

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

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



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

- **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



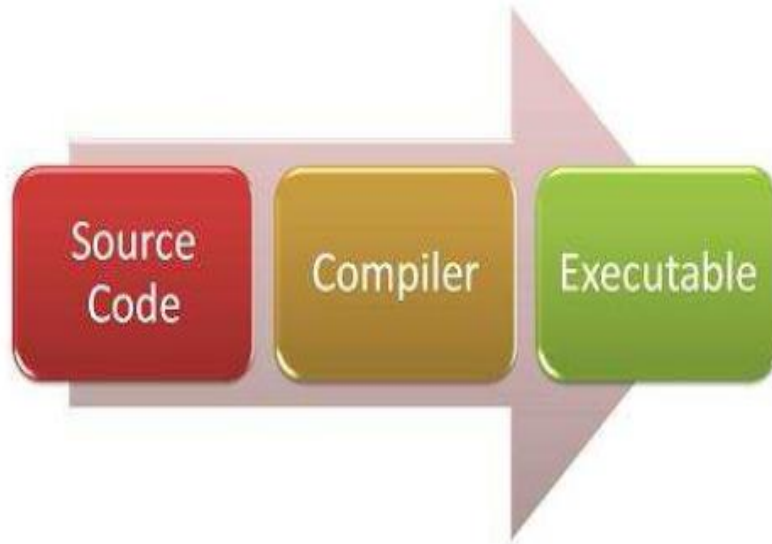
Find a needle in the heap!

```

4800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
0000 00b8 4500 0000 5dc3 0540 bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b8
bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b8
e5c7 0540 bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b8
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec10
0000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 0883
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08ff
00c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f9
0000 00e9 d901 0000 c645 0548 bf0e 0802 0000 0000 00e9 d901 0000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 0000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
0600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 0100
f701 c645 f800 c645 f900 8301 0000 c645 f701 c645 f800 c64
fc00 740f c705 48bf 0e08 c645 fa02 807d fc00 740f c705 48b
0100 00e9 5901 0000 c645 0400 0000 e95e 0100 00e9 5901 0000
c645 f900 c645 fa03 807d f701 c645 f800 c645 f900 c645 fa0
fe00 750a c705 48bf 0e08 fd00 7410 807d fe00 750a c705 48b
fc00 750a c705 48bf 0e08 0500 0000 807d fc00 750a c705 48b
fe00 740f c705 48bf 0e08 0300 0000 807d fc00 740f c705 48b
0100 901 0000 c645 0600 0000 e90e 0100 00e9 9901 0000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f901 c645 fa0
48bf 0e08 0000 c9c4 fd00 750f c705 48bf 0e08 0400 0000
0000 c645 f701 c645 f800 0000 00e9 df00 0000 c645 f701 c64
fa04 807d fc00 7410 807d c645 f900 c645 fa04 807d fc00 741
48bf 0e08 0700 0000 807d ff00 750a c705 48bf 0e08 0700 0000
ff00 740f c705 48bf 0e08 fc00 7415 807d ff00 740f c705 48b
0000 00e9 9900 0000 c645 0600 0000 e99e 0000 00e9 9900 0000
c645 f900 c645 fa05 807d f701 c645 f800 c645 f900 c645 fa0
fc00 750a c705 48bf 0e08 fd00 7410 807d fc00 750a c705 48b
fc00 750a c705 48bf 0e08 0800 0000 807d fc00 750a c705 48b
fe00 7506 807d ff00 740c 0900 0000 807d fe00 7506 807d ff0
0600 0000 eb4b eb49 c645 c705 48bf 0e08 0600 0000 eb4b eb4
c645 f901 c645 fa02 807d f701 c645 f800 c645 f901 c645 fa0
5dc3 5589 e5c7 0540 bf0e 00b8 5400 0000 5dc3 5589 e5c7 0540
1800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 5589
3000 00b8 4500 0000 5dc3 0540 bf0e 0821 0000 00b8 5800 5589
bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b8
e5c7 0540 bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b8
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec10
0000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 0883
3b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08ff
00c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f9
0000 00e9 d901 0000 c645 0548 bf0e 0802 0000 00e9 d901 0000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
18bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 0000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
3600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 0100

```

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:

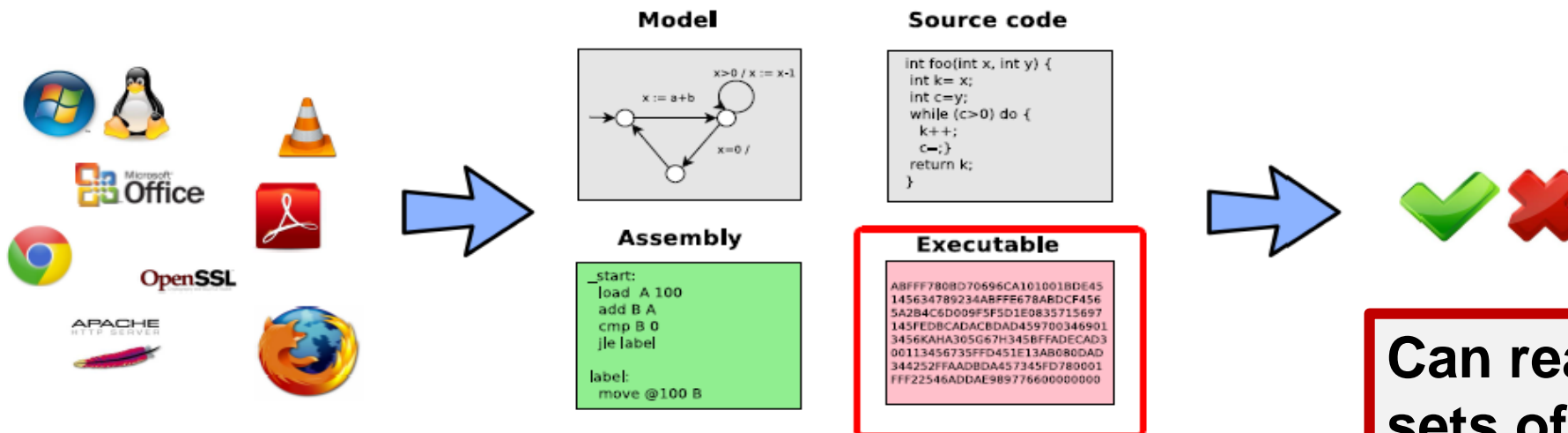
- **Check the code after compilation**

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**



Advantages

- more robust than syntactic
- more thorough than dynamic

Challenges

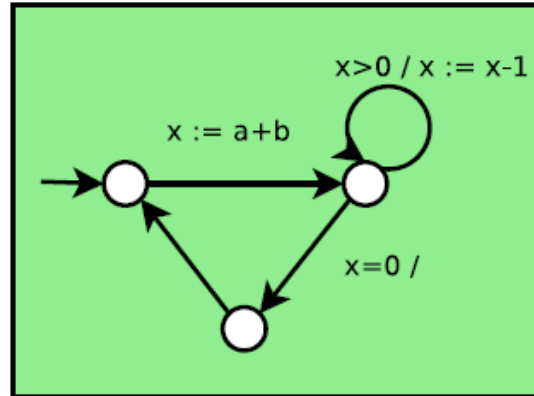
- source-level \mapsto binary-level
- safety \mapsto security
- many (complex) architectures

**Can reason about
sets of executions**

- find rare events
- prove facts

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

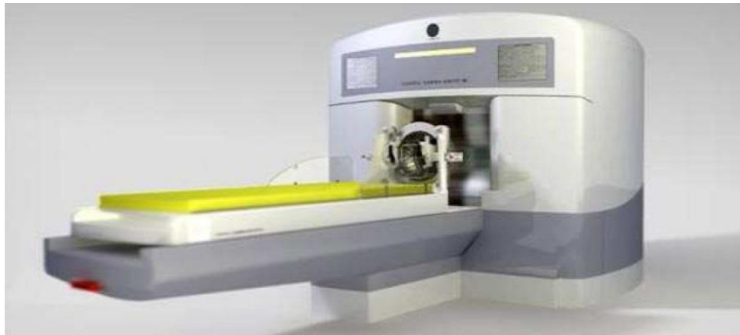
```
ABFFF780BD70696CA101001BDE45  
145634789234ABFFE678ABDCF456  
5A2B4C6D009F5F5D1E0835715697  
145FEDBCADACBDAD459700346901  
3456KAHA305G67H345BFFADECAD3  
00113456735FFD451E13AB080DAD  
344252FFAADBDA457345FD780001  
FFF22546ADDAE989776600000000
```

- 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)



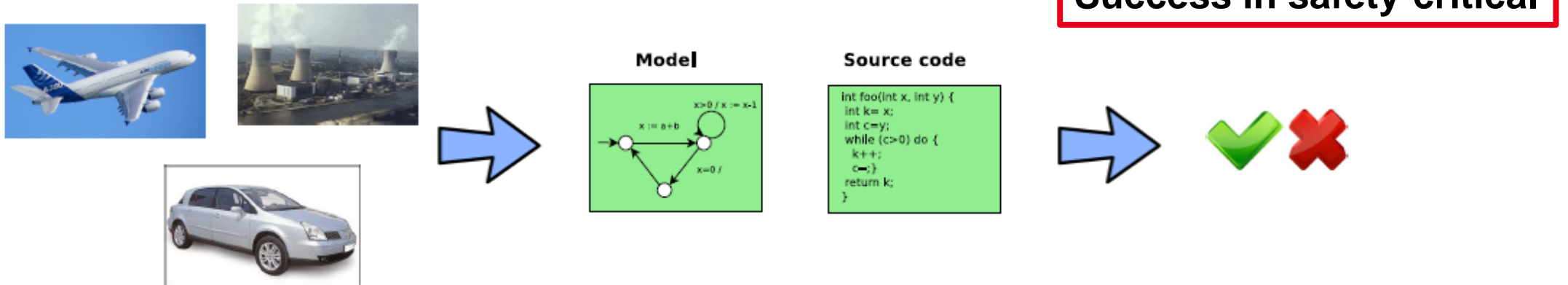
http://en.wikipedia.org/wiki/List_of_software_bugs

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



Key concepts : $M \models \varphi$

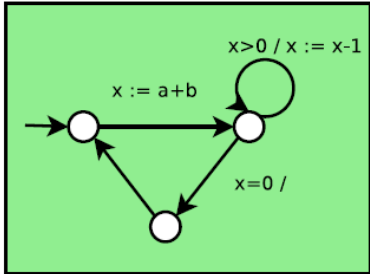
- M : semantic of the program
- φ : property to be checked
- \models : algorithmic check

Kind of properties

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

Input model?

Model



Source code

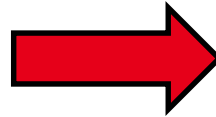
```
int foo(int x, int y) {  
  int k= x;  
  int c=y;  
  while (c>0) do {  
    k++;  
    c--;}  
  return k;  
}
```

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
- ...

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;
}
```

A set of **good** behaviours

- Reachable states
- Traces (finite or infinite)
- Execution Tree
- ...



Spec may be implicit

- Good typing
- No runtime error
- ...

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

integer overflow

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

division by zero

Examples from static analysis benchmarks

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

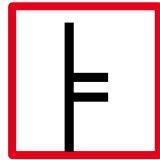
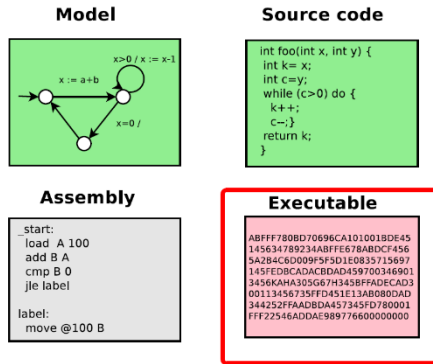
void f(void){
  int index;
  for (index = 0; index < TAILLE_TAB
; index++)
  {
    tab[index] = 0;
  }
  tab[index] = 1;
}
```

out of bounds access

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

uninitialized pointer

Algorithmic check (1)



A set of relevant behaviours

- Reachable states
- Traces (finite or infinite)
- Execution Tree
- ...



A set of **good behaviours**

- Reachable states
- Traces (finite or infinite)
- Execution Tree
- ...

Algorithmic check (2)

Problem is often undecidable

- **Over-approximation**
- **Under-approximation**

// Witness?

Two key aspects of program analysis

- Mastering abstraction
- Fighting undecidability / intractability

- 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)

The SMACCMCopter: 18-Month Assessment

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



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

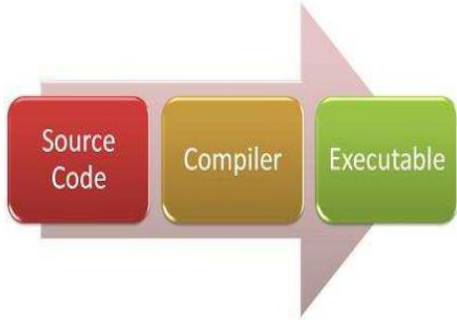
Formally-hardened drone

- memory safety
- ignores bad messages

Red team attack

- 6 weeks, access to source
- no security bug found

Other successes



**Compcert
Sel4**

```

2552 #ifndef OPENSSL_NO_HEARTBEATS
2553 int
2554 tls1_process_heartbeat(SSL *s)
2555 {
2556     /* Read type and payload length first */
2557     hbtype = *p++;
2558     hblen = *p++;
2559     p1 = p;
2560     [-]
2561     if (hbtype == TLS1_HB_REQUEST)
2562     {
2563         /* Enter response type, length and copy payload */
2564         *hp++ = TLS1_HB_RESPONSE;
2565         s2n(payload, hp);
2566         memcpy(p1, payload, hblen);
2567         bp += payload;
2568         /* Random padding */
2569         RAND_pseudo_bytes(bp, padding);
2570
2571         r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
2572             3 + payload + padding);
2573
2574         if (r >= 0 && s->msg_callback)
2575             s->msg_callback(1, s->version,
2576                 TLS1_RT_HEARTBEAT,
2577                 buffer, 3 + payload + padding,
  
```



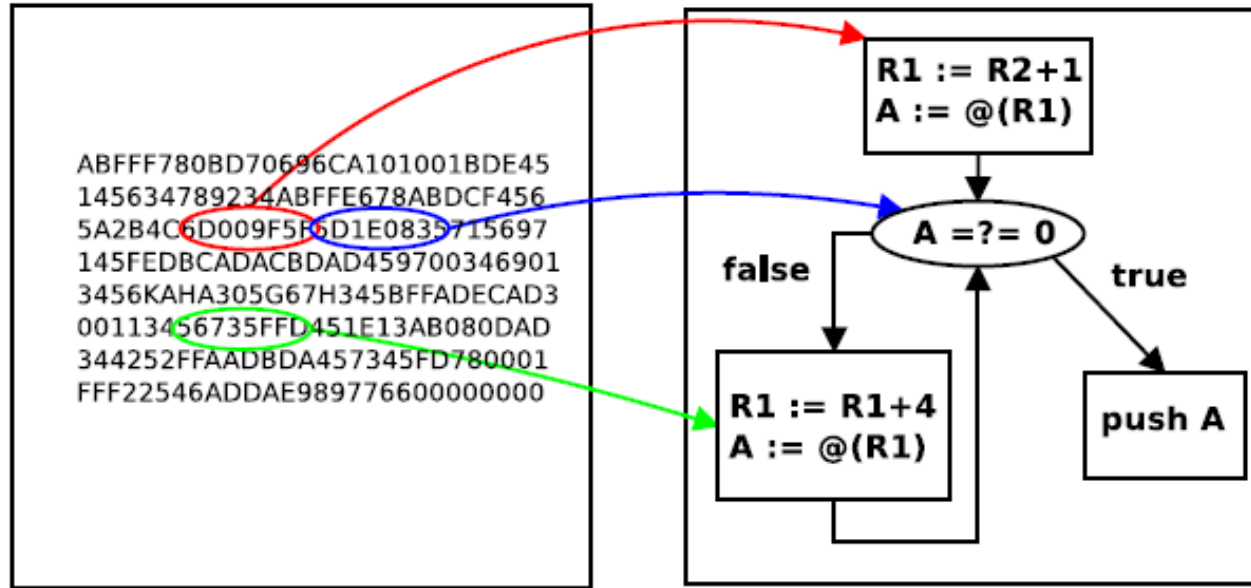
SAGE



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

- **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)

Sections

.text	8D 4C 24 04 83 E4 F0 FF 71 FC 55 89 E5 53 51 83
	EC 10 89 CB 83 EC 0C 6A 0A E8 A7 FE FF FF 83 C4
	10 89 45 F0 8B 43 04 83 C0 04 8B 00 83 EC 0C 50
	E8 C0 FE FF FF 83 C4 10 89 45 F4 83 7D F4 04 77
	3B 8B 45 F4 C1 E0 02 05 98 85 04 08 8B 00 FF E0
	C7 45 F4 00 00 00 00 EB 23 C7 45 F4 01 00 00 00
	EB 1A C7 45 F4 02 00 00 00 EB 11 C7 45 F4 03 00
	00 00 EB 08 C7 45 F4 04 00 00 00 90 83 EC 08 FF
	75 F4 68 90 85 04 08 E8 29 FE FF FF 83 C4 10 8B
	45 F4 8D 65 F8 59 5B 5D 8D 61 FC C3 66 90 66 90
	66 90 66 90 90 55 57 31 FF 56 53 E8 85 FE FF FF
	81 C3 89 12 00 00 83 EC 1C 8B 6C 24 30 8D B3 0C
	FF FF FF E8 B1 FD FF FF 8D 83 08 FF FF FF 29 C6
	C1 FE 02 85 F6 74 27 8D B6 00 00 00 8B 44 24
	38 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
	FF 94 BB 08 FF FF FF 83 C7 01 39 F7 75 DF 83 C4
	1C 5B 5E 5F 5D C3 EB 0D 90 90 90 90 90 90 90
	90 90 90 90 90 F3 C3 FF FF 53 83 EC 08 E8 13 FE
	FF FF 81 C3 17 12 00 00 83 C4 08 5B C3 03 00 00
	.fini
	.rodata
	00 01 00 02 00 76 61 6C 3A 25 64 0A 00 AB 84 04
08 B4 84 04 08 BD 84 04 08 C6 84 04 08 CF 84 04	
08 01 1B 03 3B 28 00 00 00 04 00 00 00 54 FD FF	
.eh_frame_hdr	

■ code
 ■ dead bytes
 ■ global csts
 ■ strings
 ■ pointers
 ■ other

Code (Functions)

main

unknown

_libc_csu_init

unknown

_libc_csu_fini

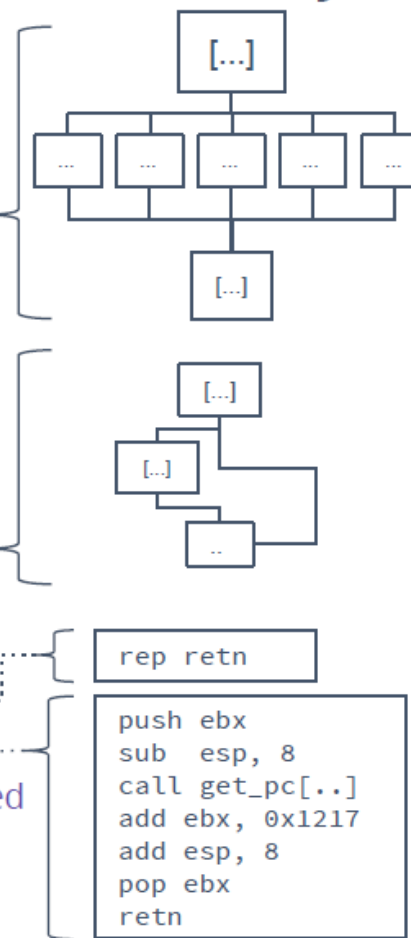
_term_proc

_fp_hw, _IO_stdin_used

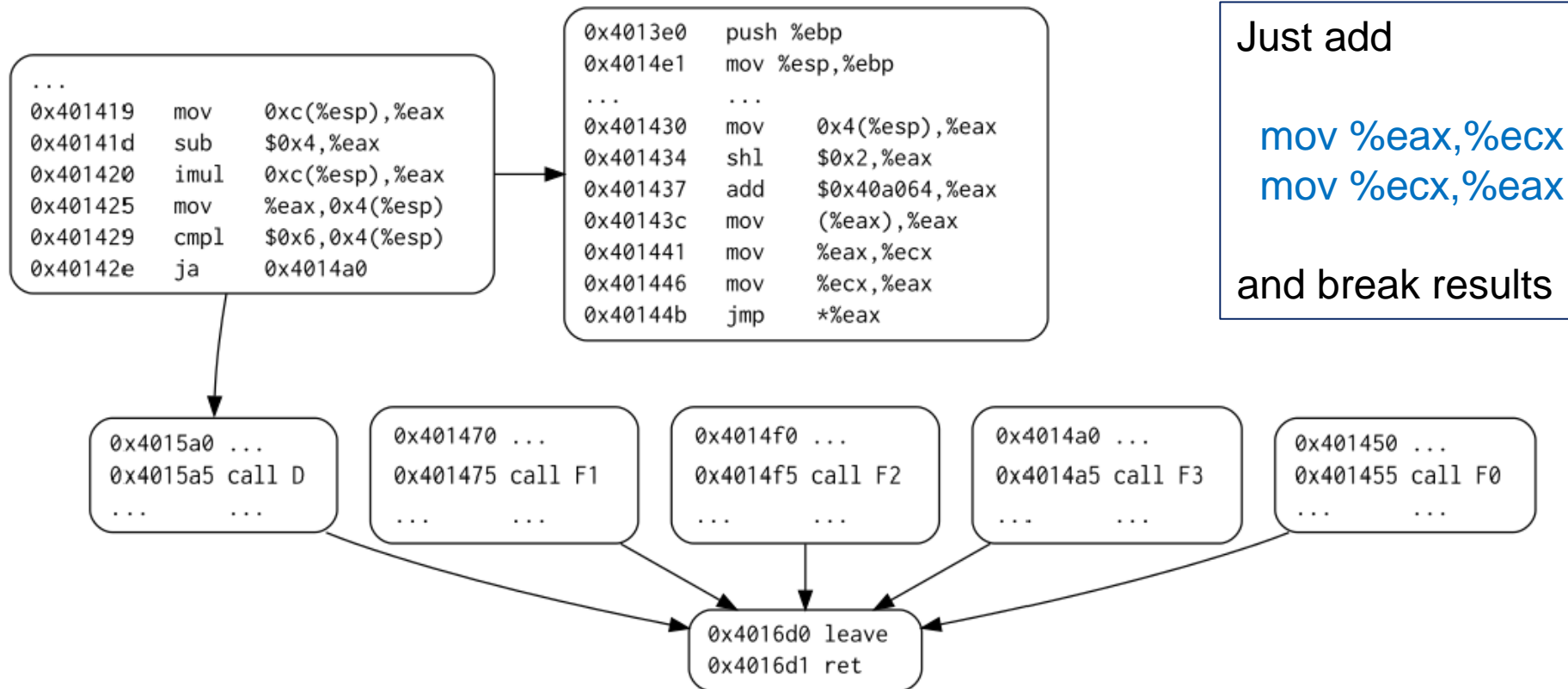
"val%d\n"

switch jump table

Assembly



STATE-OF-THE-ART TOOLS ARE NOT ENOUGH



Just add

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

and break results

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

With IDA

- 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?

```
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?

- **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

Petit langage impératif à valeurs entières

Expressions *Expr*

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

Instructions *Instr*

$i ::=$	$x := e$	affectation
	if e then i else i fi	conditionnelle
	while e do i done	boucle
	$i; i$	séquence
	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 :

$$. \text{ex} : z := a + b$$

$$. c \xrightarrow{z:=a+b} c' \text{ ssi } c' = c[z \leftarrow c[a] + c[b]]$$

- un branchement teste l'état mémoire



<Correction>

- **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

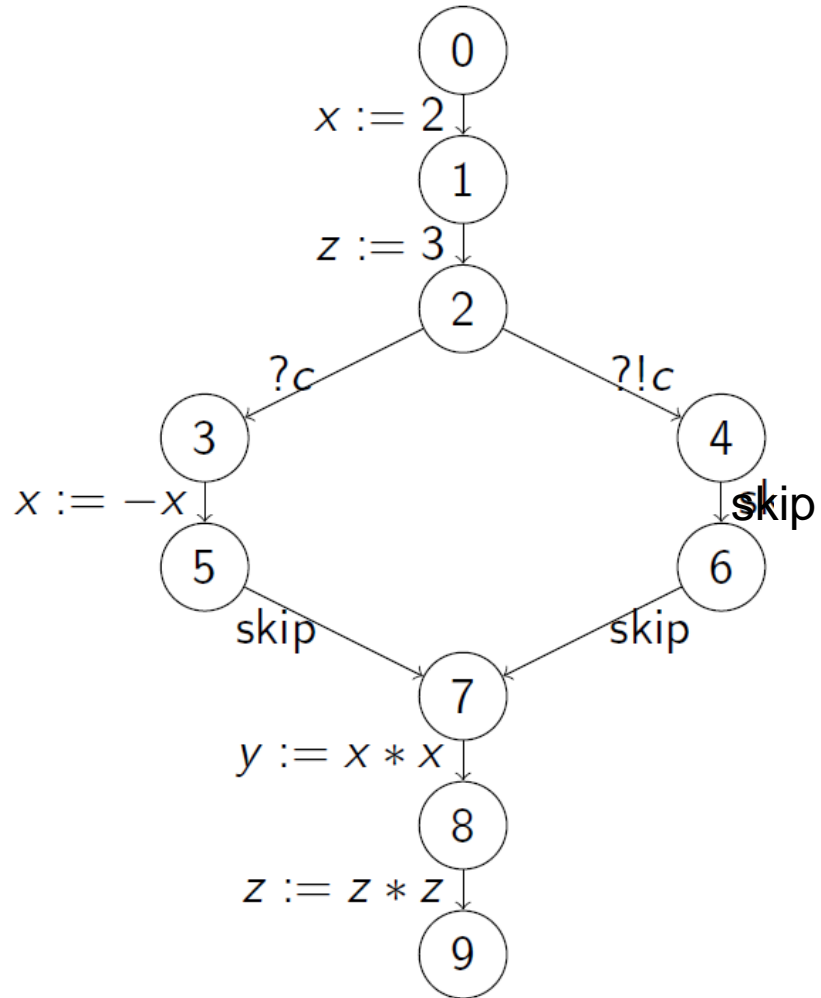
Vue informelle

- 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

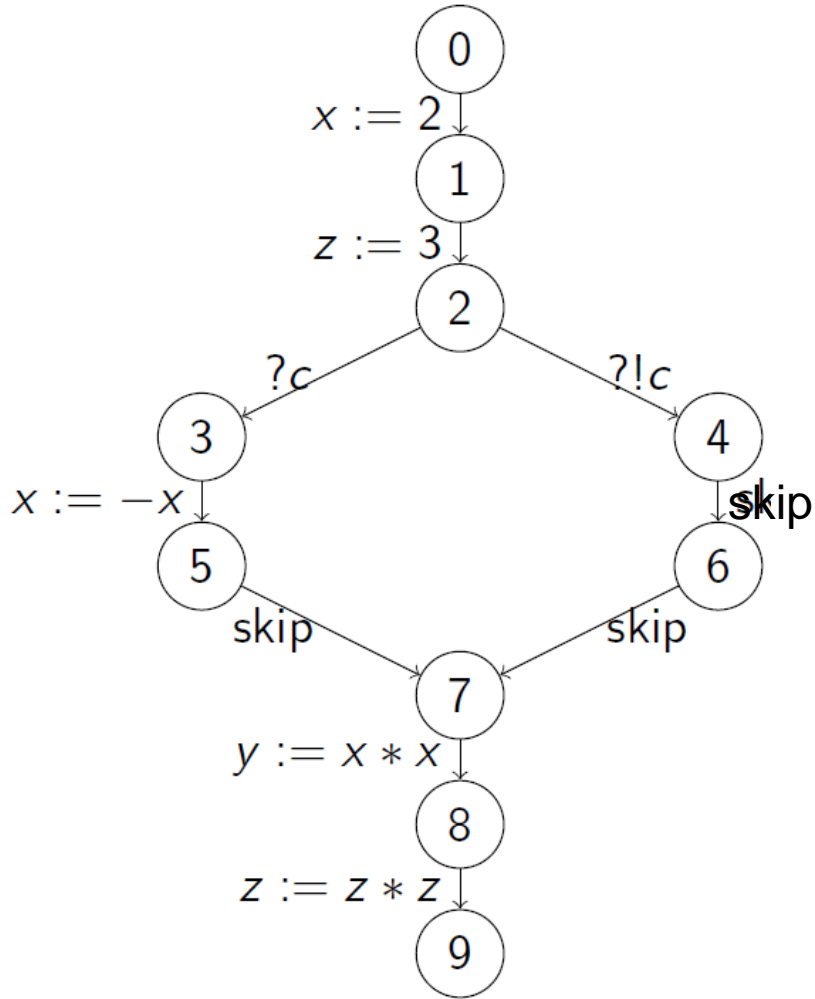
Constant propagation:

- 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

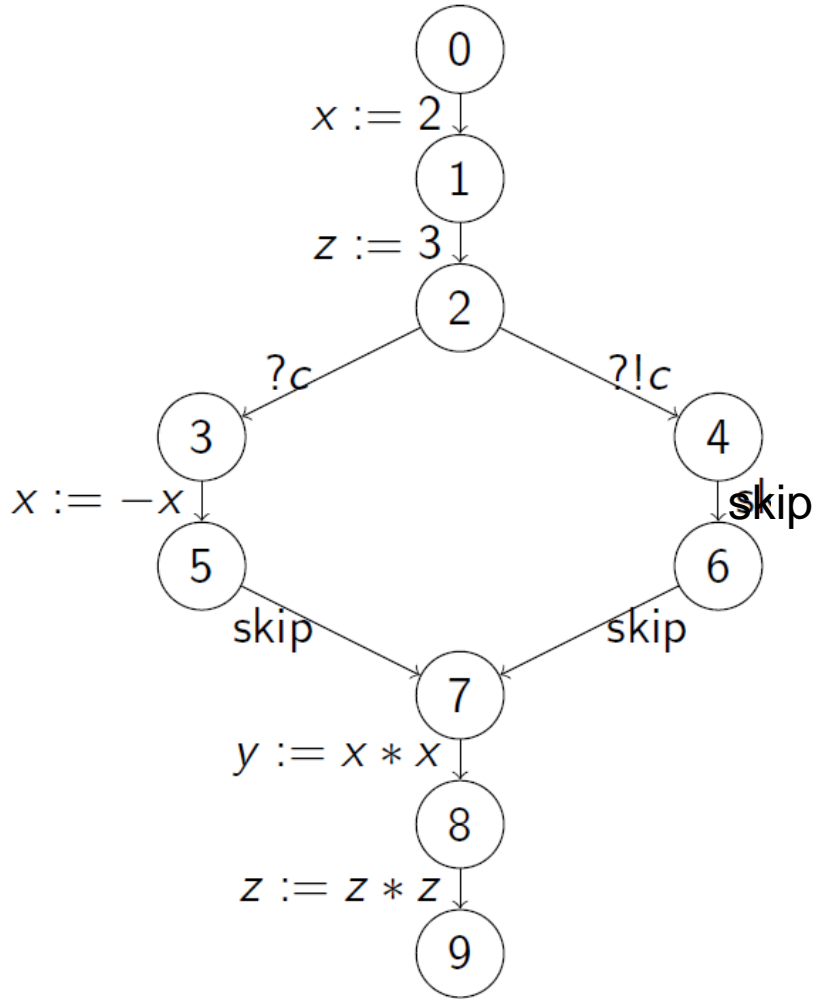


- Precision loss in practice
- Line 7 :
 - $X \in \{-2, 2\}$
 - Becomes $X \in T$

no				
1	T	2	T	T
2	T	2	T	3
3	T	2	T	3
4	0	2	T	3
5	T	-2	T	3
6	0	2	T	3
7	T	$\neg 2$	T	3
8	T	T	T	3
9	T	T	T	9

- **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



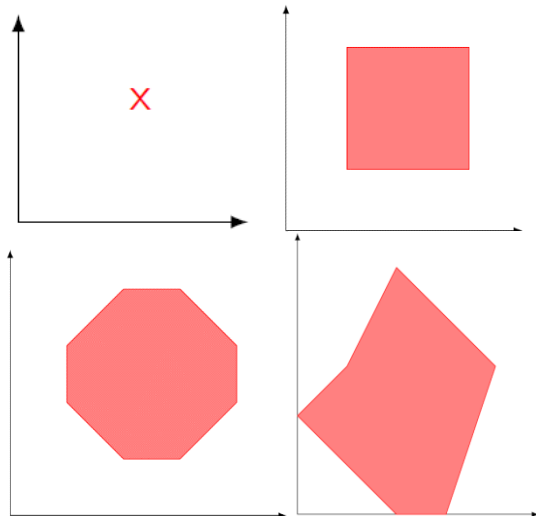
- Precision loss in practice
- Line 7 :
 - $X \in \{-2, 2\}$
 - $X \in -2..2$

no				
1	T	2	T	T
2	T	2	T	3
3	T	2	T	3
4	0	2	T	3
5	T	-2	T	3
6	0	2	T	3
7	T	$\neg 2$	T	3
8	T	T	T	3
9	T	T	T	9

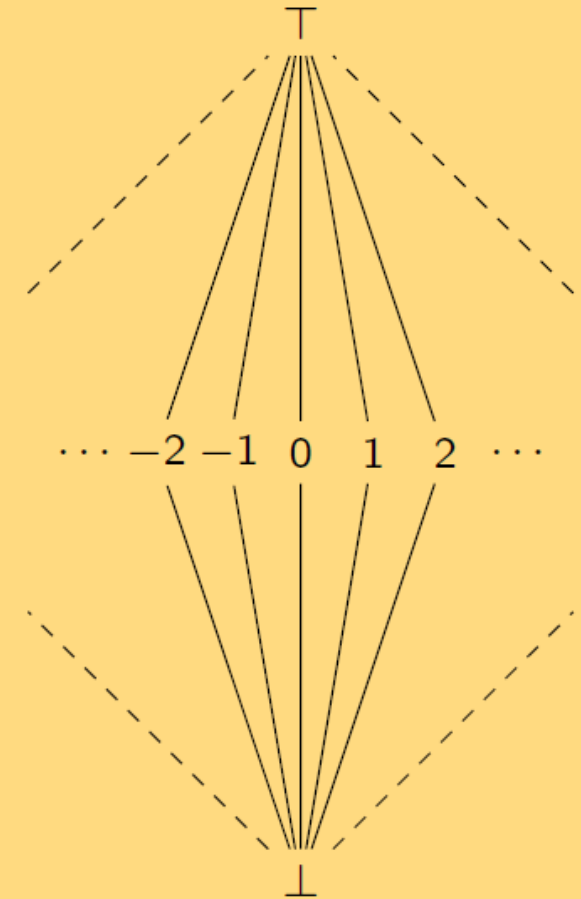
ABSTRACT INTERPRETATION (2)

Framework : abstract interpretation

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

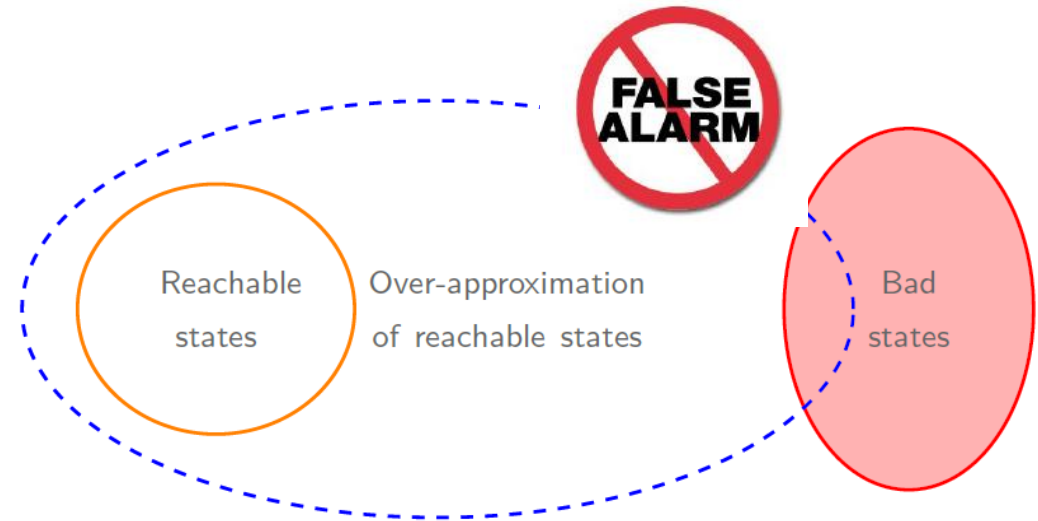
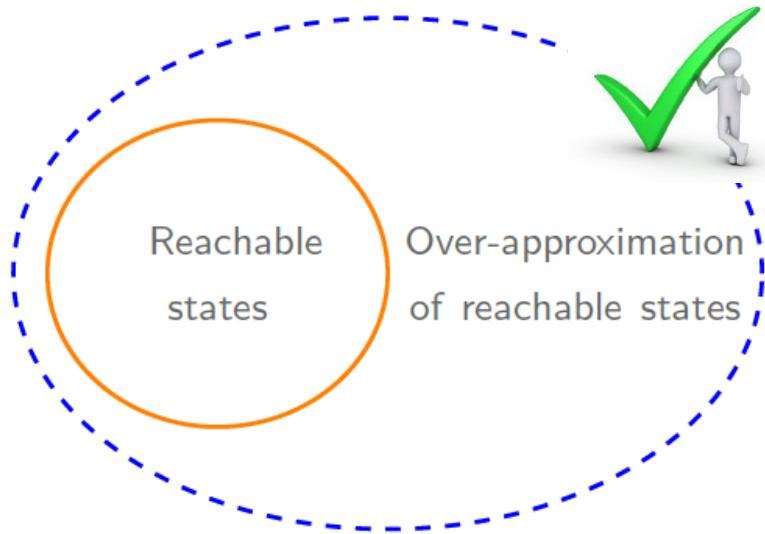


Generalize constant propagation



ABSTRACT INTERPRETATION

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



Abstract transfert: $x := a+10$

- $a == \{0\} \Rightarrow x == \{10\}$
- $a == [0..5] \Rightarrow x == [10..15]$
- $a == \top \Rightarrow x == \top$

Merge

- $[a == 5] \sqcap [a == 5] \Rightarrow [a == 5]$
- $[a == 5] \sqcap [a == 8] \Rightarrow [a == \top]$
- $[a == 5] \sqcap [a == 8] \Rightarrow [a == 5..8]$

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

EXOS & discussions

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

- Slicing

- Static vs dynamic



- Termination?



- Correct?
- Complete?

Key points:

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

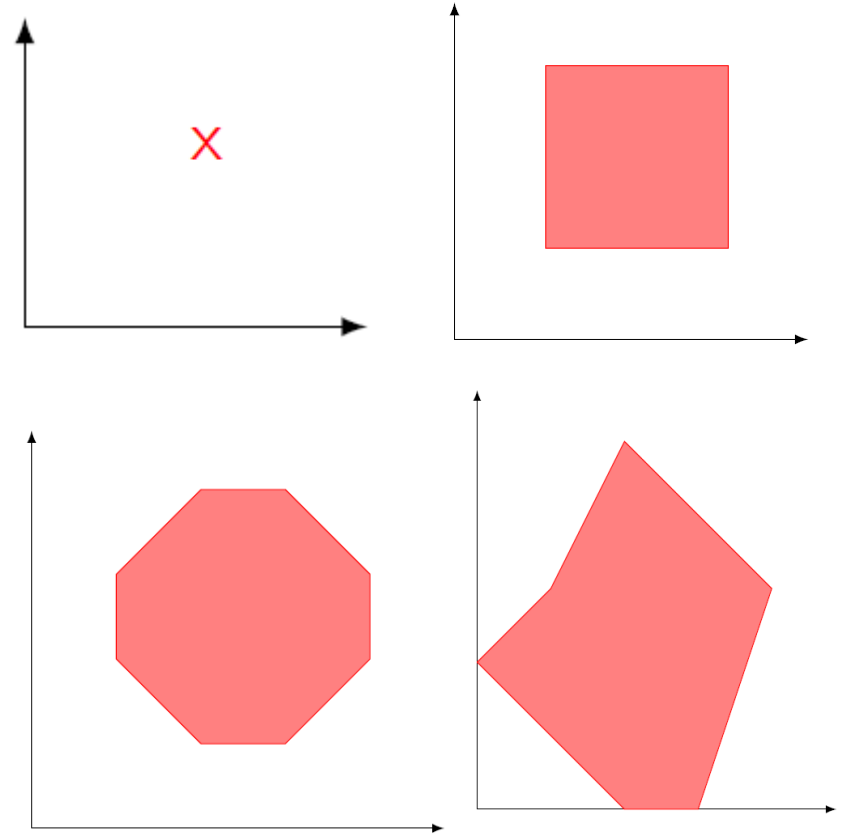
In practice:

- 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



- **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

- Correctness
- Completeness
- Efficiency
- Robustness

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

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

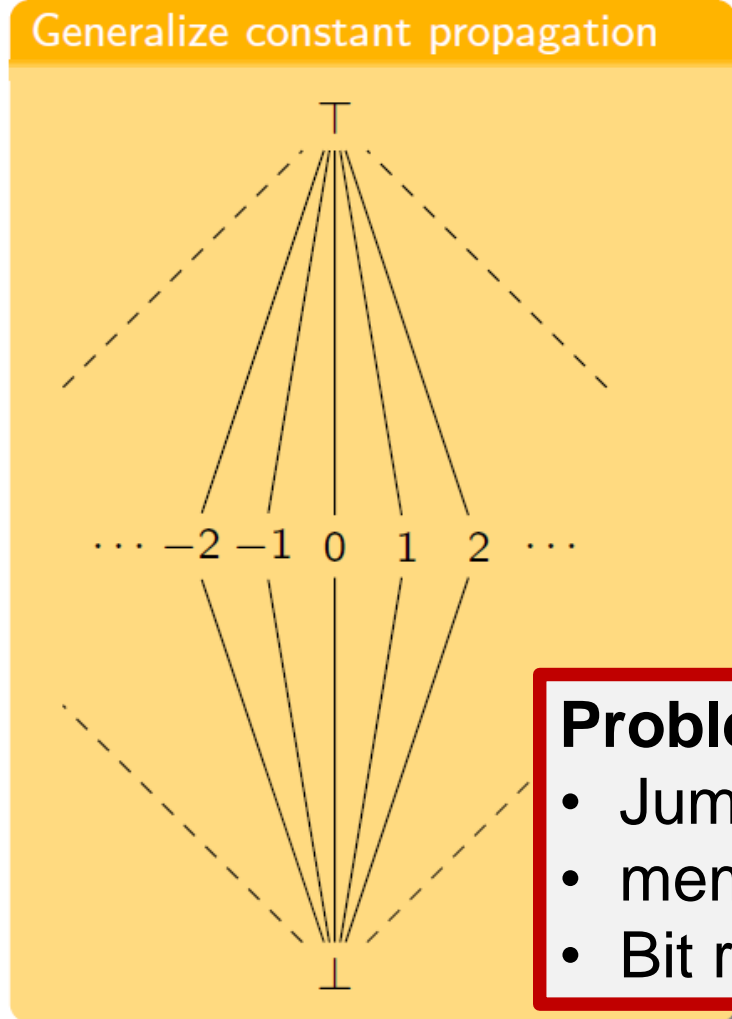
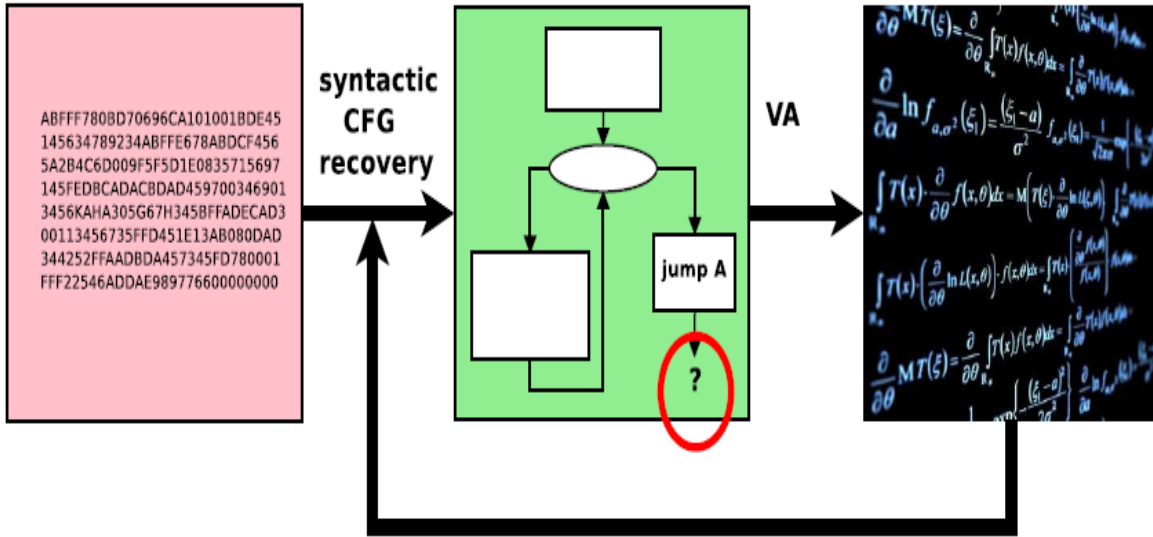
Diversity of architectures and instruction sets

- 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



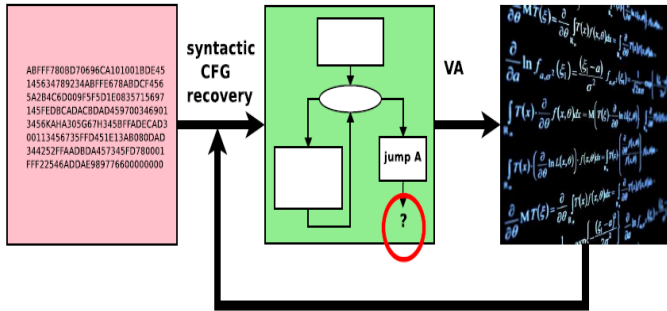
Framework : abstract interpretation

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

Problems

- Jump eax
- memory
- Bit reasoning

ABSTRACT INTERPRETATION IS VERY VERY HARD ON BINARY CODE

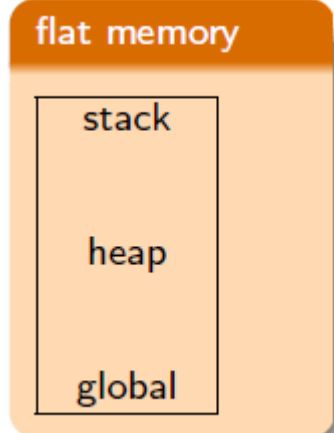


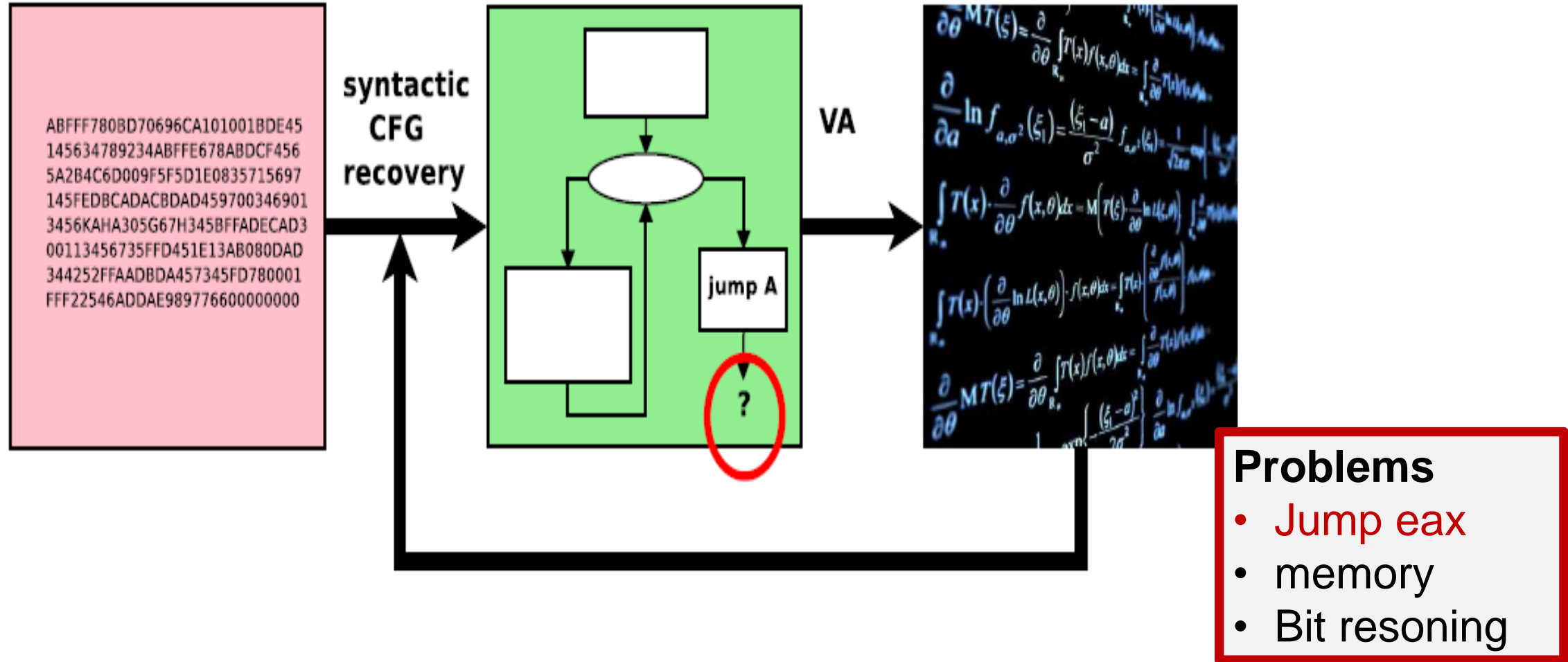
- ## Problems
- Jump eax
 - memory
 - Bit reasoning

```

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:
  
```

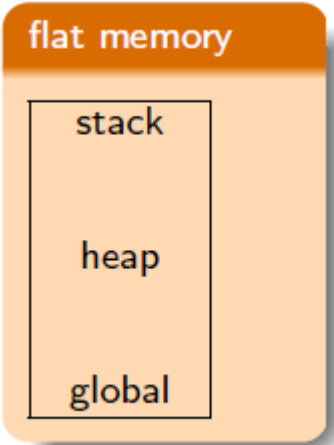
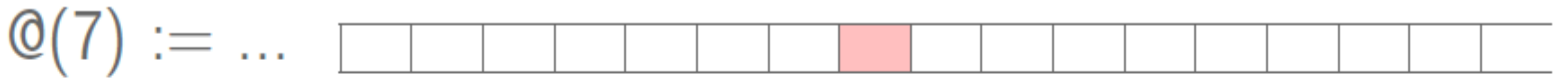




ISSUE: GLOBAL MEMORY

- ## Problems

 - Jump eax
 - **memory**
 - Bit reasoning



ISSUE: LACK of HIGH-LEVEL STRUCTURE

Problems

- Jump eax
- memory
- Bit reasoning

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)

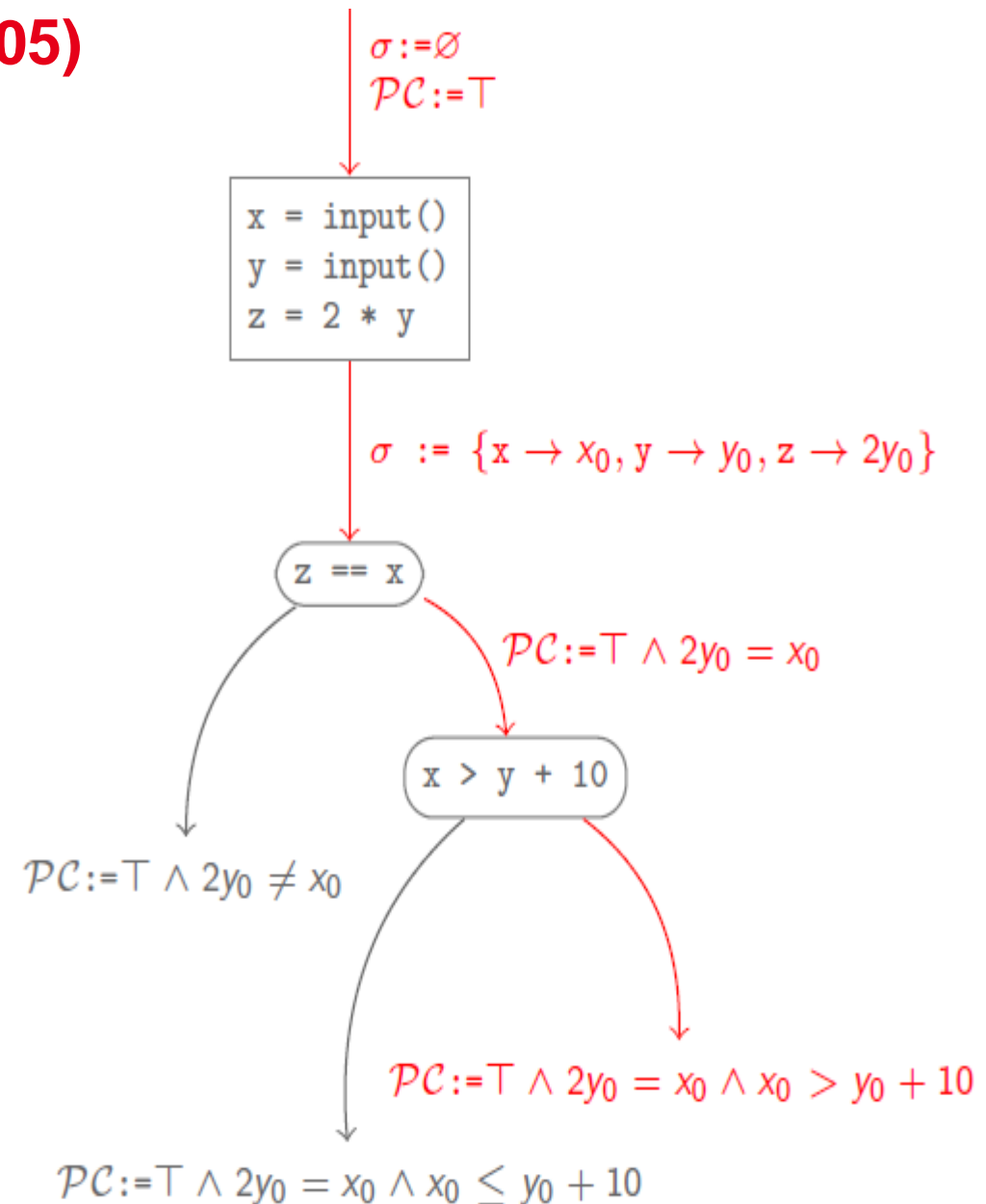
```

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 $f \Leftrightarrow$ input following the path
- Solve it with powerful existing solvers

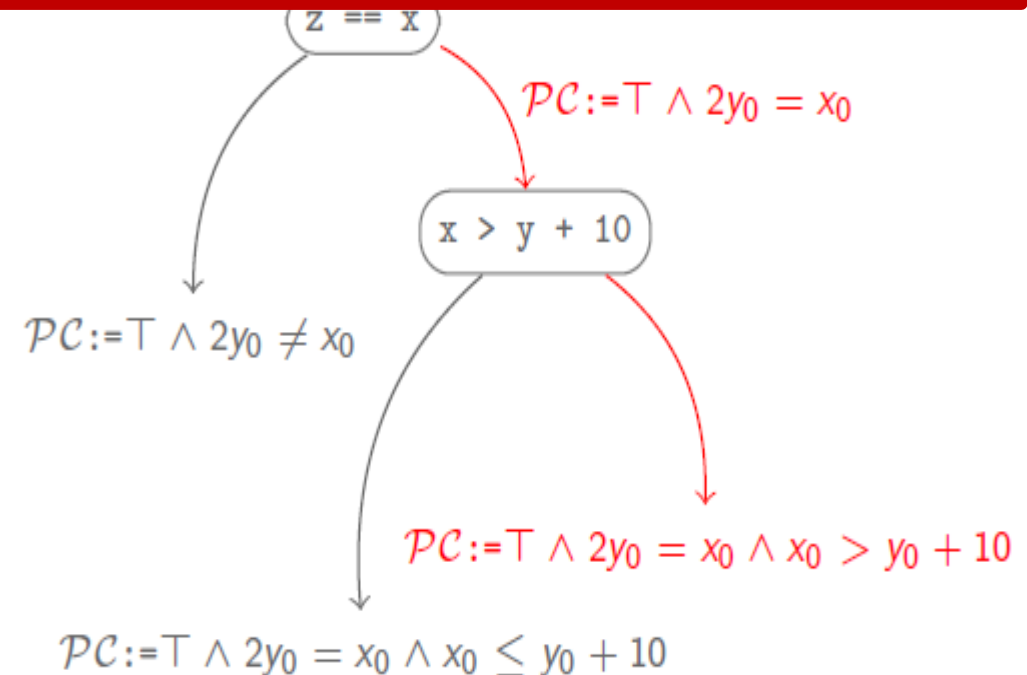


$\sigma := \emptyset$
 $PC := T$

```
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 $f \Leftrightarrow$ input following the path
- Solve it with powerful existing solvers

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)

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

SMT Solver

$Y_0 = 0 \wedge Z_0 = 3$

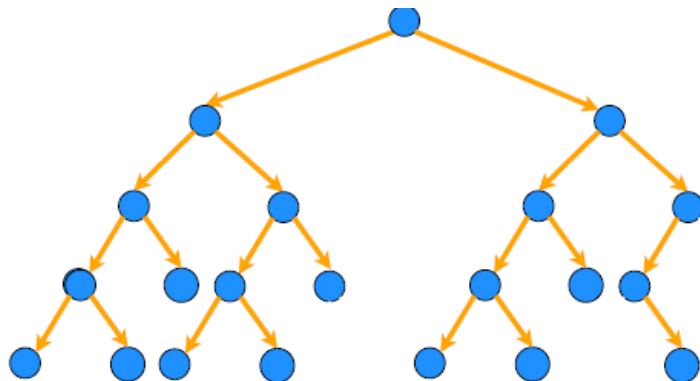
DSE: GLOBAL PROCEDURE

(DSE, Godefroid 2005)

input : a program P

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

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s)? get a new pair $\langle s, \sigma \rangle$
- 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**

x86

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

ARM

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

...

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

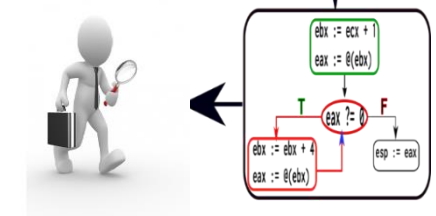
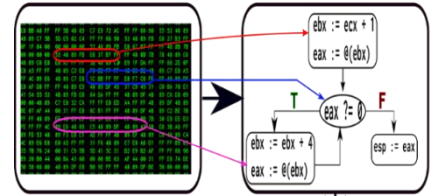
Static analysis



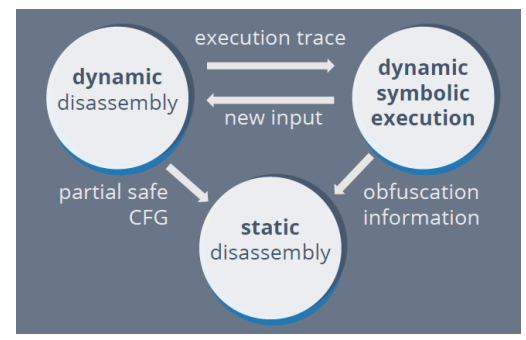
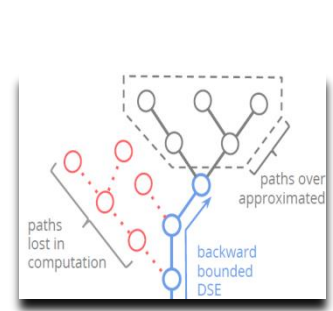
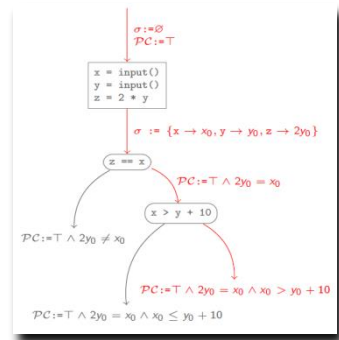
Symbolic execution

malware analysis

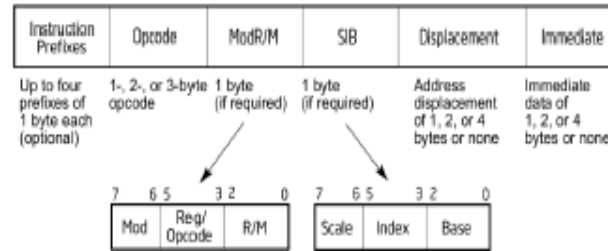
vulnerabilities



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



Key: INTERMEDIATE REPRESENTATION



81 c3 57 1d 00 00 $\xrightarrow{x86reference}$ 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)
```

Forward reasoning

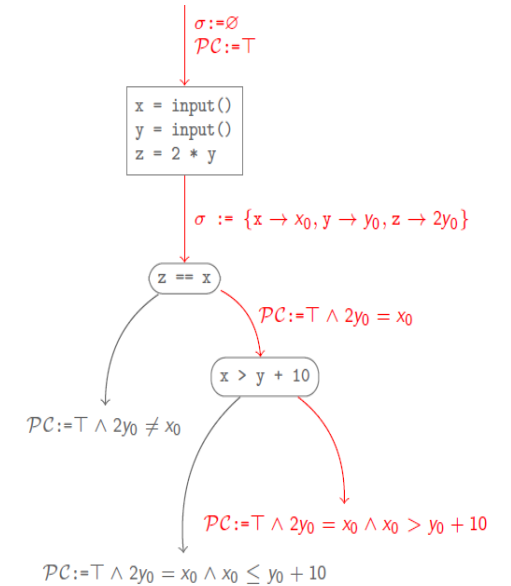
- Follows path
- Find new branch / jumps
- Standard DSE setting

Advantages

- Find new real paths
- Even rare paths

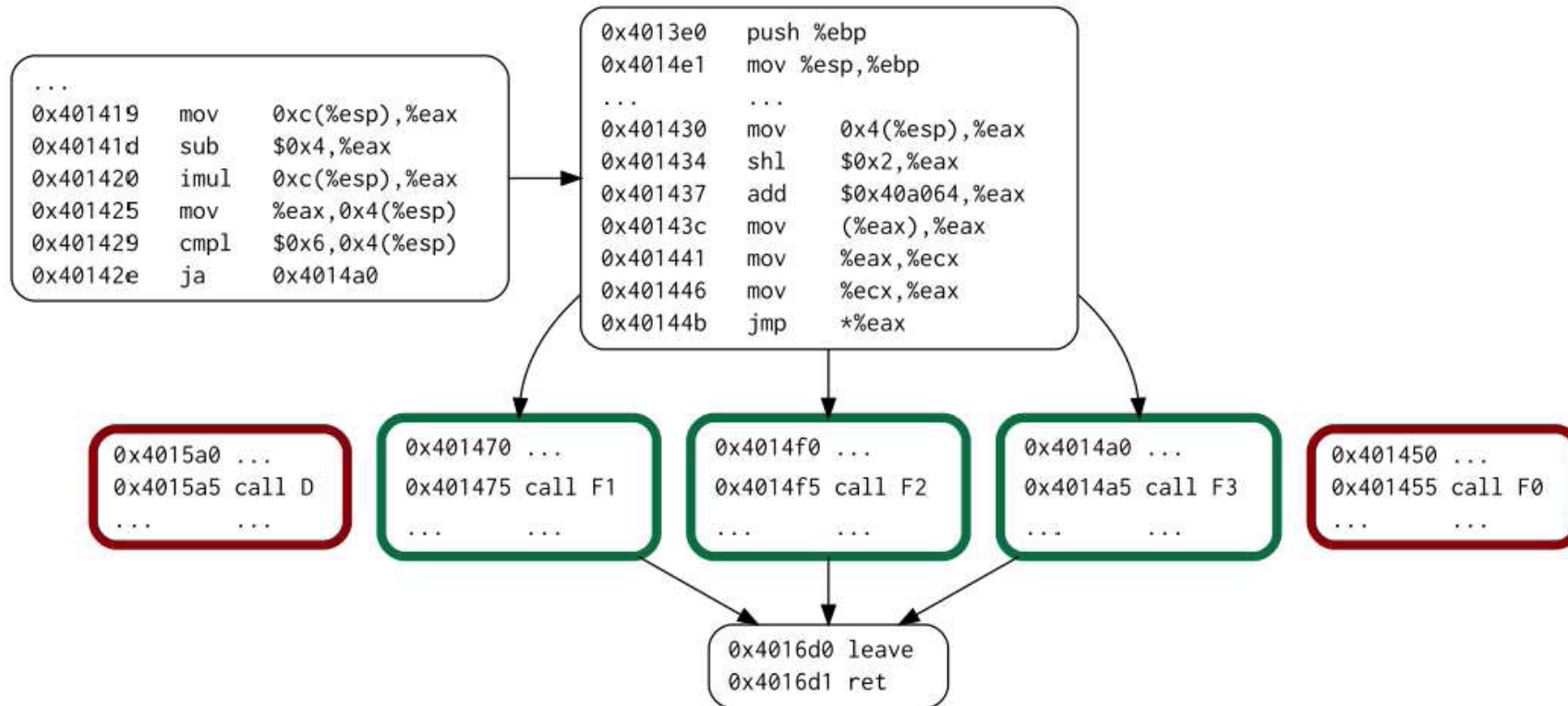
« dynamic analysis on steroids »

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



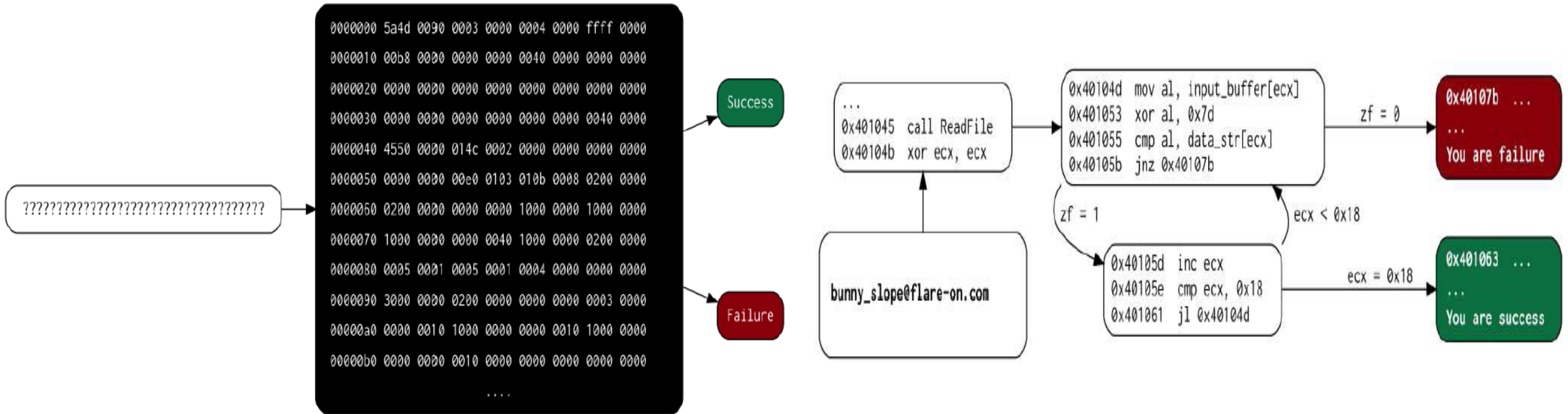
Solve for new dynamic targets

- 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 @

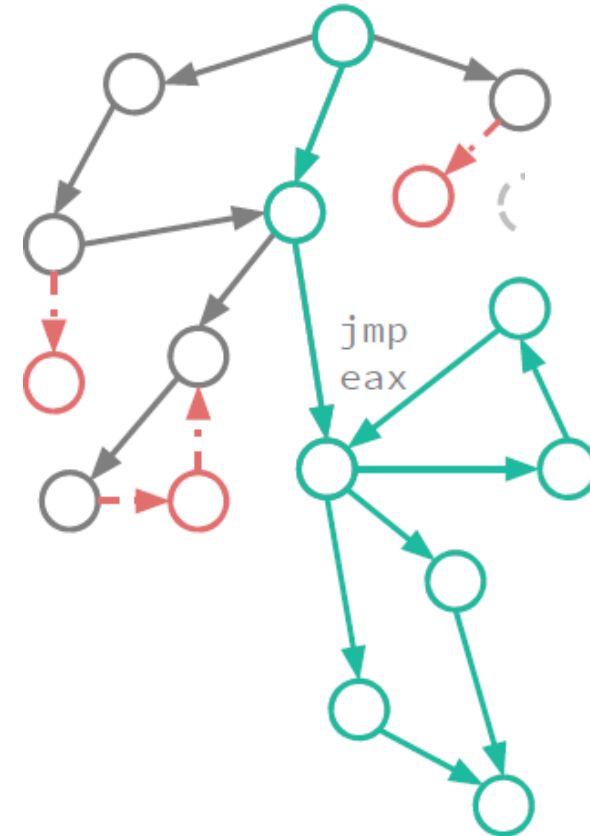


```
int main(int argc, char *argv[])
{
    struct {
        int canary;
        char buf[16];
    } state;
    my_strcpy(input, argv[1]);
    state.canary = 0;
    grub_username_get(state.buf, 16);
    if (state.canary != 0) {
        printf("This gets interesting!\n");
    }
    printf("%s", output);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

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?
- ...



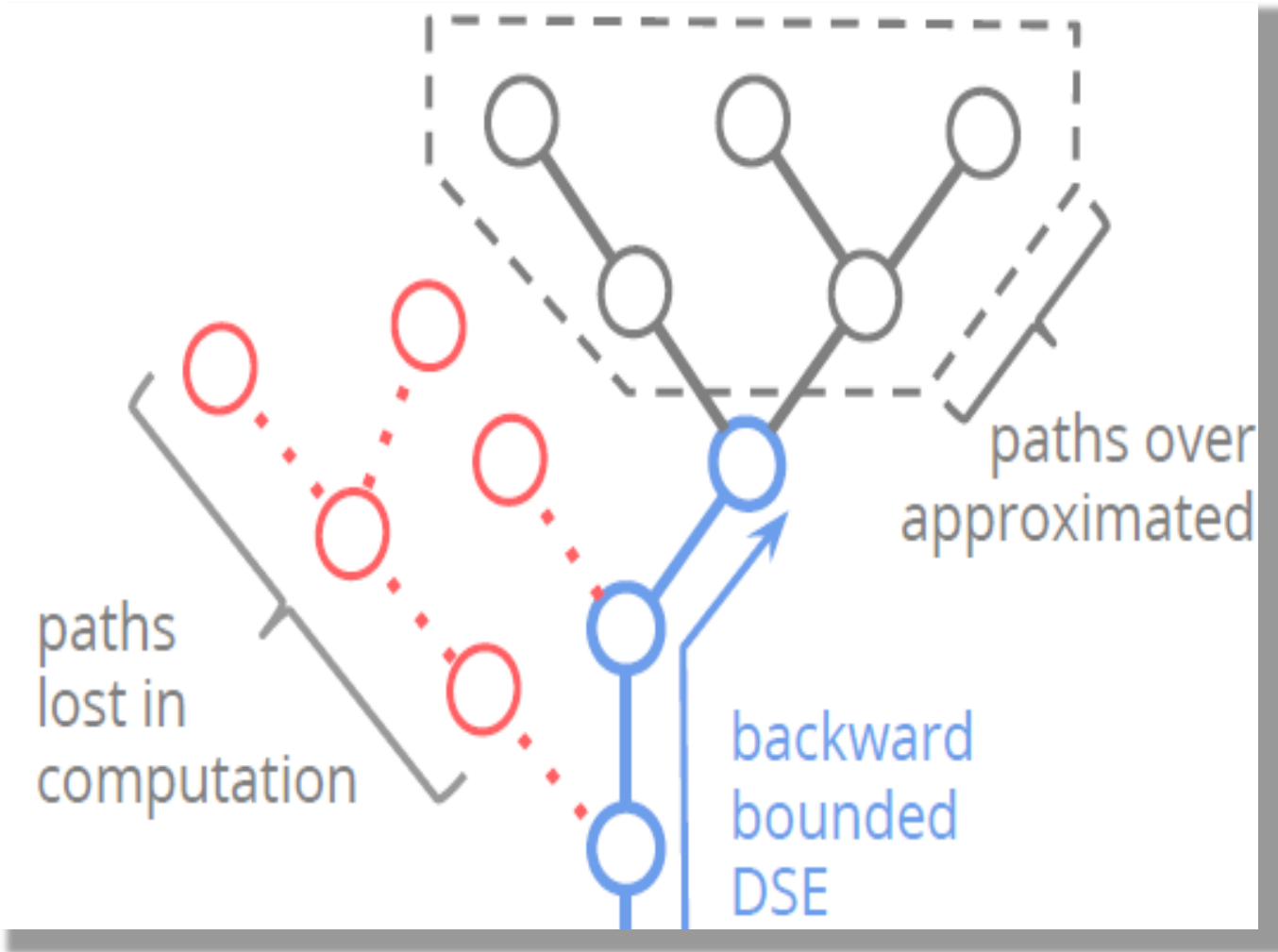
eg: $7y^2 - 1 \neq x^2$
(for any value of x, y in modular
arithmetic)

↓

```
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>
```

Not addressed by DSE

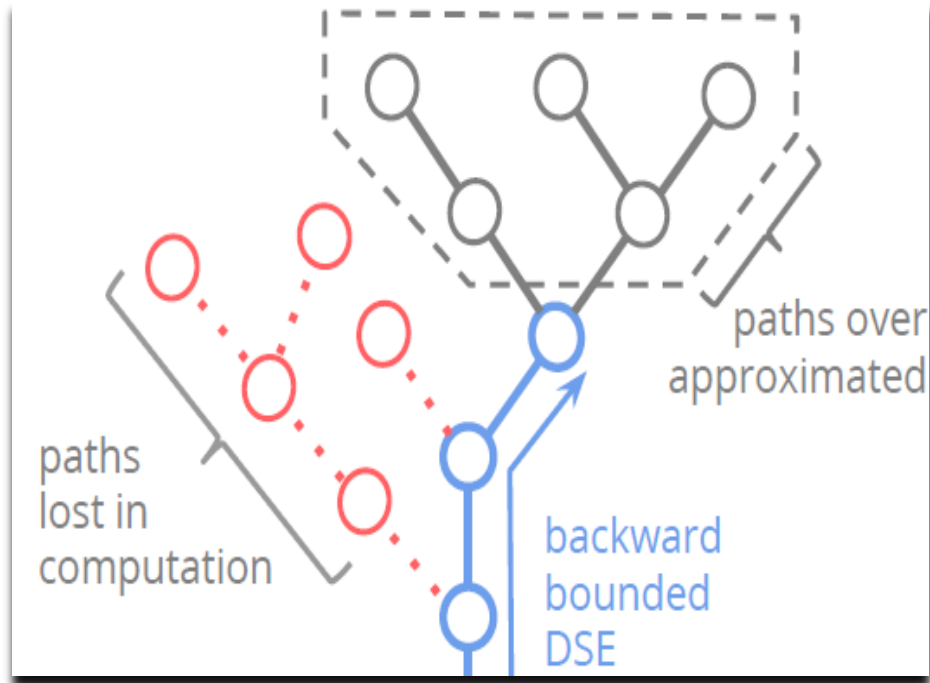
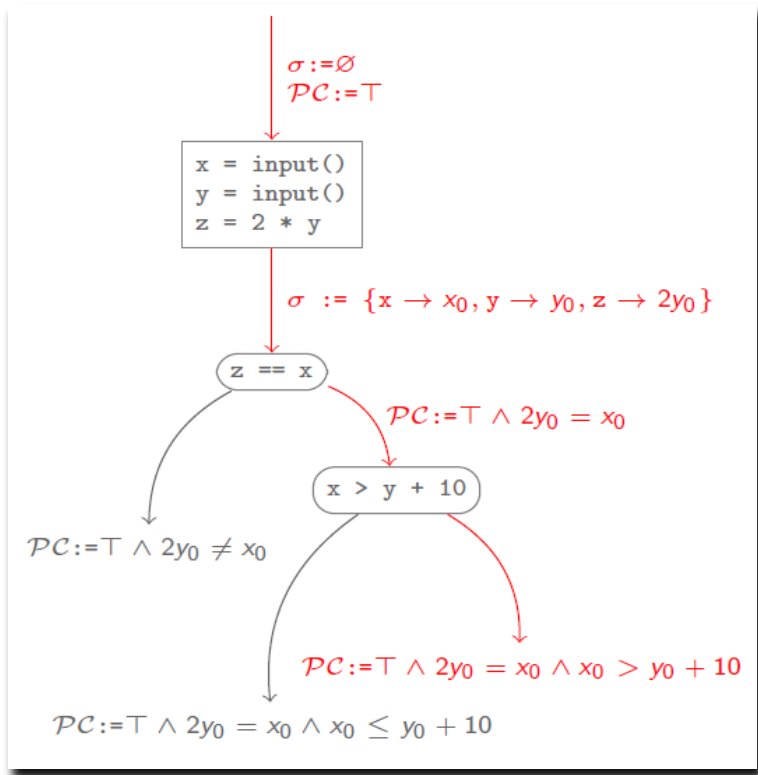
- Cannot enumerate all paths



- 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



Explore & discover

	(forward) DSE	bb-DSE
feasibility queries	●	●
infeasibility queries	●	●
scale	●	●

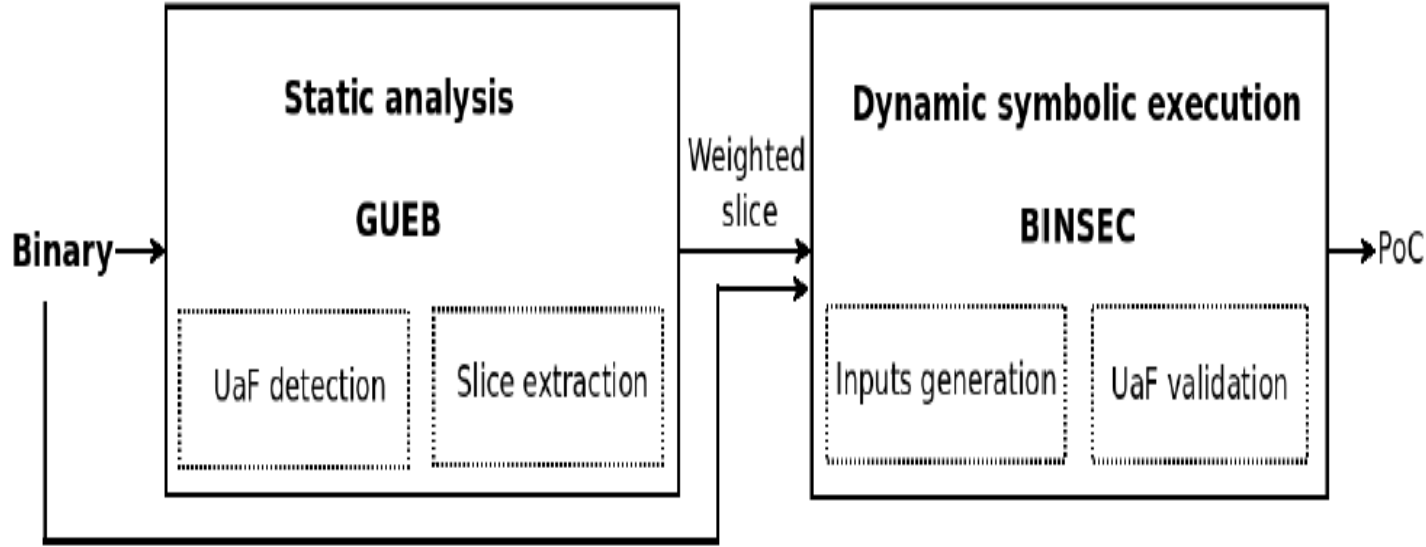
• Prove infeasible

- 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



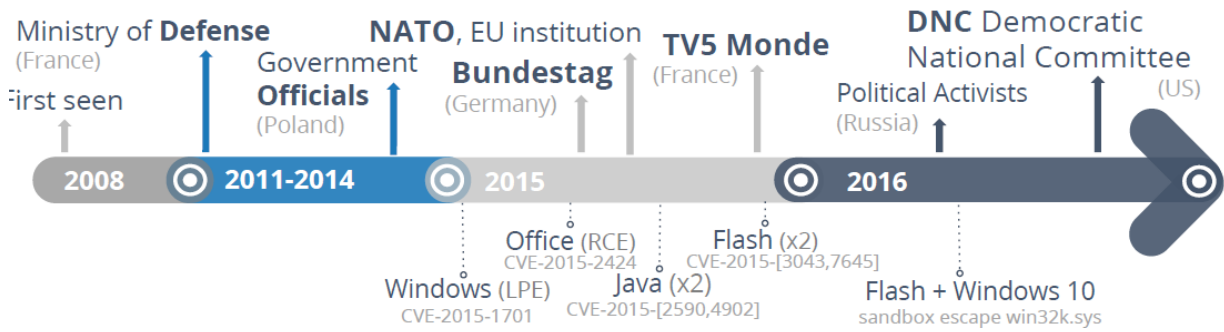
Find a needle in the heap!

Find a few new CVEs

```

4800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
0000 00b8 4500 0000 0000 00b8 820 0000 00b8 4500 000
bf0e 0821 0000 00b8 540 bf0e 0821 0000 00b
e5c7 0540 bf0e 0822 0000 0000 540 bf0e 0821 0000 00b
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec1
0000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 088
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08f
00c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
0000 00e9 d901 0000 c645 0548 bf0e 0802 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
0600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 010
f701 c645 f800 c645 f900 e301 0000 c645 f701 c645 f800 c64
fc00 740f c705 48bf 0e08 c645 fa02 807d fc00 40f c705 48b
0100 00e9 5901 0000 c645 0400 0000 e95e 0100 00e9 5901 000
c645 f900 c645 fa03 807d f701 c645 f800 c645 f900 c645 fa0
fe00 750a c705 48bf 0e08 0000 7410 807d fe00 750a c705 48b
fc00 750a c705 48bf 0e08 0500 0000 807d fc00 750a c705 48b
fe00 740f c705 48bf 0e08 0300 0000 807d fe00 740f c705 48b
0100 0000 901 0000 c645 0600 0000 e90e 0100 00e9 0901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0400 0000 e90e 0100 00e9 0901 0000 c645 f701 c64
0000 c645 f701 c645 f800 0000 00e9 d100 0000 c645 f701 c64
5004 807d fc00 7410 807d c645 f900 c645 fa04 807d fc00 741
48bf 0e08 0700 0000 807d ff00 750a c705 48bf 0e08 0700 000
ff00 740f c705 48bf 0e08 fc00 7415 807d ff00 740f c705 48b
0000 00e9 9900 0000 c645 0600 0000 e99e 0000 00e9 9900 000
c645 f900 c645 fa05 807d f701 c645 f800 c645 f900 c645 fa0
fe00 750a c705 48bf 0e08 fc00 7410 807d fe00 750a c705 48b
fc00 750a c705 48bf 0e08 0800 0000 807d fc00 750a c705 48b
fe00 750e 807d ff00 740c 0900 0000 807d fe00 750e 807d ff0
0600 0000 cb4b cb49 c645 c705 48bf 0e08 0600 0000 cb4b cb4
c645 f901 c645 fa02 807d f701 c645 f800 c645 f901 c645 fa0
5dc3 5589 e5c7 0540 bf0e 00b8 5400 0000 5dc3 5589 e5c7 054
4800 0000 5dc3 5589 e5c7 0812 0000 00b8 4800 0000 5dc3 558
0000 00b8 4500 0000 0000 00b8 5800 5589 e5c7 0540 bf0e 082
bf0e 0821 0000 00b8 5800 5589 e5c7 0540 bf0e 0821 0000 00b
e5c7 0540 bf0e 0822 0000 0000 5dc3 5589 e5c7 0540 bf0e 082
5dc3 5589 e583 ec10 c705 00b8 4900 0000 5dc3 5589 e583 ec1
0000 a148 bf0e 0883 f809 48bf 0e08 0100 0000 a148 bf0e 088
8b04 8548 e10b 08ff e0c6 0f87 0002 0000 8b04 8548 e10b 08f
90c6 45f9 00c6 45fa 00c7 45f7 00c6 45f8 00c6 45f9 00c6 45f
0000 00e9 d901 0000 c645 0548 bf0e 0802 0000 00e9 d901 000
c645 f900 c645 fa01 807d f701 c645 f800 c645 f900 c645 fa0
48bf 0e08 0300 0000 807d fb00 750a c705 48bf 0e08 0300 000
fc00 750a c705 48bf 0e08 fb00 7410 807d fc00 750a c705 48b
fc00 7415 807d fb00 740f 0900 0000 807d fc00 7415 807d fb0
9600 0000 e988 0100 00e9 c705 48bf 0e08 0600 0000 e988 010
  
```


Nicknames: APT28, Fancy Bear, Sofacy, Sednit, Pawn Storm



Two heavily obfuscated samples

- Many opaque predicates

Goal: detect & remove protections

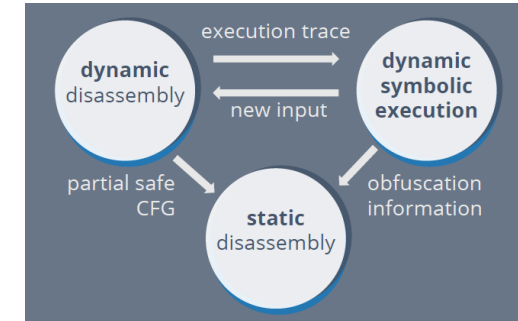
- Identify 50% of code as spurious
- Fully automatic, < 3h



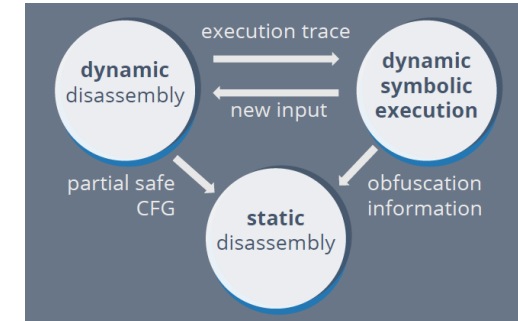
	C637 Sample #1	99B4 Sample #2
#total instruction	505,008	434,143
#alive	+279,483	+241,177

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

- **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



- **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 succinct 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

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