Obscuring Code

Unveiling and Veiling Information in Programs

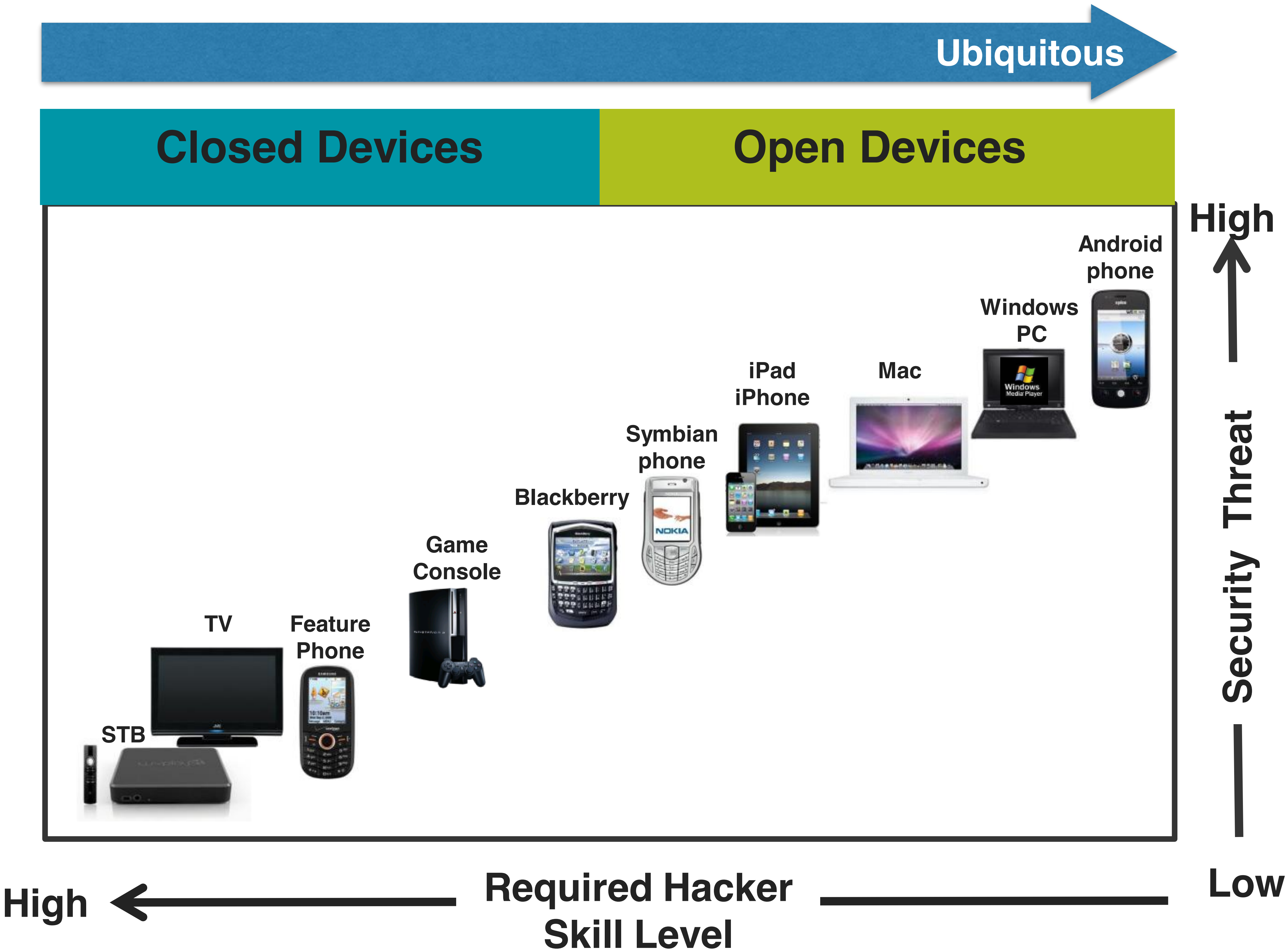
Roberto Giacobazzi

University of Verona

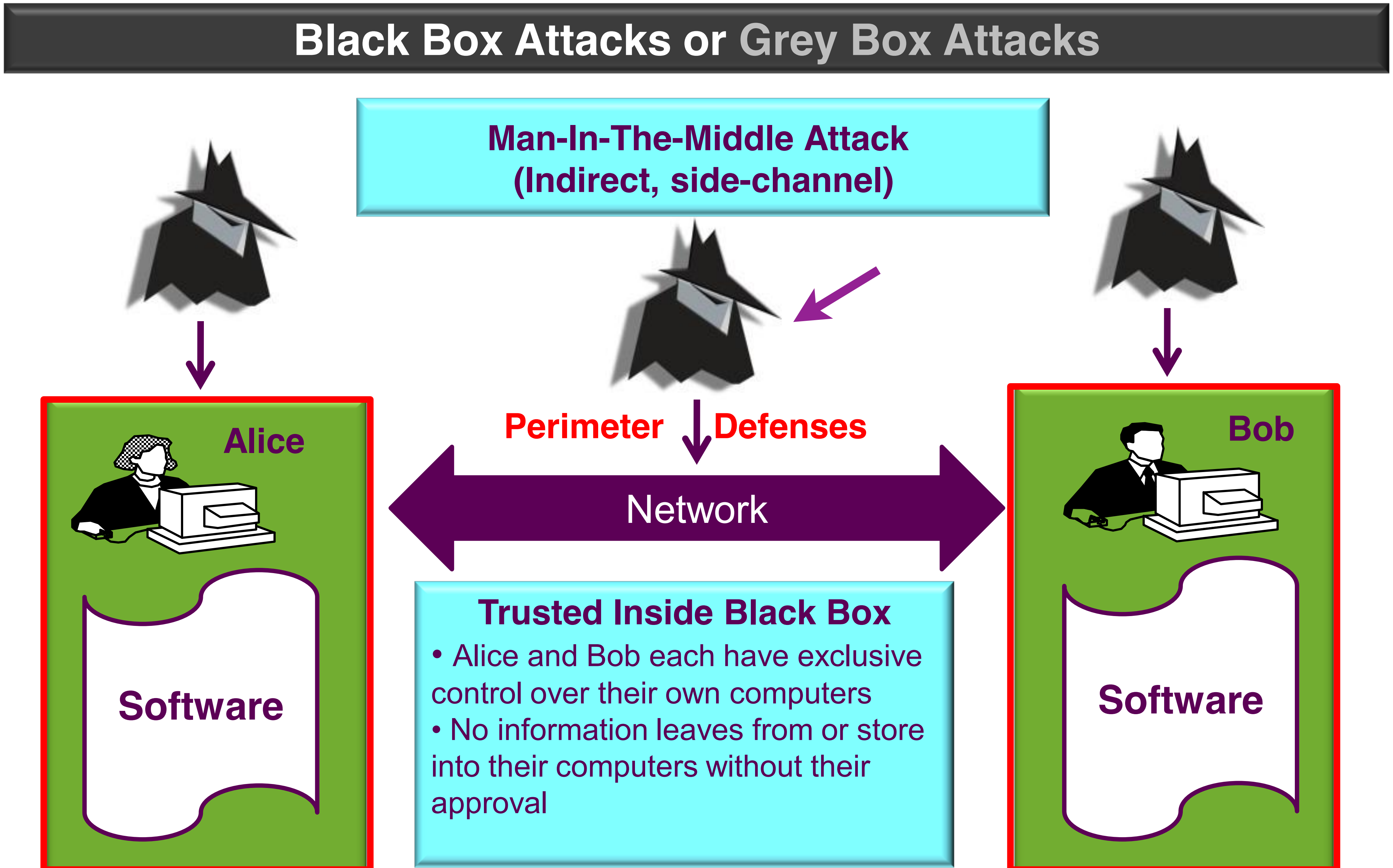
&

irdeto

DAPA
Digital Asset Protection Assn.


Ubiquitous
Closed Devices
Open Devices
STB
TV
Feature Phone
Game Console
Blackberry
Symbian phone
iPad iPhone
Mac
Windows PC
Android phone
High
Security Threat
Low
High
Required Hacker Skill Level

Crypto Assumption



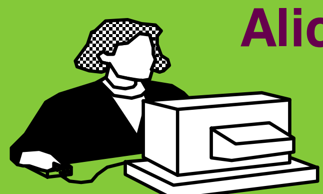
White-Box Attacks

Attackers have open-end powers to do

- Trace every program instruction
- View the contents of memory and cache
- Stop execution at any point and run an off-line process
- Alter code or memory at will
- Do all of this for as long as they want, whenever they want, in collusion with as many other attackers as they can find

Man-At-The-End Attack

Alice



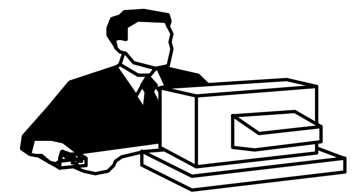
Software

Attacking has much less limitation than protection

Network

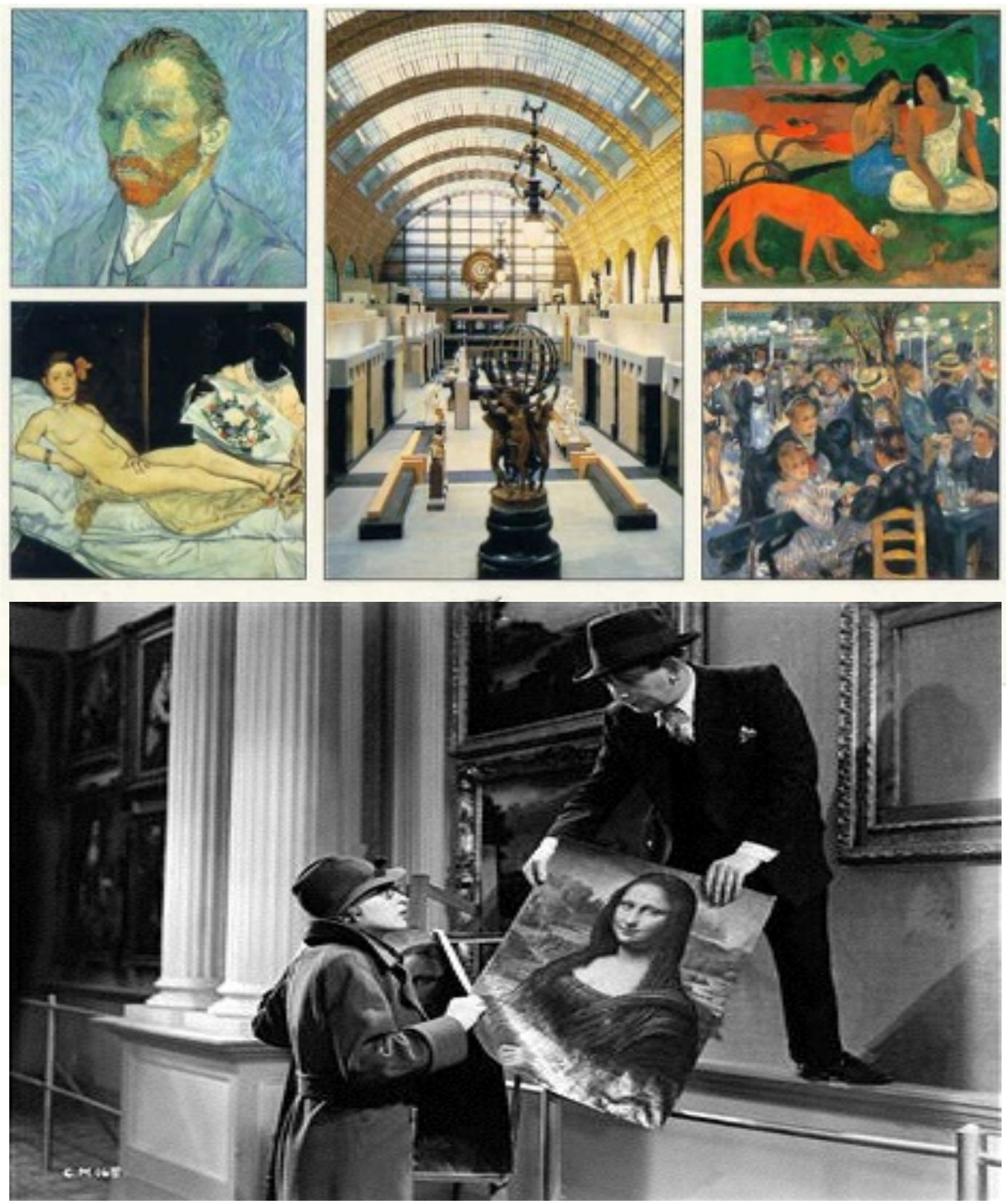
- Device and environment are un-trusted
- Attacker has direct access to the machine and software no matter whether it's running or not

Bob is the Attacker

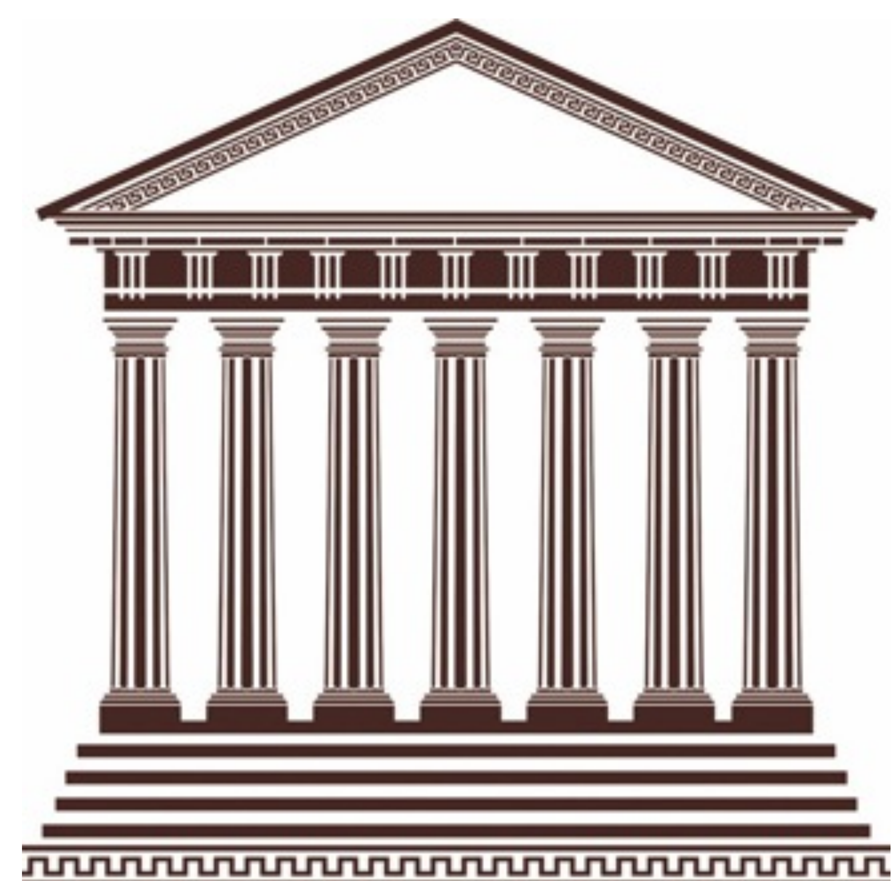


Software

Just Like in Museums



The state of the art



Obfuscation

Program Source Code → **Obfuscation** → **Obfuscated Program**

"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****)"*

A nondeterministic algorithm O is a TM obfuscator if three following conditions hold:

- **(functionality)** For every TM M, the string $O(M)$ describes the same function as M.
- **(polynomial slowdown)** The description length and running time of $O(M)$ are at most polynomially larger than that of M.
- **("virtual black box" property)** For any PPT A, there is a PPT S and a negligible function α such that for all TMs M

Attack

$$\left| Pr[A(O(M)) = 1] - Pr[S^M(1^{|M|}) = 1] \right| \leq \alpha(|M|).$$

Simulator

Impossibility

Program Source Code → **Obfuscation** → **Obfuscated Program**

"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****)"*

A nondeterministic algorithm O is a TM obfuscator if three following conditions hold:

- ⇨ **(functionality)** For every TM M, the string $O(M)$ describes the same function as M.
- ⇨ **(polynomial slowdown)** The description length and running time of $O(M)$ are at most polynomially larger than that of M.
- ⇨ **("virtual black box" property)** For any PPT A, there is a PPT S and a negligible function α such that for all TMs M:

Attack

$$\left| Pr[A(O(M)) = 1] - Pr[S^M(1^{|M|}) = 1] \right| \leq \alpha(|M|).$$

Simulator

Barak's et al. JACM 2012

Obfuscation

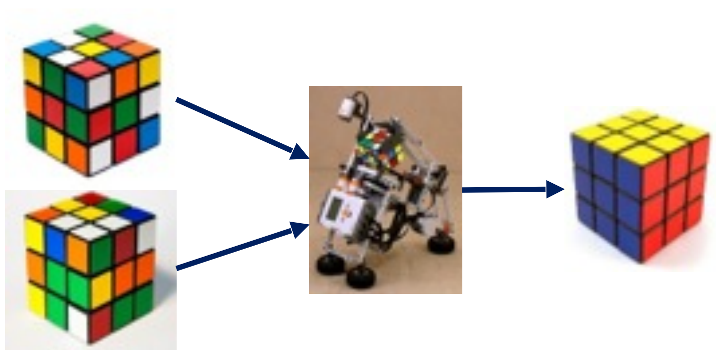
Program Source Code → **Obfuscation** → **Obfuscated Program**

PROGRAM OBFUSCATION

Security through obscurity

2013

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



Hard

1. **Program obfuscation re-cast as a rigorous mathematical science**
2. **The adversary can have full knowledge of the obfuscating theory but still cannot de-obfuscate**

Garg, Gentry, Halevy, Raykova, Sahai, Waters, 2013 "Candidate **Indistinguishability Obfuscation** and Functional Encryption for All Circuits".

What exactly do we mean when we say that we have obfuscated a program?

The value

Secured Input
Authentication, validation, integrity, confidentiality of input data

Hide Algorithms & Computations

Tamper Detection

Tamper Resistance
Makes it hard to modify the code's data and control flow

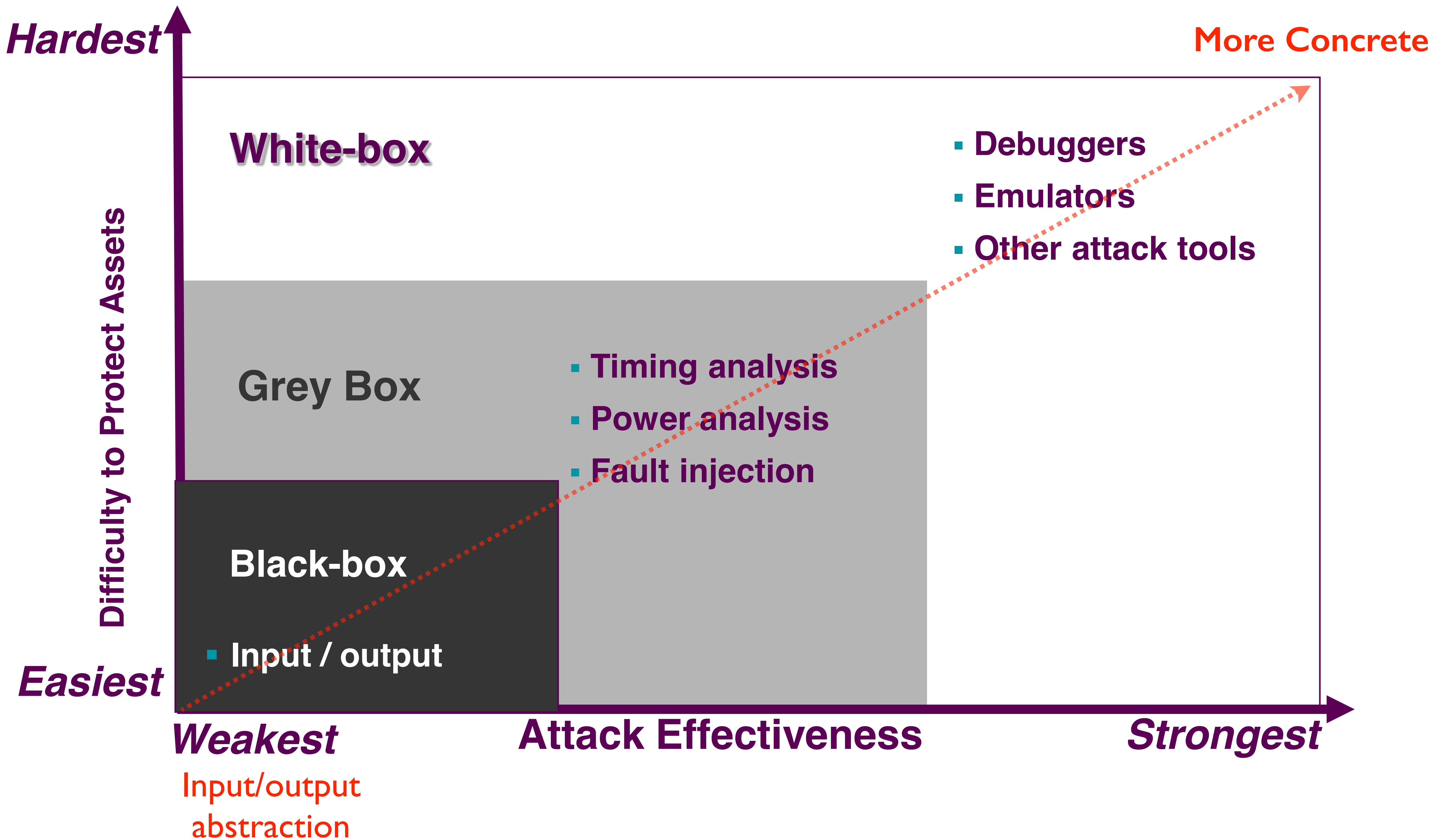
Secured Output
Authentication, validation, integrity, confidentiality of output data

Hide Internal Data
Including internally initialized data

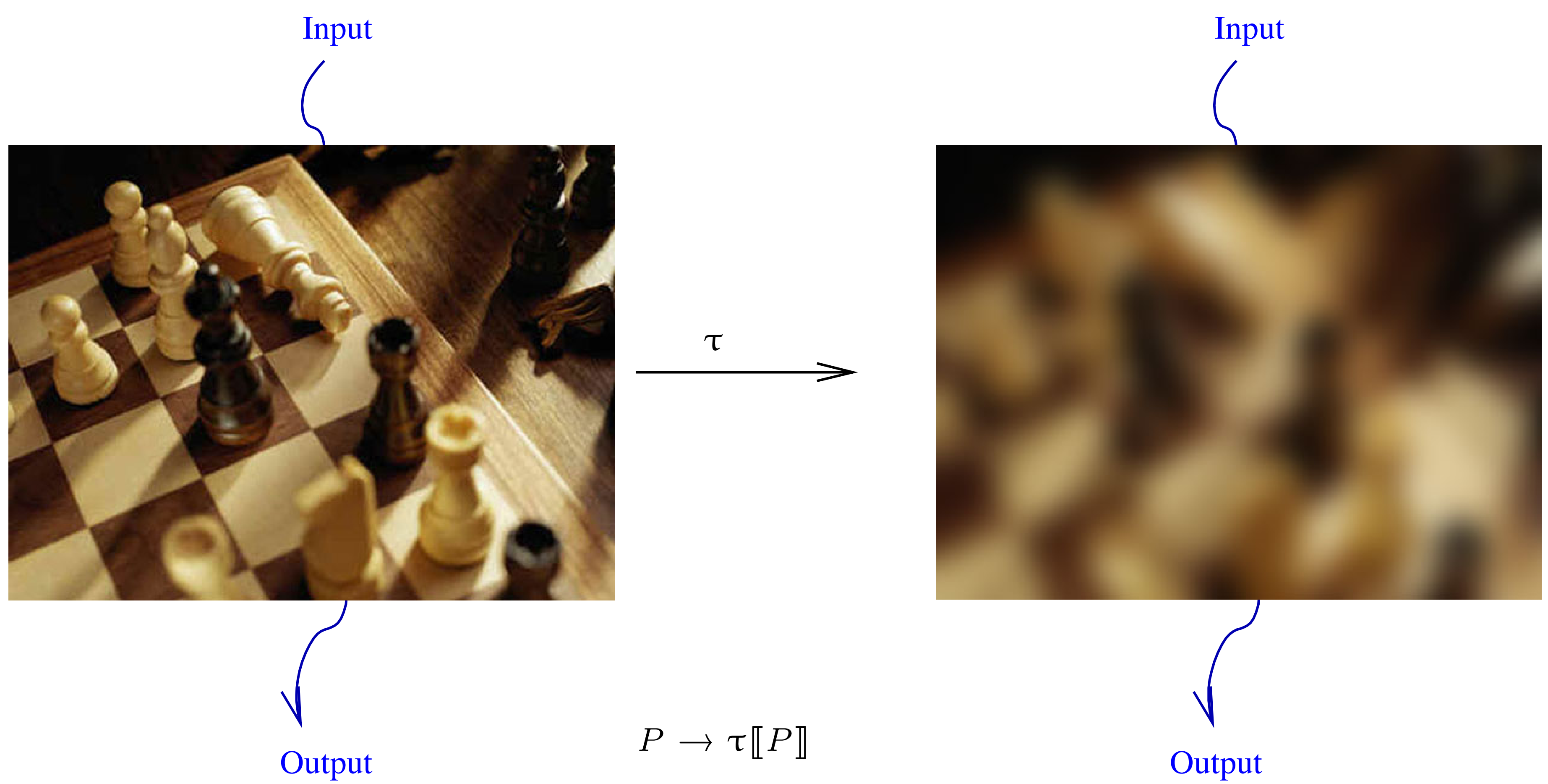
Anti Bug

Damage Mitigation

A different view from PL…

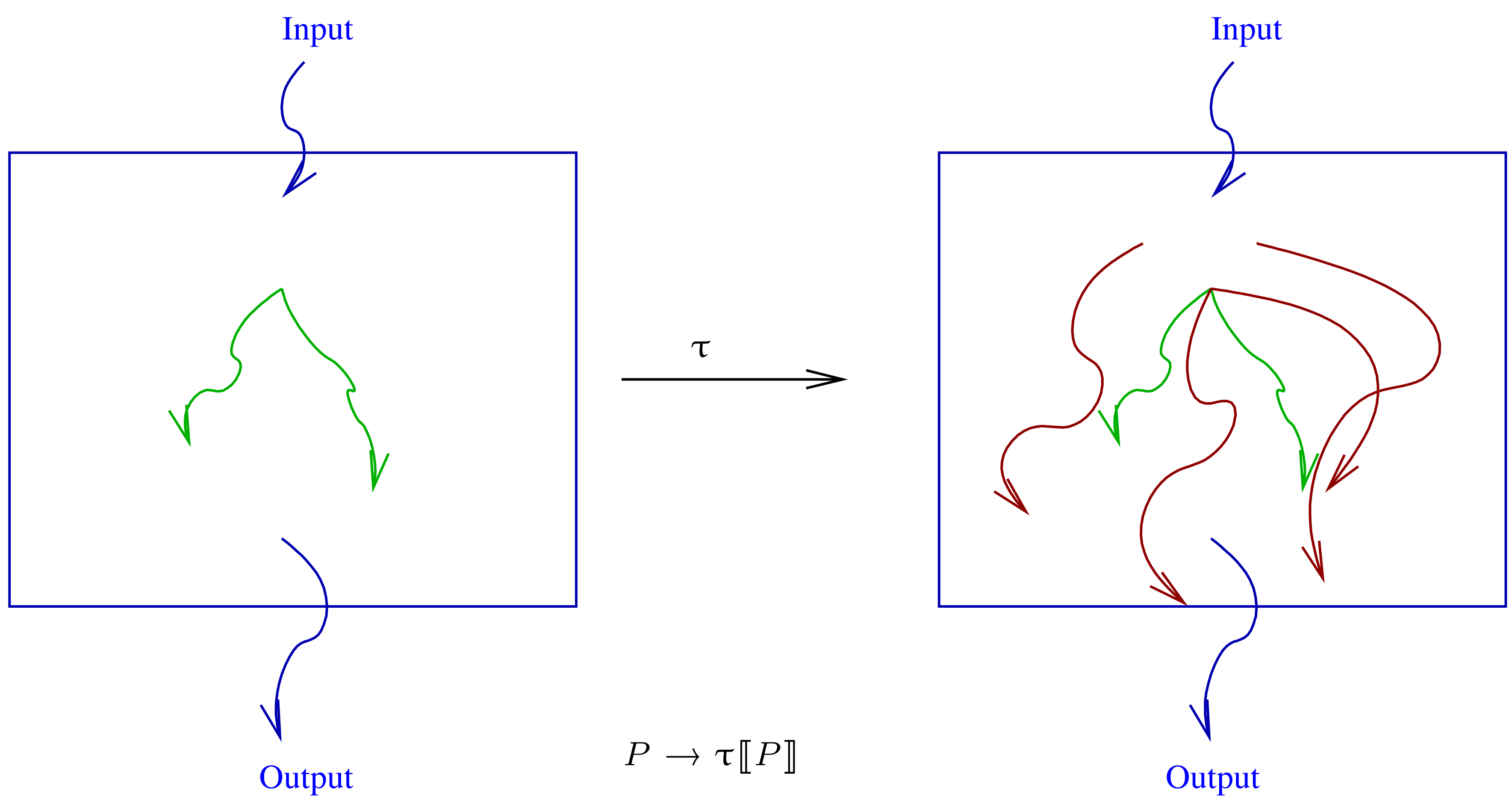

Hardest
More Concrete
White-box
Debuggers
Emulators
Other attack tools
Difficulty to Protect Assets
Grey Box
Timing analysis
Power analysis
Fault injection
Black-box
Input / output
Easiest
Weakest
Attack Effectiveness
Strongest
Input/output
abstraction

Obfuscation



$$P \rightarrow \tau[\![P]\!]$$

Obfuscation as compilation



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

(Pseudo-)Code:

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

$\xrightarrow{\tau}$

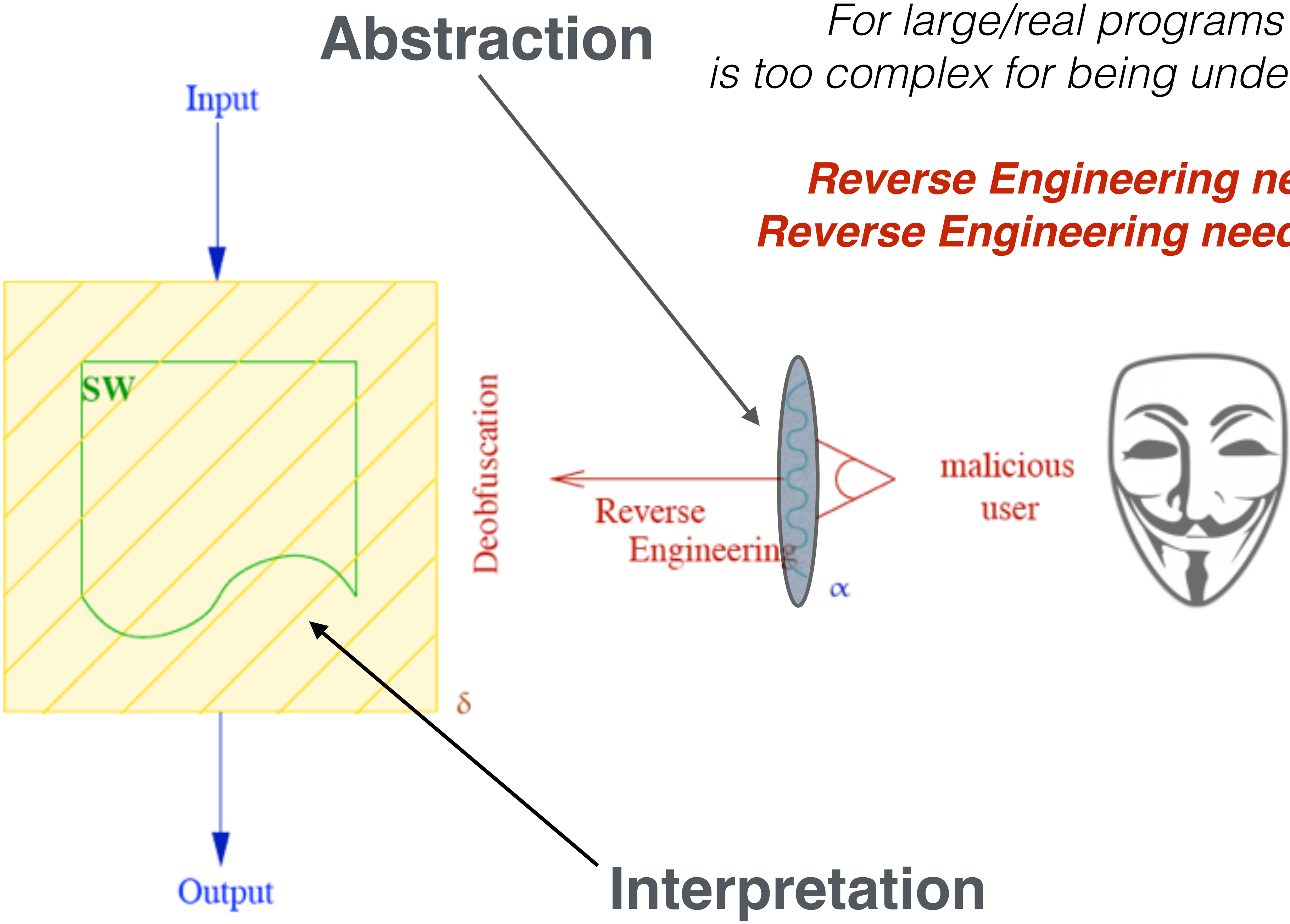
Obfuscated code (junk + reordering):

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

Computing means Interpreting

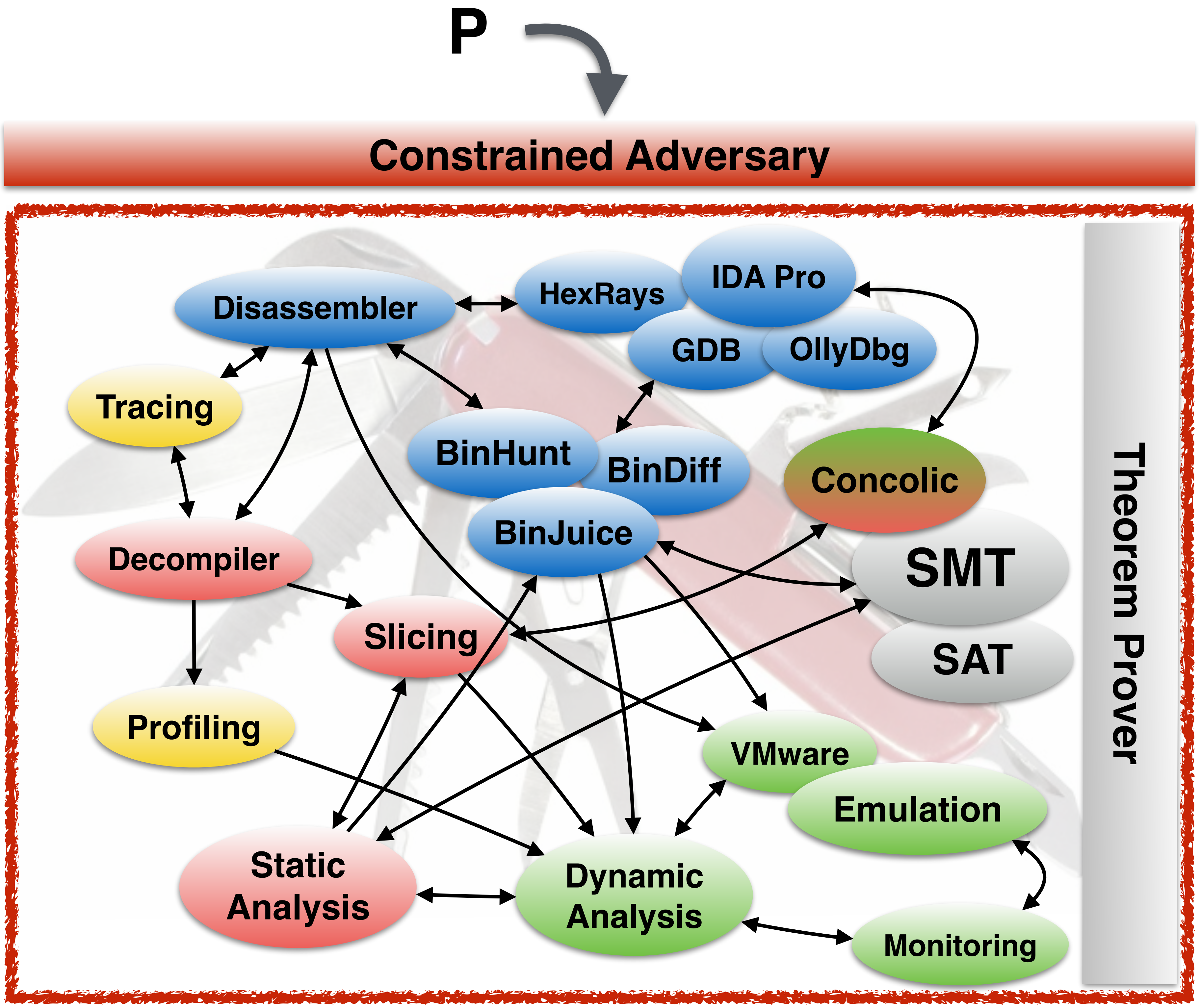
For large/real programs control/data flow
is too complex for being understandable by humans:

Reverse Engineering needs abstraction!
Reverse Engineering needs automated tools!

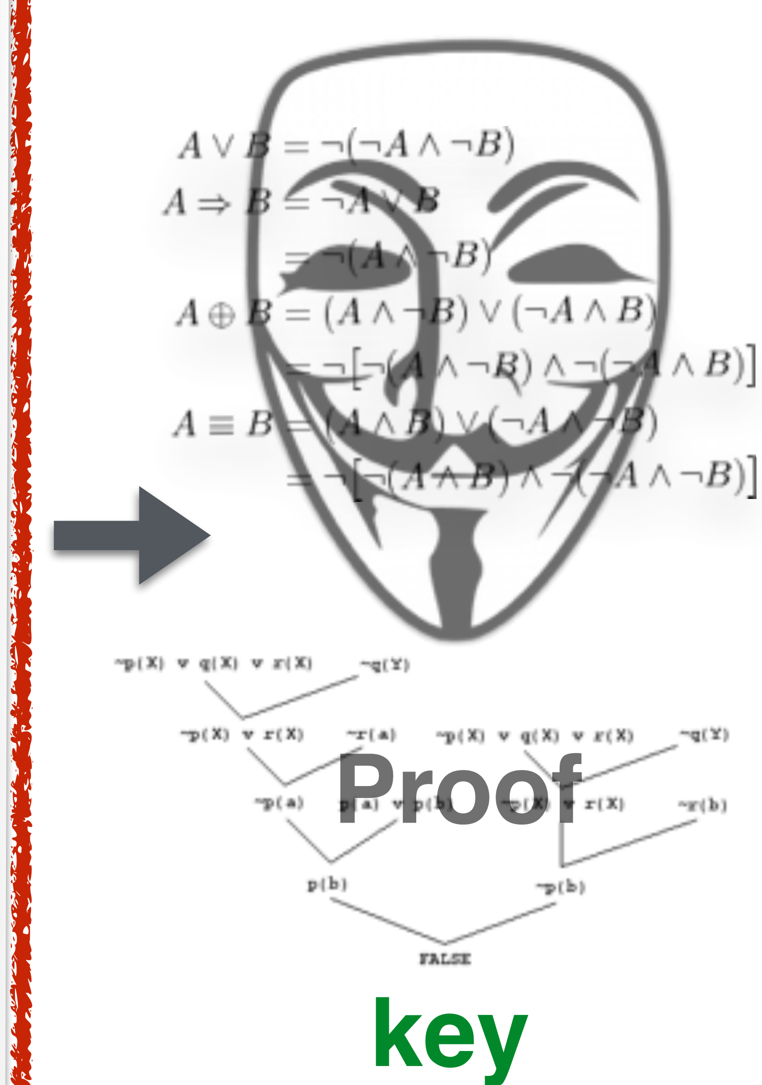


Abstract Interpretation is a general theory for approximating the semantics of dynamic systems (Cousot & Cousot 1977)

Reverse Engineering is Interpreting



- Each tool is an Abstract Interpretation



We can quantify the security achieved by looking at proof complexity!

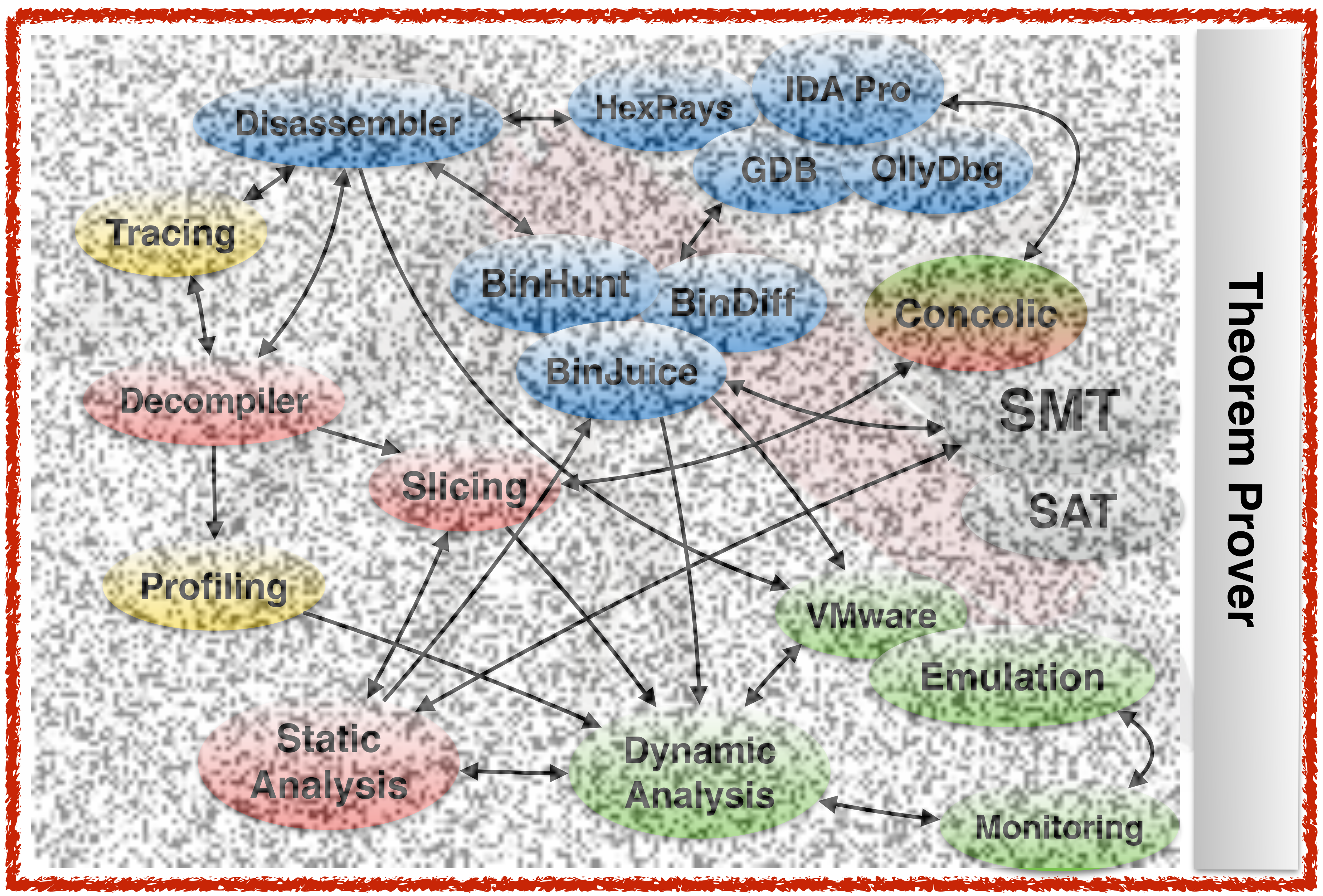
DAPA
Digital Asset Protection Assn.

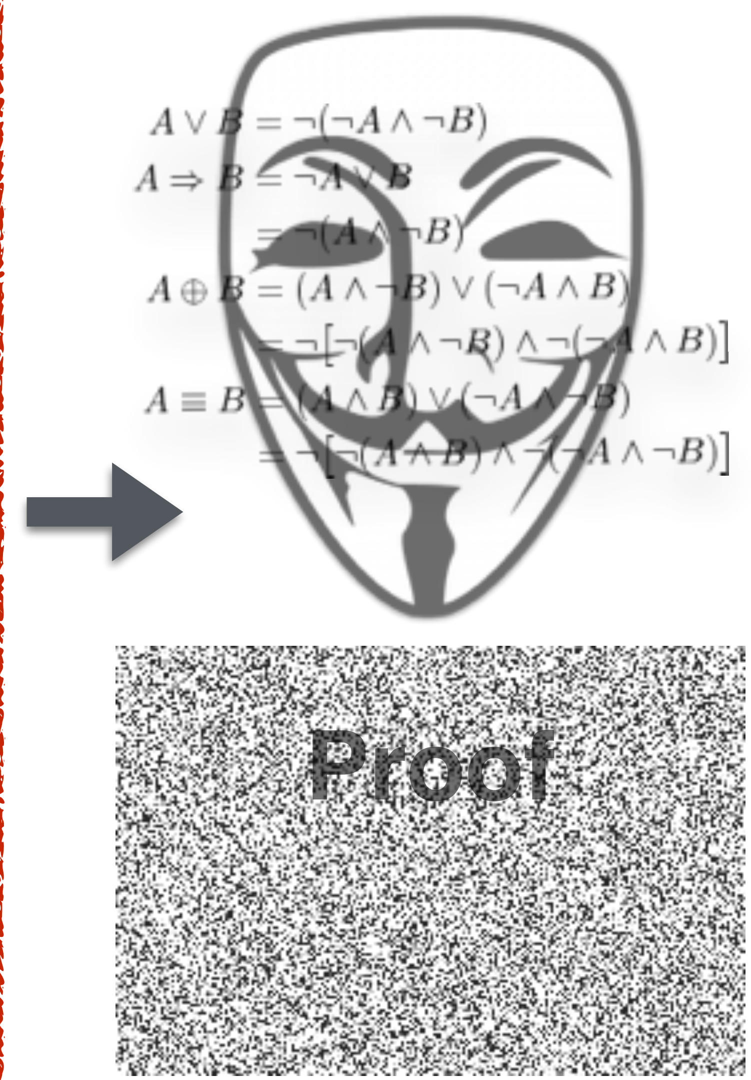
Protecting is obscuring Interpretation

$\mathcal{O}(P)$

- Transform code to make *all* tools blind

Constrained Adversary

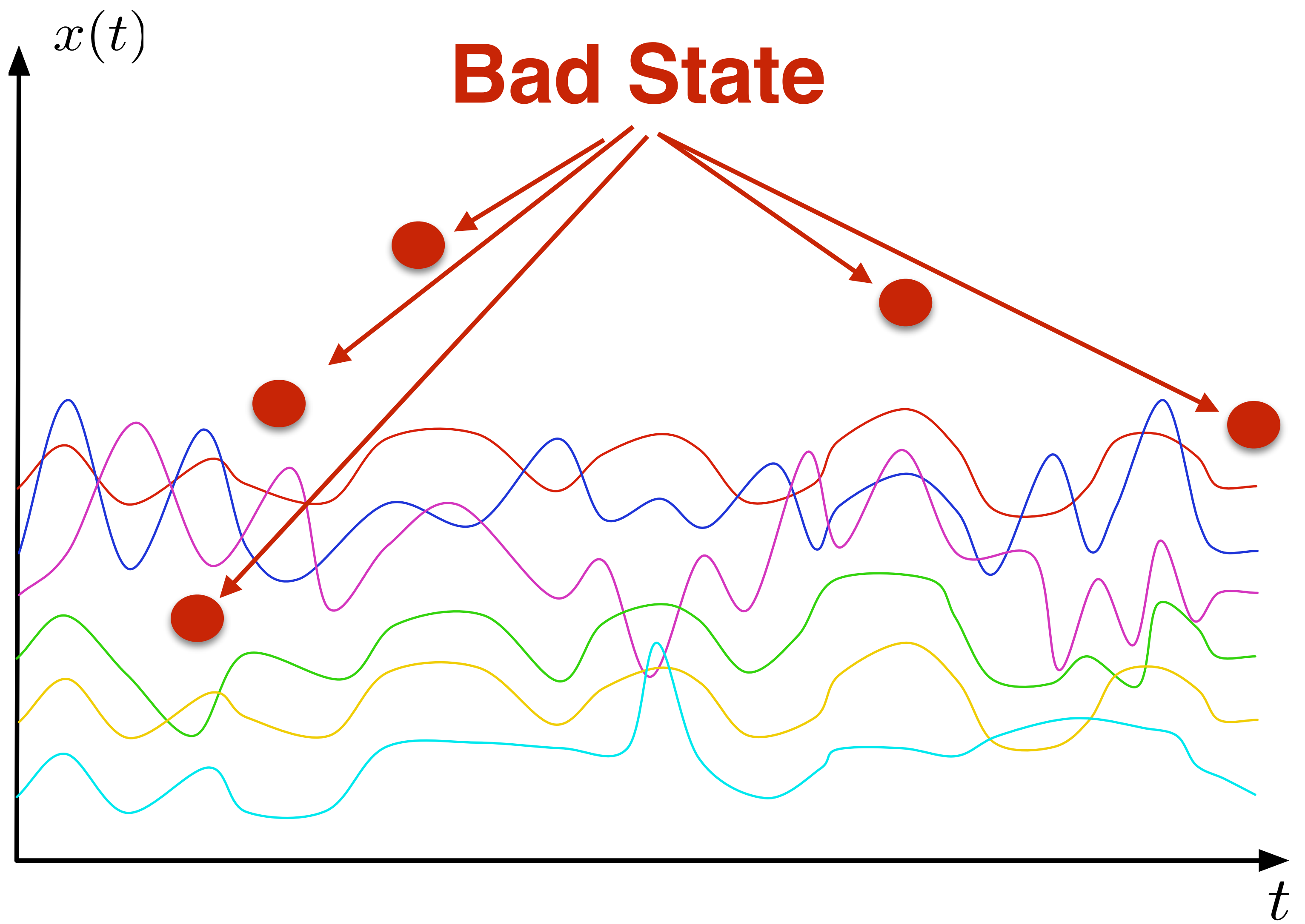




Removing noise means refining abstractions / complicating proofs! (Giacobazzi et al 2000 / 2012)

...the ingredients?

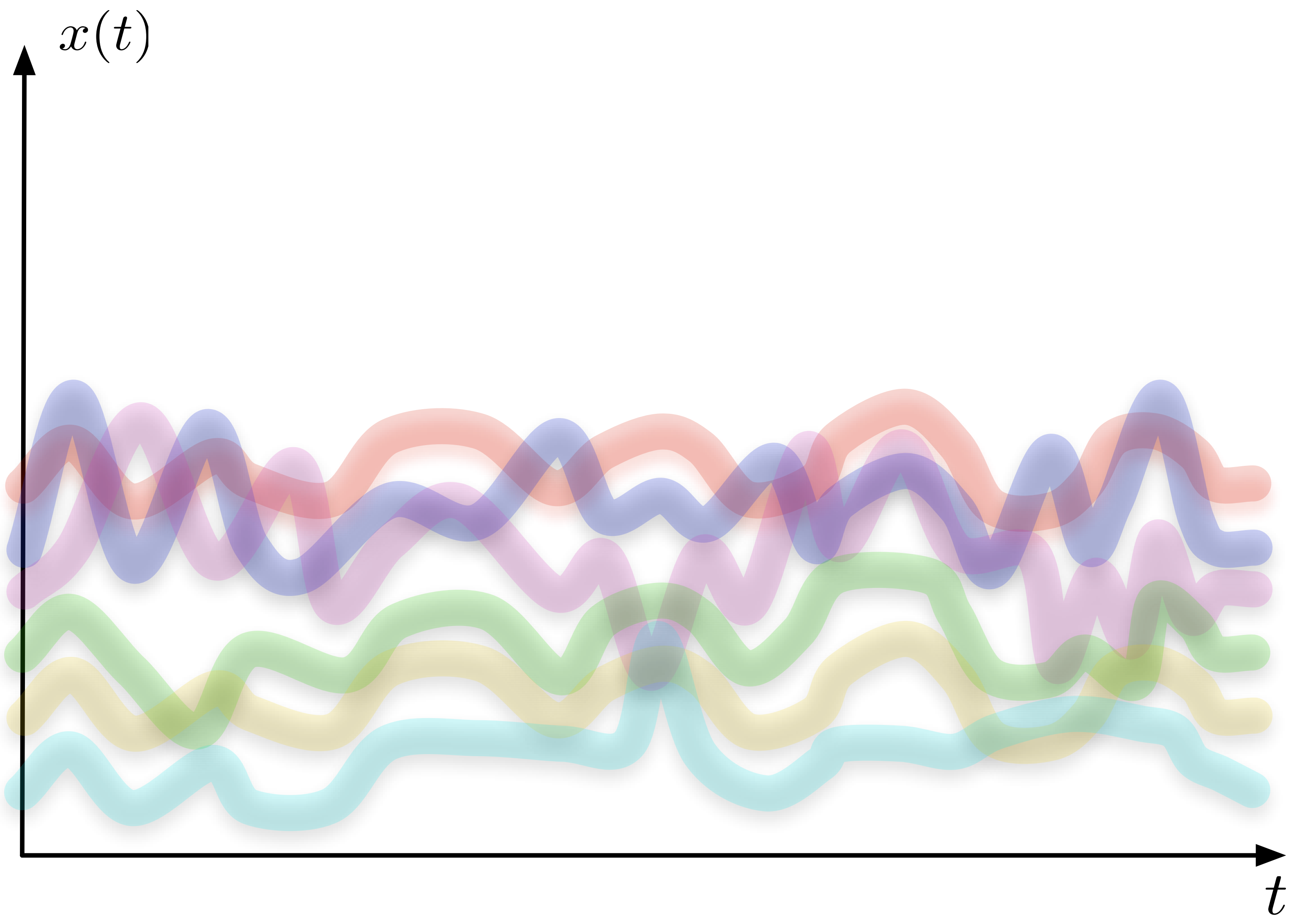
A Model



No bug!

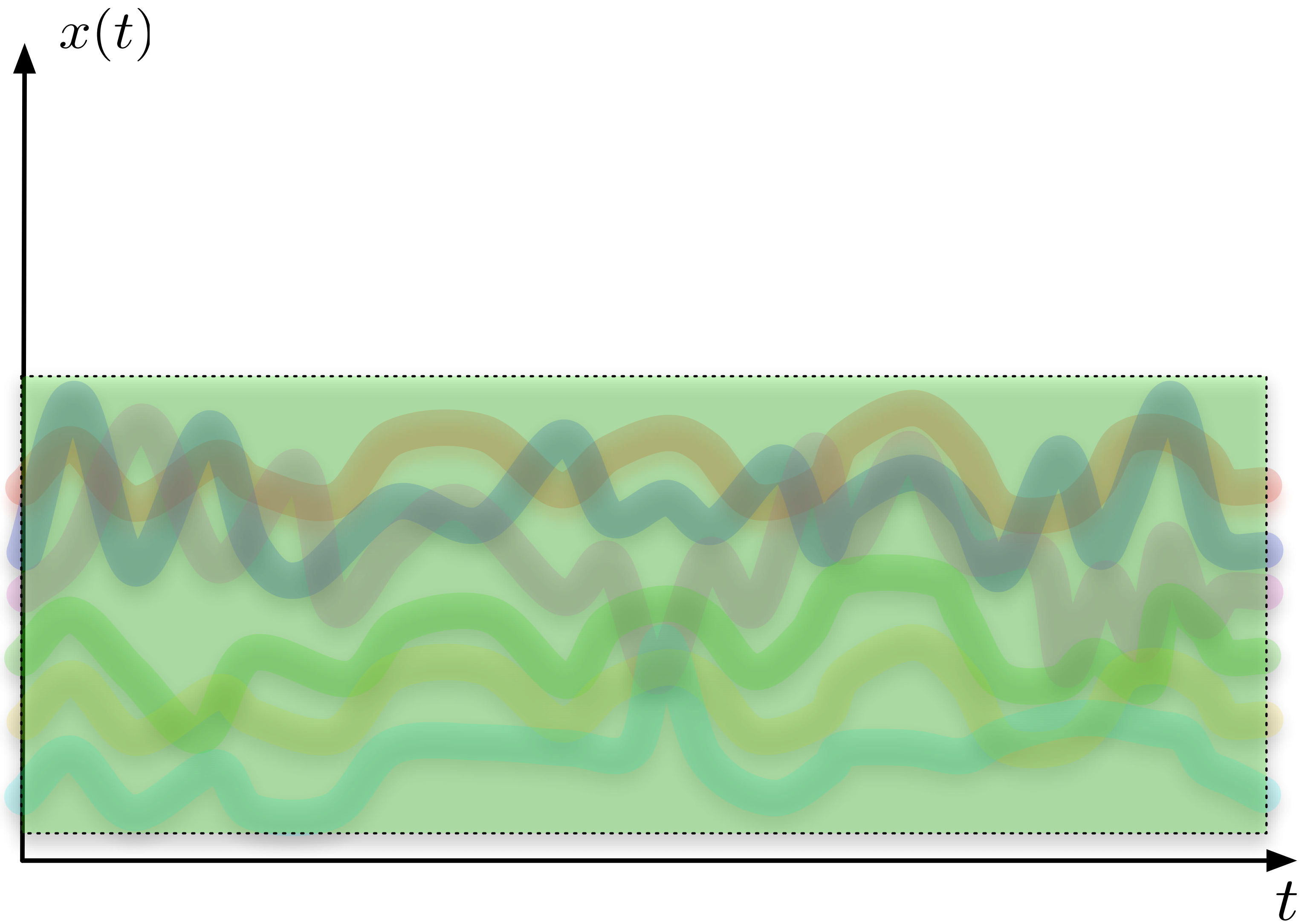
Too complicated, complex, undecidable

Abstraction



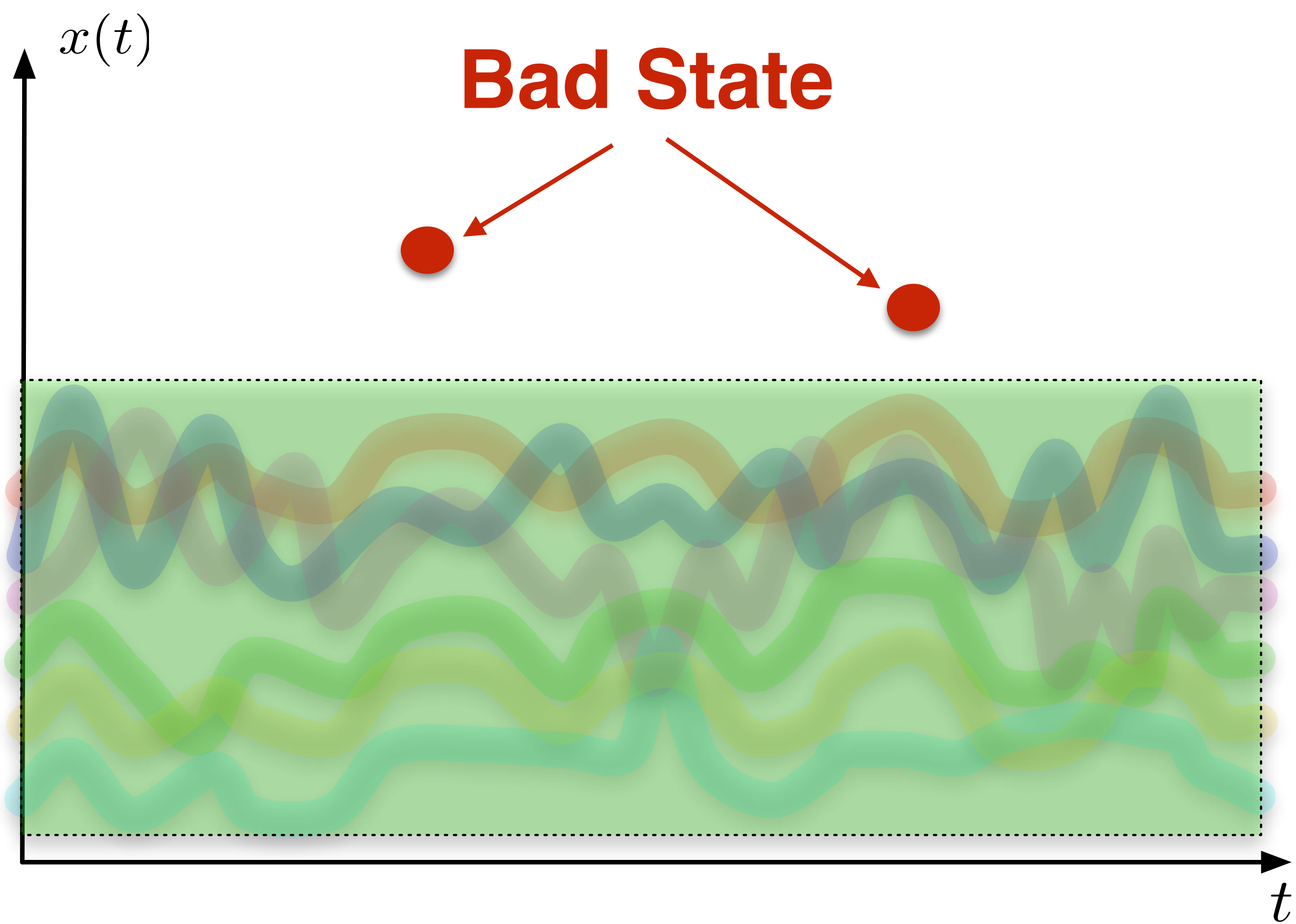
To understand code we need abstraction: *simpler and computable*

Abstraction



Abstraction ➟ (sound) loss of precision

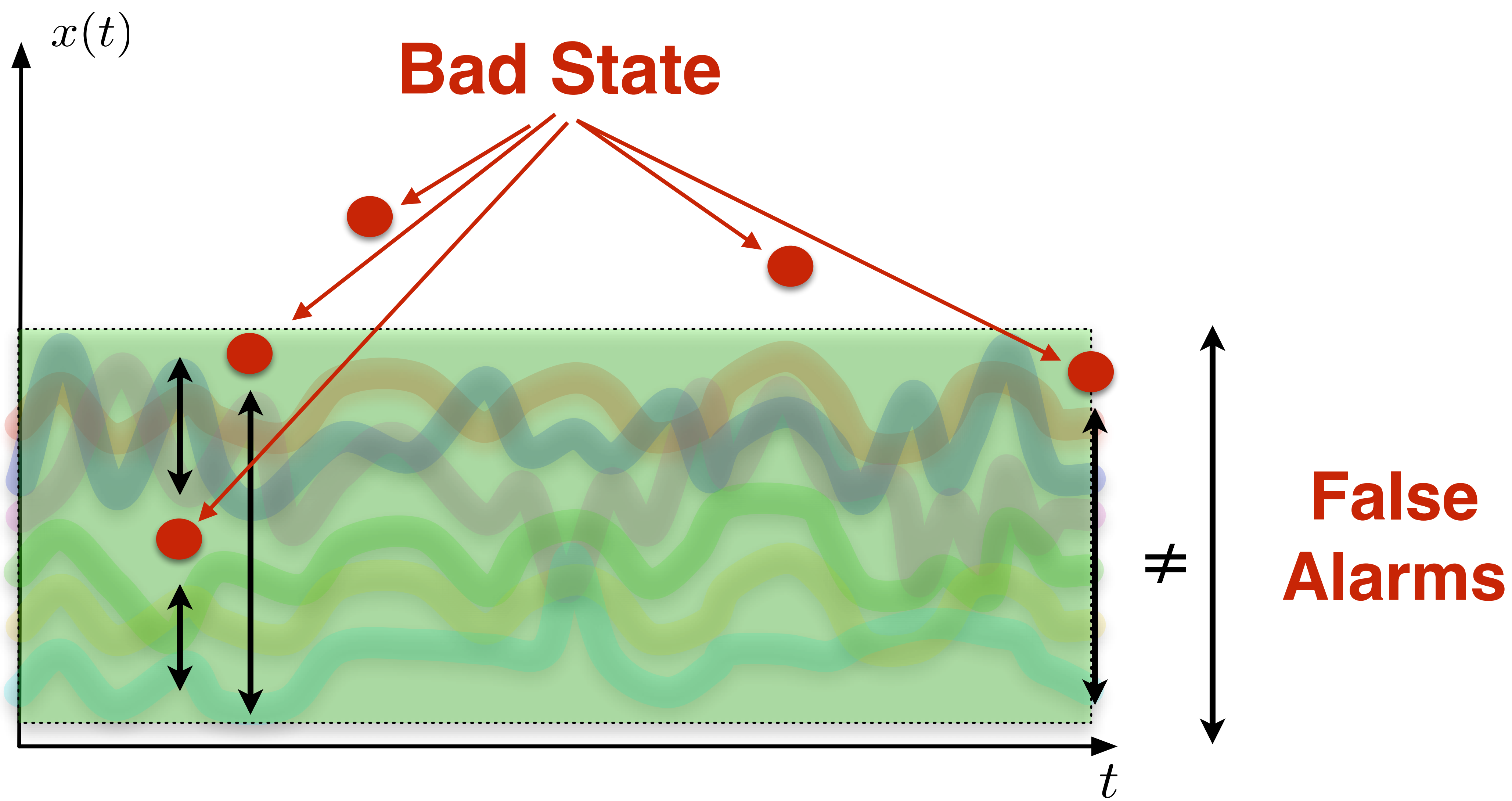
Abstraction



No bug!

Computable
Abstraction ➠ (sound) loss of precision

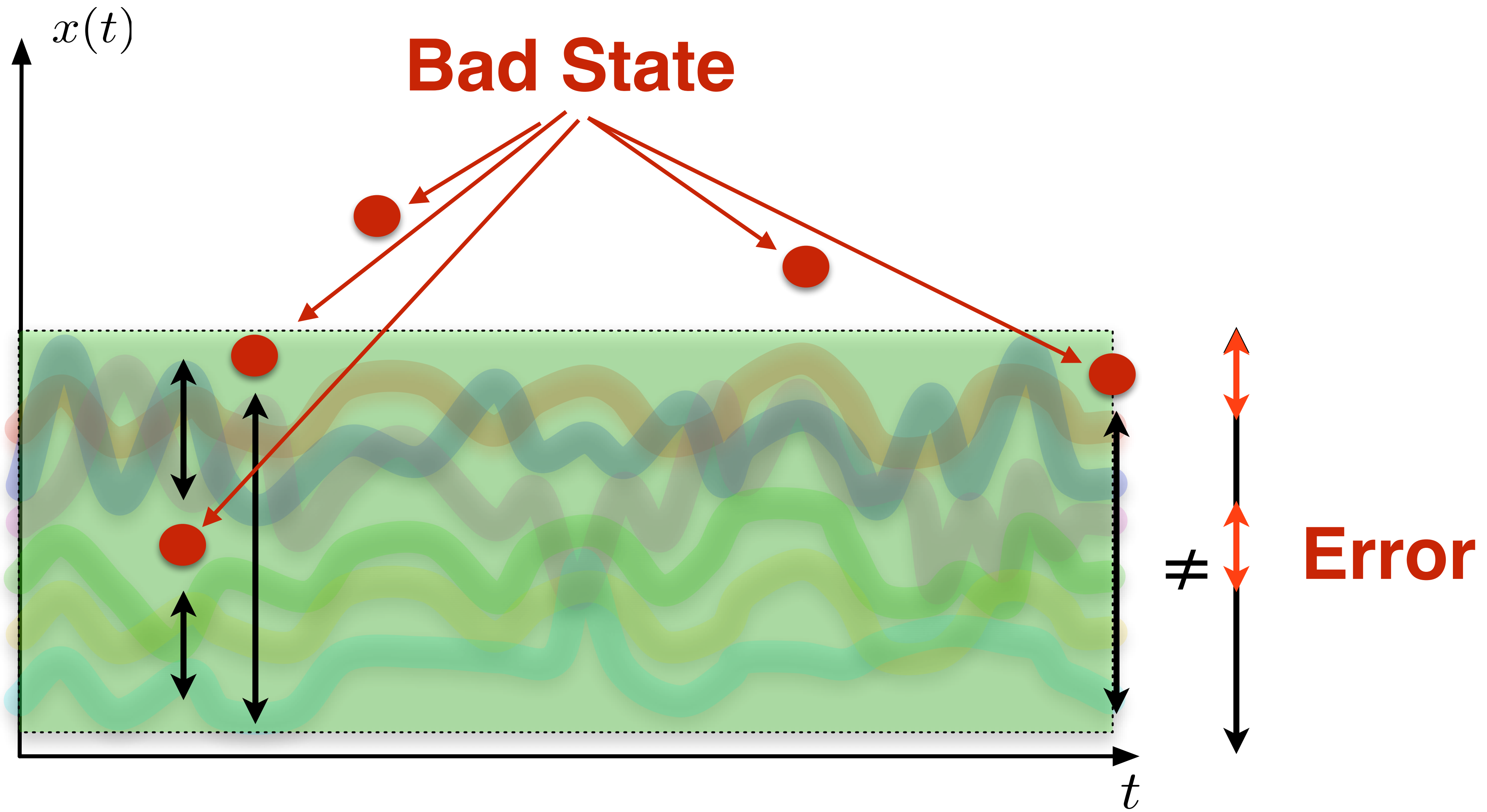
Completeness



Abstraction ➟ (sound) loss of precision

Incompleteness

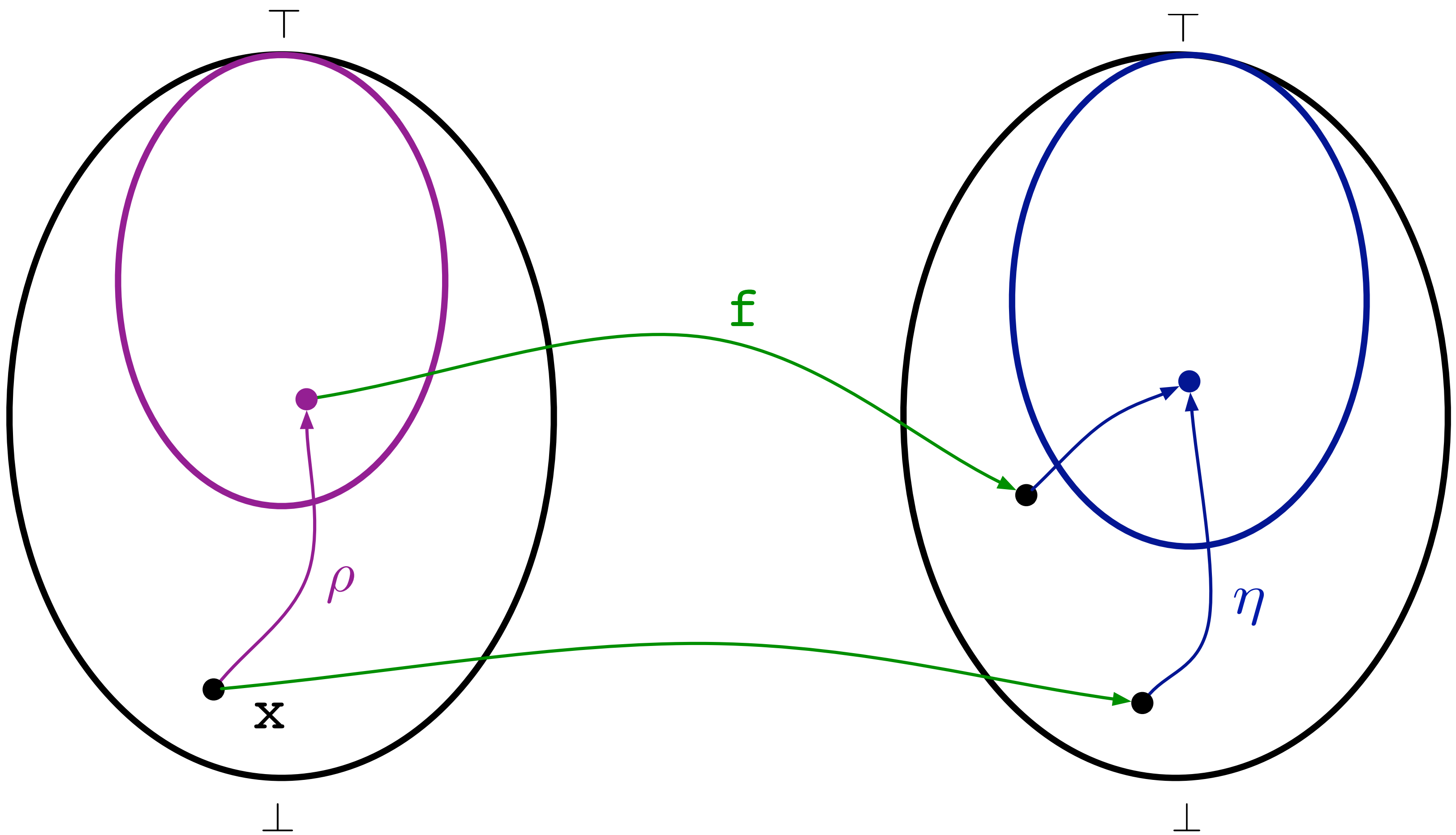
Completeness



Abstraction ➟ False alarms

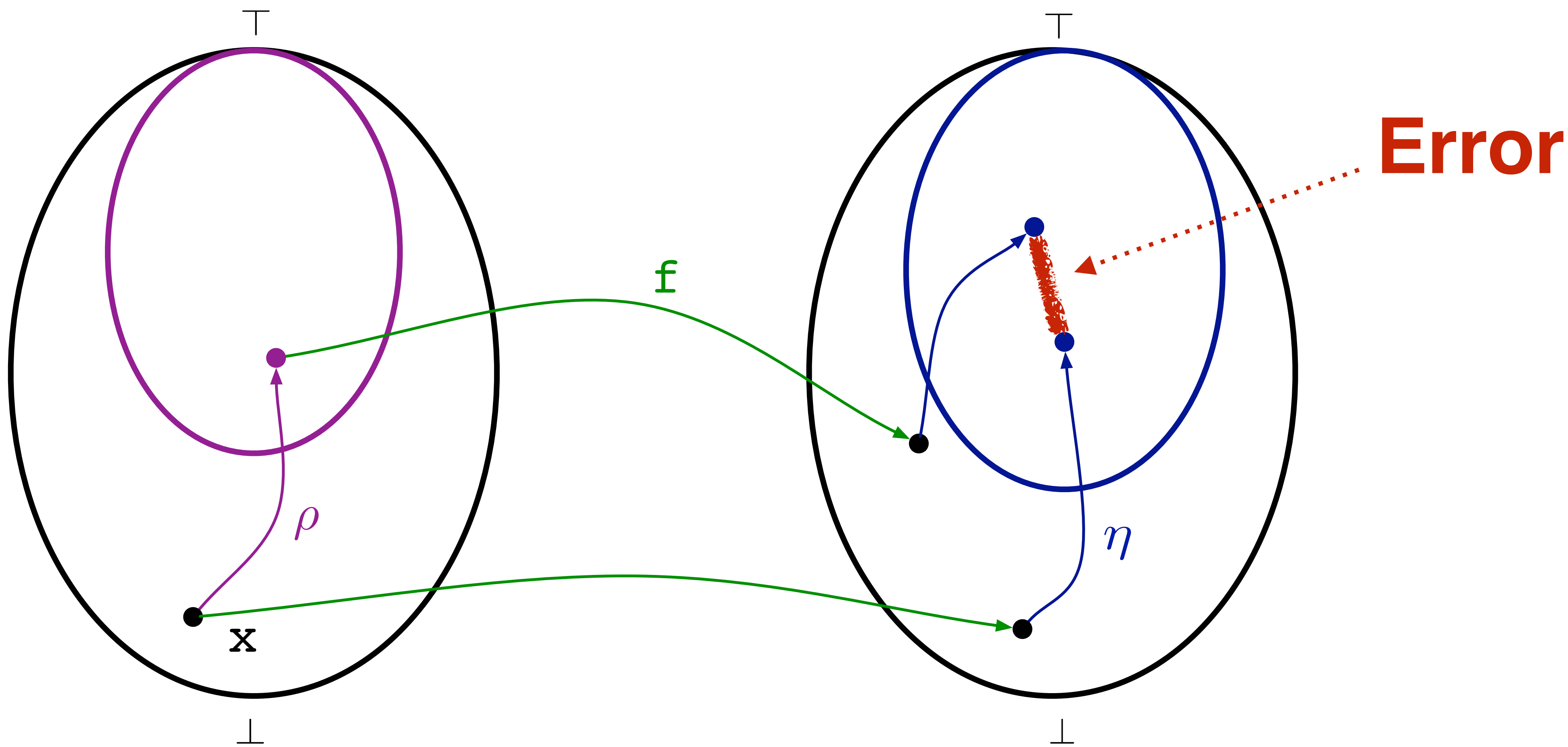
Incompleteness

Completeness



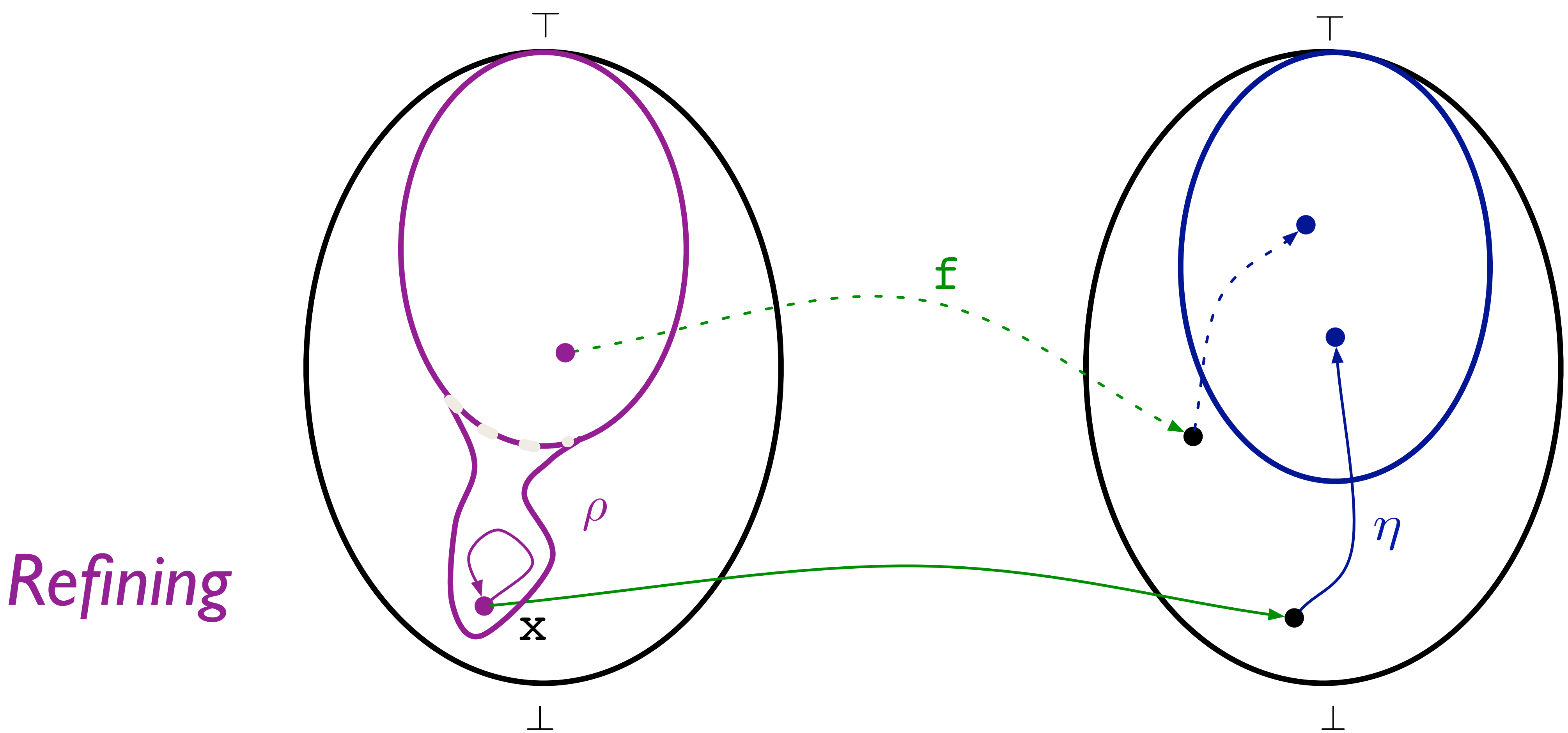
COMPLETENESS: $\eta \circ f \circ \rho = \eta \circ f$

(in)Completeness



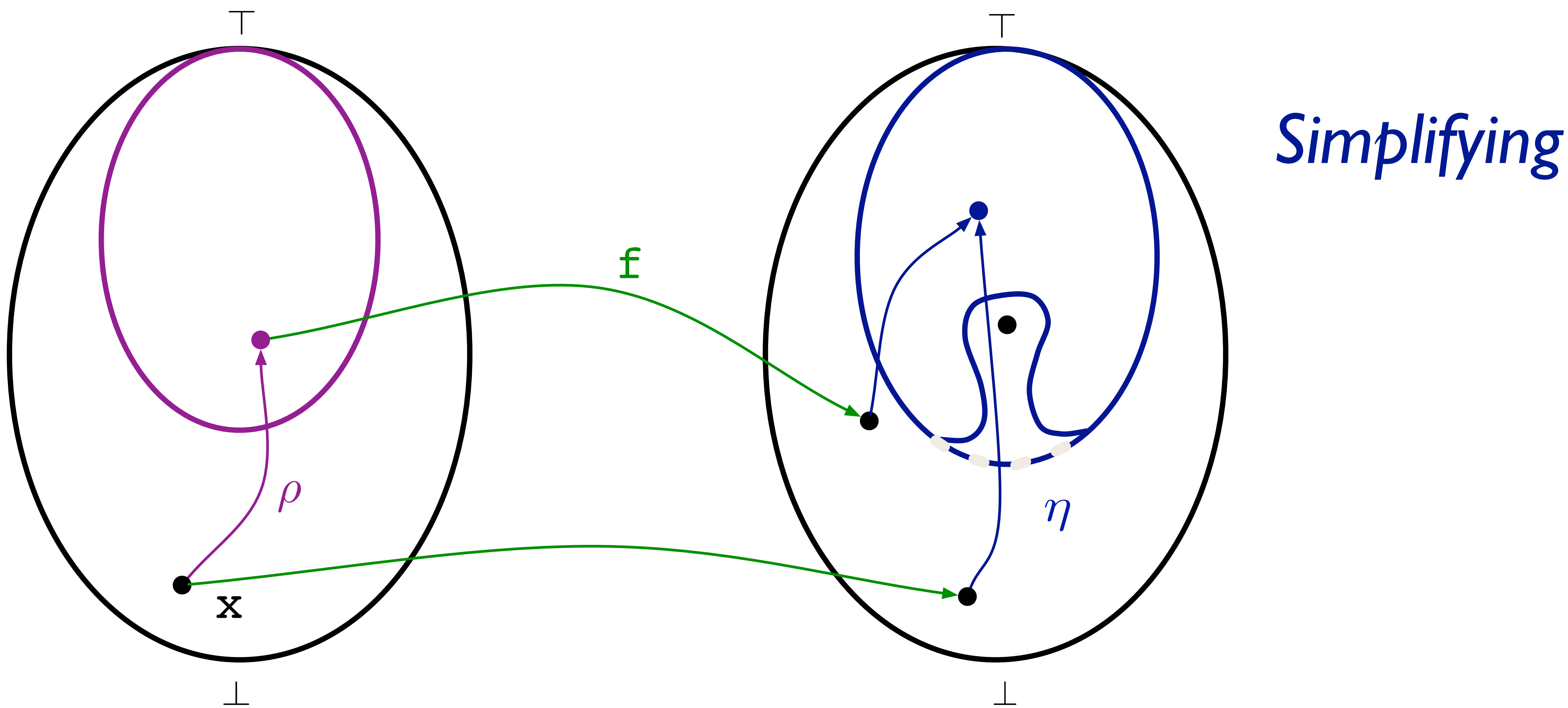
IN-COMPLETENESS: $\eta \circ f \circ \rho \geq \eta \circ f$

Making Completeness



Making ABSTRACTIONS COMPLETE: Refining input domains
[*Giacobazzi et al. JACM'00*]

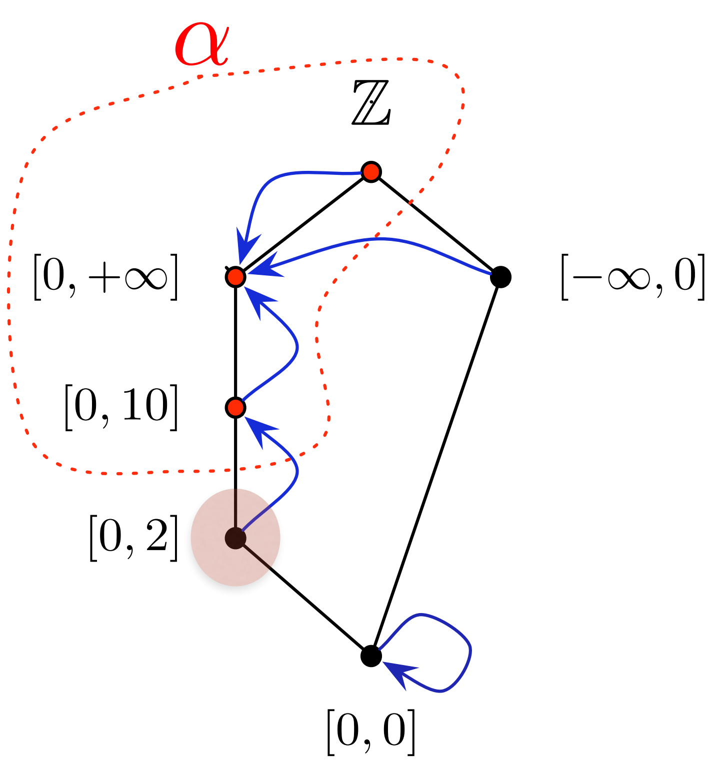
Making Completeness



Making ABSTRACTIONS COMPLETE: Simplifying output domains
[*Giacobazzi et al. JACM'00*]

Making Completeness

A simple example in Interval analysis



- A simple domain of intervals
- $sq(X) = \left\{ x^2 \;\middle|\; x \in X \right\}$
- $\{\mathbb{Z}, [0, +\infty], [0, 10]\}$ is not Backward complete

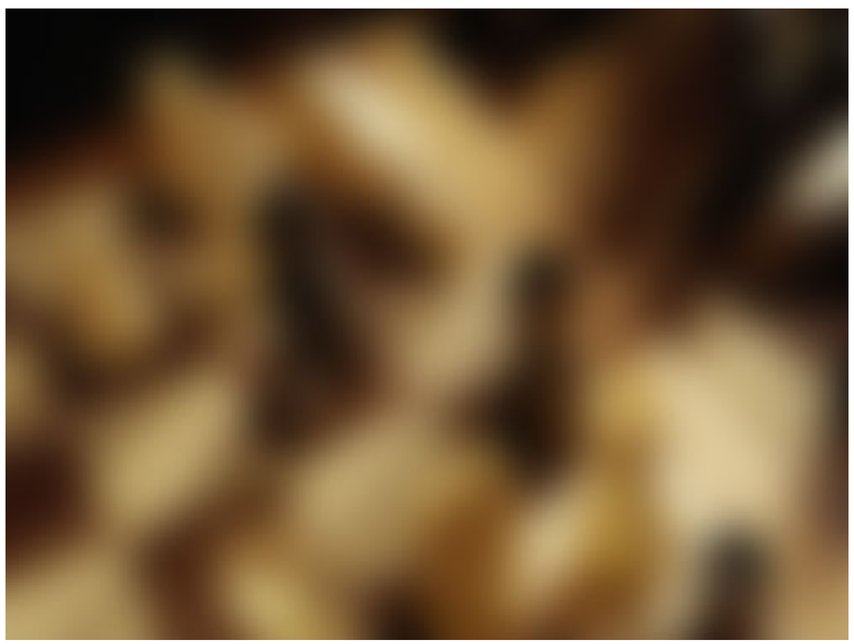
Same input & output abstraction = **fix-point refinement**

$$\mathcal{R}_f(\alpha) = \mathsf{gfp}(\lambda X.\, \alpha \sqcap R_f(X))$$

$$R_f \stackrel{\text{def}}{=} \lambda X.\, \mathcal{M}(\bigcup_{y \in X} \max(f^{-1}(\downarrow y)))$$

Transformation

A self interpreter `int` and a specializer `spec`



Defense ←

Attack →

$\texttt{target} := [\![\texttt{spec}]\!](\texttt{int}, \texttt{source})$

$\texttt{source}$

Design `int` and `spec` for obfuscating code

Challenge!

Transformation

A self interpreter `int` and a specializer `spec`

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

Defense ←

Attack →

$^{1\cdot}$**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**

$\texttt{target} := [\![\texttt{spec}]\!](\texttt{int}, \texttt{source})$

$\texttt{source}$

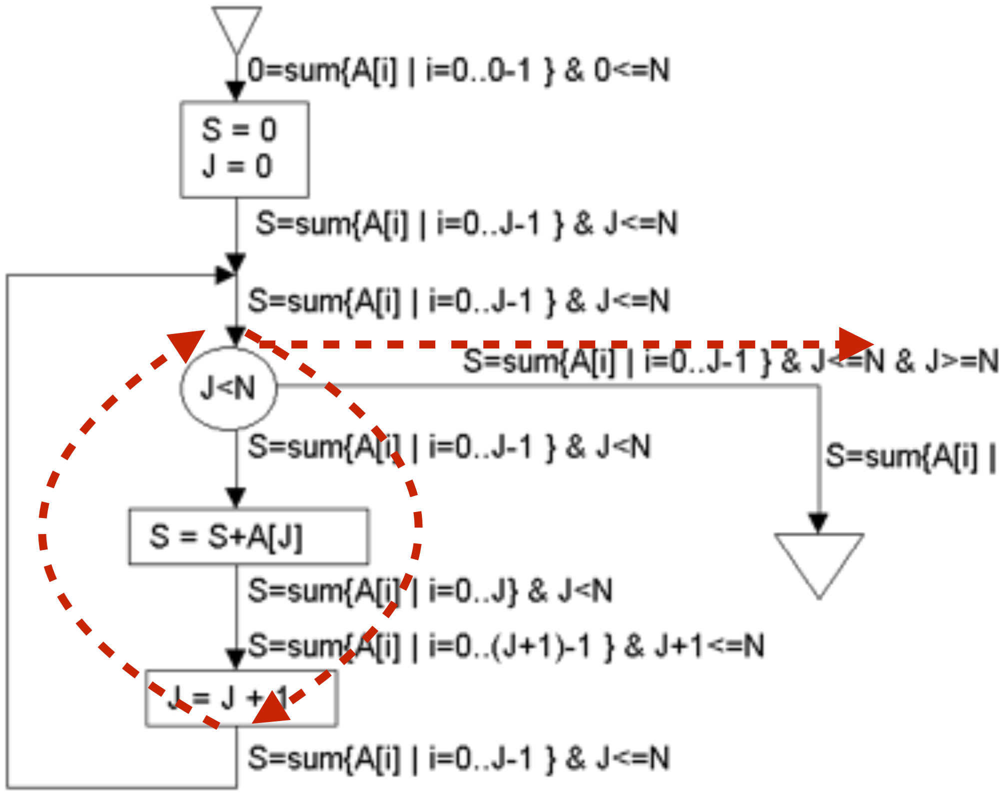
Design `obfuscation` from observation
(`transformation`) (abstraction)

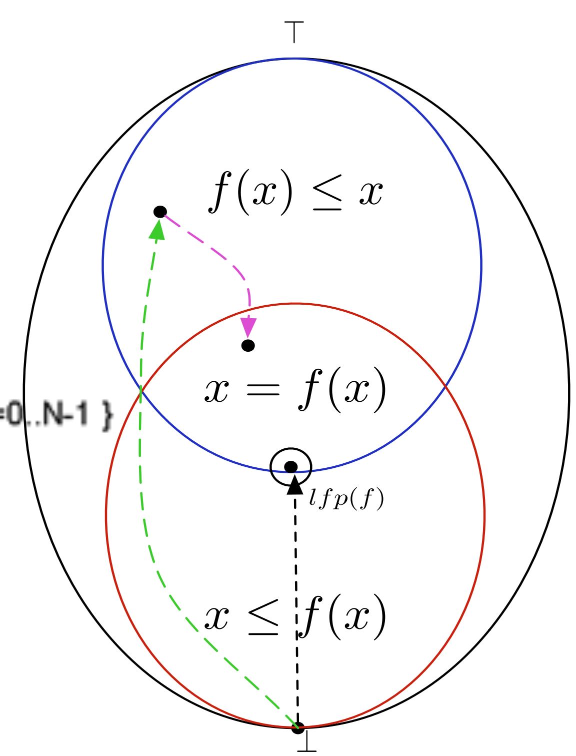
The Attacker

Why Abstract Interpretation?

Abstract Interpretation (1977) is the a general model for the (static or dynamic) approximation of semantics of discrete dynamic systems

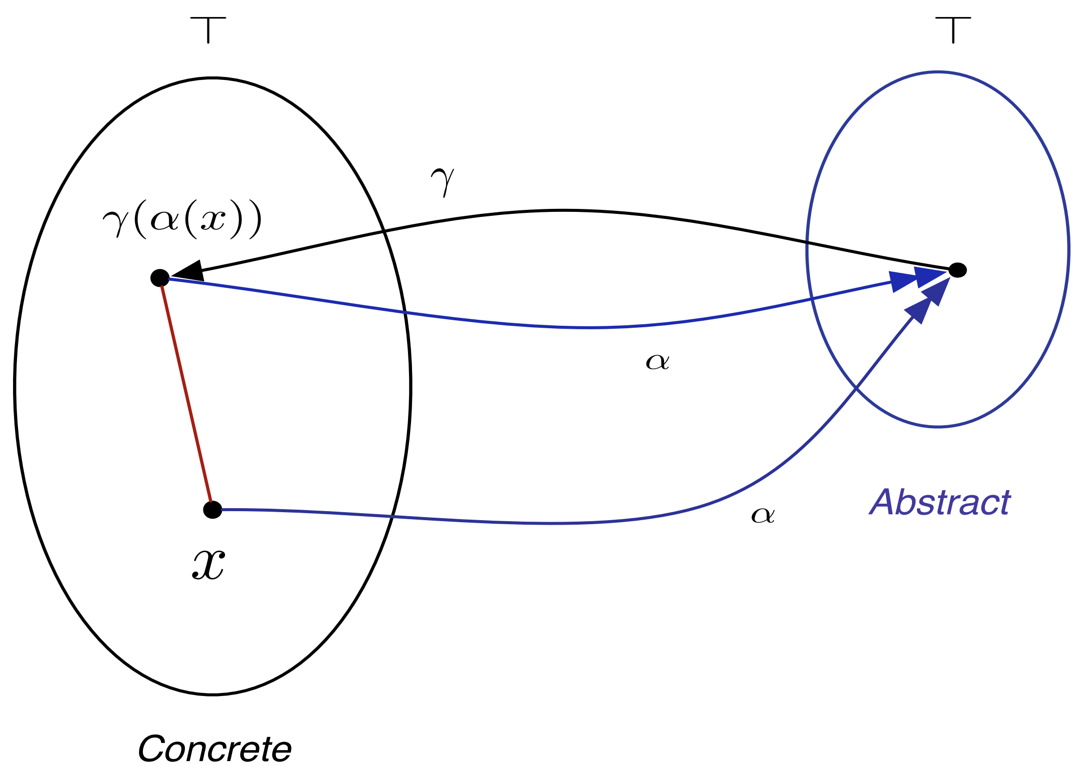
- Including: Static program analysis, dynamic analysis, profiling, debugging, tracing, compilation, de-compilation, type checking and type inference, model checking and predicate abstraction, trajectory evaluation, testing, proof systems, etc.





Abstract Interpretation

Design approximate semantics of programs [Cousot & Cousot '77, '79].

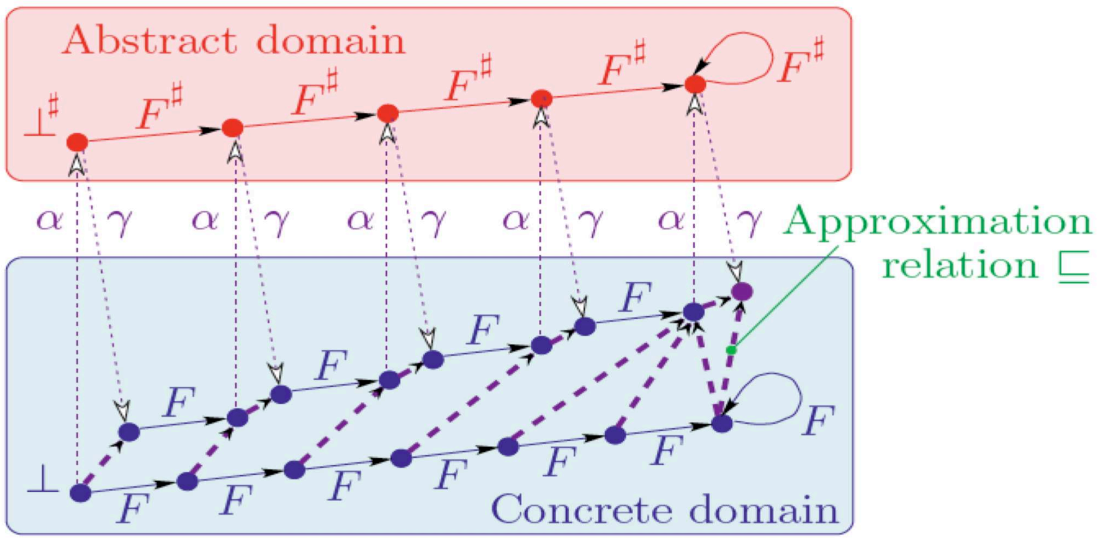


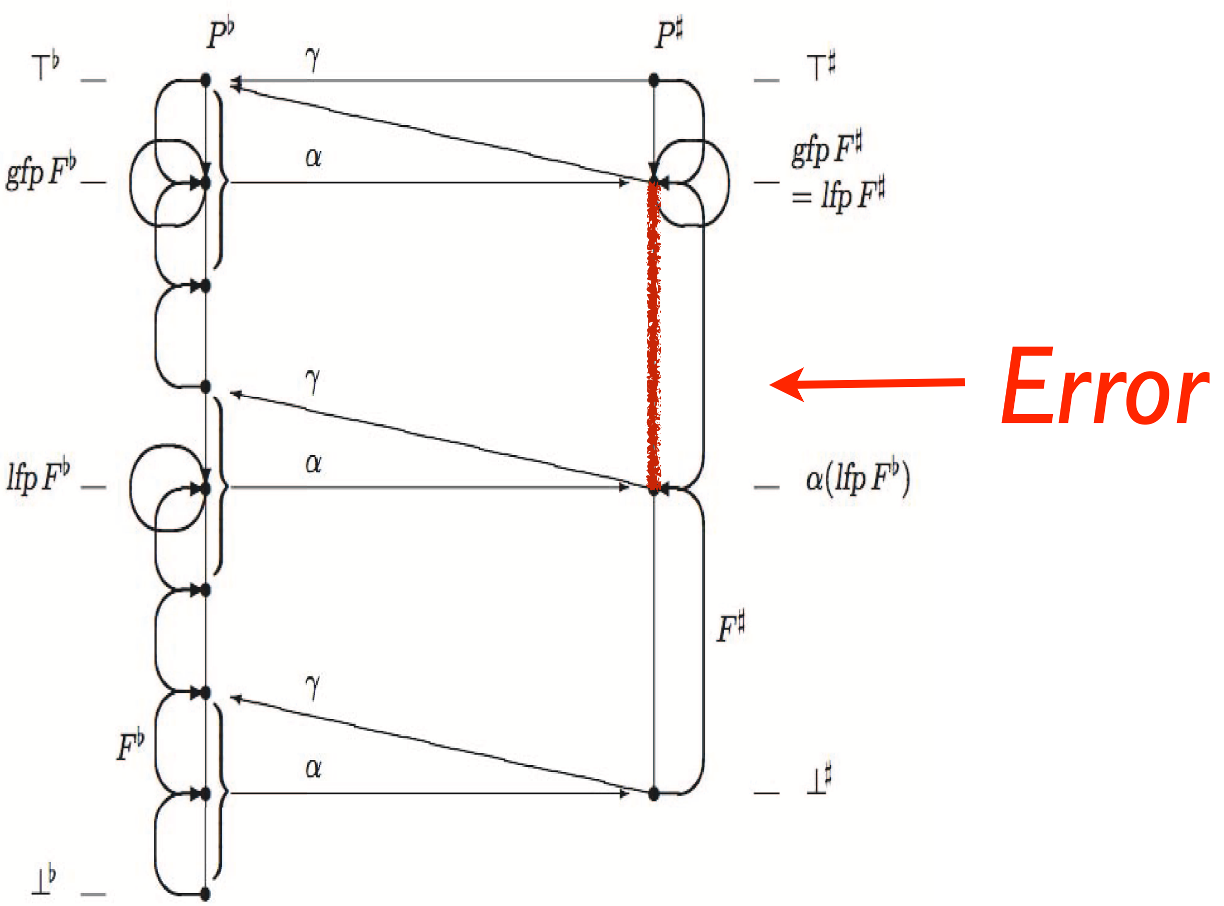
Galois Connection: $\langle C, \alpha, \gamma, A \rangle$, A and C are complete lattices.

Closures: $\langle uco(C), \sqsubseteq \rangle$ set of all possible abstract domains,

$A_1 \sqsubseteq A_2$ if A_1 is more concrete than A_2

Approximating interpretation





G is a sound approximation of F if

$$\alpha \circ F \circ \gamma \sqsubseteq G$$

Soundness and Completeness

- $\texttt{WhichChess} : Img \longrightarrow \wp(Chess)$ returns the type of chess on the chessboard.
- $\rho : Img \longrightarrow Img$ such that: $\rho\,\Big($  $\Big) =$ 
- $\eta : \wp(Chess) \longrightarrow [0, 12]$ counts an *upper bound* to the number of different types of chess

$\eta\Big(\texttt{WhichChess}\Big(\rho\Big($  $\Big)\Big)\Big) = \eta\Big(\texttt{WhichChess}\Big($  $\Big)\Big)$

$= 12$

$\geq \eta\Big(\texttt{WhichChess}\Big($  $\Big)\Big)$

$= 7$

Obscurity as Incompleteness

The **attack strategy** is a temporal formula to check against an abstraction

The **attacker** is an abstract interpreter

Failing precision means failing completeness

Obfuscating is making abstract interpreters and strategies incomplete!!

$$[\![P]\!] = [\![\tau(P)]\!] \qquad \rho([\![P]\!]) = [\![P]\!]^{\rho}$$

τ obfuscates P if $[\![P]\!]^{\rho} \sqsubset [\![\tau(P)]\!]^{\rho}$

$$[\![P]\!]^{\rho} \sqsubset [\![\tau(P)]\!]^{\rho} \iff \rho([\![\tau(P)]\!]) \sqsubset [\![\tau(P)]\!]^{\rho}$$

DAPA

Digital Asset Protection Assn.

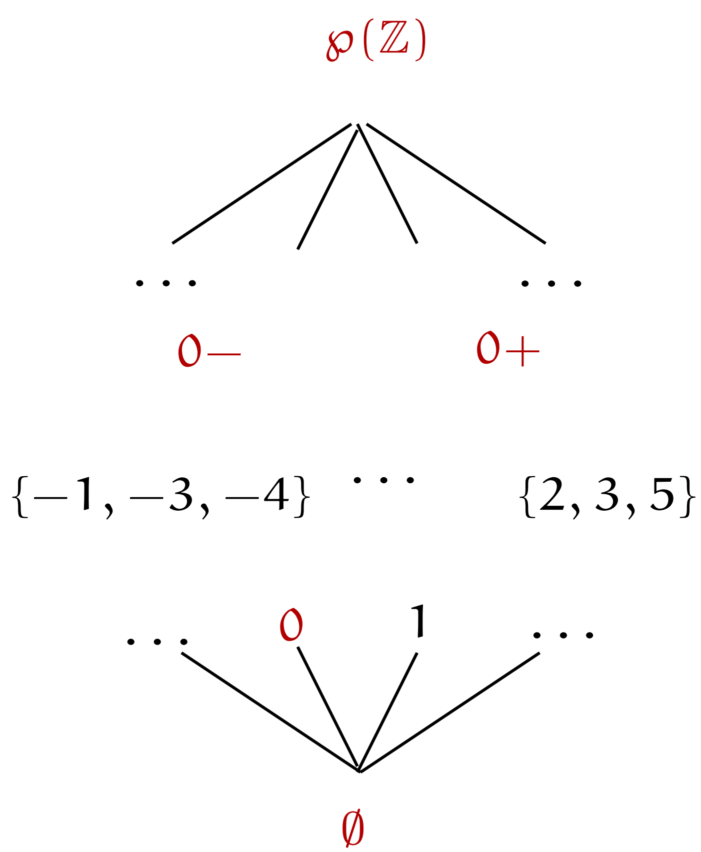
Obscurity as incompleteness

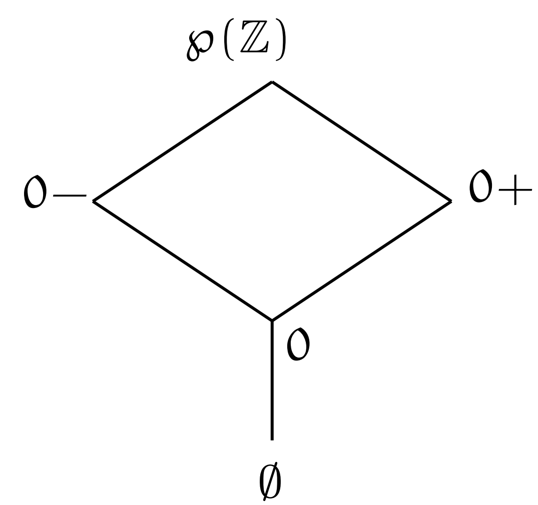
Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

$P : \texttt{x} = \texttt{a} * \texttt{b}$

$Sign$ is an obvious abstraction of $\wp(\mathbb{Z})$:





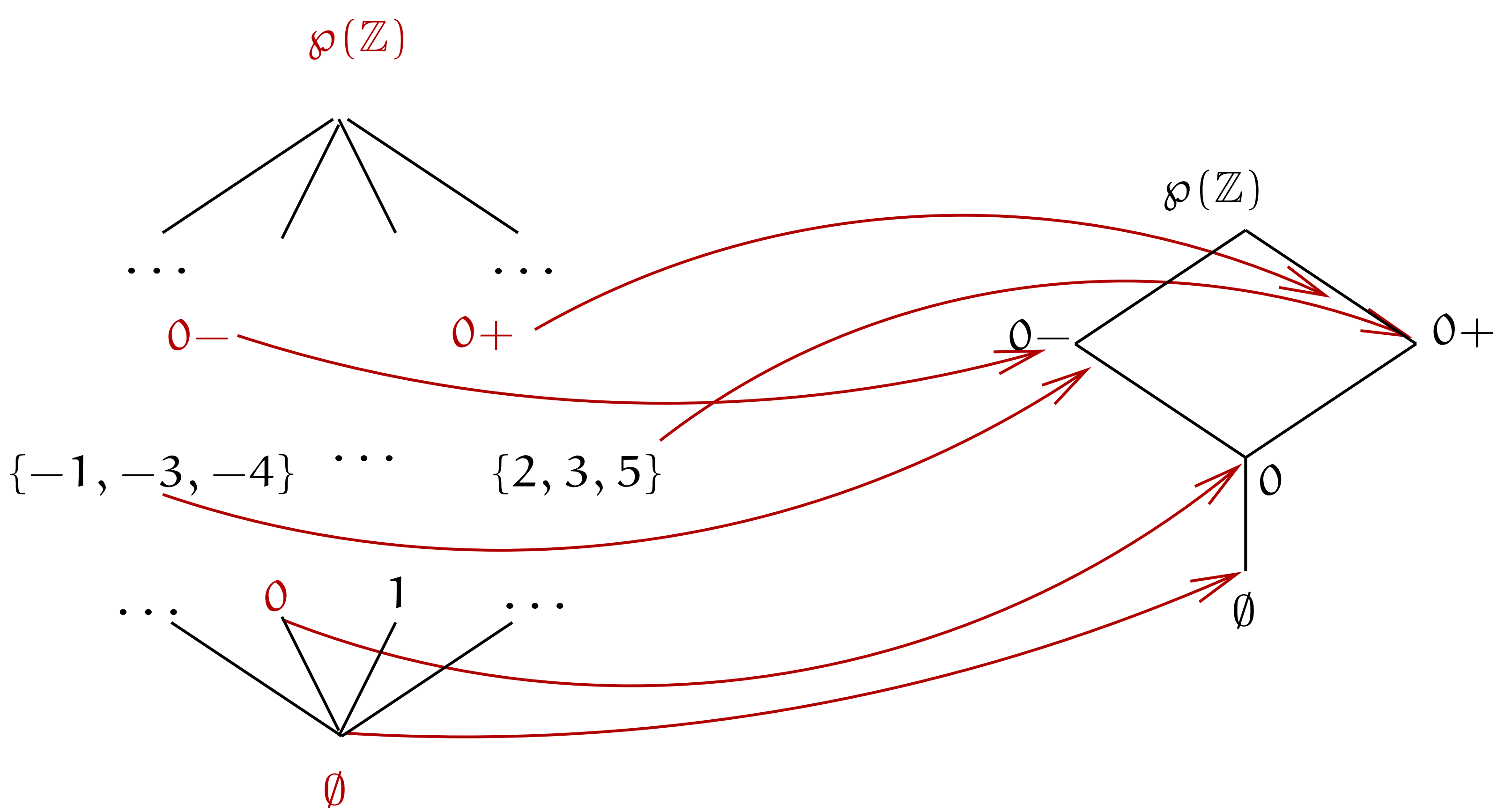
OBSCURITY AS INCOMPLETENESS

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

$P : \texttt{x} = \texttt{a} * \texttt{b}$

$Sign$ is an abstraction of $\wp(\mathbb{Z})$:



Obscurity as incompleteness

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

$$P: \quad \mathtt{x = a * b} \quad \longrightarrow \quad \tau(P): \quad \begin{array}{l} \mathtt{x} = 0; \\ \mathtt{if}\ \mathtt{b} \leq 0\ \mathtt{then}\ \{\mathtt{a} = -\mathtt{a};\ \mathtt{b} = -\mathtt{b}\}; \\ \mathtt{while}\ \mathtt{b} \neq 0\ \{\mathtt{x = a + x};\ \mathtt{b = b - 1}\} \end{array}$$

- $Sign$ is complete for P:
  - ✔ $[\![P]\!]^{Sign} = \lambda \mathtt{a}, \mathtt{b}.\ Sign(\mathtt{a * b})$
- $Sign$ is incomplete for $\tau(P)$:
  - ✔ $[\![\tau(P)]\!]^{Sign} = \lambda \mathtt{a}, \mathtt{b}. \begin{cases} 0 & \text{if } \mathtt{a} = 0 \vee \mathtt{b} = 0 \\ ? = \wp(\mathbb{Z}) & \text{otherwise} \end{cases}$

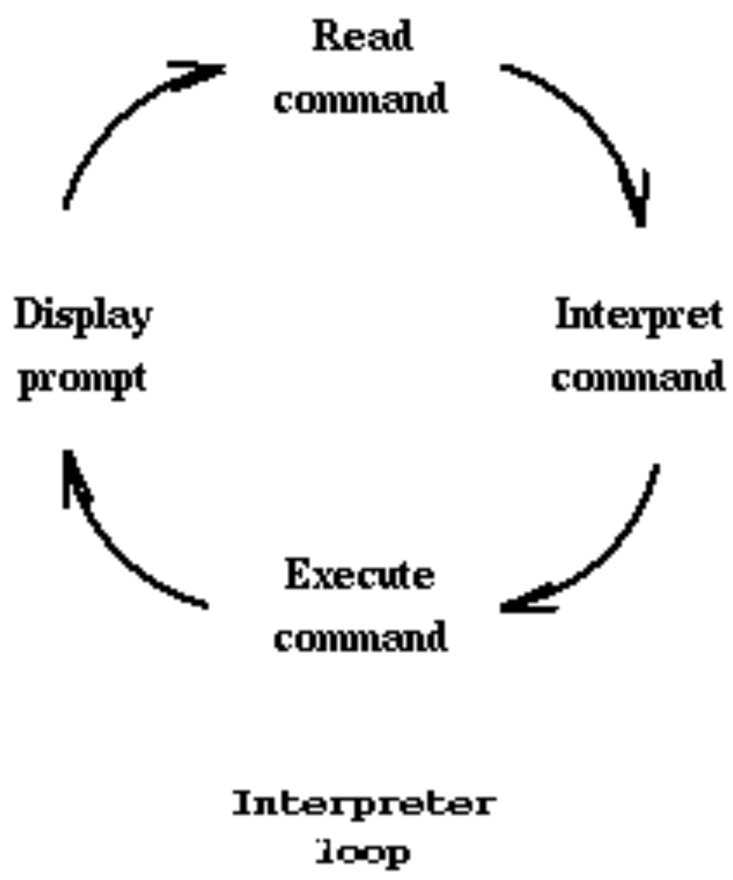
$$
\begin{aligned}
P: &\quad x := a * b \\
Q: &\quad x := a * (b-2) + a + a
\end{aligned}
$$

$$
\begin{aligned}
[\![P]\!]^{\mathsf{Sign}}(\{a \mapsto +, b \mapsto +\}) &= \{x, \mapsto +, a \mapsto +, b \mapsto +\} \\
[\![Q]\!]^{\mathsf{Sign}}(\{a \mapsto +, b \mapsto +\}) &= \{x, \mapsto \mathbb{Z}, a \mapsto +, b \mapsto +\}.
\end{aligned}
$$

Expoliting incompleteness

Maximize $[\![P]\!]^\rho$ incompleteness!

- The abstraction is the specification of the attacker
  - 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)
- Each abstraction is incomplete for a concrete enough trace semantics
- Maximize incompleteness by code transformation: Obfuscation
- Exploit incompleteness for hiding information: Steganography



Interpreter-based Obfuscation

Whole-program VIEW OF OBFUSCATION

A major conflict makes program obfuscation a subtle problem in programming:

Good programs are well-structured and have concise invariants

This is a key to

- understanding a program, and
- adapting it to new purposes.

Good structure and short invariants are necessity in order to develop, debug and perfect a program `P` in the first place.

However, instead *an obfuscated program should* **not** *be well-structured and easy to understand*.

This suggests (among other things):

obfuscation by making the program's control/data flow hard to understand

About $\mathtt{P}' = [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{P})$

1. Program $\mathtt{P}'$ inherits the *algorithm* of program $\mathtt{P}$.
2. Program $\mathtt{P}'$ inherits the *programming style* of $\mathtt{interp}$.

1: A correct interpreter $\mathtt{interp}$ must faithfully execute the operations specified by program $\mathtt{P}$. Usually: specialized program $\mathtt{P}'$ performs *the same computations in the same order* as those performed by $\mathtt{P}$.

Most interpreters do not devise new computational approaches!

2: $\mathtt{P}'$ consists of specialized code from the interpreter $\mathtt{interp}$:

$\mathtt{P}' =$ the operations of $\mathtt{interp}$ that depend on its dynamic input
(all others will be “specialized away”).

Idea

Build a *general-purpose program transformer* by programming a self-interpreter in a style to give the desired transformation

CLAIM: $[\![\texttt{P}]\!] = [\![\texttt{P}']\!]$, by simple equational reasoning:

$$
\begin{array}{lcll}
[\![\texttt{P}]\!](\texttt{d}) & = & [\![\texttt{interp}]\!](\texttt{P},\texttt{d}) & \text{definition of self-interpreter} \\
 & = & [\![[\![\texttt{spec}]\!](\texttt{interp},\texttt{P})]\!](\texttt{d}) & \text{definition of specializer} \\
 & = & [\![\texttt{P'}]\!](\texttt{d}) & \text{definition of } \texttt{P'}
\end{array}
$$

Therefore the function

$$\texttt{P} \longmapsto [\![\texttt{spec}]\!](\texttt{interp},\texttt{P})$$

is a *semantics-preserving program transformer*!!

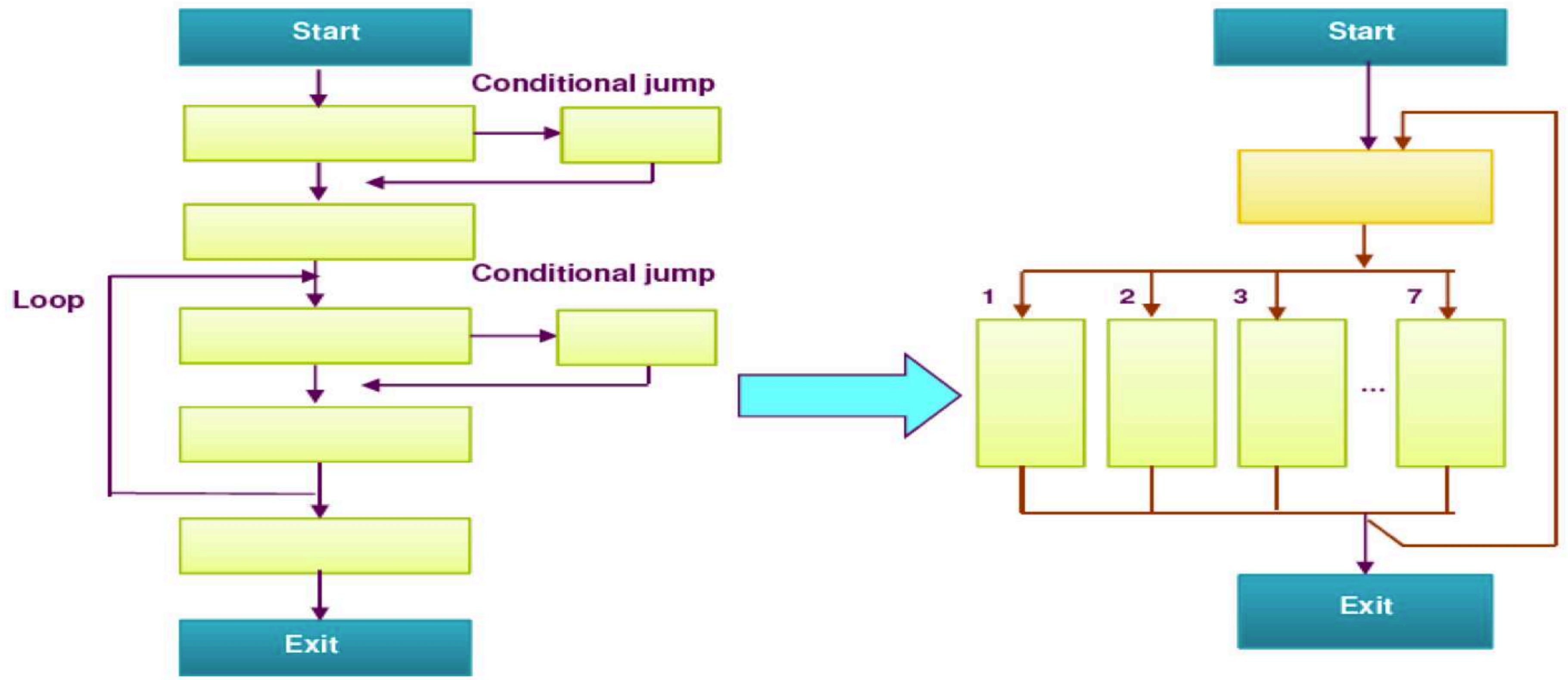
We need to *change the interpretation*: $\texttt{interp} \leadsto \texttt{interp}^{+}$

Flattening

CODE FLATTENING

[Cloackware 2000]

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



Original Program Flow

Control Flow Flattened Program

Example of Flattening

The following flattened program $\texttt{P}'$ has

- only one loop (regardless of how many loops $\texttt{P}$ has), and
- an explicit program counter pc

Original program $\texttt{P}$:

$$
\begin{array}{l}
^{1.}\textbf{input}\ x; \\
^{2.}y := 2; \\
^{3.}\textbf{while}\ x > 0\ \textbf{do} \\
\quad ^{4.}y := y + 2; \\
\quad ^{5.}x := x - 1 \\
\ \ \textbf{endw} \\
^{6.}\textbf{output}\ y; \\
^{7.}\textbf{end}
\end{array}
$$

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

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

Structure of a simple self-interpreter

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

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

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

$\textbf{while}\ pc < length(\texttt{P})\ \textbf{do}$ pc dynamic!

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

$\quad \textbf{case}\ instruction\ \textbf{of}$ Dispatch on syntax

$\quad \textbf{skip} \quad : pc := pc + 1;$

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

$\quad \ldots\ \textbf{endw};$

$\textbf{output}\ store[out];$

$eval(e, store) \ = \ \textbf{case}\ e\ \textbf{of}$ Function to evaluate expressions

$\quad constant : e$

$\quad variable : store(e)$

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

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

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

$\texttt{target} := [\![\texttt{spec}]\!](\texttt{int}, \texttt{source})$

Why?

THE CFG ABSTRACTION

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

- Theorem

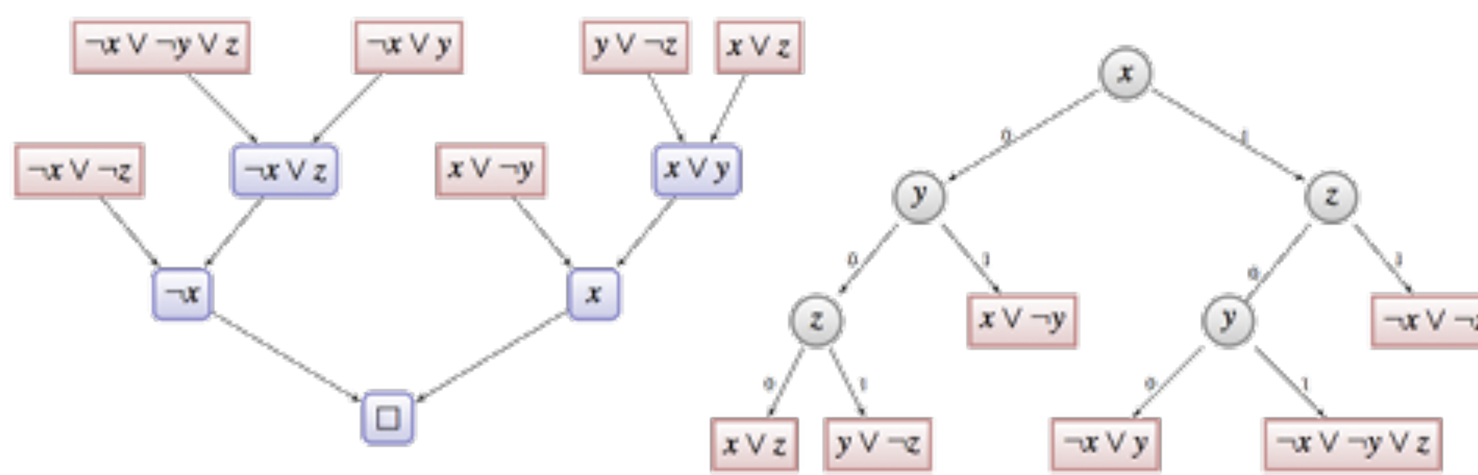
Completeness!!

pc dynamic!

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

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

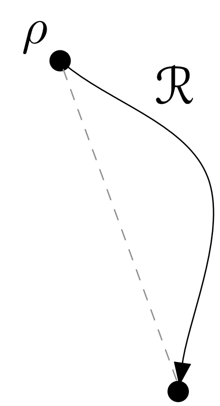
A Theory?



$\neg x \vee \neg y \vee z$
$\neg x \vee y$
$y \vee \neg z$
$x \vee z$
$\neg x \vee \neg z$
$\neg x \vee z$
$x \vee \neg y$
$x \vee y$
$\neg x$
x
$\square$
x
y
z
0
1
$x \vee \neg y$
$\neg x \vee \neg z$
$x \vee z$
$y \vee \neg z$
$\neg x \vee y$
$\neg x \vee \neg y \vee z$

Simplifying abstractions

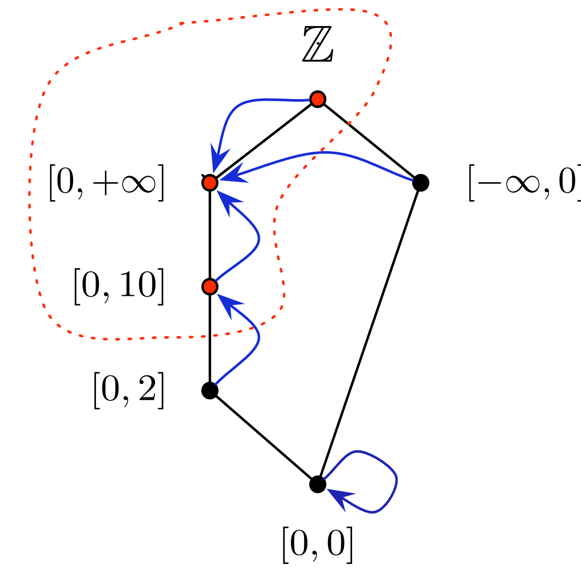
$$\mathcal{R} \stackrel{\text{def}}{=} \mathcal{R}^{\mathscr{F}}_{f,\eta} = \lambda\rho.\ \rho \sqcap \mathcal{M}(f(\eta))$$

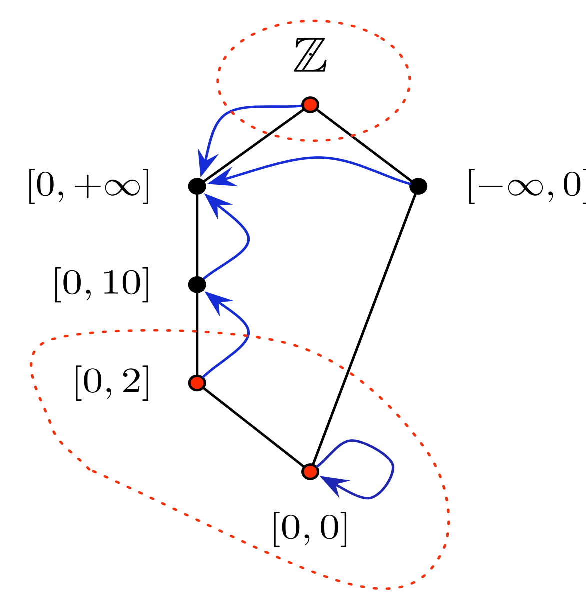


More abstract

More concrete

$$sq(X) = \left\{\, x^2 \;\middle|\; x \in X \,\right\}$$

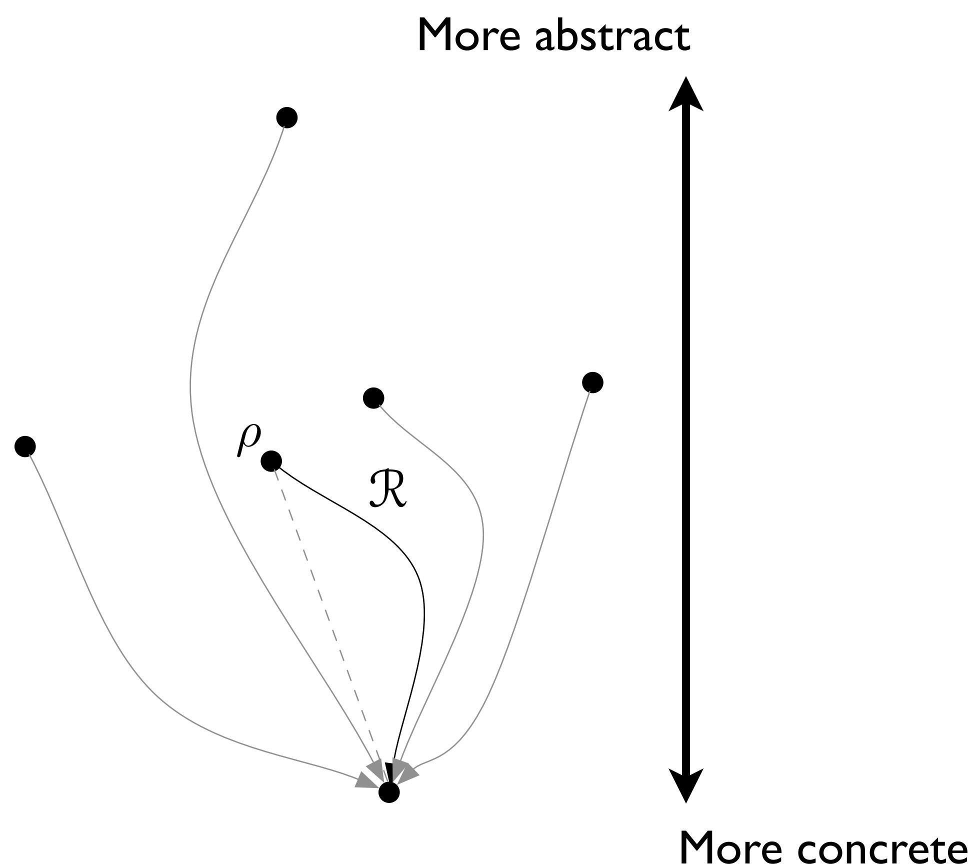


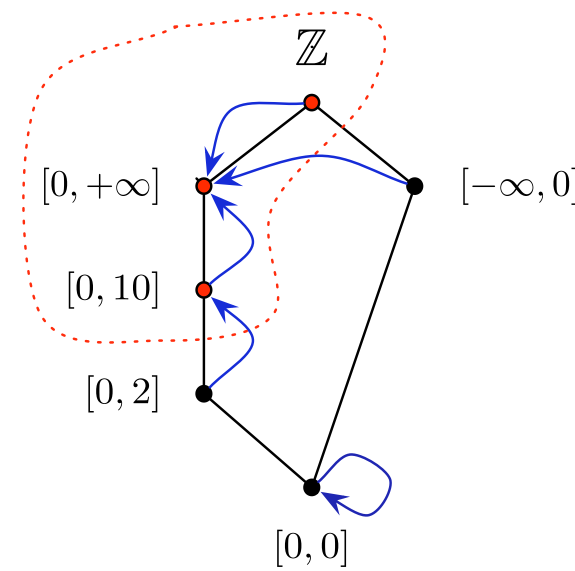


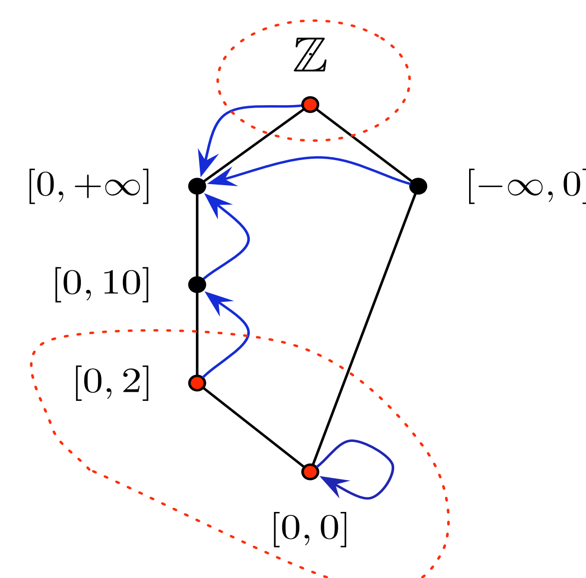
Simplifying abstractions

$$sq(X) = \left\{ x^2 \;\middle|\; x \in X \right\}$$

$$\mathcal{R} \stackrel{\text{def}}{=} \mathcal{R}^{\mathscr{F}}_{f,\eta} = \lambda\rho.\, \rho \sqcap \mathcal{M}(f(\eta))$$

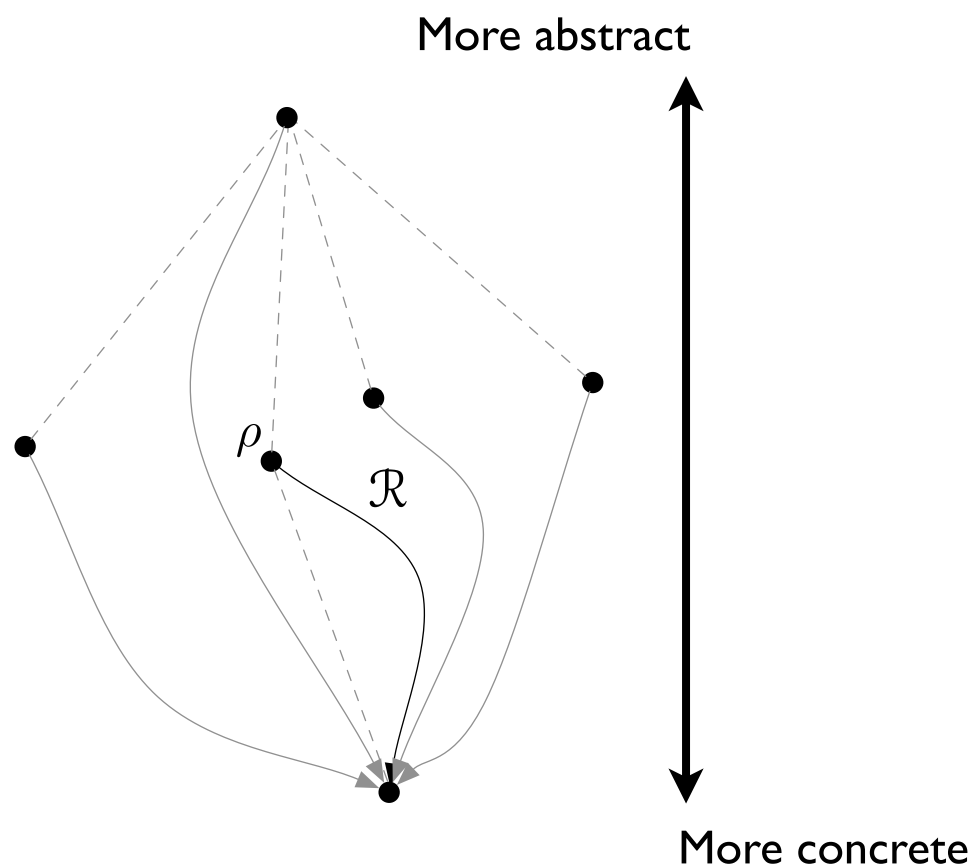






Simplifying abstractions

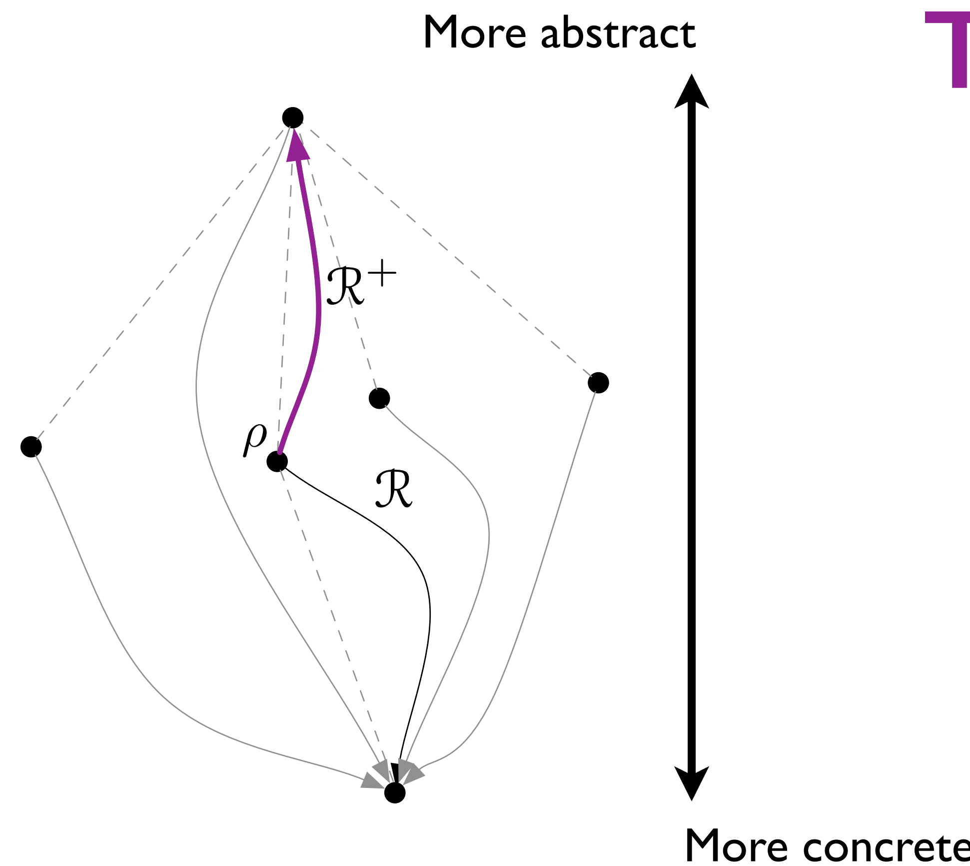
$$\mathcal{R} \stackrel{\text{def}}{=} \mathcal{R}^{\mathscr{F}}_{f,\eta} = \lambda\rho.\, \rho \sqcap \mathcal{M}(f(\eta))$$



Simplifying abstractions

The Inverse!

$$\mathcal{R} \stackrel{\text{def}}{=} \mathcal{R}^{\mathscr{F}}_{f,\eta} = \lambda\rho.\ \rho \sqcap \mathcal{M}(f(\eta))$$

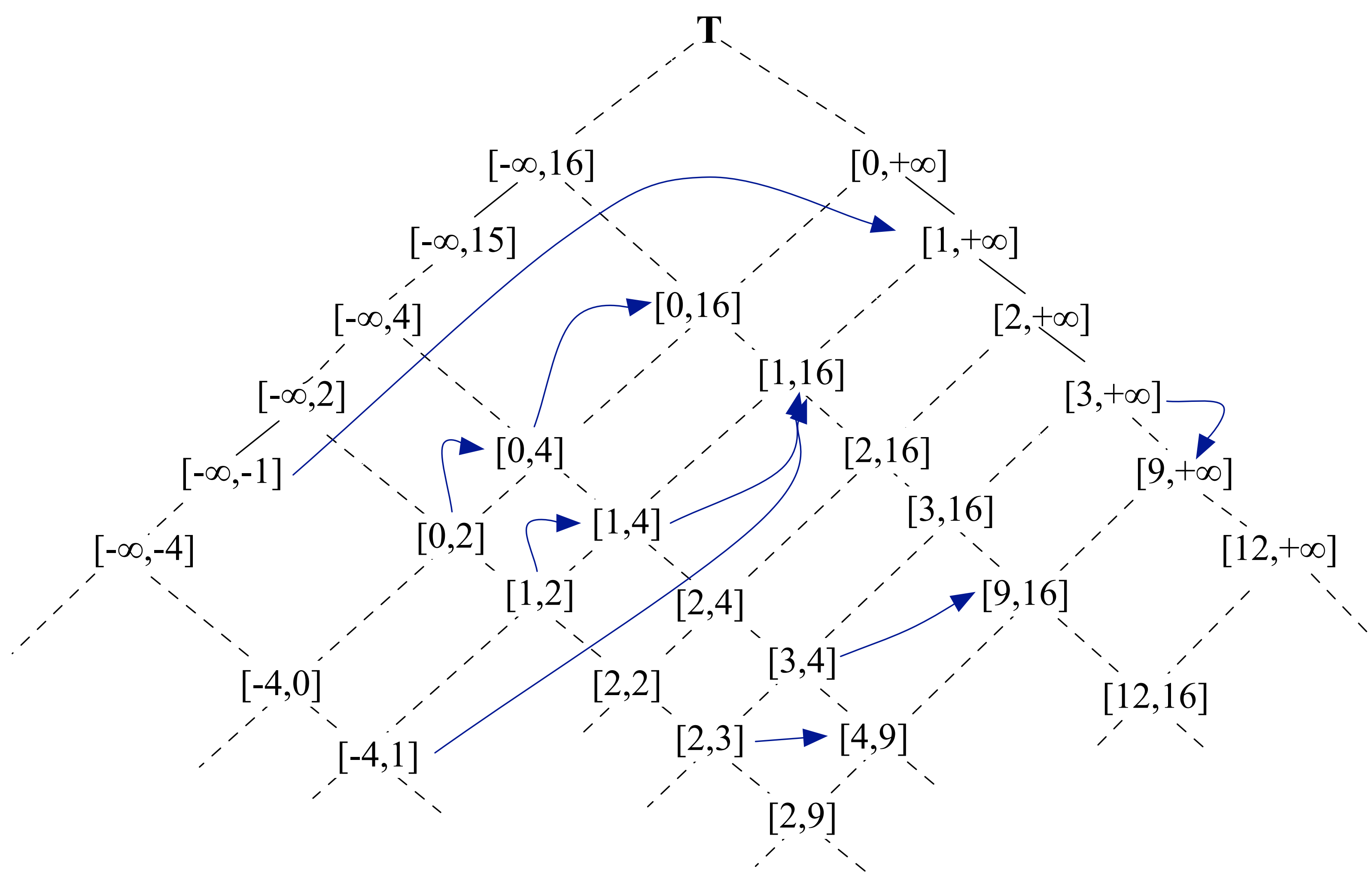


$$\mathcal{R}^+ = \lambda\rho.\ \bigsqcup \{\, \delta \mid \mathcal{R}(\delta) = \mathcal{R}(\rho) \,\} = \lambda\rho.\ \bigsqcup \{\, \delta \mid \delta \sqcap \mathcal{M}(f(\eta)) = \rho \sqcap \mathcal{M}(f(\eta)) \,\}$$

$$\rightarrow \boxed{\lambda\rho.\ \mathcal{M}(\mathit{Mirr}(\rho \sqcap \mathcal{M}(f(\eta))) \smallsetminus \mathcal{M}(f(\eta)))}$$

Example

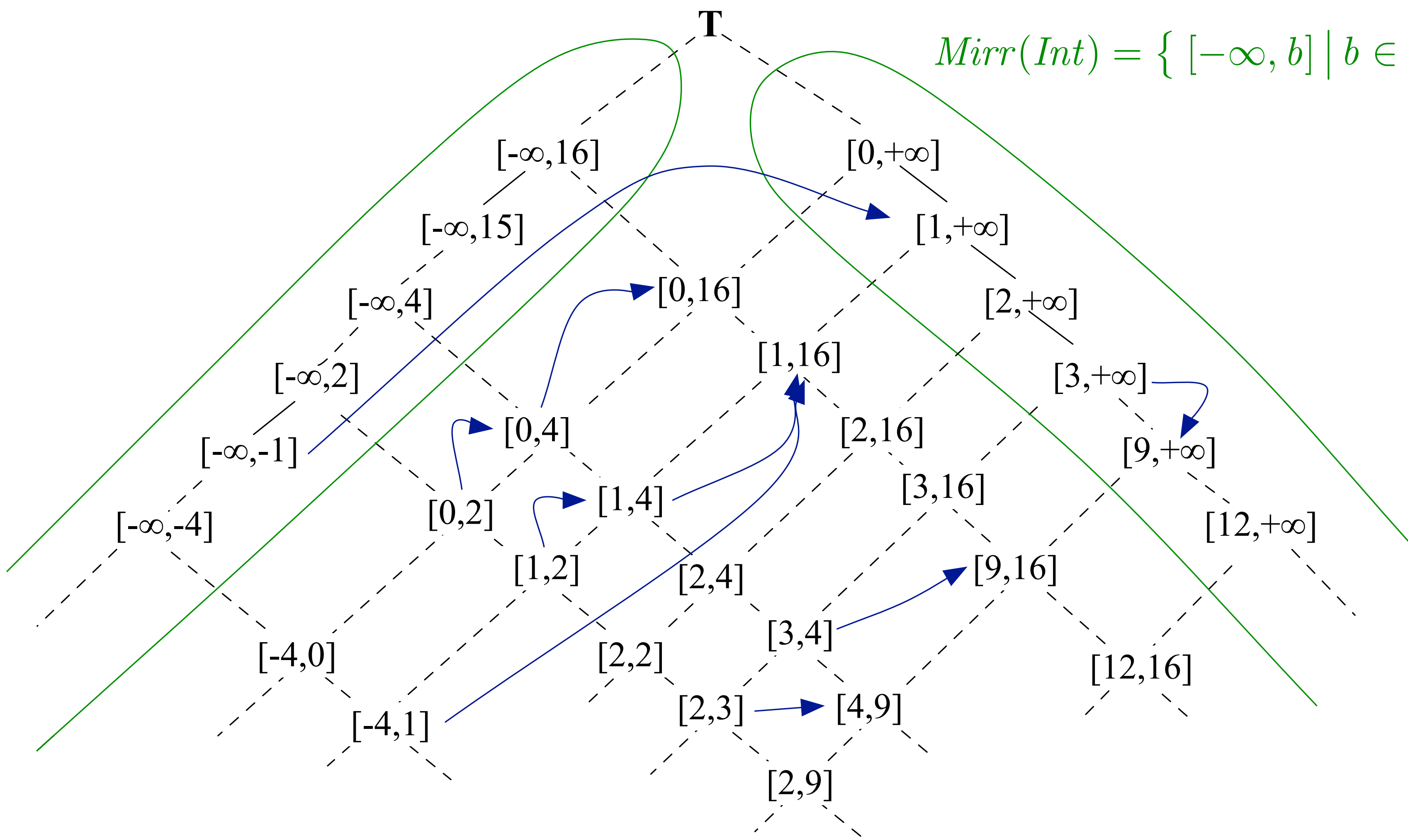
$$sq(X) = \left\{ \, x^2 \; \middle| \; x \in X \, \right\}$$



Example

$$sq(X) = \{\ x^2 \mid x \in X\ \}$$

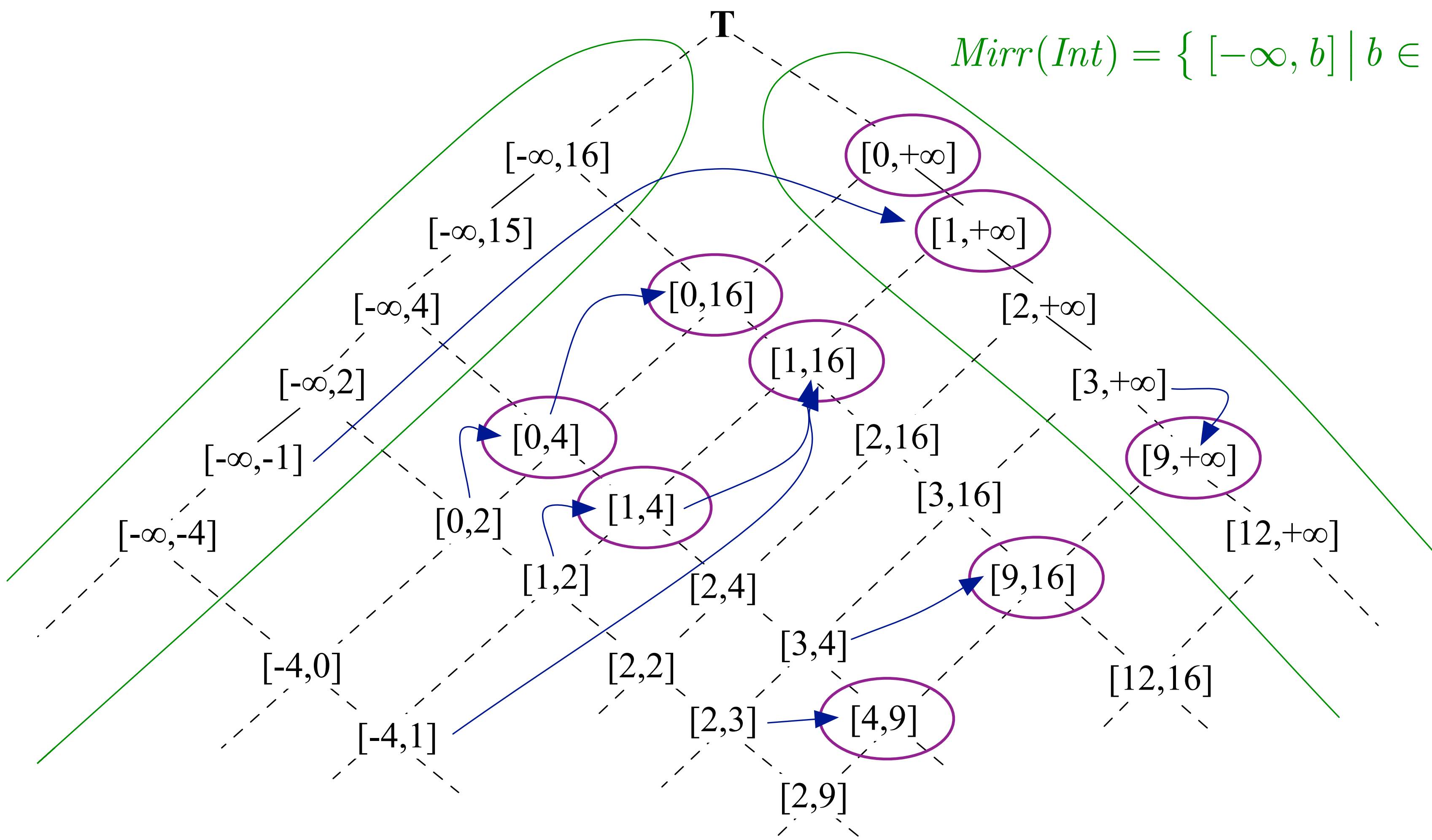
$$Mirr(Int) = \{\ [-\infty, b] \mid b \in \mathbb{Z}\ \} \cup \{\ [a, +\infty] \mid a \in \mathbb{Z}\ \}$$



Example

$$sq(X) = \left\{\ x^2 \ \middle|\ x \in X\ \right\}$$

$$\mathit{Mirr}(\mathit{Int}) = \left\{\ [-\infty, b] \ \middle|\ b \in \mathbb{Z}\ \right\} \cup \left\{\ [a, +\infty] \ \middle|\ a \in \mathbb{Z}\ \right\}$$



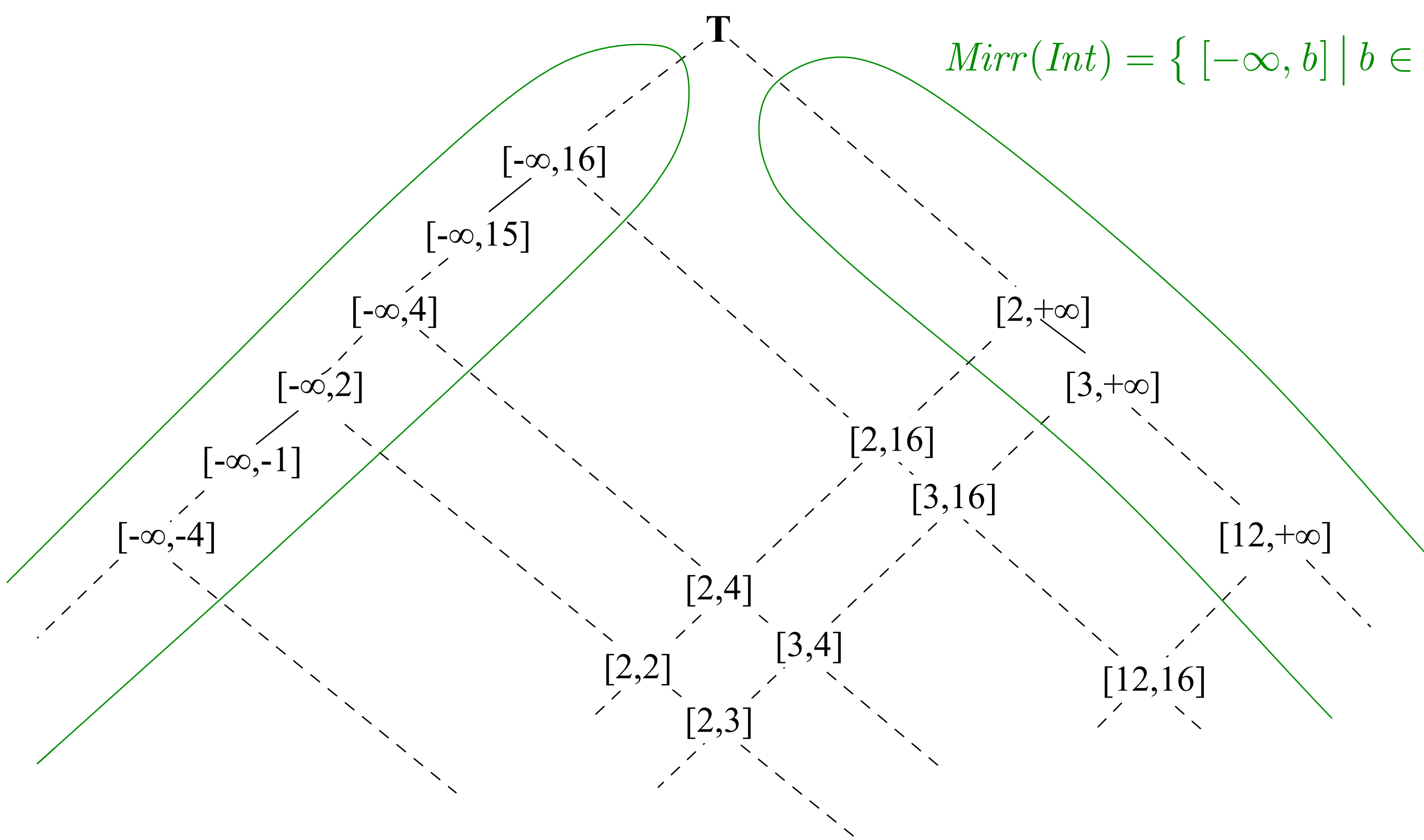
$$\mathcal{M}(sq^\sharp(\mathit{Int})) = \left\{\ [a^2, b^2] \ \middle|\ a, b \in \mathbb{Z}\ \right\} \cup \left\{\ [a^2, +\infty] \ \middle|\ a \in \mathbb{Z}\ \right\}$$

$$\mathit{Int}' = \mathcal{M}(\mathit{Mirr}(\mathit{Int}) \smallsetminus \mathcal{M}(sq^\sharp(\mathit{Int})))$$

Example

$$sq(X) = \left\{\, x^2 \;\middle|\; x \in X \,\right\}$$

$$\mathit{Mirr}(\mathit{Int}) = \left\{\, [-\infty, b] \;\middle|\; b \in \mathbb{Z} \,\right\} \cup \left\{\, [a, +\infty] \;\middle|\; a \in \mathbb{Z} \,\right\}$$



$$\mathcal{M}(sq^\sharp(\mathit{Int})) = \left\{\, [a^2, b^2] \;\middle|\; a, b \in \mathbb{Z} \,\right\} \cup \left\{\, [a^2, +\infty] \;\middle|\; a \in \mathbb{Z} \,\right\}$$

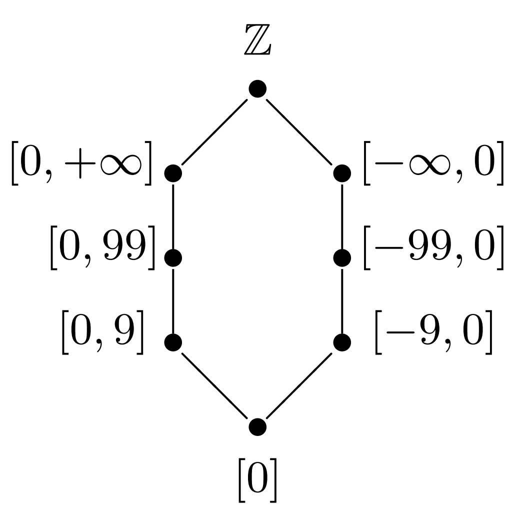
$$\mathit{Int}' = \mathcal{M}(\mathit{Mirr}(\mathit{Int}) \smallsetminus \mathcal{M}(sq^\sharp(\mathit{Int}))) =$$
$$\left\{\, [a, b] \;\middle|\; a, b \in \mathbb{Z}, \nexists c.\ a = c^2 \vee \nexists d.\ b = d^2 \,\right\} \cup \left\{\, [-\infty, b] \;\middle|\; b \in \mathbb{Z} \,\right\} \cup \left\{\, [a, +\infty] \;\middle|\; a \in \mathbb{Z}, \nexists c.\ a = c^2 \,\right\}$$

Absolute Incomplete Compressor

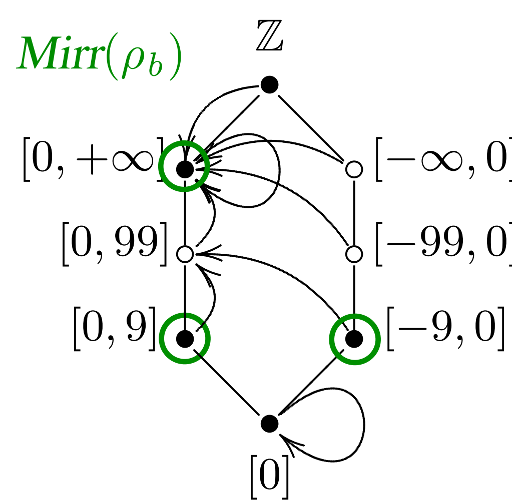
If $\mathcal{R}^+(\rho)$ is not T then

➡ $\mathcal{R}^+(\rho)$ is absolute incomplete compressor, i.e.,

$$\mathcal{R}^+(\rho) \circ f \circ \mathcal{R}^+(\rho) \neq f \circ \mathcal{R}^+(\rho)$$

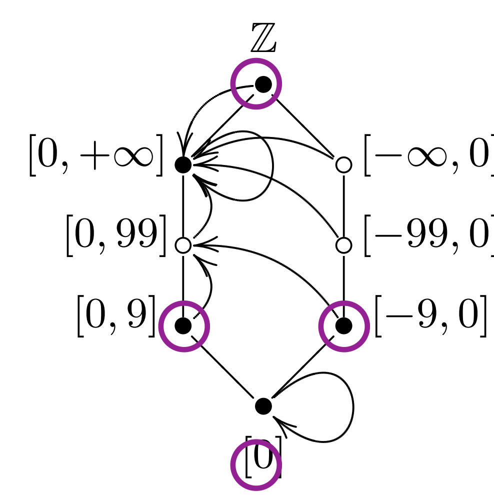


$\mathtt{S} = \rho_{\mathtt{S}}(\wp(\mathbb{Z}))$



$$\rho_b = \{\mathbb{Z}, [0,+\infty], [0,9], [-9,0], [0]\}$$

$$sq^{\mathtt{S}}(X) = \rho_{\mathtt{S}}(sq(X))$$



$S' = \mathcal{M}(Mirr(\rho_b \sqcap \mathcal{M}(sq^s(\rho_b))) \setminus \mathcal{M}(sq^s(\rho_b))$

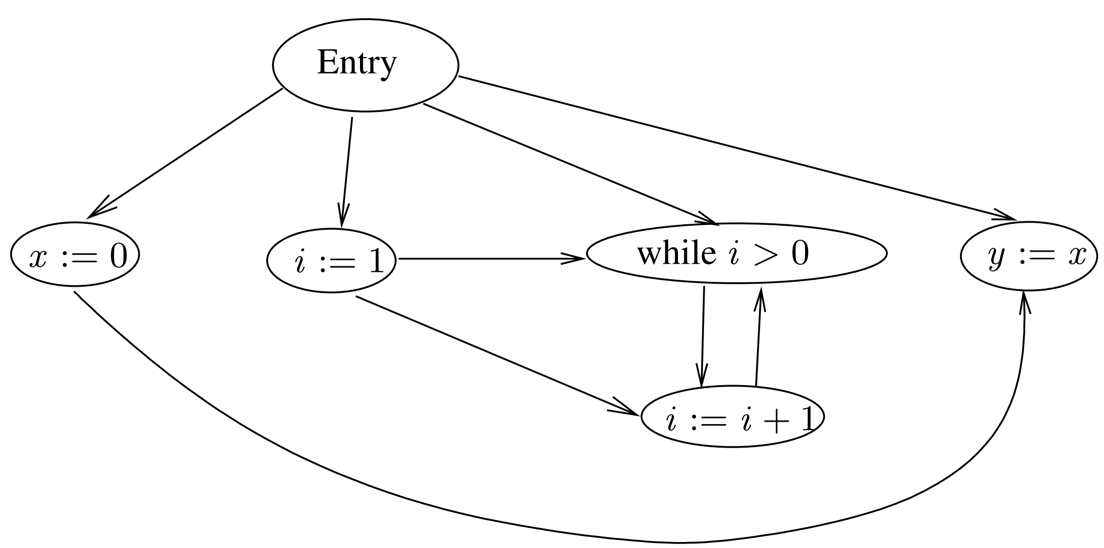
$$\mathtt{S}' \circ sq^{\mathtt{S}} \circ \mathtt{S}' \neq sq^{\mathtt{S}} \circ \mathtt{S}'$$

Slicing

Program Slicing Obfuscation

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.

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



Data Dependency (**Slicing**) 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); }
```

Data Dependency (**Slicing**) 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); }
```

Data Dependency (**Slicing**) 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); }
```

Data Dependency (Slicing) **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); }
```

Data Dependency (Slicing) **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

Data Dependency (Slicing) **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); }
```

Data Dependency (Slicing) **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); }
```

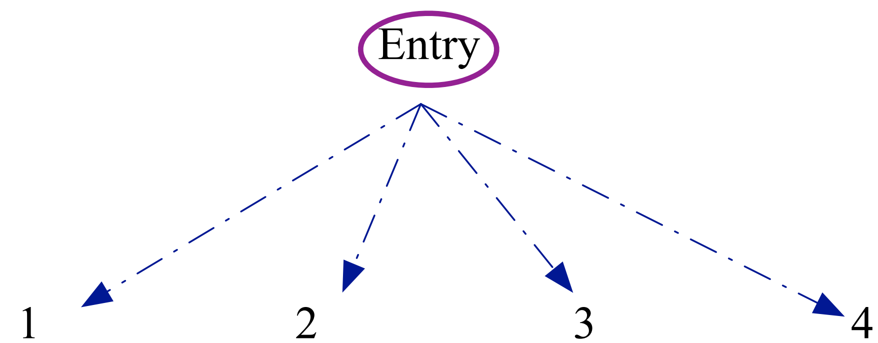
DAPA

Digital Asset Protection Assn.

Attack: Semantic PDG

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

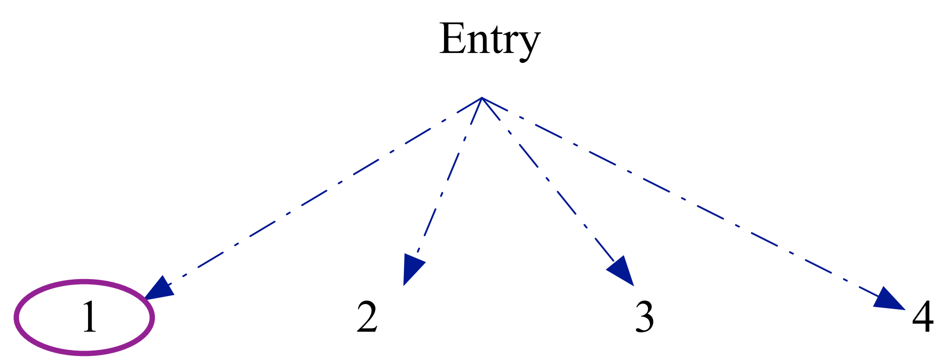
$D_{\text{Entry}} = \varnothing \qquad s = \langle \sigma, \langle \text{Entry}, 1 \rangle \rangle$



Attack: Semantic PDG

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

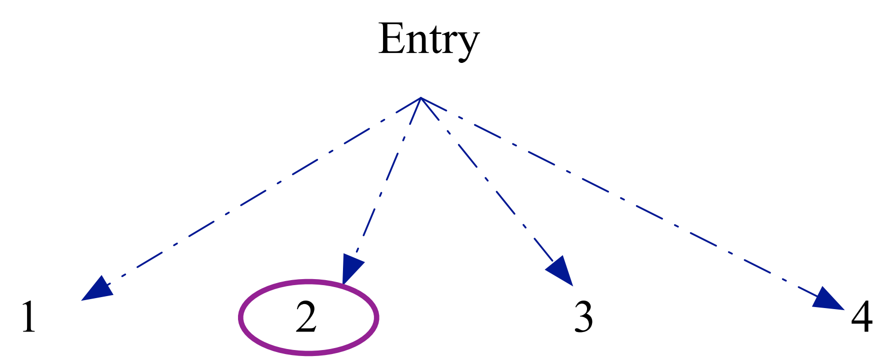
$D_1 = [D_x = 1]$ $\qquad s = \langle \sigma, \langle 1, 2 \rangle \rangle$



Attack: Semantic PDG

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

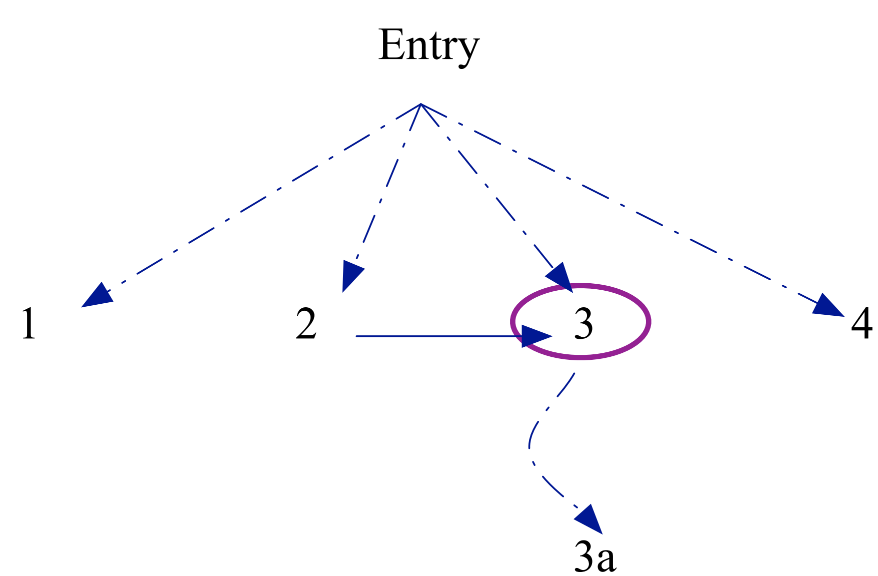
$D_2 = [D_x = 1, D_i = 2]$ $\qquad$ $s = \langle \sigma, \langle 2, 3 \rangle \rangle$



Attack: Semantic PDG

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

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

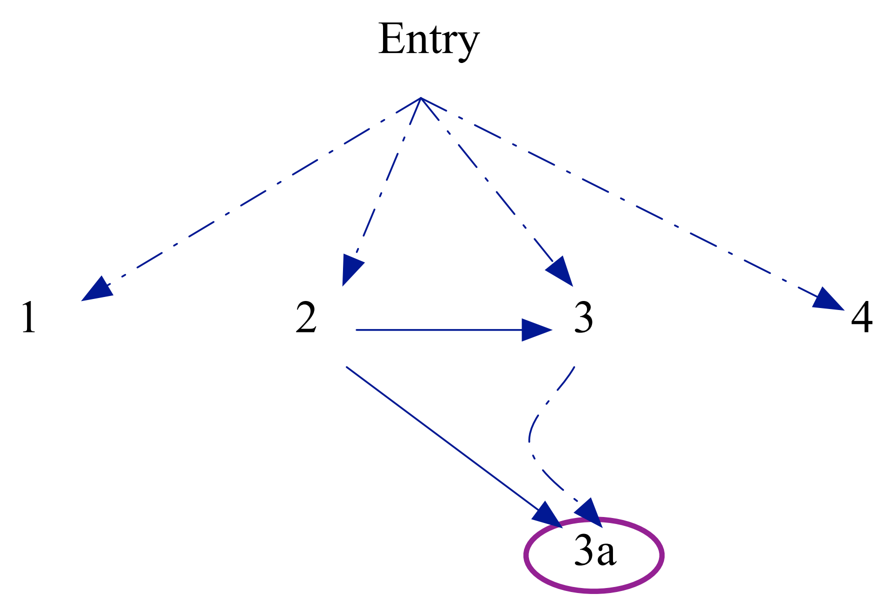


Attack: Semantic PDG

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

$D_{3a} = [D_x = 1, D_i = 3a]$

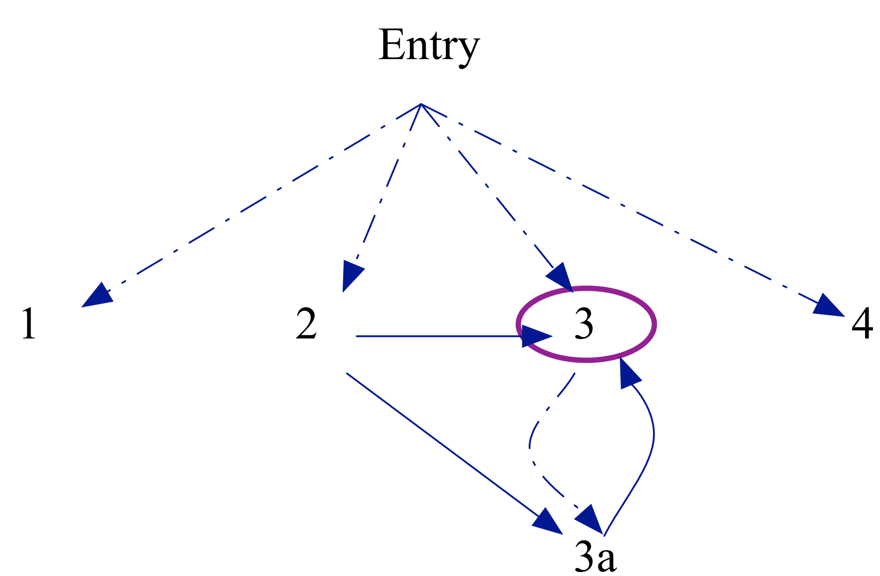
$s_1 = \langle \sigma, \langle 3a, 3 \rangle \rangle$ and $s_2 = \langle \sigma, \langle 3, 4 \rangle \rangle$



Attack: Semantic PDG

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

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



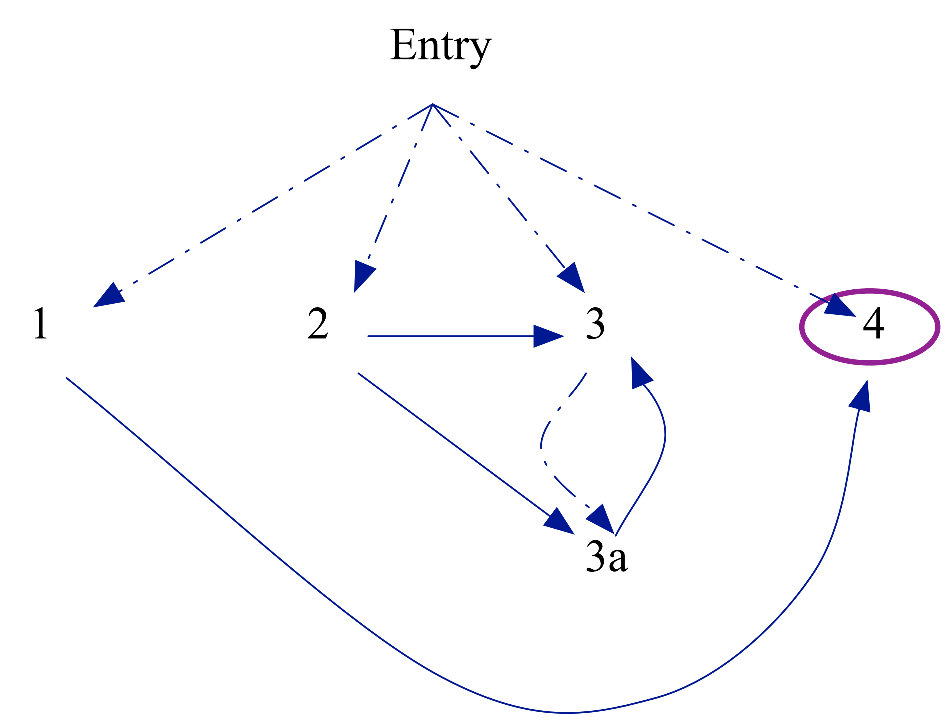
fixpoint

Attack: Semantic PDG

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

$D_4 = [D_x = 1, D_i = 2, D_y = 4]$

$s_1 = \langle \sigma, \langle 3, 3a \rangle\rangle$ and $s_2 = \langle \sigma, \langle 4, \bot \rangle\rangle$



Potency of Data obfuscation

- If a program P contains *fake dependencies* (i.e., not semantic) the analyses deceived are those analyzing syntactic dependencies only!

- The incomplete backward compressor of the abstraction of the PDG based on *semantic* dependencies is the PDG based on *syntactic* dependencies only, i.e., standard slicing!

- Namely, any obfuscator adding fake dependencies deceives slicing algorithms based on the computation of syntactic dependencies!!

The semantic PDG is complete (the PDG analysis is precise) iff the program does not contain fake dependencies

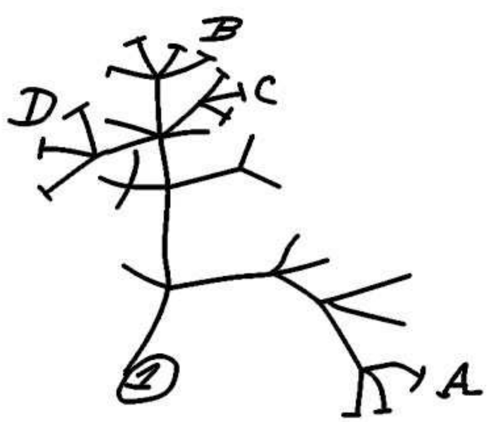
Concluding

So what?

- Any obfuscation technique is an instance of

$$[\![\texttt{spec}]\!](\texttt{interp}^{+}, \texttt{P})$$

for some $\texttt{interp}^{+}$ 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 $\texttt{P}$, what is α?

- Given α, can we derive $\texttt{interp}^{+}$ systematically?

META-LEVEL DISCUSSION

The Futamura projections are as follow for a distorted interpreter $\texttt{interp}^+$:

$$
\begin{array}{llcll}
1. & \texttt{P}' & := & [\![\texttt{spec}]\!](\texttt{interp}^+, \texttt{P}) & \text{Transform program} \\
2. & \texttt{comp} & := & [\![\texttt{spec}]\!](\texttt{spec}, \texttt{interp}^+) & \text{Generate transformer} \\
3. & \texttt{cogen} & := & [\![\texttt{spec}]\!](\texttt{spec}, \texttt{spec}) & \text{Transformer generator}
\end{array}
$$

We have just seen instances of the 1st Futamura projection.

If the set of *incomplete program structures* is Turing complete: Write the distorted interpreter as incomplete structures

If you want to act *locally*: use interference to ensure that incompleteness is propagated

An *obfuscating compiler* can also be generated, by the 2nd Futamura projection; this has been done using the UNMIX partial evaluator.

For example, if `P` is $\texttt{interp}^{\texttt{flat}}$, then `compiler` is a *stand-alone obfuscator*: a “flattening” program transformer.

Immediate consequences

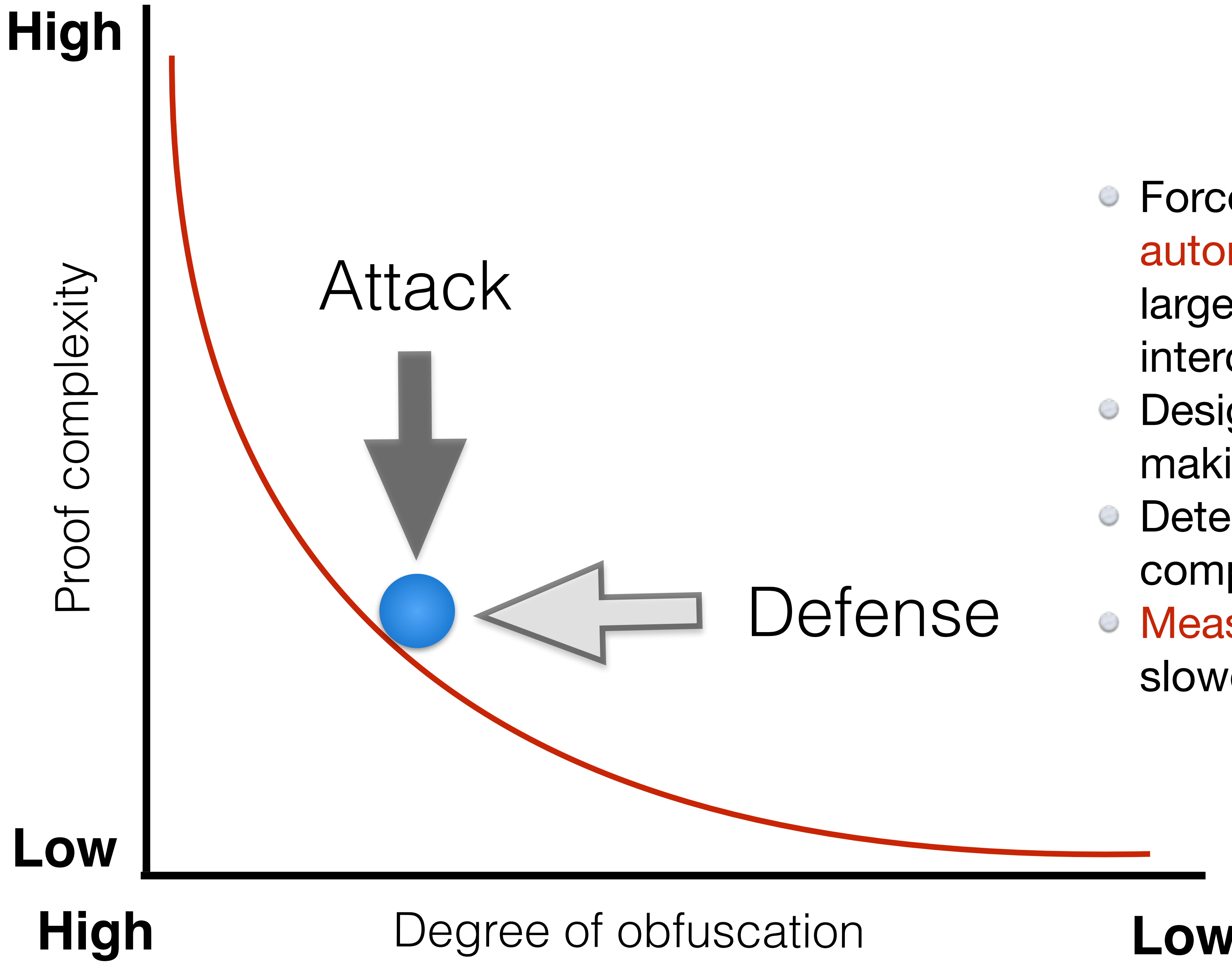
There are other better ways to obfuscate and to produce a obfuscator:

- $\mathtt{P}' = [\![\mathtt{comp}]\!](\mathtt{p})$ (obfuscate by compiler) and
- $\mathtt{comp} = [\![\mathtt{cogen}]\!](\mathtt{interp}^{+})$ (generate obfuscator).

Future developments will involve gaining a deeper understanding in expected time factors:

1. $\mathtt{time}_{\mathtt{P}'}(\mathtt{d})$ and $\mathtt{time}_{\mathtt{P}}(\mathtt{d})$, exemplifying
   the slowdown imposed by the obfuscation;
2. $\mathtt{time}_{\mathtt{spec}}(\mathtt{interp}^{+}, \mathtt{p})$ and $length(\mathtt{P})$, exemplifying the
   the amount of time required to do the obfuscation by specialization;
3. $\mathtt{time}_{\mathtt{comp}}(\mathtt{p})$ and $length(\mathtt{P})$, exemplifying the
   time to do the obfuscation by running the generated obfuscator

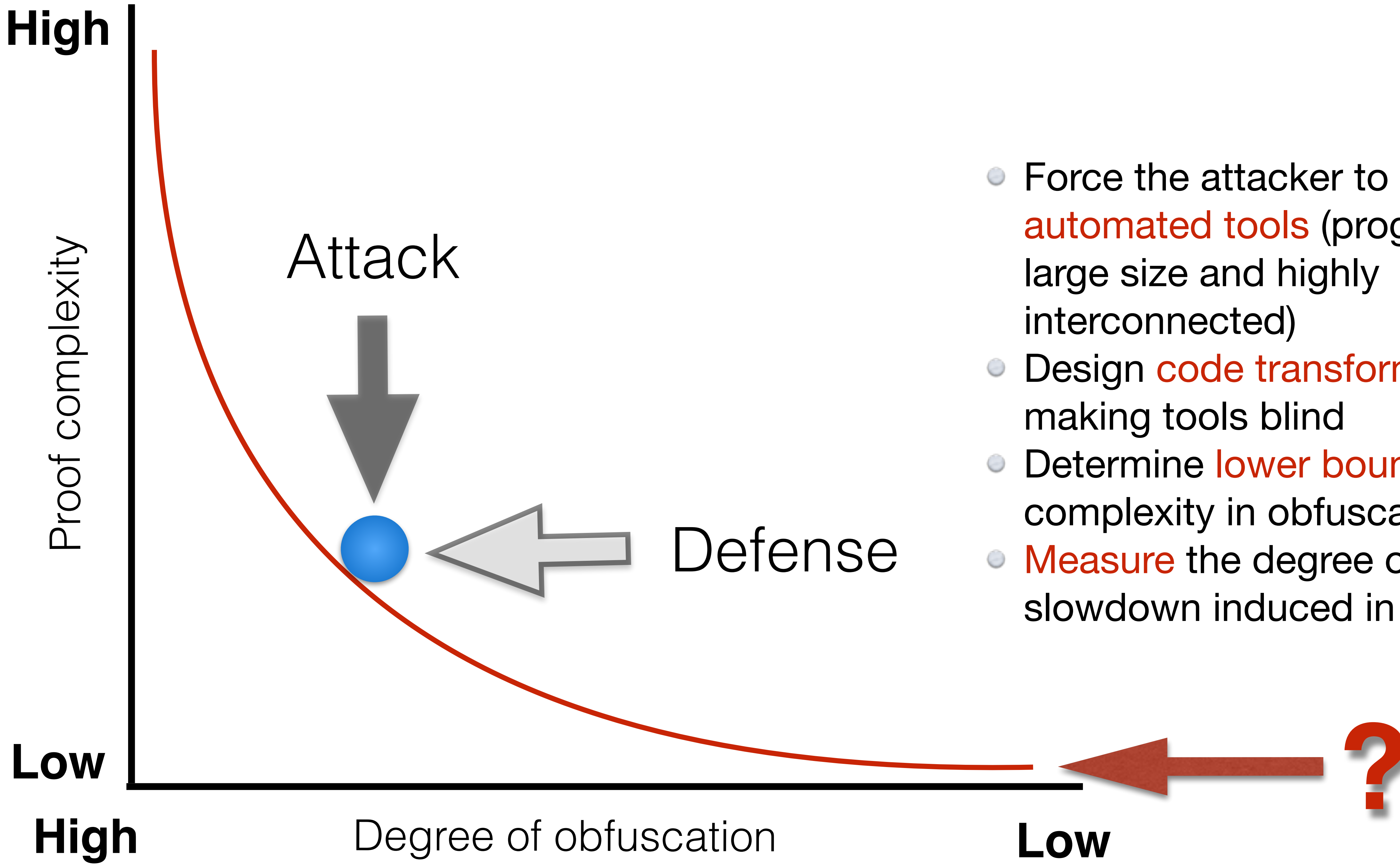
Measuring Adversary Strength?



- Force the attacker to use automated tools (programs of large size and highly interconnected)
- Design code transformations making tools blind
- Determine lower bounds for proof complexity in obfuscated code
- Measure the degree of noise/ slowdown induced in obfuscation

By constraining the adversary within a theorem prover we can quantify the security achieved from obfuscation

Measuring Adversary Strength?



- Force the attacker to use automated tools (programs of large size and highly interconnected)
- Design code transformations making tools blind
- Determine lower bounds for proof complexity in obfuscated code
- Measure the degree of noise/ slowdown induced in obfuscation

By constraining the adversary within a theorem prover we can quantify the security achieved from obfuscation

Thanks!

Mila

Neil

Isa