# BINARY-LEVEL SECURITY: SEMANTIC ANALYSIS TO THE RESCUE

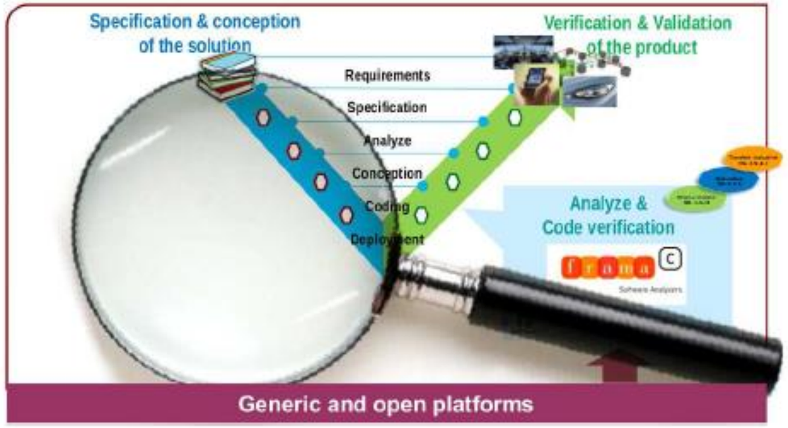
Sébastien Bardin (CEA LIST)

Joint work with
Richard Bonichon, Robin David, Adel Djoudi & many other people

# ABOUT MY LAB @CEA

## CEA LIST, Software Safety & Security Lab

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




frama C Software Analyzers

GATeL

PathCrawler

UNISIM

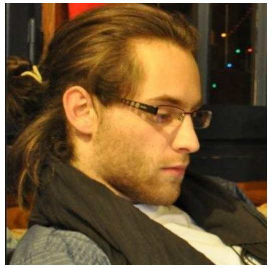
POLAR SSL

# IN A NUTSHELL

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

- **Standard techniques (dynamic, syntactic) not enough**

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



- ***A tour on how formal methods can help***
  - *Explore and discover* -- *with Josselin Feist*
  - *Prove infeasibility or validity* -- *with Robin David*
  - *Simplify* (not covered here) -- *with Jonathan Salwan*

# OUTLINE

- Why binary-level analysis?
- Some background on source-level formal methods
- The hard journey from source to binary
- A few case-studies
- Conclusion

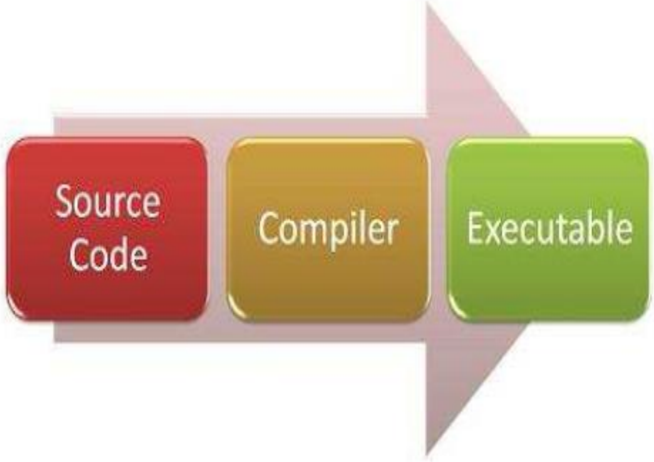
- **Focus mostly on Symbolic Execution**
- **Give hints for abstract Interpretation**

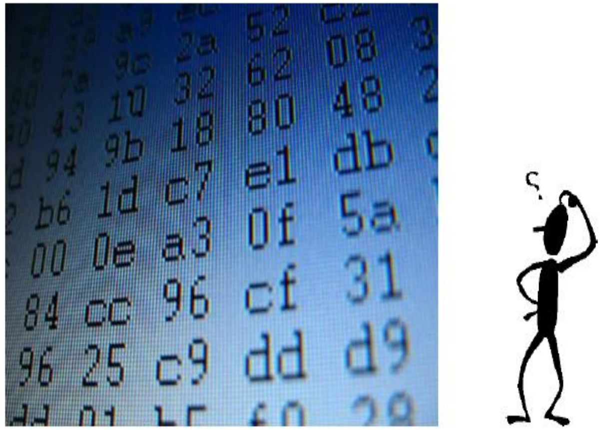
**Cover both**
- **vulnerability detection**
- **deobfuscation**

# OUTLINE

- Why binary-level analysis?
- Some background on source-level formal methods
- The hard journey from source to binary
- A few case-studies
- Conclusion

# BENEFITS

Is it Stuxnet ?

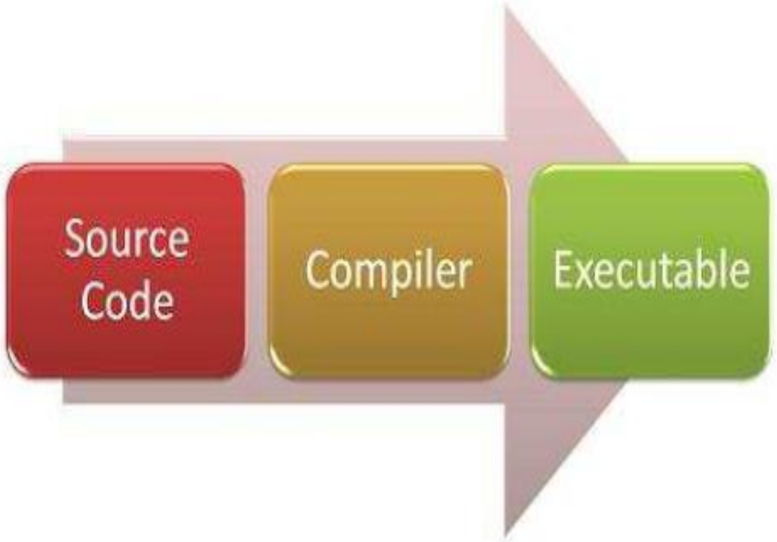
No source code

More precise analysis

Malware

What for: vulnerabilities, reverse (malware, legacy), protection evaluation, etc.

# 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

**Our goal here:**
- **Check the code after compilation**

# EXAMPLE: MALWARE COMPREHENSION

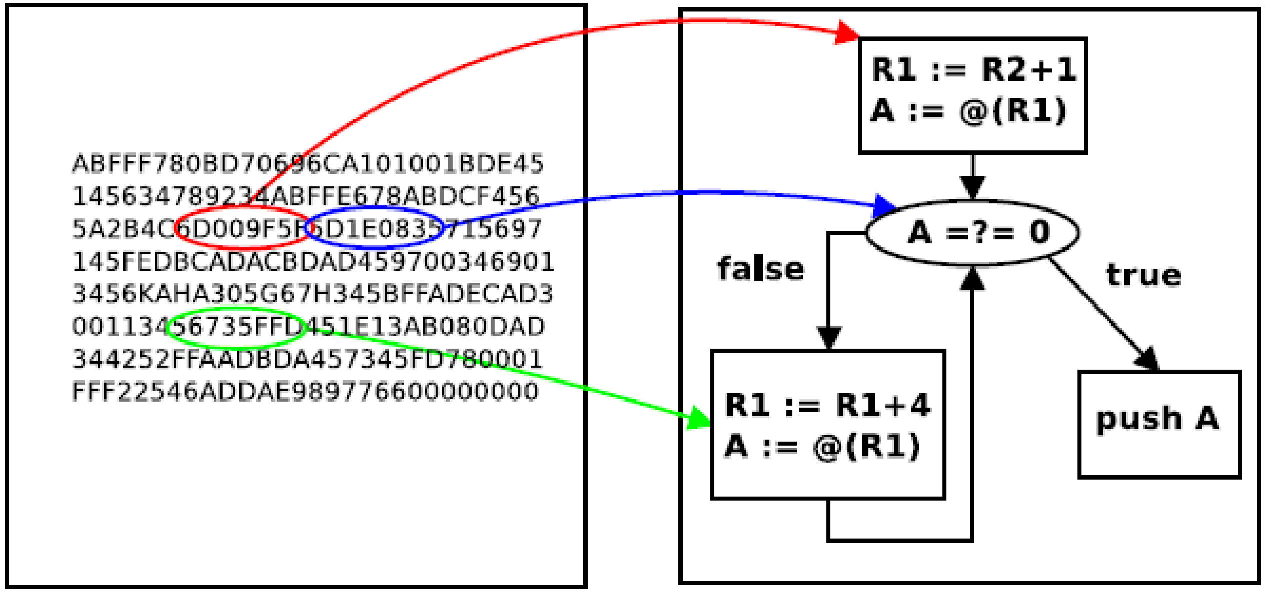
APT: highly sophisticated attacks
- **Targeted malware**
- **Written by experts**
- Attack: 0-days
- Defense: stealth, **obfuscation**
- **Sponsored by states or mafia**

The day after: **malware comprehension**
- understand what has been going on
- mitigate, fix and clean
- improve defense

**USA elections: DNC Hack**

**Highly challenging [obfuscation]**

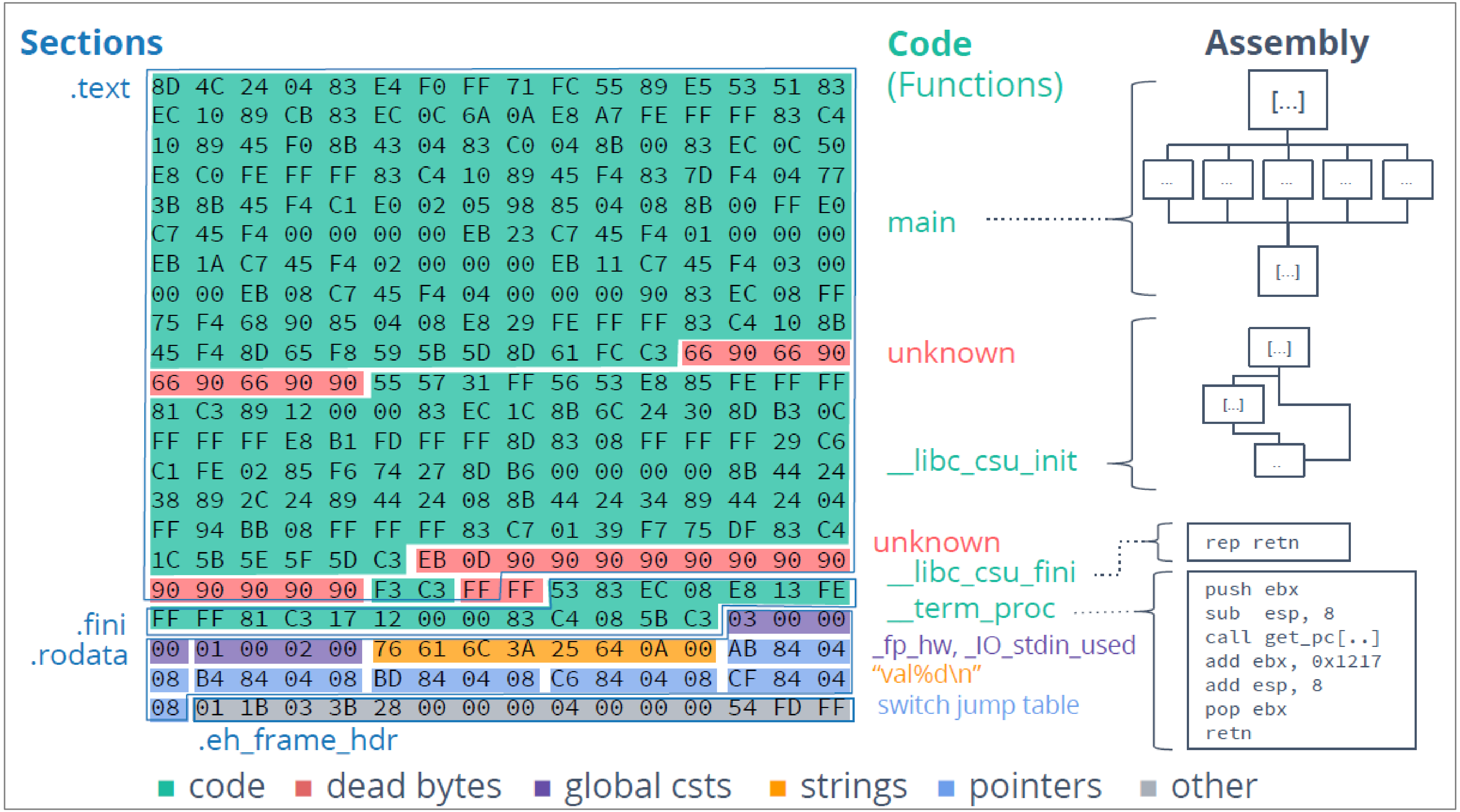
# CHALLENGE: CORRECT DISASSEMBLY



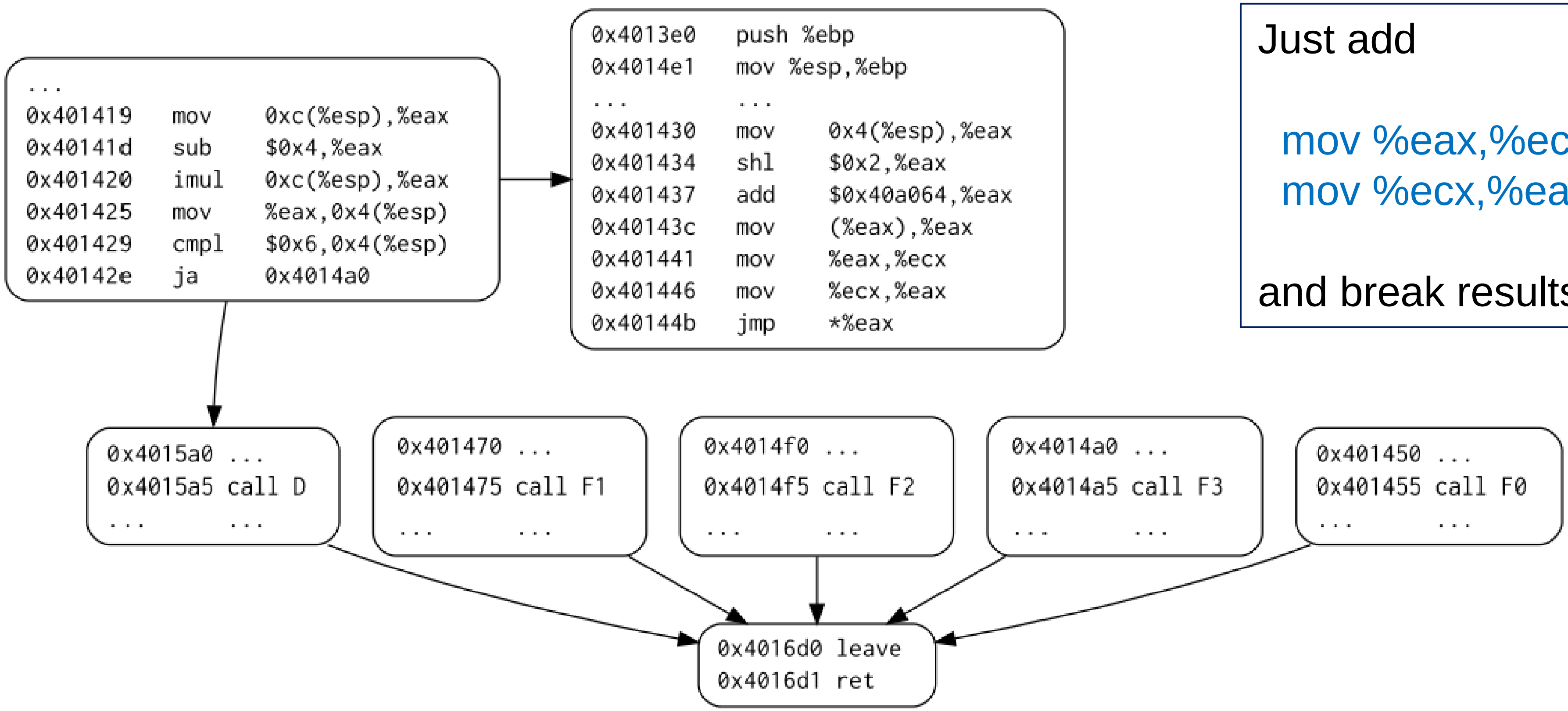

**Basic reverse problem**
- **aka model recovery**
- **aka CFG recovery**

# CAN BE TRICKY!

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



# STATE-OF-THE-ART TOOLS ARE NOT ENOUGH




With IDA

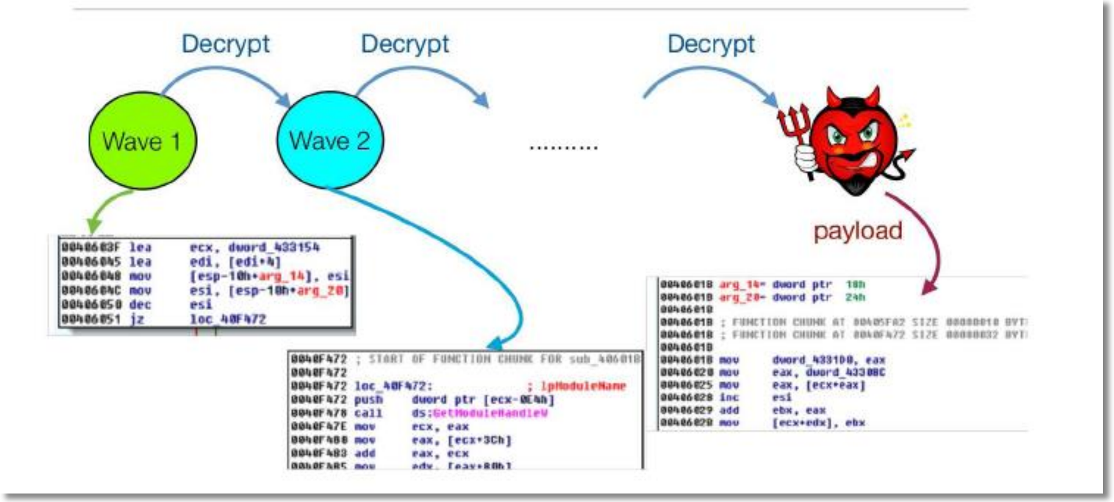
Just add

mov %eax,%ecx
mov %ecx,%eax

and break results

- **Static (syntactic): too fragile**
- **Dynamic: too incomplete**

# [See later] CAN BECOME A NIGHTMARE WHEN OBFUSCATED



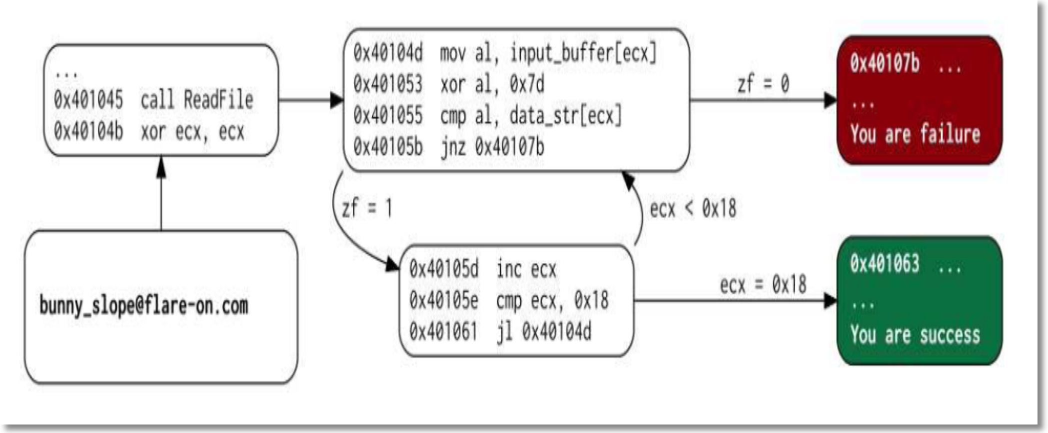

| address | instr |
|---|---|
| 80483d1 | call +5 |
| 80483d6 | pop edx |
| 80483d7 | add edx, 8 |
| 80483da | push edx |
| 80483db | ret |
| 80483dc | .byte{invalid} |
| 80483de | [...] |

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

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

↧

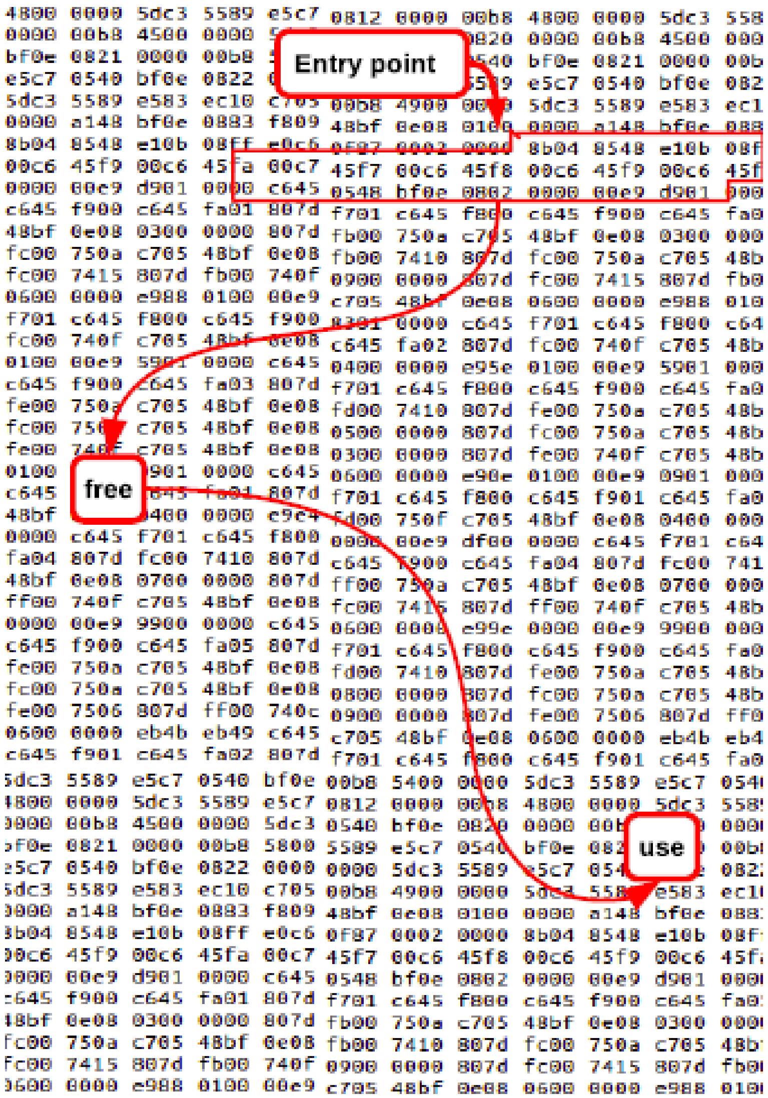
```
mov   eax, ds:X
mov   ecx, ds:Y
imul  ecx, ecx
imul  ecx, 7
sub   ecx, 1
imul  eax, eax
cmp   ecx, eax
jz    <dead_addr>
```



# EXAMPLE: VULNERABILITY DETECTION

**Find vulnerabilities before the bad guys**

- **On the whole program**
- **At binary-level**
- **Know only the entry point and program input format**



# EXAMPLE: VULNERABILITY DETECTION

## Use-after-free (UaF) – CWE-416

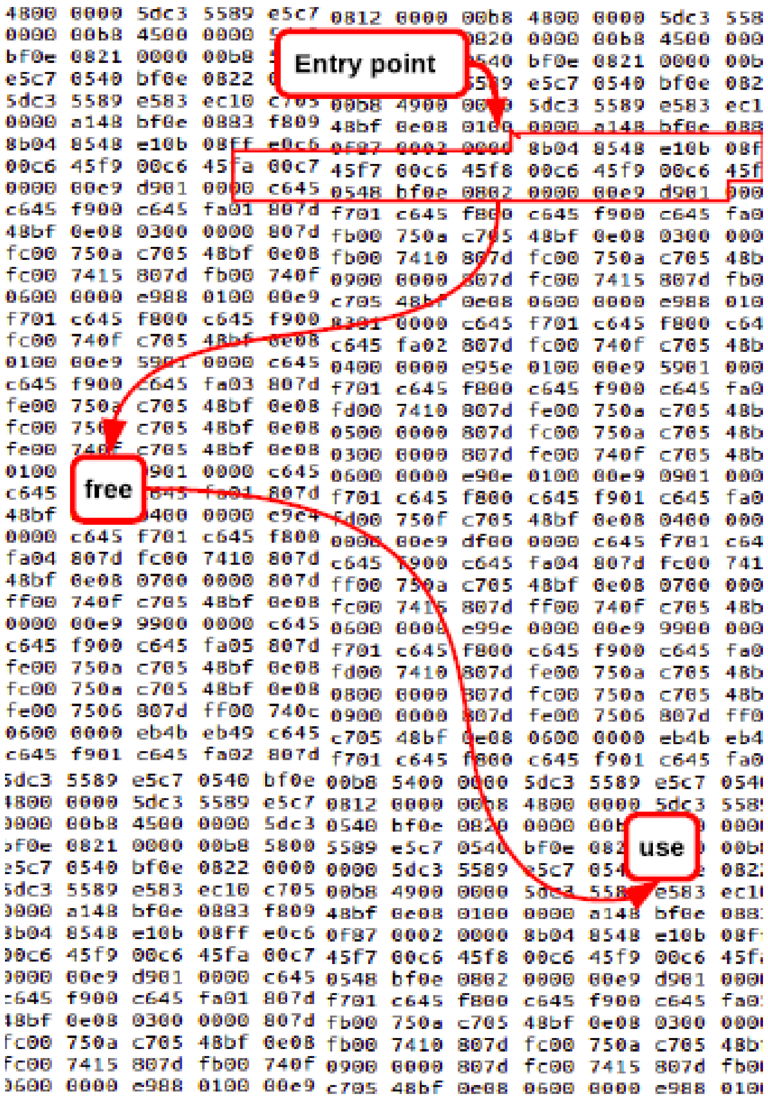
- *dangling pointer* on deallocated-then-reallocated memory
- may lead to arbitrary data/code read, write or execution
- standard vulnerability in C/C++ applications (e.g. web browsers) . firefox (CVE-2014-1512), chrome (CVE-2014-1713)

```
char *login , *passwords;
login=(char *) malloc (...);
[...]
free(login);   // login is now a dangling pointer
[...]
passwords=(char *) malloc (...);   // may re-allocate memory of *login
[...]
printf("%s\n", login);   // security threat : may print the passwords !
```

# CHALLENGE: In-depth exploration (example: use after free)

**Find a needle in the heap !**

- sequence of events, importance of aliasing
- strongly depend on implem of `malloc` and `free`

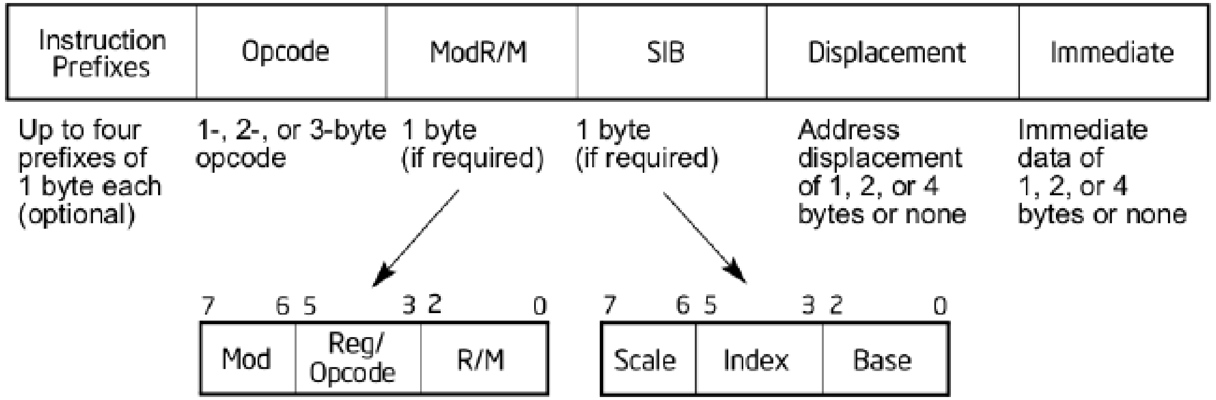




**Dynamic: not enough**

- **Too incomplete**

# BONUS: (MULTI-)ARCHITECTURE SUPPORT




Example of x86

- more than 1,000 instructions
  - . $\approx$ 400 basic
  - . + float, interrupts, mmx
- many side-effects
- error-prone decoding
  - . addressing modes, prefixes, ...

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| r8(/r) | | | AL | CL | DL | BL | AH | CH | DH | BH |
| r16(/r) | | | AX | CX | DX | BX | SP | BP | SI | DI |
| r32(/r) | | | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
| mm(/r) | | | MM0 | MM1 | MM2 | MM3 | MM4 | MM5 | MM6 | MM7 |
| xmm(/r) | | | XMM0 | XMM1 | XMM2 | XMM3 | XMM4 | XMM5 | XMM6 | XMM7 |
| sreg | | | ES | CS | SS | DS | FS | GS | res. | res. |
| eee | | | CR0 | invd | CR2 | CR3 | CR4 | invd | invd | invd |
| eee | | | DR0 | DR1 | DR2 | DR3 | DR4[1] | DR5[1] | DR6 | DR7 |
| (In decimal) /digit (Opcode) | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| (In binary) REG = | | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Effective Address | Mod | R/M | Value of ModR/M Byte (in Hex) | | | | | | | |
| [EAX] | 00 | 000 | 00 | 08 | 10 | 18 | 20 | 28 | 30 | 38 |
| [ECX] | | 001 | 01 | 09 | 11 | 19 | 21 | 29 | 31 | 39 |
| [EDX] | | 010 | 02 | 0A | 12 | 1A | 22 | 2A | 32 | 3A |
| [EBX] | | 011 | 03 | 0B | 13 | 1B | 23 | 2B | 33 | 3B |
| [sib] | | 100 | 04 | 0C | 14 | 1C | 24 | 2C | 34 | 3C |
| disp32 | | 101 | 05 | 0D | 15 | 1D | 25 | 2D | 35 | 3D |
| [ESI] | | 110 | 06 | 0E | 16 | 1E | 26 | 2E | 36 | 3E |
| [EDI] | | 111 | 07 | 0F | 17 | 1F | 27 | 2F | 37 | 3F |
| [EAX]+disp8 | 01 | 000 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 |
| [ECX]+disp8 | | 001 | 41 | 49 | 51 | 59 | 61 | 69 | 71 | 79 |
| [EDX]+disp8 | | 010 | 42 | 4A | 52 | 5A | 62 | 6A | 72 | 7A |
| [EBX]+disp8 | | 011 | 43 | 4B | 53 | 5B | 63 | 6B | 73 | 7B |
| [sib]+disp8 | | 100 | 44 | 4C | 54 | 5C | 64 | 6C | 74 | 7C |
| [EBP]+disp8 | | 101 | 45 | 4D | 55 | 5D | 65 | 6D | 75 | 7D |
| [ESI]+disp8 | | 110 | 46 | 4E | 56 | 5E | 66 | 6E | 76 | 7E |
| [EDI]+disp8 | | 111 | 47 | 4F | 57 | 5F | 67 | 6F | 77 | 7F |
| [EAX]+disp32 | 10 | 000 | 80 | 88 | 90 | 98 | A0 | A8 | B0 | B8 |
| [ECX]+disp32 | | 001 | 81 | 89 | 91 | 99 | A1 | A9 | B1 | B9 |
| [EDX]+disp32 | | 010 | 82 | 8A | 92 | 9A | A2 | AA | B2 | BA |
| [EBX]+disp32 | | 011 | 83 | 8B | 93 | 9B | A3 | AB | B3 | BB |
| [sib]+disp32 | | 100 | 84 | 8C | 94 | 9C | A4 | AC | B4 | BC |
| [EBP]+disp32 | | 101 | 85 | 8D | 95 | 9D | A5 | AD | B5 | BD |
| [ESI]+disp32 | | 110 | 86 | 8E | 96 | 9E | A6 | AE | B6 | BE |
| [EDI]+disp32 | | 111 | 87 | 8F | 97 | 9F | A7 | AF | B7 | BF |
| AL/AX/EAX/ST0/MM0/XMM0 | 11 | 000 | C0 | C8 | D0 | D8 | E0 | E8 | F0 | F8 |
| CL/CX/ECX/ST1/MM1/XMM1 | | 001 | C1 | C9 | D1 | D9 | E1 | E9 | F1 | F9 |
| DL/DX/EDX/ST2/MM2/XMM2 | | 010 | C2 | CA | D2 | DA | E2 | EA | F2 | FA |
| BL/BX/EBX/ST3/MM3/XMM3 | | 011 | C3 | CB | D3 | DB | E3 | EB | F3 | FB |
| AH/SP/ESP/ST4/MM4/XMM4 | | 100 | C4 | CC | D4 | DC | E4 | EC | F4 | FC |
| CH/BP/EBP/ST5/MM5/XMM5 | | 101 | C5 | CD | D5 | DD | E5 | ED | F5 | FD |

# THE SITUATION

- **Binary-level security analysis is necessary**
- **Binary-level security analysis is highly challenging (*)**
- **Standard tools are not enough – experts need better help!**

(*) i.e., more challenging than source code analysis

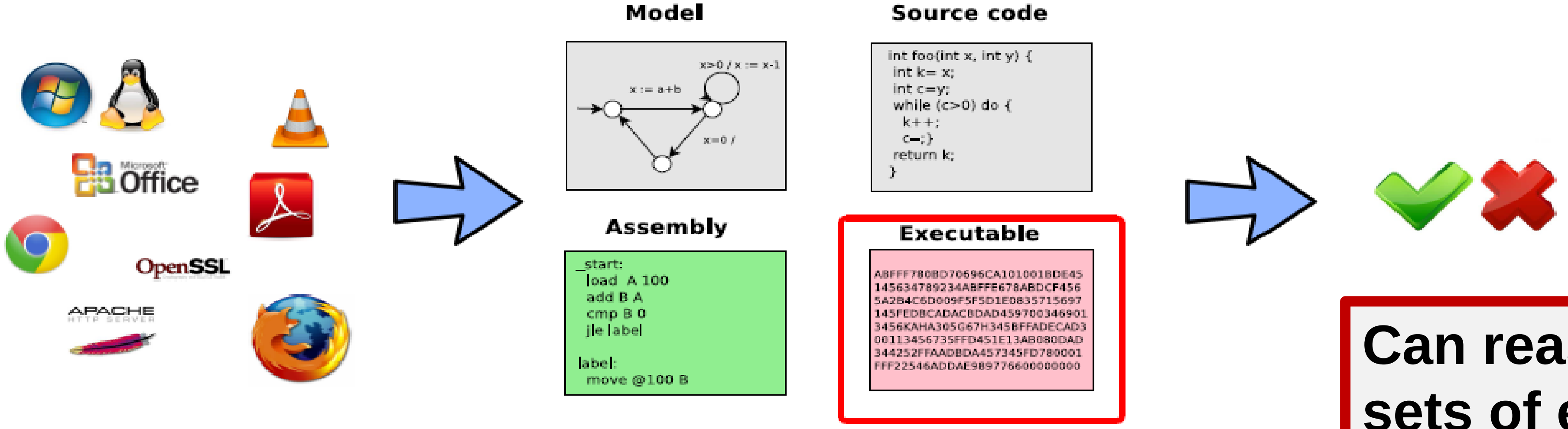
- **Static (syntactic): too fragile**
- **Dynamic: too incomplete**

# SOLUTION? BINARY-LEVEL SEMANTIC ANALYSIS

**Semantic tools** help make sense of binary

- Develop the next generation of binary-level tools !
- motto : leverage formal methods from safety critical systems

**Semantic preserved by compilation or obfuscation**




**Can reason about sets of executions**

### Advantages

- more robust than syntactic
- more thorough than dynamic

### Challenges

- source-level $\mapsto$ binary-level
- safety $\mapsto$ security
- many (complex) architectures

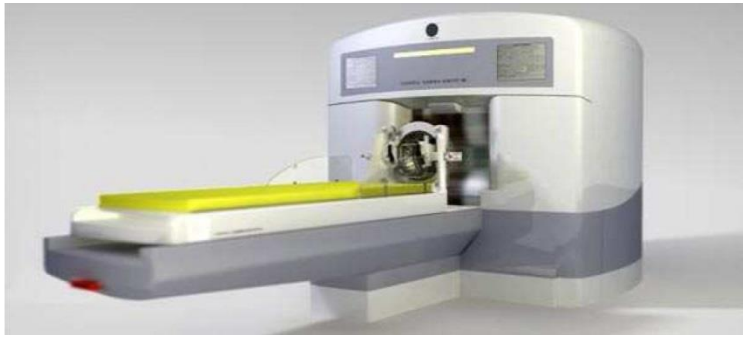
# OUTLINE

- Why binary-level analysis?
- Some background on source-level formal methods
- The hard journey from source to binary
- A few case-studies
- Conclusion

# BACK IN TIME: THE SOFTWARE CRISIS (1969)

*The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly : as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

- Edsger Dijkstra, The Humble Programmer (EWD340)



http ://en.wikipedia.org/wiki/List_of_software_bugs

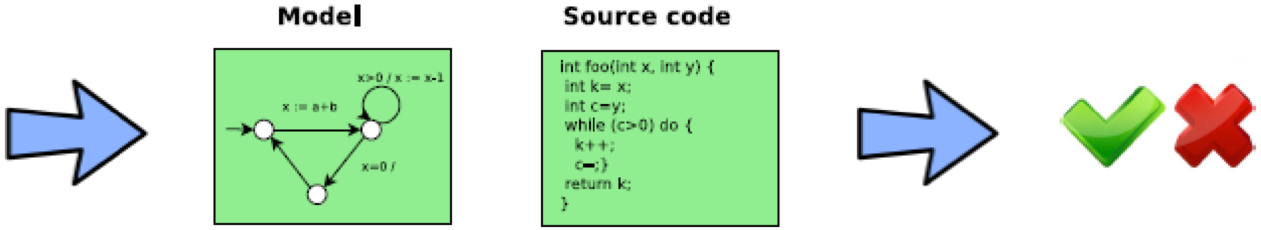
*Testing can only reveal the presence of errors but never their absence.*

- E. W. Dijkstra (Notes on Structured Programming, 1972)

# ABOUT FORMAL METHODS

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

**Success in safety-critical**




## Key concepts : $M \models \varphi$

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

## Kind of properties

- absence of runtime error
- pre/post-conditions
- temporal properties

# A DREAM COME TRUE … IN CERTAIN DOMAINS

Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

**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 DREAM COME TRUE … IN CERTAIN DOMAINS (2)

**Ex : Microsoft**

Verification of drivers [SDV]

- conformance to MS driver policy
- home developers
- and third-party developers


*Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.*

- Bill Gates (2002)

# OVERVIEW OF FORMAL METHODS

**Semantics**

- Precise meaning for the domain of evaluation and the effect of instructions
- Operational semantics = « interpreter »

**Properties**

- From Invariants / reachability to safety/liveness/hyper-properties/…
- On software: mostly invariants and reachability

**Algorithms:**

- Historically: Weakest precondition, Abstract interpretation, model checking
- Correctness: the analysis explores only behaviors of interest
- Completeness: the analysis explores at least all behaviors of interest

# OVERVIEW OF FORMAL METHODS

**Trends:**

- **Frontier between techniques disappear**
- **master abstraction (correct xor complete)**
- **reduction to logic**
- **sweet spots**

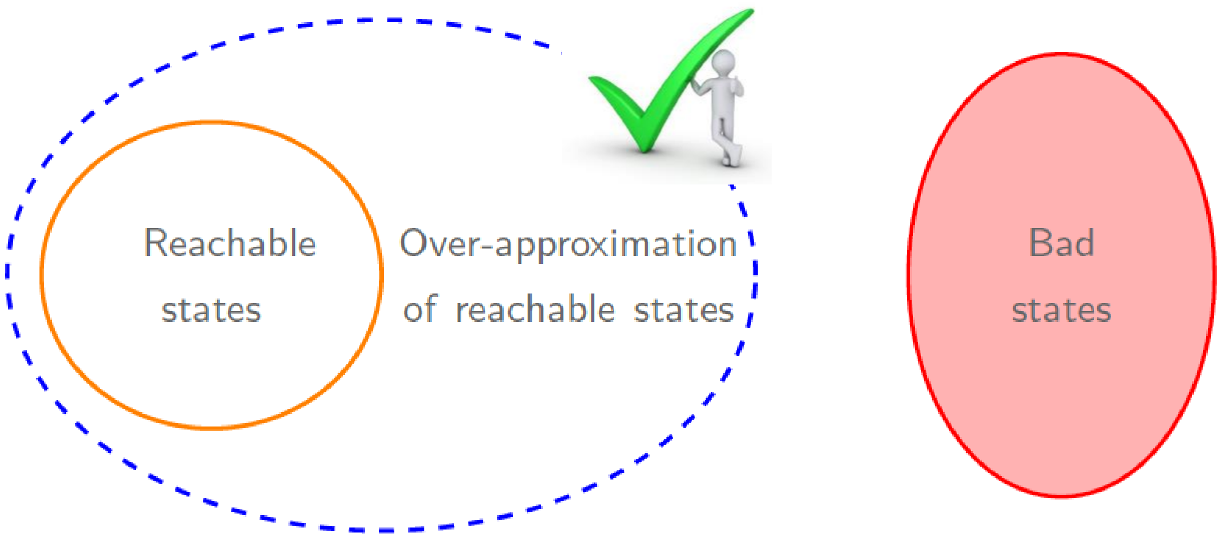
**Next:**

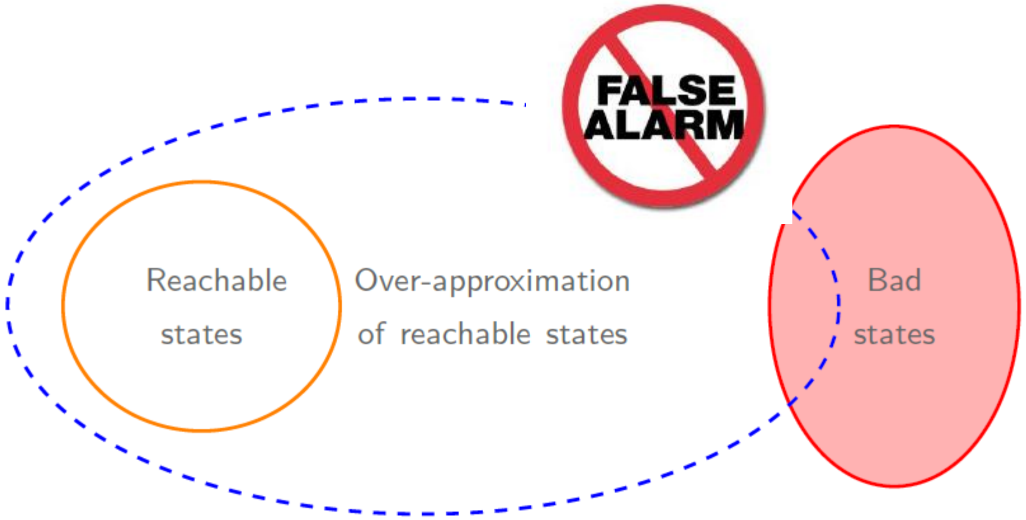
- **AI: complete (can prove invariants) -- 1977**
- **DSE: correct (can find bugs) -- 2005**

- Representative
- Industrial successes at source-level
- Adaptation to binary: very different situations

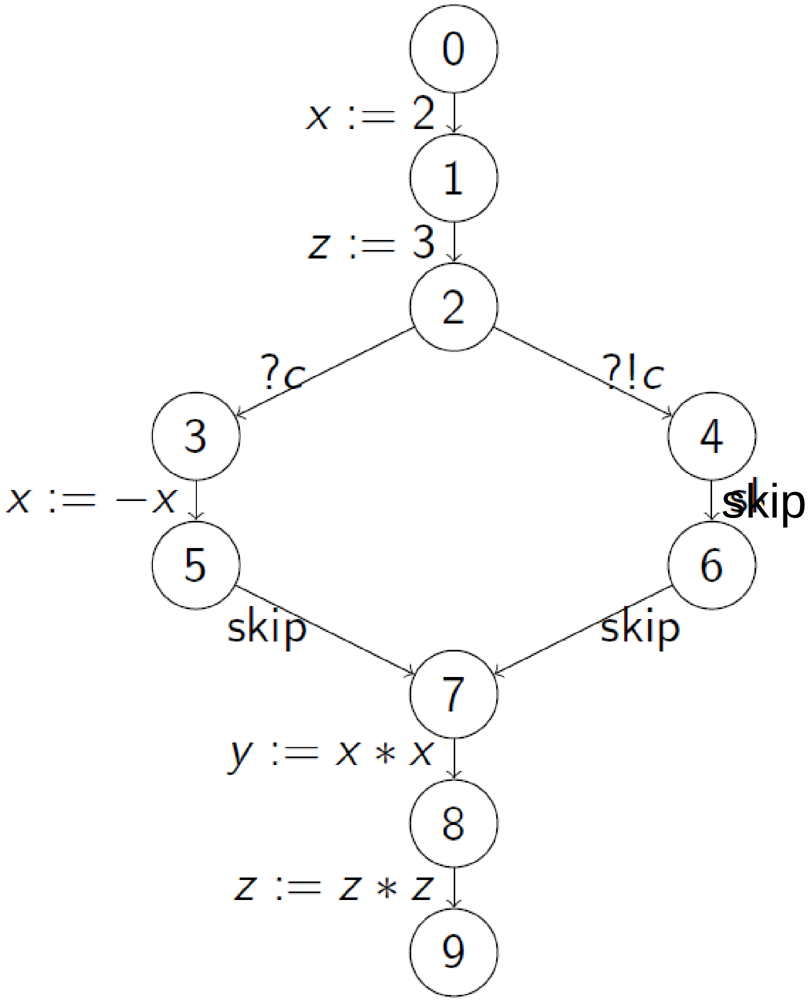
# ABSTRACT INTERPRETATION

$$(\mathcal{P}(states), \cup, \cap, \rightarrow) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (states^{\#}, \sqcup, \sqcap, \rightarrow^{\#})$$



# ABSTRACT INTERPRETATION IN PRACTICE




| nœud | $c$ | $x$ | $y$ | $z$ |
|---|---|---|---|---|
| 0 | $\top$ | $\top$ | $\top$ | $\top$ |
| 1 | $\top$ | 2 | $\top$ | $\top$ |
| 2 | $\top$ | 2 | $\top$ | 3 |
| 3 | $\top$ | 2 | $\top$ | 3 |
| 4 | 0 | 2 | $\top$ | 3 |
| 5 | $\top$ | $-2$ | $\top$ | 3 |
| 6 | 0 | 2 | $\top$ | 3 |
| 7 | $\top$ | $\top$ | $\top$ | 3 |
| 8 | $\top$ | $\top$ | $\top$ | 3 |
| 9 | $\top$ | $\top$ | $\top$ | 9 |

# ABSTRACT INTERPRETATION IN PRACTICE

**Key points:**

- **Infinite data: abstract domain**
- **Path explosion: merge**
- **Loops: widening**

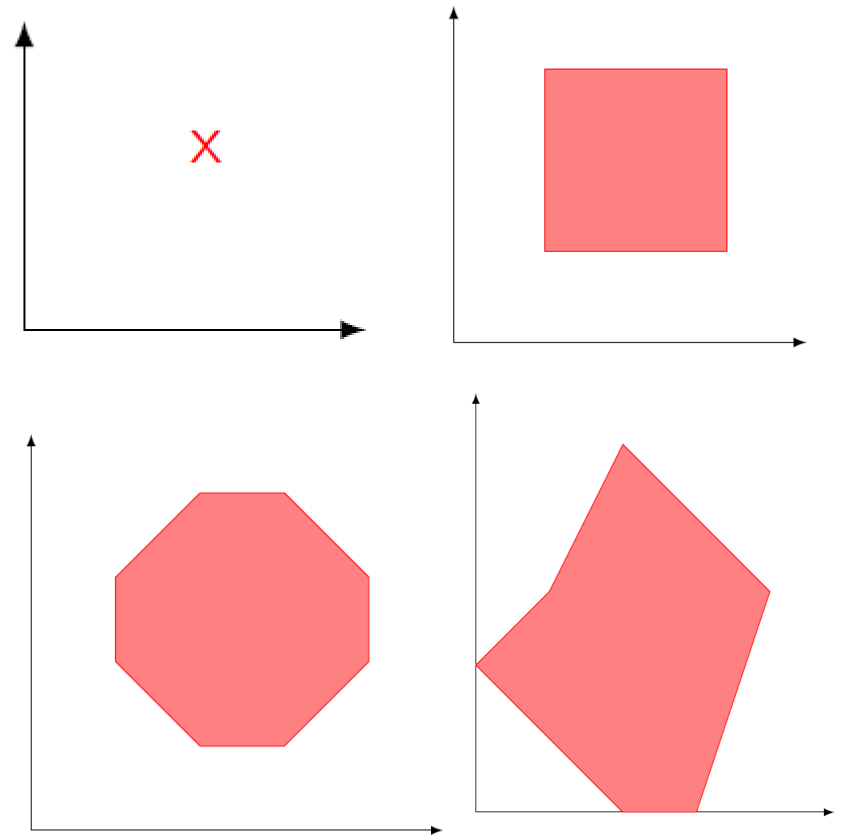
**In practice:**

- **Tradeoff between cost and precision**
- **Tradeoff between generic & dedicated domains**

**It is sometimes simple and useful**

- **taint, pointer nullness, typing**

**Big successes: Astrée, Frama-C, Clousot**



# DYNAMIC SYMBOLIC EXECUTION (DSE, Godefroid 2005)

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

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths

$\sigma := \emptyset$

$\mathcal{PC} := \top$

**Perfect for intensive testing**

- **Correct, relatively complete**
- **No false alarm**
- **Robust**
- **Scale in some ways**

**// incomplete**

$\mathcal{PC} := \top \wedge 2y_0 \neq x_0$

$\mathcal{PC} := \top \wedge 2y_0 = x_0 \wedge x_0 > y_0 + 10$

$\mathcal{PC} := \top \wedge 2y_0 = x_0 \wedge x_0 \leq y_0 + 10$

# DSE: PATH PREDICATE COMPUTATION

## (DSE, Godefroid 2005)

| 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) |

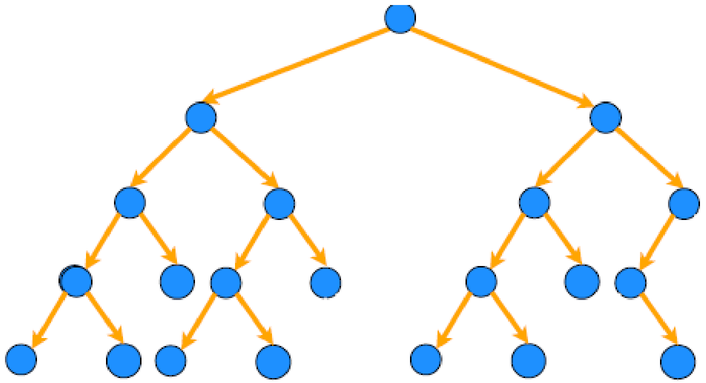
$$
\begin{array}{l}
\text{let } W_1 \triangleq Y_0 + 1 \text{ in} \\
\quad \text{let } X_2 \triangleq W_1 + 3 \text{ in} \\
\quad\quad X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0
\end{array}
$$

# DSE: GLOBAL PROCEDURE

## (DSE, Godefroid 2005)

**input** : a program P
**output** : a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- SAT(s) ? get a new pair $< \mathrm{s}, \sigma >$
- loop until no more path to cover



# ABOUT ROBUSTNESS (imo, the major advantage)

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
g(int x) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

Symbolic Execution

- create a subformula $z = x * x$, out of theory [FAIL]

Dynamic Symbolic Execution

- first concrete execution with x=3, y=5 [goto OK]
- during path predicate computation, $x * x$ not supported
  - . $x$ is concretized to 3 and $z$ is forced to 9
- resulting path predicate : $x = 3 \wedge z = 9 \wedge y = z$
- a solution is found : x=3, y=9 [goto ERROR] [SUCCESS]

**« concretization »**

- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

# DSE

**Three key ingredients**

- **Path predicate & solving**
- **Path enumeration**
- **C/S policy**

**Limits**

- **#paths -> better heuristics (?), state merging, distributed search, path pruning, adaptation to coverage objectives, etc.**
- **solving cost -> preprocessing, caching, incremental solving, aggressive concretization (good?) [wait for better solvers ☺]**
- **Preconditions/postconditions/advanced stubs**

# DSE: PATH PREDICATE MAY BE COMPLICATED

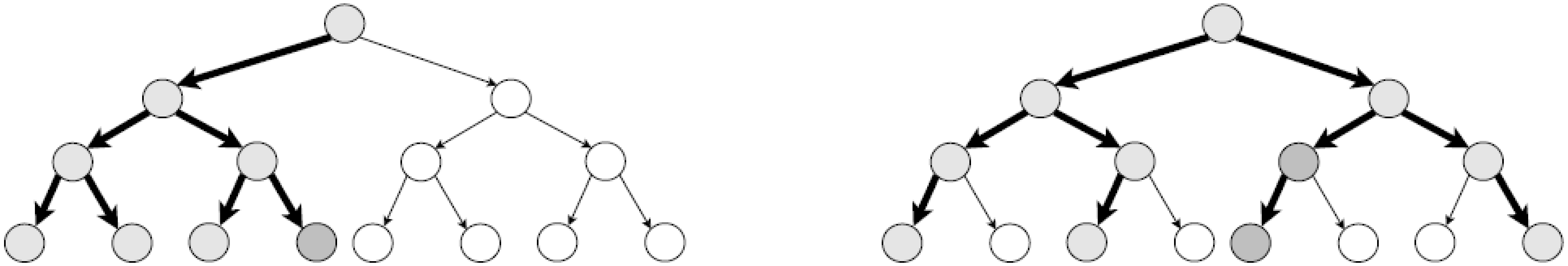
```
x := a + b
```

$$X_{n+1} = A_n + B_n$$

$$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$$

$$\begin{array}{l}
let\ tmpA = load(M,addr(A))\ @\ load(M,addr(A)+1)\ @\ load(M,addr(A)+2) \\
and\ tmpB = load(M,addr(B))\ @\ load(M,addr(B)+1)\ @\ load(M,addr(B)+2) \\
in \\
let\ nX = tmpA+tmpB \\
in \\
\quad store( \\
\quad\quad store( \\
\quad\quad\quad store(M, addr(X), nX[0]), \\
\quad\quad\quad addr(X) + 1, nX[1]), \\
\quad\quad addr(X) + 2, nX[2])
\end{array}$$

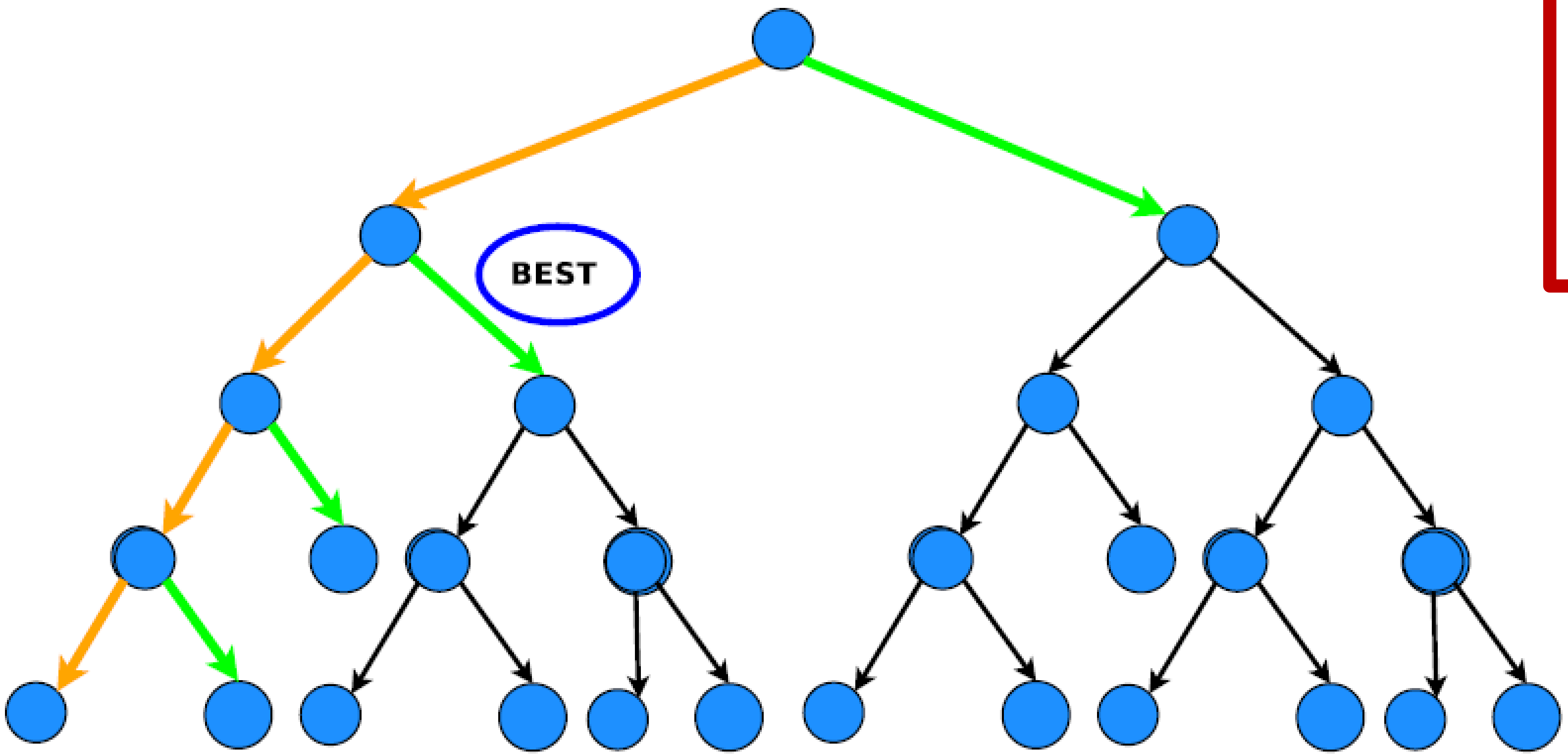
# DSE: SEARCH



(a) DFS (b) BFS

- **Search heurstics matters**
- But no good choice (hint: DFS is often the worst)
- The engine must provide flexibility

# DSE: SEARCH (2)




**Generic engine**
- Score each active prefix
- Pick the best & expand
- Easy encoding of many heuristics

# C/S POLICIES

Robustness : what if the instruction cannot be reasoned about ?

- missing code, self-modification
- hash functions, dynamic memory accesses, NLA operators

**program**

```
input: a, b
x := a × b
x := x + 1
//assert x > 10
```

**path predicate**

$$x1 = a \times b \;\wedge\; x2 = x1 + 1 \;\wedge\; x2 > 10 \quad (\varphi_1)$$

**concretization**

$$a = 5 \;\wedge\; x1 = 5 \times b \;\wedge\; x2 = x1 + 1 \;\wedge\; x2 > 10 \quad (\varphi_2)$$

**symbolization**

$$x1 = fresh \;\wedge\; x2 = x1 + 1 \;\wedge\; x2 > 10 \quad (\varphi_3)$$

Solutions

- **Concretization** : replace by runtime value [lose completeness]
- **Symbolization** : replace by fresh variable [lose correctness]

# C/S POLICIES (2)

Consider the following situation

- instruction `x := @(a * b)`
- your tool documentation says : *"memory accesses are concretized"*
- suppose that at runtime : `a = 7, b = 3`

What is the intended meaning ? [perfect reasoning : $x == select(M, a \times b)$]

**CS1 :** $x == select(M, 21)$ [incorrect]

**CS2 :** $x == select(M, 21) \wedge a \times b == 21$ [minimal]

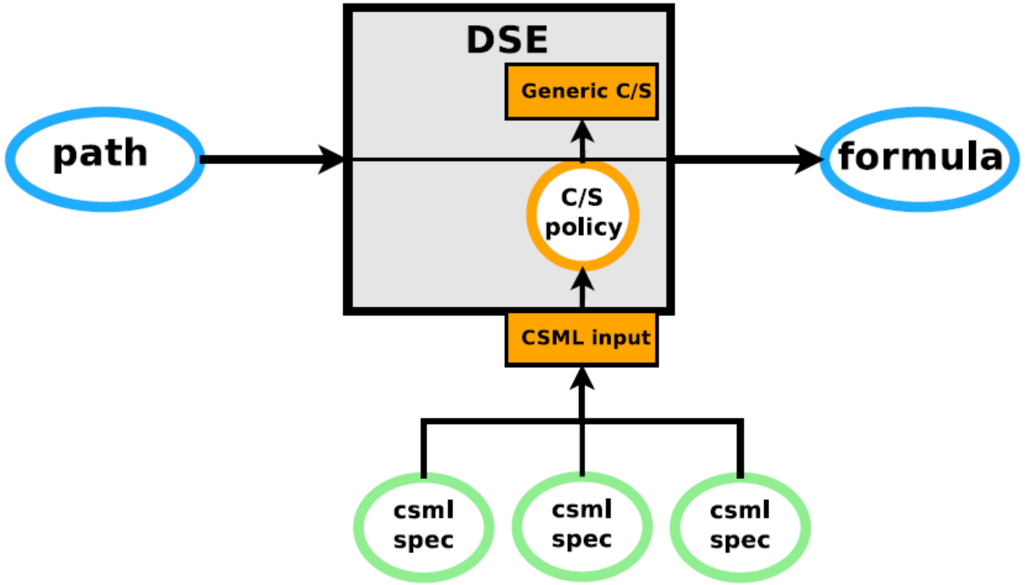
**CS3 :** $x == select(M, 21) \wedge a == 7 \wedge b == 3$ [atomic]

No best choice, depends on the context

- acceptable loss of correctness / completeness ?
- is it mandatory to get rid off $\times$ ?

- **C/S policy matters**
- But no good choice
- The engine must provide flexibility

# C/S POLICIES (3)


**Generic engine**

- C/S specification
- DSE parametrized by C/S

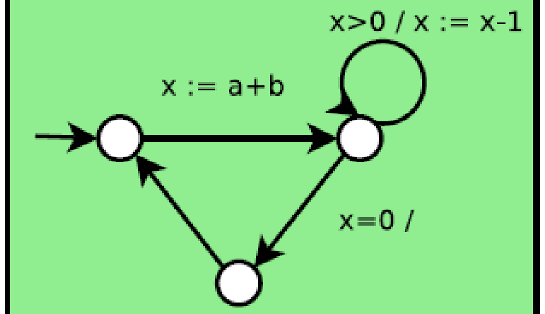
$$\mathsf{cs} : \mathsf{loc} \times \mathsf{instr} \times \mathsf{state} \times \mathsf{expr} \mapsto \left\{ \begin{array}{ll} \mathcal{C} & \text{concretization} \\ \mathcal{S} & \text{symbolization} \\ \mathcal{P} & \text{propagation} \end{array} \right\}$$

# OUTLINE

- Why binary-level analysis?
- Some background on source-level formal methods
- The hard journey from source to binary
- A few case-studies
- Conclusion

# NOW: BINARY-LEVEL 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
```

# THE HARD JOURNEY FROM SOURCE TO BINARY

## Low-level semantics of data

- machine arithmetic, bit-level operations, untyped memory
- ▶ difficult for any state-of-the-art formal technique

**Wanted**
- robustness
- precision
- scale

## Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`jmp eax`)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
- ▶ violate an implicit prerequisite for most formal techniques

## Diversity of architectures and instruction sets

- support for many instructions, modelling issues
- ▶ tedious, time consuming and error prone

# ADAPTING DSE and AI to BINARY: two very different stories

**DSE is quite easy to adapt**

- **thx to SMT solvers (arrays+bitvectors)**
- **thx to concretization**
- **yet, performance degrades**

**Problems**

- Low-level control: jump eax
- Low-level data: memory
- Low-level data: flags

**AI is much more complicated**

- **Even for « normal » code**
- **btw, cannot expect better than source-level precision**

**Problem solved: multi-architecture**

- rely on some IR

# FULL DISCLOSURE: the BINSEC tool

Still very young!

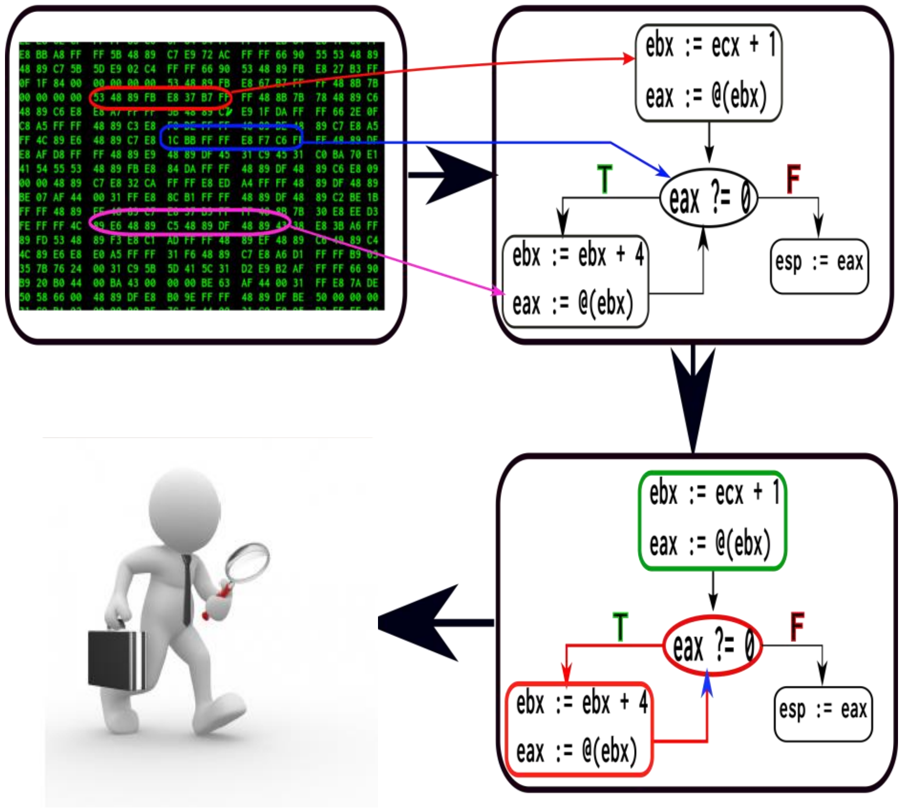
**Semantic analysis for binary-level security**

- Help make sense of binary
- more robust than syntactic
- more exhaustive than dynamic

**Some features**

- Help to recover a simple model
- Identify feasible events (+ input)
- Identify infeasible events (eg, protections)
- Multi-architecture

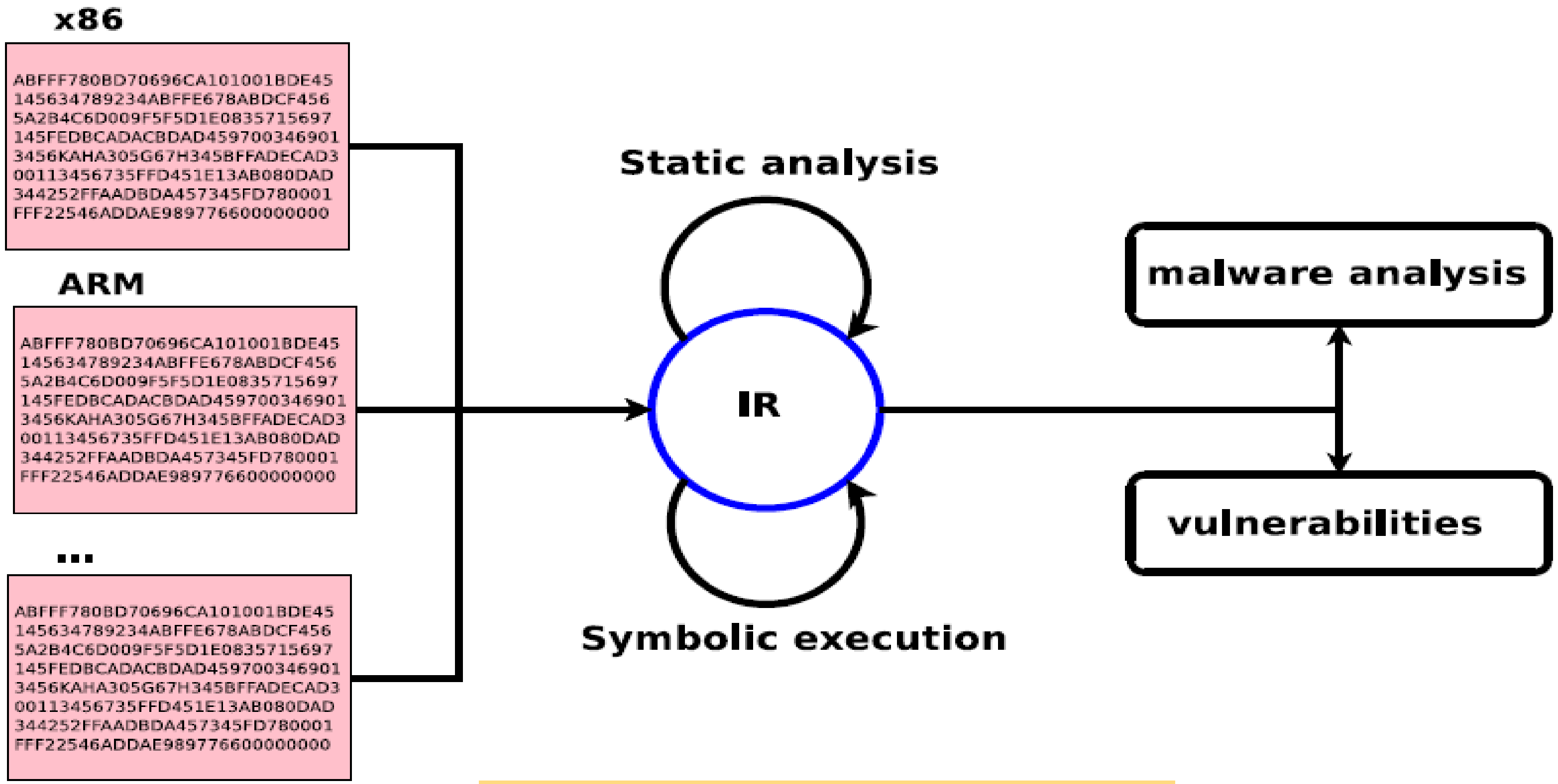
BINSEC

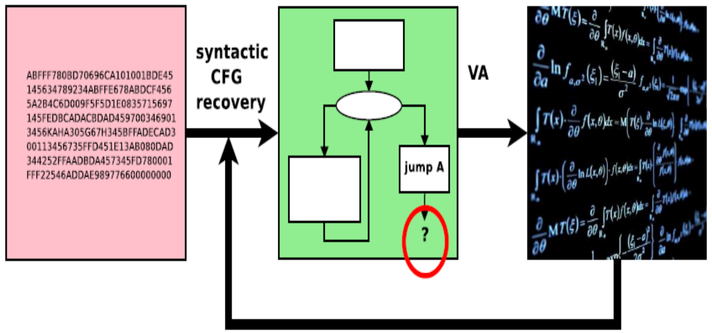


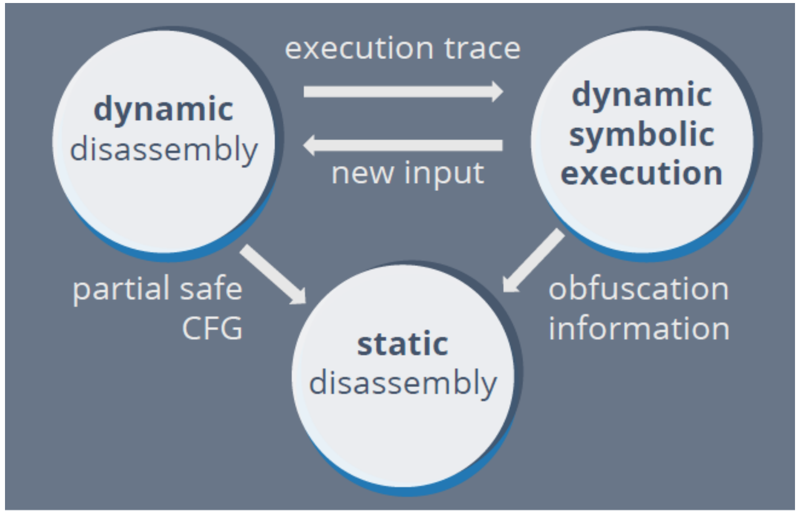

**Challenges**

- Binary analysis
- **Scalability**
- **Robustness** w.r.t obfuscation

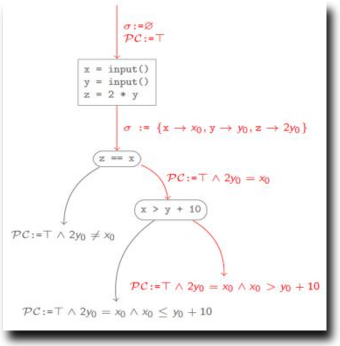
# UNDER THE HOOD

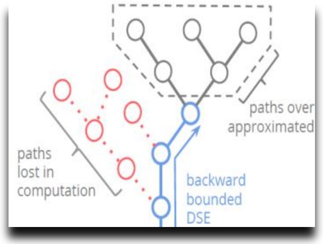


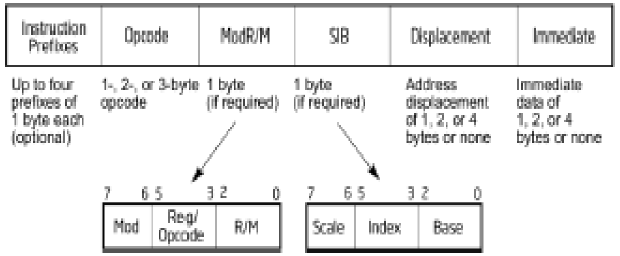

- lhs := rhs
- goto addr, goto expr
- ite(cond)? goto addr :
- assume, assert, nondet





# INTERMEDIATE REPRESENTATION

- `lhs := rhs`
- `goto addr, goto expr`
- `ite(cond)? goto addr`




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

- **Concise**
- **Well-defined**
- **Clear, side-effect free**

```
(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)
```

# INTERMEDIATE REPRESENTATION + simplifications

| program | native loc | DBA loc | opt (DBA) time | opt (DBA) loc | opt (DBA) red |
|---|---|---|---|---|---|
| bash | 166K | 559K | 673.61s | 389K | 30.45% |
| cat | 8K | 23K | 18.54s | 18K | 23.02% |
| echo | 4K | 10K | 6.96s | 8K | 24.26% |
| less | 23K | 80K | 69.99s | 55K | 30.96% |
| ls | 19K | 63K | 65.69s | 44K | 30.58% |
| mkdir | 8K | 24K | 19.74s | 17K | 29.50% |
| netstat | 17K | 50K | 52.59s | 40K | 20.05% |
| ps | 12K | 36K | 36.99s | 27K | 23.98% |
| pwd | 4K | 11K | 7.69s | 9K | 23.56% |
| rm | 10K | 30K | 24.93s | 22K | 25.24% |
| sed | 10K | 32K | 28.85s | 23K | 26.20% |
| tar | 64K | 213K | 242.96s | 154K | 27.48% |
| touch | 8K | 26K | 24.28s | 18K | 27.88% |
| uname | 3K | 10K | 6.99s | 8K | 23.62% |

| | reduction | | | |
|---|---|---|---|---|
| | time | dba instr | tmp assigns | flag assigns |
| BINSEC | 1279.81s | 28.64% | 88.00% | 75.04% |

**Approach**

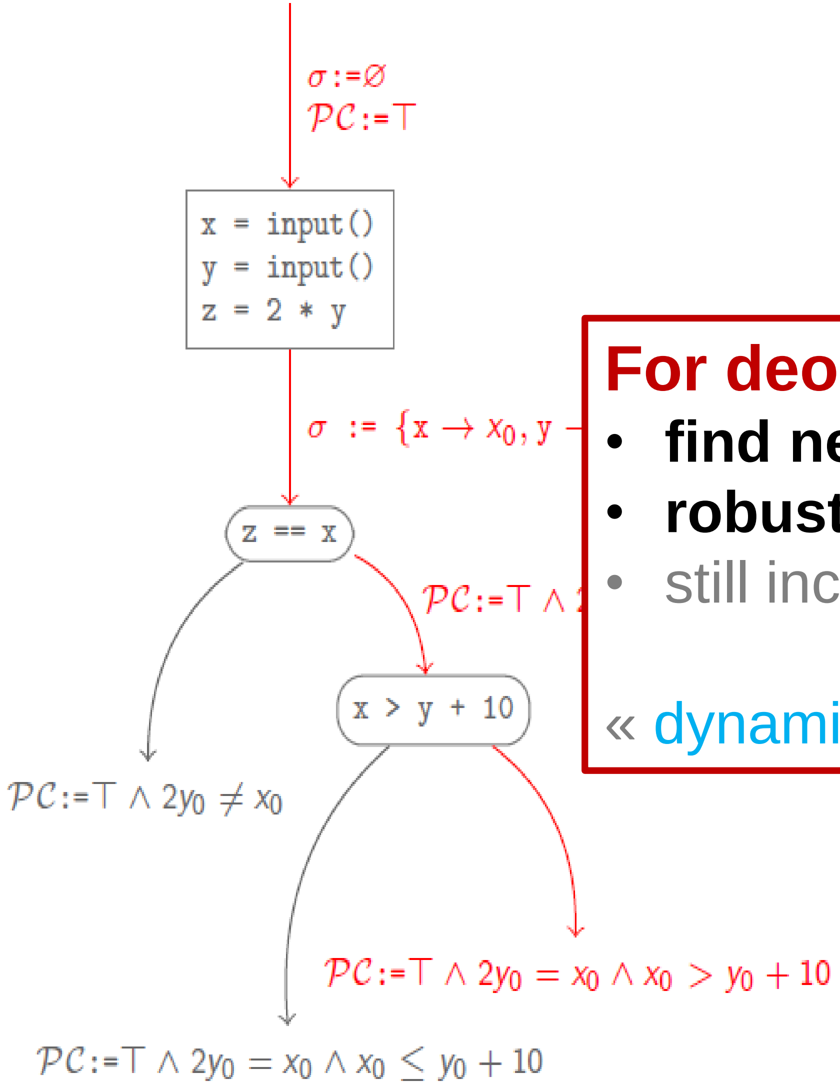
- Inspired from standard compiler optim
- Targets : flags & temp
- Sound : w.r.t. incomplete CFG
- Inter-procedural (summaries)

- **IR level**
- **machine-instruction level**
- **program level**

# BINARY-LEVEL DSE (Godefroid)

```
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 the program
- automatically find input that follows the path
- then, iterate over all paths

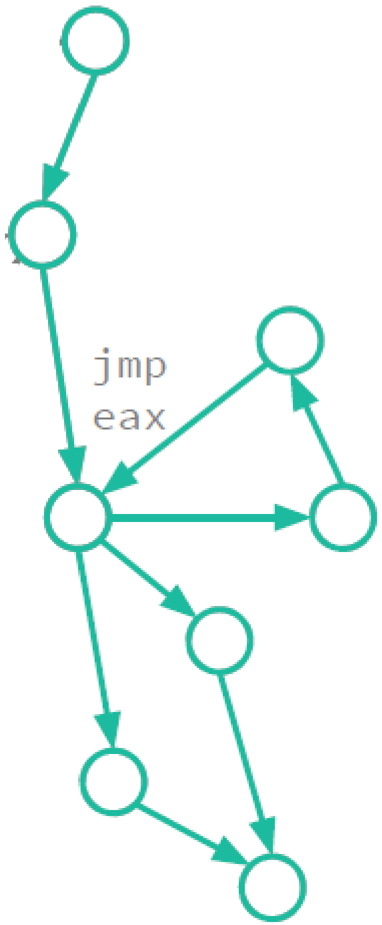


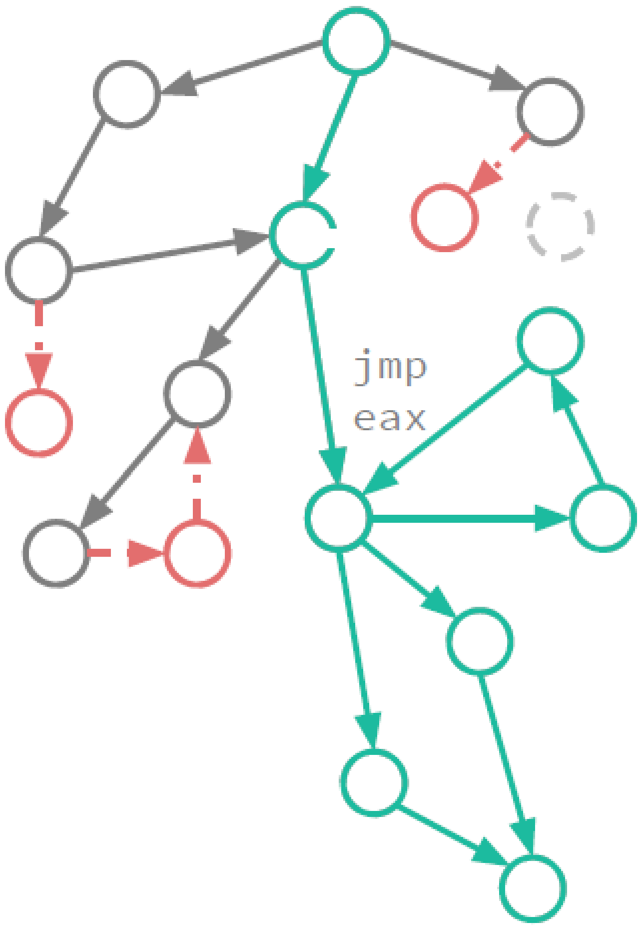

**For deobfuscation**

- **find new real paths**
- **robust**
- still incomplete

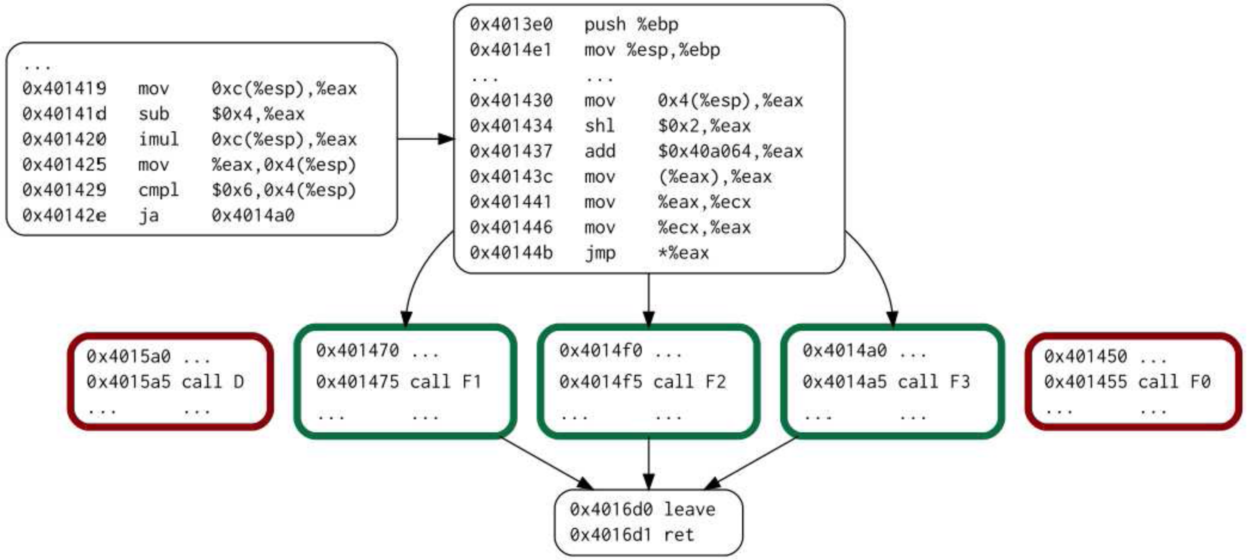
« dynamic analysis on steroids »

# DSE COMPLEMENTS DYNAMIC ANALYSIS



# IN PRACTICE




```
cmp eax ebx
cmc
jae ...
```

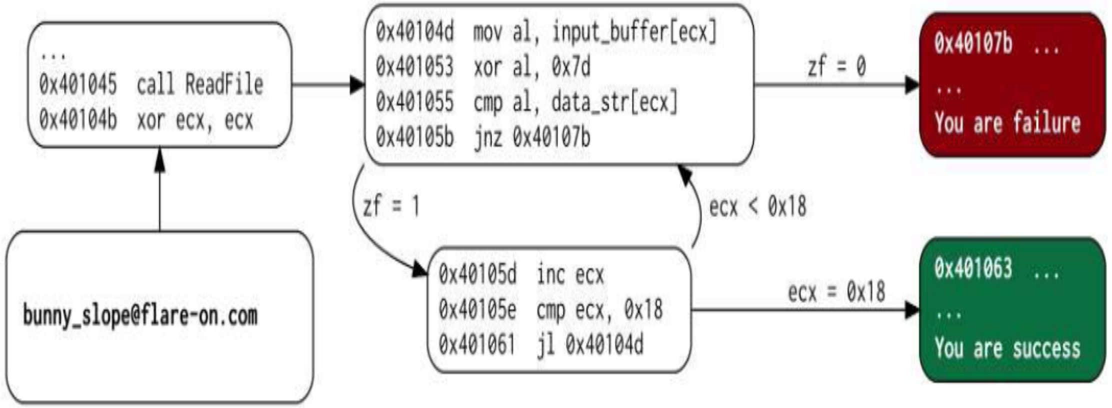
$\mathrm{CF} := (\mathrm{eax} <_u \mathrm{ebx})$

$\mathrm{CF} := \neg\mathrm{CF}$

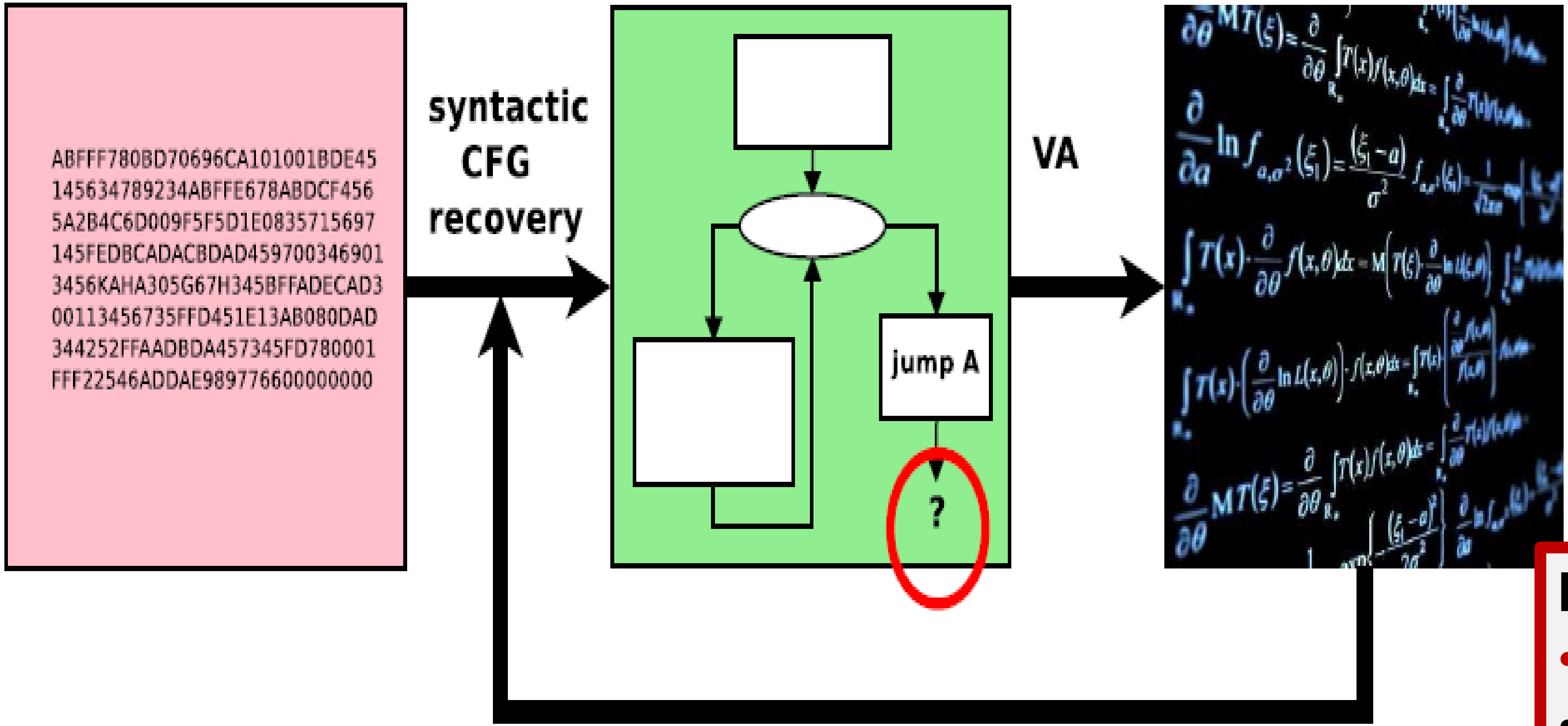
if $(\neg\mathrm{CF})$ goto ...

With IDA + BINSEC

**Can recover useful semantic information**
- **More precise disassembly**
- **Exact semantic of instructions**
- **Input of interest**
- **…**



# ABSTRACT INTERPRETATION IS VERY VERY HARD ON BINARY CODE

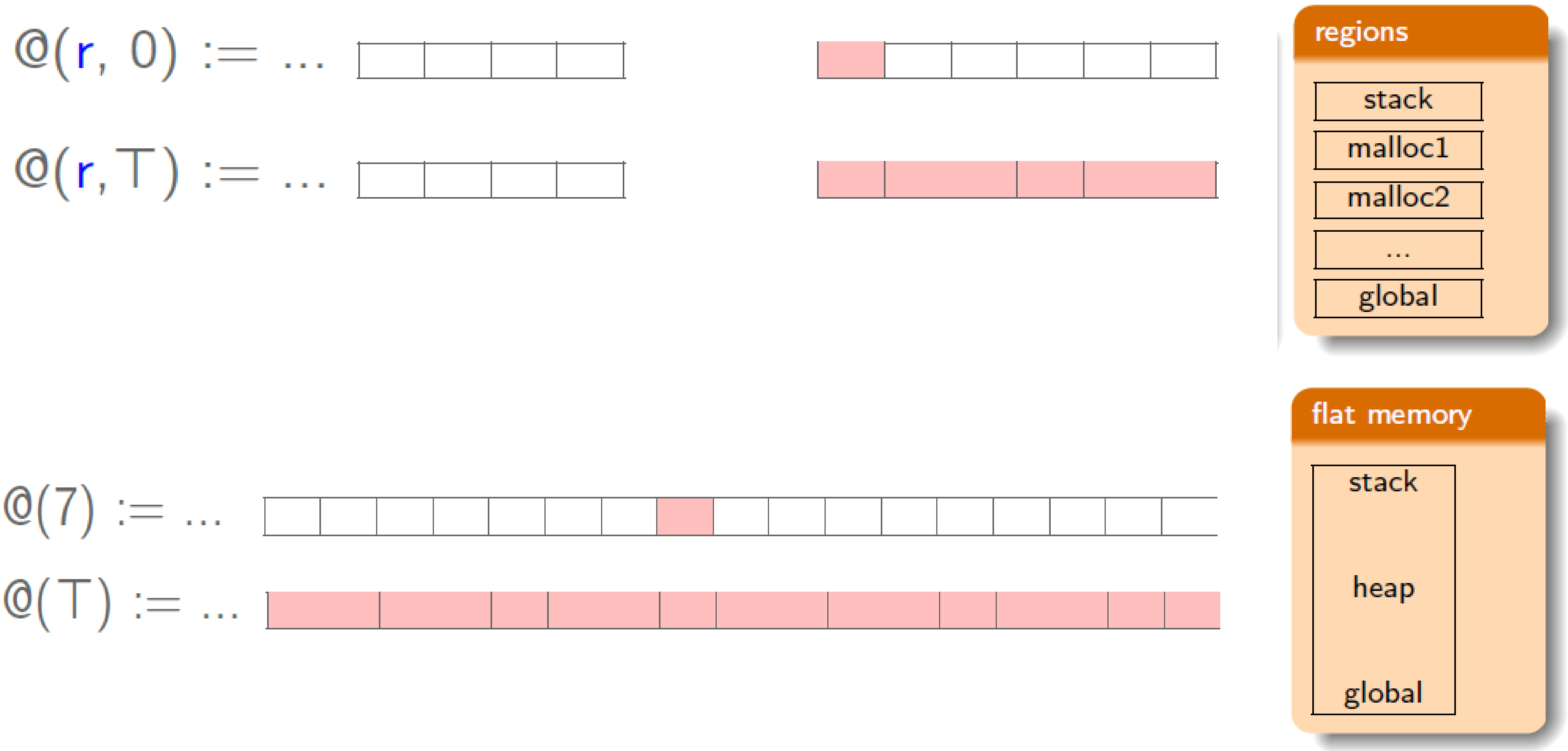



**Problems**

- Jump eax
- memory
- Bit resoning

# ISSUE: GLOBAL MEMORY

**Problems**
- Jump eax
- memory
- Bit resoning



# ISSUE: LACK of HIGH-LEVEL STRUCTURE

**Problems**
- Jump eax
- memory
- Bit resoning

High-level conditions translated into low-level flag predicates

```
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:
```

Condition on flags, not on register (nor stack)

# LOW-LEVEL CONDITIONS

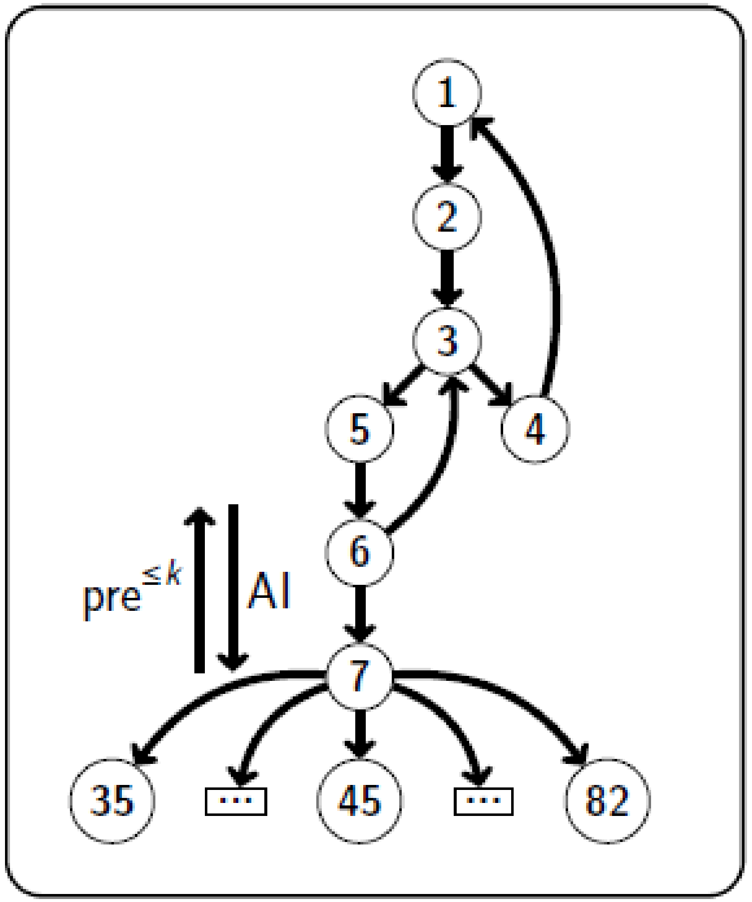
| | flag predicate | cmp x y predicate | sub x y predicate [2] | test x y predicate |
|---|---|---|---|---|
| ja, jnbe | $\neg CF \land \neg ZF$ | $x >_u y$ | $x' \neq 0$ | $x\&y \neq 0$ |
| jae, jnb, jnc | $\neg CF$ | $x \geq_u y$ | true | true |
| jb, jnae, jc | $CF$ | $x <_u y$ | $x' \neq 0$ | false |
| jbe, jna | $CF \lor ZF$ | $x \leq_u y$ | true | $x\&y = 0$ |
| je, jz | $ZF$ | $x = y$ | $x' = 0$ | $x\&y = 0$ |
| jne, jnz | $\neg ZF$ | $x \neq y$ | $x' \neq 0$ | $x\&y \neq 0$ |
| jg, jnle | $\neg ZF \land (OF = SF)$ | $x > y$ | $x' > 0$ | $(x\&y \neq 0) \land (x \geq 0 \lor y \geq 0)$ |
| jge, jnl | $(OF = SF)$ | $x \geq y$ | true | $(x \geq 0 \lor y \geq 0)$ |
| jl, jnge | $(OF \neq SF)$ | $x < y$ | $x' < 0$ | $(x < 0 \land y < 0)$ |
| jle, jng | $ZF \lor (OF \neq SF)$ | $x \leq y$ | *true* | $(x\&y = 0) \lor (x < 0 \land y < 0)$ |

# LOW-LEVEL CONDITIONS

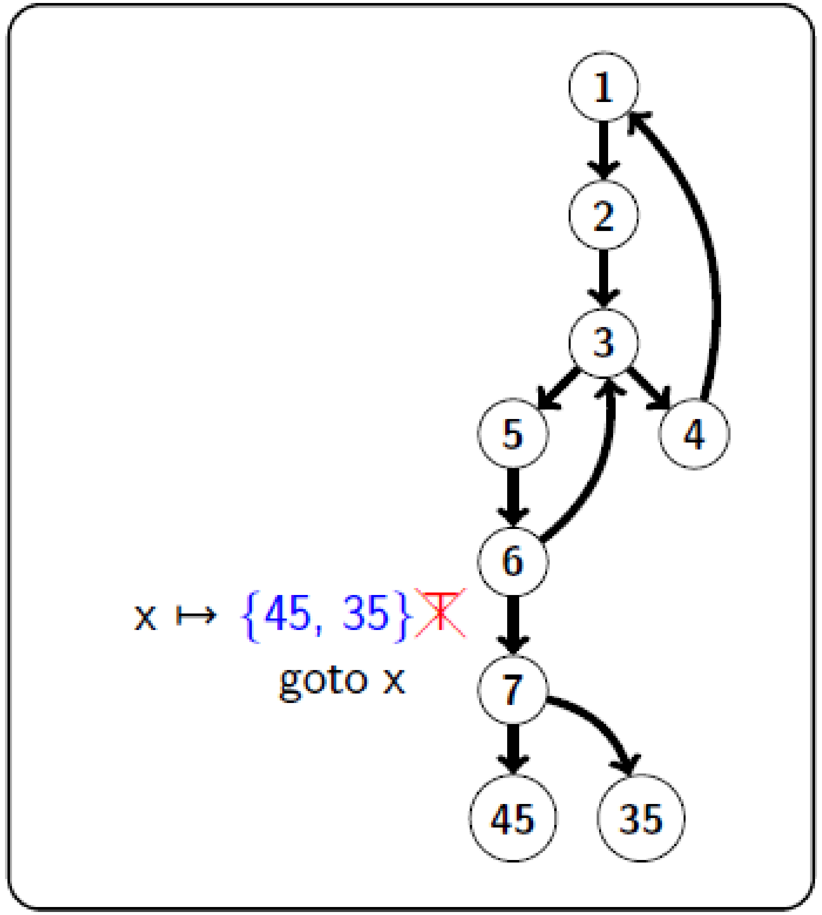
| example | retrieved condition | patterns |
| --- | --- | --- |
| `or eax, 0`<br>`je ...` | `if (eax = 0) then goto ...` | ✗ |
| `cmp eax, 0`<br>`jns ...` | `if (eax ≥ 0) then goto ...` | ✗ |
| `sar ebp, 1`<br>`je ...` | `if (ebp = 0) then goto ...` | ✗ |
| `dec ecx`<br>`jg ...` | `if (ecx ≥ 0) then goto ...` | ✗ |

# SOLUTIONS?

## Precision refinement [Brauer, 2011]




## Degraded mode [Kinder, 2012]



# SOLUTIONS? (2)

## Insights

- Complex predicates often hide simple predicates
- Only a few templates : $>_{u,s}, <_{u,s}, \geq_{u,s}, \leq_{u,s}, =, \neq$
- Try to find the appropriate one through equivalence checking
- Optimization :
  - Once per address using cache
  - Cheap pruning through filtering

| Approach | archi. independent | Sound | Complete enough |
|---|---|---|---|
| Patterns | ✗ | ✓/✗ | ✗ |
| Logic-based | ✓ | ✓ | ✗ |
| **Template-based** | ✓ | ✓ | ✓ |

# HIGH-LEVEL CONDITION RECOVERY

| method | #loc† | #cond‡ | #success⋆ | time | $time_{all}$ |
|---|---|---|---|---|---|
| templates | 242884 | 1978 | **1760 (89%)** | **22.93** | 2674.81 |
| logic-based | 247894 | 2260 | **694 (31%)** | 0.003 | 2561.64 |
| patterns | 229255 | 1987 | **1357 (68%)** | 0.014 | 2373.33 |
| templates+patterns | 242884 | 1978 | **1838 (92%)** | **9.17** | 2659.95 |

# STATIC ANALYSIS in BINSEC
## an overview

| | difficulty | solution |
|---|---|---|
| Domains | low-level arithmetic<br>ubiquitous data moves<br>low-level conditions | dual-intervals<br>equality domain<br>flag domain + condition recovery |
| Widening | no loop structure | *loop detection*<br>widening point positioning |
| CFG | unavailable<br><br>recovery scale | incremental CFG recovery<br>backward precision recovery<br>degraded mode [Kinder2012] |

# OVERVIEW

| | Correct | Complete | Efficient | Robust |
|---|---|---|---|---|
| **Static syntactic** | X | X / -- | OK | X |
| **Dynamic** | OK | XX | OK | OK |
| **DSE** | OK | -- | X | OK |
| **Static semantic** | X | OK / X | X | X |

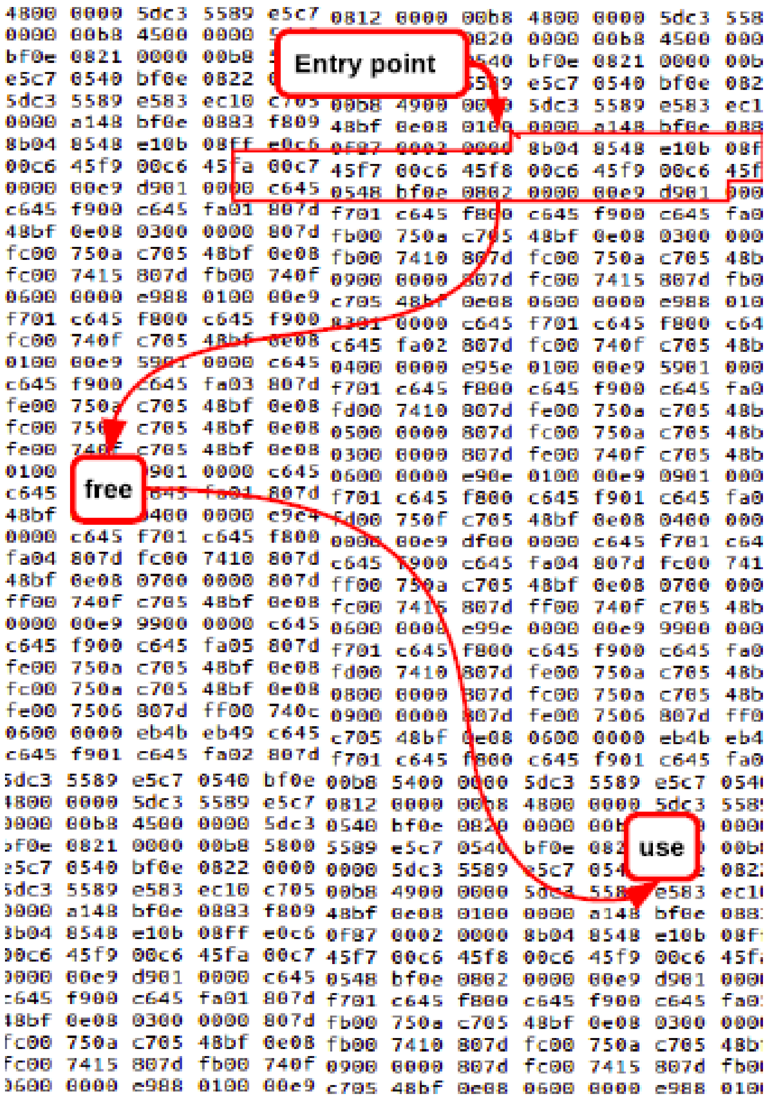
# OUTLINE

- Why binary-level analysis?
- Some background on source-level formal methods
- The hard journey from source to binary
- A few case-studies
- Conclusion

# APPLICATION: VULNERABILITY DETECTION

**Find vulnerabilities before the bad guys**

- **On the whole program**
- **At binary-level**
- **Know only the entry point and program input format**

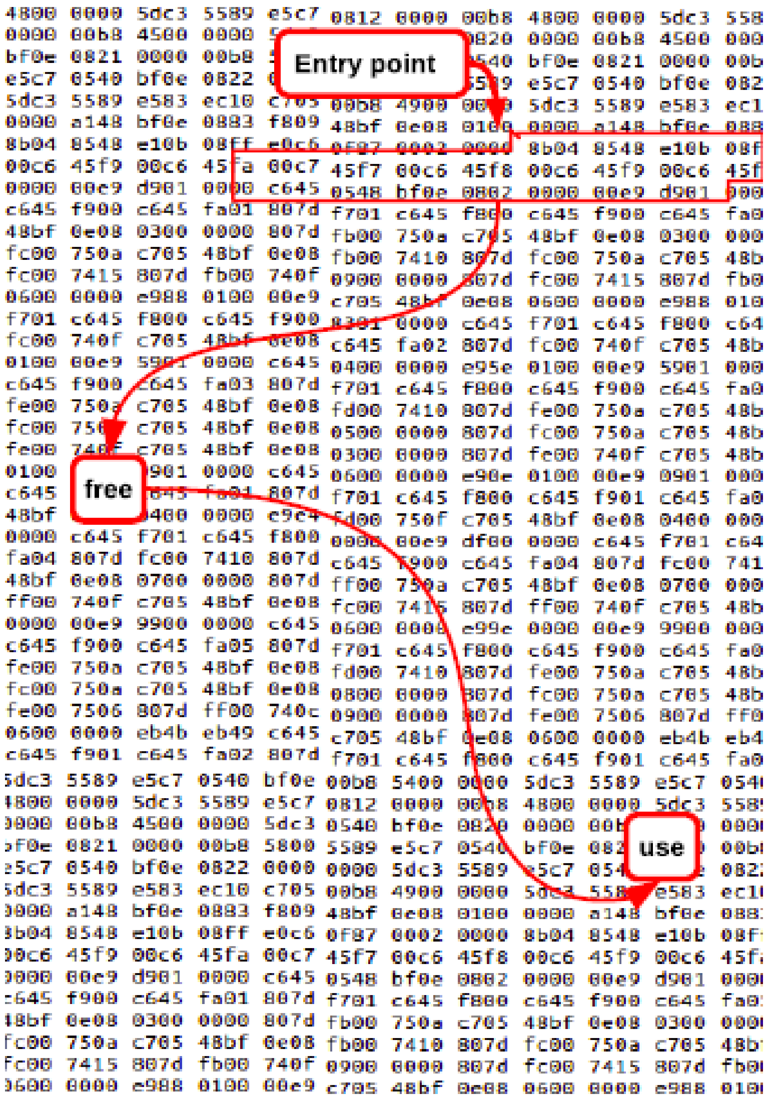



# APPLICATION: VULNERABILITY DETECTION

**Many successful applications of pure DSE**

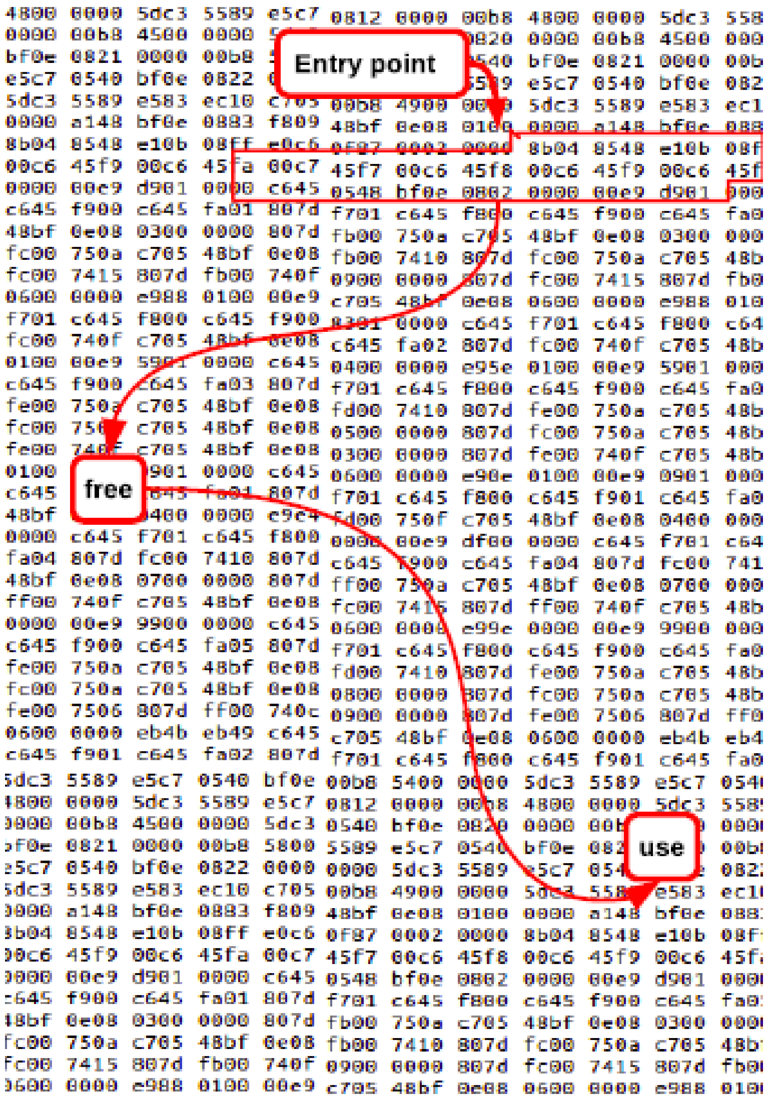
- **SAGE @ Microsoft**
- **Mayhem/VeriT @ ForallSecure**

**cf. Cyber Grand Challenge**

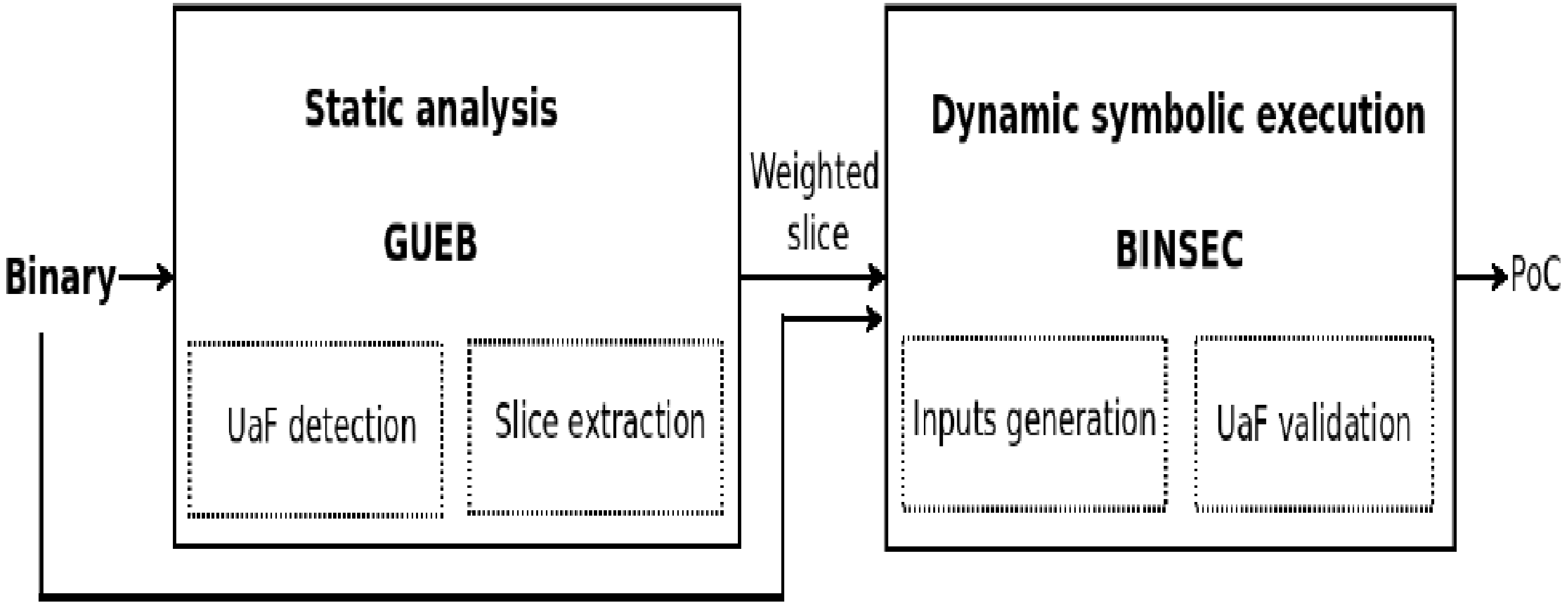



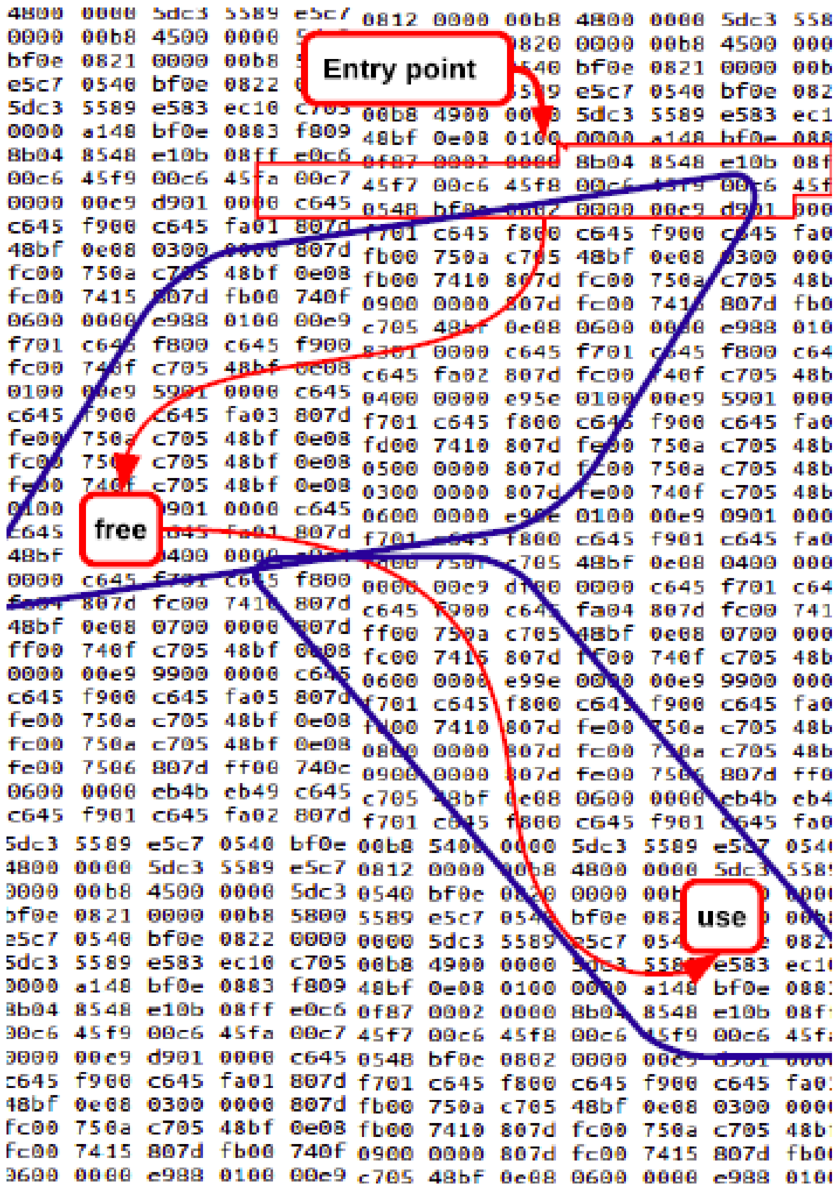
# APPLICATION: VULNERABILITY DETECTION [SSPREW 2016, with VERIMAG]

**Here:**
- **Focus on use-after-free**
- **Combine static and DSE**

# KEY IDEAS (Josselin Feist)



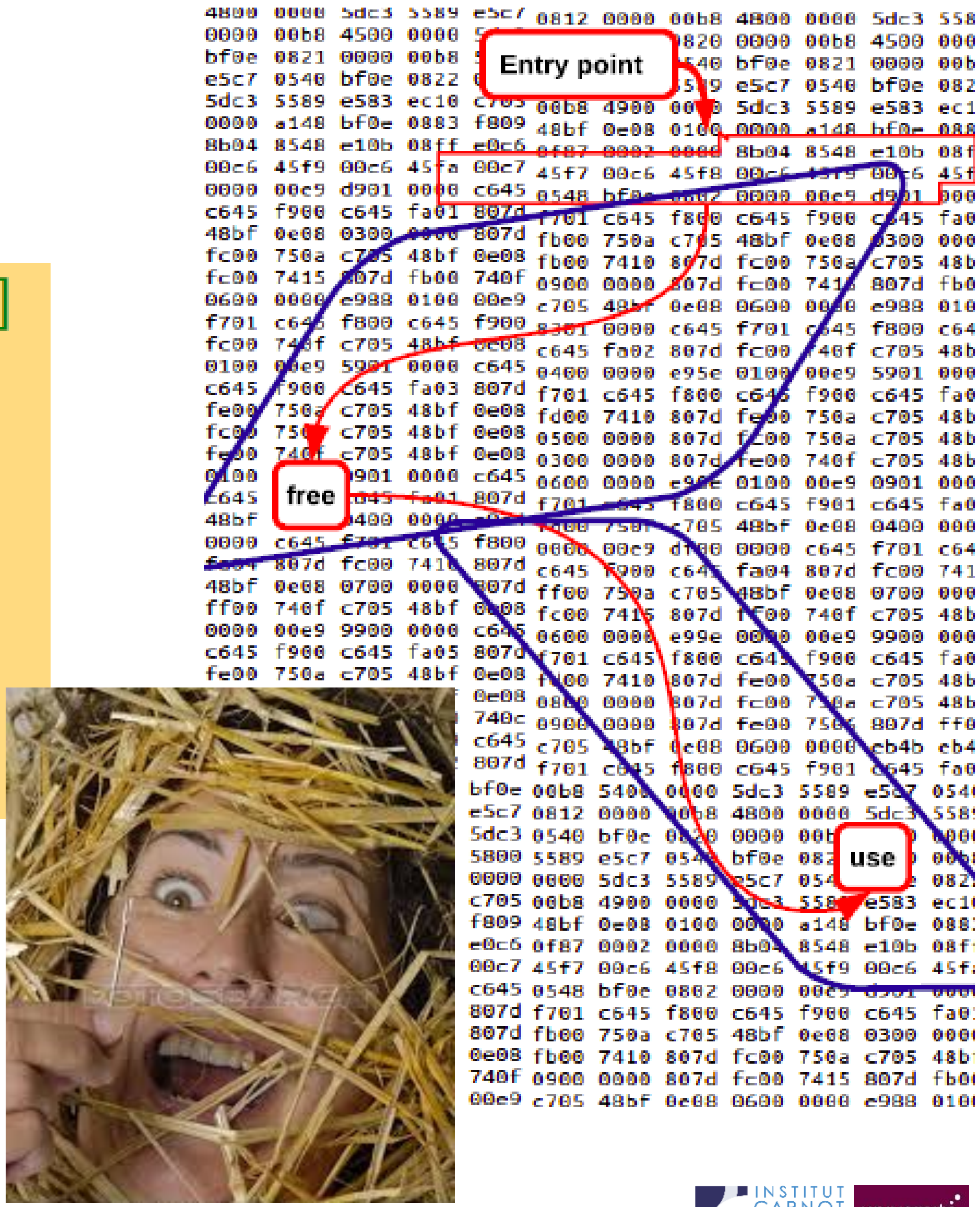




**A Pragmatic 2-step approach**

- **Static: scale, not complete, not correct**
- **Symbolic: correct, directed by static**
- **Combination: scalable and correct**

# EXPERIMENTAL EVALUATION

- GUEB + manual analysis [j. comp. virology 14]
  - tiff2pdf : CVE-2013-4232
  - openjpeg : CVE-2015-8871
  - gifcolor : CVE-2016-3177
  - accel-ppp
- GUEB + BINSE/SE [ssprew16]
  - Jasper JPEG-2000 : CVE-2015-5221




**On these examples:**
- **Better than DSE alone**
- **Better than blackbox fuzzing**
- **Better than greybox fuzzing with no seed**

# APPLICATION: MALWARE DEOBFUSCATION
# [S&P 2017, with LORIA]

APT: highly sophisticated attacks
- **Targeted malware**
- **Written by experts**
- Attack: 0-days
- Defense: stealth, **obfuscation**
- **Sponsored by states or mafia**

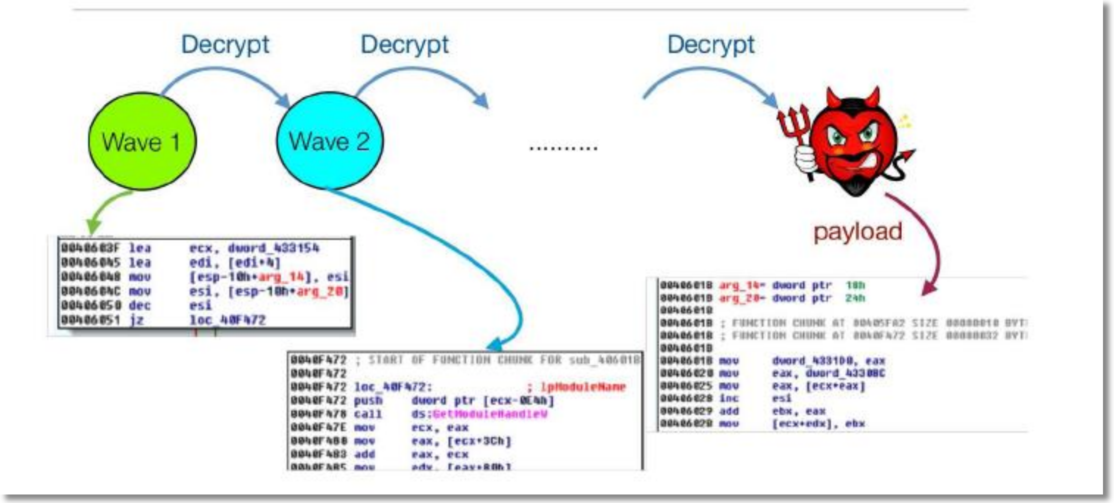
The day after: **malware comprehension**
- understand what has been going on
- mitigate, fix and clean
- improve defense


**USA elections: DNC Hack**



**Goal: help malware comprehension**
- **Reverse of heavily obfuscated code**
- **Identify and simplify protections**

# REVERSE CAN BECOME A NIGHTMARE (OBFUSCATION)




| address | instr |
|---|---|
| 80483d1 | call +5 |
| 80483d6 | pop edx |
| 80483d7 | add edx, 8 |
| 80483da | push edx |
| 80483db | ret |
| 80483dc | byte{invalid} |

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>
```

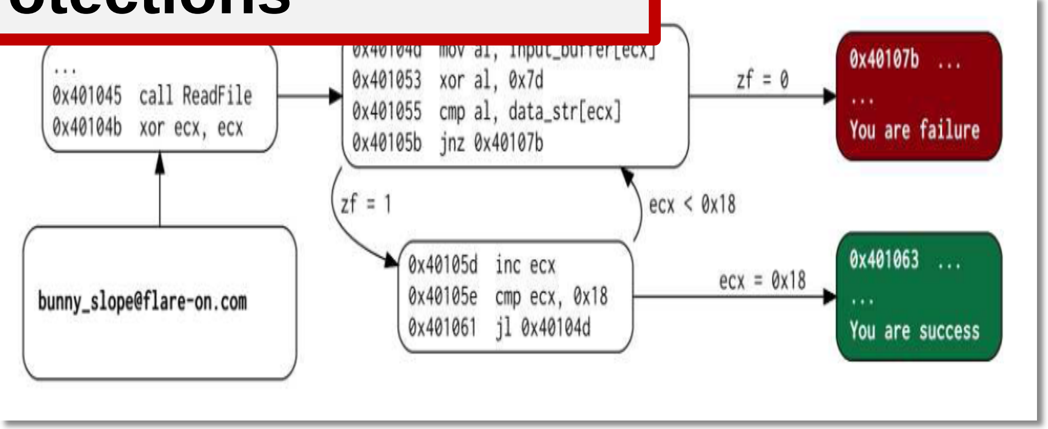
**Obfuscatio**

**hard to reve**

- self-modif
- encryption
- virtualization
- code overlapping
- opaque predicates
- callstack tampering
- …

**Goal: help malware comprehension**

- **Identify and simplify protections**
- **Ideal = revert protections**



# EXAMPLE: OPAQUE PREDICATE

**Constant-value predicates**

(always true, always false)

- dead branch points to spurious code
- goal = waste reverser time & efforts

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>
```

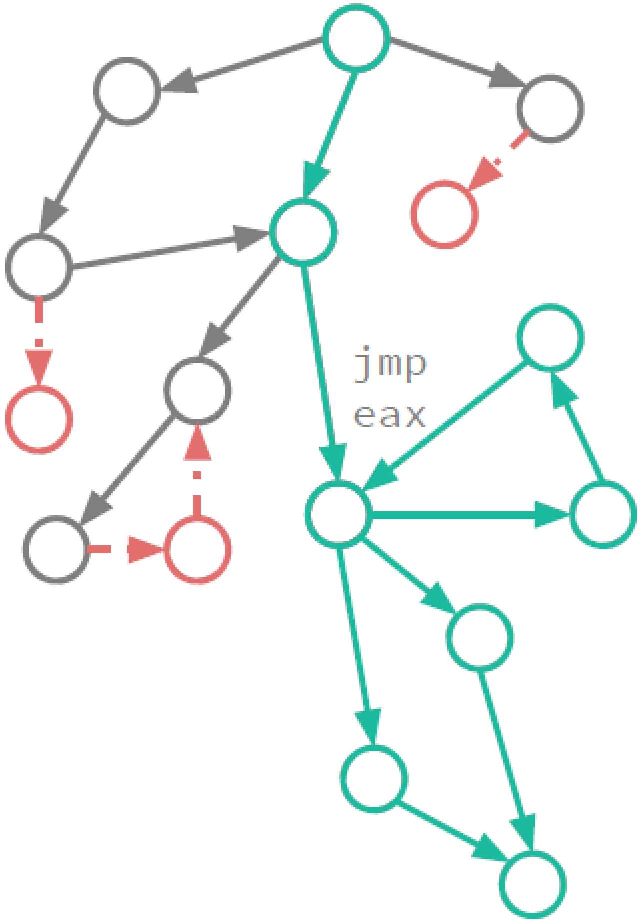
# EXAMPLE: STACK TAMPERING

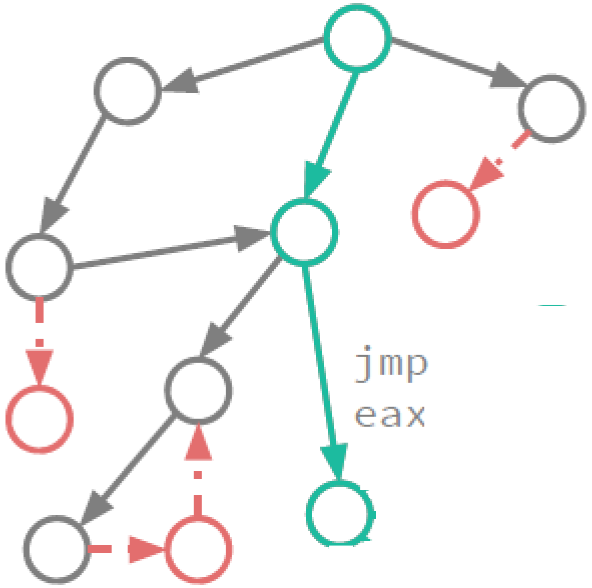
**Alter the standard compilation scheme:**
**ret do not go back to call**

- hide the real target
- return site may be spurious code

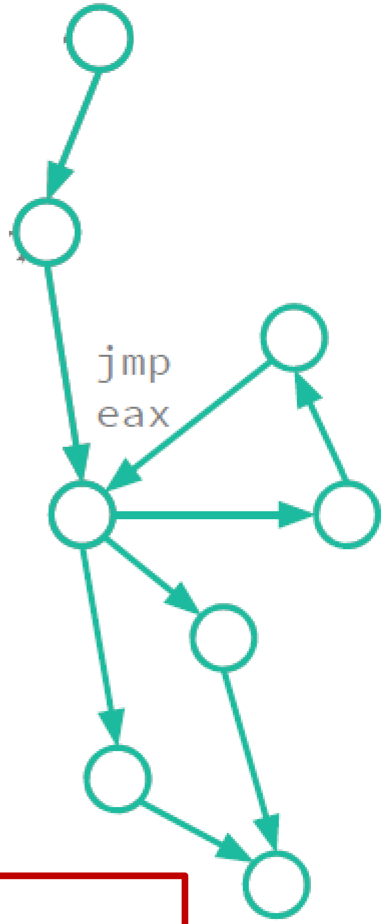
| address | instr |
|---|---|
| 80483d1 | call +5 |
| 80483d6 | pop edx |
| 80483d7 | add edx, 8 |
| 80483da | push edx |
| 80483db | ret |
| 80483dc | .byte{invalid} |
| 80483de | [...] |

# STANDARD DISASSEMBLY TECHNIQUES ARE NOT ENOUGH




**Static analysis**
- too fragile vs obfuscation
- junk instr, missed instr.

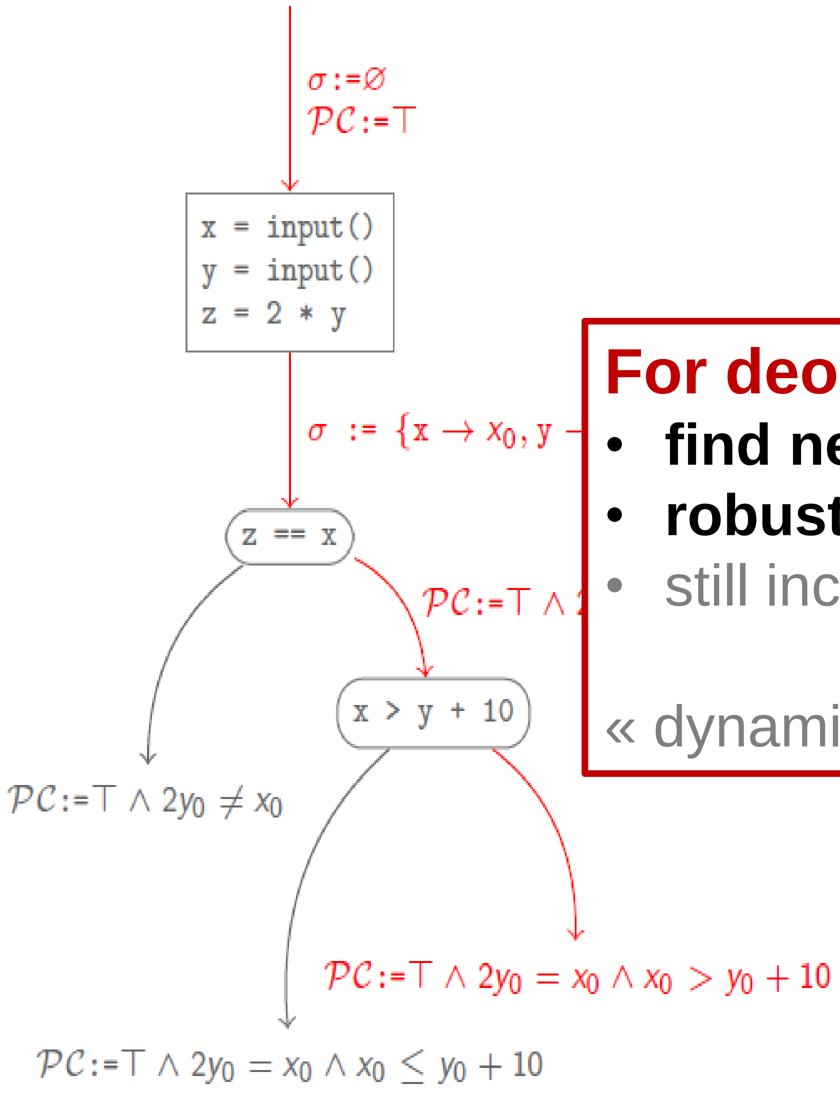



**Dynamic analysis**
- robust vs obfuscation
- too incomplete

# DYNAMIC SYMBOLIC EXECUTION CAN HELP (Debray, Kruegel, …)

```
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 the program
- automatically find input that follows the path
- then, iterate over all paths




**For deobfuscation**

- **find new real paths**
- **robust**
- still incomplete

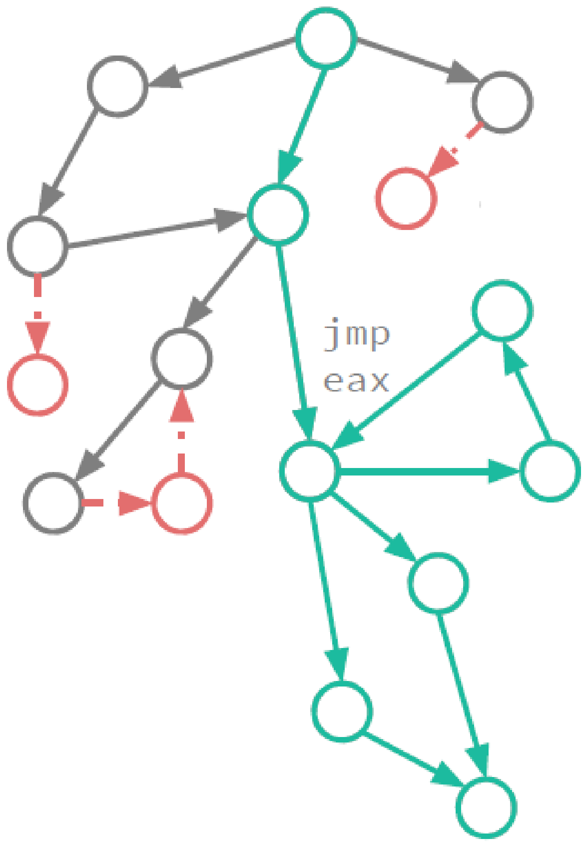
« dynamic analysis on steroids »

# YET … WHAT ABOUT INFEASIBILITY QUESTIONS?

**Prove that something is always true (resp. false)**

**Many such issues in reverse**
- **is a branch dead?**
- **does the ret always return to the call?**
- **have i found all targets of a dynamic jump?**

**And more**
- **does this malicious ret always go there?**
- **does this expression always evaluate to 15?**
- **does this self-modification always write this opcode?**
- **does this self-modification always rewrite this instr.?**
- **…**




**Not addressed by DSE**
- **Cannot enumerate all paths**

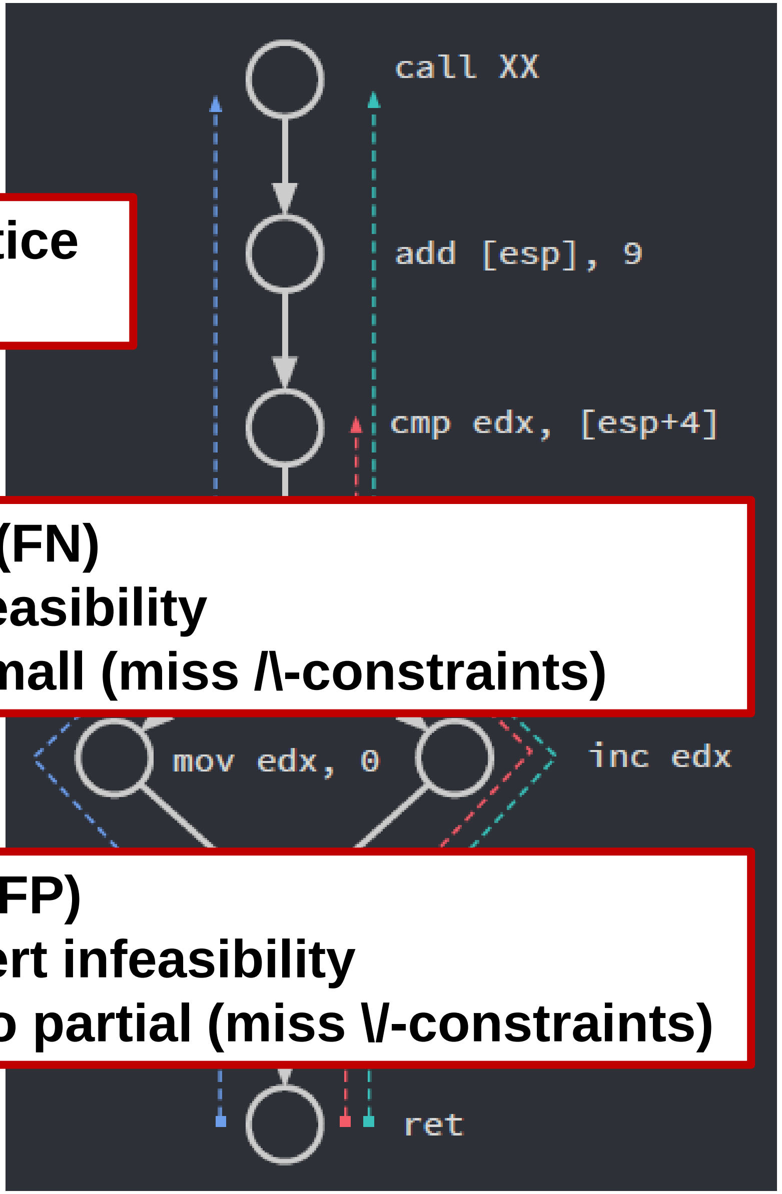
# OUR PROPOSAL: BACKWARD-BOUNDED SYMBOLIC EXECUTION

**Insight 1: symbolic reasoning**
- precision
- But: need finite #paths

**Insight 2: backward-bounded**
- pre_k(c)=0 => c is infeasible
- finite #paths
- efficient, depends on k
- But: backward on jump eax?

**Insight 3: dynamic partial CFG**
- solve (partially) dyn. jumps
- robustness

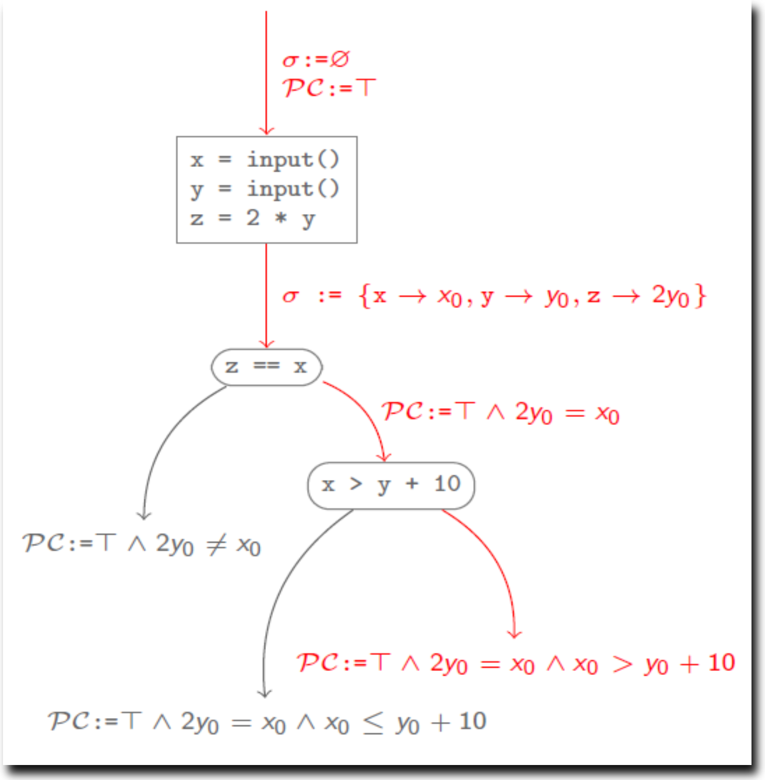
**Low FP/FN rates in practice**
- ground truth xp

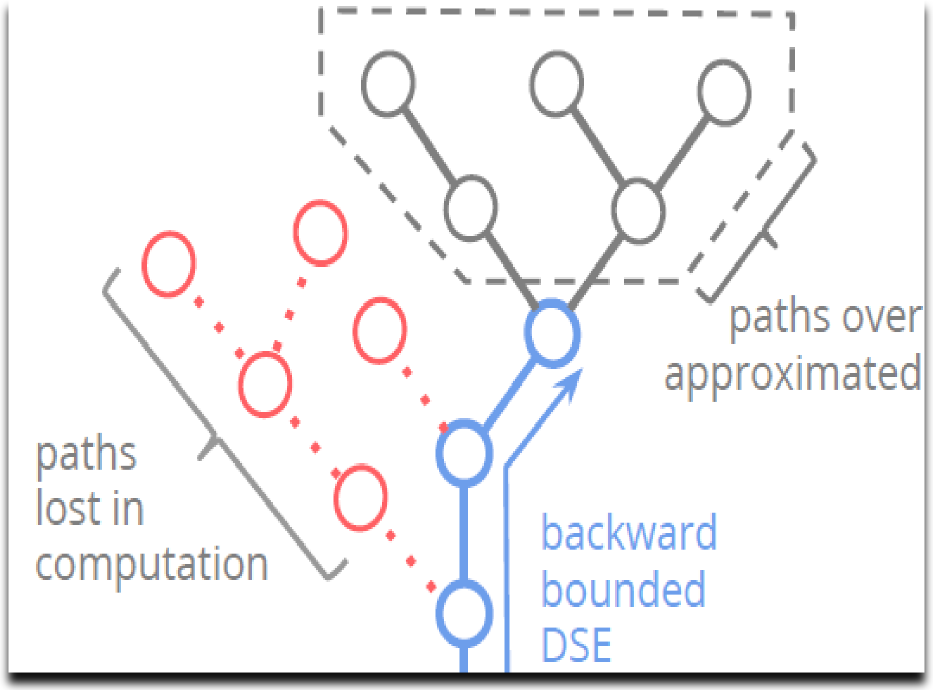



**False negative (FN)**
- can miss infeasibility
- why: k too small (miss /\-constraints)

**False positive (FP)**
- wrongly assert infeasibility
- why: CFG too partial (miss \/-constraints)

# FORWARD & BACKWARD SYMBOLIC EXECUTION




| | (forward) DSE | bb-DSE |
|---|---|---|
| feasibility queries | 🟢 | 🔴 |
| infeasibility queries | 🔴 | 🟢 |
| scale | 🟠 | 🟢 |

# EXPERIMENTAL EVALUATION

- **Controlled experiments** (ground truth) ⇒ **precision**
- **Large-scale experiment: packers** ⇒ **scalability, robustness**
- **Case-study: X-tunnel malware** ⇒ **usefulness**

# CONTROLLED EXPERIMENTS

- **Goal = assess the precision of the technique**
  - ground truth value

- **Experiment 1: opaque predicates (o-llvm)**
  - 100 core utils, 5x20 obfuscated codes
  - k=16: 3.46% error, no false negative
  - robust to k
  - efficient: 0.02s / query

- **Experiment 2: stack tampering (tigress)**
  - 5 obfuscated codes, 5 core utils
  - almost all genuine ret are proved (no false positive)
  - many malicious ret are proved « single-targets »

| k | OP (5556) | | Genuine (5183) | | TO | Error rate | Time | avg/query |
|---|---|---|---|---|---|---|---|---|
| | ok | miss (FN) | ok | miss (FP) | | (FP+FN)/Tot (%) | (s) | (s) |
| 2 | 0 | 5556 | 5182 | 1 | 0 | 51.75 | 89 | 0.008 |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 96 | 0.009 |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 120 | 0.011 |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | **152** | **0.014** |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | **197** | **0.018** |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | **272** | **0.025** |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | **384** | **0.036** |
| 32 | 5552 | 4 | 4579 | 604 | 25 | 5.66 | 699 | 0.065 |
| 40 | 5548 | 8 | 4523 | 660 | 39 | 6.22 | 1145 | 0.107 |
| 50 | 5544 | 12 | 4458 | 725 | 79 | 6.86 | 2025 | 0.189 |

- **Very precise résults**
- **Seems efficient**

| Sample | runtime genuine | | | runtime violation | | |
|---|---|---|---|---|---|---|
| | `#ret` † | proved genuine | proved a/d | `#ret` † | proved a/d | proved single |
| *obfuscated programs* | | | | | | |
| simple-if | 6 | 6 | 6/0 | 9 | 0/0 | 8 |
| bin-search | 15 | 15 | 15/0 | 25 | 0/0 | 24 |
| bubble-sort | 6 | 6 | 6/0 | 15 | 0/1 | 13 |
| mat-mult | 31 | 31 | 31/0 | 69 | 0/0 | 68 |
| huffman | 19 | 19 | 19/0 | 23 | 0/3 | 19 |
| *non-obfuscated programs* | | | | | | |
| ls | 30 | 30 | 30/0 | 0 | - | - |
| dir | 35 | 35 | 35/0 | 0 | - | - |
| mktemp | 21 | 20 | 20/0 | 0 | - | - |
| od | 21 | 21 | 21/0 | 0 | - | - |
| vdir | 49 | 43 | 43/0 | 0 | - | - |

# CASE-STUDY: PACKERS


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

The technique scale on significant traces

Many true positives. Some packers are using it intensively

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

**Packers: legitimate software protection tools**
**(basic malware: the sole protection)**

# CASE-STUDY: PACKERS (fun facts)

Several of the tricks detected by the analysis


| OP in ACProtect | | |
|---|---|---|
| 1018f7a | js | 0x1018f92 |
| 1018f7c | jns | 0x1018f92 |

(and all possible variants ja/jbe, jp/jnp, jo/jno..)

| CST in ACProtect | | |
|---|---|---|
| 1004328 | call | 0x1004318 |
| 1004318 | add | [esp], 9 |
| 100431c | ret | |

| OP in Armadillo | | |
|---|---|---|
| 10330ae | xor | ecx, ecx |
| 10330b0 | jnz | 0x10330ca |

| CST in ACProtect | |
|---|---|
| 1001000 | push 16793600 |
| 1001005 | push 16781323 |
| 100100a | ret |
| 100100b | ret |

| CST in ASPack | | |
|---|---|---|
| 10043a9 | mov | [ebp+0x3a8], eax |
| 10043af | popa | 0x10043bb at runtime |
| 10043b0 | jnz | 0x10043ba |
| Enter SMC Layer 1 | | |
| 10043ba | push | 0x10011d7 |
| 10043bf | ret | |

OP (decoy) in ASPack

```
10040fe: mov bl, 0x0
10041c0: cmp bl, 0x1
1004103: jnz 0x1004163
```

0x10040ff at runtime

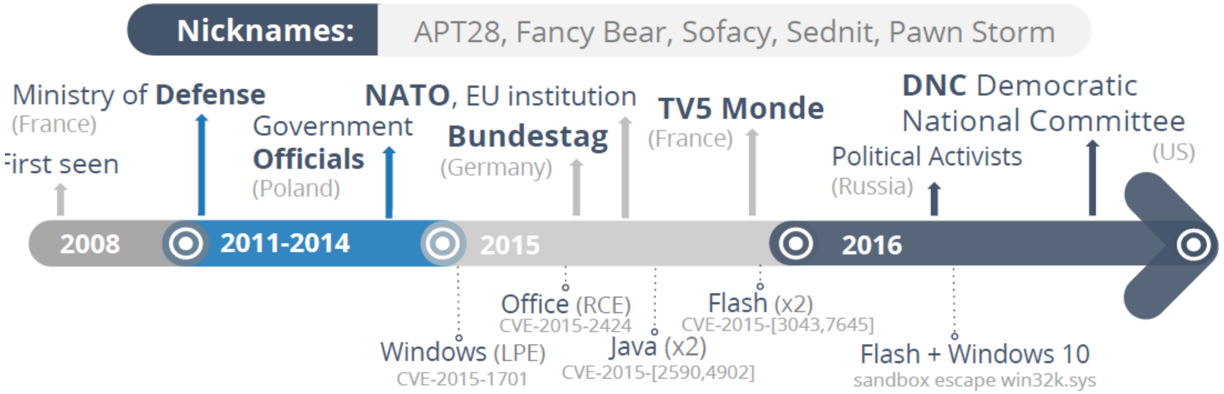
ZF = 0

```
1004163: jmp 0x100416d
[...]
```

ZF = 1

```
1004105: inc [ebp+0xec]
[...]
```

# CASE-STUDY: THE XTUNNEL MALWARE (part of DNC hack)





**Two heavily obfuscated samples**
- **Many opaque predicates**

**Goal: detect & remove protections**
- Identify 50% of code as spurious
- Fully automatic, < 3h

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

# CASE-STUDY: THE XTUNNEL MALWARE (fun facts)

- **Protection seems to rely only on opaque predicates**
- **Only two families of opaque predicates** $7y^2 - 1 \neq x^2$ $\frac{2}{x^2+1} \neq y^2 + 3$
- **Yet, quite sophisticated**
  - original OPs
  - interleaving between payload and OP computation
  - sharing among OP computations
  - possibly long dependencies chains (avg 8.7, upto 230)

# SECURITY ANALYSIS: COUNTER-MEASURES (and mitigations)

- **Long dependecy chains (evading the bound k)**
  - Not always requires the whole chain to conclude!
  - Can use a more flexible notion of bound (data-dependencies, formula size)

- **Hard-to-solve predicates (causing timeouts)**
  - A time-out is already a valuable information
  - Opportunity to find infeasible patterns (then matching), or signatures
  - Tradeoff between performance penalty vs protection focus
  - Note: must be input-dependent, otherwise removed by standard DSE optimizations

- **Anti-dynamic tricks (fool initial dynamic recovery)**
  - Can use the appropriate mitigations
  - Note: some tricks can be circumvent by symbolic reasoning

**Current state-of-the-art**
- **push the cat-and-mouse game further**
- **raise the bar for malware designers**

**Also**
- **« Probabilistic obfuscation »**
- **Covert channels**

# OUTLINE

- Why binary-level analysis?
- Some background on source-level formal methods
- The hard journey from source to binary
- A few case-studies
- Conclusion

# SUMMARY

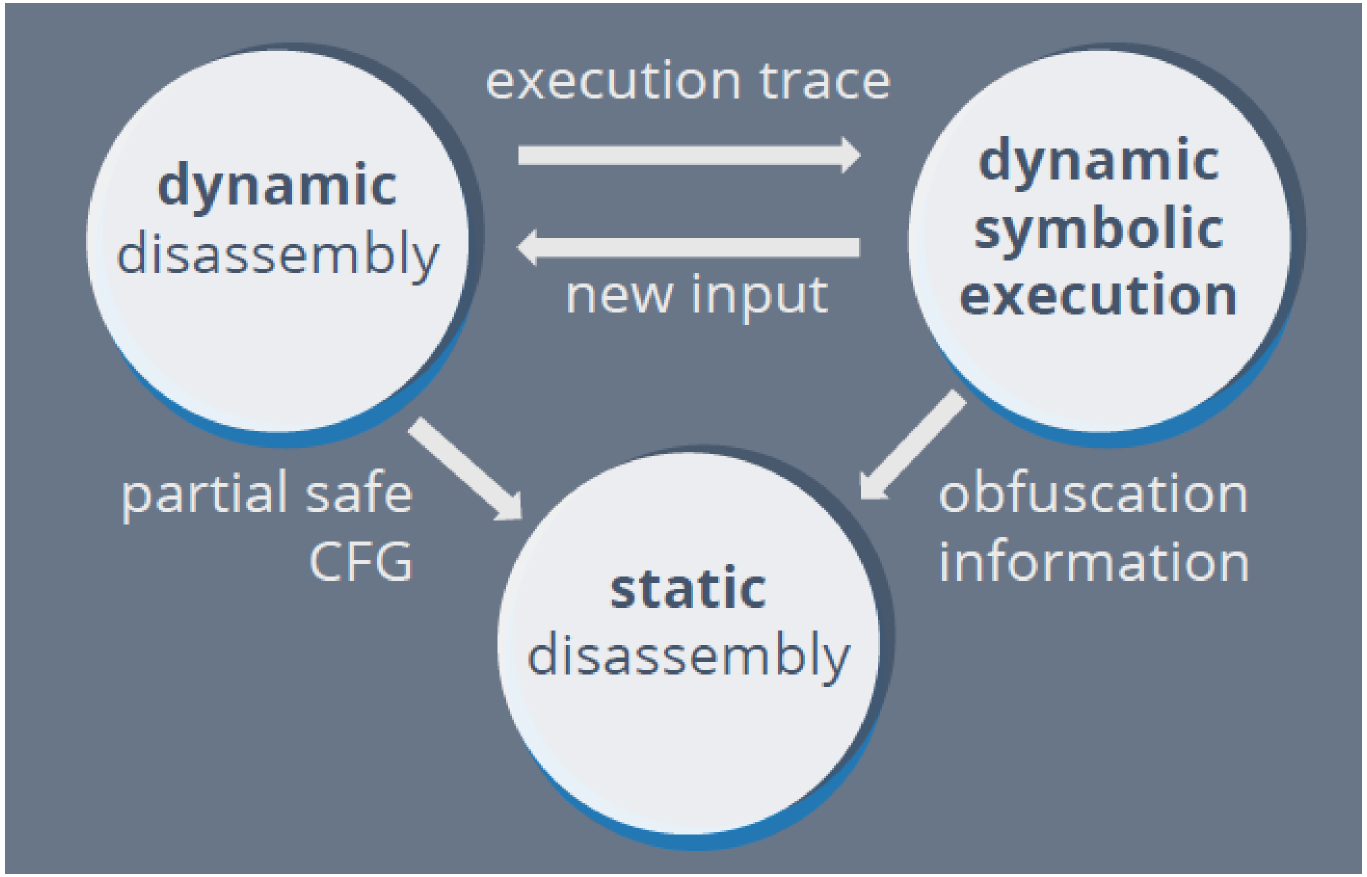
| | Feasibility | Infeasibility | Efficient | Robust |
|---|---|---|---|---|
| **Static syntactic** | X | X | OK | X |
| **Dynamic** | -- | X | OK | OK |
| **DSE** | OK | X | X | OK |
| **Static semantic** | X | OK | X | X |
| **BB-DSE** | X | OK (fp,fn) | OK | OK |

# CONCLUSION

- **Semantic analysis can change the game of binary-level security**
  - Current syntactic and dynamic methods are not enough
  - [complement existing approaches and help the expert, not replace everything]
  - Explore more, Prove invariance, Simplify

- **Yet, challenging to adapt from source-level safety-critical**
  - Need robustness, precision and scale!!
  - « Correct-enough » and « Complete-enough » are enough (room for better definition!)
  - DSE much easier to adapt than AI
  - **New challenges and variations, so much to do**

# FUTURE DIRECTION



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