Code-level Cyber-Security: Semantic attacks (program analysis)

Sébastien Bardin (CEA LIST)
Richard Bonichon (CEA LIST)

NOW

- Program analysis tools and methods
- What they are
- What they can do for security (reverse, vulnerabilities)
- Their strengths & limites

OUTLINE

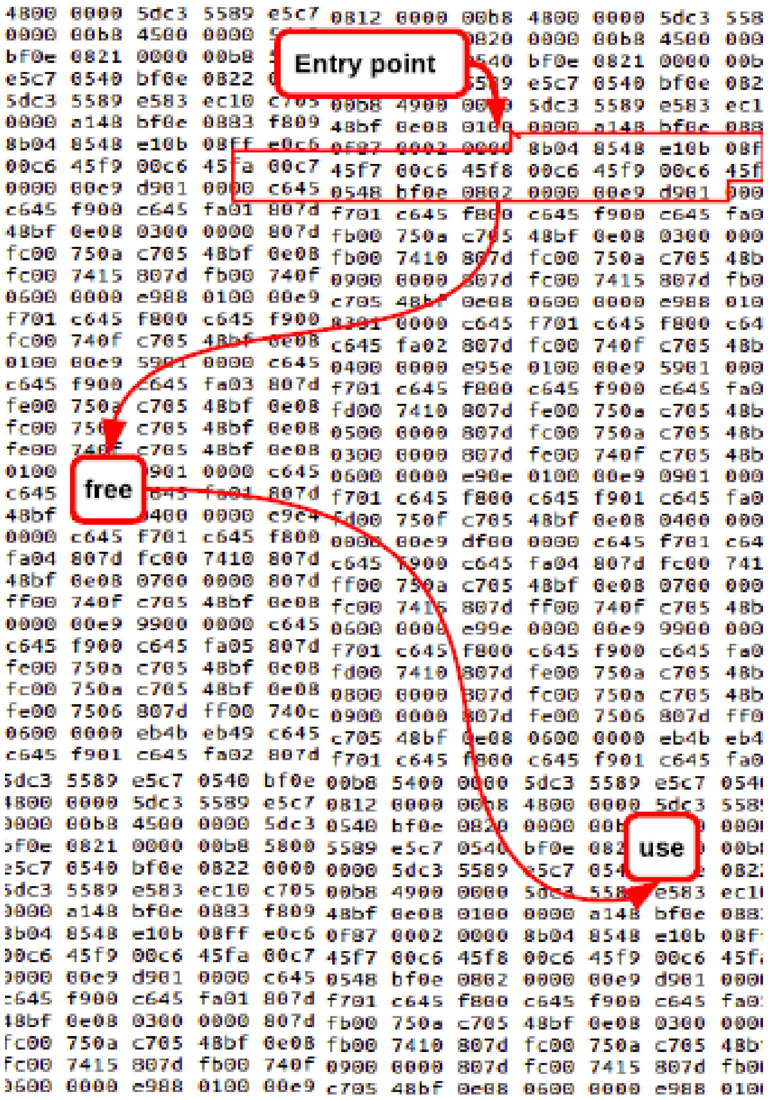
- Context
- What are formal methods?
- An overview of program analysis
- The hard journey from source to binary
- A few case-studies
- Discussion & Conclusion

EXAMPLE: VULNERABILITY DETECTION

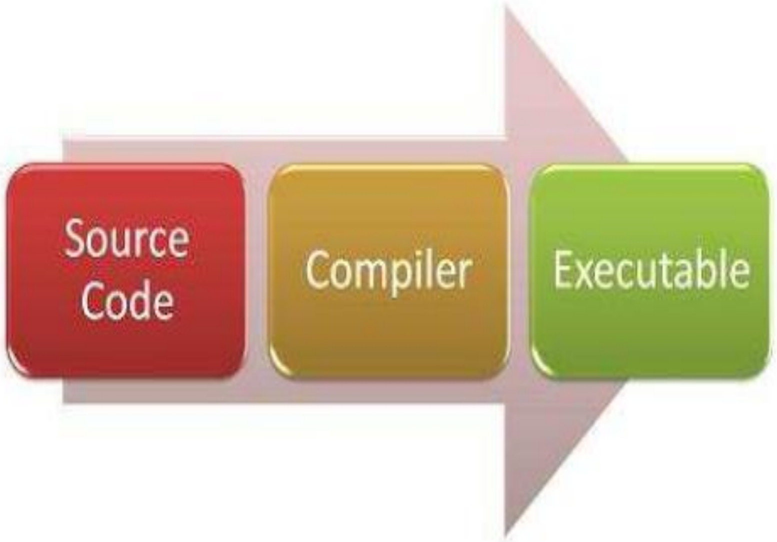
Find vulnerabilities before the bad guys

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

Find a needle in the heap !



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:

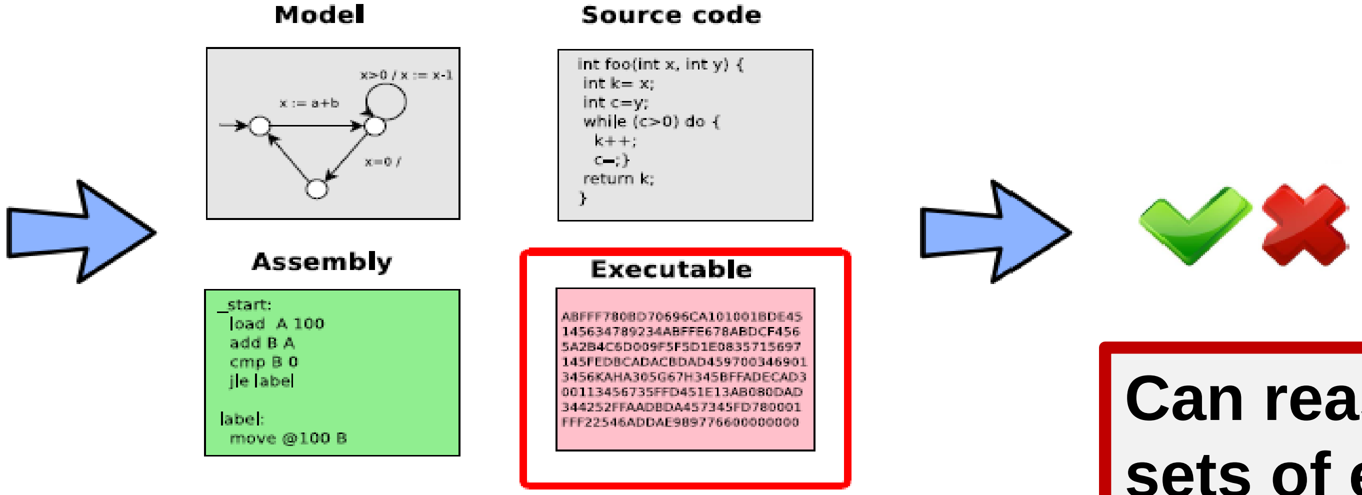
- **Check the code after compilation**

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

- **find rare events**
- **prove facts**

Advantages

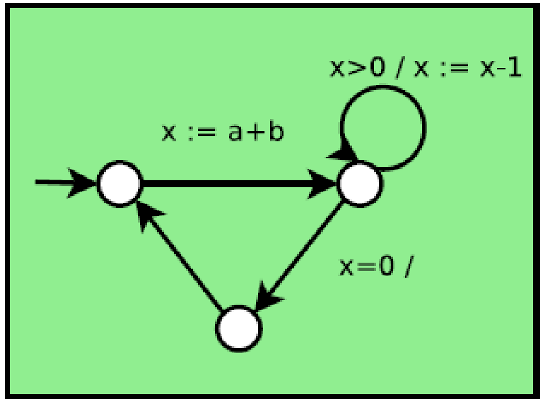
- more robust than syntactic
- more thorough than dynamic

Challenges

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

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
```

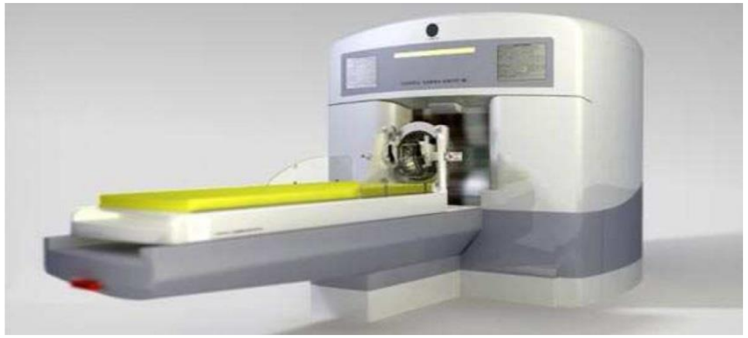
OUTLINE

- Context
- **What are formal methods?**
- An overview of program analysis
- The hard journey from source to binary
- A few case-studies
- Discussion & 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

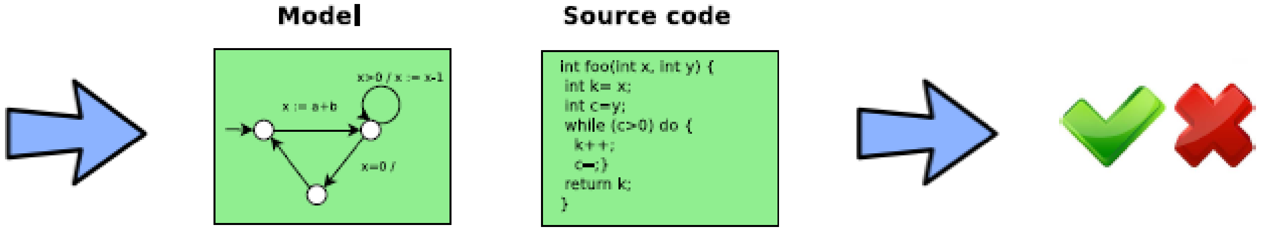
Testing can only reveal the presence of errors but never their absence.

- 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 : $M \models \varphi$

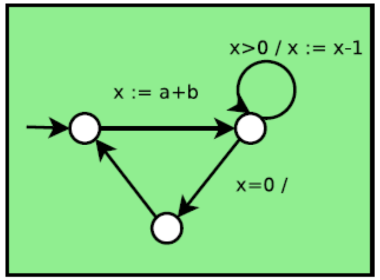
- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

Kind of properties

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

Input model?

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
```

A set of relevant behaviours

- **Reachable states**
- **Traces (finite or infinite)**
- **Execution Tree**
- **…**

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;
}
```

Specification?

Spec may be implicit
- **Good typing**
- **No runtime error**
- **…**

Examples from static analysis benchmarks

```
#define TAILLE_TAB 1024
int tab[TAILLE_TAB];

void f(void){
    int index;
    for (index = 0; index < TAILLE_TAB
; index++)
    {
        tab[index] = 0;
    }
    tab[index] = 1;
}
```

out of bounds access

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

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

integer overflow

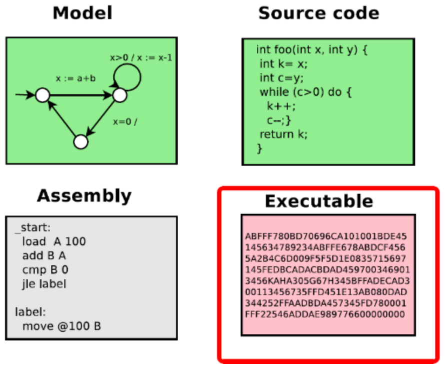
```
void main(void)
{
    int* p;
    *p = 42;
}
```

uninitialized pointer

```
void zdvs(int p)
{
    int i,j = 1;
    i = 1024 / (j-p);
}
```

division by zero

Algorithmic check (1)



$\models$

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

$\subset$

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

Algorithmic check (2)

Two key aspects of program analysis
- Mastering abstraction
- Fighting indecidability / intractability

Problem is often undecidable
- **Over-approximation**
- **Under-approximation**

// Witness?

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

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

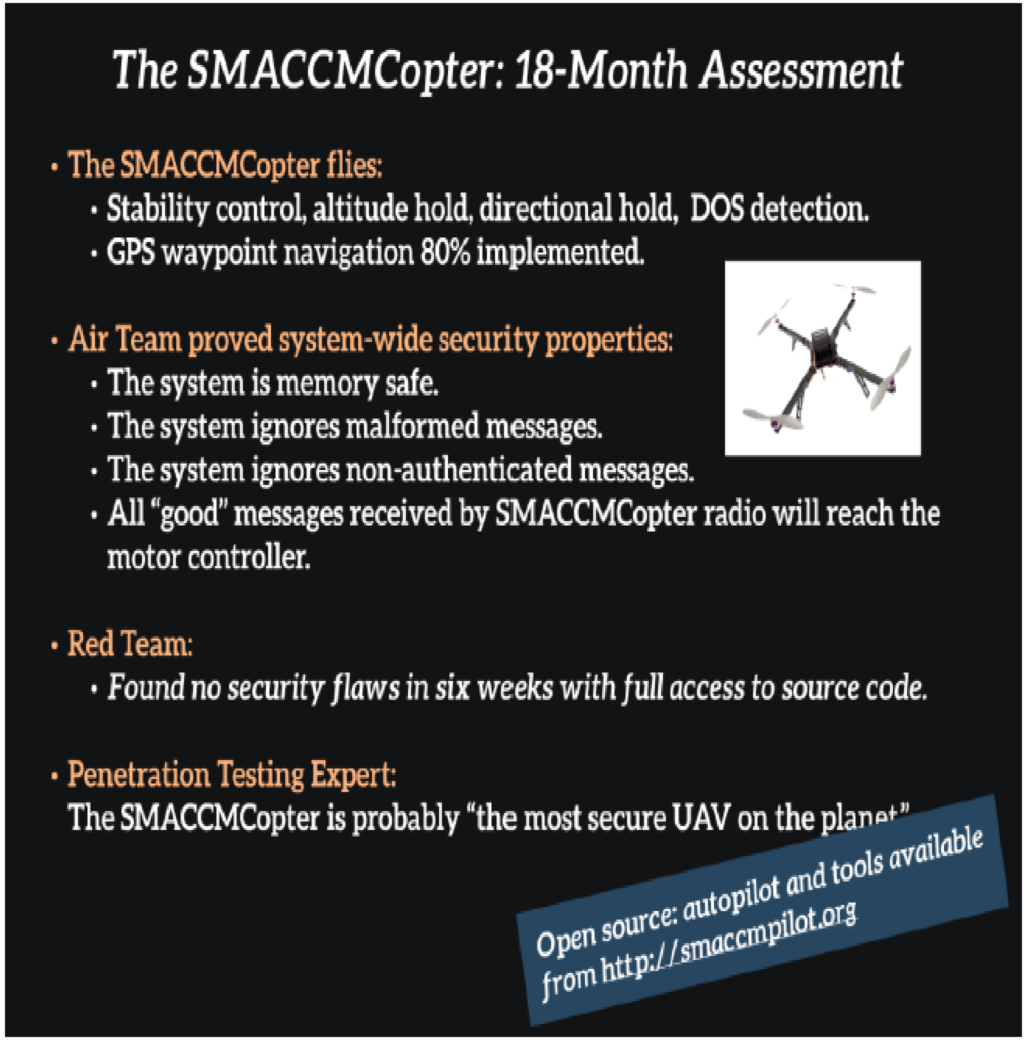
Verification of drivers [SDV]

- 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)

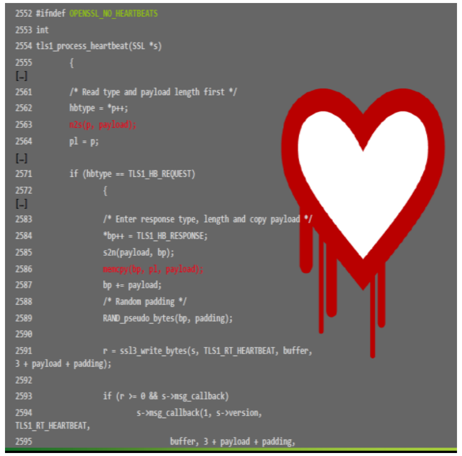
A DREAM COME TRUE … IN CERTAIN DOMAINS (3)

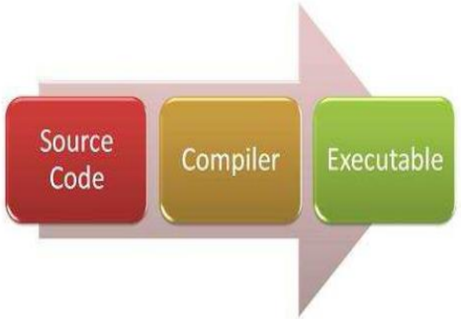


Formally-hardened drone
- **memory safety**
- **ignores bad messages**

Red team attack
- **6 weeks, access to source**
- **no security bug found**

Other successes





Compcert

Sel4

SAGE

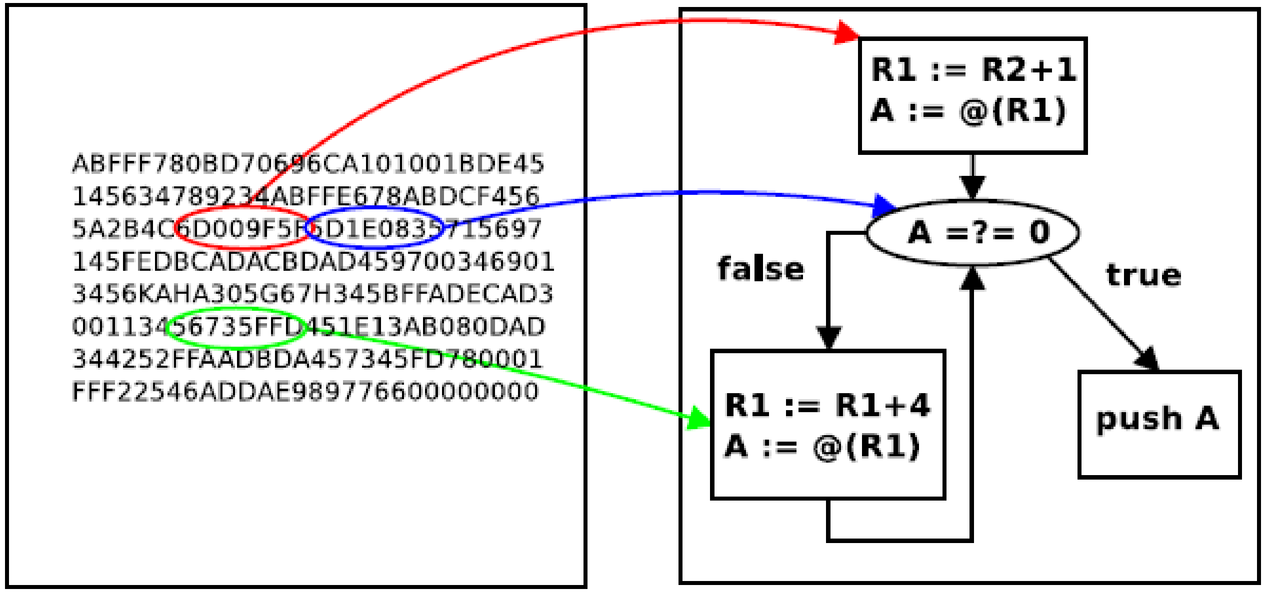
OUTLINE

- Context
- What are formal methods?
- **An overview of program analysis**
- The hard journey from source to binary
- A few case-studies
- Discussion & Conclusion

OUTLINE

- **An overview of program analysis**
  - Basic disassembly: linear sweep & recursive traversal
  - Program semantic
  - Basic static analysis: constant propagation
  - More complicated: domain propagation, taint, combination
  - Properties of program analysis

CHALLENGE: CORRECT DISASSEMBLY

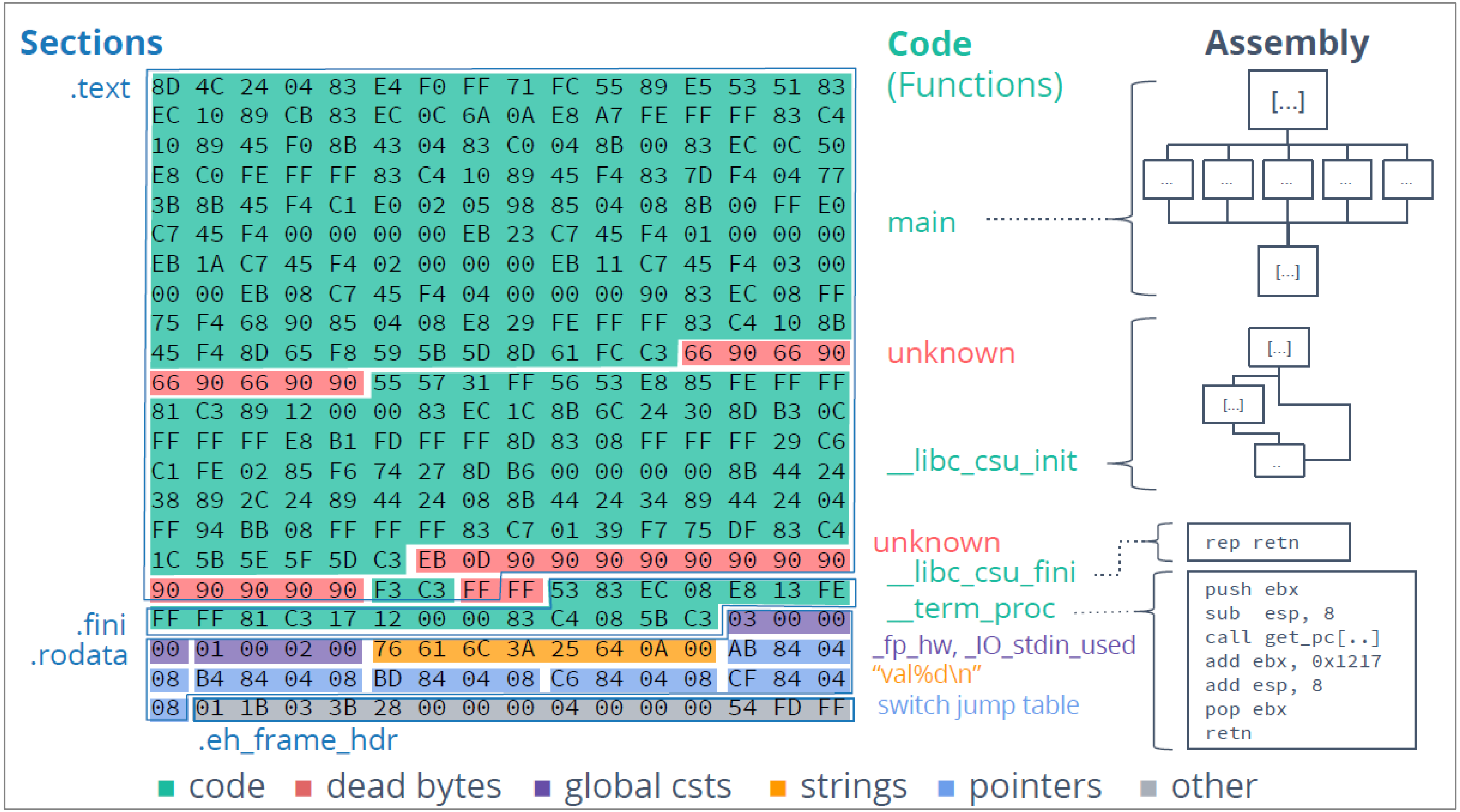


Basic reverse problem

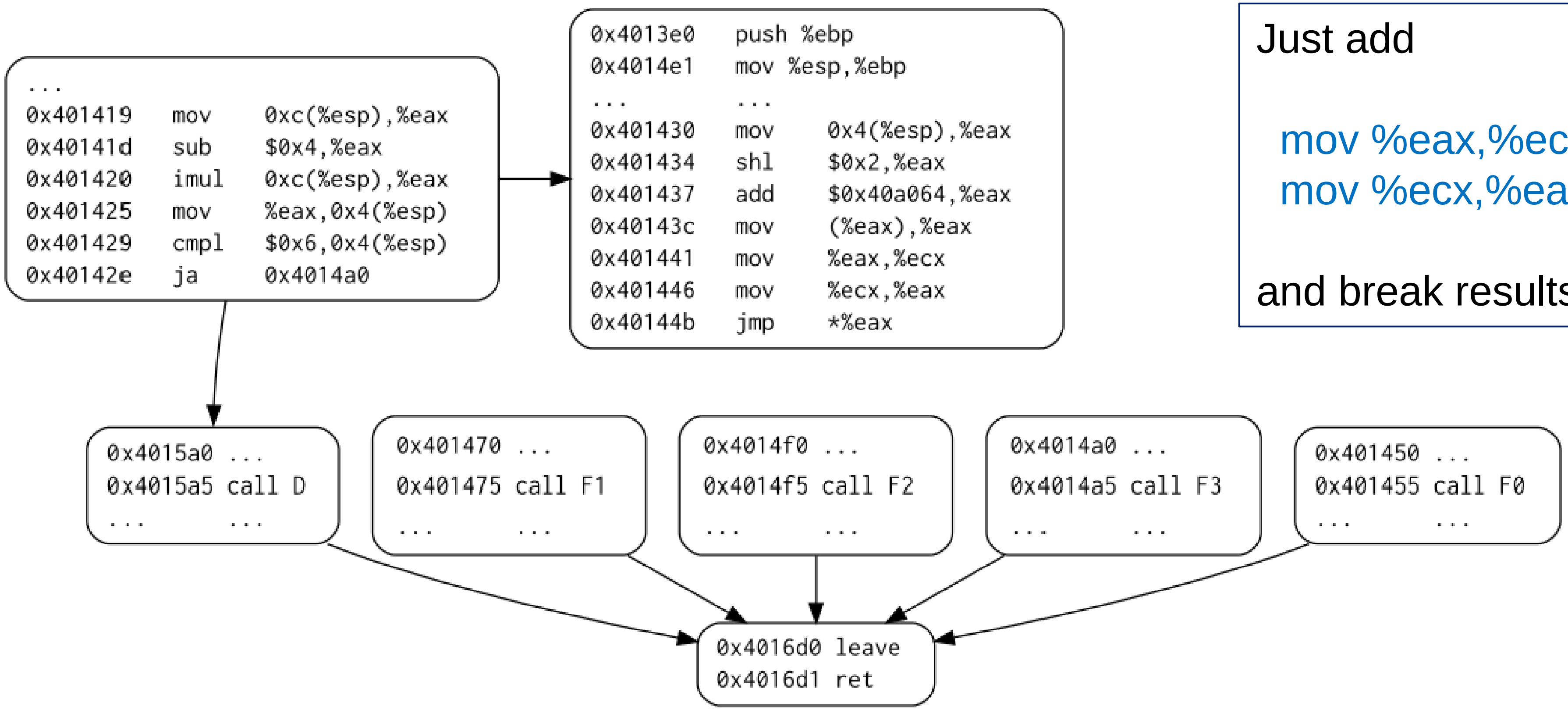
- **aka model recovery**
- **aka CFG recovery**

CAN BE TRICKY!

- code – data
- dynamic jumps (jmp eax)



STATE-OF-THE-ART TOOLS ARE NOT ENOUGH



Just add

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

and break results

With IDA

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

Basic disassembly <EXO>

- **Input : a binary code (map<addr -> byte>) and initial address addr_0**
- **Output : map <addr -> instr>**

- **Also function decode : (code, addr) -> (instr, size)**

- **Instr ::= operation | halt | jump k | ite(b,k,k') | jump x**

- **Write:**
  - *Linear sweep disassembly*
  - *Recursive disassembly*
  - *Combination*

- Correct?
- Complete?

Linear sweep

```
TODO := {addr_0}
Instr := {}                          // pairs (addr,instr)

while TODO <> empty do
          choose addr \in TODO;
          TODO := TODO - addr;
          (i,size) := decode(addr);
          Instr := Instr + (addr,i)
          if (addr+size+1  < code_limit) TODO := TODO + (addr+size+1) end if
end while

return Instr
```

Recursive traversal

```
TODO := {addr_0}
Instr := {}                              // pairs (addr,instr)

while TODO <> empty do
        choose addr \in TODO;
        TODO := TODO - addr;
        (i,size) := decode(addr);
        Instr := Instr + (addr,i)
        next :=
          match i with
                halt -> {}
                operation -> {addr+size+1}
                jump a' -> {a'}
                ite(f,a',a") -> {a',a"}
                jump EAX ->  ?????
          end match
        TODO := TODO + next
end while

return Instr
```

Combined disassembly?

OUTLINE

- **An overview of program analysis**
  - Basic disassembly: linear sweep & recursive traversal
  - Program semantic
  - Basic static analysis: constant propagation
  - More complicated: domain propagation, taint, combination
  - Properties of program analysis

Semantic: an idea

Petit langage impératif à valeurs entières

Expressions *Expr*

$e ::=$	x	variable, $x \in Var$
$\mid$	z	constante entière, $z \in \mathbb{Z}$
$\mid$	$e + e \mid - e \mid e * e \mid e / e$	expressions arithmétiques
$\mid$	$e = e \mid e <= e \mid\ !\ e \mid (e)$	expressions booléennes

Instructions *Instr*

$i ::=$	x := e	affectation
$\mid$	if e then i else i fi	conditionnelle
$\mid$	while e do i done	boucle
$\mid$	$i; i$	séquence
$\mid$	skip	sans effet

Semantic: an idea (2)

- la syntaxe décrit seulement l'ensemble des programmes qu'on peut d'écrire
- elle ne précise pas le sens de chaque entité syntaxique (ici les expressions et les instructions)
- c'est le rôle de la sémantique

EXO : écrire la sémantique du langage précédent

Cas simple : juste des variables entières a, b, c, d, ...

Sémantique opérationnelle

- une configuration du système :
  - . $c : \text{Var} \mapsto \mathbb{N}$
- une affectation modifie l'état mémoire :
  - . ex : z := a+b
  - . $c \xrightarrow{z:=a+b} c'$ ssi $c' = c[z \leftarrow c[a] + c[b]]$
- un branchement teste l'état mémoire

<Correction>

OUTLINE

- **An overview of program analysis**
  - Basic disassembly: linear sweep & recursive traversal
  - Program semantic
  - Basic static analysis: constant propagation
  - More complicated: domain propagation, taint, combination, slice
  - Properties of program analysis

Dataflow analysis & constant propagation

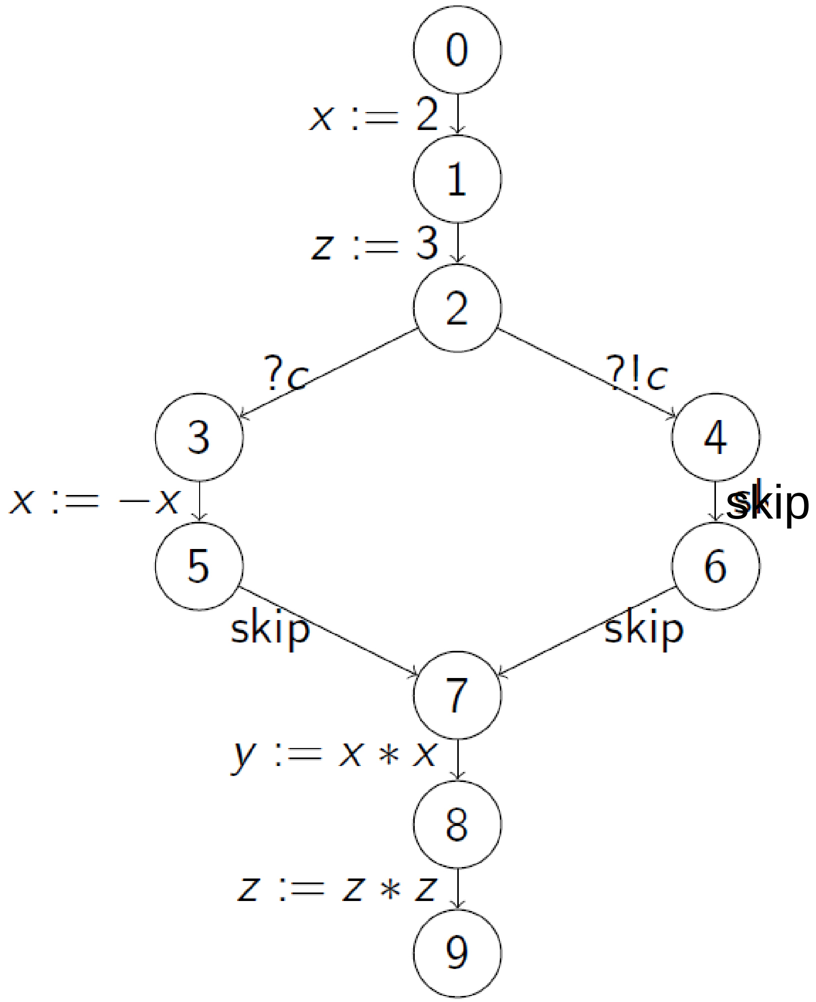
Vue informelle

- on propage des informations le long du graphe de contrôle (fonction de transition)
- lorsqu'un sommet admet plusieurs prédécesseurs (sommet de jonction), on considère la réunion des informations propagés
- on termine lorsque la propagation des informations ne fait plus "augmenter" le résultat

Constant propagation:

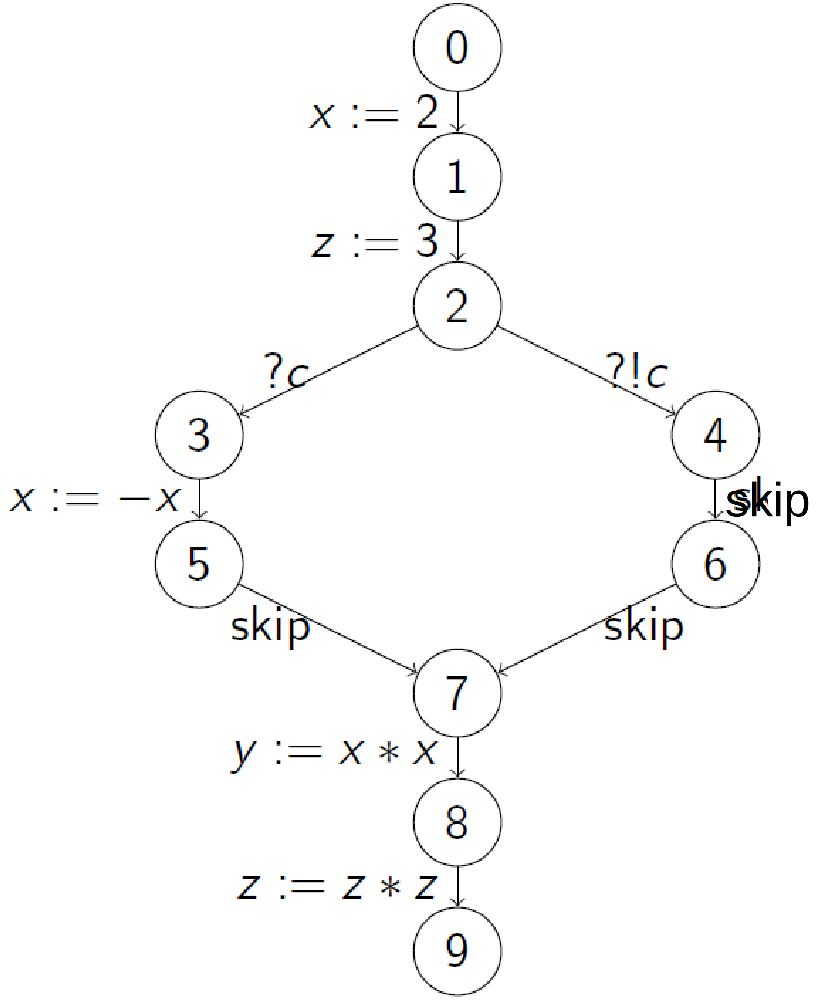
- At instruction i, x is sure to be equal to value k **for each program execution** « x=k »
- We do not know « x=T »
- Initialization: « x = _ » // still not reached

EXO: constant propagation in practice



ABSTRACT INTERPRETATION IN…

- Precision loss in practice
- Line 7 :
  - X \in {-2,2}
  - Becomes X \in T

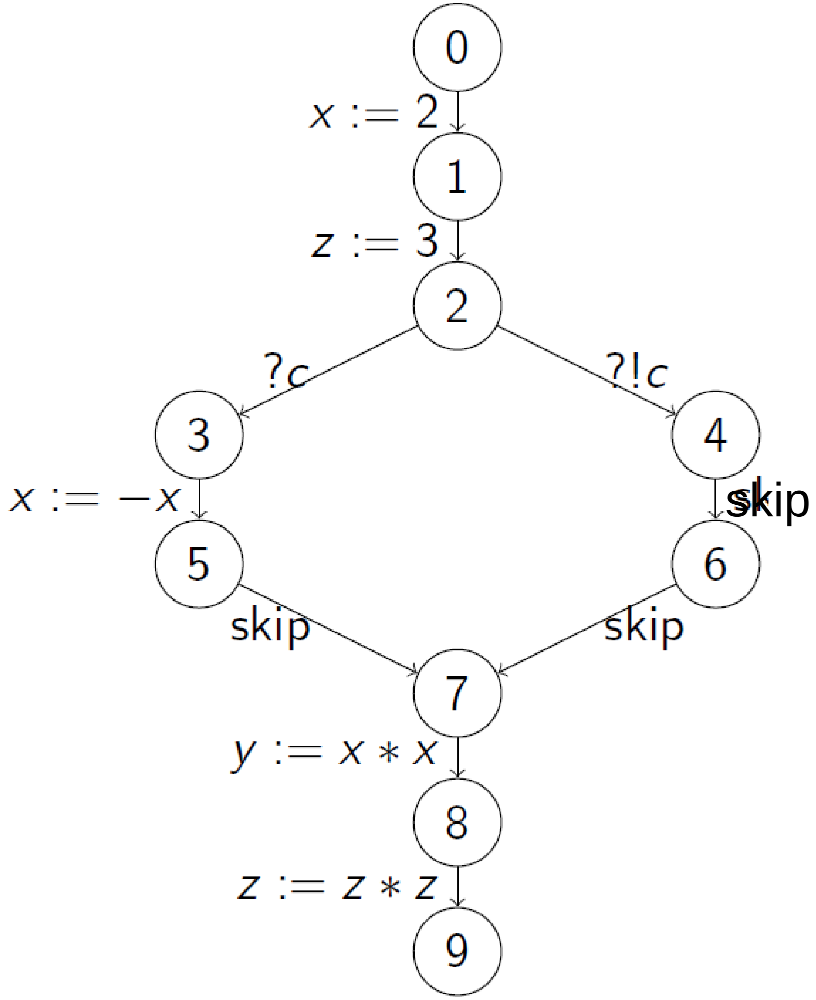


no…				
1	⊤	2	⊤	⊤
2	⊤	2	⊤	3
3	⊤	2	⊤	3
4	0	2	⊤	3
5	⊤	−2	⊤	3
6	0	2	⊤	3
7	⊤	−⊤2	⊤	3
8	⊤	⊤	⊤	3
9	⊤	⊤	⊤	9

OUTLINE

- **An overview of program analysis**
  - Basic disassembly: linear sweep & recursive traversal
  - Program semantic
  - Basic static analysis: constant propagation
  - More complicated: domain propagation, taint, combination, slice
  - Properties of program analysis

ABSTRACT INTERPRETATION IN



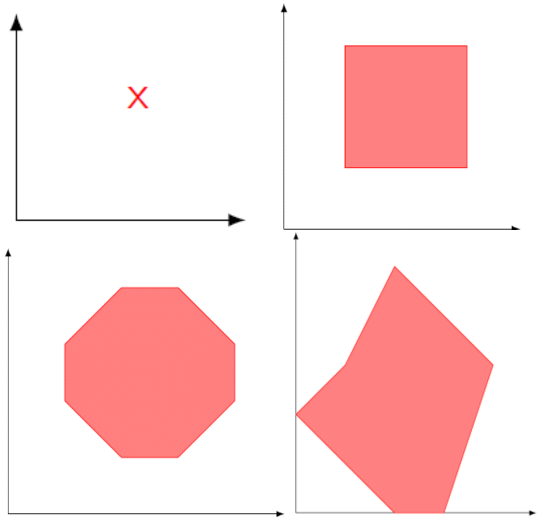
- Precision loss in practice
- Line 7 :
  - X \in {-2,2}
  - X \in -2..2

1	⊤	2	⊤	⊤
2	⊤	2	⊤	3
3	⊤	2	⊤	3
4	0	2	⊤	3
5	⊤	−2	⊤	3
6	0	2	⊤	3
7	⊤	−⊤2	⊤	3
8	⊤	⊤	⊤	3
9	⊤	⊤	⊤	9

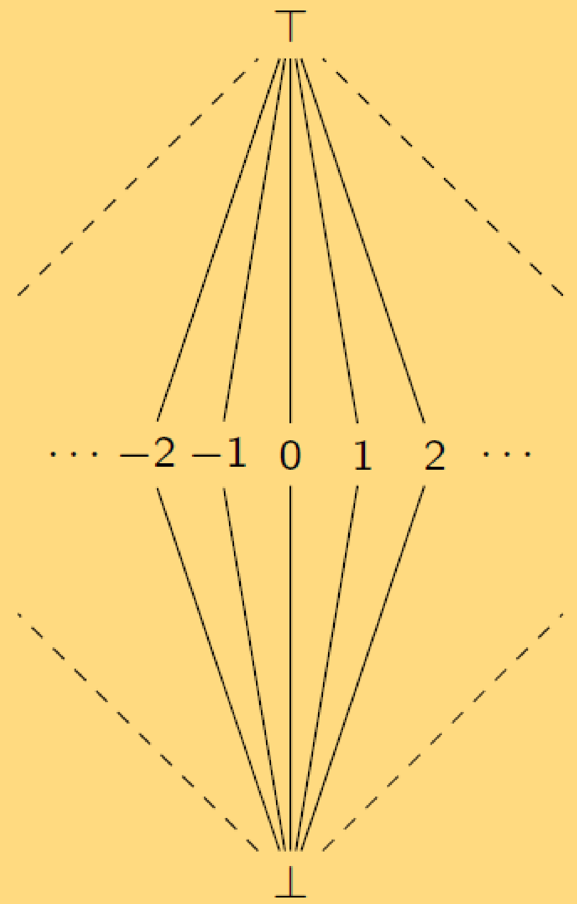
ABSTRACT INTERPRETATION (2)

Framework : abstract interpretation

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

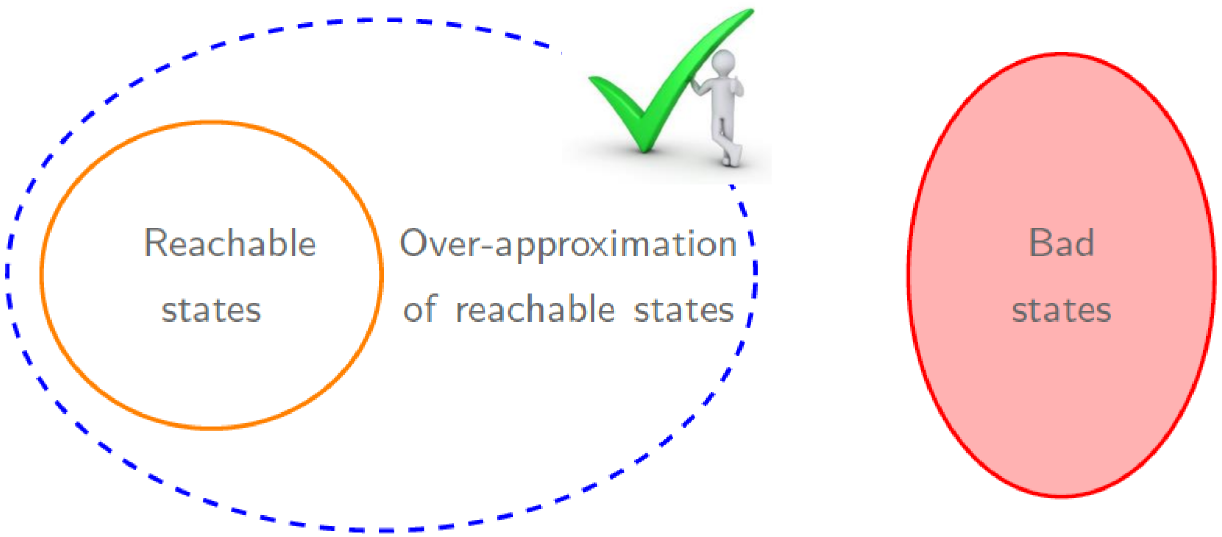


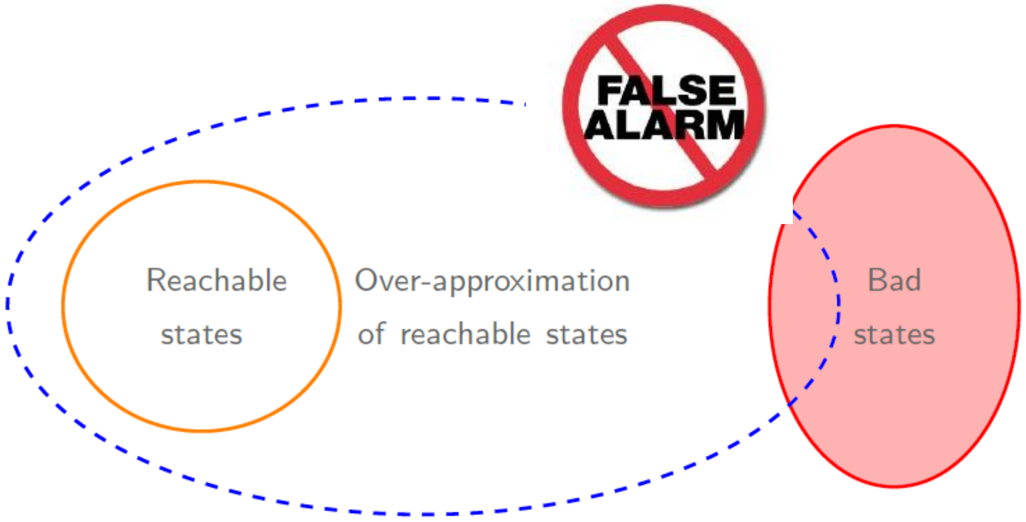
Generalize constant propagation



ABSTRACT INTERPRETATION

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





KEY INGREDIENTS

Abstract transfert: x := a+10

- **a == {0} => x == {10}**
- **a == [0..5] => x == [10..15]**
- **a == T => x == T**

Merge

- **[a == 5] ∏ [a == 5] => [a == 5]**
- **[a == 5] ∏ [a == 8] => [a == T]**
- **[a == 5] ∏ [a == 8] => [a == 5..8]**

Other key points

- **Widening ensures termination**
- **Computation in the abstract is upper-approximated (correctness)**
- **Galois connexion ensures best abstraction**

Framework : abstract interpretation

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

EXOS & discussions

- Termination?

- **Rules for interval propagation**
- **Rules for tainting**
- **Combining interval & tainting**

- **Slicing**

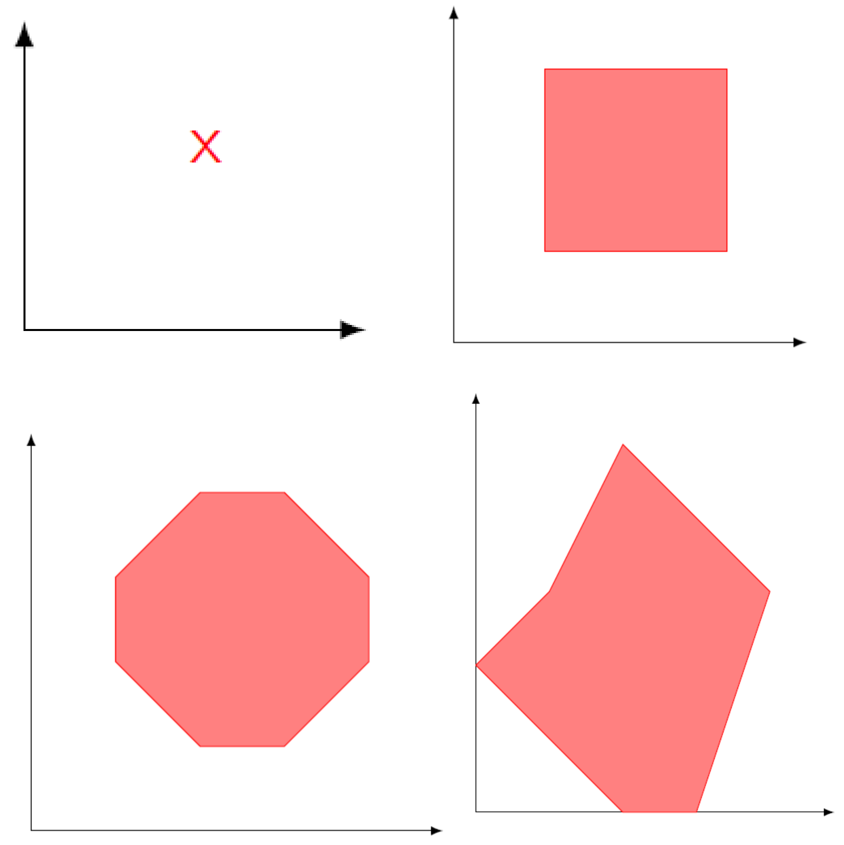
- **Static vs dynamic**

- Correct?
- Complete?

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

OUTLINE

- **An overview of program analysis**
  - Basic disassembly: linear sweep & recursive traversal
  - Program semantic
  - Basic static analysis: constant propagation
  - More complicated: domain propagation, taint, combination, slice
  - Properties of program analysis

PROPERTIES of PROGRAM ANALYSIS

- Correctness
- Completeness
- Efficiency
- Robustness

OUTLINE

- Context
- What are formal methods?
- An overview of program analysis
- **The hard journey from source to binary**
- A few case-studies
- Discussion & Conclusion

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

Low-level semantics of control

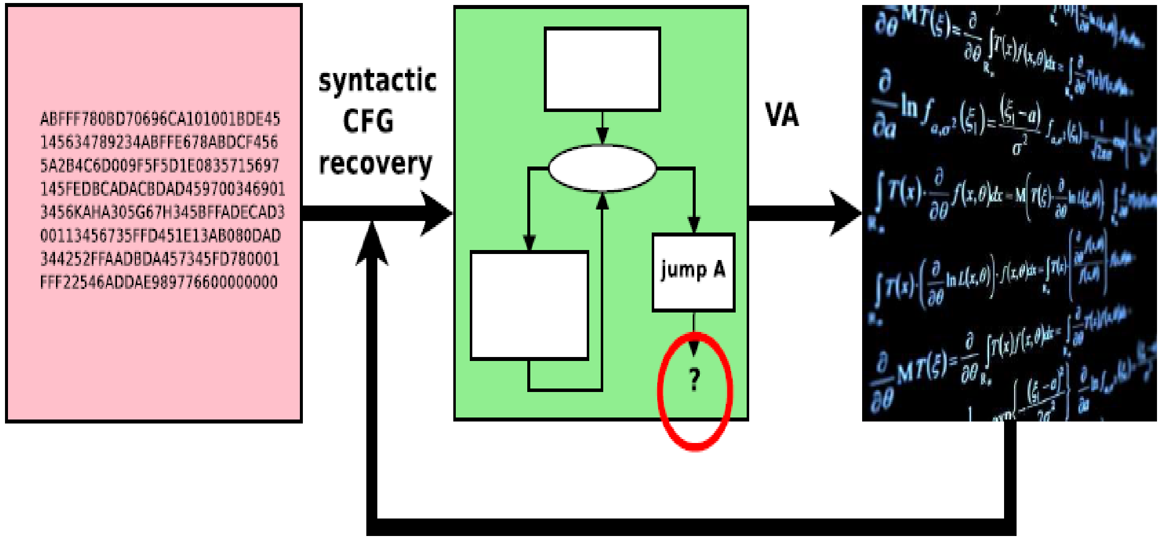
- no distinction data / instructions, dynamic jumps (`jmp eax`)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
- ▶ violate an implicit prerequisite for most formal techniques

Diversity of architectures and instruction sets

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

Wanted
- robustness
- precision
- scale

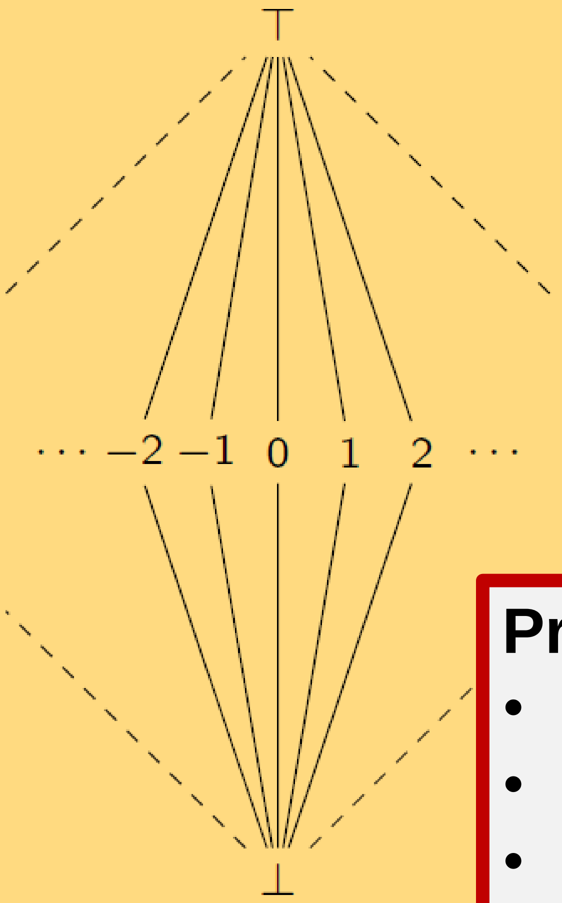
<apparté> STATIC SEMANTIC ANALYSIS IS VERY VERY HARD ON BINARY CODE



Framework : abstract interpretation

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

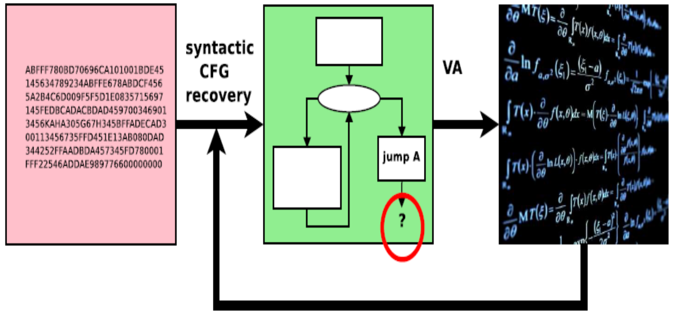
Generalize constant propagation



Problems

- Jump eax
- memory
- Bit resoning

ABSTRACT INTERPRETATION IS VERY VERY HARD ON BINARY CODE

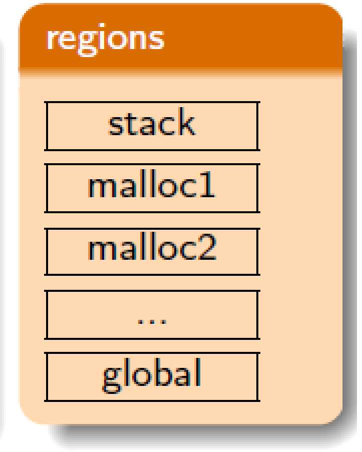


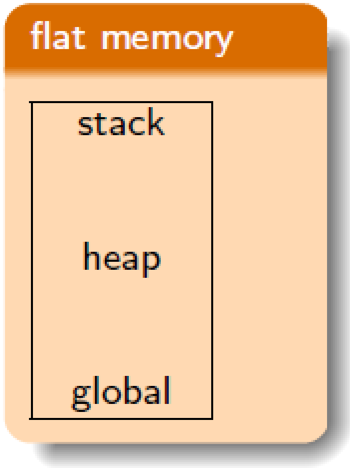
Problems

- Jump eax
- memory
- Bit resoning

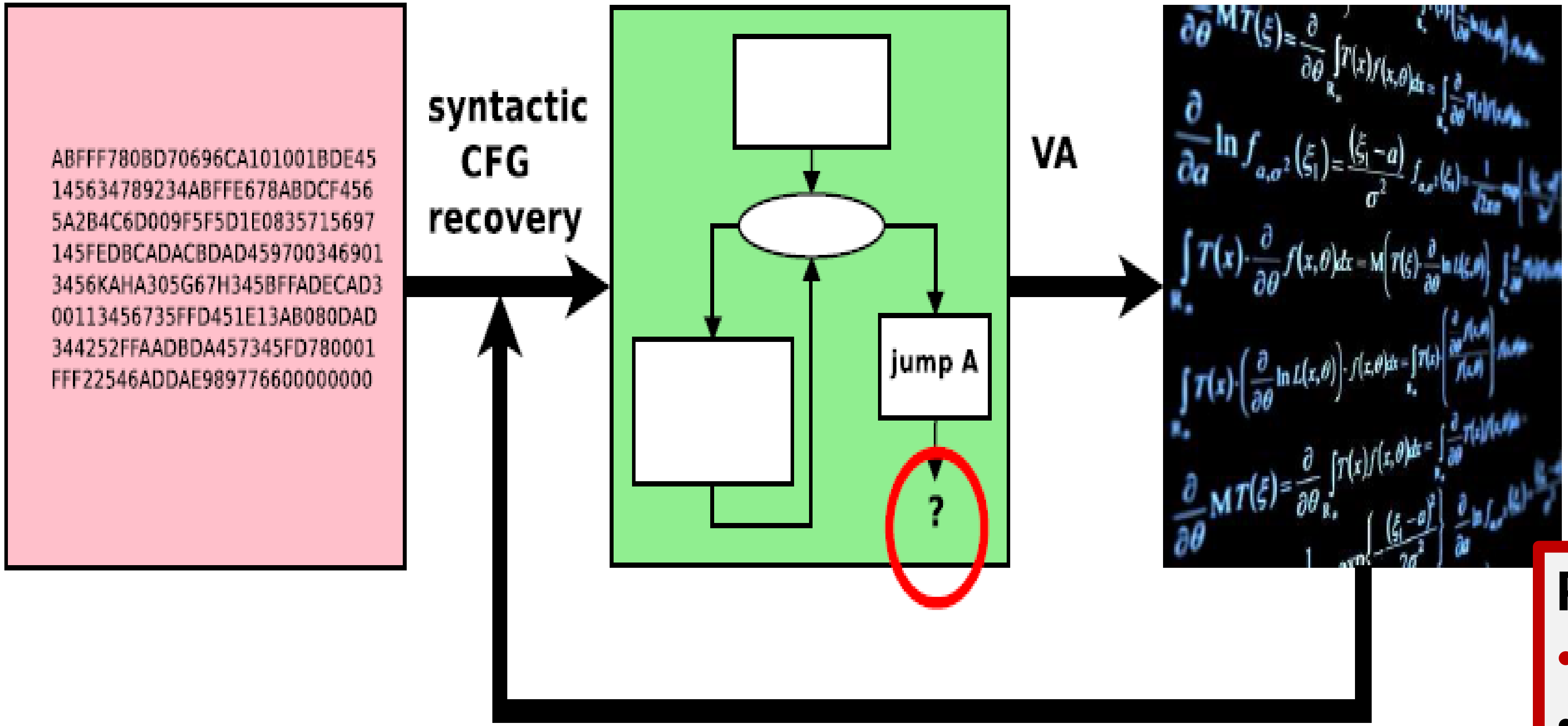
```
if (ax > bx) X = -1;
else X = 1;
```

```
      OF := ((ax{31,31}≠bx{31,31}) &
                (ax{31,31}≠(ax-bx){31,31}));
      SF := (ax-bx) < 0;
      ZF := (ax-bx) = 0;
      if (¬ ZF ∧ (OF = SF)) goto l1
      X := 1
      goto l2
l1:   X := -1
l2:
```





ISSUE: DYNAMIC JUMPS

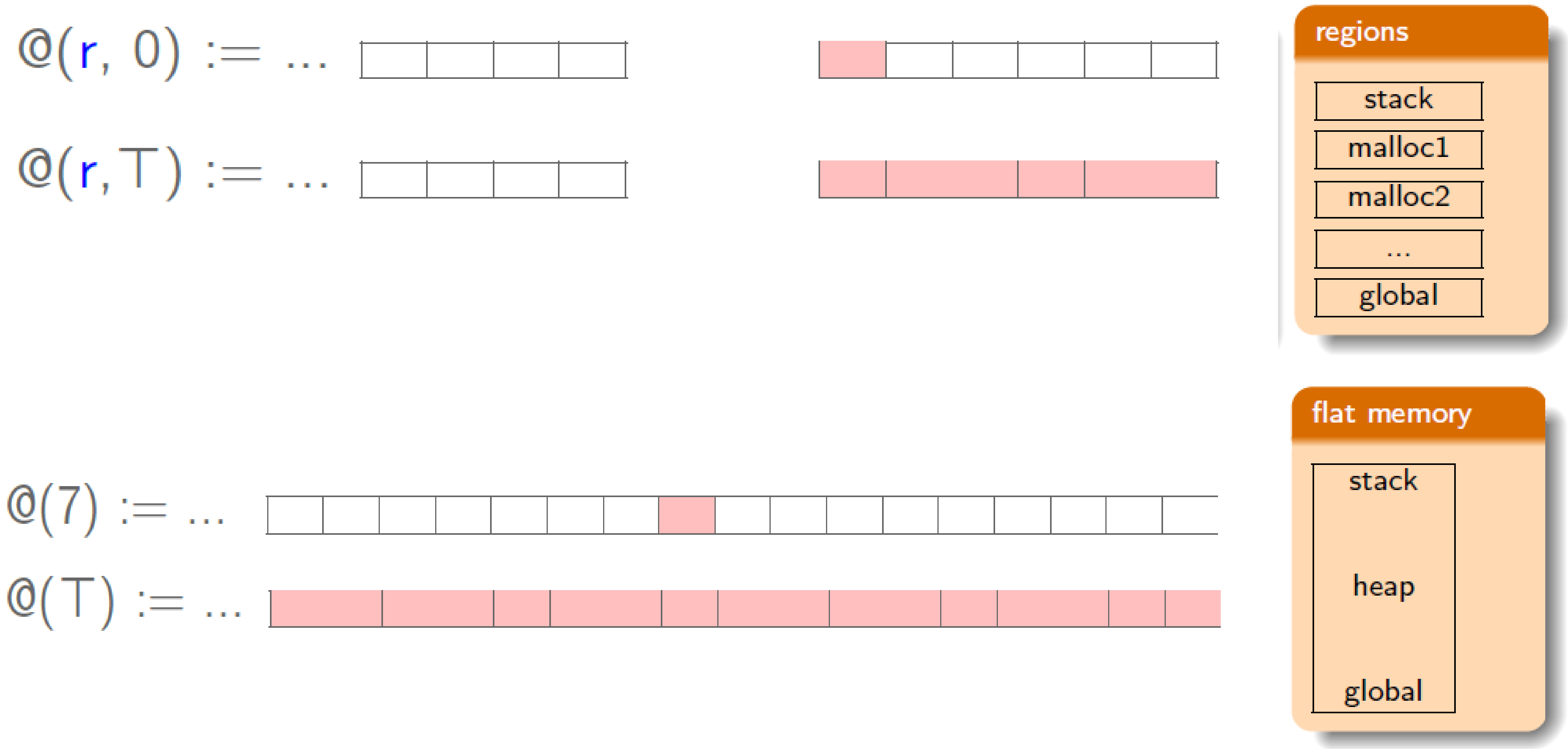


Problems

- 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)

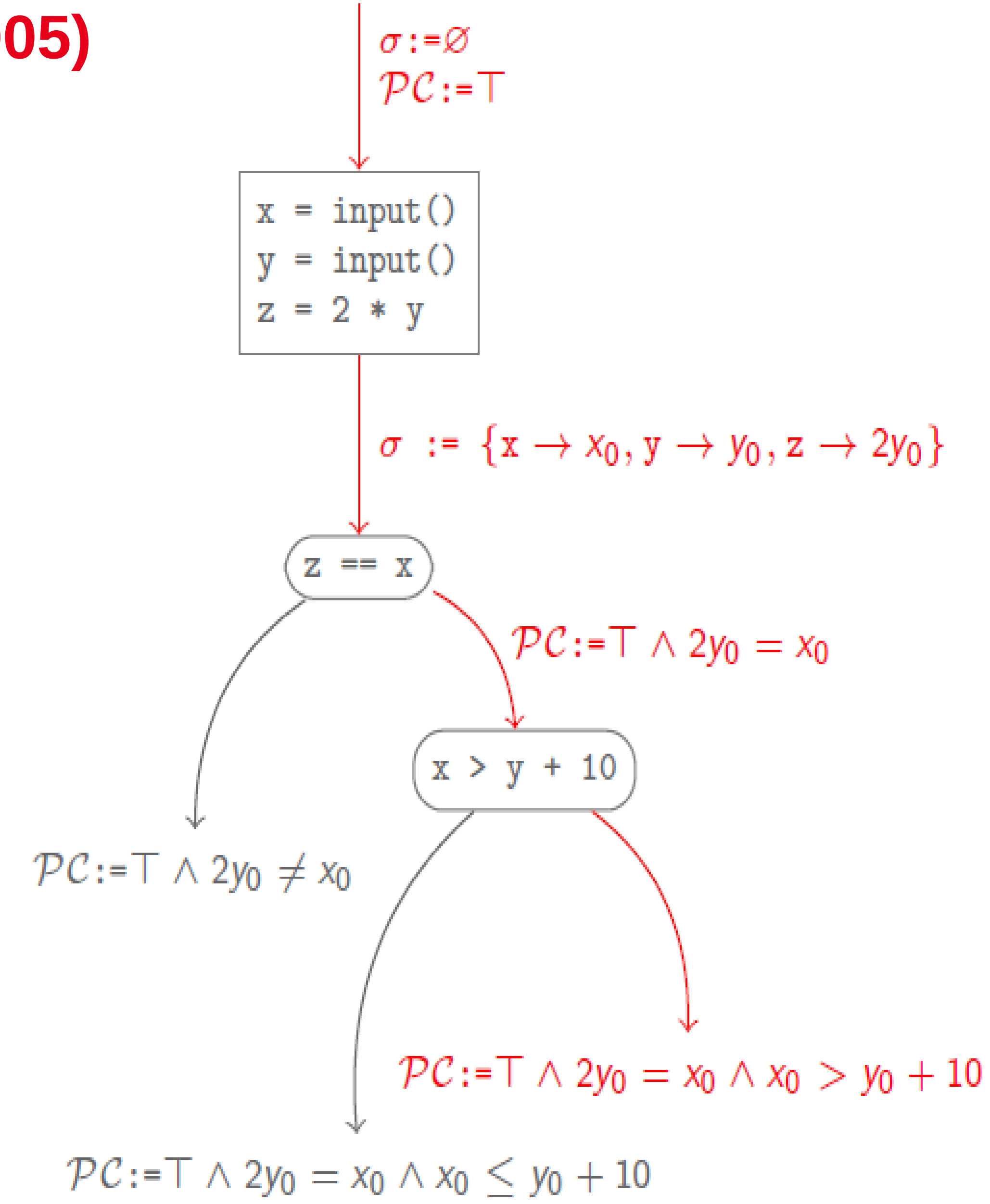
What’s next?

We need **useful** advanced binary-level semantic analysis!!

- precise, « correct-enough »
- reasonably efficient
- **robust**

SYMBOLIC EXECUTION (Godefroid, 2005)

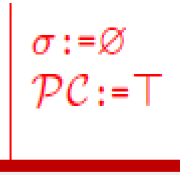
```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```



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

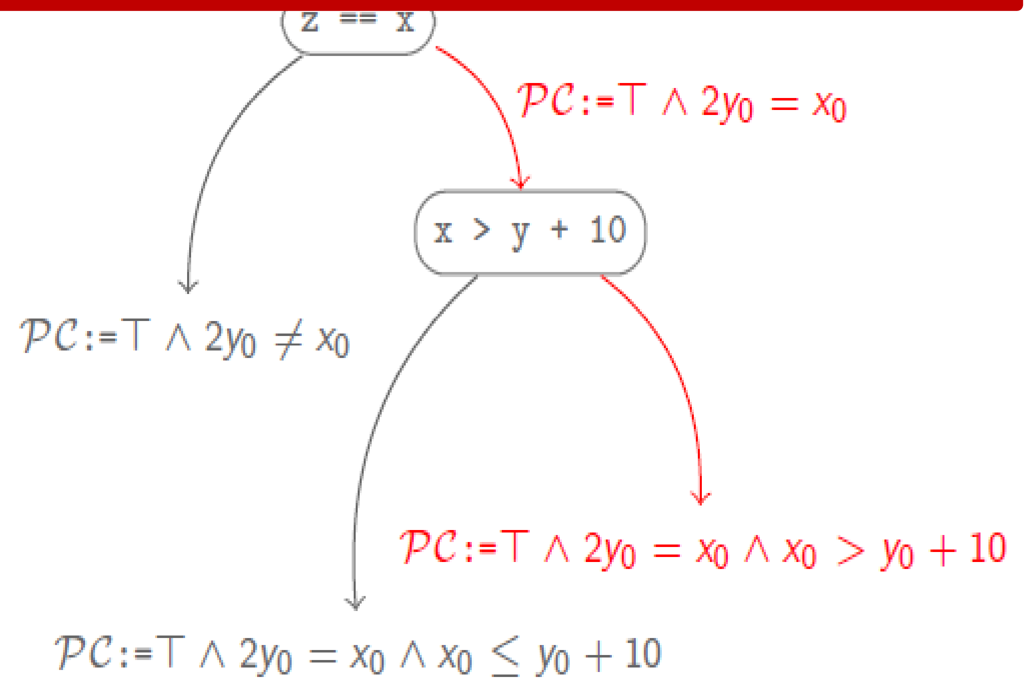
SYMBOLIC EXECUTION (Godefroid, 2005)

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



Good points:

- **No false positive = find real paths**
- **Robust (symb. + dynamic)**
- **Extend rather well to binary code**



Given a path of a program

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

DSE: PATH PREDICATE COMPUTATION & RESOLUTION

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

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

SMT Solver

Y0 = 0 /\ Z0=3

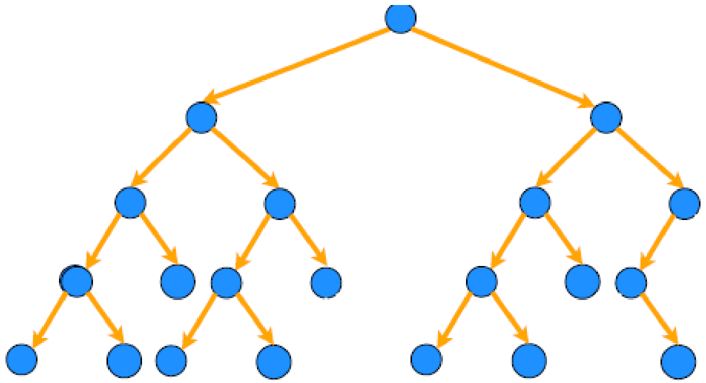
DSE: GLOBAL PROCEDURE

(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* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- 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]

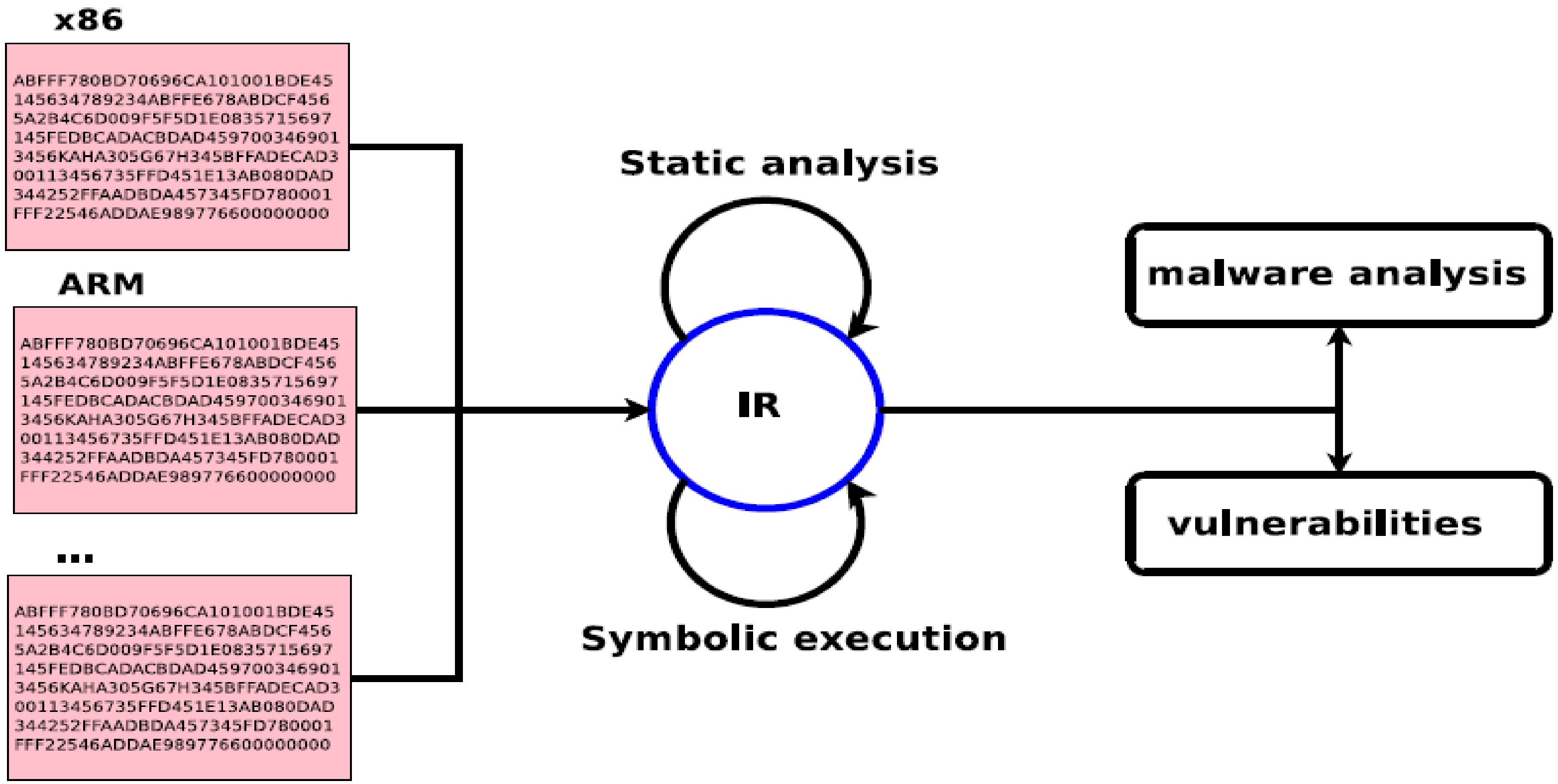
« concretization »

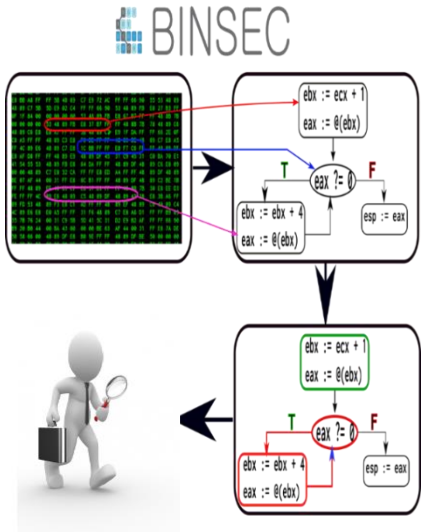
- Keep going when symbolic reasoning fails
- Tune the tradeoff genericity - cost

EXO

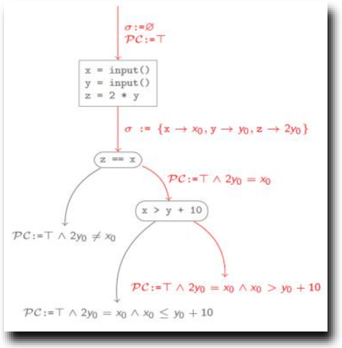
- DSE

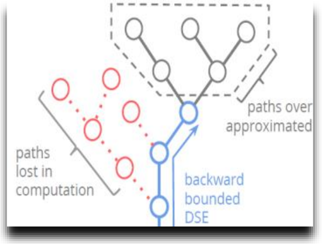
BINSEC: SYMBOLIC ANALYSIS for BINARY

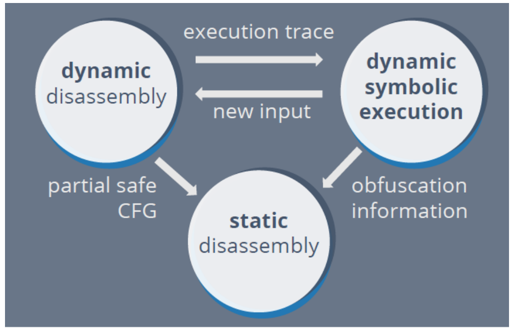




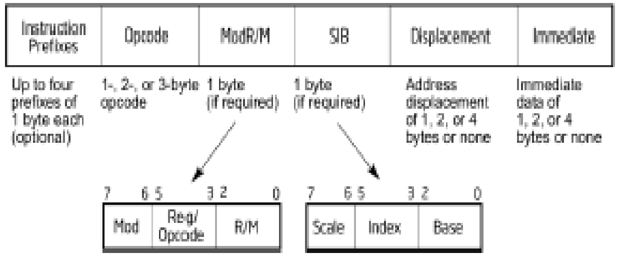
- `lhs := rhs`
- `goto addr, goto expr`
- `ite(cond)? goto addr :`
- `assume, assert, nondet`







Key: INTERMEDIATE REPRESENTATION



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

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

- **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)
```

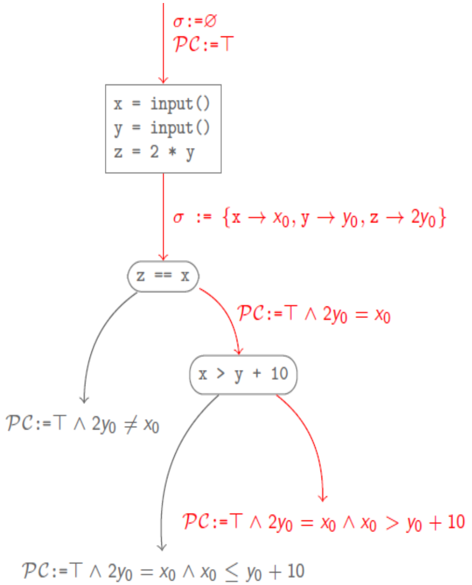
PART I: EXPLORE

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

Advantages
- Find new real paths
- Even rare paths

« dynamic analysis on steroids »

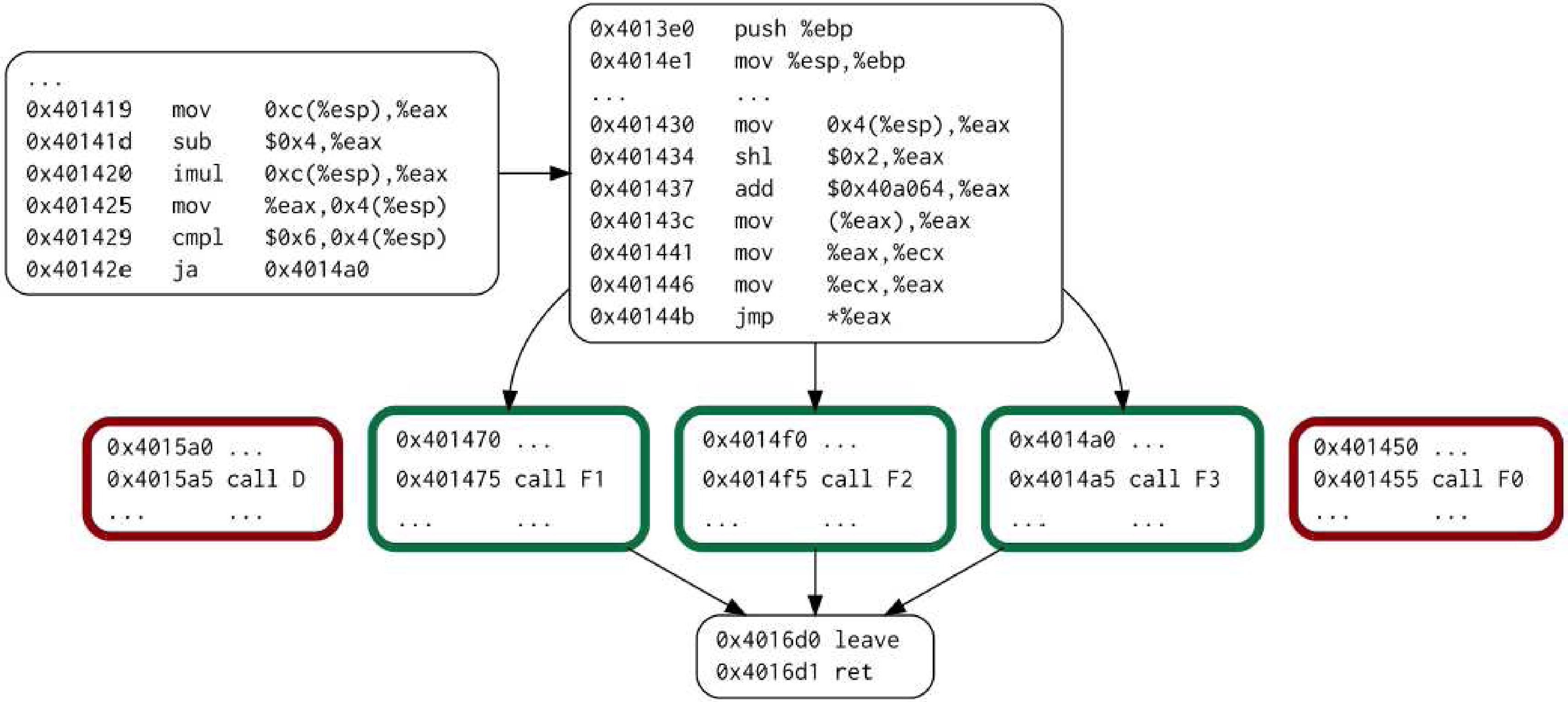
```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```



IN PRACTICE

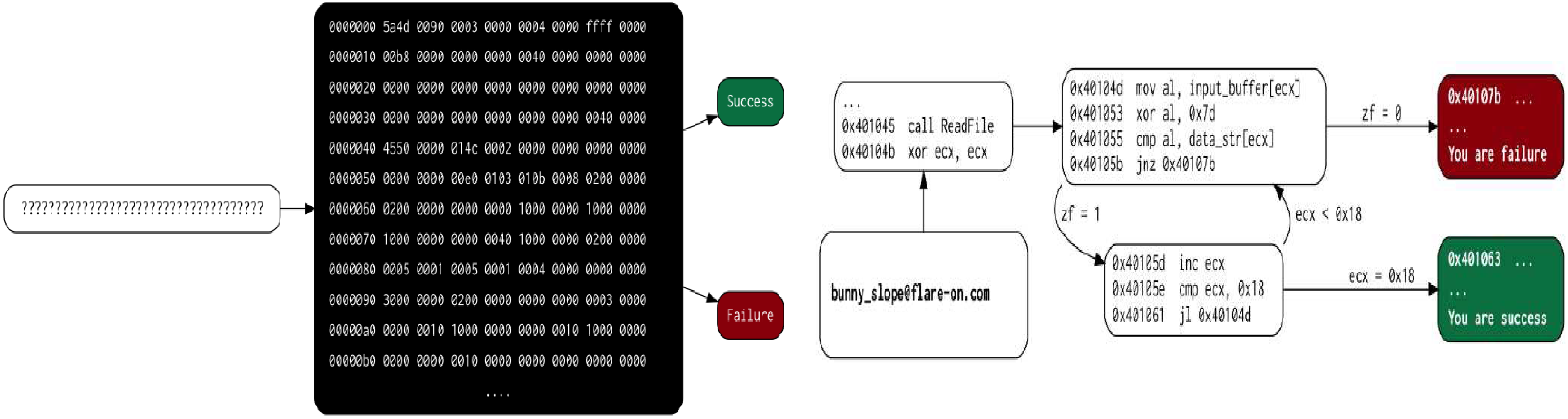
Solve for new dynamic targets

- **Get a first target**
- **Then solve for a new one**
- **Get it, solve again, …**
- **Get them all!**



With IDA + BINSEC

EXAMPLE: FIND THE GOOD PATH



Crackme challenges

- input == secret $\mapsto$ success
- input $\neq$ secret $\mapsto$ failure

EXAMPLE: FIND BUGS

GRUB2 CVE 2015-8370

- Elevation of privilege
- Information disclosure
- Denial of service

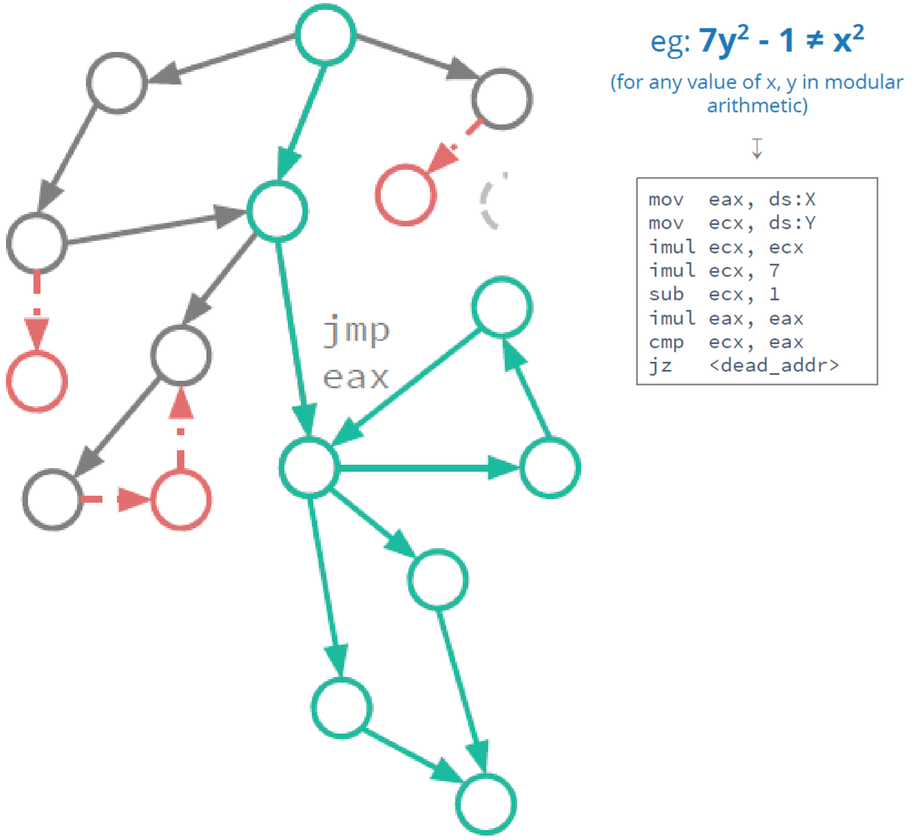
Thanks to P. Biondi @ AIRBUS GROUP

```
int main(int argc, char *argv[])
{
        struct {
                int  canary;
                char buf[16];
        } state;
        my_strcpy(input, argv[1]);
        state.canary = 0;
        grub_username_get(state.buf, 16);
        if (state.canary != 0) {
                printf("This gets interesting!\n");
        }
        printf("%s", output);
        printf("canary=%08x\n", state.canary);
}
```

PART II: PROVE

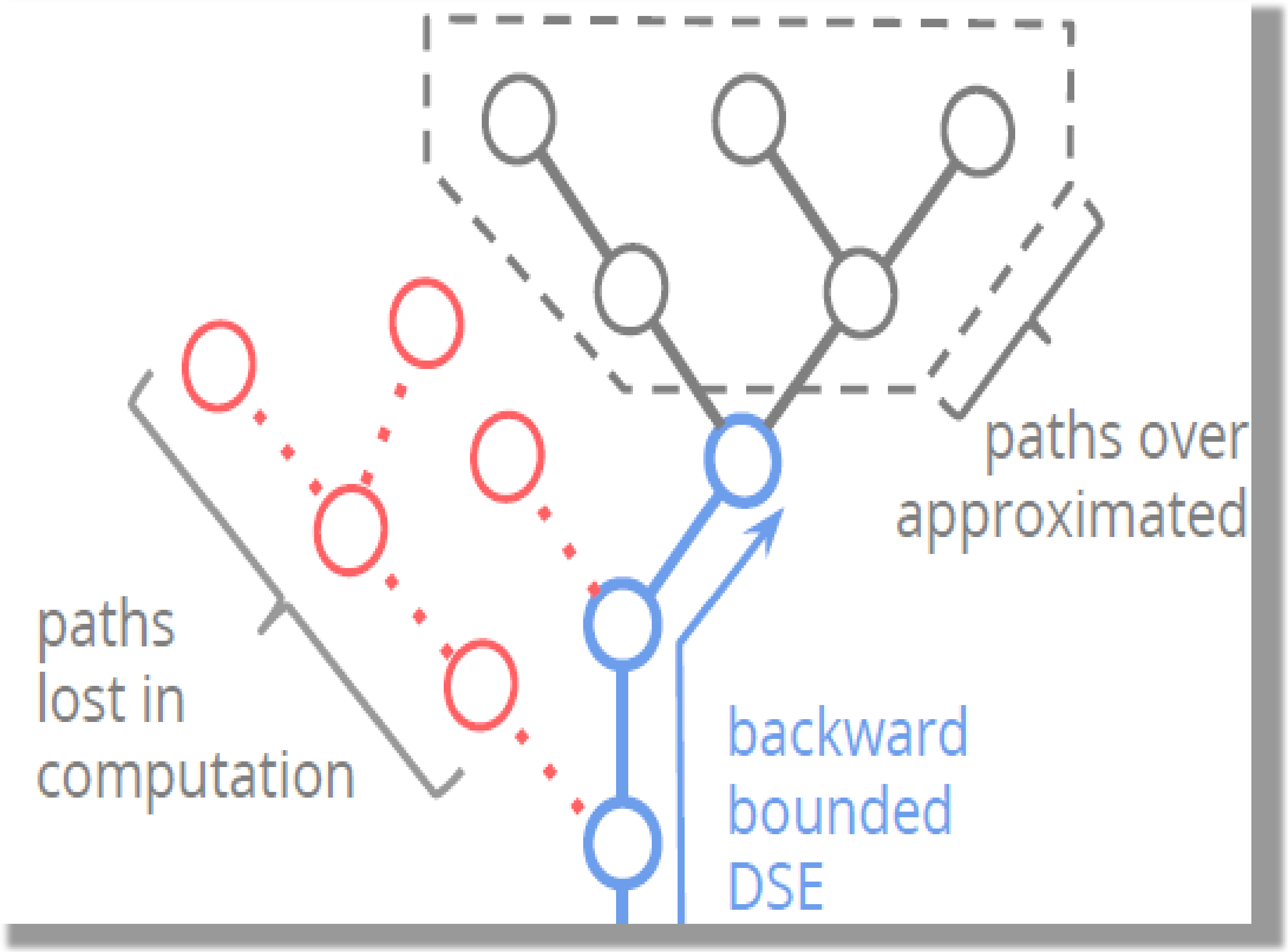
Prove that something is always true (resp. false)

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?**
- **does this expression always evaluate to 15?**
- **…**



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

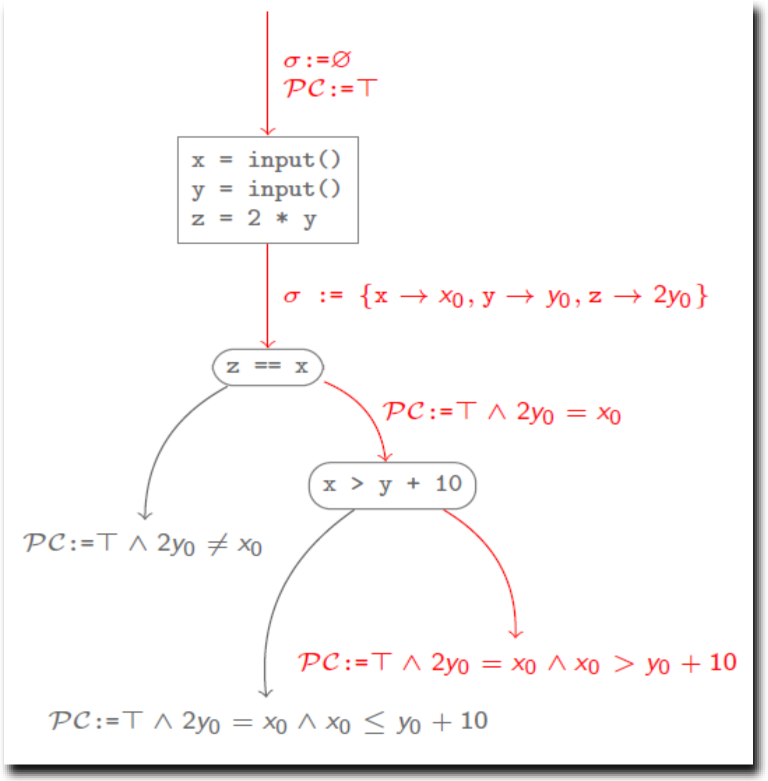
BACKWARD-BOUNDED SYMBOLIC EXECUTION

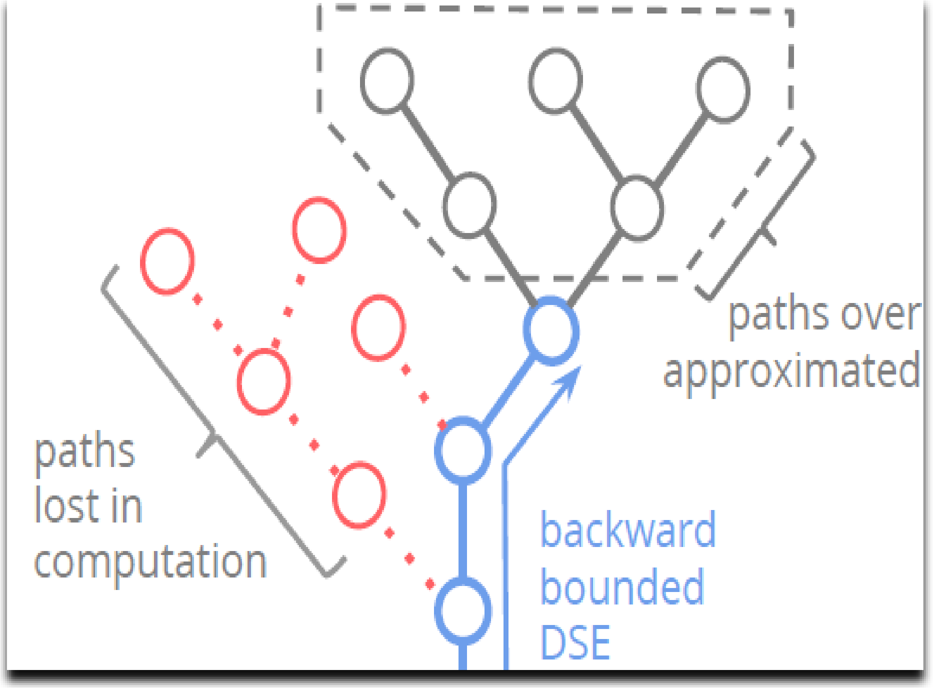


- Compute k-predecessors
- If the set is empty, no pred.
- Allows to prove things

- False Negative: k too small
- False Positive: CFG incomplete

FORWARD vs BACKWARD





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

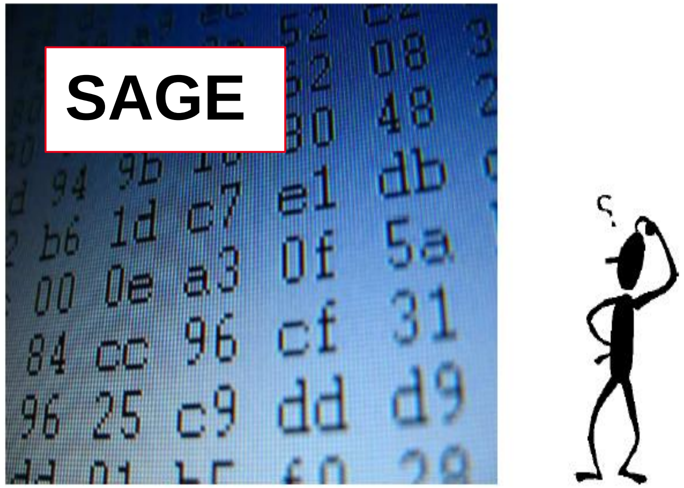
Explore & discover

- **Prove infeasible**

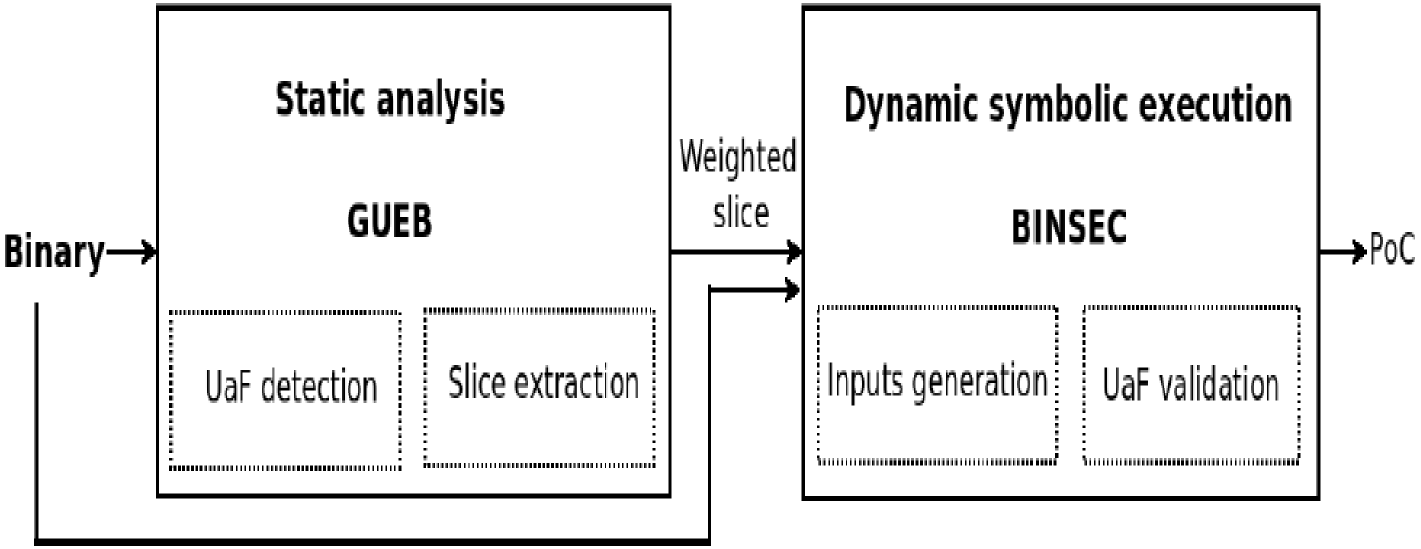
OUTLINE

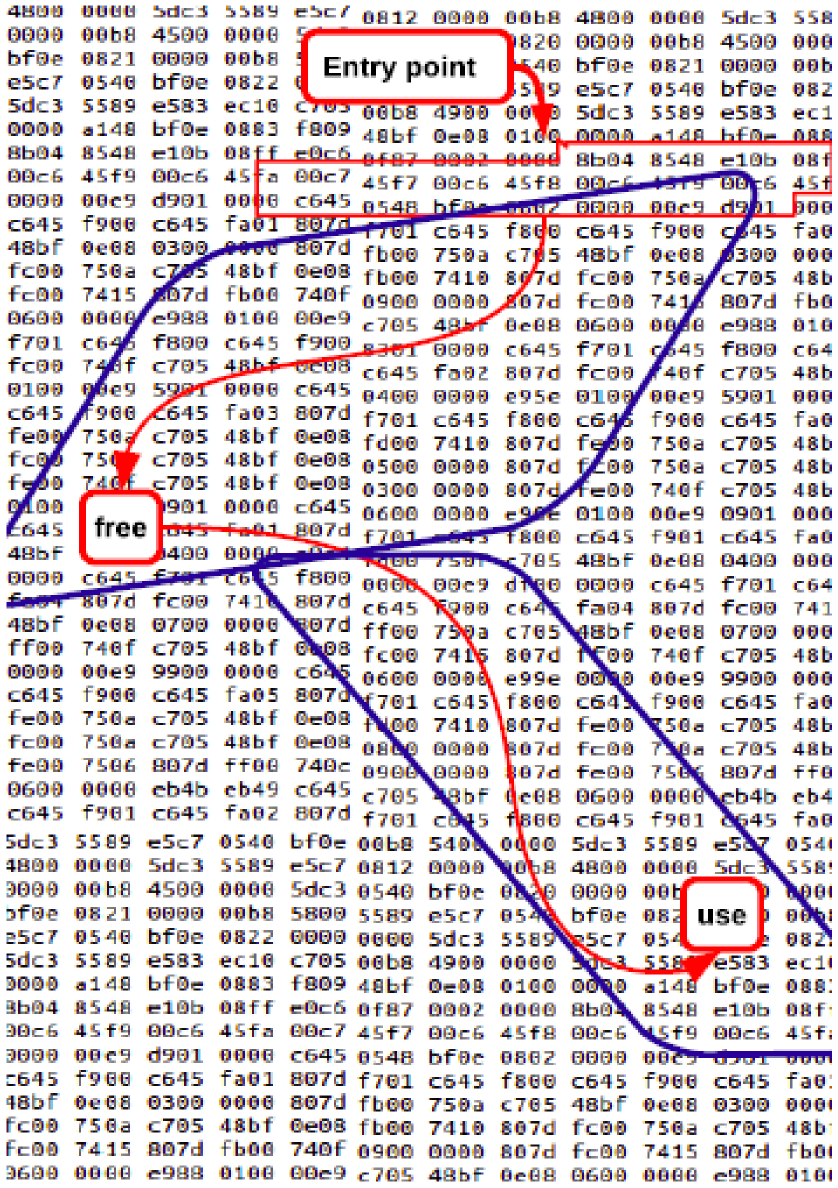
- Context
- Formal methods
- Overview of program analysis
- The hard journey from source to binary
- **A few case-studies**
- Discussion & Conclusion

Remember



VULNERABILITY DETECTION (use after free)

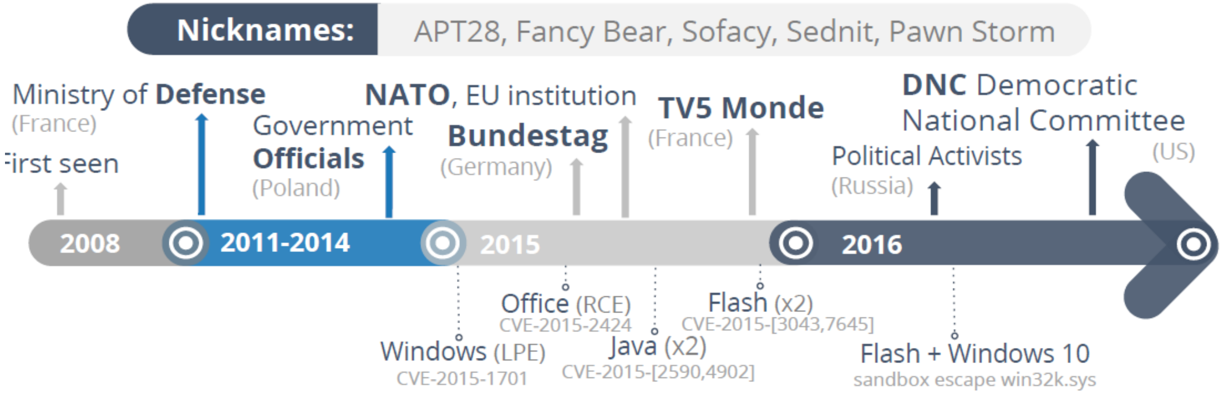




A Pragmatic 2-step approach
- **scalable and correct**

Find a few new CVEs

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

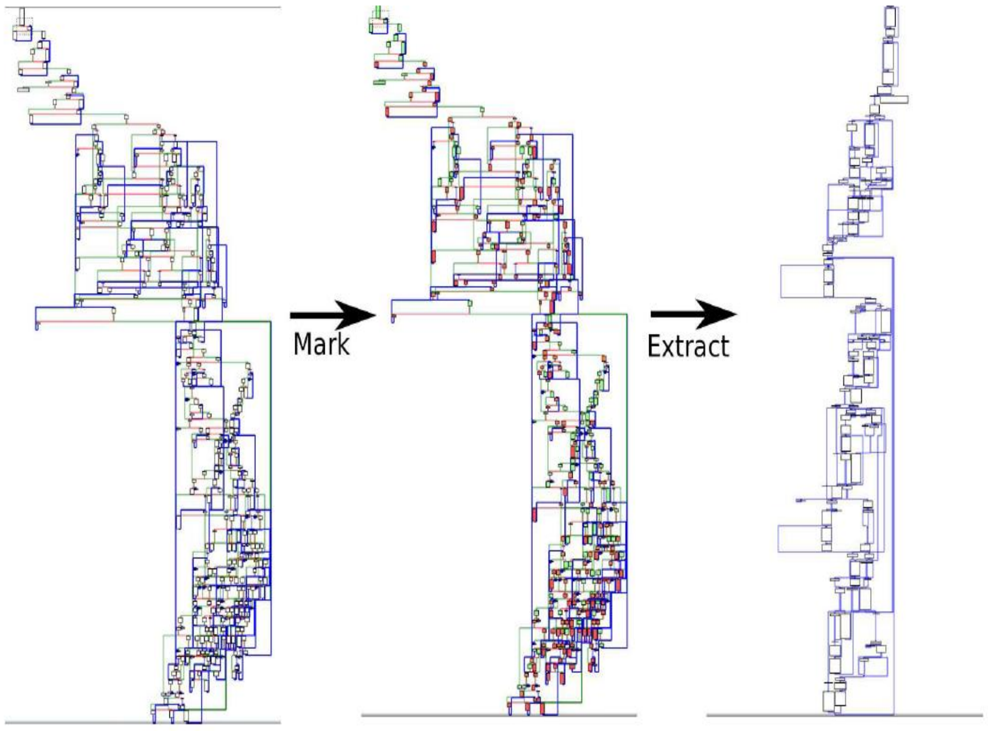


Two heavily obfuscated samples

- **Many opaque predicates**

Goal: detect & remove protections

- Identify 50% of code as spurious
- Fully automatic, < 3h



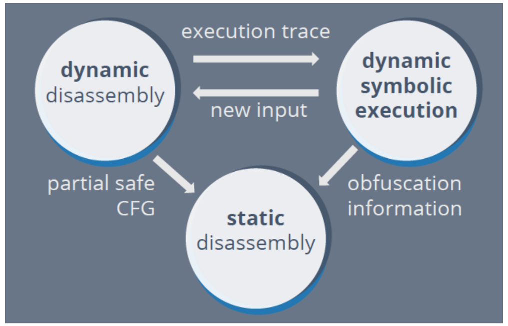
	C637 Sample #1	99B4 Sample #2
#total instruction	**505,008**	**434,143**
#alive	+279,483	+241,177

OUTLINE

- Why binary-level analysis?
- The hard journey from source to binary
- A few case-studies
- **Discussion & Conclusion**

LIMITS

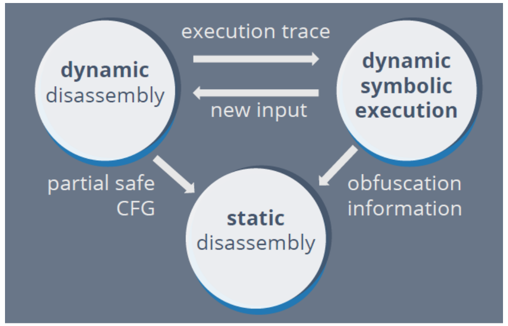
- **Trade off precision – scale – correct/complete**
  - Can be mitigated in some ways by combination



- **Semantic approaches always have weak points**
  - Diffuse tainting
  - Play with physical effects (side channels)
  - Hard to solve formulas

- **Take great care:**
  - Make everything looks like it depends from input
  - Make everything looks like it helps compute result

CONCLUSION



- **Binary-level security analysis needs advanced tooling**
  - Current syntactic and dynamic methods are not enough
- **Semantic analysis complement existing approaches**
  - Well-adapted – semantics is invariant by obfuscation or compilation
  - Explore, prove infeasible, simplify -- and allows succint reasoning
  - Promising case-studies
- **Especially: need to be taken into account by defenders**
- **Yet, challenging to adapt from source-level safety-critical**
  - Need robustness, precision and scale!!

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

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

ideas

- Example : S&P 2015 en plus, SAGE en plus, citer « can obfuscation kepp up with the pace of program analysis»
- Duality, protections