Semantics-based Obfuscation

On the Impossibility Result

Whenever we disclose the code we always disclose more that its input/output relation!!

CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS(1)

BY

H. G. RICE

1952

On the (Im)possibility of Obfuscating Programs*

Boaz Barak† Oded Goldreich† Russell Impagliazzo‡ Steven Rudich§
Amit Sahai¶ Salil Vadhan|| Ke Yang§

2001

How to verify programs?

1970

Formal Methods
Abstract Interpretation
Program Analysis
Verification

How to protect programs?

2001

?

Typical Obfuscating Transformations

Layout Obfuscation

```
y = x * 42      A = B * 42
```

Data Obfuscation

```
A[i]=10
int i , sum=0
for (i=0,i<10,i++)
      sum +=A[i]
```

data permutation

```
int P[]={7,5,1,3,8,9,0,2,4,6}
int* E(int a[], int p[], int i){
      return &(a[p[i]]);}

*(E(B,P,1))=10;
int i,sum=0;
for(i=0,i<10,i++)
      sum += *(E(B,P,i));
```

A	1	2	3	4	5	6	7	8	9	10

A[i] = B[P[i]]

B	7	3	8	4	9	2	10	1	5	6

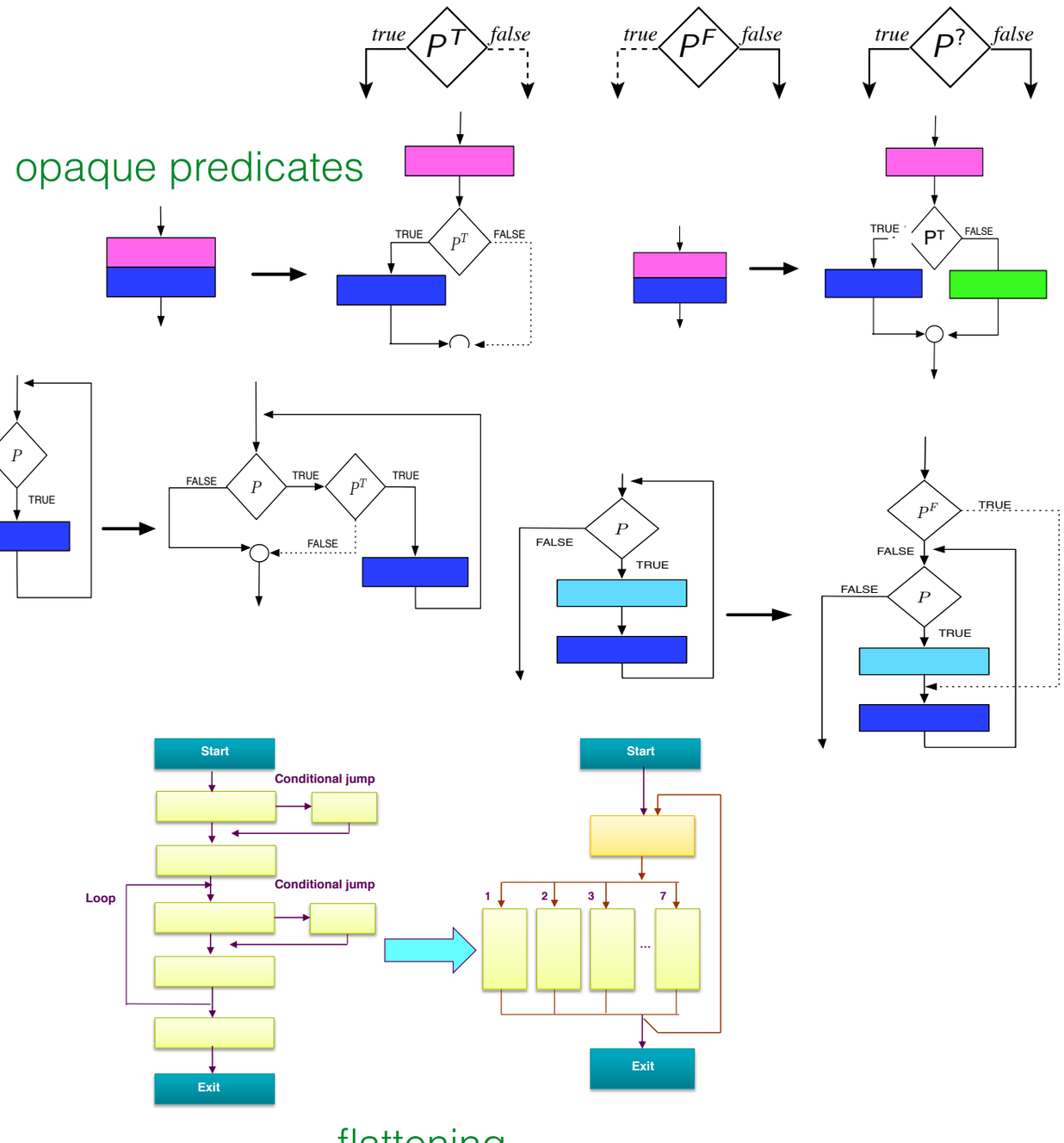
P	7	5	1	3	8	9	0	2	4	6

encoding-decoding data

P:
input x;
y := 2;
while x>0 **do**
  y:=y+2;
  x:=-x-1
endw
output y

t(P):
input x;
y := - 2;
while x>0 **do**
  y:=y+2;
  x:=-x-1
endw
if x = 0 **then** y:=y+4
output y

Control Flow Obfuscation



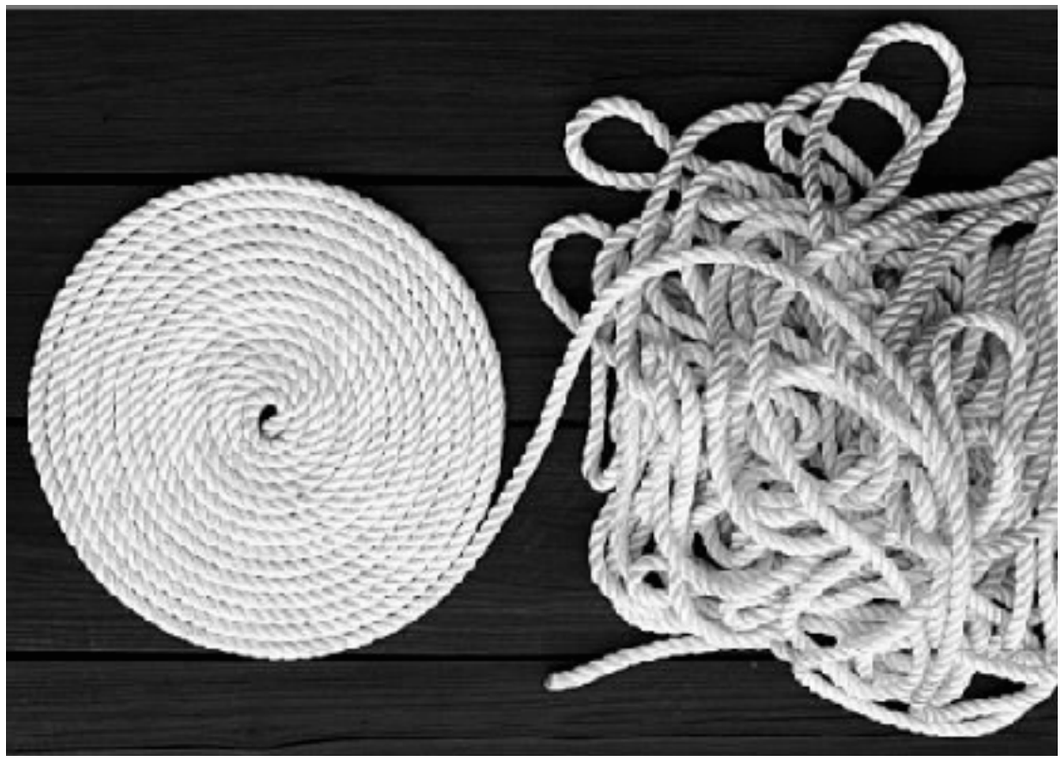
flattening

What does it mean to obfuscate/protect?

A program transformation o: Programs ⟶ Programs

is a code obfuscation if:

- o is potent, namely $o(P)$ is more complex than P
- o preserves the observational behaviour of programs



Collberg et al. POPL 1998

What does it mean to be potent?

The Attacker: Program Analysis

```
n := n0;
i := n;
while (i <> 0 ) do
      j := 0;
      while (j <> i) do
            j := j + 1
      od;
      i := i - 1
od
```

```
                                  n0 must be initially nonnegative
{n0>=0}  <-------------------     (otherwise the program does not
   n := n0;                       terminate properly)
{n0=n,n0>=0}
   i := n;
{n0=i,n0=n,n0>=0}
   while (i <> 0 ) do
      {n0=n,i>=1,n0>=i}
         j := 0;
      {n0=n,j=0,i>=1,n0>=i}
         while (j <> i) do
            {n0=n,j>=0,i>=j+1,n0>=i}
               j := j + 1  <-- j < n0 so no upper overflow
            {n0=n,j>=1,i>=j,n0>=i}
         od;
      {n0=n,i=j,i>=1,n0>=i}
         i := i - 1  <-- i > 0 so no lower overflow
      {i+1=j,n0=n,i>=0,n0>=i+1}
   od
{n0=n,i=0,n0>=0}
```

Program analysis observe semantic program properties

Ainuco(Semantics)

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

            }
      out(nl,nw,nc);}
```

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

            }
      out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw				
nl				

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

      }
out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl				

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

      }
out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

      }
out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

	c	nc	nw	nl
c				
nc	X	X		
nw				
nl				

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

      }
out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl				

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') nl++;

        }

    out(nl,nw,nc);}
```

```
Obfuscated()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') {if(nw <= nc) nl++};

        if(nl > nc) nw = nc + nl;

        elseif(nw > nc) nc = nw – nl;

        }
    out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X	X	X	X

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

            }
      out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

	c	nc	nw	nl
c				
nc	X	X		
nw	X	X	X	X
nl	X	X	X	X

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') nl++;

        }

    out(nl,nw,nc);}
```

```
Obfuscated()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') {if(nw <= nc) nl++};

        if(nl > nc) nw = nc + nl;

        elseif(nw > nc) nc = nw – nl;
    }
out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

	c	nc	nw	nl
c				
nc	X	X	X	X
nw	X	X	X	X
nl	X	X	X	X

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw – nl;

      }
out(nl,nw,nc);}
```

Always true

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }

      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;
Always false
            elseif(nw > nc) nc = nw – nl;
            }
      out(nl,nw,nc);}
```

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }
      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;
Always false elseif(nw > nc) nc = nw – nl;
            }
      out(nl,nw,nc);}
```

Example: add fake dependences

[Majumdar et al. ACM workshop on Digital Rights Management 2007]

```
Original()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') nl++;

            }
      out(nl,nw,nc);}
```

```
Obfuscated()
      int c, nl = 0, nw = 0, nc = 0, in;
      in = false;

      while (( c = getchar()) != EOF){
            nc++;

            if(c == ' ' || c = '\n' || c = '\t') in = false;

            elseif(in  == false) {in = true; nw++}

            if(c == '\n') {if(nw <= nc) nl++};

            if(nl > nc) nw = nc + nl;

            elseif(nw > nc) nc = nw - nl;

      }
      out(nl,nw,nc);}
```

Always false

The Attacker: Program Analysis

Majumdar et al. ACM Workshop on digital Rights Management 2007

```
Original()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') nl++;

        }

    out(nl,nw,nc);}
```

```
Obfuscated()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') {if(nw <= nc) nl++};

        if(nl > nc) nw = nc + nl;

        elseif(nw > nc) nc = nw – nl;

    }
out(nl,nw,nc);}
```

opaque predicate insertion that confuses data flow analysis

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

	c	nc	nw	nl
c				
nc	X	X	X	X
nw	X	X	X	X
nl	X	X	X	X

data obfuscation that confuses the analysis of Sign

P:
```
input x;
y := 2;
while x>0 do
   y:=y+2;
   x:= x-1
endw
output y
```

t(P):
```
input x;
y := - 2;
while x>0 do
   y:=y+2;
   x:= x-1
endw
if x = 0 then y:=y+4
output y
```

- $Sign([\![P]\!]) = \{\sigma \mid \exists n \geq 0.\, \sigma \in \langle 0+, \bot\rangle \to \langle 0+, +\rangle \to \langle +, +\rangle^n \to \langle 0, +\rangle\}$
- $Sign([\![t(P)]\!]) = \left\{ \sigma \;\middle|\; \begin{array}{l} \exists n \geq 0. \\ \sigma \in \langle 0+, \bot\rangle \to \langle 0+, -\rangle \to \langle +, 0+\rangle^n \to \langle 0, +\rangle \end{array} \right\}$
- $\mathfrak{I}([\![t(P)]\!]) = \mathfrak{I}([\![P]\!])$

Attack as Abstract Domains

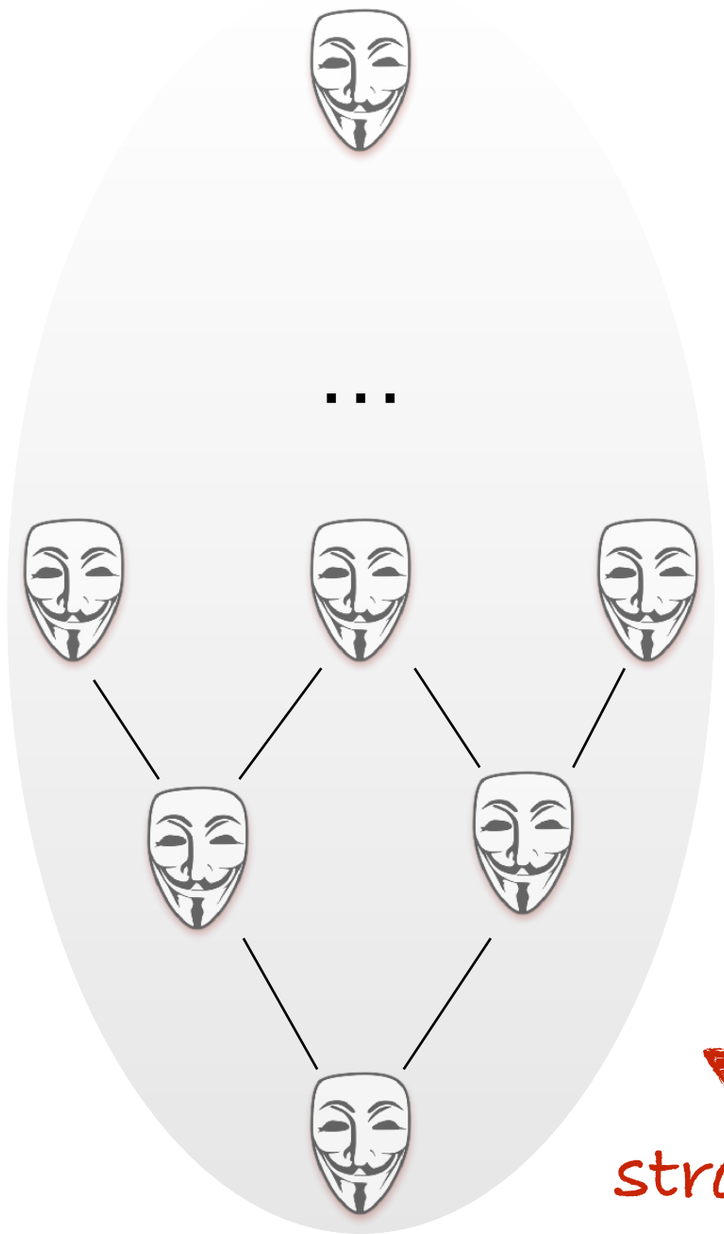
✓ Static analysis corresponds to an abstract execution of the program (the abstract domain is the one representing the property of interest of the attacker)

✓ Dynamic analysis corresponds to an abstract observation of the concrete execution (considers only certain aspects of certain executions)

Consider a property $\varphi \in \mathit{uco}(\wp(\Sigma^*))$ then:

- static attack $\varphi \circ [\![P]\!] \circ \varphi$
- dynamic attack $\varphi \circ [\![P]\!]$

The Attacker: Program Analysis

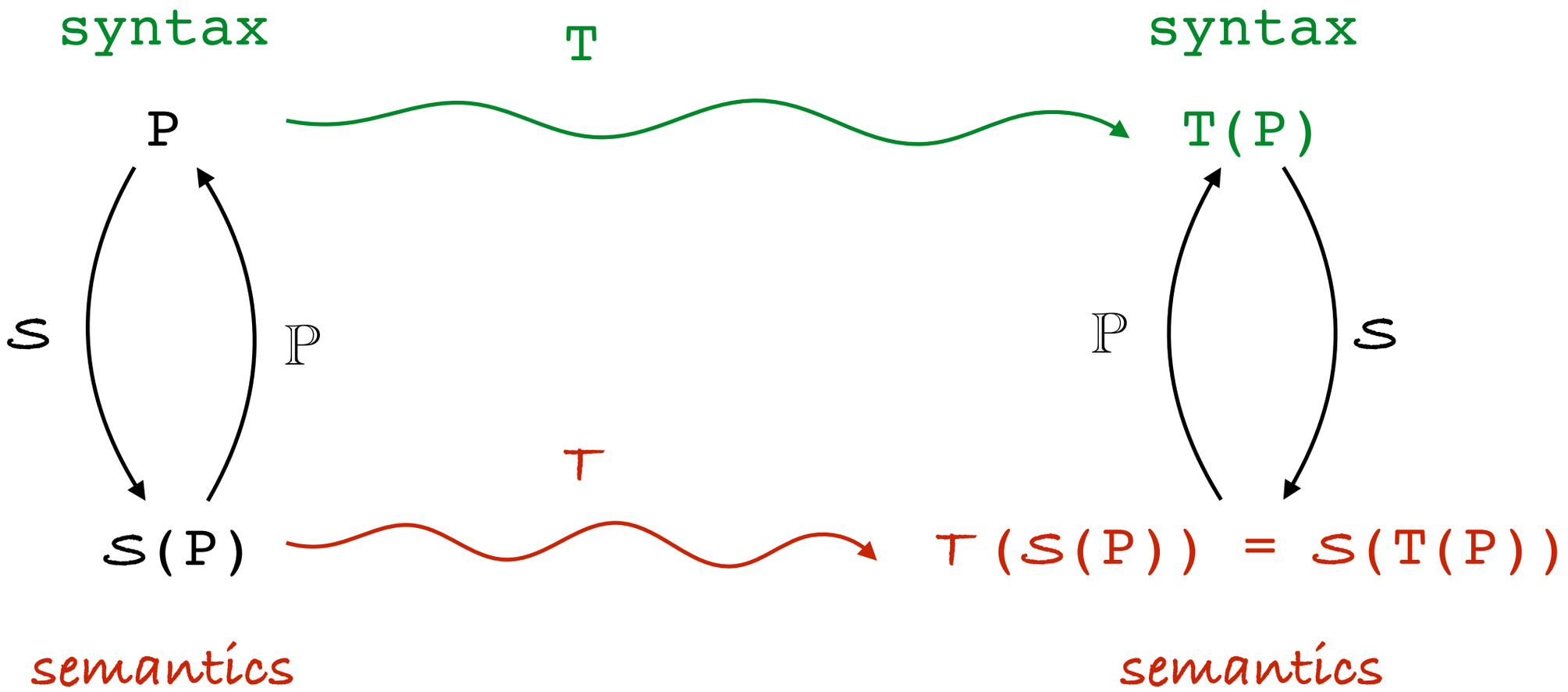


Attackers are ordered wrt the precision of their observation of program semantics

Program analysis observes semantic program properties

Ainuco (Semantics)

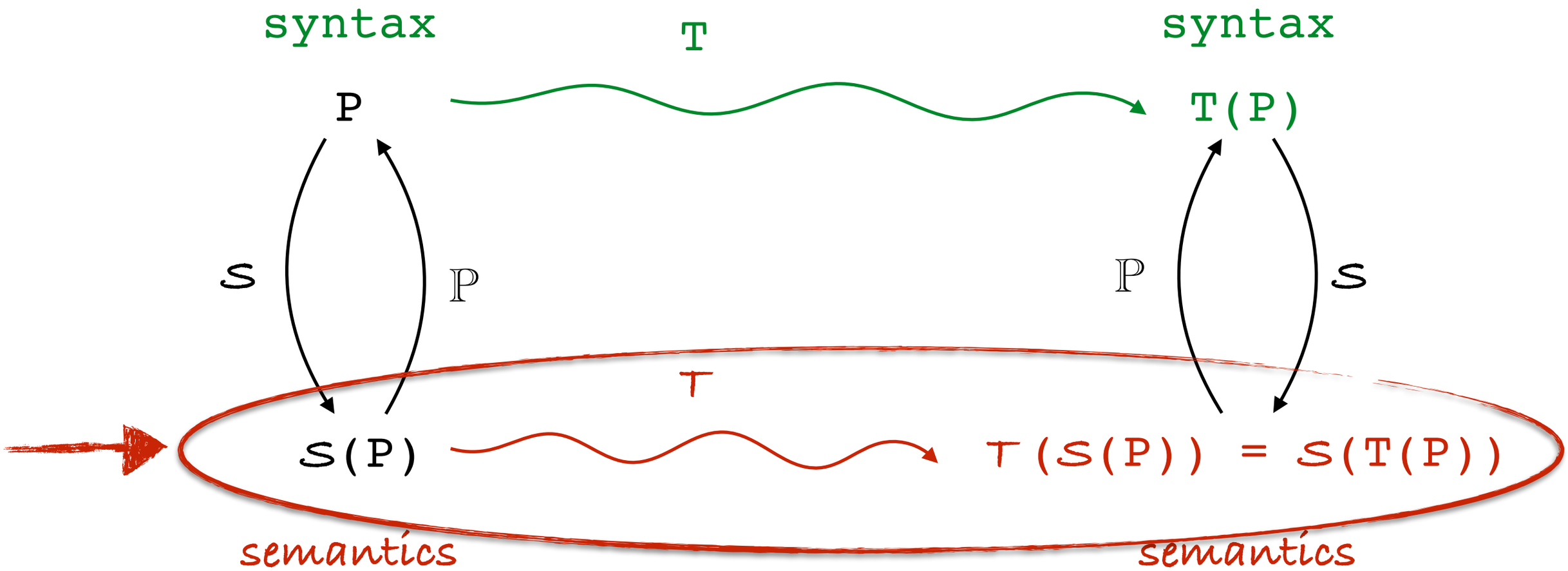
Cousot's Program Transformations



Programs can be seen as abstractions of their semantics.
Program transformations abstract semantic transformations.

Cousot and Cousot POPL 2002

Cousot's Program Transformations



Programs can be seen as abstractions of their semantics.
Program transformations abstract semantic transformations.

Cousot and Cousot POPL 2002

Semantic Notion of Potency

A program transformation $\mathtt{T}$ is **potent** with respect to an attacker $\mathcal{A}$ *in uco (Semantics)* and a program P when:

$$\mathcal{A}(\mathcal{S}(P)) \neq \mathcal{A}(\mathcal{S}(\mathtt{T}(P))) = \mathcal{A}(\mathtt{T}(\mathcal{S}(P)))$$

Characterize the obfuscating behavior of a program transformation T in terms of the most concrete property it preserves on program semantics

$\mathcal{O}_{\mathtt{T}}$ *in uco (Semantics)*

Semantic Code Obfuscation

Constructive characterization of the most concrete property preserved by a program transformation T, where τ(s(P)) = s(T(P)))

$$o_\tau = \textbf{glb}\ \{o \text{ in uco (Semantics)} \mid \text{for every P: } o(s(P)) = o(\tau(s(P)))\}$$

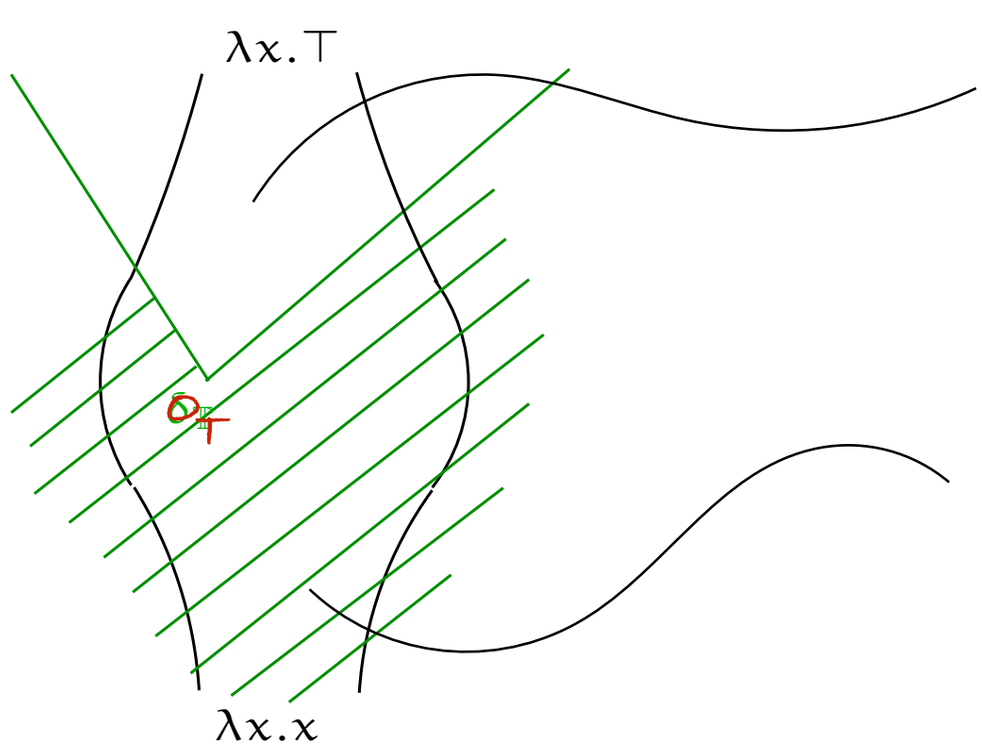
Characterize the obfuscating behavior of a program transformation T in terms of the most concrete property it preserves

o_τ in uco (Semantics)

Semantic Code Obfuscation

Constructive characterization of the most concrete property preserved by a program transformation $\mathtt{T}$, where $\tau(s(P)) = s(\mathtt{T}(P))$

$$o_\tau = \textbf{glb}\ \{o \in uco\ (Semantics) \mid \text{for every } P: o(s(P)) = o(\tau(s(P)))\}$$

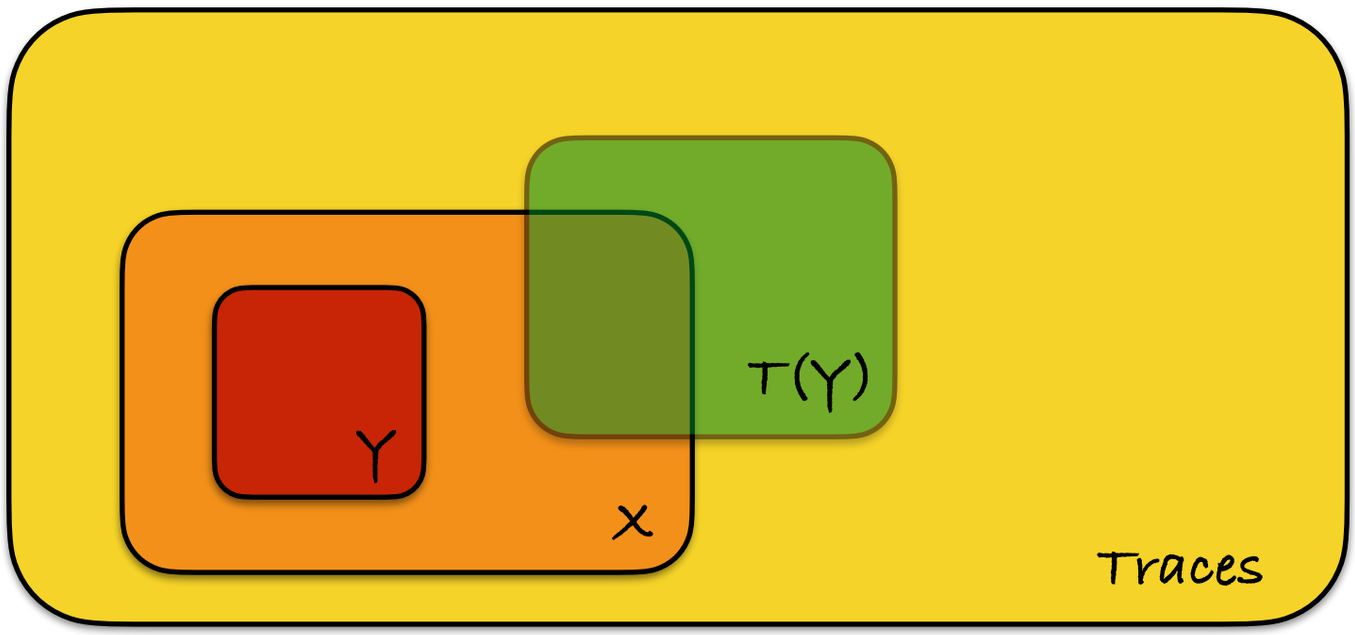


Preserved, not obfuscated

Not preserved, obfuscated

Costruttive Characterization

$T: \wp(Traces) \longrightarrow \wp(Traces)$



$Pres_{T,P}(X) = \text{true}$

$\forall Y \subseteq Traces(P) : Y \subseteq X \Rightarrow \mathcal{T}(Y) \subseteq X$

Most concrete property preserved by τ on program P

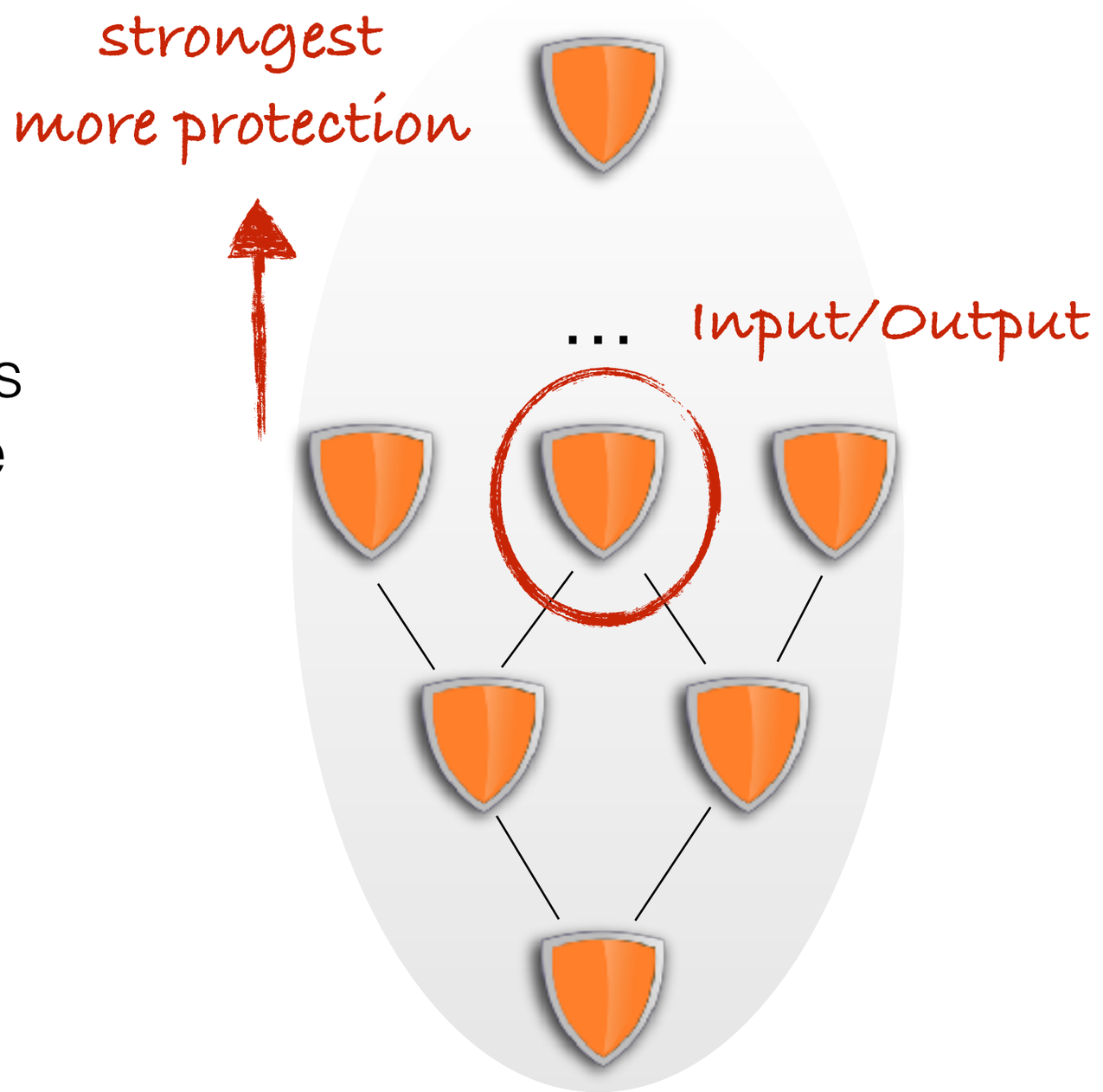
$$K_{P,\mathcal{T}} = \{X \in \mathcal{P}(Traces) \mid Pres_{P,\mathcal{T}}(X)\}$$

Most concrete property preserved by τ on all programs

$$\mathcal{O}_{\mathcal{T}} = \bigvee_{P} K_{P,\mathcal{T}}$$

Semantic Code Obfuscation

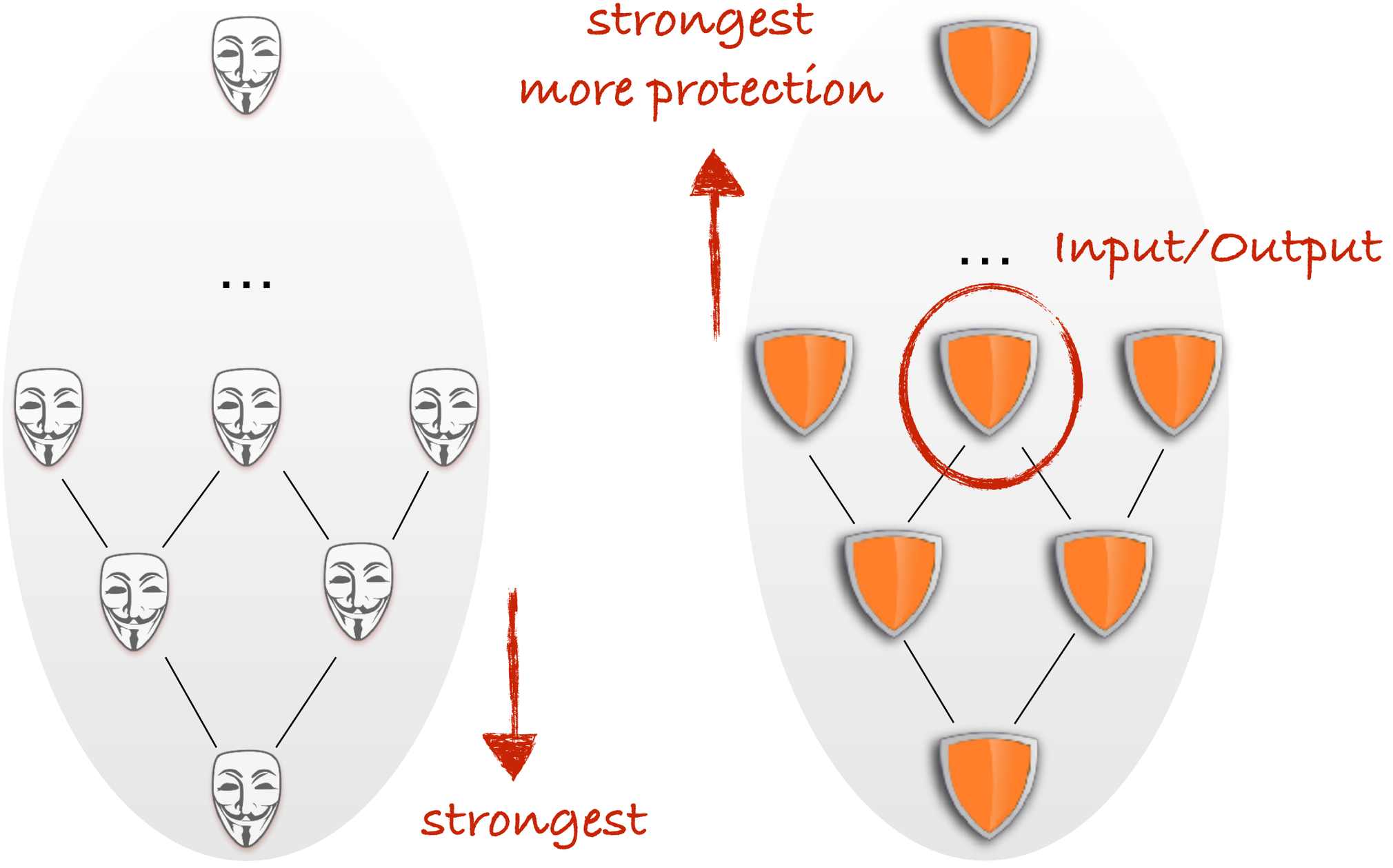
Obfuscating techniques can be ordered wrt the properties that they preserve of program's semantics



Program transformations preserve semantic properties

O_T in uco (Semantics)

Semantic Code Obfuscation



Semantics-based Code Obfuscation

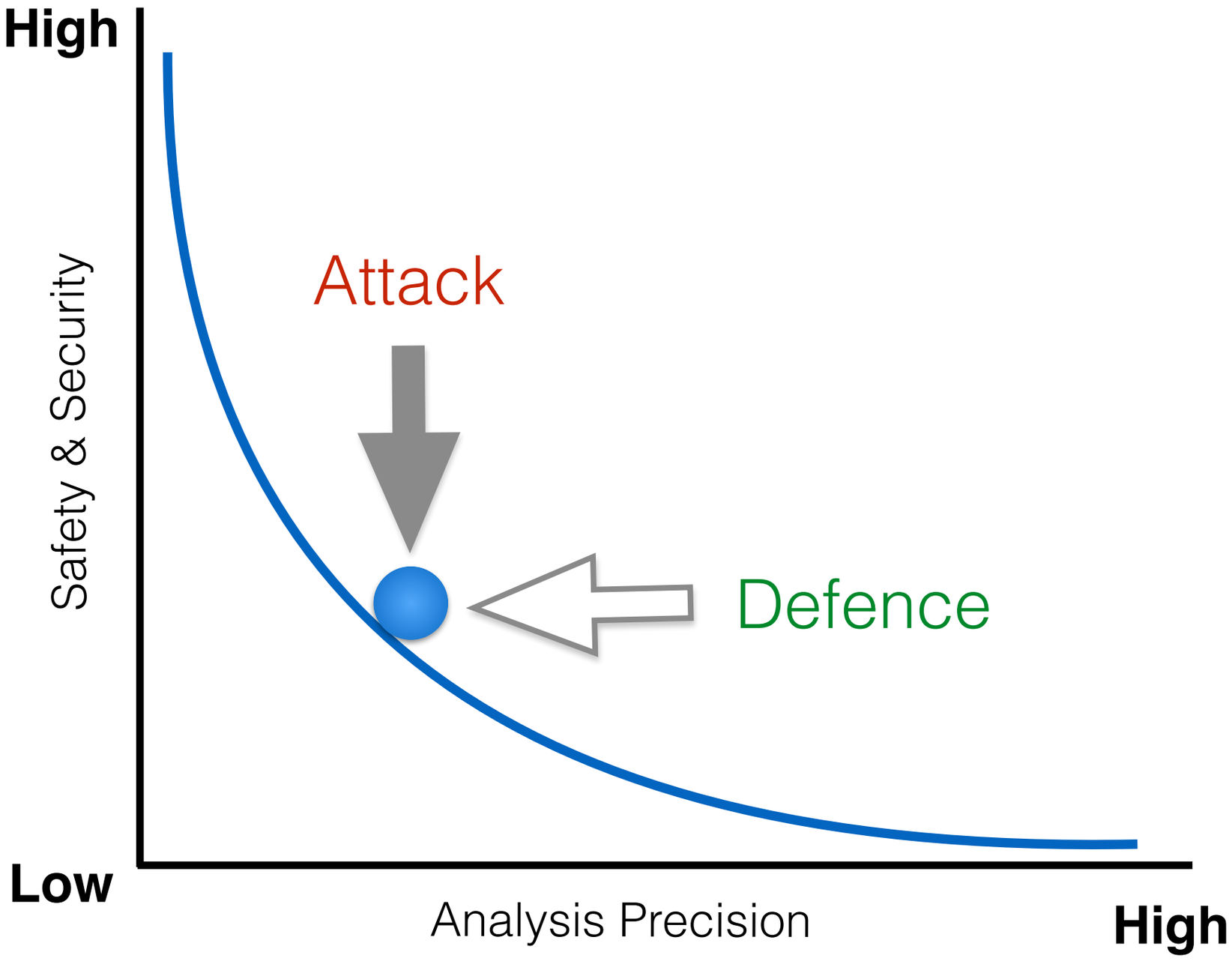
THUS

* Formally prove if a transformation T is potent wrt an attacker A

* Compare the potency of obfuscations, by comparing the class of attackers that they are able to defeat

* Property-Based obfuscation strategy based on the specification of the properties to conceal/protect and reveal/disclose

Code Protection



The attacker reverse engineer code

Breaking Opaque Predicates

Breaking opaque Predicates

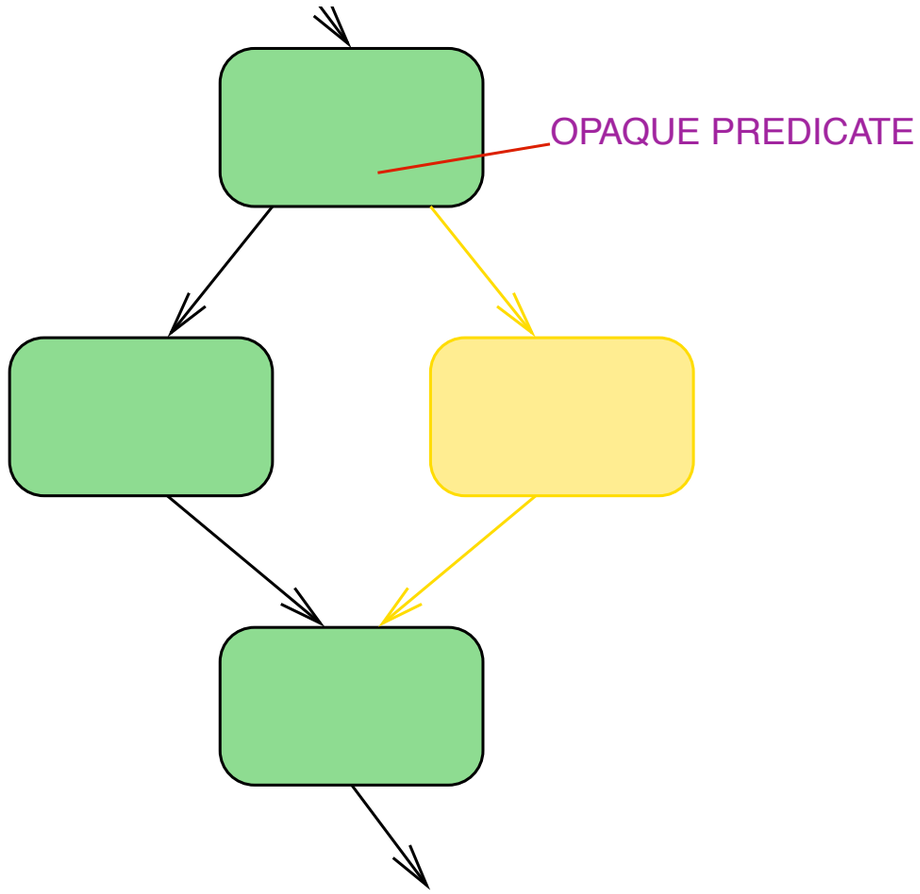
```
...
x1 ← ...;
x2 ← ...;
...
b ← f(x1, x2, ...);
if b goto ...
```

- ✓ Find the instruction to make up *f(x₁,x2,…)*
- ✓ Find the inputs to *f*, i.,e., *x₁,x₂…*
- ✓ Find the range of values *R₁* of *x₁*,…
- ✓ Compute the outcome of *f* for all possible values
- ✓ Kill the branch if *f* is always TRUE

Breaking opaque Predicates

- ✓ How to make attacker's task *more difficult*? Make it harder to:
  - Find $f(x_1,x2,\ldots)$
  - Find the inputs $x_1,x_2\ldots$ to f
  - Find the ranges $R_{1,}R_{2,\ldots}$ of $x_1,x_2\ldots$
  - Determine the outcome of f *for all argument values*

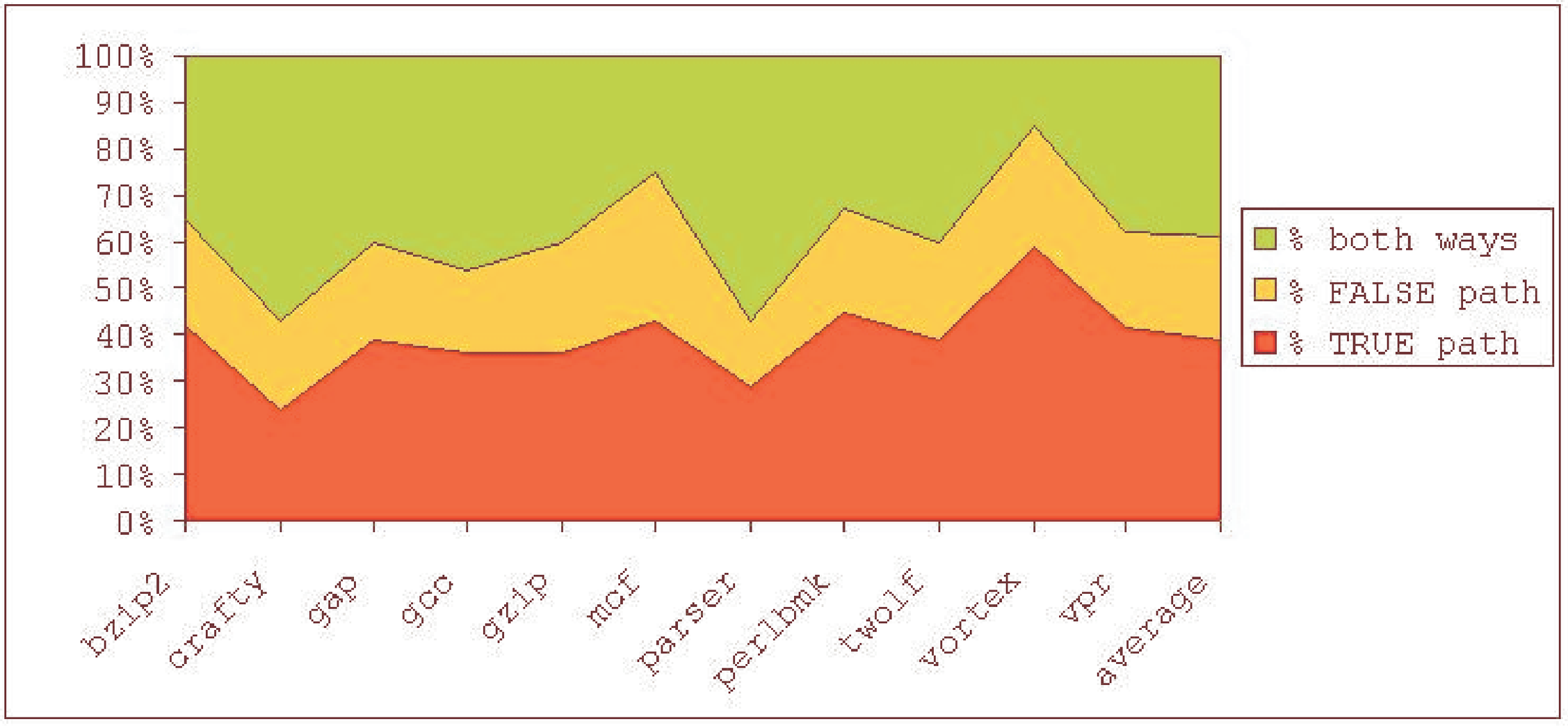
Dynamic Attack



- Observation of the program execution on *several inputs*
- *Limited set of inputs* leads to *false positives*

Dynamic Attack Results

SPECint2000 benchmarks on reference input



Dynamic attack is imprecise

Breaking numerical opaque predicates

- ✓ Attack opaque predicates confined to a single basic block
- ✓ Assume that the instructions that make up the predicate are contiguous
- ✓ Start at a conditional jump instruction j and incrementally extend it with the 1,2,… instructions until an opaque predicate (or beginning of a basic block) is found

- ▸ Brute force evaluation
- ▸ Abstract interpretation

Breaking numerical opaque predicates

Consider the simple opaque predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

(1)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(2)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(3)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(4)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

Consider every possible value of variable x = 2^{16}

The approach is *not efficient*: It takes 8.83 seconds to recognize 1 opaque predicate

Abstract domain of Parity

Consider the simple opaque predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

Assume that **x** is **odd**

```
x = odd number;
y = x * x;
y = y + x;
z = y % 2;
b = z==0;
if b ...
```

```
x                = odd;
y = x *_a x      = odd *_a odd = odd;
y = y +_a x      = odd +_a odd = even;
z = y %_a 2      = even mod 2 = 0;
b = z==0;        = true
if b ...
```

Abstract domain of Parity

Consider the simple opaque predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

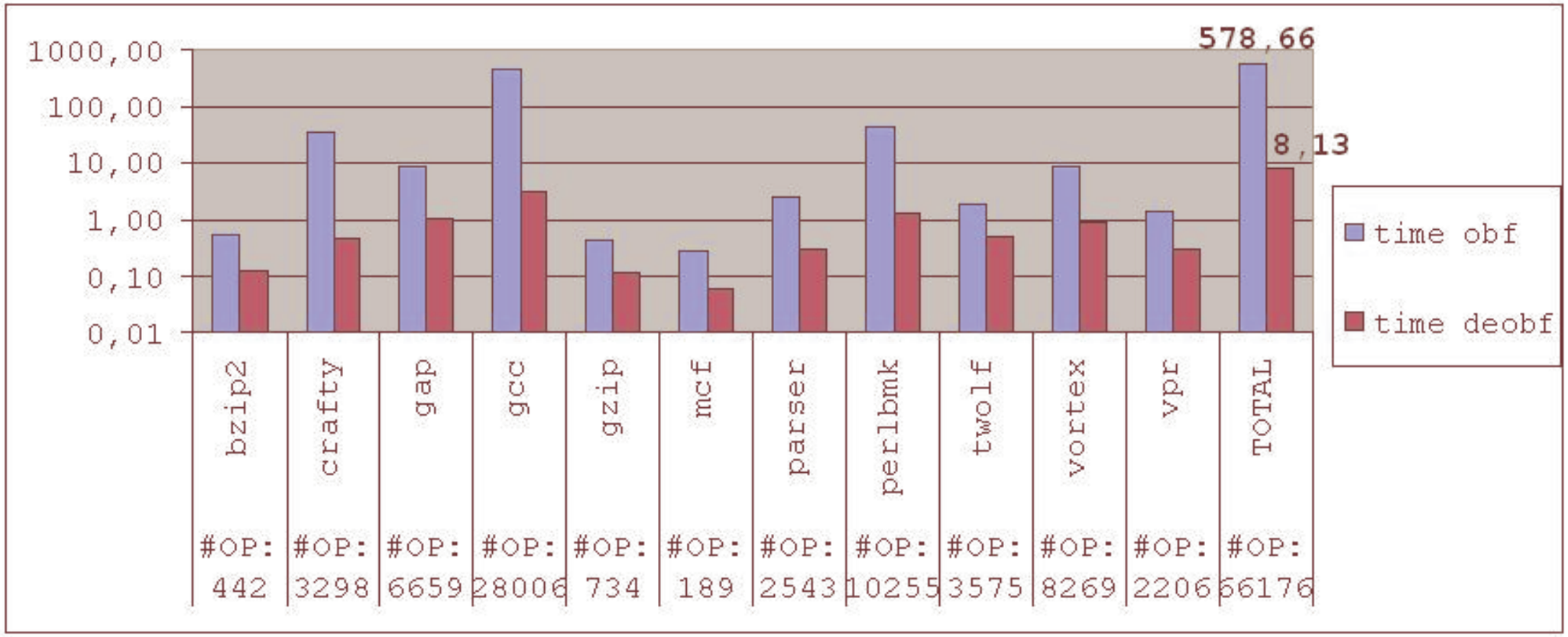
Assume that **x** is **even**

```
x = even number;
y = x * x;
y = y + x;
z = y % 2;
b = z==0;
if b ...
```

```
x                = even;
y = x *_a x      = even *_a even = even;
y = y +_a x      = even +_a even = even;
z = y %_a 2      = even mod 2 = 0;
b = z==0;        = true
if b ...
```

We have *efficiently* proved the opaqueness of the predicate by considering only two possible values for each variable

Abstract Attack Results



SPECInt2000 benchmarks obfuscated with:
$\forall x \in \mathbb{Z} : 2 \ mod \ (x^2 + x)$ and $\forall x \in \mathbb{Z} : 2 \ mod \ (x + x)$

The abstract approach detects 66176 opaque predicates in about 8 sec
(Brute force almost 9 seconds for detecting one opaque predicate)

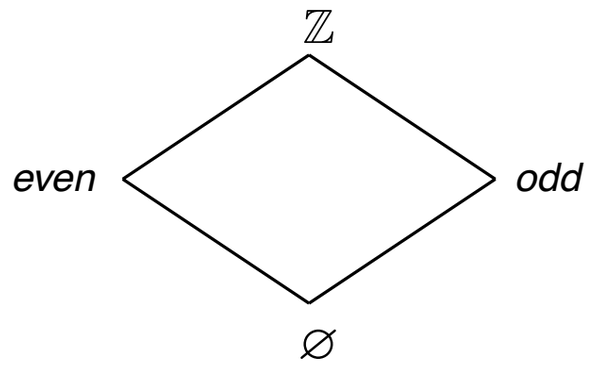
Abstract Attack

We consider a specific class of opaque predicates $\forall x \in \mathbb{Z} : n \ \textit{mod}\ f(x)$

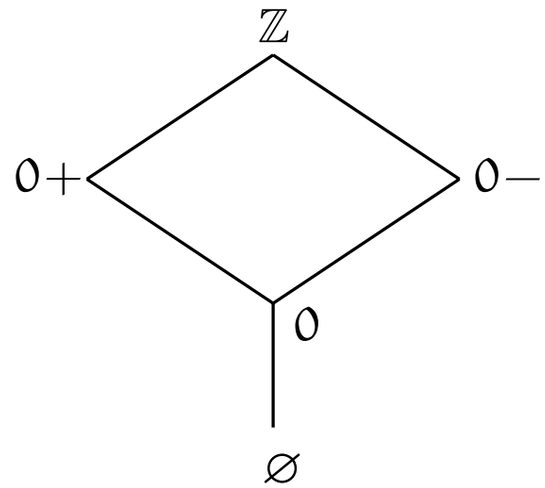
$$\forall x \in \mathbb{Z} : 3 \ \textit{mod}\ (x^3 - x)$$
$$\forall x \in \mathbb{Z}^+ : 24 \ \textit{mod}\ (2,7^x + 3,4^x - 5)$$

We consider attackers as modeled as abstract domains

Parity



Sign



Given an opaque predicate design an abstract domain able to break it

Breaking Opaque Predicates

- $\forall x \in \mathbb{Z} : n \ \textit{mod}\ f(x)$

  Concrete Test: $\forall x \in \mathbb{Z} : f(x) \in n\mathbb{Z}$

- Abstract domains that express being a multiple of n

  $A \in \textit{Abs}(\wp(\mathbb{Z}))$ such that: $\exists a_n \in A : \gamma_A(a_n) = n\mathbb{Z}$

  Abstract Test: $\forall x \in \mathbb{Z} : f^\sharp(\alpha_A(\{x\})) \leq_A a_n$

- $f(\vec{x}) = h(g_1(\vec{x}) \ldots g_k(\vec{x}))$

- $f^\sharp(a)$: composition of the BCA of $h, g_1, \ldots, g_k$

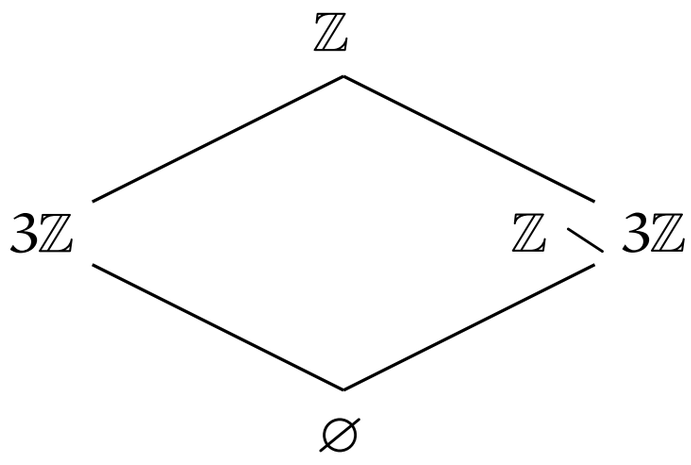
Breaking Opaque Predicates

- Abstract test is sound if: Abstract test $\Rightarrow$ Concrete test
- Abstract test is complete if: Abstract test $\Leftrightarrow$ Concrete test

When the abstract test is complete A breaks $\forall x \in \mathbb{Z} : n \ \mathit{mod}\ f(x)$

- Example $\forall x \in \mathbb{Z} : 3 \ \mathit{mod}\ (x^3 - x)$

A_3



A_3 is not complete

$f^\sharp(\mathbb{Z} \setminus 3\mathbb{Z}) = \mathbb{Z}$

$(\mathbb{Z} \setminus 3\mathbb{Z})^3 - \mathbb{Z} \setminus 3\mathbb{Z} = \mathbb{Z}$

Breaking Opaque Predicates

$f^\sharp$ is a sound approximation of f	$\Rightarrow$	Abstract test sound
$f^\sharp$ is a complete approximation of f	$\Rightarrow$	Abstract test complete

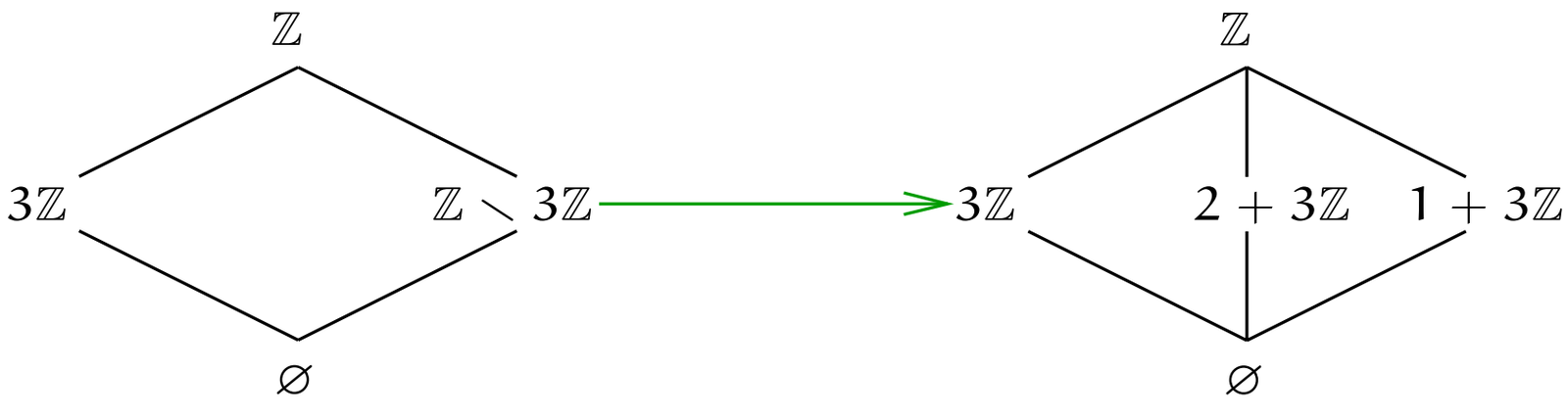
- $f^\sharp$ is sound by definition
- Completeness is preserved by composition

if $\mathcal{A}$ is complete for the elementary functions $h, g_1, ..., g_k$ composing f
then $\mathcal{A}$ breaks the opaque predicate $\forall x \in \mathbb{Z} : n \ \textit{mod}\ f(x)$

Completeness domain refinement

Example

$$\forall x \in \mathbb{Z} : 3 \; mod \; x^3 - x$$



A_{3+}

$f^\sharp(3\mathbb{Z}) = 3\mathbb{Z}$

$f^\sharp(\mathbb{Z} \setminus 3\mathbb{Z}) = \mathbb{Z}$

Not complete for addition

$A_{3+}(4-1) = A_{3+}(3) = 3\mathbb{Z}$

$A_{3+}(4) - A_{3+}(1) = \mathbb{Z} \setminus 3\mathbb{Z} - \mathbb{Z} \setminus 3\mathbb{Z} = \mathbb{Z}$

3-arity

$f^\sharp(3\mathbb{Z}) = 3\mathbb{Z}$

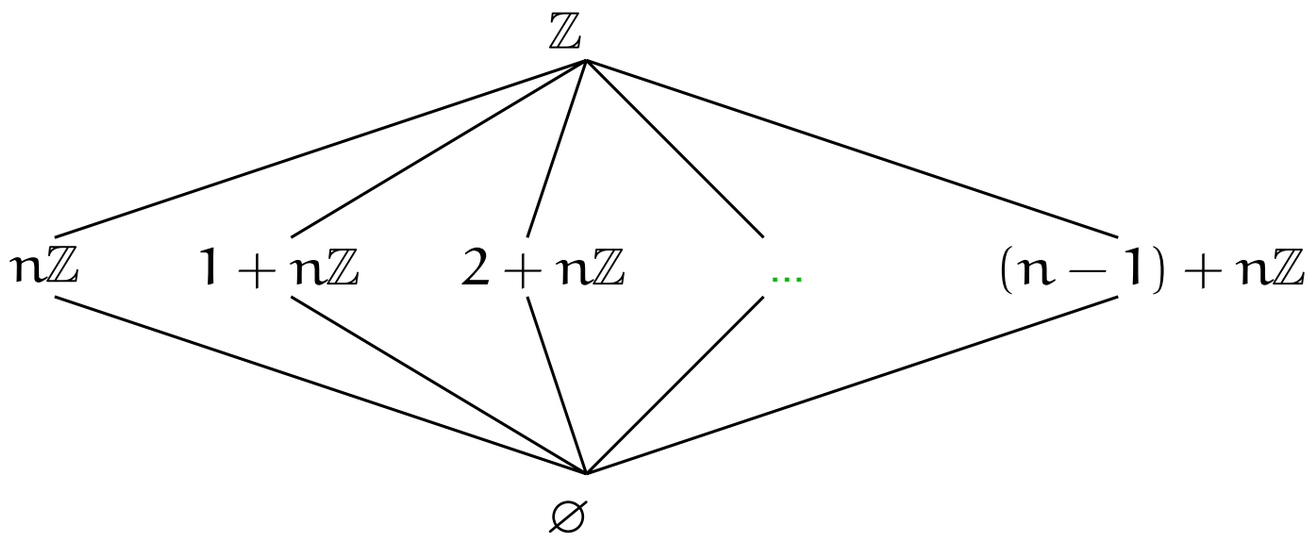
$f^\sharp(1 + 3\mathbb{Z}) = 3\mathbb{Z}$

$f^\sharp(2 + \mathbb{Z}) = 3\mathbb{Z}$

COMPLETE

Abstract Attack to Opaque Predicate

- Given an opaque predicate $\forall x \in \mathbb{Z} : n \ \textit{mod}\ f(x)$:
  - Refine $A_n = \left\{ \ \mathbb{Z}, n\mathbb{Z} \ \right\}$ w.r.t. elementary function composing f
- $n\text{-arity}$ breaks $\forall x \in \mathbb{Z} : n \ \textit{mod}\ f(x)$ where $f(x)$ is a polynomial function:



- Given an opaque predicate $\forall x \in \mathbb{Z} : f(x) \subseteq P$ with $P \in \wp(\mathbb{Z})$:
  - Refine $A_P = \left\{ \ \mathbb{Z}, P \ \right\}$ w.r.t. elementary function composing f

Code Protection

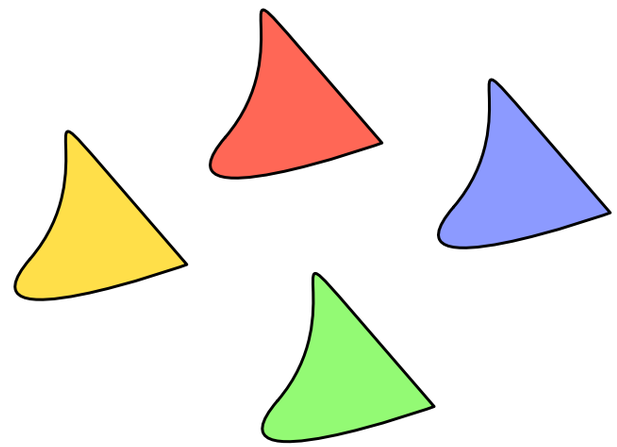
Code Obfuscation

Malware Detection

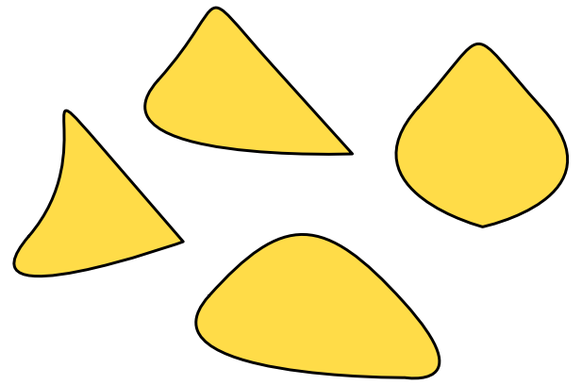
Metamorphism
Code Obfuscation

Malware Evasion Techniques

* Static evasion: Polymorphic malware contain decryption routines which decrypt encrypted constant parts of their body

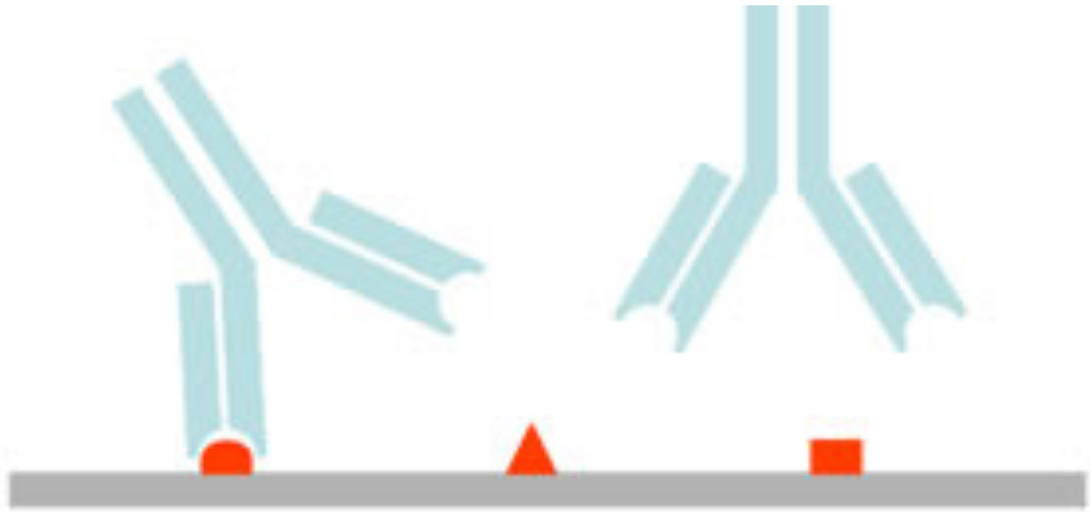


* Static/dynamic evasion: Metamorphic malware typically do not use encryption, but mutates (obfuscate) forms in subsequent generations



* Dynamic evasion: Anti-emulation, anti-instrumentation, dormant behaviors

Metamorphic Malware



Original code

```
E8 00000000      call 0h
5B               pop ebx
8D 4B 42         lea ecx, [ebx + 42h]
51               push ecx
50               push eax
50               push eax
0F01 4C 24 FE    sidt [esp - 02h]
5B               pop ebx
83 C3 1C         add ebx, 1Ch
FA               cli
8B 2B            mov ebp, [ebx]
```

Signature

```
E900 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

Obfuscated code

```
E8 00000000      call 0h
5B               pop ebx
8D 4B 42         lea ecx, [ebx + 45h]
90               nop
51               push ecx
50               push eax
50               push eax
90               nop
0F01 4C 24 FE    sidt [esp - 02h]
5B               pop ebx
83 C3 1C         add ebx, 1Ch
90               nop
FA               cli
8B 2B            mov ebp, [ebx]
```

New signature

```
E900 0000 005B 8D4B 4290 5150
5090 0F01 4C24 FE5B 83C3 1C90
FA8B 2B
```

[SYMANTEC 2013]

2011 variants per malware rate **5:1**

2012 variants per malware rate **38:1**

Behavioral Models

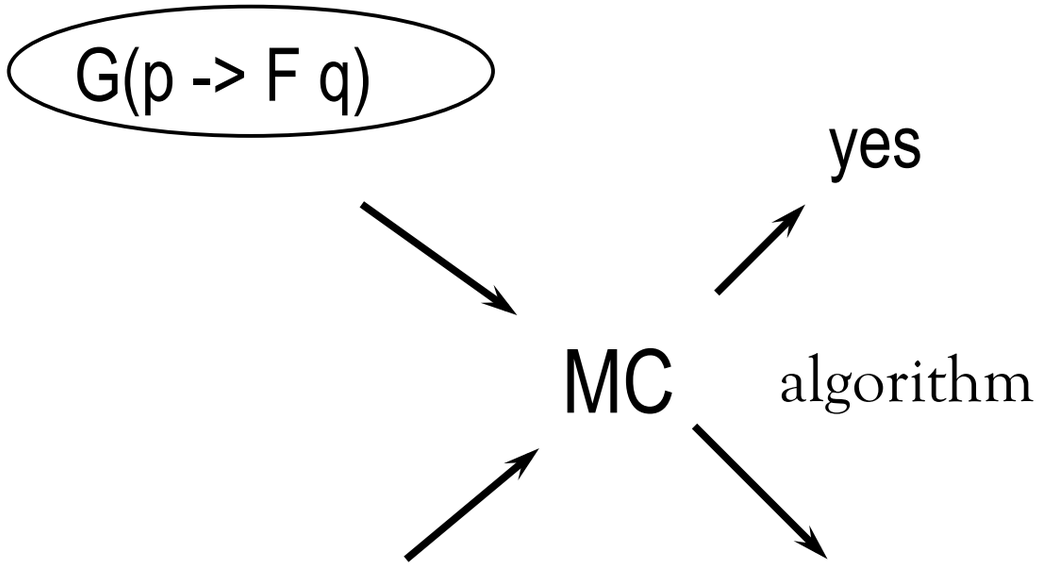
* Model checking and Malware Detection
* Automata and Malware Detection
* Normalization
* Mining/Learning
* …

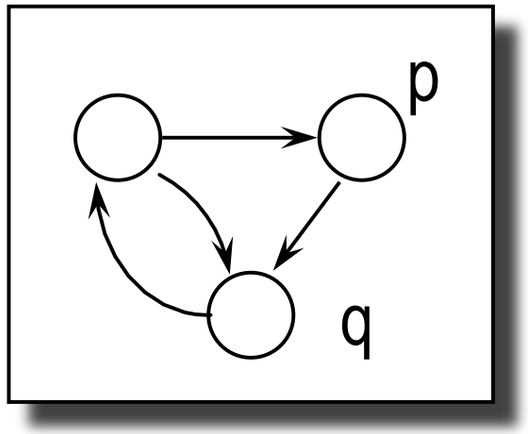
Model Checking and MD

* Abstract model of *malware*: *temporal logic formula* that specifies some temporal property of the malicious behaviour
* Abstract model of the possible infected *program*: *Kripke structure* derived from the CFG of the program
* Detection strategy: *model checking algorithm* that decides whether the Kripke structure modelling the program satisfies the logic formula modelling the malware
* Some works:
  * Singh and Lakhotia. “Static verification of worm and virus behavior in binary executables using model checking” IAW 2003
  * **Kinder et al. “Detecting malicious code by model checking” DIMVA 2005**
  * Kinder et al. “Using verification technology to specify and detect malware” EUROCAST 2007

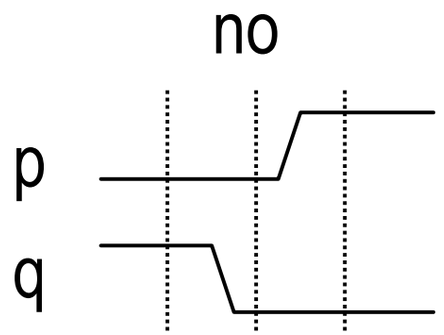
Kinder et al. “Detecting malicious code by model checking” DIMVA 2005

Model Checking





finite-state model



counterexample

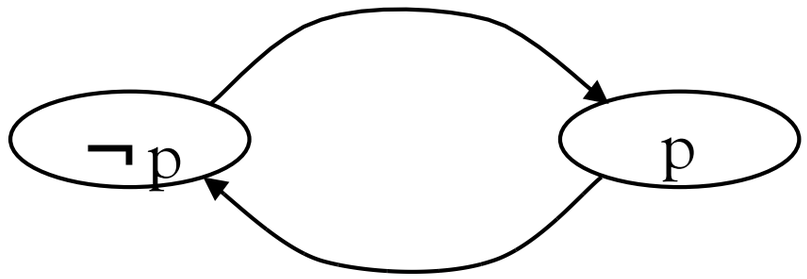
Model must now represent all behaviors

Kripke Models

- A Kripke model (S,R,L) consists of
  - set of states S
  - set of transitions R ⊆ S **x** S
  - labeling L ⊆ S **x** AP

- Kripke models from programs

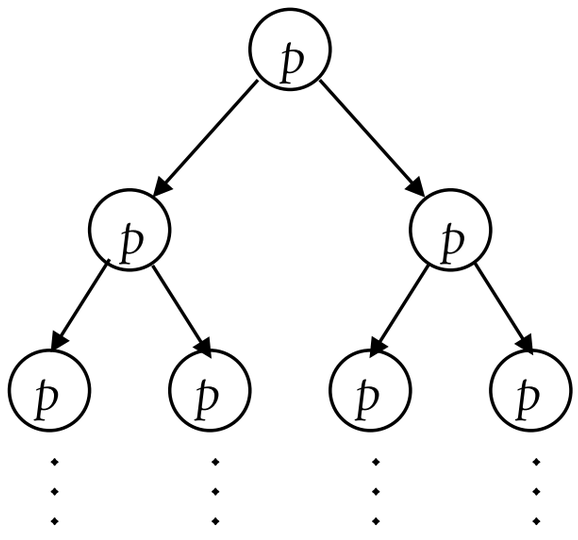
```
repeat
      p := true;
      p := false;
end
```



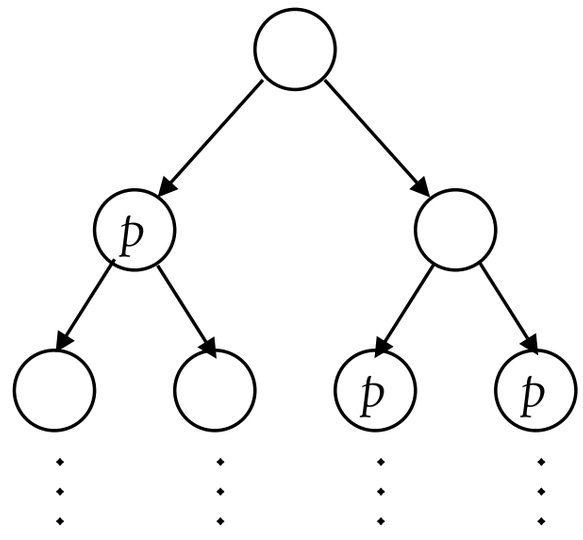
Computation Tree Logic

- Every operator F, G, X, U preceded by A or E
  - Universal modalities …

$AG\ p$



$AF\ p$



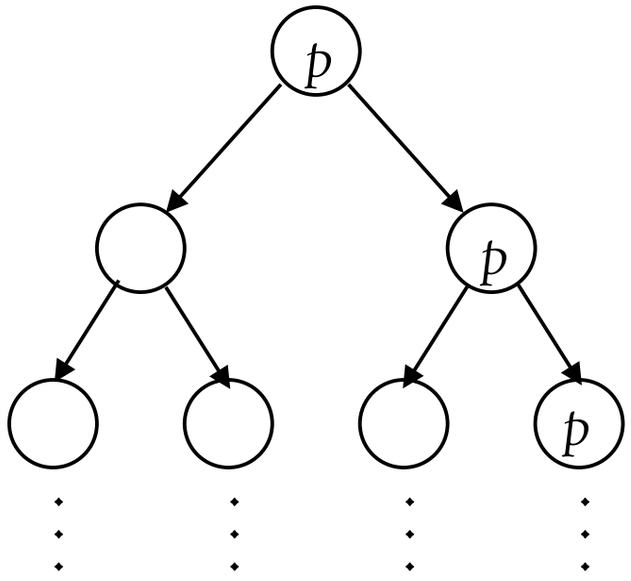
- Other modalities

AX p, EX p, A(p U q), E(p U q)

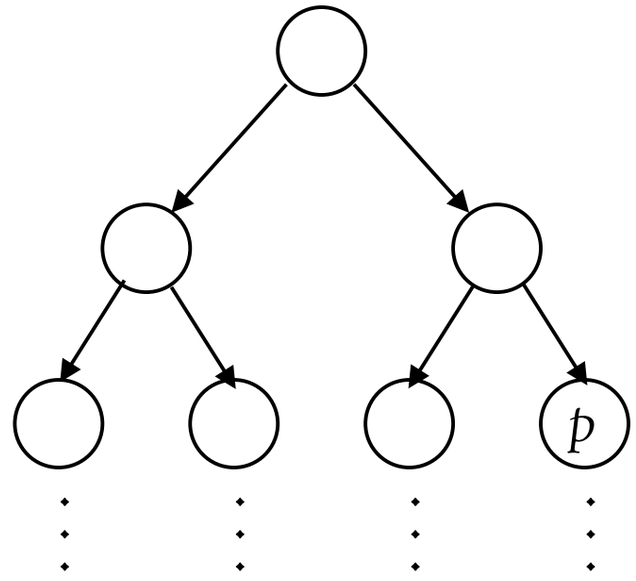
Computation Tree Logic

- Every operator F, G, X, U preceded by A or E
  - Existential modalities …

$EG\ p$



$EF\ p$



Model Checking and MD

- *Abstract model of the malware*: formula in CTPL, a temporal logic that extends CTL by taking into account register renaming in order to allow a succinct and natural representation of malicious code patterns
  - Atomic predicates in CTPL have the form `p(x1...xn)` where `xi` are free variables or constants and the predicate names represent assembly instructions in the natural way. For example, instruction `cmp ebx, [bp–1]` is represented as `cmp(ebx,[bp-1])`
  - CTPL has the existential and universal quantifiers that allow to quantify over the free variables in a predicate

Kinder et al. “Detecting malicious code by model checking” DIMVA 2005

Model Checking and MD

* Example: “In the code there exists a `mov` instructions that loads the constant `937` into some register; later the value contained in this register is always pushed onto the stack”
* CTL formula

```
EF(mov eax, 937 AND AF(push eax)) OR
EF(mov ebx, 937 AND AF(push ebx)) OR
EF(mov ecx, 937 AND AF(push ecx)) OR
…
```

* CTPL formula

```
Exits r EF(mov r, 937 AND AF(push r))
```

Kinder et al. “Detecting malicious code by model checking” DIMVA 2005

Model Checking and MD

* *Abstract model of the program*: Kripke structure where each node is labeled by a predicate denoting an assembly instruction and the corresponding location in memory.
* *Detection strategy*: verify whether a Kripke structure satisfies a CTPL formula. This is achieved by extending the standard model checking algorithm for CTL in order to keep track of *variable bindings*.
* The model checking algorithm is PSPACE-complete but efficient in real world settings
  * Exponential in the size of specification
  * Linear in size of the model

Kinder et al. "Detecting malicious code by model checking" DIMVA 2005

Model Checking and MD

```
c:   cmp ebx,[bp-4]
     jz j
     dec ebx
     jmp c
j:   mov eax,[bp+8]
```

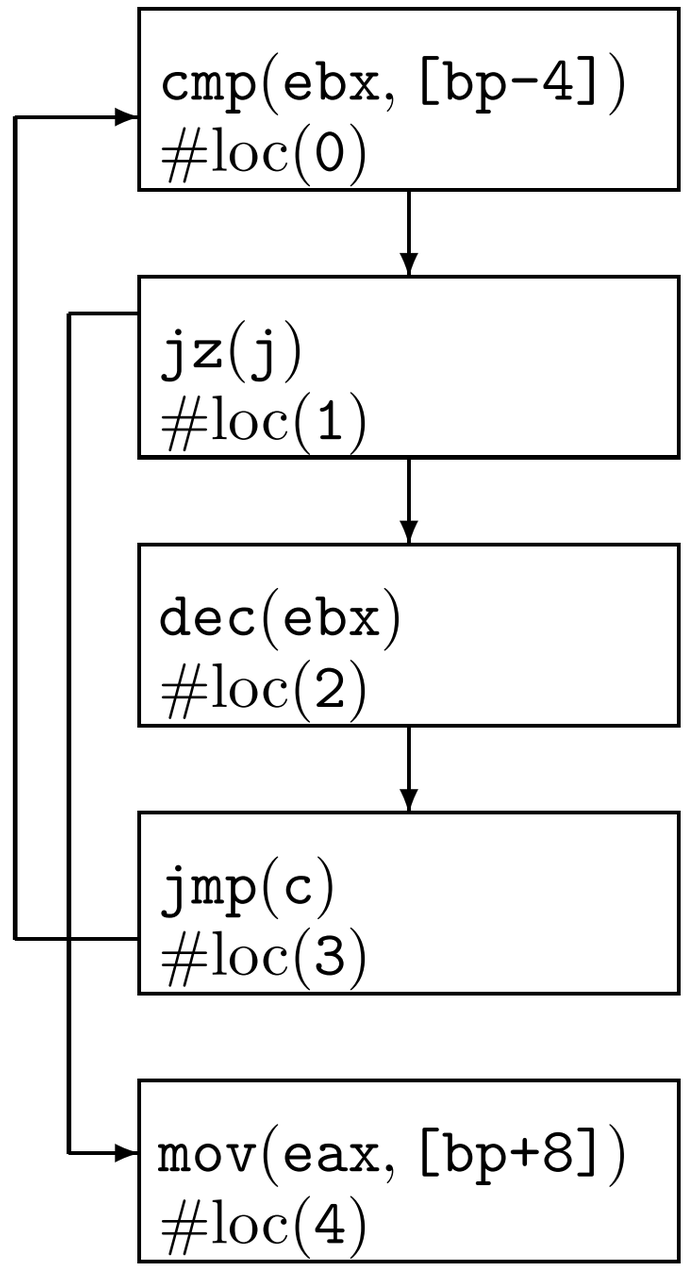


Fig. 4. Executable code sequence and corresponding Kripke structure.

Kinder et al. “Detecting malicious code by model checking” DIMVA 2005

Model Checking and MD

Example: CTPL formula that specifies a typical worm behaviour, namely that matches code creating copies of its own behaviour

$$
\begin{aligned}
&\exists L_m \exists L_c \exists v_{File} (\\
&\quad \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0 (\\
&\quad\quad \mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0,t) \vee \texttt{lea}(r_0,t)))\mathbf{U}\#\mathrm{loc}(L_0)) \wedge \\
&\quad\quad \mathbf{EF}(\texttt{mov}(r_1, 0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_1,t) \vee \texttt{lea}(r_1,t)))\mathbf{U}\#\mathrm{loc}(L_1)) \wedge \\
&\quad\quad \mathbf{EF}(\texttt{push}(c_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
&\quad\quad\quad \mathbf{U}(\texttt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
&\quad\quad\quad\quad \mathbf{U}(\texttt{push}(r_1) \wedge \#\mathrm{loc}(L_1) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
&\quad\quad\quad\quad\quad \mathbf{U}(\texttt{call}(\texttt{GetModuleFileNameA}) \wedge \#\mathrm{loc}(L_m))))) \\
&\quad\quad) \\
&\quad \wedge (\exists r_0 \exists L_0 (\\
&\quad\quad \mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0,t) \vee \texttt{lea}(r_0,t)))\mathbf{U}\#\mathrm{loc}(L_0)) \wedge \\
&\quad\quad \mathbf{EF}(\texttt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
&\quad\quad\quad \mathbf{U}(\texttt{call}(\texttt{CopyFileA}) \wedge \#\mathrm{loc}(L_c))) \\
&\quad\quad)) \\
&\quad \wedge \mathbf{EF}(\#\mathrm{loc}(L_m) \wedge \mathbf{EF}\#\mathrm{loc}(L_c)) \\
&)
\end{aligned}
$$

Kinder et al. “Detecting malicious code by model checking” DIMVA 2005

Model Checking and MD

- Advantages:
  - Succinct representation of malicious behavior that matches many malware variants
  - Formal technique for deciding infection and efficient in real scenarios
  - Low risk of false positives
- Limits:
  - Malicious specifications are *difficult* to write and might be very complex
  - The design of the malicious CTPL formula is based on the observation of common features of existing malware

Kinder et al. “Detecting malicious code by model checking” DIMVA 2005

Automata and MD

* Abstract model of the malware and of the possibly infected program: *automata on an alphabet of abstract instructions*, derived from the CFG of the program. Thus, programs are represented as regular languages.

* Detection strategy: Comparison between the languages recognized by the automata representing the malware and the automata representing the possibly infected program (containment, intersection not empty....)

* Some works:

  * **Christodorescu et al. "Static analysis of executables to detect malicious code patterns." USENIX 2003**

  * Christodorescu et al. "Semantics-aware malware detector" S&P 2005

  * Beaucamps et al. "Behavior abstraction in malware analysis RV 2010

Automata and MD

- MD system based on *language containment* and *unification*
- Abstract patterns: represent properties of sequences of instructions that are obtained through static analysis.
- Abstract model of malware: automata M on the alphabet of abstract patterns with symbolic names for memory locations (in order to handle dependences between variables without referring to their storage locations). The automata is derived by the CFG of the malware where sequences of instructions are approximated with abstract patterns

Static analysis of executables to detect malicious code patterns. USENIX'03

Automata and MD

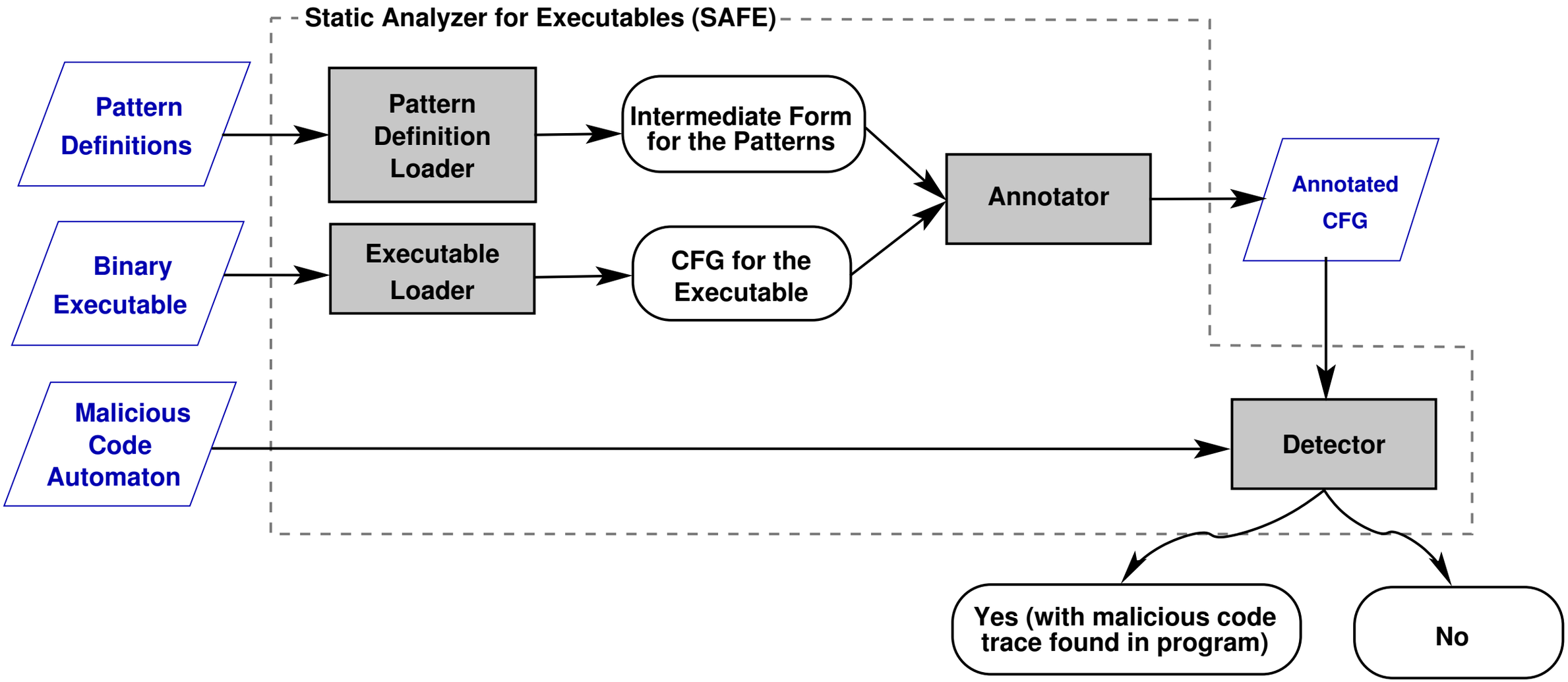


Figure 4: Architecture of the static analyzer for executables (SAFE).

Static analysis of executables to detect malicious code patterns. USENIX'03

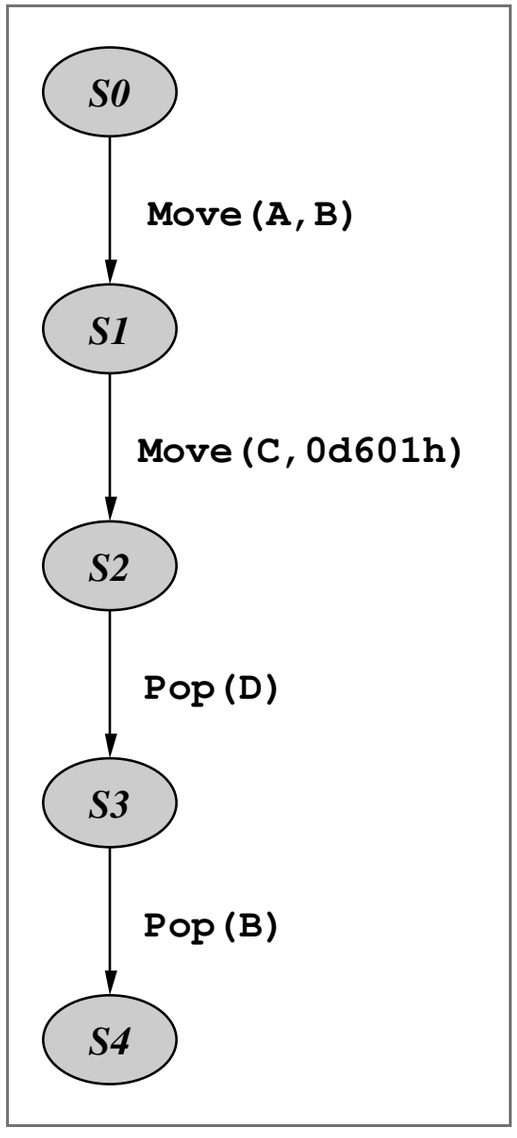
Automata and MD

Some of the abstract patterns used:

- `Dominators(B)`: the set of basic blocks that dominate the basic block B
- `PostDominators(B)`: the set of basic blocks that are dominated by the basic block B
- `Pred(B)`: the set of basic blocks that immediately precede B
- `Succ(B)`: the set of basic blocks that immediately follow B
- `First(B)`: the first instruction of the basic block B
- `Last(B)`: the last instruction of the basic block B
- `Kill(p,a)`: true if the instruction at programs point p kills variable a
- `Uses(p,a)`: true if the instruction at programs point p uses variable a
- `Alias(p,x,y)`: true if variable x is an alias of variable y at program point p
- `PointTo(p,x,a)`: true if variable x points to a location a at program point p

Automata and MD

Malicious code automaton for a code fragment of the Chernobyl virus, and instantiations with different register assignments, shown with their respective bindings.




```
mov     esi, ecx
mov     eax, 0d601h
pop     edx
pop     ecx
```

$\mathcal{B}_1 = \{$ [A, esi],
[B, ecx],
[C, eax],
[D, edx] $\}$

```
mov     esi, eax
mov     ebx, 0d601h
pop     ecx
pop     eax
```

$\mathcal{B}_2 = \{$ [A, esi],
[B, eax],
[C, ebx],
[D, ecx] $\}$

Static analysis of executables to detect malicious code patterns. USENIX'03

Automata and MD

* Abstract model of possibly infected programs: a program P is modelled as a set of automata A1…An on the alphabet of abstract patterns with instantiated memory location. Each automata Ai represent one of the procedures P1…Pn of program P

* Detection strategy: a program P presents a malicious behaviour A if the following language is not empty:

$$L(P_\Sigma) \cap \left(\bigcup_{\mathcal{B} \in \mathcal{B}_{All}} L(\mathcal{B}(\mathcal{A})) \right)$$

Where B denotes a possible binding of the symbolic names of A to specific values

Static analysis of executables to detect malicious code patterns. USENIX'03

Automata and MD

Advantages:

* Formal behavioural representation of malicious behaviour that matches many malware variants
* Formal and efficient technique for deciding infection

Limits:

* The efficiency of the proposed detection strategy strongly depends on the set of abstract patterns used to approximate program behaviours
* Researchers choose the abstract patterns based on their observation of the salient features of existing malware and on the metamorphic techniques that they use to foil detection (for example: symbolic names, register renaming)

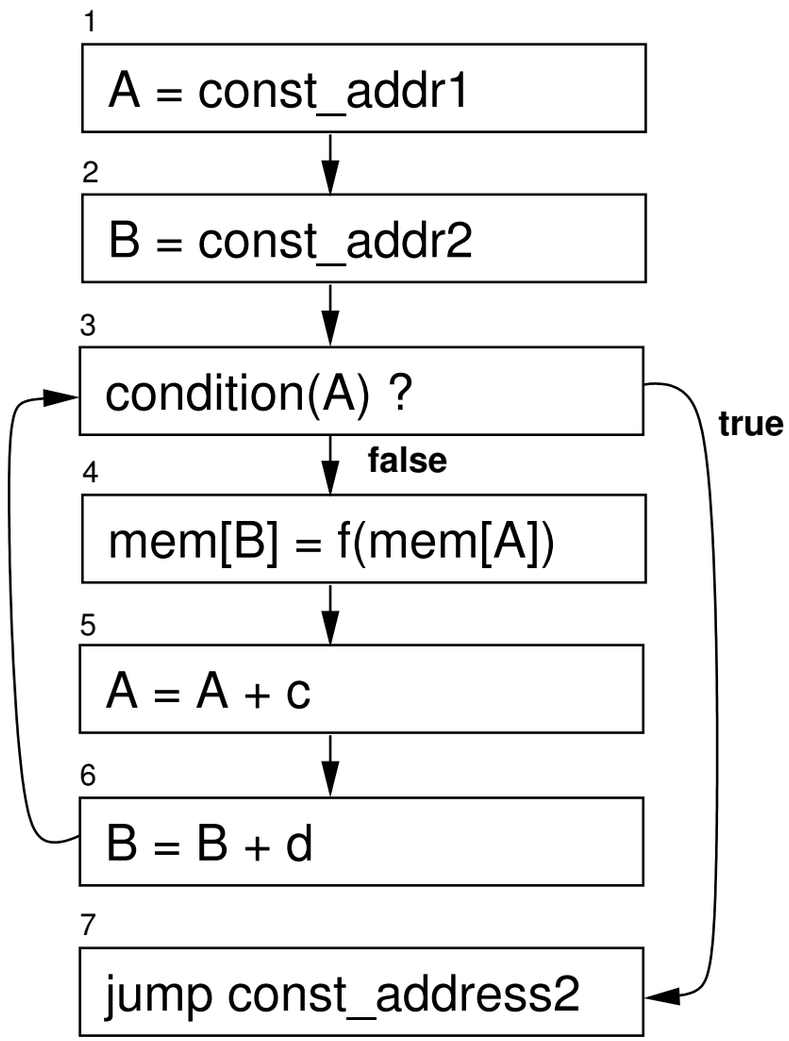
Static analysis of executables to detect malicious code patterns. USENIX'03

Semantics-Aware MD

- The basic deficiency in the pattern matching approach to malware detection is that they *ignore the semantics of instructions*
- Design a malware detection algorithm that uses semantics of instructions
- Such an algorithm will be resilient to minor obfuscations and variations

Semantics-aware Malware Detection S&P 2005

Semantics-Aware MD: Specifying the malicious behaviour

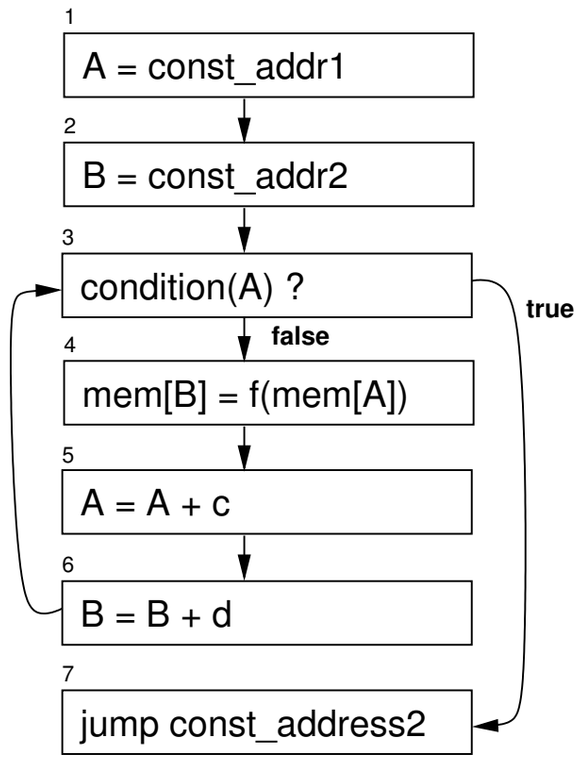


(a) Template of malicious behavior.

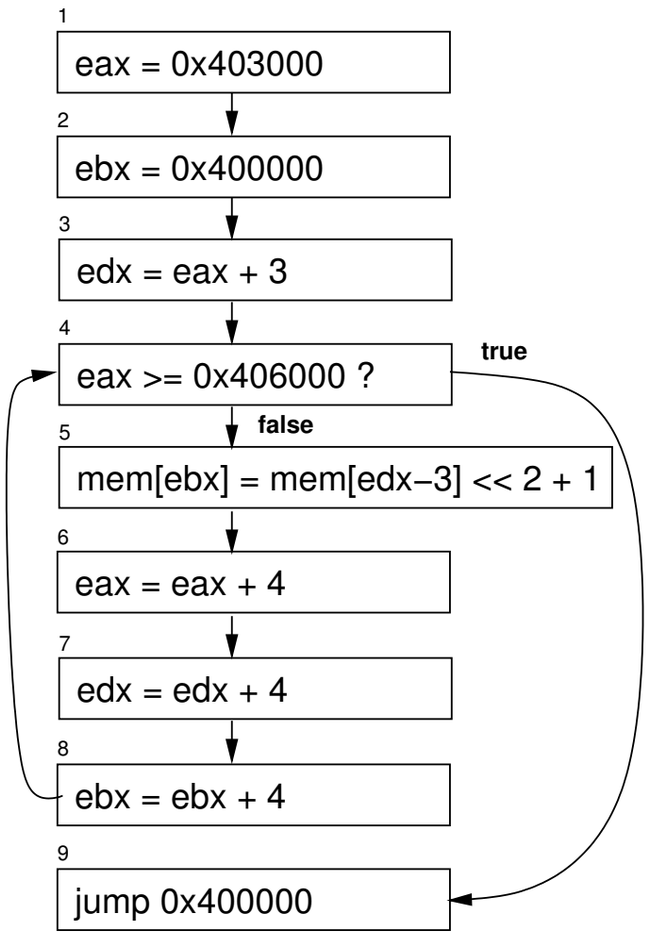
* Malicious behaviours are described using templates, which are instruction sequences where variables and symbolic constants are used

* On the left we see a template that describes a simplified version of a decryption loop found in polymorphic worms: it decrypts memory starting from `const_addr1` and writes the decrypted data at `const_addr2`

Semantics-Aware MD: Specifying the malicious behaviour



(a) Template of malicious behavior.



(b) Malware instance.

$$
\begin{aligned}
\text{const\_addr1} &\leftarrow 0\text{x}403000\\
\text{const\_addr2} &\leftarrow 0\text{x}400000\\
\text{condition(X)} &\leftarrow \text{X} \geq 0\text{x}406000\\
\text{f(X)} &\leftarrow \text{X} \ll 2 + 1\\
\text{c} &\leftarrow 4\\
\text{d} &\leftarrow 4
\end{aligned}
$$

(c) Execution context.

const_addr1 : F(0)
const_addr2 : F(0)
c : F(0)
d : F(0)
f : F(1)
condition : P(1)

(d) Symbolic constant types.

* If the template and the instruction sequence have the same effect on memory, namely if they are executed from a state where the contents of the memory are the same, then after both the executions the state of the memory is the same.

Semantics-aware Malware Detection S&P 2005

Semantics-Aware MD

* If the template and the instruction sequence shown in Figure 1(a) and 1(b) are executed from a state where the contents of the memory are the same, then after both the executions the state of the memory is the same.

* *The template and the instruction sequence have the same effect on the memory.*

* There is an execution of instruction sequence shown in Figure 1(b) that exhibits the behavior specified by the template given in Figure 1(a).

* *The malicious behavior specified by the template is demonstrated by the instruction sequence.*

* Note that this intuitive notion of an instruction sequence demonstrating a specified malicious behavior is not affected by program transformations, such as register renaming, inserting irrelevant instruction sequences, and changing starting addresses of memory blocks.

Semantics-Aware MD

Advantages:

* Resilient to commonly used obfuscations

Limits:

* Manual design of the malicious template

* The current implementation requires all of the IR instructions in the template to appear in the same form in the program (no equivalent instructions)

* The def-use relations in the malicious template effectively encode a specific ordering of memory updates – thus, the algorithm detects only those program that exhibit the same ordering of memory updates. (no equivalent functions)

Semantics-aware Malware Detection S&P 2005

Normalization and MD

- Fix a formal model for programs (CFG, trees,…)
- Define transformations of such models that lead to an unique normal form (ideally these transformations should undo the effects of obfuscations on the model)
- Detection strategy: verify whether the normal form of the model of the program matches the one of the malware

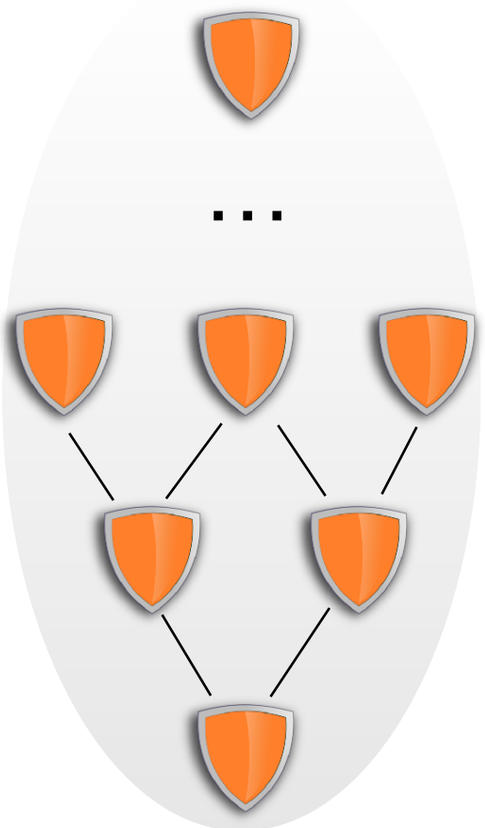
- Some works:
  - Lakhotia et al. “Imposing order on program statements to assist anti-virus scanners” WCRE 2004
  - **Walenstein et al. “Normalizing metamorphic malware using term rewriting” SCAM 2006**
  - Bruschi et al. “Detecting self-mutating malware using control flow graph matching” DIMVA 2006
  - Bruschi et al. “Using code normalization for fighting self-mutating malware” ISSSE 2006
  - **Bonfante et al. “Architecture of a morphological malware detector” Journal in Computer Virology 2008**

“Normalizing metamorphic malware using term rewriting” SCAM 2006

Semantic Malware Detector

To detect metamorphic malware variants we need a malware detector that observes

semantic properties preserved by obfuscation



Let M be a known malware

A **semantic malware detector AV** is defined wrt a semantic property *D in uco (Semantics)*

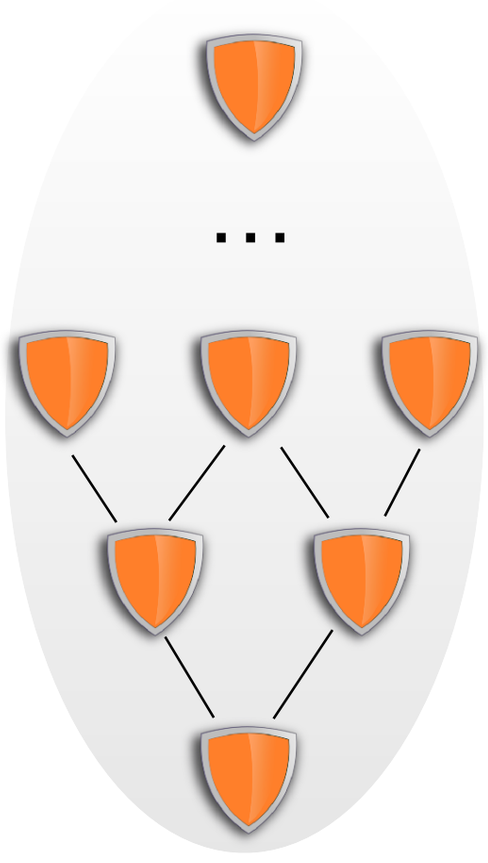
$$AV(D,M,P) = \textbf{alarm} \quad \text{iff} \quad D(S(P)) = D(S(M))$$

Dalla Preda at al. POPL 2007

THUS

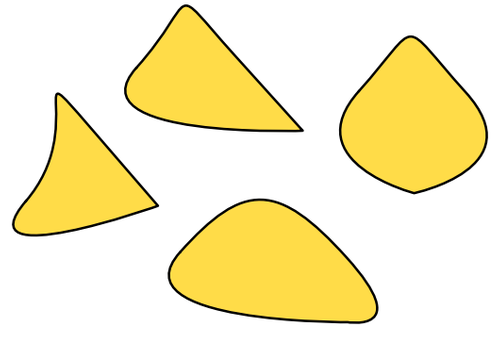
Semantic Malware Detector

- AV is *complete* for o if
  P infected with o(M) → AV($\mathcal{D}$,M,P) = alarm
  if $\mathcal{D}$ is preserved by o we have completeness



- AV is *sound* for o if
  AV($\mathcal{D}$,M,P) = alarm → P infected with o(M)

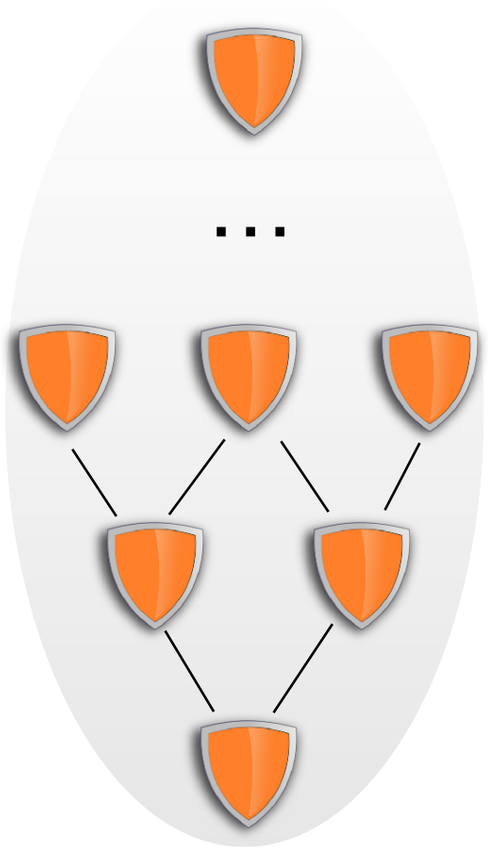
- Formally prove if a malware detector AV is sound (no false positives) or complete (no false negatives) wrt an obfuscating transformation τ, namely if AV can recognize metamorphic variants generated by τ



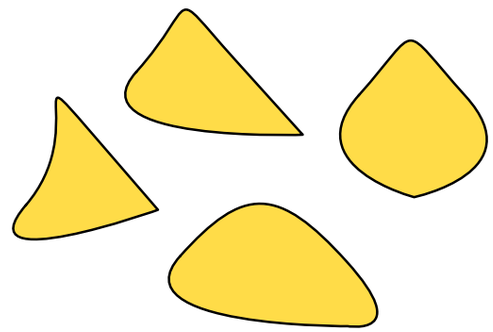
Dalla Preda at al. POPL 2007

Semantic Malware Detector

*Compare the efficiency of different AVs, by comparing the class of obfuscations that they are able to handle



Obfuscation	Signature-based	Semantics-aware 2005	Model Checking 2005
code reordering	sound	sound & complete	sound & complete
semantic nop insertion	sound	sound & complete	sound
equivalent instruticons	sound	sound	sound
variable renaming	sound	sound & complete	sound & complete



Dalla Preda at al. POPL 2007

The Challenge

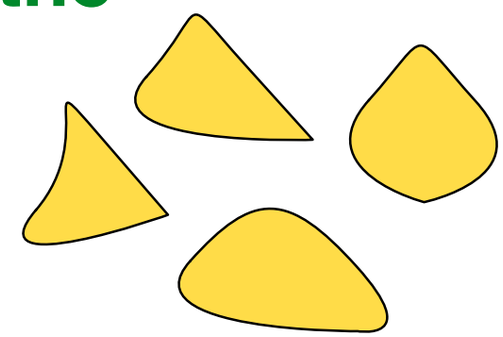
Limit: need of a priori knowledge of the metamorphic transformations used by malware

* The malware code contains the metamorphic engine (70%), it is a self-modifying program

* Metamorphic signature: is a language **L** of possible malware variants generated by the self-modifying malware

s in **L** then **s** is a metamorphic malware variant

Is there a way for systematically extracting a metamorphic signature without a priori knowledge of the transformations used?



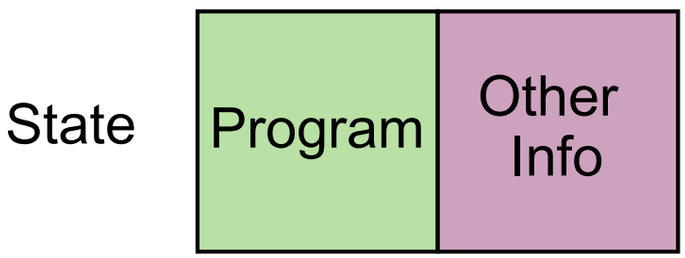
The Idea

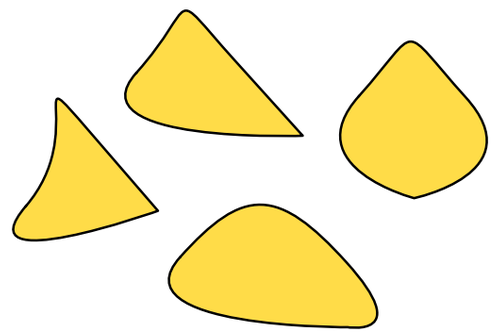
Extract L by abstract interpretation of the semantics of the self-modifying malware

* The description of code evolution is *inside* the trace semantics of the metamorphic engine

* Program states do not distinguish between *code* and *data*, it is possible to write in a location and then execute it

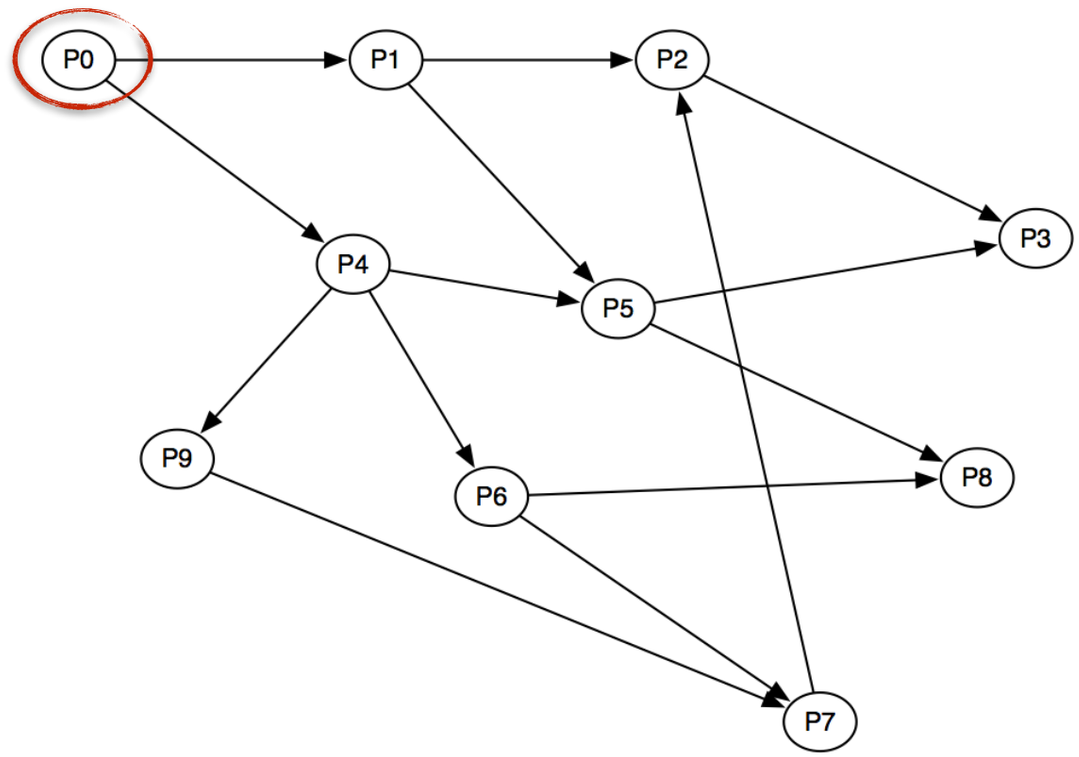
* The state of a program contains a description of the program being executed





Program Evolution Graph

Program Evolution Graph
Nodes: phases, code snapshots
Edges: possible code evolutions

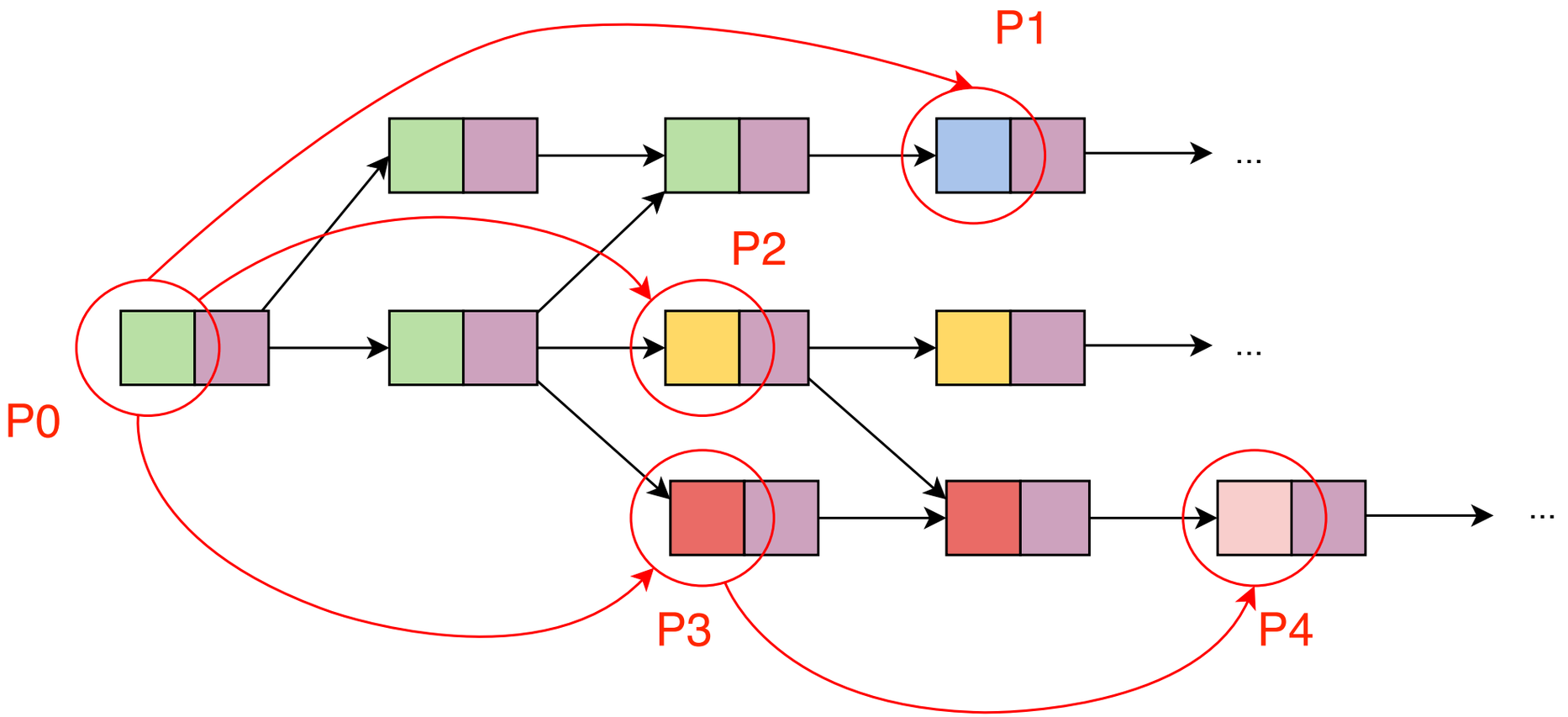


Precise description of the evolution of the code during execution

We would like to extract a (sound approximation) of the program evolution graph of a self-modifying code

Phase Semantics

A phase collects the computation that belongs to the same malware version



Phase semantics can be computed as a sound fix-point abstraction of standard trace semantics

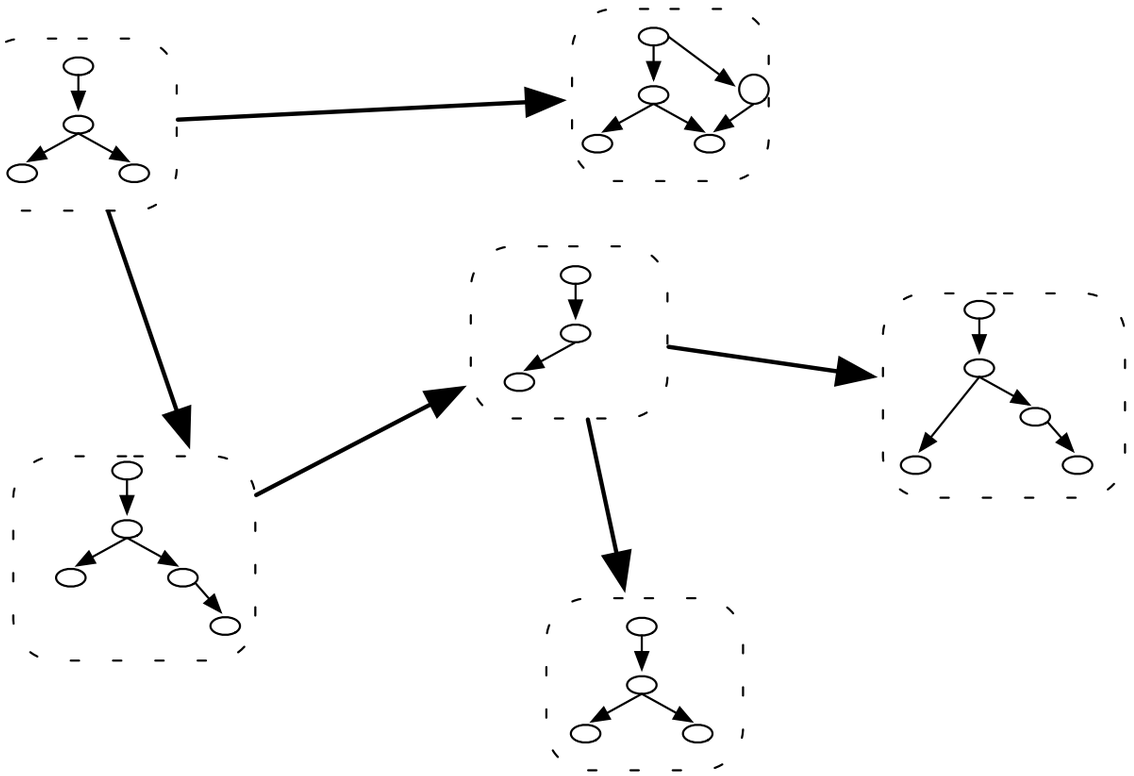
Phase Semantics

Phase semantics is precise but leads to an
undecidable test of metamorphism
(infinite traces)

P is a metamorphic variant of Q
iff
P is a phase in the phase semantics of Q

Need to design suitable abstract domains for the approximation of phase semantics

FSA Abstraction



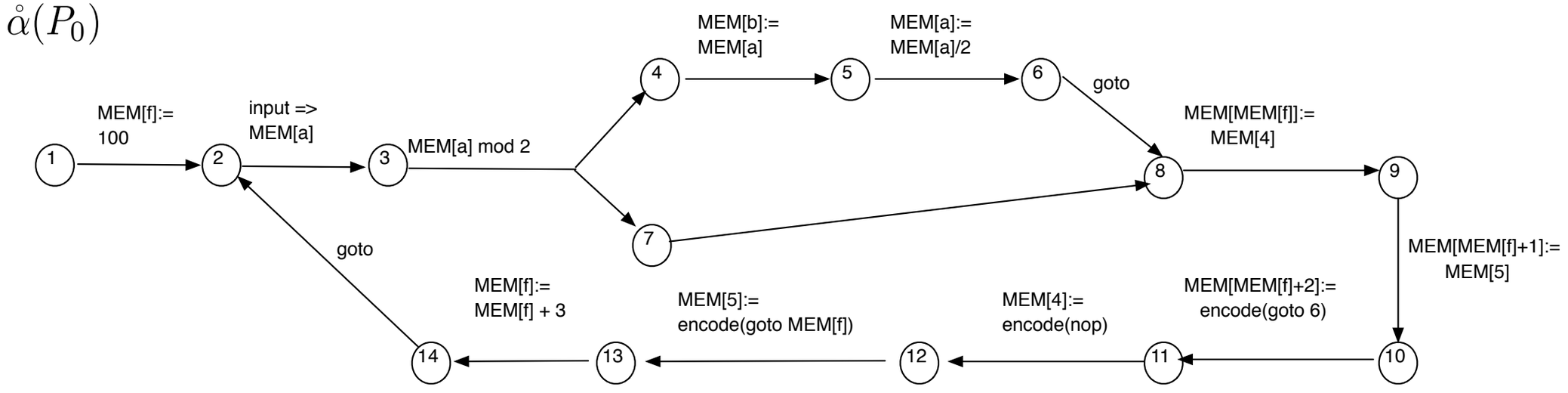
We represent programs as FSA, so the phase semantics becomes a graph of FSA (infinite traces)

FSA Abstraction

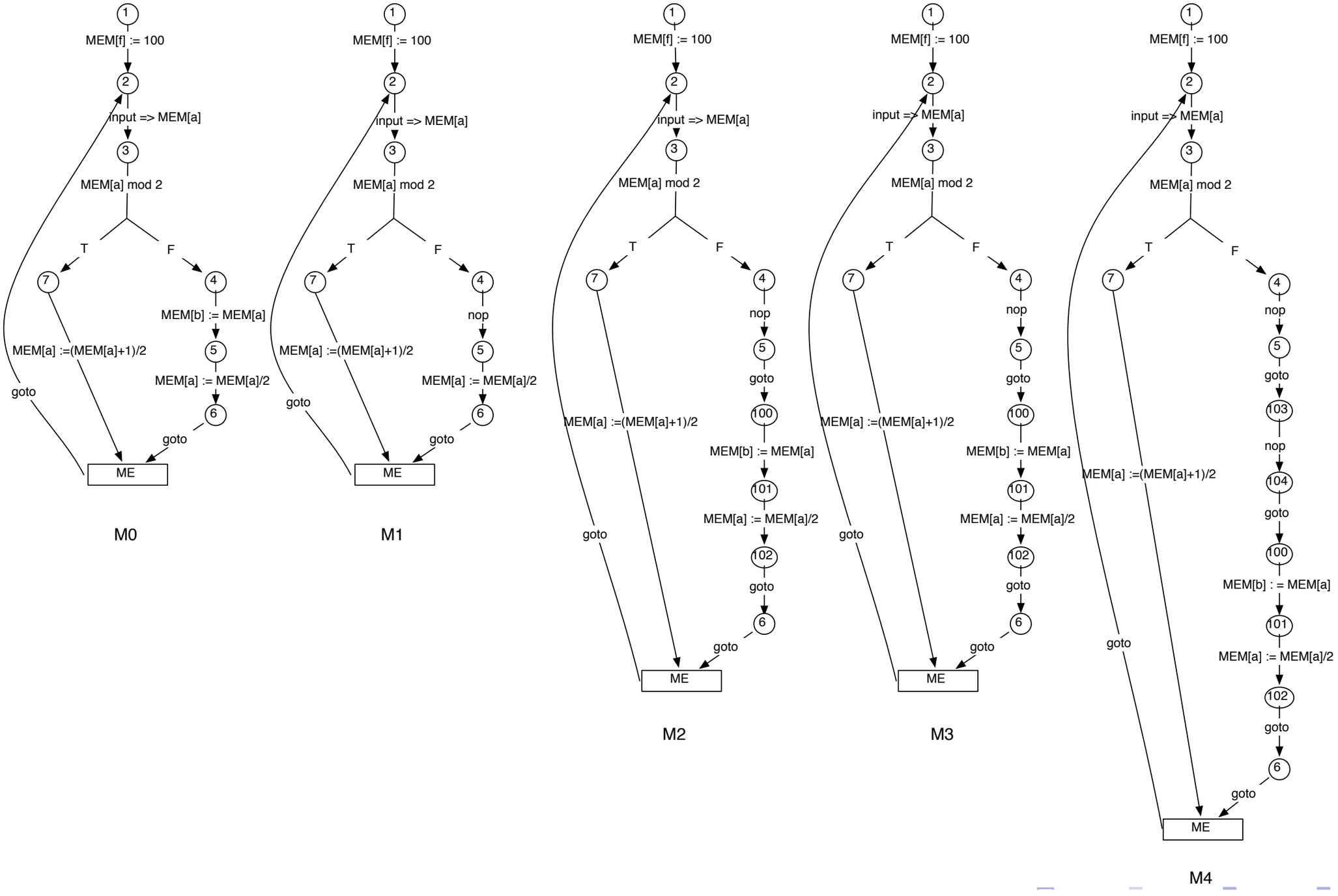
P_0

```
1:    mov f,100
2:    input ⇒ MEM[a]
3:    if (MEM[a] mod 2) goto 7
4:    mov b,MEM[a]
5:    mov a,MEM[a]/2
6:    goto 8
7:    mov a,(MEM[a]+1)/2
8:    mov MEM[f], MEM[4]
9:    mov MEM[f+1], MEM[5]
10:   mov MEM[f+2], encode(goto 6)
11:   mov 4, encode(nop)
12:   mov 5, encode(goto MEM[f])
13:   mov f, MEM[f]+3
14:   goto 2
```

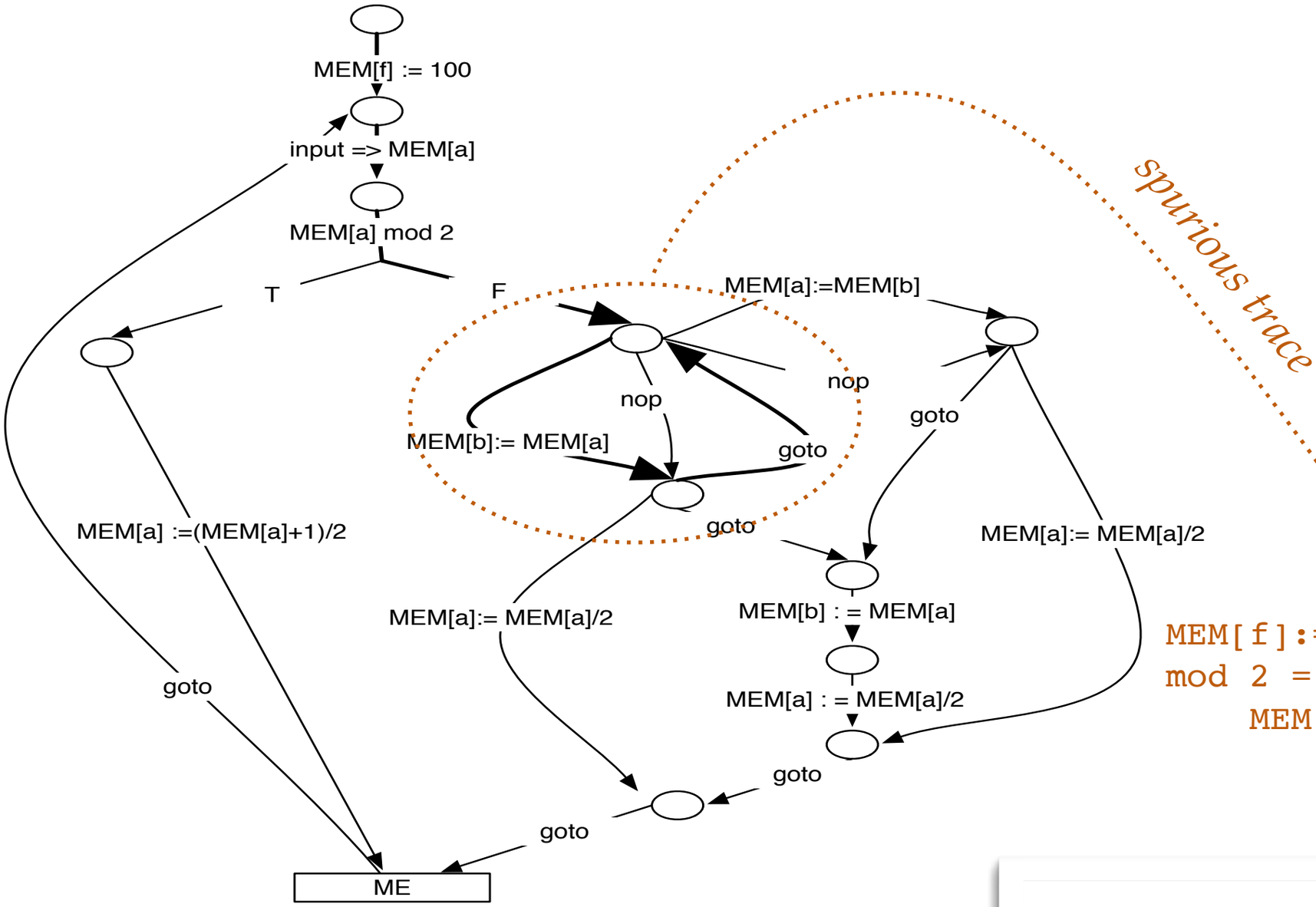
$\mathring{\alpha}(P_0)$



Traces of FSA



Merge FSA Phases




```
MEM[f]:=100;input=>MEM[a];MEM[a]
mod 2 = 0; MEM[b]:=MEM[a]; goto;
    MEM[b]:=MEM[a]; goto;...
```

decidable approximation

P is a metamorphic variant of Q

iff

$\mathbf{L}(P) \subseteq \mathbf{L}(Q)$

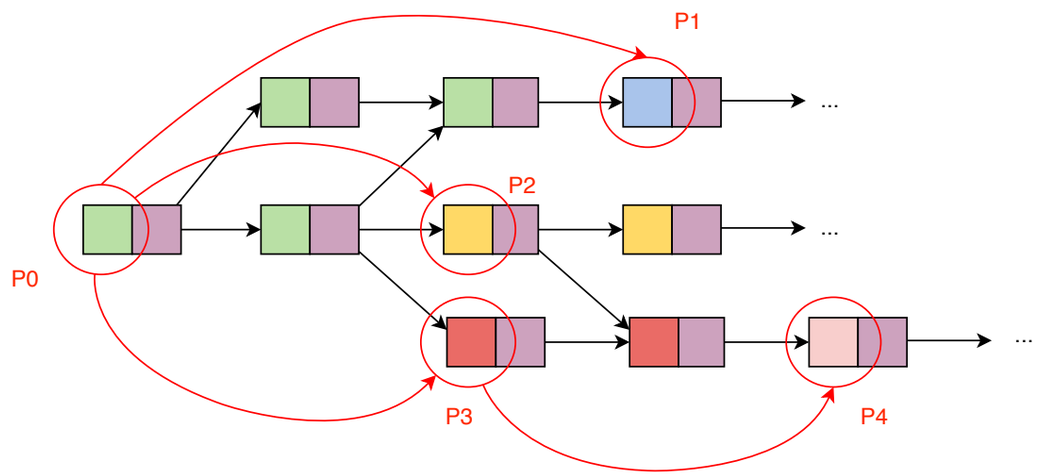
Phase Semantics

* Precise model of the evolution of the code of a self-modifying program with **no a priori knowledge** of the transformations used

* Sound abstractions of phase semantics lead to decidable and sound abstract metamorphism tests (we may have false positives)

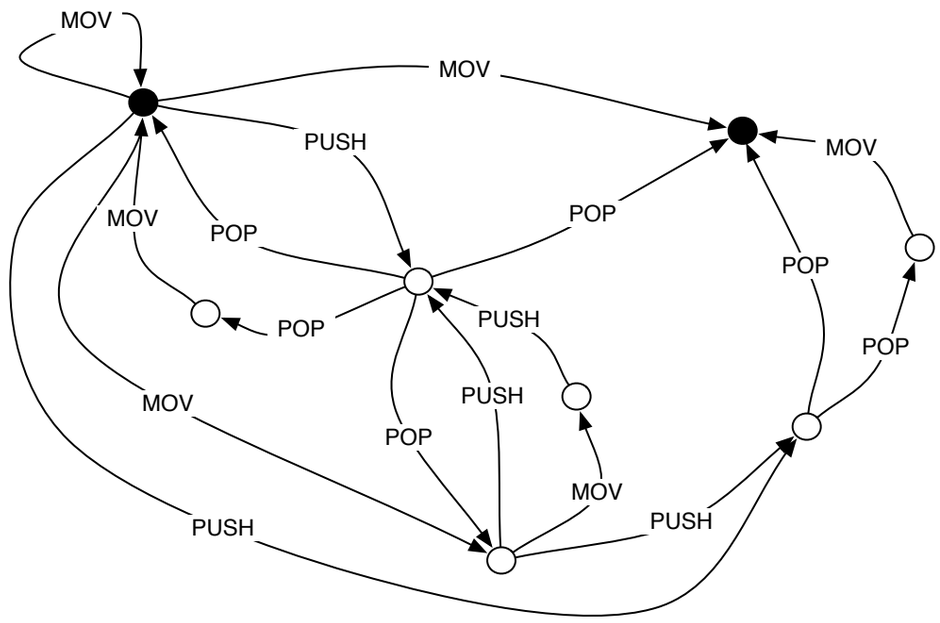
* Need to design an abstract domain for the abstraction of code features

the code is also data, namely the object of computation and abstraction



Learn the metamorphic engine?

$P =$ `mov e,10`



Approximated rules:

`push; pop` $\rightarrow$ `mov`

`mov; mov` $\rightarrow$ `mov`

Compression rules:

`push e2; pop e1` $\rightarrow$ `mov e1,e2`

`mov e2,e1; push e2` $\rightarrow$ `push e1`

`pop e2; mov e1,e2` $\rightarrow$ `pop e1`

```
mov
mov,mov
push,pop
mov,mov,mov
mov,push,pop
push,pop,mov
mov,mov,mov,mov
mov,mov,push,pop
mov,push,pop,mov
push,pop,push,pop
...
```

Code De-Obfuscation

* Generic approaches to code de-obfuscation

* Static analysis mostly ineffective

* “Hence, recent de-obfuscation proposals have shifted more towards **dynamic analysis**. Commonly, they produce an execution trace and use techniques such as (dynamic) taint analysis or symbolic execution to distinguish input dependent instructions. Based on their results, the program code can be reduced to only include relevant, input dependent instructions. This effectively strips the obfuscation layer. Even though such deobfuscation approaches sound promising, recent work proposes several ways to effectively thwart underlying techniques, such as symbolic execution” USENIX SECURITY 2017

Code De-Obfuscation

- COOGAN K, LU G., DEBRAY S. Deobfuscation of Virtualization-obfuscated Software: A Semantics-Based Approach. In ACM Conference on Computer and Communications Security (CCS) (2011).

- YADEGARI, B., AND DEBRAY, S. Symbolic Execution of Obfuscated Code. In ACM Conference on Computer and Communications Security (CCS) (2015)

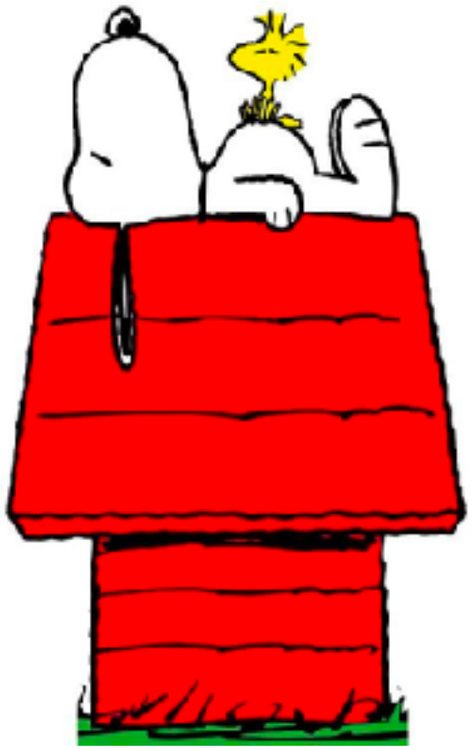
- B. Yadegari, B. Johannesmeyer, B. Whitely and S. Debray, A Generic Approach to Automatic Deobfuscation of Executable Code 2015 IEEE Symposium on Security and Privacy,

- Jhoannes Kinder. Towards Static Analysis of Virtualization-Obfuscated Binaries. WCRE 2012

- Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: synthesizing the semantics of obfuscated code. In Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)

Formal framework for dynamic analysis?

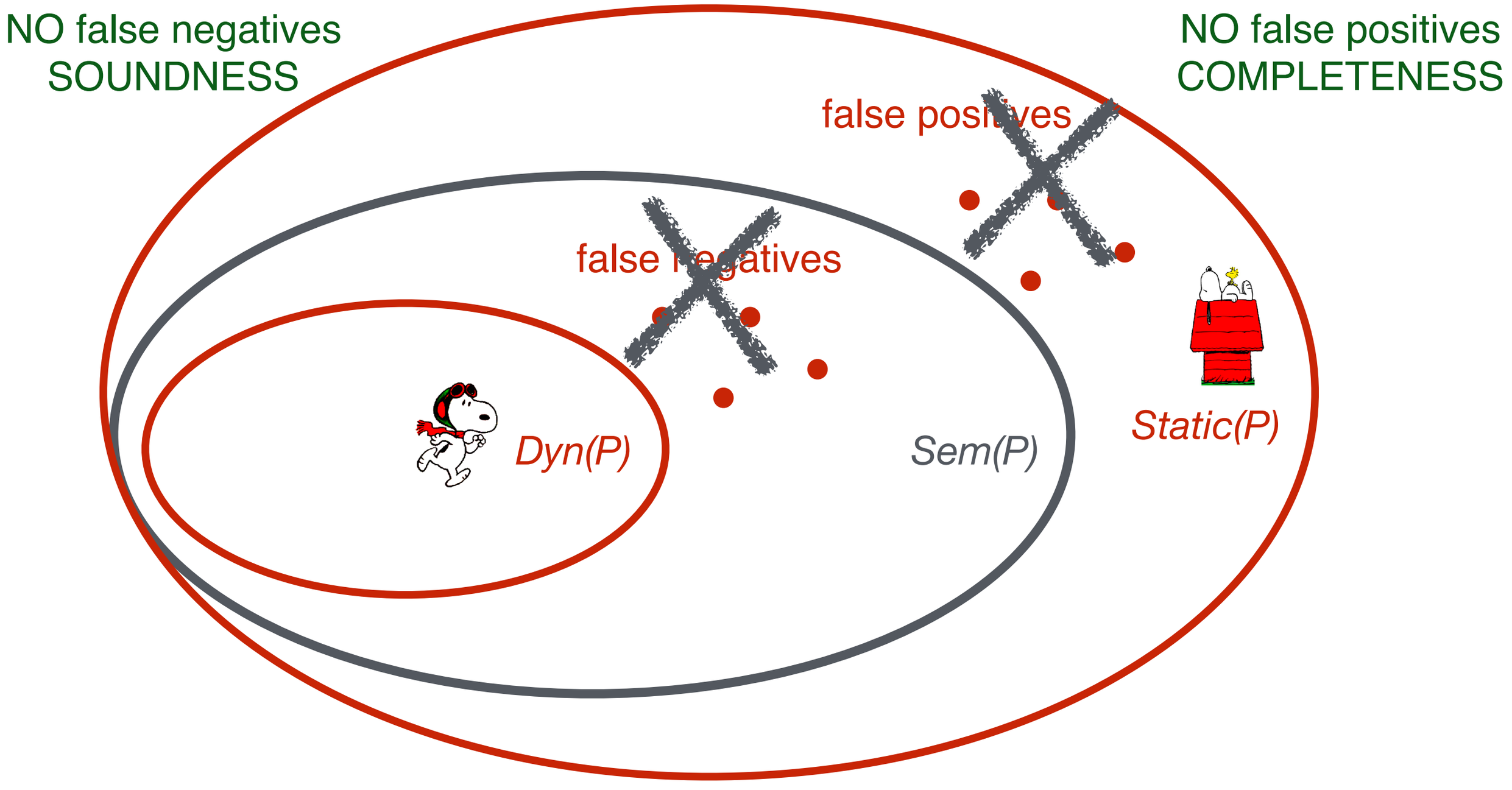
Program Analysis



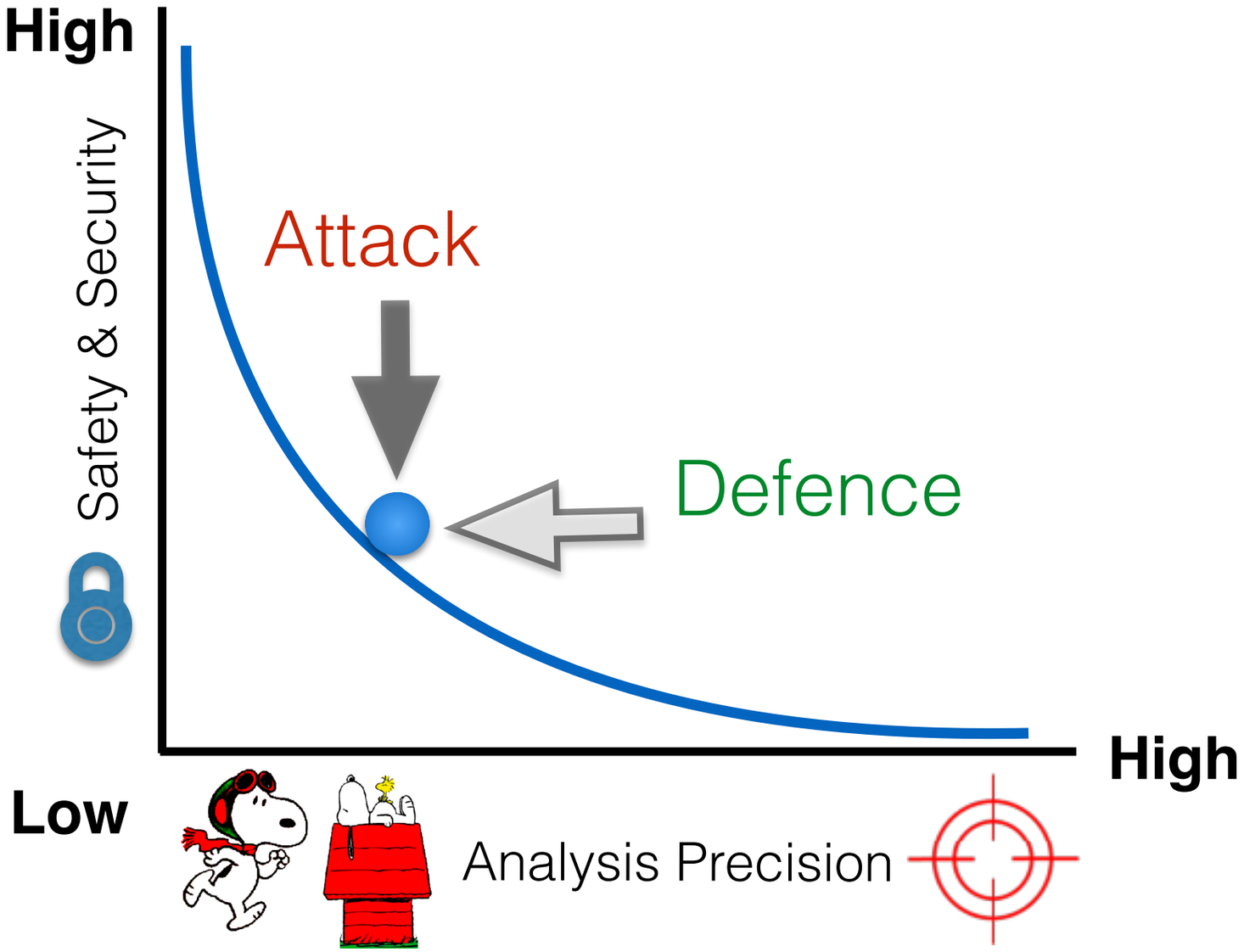
Static program analysis

Dynamic program analysis

Program Analysis



Software Protection

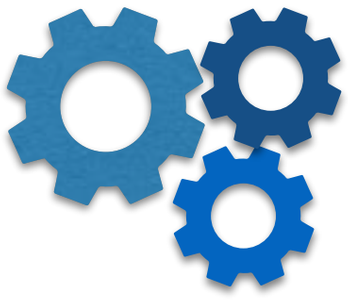


The attacker reverse engineer code

Code Obfuscation

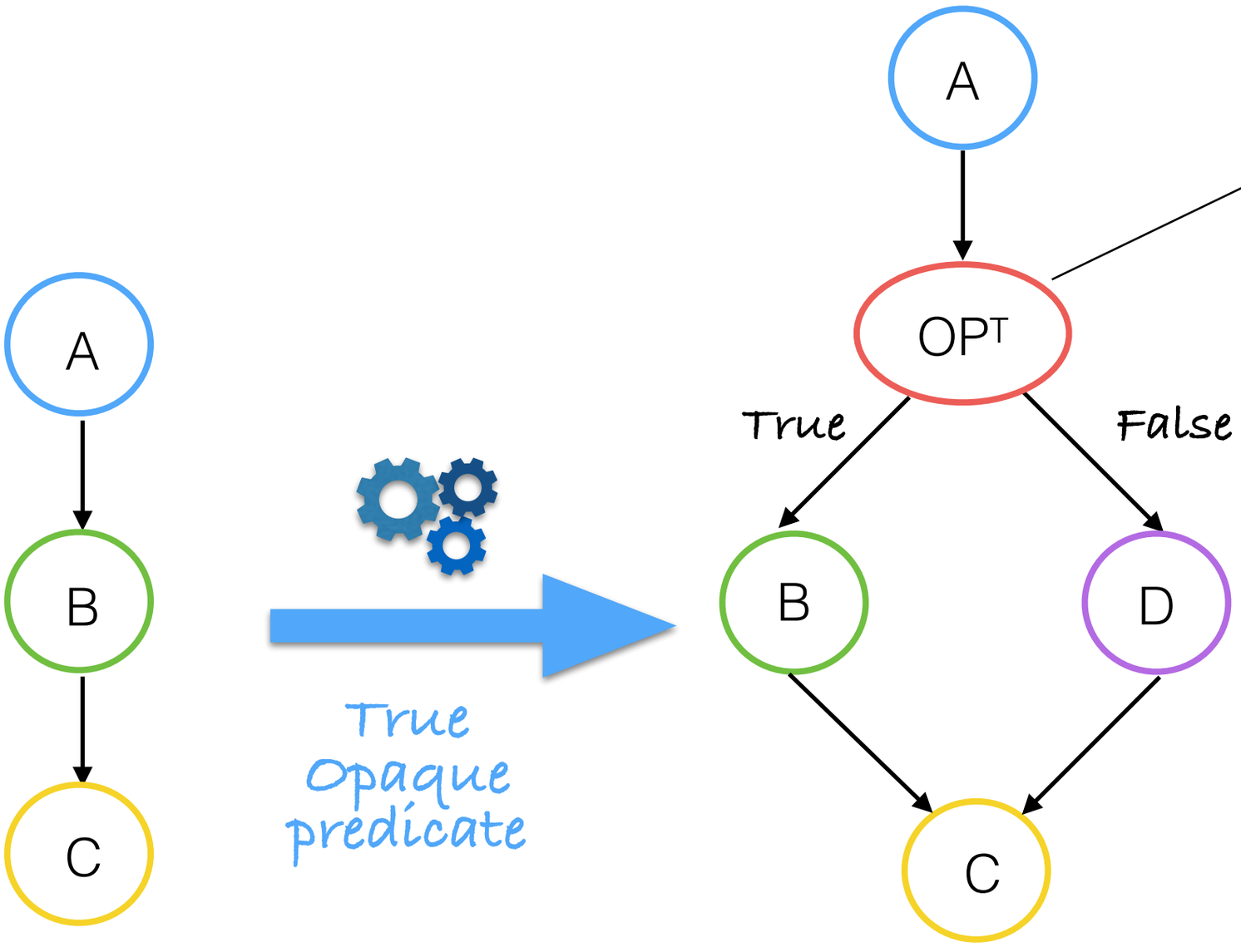
[Collberg et al. POPL 1998]
A program transformation O: Programs ⟶ Programs is a code obfuscation if:

- O **preserves the observational behavior** of programs
- O(P) is **more difficult to analyse**



Code obfuscation

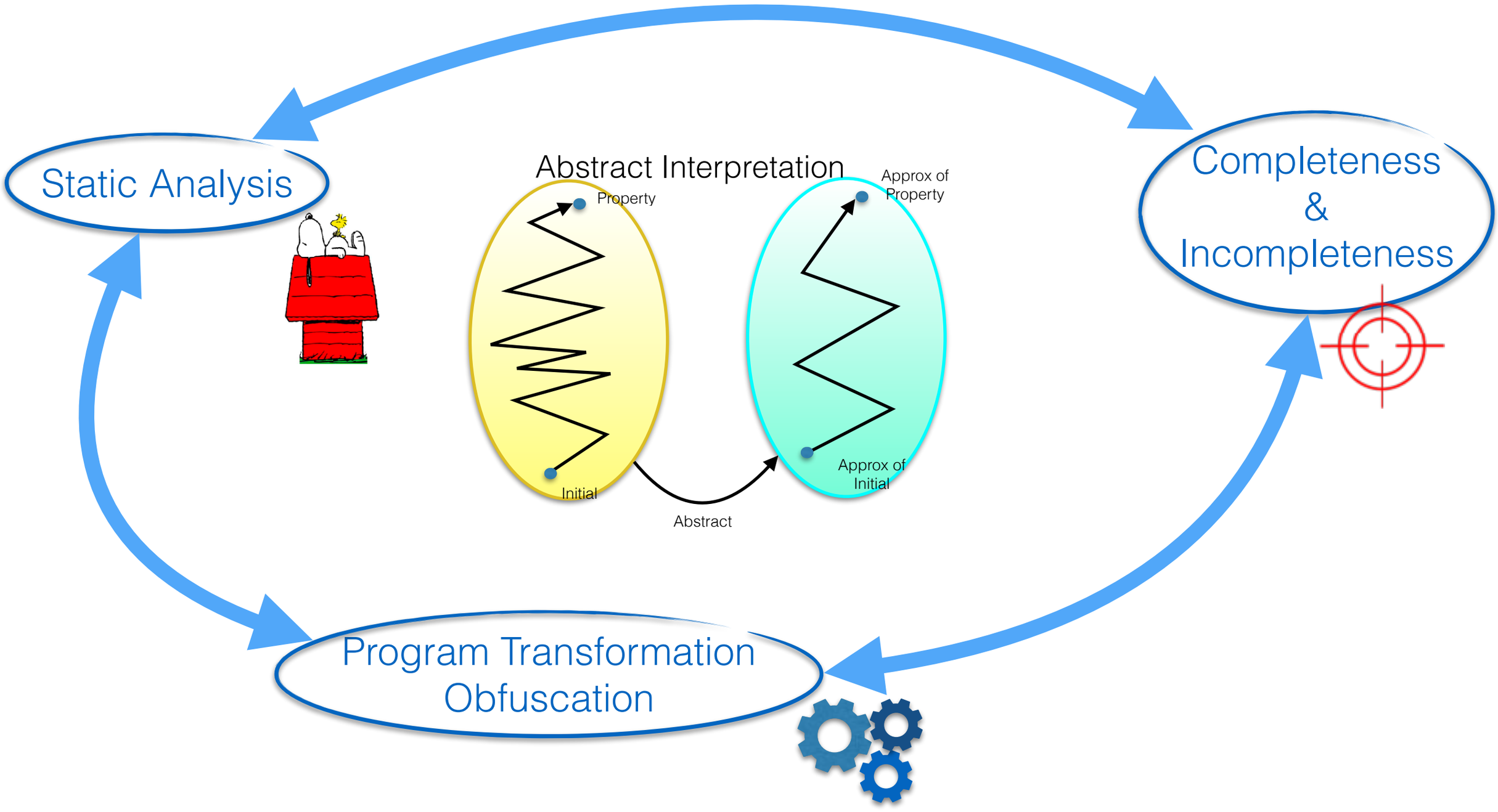
Code Obfuscation & Static Analysis



Add *false positive* for the control flow analysis

Incompleteness

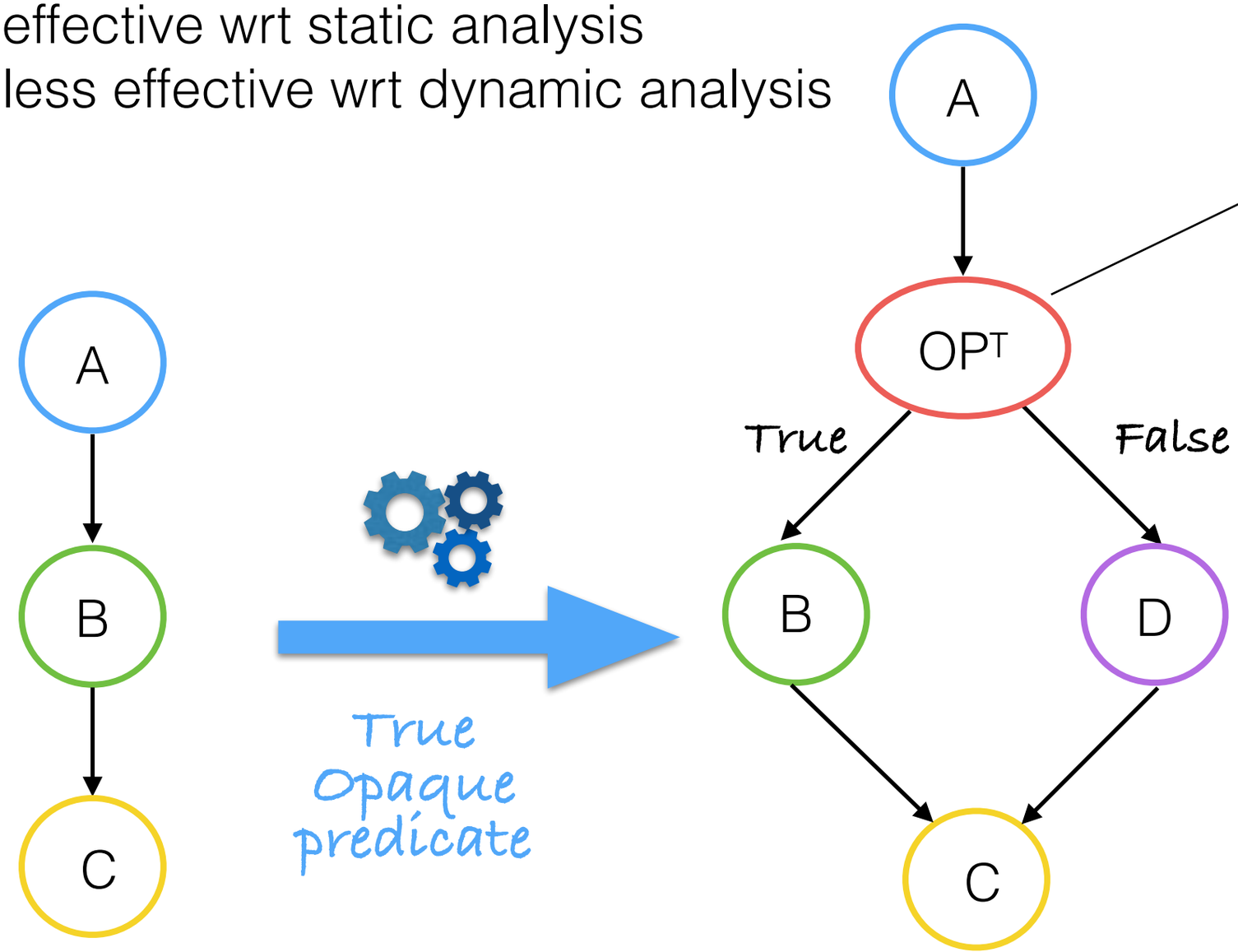
Code Obfuscation & Static Analysis



Code Obfuscation & Static Analysis

➡ effective wrt static analysis

➡ less effective wrt dynamic analysis



Add *false positive* for the control flow analysis

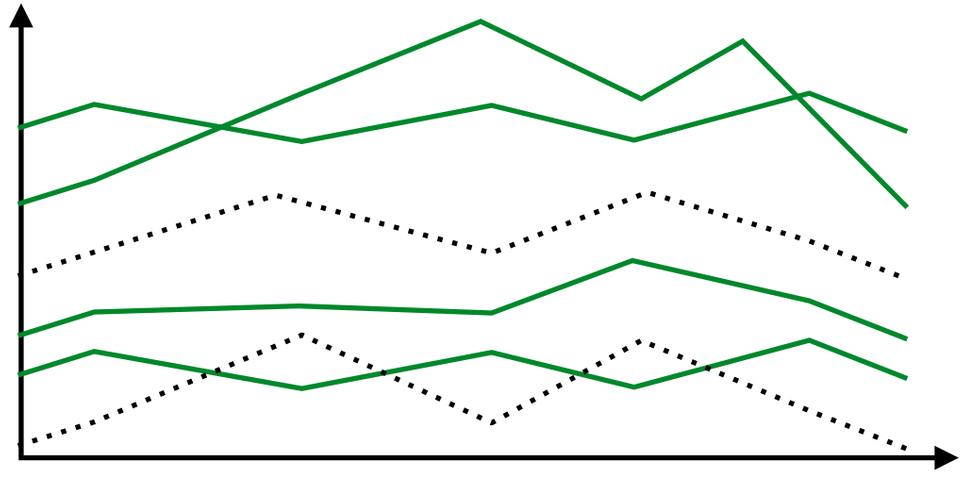
Incompleteness

Code Obfuscation & Dynamic Analysis

What does it mean to complicate/confuse dynamic analysis?

Dynamic Analysis

Analyze a finte subset of finte program traces to infer informations of the whole program, like in program testing and fuzzing

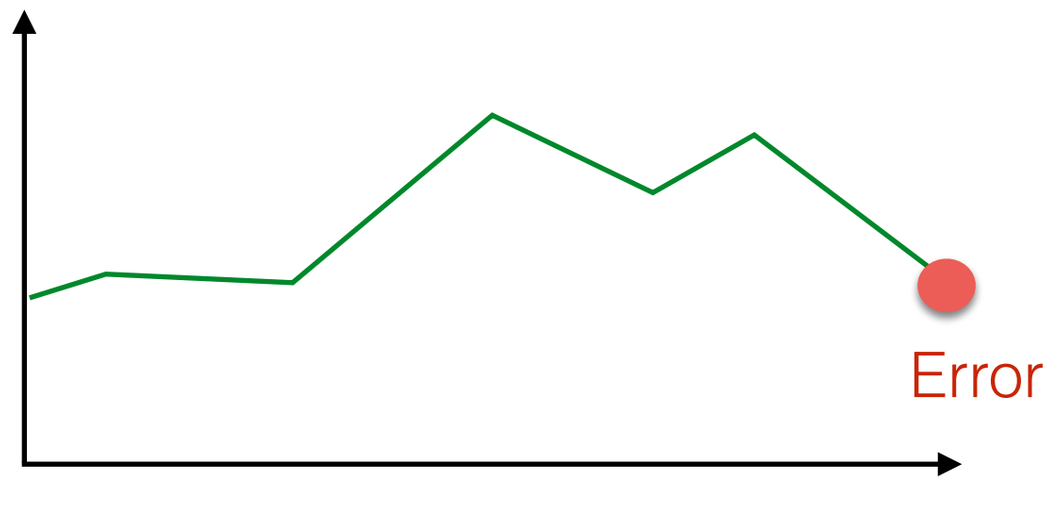


➡ Increase false negatives?

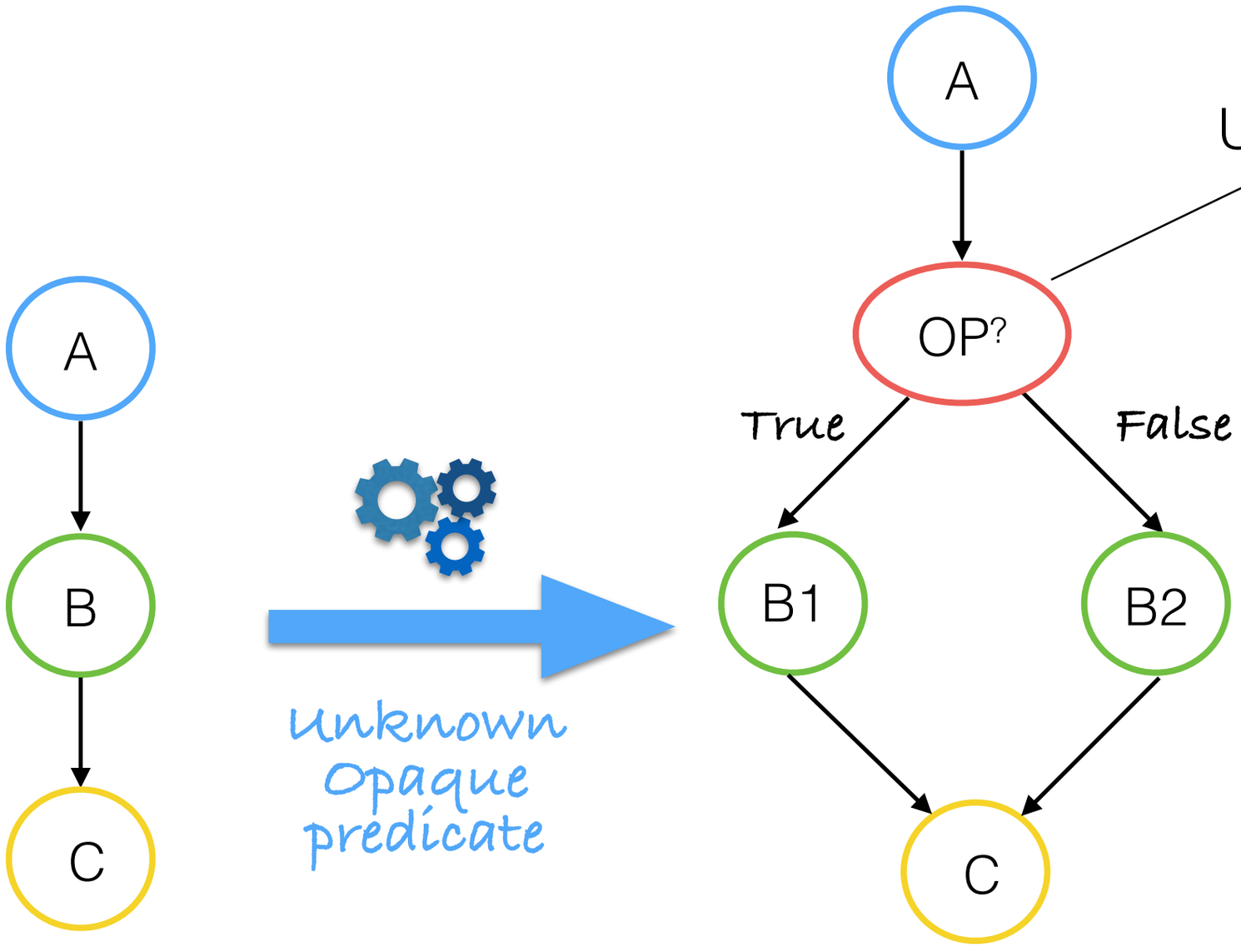
➡ Soundness can be forced or harmed by transforming the analysis or the program?

Analyze a single trace to better understand what went wrong or for runtime monitoring/verification

?



Code Obfuscation & Dynamic Analysis



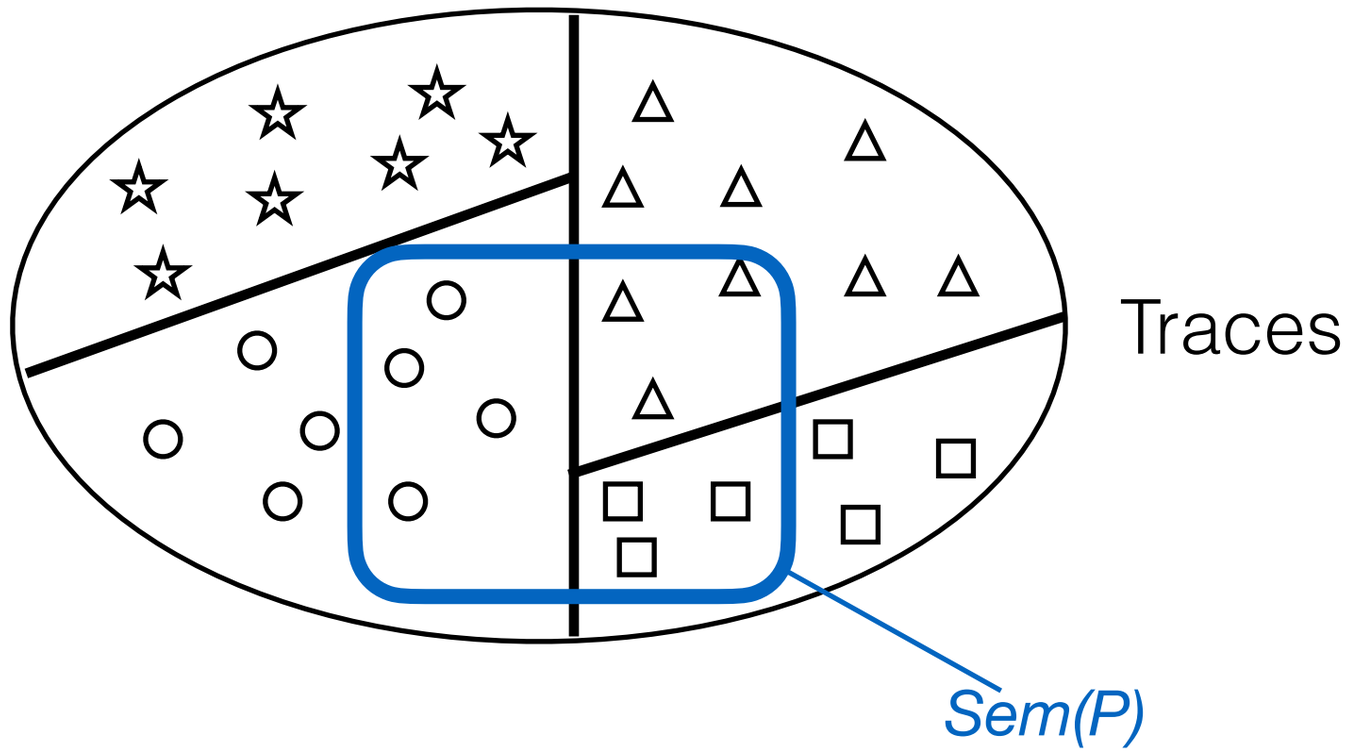
Add *diversification* for the control flow analysis

Unsoundness?

Formalizing Dynamic Analysis

Property of single trace (no properties of sets of traces)

Equivalence Relation $\mathcal{A}$



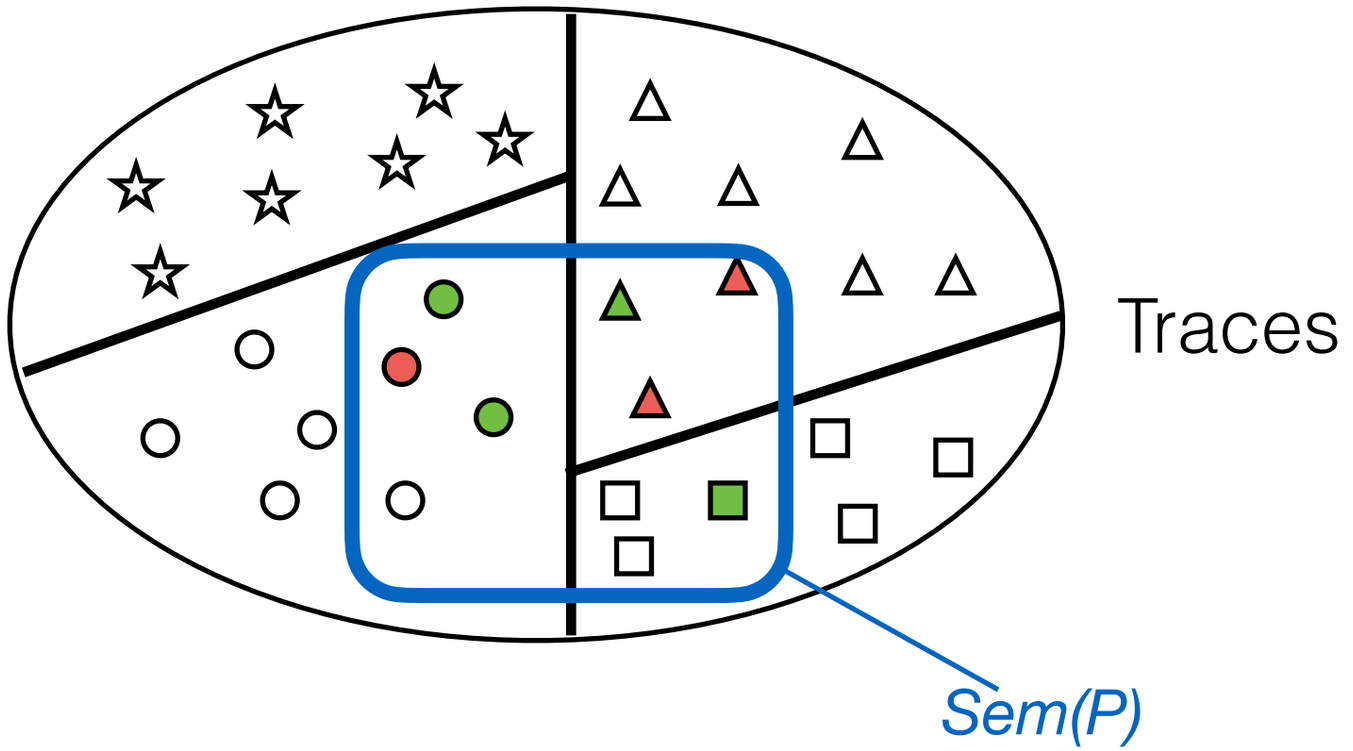
$$\mathcal{A}(Sem(P)) = \{[\sigma]_{\mathcal{A}} \mid \sigma \in Sem(P)\}$$

Formalizing Dynamic Analysis

Property of single trace (no properties of sets of traces)

Equivalence Relation $\mathcal{A}$

Dynamic Analysis
(Exe(P),$\mathcal{A}$)
sound
unsound



Soundness

$$\mathcal{A}(Sem(P)) = \mathcal{A}(Exe(P))$$

$$\mathcal{A}(Sem(P)) = \{[\sigma]_{\mathcal{A}} \mid \sigma \in Sem(P)\}$$

Formalizing Dynamic Analysis

Soundness

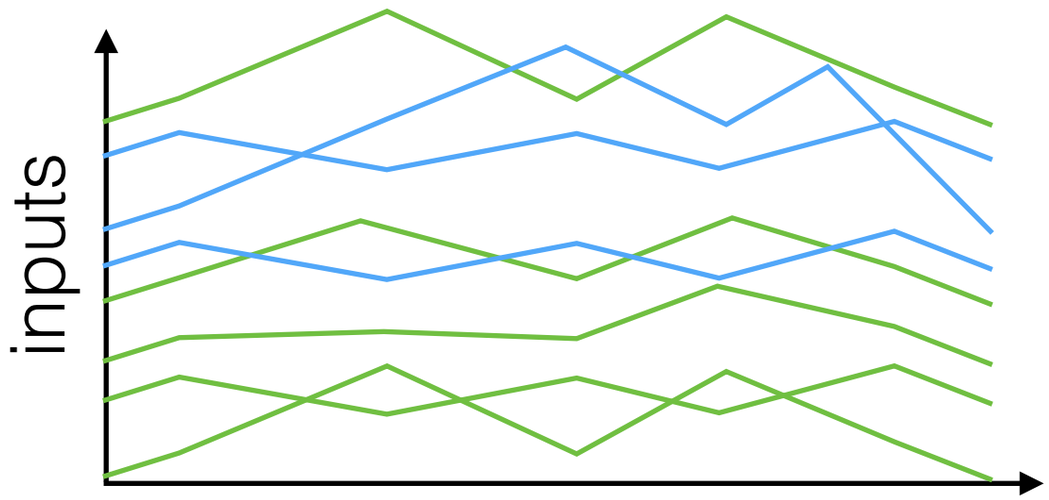
$$\mathcal{A}(Sem(P)) = \mathcal{A}(Exe(P))$$

Every equivalence class of Sem(P) is represented in Exe(P)

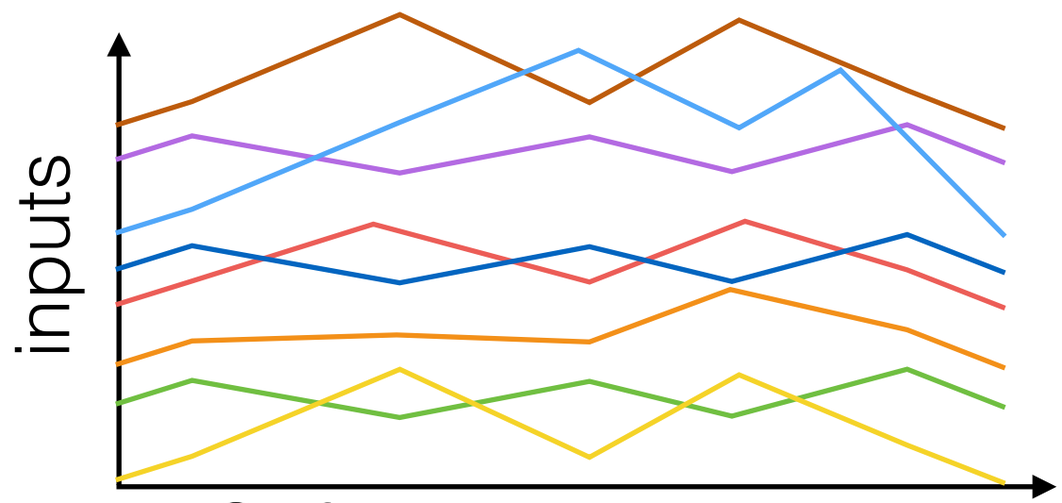
Exe(P) covers P wrt $\mathcal{A}$

Obfuscating Dynamic Analysis

The key for harming dynamic analysis is **diversification wrt the property being analysed**



Original program



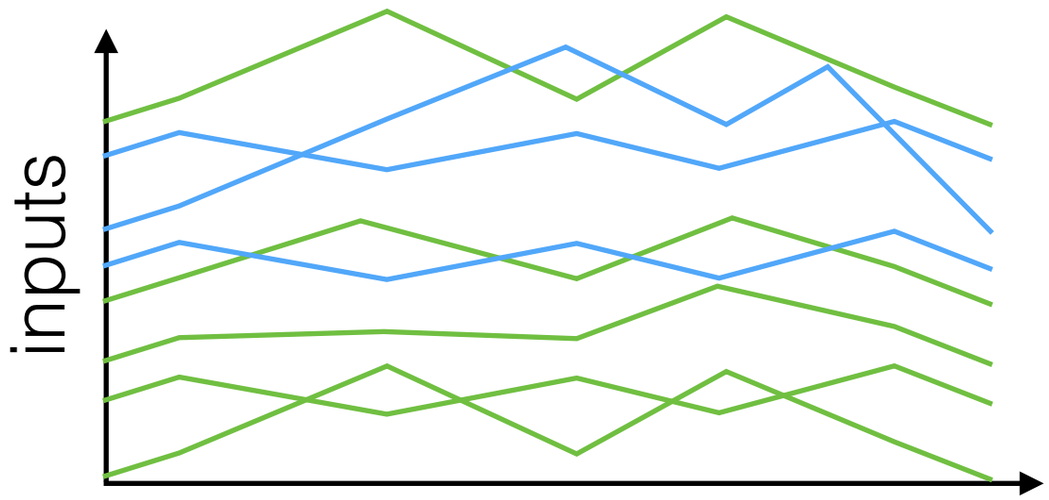
Obfuscated program

Colours represents the equivalence classes wrt $\mathcal{A}$

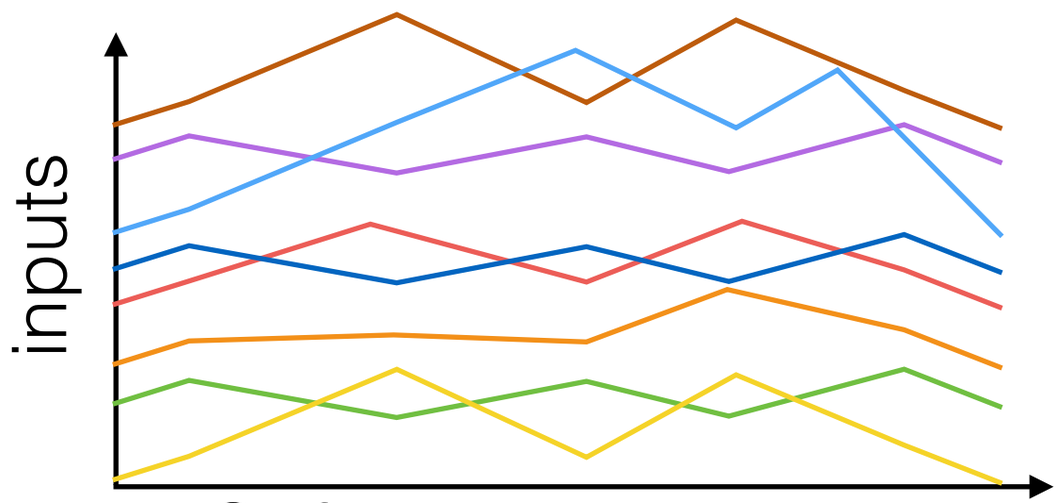
Ideally: specialise the program for every input wrt $\mathcal{A}$

Obfuscating Dynamic Analysis

The key for harming dynamic analysis is **diversification wrt the property being analysed**



Original program

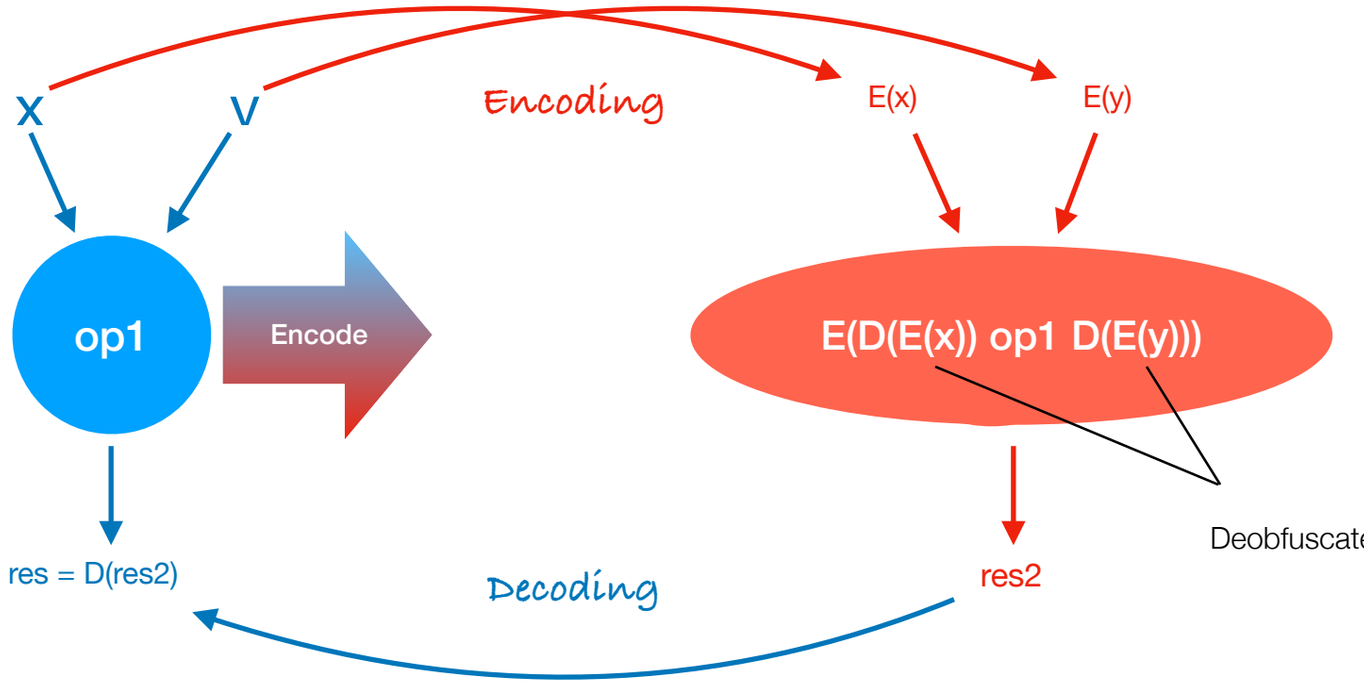


Obfuscated program

A program transformation O: Programs ⟶ Programs obfuscates property $\mathcal{A}$:

- O **preserves the observational behavior** of programs
- The property $\mathcal{A}$ of O(P) is **diversified** wrt P

Data Obfuscation



Data Obfuscation

P

```
input x;
sum := 0;
while x < 50
```

• $X = [x, 49]$

```
    sum := sum + x;
    x := x + 1;
```

$\mathcal{T}(P)$

```
input x;
x := 2*x;        (Encoding)
sum := 0;
while x < 2*50
```

• $X = [x, 2 * 50 - 1]$

```
    sum := sum + x/2;
    x := x + 2;
x:= x/2;         (Decoding)
```

No effects on dynamic analysis

Dynamic Data Obfuscation

P

```
input x;
sum := 0;
while x < 50
• X = [x, 49]
    sum := sum + x;
    x := x + 1;
```

Diversification
Parametric data encoding

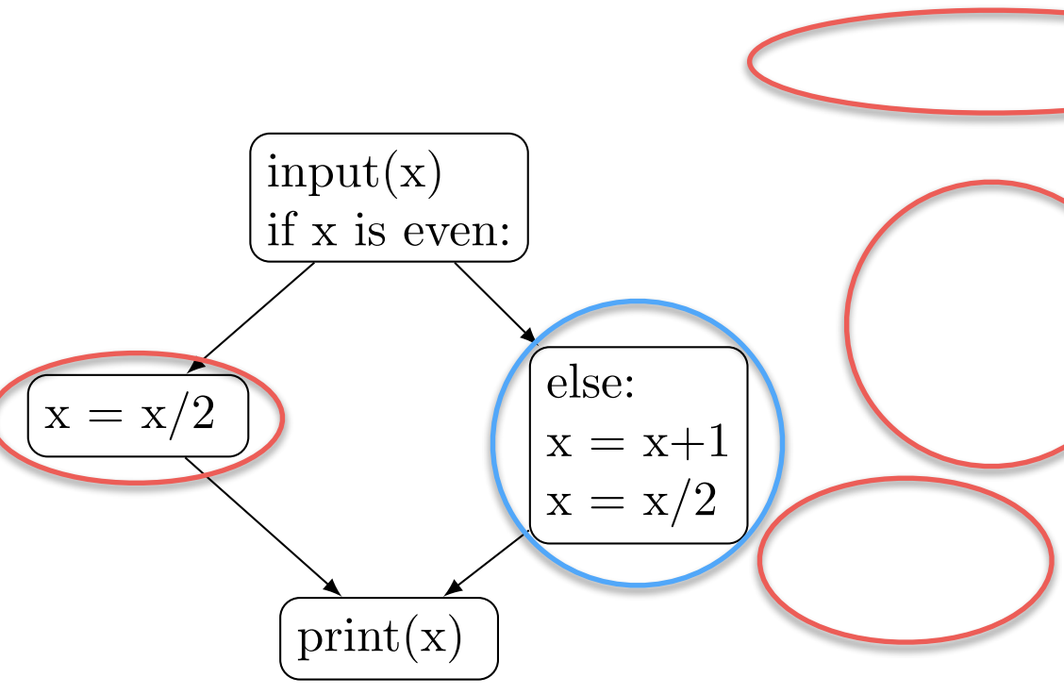
Dynamic analysis needs to observe at least n traces

$\mathcal{T}_n(P)$

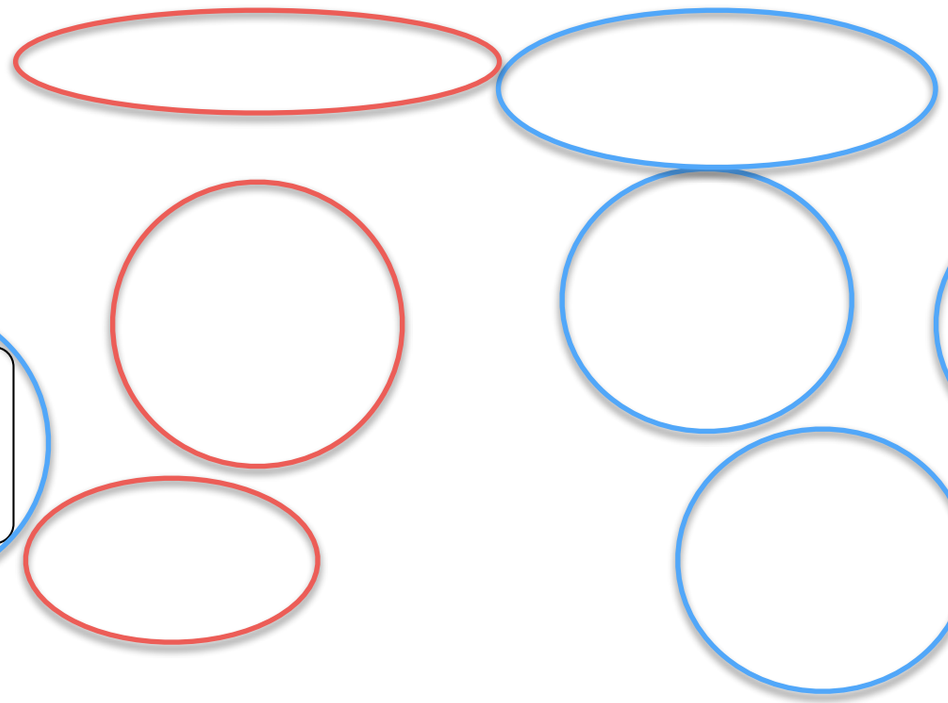
```
input x;
n := select(N,x);   Encoding
x := n*x;
sum := 0;
while x < n*50
• X = [x, n * 50 − 1]
    sum := sum + x/n;
    x := x + n;
x:= x/n;   Decoding
```

Dynamic CFG Obfuscation

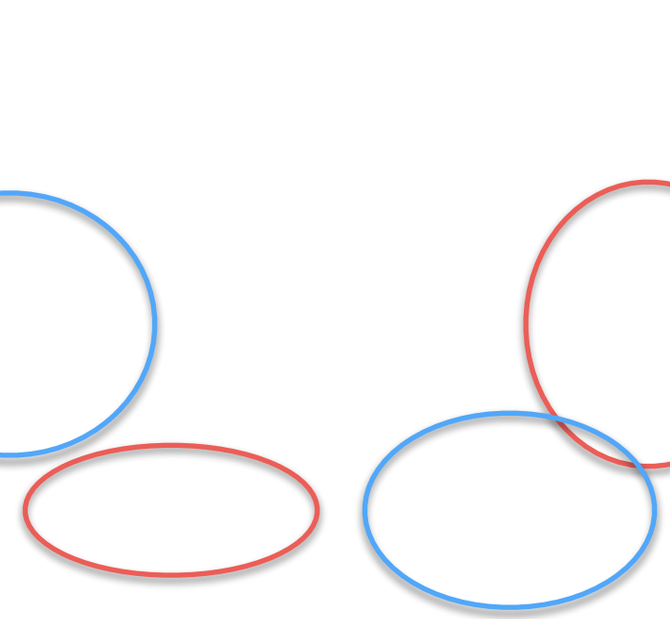
Diversification



P



Range Dividers 2016



Gadget diversification 2011

Coverage Criteria

Statement Coverage

Count-statement Coverage

Path Coverage

Count-path Coverage

Testing

Fuzzing

...

Equivalence Relation $\mathcal{C}$

Compare coverage criteria

$\mathcal{C}1 \quad \{<, =, >\} \quad \mathcal{C}2$

Compare coverage criteria
and properties under analysis

$\mathcal{C} < \mathcal{A}$

Soundness!!

Open Issues

➡ Properties of set of traces, other properties? Topological characterisation wrt to the kind of property being analysed

➡Model validation (ORAM, fuzzers, input generator and recognisers,…)

➡Potentiality and limits of code obfuscation for dynamic analysis

➡ Extend the model with measure of likelihodd of the inputs (probability distribution over the input)