Cybersecurity meets Automated reasoning

Sébastien Bardin
CEA LIST

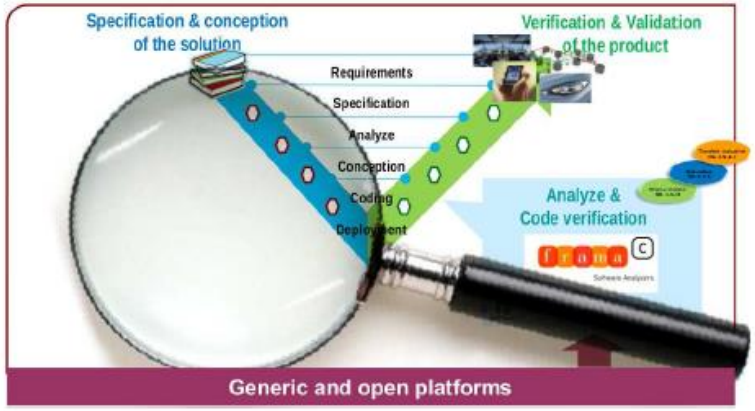
WWW.FORUM-FIC.COM

About my lab @ CEA

CEA LIST, Software Safety & Security Lab

- Located in Saclay, France

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



frama C Software Analyzers

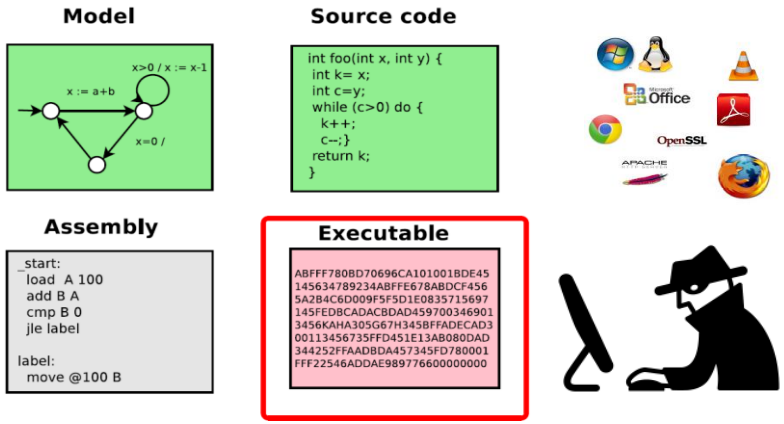
GATeL

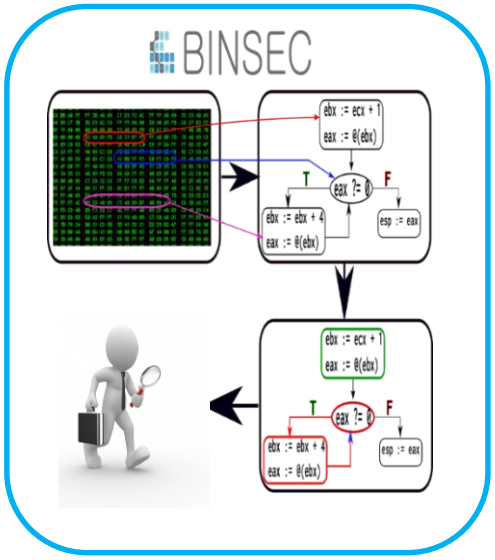
PathCrawler

POLAR SSL

me, myself and I

ADAPT FORMAL METHODS
TO BINARY-LEVEL SECURITY ANALYSIS





Malware comprehension

Protection evaluation

Vulnerability analysis

THE TALK IN A NUTSHELL

- **Securing software and systems is extremely hard**
  - « Black hat » hackers take advantage of system flaws at various levels
- **Meanwhile, powerful advanced code analysis techniques for safety**

➔**Advanced automated code reasoning as a game changer in cybersecurity?**
- Principles, achievements, strengths & limits, future trends

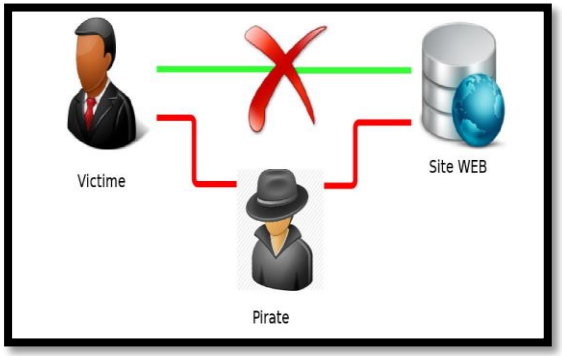
OUTLINE

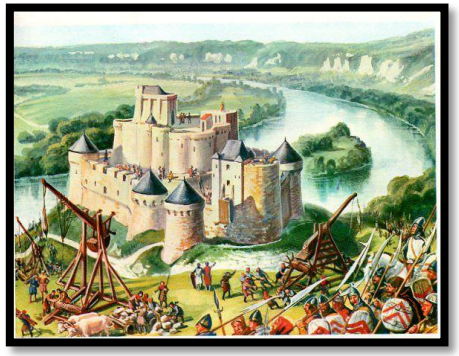
- Context
- Reasoning about software
- What about security?
- Limitations and future trends

WE ARE UNDER ATTACKS!

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



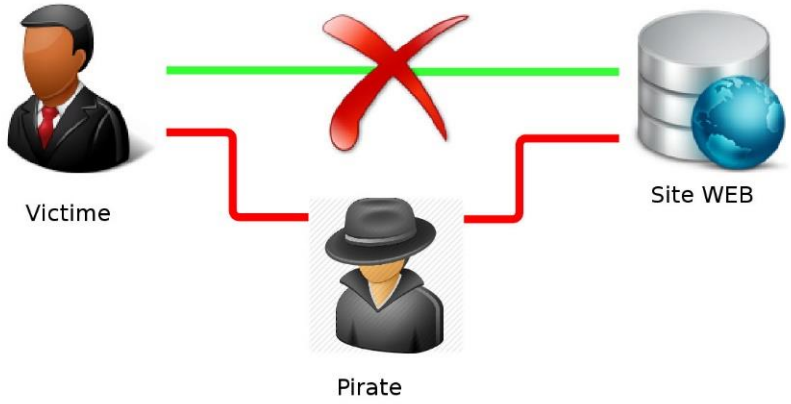


CLASSIFICATION OF ATTACKS

MITM: Man-In-The-Middle

Attacker is on the network

- **Observe messages**
- **Forge messages**

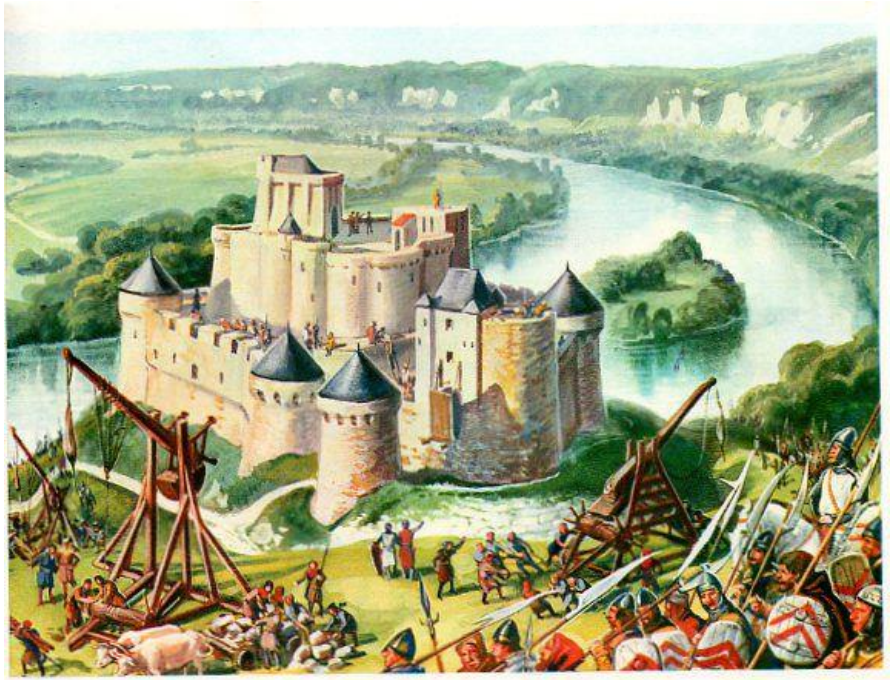


CLASSIFICATION OF ATTACKS

« Man-Beyond-The-Door »

Attacker has limited access

- **Try to escalate**
- **Forge specially crafted files/queries**

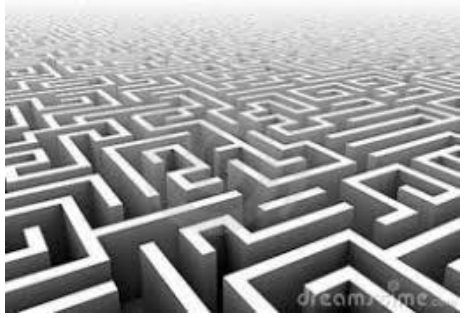


CLASSIFICATION OF ATTACKS

MATE: Man-At-The-End

Attacker is *on the computer*

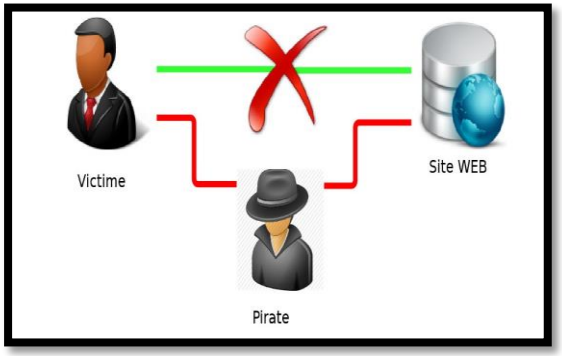
- **R/W the code**
- **Execute step by step**
- **Patch on-the-fly**

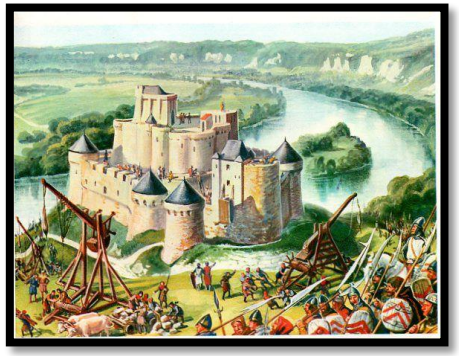


THE SECURITY GAME

- **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

Easy: remove bugs!!





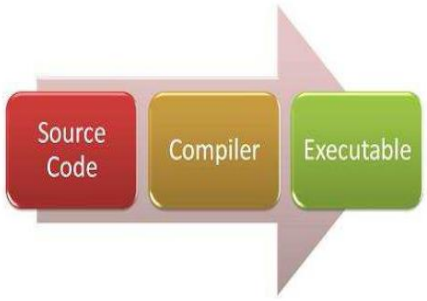
WHY BUGS?

Programs are the most complicated artifacts ever created by humans

- Huge sizes
- Many different layers & APIs: protocoles, OS, hardware, …
- Third-party components
- Myriads of languages
- Compilers
- Many lurking corner cases
- …

EXAMPLE: LANGUAGE CORNER CASES

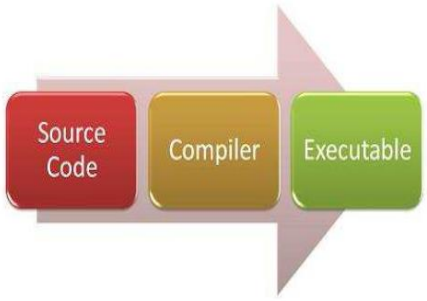
```c
#include "stdio.h"

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

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



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

EXAMPLE: LANGUAGE CORNER CASES

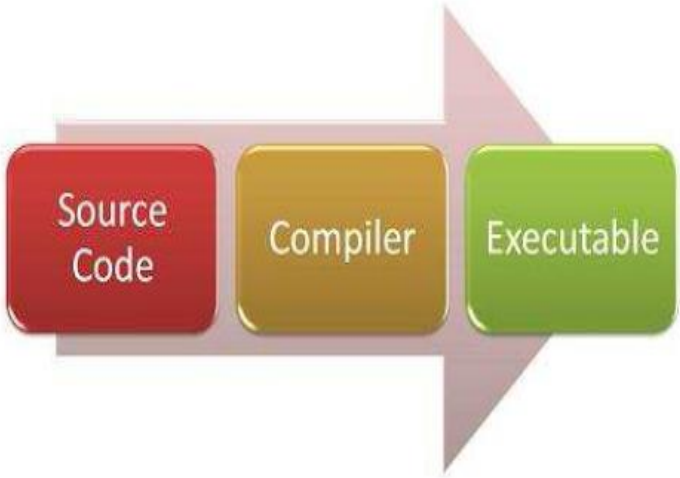
```c
#include "stdio.h"

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

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



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

Both are right!

EXAMPLE: COMPILER BUG (?)



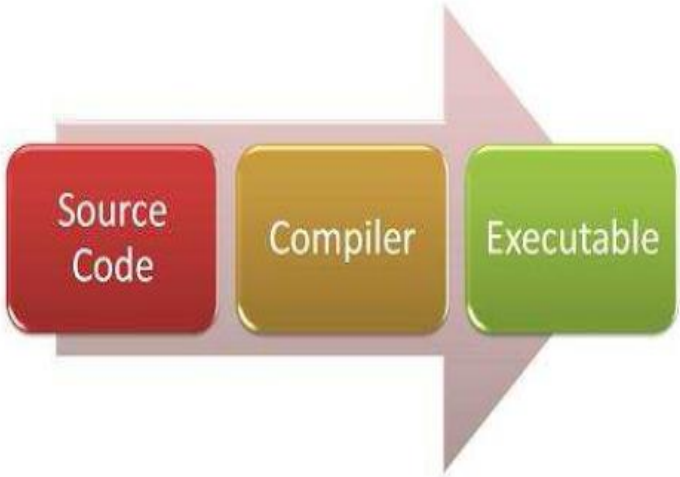
- Optimizing compilers may remove dead code
- pwd never accessed after memset
- Thus can be safely removed
- And allows the password to stay longer in memory

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

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

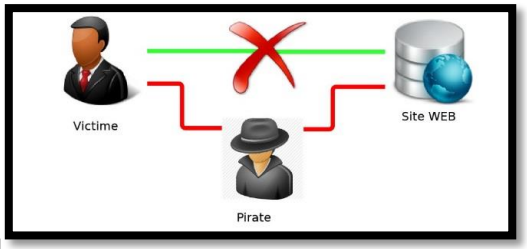
Security bug introduced by a non-buggy compiler

```
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**

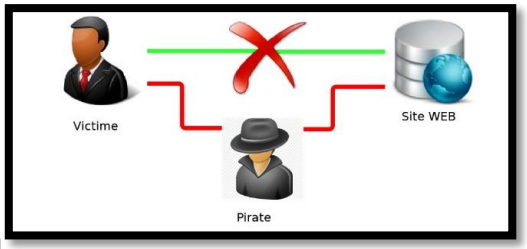
THE (SAD) STATE OF WAR



- **In a few situations, the defender has a clear advantage**
  - The miracle of « provable crypto »
  - Defender can reveal its method, no efficient way to break it (if well implemented)

- **In most situations: advantage to attacker and cat-and-mouse game**
  - **Defender tries to be one step ahead**
  - **Raise the bar enough**

THE (SAD) STATE OF WAR



- **In a few situations, the defender has a clear advantage**
  - The miracle of « provable crypto »
  - Defender can reveal its method, no efficient way to break it (if well implemented)

- **In most situations: advantage to attacker and cat-and-mouse game**
  - **Defender tries to be one step ahead**
  - **Raise the bar enough**

- **Most attacks come from implementation bugs**

- **Bugs are inevitable**

OUR VIEW

- Most attacks come from implementation bugs
  - ➔ Automated Program Analysis
- Bugs are inevitable ➔ Are they?

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

OUTLINE

- Context
- Reasoning about software
- What about security?
- Limitations and future trends

BY THE WAY

SOFTWARE IS BUGGY, with consequences

WHITE PAPERS

Software Fail Watch: 5th Edition

White Paper

The Software Fail Watch is an analysis of software failures found in a year's worth of English language news articles. The result is an extraordinary reminder of the role software plays in our daily lives, the necessity of software testing in every industry, and the far-reaching impacts of its failure.

The 5th Edition of the Software Fail Watch identified 606 recorded software failures, impacting half of the world's population (3.7 billion people), $1.7 trillion in assets, and 314 companies. And this is just scratching the surface—there are far more software defects in the world than we will likely ever know about.

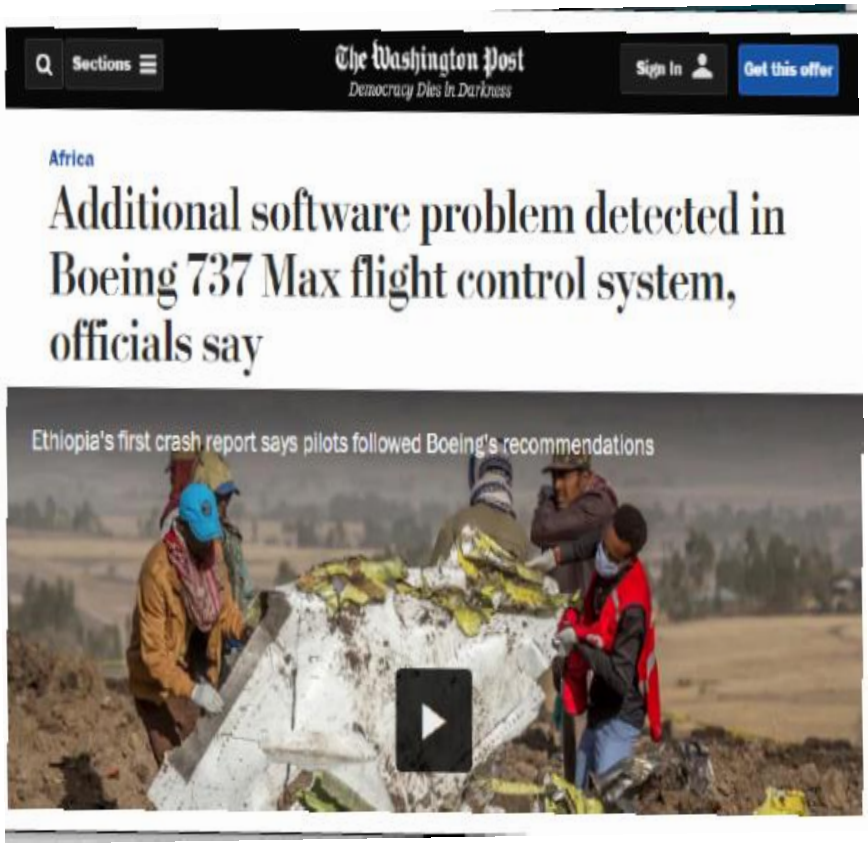
LOSSES FROM SOFTWARE FAILURES (USD)

1,715,430,778,504

ONETRILLIONSEVENHUNDREDFIFTEENBILLIONFOURHUNDREDTHIRTYMILLIONSEVENHUNDREDSEVENTY-EIGHTTHOUSANDFIVEHUNDREDFOUR

Download the report for a detailed analysis of the year's software failures, including:

Tricentis



BUT WE KNOW HOW TO

How does one develop high-quality software?

- Correct and complete specifications/design
- Good software development process
- Testing

BUT WE KNOW HOW TO

How does one develop high-quality software?

- Correct and complete specifications/design
- Good software development process
- Testing

« Testing can only find bugs,
not prove their absence »
-- E. Dijkstra

BUT WE KNOW HOW TO

How does one develop high-quality software?

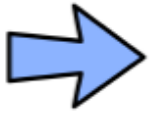
- Correct and complete specifications/design
- Good software development process
- Testing
- Formal verification

Dream of absolute,
« mathematical » guarantee

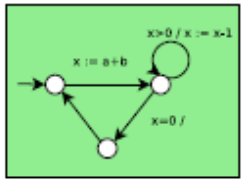
THEN COMES FORMAL METHODS

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

Success in safety-critical

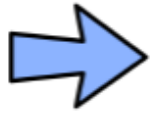


Model



Source code

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



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

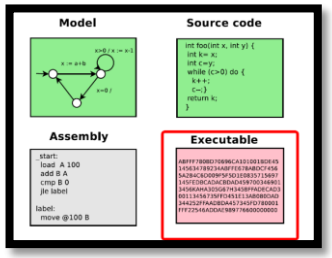
A GLIMPSE UNDER THE HOOD

ABOUT FORMAL METHODS

Key concepts : $M \models \varphi$

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

A GLIMPSE UNDER THE HOOD



A set of relevant behaviours
- **Reachable states**
- **Traces (finite or infinite)**
- **Execution Tree**
- **…**

- Mathematically defined
- Automatically computed

$\subset$

The HARD part

A set of good behaviours
- **Reachable states**
- **Traces (finite or infinite)**
- **Execution Tree**
- **…**

- Mathematically defined
- User defined or implicit

Specification?

- Mathematically defined!
- Given by user, or implicit

A set of good behaviours
- Reachable states
- Traces (finite or infinite)
- Execution Tree
- …

- Clearly specified
- Logic, automata, etc.

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

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

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

Specification?

A set of good behaviours
- Reachable states
- Traces (finite or infinite)
- Execution Tree
- …

- Clearly specified
- Logic, automata, etc.

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

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

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

Did you see the bug?

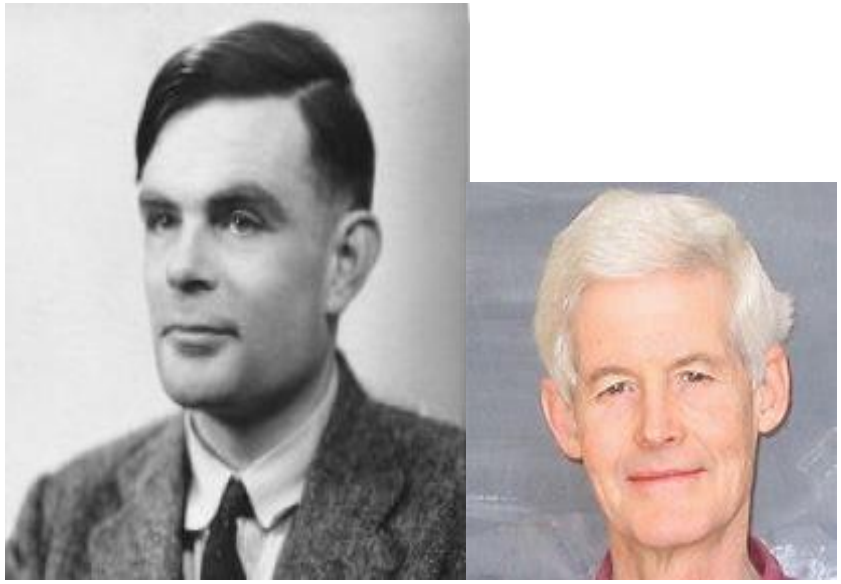
WAIT!!!

The Verification problem is either

- **undecidable, (1936)**
- **or intractable (1969)**

They didn't know […], so they did it

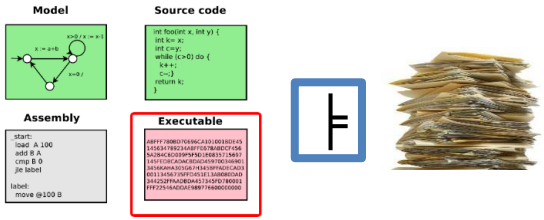
The Verification problem is either
- **undecidable, (1936)**
- **or intractable (1969)**



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

They didn’t know […], so they did it

KEY: WHICH GUARANTEES



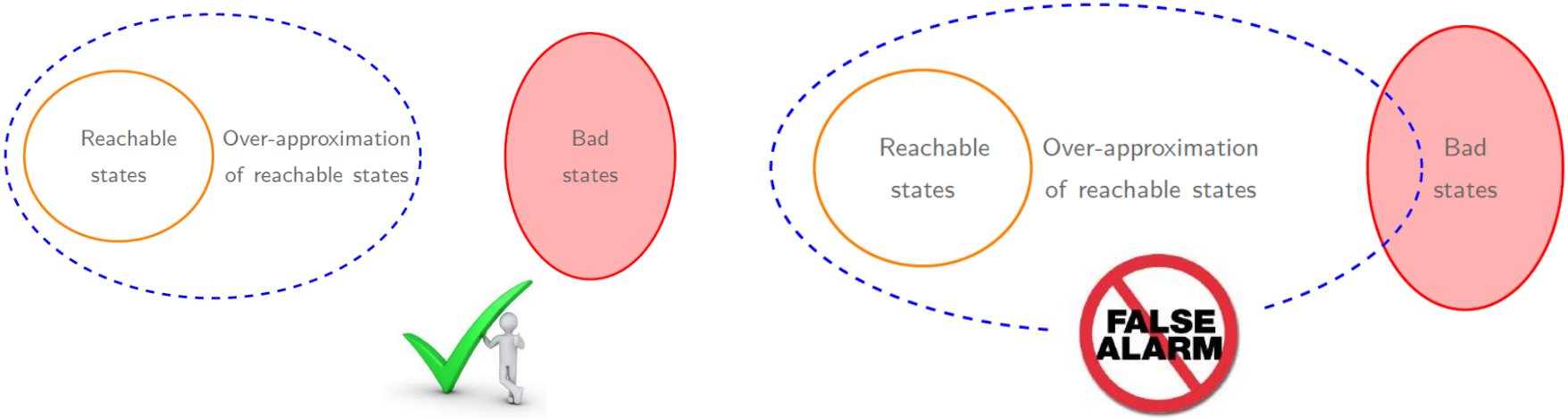
Cannot have an analysis that
- Terminates
- Is perfectly precise, answers YES or NO

On all programs

Answers
- Forget perfect precision → YES or MAYBE
- Or forget termination → YES or NO or LOOP
- Focus only on « interesting » programs
- Put a human in the loop

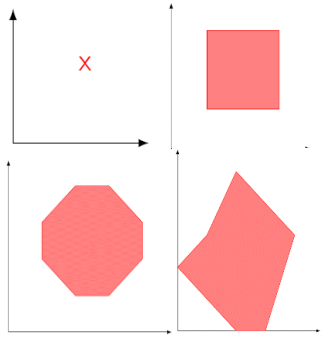
COCORICO EXAMPLE

<abstraction> ABSTRACT INTERPRETATION



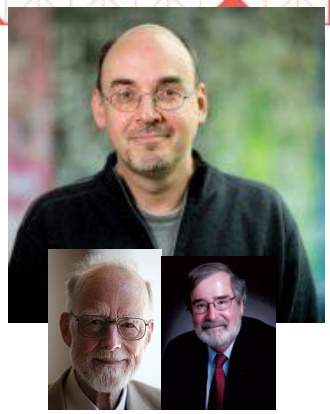
Compute an over-approximated semantic of the code

- Contain all possible behaviors
- Possibly more ➔ false alarms
- Tradeoff precision – efficiency



EXAMPLE 2

<reduction to logic> SYMBOLIC EXECUTION



```
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

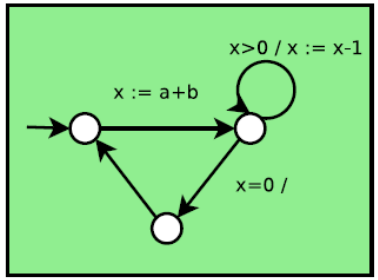
Ok, it looks complicated, but in the end …

OUTLINE

- Context
- Reasoning about software
- What about security?
- Limitations and future trends

MOVING TO SECURITY

Model



Source code

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

Assembly

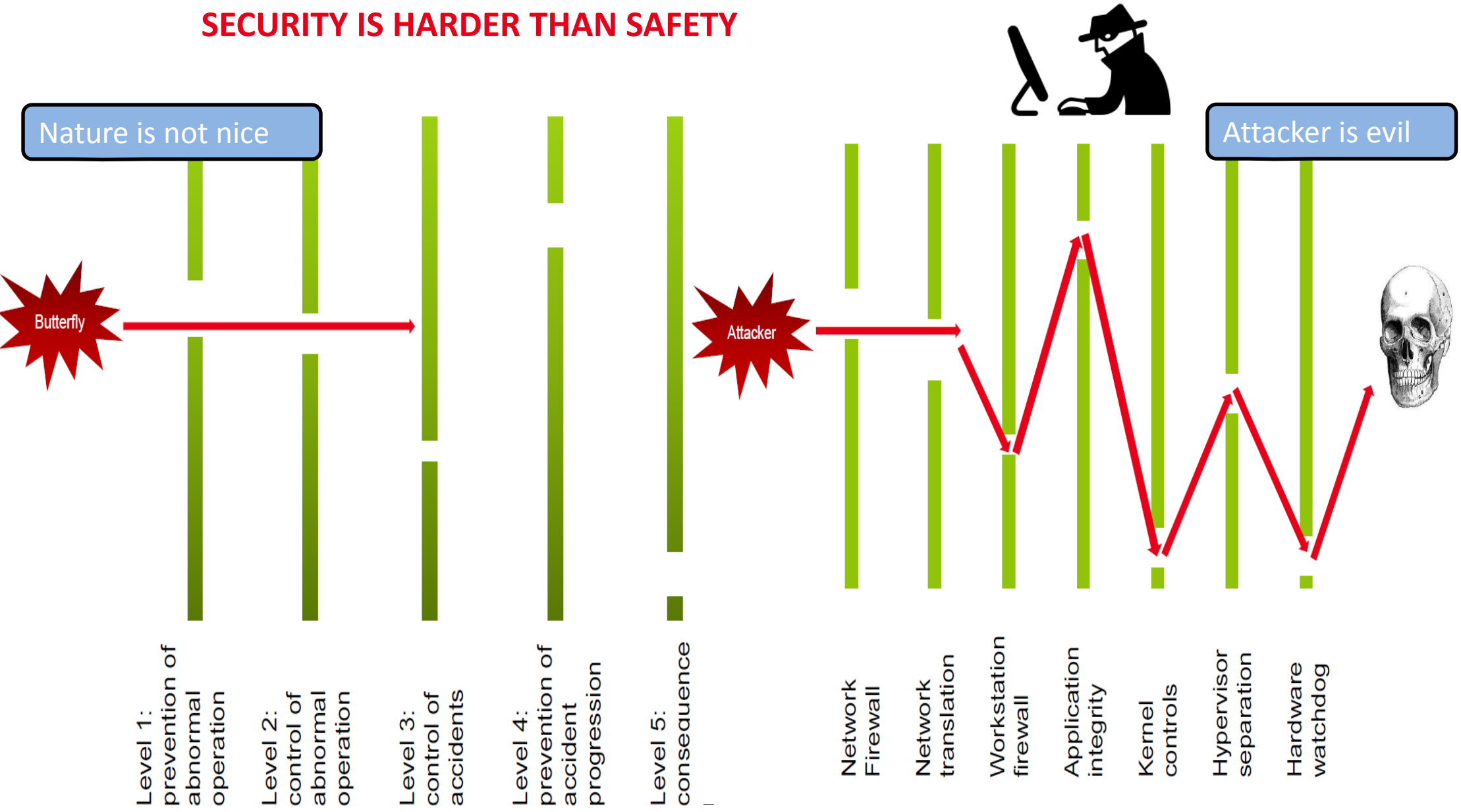
```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

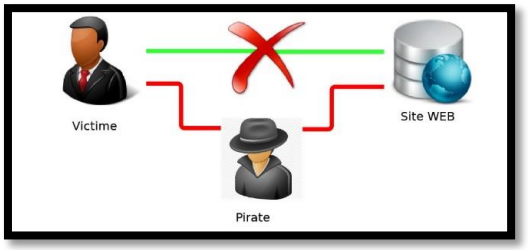
```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

One challenge among others

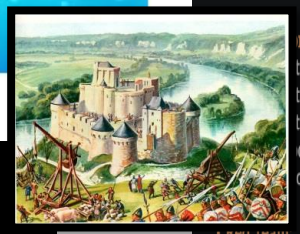
SECURITY IS HARDER THAN SAFETY

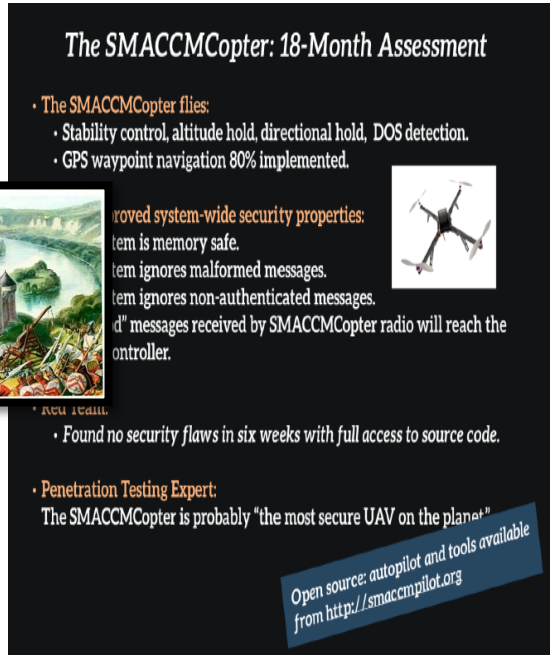


Still, another dream comes true



TLS 1.3





DEFENSE

- Security proven by design
- Security proven after development (harder)
- Intensive bug hunting + automated patching
- Targeted protection insertion (runtime monitoring)

Early attempts @ MS

Driver verification

- drivers are priviledged
- drivers are tricky

Automatic verif. of API conformance

Smart fuzzing

- Parsers are error prone input processing parts of software

- **Use symbolic execution for bug finding**
- **All security-critical products**

Cybersecure Drone

Formally hardened UAV

- **Developped from scratch**
- **Based on sel4 certified kernel**
- **Memory safety, task separation, input sanitization, etc.**

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

The SMACCMCopter: 18-Month Assessment

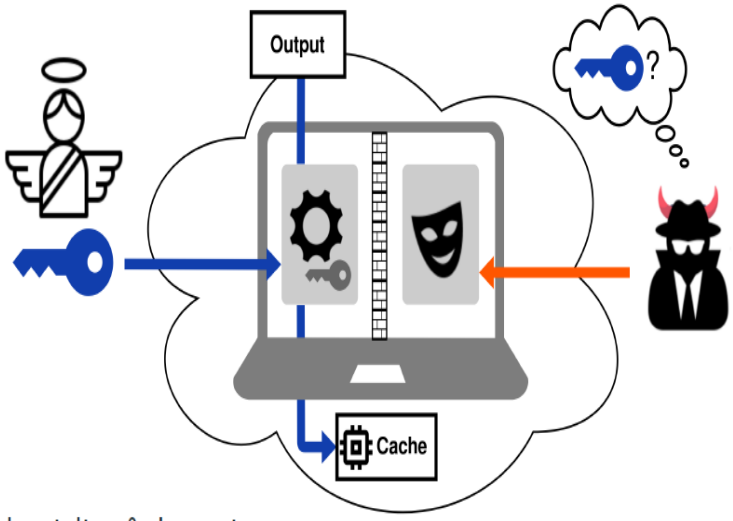
- The SMACCMCopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.

- 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

CASE-STUDY: SECURING CRYPTO-PRIMITIVES

-- [Security & Privacy 2020, Lesly-Ann Daniel]



- Confidentiality & Integrity
- Erase secrets from memory
- Spectre attacks
- Constant-time crypto

		#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	-

	#I	#I/s	#Q	Time	⌛	✓	✗
SC	252k	3.9	170k	65473	15	282	41
RelSE	320k	5.4	97k	59316	14	283	42
BINSEC/REL	22.8M	3861	3.9k	5895	0	296	42

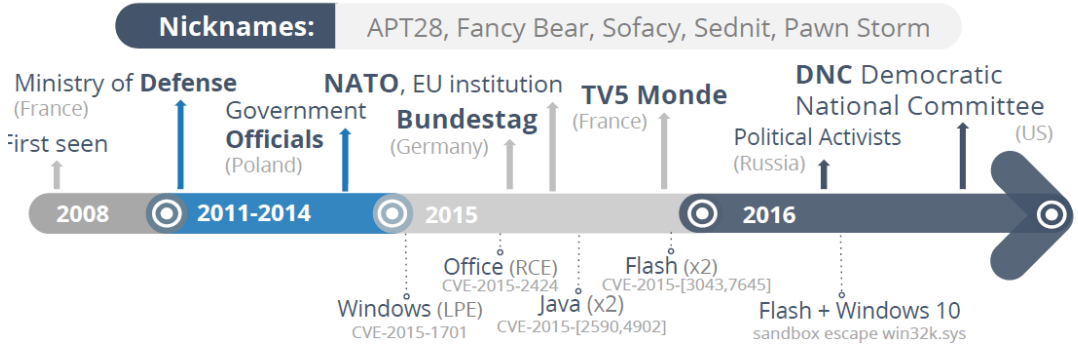
ATTACK

- Intensive bug hunting
- Identify weak parts of codes
- Help identify and bypass protections
- Transform poc exploits into weaponized exploits

REVERSE & DEOBFUSCATION

CASE-STUDY: THE XTUNNEL MALWARE

-- [BlackHat EU 2016, Security & Privacy 2017]



Two heavily obfuscated samples

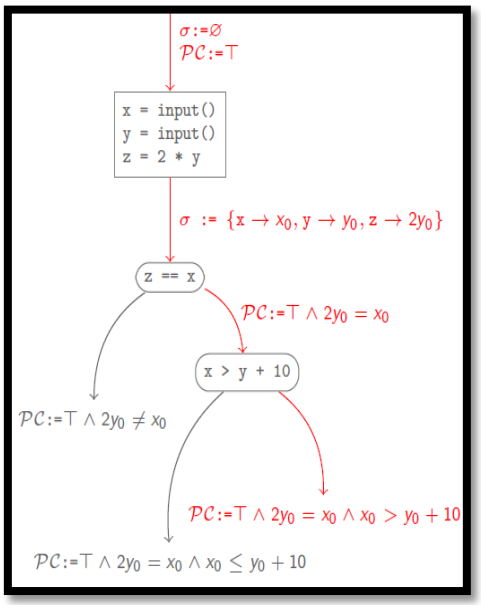
- **Many opaque predicates**

Goal: detect & remove protections

- Identify 40% of code as spurious
- Fully automatic, < 3h

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

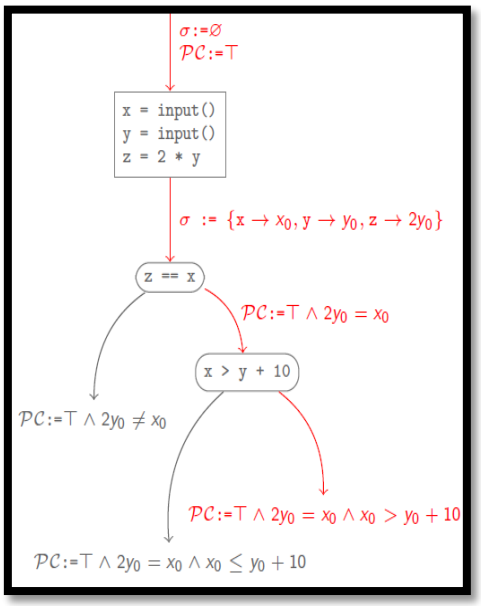
VULNERABILITIES



Find (new) CVEs

- Intensive bug hunting

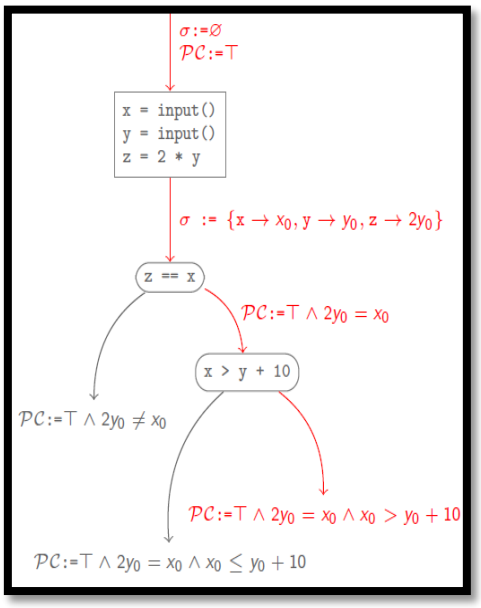
Bugs? Who cares?



Find (new) CVEs

- Intensive bug hunting

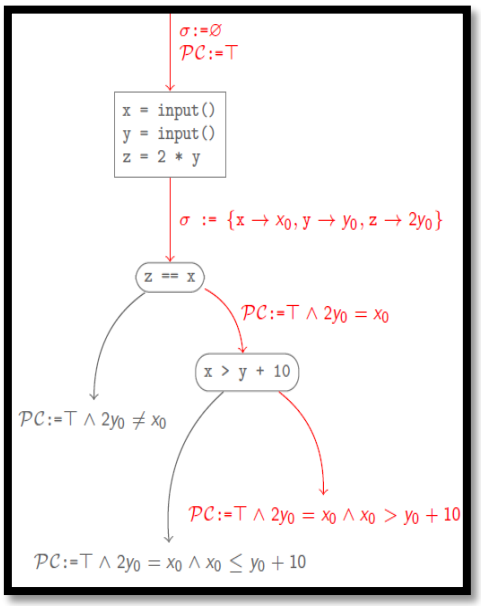
Bugs? Who cares?



Find (new) CVEs

- Intensive bug hunting
- Can target critical bugs

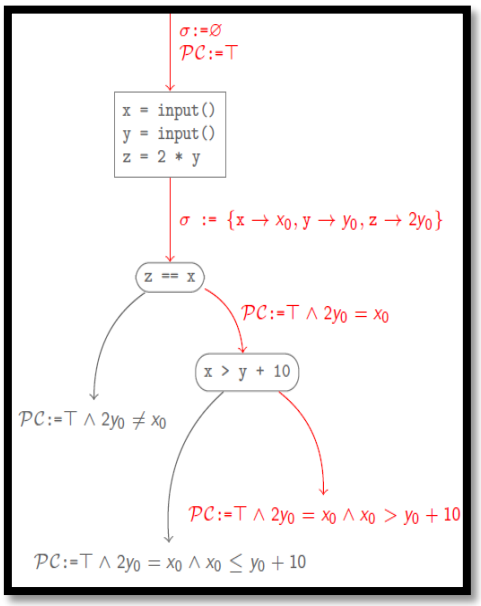
Bugs? Who cares?



Find (new) CVEs

- Intensive bug hunting
- Can target critical bugs
- Directly create simple exploits

Bugs? Who cares?



Find (new) CVEs

- Intensive bug hunting
- Can target critical bugs
- Directly create simple exploits
- Automatic exploit hardening

PUTTING ALL TOGETHER

The future is today

CYBER-REASONING SYSTEMS

DARPA 2016 challenges

- **Standard capture the flag competition**

The future is today

CYBER-REASONING SYSTEMS

DARPA 2016 challenges

- **Standard capture the flag competition**
- **Played by autonomous programs**

The future is today

CYBER-REASONING SYSTEMS

DARPA 2016 challenges

- **Standard capture the flag competition**
- **Played by autonomous programs**
  - **Find errors & exploit them**
  - **Defend their own system**

OUTLINE

- Context
- Reasoning about software
- What about security?
- Limitations and future trends

TAKE AWAY

YOU SHALL NOT IGNORE FORMAL METHODS!

They help secure the system

They help find flaws and devise attacks

STATE OF WAR

DEFENDER

- Remove whole classes of attacks
- Require sound analysis
- Works on source code
- Source could be tailored to analysis

ATTACKER

- Explore large sets of input
- Unsound analysis is fine
- Works on binary code (?)
- Code should be tailored vs analysis

STATE OF WAR

DEFENDER

- Remove whole classes of attacks
- Require sound analysis
- Works on source code
- Source could be tailored to analysis

ATTACKER

- Explore large sets of input
- Unsound analysis is fine
- Works on binary code (?)
- Code should be tailored vs analysis

Automated code analysis as an equalizer?

CURRENT LIMITS

- **Easier to use, yet still a learning curve, often require strong expertise**
- **Mastery can make a huge difference**
- **Scalability and applicability issues**

TRENDS

Scalability

- **Pure deductive improvements**
- **Combine with humans**
- **unsound methods**
- **Targeted reasoning (1day)**
- **…**

TRENDS

Scalability

- Pure deductive improvements
- Combine with humans
- unsound methods
- Targeted reasoning (1day)
- …

Usability / Expertise

- Choose right approach
- Tune parameters
- Prioritize targets
- Filter results
- …

TRENDS

Scalability

- Pure deductive improvements
- Combine with humans
- unsound methods
- Targeted reasoning (1day)
- …

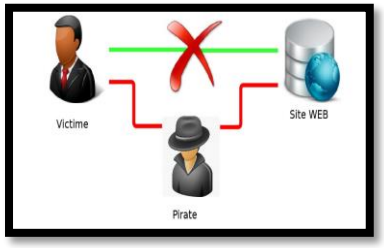
ML could be good for any of that

Usability / Expertise

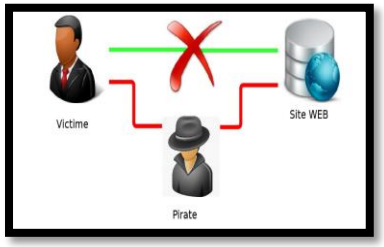
- Choose right approach
- Tune parameters
- Prioritize targets
- Filter results
- …

ML could be good for any of that

TRENDS

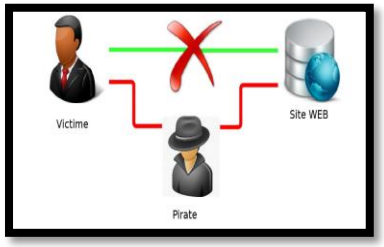


TRENDS



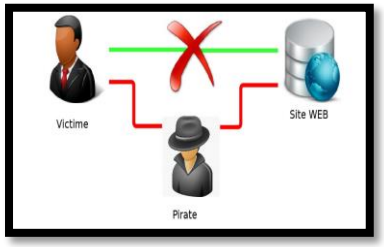
Deduction (logic)

TRENDS



Induction (data)

TRENDS



Deduction + Induction ?

CONCLUSION

- **Advanced automated reasoning as a game changer in cybersecurity**
  - Leverage and adapt best methods from safety-critical domains
  - Still, several challenges ahead
- **Can be used for both defense and attack: cannot ignore them**
- **Easier to use, yet still a learning curve, often require strong expertise**
  - Mastery can make a big difference
- **ML could be a strong enabler for advanced automated reasoning**
- **CEA LIST is a major actor, with strong international visibility**
  - Lead of the SPARTA « European network of excellence in cybersecurity »
  - Collaborations with USC, NASA, SRI, Franhaufer, etc.