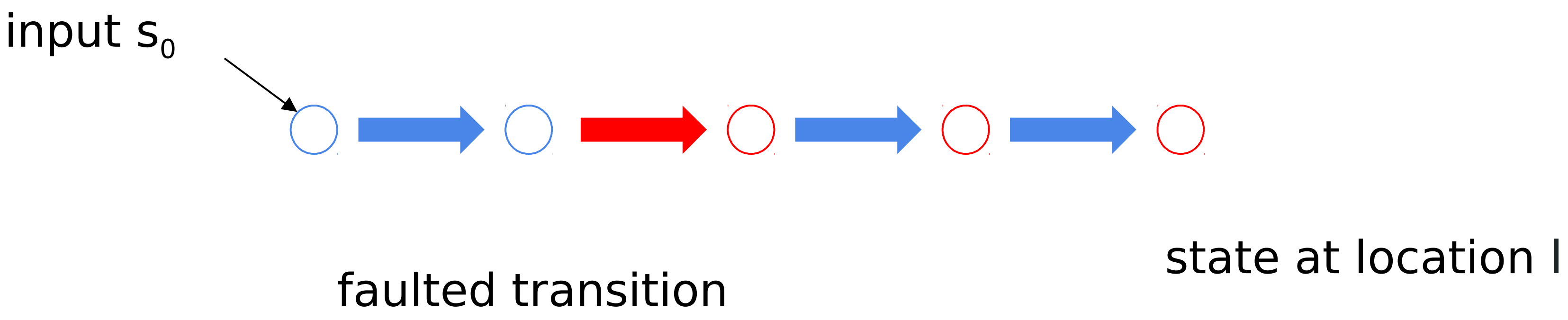
Contributions

- We formalize the **Adversarial Reachability** problem
- We propose **Adversarial Symbolic Execution,** with dedicated **optimizations.**
- We propose an **implementation** and **evaluation** of our technique.
- We perform a security analysis of the **WooKey bootloader**.

Adversarial reachability

Goal: have a formalism extending standard reachability to reason about a program execution in presence of an advanced attacker.

Adversarial reachability: A location l is adversarialy reachable in a program P for an attacker model A if $S_0 \mapsto^* l$,
where $\mapsto^*$ is a succession of program instructions interleaved with faulty transitions.



Forking encodings

Original

x := y

Forking

Non deterministic choice between fault or normal if $nb_f < max_f$

x := y

x := $fault_i$

nb_f ++

- ❑ Covers all adversarial behaviors
- ❑ Number of path exponential with # fault injection points

Forkless encodings and FASE

Original

$x := y$

Forkless

$x := \text{ite } here_i \text{ ? } fault_i : y$

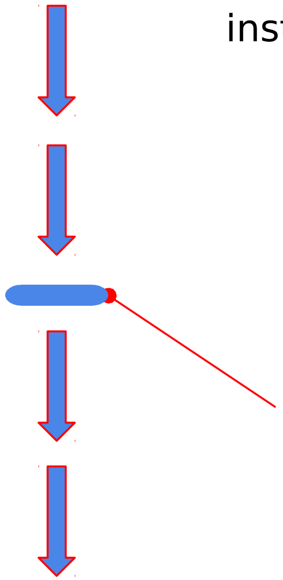
$here_i \in [0,1]$, $\Sigma\ here_i \leq max_f$

- ❑ Covers all adversarial behaviors
- ❑ **Only 1 path**
- ❑ **More complex formulas**

Early Detection of fault Saturation (EDS)

FASE

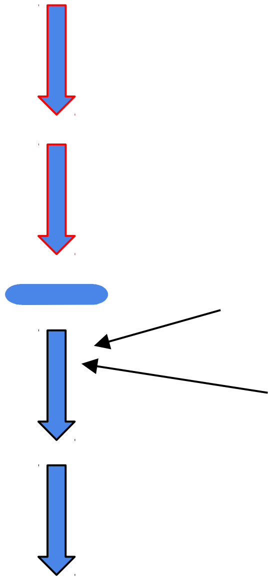
Potentially faulted instruction (with ite)



We need max_f faults to go beyond that point on that path.

FASE-EDS

SAT with a fault margin
or SAT with exactly the fault budget
or infeasible



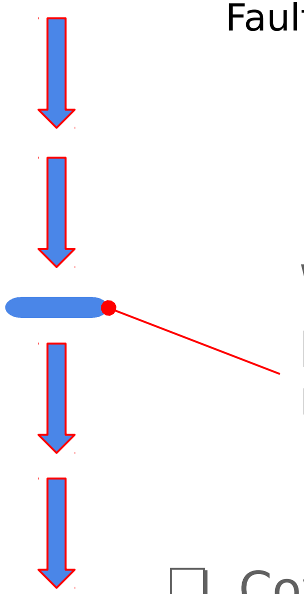
Instruction not faulted

- ❑ Covers all adversarial behaviors, as complete as FASE
- ❑ Only 1 path
- ❑ Reduce number of fault injections along a path

Injection On Demand (IOD)

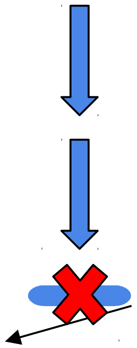
FASE

Faulted instruction



We can’t go beyond that point on that path without more faults.

FASE-IOD

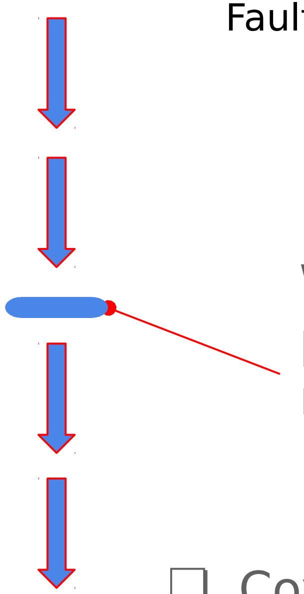


- ❑ Covers all adversarial behaviors, as complete as FASE
- ❑ Only 1 path
- ❑ Reduce number of fault injections
- ❑ Additional queries

Injection On Demand (IOD)

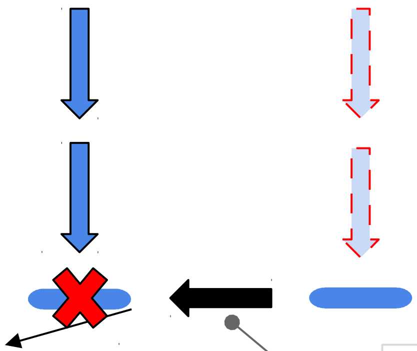
FASE

Faulted instruction



We can't go beyond that point on that path without more faults.

FASE-IOD



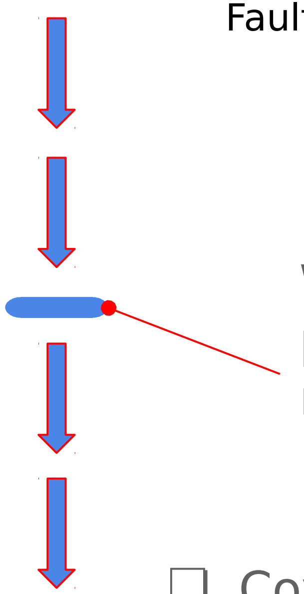
Path predicate switched for the faulted one

- ❑ Covers all adversarial behaviors, as complete as FASE
- ❑ Only 1 path
- ❑ Reduce number of fault injections
- ❑ Additional queries

Injection On Demand (IOD)

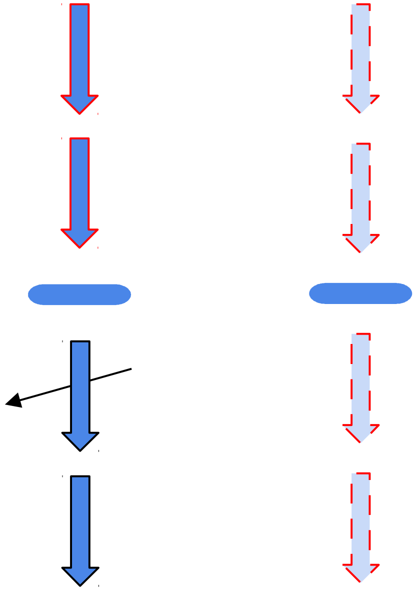
FASE

Faulted instruction



We can't go beyond that point on that path without more faults.

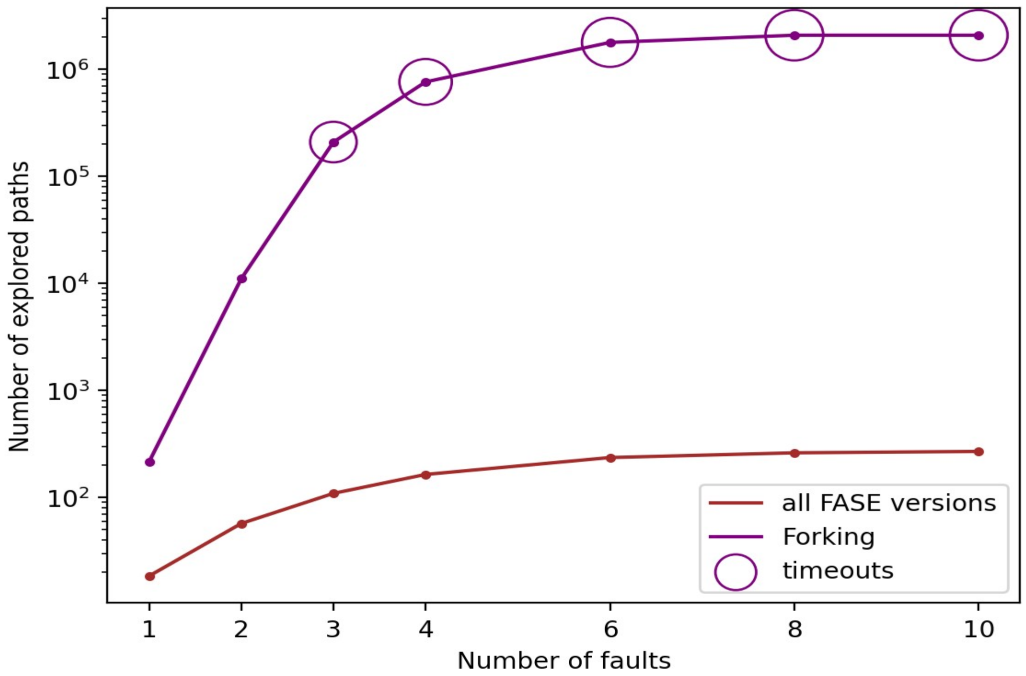
FASE-IOD

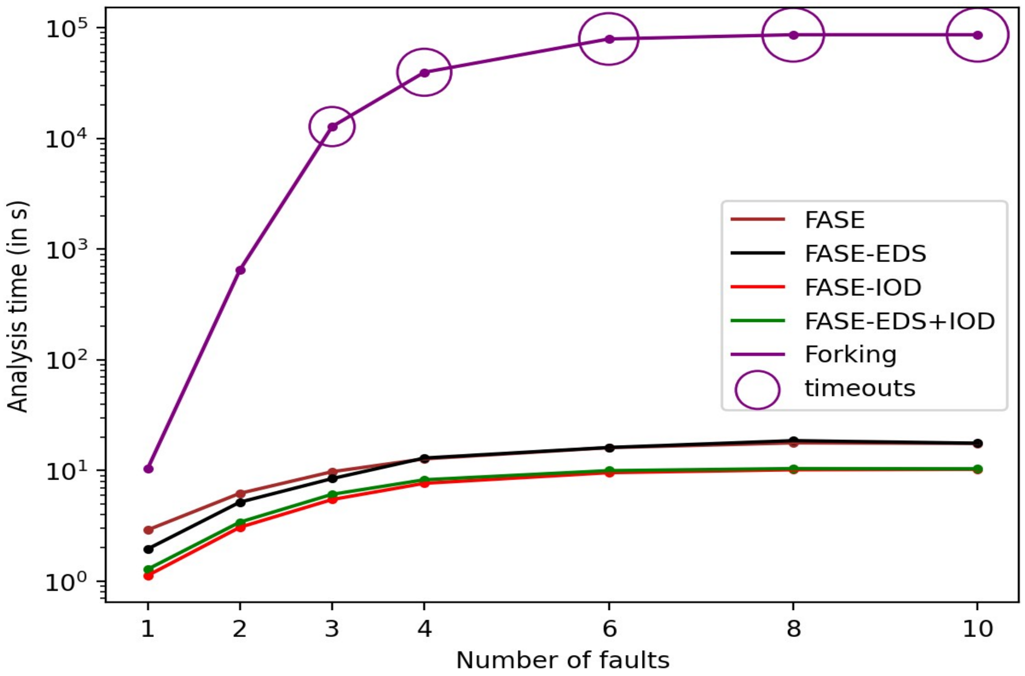


Bonus: under-approximation of nb_f

- ❑ Covers all adversarial behaviors, as complete as FASE
- ❑ Only 1 path
- ❑ Reduce number of fault injections
- ❑ Additional queries

RQ2 - scaling without path explosion





- ➔ Forking explodes in explored paths while FASE doesn't.
- ➔ Translates to improved analysis time overall.

Security scenarios using different fault models

CRT-RSA: [1]

- basic vulnerable to 1 reset → OK
- Shamir (vulnerable) and Aumuler (resistant) → TO

Secret-keeping machine: [2]

- Linked-list implementation vulnerable to 1 bit-flip in memory → OK
- Array implementation resistant to 1 bit-flip in memory → OK
- Array implementation vulnerable to 1 bit-flip in registers → OK

Secswift countermeasure: llvm-level CFI protection by STMicroelectronics [3]

- SecSwift impementation [4] applied to VerifyPIN_0 → early loop exit attack with 1 arbitrary data fault or test inversion in valid CFG

[1] Puys, M., Riviere, L., Bringer, J., Le, T.h.: High-level simulation for multiple fault injection evaluation. In: Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance. Springer (2014)
[2] Dullien, T.: Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing (2017)
[3] de Ferrière, F.: Software countermeausres in the llvm risc-v compiler (2021), https://open-src-soc.org/2021-03/media/slides/3rd-RISC-V-Meeting-2021-03-30-15h00-Fran%C3%A7ois-de-Ferri%C3%A8re.pdf
[4] Lacombe, G., Feliot, D., Boespflug, E., Potet, M.L.: Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In: PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS) (2021)

Case study

WooKey bootloader: secure data storage by ANSSI, 3.2k loc.

Goals:

1. Find known attacks (from source-level analysis)
    a. Boot on the old firmware instead for the newest one [1]
    b. A buffer overflow triggered by fault injection [1]
    c. An incorrectly implemented countermeasure protecting against one test inversion [2]

2. Evaluate countermeasures from [1]
    a. Evaluate original code → **We found an attack not mentioned before**
    b. Evaluate existing protection scheme [1] **(not enough)**
    c. **Propose and evaluate our own protection scheme**

[1] Lacombe, G., Feliot, D., Boespflug, E., Potet, M.L.: Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In: PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS) (2021)
[2] Martin, T., Kosmatov, N., Prevosto, V.: Verifying redundant-check based countermeasures: a case study. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. (2022)

Stepping back

- Adversarial reachability takes an active attacker into account
- Well known in cryptographic protocol verification, not for code

- generic: reachability, hyper-reachability, non termination

- Scalability ?
- Which capabilities for the attacker? [link with Harware security community]
- Strong link with robust reachability

OUTLINE

- Introduction

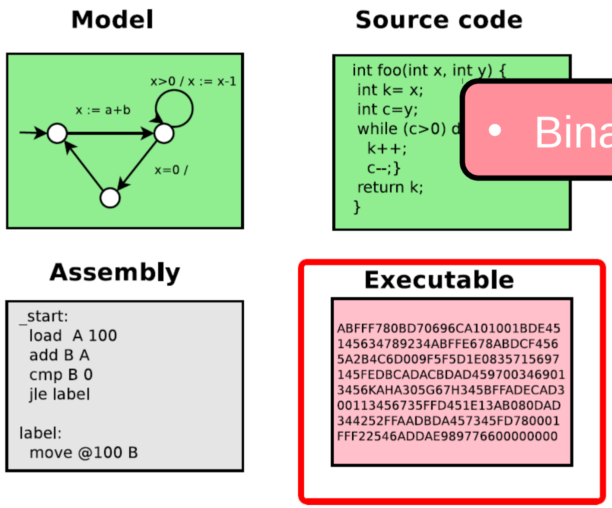
- Challenges of automated binary-level security analysis
- BINSEC & Symbolic Execution for Binary-level Security

- Robust reachability and bugs that matter
- Adversarial reachability

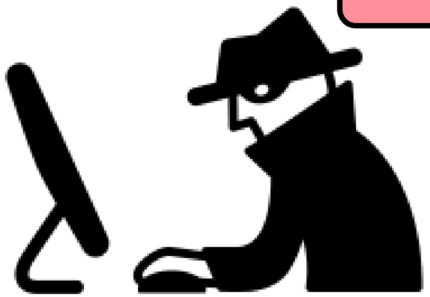
- Conclusion, Take away and Disgression

TAKE AWAY: SECURITY IS NOT SAFETY

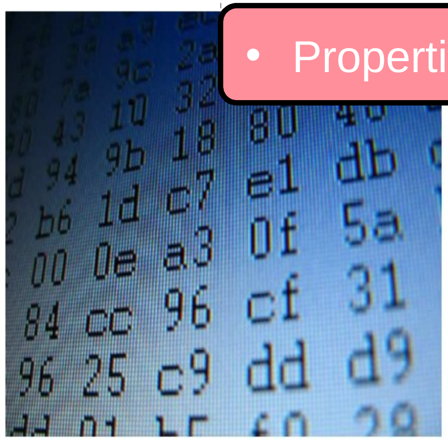
- - Fun for FM/PL researchers
- - Important applications



- Binary code



- Attacker



- Properties

- Reachability is well suited for safety, yet
- security leads to many new interesting variations
- Still many things to do !!
- Symbolic Execution appears to be versatil enough
- BINSEC is open source, check it [with us]

https://binsec.github.io/

- Looking for postdoc & PhD

THANK YOU