Foundations of Code Obfuscation

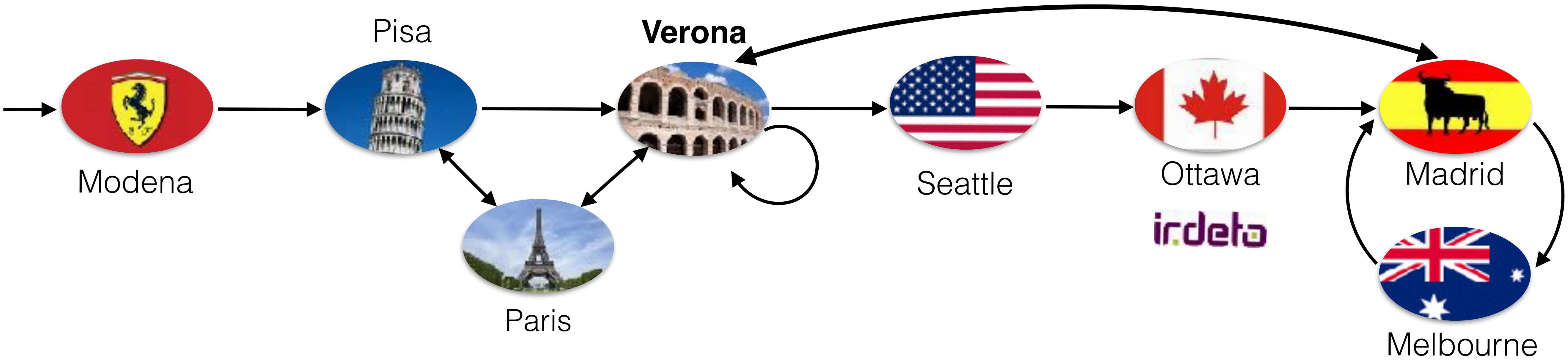
a hacking view on program analysis and understanding

Roberto Giacobazzi

ISSISP 2017

Gif sur Yvette 2017

About me



Activities

- Professor in Verona & IMDEA SW Institute
- Co-founder of JULIA & TESTOODO
- Partner in cythereal

Programming (coding)

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

Understanding

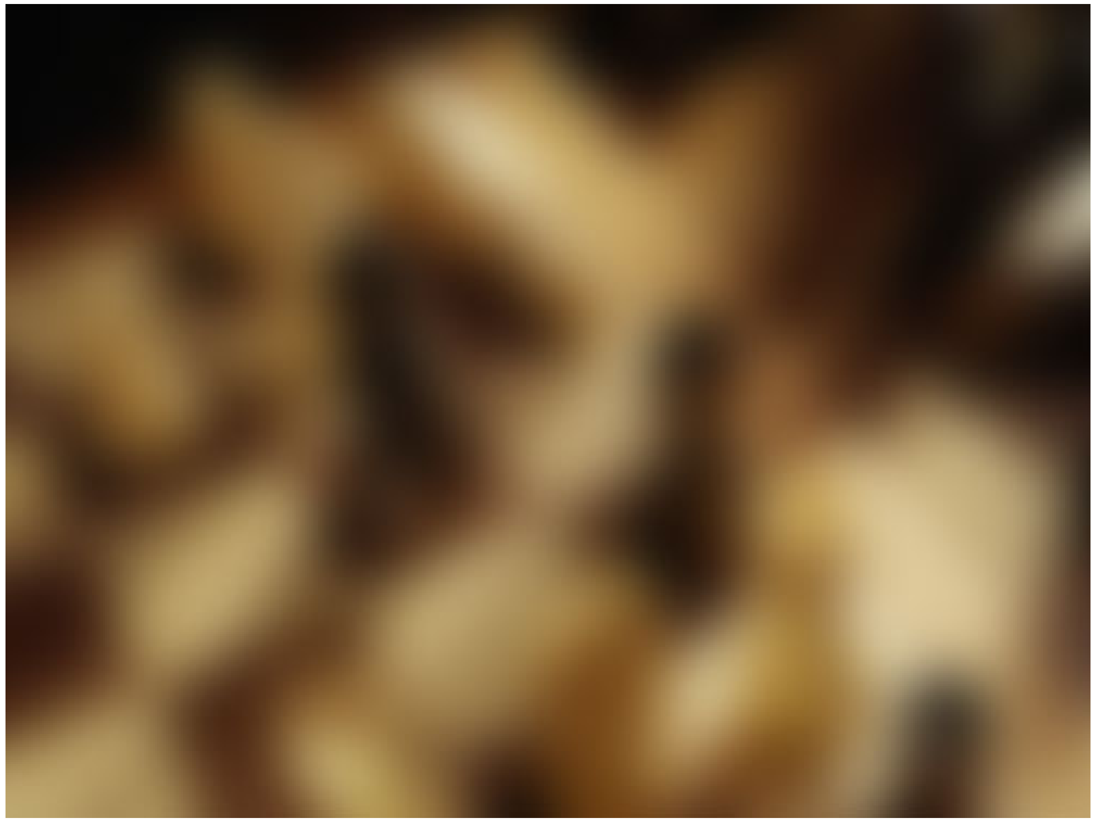
```
{n0>=0}
   n := n0;
{n0=n,n0>=0}
   i := n;
{n0=i,n0=n,n0>=0}
   while (i <> 0 ) do
      {n0=n,i>=1,n0>=i}
         j := 0;
      {n0=n,j=0,i>=1,n0>=i}
         while (j <> i) do
            {n0=n,j>=0,i>=j+1,n0>=i}
               j := j + 1
            {n0=n,j>=1,i>=j,n0>=i}
         od;
     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

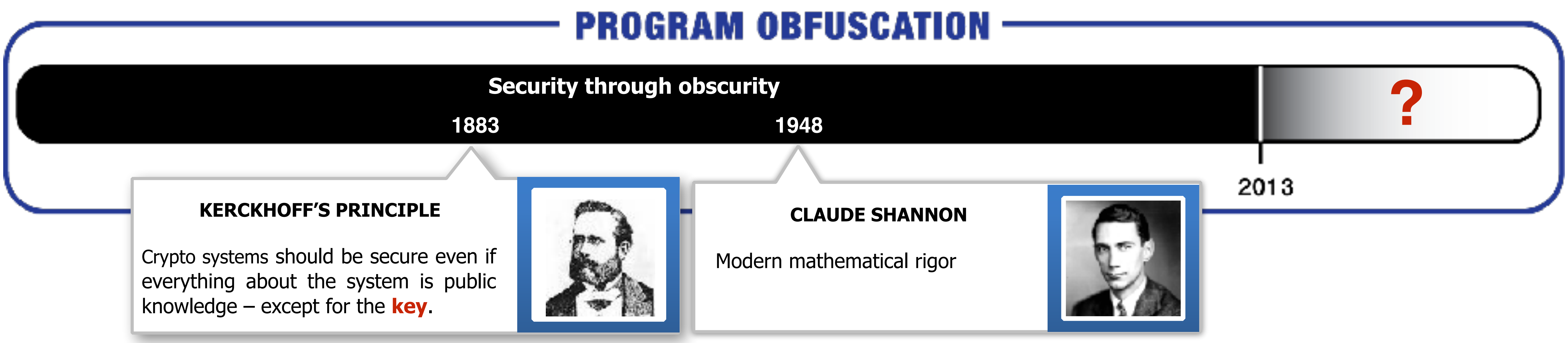
Obfuscating

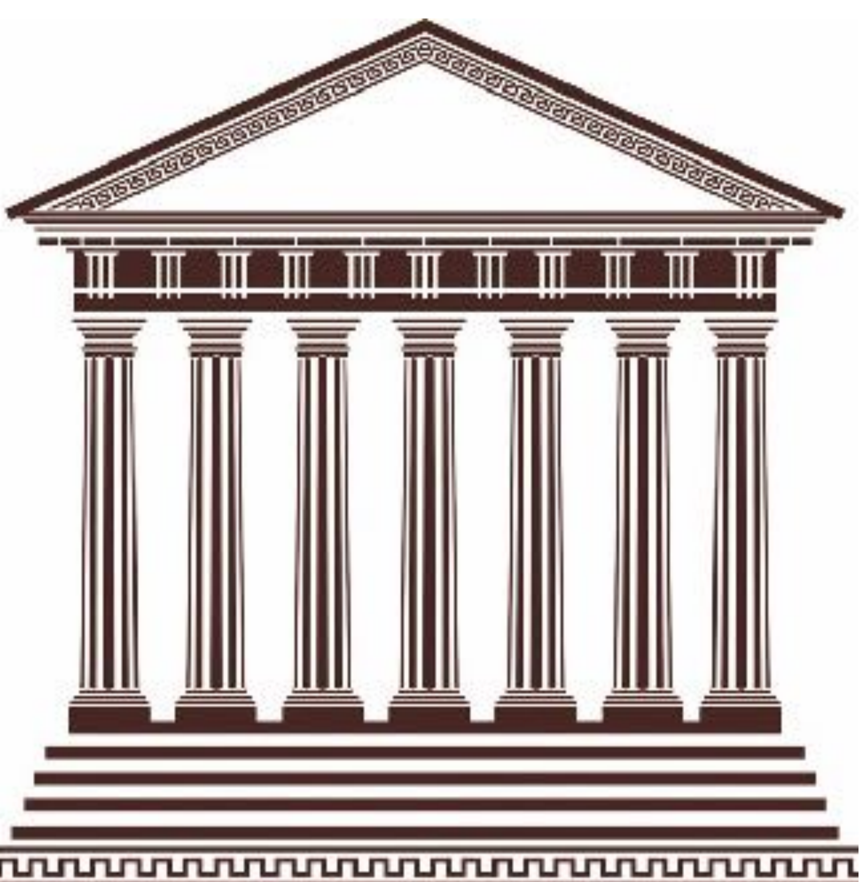
```
{n0>=0}
   n := n0;
{n0=n,n0>=0}
   i := n;
{n0=i,n0=n,n0>=0}
   while (i <> 0 ) do
      {n0=n,i>=1,n0>=i}
         j := 0;
      {n0=n,j=0,i>=1,n0>=i}
         while (j <> i) do
            {n0=n,j>=0,i>=j+1,n0>=i}
               j := j + 1
            {n0=n,j>=1,i>=j,n0>=i}
         od;
     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

Obfuscating

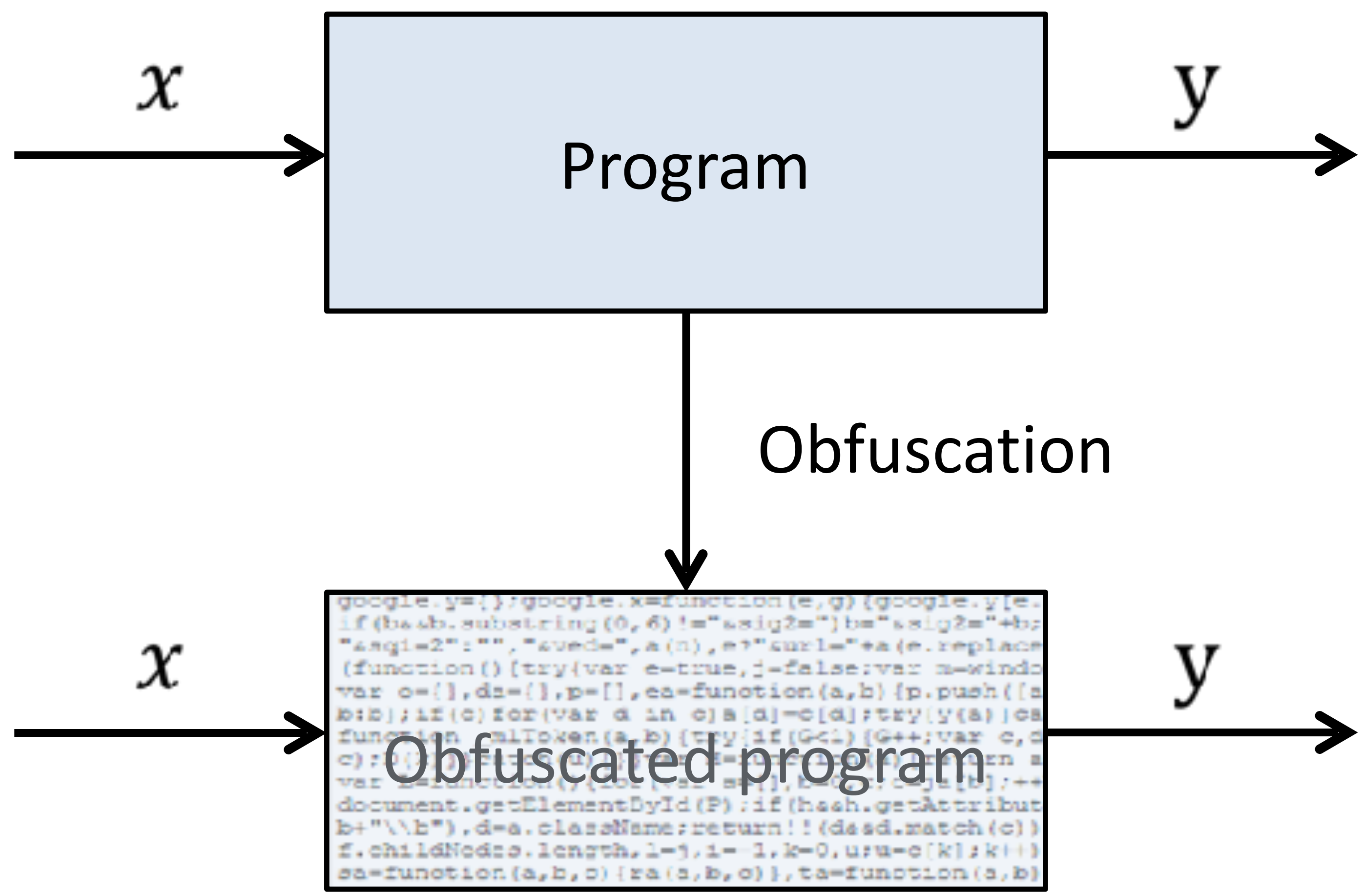
Obfuscating



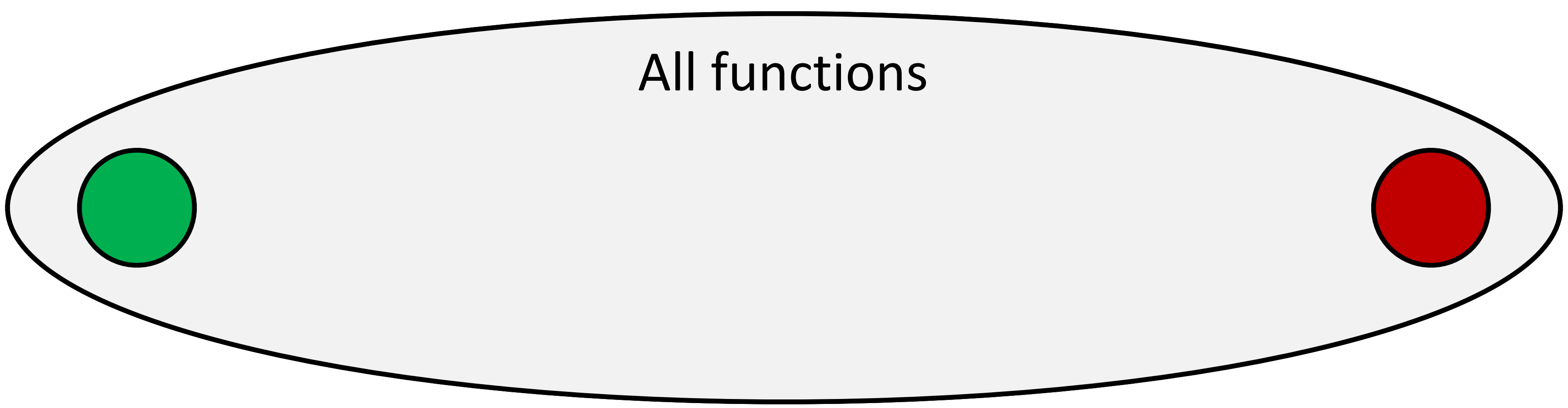




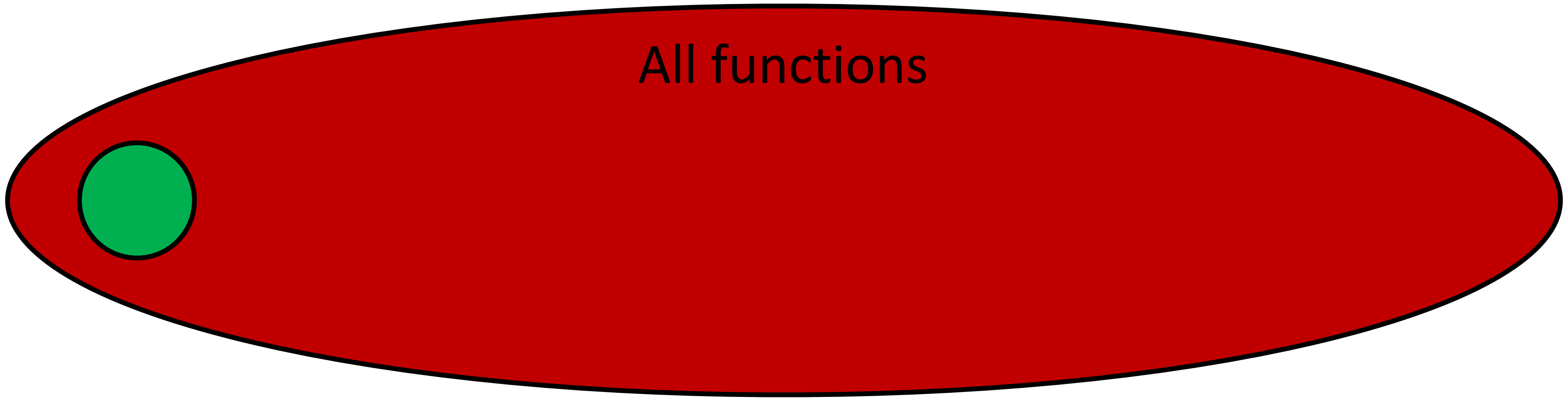
A Crypto-like Foundation of Obfuscation


x
Program
y
Obfuscation
x
Obfuscated program
y

< 2001: No general solution



< 2001: *No general solution*

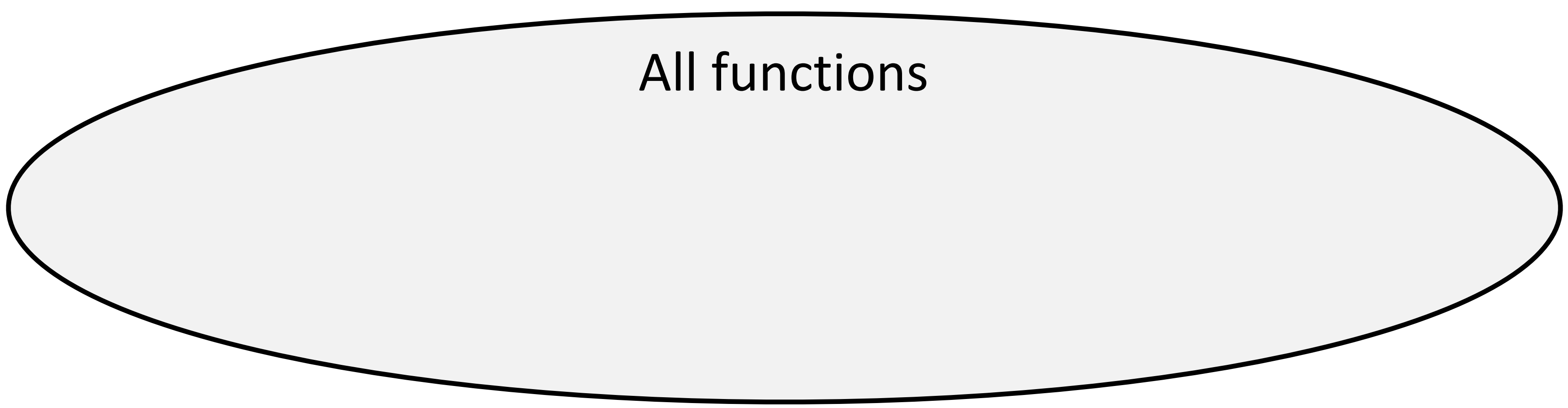


> 2001

Hides everything about the program except for its input\output behaviour

> 2001

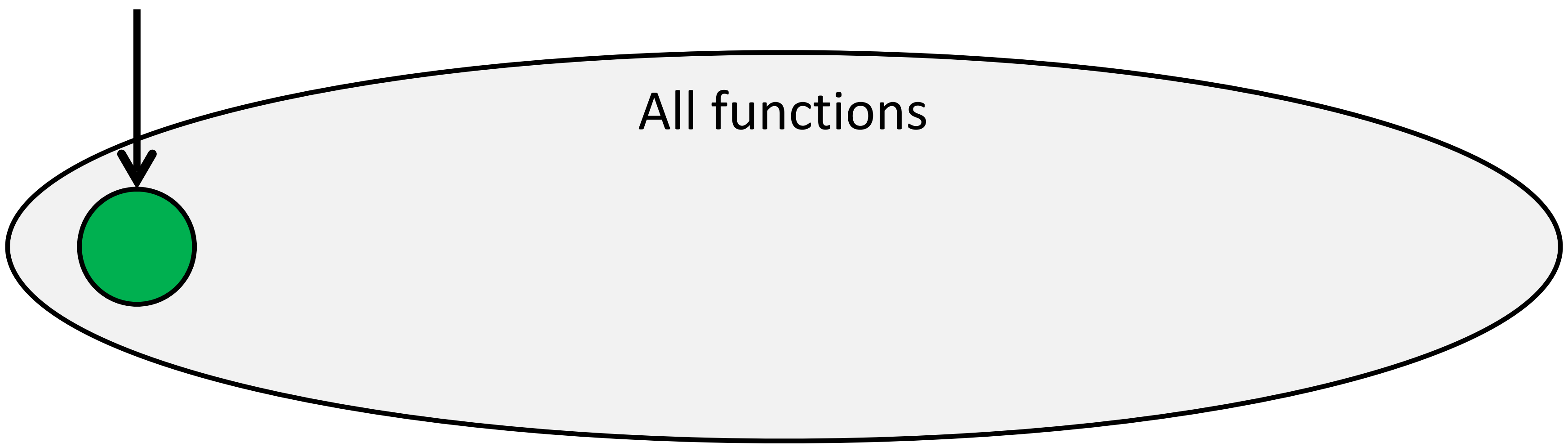
Hides everything about the program except for its input\output behaviour



> 2001

Hides everything about the program except for its input\output behaviour

Point Function etc.
[Canetti 97, Wee 05, Bitansky-Canetti 10, Canetti-Rothblum-Varia 10]

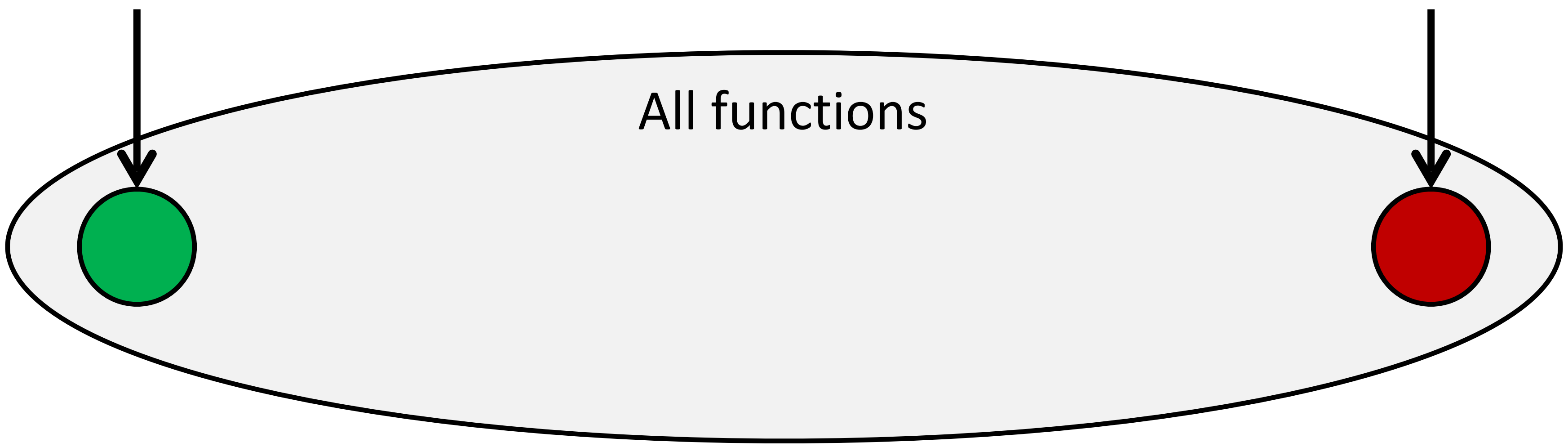


> 2001

Hides everything about the program except for its input\output behaviour

Point Function etc.
[Canetti 97, Wee 05, Bitansky-Canetti 10, Canetti-Rothblum-Varia 10]

Unobfuscatable Functions
[Barak-Goldreich-Impagliazzo-Rudich-Sahai-Vadhan-Yang 01]

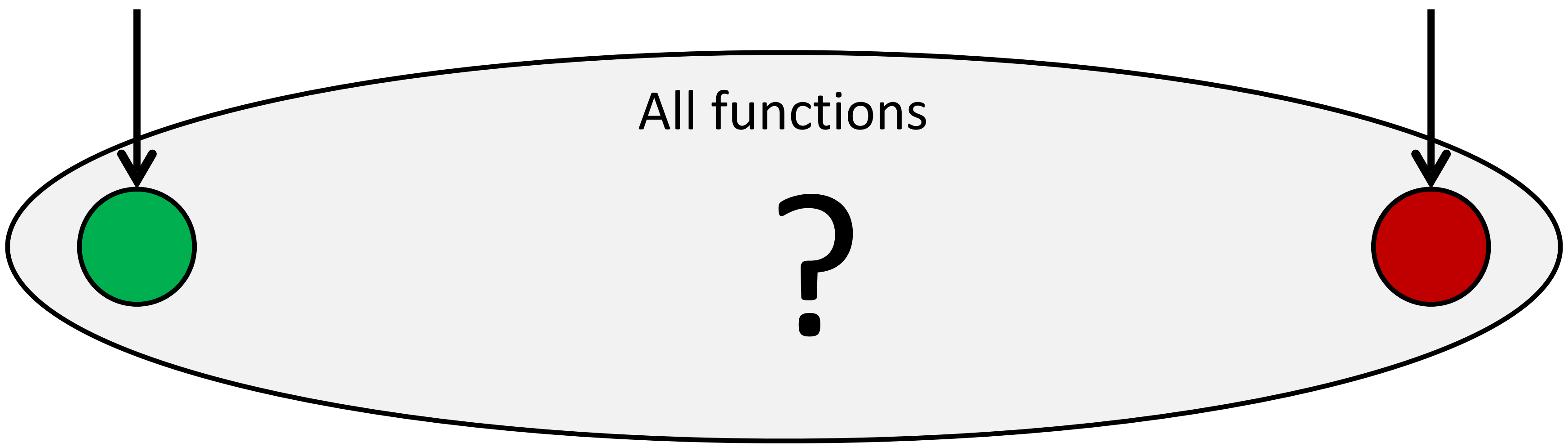


> 2001

Hides everything about the program except for its input\output behaviour

Point Function etc.
[Canetti 97, Wee 05, Bitansky-Canetti 10, Canetti-Rothblum-Varia 10]

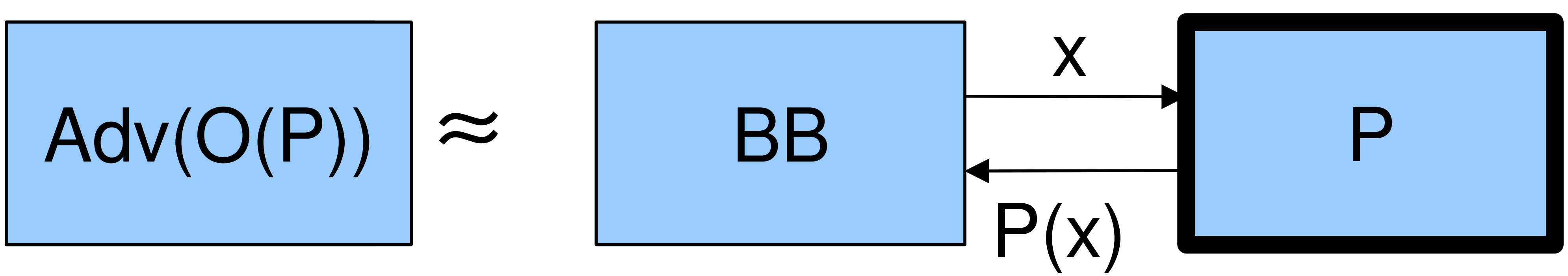
Unobfuscatable Functions
[Barak-Goldreich-Impagliazzo-Rudich-Sahai-Vadhan-Yang 01]



What is an Obfuscator?

An **obfuscator** is an algorithm **O** such that for any program **P, O(P)** is a program such that:

- **O(P)** has the **same functionality** as **P**
- **O(P)** is **hard** to analyse / “reverse-engineer”.



The model

We are interested in 2 types of polynomial-time analyzers:

- ⇨ Ana is a source-code analyzer that can read the program.

$$Ana(P)$$

- ⇨ BAna is a black-box analyzer that only queries the program as an oracle.

$$BAna^{P}(time(P))$$

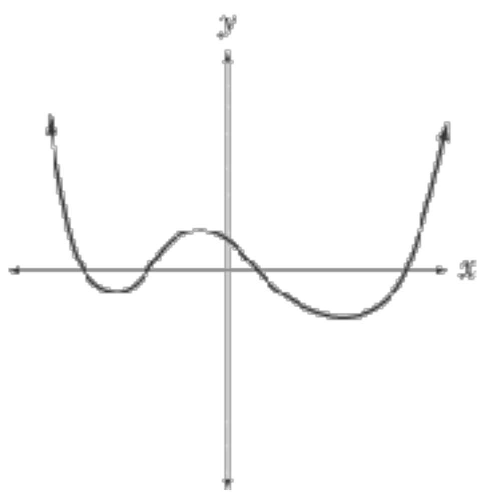
Black-Box security

Ana can't get more information than BAna could ⇦ VBB

Obfuscation

Functionality

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

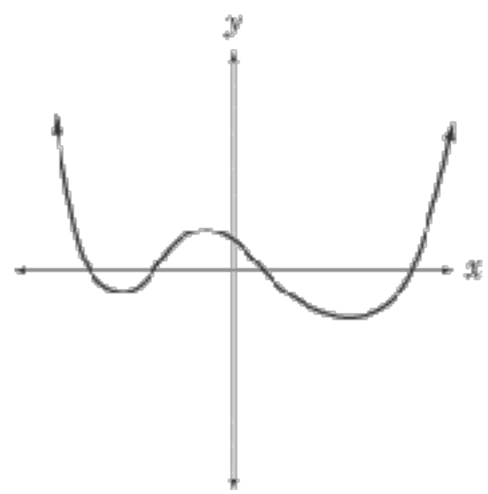


*“Anything that can be learned from the obfuscated form, could have been learned by merely observing the program’s input-output behavior (i.e., by treating the program as a **black-box**)”*

Obfuscation

Functionality

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



*“Anything that can be learned from the obfuscated form, could have been learned by merely observing the program’s input-output behavior (i.e., by treating the program as a **black-box**)”*

Obfuscation

Polynomial Slowdown

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

*"Anything that can be learned from the obfuscated form, could have been learned by merely observing the program's input-output behavior (i.e., by treating the program as a **black-box**)"*

Obfuscation

Polynomial Slowdown

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

"Anything that can be learned from the obfuscated form, could have been learned by merely observing the program's input-output behavior (i.e., by treating the program as a ***black-box****)"*

|O(P)| ≤ poly(|P|) for some polynomial **poly( )**

O is **efficient** if it runs in polynomial time

Obfuscation

Virtual Black-Box

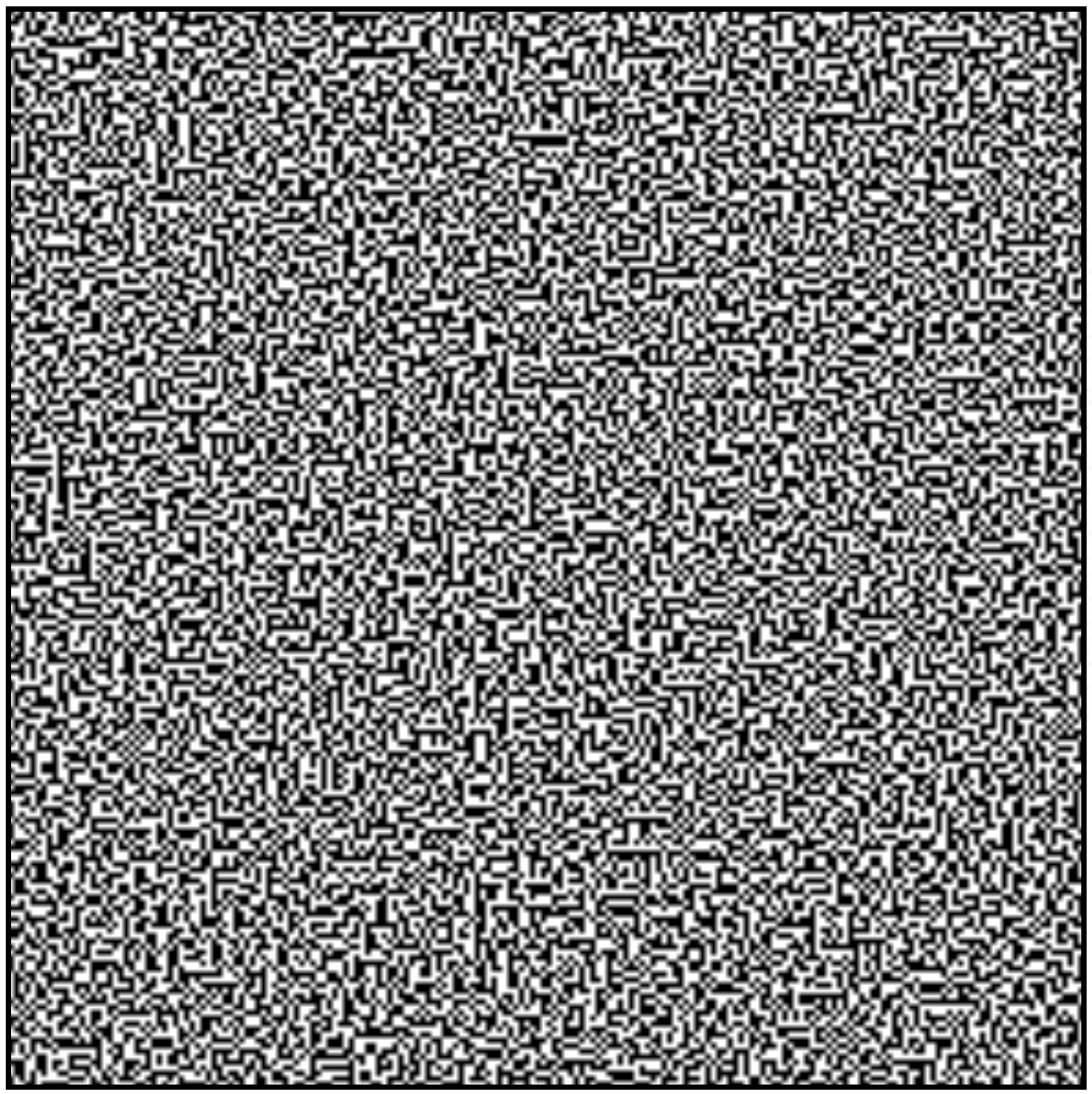
```
n := n0;
i := n;
while (i <> 0 ) do
    j := 0;
    while (j <> i) do
        j := j + 1
    od;
    i := i - 1
od
```

*"Anything that can be learned from the obfuscated form, could have been learned by merely observing the program's input-output behavior (i.e., by treating the program as a **black-box**)"*

$$|Pr[A(\mathcal{O}(M)) = 1] - Pr[S^M(1^{|M|}) = 1]| \leq \varepsilon(|M|)$$

Obfuscation

Virtual Black-Box



*"Anything that can be learned from the obfuscated form, could have been learned by merely observing the program's input-output behavior (i.e., by treating the program as a **black-box**)"*

$$|Pr[A(\mathcal{O}(M)) = 1] - Pr[S^M(1^{|M|}) = 1]| \leq \varepsilon(|M|)$$

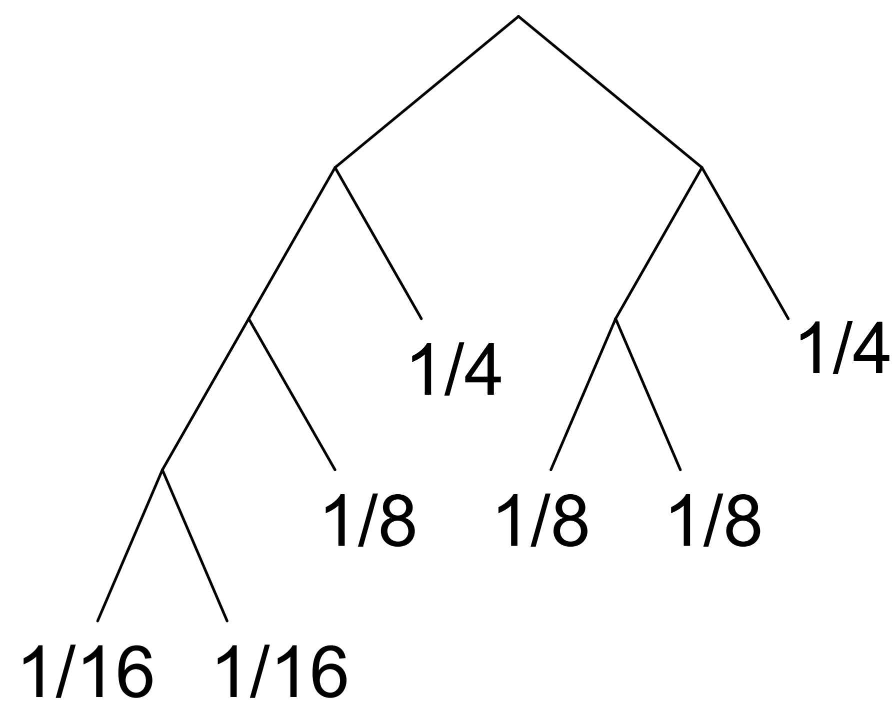
Is this possible?

Probabilistic Polynomial Time TM
PPT-TM

- New kind of NTM, in which each nondeterministic step is a coin flip: has exactly 2 next moves, to each of which we assign probability ½.
- Example:
  - To each maximal branch, we assign a probability:

$$\underbrace{\tfrac{1}{2} \times \tfrac{1}{2} \times \ldots \times \tfrac{1}{2}}_{\text{number of coin flips on the branch}}$$

Computation on input w



- Has accept and reject states, as for NTMs.
- Now we can talk about probability of acceptance or rejection, on input w.

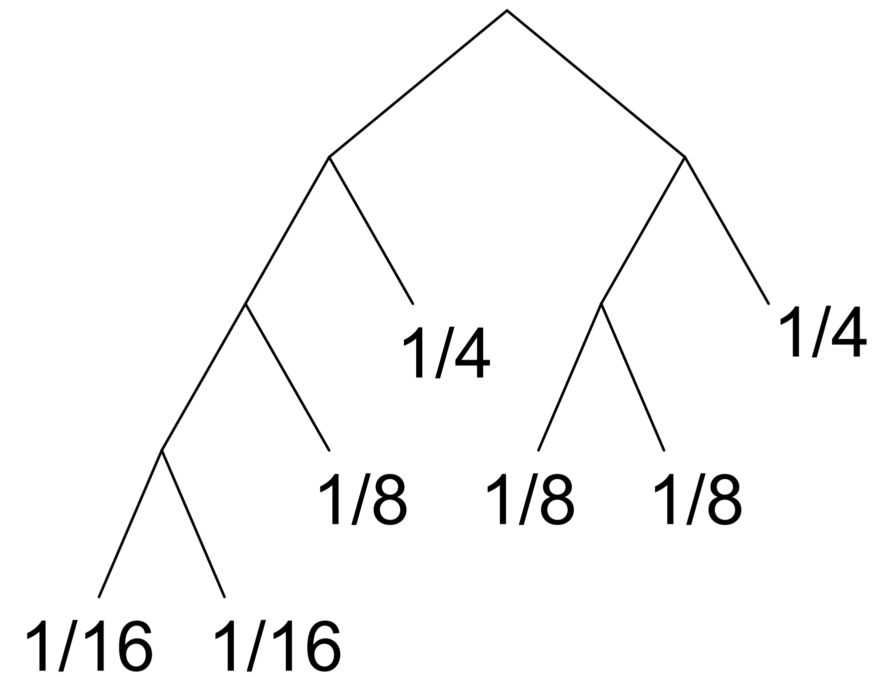
Probabilistic Polynomial Time TM
PPT-TM

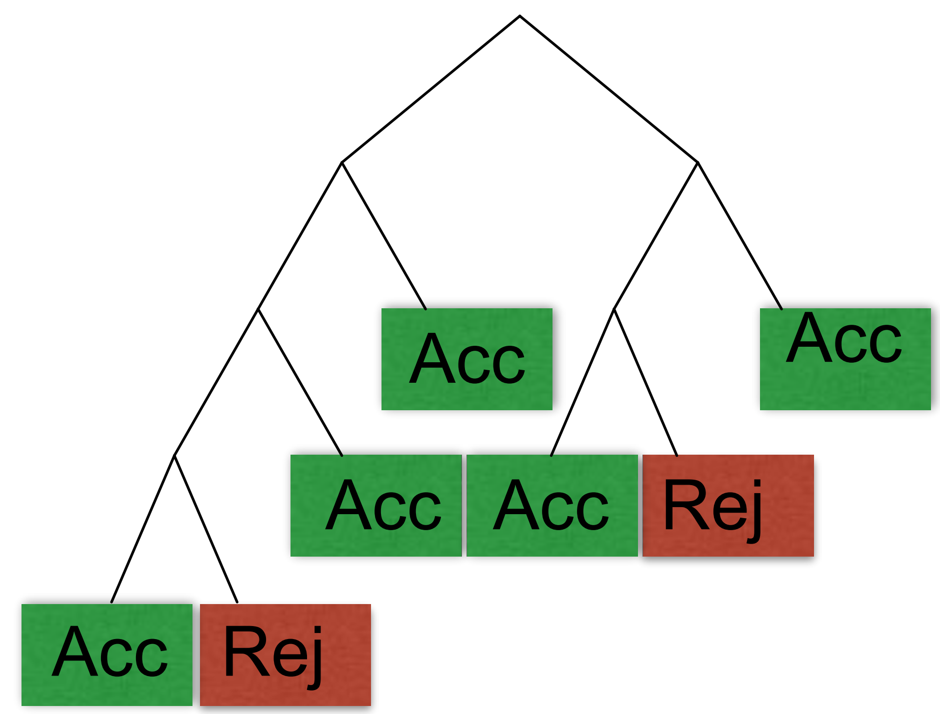
- Probability of acceptance =

  $\Sigma_{\text{b an accepting branch}}$ Pr(b)
- Probability of rejection =

  $\Sigma_{\text{b a rejecting branch}}$ Pr(b)
- Example:
  - Add accept/reject information
  - Probability of acceptance = 1/16 + 1/8 + 1/4 + 1/8 + 1/4 = 13/16
  - Probability of rejection = 1/16 + 1/8 = 3/16
- We consider TMs that halt (either accept or reject) on every branch---deciders.
- So the two probabilities total 1.

Computation on input w





One-way Functions

A one-way function is a function that is easy to compute but computationally hard to reverse

- Easy to calculate $f(x)$ from x
- Hard to invert: to calculate x from $f(x)$

There is no proof that one-way functions exist, or even real evidence that they can be constructed

Definition A function $f\colon \{0,1\}^* \to \{0,1\}^*$ is *one-way* if:

(1) there exists a PPT that on input x output $f(x)$;

(2) For every PPT algorithm A there is a negligible function ν_A such that for sufficiently large k,

$$\mathbf{Pr}\left[\, f(z) = y \;:\; x \xleftarrow{\$} \{0,1\}^k \;;\; y \leftarrow f(x) \;;\; z \leftarrow A(1^k, y) \right] \;\le\; \nu_A(k)$$

Obfuscating Point Functions

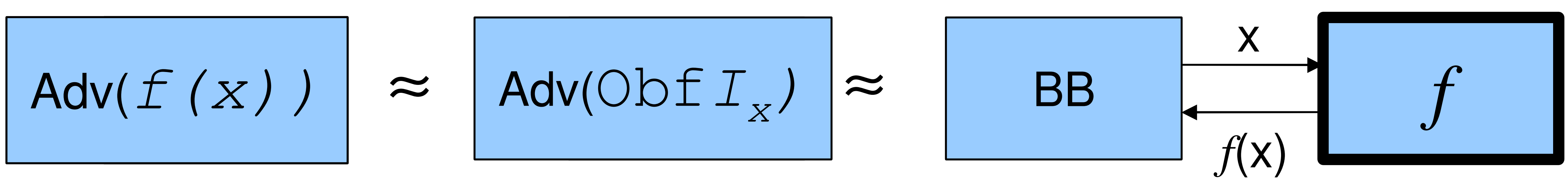
Point function: $$I_x(w) = \begin{cases} 1 & \text{if } w = x \\ 0 & \text{otherwise} \end{cases}$$

One-way functions f obfuscate I_x

Let y = f(x) then Obf-I_x obfuscates I_x

```
Program Obf-I_x(w): {if y=f(w) then 1 else 0}
```

Idea: y = f(x) reveals no more than VBB access to I_x



Is it possible for all programs?

Obfuscation for arbitrary TM

Impossible!

- ~~Functionality~~
- ~~Polynomial Slowdown~~
- ~~Virtual Black-Box~~

There exists an attacker A and a program P for which NO VBB obfuscation O does work!

Lemma: Proof for 2TMs (C & D)

$\alpha, \beta \in \{0,1\}^k$ Secrets!!

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(X) = \begin{cases} 1 & \text{if } X \equiv C_{\alpha,\beta} \\ 0 & \text{otherwise} \end{cases}$$

Distinguish if X computes $C_{\alpha, \beta}$ from $C_{\alpha',\beta'}$ for any $(\alpha, \beta) \neq (\alpha',\beta')$ is NON COMPUTABLE!

$$Z_k(x) = 0^k$$

→ Simply compute $X(\alpha)$ for Poly(k) steps and check!

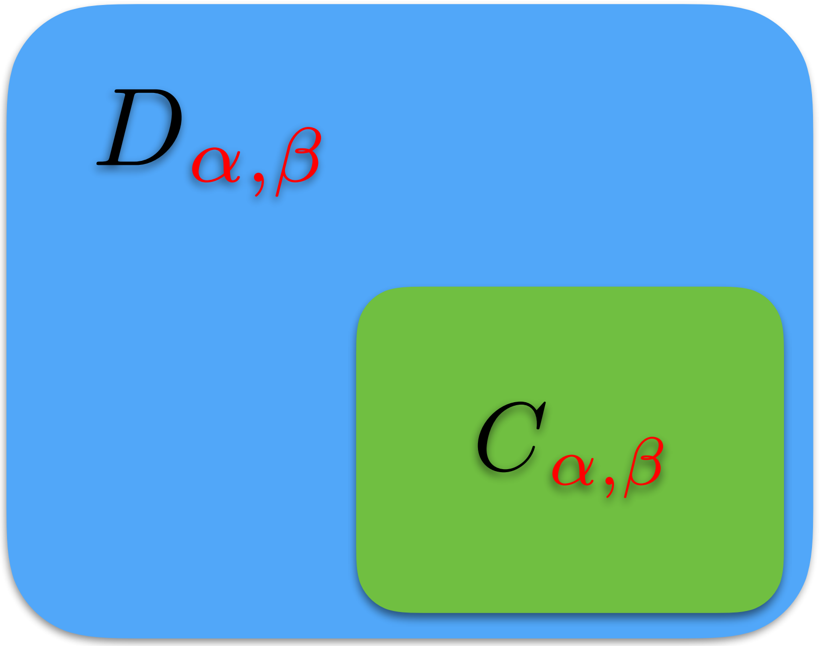
Idea: It is difficult distinguish ($C_{\alpha, \beta}$, $D_{\alpha, \beta}$) from (Z_k , $D_{\alpha, \beta}$) by VBB access to these programs!!

Idea: Proof for 2TMs (C,D)

The Functionality
preserves behaviour

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}), \mathcal{O}(D_{\alpha,\beta})) = 1] - Pr[S^{C_{\alpha,\beta}, D_{\alpha,\beta}}(1^k) = 1] \leq 2^{-\Omega(k)}$$

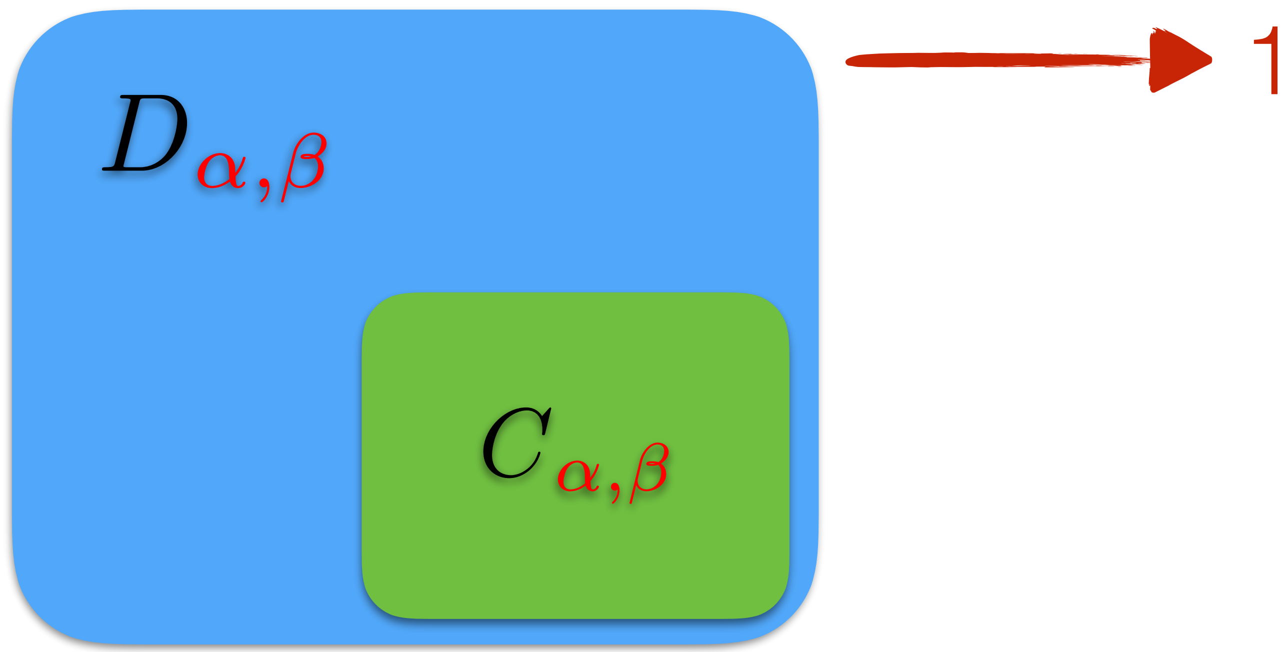
Idea: Proof for 2TMs (C,D)



The Functionality preserves behaviour

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]-Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]\leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)



The Functionality
preserves behaviour

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]-Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]\leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)

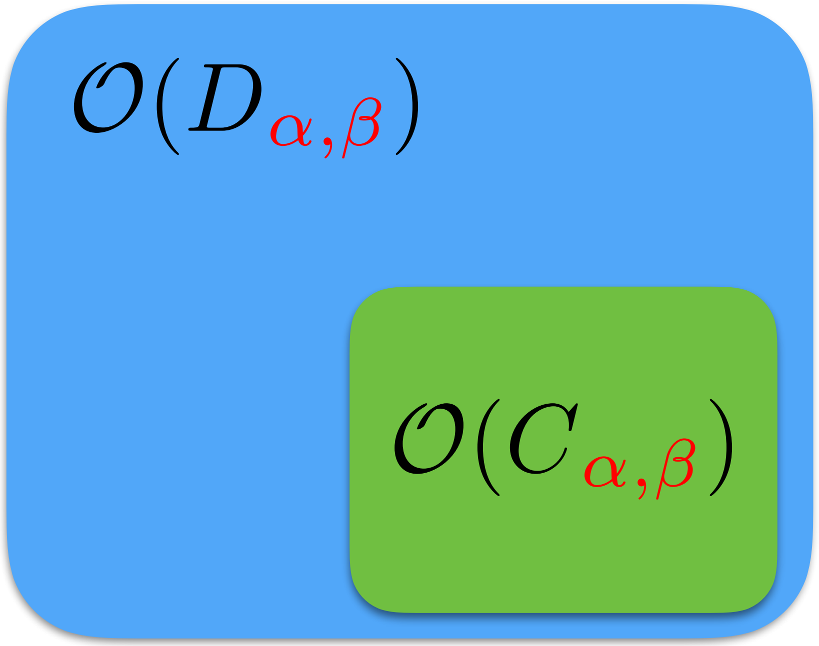
$\mathcal{O}(D_{\alpha,\beta})$

$\mathcal{O}(C_{\alpha,\beta})$

The Functionality
preserves behaviour
even if obfuscated

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]-Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]\leq 2^{-\Omega(k)}$$

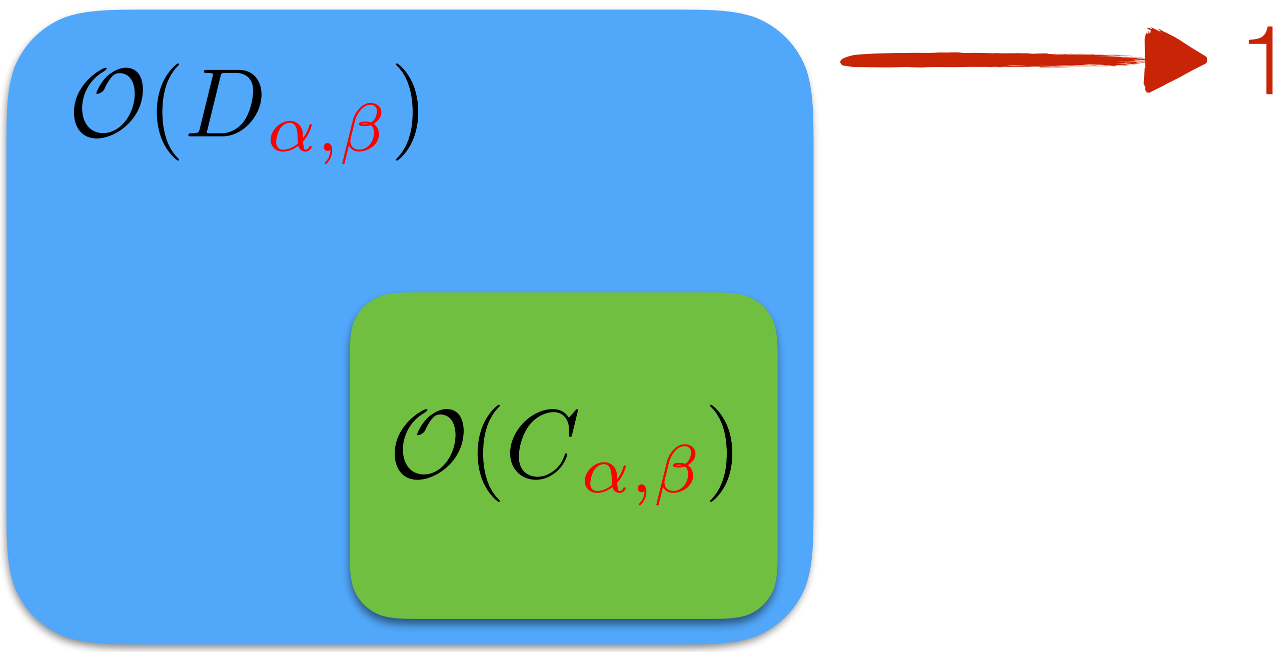
Idea: Proof for 2TMs (C,D)



The Functionality
preserves behaviour
even if obfuscated

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]-Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]\leq 2^{-\Omega(k)}$$

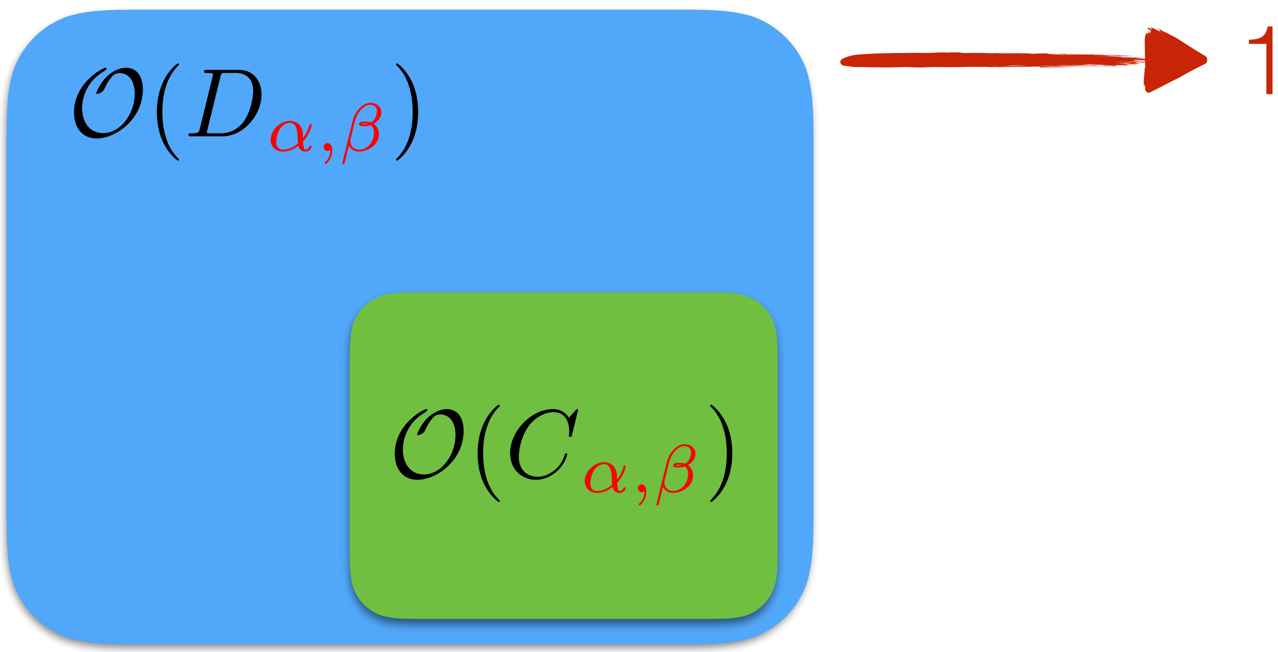
Idea: Proof for 2TMs (C,D)



The Functionality
preserves behaviour
even if obfuscated

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]-Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]\leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)

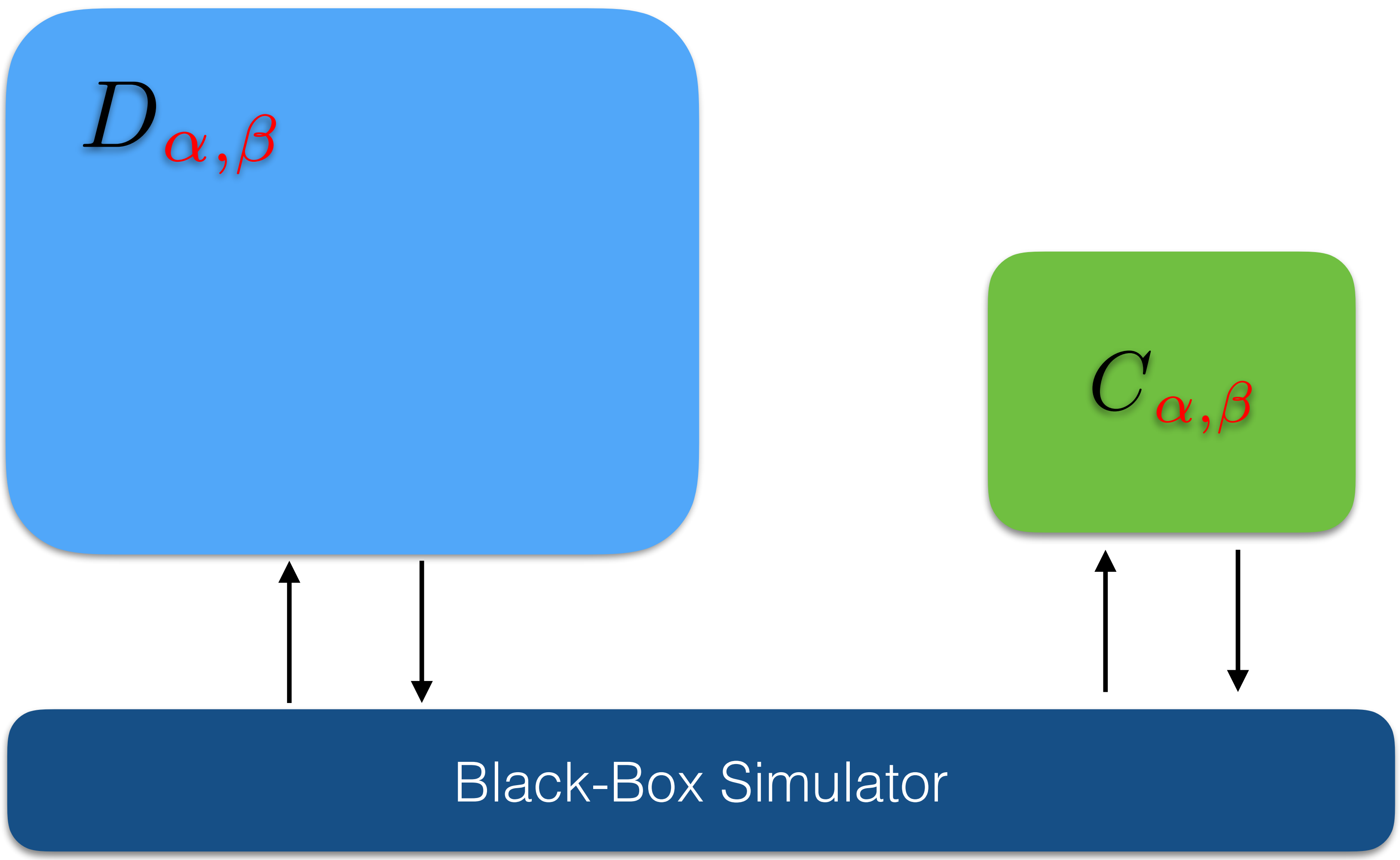


The Functionality preserves behaviour even if obfuscated

1

$$\boxed{Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]} - Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1] \leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)

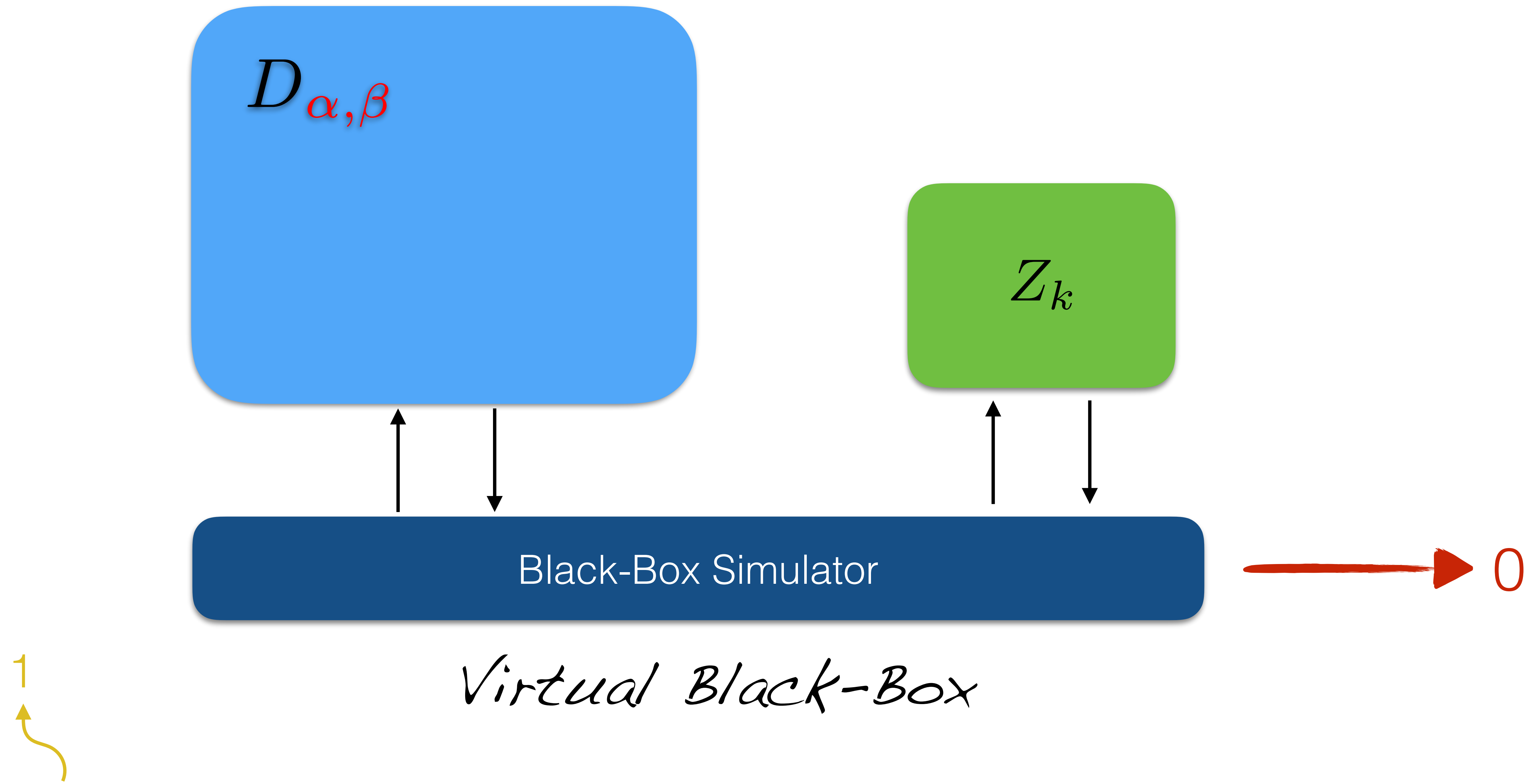


Virtual Black-Box

1

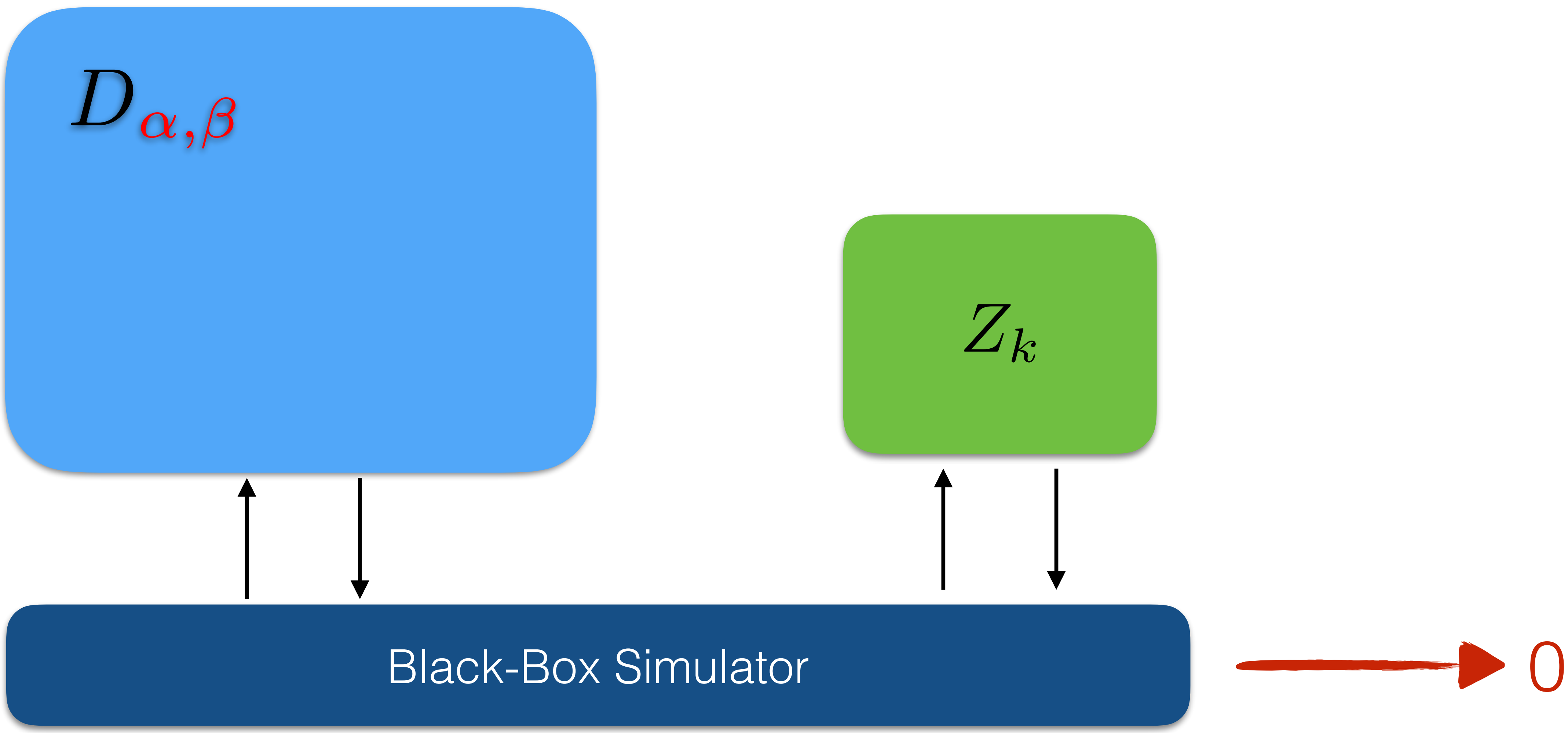
$$\boxed{Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]} - Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1] \leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)



$$\boxed{Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]} - Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1] \leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)

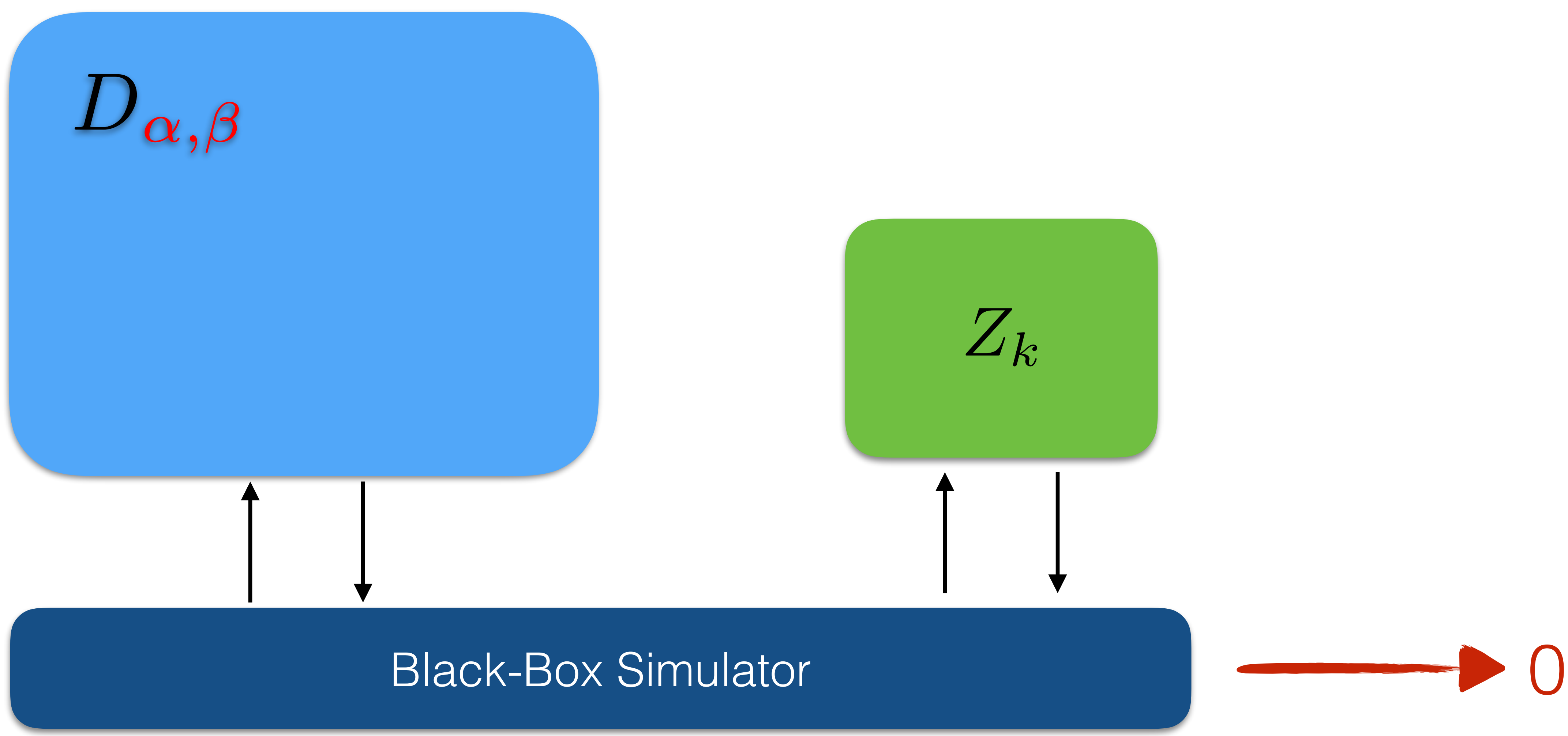


Virtual Black-Box

1 0

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1] - Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1] \leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)



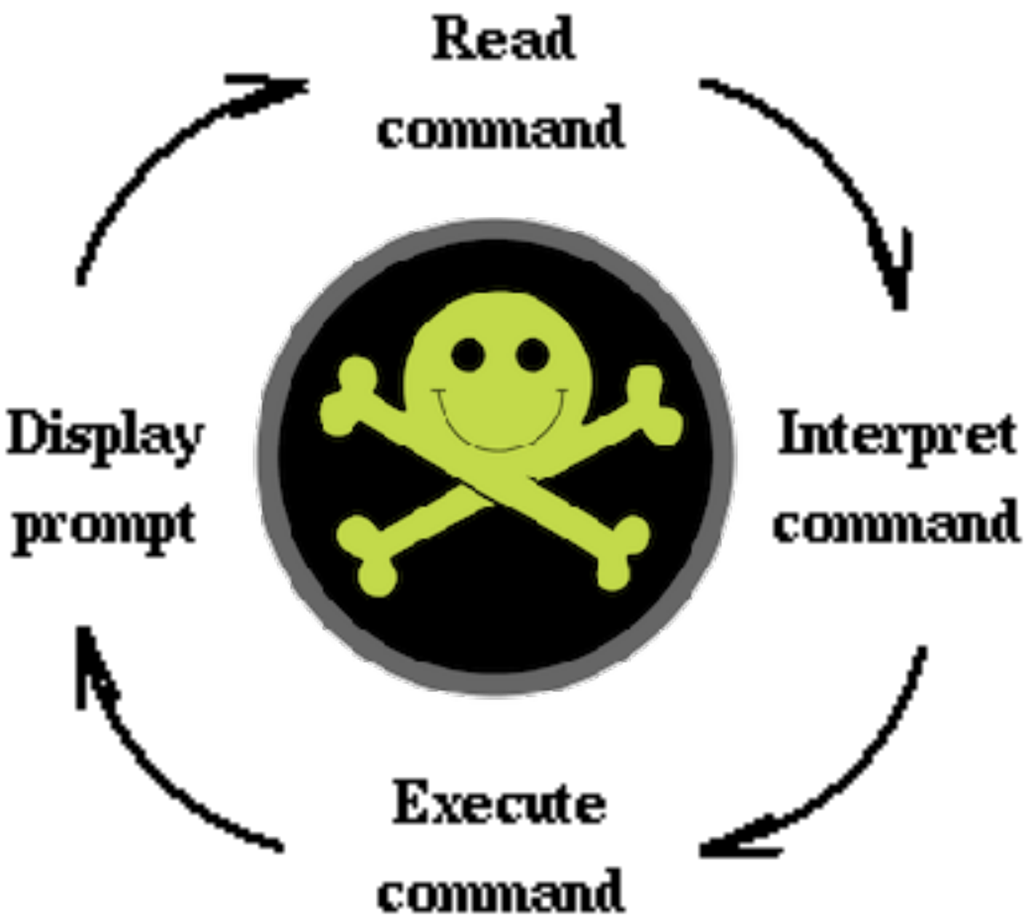
1 Virtual Black-Box 0

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1] - Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1] \le 2^{-\Omega(k)}$$

How about ONE generic program?

Virtualisation

i.e., Interpreters & specialisers



DETOUR

Interpretation

the function computed by x

Data

$$\mathbb{U}(x,y) = \begin{cases} \boxed{\varphi_x}(y) & \textbf{if } \boxed{\varphi_x}(y)\downarrow \\ \uparrow & \textbf{otherwise} \end{cases}$$

Code

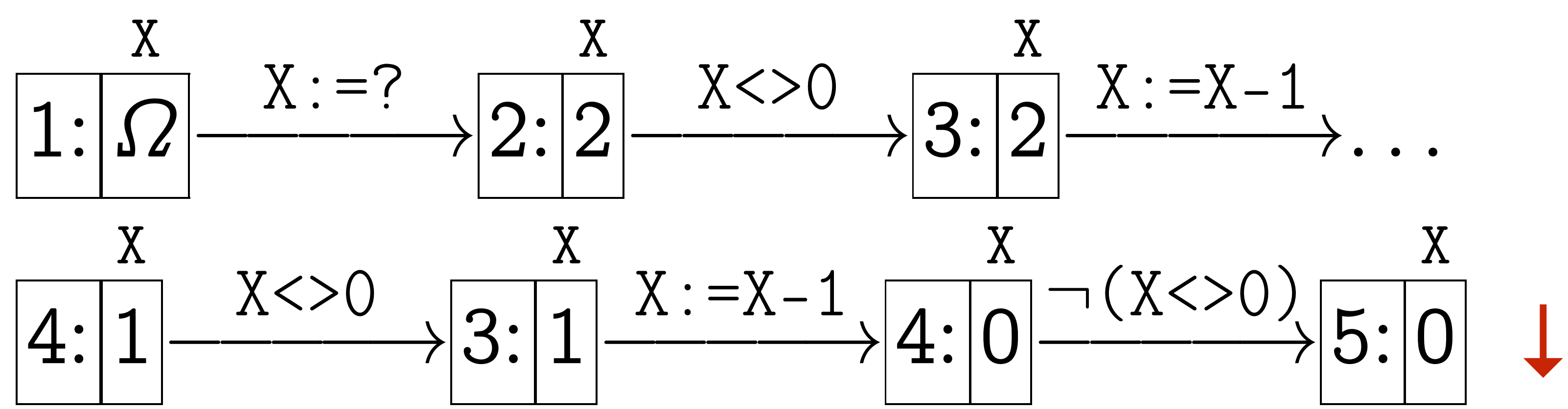
the Universal Turing Machine

$\mathbb{U} \in$ Code

...is the interpreter!!

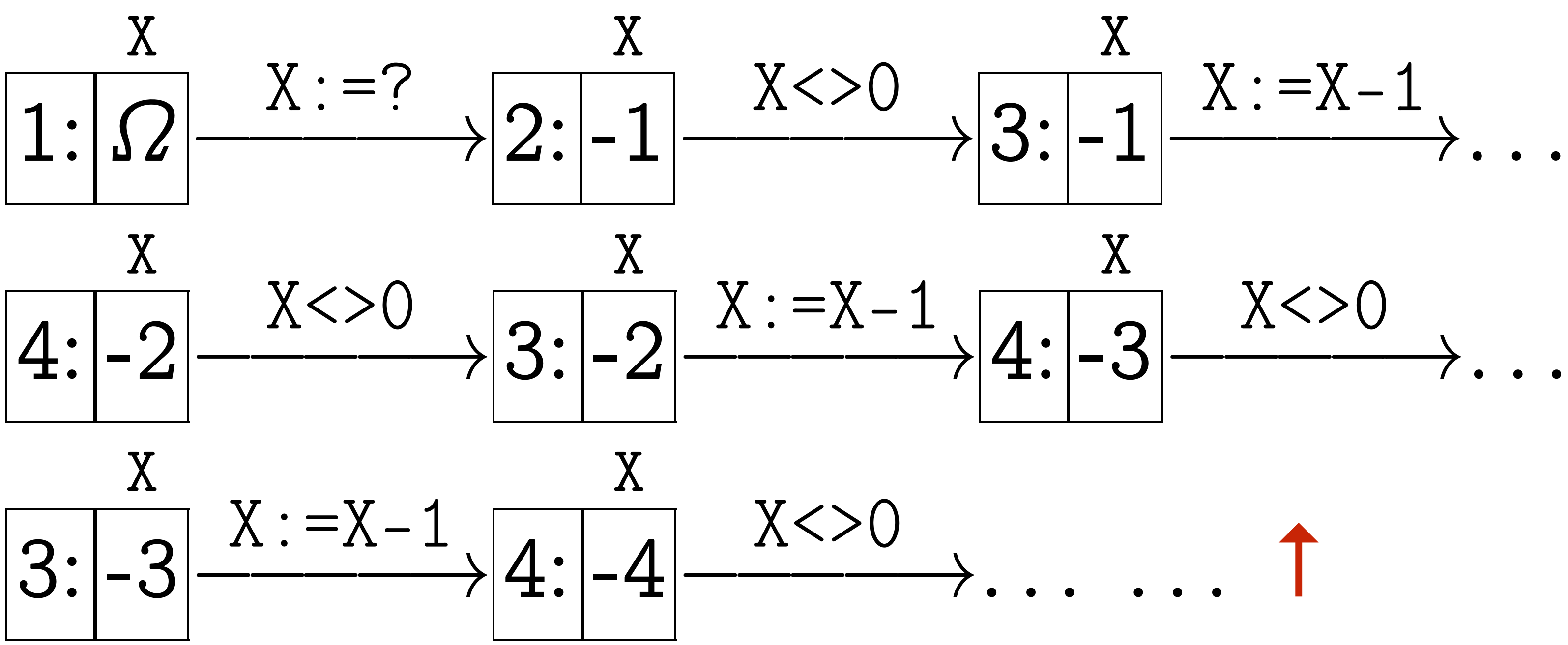
Interpretation

```
1:
   X:=?;
2:
   while (X>0) do
3:
      X:=X-1
4:
   od
5:
```



Interpretation

```
1:
    X:=?;
2:
    while (X>0) do
3:
        X:=X-1
4:
    od
5:
```



Interpretation

$$
\begin{array}{lcl}
\mathrm{Exp} & \rightarrow & x \mid d \mid \mathsf{cons}(\mathrm{Exp}_1, \mathrm{Exp}_2) \mid \mathsf{hd}(\mathrm{Exp}) \mid \mathsf{tl}(\mathrm{Exp}) \mid (\mathrm{Exp}_1 = \mathrm{Exp}_2) \\
\mathrm{Com} & \rightarrow & x := \mathrm{Exp} \mid \mathrm{Com}_1; \mathrm{Com}_2 \mid \mathsf{skip} \mid \mathbf{while}\ \mathrm{Exp}\ \mathbf{do}\ \mathrm{Com}\ \mathbf{endw} \\
\mathrm{Prog} & \rightarrow & \mathbf{read}(\mathrm{Listavar}); \mathrm{Com}; \mathbf{write}(\mathrm{Listavar}) \\
\mathrm{Listavar} & \rightarrow & x \mid x,\ \mathrm{Listavar}
\end{array}
$$

$$\mathcal{E}[\![x]\!]\sigma = \sigma(x) \qquad \mathcal{E}[\![d]\!]\sigma = d$$

$$\mathcal{E}[\![\mathsf{cons}(E_1, E_2)]\!]\sigma = (\mathcal{E}[\![E_1]\!]\sigma.\mathcal{E}[\![E_2]\!]\sigma) \qquad \mathcal{E}[\![E_1 = E_2]\!]\sigma = (\mathcal{E}[\![E_1]\!]\sigma = \mathcal{E}[\![E_2]\!]\sigma)$$

$$\mathcal{E}[\![\mathsf{tl}(E)]\!]\sigma = \begin{cases} c & \textbf{if } \mathcal{E}[\![E]\!]\sigma = (t.c) \\ \mathsf{nil} & \textbf{otherwise} \end{cases} \qquad \mathcal{E}[\![\mathsf{hd}(E)]\!]\sigma = \begin{cases} t & \textbf{if } \mathcal{E}[\![E]\!]\sigma = (t.c) \\ \mathsf{nil} & \textbf{otherwise} \end{cases}$$

Interpretation

$$\frac{\mathcal{E}[\![E]\!]\sigma = d}{\langle x := E, \sigma\rangle \longrightarrow \sigma[d/x]} \qquad \frac{}{\langle \mathsf{skip}, \sigma\rangle \longrightarrow \sigma}$$

$$\frac{\langle C_1, \sigma\rangle \longrightarrow \sigma'}{\langle C_1; C_2, \sigma\rangle \longrightarrow \langle C_2, \sigma'\rangle}$$

$$\frac{\mathcal{E}[\![E]\!]\sigma = \mathsf{nil}}{\langle \mathbf{while}\ \textsc{e}\ \mathbf{do}\ \textsc{c}\ \mathbf{endw}, \sigma\rangle \longrightarrow \sigma}$$

$$\frac{\mathcal{E}[\![E]\!]\sigma \neq \mathsf{nil}}{\langle \mathbf{while}\ \textsc{e}\ \mathbf{do}\ \textsc{c}\ \mathbf{endw}, \sigma\rangle \longrightarrow \langle \textsc{c}; \mathbf{while}\ \textsc{e}\ \mathbf{do}\ \textsc{c}\ \mathbf{endw}, \sigma\rangle}$$

Interpretation

DETOUR

$$\boxed{[\![\cdot]\!]^W : \text{Prog} \longrightarrow (\mathbb{D}_A^n \longrightarrow \mathbb{D}_A^m \cup \{\uparrow\})}$$

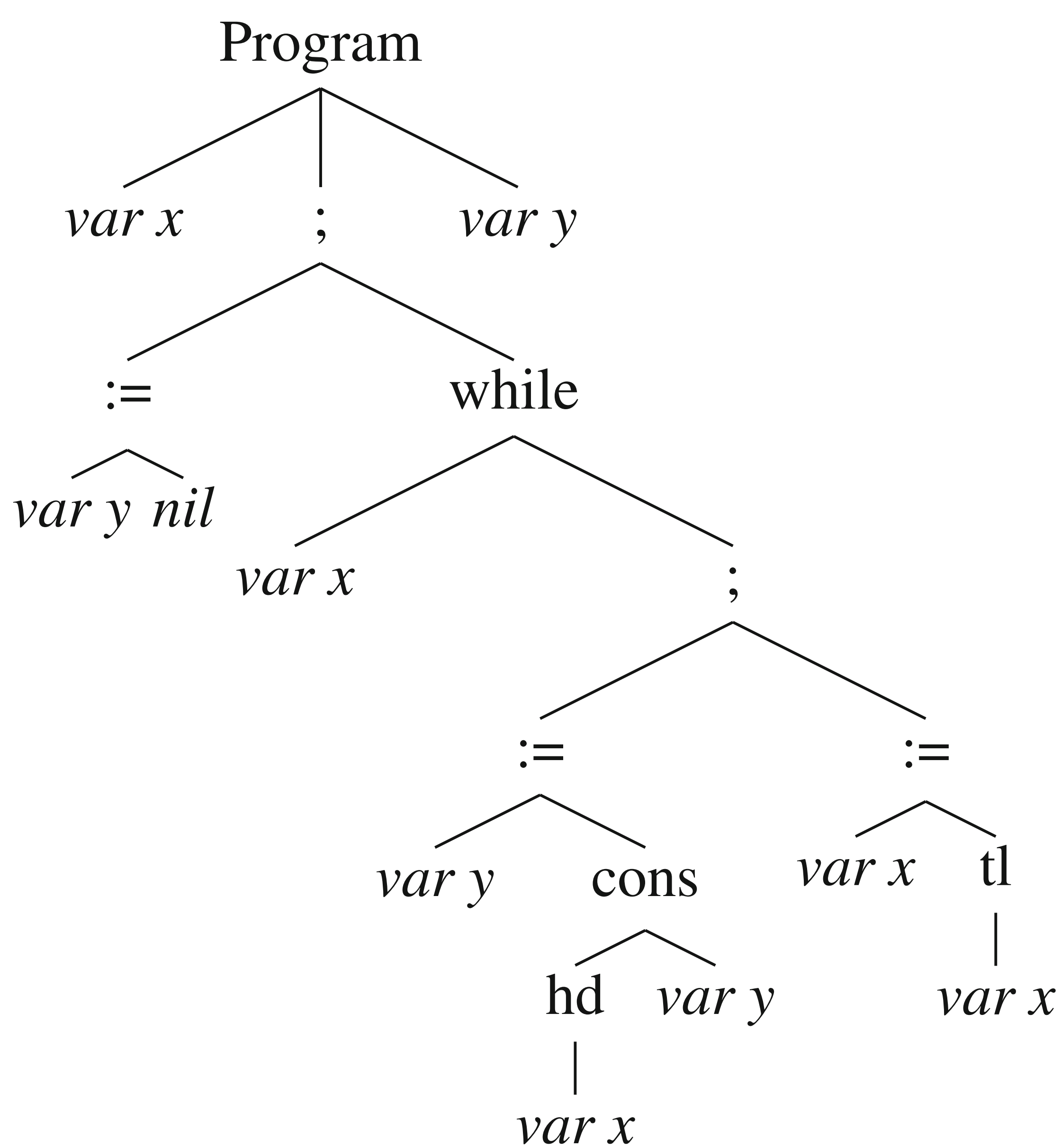
$$P = \textbf{read}(x_1, \ldots, x_n);\ C;\ \textbf{write}(y_1, \ldots, y_m)$$

$$[\![P]\!]^W(d_1, \ldots, d_n) = \begin{cases} e_1, \ldots, e_m & \textbf{if } \langle C, [d_1/x_1, \ldots, d_n/x_n]\rangle \longrightarrow^* \sigma' \\ & \quad \textbf{and } \sigma'(y_1) = e_1, \ldots, \sigma'(y_m) = e_m \\ \uparrow & \textbf{otherwise} \end{cases}$$

Interpretation

DETOUR

representing code as data!



AST

DETOUR

Interpretation

representing code as data!

$$
\begin{array}{rcl}
\underline{\mathbf{read}(v_i); C; \mathbf{write}(v_j)} & = & ((\mathrm{var}.\underline{i}).(\underline{C}.(\mathrm{var}.\underline{j}))) \\
\underline{C_1; C_2} & = & (;.(\underline{C_1}.\underline{C_2})) \\
\underline{\mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{endw}} & = & (\mathbf{while}(\underline{E}.\underline{C})) \\
\underline{v_i := E} & = & (:=.((\mathrm{var}.\underline{i}).\underline{E})) \\
\underline{v_i} & = & (\mathrm{var}.\underline{i}) \\
\underline{d} & = & (\mathrm{quote}.d) \\
\underline{\mathrm{cons}(E_1, E_2)} & = & (\mathrm{cons}.(\underline{E_1}.\underline{E_2})) \\
\underline{\mathrm{hd}(E)} & = & (\mathrm{hd}.\underline{E}) \\
\underline{\mathrm{tl}(E)} & = & (\mathrm{tl}.\underline{E}) \\
\underline{(E_1 = E_2)} & = & (=.(\underline{E_1}.\underline{E_2}))
\end{array}
$$

Interpretation

representing code as data!

`reverse read X {`
`  Y:= nil;`
`  while X {`
`    Y:= cons hd X Y;`
`    X:= tl X`
`    }`
`}`
`write Y`
	`[0,`
	` [[:=,1,[quote,nil]],`
	`  [while,[var,0],`
	`      [ [:=,1,[cons,[hd,[var,0]],[var,1]]],`
	`        [:=,0,[tl,[var,0]]]`
	`      ]`
	` ]],`
	` 1]`

DETOUR (Self) Interpretation

$$\mathbb{U}(x,y) = \begin{cases} \varphi_x(y) & \textbf{if } \varphi_x(y)\downarrow \\ \uparrow & \textbf{otherwise} \end{cases}$$

```
read PD;                                  (* Input (p.d)                  *)
 Pgm := hd PD;                            (* p = ((var i) c (var j))      *)
 D   := tl PD;                            (* D = d  (input value)         *)
 I   := hd (tl (hd Pgm))                  (* I = i  (input variable)      *)
 J   := hd (tl (hd (tl (tl Pgm))));       (* J = j  (output variable)     *)
 C   := hd (tl Pgm))                      (* C = c, program code          *)
 Vl  := update I D nil     (* (var i) initially d, others nil             *)
 Cd  := cons C nil;        (* Cd = (c.nil), Code to execute is c          *)
 St  := nil;               (* St = nil, computation Stack empty           *)
 while Cd do STEP;         (* do while there is code to execute           *)
 Out := lookup J Vl        (* Output is the value of (var j)              *)
write Out;
```

DETOUR (Self) Interpretation

Code Stack

Value Stack

STEP

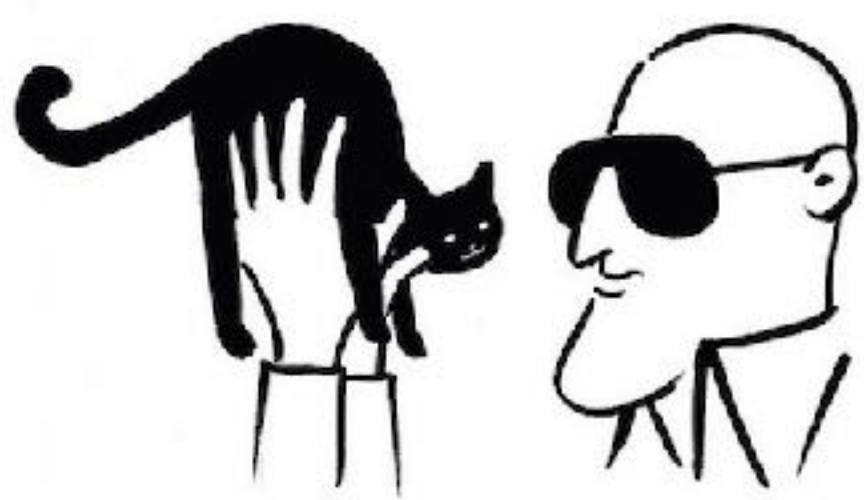
```
rewrite [Cd, St] by
[((quote D).Cr),              St  ]  ⇒ [Cr,            cons D St]
[((var 1).Cr),                St  ]  ⇒ [Cr,           cons Vl St]

[((hd E).Cr),                 St  ]  ⇒ [cons* E dohd Cr,       St]
[(dohd.Cr),               (T.Sr)]  ⇒ [Cr,         cons (hd T) Sr]

[((tl E).Cr),                 St  ]  ⇒ [cons* E dotl Cr,       St]
[(dotl.Cr),               (T.Sr)]  ⇒ [Cr,         cons (tl T)  Sr]

[((cons E1 E2).Cr),          St  ]  ⇒ [cons* E1 E2 docons Cr, St]
[(docons.Cr),          (U.(T.Sr)) ]  ⇒ [Cr,  cons (cons T U) Sr]

[((=? E1 E2).Cr),             St  ]  ⇒ [cons* E1 E2 do=? Cr, St]
[(do=?.Cr),            (U.(T.Sr)) ]  ⇒ [Cr,      cons (=? T U)  Sr]

[((; C1 C2).Cr),              St  ]  ⇒ [cons* C1 C2 Cr,        St]

[((:= (var 1) E).Cr),         St  ]  ⇒ [cons* E doasgn Cr,     St]
[(doasgn.Cr),             (W.Sr) ] ⇒{Cd := Cr; St  := Sr; Vl:= W;}

[((while E C).Cr),       St ]  ⇒  [cons* E dowh (while E C) Cr, St]

[(dowh.((while E C).Cr)), (nil.Sr)]  ⇒  [Cr,                  Sr]

[(dowh.((while E C).Cr)),((D.E).S)]⇒[cons* C (while E C) Cr, S]

[nil, St]                            ⇒  [nil, St]
```

© Giacobazzi

DETOUR

Specialisation

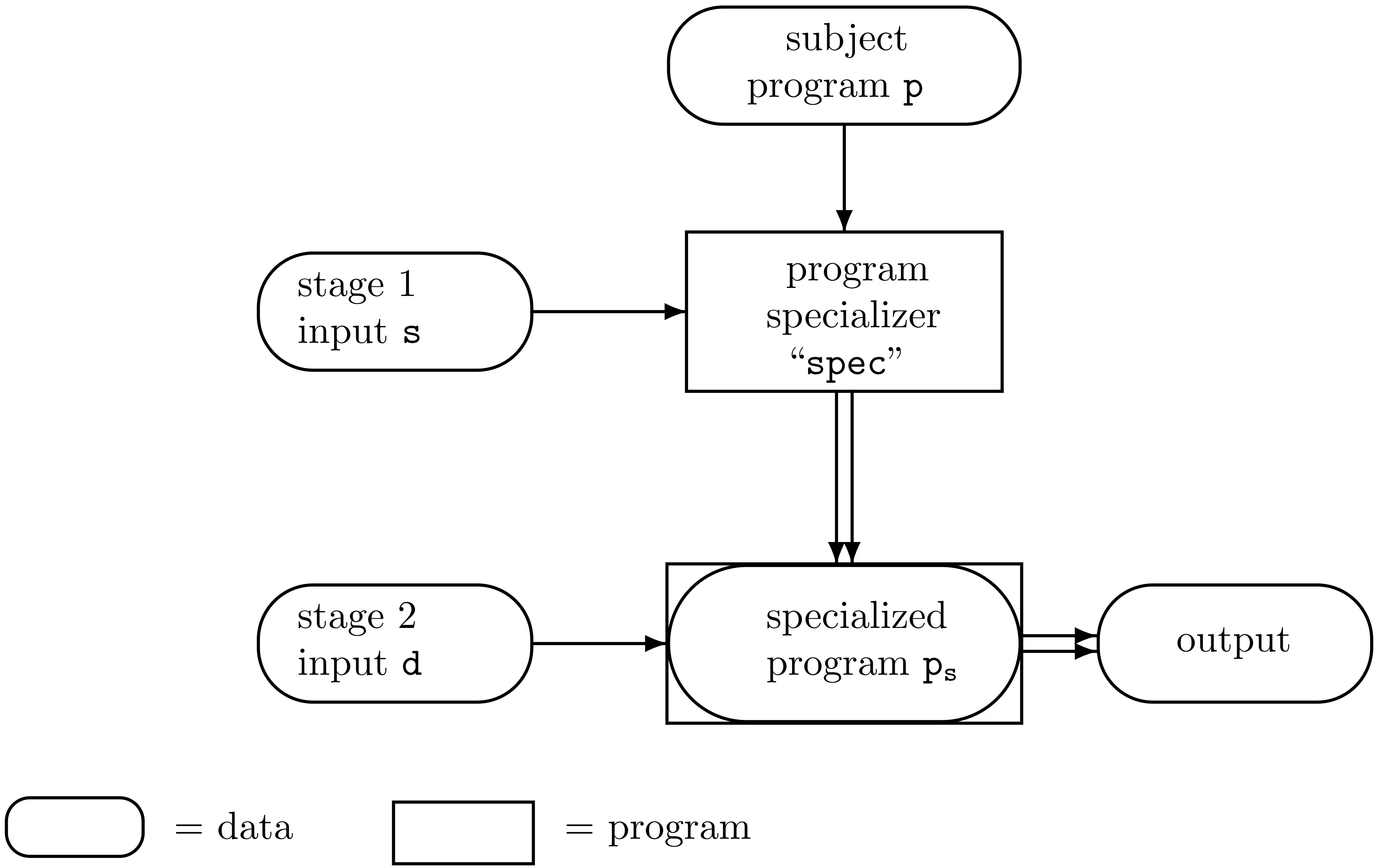
Input Code



Residual Code

Can we generate code?

Specialisation



$$[\![\,[\![\mathtt{spec}]\!]\,(\mathtt{p}.\mathtt{s})\,]\!]\,(\mathtt{d}) = [\![\mathtt{p}]\!]\,(\mathtt{s}.\mathtt{d})$$

Specialisation

DETOUR

```
spec read PS  {            (* input list of form [P, S] *)
    P := hd PS;            (* P is program  [X,B,Y] *)
    S := hd tl PS;         (* S is input *)
    X := hd P;             (* X is input var of P *)
    B := hd tl P;          (* B is statement block of program P *)
    Y := hd tl tl P;       (* Y is output var of P *)
    expr:= [cons,[quote, S],[cons,[var, X],[quote,nil]]]
    newAsg := [:=, X, expr];
                     (* assemble asgmnt: "X := [S, X]" *)
    newB := cons newAsg B;
                     (* add assignment to old code *)
    newProg := [X, newB, Y]
                     (* assemble new program *)
}
write newProg
```

Specialisation

```
P: f(n,x) = if n =? 0 then 1
                      else x * f(n-1,x)
```

$$[\![\mathtt{spec}]\!](\mathtt{P}, 5) = \mathtt{x} * (\mathtt{x} * (\mathtt{x} * (\mathtt{x} * (\mathtt{x} * 1))))$$

No function call and no test $\Rightarrow$ runs faster!

Partial evaluation: a form of specialization that optimizes program `P` with respect to s.

- Analysis: What depends only on s?
- Specialization: Simplify what depends only on s and construct a $\mathcal{T}$-program for the rest.

Specialisation

Specialization of a program to compute Ackermann's function:

```
a(m,n) = if m =? 0 then n+1 else
         if n =? 0 then a(m-1,1)
           else a(m-1,a(m,n-1))
```

$m = 2$ but unknown n:

```
a2(n) = if n =? 0 then 3 else a1(a2(n-1))
a1(n) = if n =? 0 then 2 else a1(n-1)+1
```

Specialisation

Specialization of a program to compute the power $f(n,x) = x^n$:

```
f(n,x) =
  if n=0 then 1
         else if odd(n) then x*f(n-1,x)
                        else f(n/2)**2
```

$n = 13$ but unknown x:

```
f_13(x) = x*((x*(x**2))**2)**2
```

We need Binding-time Analysis (BTA)

Specialisation

- Static: compute at specialization time
- Dynamic: generate code to compute at run time

```
power =
f(n,x) =
if n=0            then 1
else if odd(n) then x*f(n-1,x)
else f(n/2,x)**2
```

We know `n` and we will not know `x`

Specialisation

- Static: compute at specialization time
- Dynamic: generate code to compute at run time

```
power =
        f(n,x) =
        if n=0              then 1
        else if odd(n) then x*f(n-1,x)
        else f(n/2,x)**2
```

If we know `n` we can decide if it is zero

Specialisation

DETOUR

- Static: compute at specialization time
- Dynamic: generate code to compute at run time

```
power =
        f(n,x) =
        if n=0          then 1
        else if odd(n) then x*f(n-1,x)
        else f(n/2,x)**2
```

.... and if it is odd

Specialisation

- Static: compute at specialization time
- Dynamic: generate code to compute at run time

```
power =
  f(n,x) =
  if n=0             then 1
  else if odd(n) then x*f(n-1,x)
  else f(n/2,x)**2
```

.... and we can compute $n - 1$ or $n/2$

Specialisation

- Static: compute at specialization time
- Dynamic: generate code to compute at run time

```
power =
        f(n,x) =
        if n=0            then 1
        else if odd(n) then x*f(n-1,x)
        else f(n/2,x)**2
```

.... and unfold function calls being n bounded below!

DETOUR Futamura Projections 1971

First Projection

$$\begin{aligned}\mathtt{out} &= [\![\mathtt{source}]\!]^{\mathtt{S}}(\mathtt{in})\\ &= [\![\mathtt{int}]\!]^{\mathtt{L}}(\mathtt{source.in})\\ &= [\![[\![\mathtt{spec}]\!]^{\mathtt{Imp}}(\mathtt{int.source})]\!]^{\mathtt{T}}(\mathtt{in})\\ &= [\![\mathtt{target}]\!]^{\mathtt{T}}(\mathtt{in})\end{aligned}$$

$$\mathtt{target} = [\![\mathtt{spec}]\!]^{\mathtt{Imp}}(\mathtt{int.source})$$

DETOUR Futamura Projections 1971

Second Projection

$$\begin{aligned}
\mathtt{target} &= [\![\mathtt{spec}]\!]^{\mathtt{L}}(\mathtt{int.source}) \\
&= [\![[\![\mathtt{spec}]\!]^{\mathtt{L}}(\mathtt{spec.int})]\!]^{\mathtt{T}}(\mathtt{source}) \\
&= [\![\mathtt{compiler}]\!]^{\mathtt{T}}(\mathtt{source})
\end{aligned}$$

$$\mathtt{compiler} = [\![\mathtt{spec}]\!]^{\mathtt{L}}(\mathtt{spec.int})$$

DETOUR Futamura Projections 1971

Third Projection

$$\begin{aligned}\texttt{compiler} &= [\![\texttt{spec}]\!]^{\mathtt{L}}(\texttt{spec.int}) \\ &= [\![[\![\texttt{spec}]\!]^{\mathtt{L}}(\texttt{spec.spec})]\!]^{\mathtt{T}}(\texttt{int}) \\ &= [\![\texttt{cogen}]\!]^{\mathtt{T}}(\texttt{int})\end{aligned}$$

$$\texttt{cogen} = [\![\texttt{spec}]\!]^{\mathtt{L}}(\texttt{spec.spec})$$

DETOUR Futamura Projections 1971

$$
\begin{array}{llll}
\texttt{out} & = & [\![\texttt{int}]\!](\texttt{source.input}) & = & [\![\texttt{target}]\!](\texttt{input}) \\
\texttt{target} & = & [\![\texttt{spec}]\!](\texttt{int.source}) & = & [\![\texttt{compiler}]\!](\texttt{source}) \\
\texttt{compiler} & = & [\![\texttt{spec}]\!](\texttt{spec.int}) & = & [\![\texttt{cogen}]\!](\texttt{int}) \\
\texttt{cogen} & = & [\![\texttt{spec}]\!](\texttt{spec.spec}) & = & [\![\texttt{cogen}]\!](\texttt{spec})
\end{array}
$$

Compilation

DETOUR

A simple programming language:

```
;; A NORMA program works on two registers, x and y,
;; each holding a number ( n = list of n 1's )
;; INITIALLY x = input, y = 0.
;; AT END: output is y's final value.
;;
;; Norma syntax: (only 7 instructions)
;;
;; pgm ::= ( instr* )
;; instr ::= X:=X+1 | X:=X-1 | Y:=Y+1 | Y:=Y-1
;; | ifX=0goto addr) | ifY=0goto addr
;; | goto addr
;; addr ::= 1*
```

Still a Turing-complete language!!!

Compilation

```
;; Data: a NORMA program. It computes 2 * x + 2.
0: (Y:=Y+1 ;
1:  Y:=Y+1 ;
2:  ifX=0goto 1 1 1 1 1 1 1 ;
3:  Y:=Y+1 ;
4:  Y:=Y+1 ;
5:  X:=X-1 ;
6:  goto 1 1 ;
7: )
```

$$[\![P]\!](2) = 6$$

DETOUR

Compilation

Functional interpreter

```
execute(pgm,x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) =  case head(rest) of
"X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
"goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
   if x ≠ 0 then run(tail(rest), pgmcopy, x, y)
   else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = - find suffix of pgm starting with
                   the lth instruction -
```

Compilation

DETOUR

```
execute(pgm,x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) =  case head(rest) of
"X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
"goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
   if x ≠ 0 then run(tail(rest), pgmcopy, x, y)
   else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = - find suffix of pgm starting with
                   the lth instruction -
```

Compilation

DETOUR

```
execute(pgm, x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) =  case head(rest) of
"X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
"goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
   if x ≠ 0 then run(tail(rest), pgmcopy, x, y)
   else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = - find suffix of pgm starting with
                   the lth instruction -
```

Compilation

```
execute(pgm, x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) = case head(rest) of
"X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
"goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
    if x ≠ 0 then run(tail(rest), pgmcopy, x, y)
    else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = - find suffix of pgm starting with
                   the lth instruction -
```

DETOUR

Compilation

```
execute(pgm, x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) =  case head(rest) of
"X:=X+1"  : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1"  : run(tail(rest), pgmcopy, x-1, y)
"goto l"  : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
   if x ≠ 0 then run(tail(rest), pgmcopy, x, y)
   else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = - find suffix of pgm starting with
                   the lth instruction -
```

Compilation

DETOUR

```
execute(pgm, x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) = case head(rest) of
"X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
"goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
    if x ≠ 0 then run(tail(rest), pgmcopy, x, y)
    else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = - find suffix of pgm starting with
                   the lth instruction -
```

Compilation

DETOUR

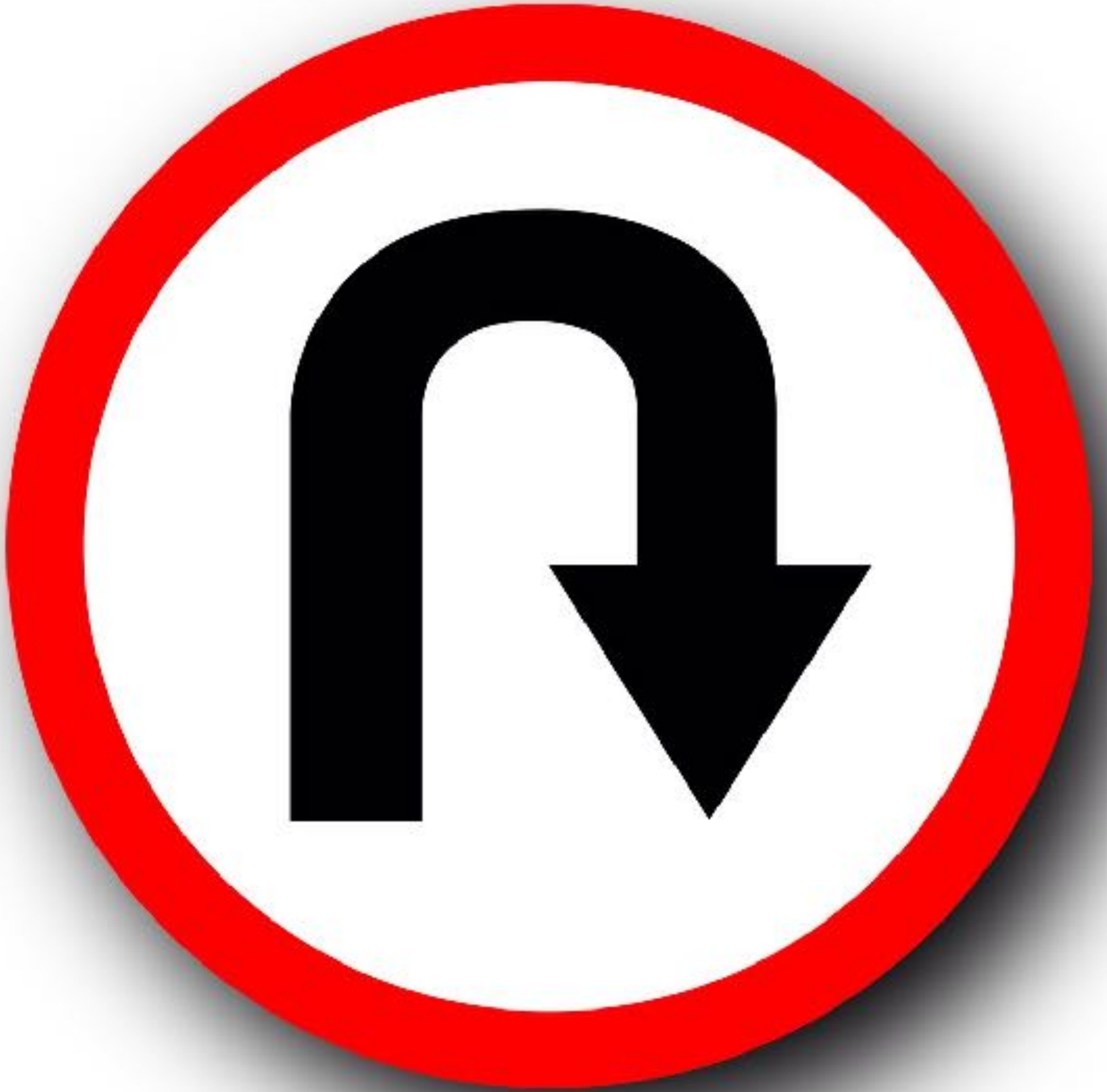
source =

```
0: (Y:=Y+1 ;
1:  Y:=Y+1 ;
2:  ifX=0goto 1 1 1 1 1 1 1 ;
3:  Y:=Y+1 ;
4:  Y:=Y+1 ;
5:  X:=X-1 ;
6:  goto 1 1 ;
7: )
```

target =

```
execute(x) = run0(x, 0)
run0(x, y) = run2(x, y+1 +1)
run2(x, y) = if x=0 then run5(x, y)
                     else run3(x,y)
run3(x, y) = run0(x, y)
run5(x, y) = y
```

...still computes $2 \cdot x + 2$ but it is a functional program!!



Back to the Proof for 2TMs (C & D)

$\alpha, \beta \in \{0,1\}^k$ Secrets!!

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(X) = \begin{cases} 1 & \text{if } X \equiv C_{\alpha,\beta} \\ 0 & \text{otherwise} \end{cases}$$

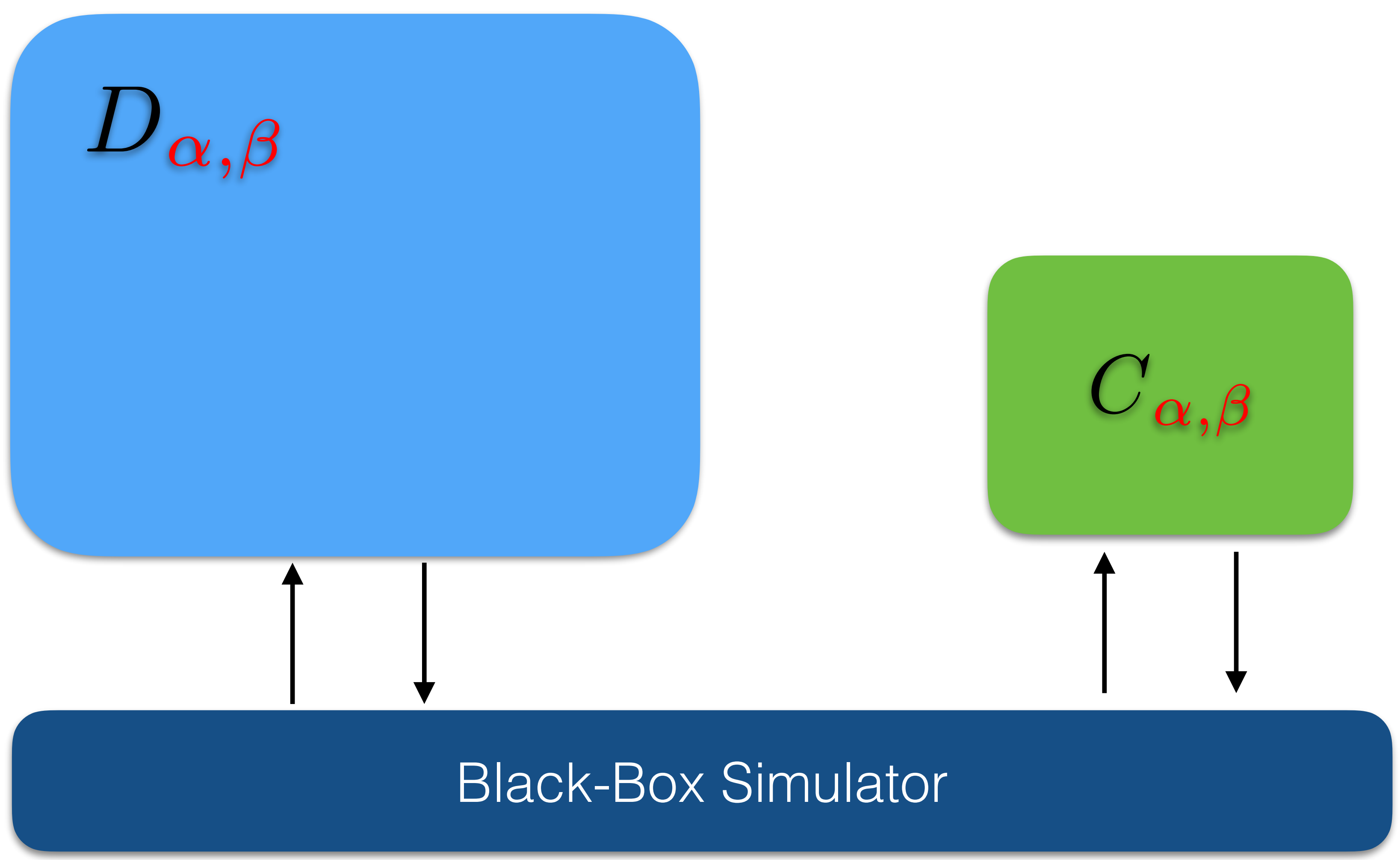
Distinguish if X computes $C_{\alpha,\beta}$ from $C_{\alpha',\beta'}$ for any $(\alpha,\beta)\neq(\alpha',\beta')$ is NON COMPUTABLE!

$$Z_k(x) = 0^k$$

→ Simply compute $X(\alpha)$ for Poly(k) steps and check!

Idea: It is difficult distinguish $(C_{\alpha,\beta}, D_{\alpha,\beta})$ from $(Z_k, D_{\alpha,\beta})$ by VBB access to these programs!!

Idea: Proof for 2TMs (C,D)

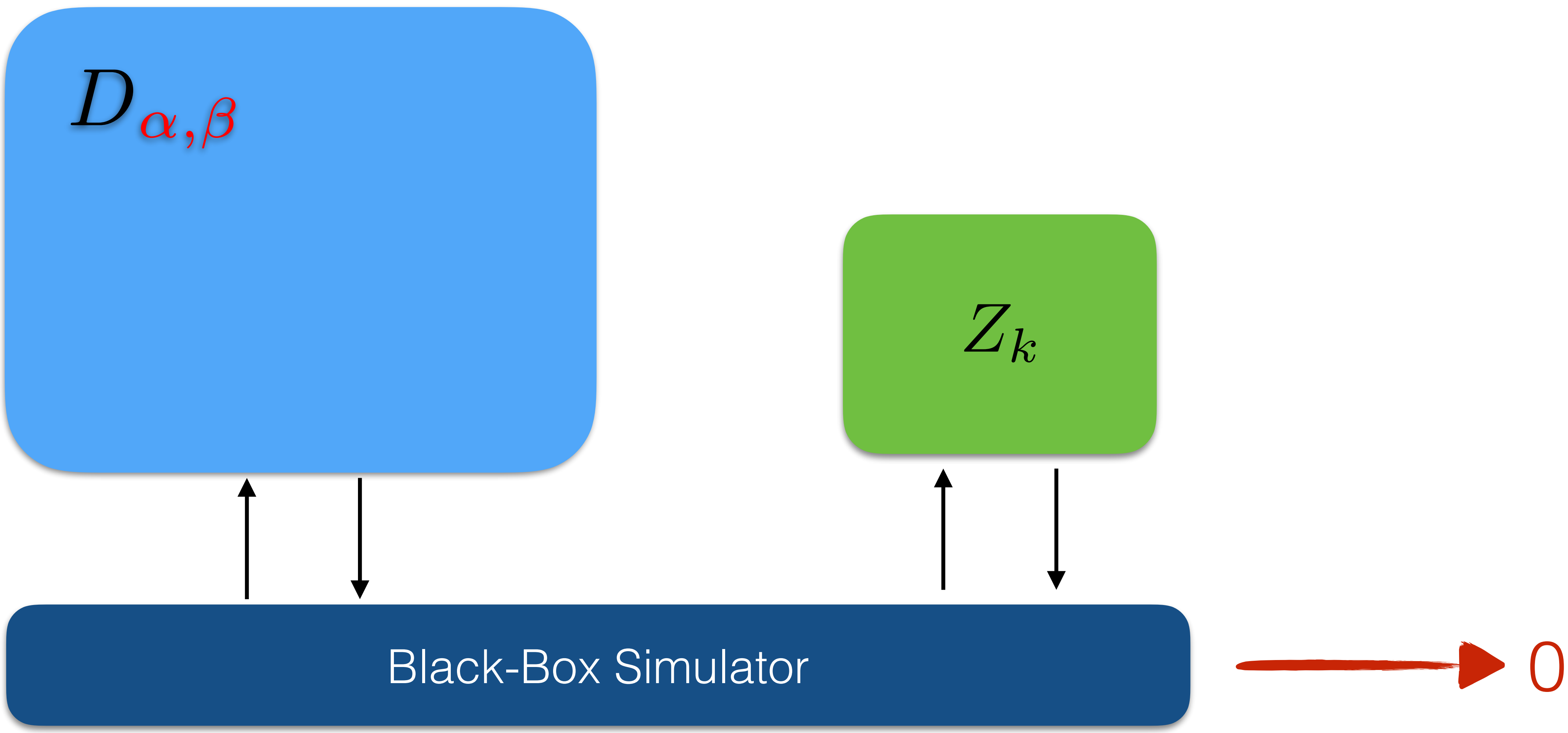


Virtual Black-Box

1

$$\boxed{Pr[A(\mathcal{O}(C_{\alpha,\beta}), \mathcal{O}(D_{\alpha,\beta})) = 1]} - Pr[S^{C_{\alpha,\beta}, D_{\alpha,\beta}}(1^k) = 1] \leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)

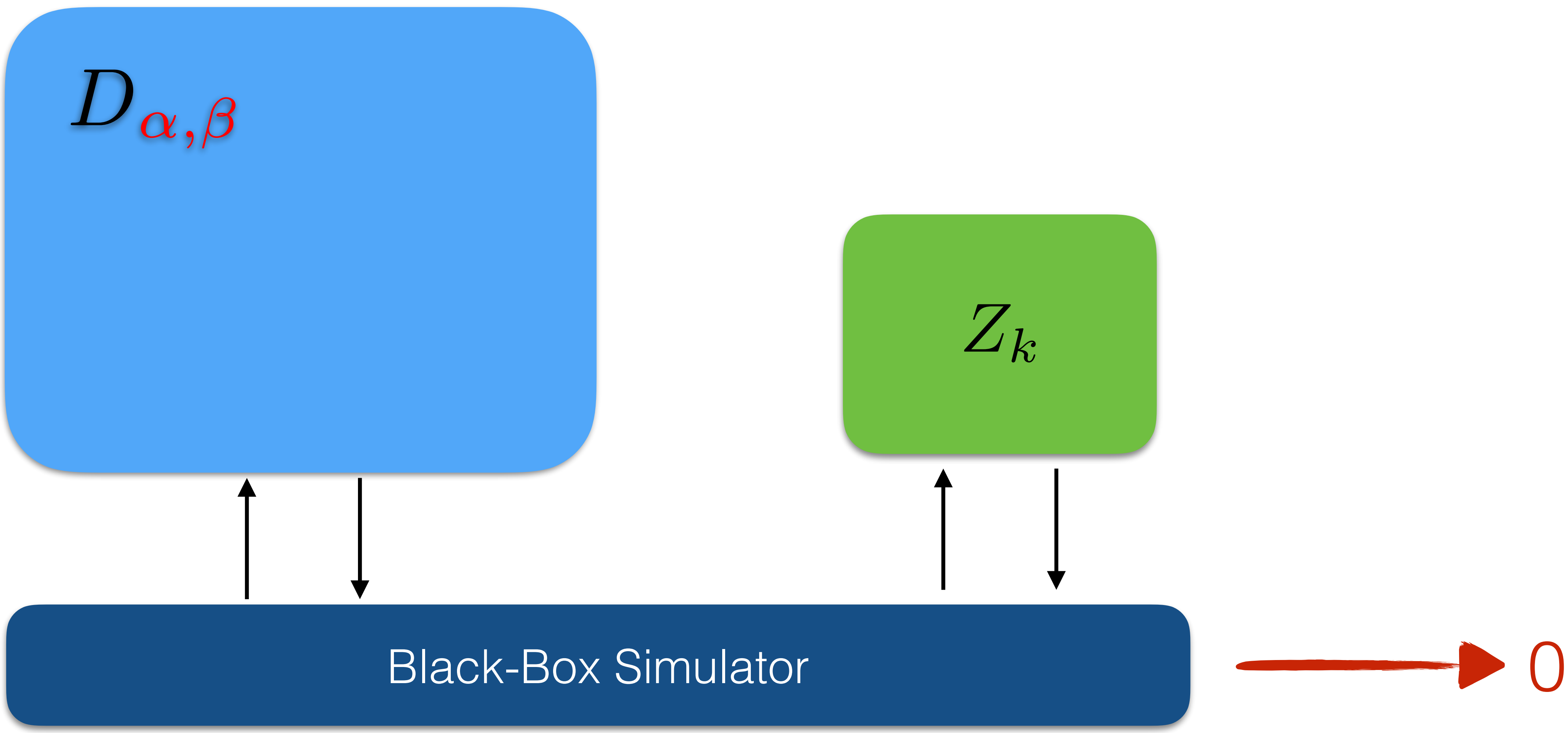


Virtual Black-Box

1

$$Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1] - Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1] \leq 2^{-\Omega(k)}$$

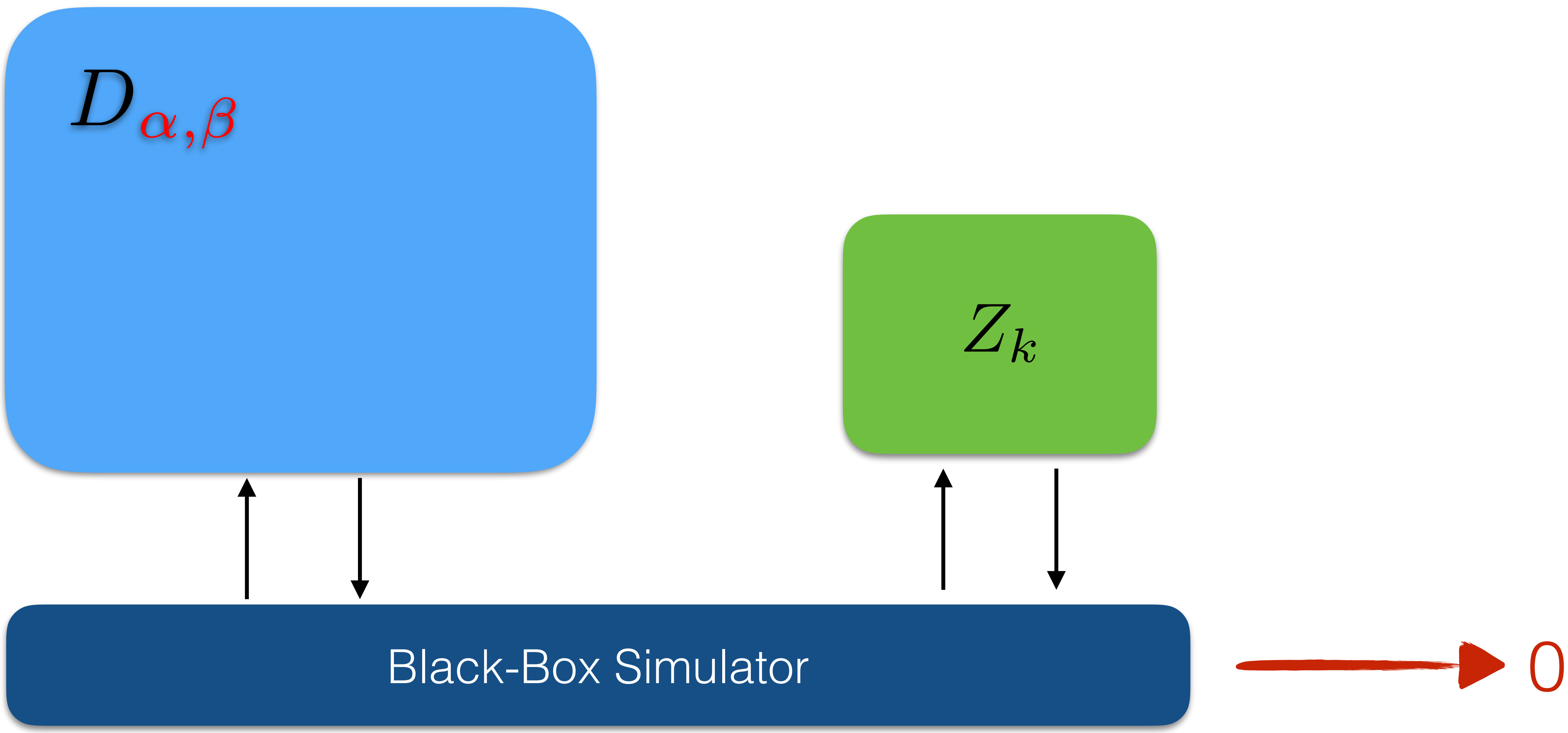
Idea: Proof for 2TMs (C,D)



Virtual Black-Box

$$\overset{1}{\boxed{Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]}} - \overset{0}{\boxed{Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]}} \leq 2^{-\Omega(k)}$$

Idea: Proof for 2TMs (C,D)



Virtual Black-Box

$$\overset{1}{Pr[A(\mathcal{O}(C_{\alpha,\beta}),\mathcal{O}(D_{\alpha,\beta}))=1]} - \overset{0}{Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]} \leq 2^{-\Omega(k)}$$

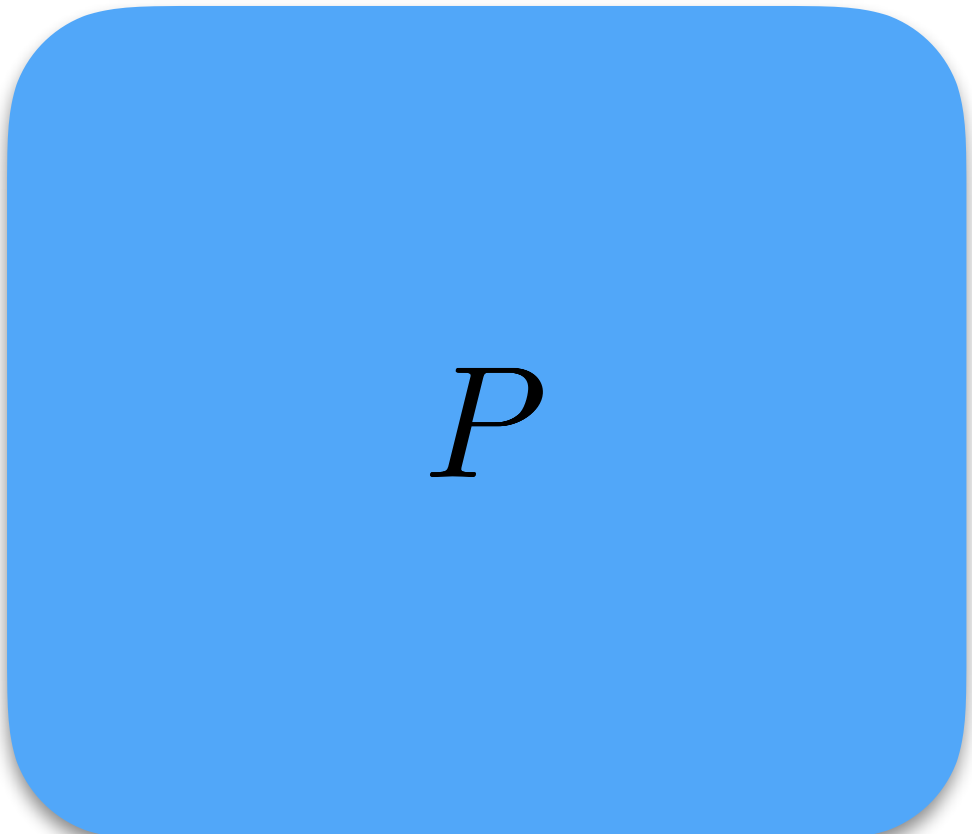
The Proof for one generic TM

P

Any program P is the partial evaluation of an interpreter Int wrt a residual program Q

$$\texttt{target} = [\![\texttt{spec}]\!]^{\texttt{Imp}}(\texttt{int.source})$$

The Proof for one generic TM



Any program P *is the partial evaluation of an interpreter* Int *wrt a residual program* Q

$$\texttt{target} = [\![\texttt{spec}]\!]^{\texttt{Imp}}(\texttt{int.source})$$

The Proof for one generic TM

Any program P is the partial evaluation of an interpreter Int wrt a residual program Q

$$\texttt{target} = [\![\texttt{spec}]\!]^{\texttt{Imp}}(\texttt{int.source})$$

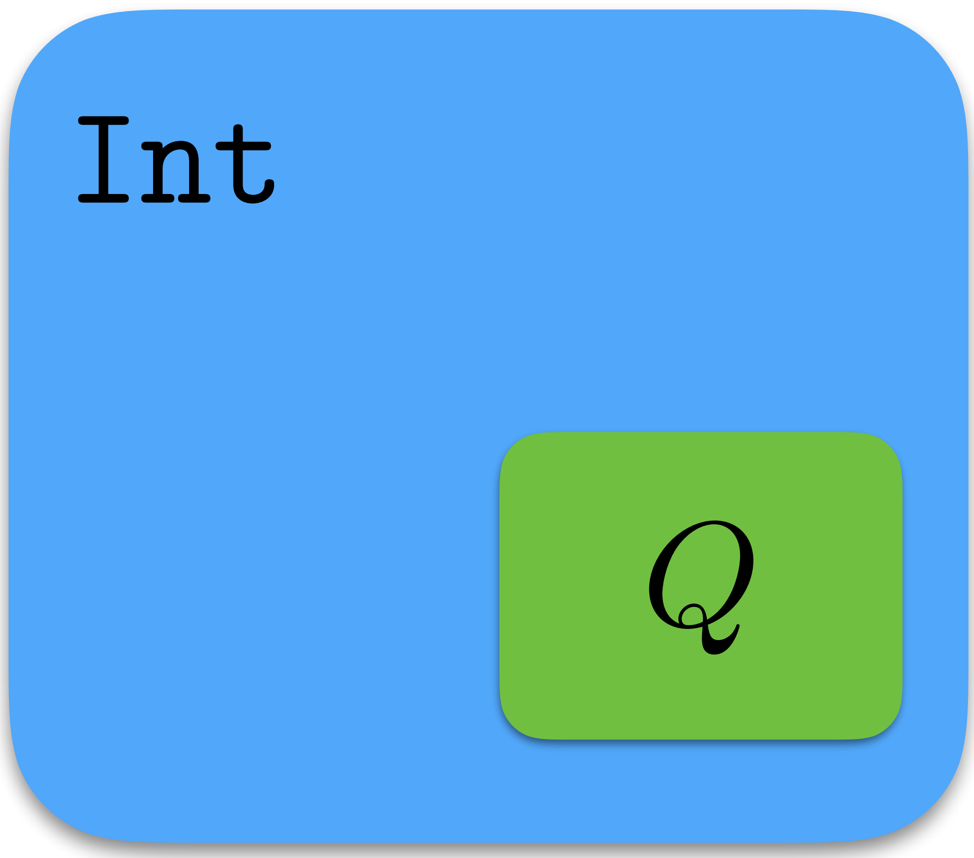
The Proof for one generic TM

`Int`

Any program P is the partial evaluation of an interpreter Int wrt a residual program Q

$$\mathtt{target} = [\![\mathtt{spec}]\!]^{\mathtt{Imp}}(\mathtt{int.source})$$

The Proof for one generic TM



Any program P is the partial evaluation of an interpreter Int wrt a residual program Q

$$\mathtt{target} = [\![\mathtt{spec}]\!]^{\mathtt{Imp}}(\mathtt{int.source})$$

The Proof for one generic TM

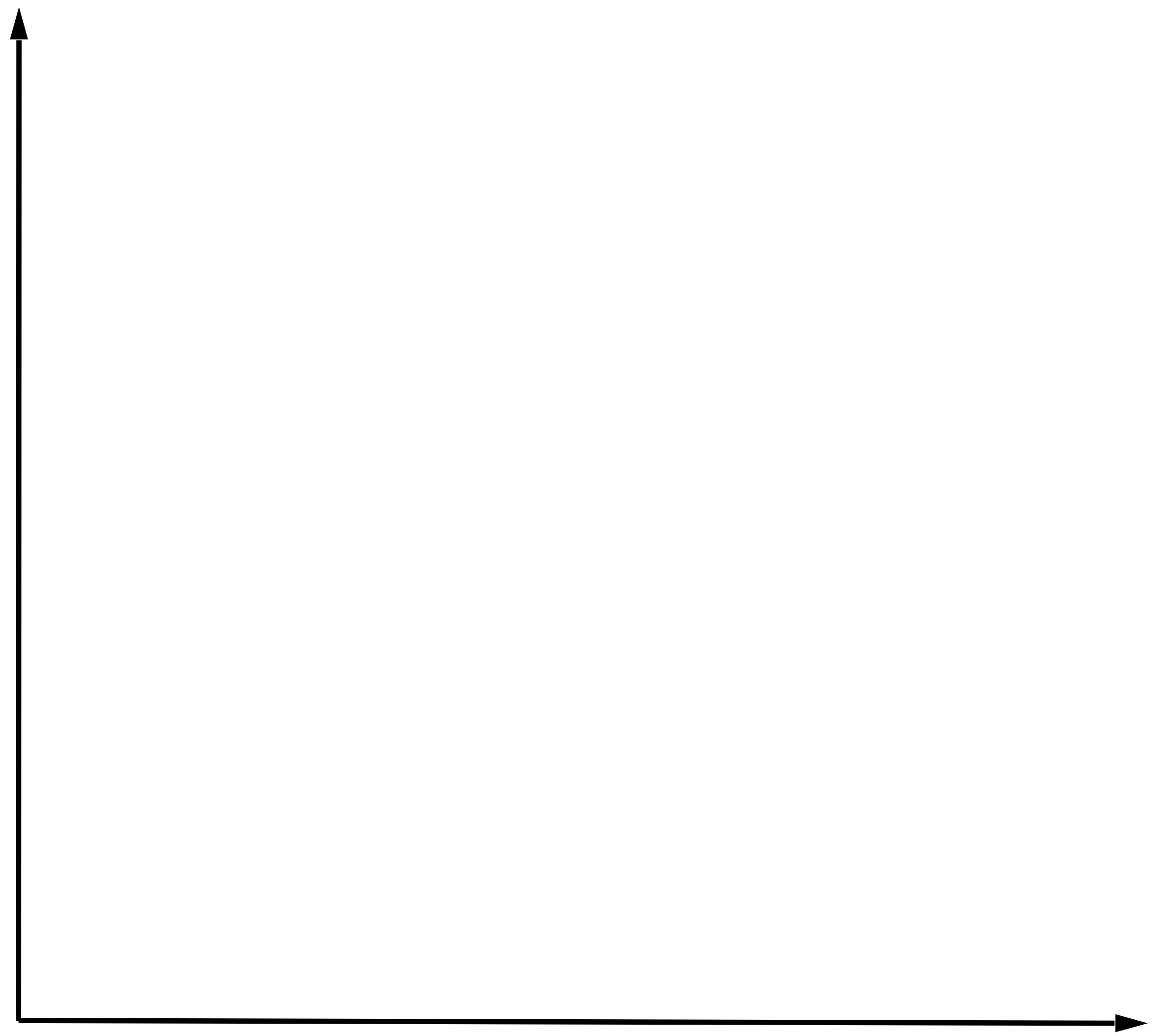
Any program P is the partial evaluation of an interpreter Int wrt a residual program Q

$$\texttt{target} = [\![\texttt{spec}]\!]^{\texttt{Imp}}(\texttt{int.source})$$

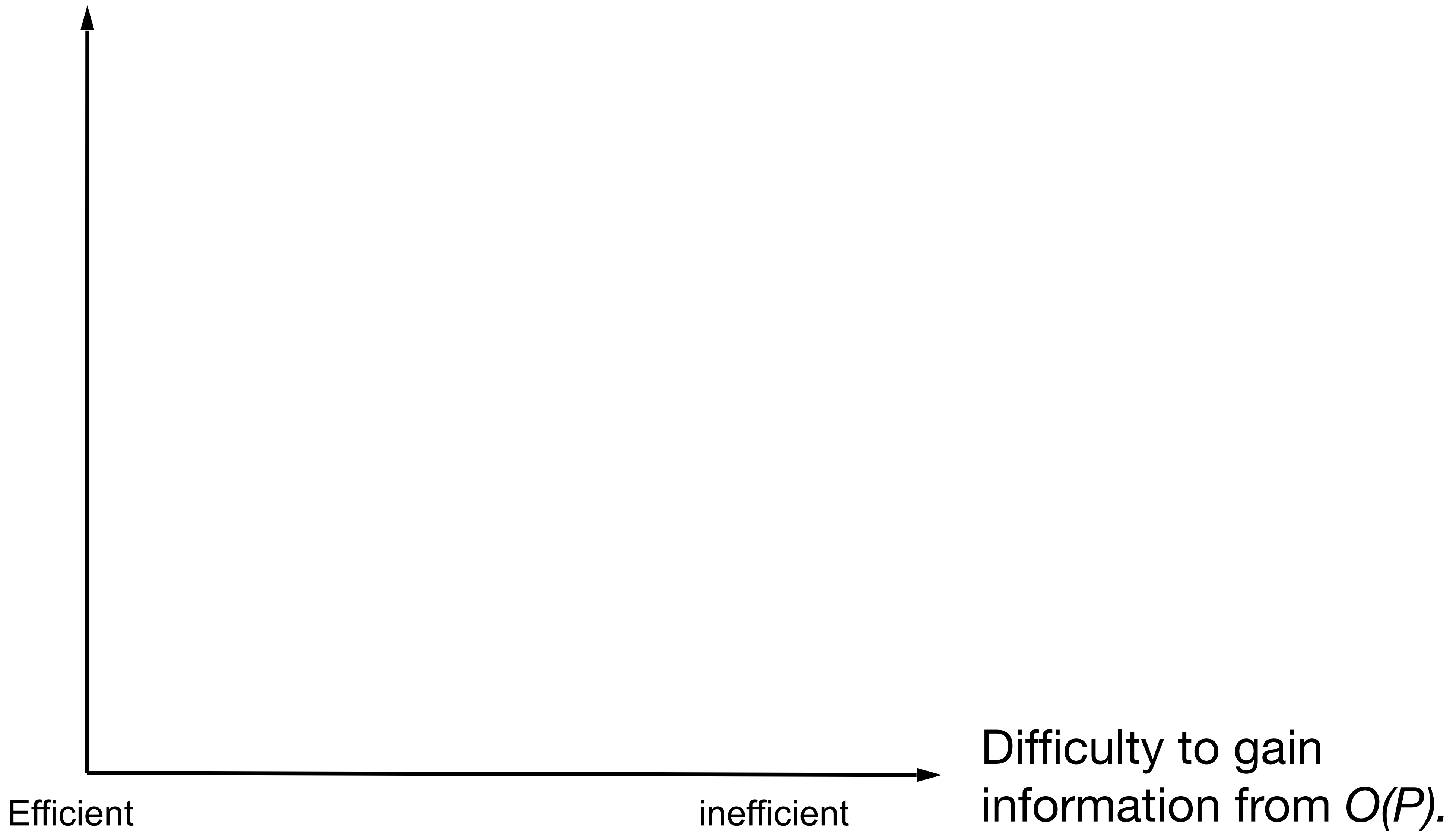
Whenever we disclose the code we always disclose more that its input/output relation!!

The notion of interpretation is fundamental here!

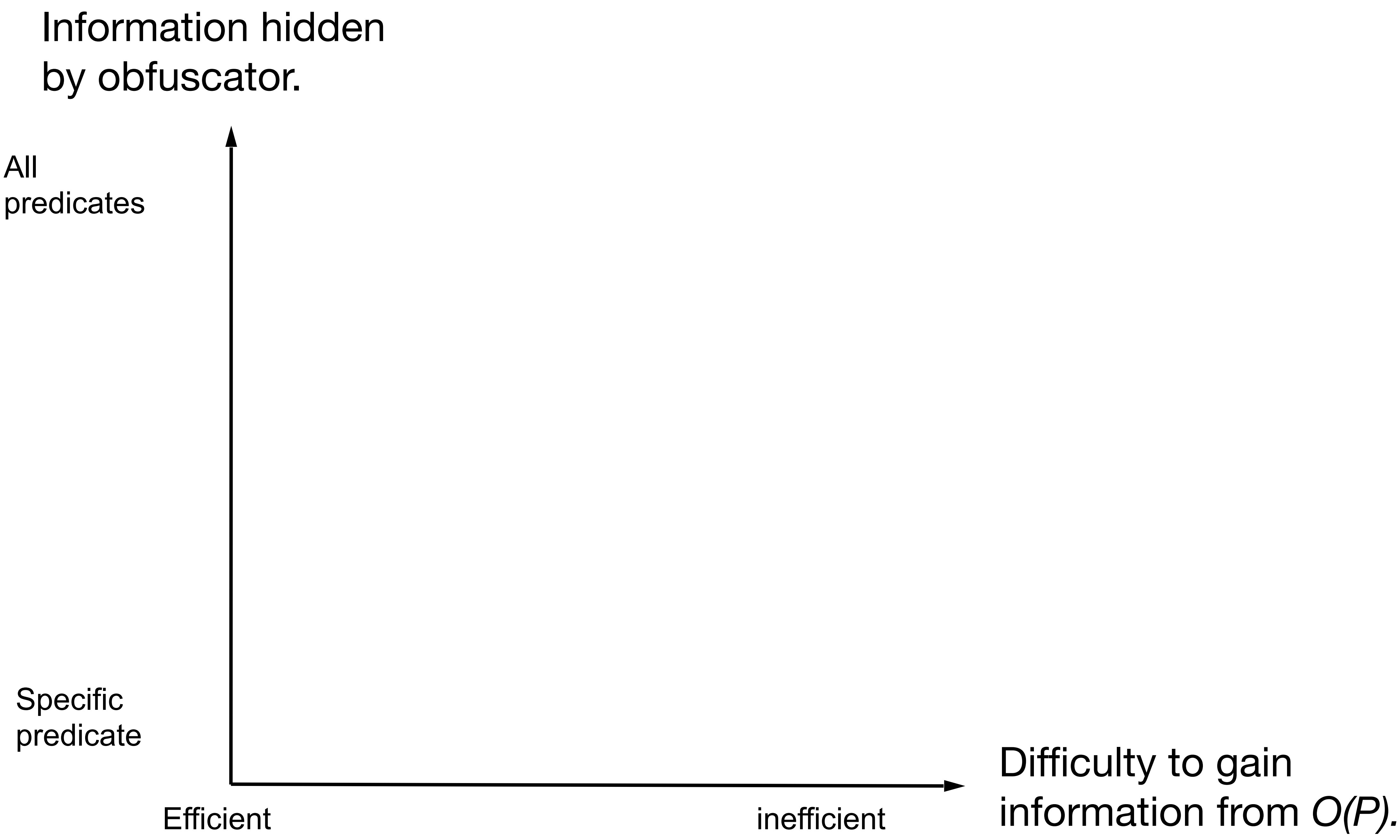
Obfuscation Model Space



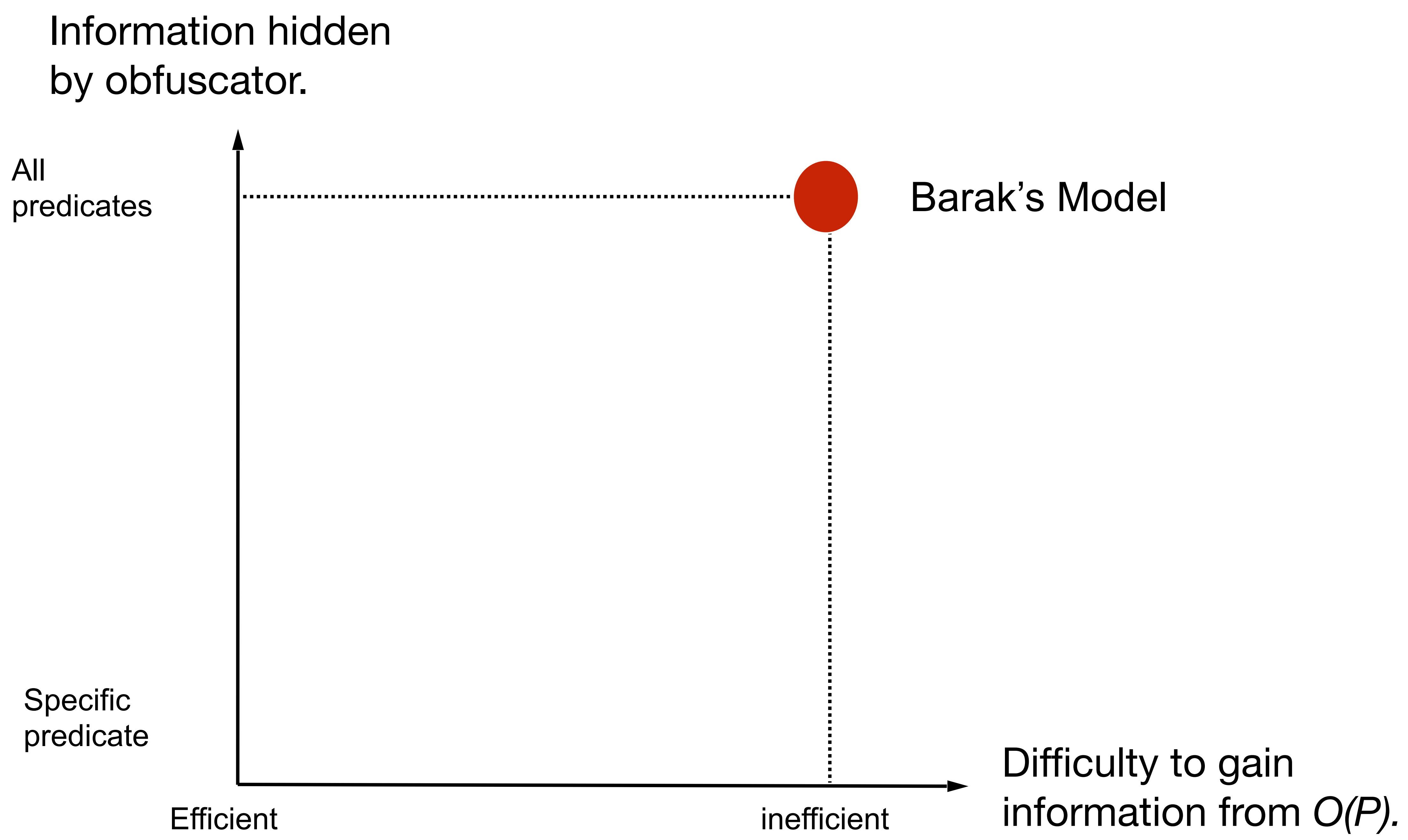
Obfuscation Model Space



Obfuscation Model Space



Obfuscation Model Space



TM Obfuscator

A Turing machine O is a TM obfuscator if for any Turing machine M:

1. $O(M)$ computes the same function as M.
2. $O(M)$ running time[1] is the same as M.
3. For any efficient algorithm[2] A (Analysis) that computes a predicate $p(M)$, there is an efficient (oRacle) algorithm[2] R^M that for all M computes $p(M)$:

$$Pr[A(O(M)) = p(M)] \approx Pr[R^M(1^{|M|}) = p(M)]$$

[1]Polynomial slowdown is permitted

[2]Probabalistic polynomial-time Turing machine

TM Obfuscator

A Turing machine O is a TM obfuscator if for any Turing machine M:

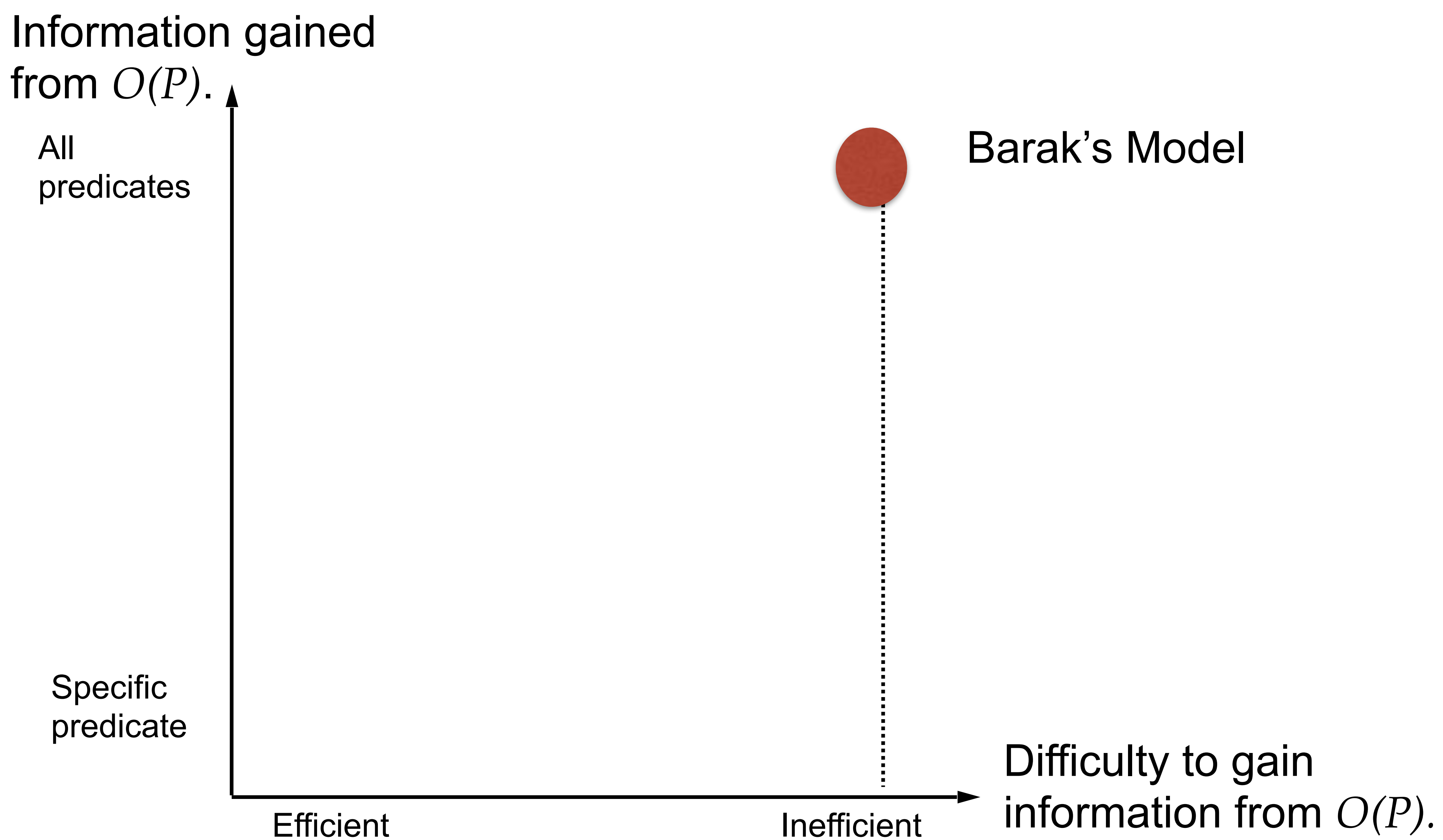
1. $O(M)$ computes the same function as M.
2. $O(M)$ running time[1] is the same as M.
3. For any efficient algorithm[2] A (Analysis) that computes a predicate $p(M)$, there is an efficient (oRacle) algorithm[2] R^M that for all M computes $p(M)$:

$$Pr[A(O(M)) = p(M)] \approx Pr[R^M(1^{|M|}) = p(M)]$$

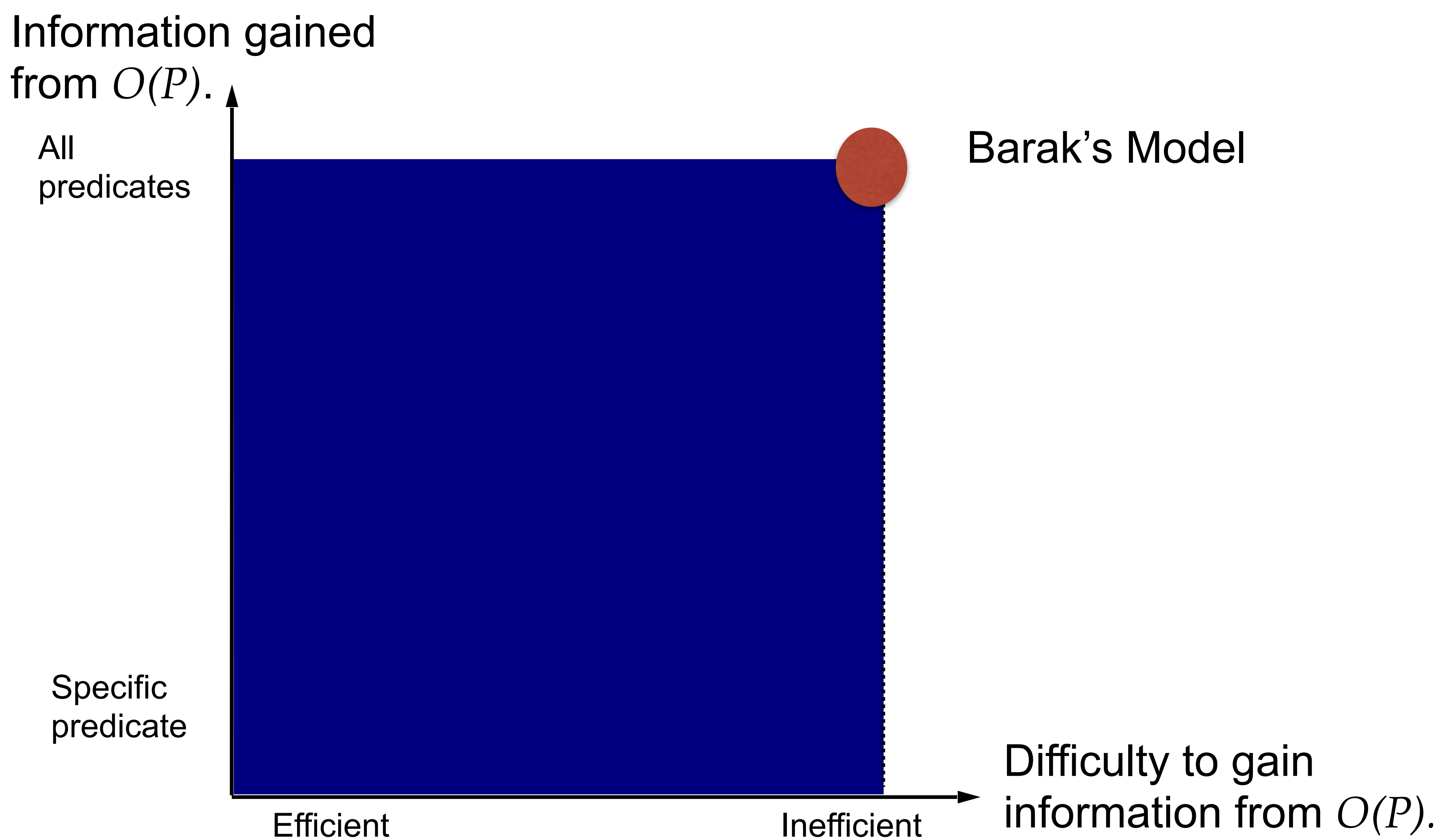
[1]Polynomial slowdown is permitted

[2]Probabalistic polynomial-time Turing machine

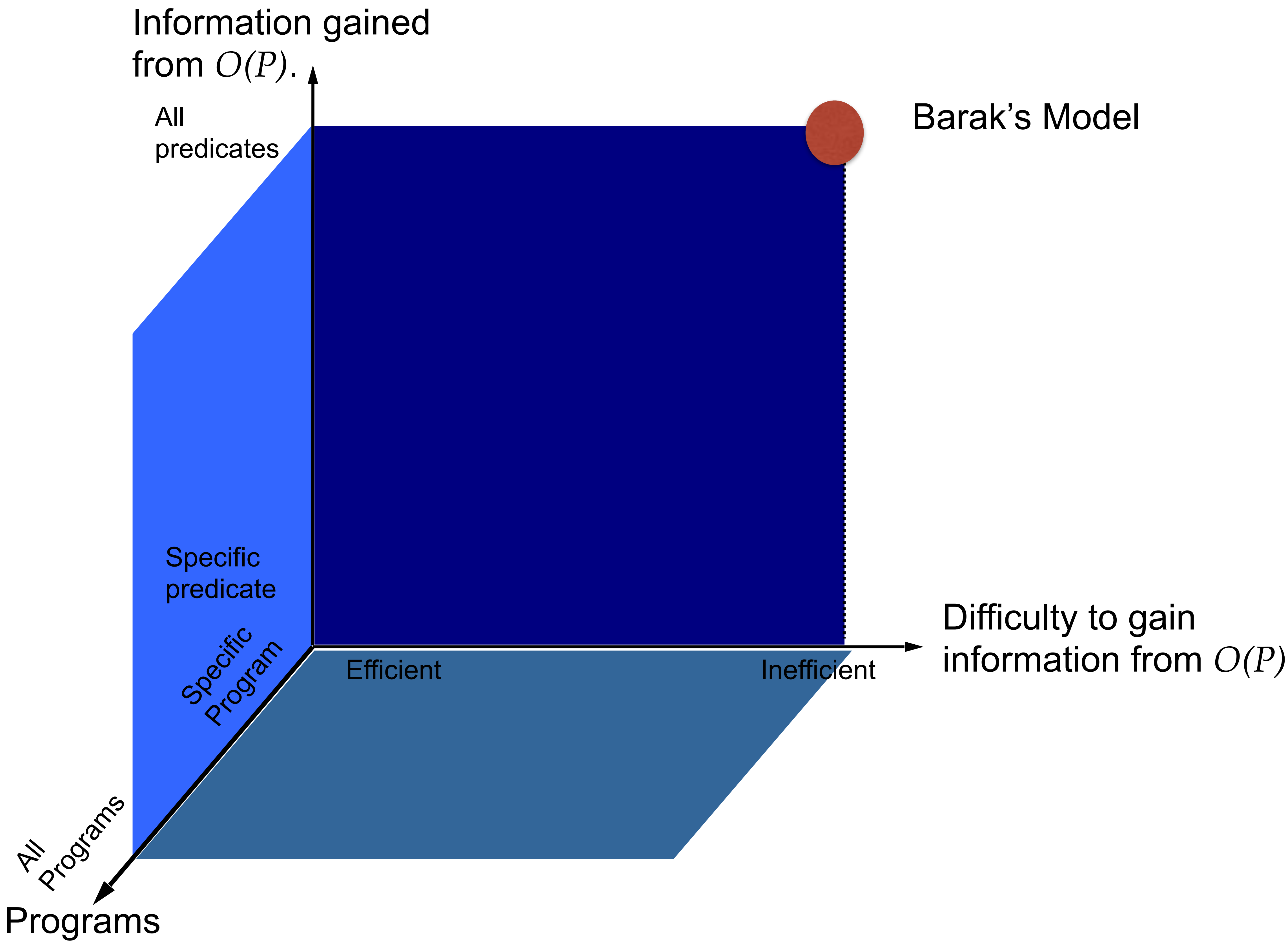
Obfuscation Model Space



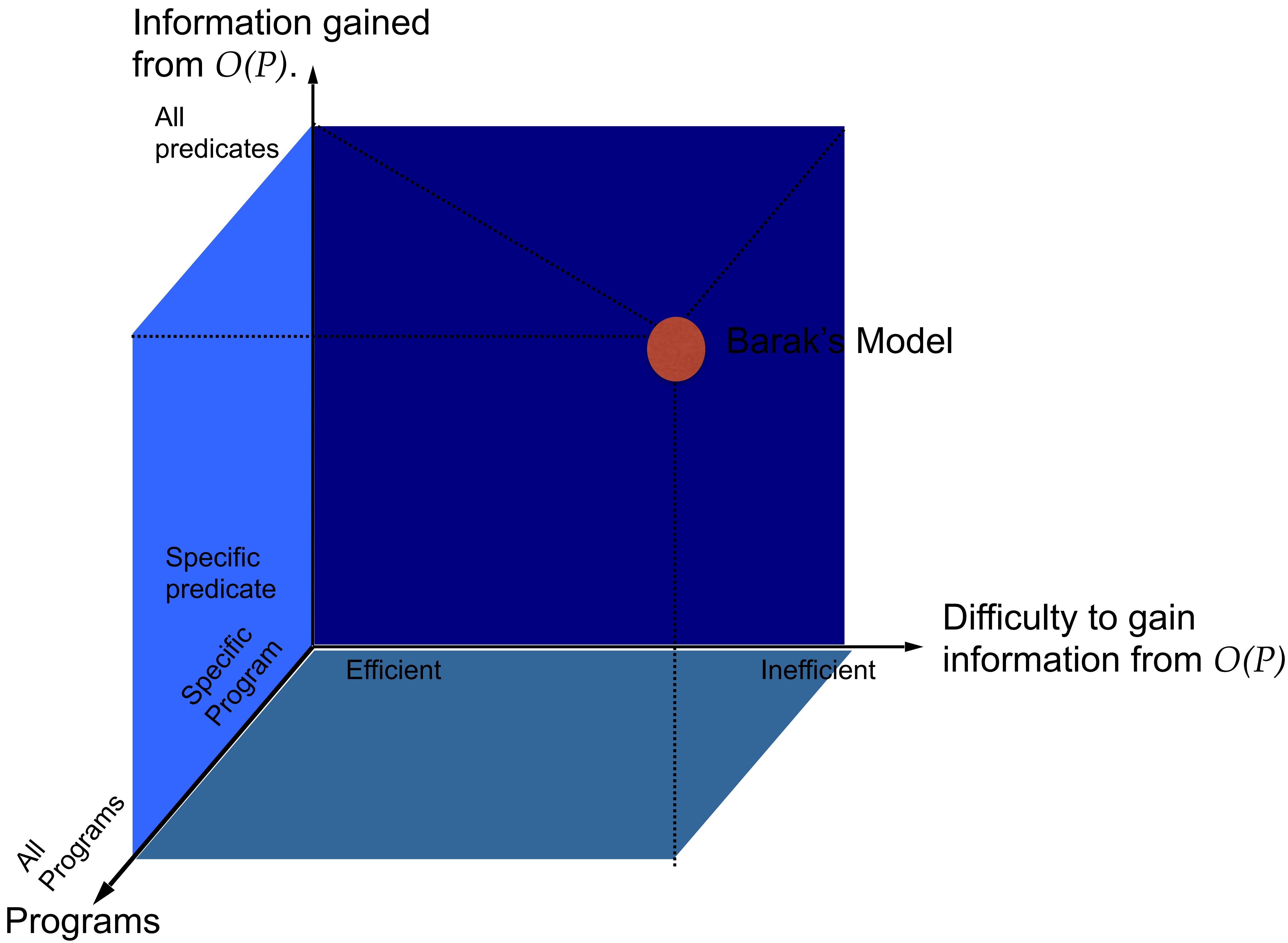
Obfuscation Model Space



Obfuscation Model Space



Obfuscation Model Space



TM Obfuscator

A Turing machine O is a TM obfuscator if for any Turing machine M:

1. $O(M)$ computes the same function as M.
2. $O(M)$ running time[1] is the same as M.
3. For any efficient algorithm[2] A (Analysis) that computes a predicate $p(M)$, there is an efficient (oRacle) algorithm[2] R^M that for all M computes $p(M)$:

$$Pr[A(O(M)) = p(M)] \approx Pr[R^M(1^{|M|}) = p(M)]$$

[1]Polynomial slowdown is permitted

[2]Probabalistic polynomial-time Turing machine

TM Obfuscator

A Turing machine O is a TM obfuscator if for any Turing machine M:

1. $O(M)$ computes the same function as M.
2. $O(M)$ running time[1] is the same as M.
3. For any efficient algorithm[2] A (Analysis) that computes a predicate $p(M)$, there is an efficient (oRacle) algorithm[2] R^M that for all M computes $p(M)$:

$$Pr[A(O(M)) = p(M)] \approx Pr[R^M(1^{|M|}) = p(M)]$$

[1]Polynomial slowdown is permitted

[2]Probabalistic polynomial-time Turing machine

TM Obfuscator

A Turing machine O is a TM obfuscator if for any Turing machine M:

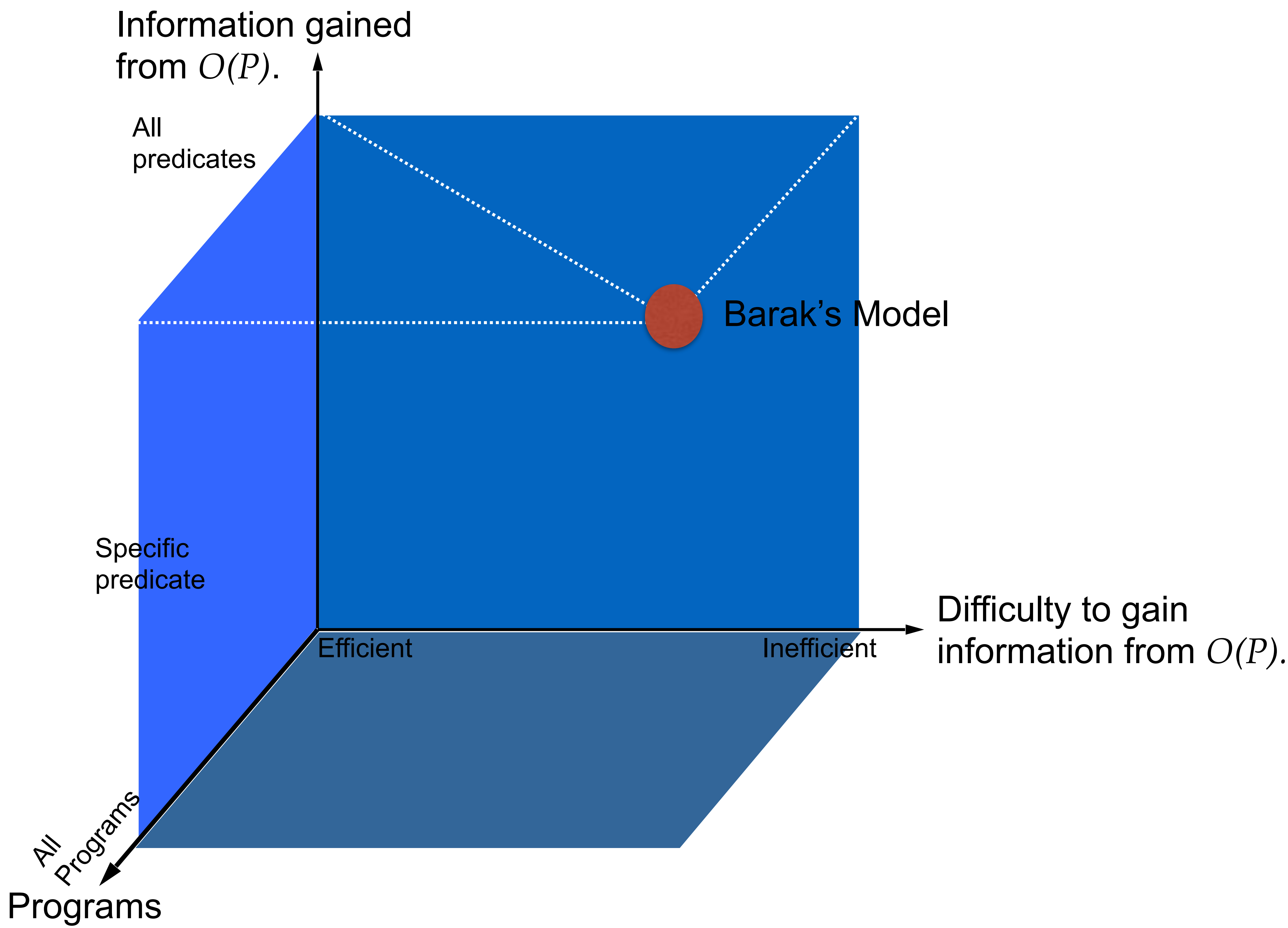
1. $O(M)$ computes the same function as M.
2. $O(M)$ running time[1] is the same as M.
3. For any efficient algorithm[2] A (Analysis) that computes a predicate $p(M)$, there is an efficient (oRacle) algorithm[2] R^M that for all M computes $p(M)$:

$$Pr[A(O(M)) = p(M)] \approx Pr[R^M(1^{|M|}) = p(M)]$$

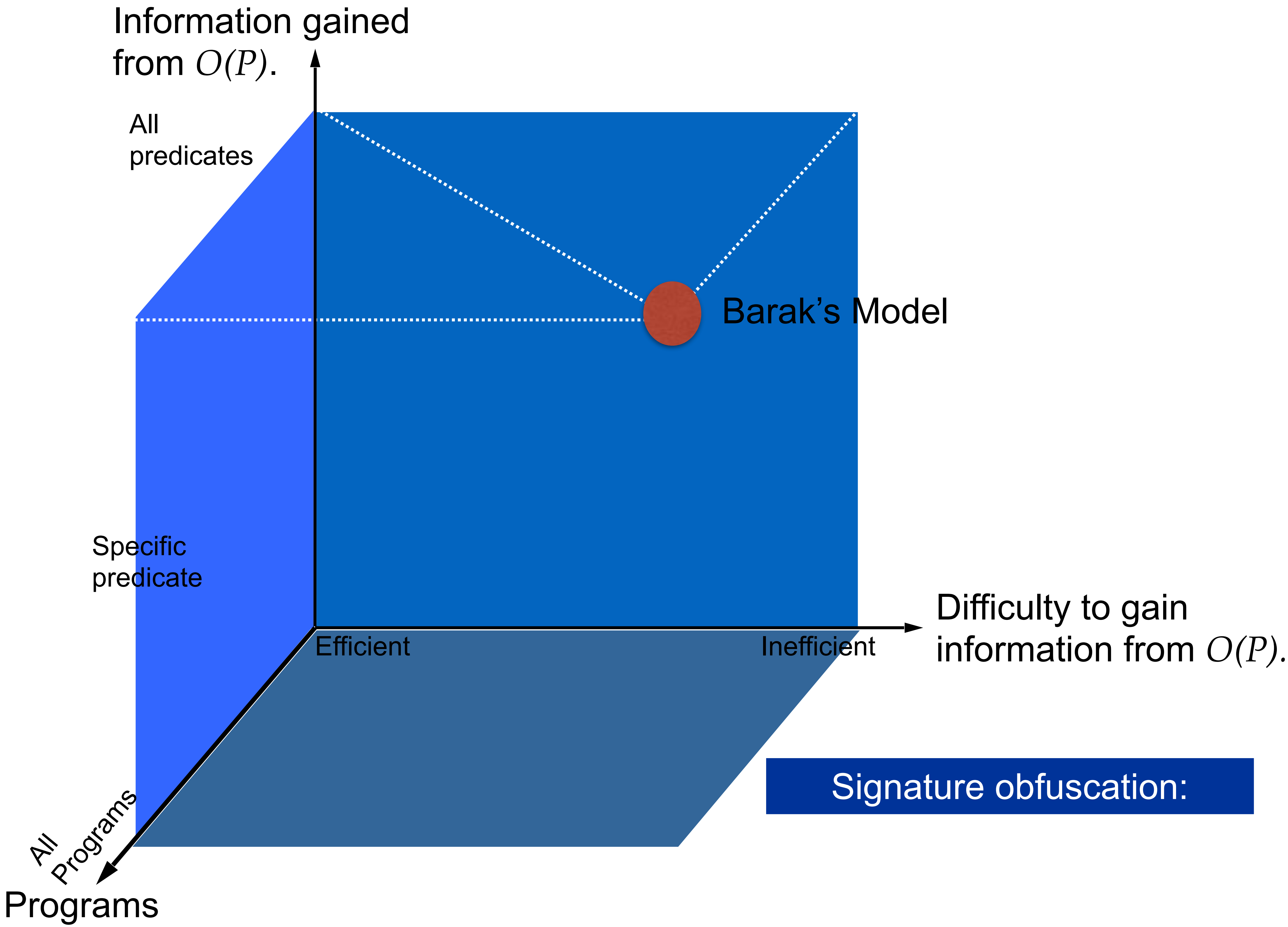
[1]Polynomial slowdown is permitted

[2]Probabalistic polynomial-time Turing machine

Other Obfuscation Models

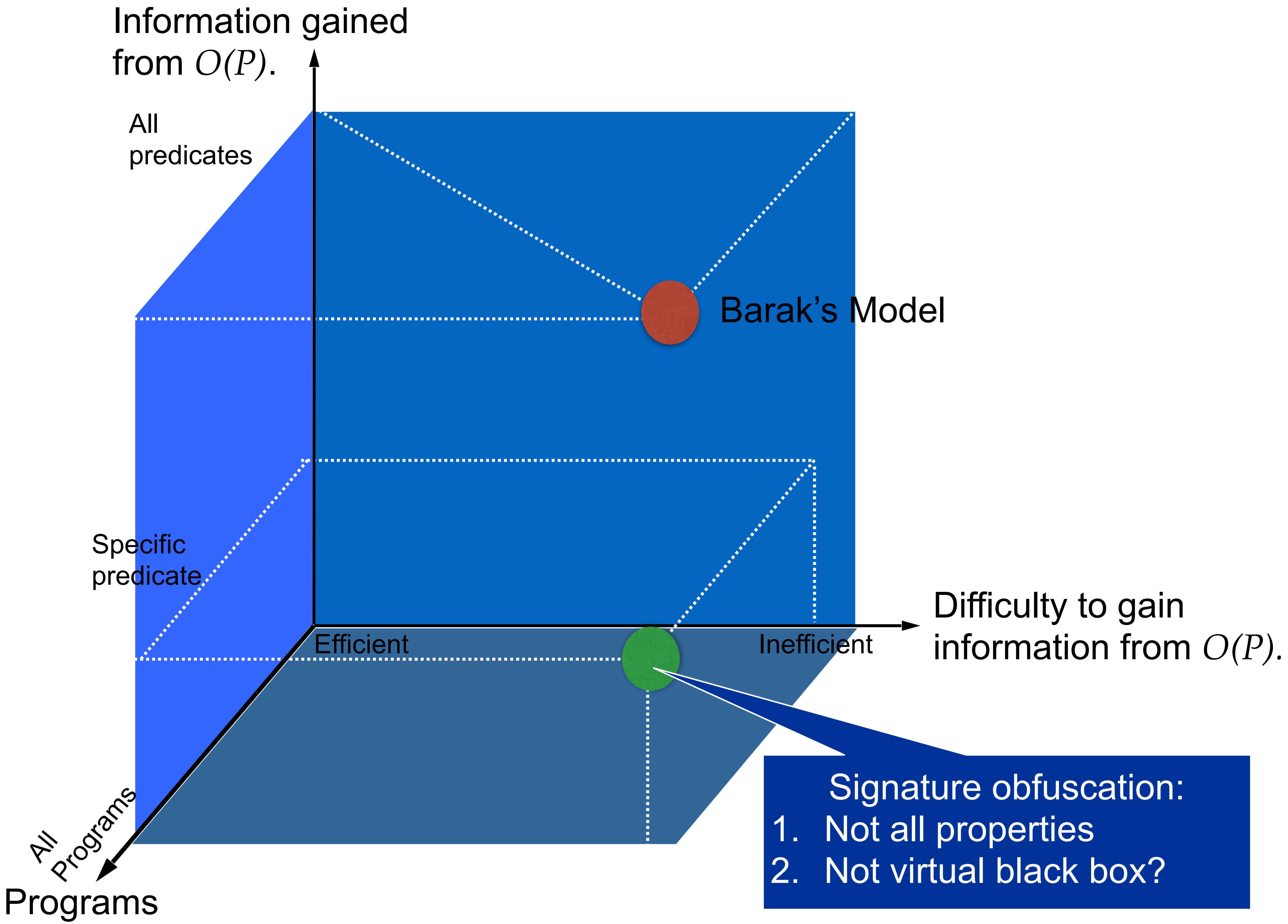


Other Obfuscation Models

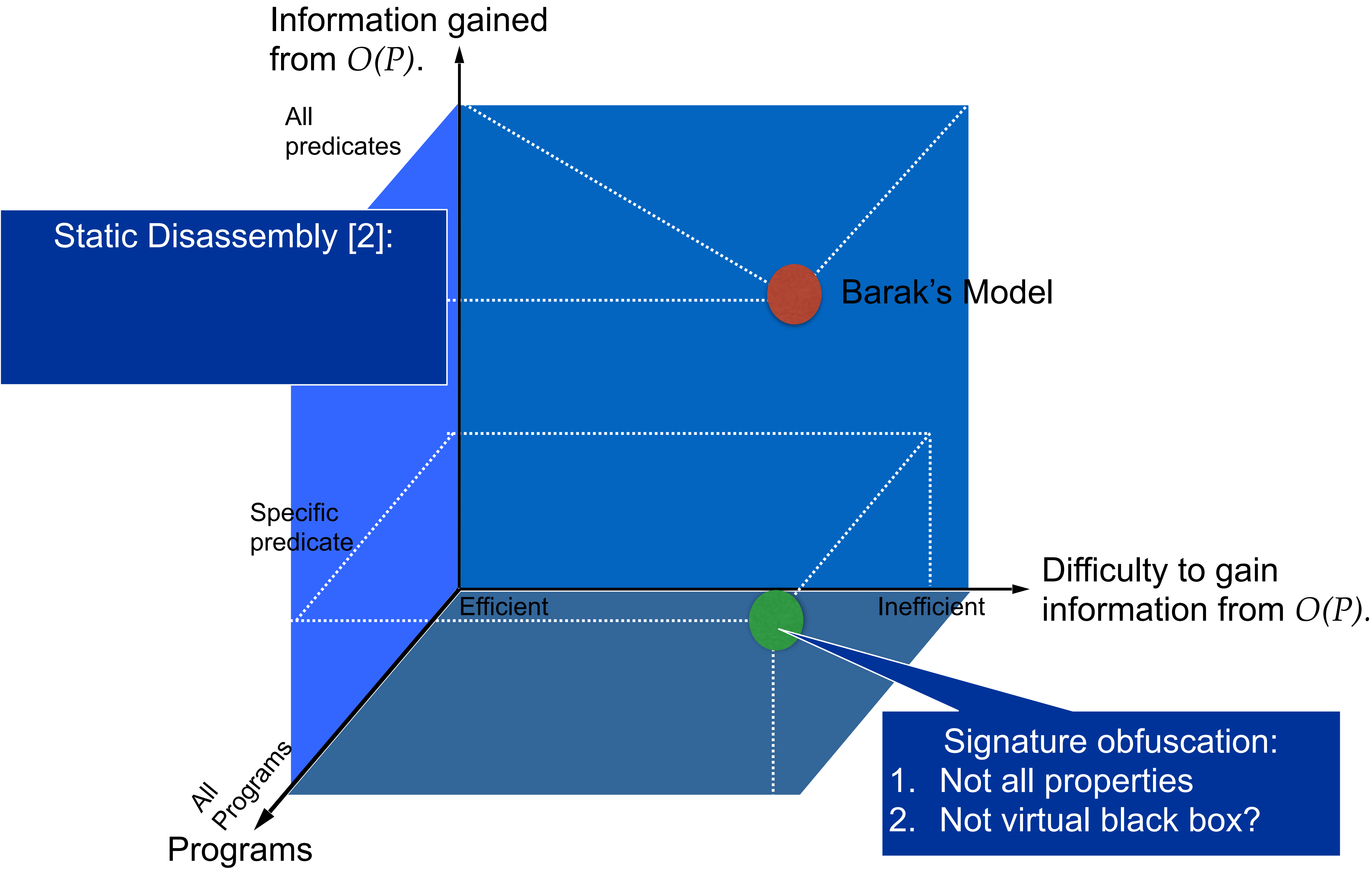


Signature obfuscation:

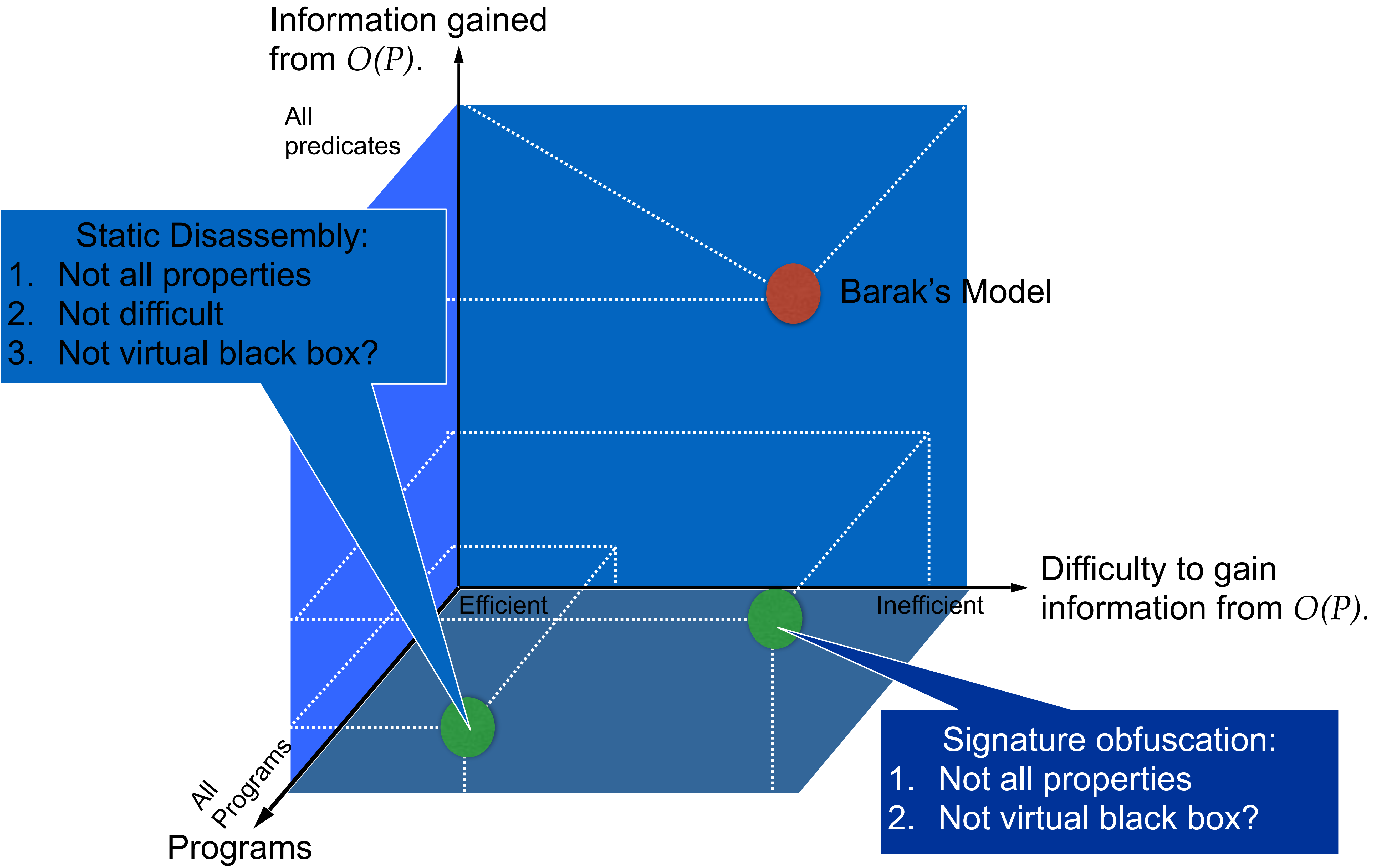
Other Obfuscation Models



Other Obfuscation Models



Other Obfuscation Models



Barak's Model Limitation

- Virtual Black Box:
  - Not surprising in some sense (but, still excellent work)
  - Does not corresponds to what attackers/researchers are doing: "the virtual black box paradigm for obfuscation is inherently flawed"
- Too general:
  - obfuscator must work for all programs
  - for any property (Barak addresses this in the extensions)

The Future?

Indistinguishability Obfuscation

Instance: two families of programs Π_1 and Π_2

Adversary task: given a program $P \in \Pi_1 \cup \Pi_2$ to decide whether $P \in \Pi_1$ or $P \in \Pi_2$.

Desirable protection: make adversary task as difficult as well-known computationally hard problem is.

Garg et al., CRYPTO 2013

The Future?

Indistinguishability Obfuscation

Indistinguishability:
If P and Q compute
the same function
then $\mathcal{O}(P) \approx \mathcal{O}(Q)$

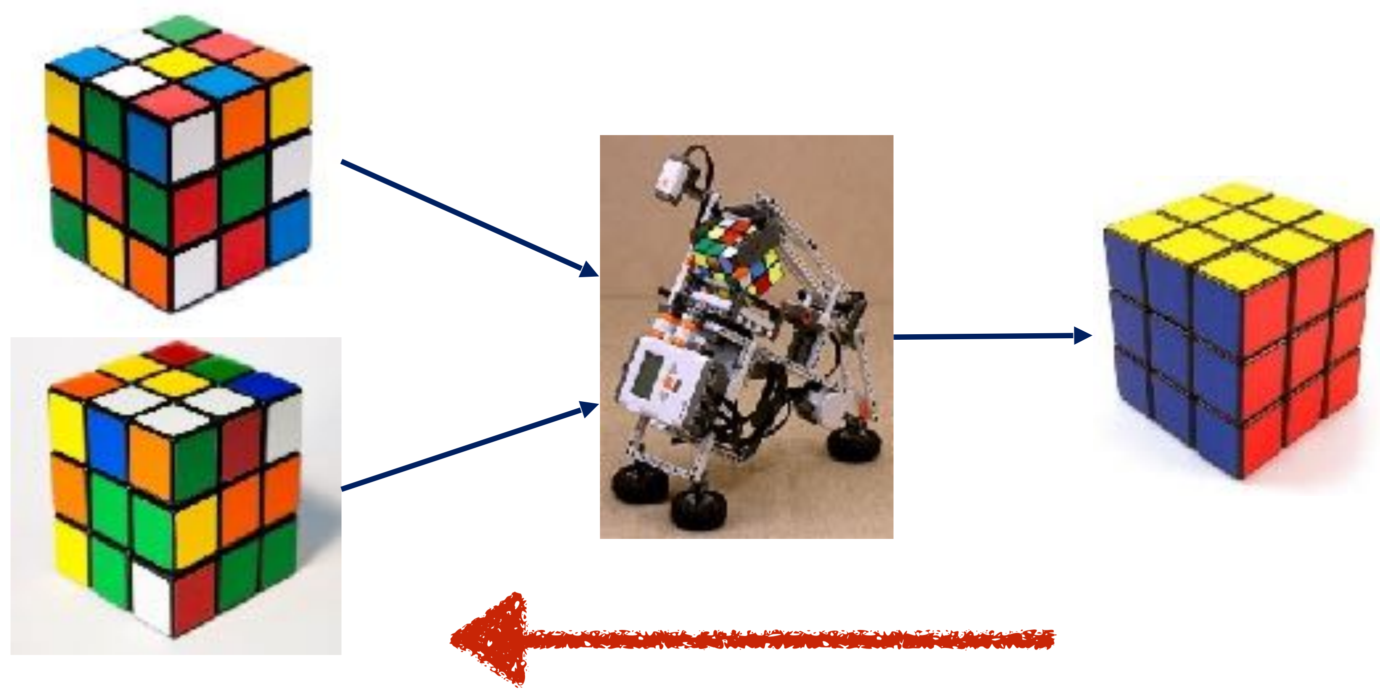
Hard

Garg et al., CRYPTO 2013

The Future?

Indistinguishability Obfuscation

Indistinguishability:
If P and Q compute
the same function
then $\mathcal{O}(P) \approx \mathcal{O}(Q)$



Hard

Garg et al., CRYPTO 2013

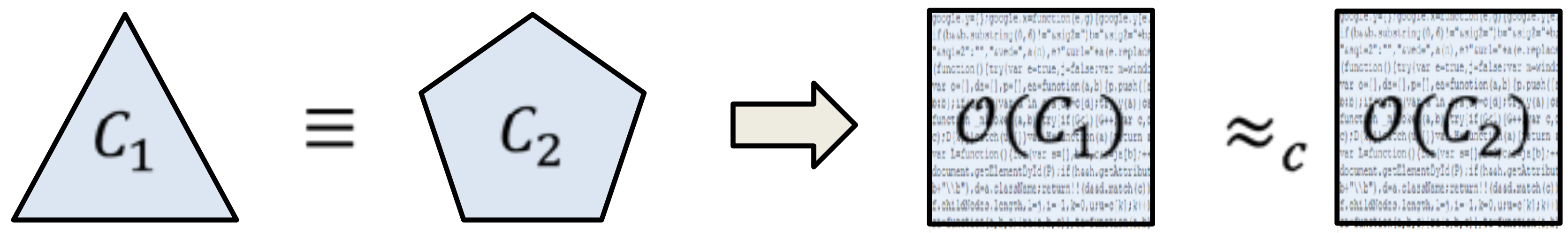
The Future?

Indistinguishability Obfuscation

Garg et al., CRYPTO 2013

The Future?

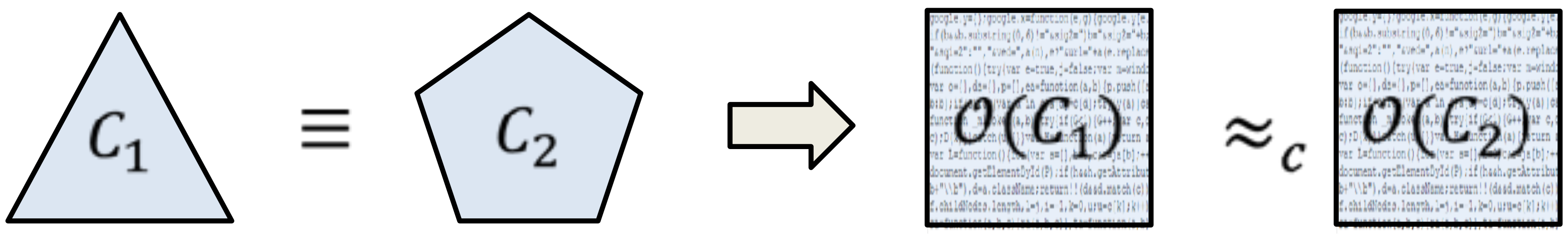
Indistinguishability Obfuscation



Garg et al., CRYPTO 2013

The Future?

Indistinguishability Obfuscation

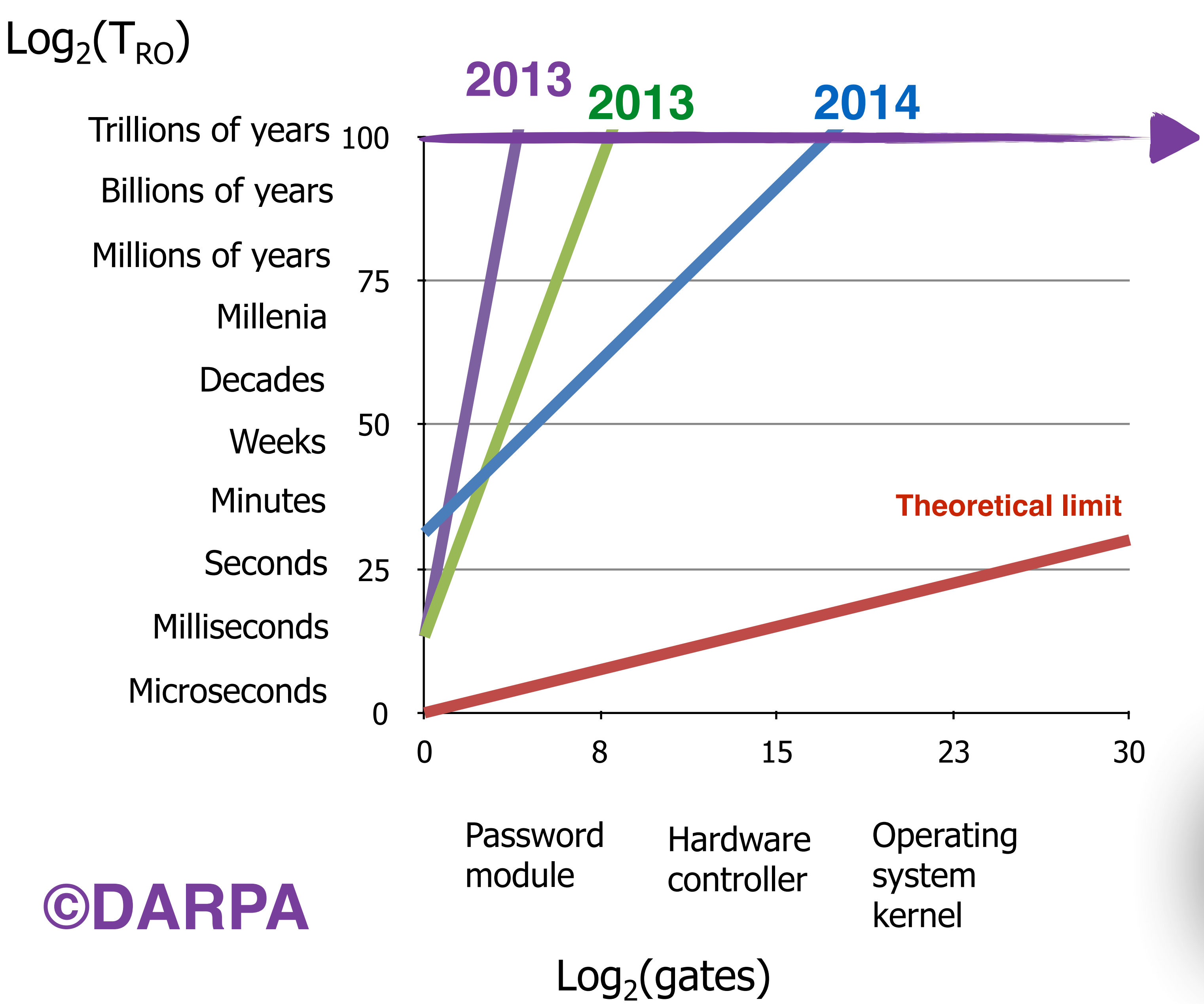


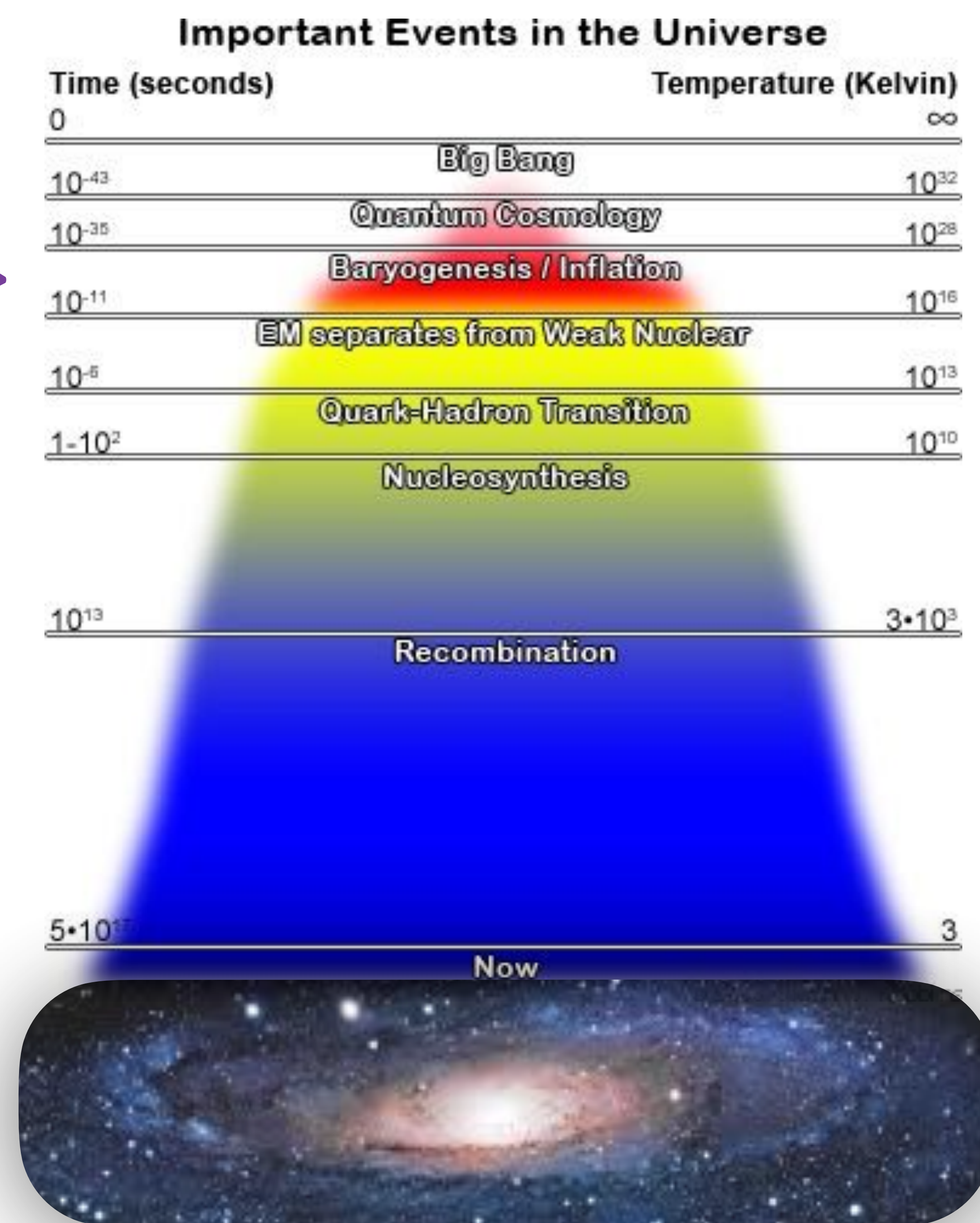
Assumption:
indistinguishability obfuscation for all circuits

Garg et al., CRYPTO 2013

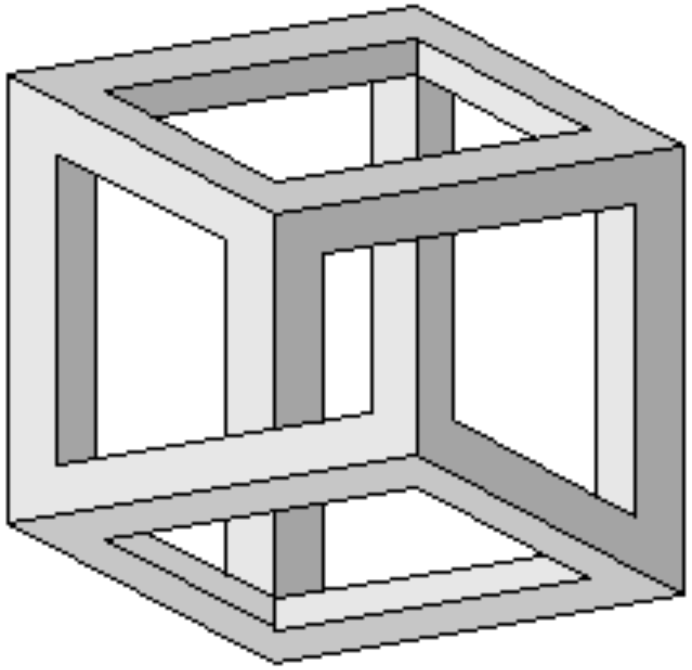
Current limitations

Assumption: 1 GHz processor





©DARPA



About (im)possibility

On THE (im)possibility result!

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

BY

H. G. RICE

1952

$$\underset{\text{Code}}{W} \longrightarrow \quad W \overset{?}{\in} \{\, P \mid P \approx Q \,\} \quad \longrightarrow \; ?$$

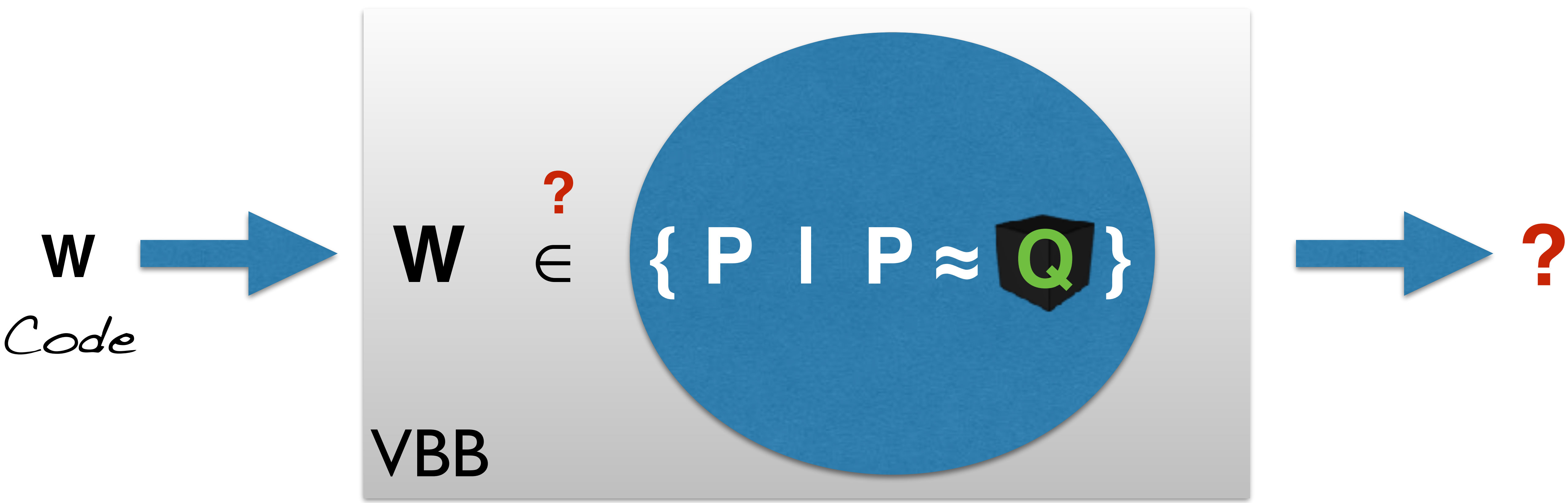
We can only approximate!!!

Another (im)possibility result!

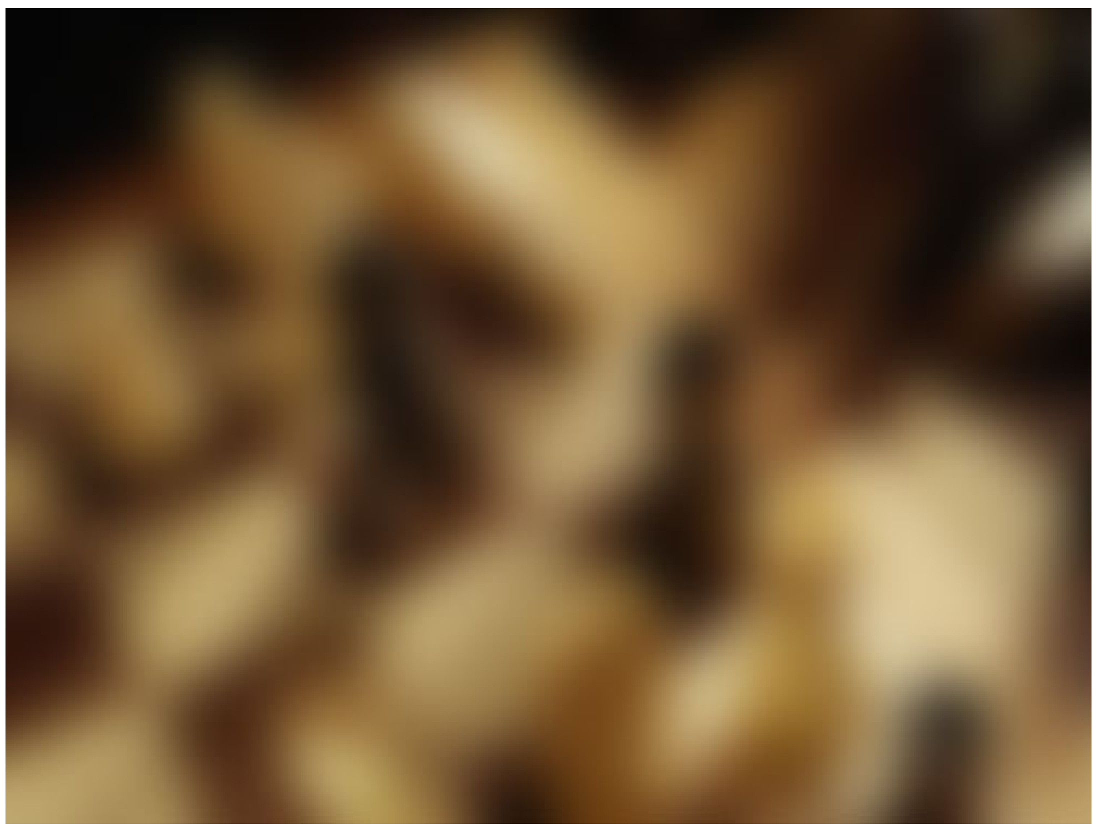
On the (Im)possibility of Obfuscating Programs*

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

2001



We can only partially obfuscate!!!



What does it mean being obscure?

...a different viewpoint from PL

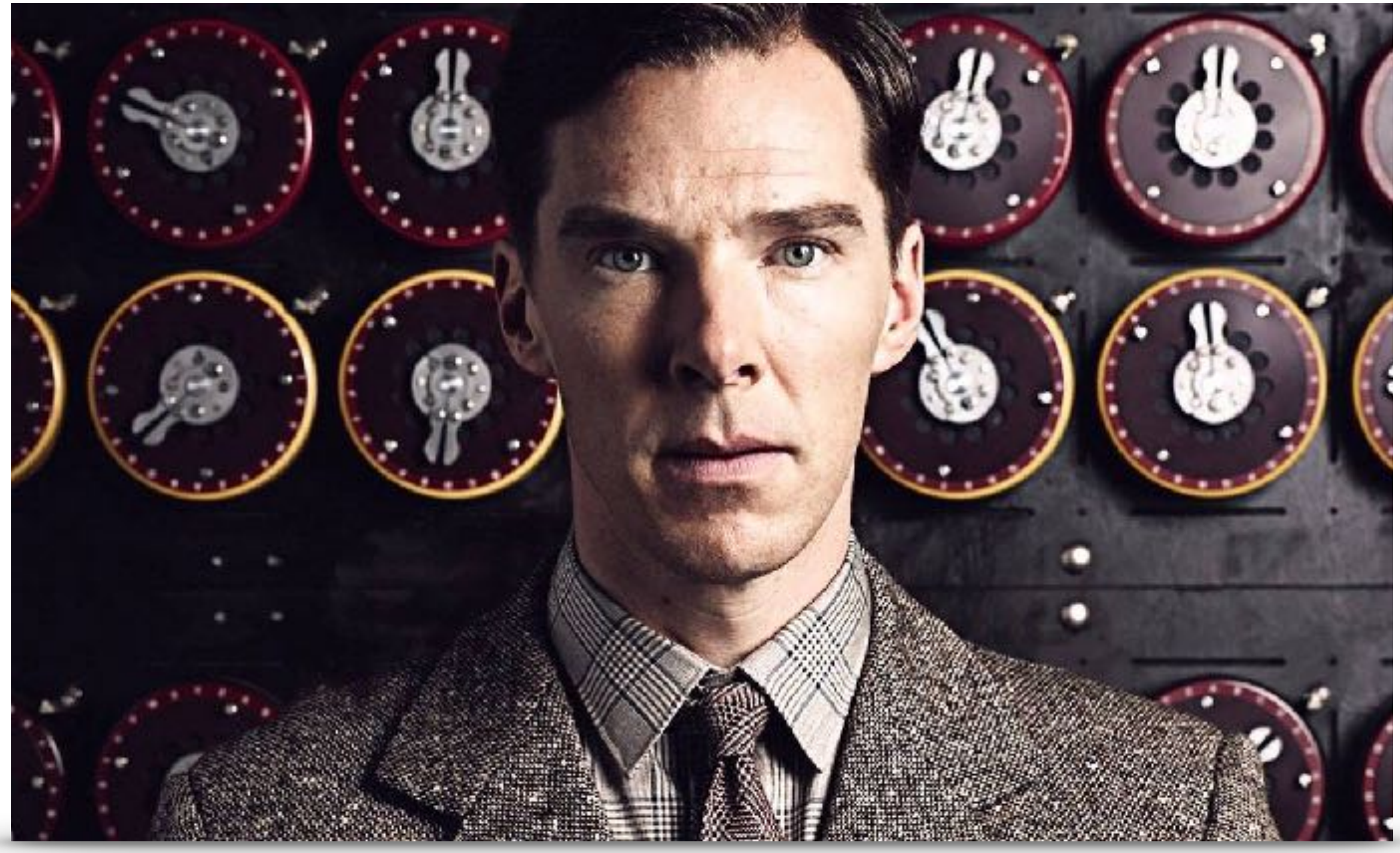
The Attack Model

The computing power and memory size of computers *double* every 18 months

Size of programs grows proportionally

Analysis is *exponential* in the program size

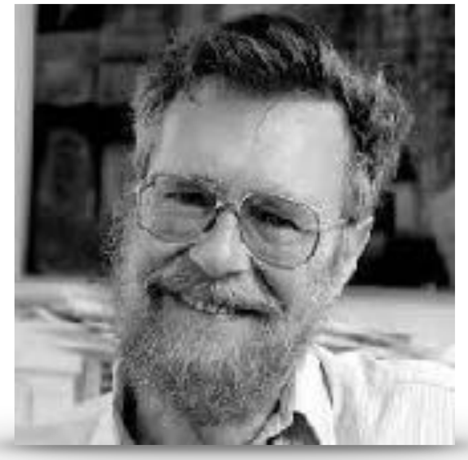
Attackers need computers to attack computers



Attackers need computers to attack computers

Whole-program view

Good programs are well-structured and have concise invariants



Obscure programs are badly-structured and have messy invariants

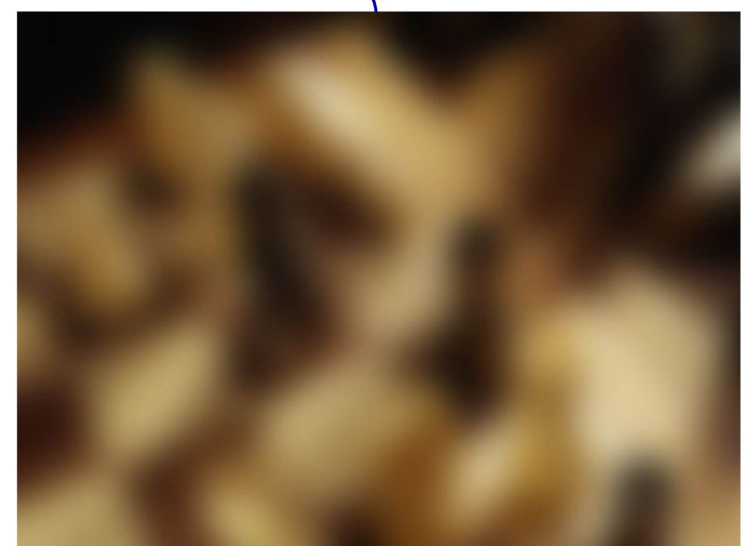
Obfuscation as Compilation

Input

Output

$$\xrightarrow{\tau}$$

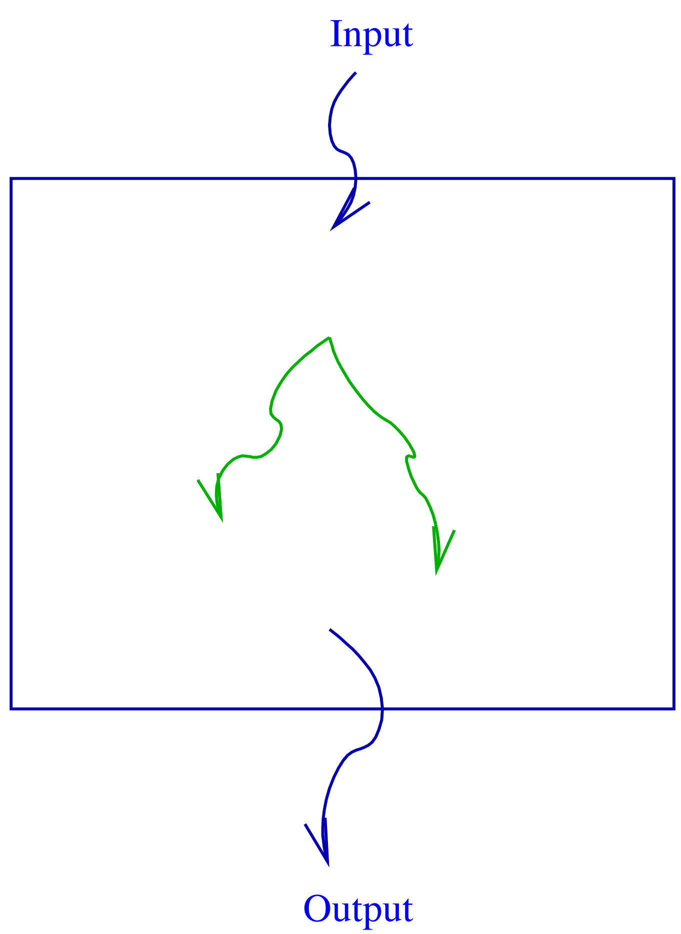
Input



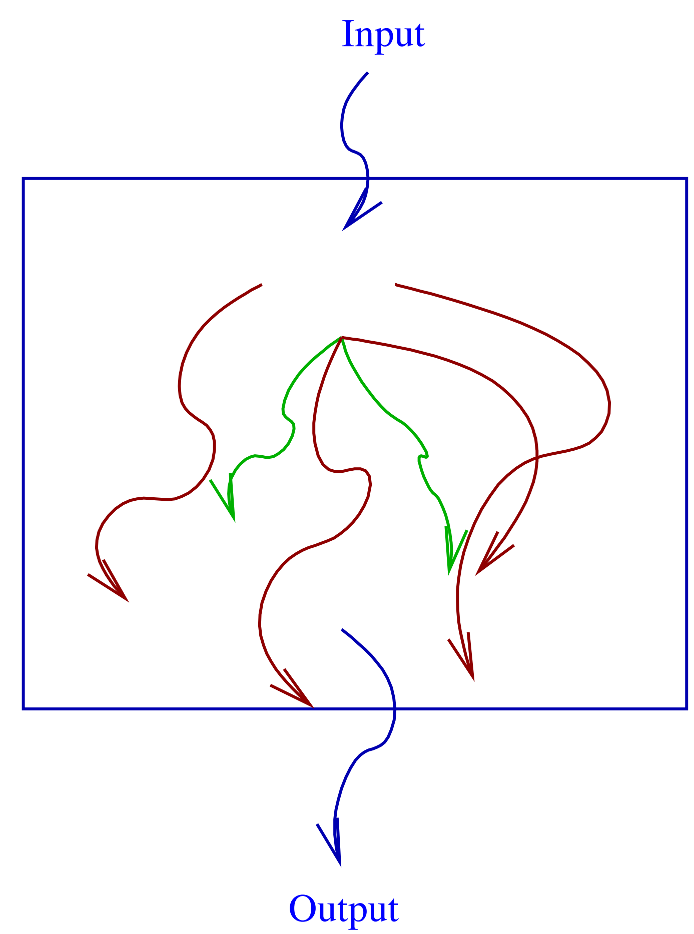
Output

$$P \to \tau[\![P]\!]$$

Obfuscation as Compilation



$$\xrightarrow{\tau}$$



$$P \longrightarrow \tau[\![P]\!]$$

(Pseudo-)Code:

```
mov eax, [edx+0Ch]
push ebx
push [eax]
call ReleaseLock
```

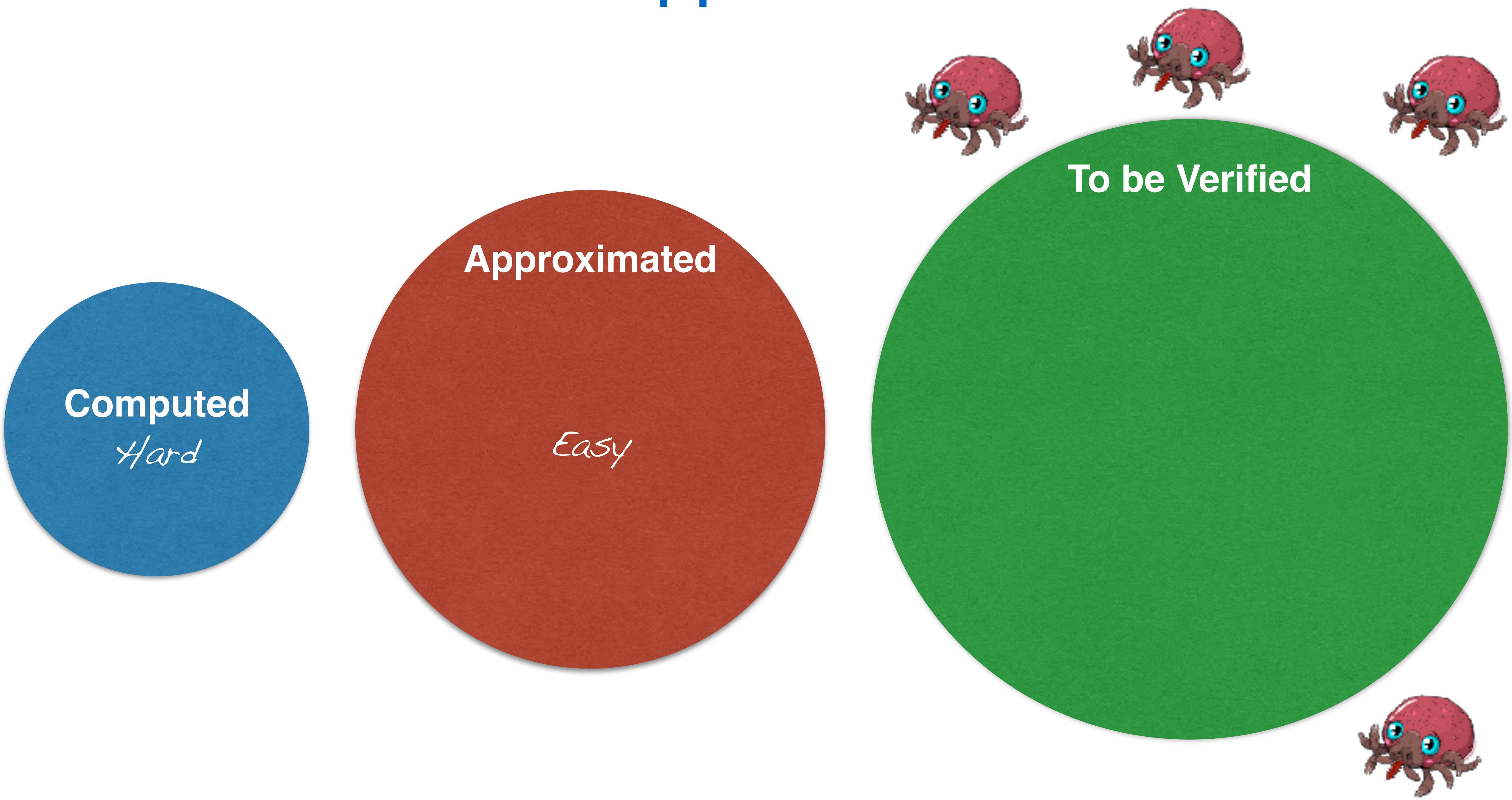
$$\xrightarrow{\tau}$$

Obfuscated code (junk + reordering):

```
mov eax, [edx+0Ch]
jmp +3
push ebx
dec eax
jmp +4
inc eax
jmp -3
call ReleaseLock
jmp +2
push [eax]
jmp -2
```

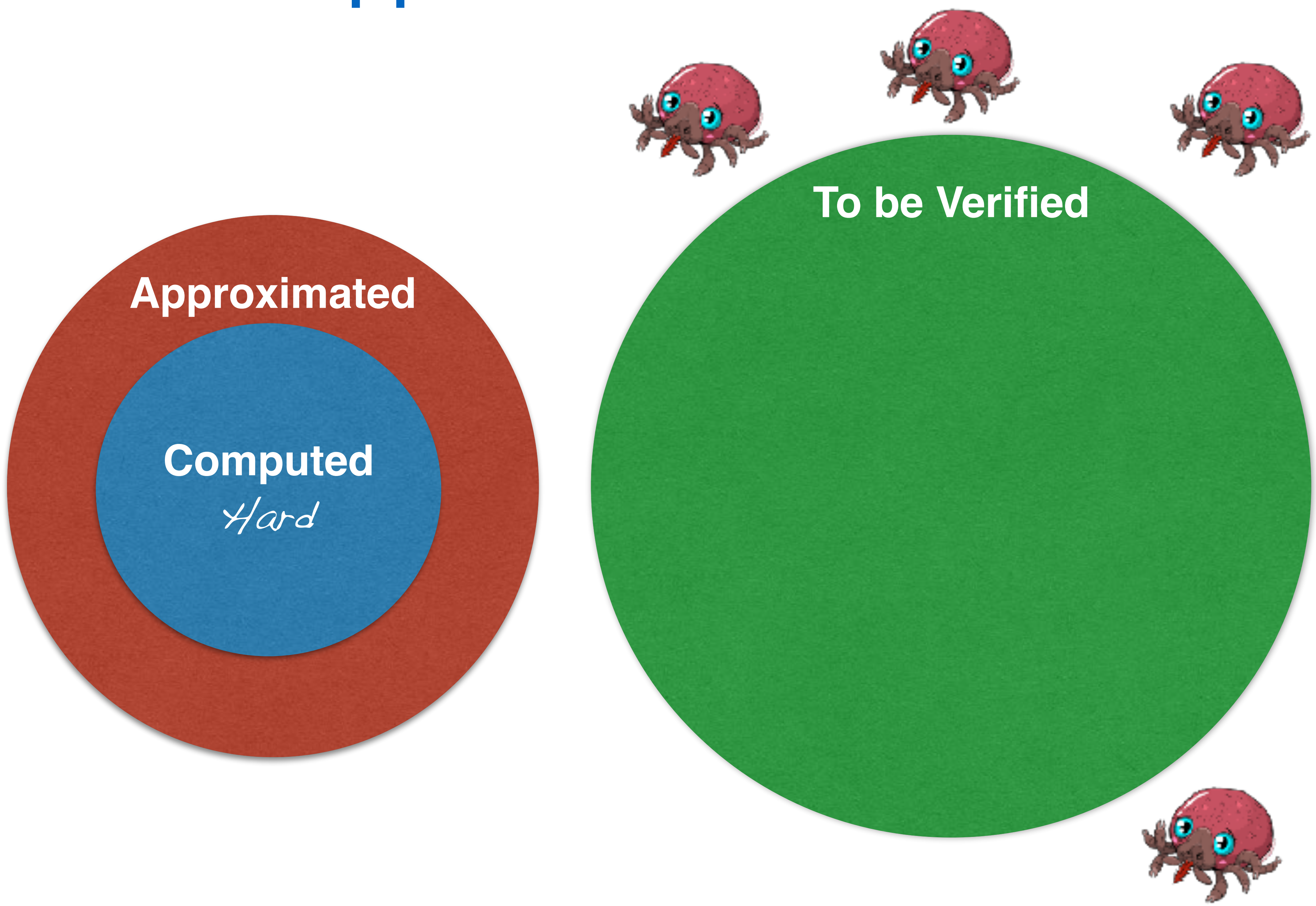
Attacking code is Analysing code

Sound Approximation



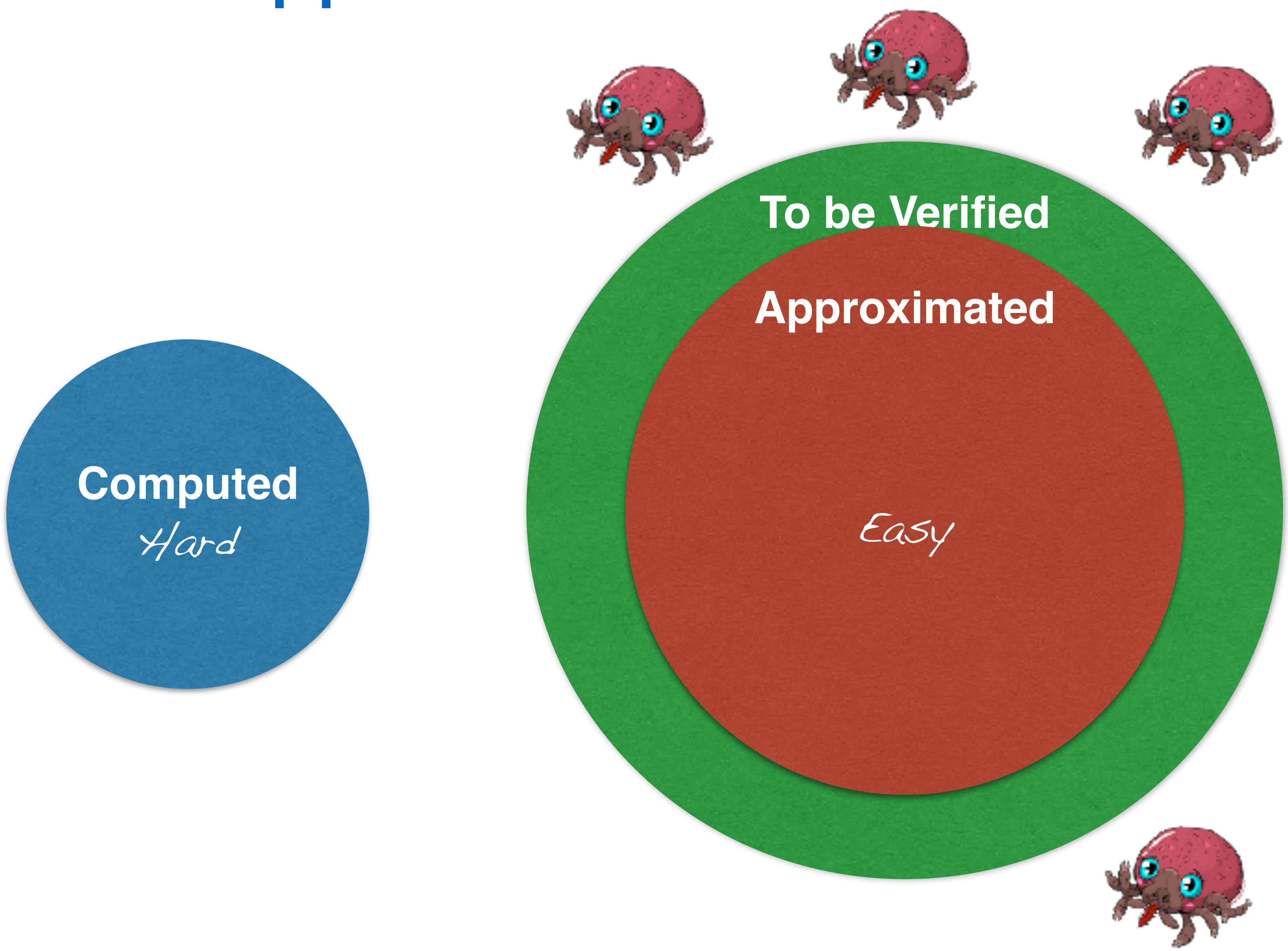
$$[\![P]\!] \subseteq [\![P]\!]^{\alpha} \models \varphi$$

Sound Approximation



$$[\![P]\!] \quad \subseteq \quad [\![P]\!]^{\alpha} \quad \models \quad \varphi$$

Sound Approximation



$$[\![P]\!] \quad \subseteq \quad [\![P]\!]^{\alpha} \quad \models \quad \varphi$$

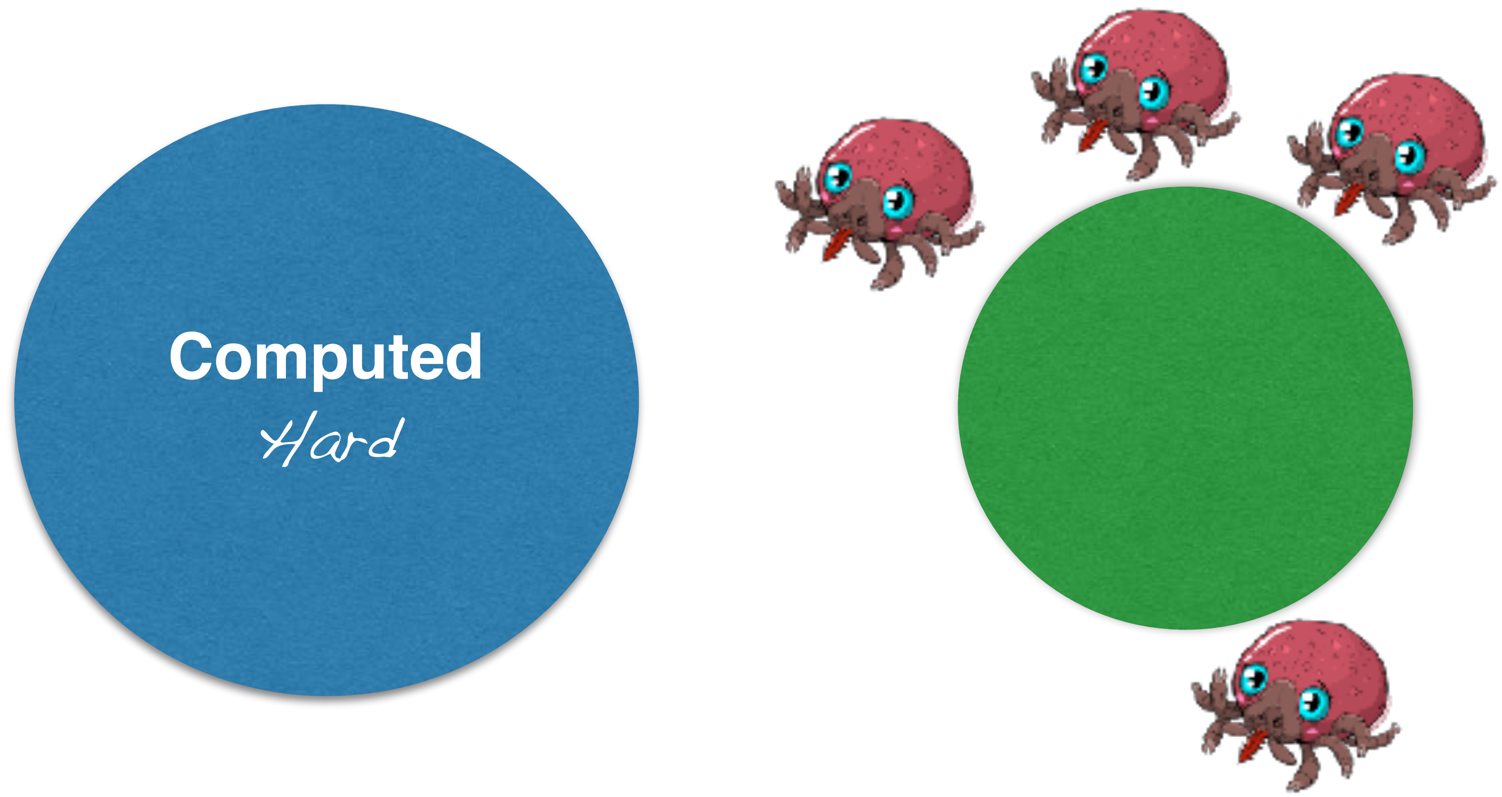
Sound Approximation

$$[\![P]\!] \subseteq [\![P]\!]^{\alpha} \models \varphi$$

Unsound Approximation

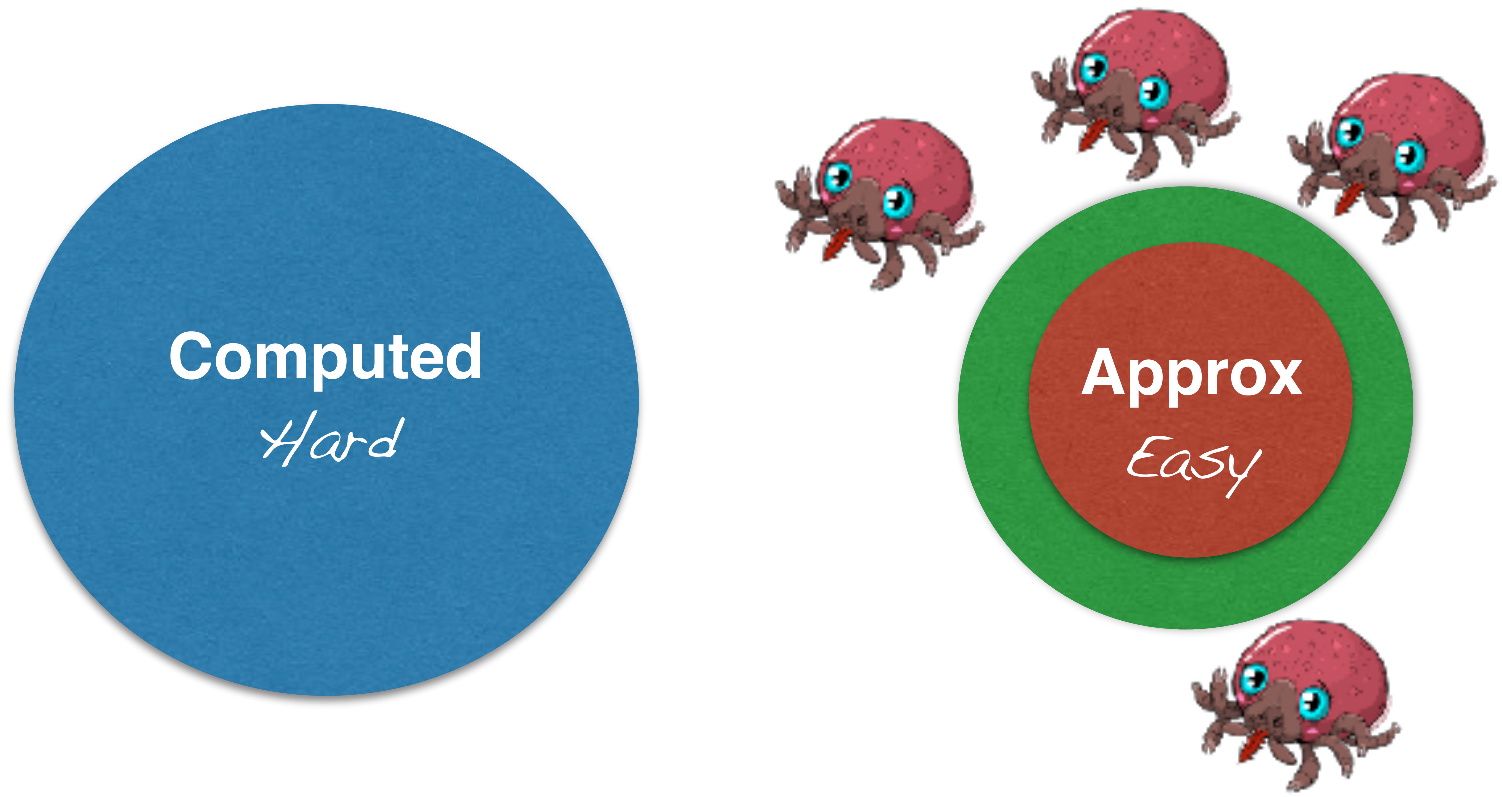
$$[\![P]\!] \not\models \varphi$$

Unsound Approximation



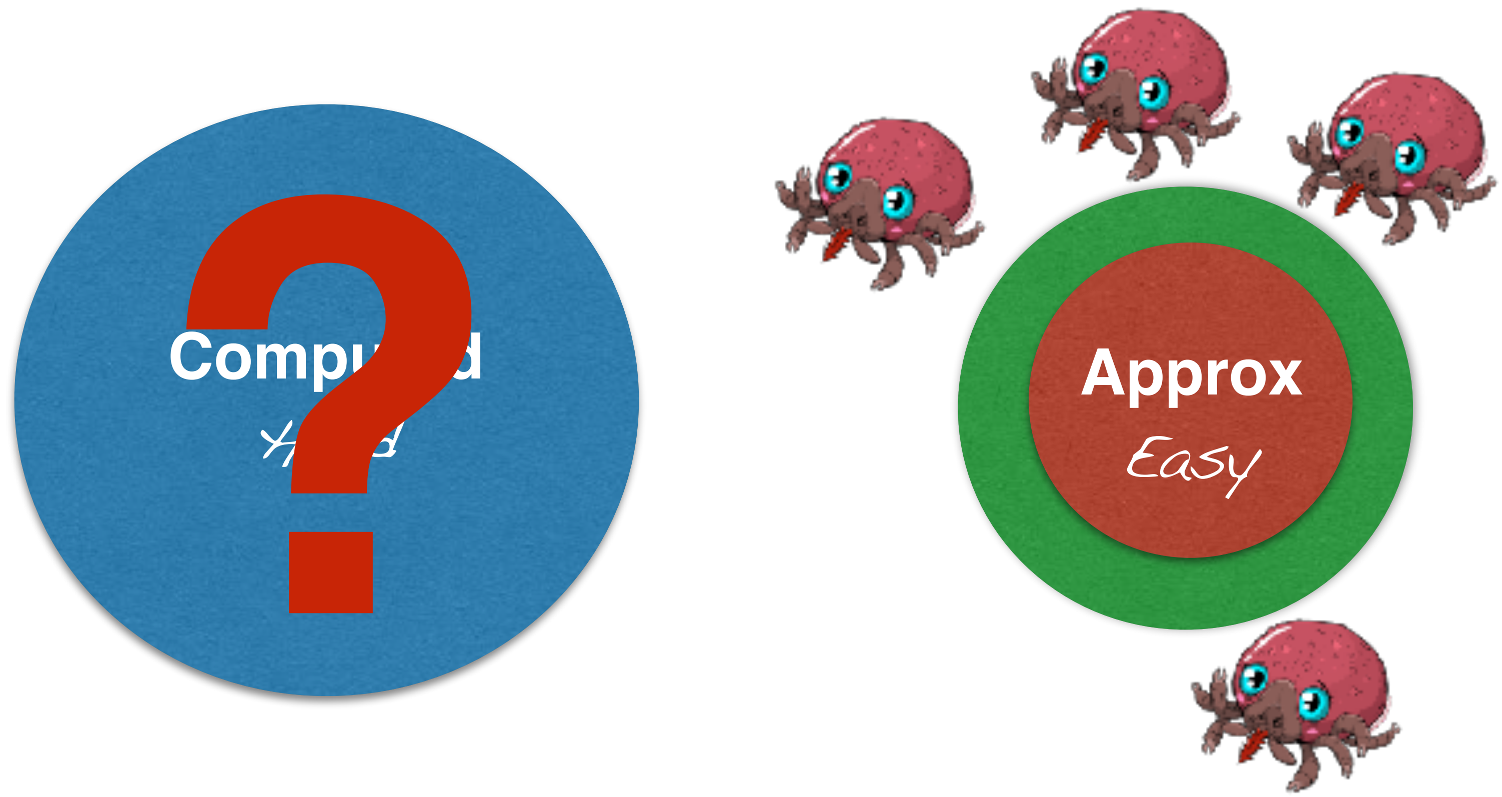
$$[\![P]\!] \;\cancel{\models}\; \varphi$$

Unsound Approximation



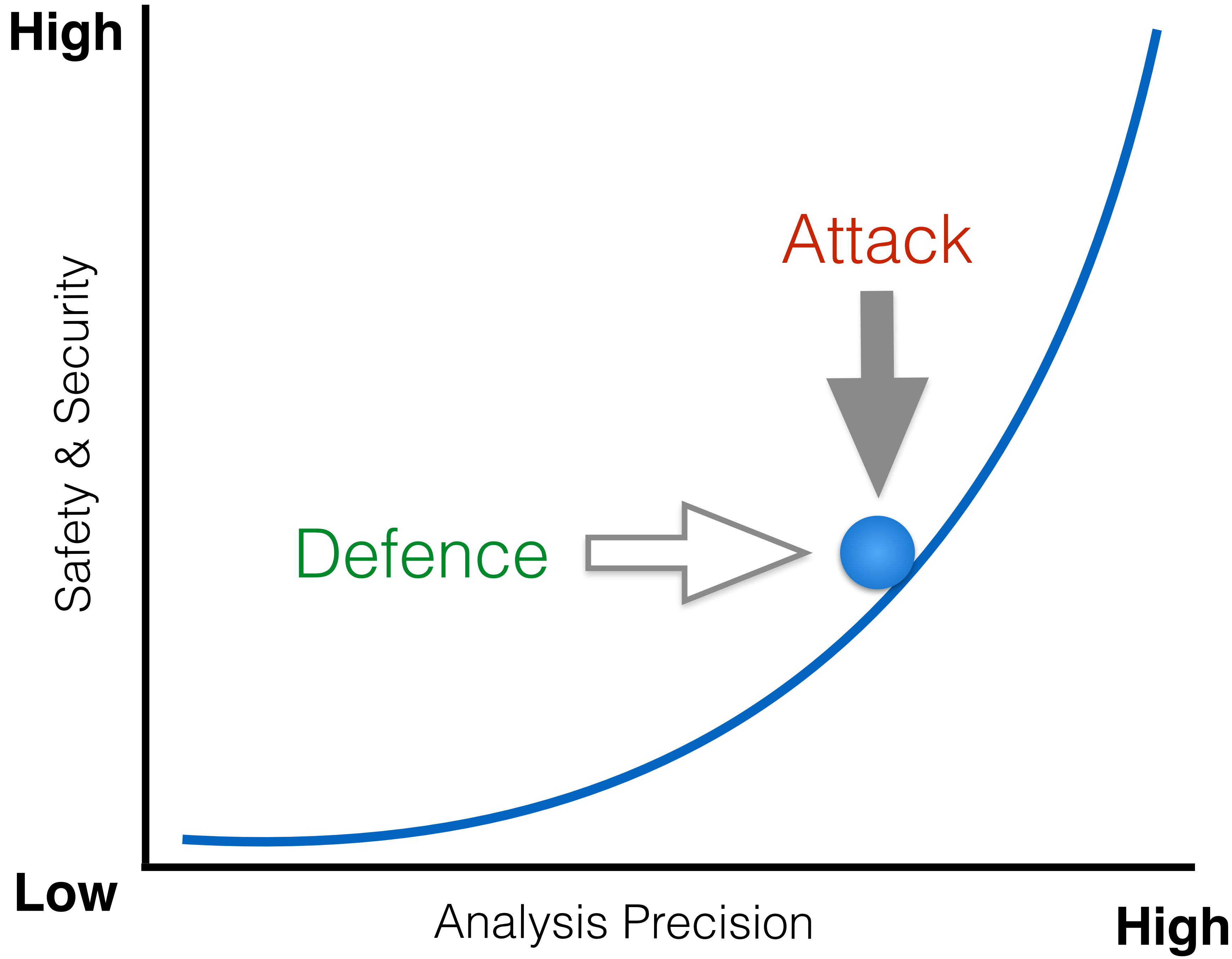
$$\llbracket P \rrbracket \;\cancel{\models}\; \varphi$$

Unsound Approximation



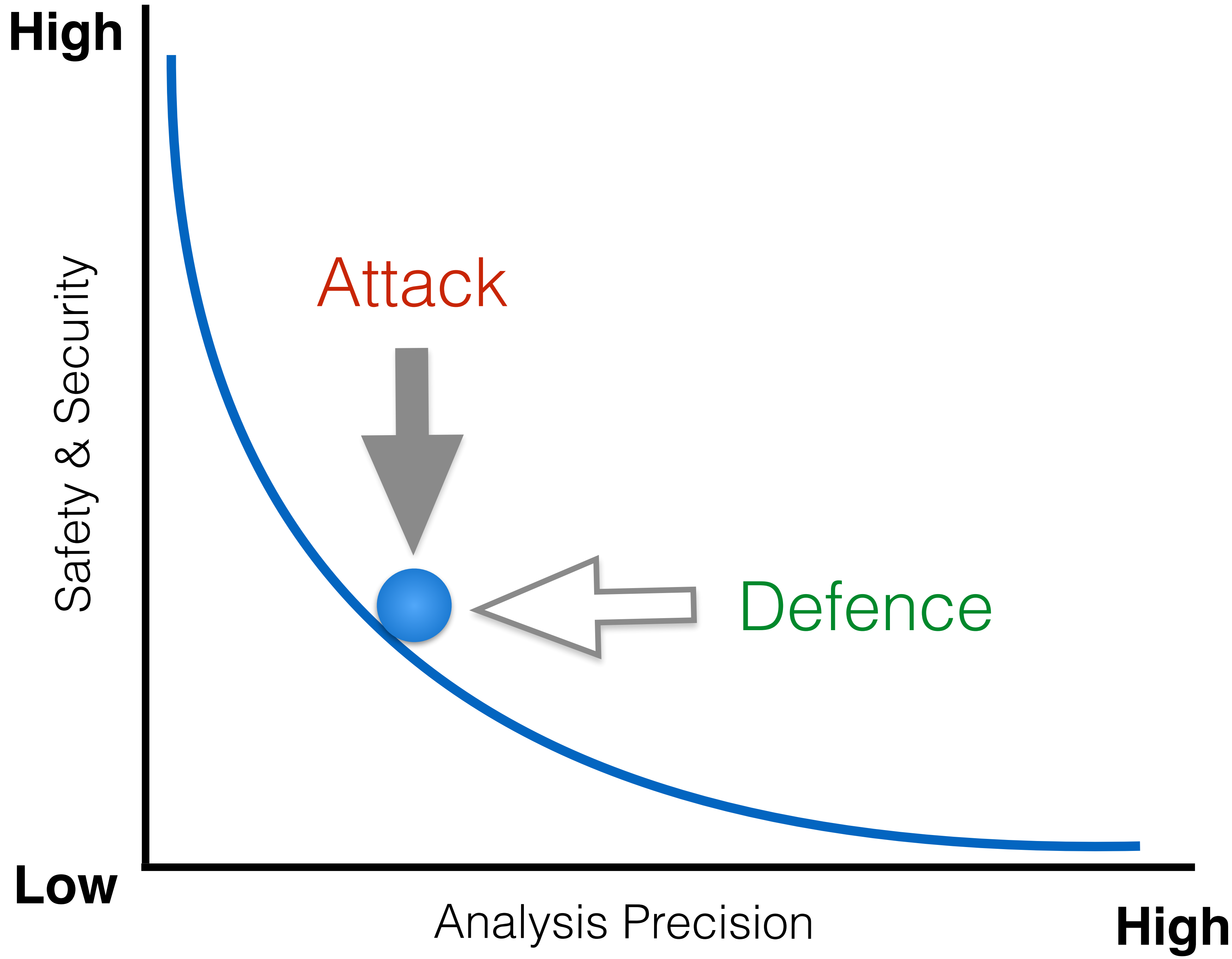
$$[\![P]\!] \;\cancel{\models}\; \varphi$$

Code Debugging



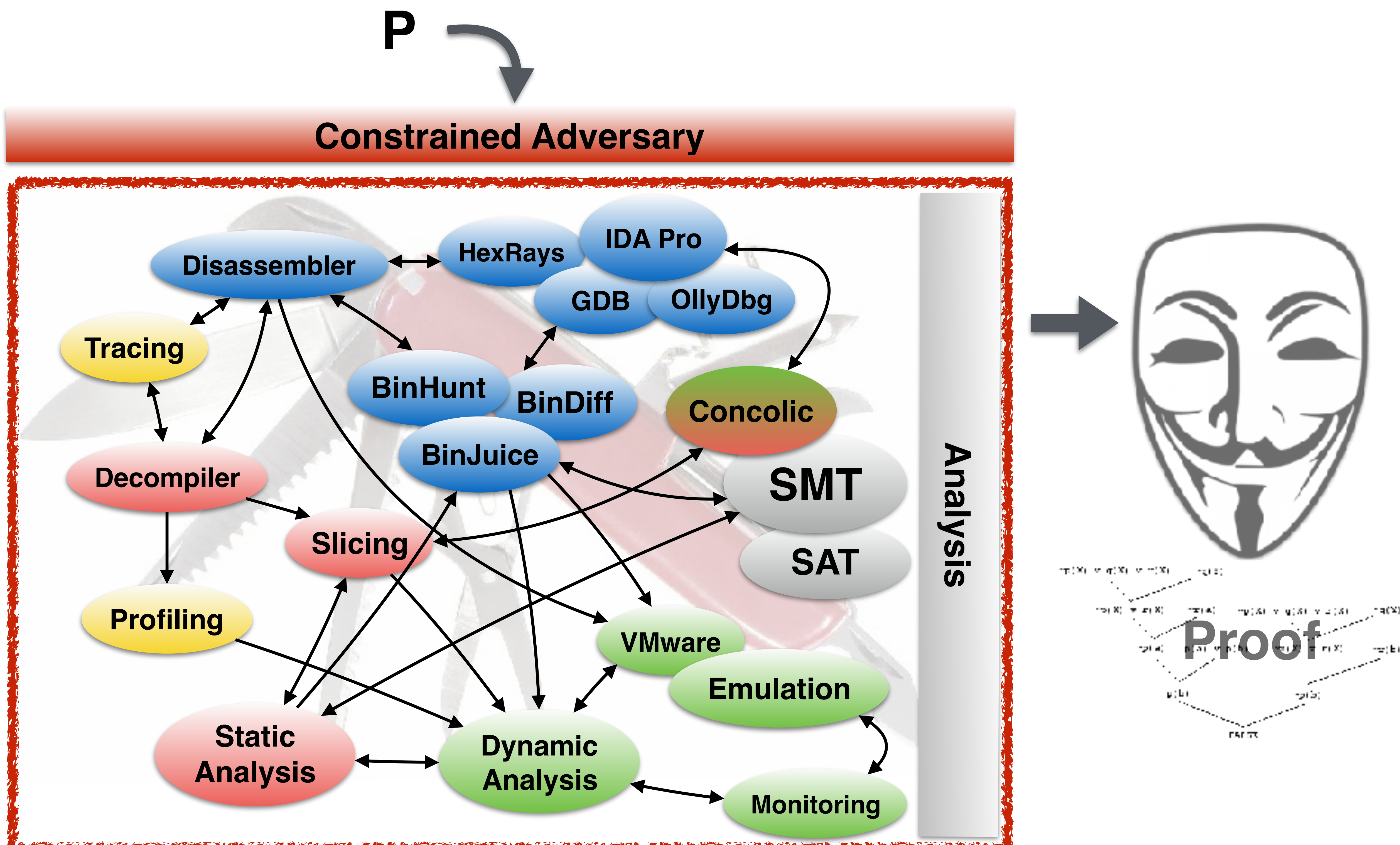
The attacker exploits bugs

Code Protection



The attacker reverse engineer code

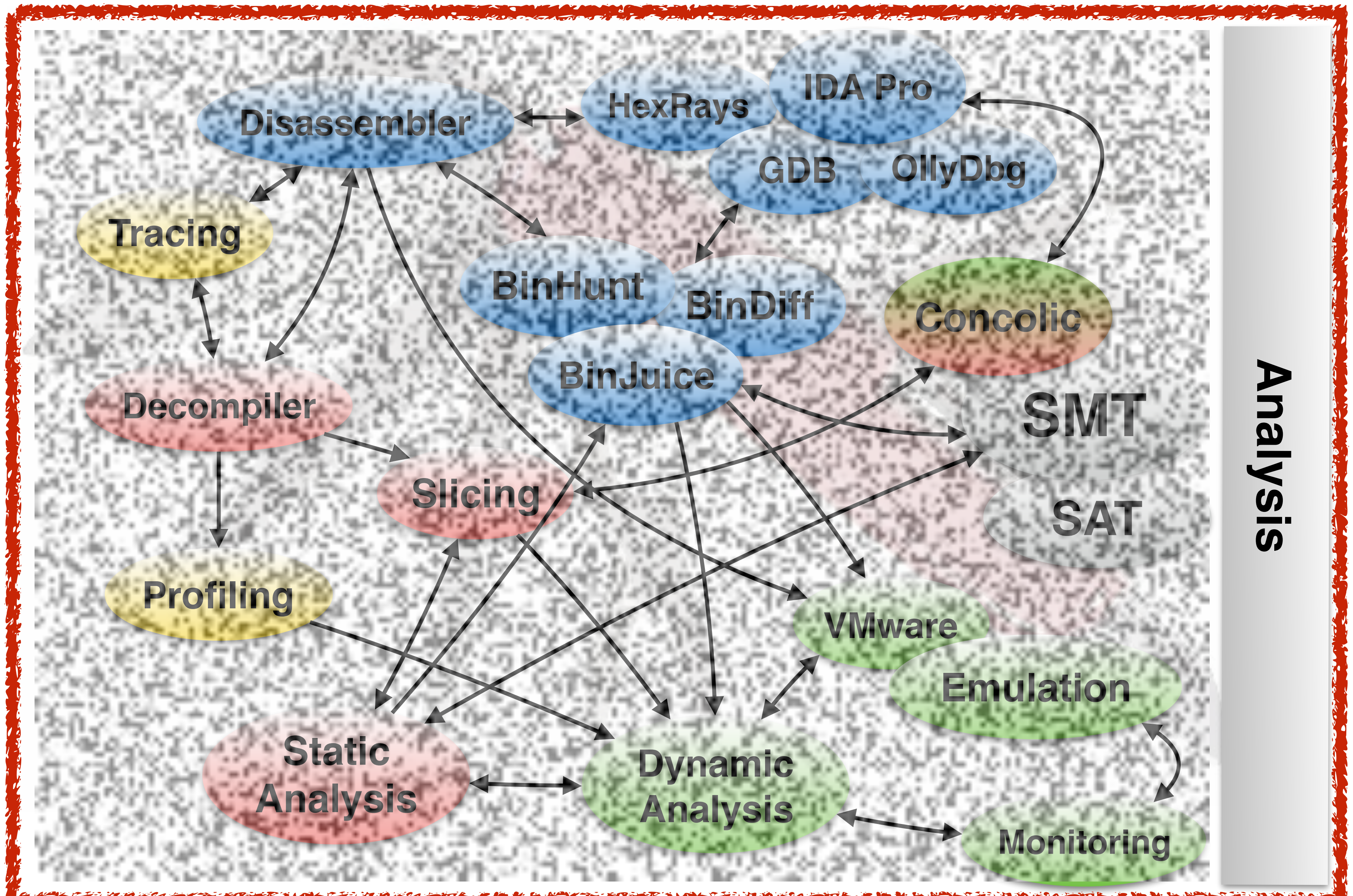
Understanding is Interpreting

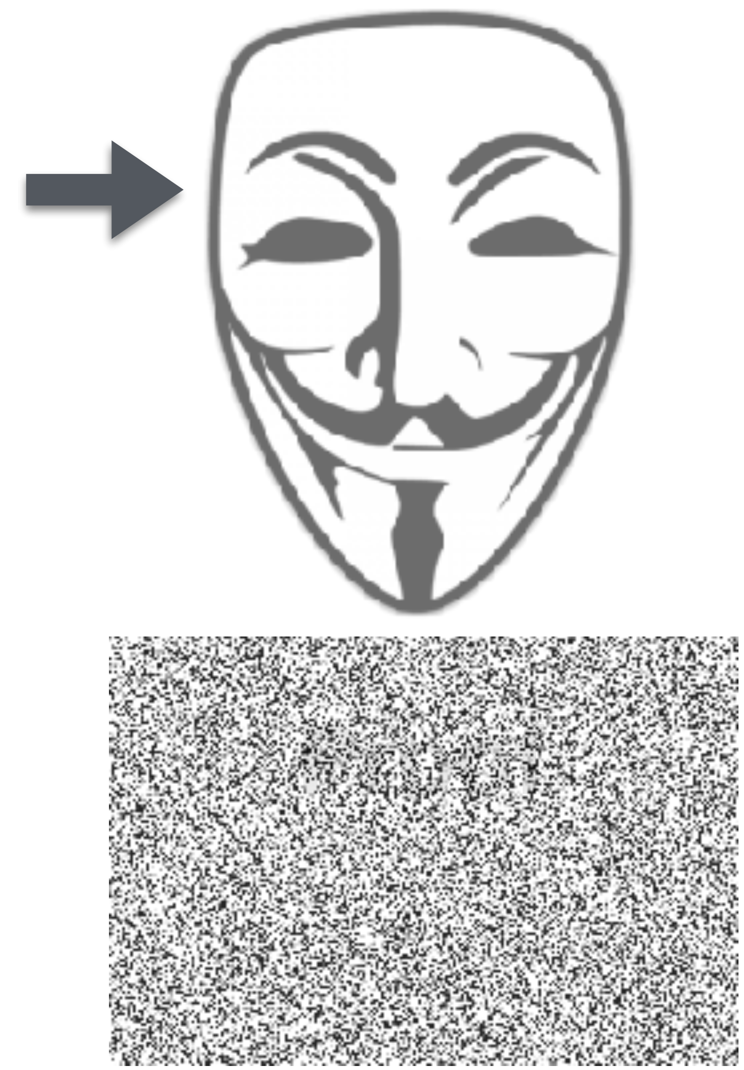


Protecting is obscuring Interpreters

(P)

Constrained Adversary



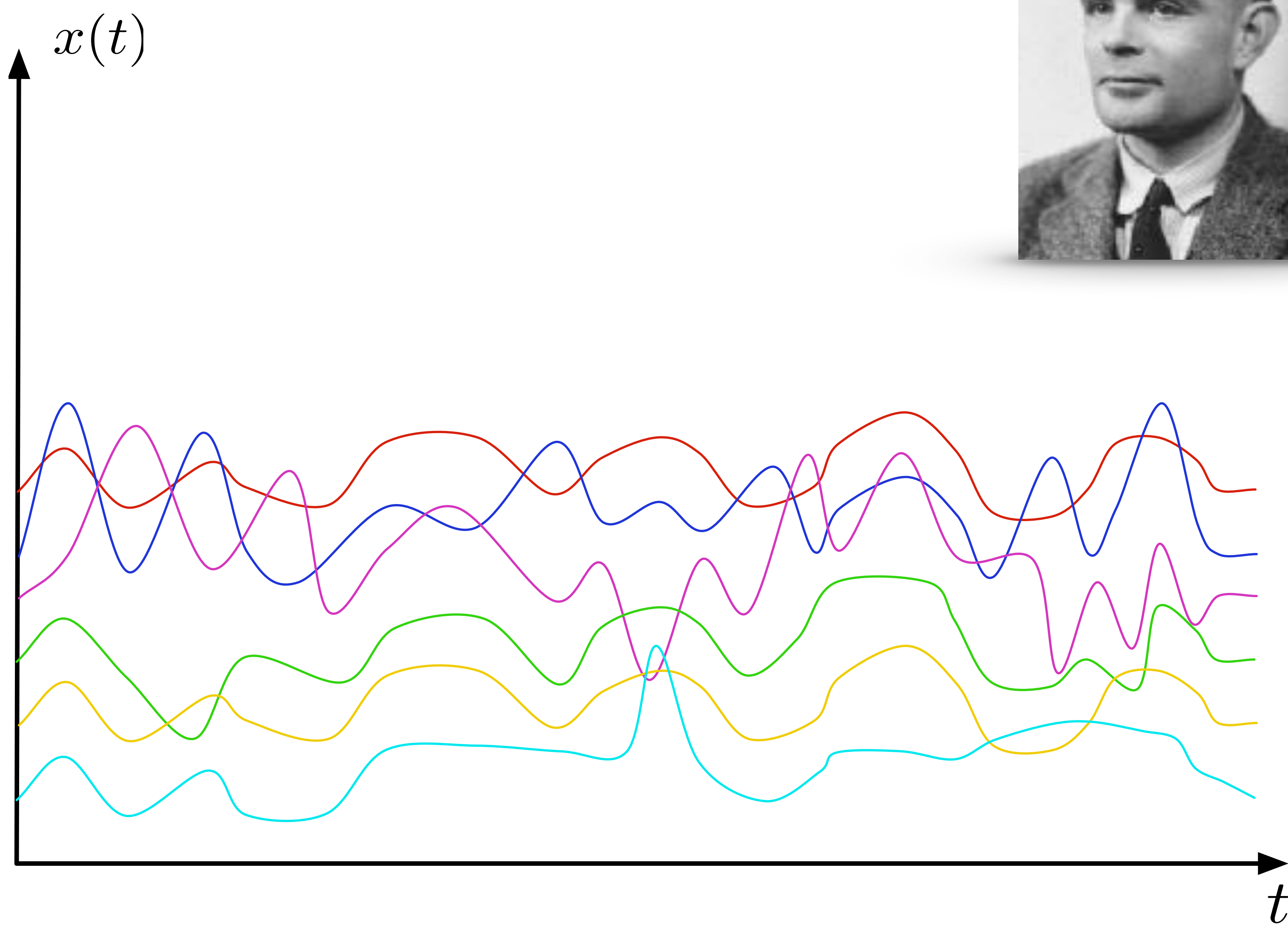


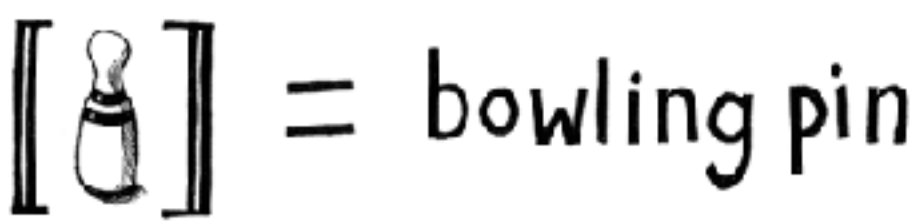
Can we build a theory in PL?

(outside crypto)

The Concrete Model

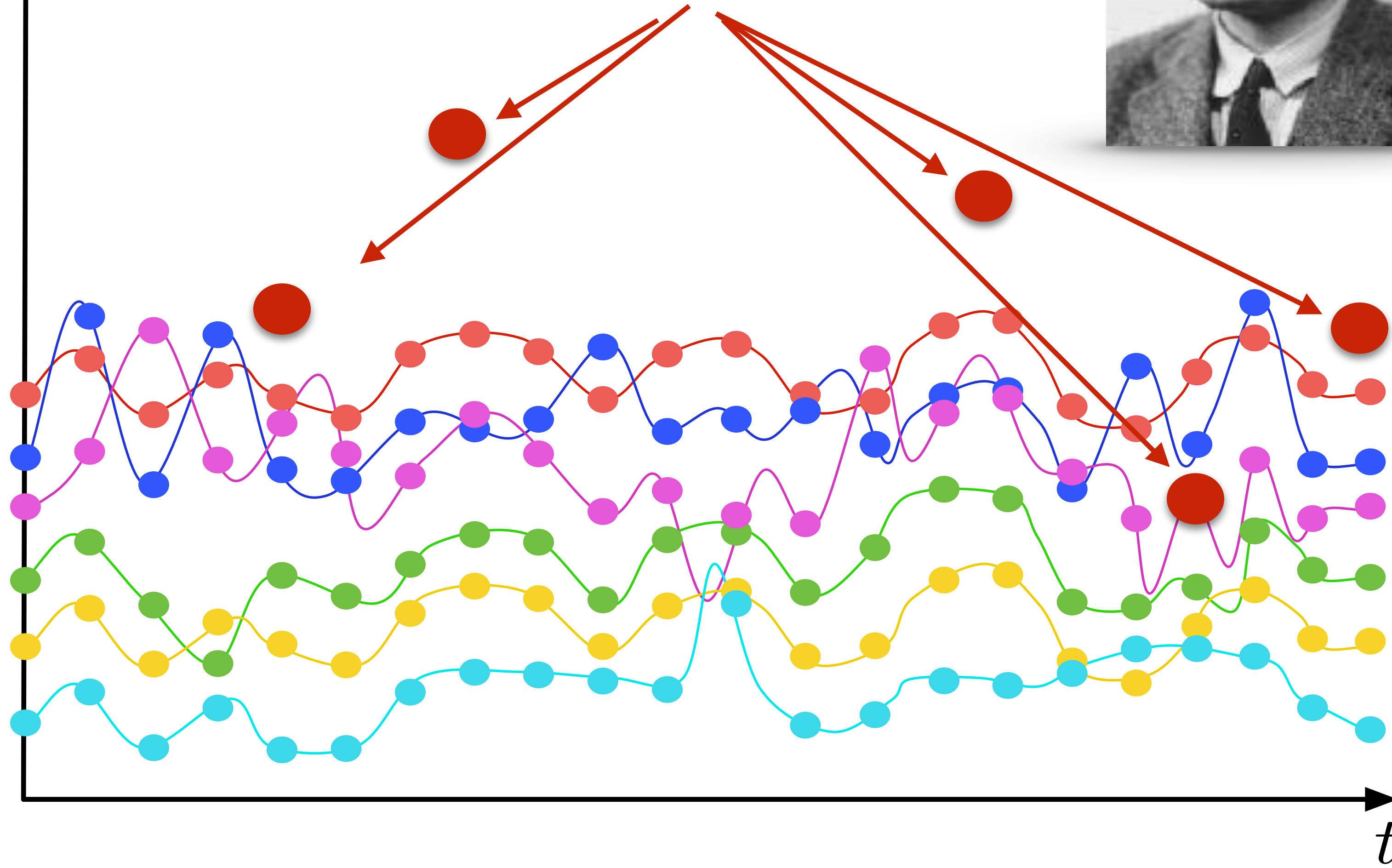
$$\llbracket P \rrbracket$$





The Concrete Model

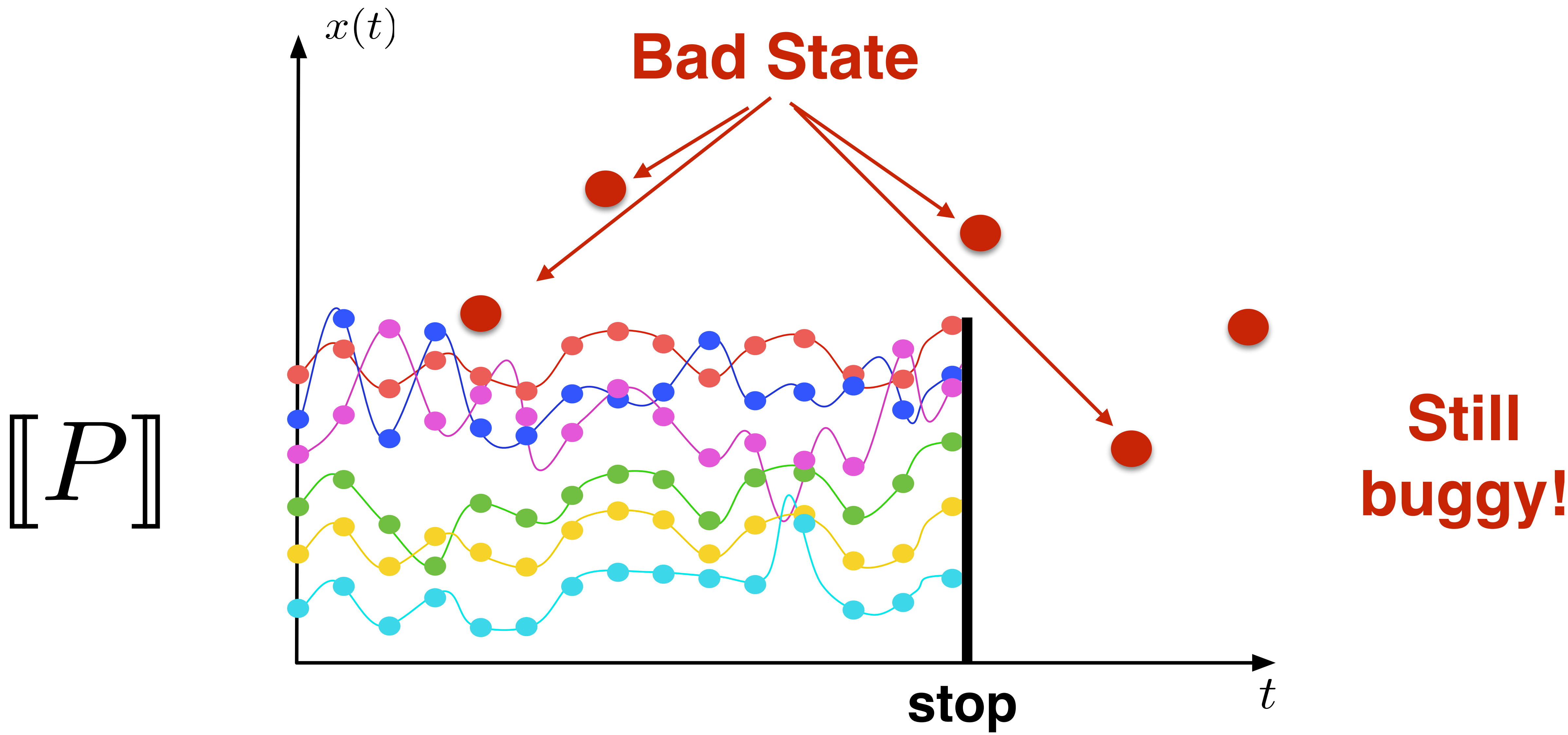
$\llbracket P \rrbracket$



1 bug!

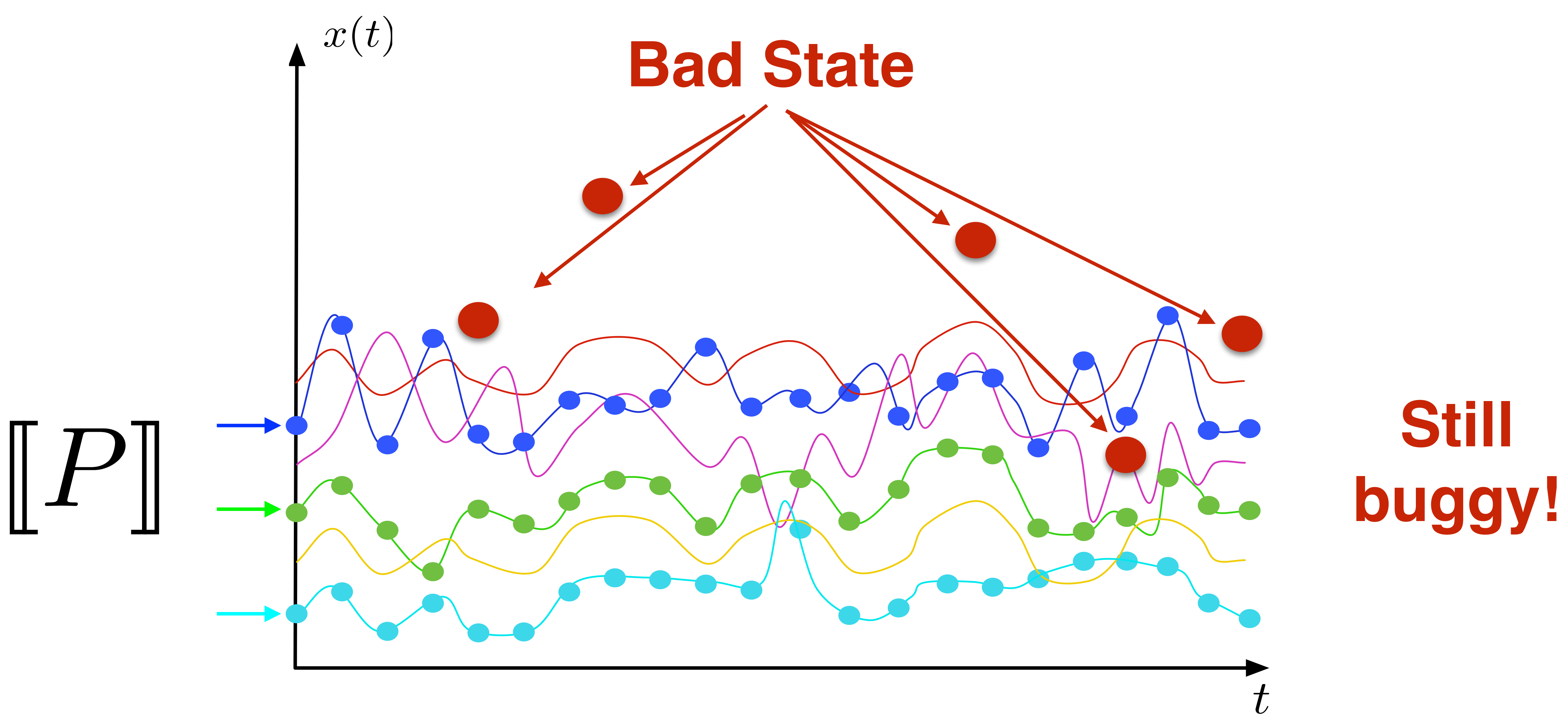
We need computers to reason about computers

Partial Execution



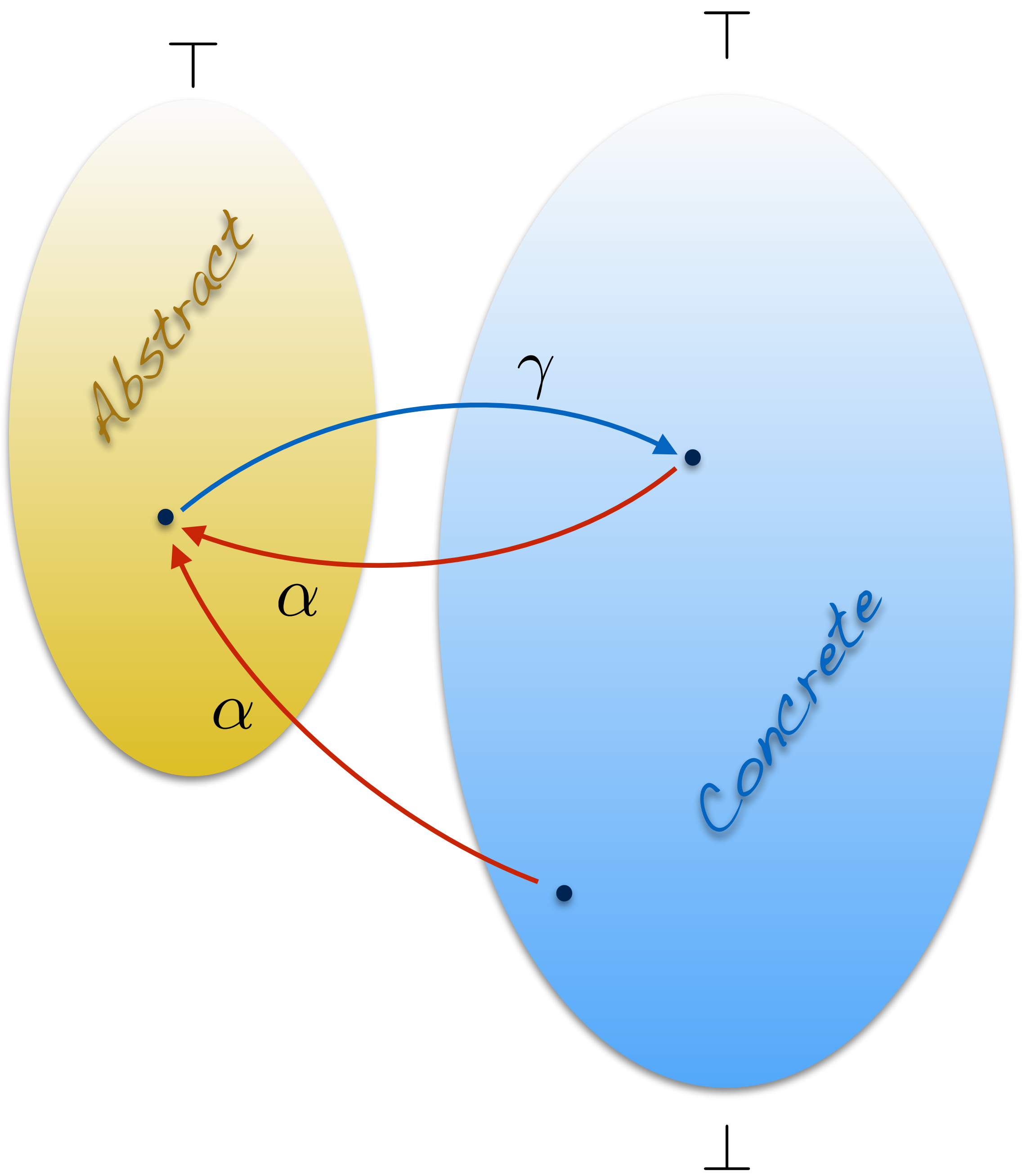
Cheap, efficient, but unsound!!!

Testing & Dynamic analysis



Efficient but unsound!

The idea of Abstraction



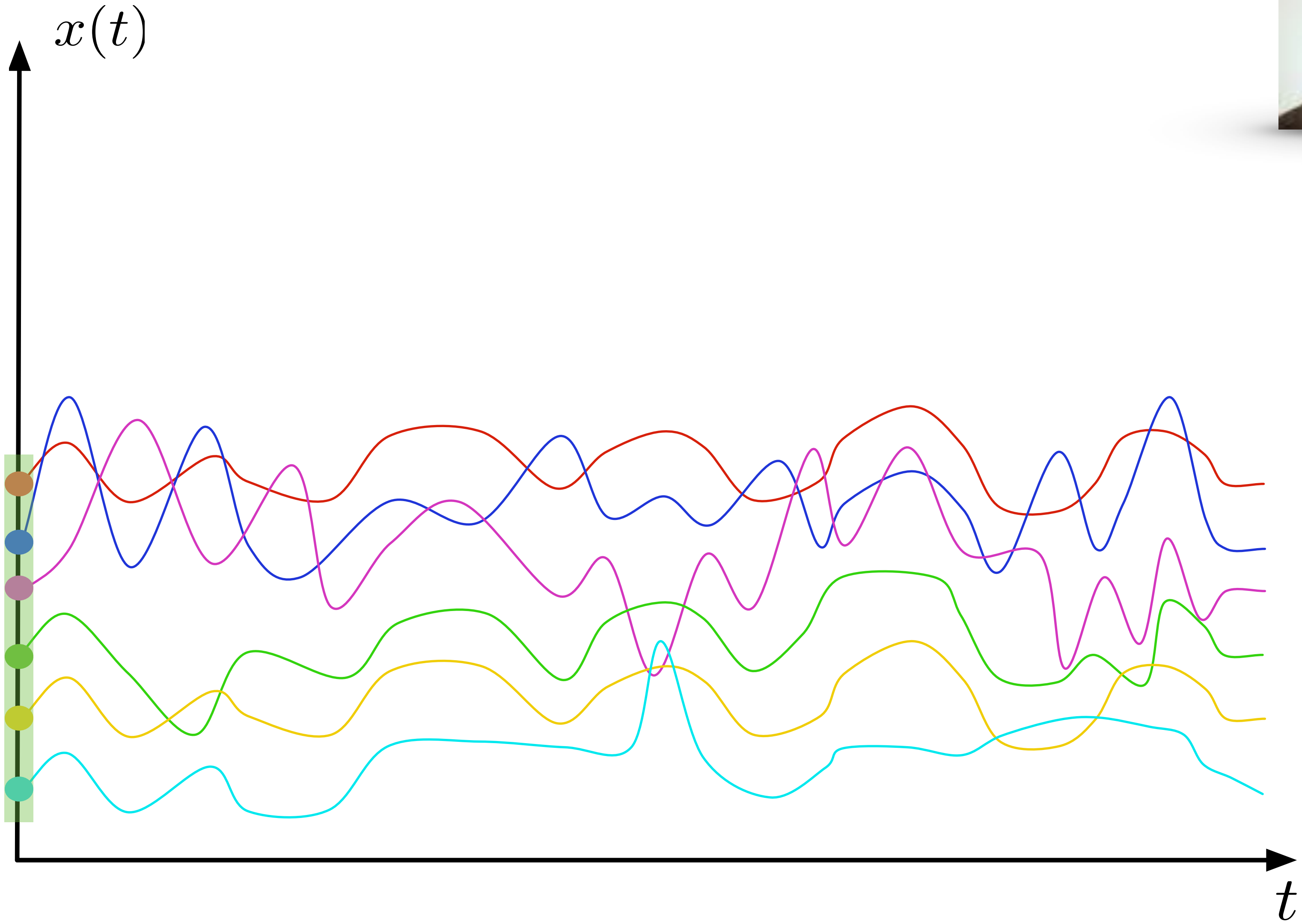
α & γ

Abstractions are
Galois Connections between
Complete Lattices of
concrete/abstract denotations

Abstract Interpretation by Cousot & Cousot ACM POPL 1977

Abstracting the Model

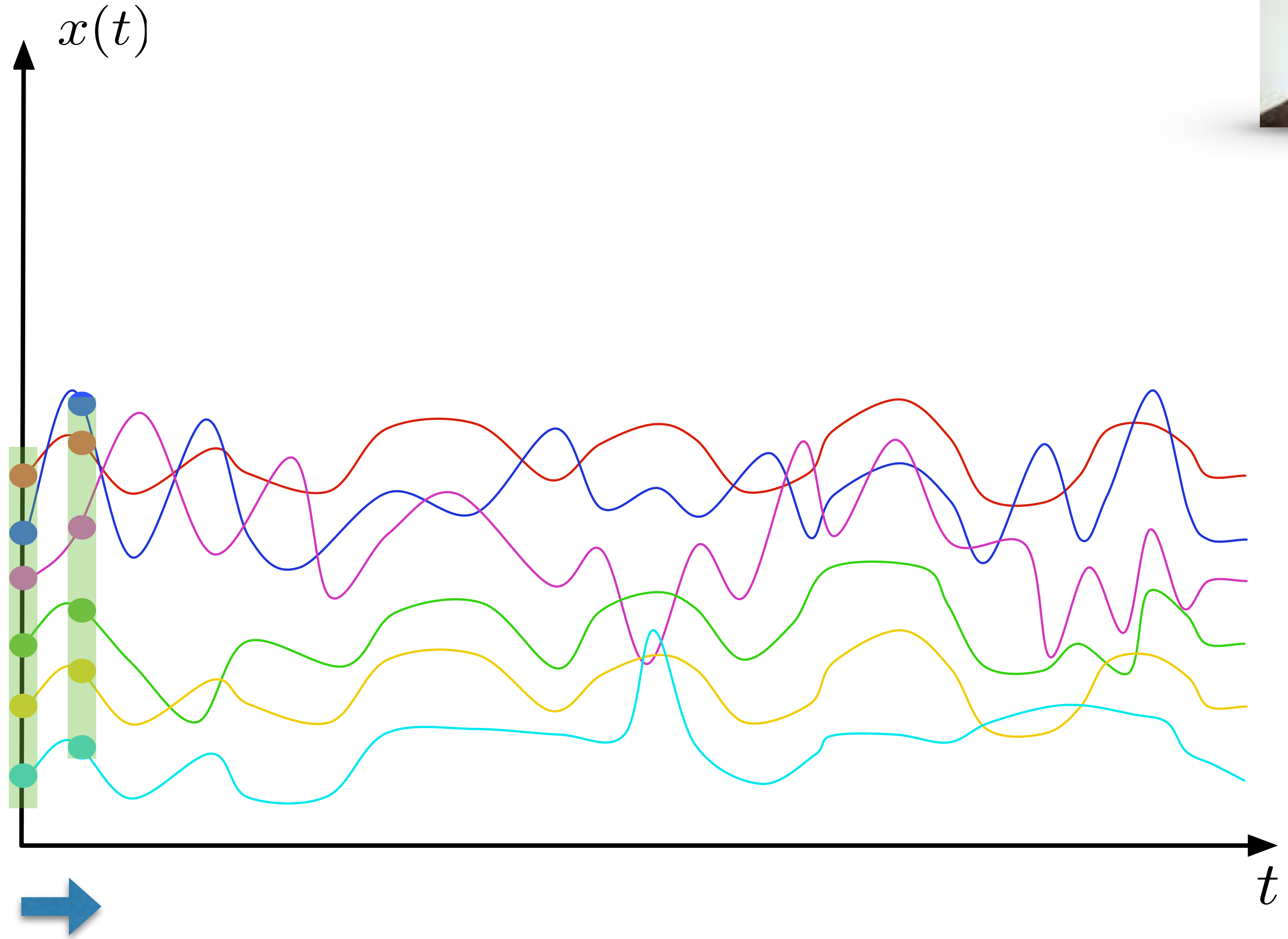
$\alpha([\![P]\!])$



Still too complicated, complex, undecidable

Abstracting the Model

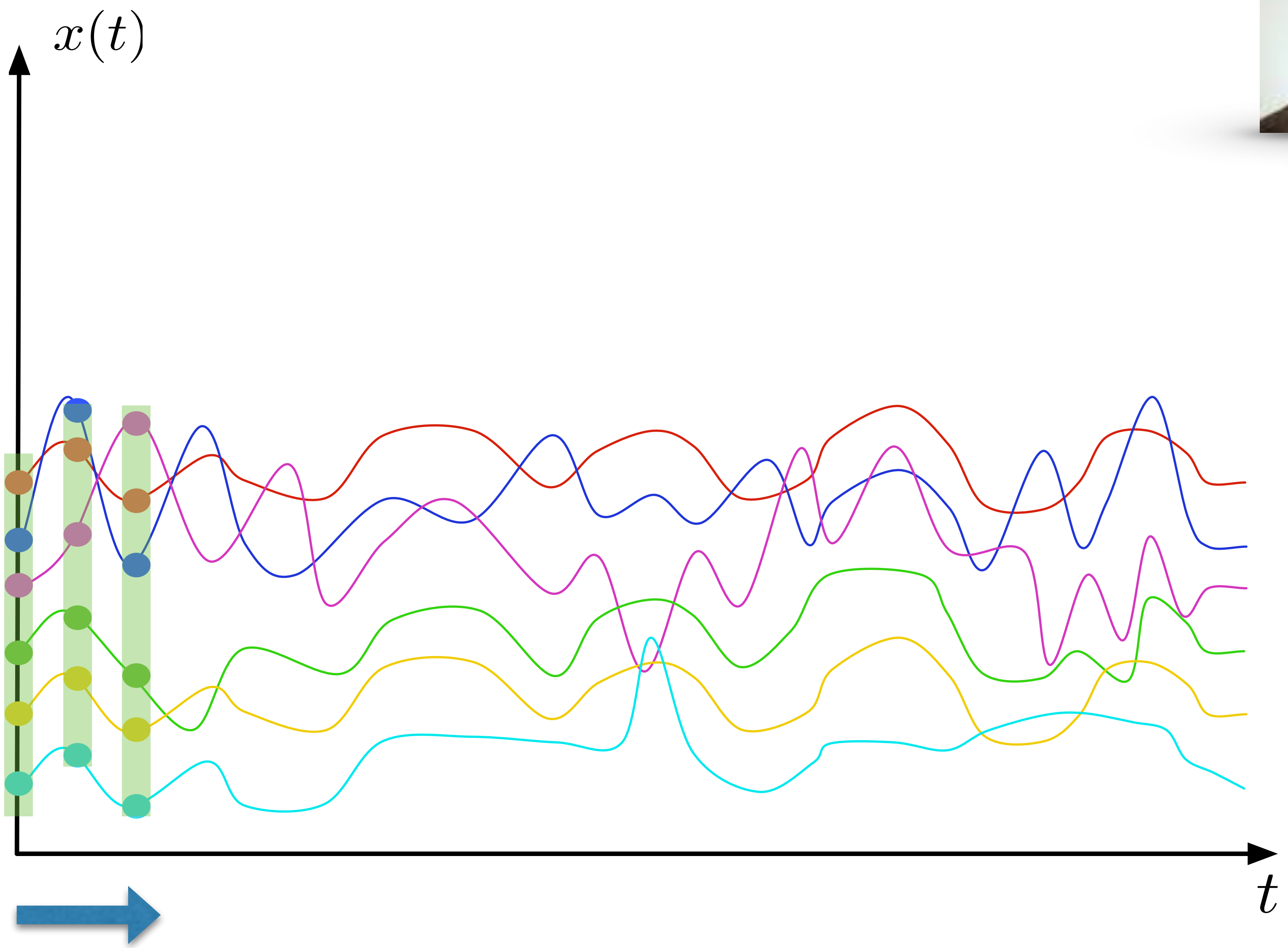
$$\alpha(\llbracket P \rrbracket)$$



Still too complicated, complex, undecidable

Abstracting the Model

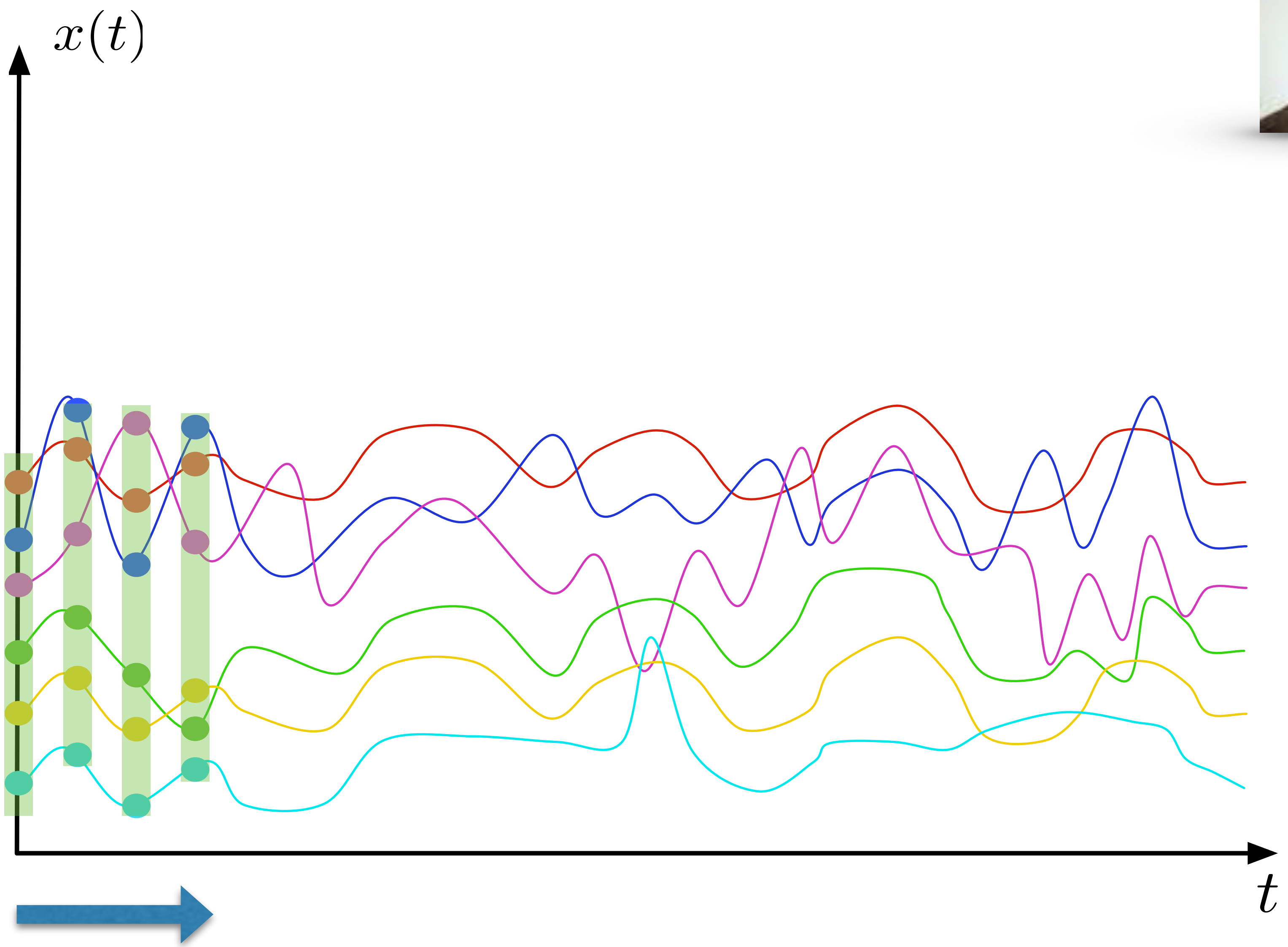
$$\alpha(\llbracket P \rrbracket)$$



Still too complicated, complex, undecidable

Abstracting the Model

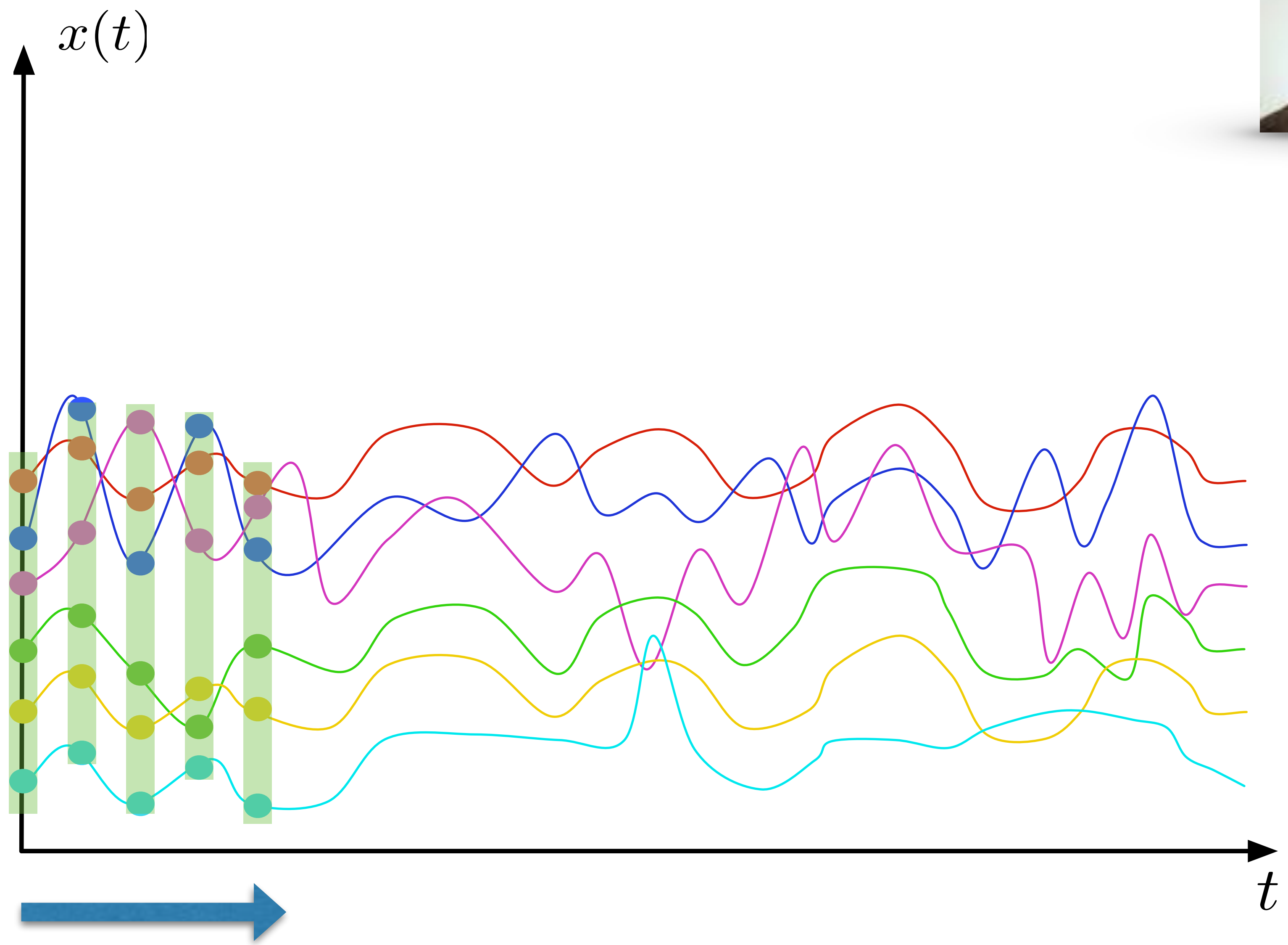
$$\alpha(\llbracket P \rrbracket)$$



Still too complicated, complex, undecidable

Abstracting the Model

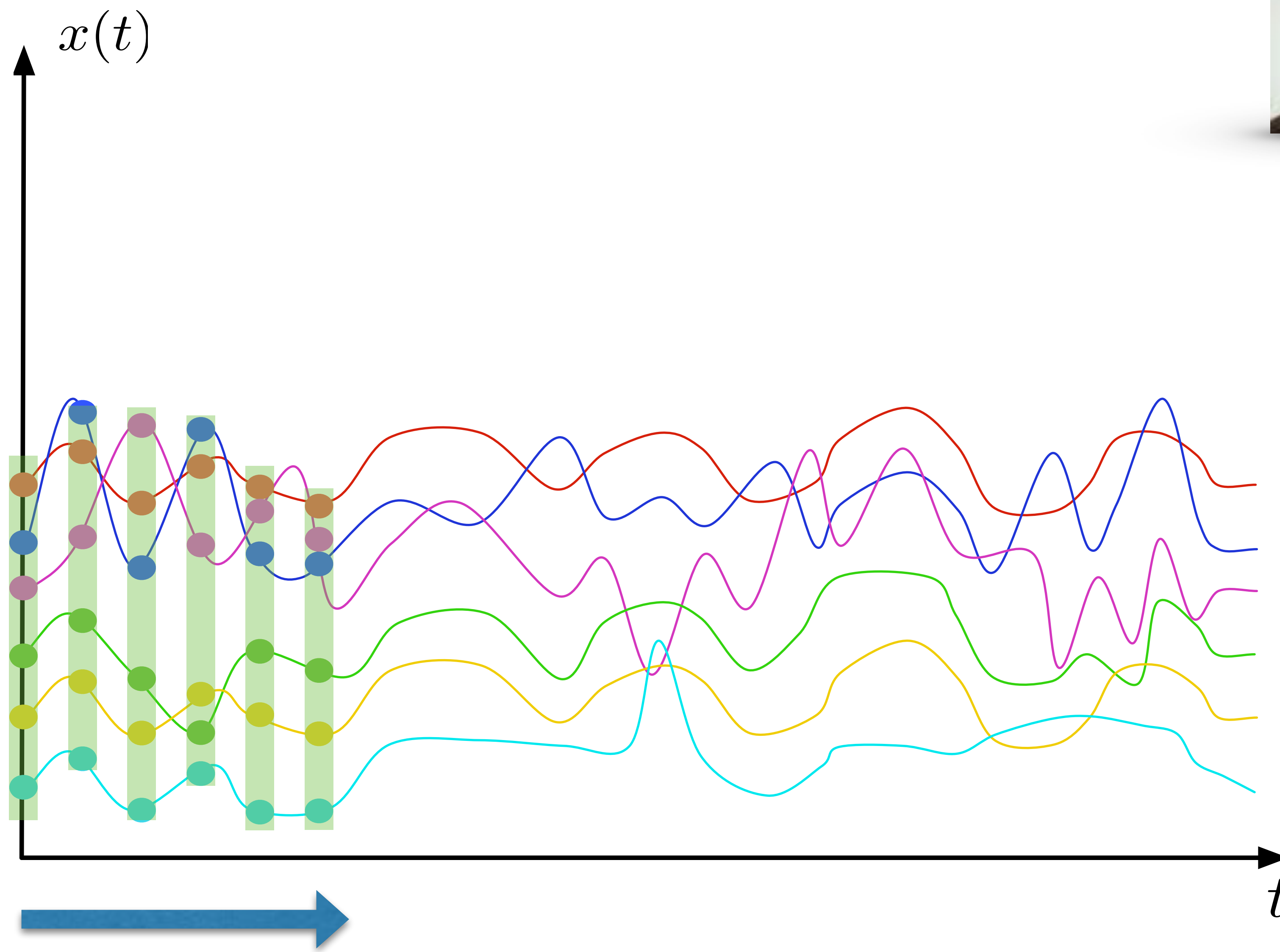
$\alpha(\llbracket P \rrbracket)$



Still too complicated, complex, undecidable

Abstracting the Model

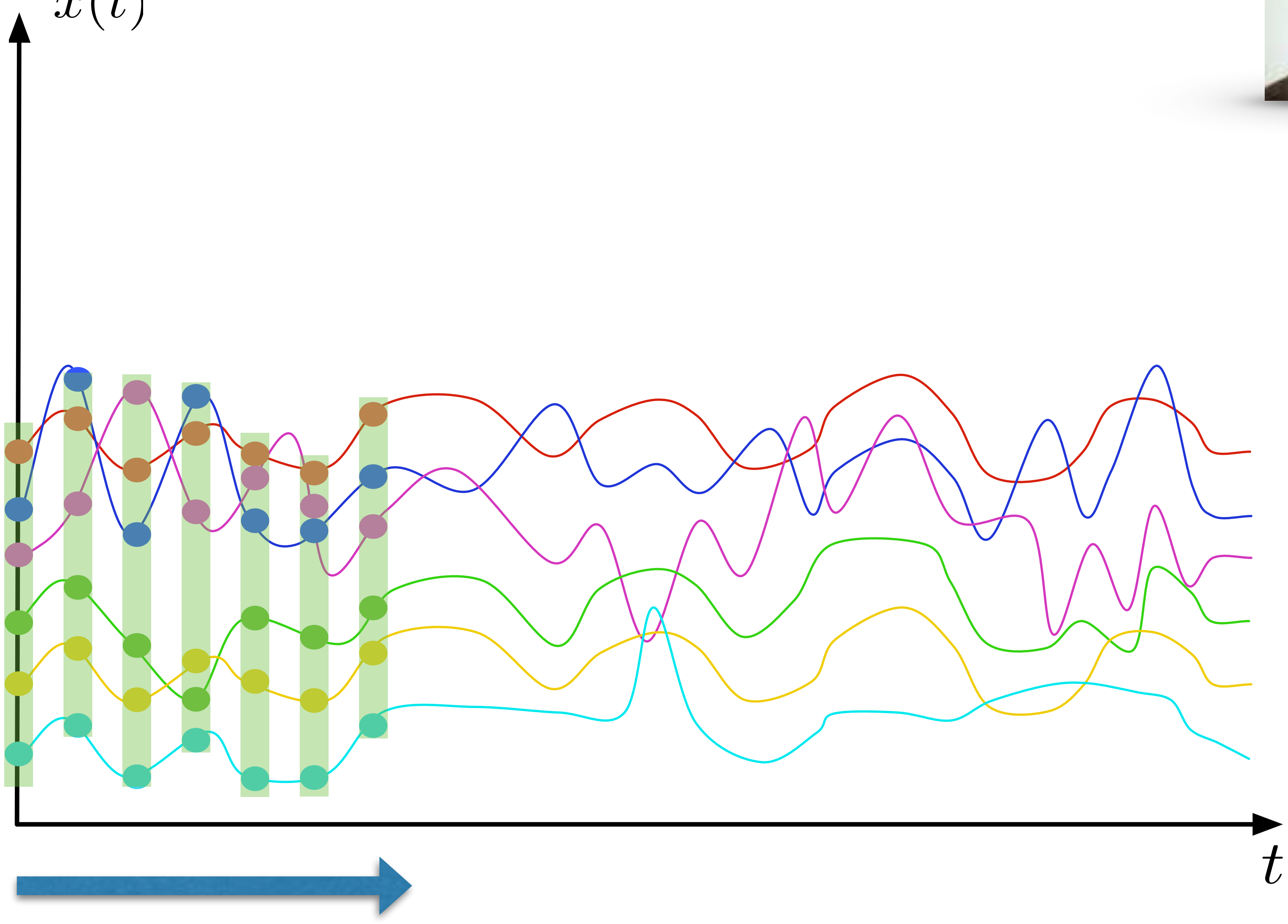
$$\alpha(\llbracket P \rrbracket)$$



Still too complicated, complex, undecidable

Abstracting the Model

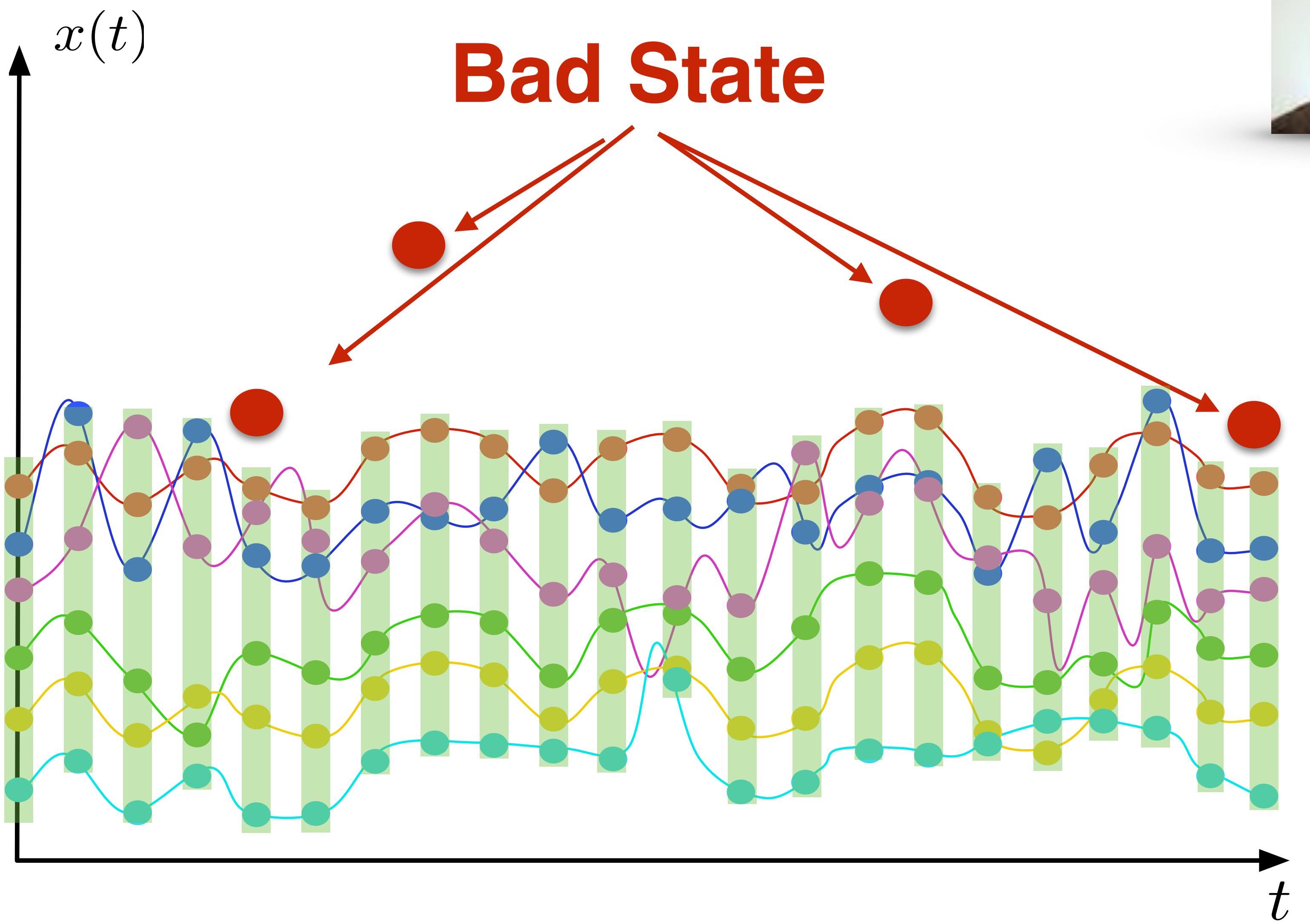
$$\alpha(\llbracket P \rrbracket)$$



Still too complicated, complex, undecidable

Abstracting the Model

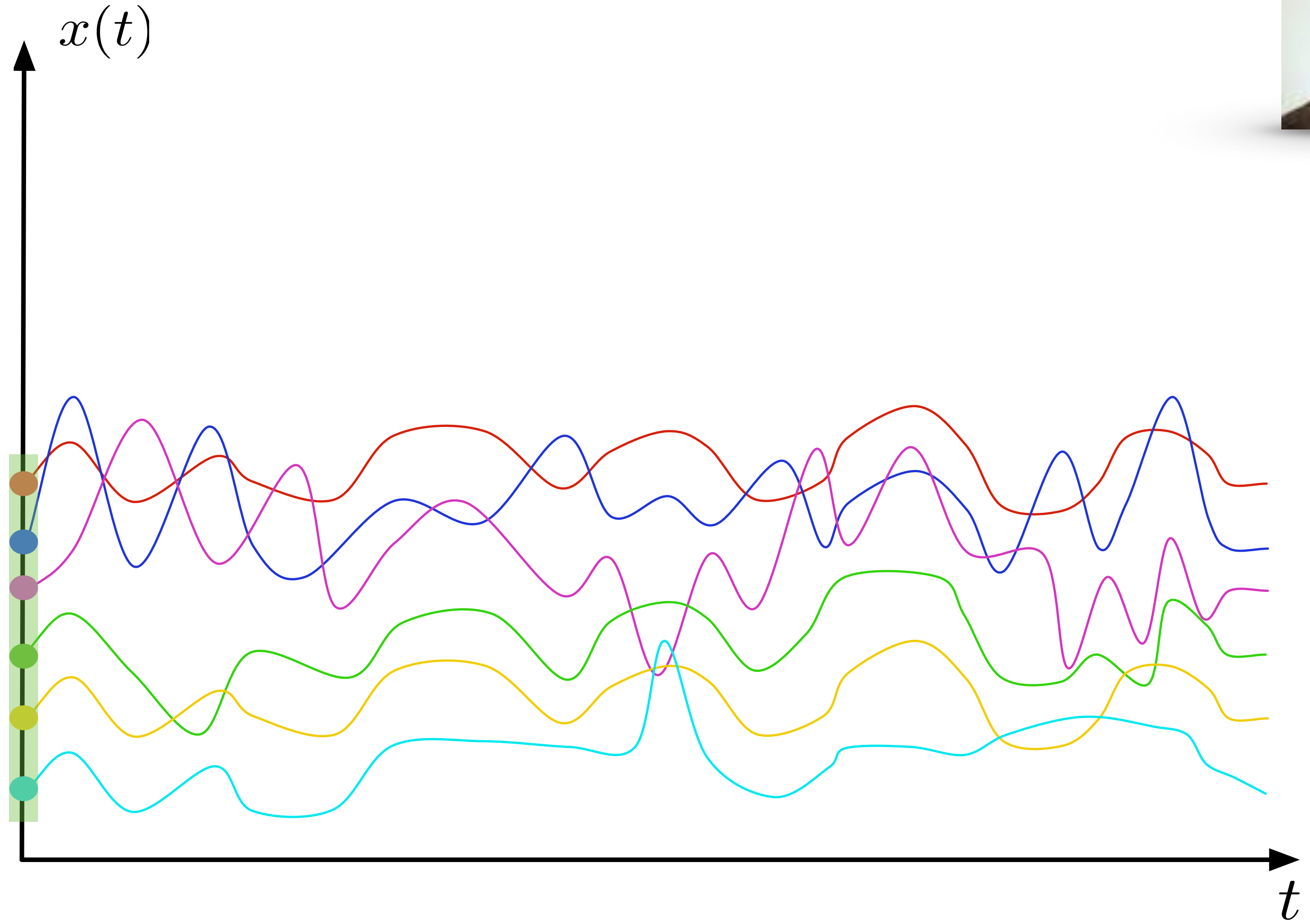
$$\alpha(\llbracket P \rrbracket)$$



No bug!

This is NOT Abstract Interpretation!!!

Abstract Interpretation



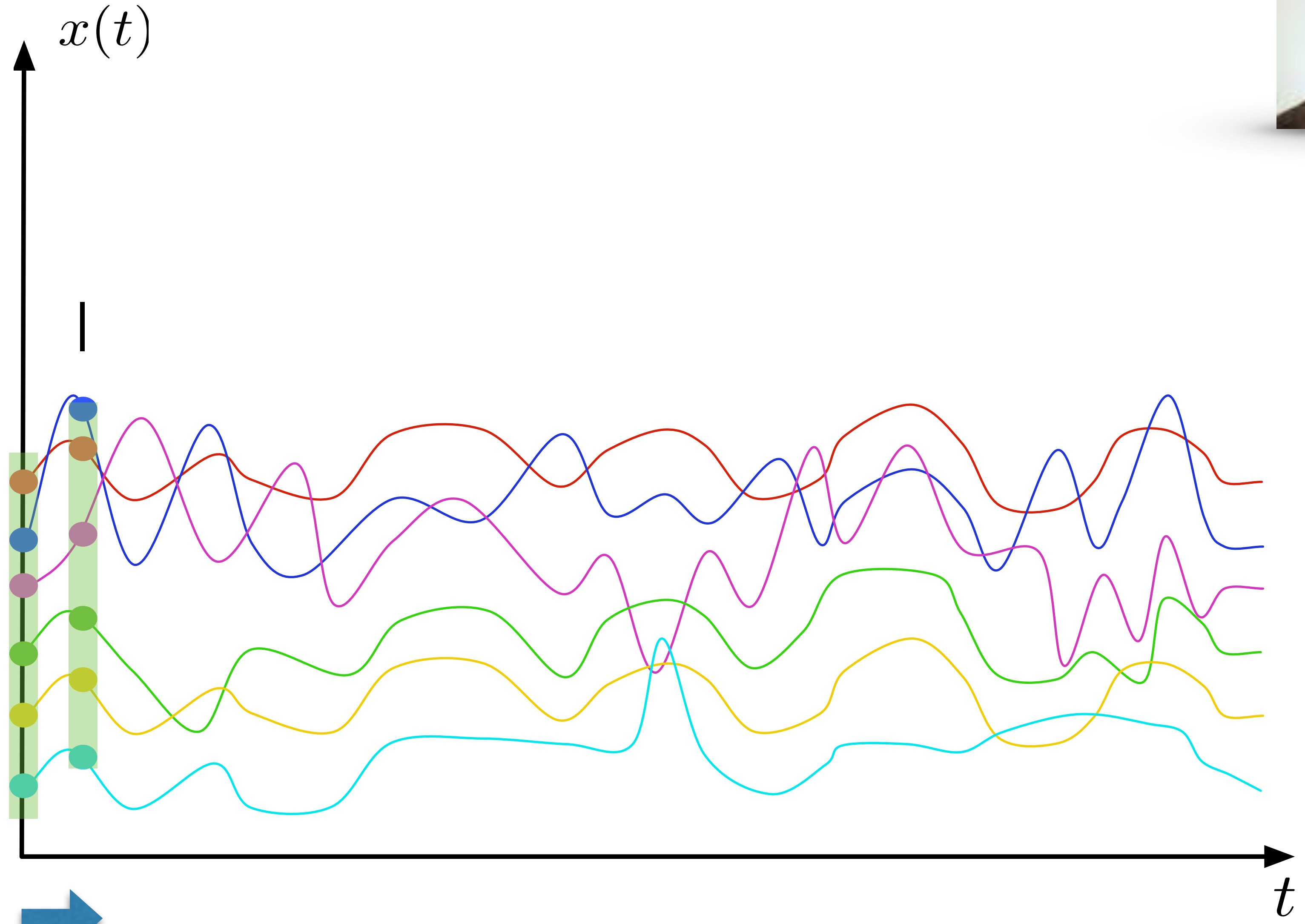
$$[\![P]\!]^{\alpha}$$

Affordable (sound) loss of precision

Abstract Interpretation by Cousot & Cousot ACM POPL 1977

Abstract Interpretation

$[\![P]\!]^{\alpha}$

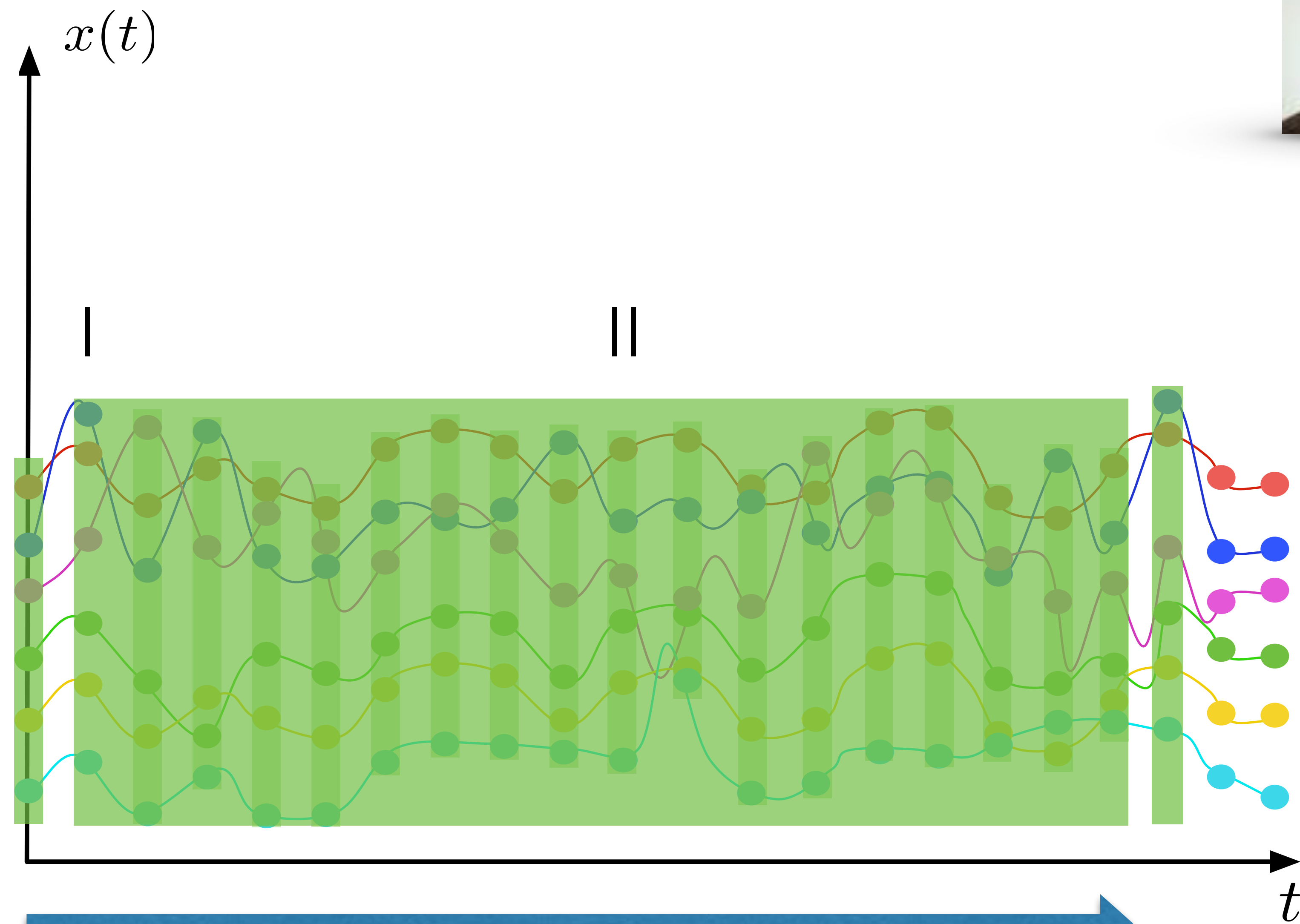


Affordable (sound) loss of precision

Abstract Interpretation by Cousot & Cousot ACM POPL 1977

Abstract Interpretation

$$[\![P]\!]^{\alpha}$$

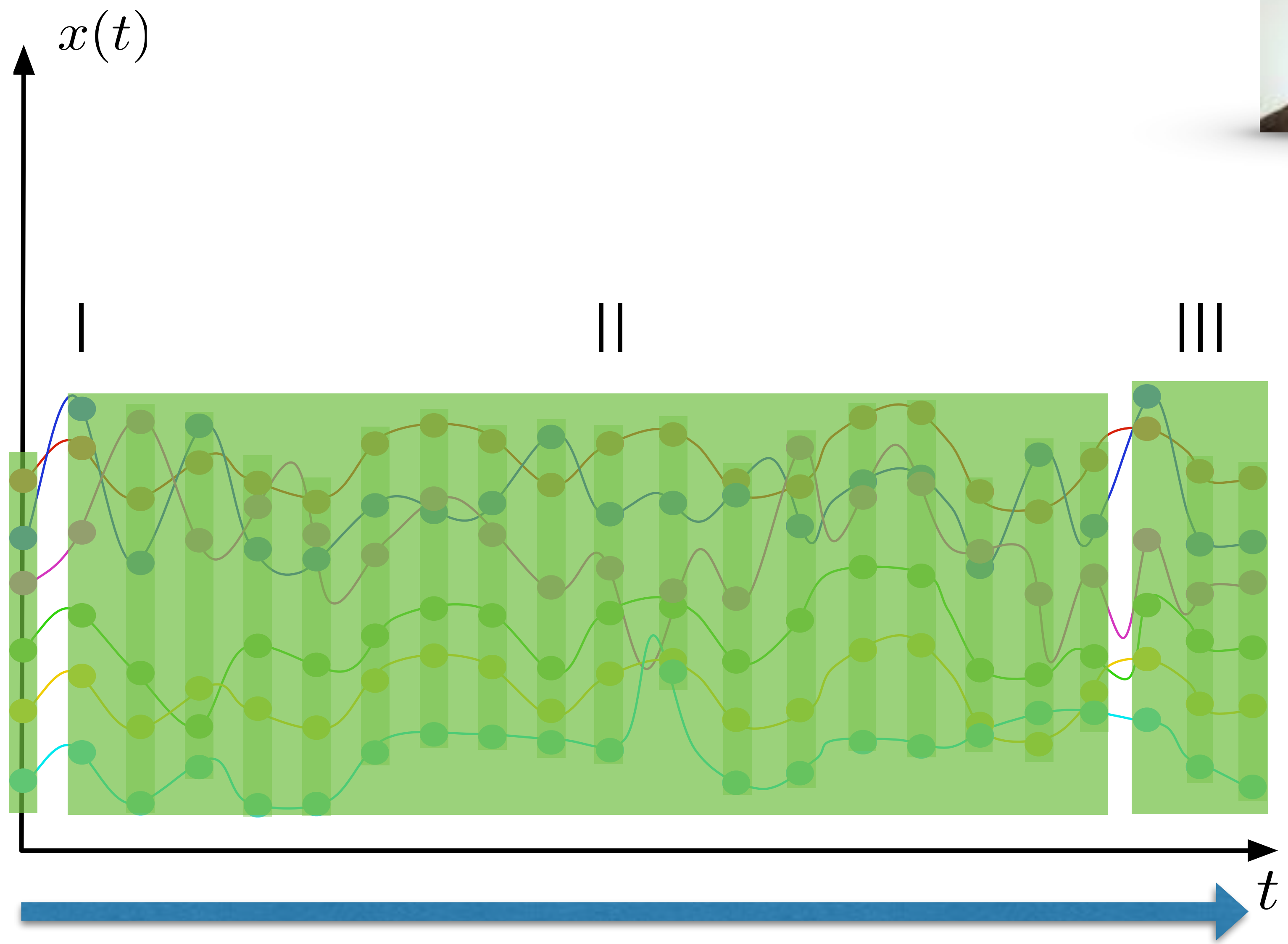


Affordable (sound) loss of precision

Abstract Interpretation by Cousot & Cousot ACM POPL 1977

Abstract Interpretation

$$[\![P]\!]^{\alpha}$$

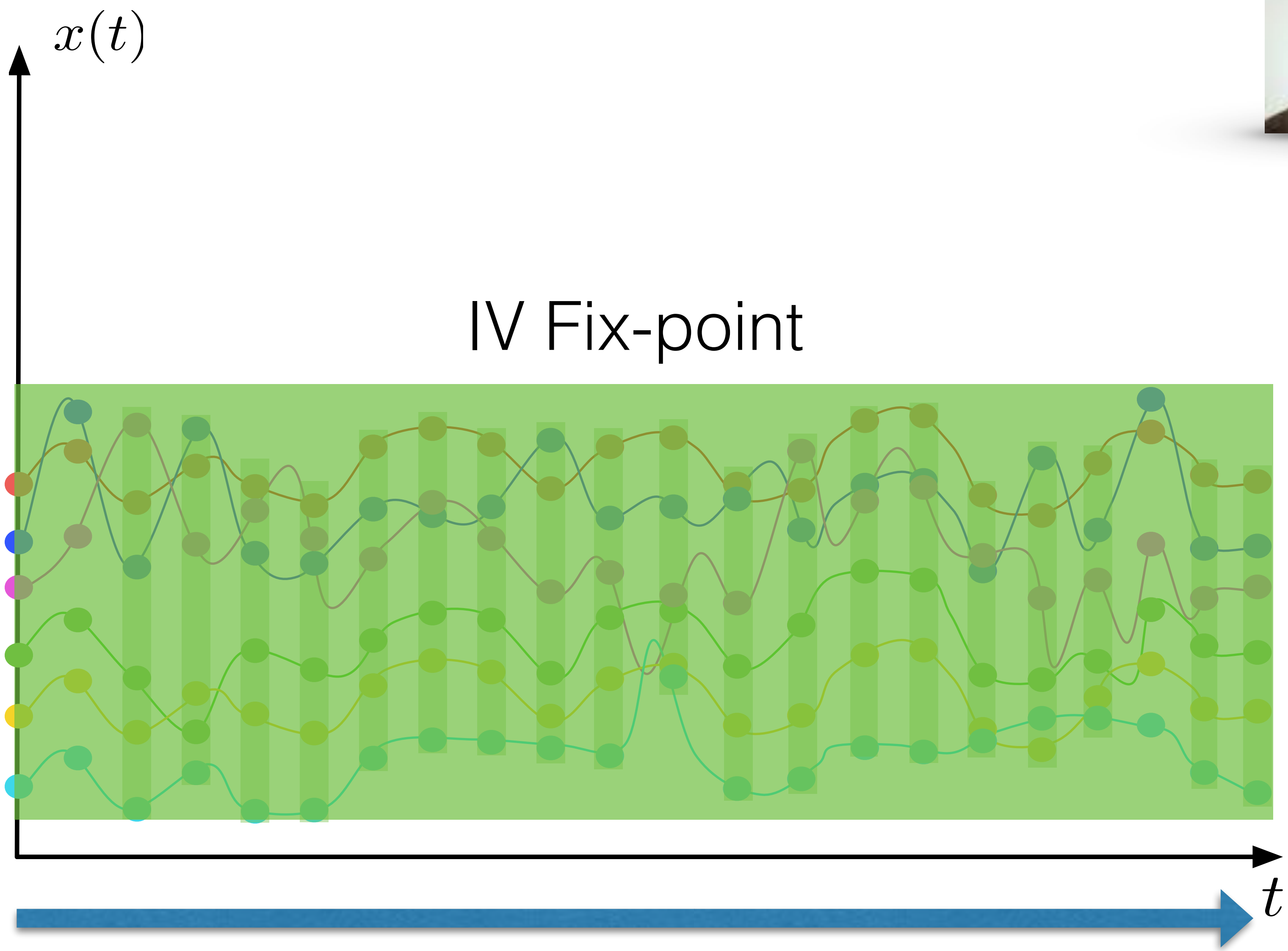


Affordable (sound) loss of precision

Abstract Interpretation by Cousot & Cousot ACM POPL 1977

Abstract Interpretation

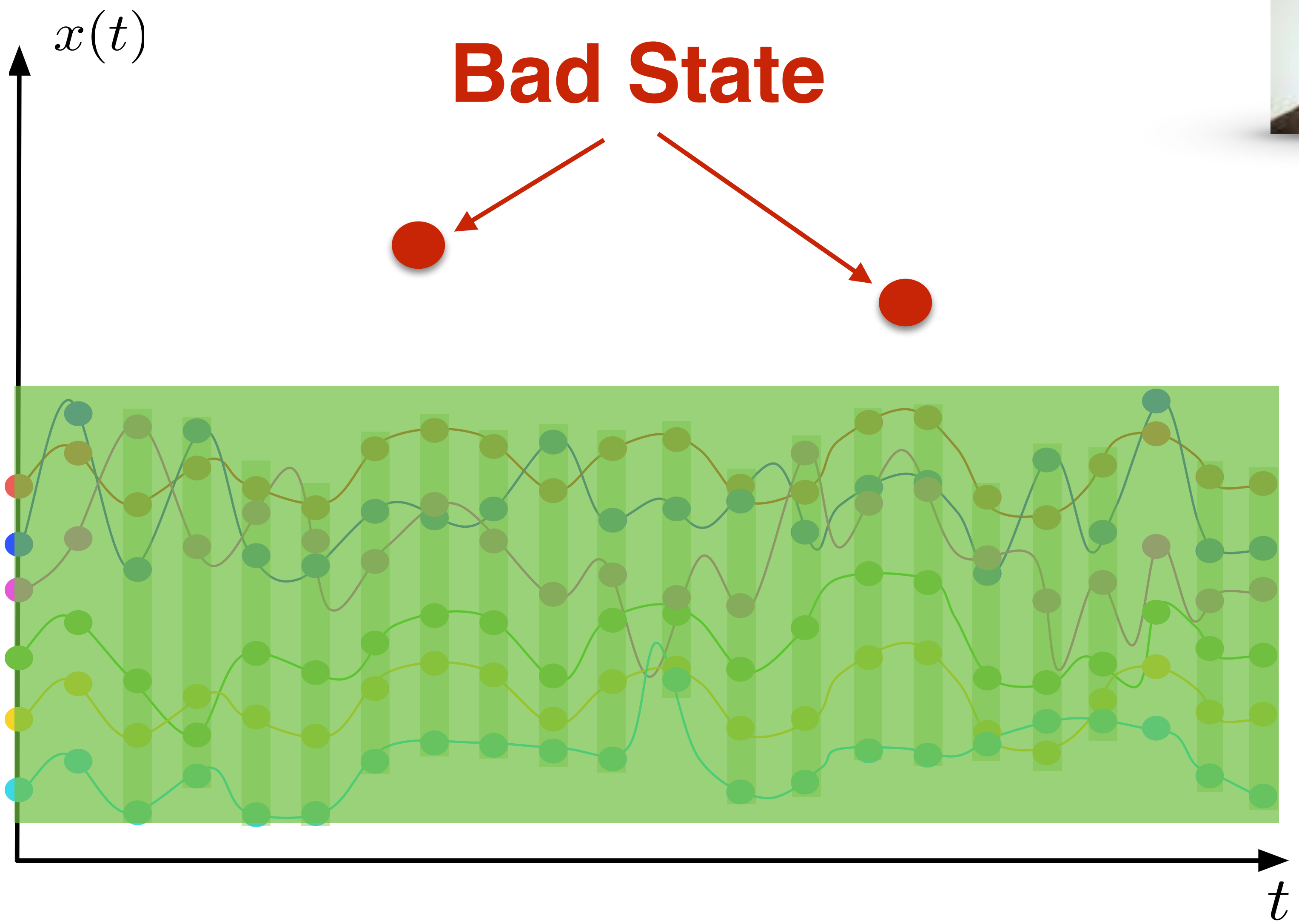
$$[\![P]\!]^{\alpha}$$



Affordable (sound) loss of precision

Abstract Interpretation by Cousot & Cousot ACM POPL 1977

Soundness



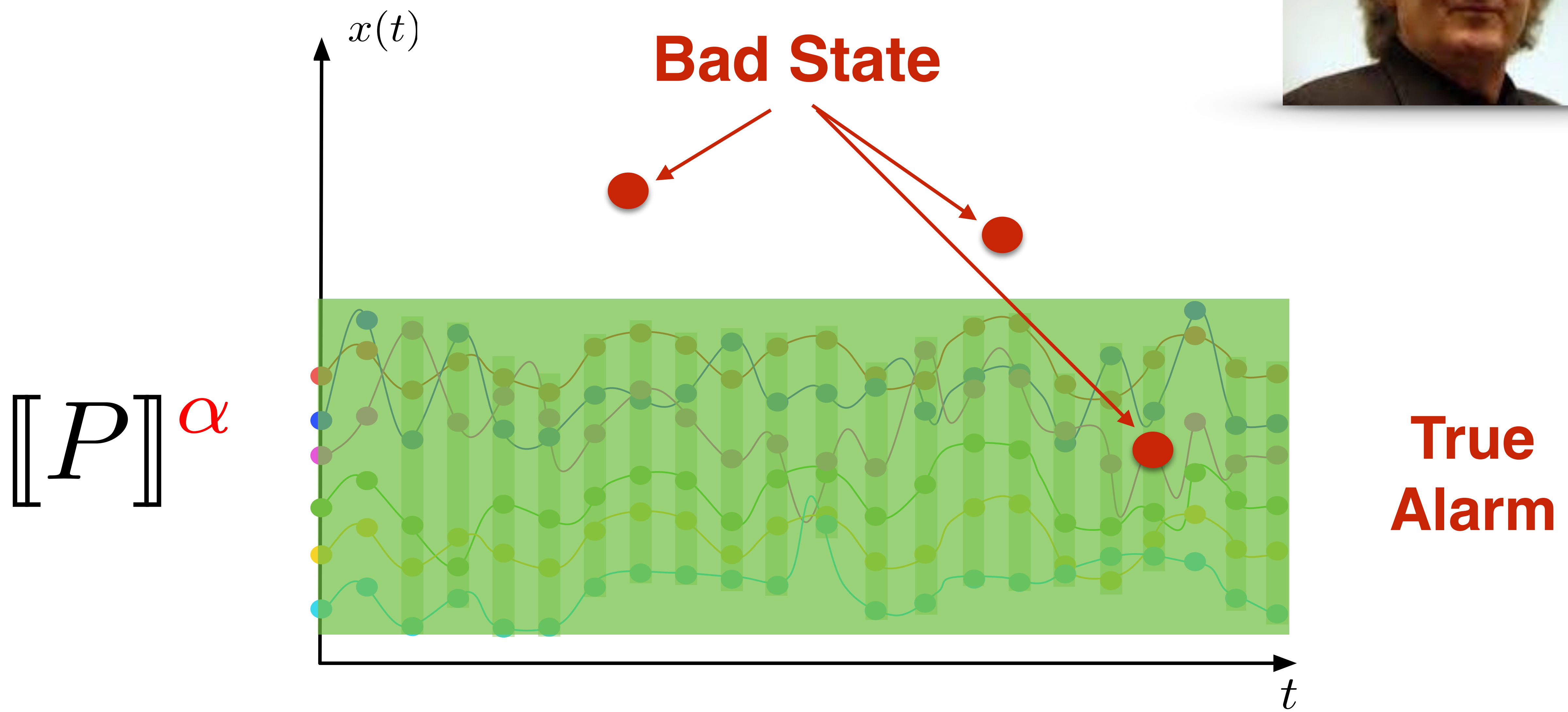
$$[\![P]\!]^{\alpha}$$

Guaranteed Security

Affordable (sound) loss of precision

$$\alpha([\![P]\!]) \subseteq [\![P]\!]^{\alpha}$$

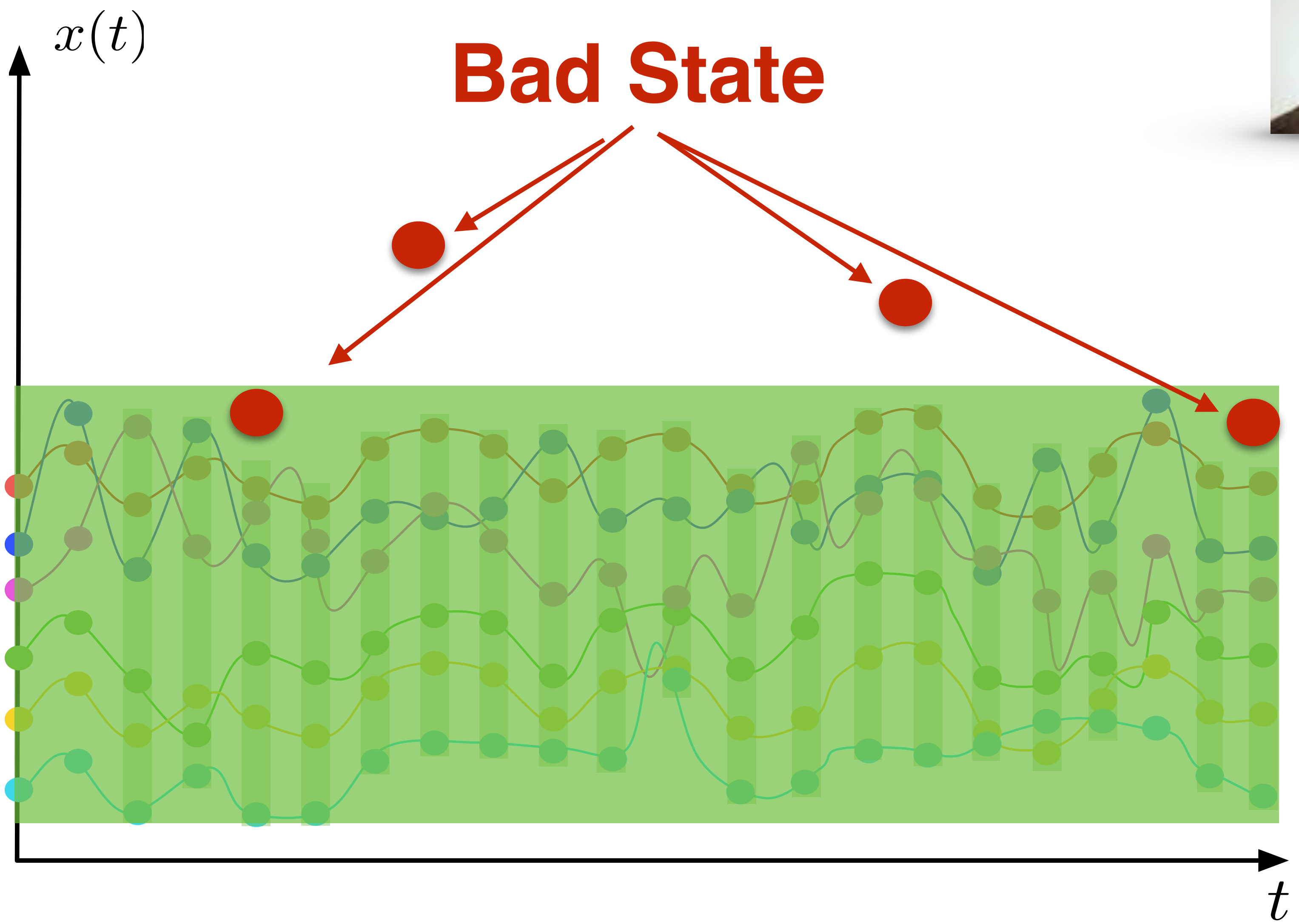
Soundness



Affordable (sound) loss of precision

$$\alpha([\![P]\!]) \subseteq [\![P]\!]^{\alpha}$$

(In)completeness



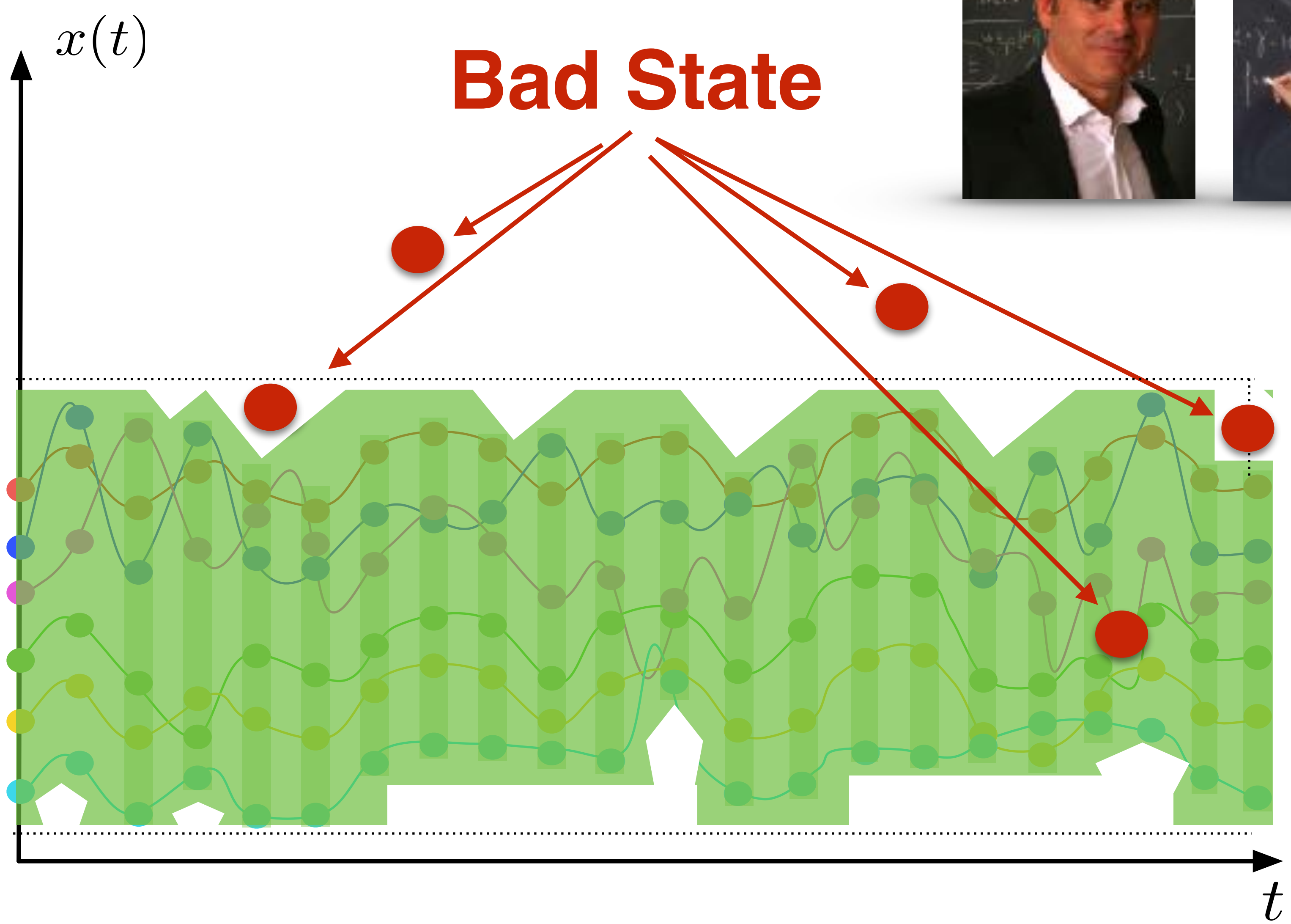
$$[\![P]\!]^{\alpha}$$

False Alarms

Affordable (sound) loss of precision

$$\alpha([\![P]\!]) \subseteq [\![P]\!]^{\alpha}$$

You can always refine!!!



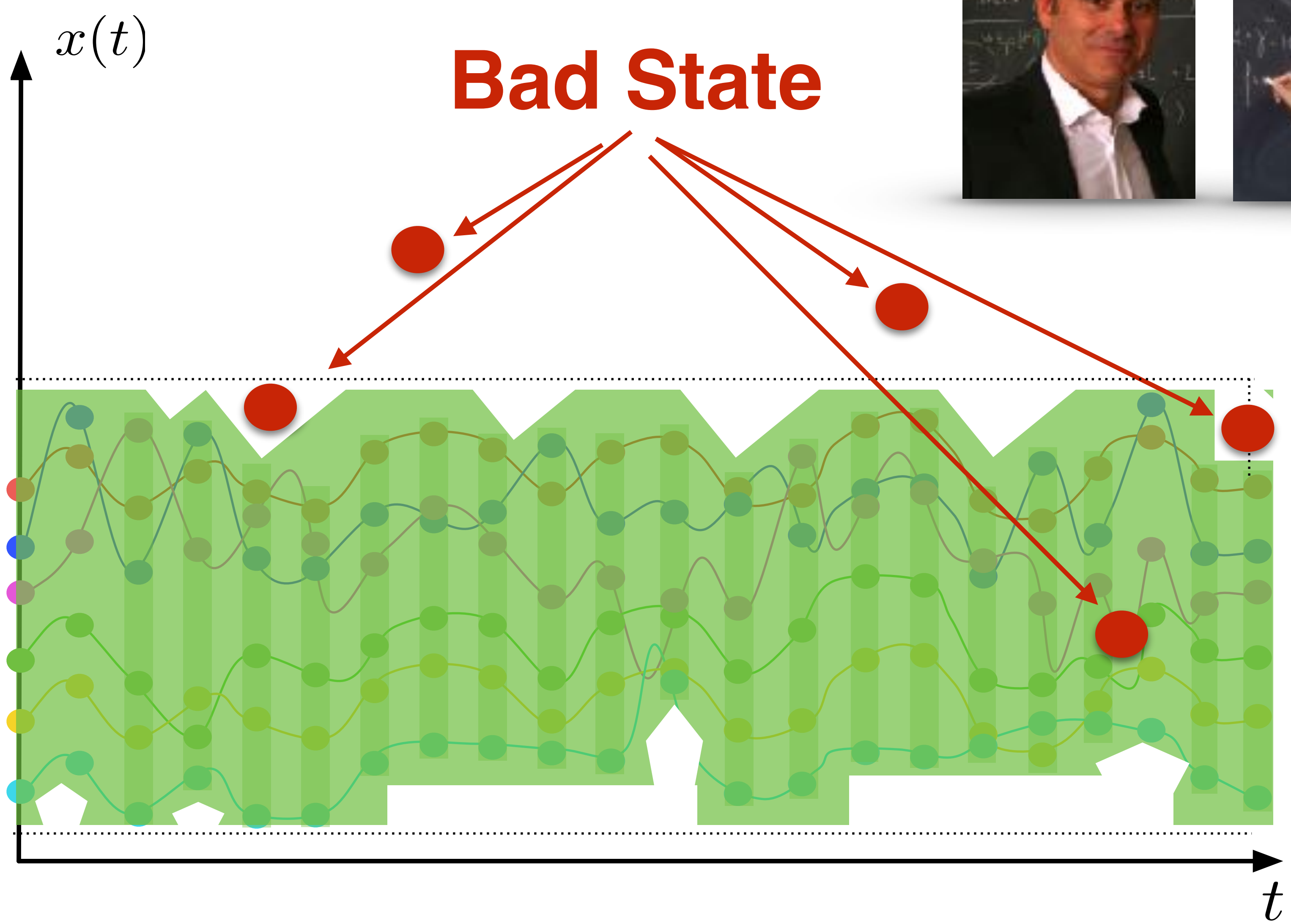
$$[\![P]\!]^{\alpha}$$

Just true bugs!

Completeness Domain Refinement

Giacobazzi et al. JACM 2000

You can always refine!!!



$$[\![P]\!]^{\alpha}$$

Just true bugs!

Completeness Domain Refinement

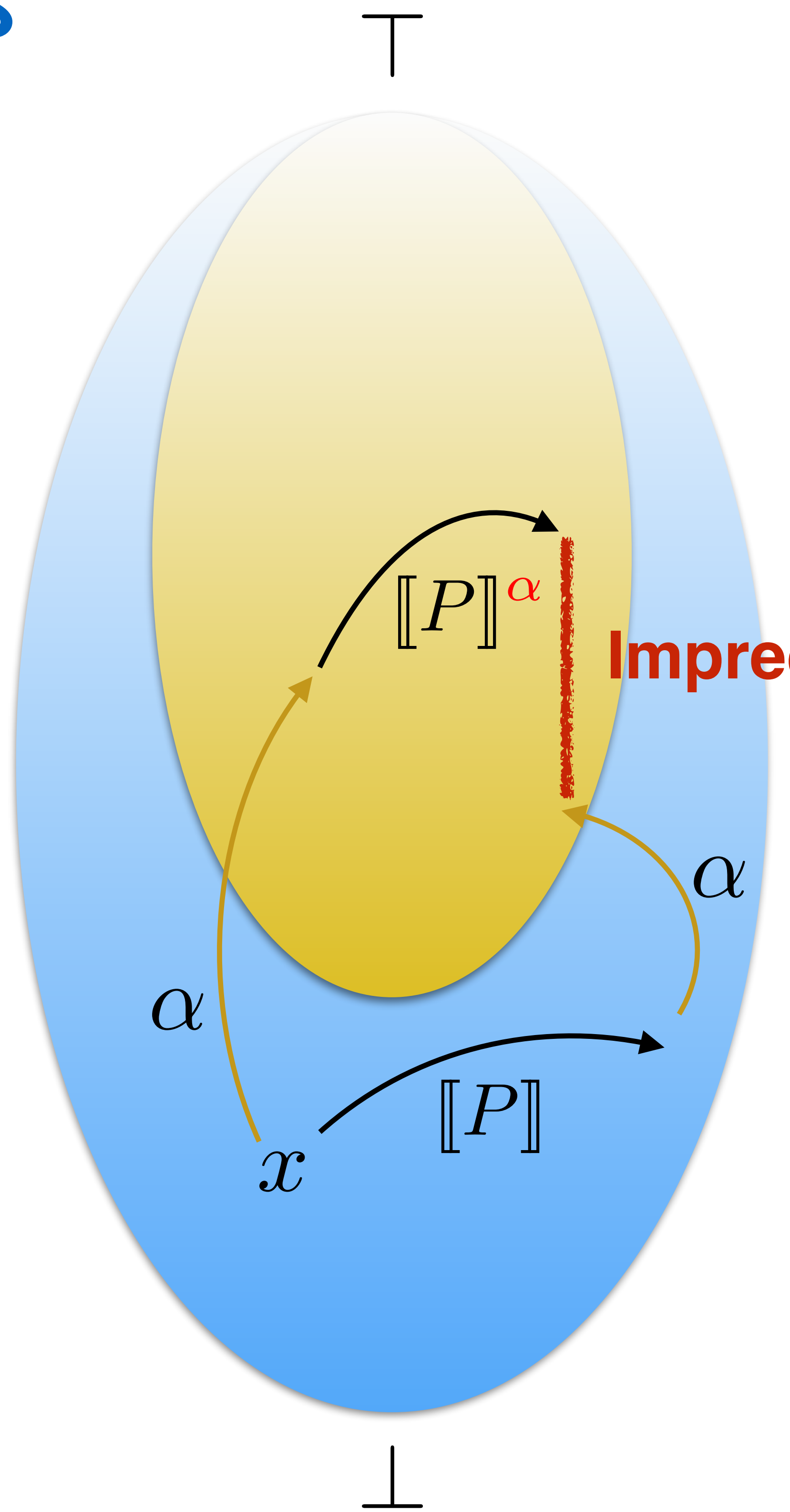
$$\alpha([\![P]\!]) = [\![P]\!]^{\alpha}$$

Soundness

- Analyses are designed to be **sound**

$$\alpha(\llbracket P \rrbracket) \subseteq \llbracket P \rrbracket^{\alpha}$$

- False alarms are due to **imprecision**

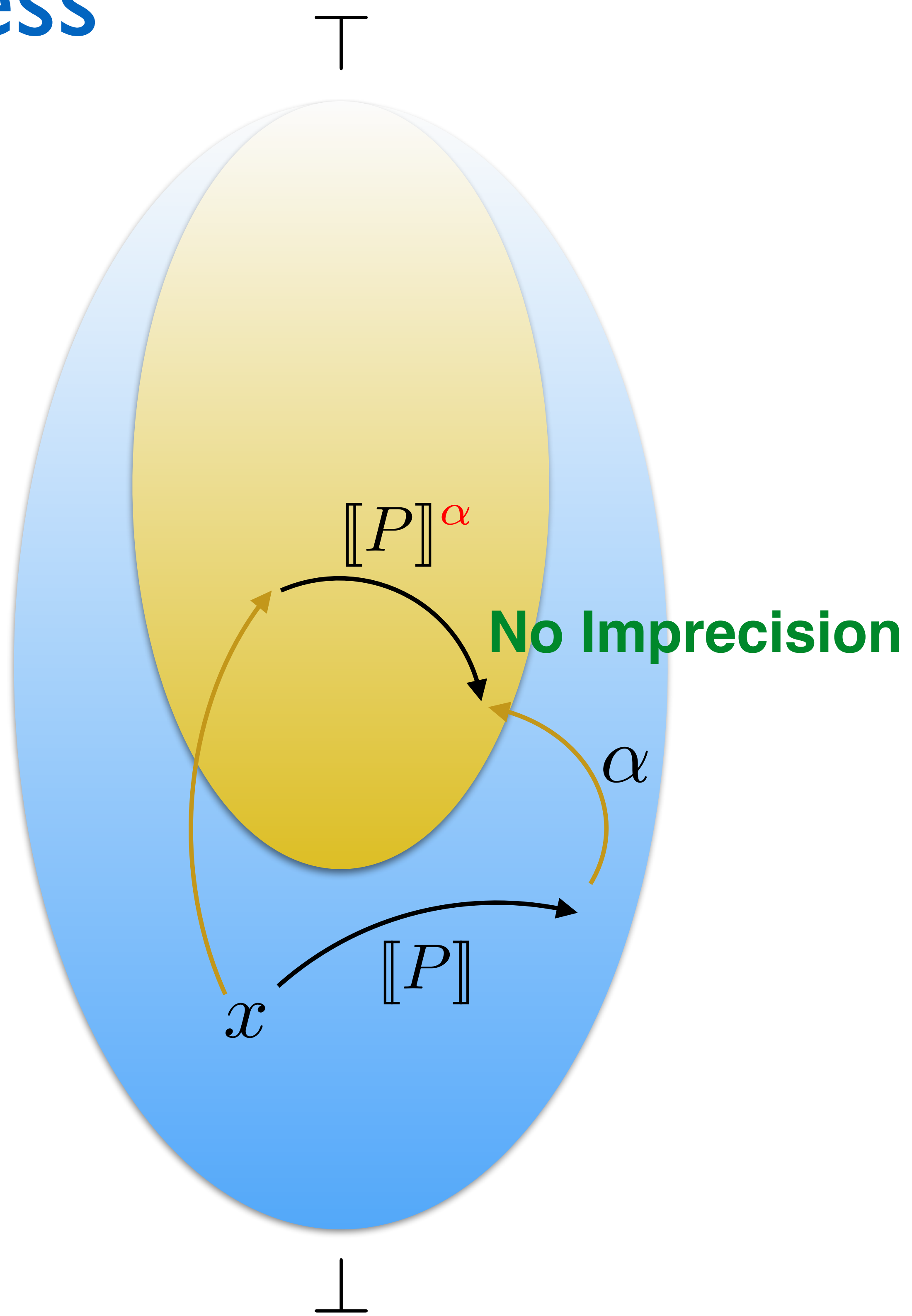


Completeness

- Some analyses may be **complete**

$$\alpha(\llbracket P \rrbracket) = \llbracket P \rrbracket^{\alpha}$$

- Completeness **may happen!**



Example

⊗	+	-	0
+	+	-	0
−	-	+	0
0	0	0	0

$$\begin{tikzcd} 135567 \arrow[d, "\alpha"'] & \times & -145673 \arrow[d, "\alpha"'] & = & -19748451591 \arrow[d, "\alpha"'] \\ + & \circledast & - & = & - \end{tikzcd}$$

⊕	+	-	0
+	+	?	+
−	?	-	-
0	+	-	0

$$\begin{tikzcd} 135567 \arrow[d, "\alpha"'] & + & -145673 \arrow[d, "\alpha"'] & = & -10106 \arrow[d, "\alpha"'] \\ + & \oplus & - & = & ? \end{tikzcd}$$

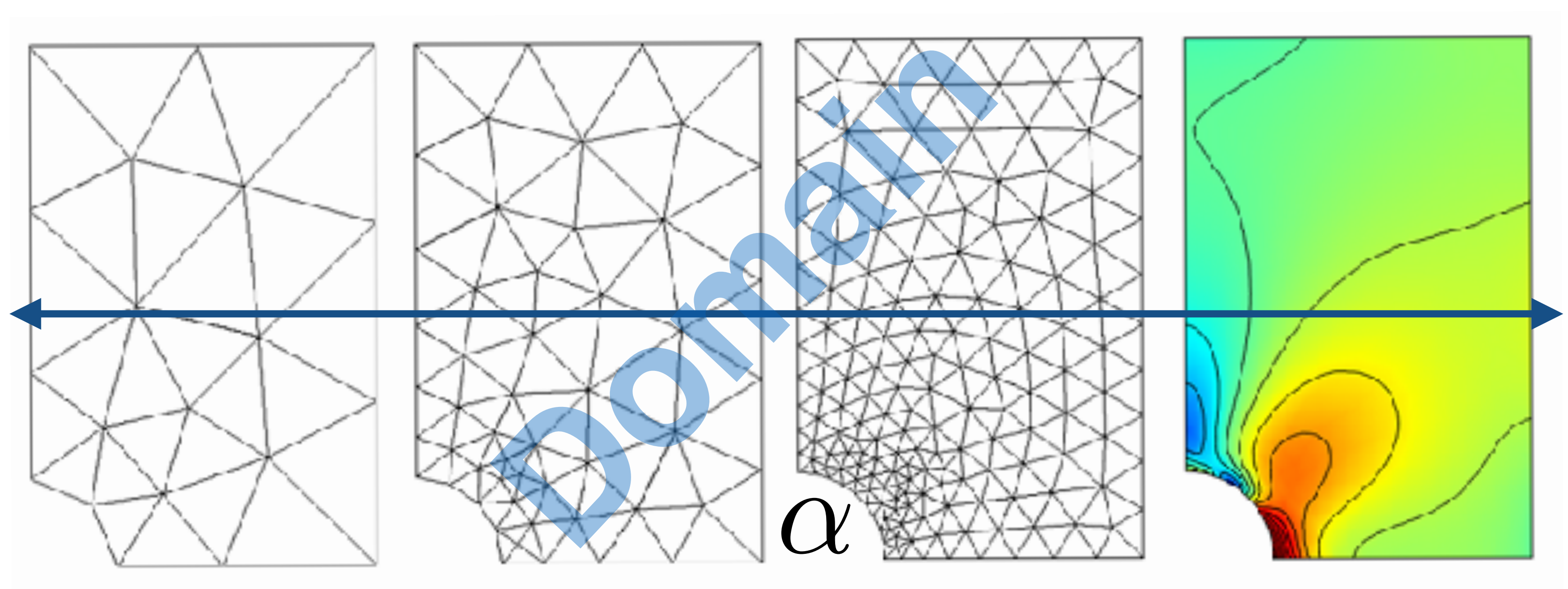
Can we tune precision?

Exploiting the (im)possibility results!

W → $W \overset{?}{\in} \{ P \mid P \approx Q \}$ → ! / ?

Code

Simplify

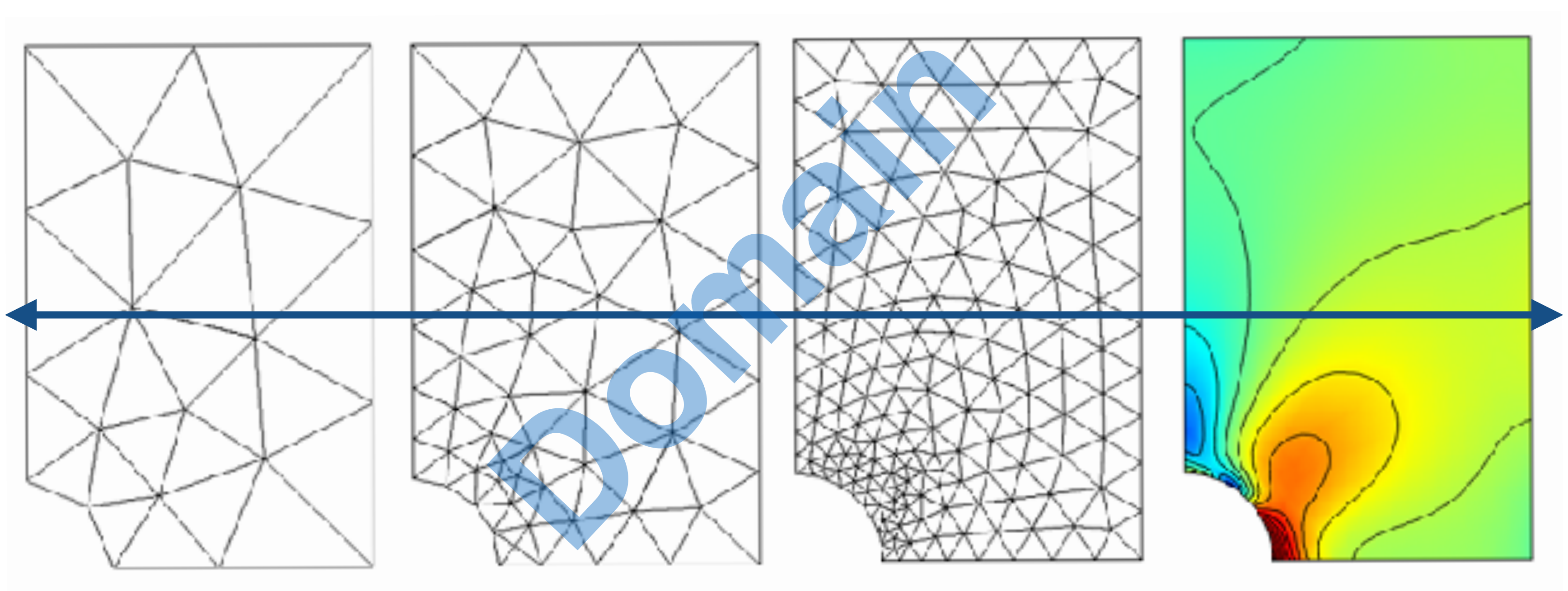


Refine

Exploiting the (im)possibility results!

W Code → $W \overset{?}{\in} \{P \mid P \approx Q\}_{\alpha}$ → **! / ?**

Simplify ⟷ Refine



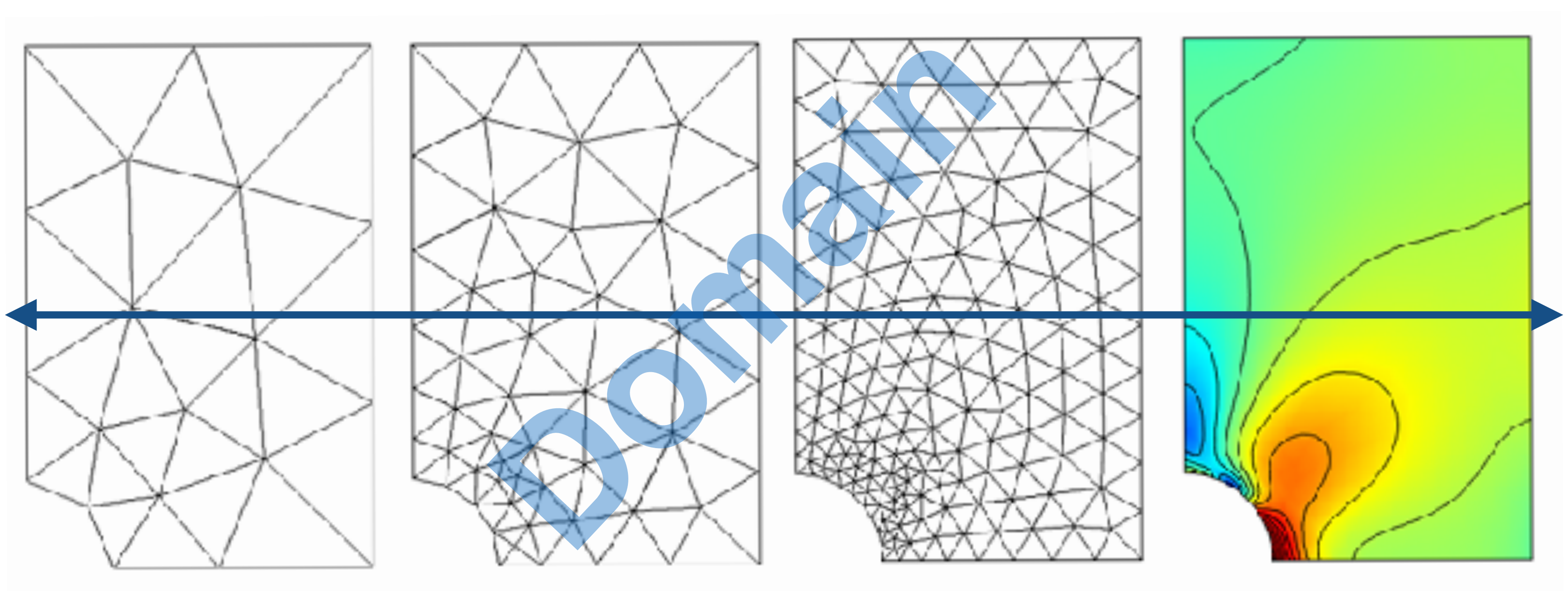
Exploiting the (im)possibility results!

W
Code

$$W \overset{?}{\in} \{ P \mid P \approx_{\alpha} Q \}$$

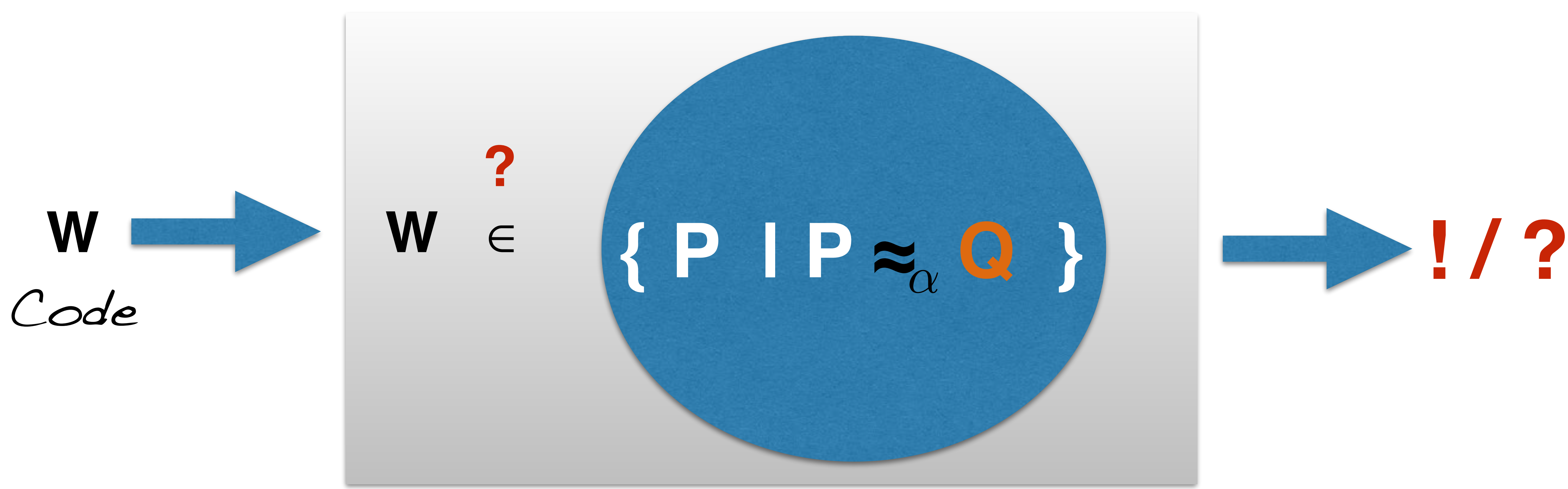
! / ?

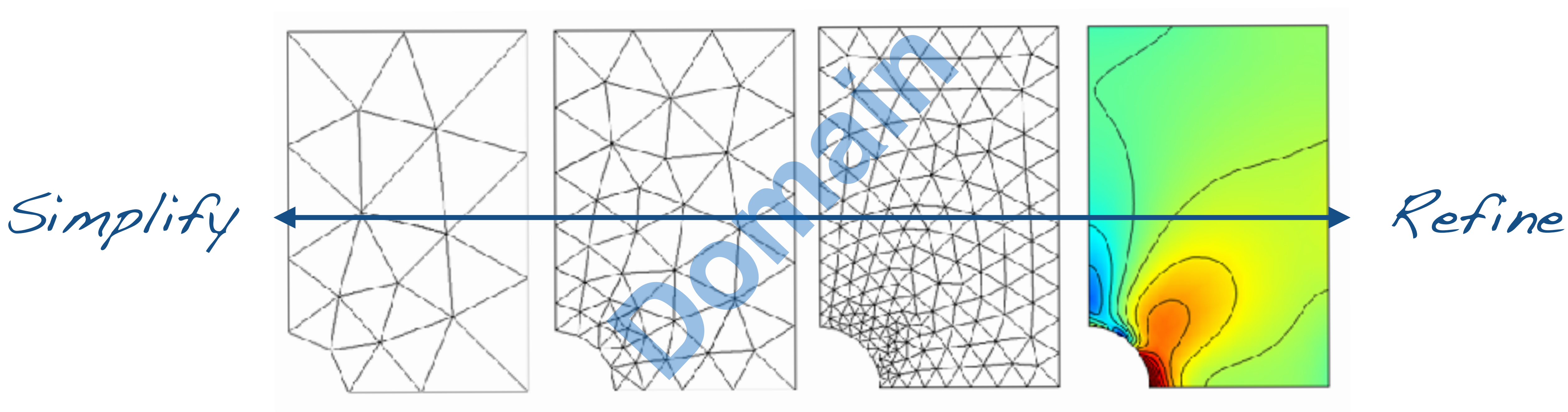
Simplify



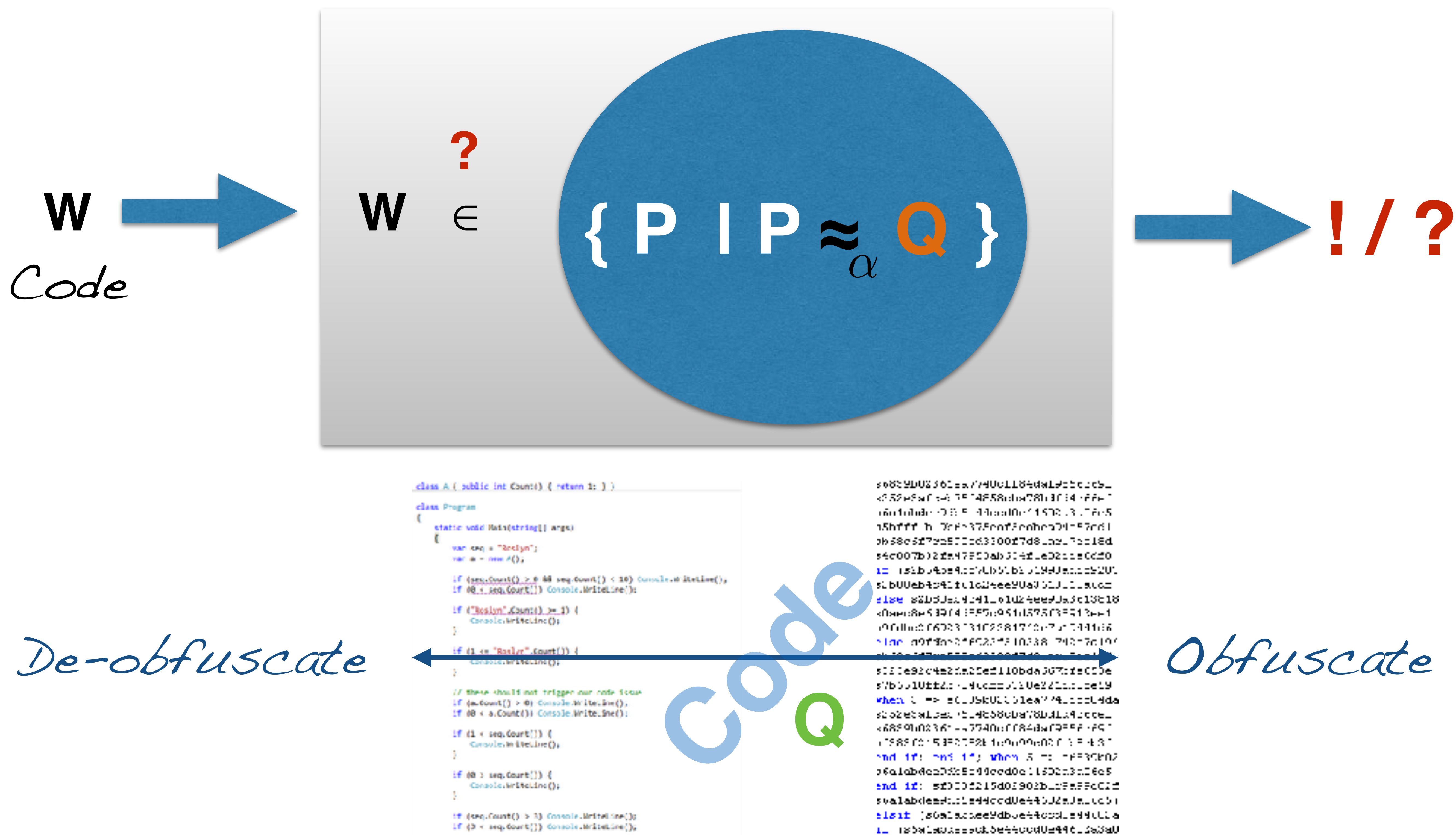
Refine

Exploiting the (im)possibility results!





Exploiting the (im)possibility results!



Exploiting the (im)possibility results!

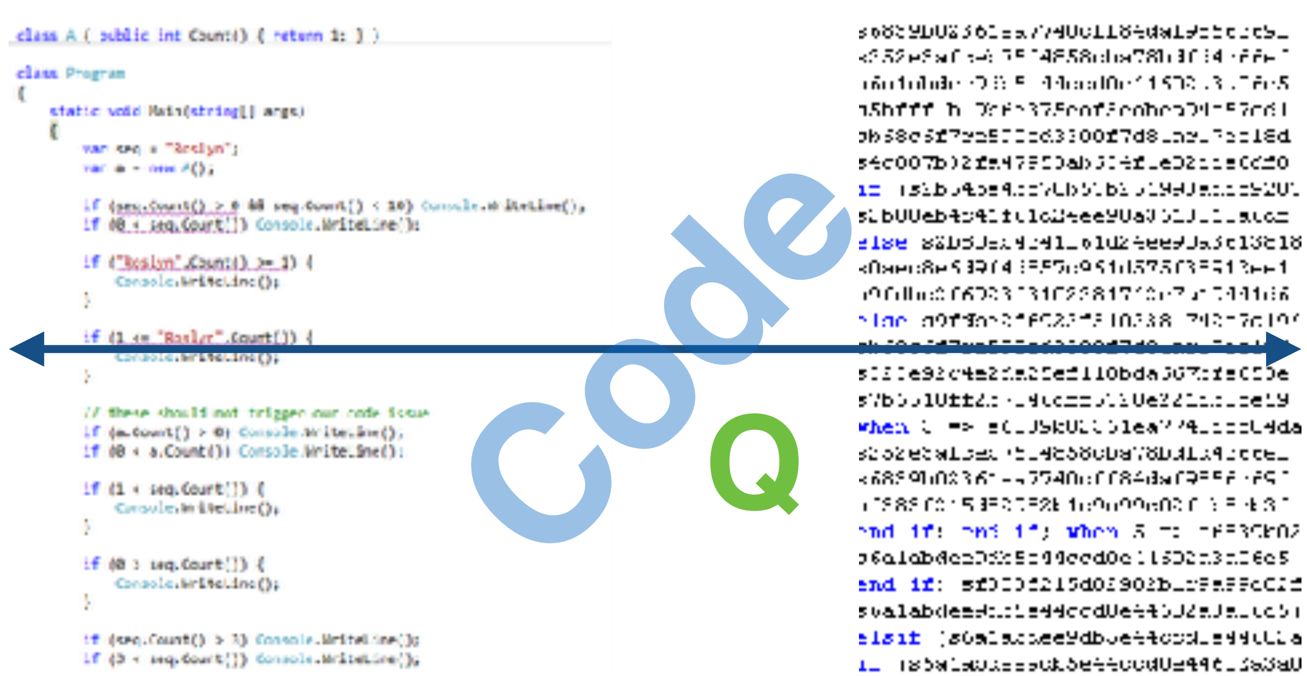
W

Code

$W \overset{?}{\in} \{\, P \mid P \approx_{\alpha} \,\}$

! / ?

De-obfuscate



Obfuscate

Exploiting the (im)possibility results!

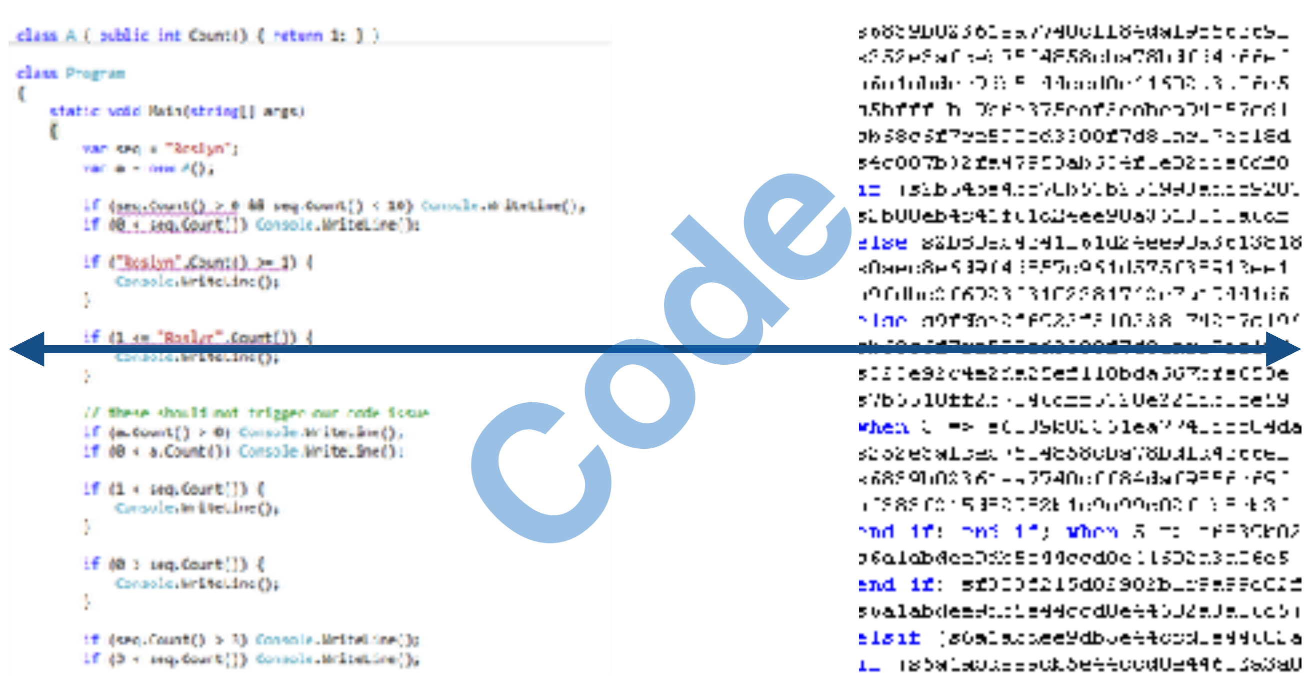
W

Code

$$W \overset{?}{\in} \{ P \mid P \approx_{\alpha} Q \}$$

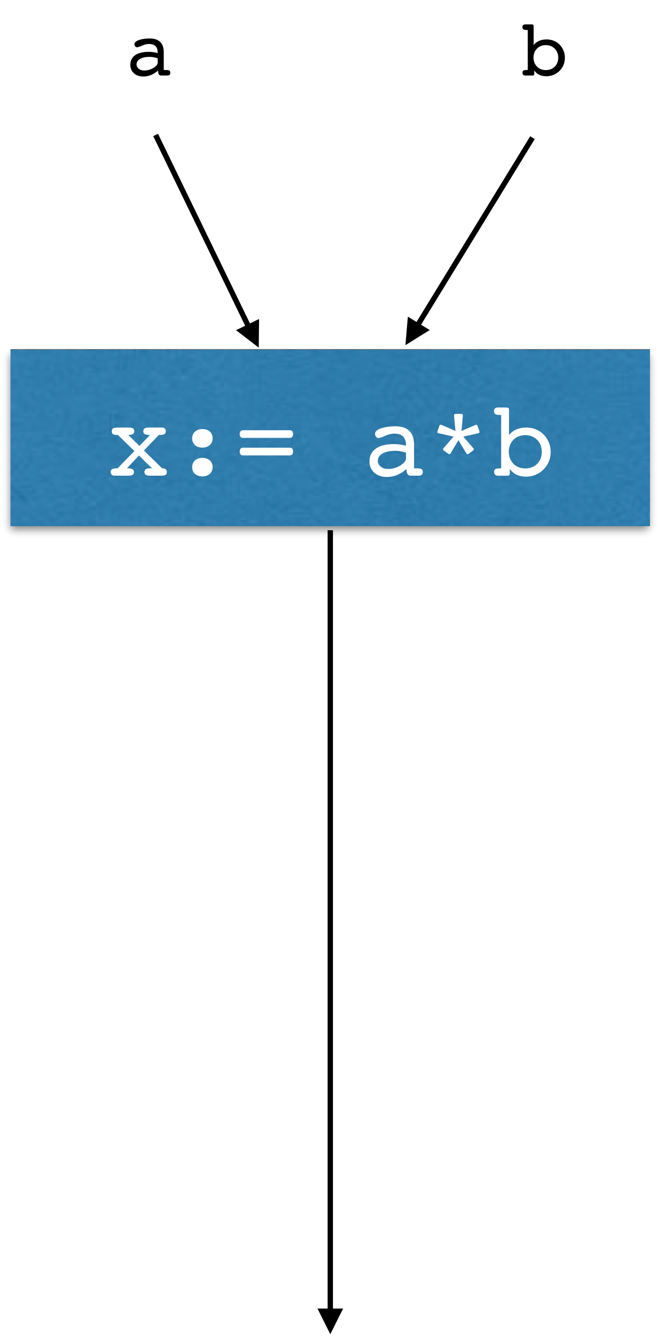
! / ?

De-obfuscate

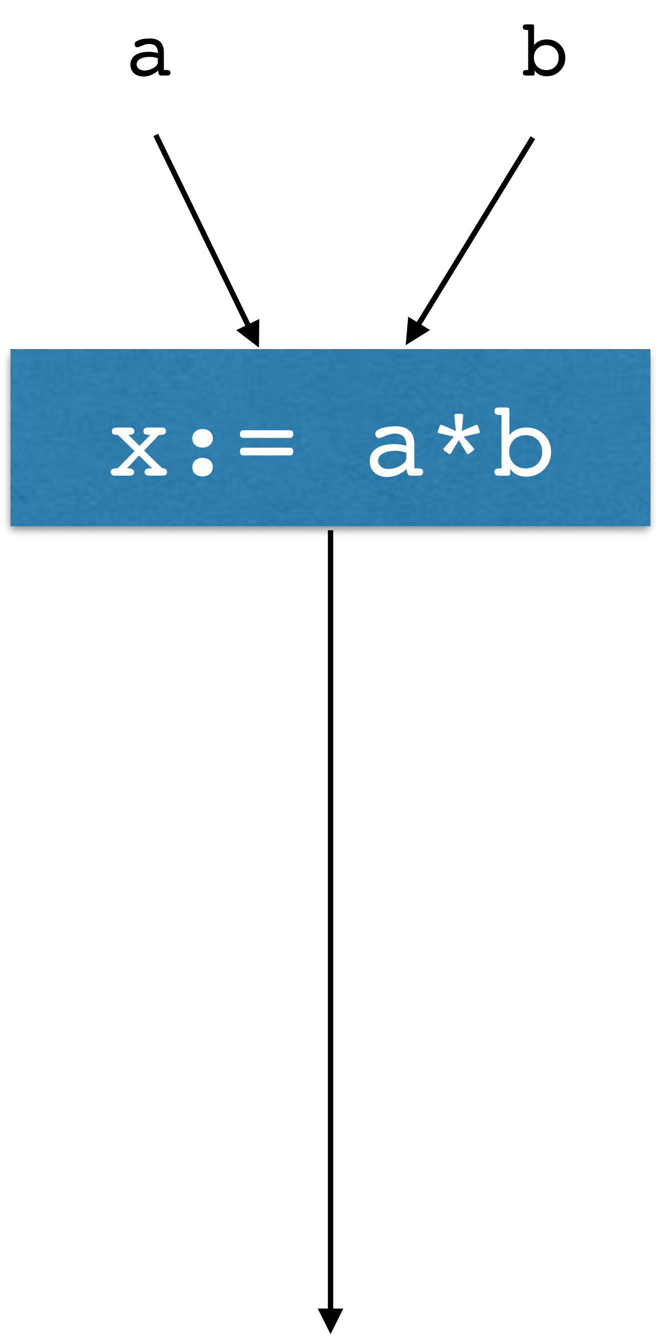


Obfuscate

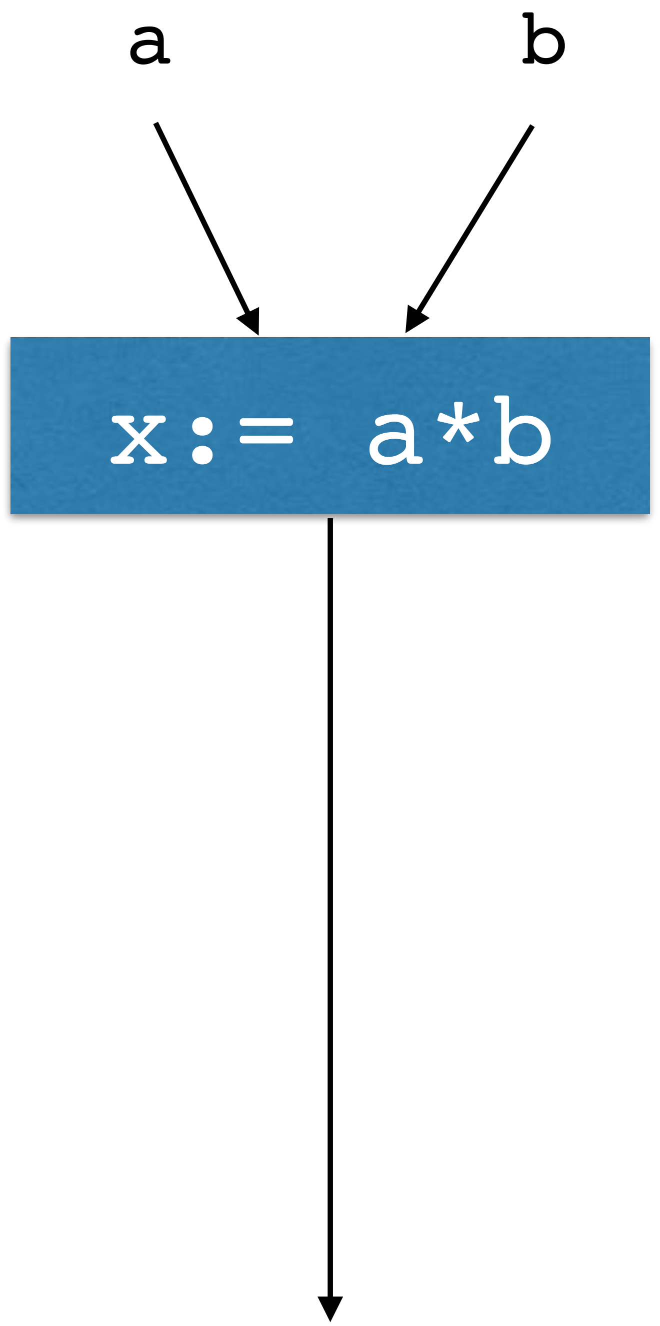
Obscurity as Incompleteness



Obscurity as Incompleteness

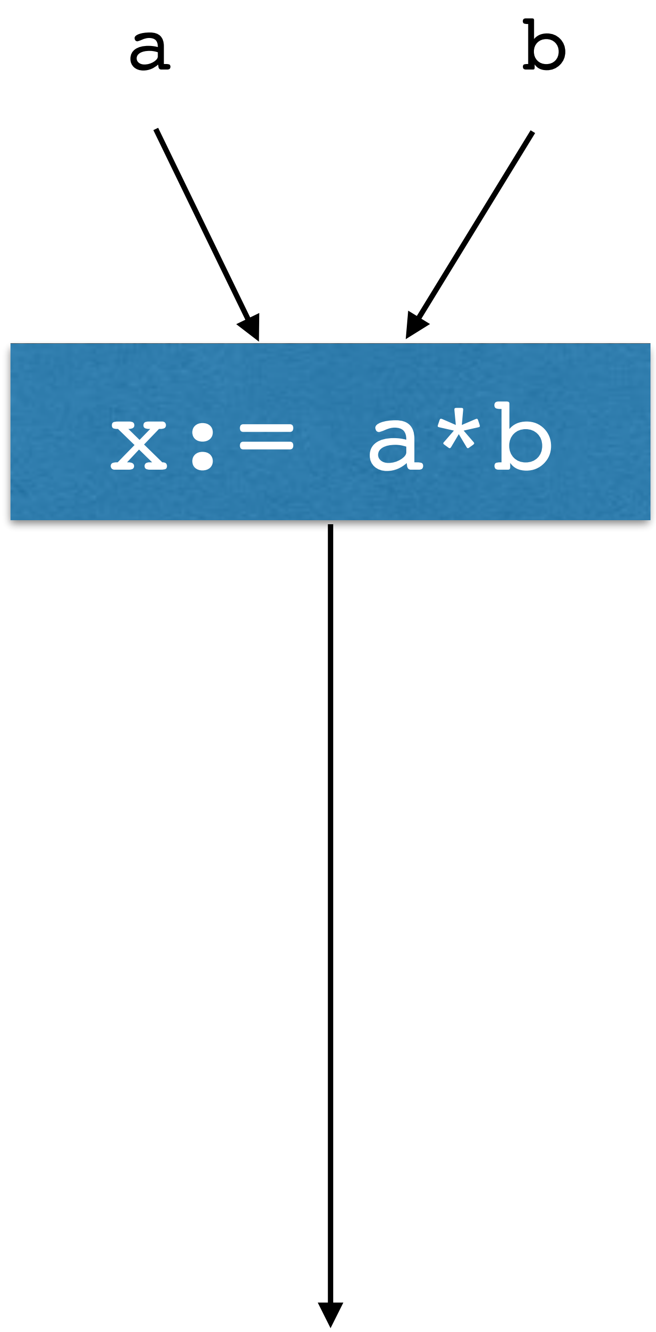


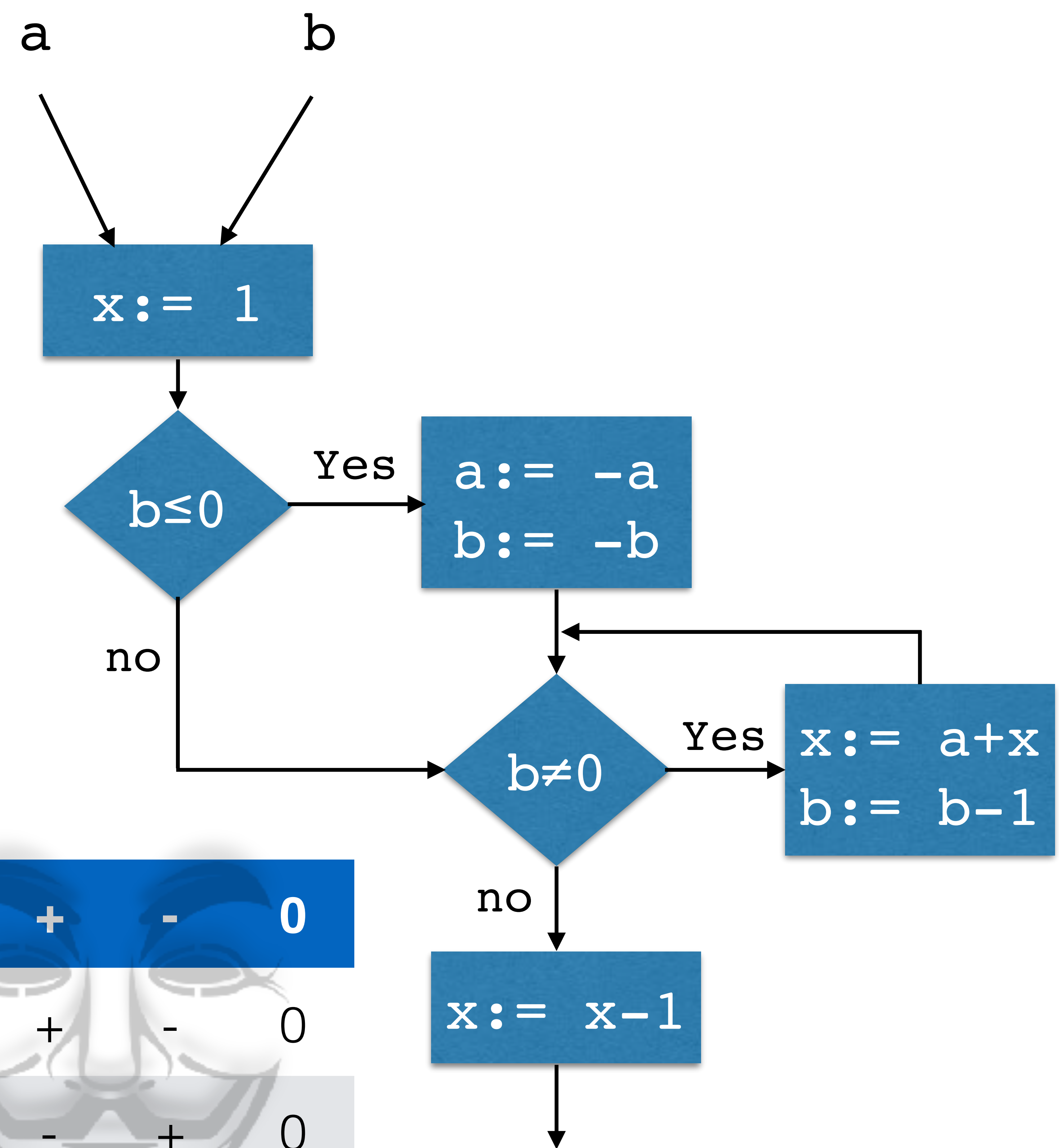
Obscurity as Incompleteness



*	+	-	0
+	+	-	0
−	-	+	0
0	0	0	0

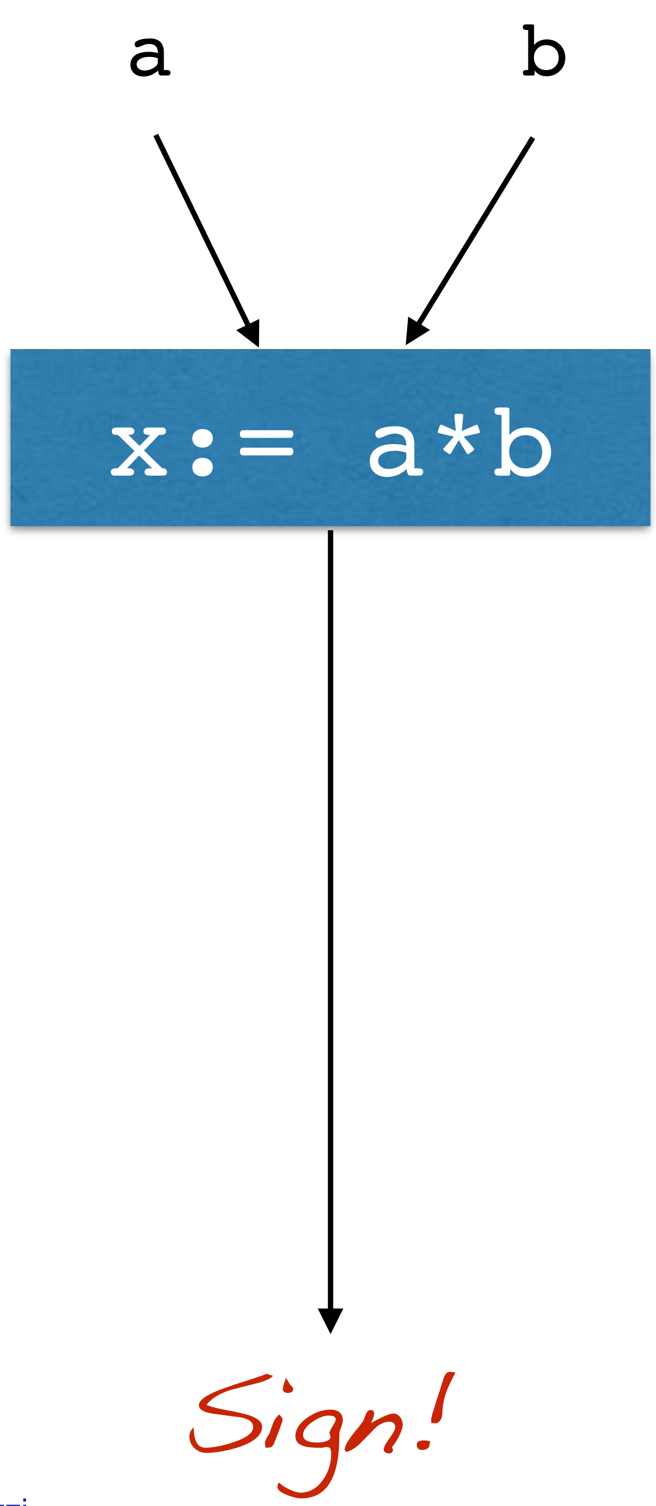
Obscurity as Incompleteness

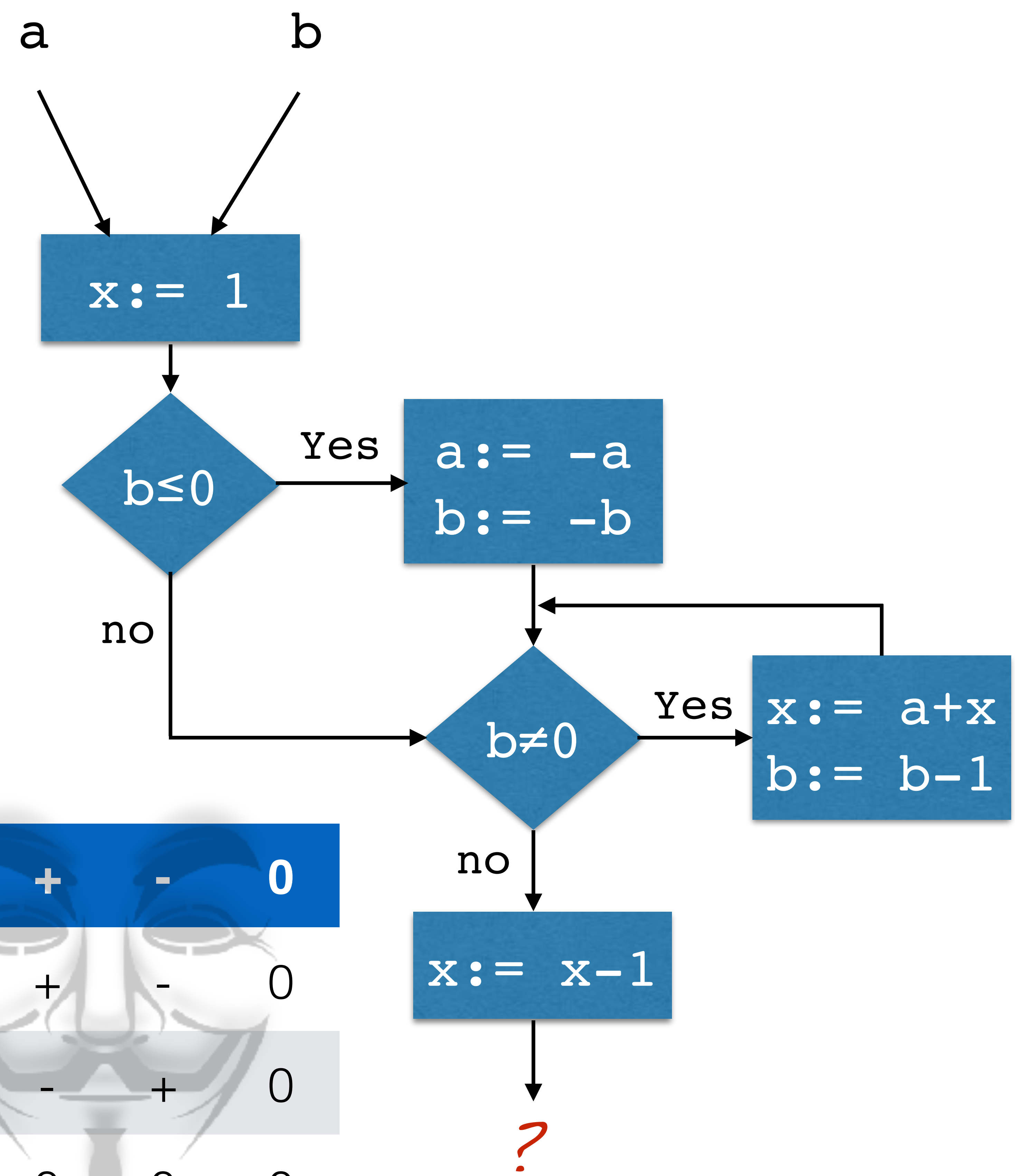




*	+	-	0
+	+	-	0
−	-	+	0
0	0	0	0

Obscurity as Incompleteness





*	+	-	0
+	+	-	0
−	-	+	0
0	0	0	0

On the Completeness Class

Obfuscation/De-obfuscation is compilation between completeness classes

Incomplete

$$\begin{tikzcd} \overline{\mathbb{C}}_\alpha \arrow[r, bend right, "\text{De-obfuscate}"'] & \mathbb{C}_\alpha \arrow[l, bend right, "\text{Obfuscate}"'] \end{tikzcd}$$

Complete

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha([\![P]\!]) = [\![P]\!]^\alpha\}$$

Giacobazzi et al. ACM POPL 2015

On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha(\llbracket P \rrbracket) = \llbracket P \rrbracket^{\alpha}\}$$

Infinite

$\textbf{skip;}$

On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha(\llbracket P \rrbracket) = \llbracket P \rrbracket^{\alpha}\}$$

Infinite

$$\begin{array}{rr} & \textbf{skip;} \\ \textbf{skip;} & \textbf{skip;} \end{array}$$

On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha(\llbracket P \rrbracket) = \llbracket P \rrbracket^{\alpha}\}$$

Infinite

$$\begin{array}{rrr} & & \textbf{skip;} \\ & \textbf{skip;} & \textbf{skip;} \\ \textbf{skip;} & \textbf{skip;} & \textbf{skip;} \end{array}$$

On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha([\![P]\!]) = [\![P]\!]^\alpha\}$$

Infinite

$$\begin{array}{r} \textbf{skip;} \\ \textbf{skip; skip;} \\ \textbf{skip; skip; skip;} \\ \textbf{skip; skip; skip; skip;} \\ \\ \cdots\cdots \end{array}$$

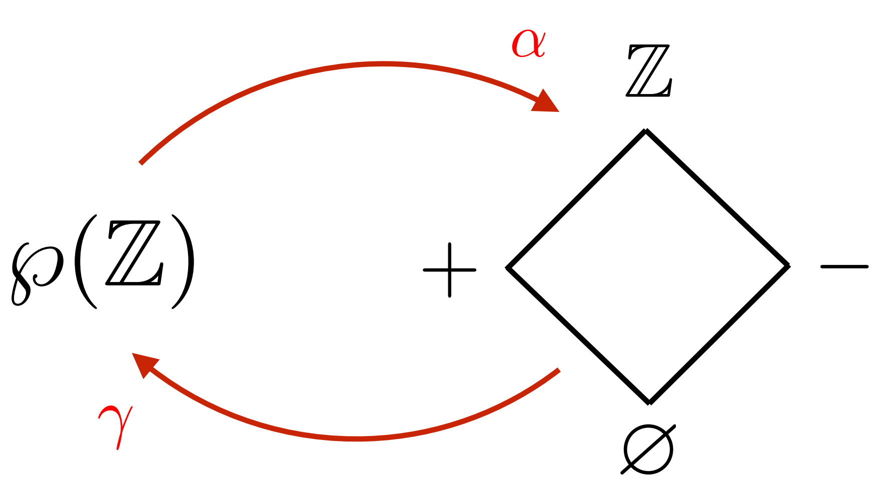
On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha(\llbracket P \rrbracket) = \llbracket P \rrbracket^{\alpha}\}$$

Non Extensional

$$P \text{ complete}, \ \llbracket P \rrbracket = \llbracket Q \rrbracket \ \not\Rightarrow \ Q \text{ complete}$$

$$P : x := y$$
$$Q : x := y + 1; \ x := x - 1$$



$$\llbracket P \rrbracket^{\mathsf{Sign}}\{y/+\} = \{x/+, \ y/+\}$$
$$\llbracket Q \rrbracket^{\mathsf{Sign}}\{y/+\} = \{x/\mathbb{Z}, \ y/+\}$$

As well as complexity!

On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha([\![P]\!]) = [\![P]\!]^{\alpha}\}$$

Non Trivial $\mathbb{C}(\alpha) = \text{All Programs} \;\Leftrightarrow\; \alpha \in \{\lambda x.x,\ \lambda x.\top\}$

$\Rightarrow$ For any nontrivial abstraction α
there always exists an **incomplete** program!

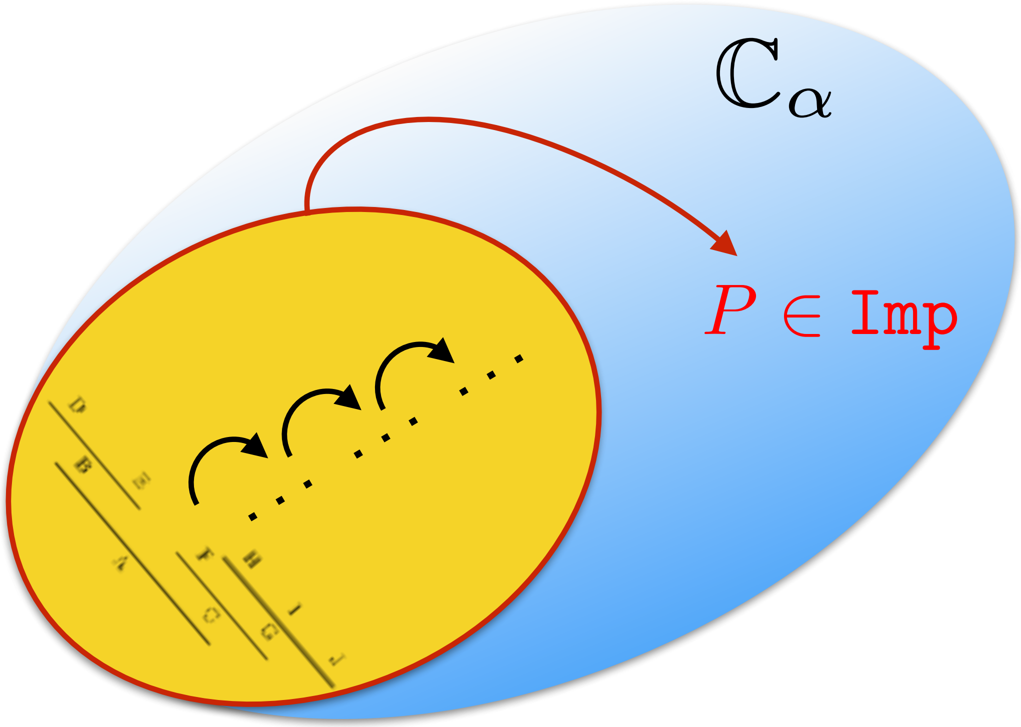
Similar to Rice's Theorem [1952]

On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha(\llbracket P \rrbracket) = \llbracket P \rrbracket^{\alpha}\}$$

Hard

If α non trivial $(\alpha \neq id \; \& \; \alpha \neq \top)$
$\mathbb{C}_\alpha$ and $\overline{\mathbb{C}_\alpha}$ are productive sets



Completeness is harder to prove than termination

On the Completeness Class

$$\mathbb{C}(\alpha) \stackrel{\text{def}}{=} \{P \text{ program} \mid \alpha(\llbracket P \rrbracket) = \llbracket P \rrbracket^{\alpha}\}$$

Hard

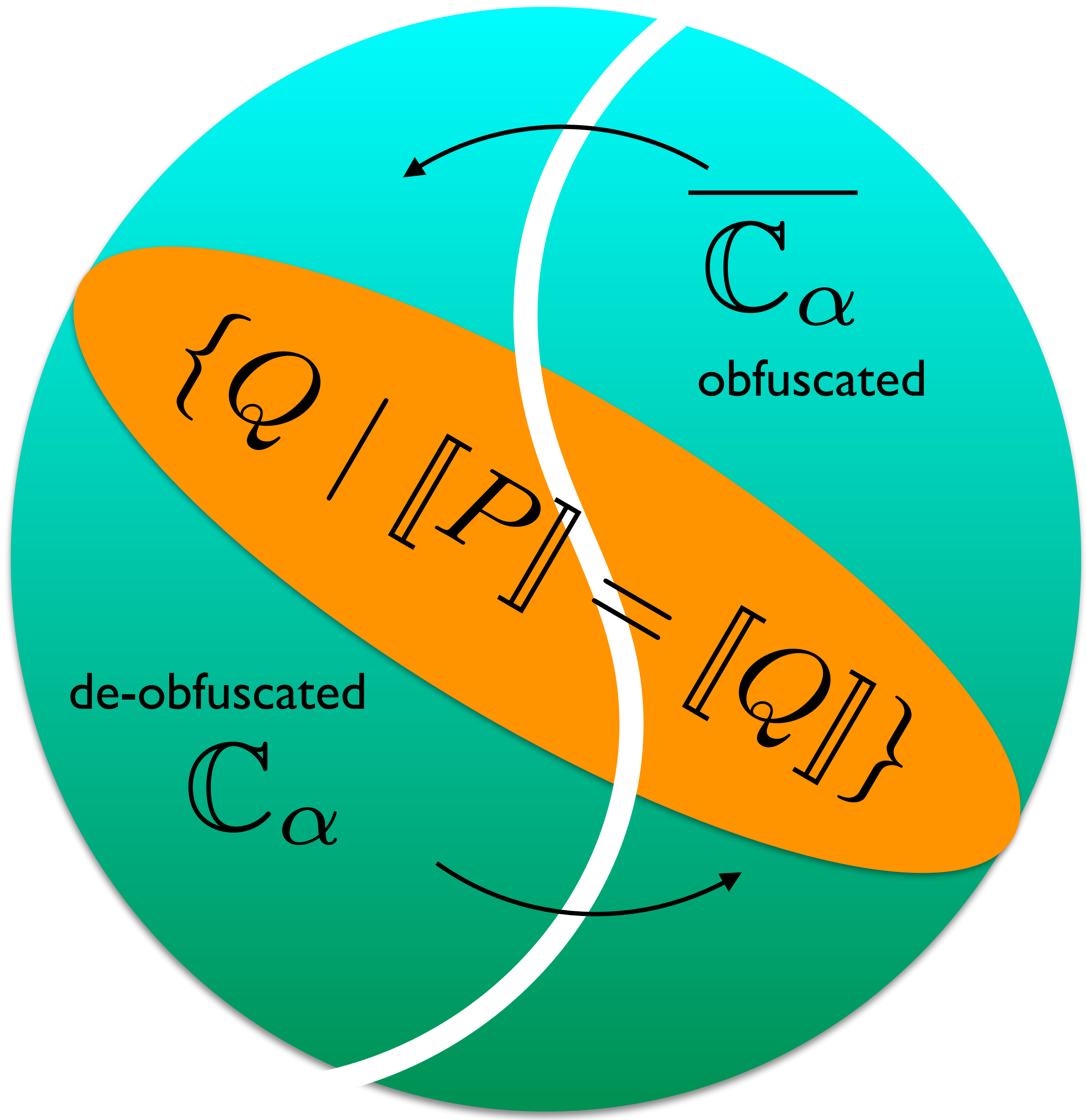
If α *non trivial* $(\alpha \neq id \;\&\; \alpha \neq \top)$
$\mathbb{C}_\alpha$ *and* $\overline{\mathbb{C}_\alpha}$ *are productive sets*

⇨ Automating the proof that α is complete for P is **impossible**

Completeness is harder to prove than termination

On Completeness

and impossibility



$\mathbb{C}_\alpha \not\leq_m \overline{\mathbb{C}_\alpha}$ and $\overline{\mathbb{C}_\alpha} \not\leq_m \mathbb{C}_\alpha$

Let us mix all this together

How to make code obscure

via Yoshihiko Futamura 1971

Programming style

$$\begin{array}{rcl} [\![P]\!](d) & = & [\![\mathtt{interp}]\!](P, d) \\ & = & [\![\ [\![\mathtt{spec}]\!](\mathtt{interp}, P)\ ]\!](d) \end{array}$$

Algorithm

How to make code obscure

via Yoshihiko Futamura 1971

Programming style

$$
\begin{aligned}
[\![P]\!](d) &= [\![\texttt{interp}]\!](P, d) \\
&= [\![\,[\![\texttt{spec}]\!](\texttt{interp}, P)\,]\!](d)
\end{aligned}
$$

Idea!!

$=$

Algorithm

$$P \longrightarrow \texttt{Obf}_{\alpha}(P)$$

I: Data Obfuscation

$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

Weird Interpretation: v → 2v

input $\mathtt{P}, d$; Program to be interpreted, and *its* data

$pc := 2$; Initialise program counter and obfuscated store:

$store := [in \mapsto obf(d), out \mapsto obf(0), x_1 \mapsto obf(0), \ldots]$;

while $pc < length(\mathtt{P})$ **do**

$instruction := lookup(\mathtt{P}, pc)$;

case $instruction$ **of** Dispatch on syntax

skip : $pc := pc + 1$; Obfuscate values when stored:

$x := e$: $store := store[x \mapsto obf(eval(e, store))]$; $pc := pc + 1$;

$\ldots$ **endw** ;

output $dob(store[out])$;

$obf(V) = 2 * V$; $dob(V) = V/2$ Obfuscation/de-obfuscation

$eval(e, store) =$ **case** e **of**

$constant : obf(e)$

$variable : dob(store(e))$ De-obfuscate variable values

$e1 + e2 : eval(e1, store) + eval(e2, store)$

$e1 - e2 : eval(e1, store) - eval(e2, store)$

$\ldots$

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Weird Interpretation: v → 2v

The source program is automatically transformed into this equivalent obfuscated one

1. **input** x;		1. **input** x;
		1.5. $x := 2 * x$; Obfuscate input x
2. $y := 2$;		2. $y := 2 * 2$; Obfuscate $y := 2$
3. **while** $x > 0$ **do**		3. **while** $x/2 > 0$ **do** De-obfuscate x
4. $y := y + 2$;	$\longmapsto$	4. $y := 2 * (y/2 + 2)$;
5. $x := x - 1$		5. $x := 2 * (x/2 - 1)$
endw		**endw**
6. **output** y;		6. **output** $y/2$; De-obfuscate output
7. **end**		7. **end**

$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

Sign Attack

- Sign analysis is complete for multiplication $*$: exact information.
- Sign analysis is incomplete for addition $+$: imprecise information

$*$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

$+$	$-$	0	$+$
$-$	$-$	$-$	$\top(!)$
0	$-$	0	$+$
$+$	$\top(!)$	$+$	$+$

Our trick: ...let the interpreter evaluate!

$$\begin{array}{l} eval(e, store) \;=\; \textbf{case } e \textbf{ of} \\ \quad e1 + e2 \;:\; eval(e1, store) + eval(e2, store) \\ \quad e1 * e2 \;:\; \textbf{let } v1 = eval(e1, store), v2 = eval(e2, store) \\ \qquad\qquad\qquad\qquad \textbf{in } \; v1 * (v2 - 1) + v1 \end{array}$$

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Sign Attack

- Sign analysis is complete for multiplication $*$: exact information.
- Sign analysis is incomplete for addition $+$: imprecise information

P:

$^{1.}$**input** x;
$^{2.}$$y := 2$;
$^{3.}$**while** $x > 0$ **do**
  $^{4.}$$y := y * y$;
  $^{5.}$$x := x - 1$
 endw
$^{6.}$**output** y;
$^{7.}$**end**

$\longmapsto$

P′:

$^{1.}$**input** x;
$^{2.}$$y := 2$;
$^{3.}$**while** $x > 0$ **do**
  $^{4.}$$y := y * (y - 1) + y$;
  $^{5.}$$x := x - 1$
 endw
$^{6.}$**output** y;
$^{7.}$**end**

Sign analysis yields $y \mapsto +$ in P, but it yields $y \mapsto \top$ in P′.

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Interval Attack

We consider variable splitting:

$v \in Var(P)$ is split into $\langle v_1, v_2 \rangle$ such that
$v_1 = f_1(v)$, $v_2 = f_2(v)$ and $v = g(v_1, v_2)$

$$f_1(v) = v \div 10$$
$$f_2(v) = v \mod 10$$
$$g(v_1, v_2) = 10 \cdot v_1 + v_2$$

And the interval analysis: $\iota(x) = [\min(x), \max(x)]$

$$P : \left[\begin{array}{l} v = 0; \\ \textbf{while } v < N \; \{v++\} \end{array} \right. \qquad [\![P]\!]^{\iota} = \lambda v.\, [0, N]$$

Interval Attack

We consider variable splitting:

$v \in Var(P)$ is split into $\langle v_1, v_2 \rangle$ such that
$v_1 = f_1(v)$, $v_2 = f_2(v)$ and $v = g(v_1, v_2)$

$$f_1(v) = v \div 10$$
$$f_2(v) = v \mod 10$$
$$g(v_1, v_2) = 10 \cdot v_1 + v_2$$

And the interval analysis: $\iota(x) = [\min(x), \max(x)]$

$$\tau(P) : \left[\begin{array}{l} v_1 = 0; \\ v_2 = 0; \\ \textbf{while} \quad 10 \cdot v_1 + v_2 < N \; \{ \\ \qquad v_1 = v_1 + (v_2 + 1) \div 10 \\ \qquad v_2 = (v_2 + 1) \mod 10 \\ \quad \}; \end{array} \right.$$

$$c : \quad v = 10 \cdot v_1 + v_2$$

$$\begin{array}{ll} [\![\tau(P); c]\!]^{\iota} & = \\ \lambda v.\, 10 \odot [0, \frac{N \ominus [0,9]}{10}] \oplus [0, 9] & = \\ \lambda v.\, [0, N] \oplus [0, 9] & = \\ \lambda v.\, [0, N+9] & \end{array}$$

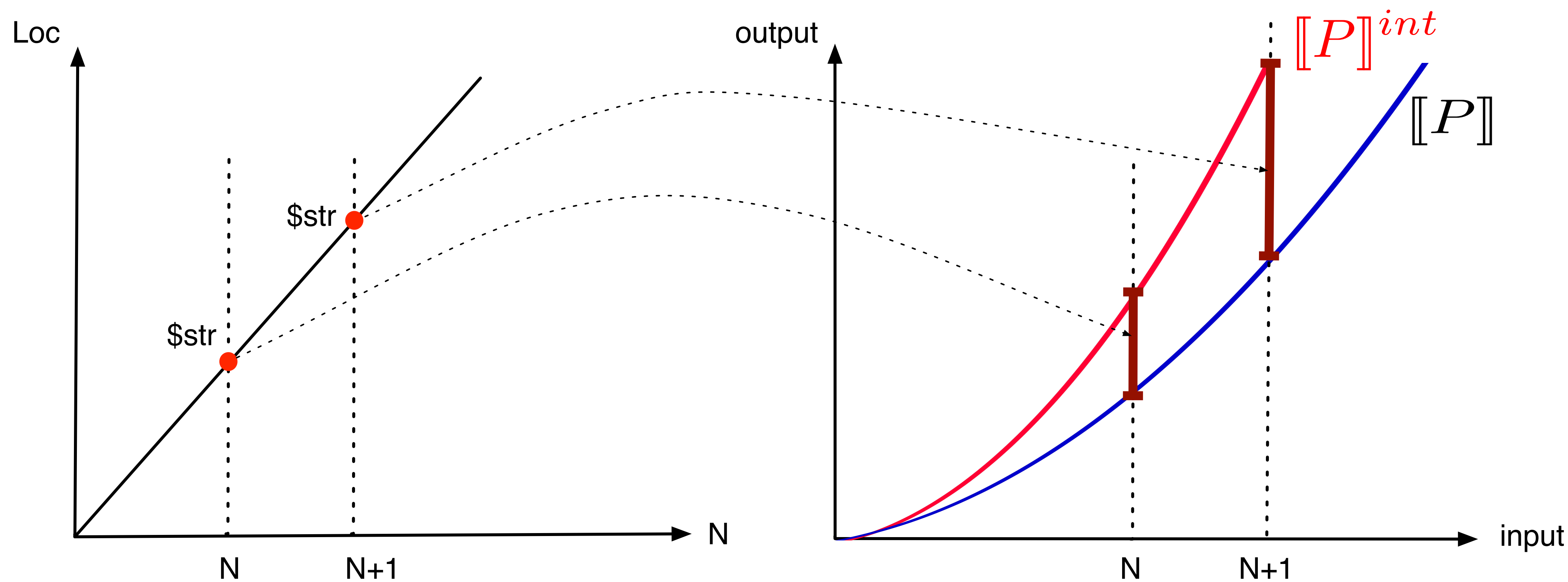
Obfuscation induces errors

Interval Attack

Dynamic Obfuscation

```
<?php
$w = 0; $n=1;
while ($n<=N) {
$z = rand(2, 10);
$str = ’$x=0;$y=0;
          while (’.$z.’*$x+$y+1<=’.$n.’) {
                 $x=intval((’.$z.’*$x+$y+1)/’.$z.’);
                 $y=($y+1)%’.$z.’;};
          $w=$w+’.$z.’*$x+$y;’. $str; ++$n;};
eval($str.’;’); echo $w. "\n";
?>
```

$$(N^2 + N)/2$$

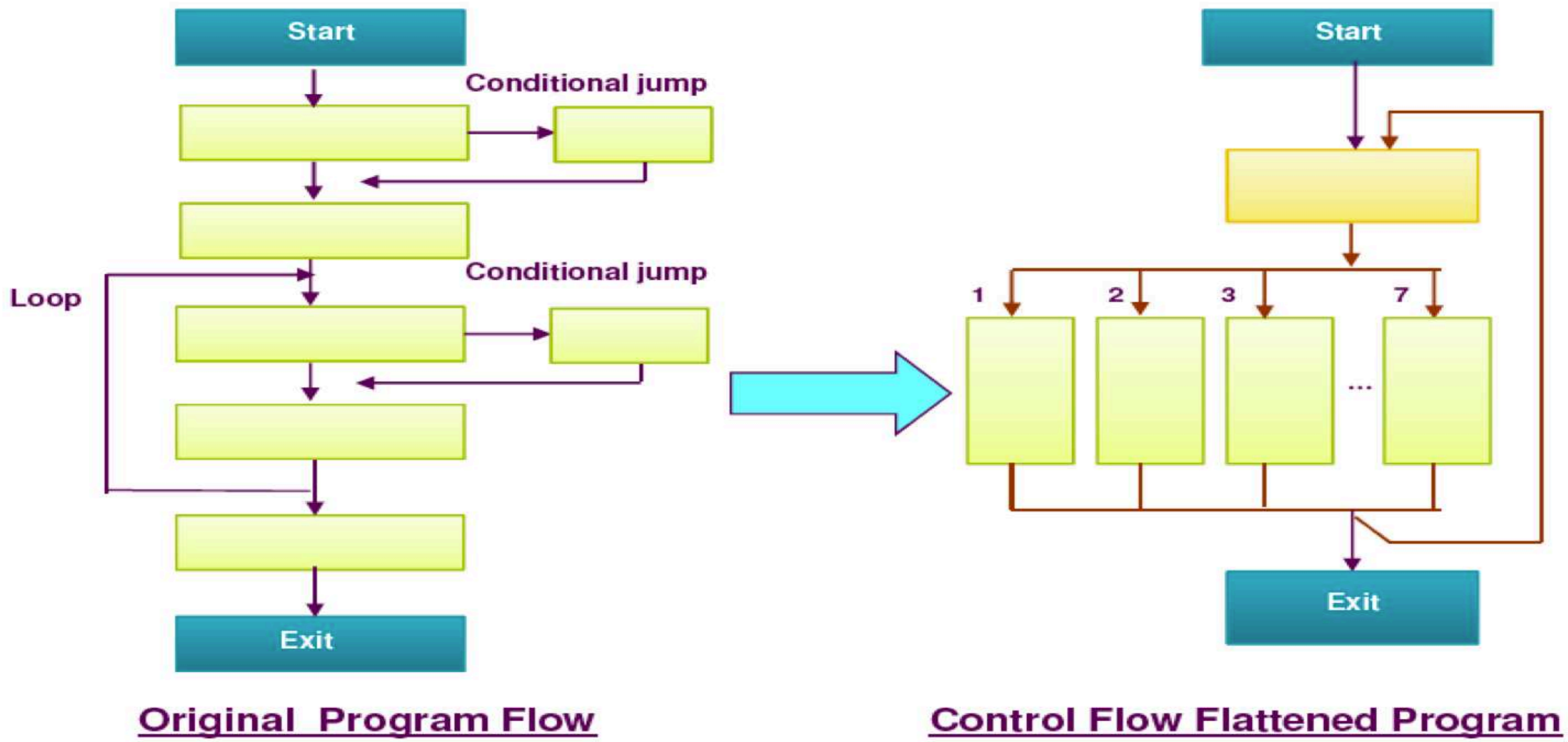


II: Flattening

$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

Code Flattening

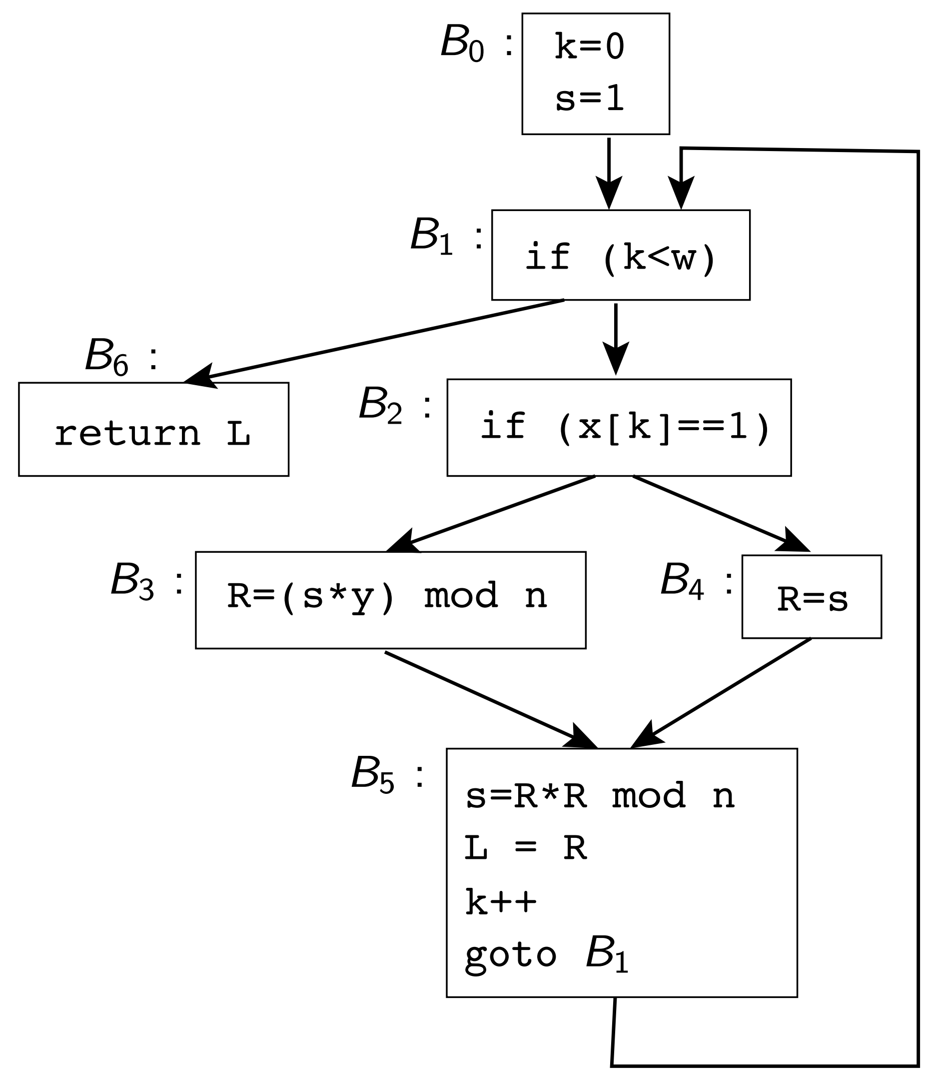
Idea: "scramble" or "distort" the control flow of input program `P`, without changing its whole-program semantics



$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Code Flattening

```
int modexp(int y,int x[],
           int w,int n) {
    int R, L;
    int k = 0;
    int s = 1;
    while (k < w) {
        if (x[k] == 1)
            R = (s*y) % n;
        else
            R = s;
        s = R*R % n;
        L = R;
        k++;
    }
    return L;
}
```



$$\mathtt{Obf}_{\alpha}(P) = [\![\mathsf{spec}]\!](\mathtt{interp}, P)$$

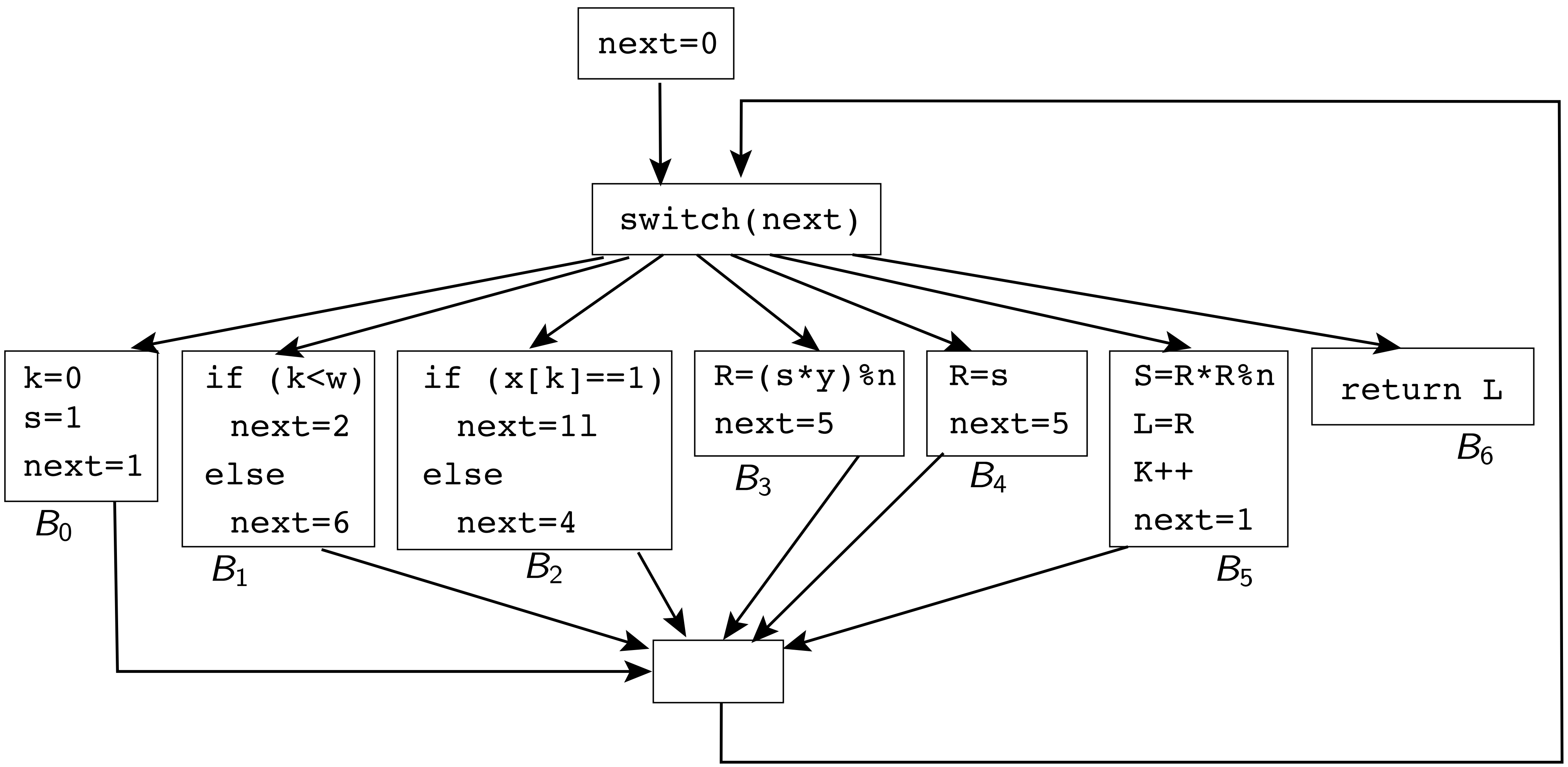
Code Flattening

irdeto

```
int modexp(int y, int x[], int w, int n) {
    int R, L, k, s;
    int next=0;
    for(;;)
        switch(next) {
            case 0 : k=0; s=1; next=1; break;
            case 1 : if (k<w) next=2; else next=6; break;
            case 2 : if (x[k]==1) next=3; else next=4; break;
            case 3 : R=(s*y)%n; next=5; break;
            case 4 : R=s; next=5; break;
            case 5 : s=R*R%n; L=R; k++; next=1; break;
            case 6 : return L;
        }
}
```

$$\texttt{Obf}_{\alpha}(P) = [\![\text{spec}]\!](\texttt{interp}, P)$$

Code Flattening



$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

Code Flattening

Original program $\mathtt{P}$:

$^{1\cdot}$**input** x;
$^{2\cdot}y := 2$;
$^{3\cdot}$**while** $x > 0$ **do**
  $^{4\cdot}y := y + 2$;
  $^{5\cdot}x := x - 1$
 endw
$^{6\cdot}$**output** y;
$^{7\cdot}$**end**

Flattened equivalent program $\mathtt{P}'$:

$^{1\cdot}$**input** x; $^{2\cdot}pc := 2$;
$^{3\cdot}$**while** $pc < 6$ **do**
  $^{4\cdot}$**case** pc **of**
    $2 :\ ^{5\cdot}y := 2;\ ^{6\cdot}pc := 3$;
    $3 :\ ^{7\cdot}$**if** $x > 0$ **then** $^{8\cdot}pc := 4$ **else** $^{9\cdot}pc := 6$;
    $4 :\ ^{10\cdot}y := y + 2;\ ^{11\cdot}pc := 5$;
    $5 :\ ^{12\cdot}x := x - 1;\ ^{13\cdot}pc := 3$;
 endw
$^{14\cdot}$**output** y
$^{15\cdot}$**end**

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

A self-Interpreter

input $\mathtt{P}, d$; Program to be interpreted, and *its* data

$pc := 2$; Initialise program counter and store

$store := [in \mapsto d, out \mapsto 0, x_1 \mapsto 0, \ldots]$;

while $pc < length(\mathtt{P})$ **do**

$instruction := lookup(\mathtt{P}, pc)$; Find the pc-th instruction

case $instruction$ **of** Dispatch on syntax

skip : $pc := pc + 1$;

$x := e$: $store := store[x \mapsto eval(e, store)]$; $pc := pc + 1$;

$\ldots$ **endw** ;

output $store[out]$;

$eval(e, store) \;=\;$ **case** e **of** Function to evaluate expressions

$constant$: e

$variable$: $store(e)$

$e1 + e2$: $eval(e1, store) + eval(e2, store)$

$e1 - e2$: $eval(e1, store) - eval(e2, store)$

$e1 * e2$: $eval(e1, store) * eval(e2, store)$ $\ldots$

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Code Flattening

Original program P:

1. **input** x;
2. $y := 2$;
3. **while** $x > 0$ **do**
   4. $y := y + 2$;
   5. $x := x - 1$

   endw
6. **output** y;
7. **end**

Original program P:

1. **input** x;
2. $y := 2$;
3. **while** $x > 0$ **do**
   4. $y := y + 2$;
   5. $x := x - 1$

   endw
6. **output** y;
7. **end**

What's wrong?

$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

The right self-Interpreter

input $\mathtt{P}, d$; Program to be interpreted, and *its* data

$pc := 2$; Initialise program counter and store

$store := [in \mapsto d, out \mapsto 0, x_1 \mapsto 0, \ldots]$;

while $pc < length(\mathtt{P})$ **do**

$instruction := lookup(\mathtt{P}, pc)$; Find the pc-th instruction

case $instruction$ **of** Dispatch on syntax

skip : $pc := pc + 1$;

$x := e$: $store := store[x \mapsto eval(e, store)]$; $pc := pc + 1$;

$\ldots$ **endw** ;

output $store[out]$;

$eval(e, store) =$ **case** e **of** Function to evaluate expressions

$constant$: e

$variable$: $store(e)$

$e1 + e2$: $eval(e1, store) + eval(e2, store)$

$e1 - e2$: $eval(e1, store) - eval(e2, store)$

$e1 * e2$: $eval(e1, store) * eval(e2, store)$ $\ldots$

pc dynamic!

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Code Flattening

Original program $\texttt{P}$:

$^{1.}$**input** x;
$^{2.}$$y := 2$;
$^{3.}$**while** $x > 0$ **do**
  $^{4.}$$y := y + 2$;
  $^{5.}$$x := x - 1$
 endw
$^{6.}$**output** y;
$^{7.}$**end**

Flattened equivalent program $\texttt{P}'$:

$^{1.}$**input** x; $^{2.}$$pc := 2$;
$^{3.}$**while** $pc < 6$ **do**
  $^{4.}$**case** pc **of**
    2 : $^{5.}$$y := 2$; $^{6.}$$pc := 3$;
    3 : $^{7.}$**if** $x > 0$ **then** $^{8.}$$pc := 4$ **else** $^{9.}$$pc := 6$;
    4 : $^{10.}$$y := y + 2$; $^{11.}$$pc := 5$;
    5 : $^{12.}$$x := x - 1$; $^{13.}$$pc := 3$;
 endw
$^{14.}$**output** y
$^{15.}$**end**

$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

Why?

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The CFG Abstraction

$$\mathtt{P}_1 \left[\begin{array}{l} ^{1.}x := 0\,; \\ ^{2.}i := 1\,;\ ^{3.}\mathbf{while}\ i > 0\ \mathbf{do}\ ^{3B.}i := i + 1\,; \\ ^{4.}y := x\,;\ ^{5.} \end{array} \right.$$

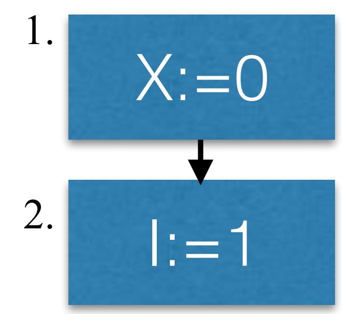
1. X:=0

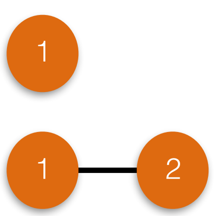
1

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The CFG Abstraction

$$\texttt{P}_1 \left[\begin{array}{l} ^{1.}\, x := 0\,; \\ ^{2.}\, i := 1\,;\ ^{3.}\,\textbf{while}\ i > 0\ \textbf{do}\ ^{3B.}\, i := i + 1\,; \\ ^{4.}\, y := x\,;\ ^{5.} \end{array} \right.$$

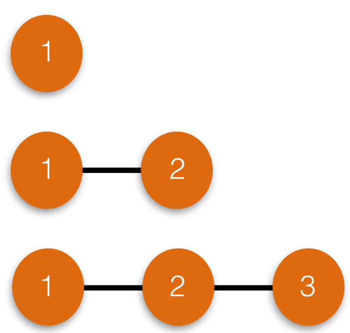


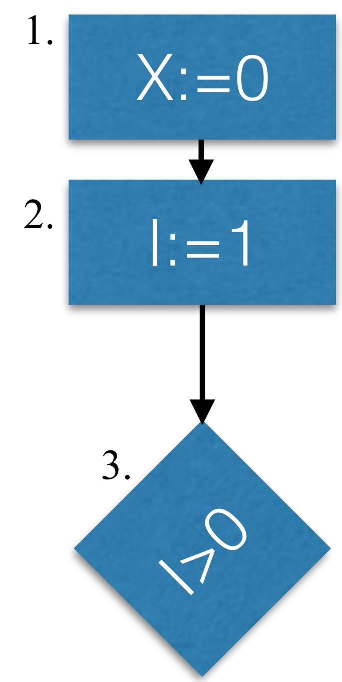


$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

The CFG Abstraction

$$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,; {}^{3.}\textbf{while } i > 0 \textbf{ do } {}^{3B.}i := i+1\,; \\ {}^{4.}y := x\,; {}^{5.} \end{array}\right.$$

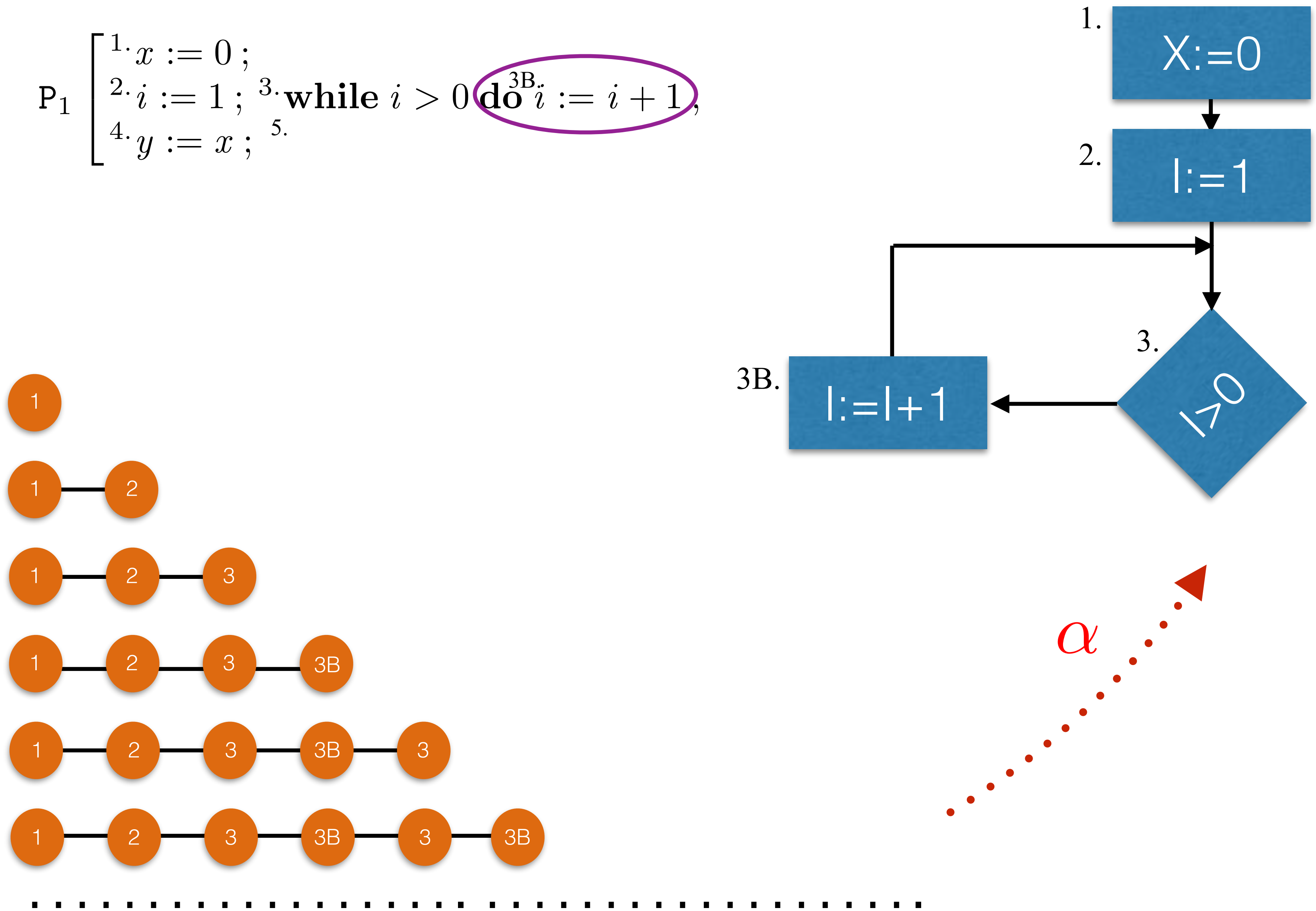




$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The CFG Abstraction

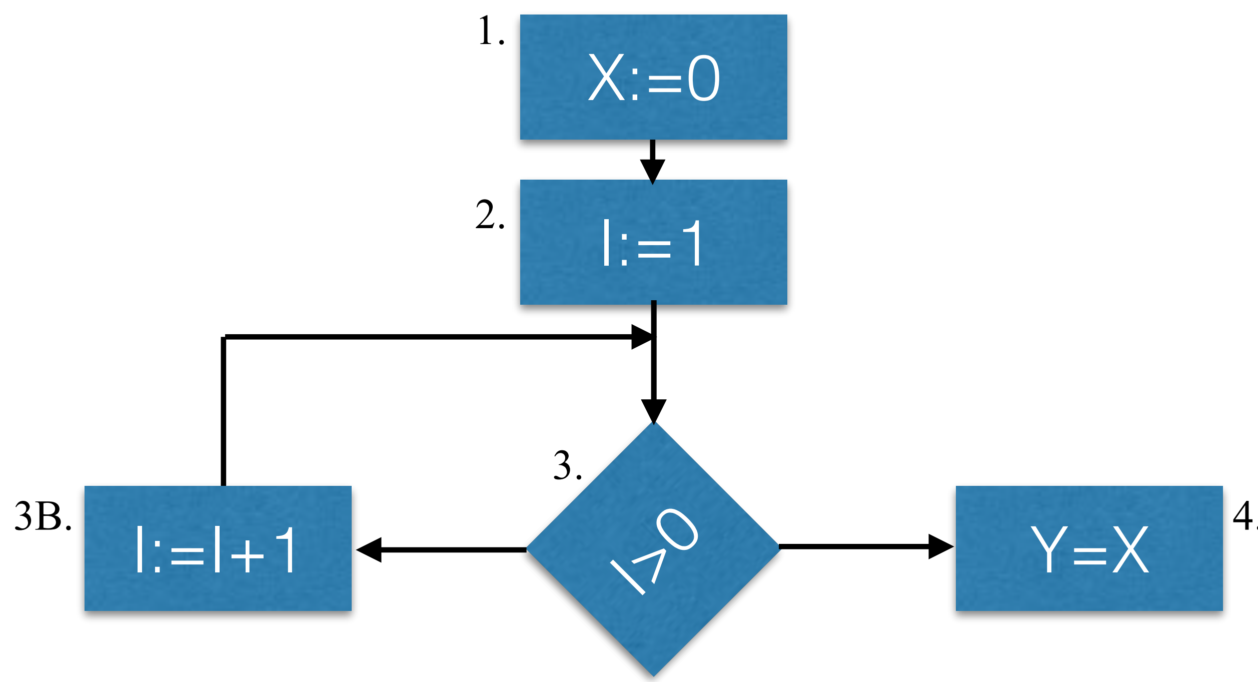
$$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\textbf{while}\ i > 0\ \textbf{do}\ {}^{3B.}i := i + 1, \\ {}^{4.}y := x\,;\ {}^{5.} \end{array} \right.$$

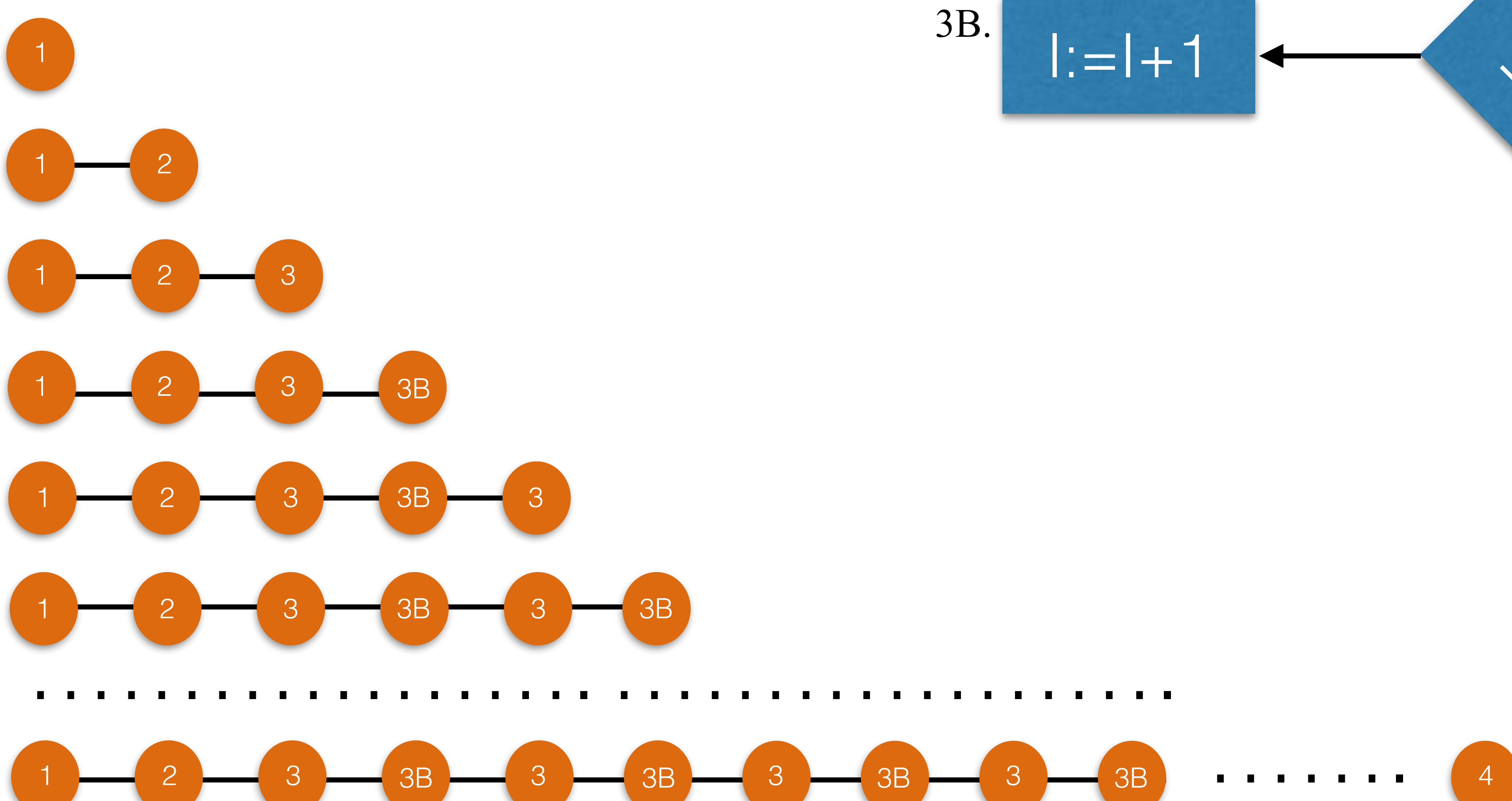


$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The CFG Abstraction

$$P_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\textbf{while } i > 0 \textbf{ do } {}^{3B.}i := i + 1\,; \\ {}^{4.}y := x\,;\ {}^{5.} \end{array}\right.$$

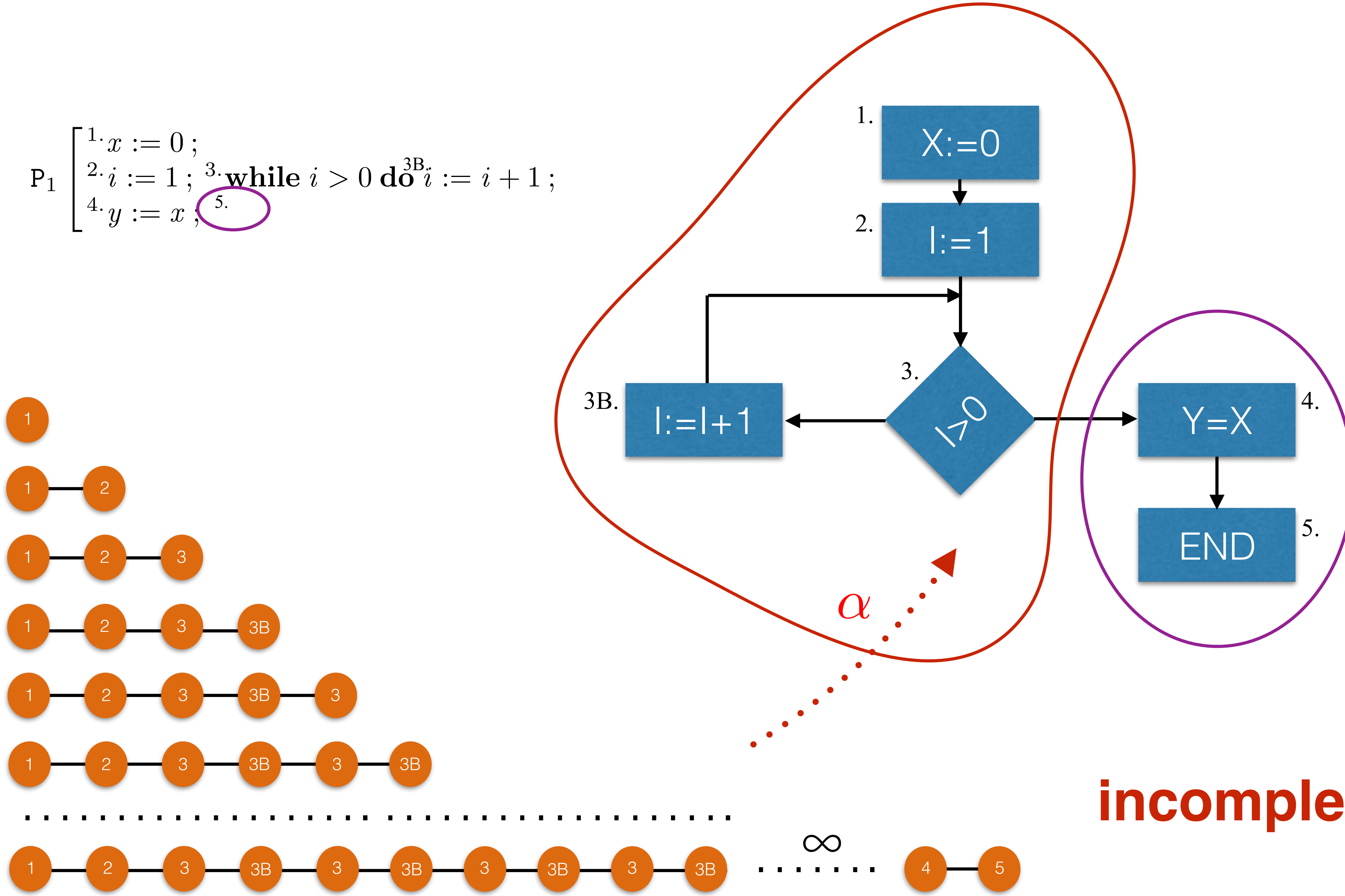




$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

The CFG Abstraction

$$\mathrm{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\mathbf{while}\ i > 0\ \mathbf{do}\ {}^{3B.}i := i + 1\,; \\ {}^{4.}y := x\,;\ {}^{5.} \end{array} \right.$$



incomplete!!

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The CFG Abstraction

- The attacker is an abstract interpreter extracting the CFG from `P`
  - ✔ forgets the computed memory $\mathbb{M}$: $\mathcal{C} = \lambda\sigma.\,\mathbb{M}$
  - ✔ forgets the branch computation when involving the `pc`: η
  - ✔ Fixpoint Graph semantics: $[\![\mathtt{P}]\!]_{\mathbb{G}} = \mathit{lfp}(\mathbb{G}_{\mathtt{P}})$

Static CFG extraction

- Theorem

pc dynamic!

$\mathcal{C}([\![\mathtt{P}]\!]_{\mathbb{G}}) = [\![\mathtt{P}]\!]_{\mathbb{G}}^{\mathcal{C},\eta}$ iff `pc` is not a program variable

Completeness!!

Flattening is distorting an interpreter making an abstract interpreter extracting the CFG incomplete

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

III: Slicing

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

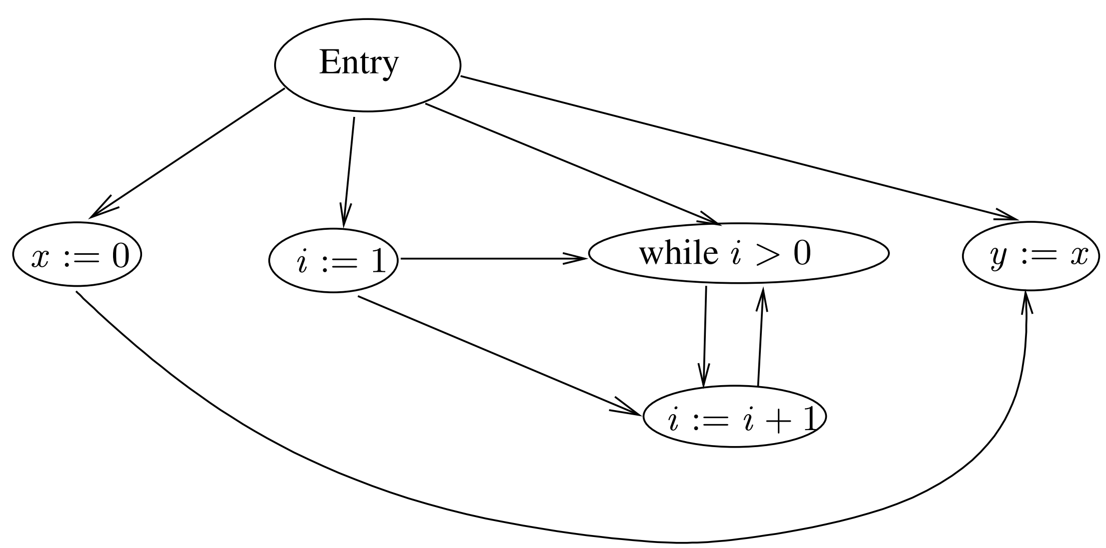
Program Slicing

M. Weiser 1981

*For a variable v and a statement (program point) s (final use of v), the **slice** S of program P with respect to the slicing criterion ⟨s,v⟩ is **any executable program such that S can be obtained by deleting zero or more statements from P and if P halts on input I then the value of v at the statement s, each time is reached in P, is the same in P and in S**.*

$$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\textbf{while}\ i > 0\ \textbf{do}\ i := i+1\,; \\ {}^{4.}y := x\,; \end{array}\right. \qquad \mathtt{P}_2 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ \\ {}^{4.}y := x\,; \end{array}\right.$$

Program Dependency Graph



$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Data Dependency Obfuscation

Word Count program
which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):

```
original() {
   int c, nl = 0, nw = 0, nc =0, in;
   in = F;
   while ((c = getchar()) ! = EOF) {
      nc ++;
      if (c == ' ' || c == '\n' || c == '\t') in = F;
      else if (in == F) {in = T; nw ++; }
      if (c == '\n') nl ++;
      }
   out(nl, nw, nc); }
```

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Data Dependency Obfuscation

Word Count program
which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):
Slicing criterion: nl

```
original() {
    int c, nl = 0, nw = 0, nc =0, in;
    in = F;
    while ((c = getchar()) ! = EOF) {
        nc ++;
        if (c == ‘ ’ || c == ‘\n’ || c == ‘\t’) in = F;
        else if (in == F) {in = T; nw ++; }
        if (c == ‘\n’) nl ++;
        }
    out(nl, nw, nc); }
```

$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

Data Dependency Obfuscation

Word Count program

which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):

Slicing criterion: nw

```
original() {
  int c, nl = 0, nw = 0, nc =0, in;
  in = F;
  while ((c = getchar()) ! = EOF) {
    nc ++;
    if (c == ' ' || c == '\n' || c == '\t') in = F;
    else if (in == F) {in = T; nw ++; }
    if (c == '\n') nl ++;
    }
  out(nl, nw, nc); }
```

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Data Dependency Obfuscation

Word Count program

which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):

```
obfuscated() {
   int c, nl = 0, nw = 0, nc =0, in;
   in = F;
   while ((c = getchar()) ! = EOF) {
      nc ++;
      if (c == ‘ ’ || c == ‘\n’ || c == ‘\t’) in = F;
      else if (in == F) {in = T; nw ++; }
      if (c == ‘\n’) {if (nw <= nc) nl ++; }
      if (nl > nc) nw = nc + nl;
      else {if (nw > nc) nc = nw – nl; }
      }
   out(nl, nw, nc); }
```

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathsf{spec}]\!](\mathtt{interp}, P)$$

Data Dependency Obfuscation

Word Count program
which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):

```
obfuscated() {
  int c, nl = 0, nw = 0, nc =0, in;
  in = F;
  while ((c = getchar()) ! = EOF) {
    nc ++;
    if (c == ‘ ’ || c == ‘\n’ || c == ‘\t’) in = F;
    else if (in == F) {in = T; nw ++; }
    if (c == ‘\n’) {if (nw <= nc) nl ++; }
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw – nl; }
    }
  out(nl, nw, nc); }
```

Always true

Always false

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Data Dependency Obfuscation

Word Count program

which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):

Slicing criterion: nl

```
obfuscated() {
  int c, nl = 0, nw = 0, nc =0, in;
  in = F;
  while ((c = getchar()) ! = EOF) {
    nc ++;
    if (c == ‘ ’ || c == ‘\n’ || c == ‘\t’) in = F;
    else if (in == F) {in = T; nw ++; }
    if (c == ‘\n’) {if (nw <= nc) nl ++; }
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw − nl; }
    }
  out(nl, nw, nc); }
```

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Data Dependency Obfuscation

Word Count program

which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):

Slicing criterion: nw

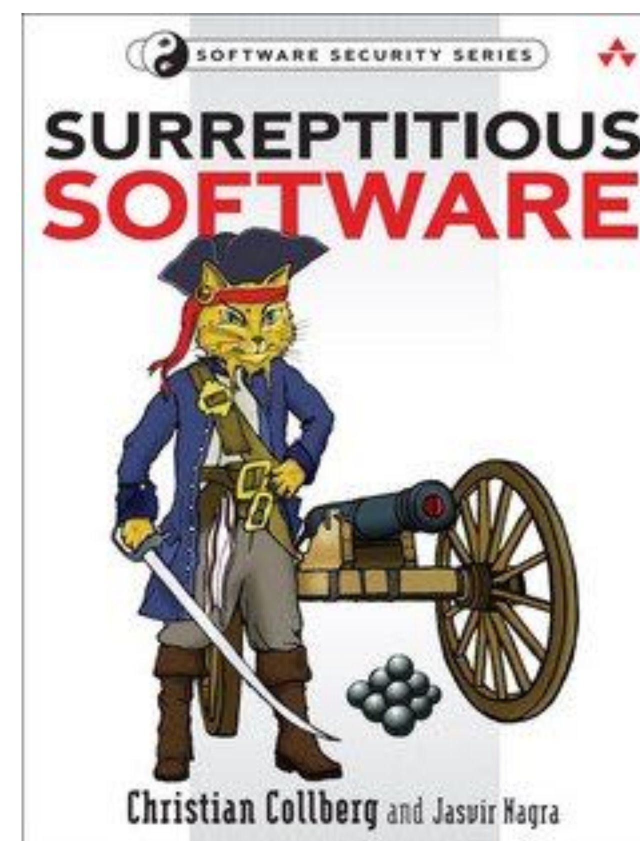
```
obfuscated() {
  int c, nl = 0, nw = 0, nc =0, in;
  in = F;
  while ((c = getchar()) ! = EOF) {
    nc ++;
    if (c == ' ' || c == '\n' || c == '\t') in = F;
    else if (in == F) {in = T; nw ++; }
    if (c == '\n') {if (nw <= nc) nl ++; }
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw – nl; }
    }
  out(nl, nw, nc); }
```

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Opaque Predicates

Examples of opaque predicates from number theory

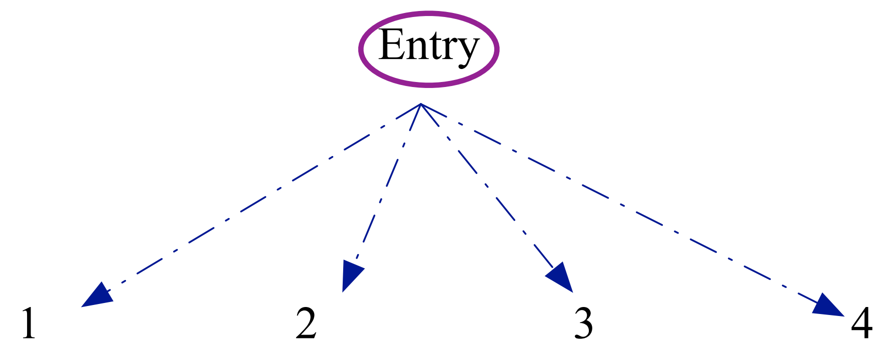
$$
\begin{aligned}
\forall x, y \in \mathbb{Z}: &\quad 7y^2 - 1 \neq x^2 \\
\forall x \in \mathbb{Z}: &\quad 2 \mid (x + x^2) \\
\forall x \in \mathbb{Z}: &\quad 3 \mid (x^3 - x) \\
\forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: &\quad (x - y) \mid (x^n - y^n) \\
\forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: &\quad 2 \mid n \ \vee \ (x + y) \mid (x^n + y^n) \\
\forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: &\quad 2 \nmid n \ \vee \ (x + y) \mid (x^n - y^n) \\
\forall x \in \mathbb{Z}^+: &\quad 9 \mid (10^x + 3 \cdot 4^{(x+2)} + 5) \\
\forall x \in \mathbb{Z}: &\quad 3 \mid (7x - 5) \ \Rightarrow \ 9 \mid (28x^2 - 13x - 5) \\
\forall x \in \mathbb{Z}: &\quad 5 \mid (2x - 1) \ \Rightarrow \ 25 \mid (14x^2 - 19x - 19) \\
\forall x, y, z \in \mathbb{Z}: &\quad (2 \nmid x \wedge 2 \nmid y) \ \Rightarrow \ x^2 + y^2 \neq z^2 \\
\forall x \in \mathbb{Z}^+: &\quad 14 \mid (3 \cdot 7^{4x+2} + 5 \cdot 4^{2x-1} - 5)
\end{aligned}
$$



The PDG Abstraction

$$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\textbf{while}\ i > 0\ \textbf{do}\ i := i+1\,; \\ {}^{4.}y := x\,; \end{array} \right.$$

$D_{\text{Entry}} = \varnothing \qquad s = \langle \sigma, \langle \text{Entry}, 1 \rangle \rangle$

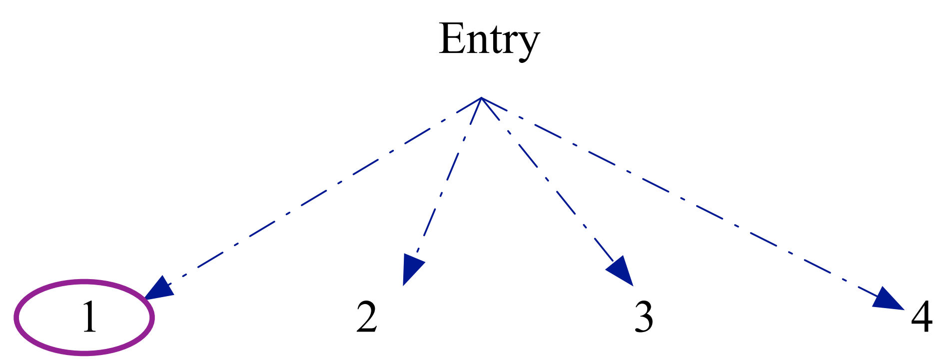


$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The PDG Abstraction

$$\texttt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\mathbf{while}\ i > 0\ \mathbf{do}\ i := i + 1\,; \\ {}^{4.}y := x\,; \end{array}\right.$$

$D_1 = [D_x = 1]$ $\qquad s = \langle \sigma, \langle 1, 2 \rangle \rangle$

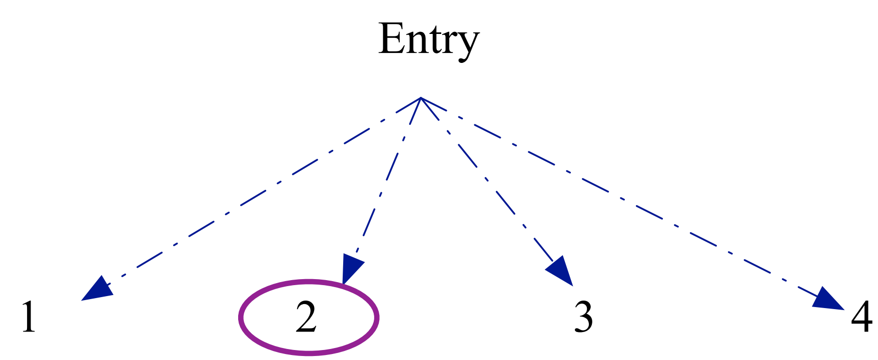


$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

The PDG Abstraction

$$\mathtt{P}_1 \left[\begin{array}{l} ^{1.}x := 0\,; \\ ^{2.}i := 1\,; \; ^{3.}\textbf{while } i > 0 \textbf{ do } i := i+1\,; \\ ^{4.}y := x\,; \end{array}\right.$$

$D_2 = [D_x = 1, D_i = 2]$ $s = \langle \sigma, \langle 2, 3 \rangle \rangle$

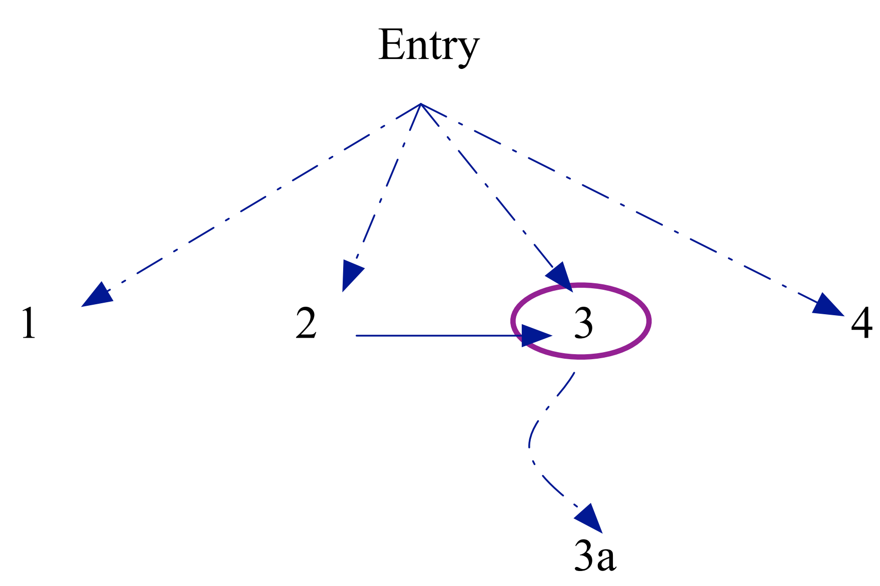


$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The PDG Abstraction

$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,; {}^{3.}\textbf{while } i > 0 \textbf{ do } i := i + 1\,; \\ {}^{4.}y := x\,; \end{array} \right.$

$D_3 = [D_x = 1, D_i = 2]$ $\quad$ $s_1 = \langle \sigma, \langle 3, 3a \rangle \rangle$ and $s_2 = \langle \sigma, \langle 3, 4 \rangle \rangle$

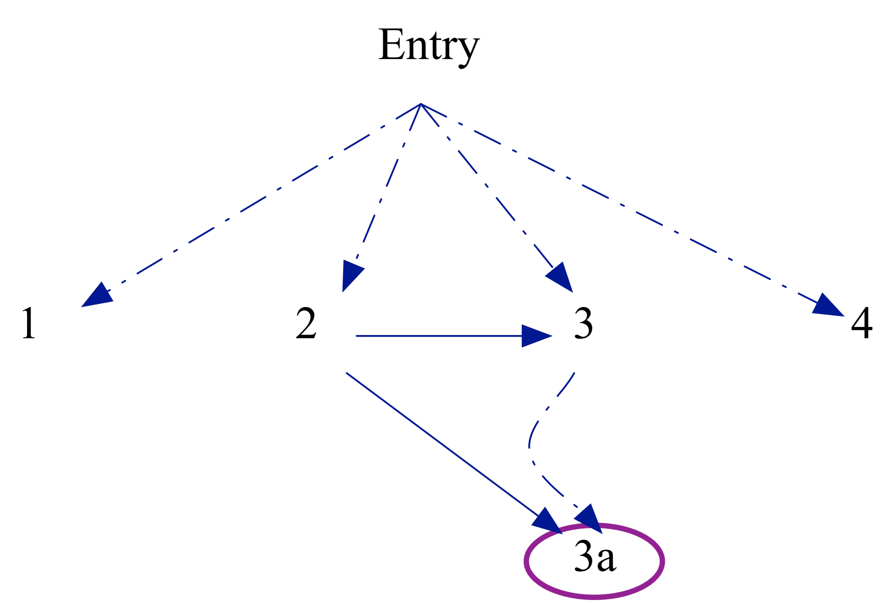


$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The PDG Abstraction

$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.} x := 0\,; \\ {}^{2.} i := 1\,;\ {}^{3.}\textbf{while } i > 0 \textbf{ do } i := i + 1\,; \\ {}^{4.} y := x\,; \end{array} \right.$

$D_{3a} = [D_x = 1, D_i = 3a]$ $\quad$ $s_1 = \langle \sigma, \langle 3a, 3 \rangle \rangle$ and $s_2 = \langle \sigma, \langle 3, 4 \rangle \rangle$

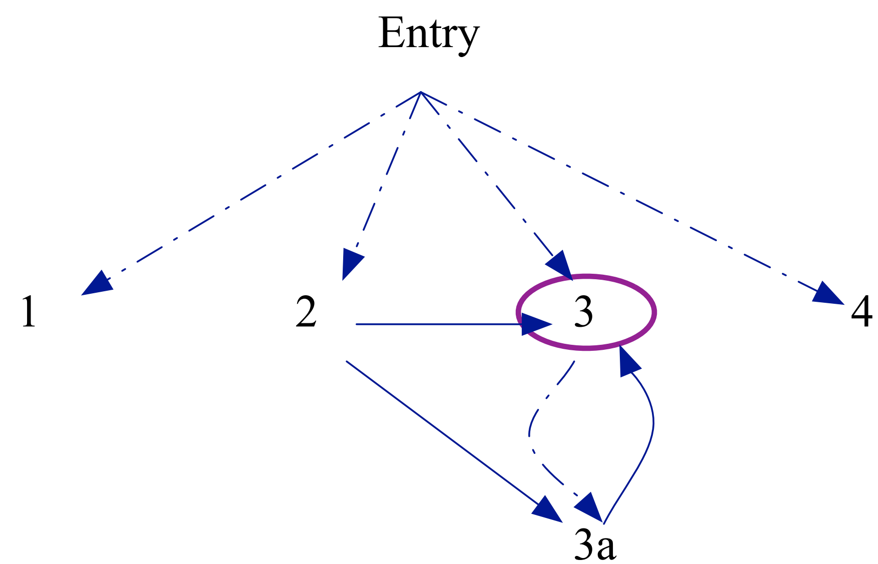


$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The PDG Abstraction

$$\mathtt{P}_1 \left[\begin{array}{l} {}^{1.}x := 0\,; \\ {}^{2.}i := 1\,;\ {}^{3.}\textbf{while}\ i > 0\ \textbf{do}\ i := i + 1\,; \\ {}^{4.}y := x\,; \end{array}\right.$$

$D_{3a} = [D_x = 1, D_i = 3a]$ $s_1 = \langle \sigma, \langle 3, 3a \rangle\rangle$ and $s_2 = \langle \sigma, \langle 3, 4 \rangle\rangle$



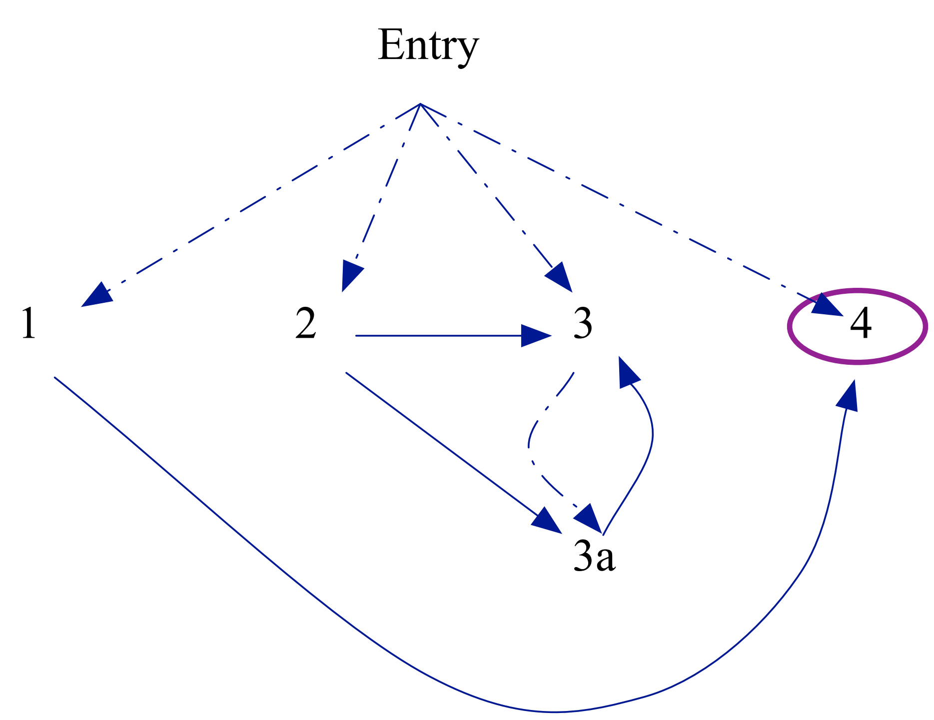
fixpoint

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

The PDG Abstraction

$$\mathtt{P}_1 \left[\begin{array}{l} ^{1.} x := 0\,; \\ ^{2.} i := 1\,;\ ^{3.}\textbf{while } i > 0 \textbf{ do } i := i + 1\,; \\ ^{4.} y := x\,; \end{array} \right.$$

$D_4 = [D_x = 1, D_i = 2, D_y = 4]$ $s_1 = \langle \sigma, \langle 3, 3a \rangle \rangle$ and $s_2 = \langle \sigma, \langle 4, \perp \rangle \rangle$

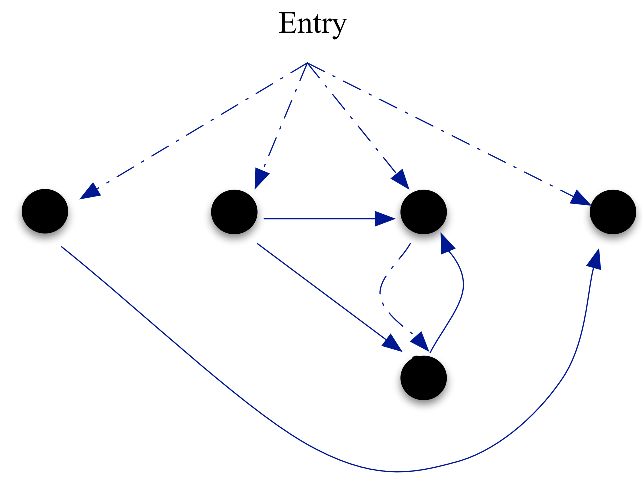


$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Slicing Obfuscation

$M_1 \quad \ldots \quad M_n$

Syntactic PDG



True dependencies

Theorem

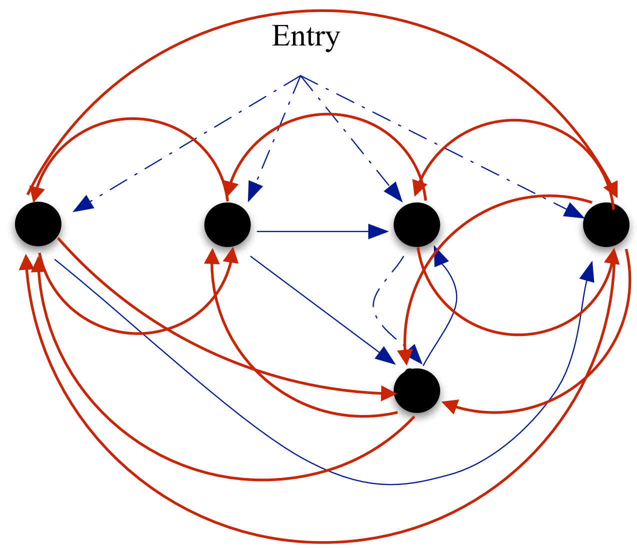
The Semantic PDG is complete (i.e., the PDG analysis is precise) iff the program does not contain *fake* dependencies

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Slicing Obfuscation

$\mathcal{O}(P)$

Syntactic PDG



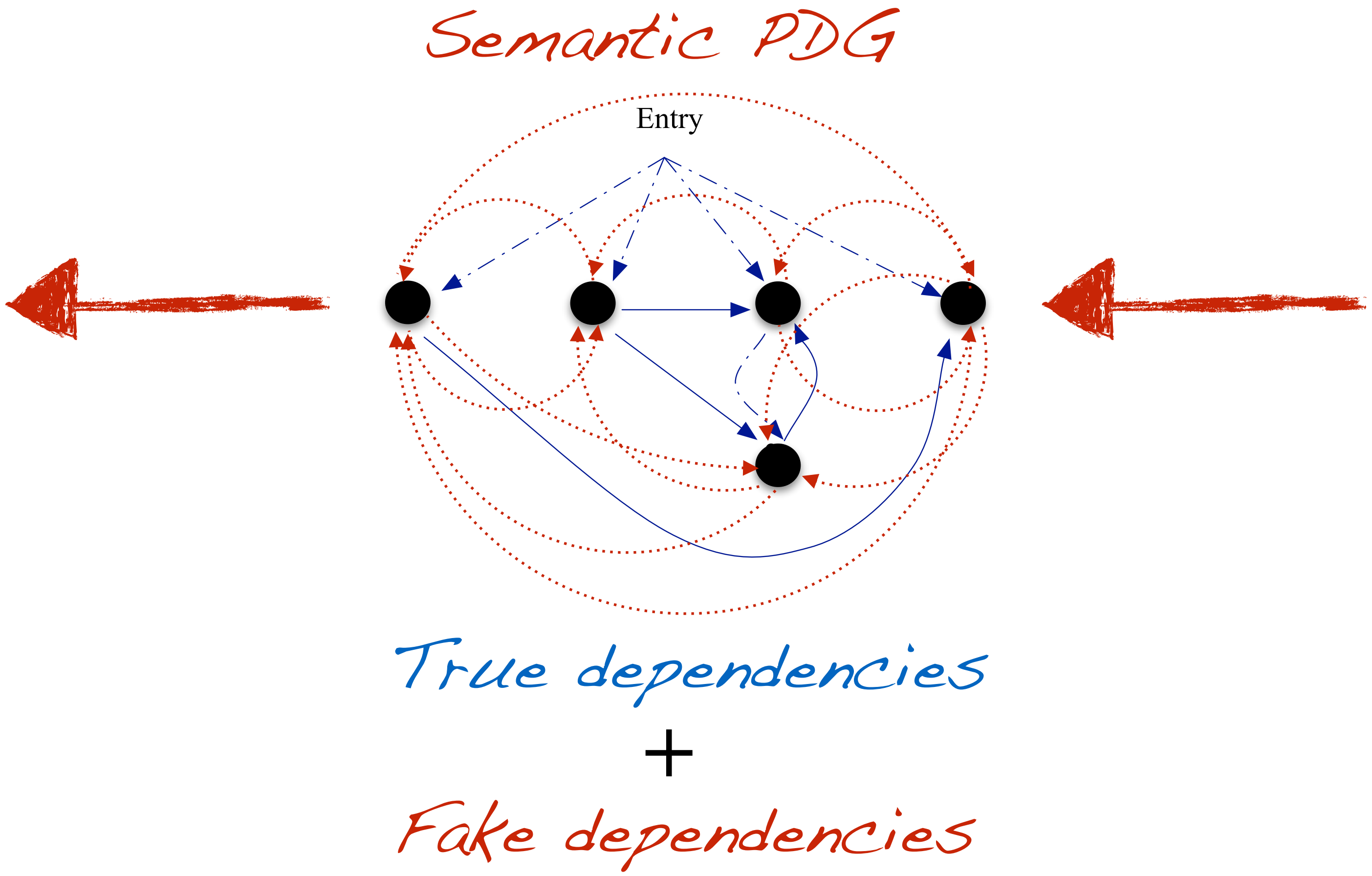
True dependencies
+
Fake dependencies

Theorem

The Semantic PDG is complete (i.e., the PDG analysis is precise) iff the program does not contain *fake* dependencies

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

Slicing Obfuscation



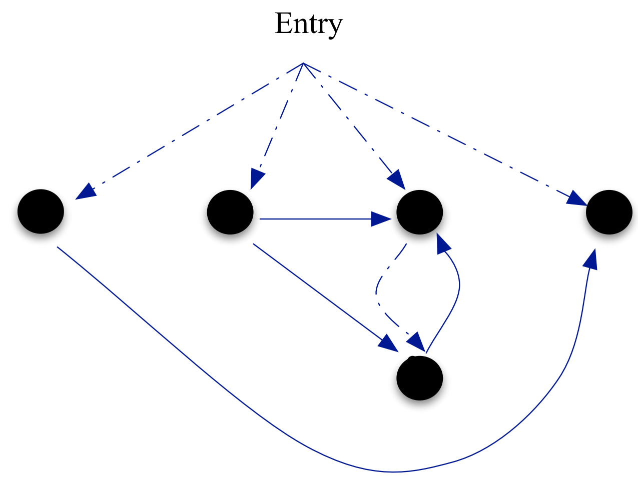
Theorem

The Semantic PDG is complete (i.e., the PDG analysis is precise) iff the program does not contain *fake* dependencies

$$\texttt{Obf}_{\alpha}(P) = [\![\texttt{spec}]\!](\texttt{interp}, P)$$

Slicing Obfuscation

Semantic PDG



$\mathcal{O}(P)$

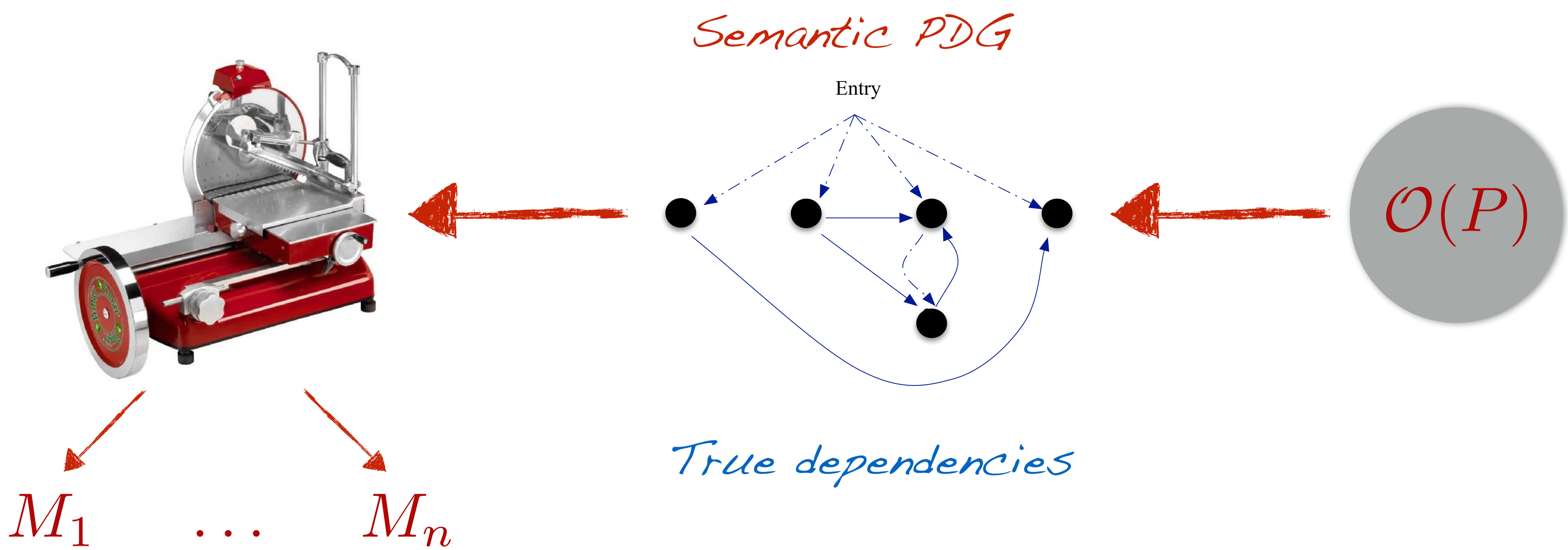
True dependencies

Theorem

The Semantic PDG is complete (i.e., the PDG analysis is precise) iff the program does not contain *fake* dependencies

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)$$

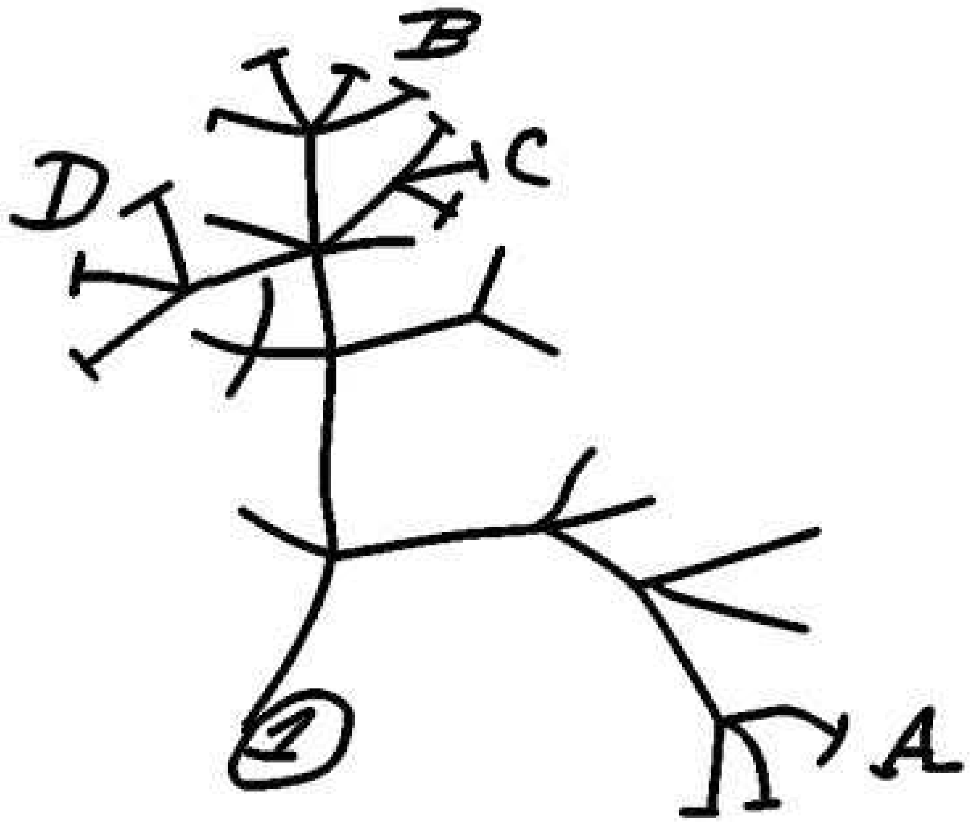
Slicing Obfuscation



Theorem

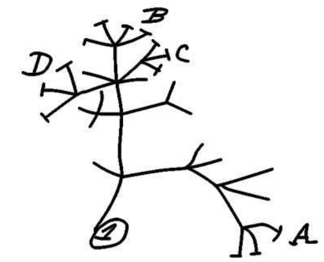
The Semantic PDG is complete (i.e., the PDG analysis is precise) iff the program does not contain *fake* dependencies

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathrm{spec}]\!](\mathtt{interp}, P)$$



Conclusions

So what?

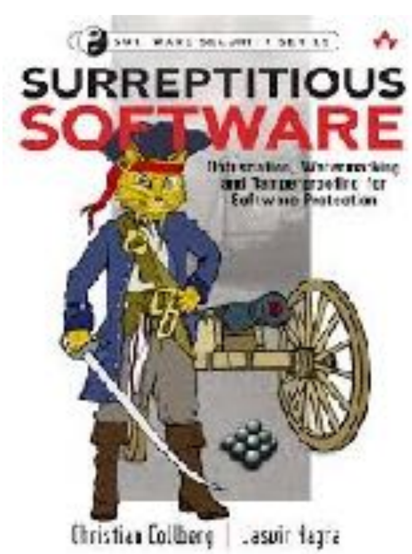


Any obfuscation technique is an instance of

$$\mathtt{Obf}_{\alpha}(P) = [\![\mathtt{spec}]\!](\mathtt{interp}, P)^{\alpha}$$

!

for some $\mathtt{interp}^{\alpha}$ making an abstraction α incomplete!



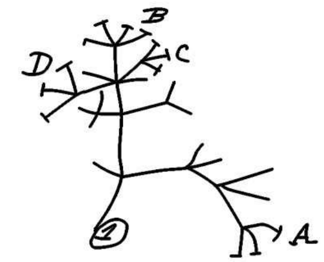
- Profiling: Abstract memory keeping only (partial) resource usage
- Tracing: Abstraction of traces (e.g., by trace compression)
- Slicing: Abstraction of traces (relative to variables)
- Monitoring: Abstraction of trace semantics ([Cousot&Cousot POPL02])
- Decompilation: Abstracts syntactic structures (e.g., reducible loops)
- Disassembly: Abstracts binary structures (e.g., recursive traversal)

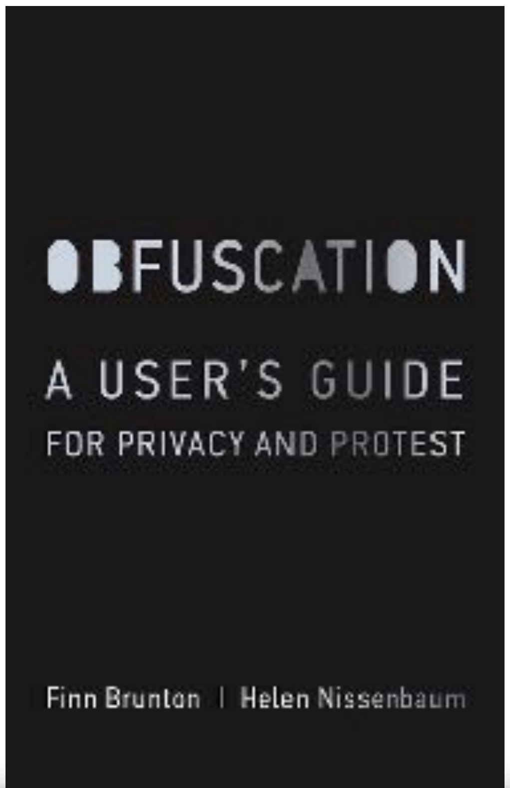
Given an obfuscated code P, what is α?

Given α, can we derive $\mathtt{interp}^{\alpha}$ systematically?

?

The Challenges





- **Correctness**: a provably correct obfuscation?
- Is obscured code still **secure**?
- **Adaptable** interpreters and specialisers for dynamic obfuscation?
- **Measures** of behaviour leakage ?
- **Automated Machine Learning** reverse engineering as attack models?

$$
\begin{array}{llll}
1. & \mathtt{P}' & := \llbracket \mathtt{spec} \rrbracket (\mathtt{interp}^{\alpha}, \mathtt{P}) & \text{Transform program} \\
2. & \mathtt{comp} & := \llbracket \mathtt{spec} \rrbracket (\mathtt{spec}, \mathtt{interp}^{\alpha}) & \text{Generate transformer} \\
3. & \mathtt{cogen} & := \llbracket \mathtt{spec} \rrbracket (\mathtt{spec}, \mathtt{spec}) & \text{Transformer generator}
\end{array}
$$

The Challenges

NII 湘南会議 SHONAN MEETING

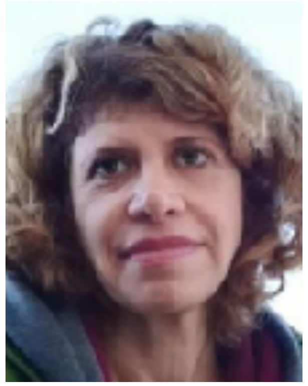
Intentional and extensional aspects of computation:
From computability and complexity to program analysis and security

Shonan Village Center, Japan
January 22-25, 2018

Thanks!

Isabella Mila Neil Sandrine

Roberta Roberto

&

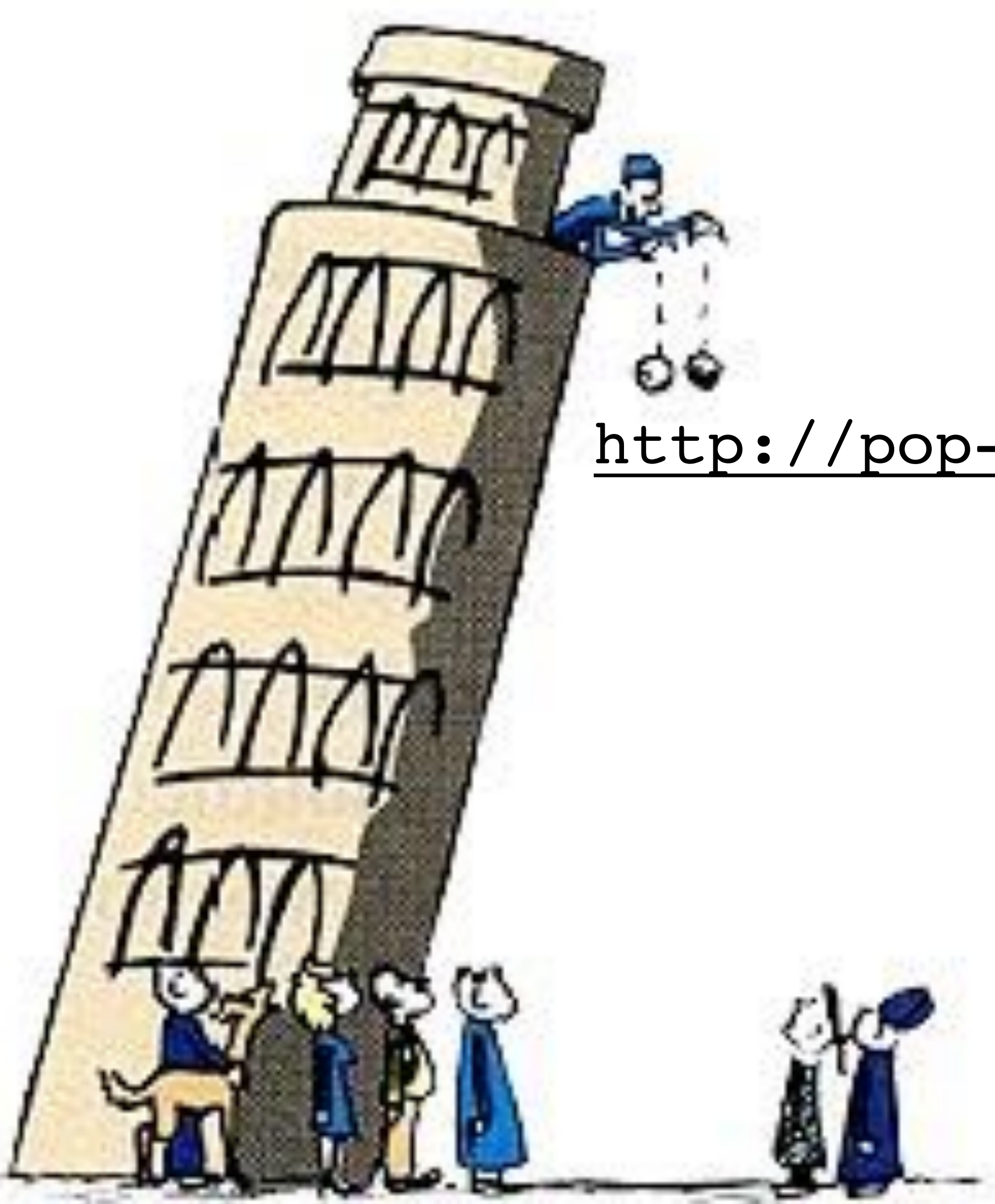
 

Francesco Francesco

Obfuscation & Security

Completeness

Experiments



The InterProc Analyser

http://pop-art.inrialpes.fr/interproc/interprocweb.cgi

OCaml

Example

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
  if (n>100) then
     r = n-10;
  else
     t1 = n + 11;
     t2 = MC(t1);
     r = MC(t2);
  endif;
end

var
a:int, b:int;
begin
  b = MC(a);
end
```

Nested recursive function by John McCarthy!

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$$

Example

```
Annotated program after forward analysis
proc MC (n : int) returns (r : int) var t1 : int, t2 : int;
begin
  /* (L5 C5) top */
  if n > 100 then
     /* (L6 C17) [|n-101>=0|] */
     r = n - 10; /* (L7 C14)
                 [|n-101>=0; r-91>=0|] */
  else
    /* (L8 C6) [|-n+100>=0|] */
    t1 = n + 11; /* (L9 C17)
                 [|-n+100>=0; -t1+111>=0|] */
    t2 = MC(t1); /* (L10 C17)
                 [|-n+100>=0; -t1+111>=0; t2-91>=0|] */
    r = MC(t2); /* (L11 C16)
                [|-n+100>=0; r-91>=0; -t1+111>=0; t2-91>=0|] */
  endif; /* (L12 C8) [|r-91>=0|] */
end

var a : int, b : int;
begin
  /* (L17 C5) top */
  b = MC(a); /* (L18 C12) [|b-91>=0|] */
end
```

!!

Obfuscation

Source

```
var i:int;
begin
  i = 0;
  while (i<=200) do
    i = i+1;
  done;
end
```

```
var i:int, j:int;
begin
  i = 0;
  j = 0;
  while ((10*i+j)<=200) do
    i = i+(j+1) /_i,0 10 ;
    j = (j + 1) % 10;
  done;
i = 10*i+j;
end
```

Result

```
Annotated program after forward analysis
var i : int;
begin
  /* (L2 C5) top */
  i = 0; /* (L3 C8) [|i>=0; -i+201>=0|] */
  while i <= 200 do
    /* (L4 C19) [|i>=0; -i+200>=0|] */
    i = i + 1; /* (L5 C12)
                [|i-1>=0; -i+201>=0|] */
  done; /* (L6 C7) [|i-201=0|] */
end
```

```
Annotated program after forward analysis
var i : int, j : int;
begin
  /* (L2 C5) top */
  i = 0; /* (L3 C8) [|i=0|] */
  j = 0; /* (L4 C8)
            [|i>=0; -i+21>=0; j>=0; -j+10>=0|] */
  while 10 * i + j <= 200 do
    /* (L5 C26)
       [|i>=0; -i+20>=0; j>=0; -j+10>=0|] */
    i = i + (j + 1) /_i,0 10; /* (L6 C26)
                                 [|i>=0; -i+21>=0; j>=0; -j+10>=0|] */
    j = (j + 1) % 10; /* (L7 C21)
                         [|i>=0; -i+21>=0; j>=0; -j+10>=0|] */
  done; /* (L8 C7)
           [|i-20>=0; -i+21>=0; j>=0; -j+10>=0|] */
  i = 10 * i + j; /* (L9 C11)
                     [|i-200>=0; -i+220>=0; j>=0; -j+10>=0|] */
end
```

Obscure!

Obfuscation

Source

```
var i:int;
begin
  i = 0;
  while (i<=10) do
    i = i+1;
  done;
end
```

```
var i:int, j:int;
begin
  i = 0; j=0;
  while (j<=10) do
    i =i+1;
    j =j+1;
  done;
end
```

Result

```
Annotated program after forward analysis
var i : int;
begin
  /* (L2 C5) top */
  i = 0; /* (L3 C8) [|i>=0; -i+11>=0|] */
  while i <= 10 do
    /* (L4 C18) [|i>=0; -i+10>=0|] */
    i = i + 1; /* (L5 C12)
                  [|i-1>=0; -i+11>=0|] */
  done; /* (L6 C7) [|i-11=0|] */
end
```

```
Annotated program after forward analysis
var i : int, j : int;
begin
  /* (L2 C5) top */
  i = 0; /* (L3 C8) [|i=0|] */
  j = 0; /* (L3 C13) [|i>=0; j>=0; -j+11>=0|] */
  while j <= 10 do
    /* (L4 C18) [|i>=0; j>=0; -j+10>=0|] */
    i = i + 1; /* (L5 C11)
                  [|i-1>=0; j>=0; -j+10>=0|] */
    j = j + 1; /* (L6 C12)
                  [|i-1>=0; j-1>=0; -j+11>=0|] */
  done; /* (L7 C7) [|i>=0; j-11=0|] */
end
```

Obscure!

With Octagons

Source

```
var i:int;
begin
  i = 0;
  while (i<=10) do
    i = i+1;
  done;
end
```

```
var i:int, j:int;
begin
  i = 0; j=0;
  while (j<=10) do
    i =i+1;
    j =j+1;
  done;
end
```

Result

```
Annotated program after forward analysis
var i : int;
begin
  /* (L2 C5) top */
  i = 0; /* (L3 C8) [|i>=0; -i+11>=0|] */
  while i <= 10 do
    /* (L4 C18) [|i>=0; -i+10>=0|] */
    i = i + 1; /* (L5 C12)
                  [|i-1>=0; -i+11>=0|] */
  done; /* (L6 C7) [|i-11>=0; -i+11>=0|] */
end
```

```
Annotated program after forward analysis
var i : int, j : int;
begin
  /* (L2 C5) top */
  i = 0; /* (L3 C8) [|i>=0; -i>=0|] */
  j = 0; /* (L3 C13)
            [|i>=0; -i+11>=0; -i+j>=0; i+j>=0; j>=0; -i-j+22>=0; i-j>=0;
              -j+11>=0|] */
  while j <= 10 do
    /* (L4 C18)
       [|i>=0; -i+10>=0; -i+j>=0; i+j>=0; j>=0; -i-j+20>=0; i-j>=0; -j+10>=0|] */
    i = i + 1; /* (L5 C11)
                  [|i-1>=0; -i+11>=0; -i+j+1>=0; i+j-1>=0; j>=0; -i-j+21>=0;
                    i-j-1>=0; -j+10>=0|] */
    j = j + 1; /* (L6 C12)
                  [|i-1>=0; -i+11>=0; -i+j>=0; i+j-2>=0; j-1>=0; -i-j+22>=0;
                    i-j>=0; -j+11>=0|] */
  done; /* (L7 C7)
           [|i-11>=0; -i+11>=0; -i+j>=0; i+j-22>=0; j-11>=0; -i-j+22>=0;
             i-j>=0; -j+11>=0|] */
end
```

Let us play!!!

http://pop-art.inrialpes.fr/interproc/interprocweb.cgi

Obfuscation on Linear Relations

Source

```
proc incr (x:int) returns (y:int)
begin
  y = x+1;
end

var i:int;
begin
  i = 0;
  while (i<=10) do
    i = incr(i);
  done;
end
```

?

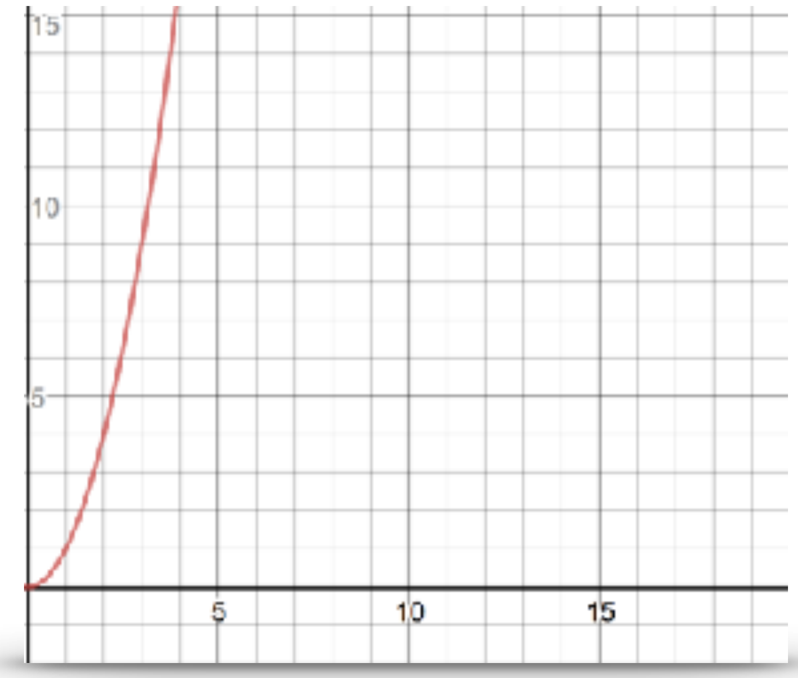
Result

```
Annotated program after forward analysis
proc incr (x : int) returns (y : int) var ;
begin
  /* (L3 C5) top */
  y = x + 1; /* (L4 C10) [|-x+y-1=0|] */
end

var i : int;
begin
  /* (L8 C5) top */
  i = 0; /* (L9 C8) top */
  while i <= 10 do
    /* (L10 C18) top */
    i = incr(i); /* (L11 C16) top */
  done; /* (L12 C7) top */
end
```

Obfuscation on Octagons

Source

```
var i:int;
begin
  i = 0;
  while (i<=10) do
    i = i+1;
  done;
end
```



Result

```
Annotated program after forward analysis
var i : int, j : int;
begin
  /* (L2 C5) top */
  i = 0; /* (L3 C8) [|i>=0; -i>=0|] */
  j = 1; /* (L3 C12)
            [|i>=0; -i+1024>=0; -i+j-1>=0; i+j-1>=0; j-1>=0; -i-j+3072>=0;
             i-j+2047>=0; -j+2048>=0|] */
  while j <= 1024 do
    /* (L4 C20)
       [|i>=0; -i+1023>=0; -i+j-1>=0; i+j-1>=0; j-1>=0; -i-j+2047>=0;
        i-j+1024>=0; -j+1024>=0|] */
    j = j * 2; /* (L5 C12)
                  [|i>=0; -i+1023>=0; -i+j-2>=0; i+j-2>=0; j-2>=0;
                   -i-j+3071>=0; i-j+2048>=0; -j+2048>=0|] */
    i = i + 1; /* (L6 C12)
                  [|i-1>=0; -i+1024>=0; -i+j-1>=0; i+j-3>=0; j-2>=0;
                   -i-j+3072>=0; i-j+2047>=0; -j+2048>=0|] */
  done; /* (L7 C7)
           [|i>=0; -i+1024>=0; -i+j-1>=0; i+j-1025>=0; j-1025>=0;
            -i-j+3072>=0; i-j+2047>=0; -j+2048>=0|] */
end
```

```
var i:int, j:int;
begin
  i = 0;j=1;
  while (j<=1024) do
    j = j*2;
    i = i+1;
  done;
end
```

Hacking McCarthy 91

Source

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
  if (n>100) then
     r = n-10;
  else
     t1 = n + 11;
     t2 = MC(t1);
     r = MC(t2);
  endif;
end

var
a:int, b:int;
begin
  b = MC(a);
end
```

→ ?

Hacking McCarthy 91

Source

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
  if (n>100) then
     r = n-10;
  else
     t1 = n + 11;
     t2 = MC(t1);
     r = MC(t2);
  endif;
end

var
a:int, b:int;
begin
  b = MC(a);
end
```

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int, p:int;
begin
  p = n*2;
  if (p>200) then
     r = (p-20)/2;
  else
     if (n*(n-1))%2==0 then
        t1 = (p + 22)/2;
        t2 = MC(t1);
        r = MC(t2);
     else
        r=50;
     endif;
  endif;
end

var
a:int, b:int;
begin
  b = MC(a);
end

See More from Vivek Notani
```

Try it!!!

Hacking McCarthy 91

Source

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
  if (n>100) then
     r = n-10;
  else
     t1 = n + 11;
     t2 = MC(t1);
     r = MC(t2);
  endif;
end

var
a:int, b:int;
begin
  b = MC(a);
end
```

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int, p:int;
begin
  p = n*n;
  if (p>10000) then
     r = (p/n-10);
  else
     t1 = (p + 11*n)/n;
     t2 = MC(t1);
     r = MC(t2);
  endif;
end

var
a:int, b:int;
begin
  b = MC(a);
end
```

Try it!!!

Why this works?

```
var i : int;
begin
  /* (L4 C5) top */
  i = 0; /* (L5 C8) [|i>=0; -i+11>=0|] */
  while i <= 10 do
    /* (L6 C18) [|i>=0; -i+10>=0|] */
    i = i + 1; /* (L7 C14)
                  [|i-1>=0; -i+11>=0|] */
  done; /* (L8 C7) [|i-11=0|] */
end
```

(a) Complete Analysis

```
var i : int;
begin
  /* (L4 C5) top */
  i = 0; /* (L5 C8) [|i>=0; -i+106>=0|] */
  while i <= 10 do
    /* (L6 C18) [|i>=0; -i+10>=0|] */
    if i == 5 then
      /* (L7 C18) [|i-5=0|] */
      i = i + 100; /* (L8 C19) [|i-105=0|] */
    endif; /* (L9 C10) [|i>=0; -i+105>=0|] */
    if i == 105 then
      /* (L10 C20) [|i-105=0|] */
      i = i - 100; /* (L11 C19) [|i-5=0|] */
    endif; /* (L12 C10) [|i>=0; -i+105>=0|] */
    i = i + 1; /* (L13 C14)
                  [|i-1>=0; -i+106>=0|] */
  done; /* (L14 C7) [|i-11>=0; -i+106>=0|] */
end
```

(b) Incomplete Analysis

Ideas??