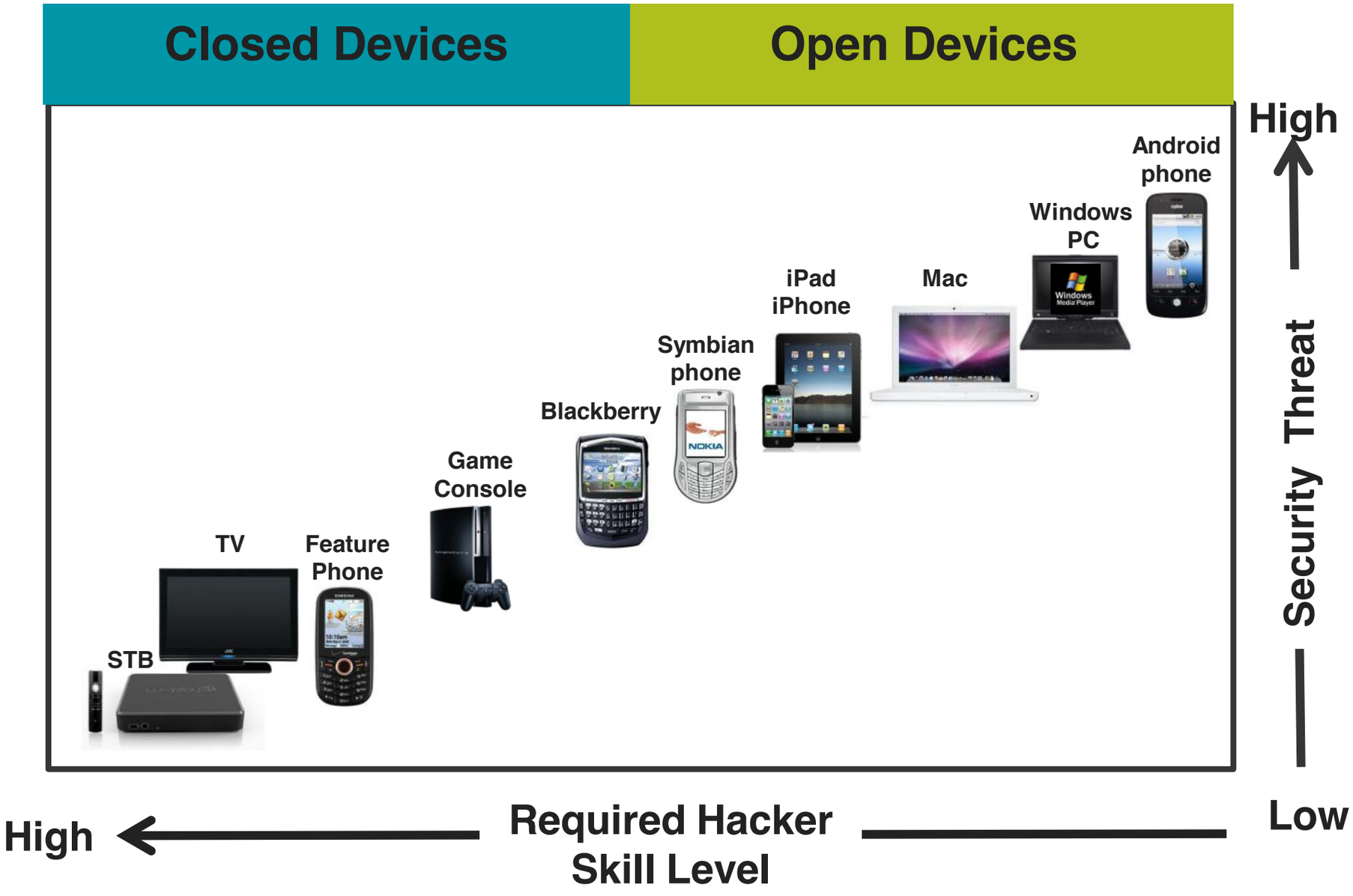
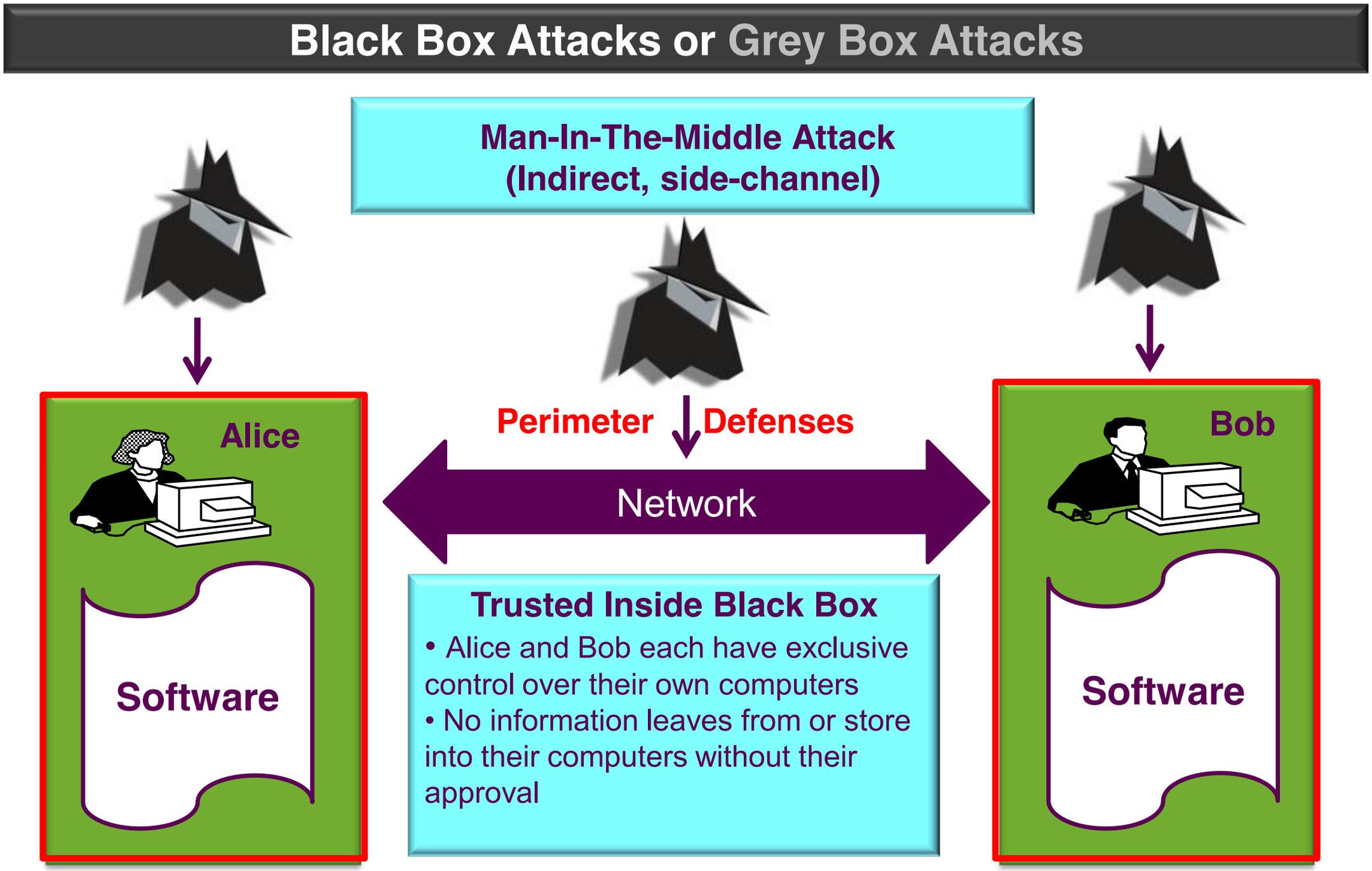
# Software Protection

# Open platforms pose opportunities & risks




- Ongoing trend towards openness
- Open devices/platforms attracts more developers and consumers
- Unfortunately they decrease  the required hacker skill level

# Cryptographic Assumption and Traditional Attacks



# 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

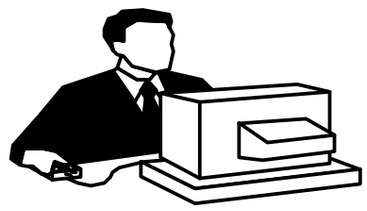
**Attacking has much less limitation than protection**


Network

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

**Man-At-The-End Attack**

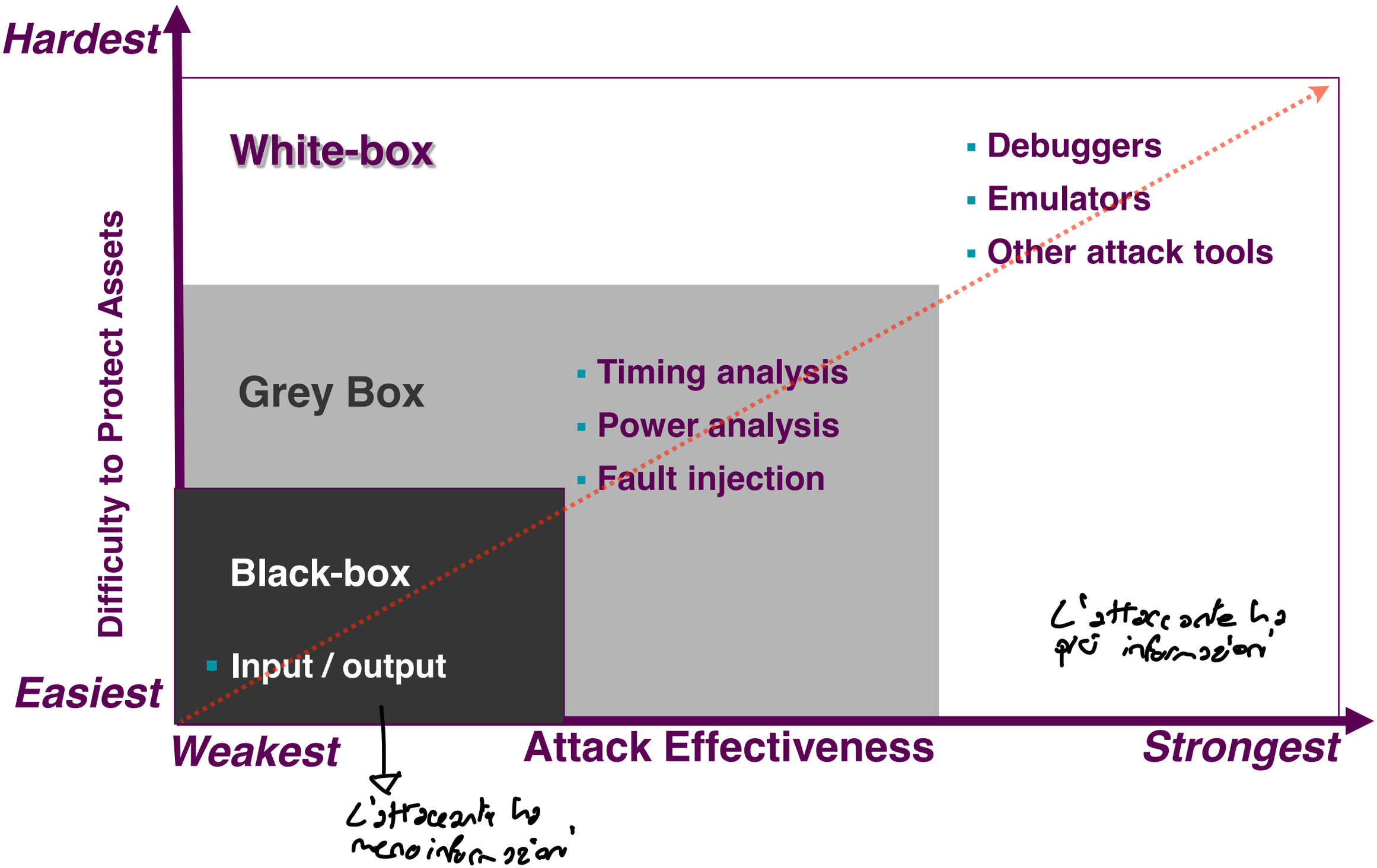
**Bob is the Attacker**



**Software**

più si schiarisce la box più la protezione è difficile

# Attacks



# New challenges for information security

**Traditional security is more about security of trusted environments**

Siamo in uno scenario
Non si the End non
ci fidiamo dell'Attacker
in cui il codice
viene eseguito

**White-Box Security**

**Dynamic Security**

**Digital Asset Protection is More About Security of Un-Trusted Environments**

siamo circondati da ambienti potenzialmente NON attendibili

# Untrusted environments reality

**Cloud Computing Environments**

Un-trusted environments are everywhere and even becoming more dominated in digital world

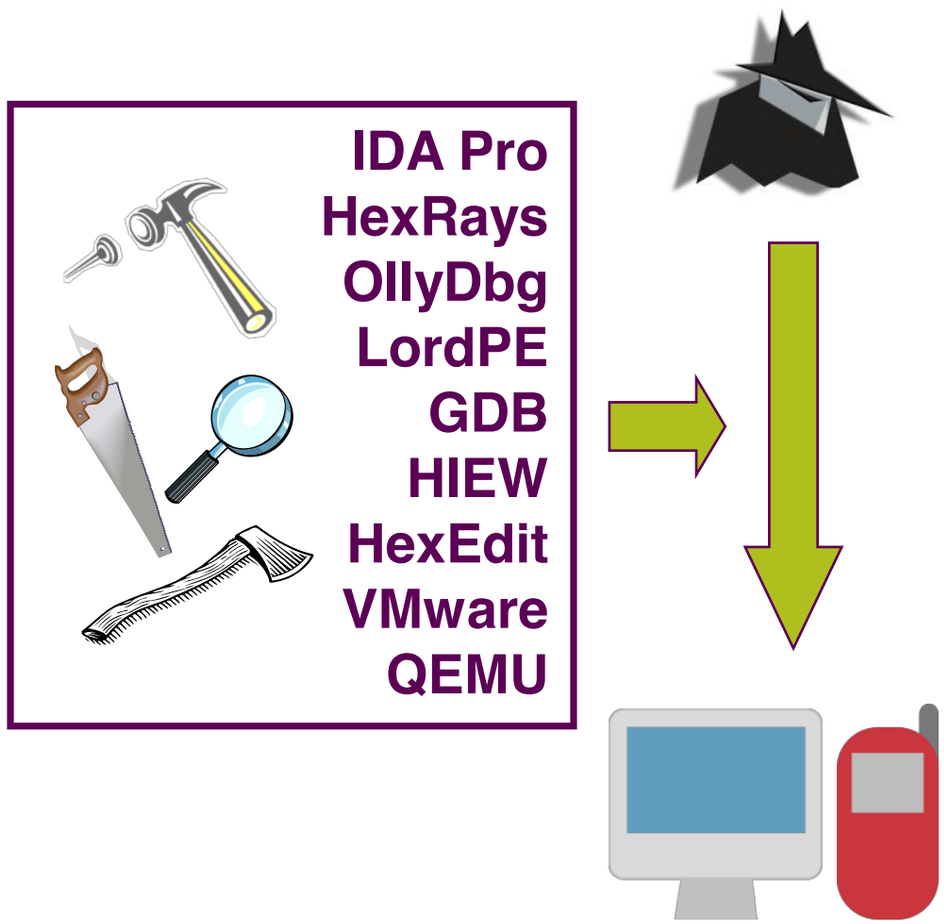
**Public Internet (HTML5)**

Persistent security on un-trusted environments is becoming #1 concern

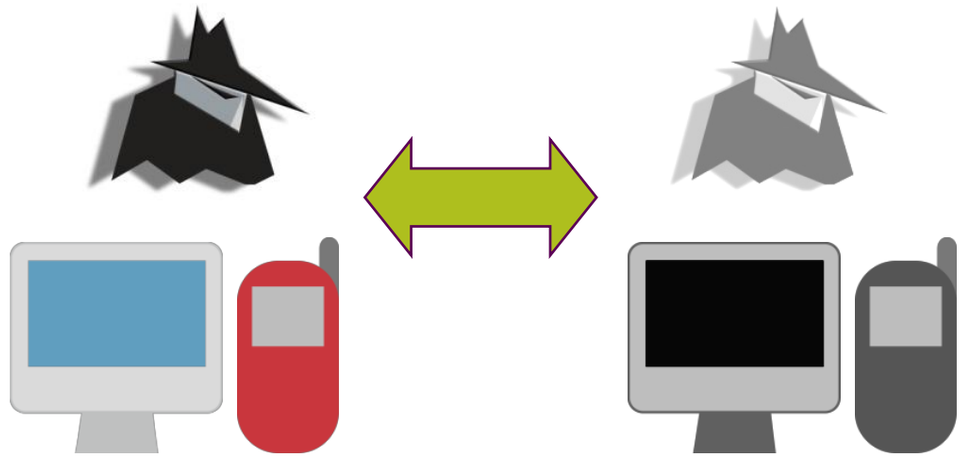

**Consumer Devices & Home Networks & Internet of Things**

# What are the threats?

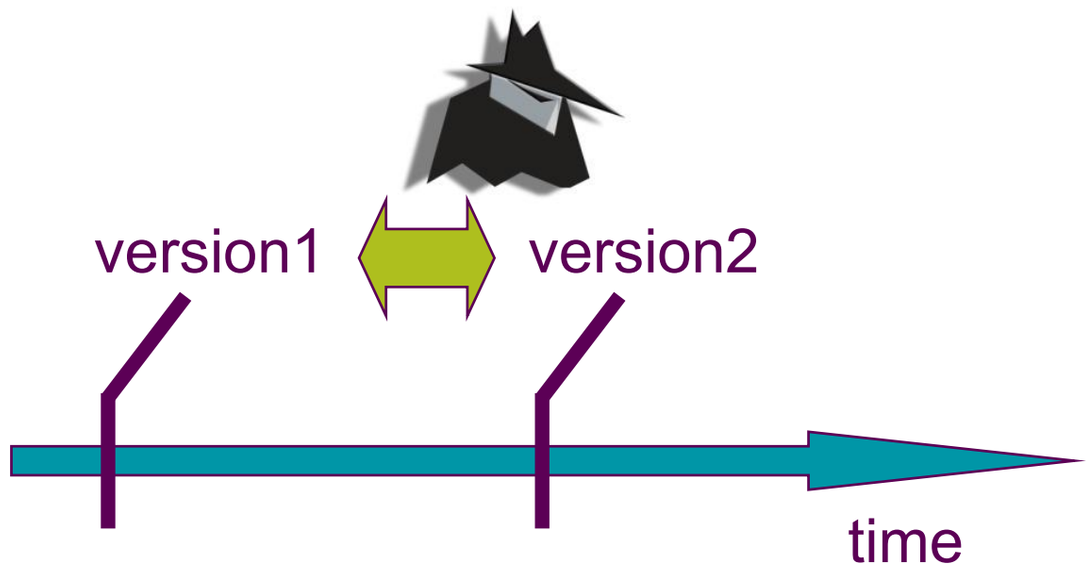
## Direct WhiteBox Attack




## Colluding Attack

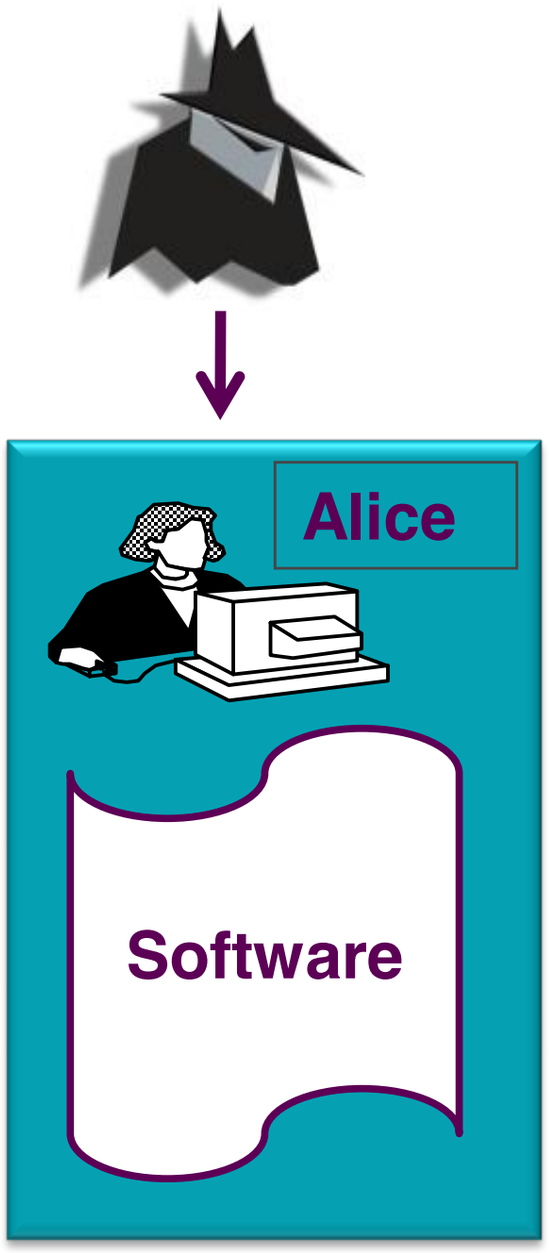


## Differential Attack



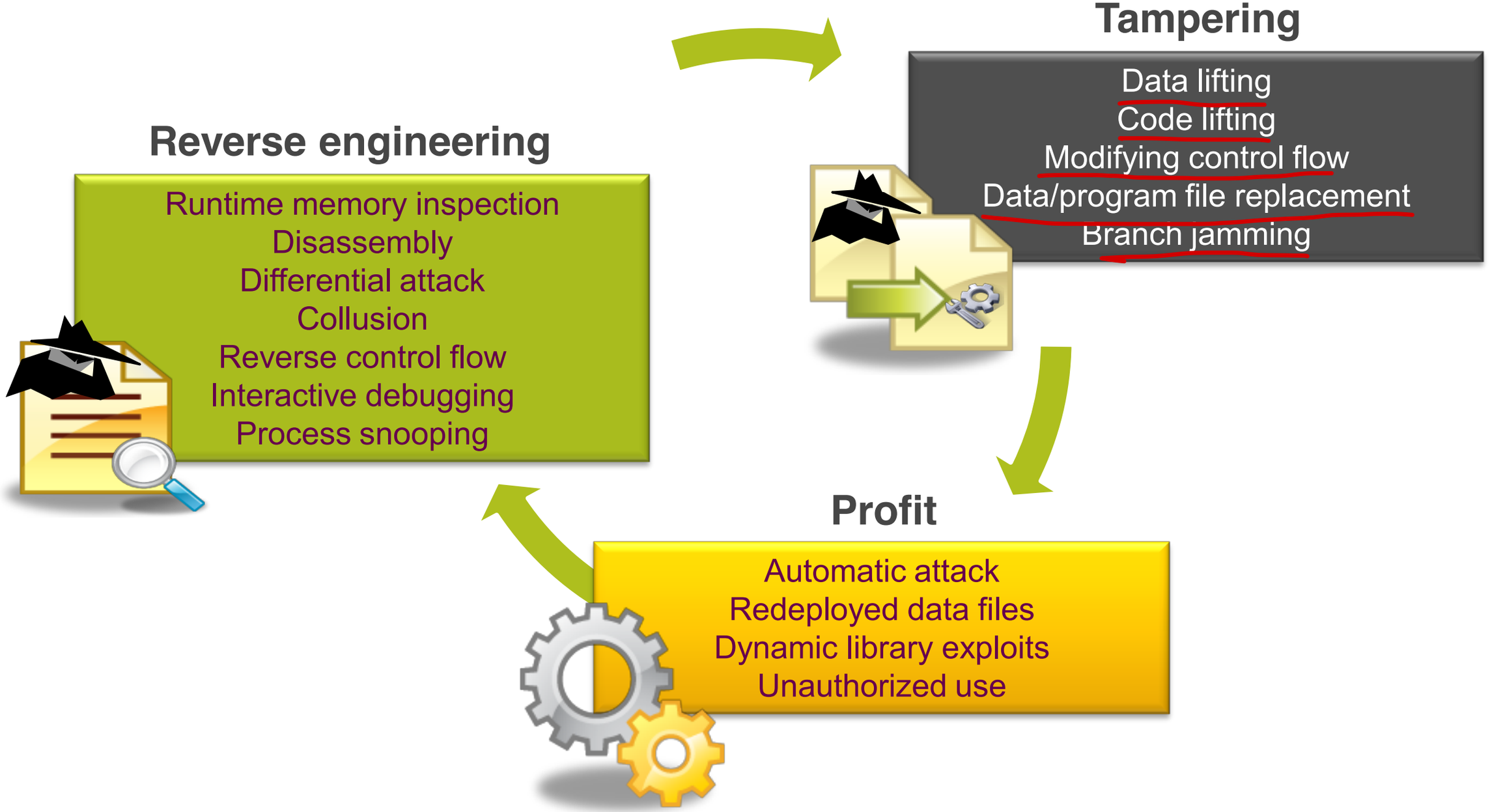
# Software Protection Challenges

↳ Necessario per rendere più sicura la protezione in ambienti non affidabili




**How to provide necessary trustworthy to application software in the Un-Trusted Environment**

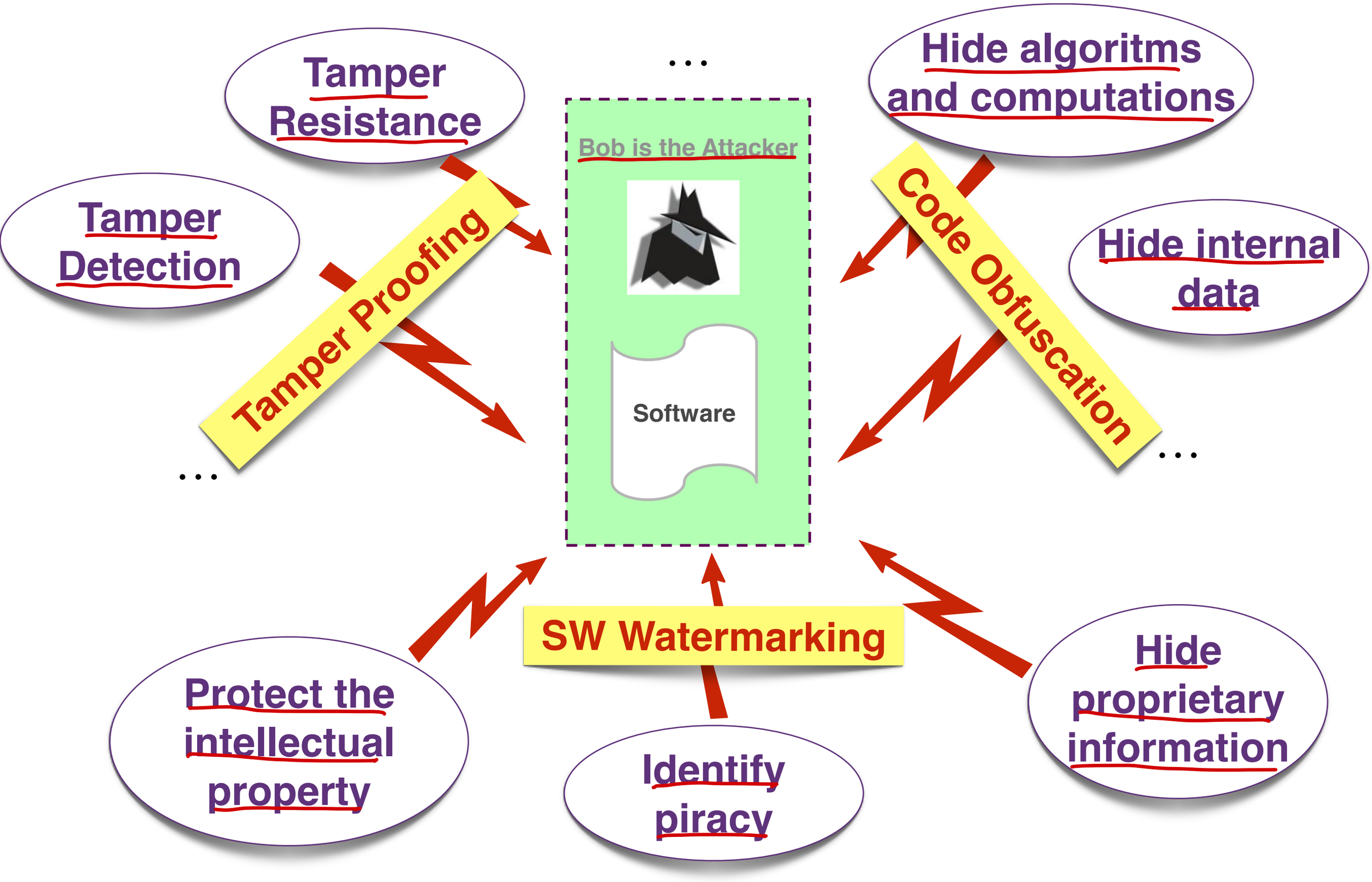

# Attacks on Software



# Key Objectives of Software Protection

- Resist static and dynamic *reverse-engineering*
- Resist *tampering* (i.e. unauthorized modifications)
- Resist *cloning* (i.e. moving software to a node it is not authorized to run on)
- Resist *spoofing* (i.e. having software use false identification, such as over a network)
- Hide both static and dynamic *secrets*, as they are created, moved and used
- Impede the production and distribution of useful "crack" programs
- Facilitate timely, intelligent *responses* to crack incidents

# Software Protection



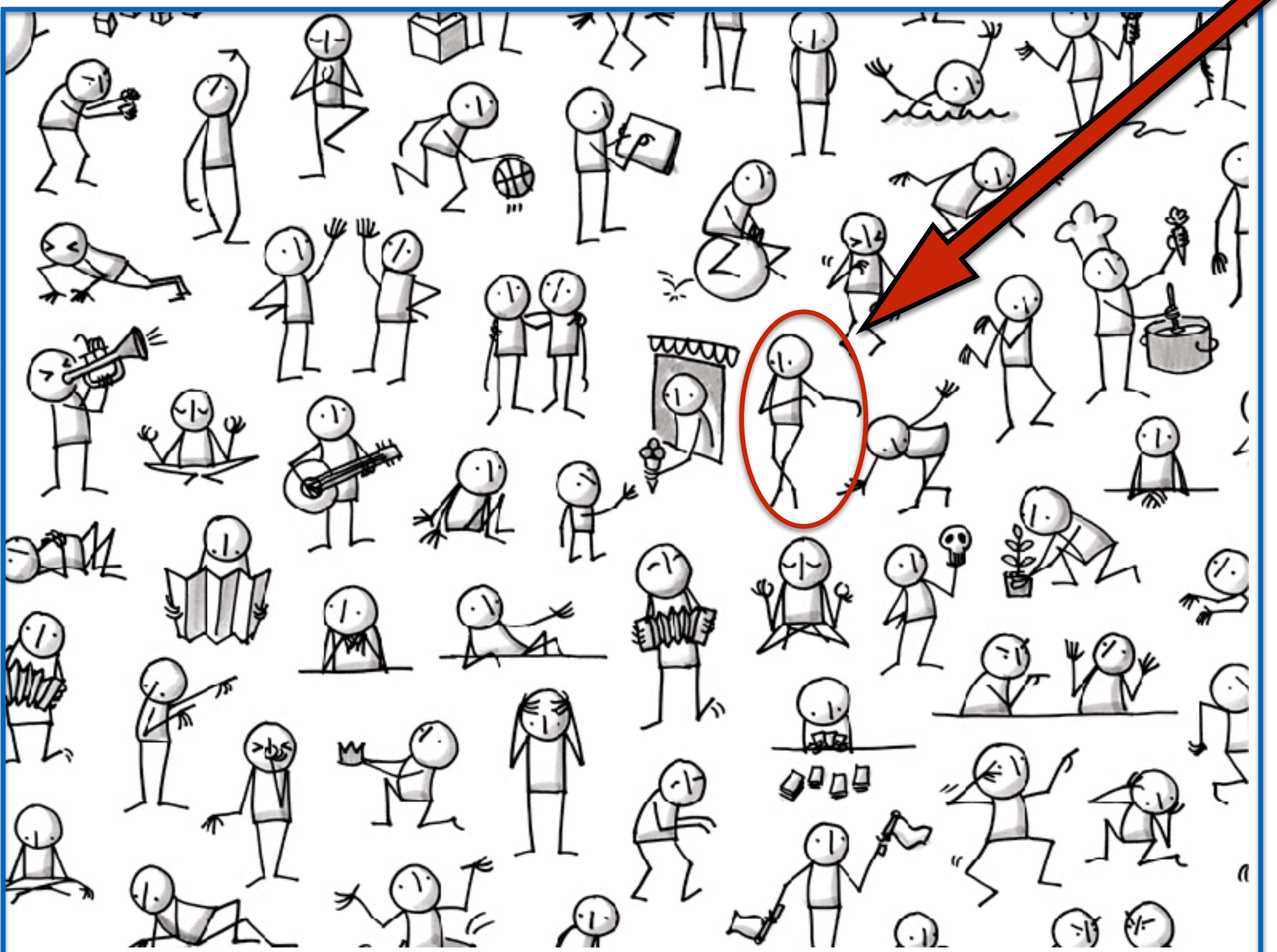
1

# Code Obfuscation

Real Code

# 1

# Code Obfuscation



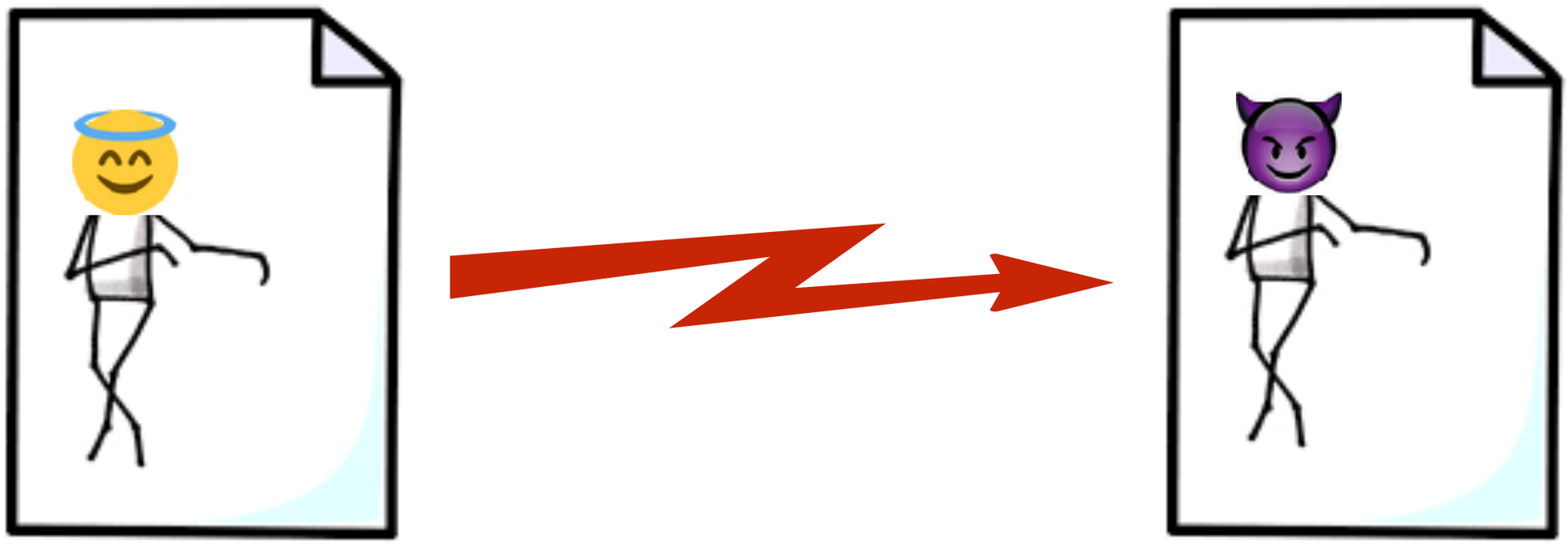
**Real Code**

Il codice deve fare le stesse azioni scritte in precedenza

**Obfuscated Code**

# 2

# Tamper Detection



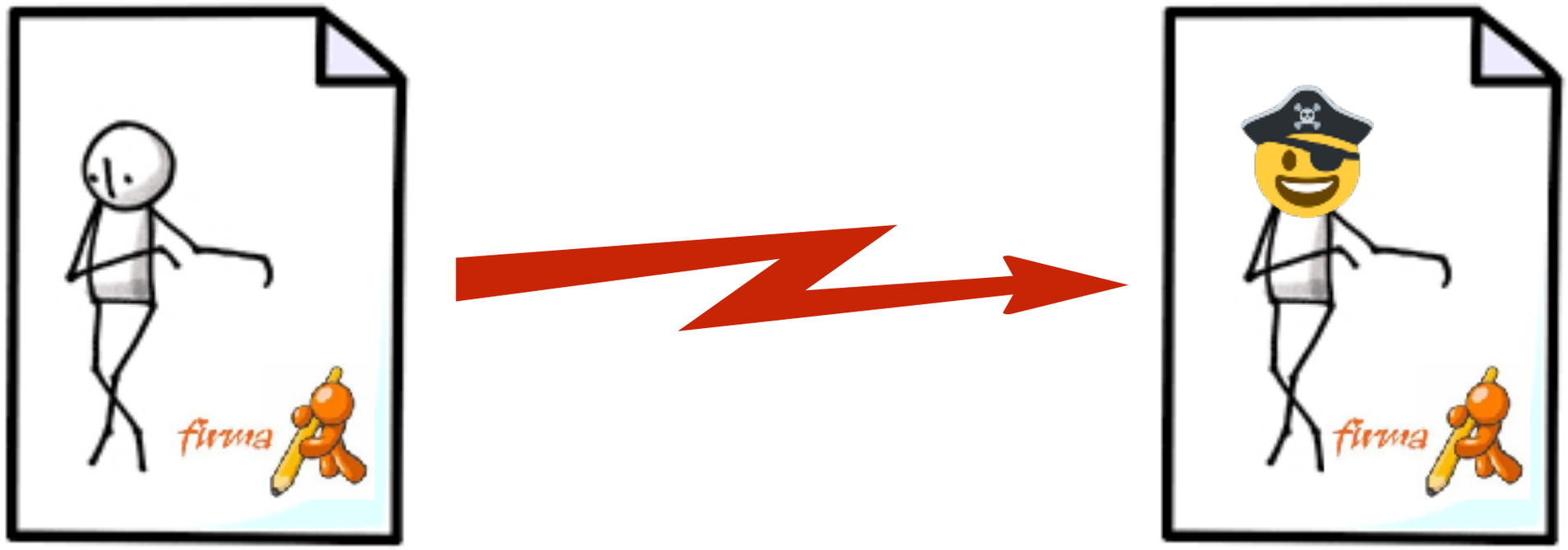
**Original Code**

**Tampered Code**

**Detect when tampering occurs and respond to attack (checksum)**

3

# SW Watermarking



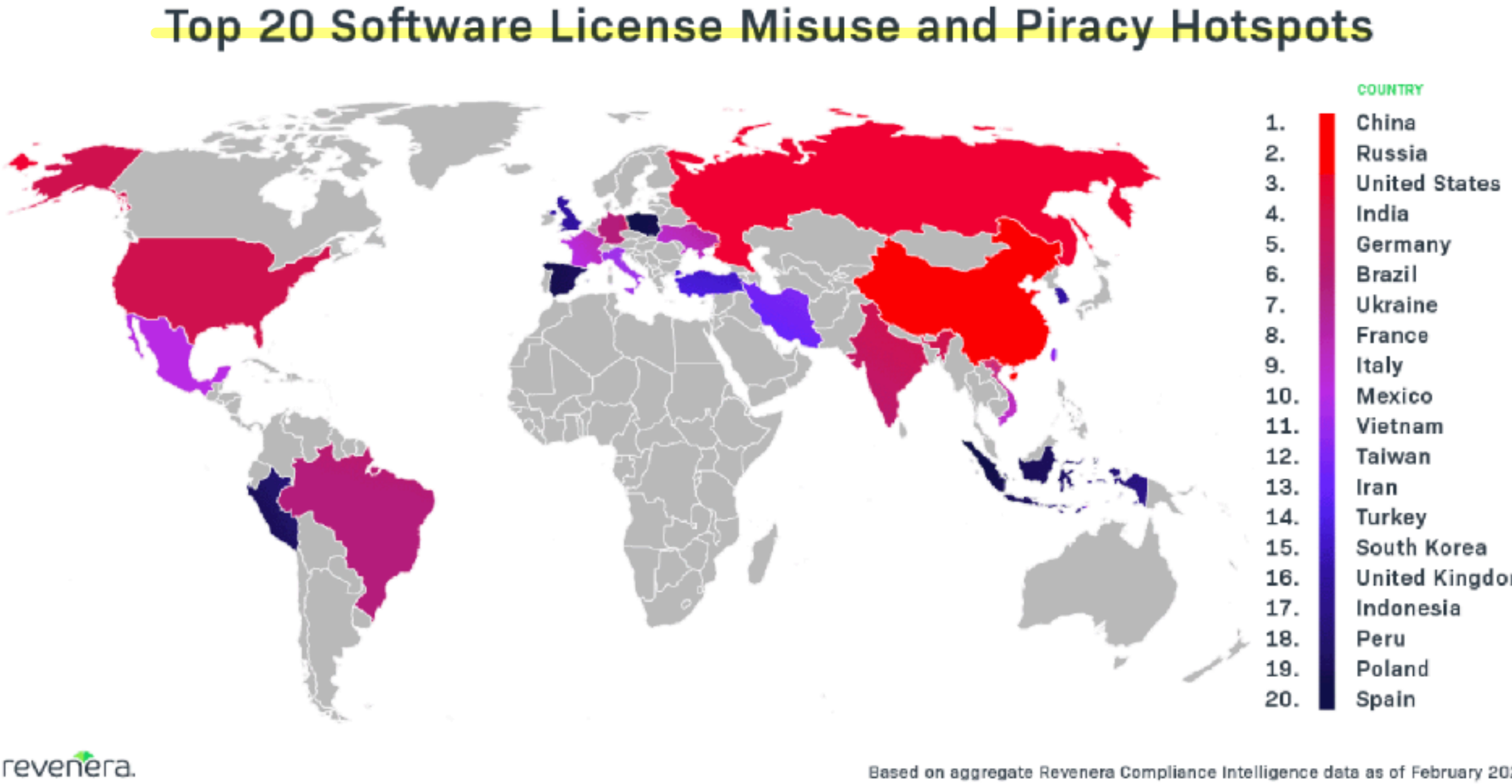

**Proprietary Code**

**Pirated Copy**

**Watermarking: Identify illegal copies**

**Fingerprint: Identify also the intellectual property violator**

# February 2022



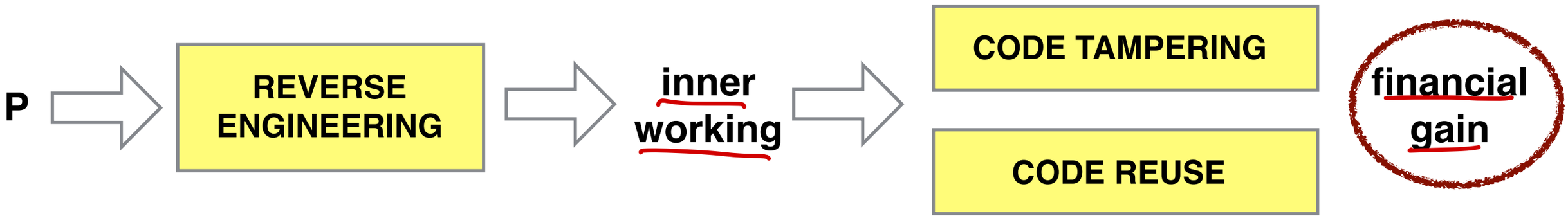

Recent Global Software Survey revealed that 37 percent of software worldwide is unlicensed and the BSA estimated that the commercial value of unlicensed software worldwide was $46.3 billion.

https://www.revenera.com/blog/software-monetization/software-piracy-stat-watch/

# SW Protection

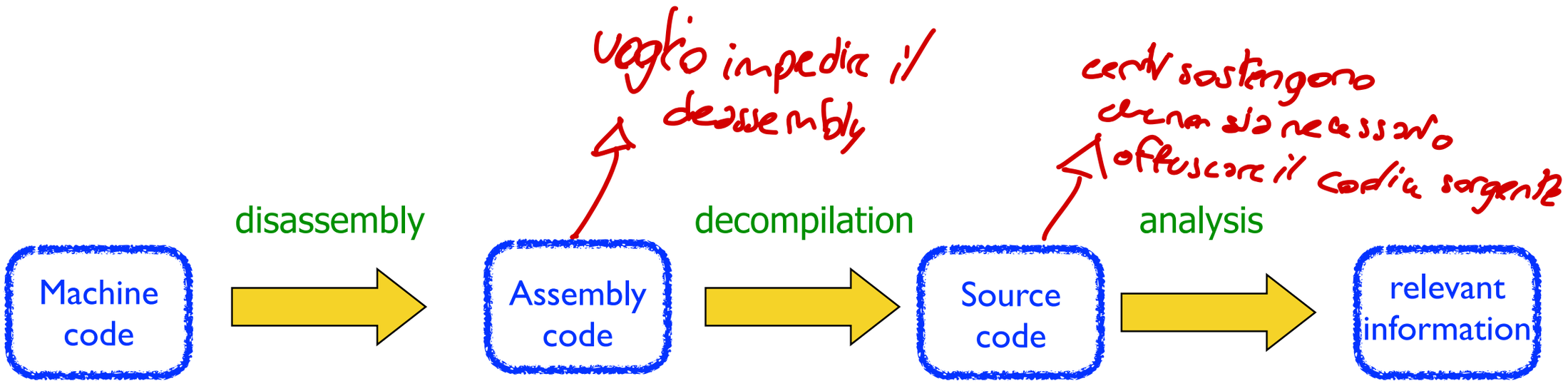
**In SW much of the know-how is located in the product itself!**

Attacks to (proprietary) programs:




It is estimated that unlicensed software is 37% of all software installed on personal computers. Frontier Economics estimated that the global value of **pirated software in 2022 will be 42$ - 95$ billion.**

# Reverse Engineering

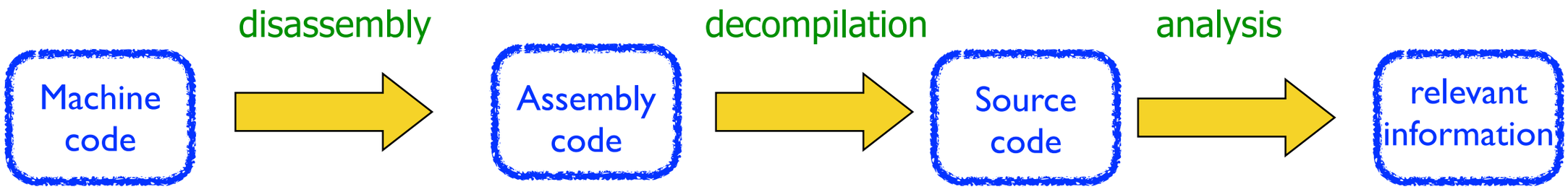



Reverse engineering typically performs static and dynamic analyses in order to understand the inner workings of a proprietary program of interest

*Given enough time, effort and determination a competent programmer can always reverse engineer any application!!!*

# Reverse Engineering

**Protection Needed at all Levels!!!!**



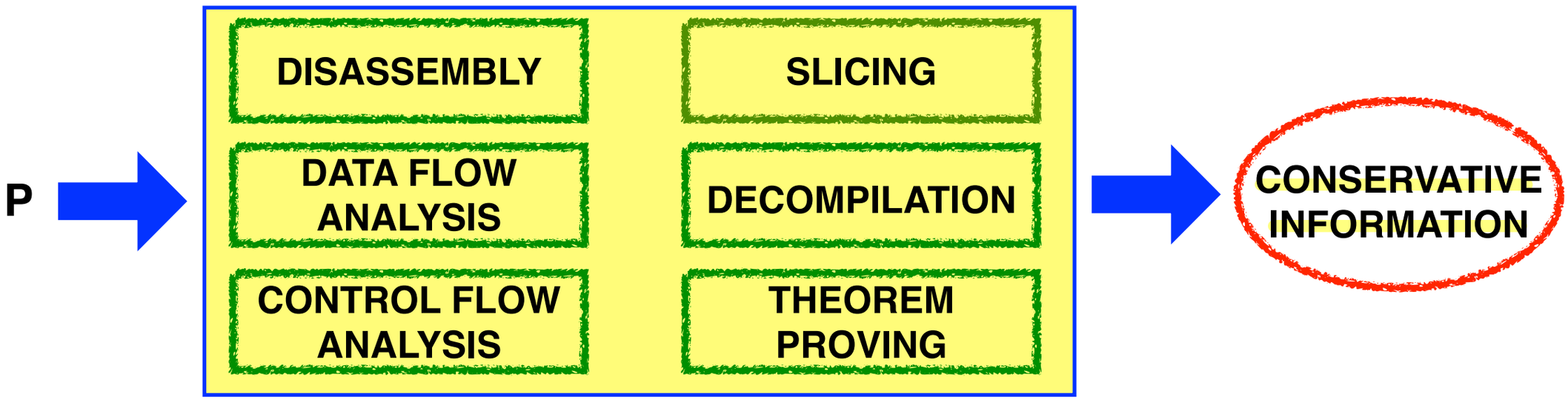

Reverse engineering typically performs static and dynamic analyses in order to understand the inner workings of a proprietary program of interest

Given enough time, effort and determination a competent programmer can always reverse engineer any application!!!

vogliamo rallentare il processo per renderlo sconveniente

# Static Analysis

Static analysis takes as input the program and analyses it without executing it



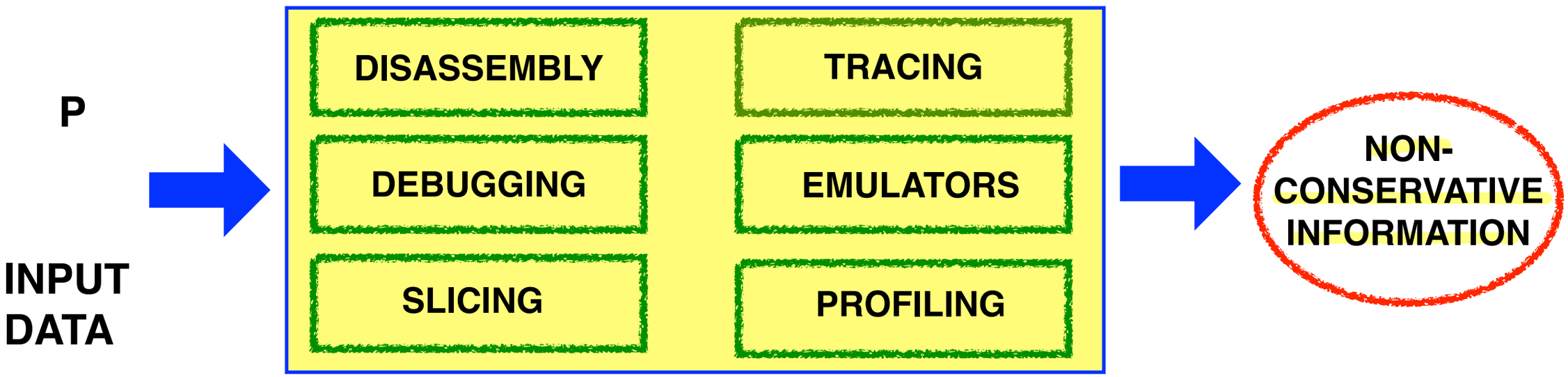
vede tutto benissimo

# Dynamic Analysis

↳ Considera alcune tracce e analizza

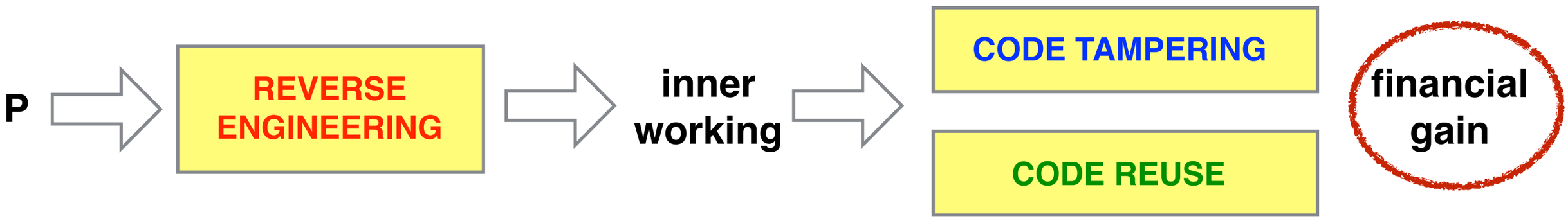
Dynamic analysis takes as input the program and a sample of input data set and analyses the program by executing it on the considered set of inputs

Offuscamento funziona bene sull'analisi statica e molto meno quella dinamica




The accuracy of the generated information depends on the completeness of the input data (coverage problem)

# Protection Techniques



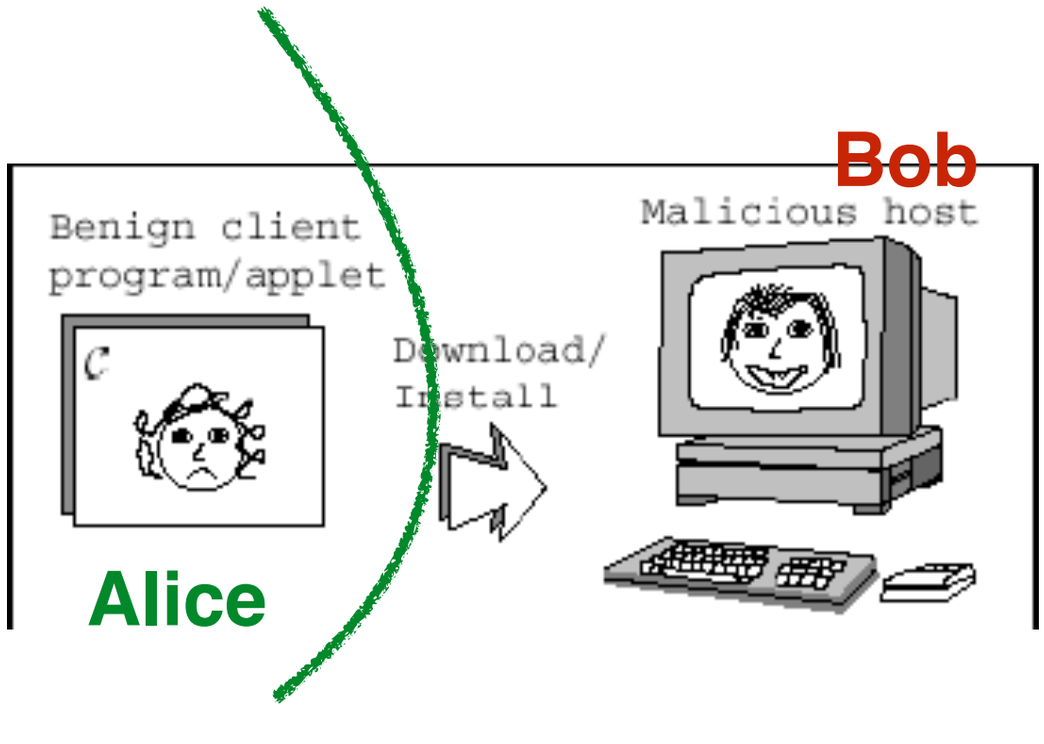

- Legal Measures:
  - ✓ copyright protects the form but not the idea
  - ✓ patents protect also the idea
  - ✓ licenses establish client rights and limitations
  - ✓ lows are different from state to state
  - ✓ costly and time consuming
- Code Obfuscation: obstructs reverse engineering
- SW Watermarking: insert a mark against theft
- TamperProofing: detect integrity violation or unlegal modifications

brevetti

# Who is interested in SW protection?

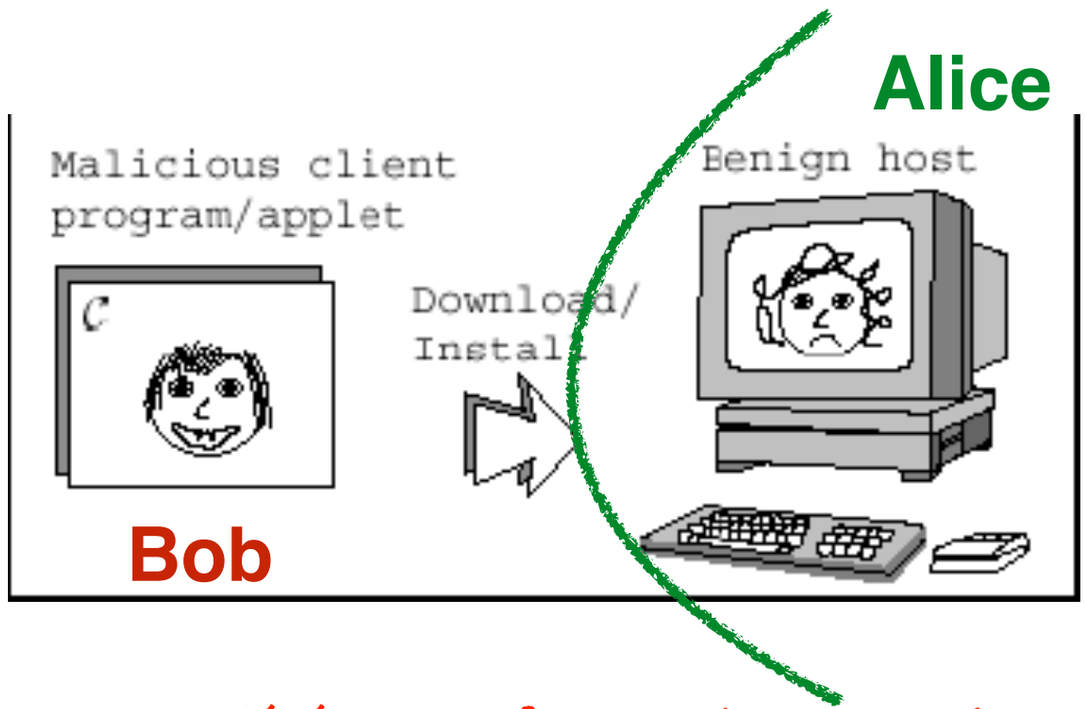
- **Companies**: Microsoft, Apple, Intel, Arxan, Cloakware, Irdeto, Skype…..
- **Academia** researchers have approached the software protection problem from a variety of angles (crypto, recursion theory, programming languages and compilers…)
- **Military** around 2008 the US DoD (Department of Defense) initiated a program to investigate technologies such as obfuscation and tamper proofing in order to protect sensitive weapons system software
- **Bad guys** malware writers recur to obfuscation to avoid automatic detection form anti-malware tools

# Malicious Host vs Malicious Software




A *malicious host* may try to gain privileged access to an application or to disclose secret information, usually in order to violate the intellectual property of programs.

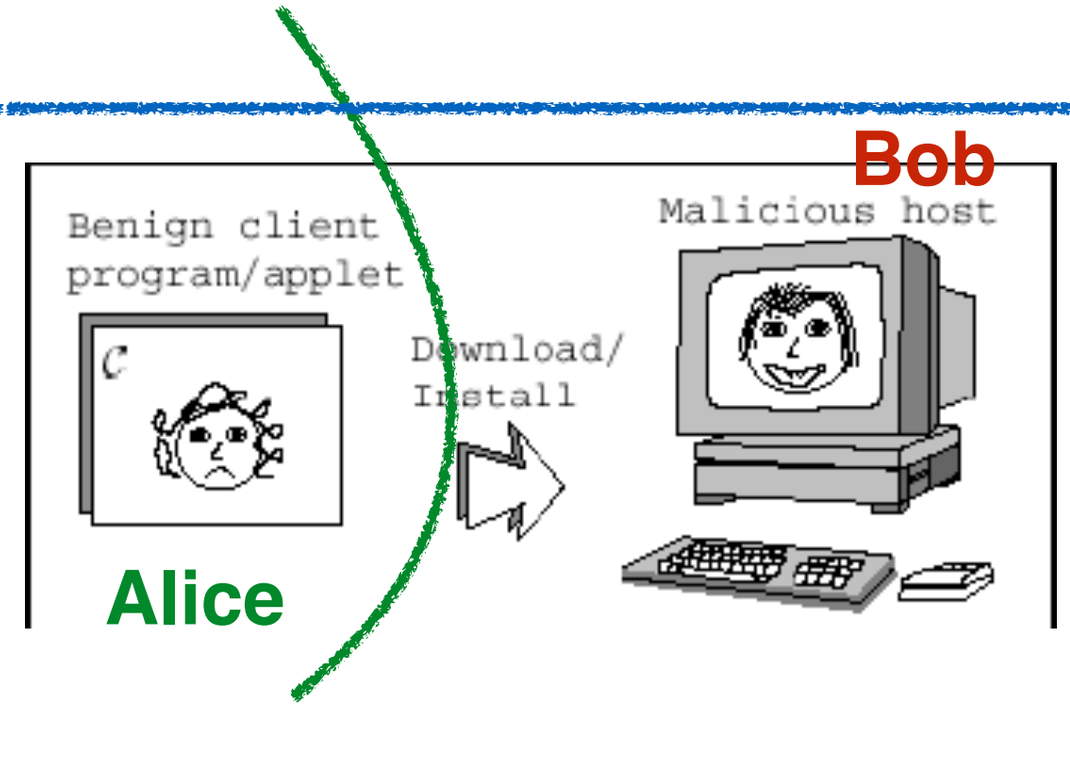
Alice uses software protection techniques to defend her product




A *malicious software (malware)* is a program with a malicious intent that propagates with no user consent and produces some damage.

Alice adds *protective layers* to prevent someone from entering, to detect that someone has entered, or to stop someone from doing harm once they've entered

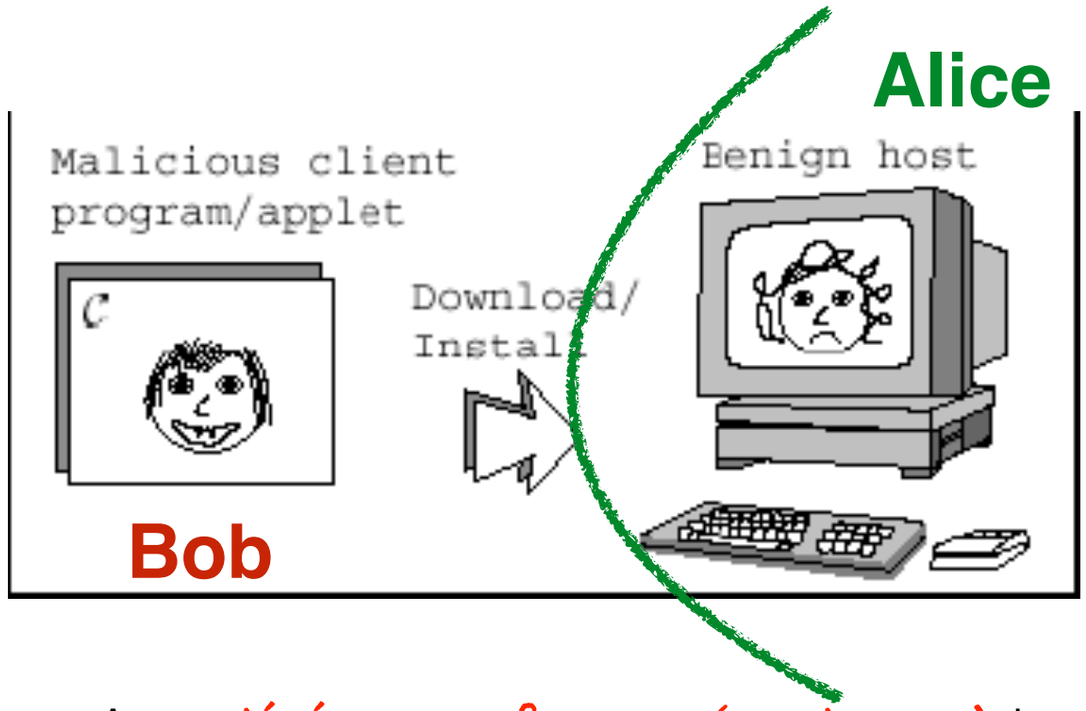
# Malicious Host vs Malicious Software




A *malicious host* may try to gain privileged access to an application or to disclose secret information, usually in order to violate the intellectual property of programs.

Alice uses software protection techniques to defend her product

**Software Protection**



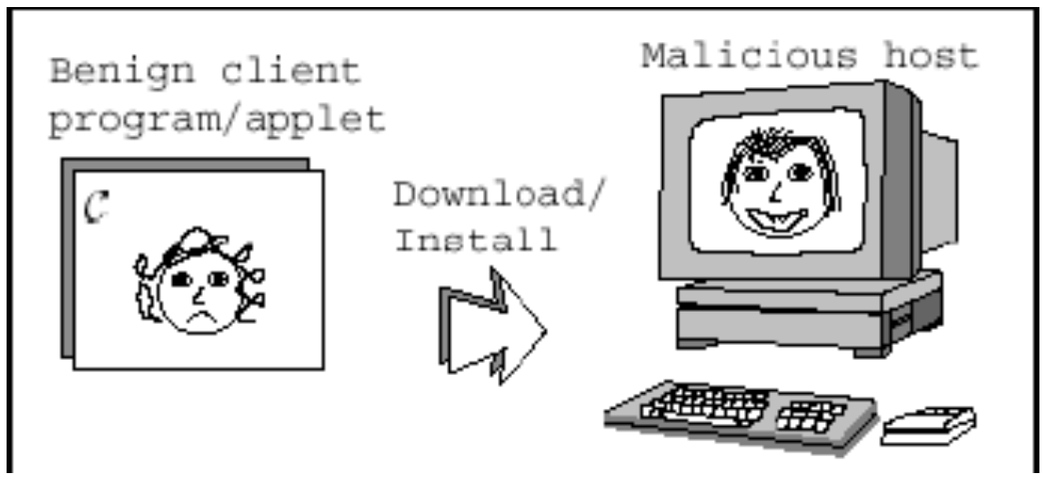

A *malicious software (malware)* is a program with a malicious intent that propagates with no user consent and produces some damage.

Alice adds *protective layers* to prevent someone from entering, to detect that someone has entered, or to stop someone from doing harm once they've entered

# Software Protection

Consider a benign developer Alice that sells a benign program to Bob to run. Assume that the program contains some secrets $S$ and that Bob can get some economic advantage over Alice by extracting such secret.

Cryptography, namely encrypting the code does not work in this case because Bob needs to execute the program at hence, at some point, it must be in clear text.




There is no limit to what Bob can do on the program of Alice once he has access to it

# Software Protection

Software protection has a lot in common with Steganography, the branch of cryptography that studies how to transfer a secret stealthily (prisoners' problem).

In a typical defense scenario Alice:

✓ adds tamper-protection to prevent Bob from modifying it

✓ adds confusion to her code to make it more difficult for Bob to analyze

✓ marks the code to assert her intellectual property rights

# Obstruct Reverse Engineering

Technical protections: Obstruct the reverse engineering process

- ✓ HW devices
- ✓ Code encryption
- ✓ (Partial) Serves side execution
- ✓ Obfuscation

# HW Devices

Alice protects her code by relating its execution to the presence of certain *hardware features,* when the dedicated hardware is not present the

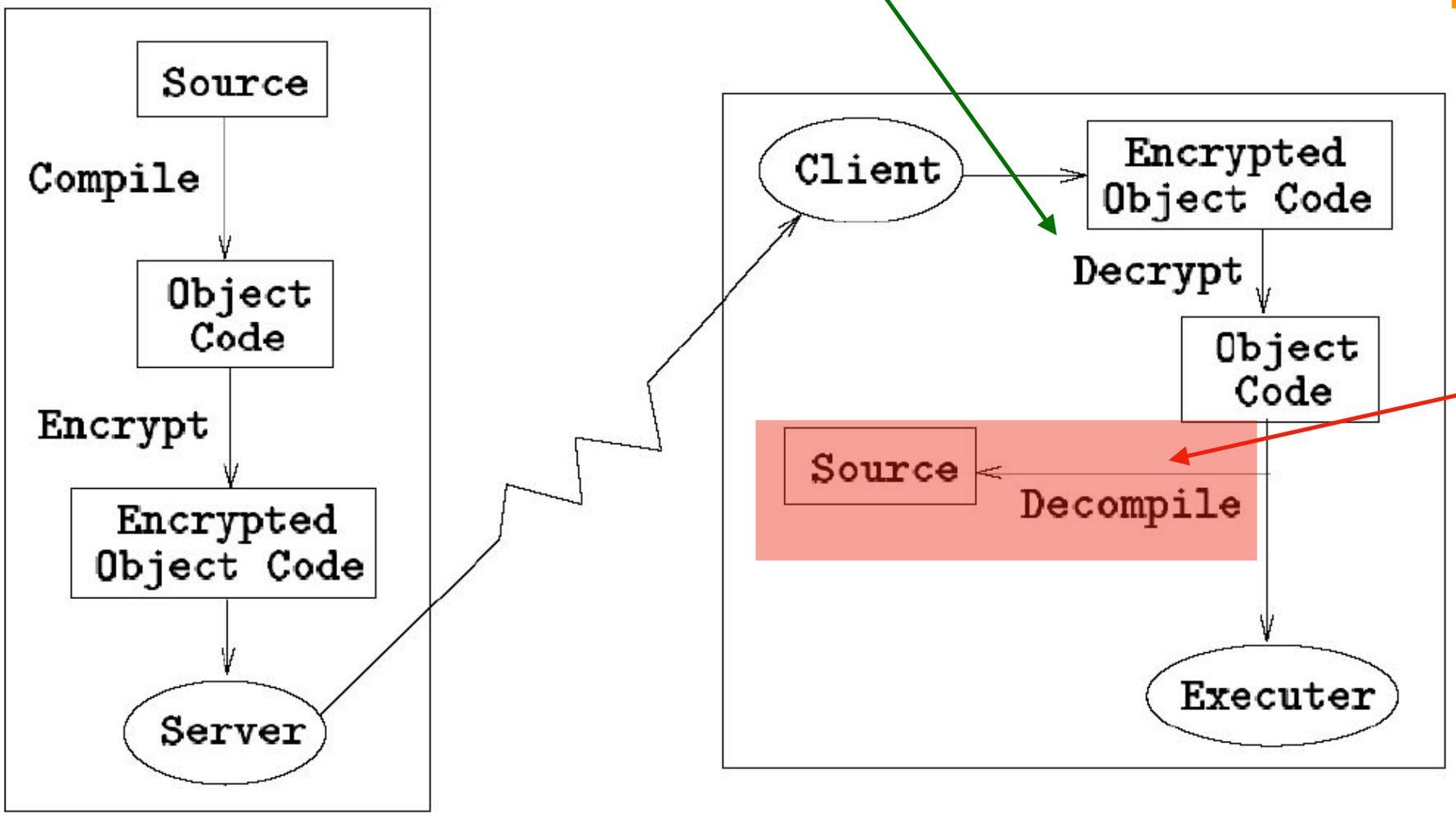
*Application refuses to run or runs in a restricted mode*

Typical example: **dongle** -- small hardware device that plugs into the serial or USB port on a computer

These physical devices are not convenient in a mobile code environment since they *limit portability*

# Encryption

**Decryption keys hold via dongles or Internet**



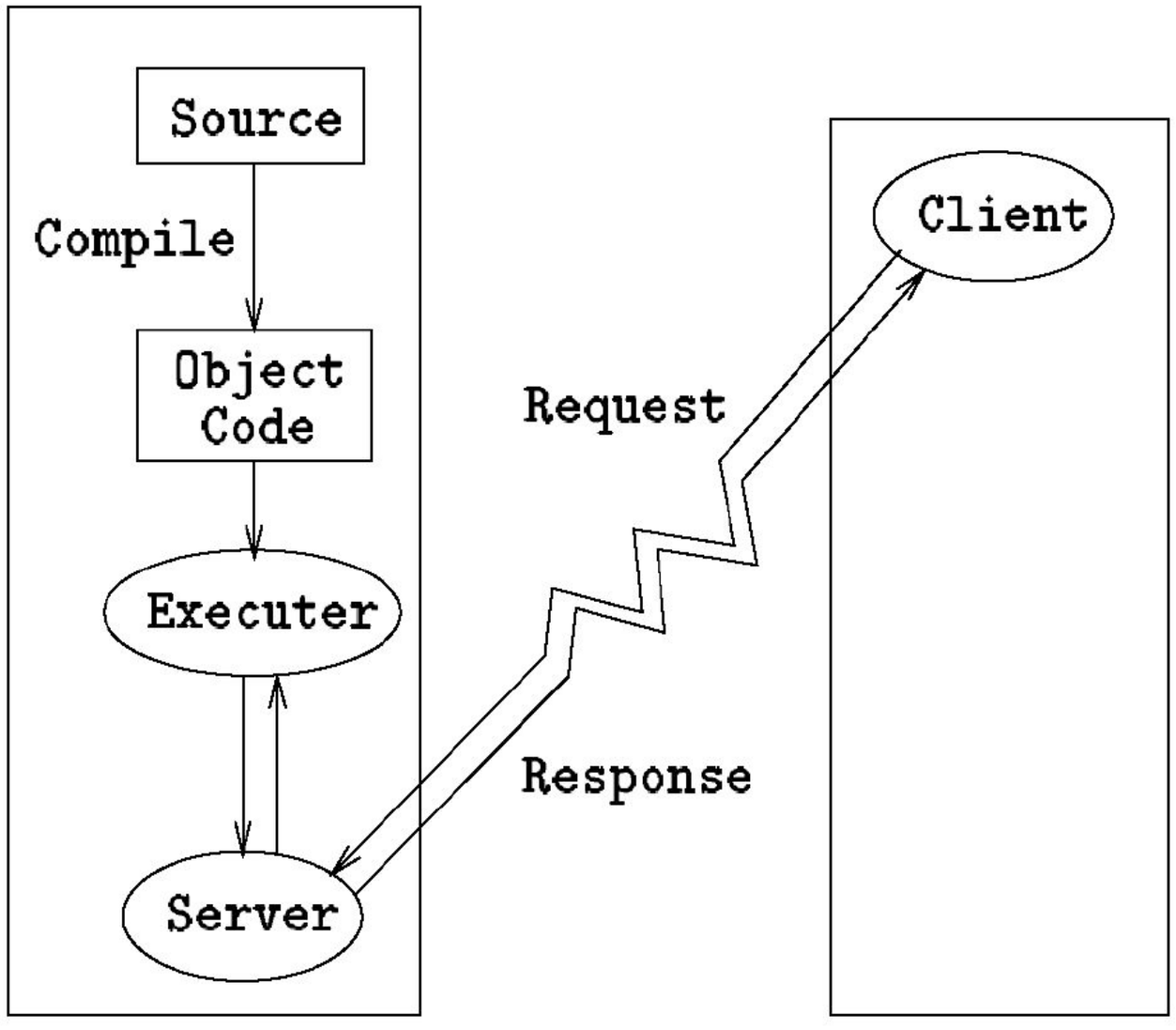

**Low security!!**

**MATE Attack**

**Code eventually appears clear in memory!!**

*Limits portability*

# Remote Execution




**Remote Execution**

**Highly secure!!**

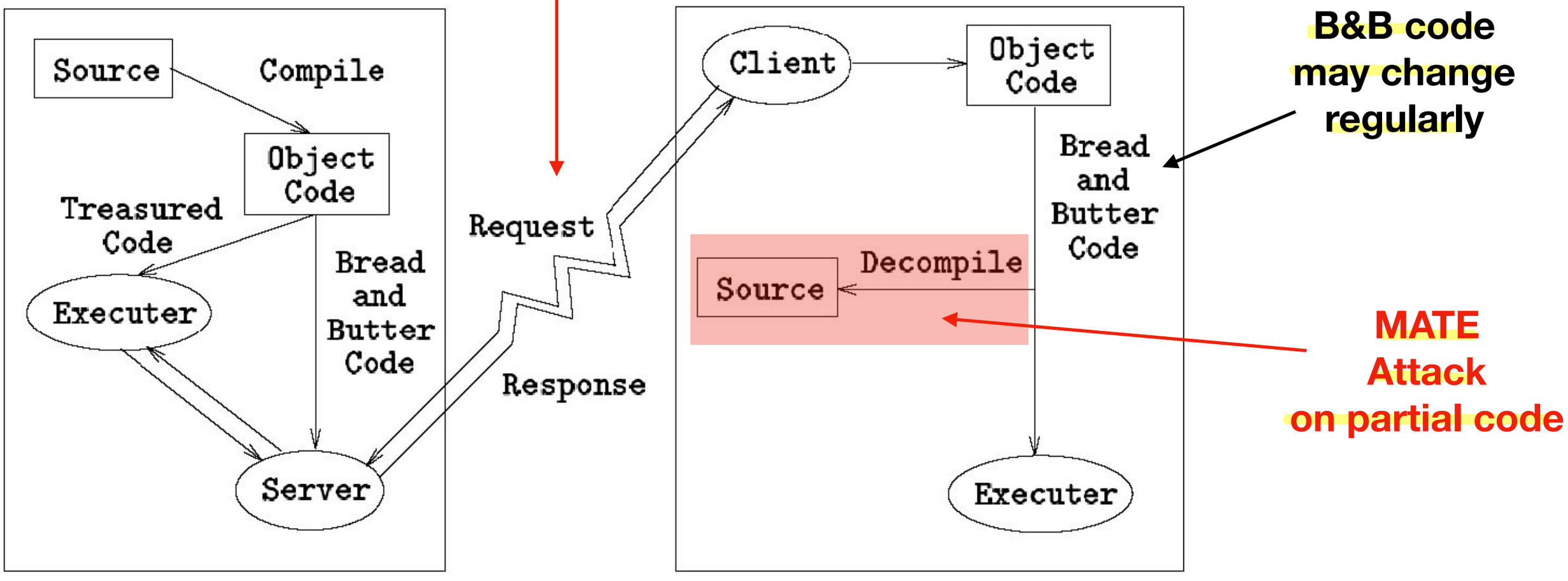
- **No code disclosure**
- **All secrets in house**

- **Requires connection client-server**
- **High bandwidth**

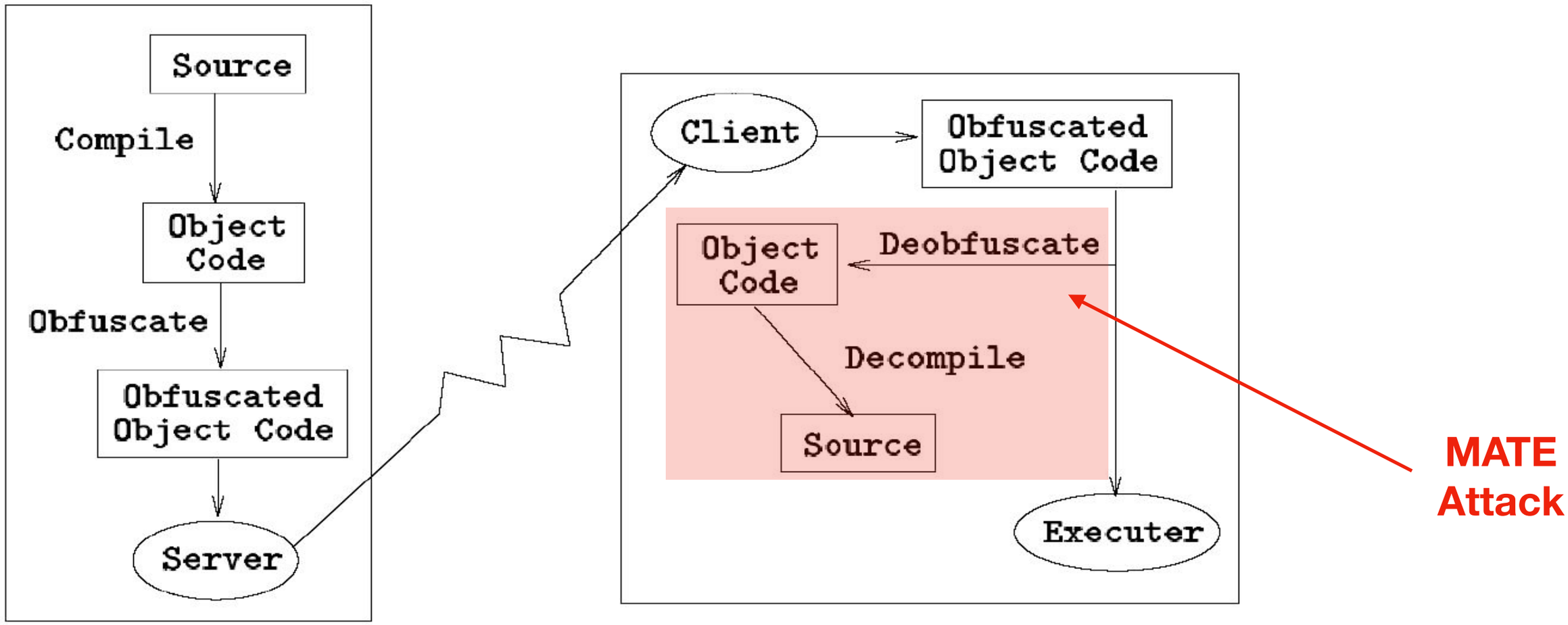
# Partial Remote Execution

**Continuous communication**

**Good security!!**



# Software Protection: Full Code Deployed




**with low diversification**

**High Risk!!**

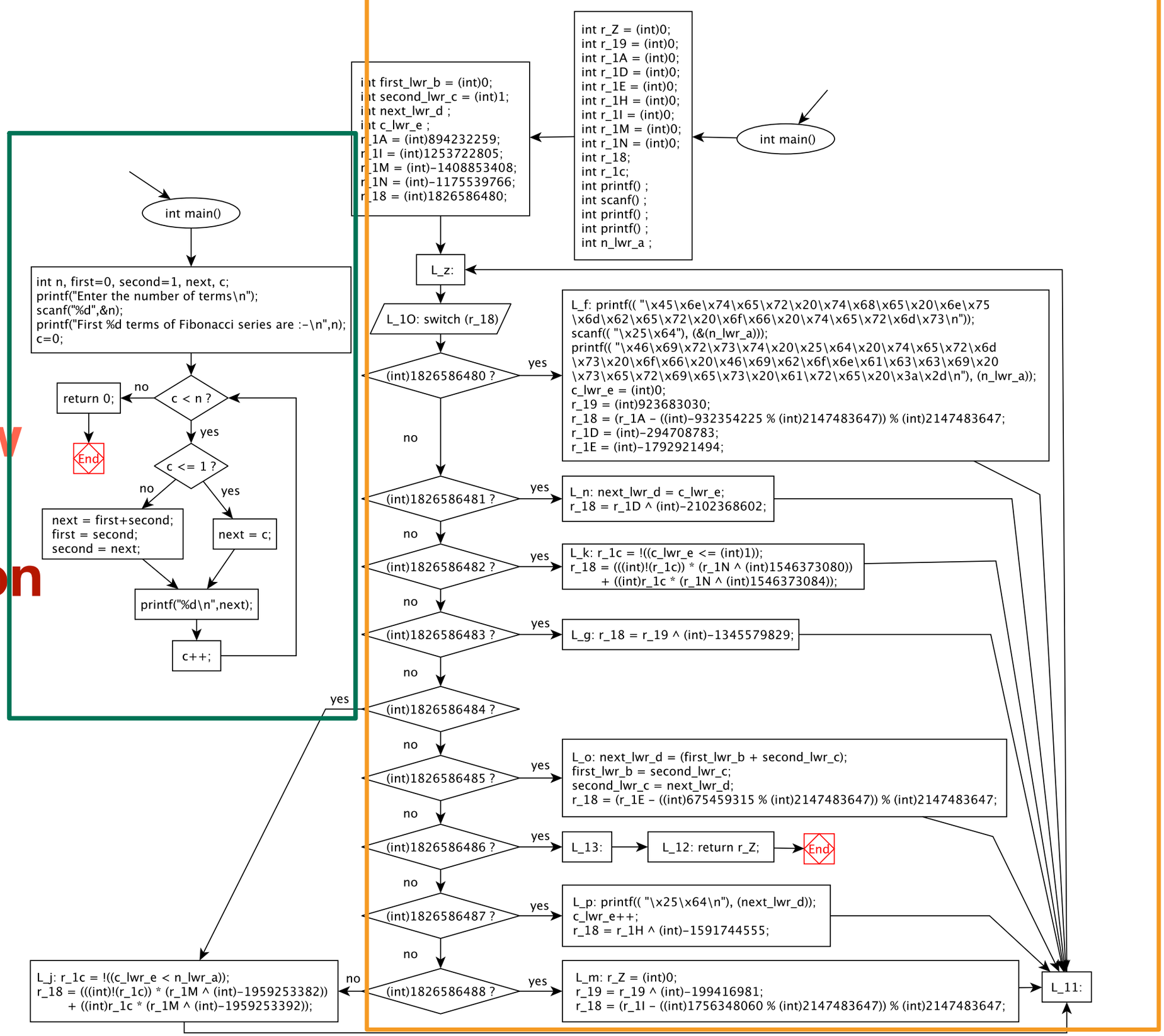
# Possible Solutions

- ✓ **Continuous diversification by obfuscation**
- ✓ **Hybrid solutions**
  - ✓ Partial execution
- ✓ **Combination with other protection techniques**
  - ✓ watermarking/fingerprinting
  - ✓ tamper proofing

# Code Obfuscation

## Methods

- Renaming
- Encryption
- Control flow
- Data flow
- Virtualisation



# Code Obfuscation

## Method

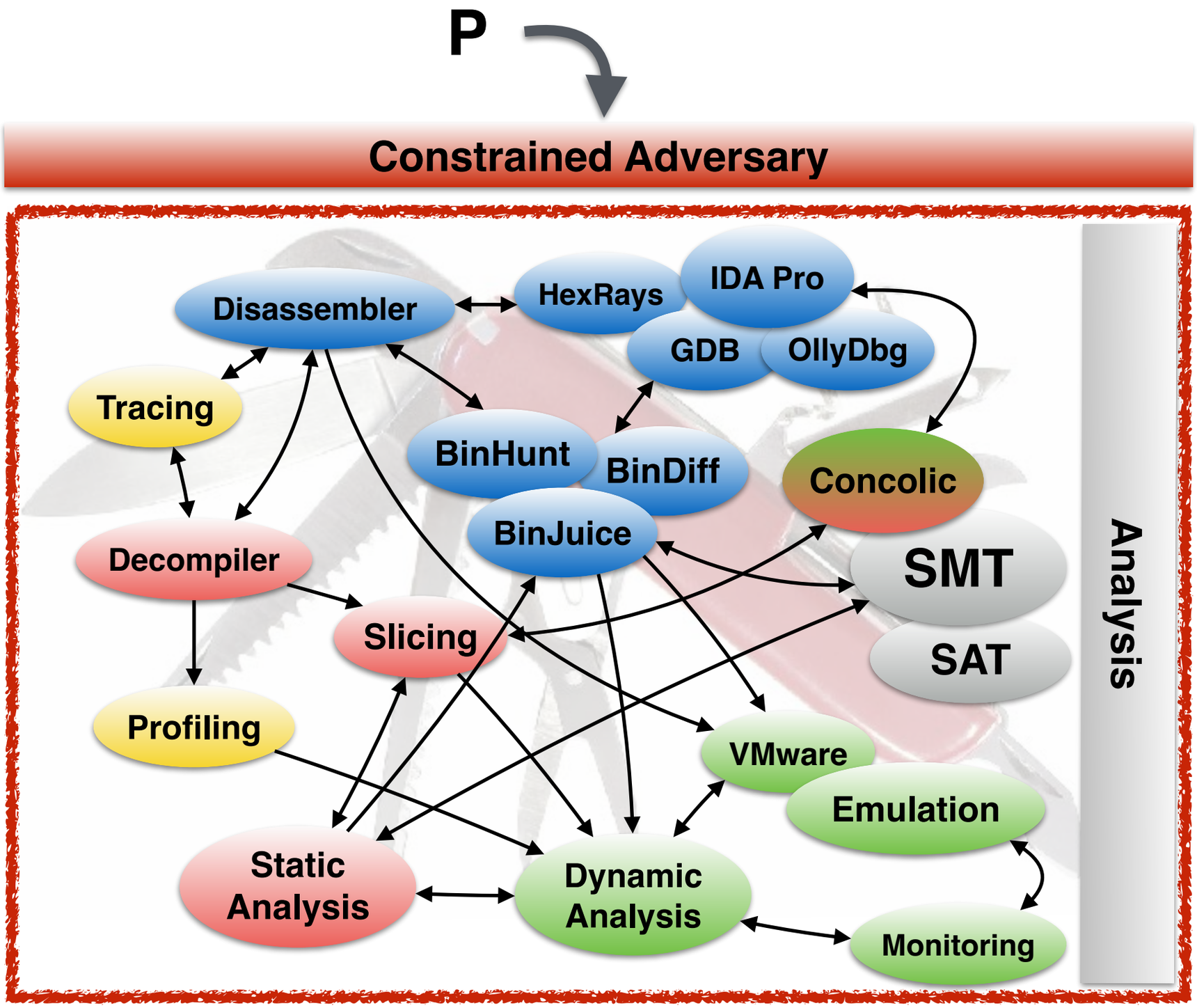
- Renaming
- Encryption
- Control flow
- Data flow
- Virtualisation

Can defeat automatic analysis fixed tool

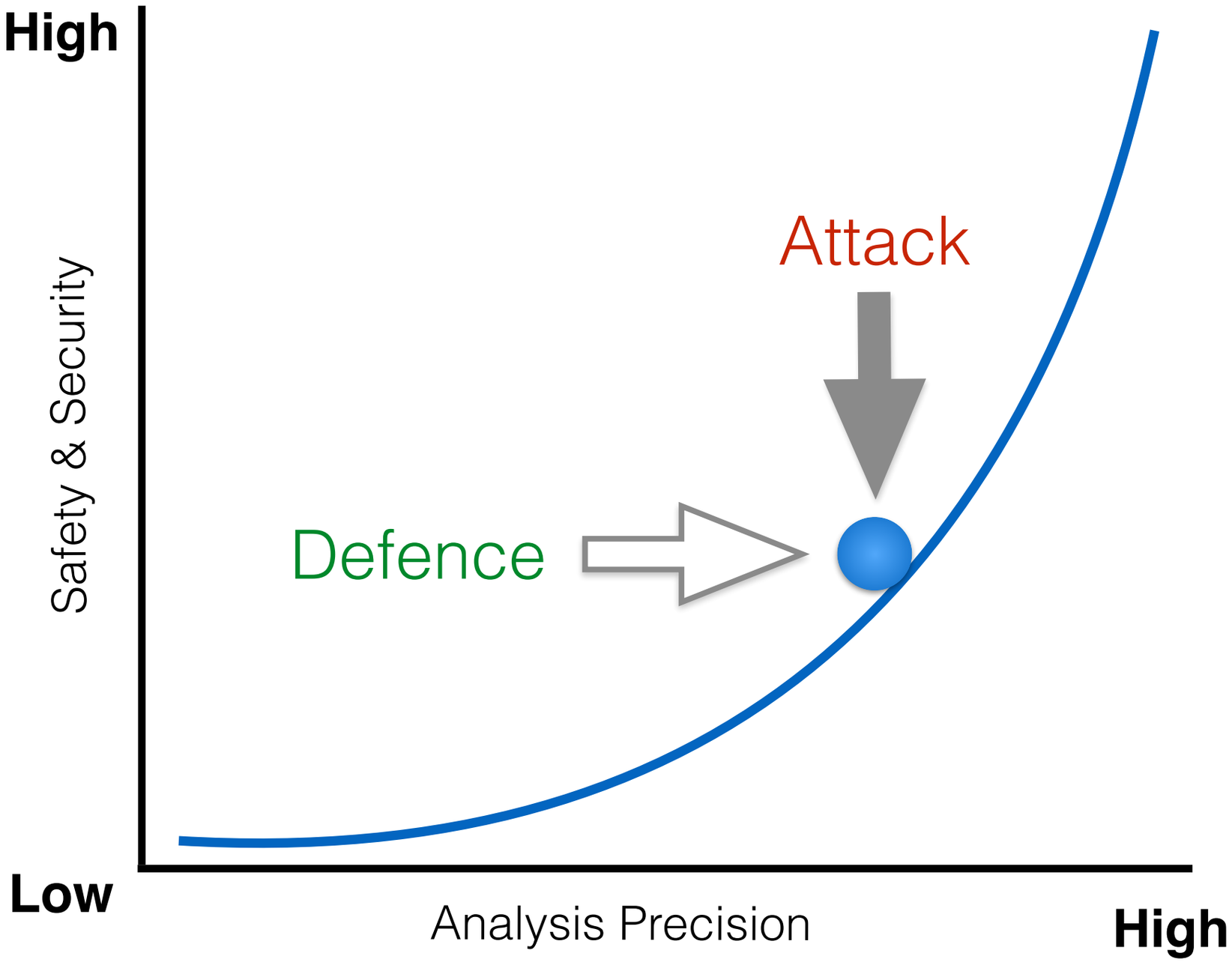
Cannot defeat all attacks!!!

*Make human analysis harder!*

# Understanding is Interpreting

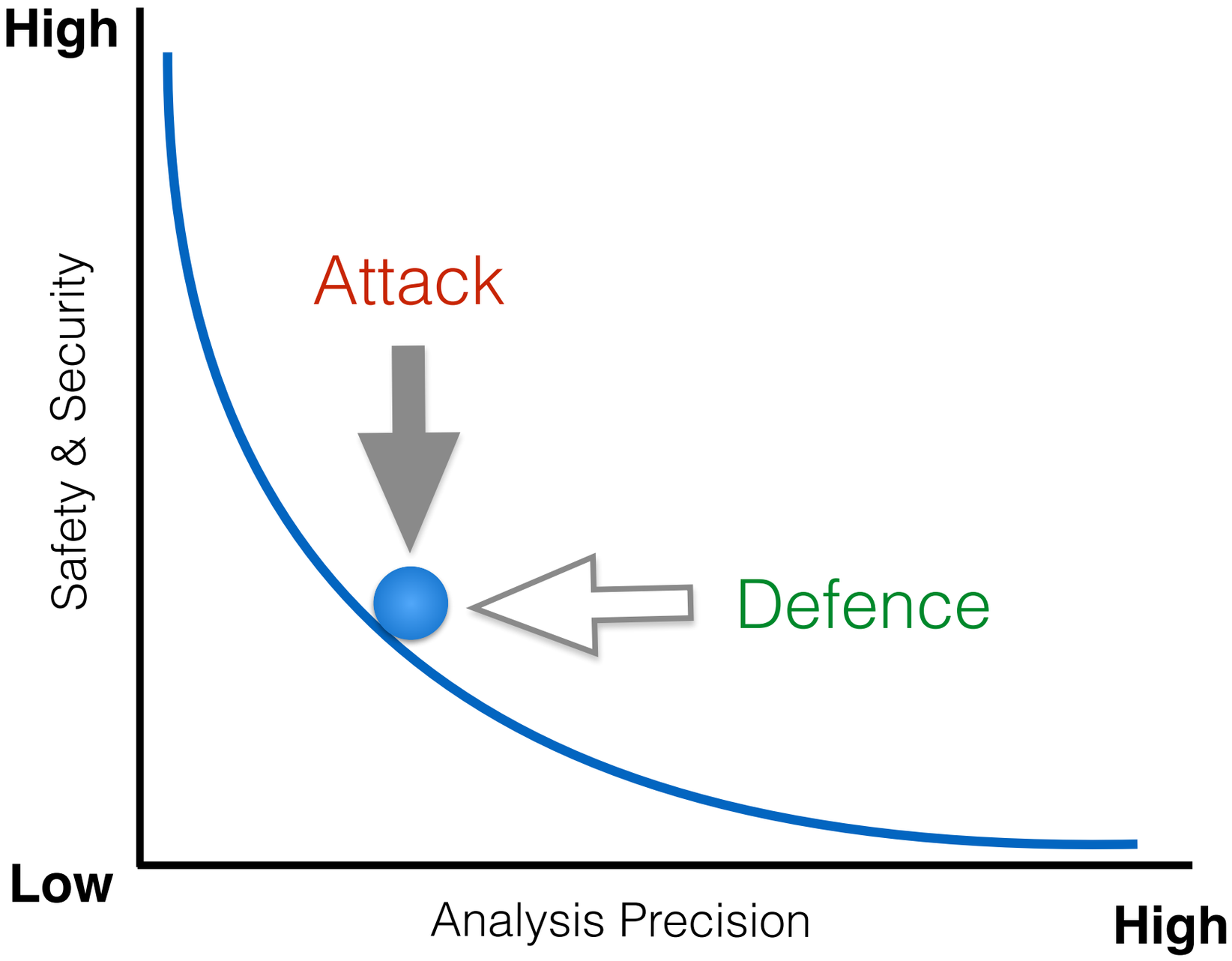



# Code Debugging



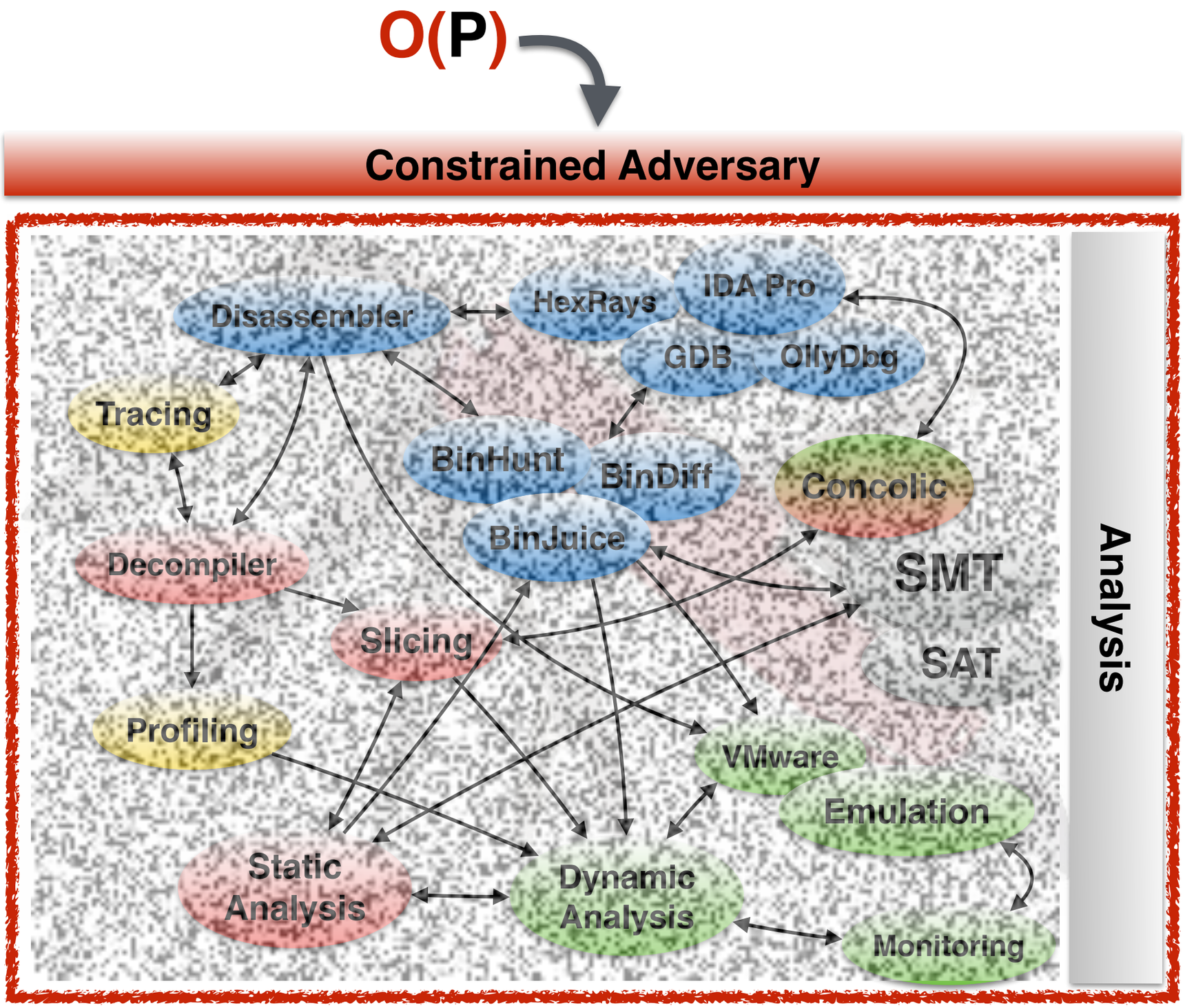

The attacker exploits bugs

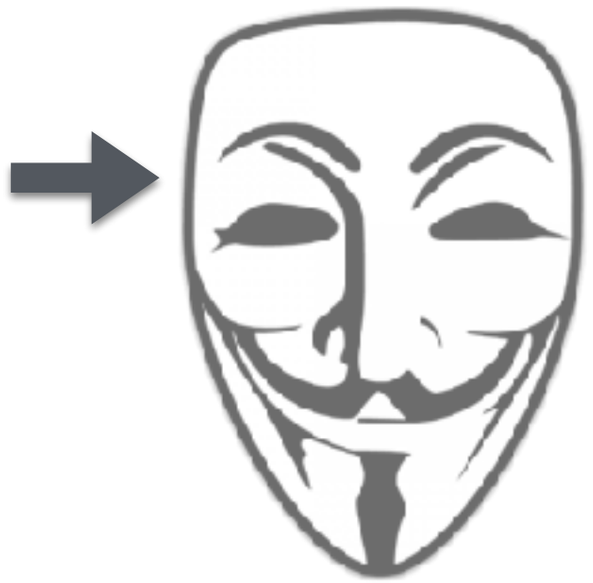
# Software Protection

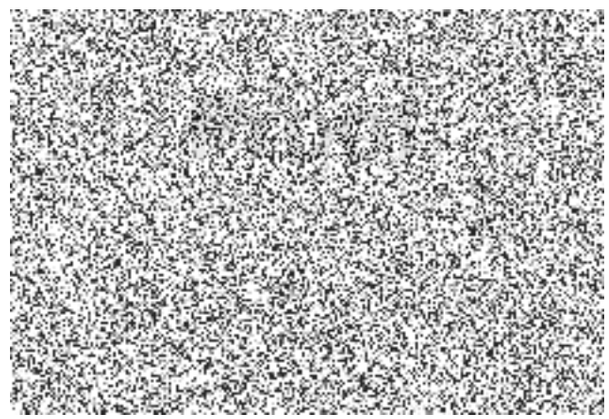



The attacker reverse engineer code

# Protecting is obscuring Interpreters

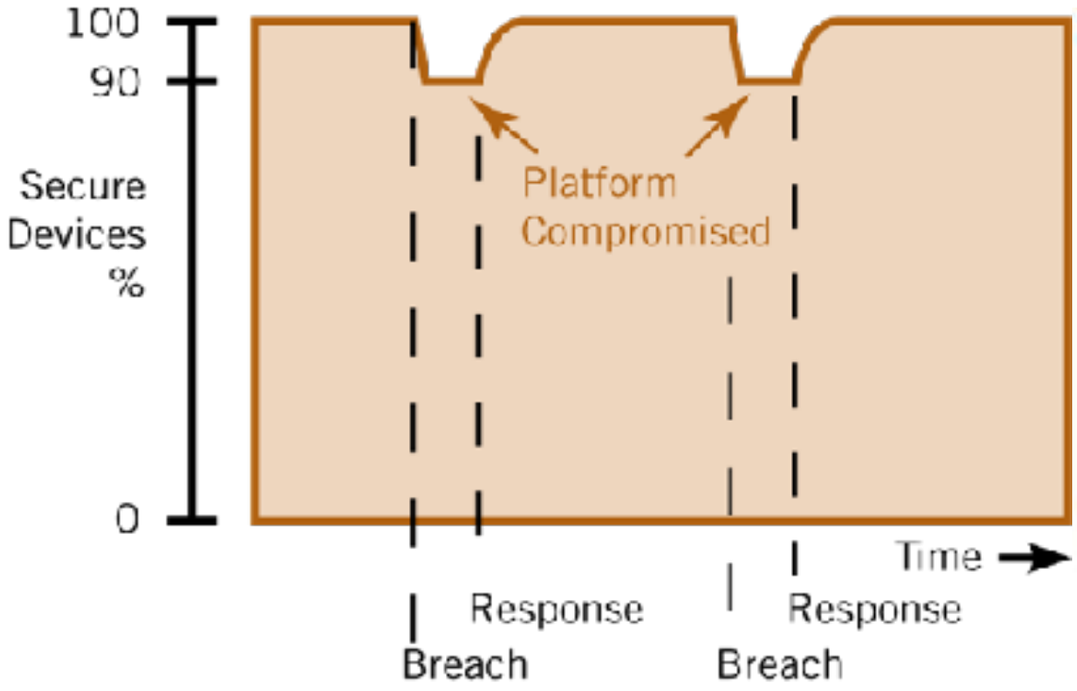




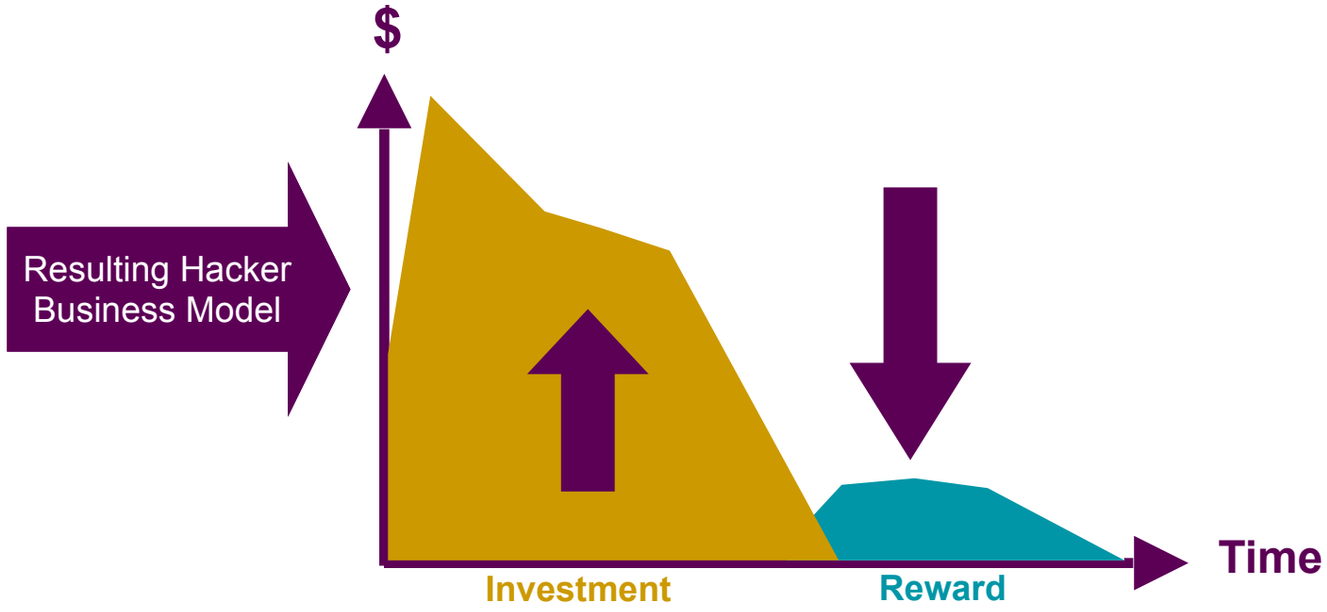
# Mitigation

- **Strong attack response**
- **Reduces duration of attack**




**Tamper resistance**
Raises cost of attack

**Diverse production**
Reduces scope of attack




## Software Diversity Benefits

Minimise scope of attack -- Prevent automated attacks

Provide rapid recovery in the event of an attack

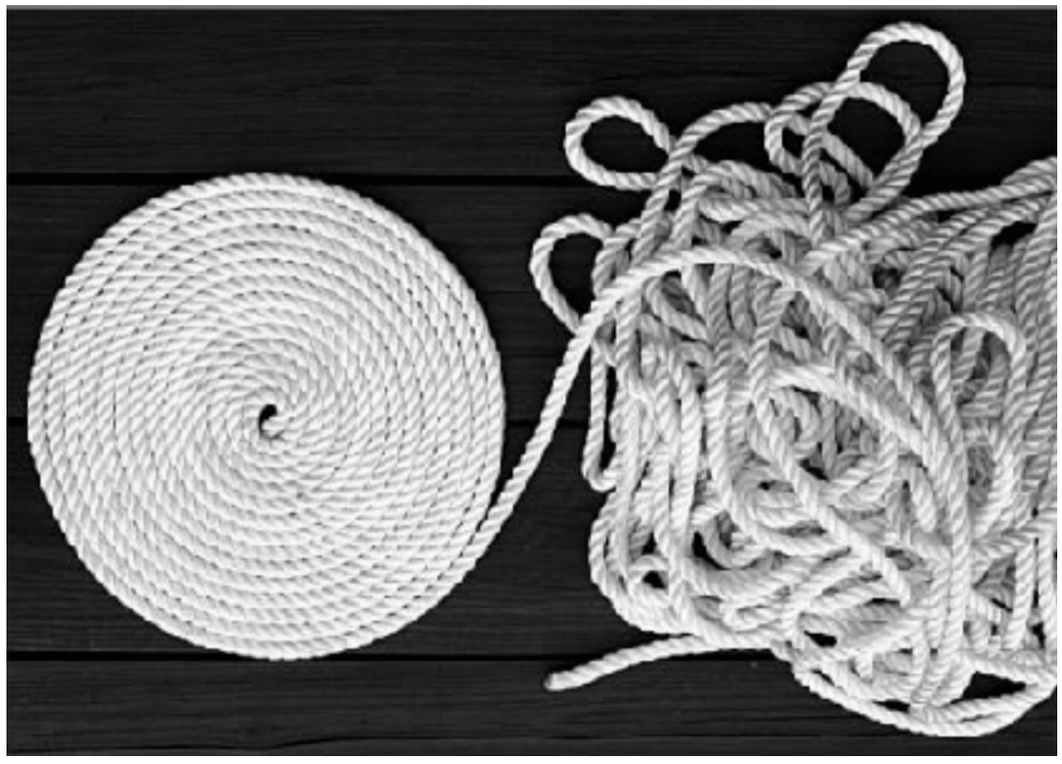
Make the business unattractive to the hacker

# Code Obfuscation

[Collberg et al. POPL 1998]

A program transformation O: Programs ⟶ Programs is a code obfuscation if:

- O preserves the observational behavior of programs
- O is potent, makes the program more complex to analyse



# Code Obfuscation

The level of security from reverse engineering that an obfuscator adds to an application depends on:

- ✓the sophistication of the obfuscating transformation
- ✓the power of the deobfuscator
- ✓the amount of resources (time and space) available to the deobfuscator

✓Observe that:

- ✓it is difficult to measure the goodness/efficiency of an obfuscation
- ✓obfuscation can never completely protect an application

# Evaluating Code Obfuscation

An obfuscating transformation T is evaluated according to:

- potency: how much obscurity obfuscation T adds to the program (max)
- resilience: how difficult T is for a deobfuscator to undo (max)
- cost: how much computational overhead (time and space) T adds to the obfuscated application (min)
- stealth: how well code introduced by T fits in with the original code (max)

Quality(T) = potency + resilience + cost + stealth

Seminar Calendar
All Events
Dagstuhl Seminars
Dagstuhl Perspectives
GI-Dagstuhl Seminars
Summer Schools
Events
Research Guests
Expenses
Planning your visit

https://www.dagstuhl.de/19331

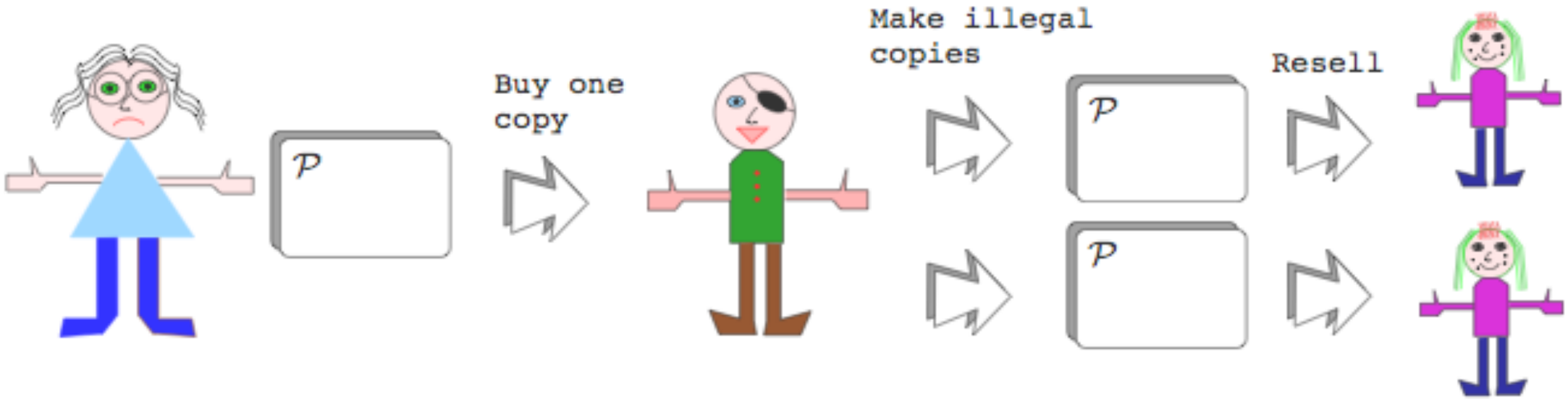
**August 11 – 16 , 2019, Dagstuhl Seminar 19331**

# Software Protection Decision Support and Evaluation Methodologies

## Organizers

Christian Collberg (University of Arizona – Tucson, US)
Mila Dalla Preda (University of Verona, IT)
Bjorn De Sutter (Ghent University, BE)
Brecht Wyseur (NAGRA Kudelski Group SA – Cheseaux, CH)

# SW Piracy



# SW Watermarking: idea

- Embed a unique identifier in a program to trace SW pirates
- Watermarking:
  - ✓ Discourages theft
  - ✓ Allows us to prove theft
- Fingerprinting
  - ✓ Trace violators
- Algorithms and procedure that are:
  - ✓ hard to destroy
  - ✓ stealthy
  - ✓ high bit-rate
  - ✓ little performance overhead.

# Steganography

- Steganography is the art and science of writing *hidden messages* in such a way that no one apart from the intended recipient knows of the existence of the message.

- Steganography means "covered, or hidden writing”

  - ✓ The Histories of Herodotus: Demeratus sent a hidden message about a forthcoming attack to Greece by writing a wooden panel which was covered in a wax.

  - ✓ Hidden messages on paper written with secret inks

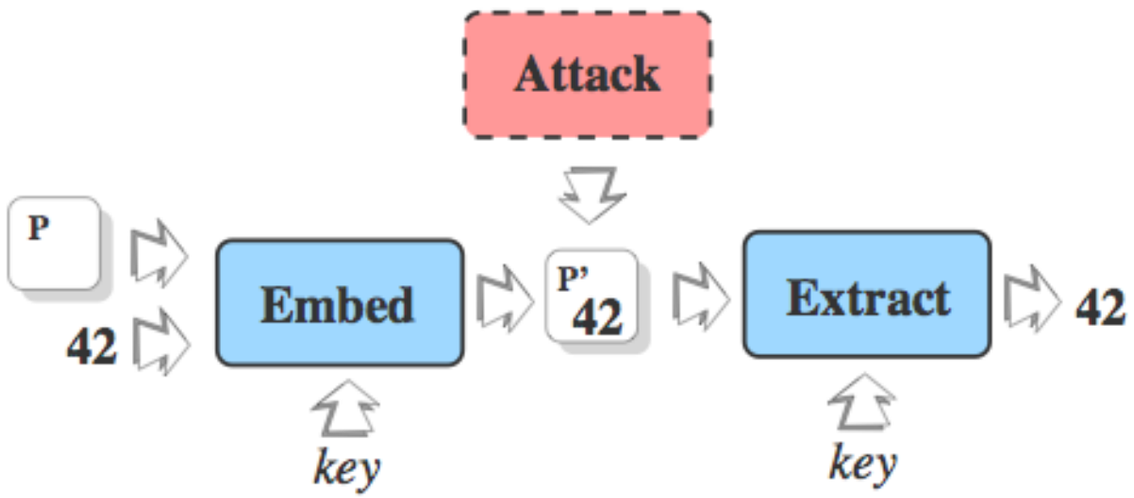
- Important: *Steganography hides the fact that the message exists!!*

- Watermarking and fingerprints are examples!

# Watermarking

- Watermarks were first introduced in Bologna by Fabriano, in 1282
- Watermarks appeared on British stamps from 1840
- The term "digital watermark" was first coined in 1992 by Tirkel and Osborne, in their paper: A.Z.Tirkel, G.A. Rankin, R.M. Van Schyndel, W.J.Ho, N.R.A.Mee, C.F.Osborne. “Electronic Water Mark”. DICTA 93, Macquarie University. p.666-673

# Watermarking




Embed an integer $W$ in program $P$ such that

- $W$ is resilient against automated attacks
- $W$ is stealthy
- $W$ is large (high bitrate)
- the overhead (space and time) is low

# Watermarking

Embed: `Init_RND(`$\mathcal{K}$`);`

**repeat** $n \approx 10000$ **times**

$i \leftarrow$ **RND**(); $j \leftarrow$ **RND**();

**fix brightness** : $a_i \leftarrow a_i + \delta$; $b_j \leftarrow b_j - \delta$;

Extract: `Init_RND(`$\mathcal{K}$`);` $S' \leftarrow 0$;

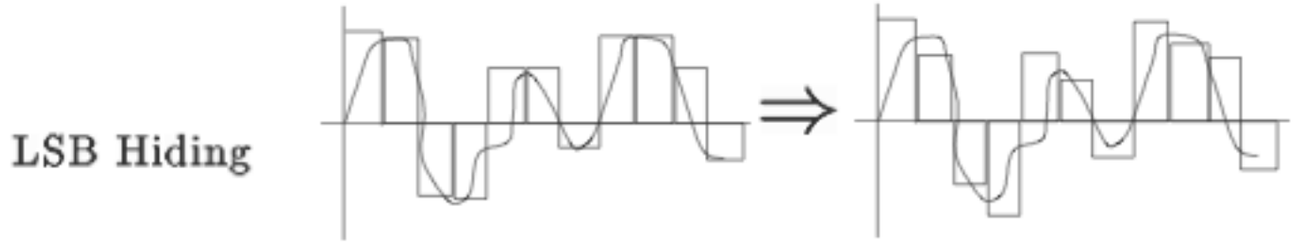
**repeat** $n \approx 10000$ **times**

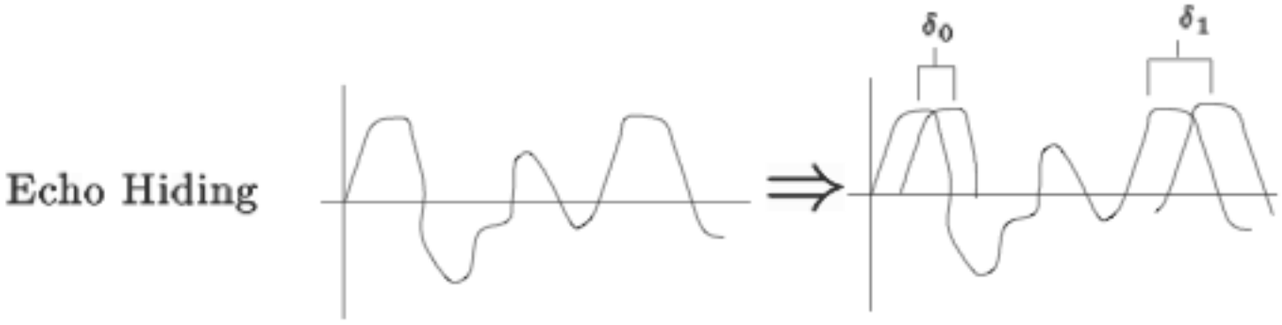
$i \leftarrow$ **RND**(); $j \leftarrow$ **RND**();

**sum brightness** : $S' \leftarrow S' + a_i - b_j$;

- $\sum_i^n (a_i - b_i) \gg 0 \Rightarrow$ watermark!
- Bit-rate: 1 bit per image.

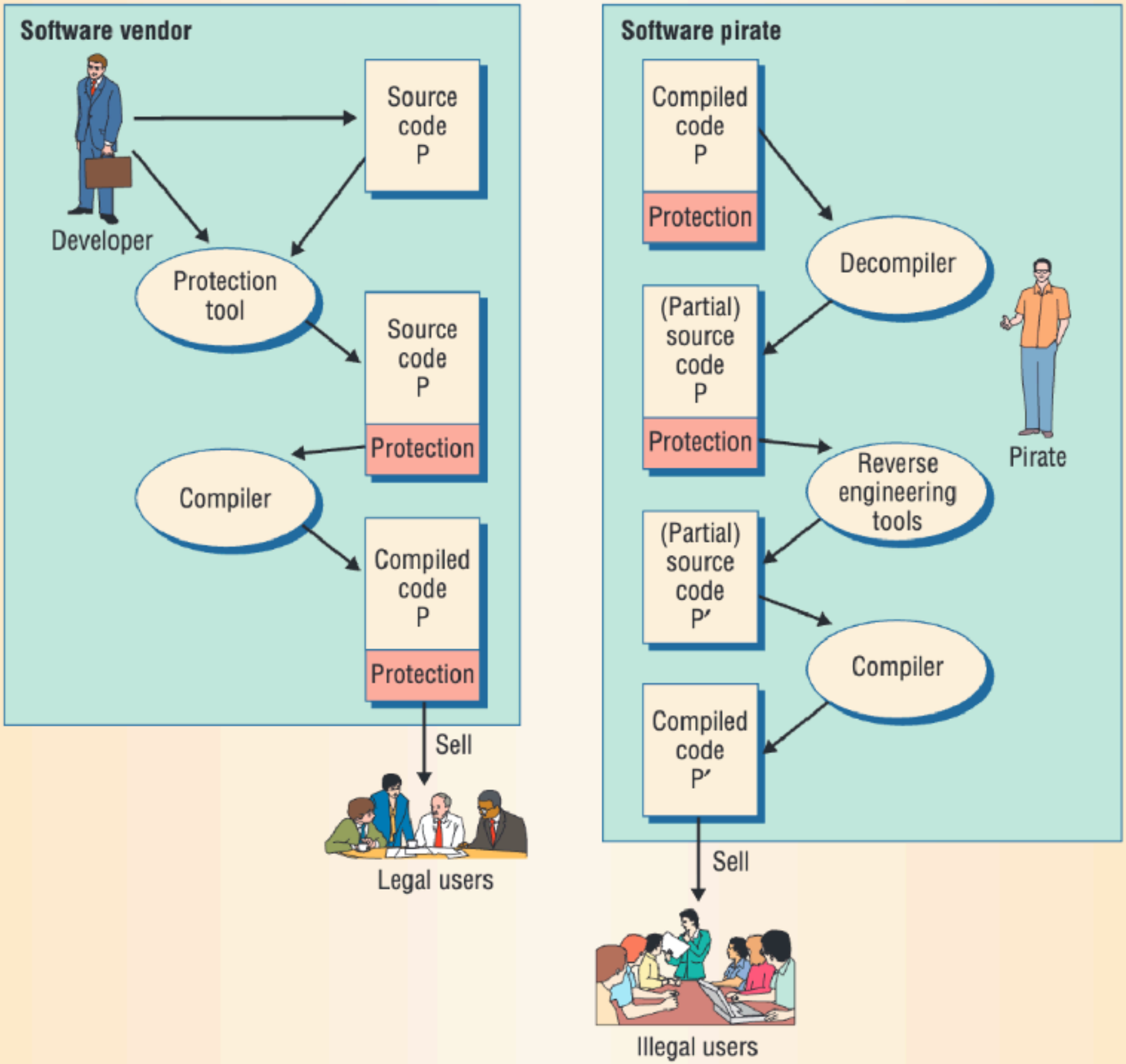
# Audio Watermarking



# Text Watermarking

| | |
|---|---|
| Hard-Copy | 12pt { I saw the best minds<br>of my generation,<br>14pt { starving hysterical naked |
| White-Space | I␣␣␣saw␣␣the␣best␣␣␣minds<br>of␣␣␣␣␣␣␣my␣␣␣generation,<br>starving␣hysterical␣naked |
| Syntactic | It was the best minds<br>of my generation that I saw,<br>starving hysterical naked |
| Semantic | I observed the choice intellects<br>of my generation,<br>starving hysterical nude |


Software vendor
Developer
Source code P
Protection tool
Source code P
Protection
Compiler
Compiled code P
Protection
Sell
Legal users
Software pirate
Compiled code P
Protection
Decompiler
(Partial) source code P
Protection
Reverse engineering tools
Pirate
(Partial) source code P′
Compiler
Compiled code P′
Sell
Illegal users

# SW Tampering: idea

- goal: ensure that the program executes *as intended* even in presence of an adversary
- it should *detect* when tampering occurs and *respond* to the attack
- related to obfuscation and software watermarking

```
if (license-expired()){
   printf ("pay m1!!");
   abort();
}
```

```
if (false){
   printf ("pay m1!!");
   abort();
}
```

program crashes

- Sw developer's revenue depends on their programs executing as aspected, while the adversary's revenue depends on modifying the semantics of these programs to execute the way they wanted to

# What is tamper proofing?

- Ensure that programs executes as intended, even in the presence of an adversary who tries to disrupt, monitor, or change the execution

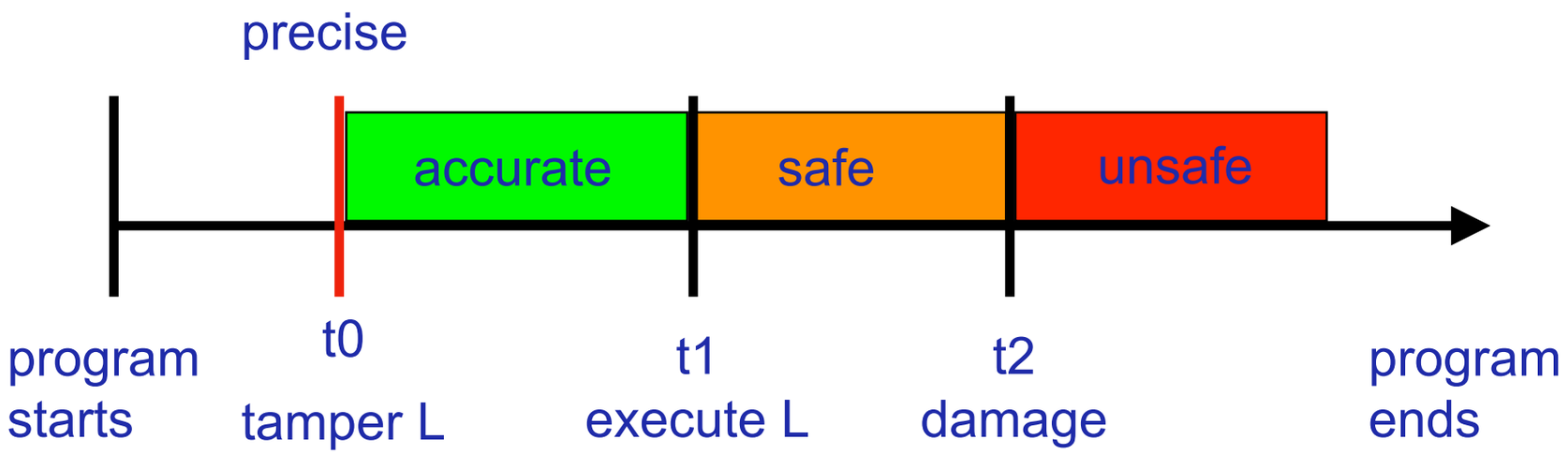
- A tamper-proofing algorithm:
  - CHECK: that monitors the health of the system by testing a set of invariants and returning **true** if nothing suspicious is found
  - RESPOND: queries CHECK to see if P is running as expected, and if it's not, issues a tamper response (e.g. terminating the program)

# Checking for Tampering

CHECK tests a set of invariant

- Code checking: check that P's code hashes to a known value
- Result checking: test that the result of computation is correct. Checking the validity of the computation result is often computationally cheaper than performing the computation itself
- Environment checking: the hardest thing for a program to check is the validity of its execution environment.
  - Running under an emulator
  - Operating system
  - …

# Precision




- ✓ Precise: the checker detects the attack immediately after the modification has taken place
- ✓ Accurate: the checker detects the attacks before the modified code is executed
- ✓ Safe: the checker waits until after the modified code has been executed, but before the first interaction event (before damage)
- ✓ Unsafe: the checker identifies the attack after the damage has taken place…

# Respond Scenarios

- ✓ Terminate the program
- ✓ Restore the program (patch the code)
- ✓ Run incorrect results / deteriorate performance
- ✓ Report an attack
- ✓ Punish the attacker: destroy code / system (malware!!)

# Attack: Program Analysis

```
n := n0;

i := n;

while (i <> 0 ) do

      j := 0;

      while (j <> i) do

            j := j + 1

      od;

      i := i - 1

od
```

```
   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}
```

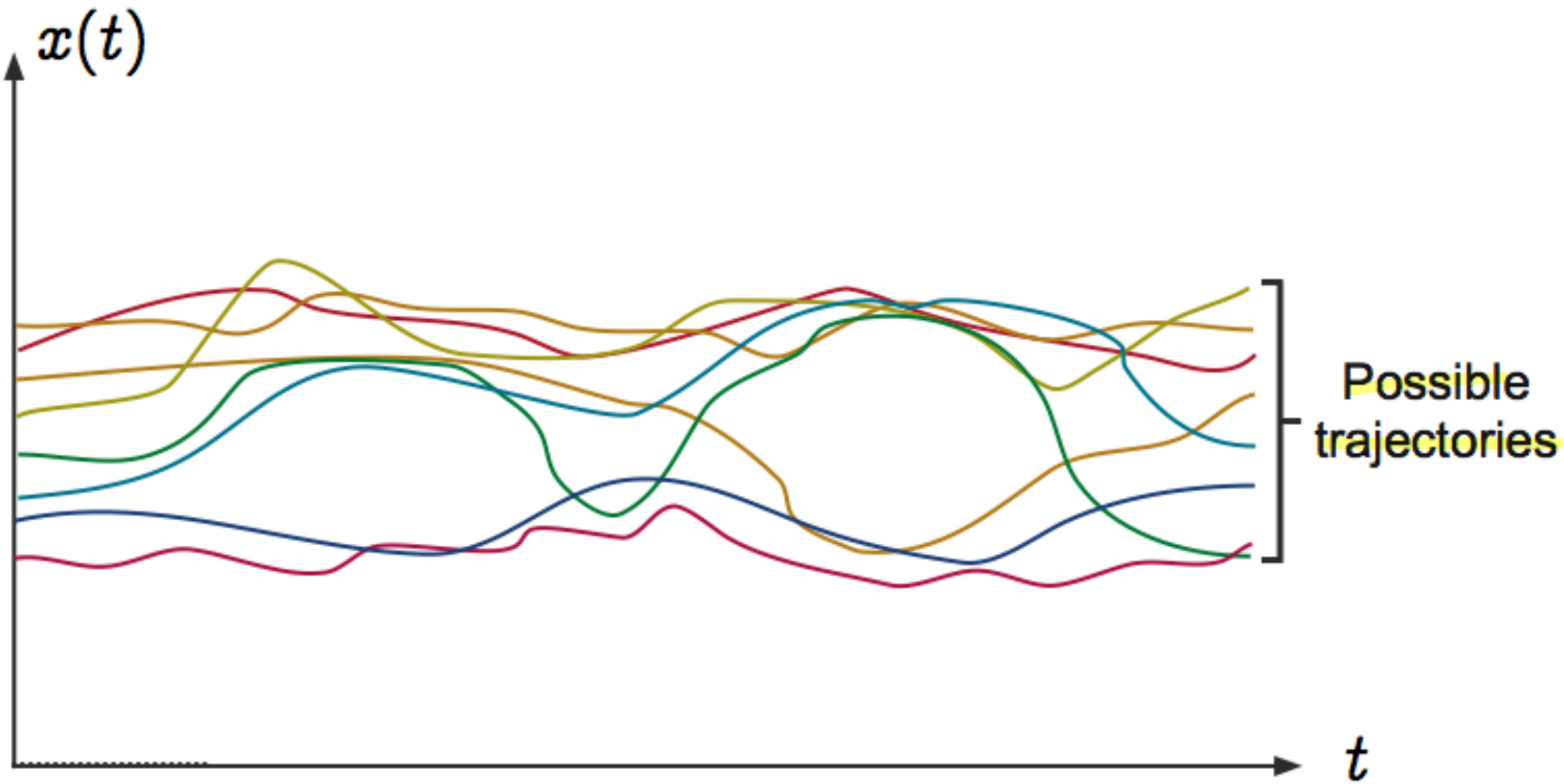
# Attack: Program Analysis

## Example of static analysis (safety)

```
{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        ⟵  j < n0 so no upper overflow
            {n0=n,j>=1,i>=j,n0>=i}
         od;
      {n0=n,i=j,i>=1,n0>=i}
         i := i - 1              ⟵  i > 0 so no lower overflow
      {i+1=j,n0=n,i>=0,n0>=i+1}
   od
{n0=n,i=0,n0>=0}
```

n0 must be initially nonnegative
(otherwise the program does not
terminate properly)

# Possible Behaviors



# Semantics

* The *concrete semantics* of a program formalizes (is a mathematical model of) the set of all its possible executions in all possible execution environments.

* The concrete mathematical semantics of a program is an *infinite* mathematical object, *not computable*;

* All non trivial questions on the concrete program semantics are *undecidable*

# Safety Properties

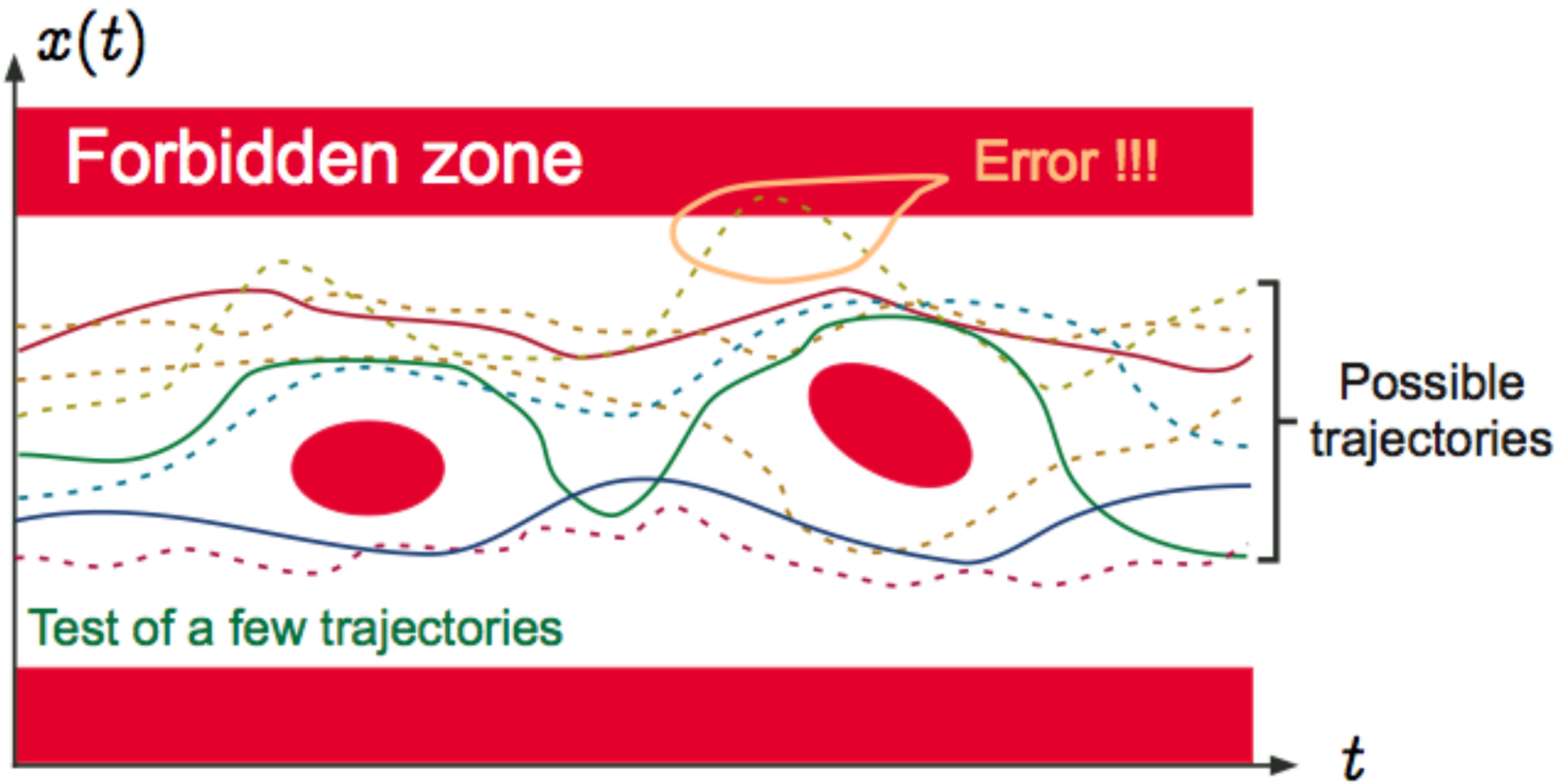
- The *safety properties* of a program express that no possible execution in any possible execution environment can reach an erroneous state
- A *safety proof* consists in proving that the intersection of the program concrete semantics and the forbidden zone is empty
- *Undecidable problem* (the concrete semantics is not computable)
- *Impossible* to provide completely automatic answers with finite computer resources and neither *human interaction* not *uncertainty* on the answer

# Safety Properties

# Dynamic Analysis
# Testing, Debugging

- Consists in considering a subset of the possible executions
- Not a correctness proof
- Absence of coverage is the main problem

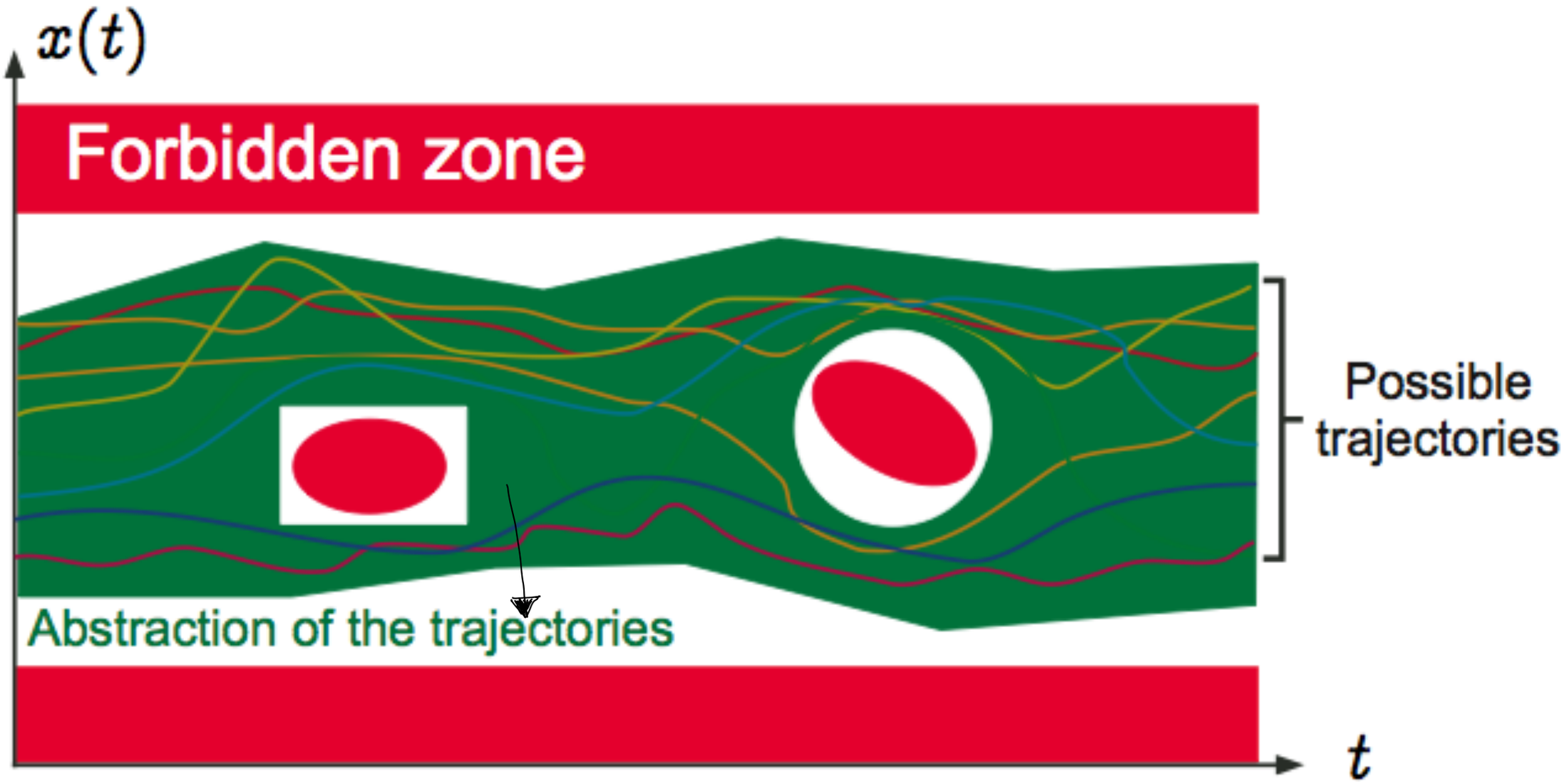
# Dynamic Analysis
# Testing, Debugging



# Abstract Interpretation

- Consists in considering an *abstract semantics*, that is to say a superset of the concrete semantics of the program
- Hence the abstract semantics *covers all possible concrete cases*
- *Correct*: if the abstract semantics is safe (does not intersect the forbidden zone) then so is the concrete semantics

# Abstract Interpretation



# Formal Methods

Formal methods are abstract interpretations, which differ in the way to obtain the abstract semantics

* Model checking:

  The abstract semantics is given manually by the user in the form of a finitely model of the program execution. Can be computed automatically, by techniques relevant to static analysis.

* Deductive methods:

  The abstract semantics is specified by verification conditions. The user must provide the abstract semantics in the form if inductive arguments (e.g. invariants). Can be computed automatically, by techniques relevant to static analysis

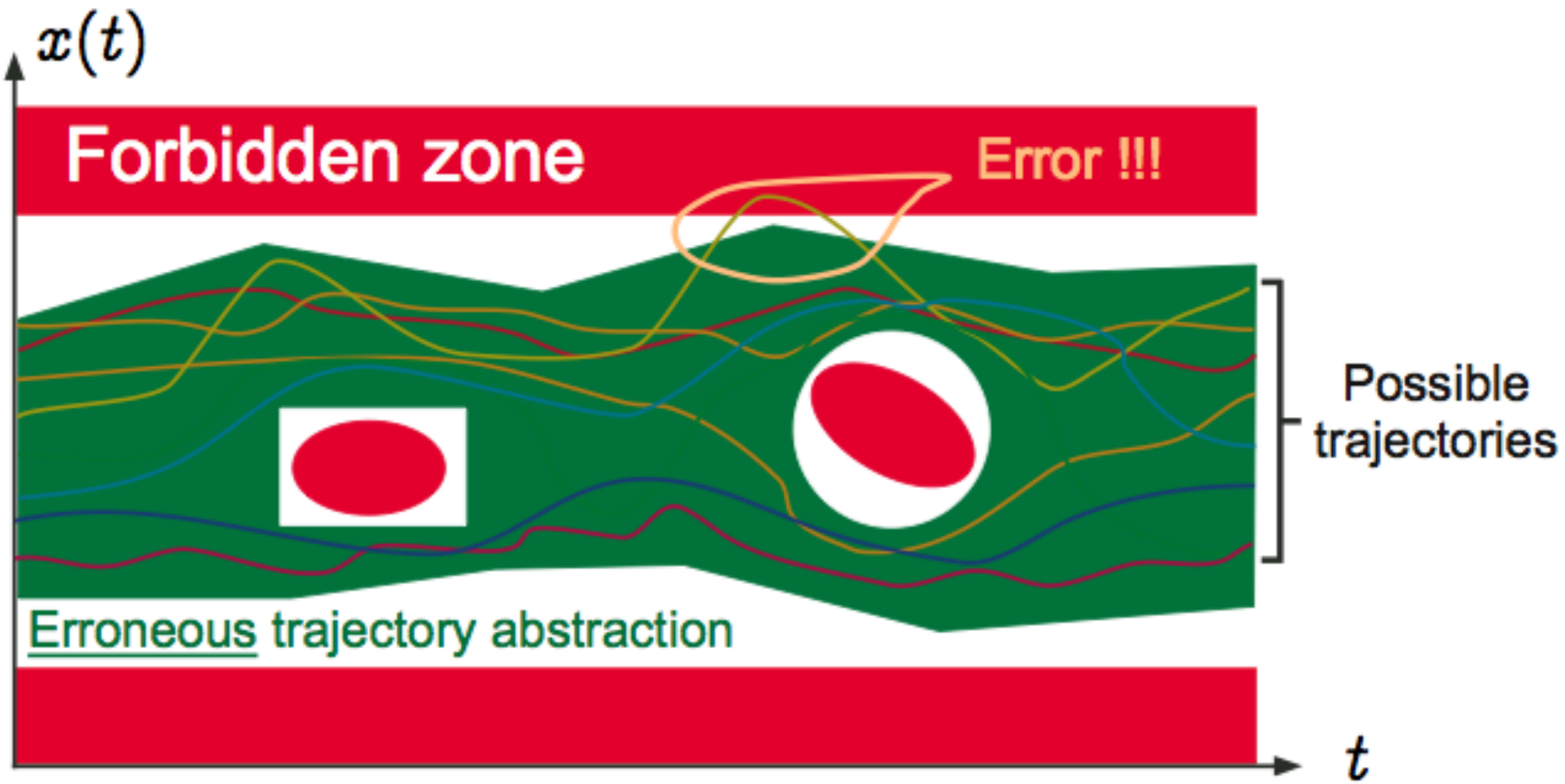
* Static analysis:

  The abstract semantic is computed automatically form the program text according tho predefined abstractions (that can sometimes be tailored automatically/manually by the user)

# Required properties of the abstract semantics

- Sound so that no possible error can be forgotten
- Precise enough (to avoid false alarms)
- As simple/abstract as possible (to avoid combinatorial explosion phenomena)

# Erroneous/Unsound Abstraction



# Imprecision & false alarms