Automated Program Analysis for Security : What About the Attacker ?

FROM RESEARCH TO INDUSTRY

Sébastien Bardin

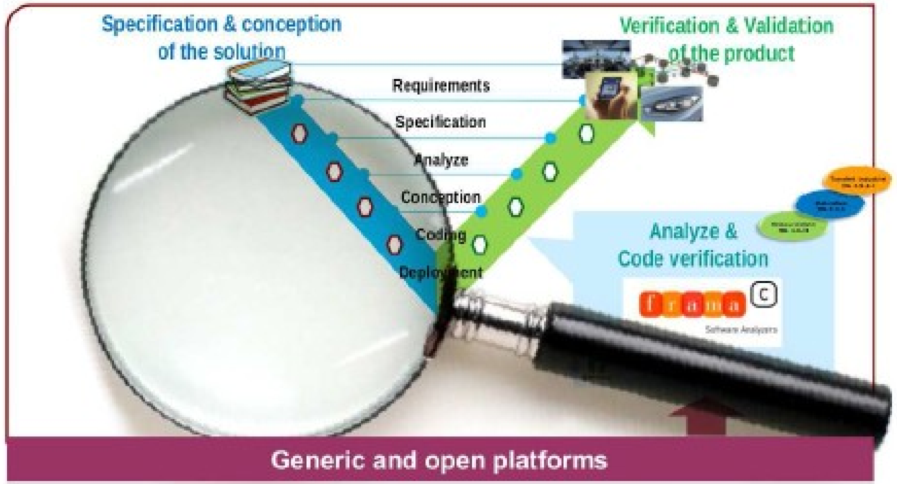
Senior Researcher, CEA Fellow

Head of the BINSEC team

Laboratoire de Sûreté et Sécurité des Logiciels

DILS/LSL : Lab. For Software Security and Safety

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



frama C Software Analyzers

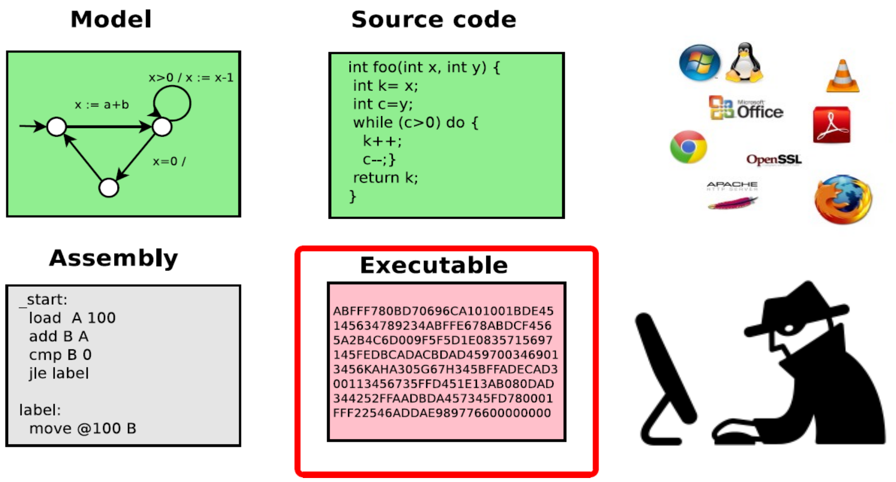
GATeL

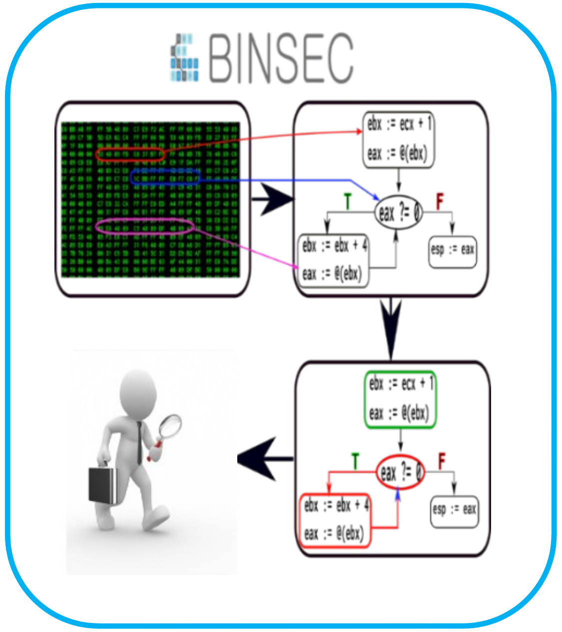
PathCrawler

UNISIM

POLAR SSL

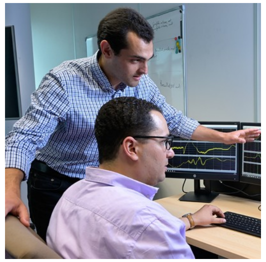
The BINSEC Group:
ADAPT FORMAL METHODS TO BINARY-LEVEL SECURITY ANALYSIS





https://binsec.github.io/

TEAM WORK SINCE 2012 [+ UGA, LORIA, INRIA]





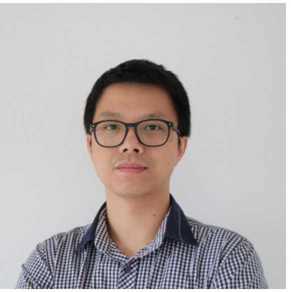












This Talk in a Nutshell

- **Program-level security is a key aspect** [yet, a single bug can ruin everything]
-

- **Program Analysis (PL) and Formal Methods come from critical safety needs**
  - Damn good there (in the hands of experts)
  - Allow to prove the absence of bugs, or find them thoroughly

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

Questions: how does security differ from safety?

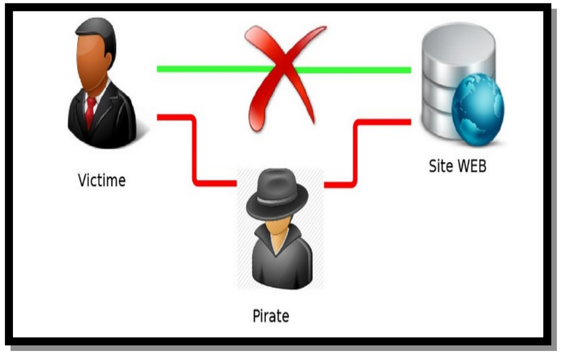
- ***Answer : the attacker***
- ***This talk: share some insights and results from the BINSEC team @DILS***

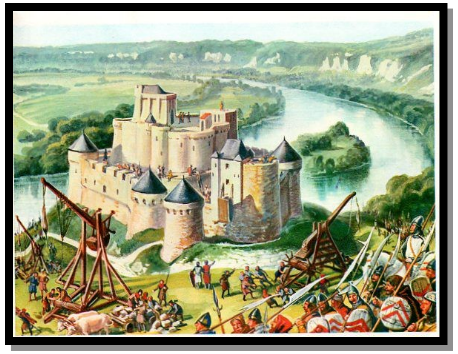
THE SECURITY GAME

- **The defender:** try to secure the whole system

- **The attacker:** try to abuse the system
  - Why: for fun & **profit**
  - How: by taking advantage of a single flaws (bugs)

- **The user:** collateral damage

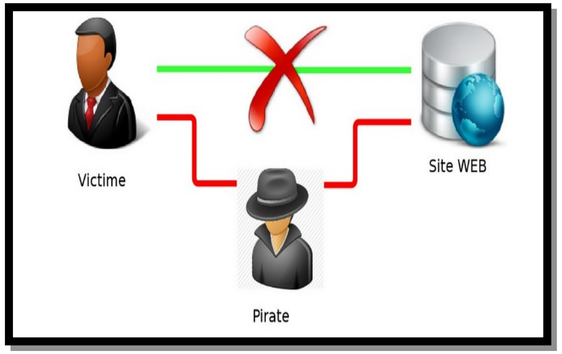


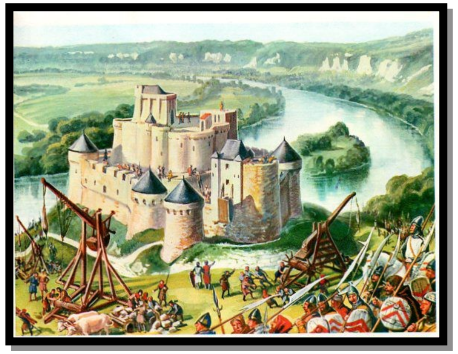


THE SECURITY GAME

- **The defender:** try to secure the **whole** system
- **The attacker:** try to abuse the system
  - Why: for fun & **profit**
  - How: by taking advantage of a **single** flaws (bugs)
- **The user:** collateral damage

Dissymetric battlefield
Advantage to the attacker
(in most cases)



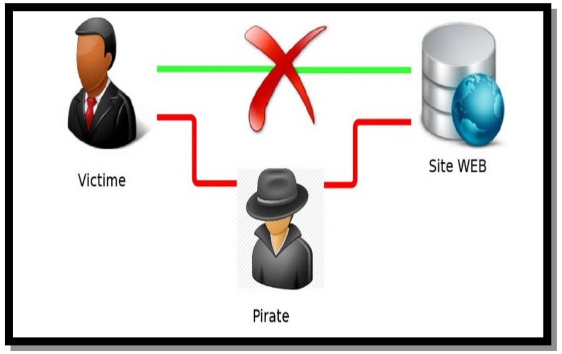


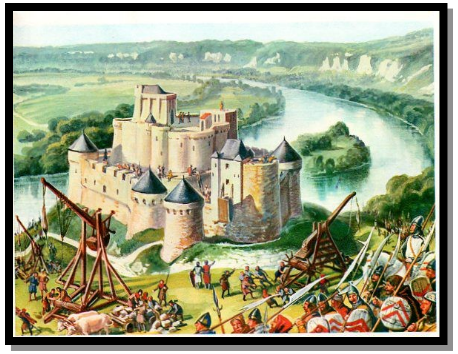
THE SECURITY GAME

- **The defender:** try to secure the **whole** system
- **The attacker:** try to abuse the system
  - Why: for fun & **profit**
  - How: by taking advantage of a **single** flaws (bugs)
- **The user:** collateral damage

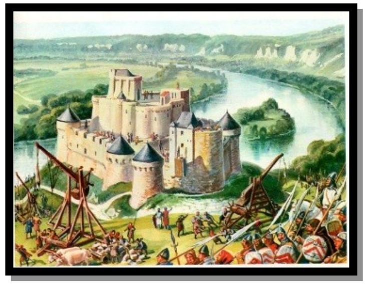
Dissymetric battlefield
Advantage to the attacker
(in most cases)

- most attacks come from implementation bugs
- bugs are inevitable





OUR VIEW



Dissymetric battlefield
Advantage to the attacker
(in most cases)

- most attacks come from implementation bugs
- bugs are inevitable

Quite depressing ...

What if software could be immune to large classes of bugs?
What if bugs could be found (and patch) automatically?

OUTLINE

- **Introduction [The Sad Truth]**
- **Reasoning about programs [A New Hope]**
- **What about the attacker? [The Evil Returns]**
- **Some results [Hard Battle In Progress]**
- **Conclusion, Take away and Disgression**

THEN CAME FORMAL METHODS

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

Key concepts : $M \models \varphi$

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

Success in (regulated) safety-critical domains

THEN CAME FORMAL METHODS

- 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

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

- Reason about infinite sets of behaviours

Success in (regulated) safety-critical domains

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

Simple example

- **Goal : prove result is positive**

```
int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
}
```

Simple example

- Goal : prove result is positive

```
int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
```

- X >=0 hence r >=0

- X <0 hence r >=0

- r >=0

PLEASE, PAY ATTENTION

- Goal : prove result is positive

```
int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
```

- False cause of integer underflow on x = minINT

- X >=0 hence r >=0

- X <0 hence r >=0

- R >=0 ???????

Please

Can prove things
Can help find bugs

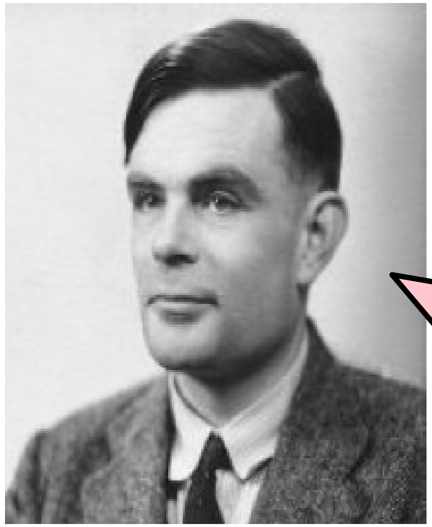
```c
int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
}
```

```c
/*@ requires -1000 <= x <= 1000;
    ensures \result >= 0;
    */

int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
}
```

- A correct version

- False because of possible underflow

They knew it was impossible, so they did it anyway



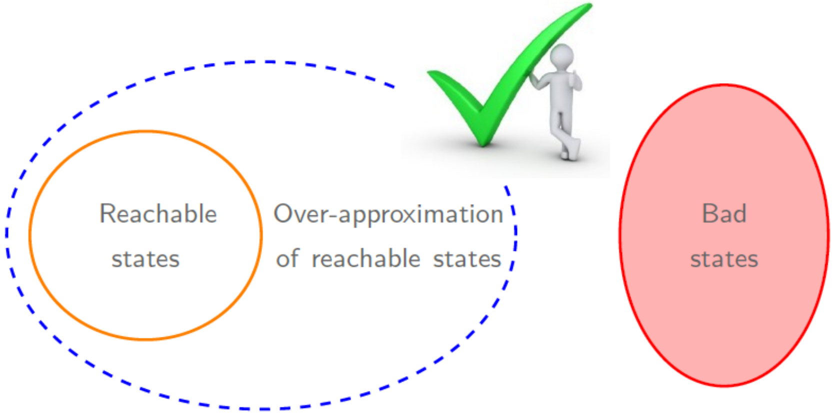
Cannot have analysis that
- Terminates
- Is perfectly precise

On all programs

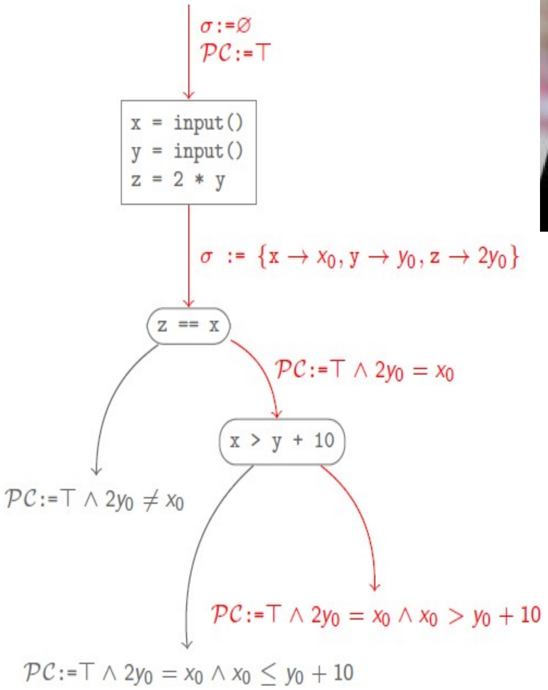
- **Weakest precondition calculi** [1969, Hoare]
- **Abstract Interpretation** [1977, Cousot & Cousot]
- **Model checking** [1981, Clarke - Sifakis]

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

Formal methods zoo : so many of them, so little time for the talk

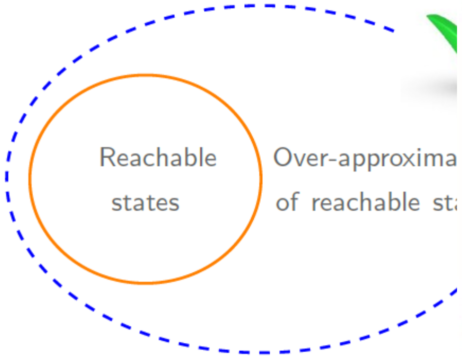


Full proofs



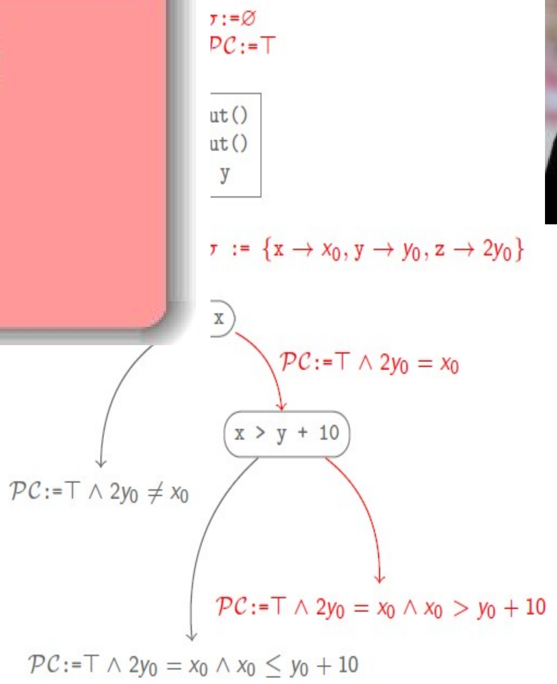
Bounded verification – bug finding

Formal methods zoo : so many of them, so little time for the talk



Key concepts : $M \models \varphi$

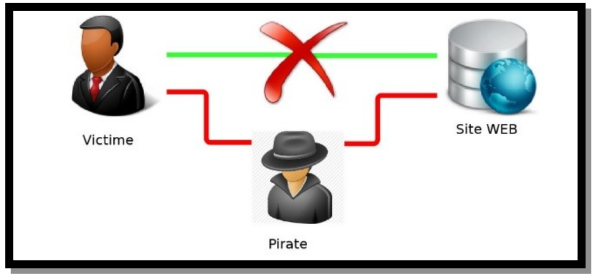
- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check



Full proofs

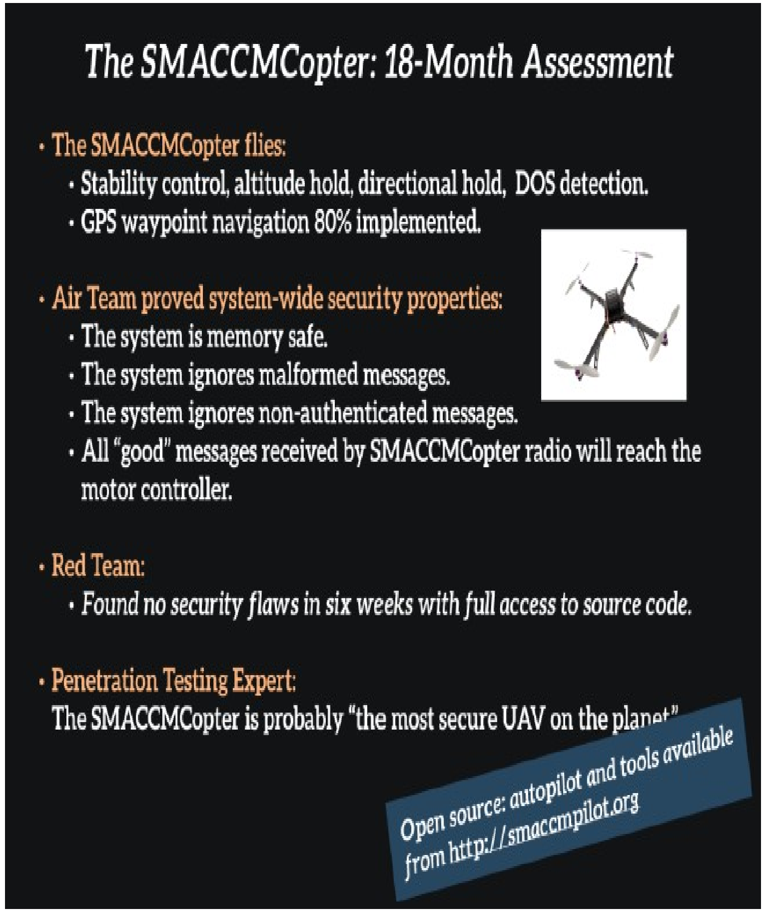
Bounded verification – bug finding

WHAT ABOUT USING THEM IN SECURITY ?



TLS 1.3

Good Idea !



The SMACCMCopter: 18-Month Assessment

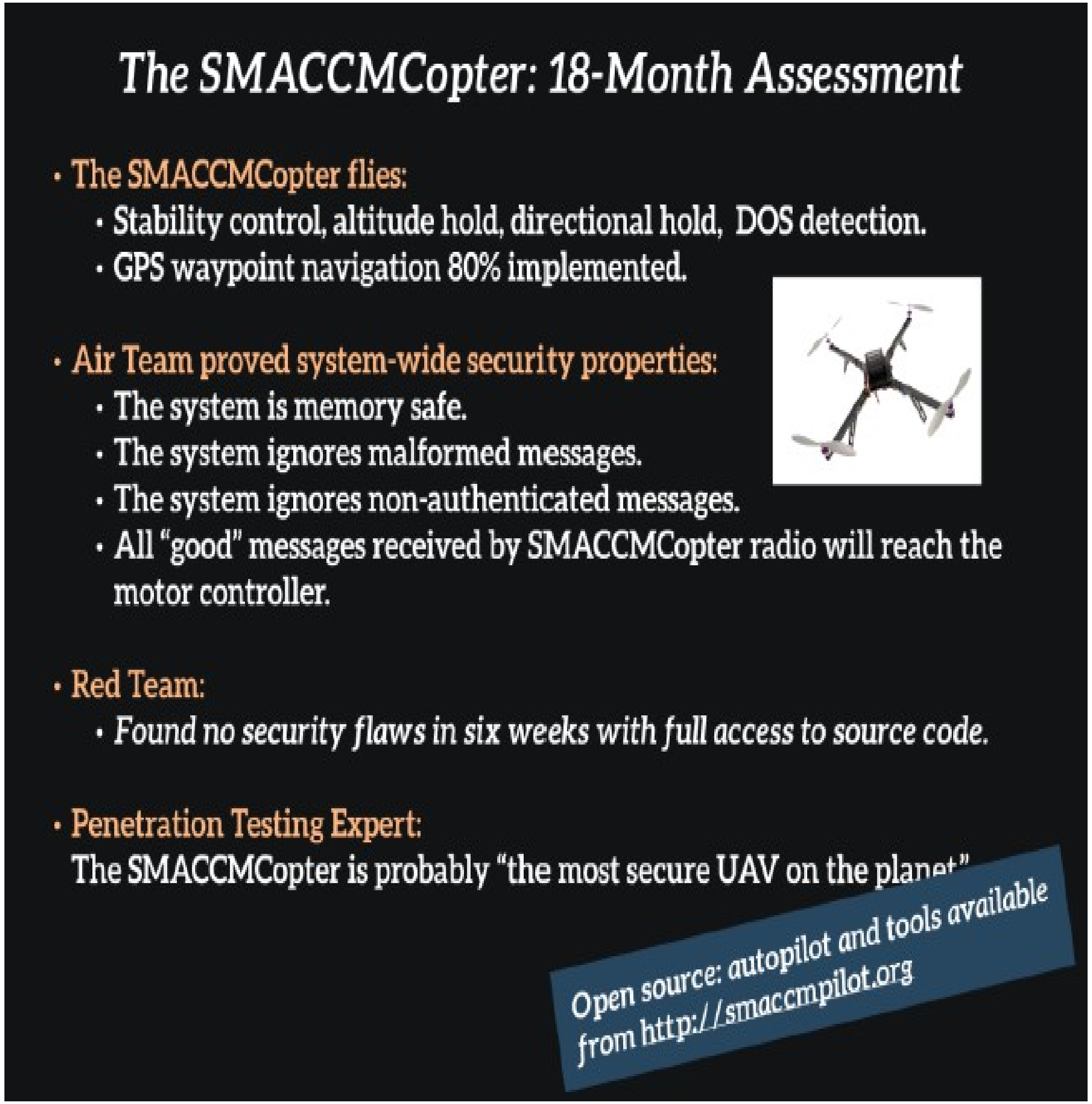
- The SMACCMCopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.
- Air Team proved system-wide security properties:
  - The system is memory safe.
  - The system ignores malformed messages.
  - The system ignores non-authenticated messages.
  - All "good" messages received by SMACCMCopter radio will reach the motor controller.
- Red Team:
  - *Found no security flaws in six weeks with full access to source code.*
- Penetration Testing Expert:
  The SMACCMCopter is probably "the most secure UAV on the planet"

Open source: autopilot and tools available from http://smaccmpilot.org

Formally hardened UAV

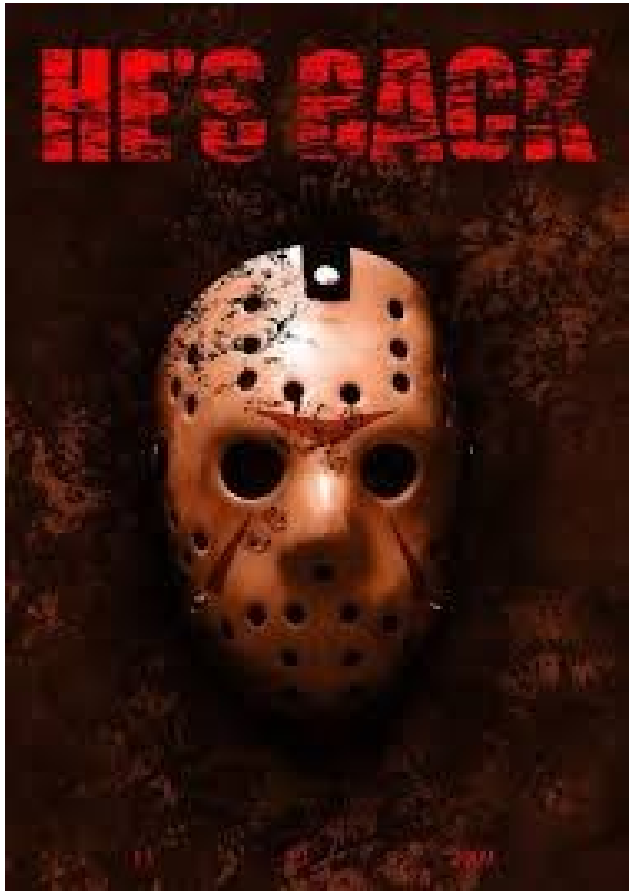
- Developped from scratch

Survives 6 weeks of red team attacks with full code & doc access



End of the story ?

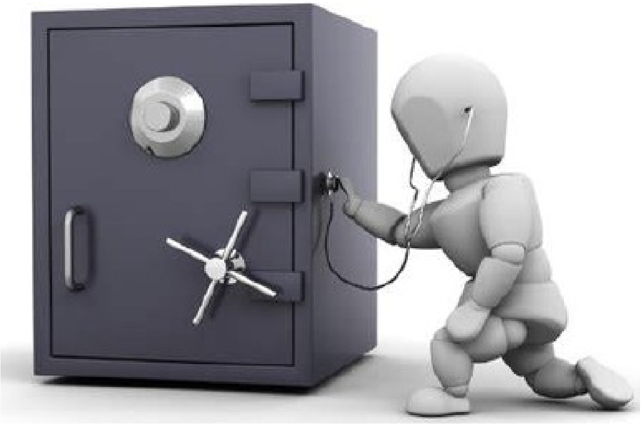
End of the story ? Not yet ...



OUTLINE

- Introduction [The Sad Truth]
- Reasoning about programs [A New Hope]
- What about the attacker? [The Evil Returns]
- Some results [Hard Battle In Progress]
- Conclusion, Take away and Disgression

EXAMPLE: side channel attacks

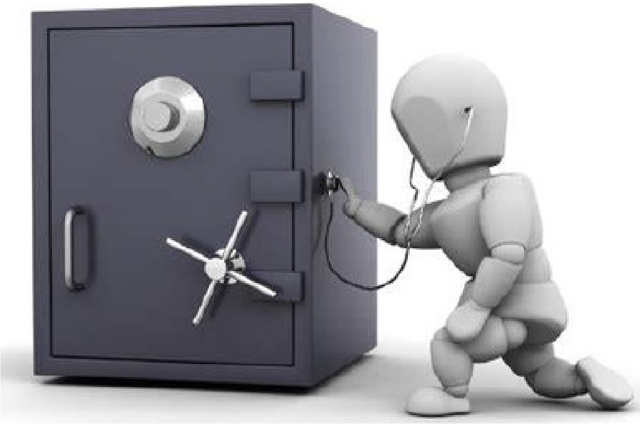


```
private char[4] secret;

boolean CheckPassword (char[4] input) {

}
```

- Yes, sometimes
- Come from the implementation

- Can you retrieve the secret with blackbox access?

EXAMPLE 1: side channel attacks



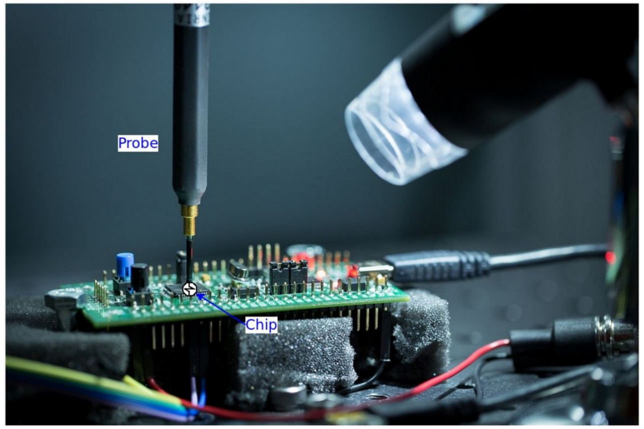
```
private char[4] secret;

boolean CheckPassword (char[4] input) {
 for (i=0 to 3) do
    if(input[i] != secret[i]) then
       return false;
    endif
 endfor
 return true;
}
```

- Can you retrieve the secret with blackbox access?

- Here, yes

EXAMPLE 2: fault injection attacks

```
private char[4] secret;

void CheckandPrint (char[4] input) {

 If (input == secret) then get-access() else stop() ;
}
```



- Can you get access without knowning secret?

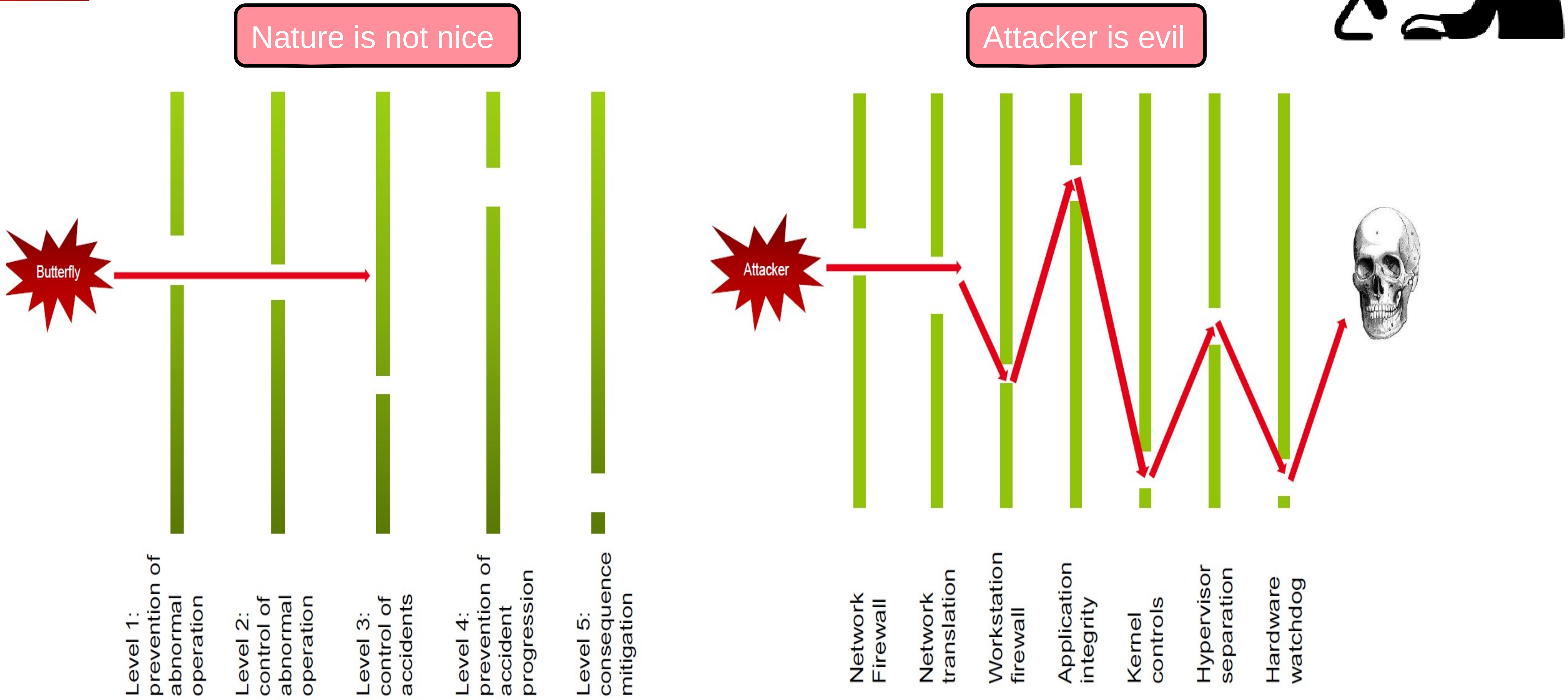
- Here, yes –
- not enough software counter measures

STANDARD PROGRAM ANALYSIS IS NOT (always) ENOUGH FOR SECURITY

Related to the safety vs security question

Introducing the attacker

CHALLENGE: ATTACKER



- We are reasoning worst case: seems very powerful!

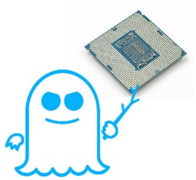
ATTACKER in Standard Program Analysis

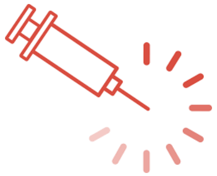
- **We are reasoning worst case: seems very powerful!**

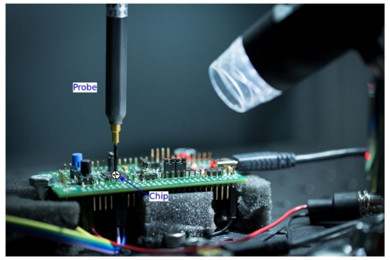
- **Still, our current attacker plays the rules: respects the program interface**
  - Can craft **very smart input**, but only through **expected input sources**

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







HOW TO TAKE THE ATTACKERs INTO ACCOUNT ?

What they can do

What they can observe

What they look for

Expressivity
vs
How to handle it efficiently

Key concepts : $M \models \varphi$

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

HOW TO TAKE THE ATTACKERs INTO ACCOUNT

NEXT : a few examples of how to take the attacker into account

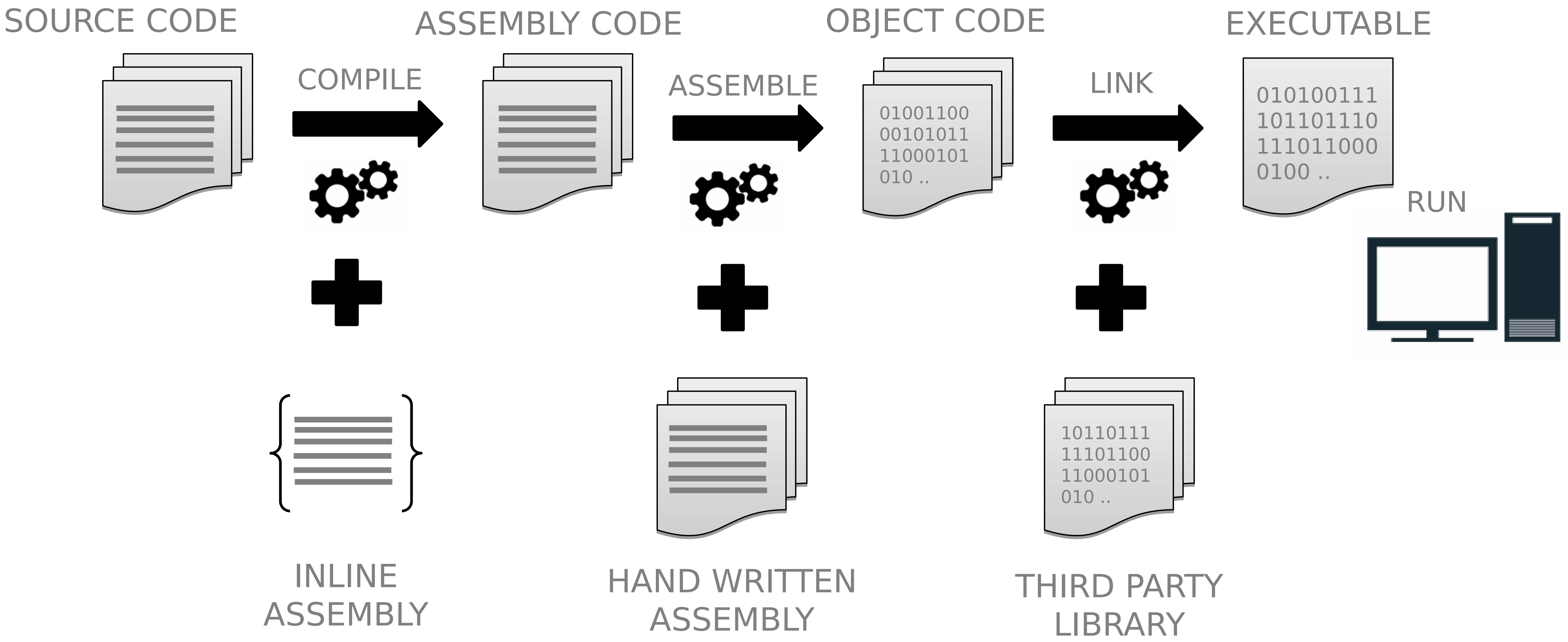
Taken from our experience with the BINSEC platform
Binary-level security analysis

OUTLINE

- Introduction [The Sad Truth]
- Reasoning about programs [A New Hope]
- What about the attacker? [The Evil Returns]
- Some results [Hard Battle In Progress]
- Conclusion, Take away and Disgression

OUTLINE

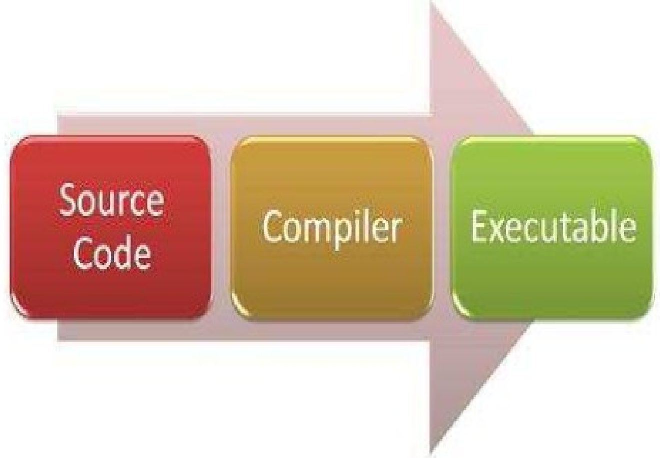
- Introduction [The Sad Truth]
- Reasoning about programs [A New Hope]
- What about the attacker? [The Evil Returns]
- Some results [Hard Battle In Progress]
  - detour : BINSEC
  - Taking the attacker into account in BINSEC
- Conclusion, Take away and Disgression


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

WHY GOING DOWN TO BINARY-LEVEL SECURITY ANALYSIS?

No source code

Post-compilation

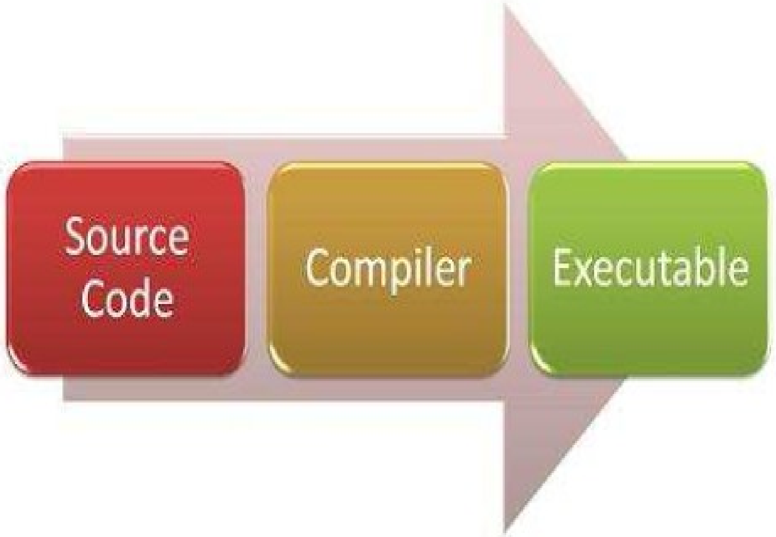


Malware comprehension

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

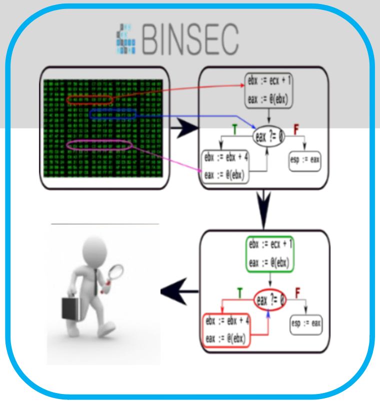
Security bug introduced by a non-buggy compiler

```
void  getPassword(void) {
char pwd [64];
if (GetPassword(pwd,sizeof(pwd))) {
/* checkpassword */
}
memset(pwd,0,sizeof(pwd));
}
```

OpenSSH CVE-2016-0777

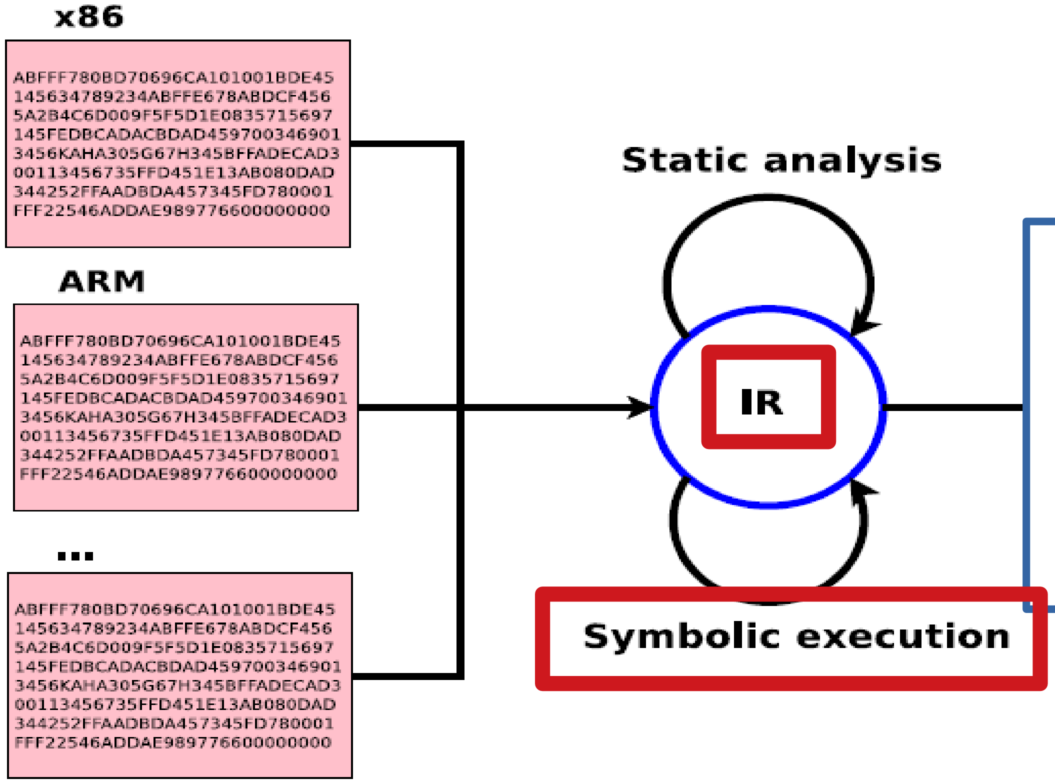
- **secure source code**
- **insecure executable**

BINSEC: brings formal methods to binary-level security analysis



Break | Prove | Protect

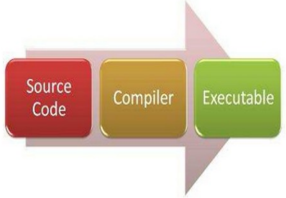
- Explore many input at once
  - Find bugs
  - Prove security
- Multi-architecture support
  - x86, ARM, RISC-V
  - 32bit, 64bit



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

https://binsec.github.io/






SYMBOLIC EXECUTION (Godefroid 2005)

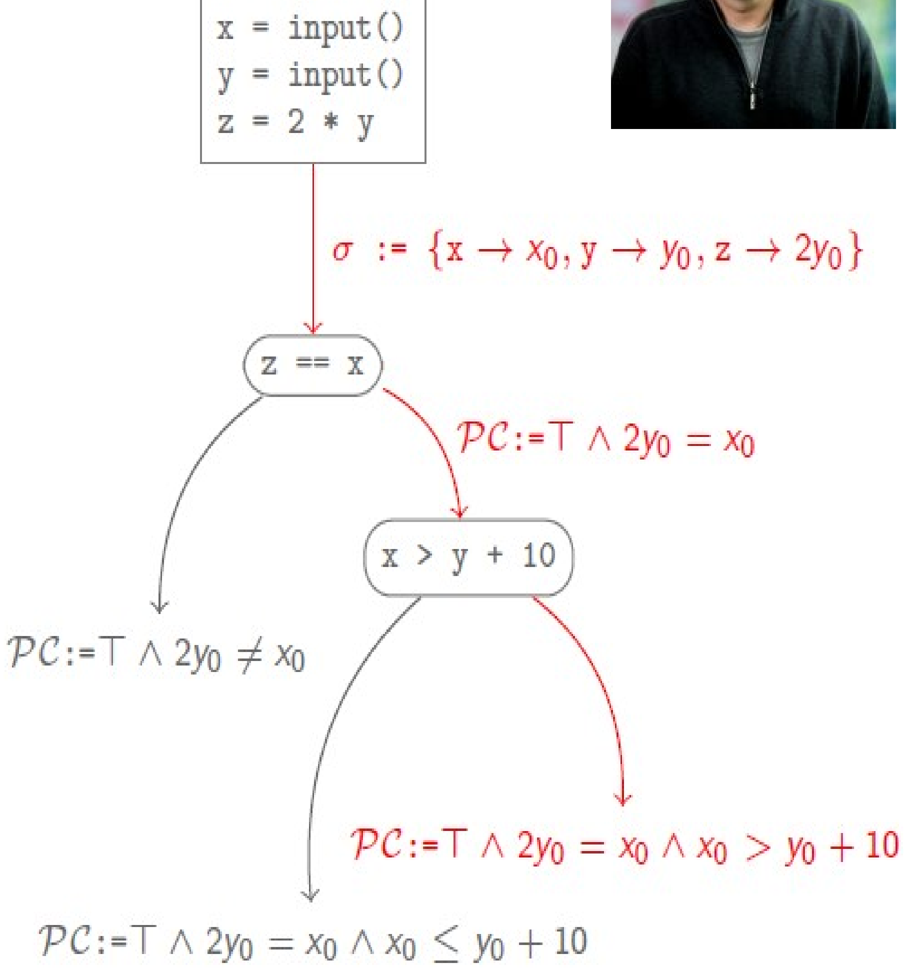
Find real bugs

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



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

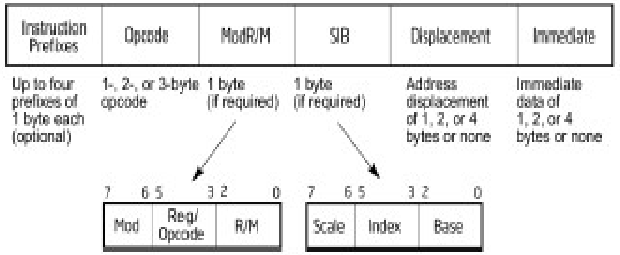
x86-32bit – ARMv7

- 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**



$$\boxed{81\ c3\ 57\ 1d\ 00\ 00} \overset{x86reference}{\Rightarrow} \boxed{ADD\ EBX\ 1d57}$$

```
(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)
```

BINSEC / SOME HIGHLIGHTS

- Vulnerability finding in open source code

- Fuzzing + program analysis
- Use-after-free, patch issues
- 15 CVE, 37 bugs

- Black Hat 2020, RAID 2020

Find a needle in the heap !

- Help reverse advanced malware

- Obfuscation detection & simplif
- 12 min for +400k instr.

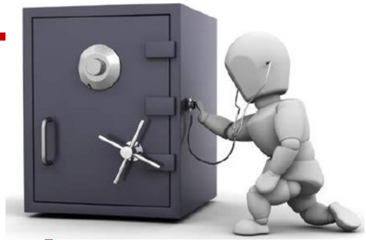
- Black Hat EU 2016, IEEE S&P 2017

- Verify cryptographic implementations
- Side channels and Spectre attacks

- Check 350+ crypto implementations

- 3 vulnerabilities introduced by compilers
- report possible flaws in standard protections

- IEEE S&P 2020, NDSS 2021



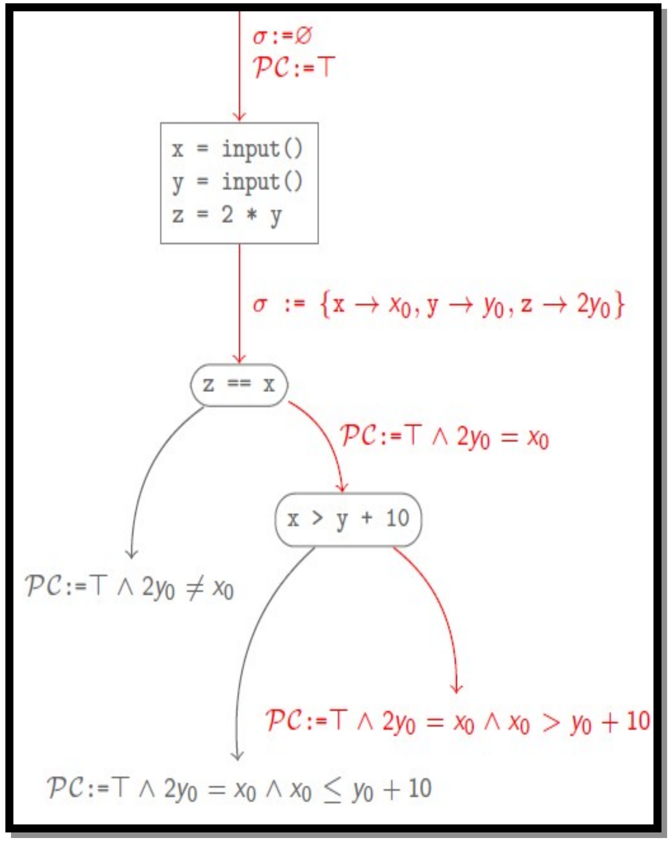
- Help handle inline assembly

- Verification-oriented decompilation
  - Tested on all Debian C+asm chunks
- Interface conformance checking
  - Found 100's of errors
  - propose patch, 10's got accepted

- **Security scenarios**
  - **Vulnerability analysis and automated exploit generation**
  - **Side channel attacks**
  - **Speculative side channel attacks**
  - **Physical fault injection**
  - **Bug priorisation**

Basic power

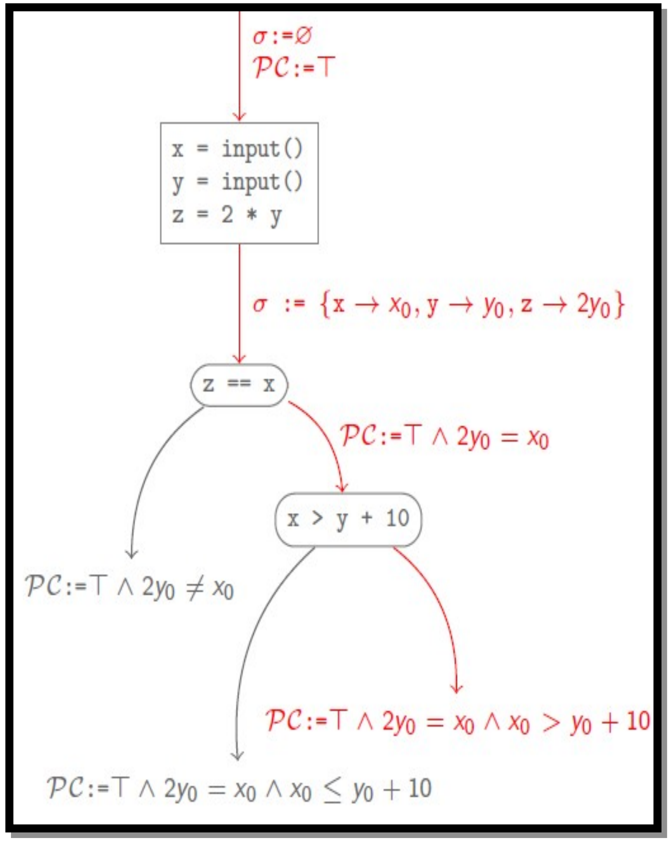
Vulnerability finding with symbolic execution (Godefroid et al., Cadar et al., Sen et al., etc.)



- Intensive path exploration

Challenge = path explosion

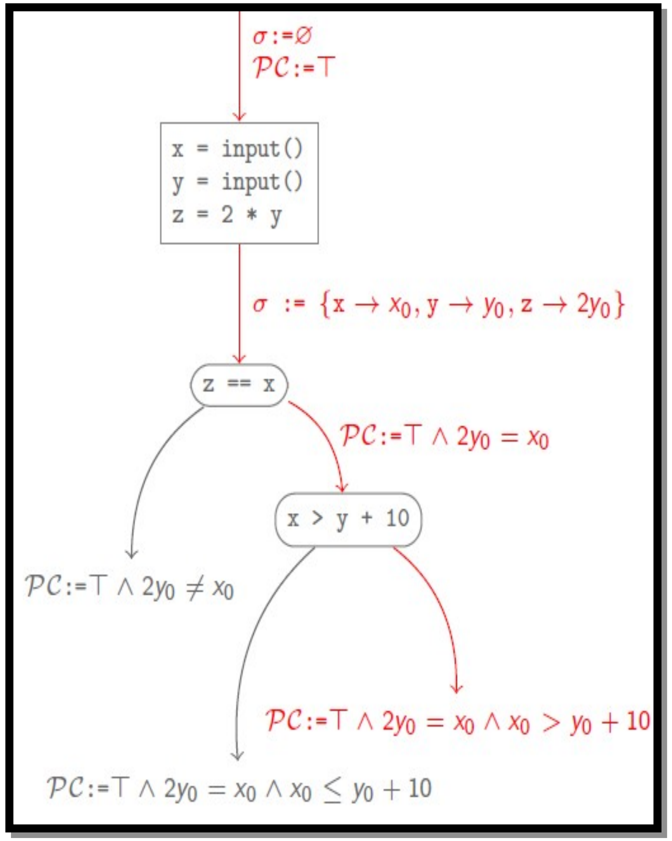
Vulnerability finding with symbolic execution (Godefroid et al., Cadar et al., Sen et al., etc.)



- 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

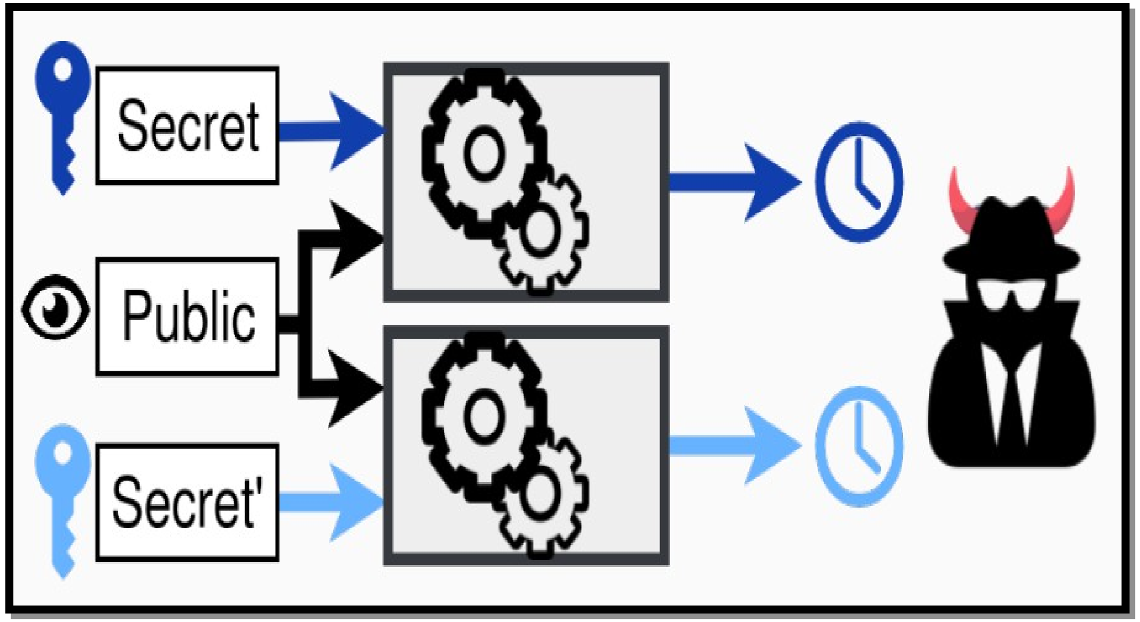
- Security scenarios
  - Vulnerability analysis and automated exploit generation
  - Side channel attacks
  - Speculative side channel attacks
  - Physical fault injection
  - Bug priorisation

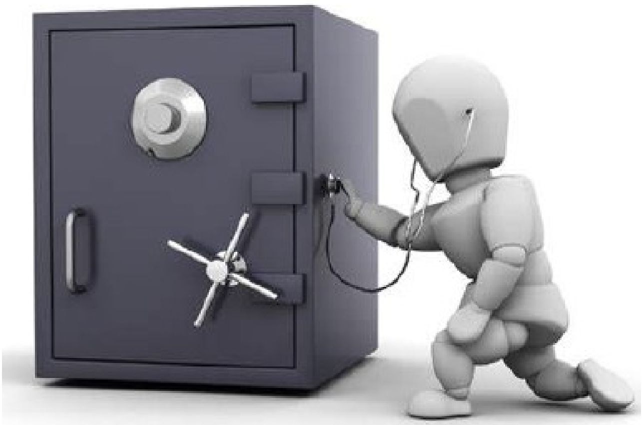
Can compare executions

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

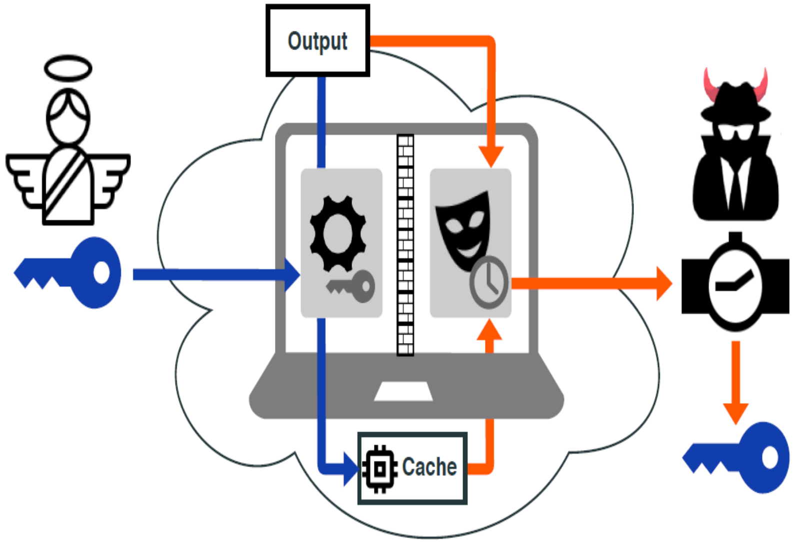
Information leakage

Properties over pairs of executions





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



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

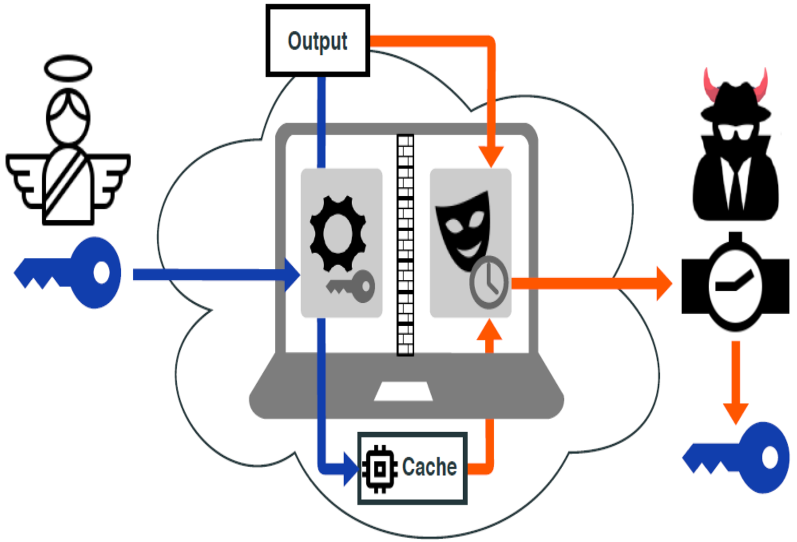
- Relational symbolic execution
- Follows paires of execution
- Check for divergence

- Property over paires

Key concepts : $M \models \varphi$

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

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	-

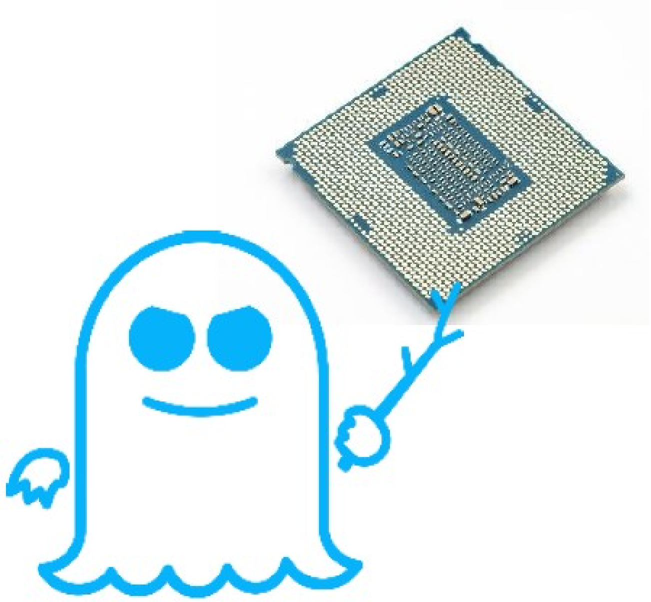
- Relational symbolic execution
- Follows paires of execution
- Check for divergence
- Sharing, dedicated preprocessing

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

- **Security scenarios**
  - **Vulnerability analysis and automated exploit generation**
  - **Side channel attacks**
  - **Speculative side channel attacks**
  - **Physical fault injection**
  - **Bug priorisation**

Can observe more

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-**

    load/

- Extends M into M_spec
- Property over paires

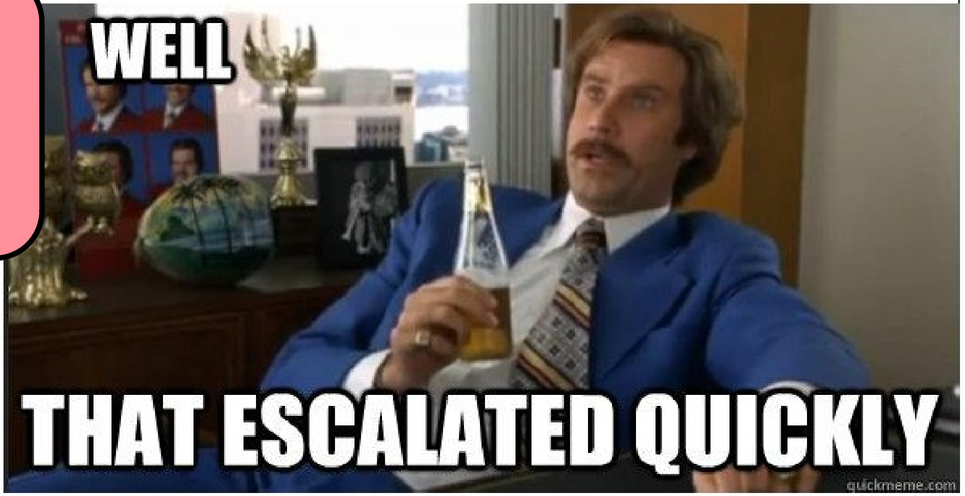
Key concepts : $M \models \varphi$

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

binary-level

Path explosion for Spectre-STL on Litmus tests (**328** instr.)

Semantics	Paths
Sequential semantics	**14**
Speculative semantics (Spectre-STL)	**37M**



Challenge !

- Some key finding : vulnerability in well known protection schemes spectre-pht protections may be vulnerable to spectre-stl

s tests (**328** instr.)

- Counte

- Path explosion:

  - **Spectre-**

  load/

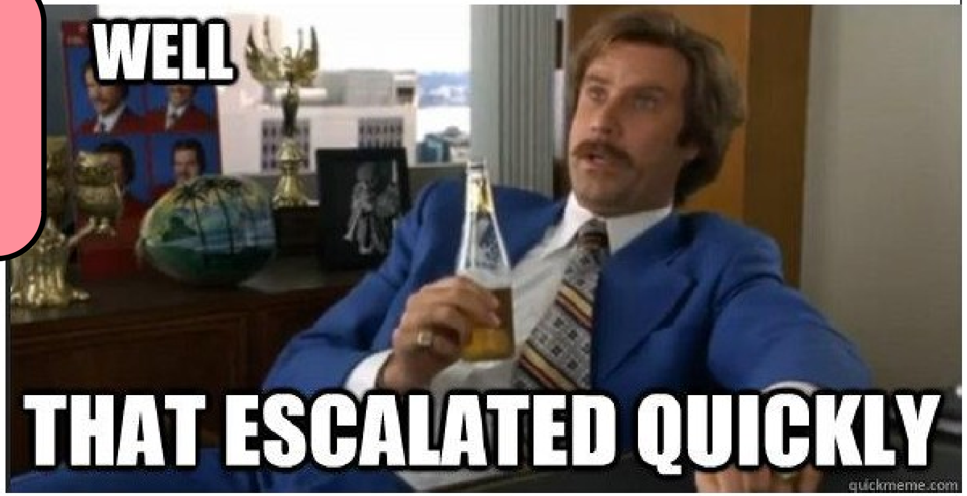
- Extends M into M_spec
- Property over paires

Key concepts : $M \models \varphi$

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

binary-level

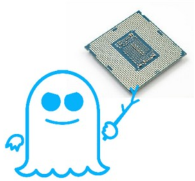
Semantics	Paths
Sequential semantics	**14**
Speculative semantics (Spectre-STL)	**37M**

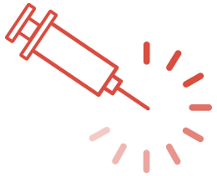


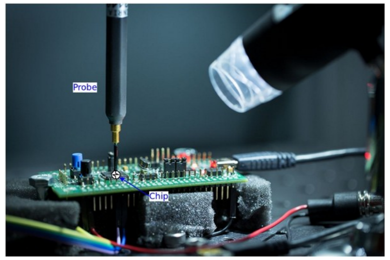
- Security scenarios
  - Vulnerability analysis and automated exploit generation
  - Side channel attacks
  - Speculative side channel attacks
  - Physical fault injection
  - Bug priorisation

Can act on the execution

Context







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

Hardware attacks

Electromagnetic pulses | Power glitch | Clock glitch | Laser beam

Software-implemented hardware attacks

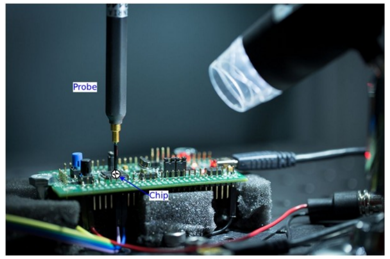
Faultline | DVFS | Rowhammer

Micro-architectural attacks

Race condition | Load Value Injection | Spectre

Man-At-The-End attacks

FOCUS: Fault injection
-- [ESOP 23, POPL 24, PLDI 24]



BINSEC

- Waht about advanced attackers ?

- Recent work :
  - support for attacker model
  - Fault injection-like capabilities

WooKey bootloader

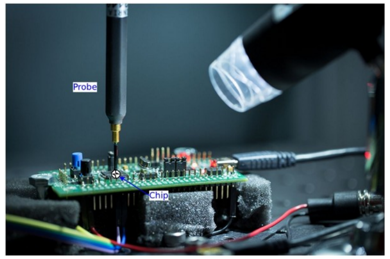
1. Find known attacks
2. Evaluate countermeasures from prior work
3. Find previously unreported attack path
4. Propose and check mitigation

- Goal

- Help security evaluators
- Help mitigation designers
-

FOCUS: Fault injection

-- [ESOP 23, POPL 24, PLDI 24]



BINSEC

- Waht about advanced attackers ?
- Recent work :
  - support for attacker model
  - Fault injection-like capabilities

WooKey bootloader

1. Find known attacks
2. ... m prior work
3. ... tack path
4. ... d check mitigation

- Extends M into M_spec
- Property over paires

- Goal
- Help security evaluators
- Help mitigation designe
-

Key concepts : $M \models \varphi$

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

- Path explosion
- Dedicated optimizations

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)

- **Security scenarios**
  - **Vulnerability analysis and automated exploit generation**
  - **Side channel attacks**
  - **Speculative side channel attacks**
  - **Physical fault injection**
  - **Bug priorisation**

Looks for strong attacks

Bug Priorization / Criticity Evaluation [CAV 18, CAV 21, FMSD 22, POPL 24, PLDI 24]

- Too many bugs. Which ones are relevant ?
- Defender can focus on these ones

- From the attacker point of view
  - **replicability**
  - Level of control
  -

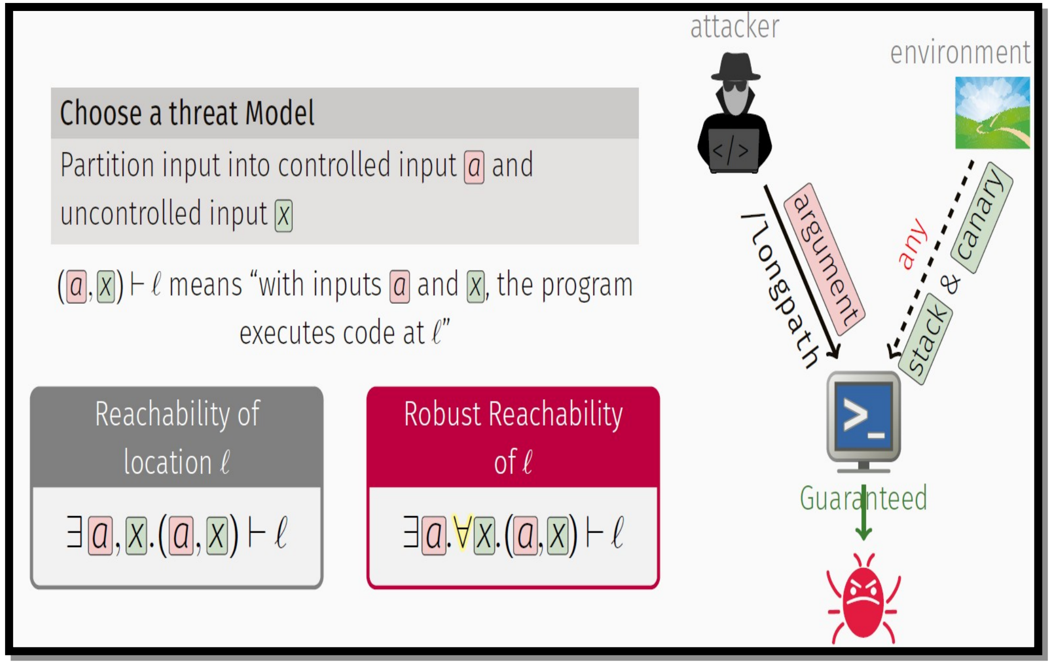
Bug Priorization / Criticity Evaluation [CAV 18, CAV 21, FMSD 22, POPL 24, PLDI 24]

- Too many bugs. Which ones are relevant ?
- Defender can focus on these ones

- From the attacker point of view
  - **replicability**
  - Level of control
  -

- Especially, bugs reported by standard program analysis may be poorly replicable
  - Ex : fault injection with very specific values
  - Ex : bugs depending on uninitialized memory
  - Ex : bugs depending on random values
  - ...
  -

Bug Priorization / Criticity Evaluation [CAV 18, CAV 21, FMSD 22, POPL 24, PLDI 24]



	PyAbd^O	Binsec/RSE	Binsec	Qemu
unknown	170	273	170	243
not vulnerable (0 input)	4414	4419	3921	4398
vulnerable (≥ 1 input)	226	118	719	169
≥ 0.0001%	226	118	–	–
≥ 0.01%	209	118	–	–
≥ 0.1%	173	118	–	–
≥ 1.0%	167	118	–	–
≥ 5.0%	166	118	–	–
≥ 10.0%	118	118	–	–
≥ 50.0%	118	118	–	–
100.0%	118	118	–	–

Key concepts : $M \models \varphi$

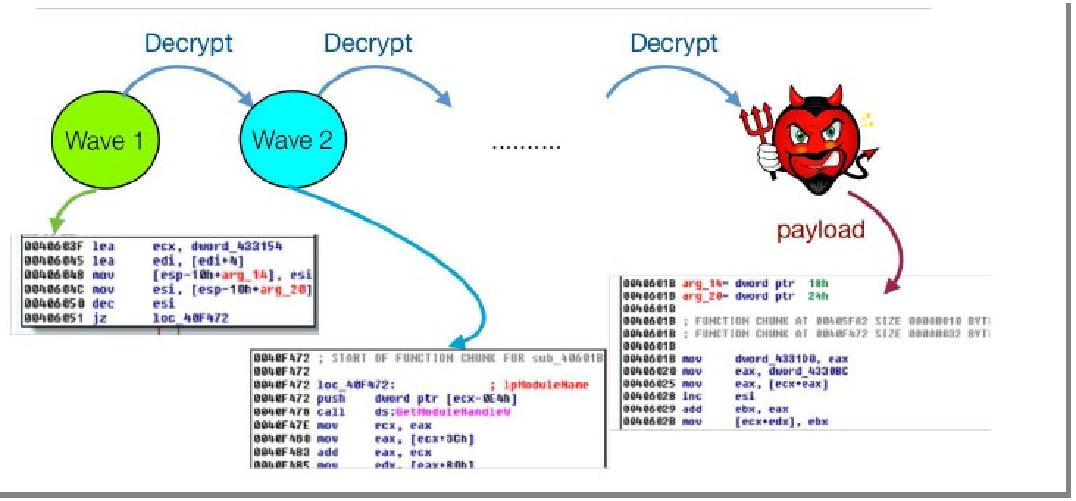
- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

- Modify the satisfaction relation

- **Security scenarios**
  - **Vulnerability analysis and automated exploit generation**
  - **Side channel attacks**
  - **Speculative side channel attacks**
  - **Physical fault injection**
  - **Bug priorisation**
  - **BONUS : reverse of malware**

Craft its own code

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	[...]

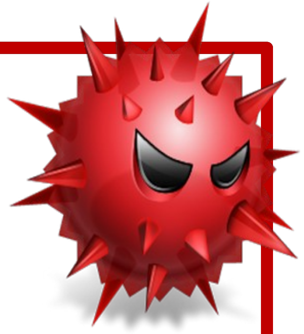
eg: $7y^2 - 1 \neq x^2$

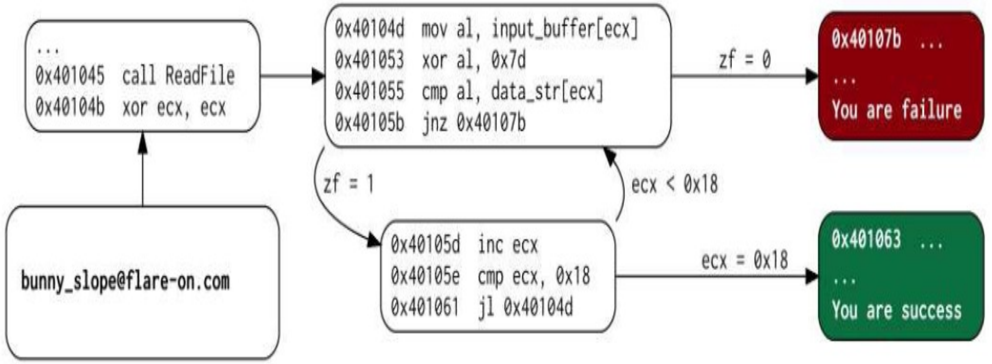
(for any value of x, y in modular arithmetic)

↧

```
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**
- **...**





FOCUS Reverse: THE XTUNNEL MALWARE

-- [BlackHat EU 2016, S&P 2017, ACSAC 2019, CCS 2022]

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

- **Introduction [The Sad Truth]**
- **Reasoning about programs [A New Hope]**
- **What about the attacker? [The Evil Returns]**
- **Some results [Hard Battle In Progress]**
- **Conclusion, Take away and Disgression**

STEP BACK

- **Taking the attacker into account in program analysis**

- Actions
- Observations
- Goal

- **New scientific challenges grounded in real security**
  - Fruitful – Useful – Fun

Key concepts : $M \models \varphi$

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

BINSEC

https://binsec.github.io/

FRANCE 2030

PEPR CYBERSECURITE

Secureval – Defmal – Rev

SUMMARY

Empower experts
Help build highly secure systems

- **Advanced automated reasoning as a game changer in cybersecurity**
  - Leverage and adapt best methods from safety-critical domains
  - Fruitful !
  - Beware of scalability and learning curve

- **Yet, security is not safety**
  - the attacker must be taken into account
  - field in progress

- **Toward truly security-oriented program analysis !**

- Actions
- Observations
- Goal

https://binsec.github.io/

THANK YOU