Malware Analysis: Overview

Malicious Software

Vijay Ganesh
ECE458, Winter 2013
University of Waterloo

Today's Lecture: Detecting Malware

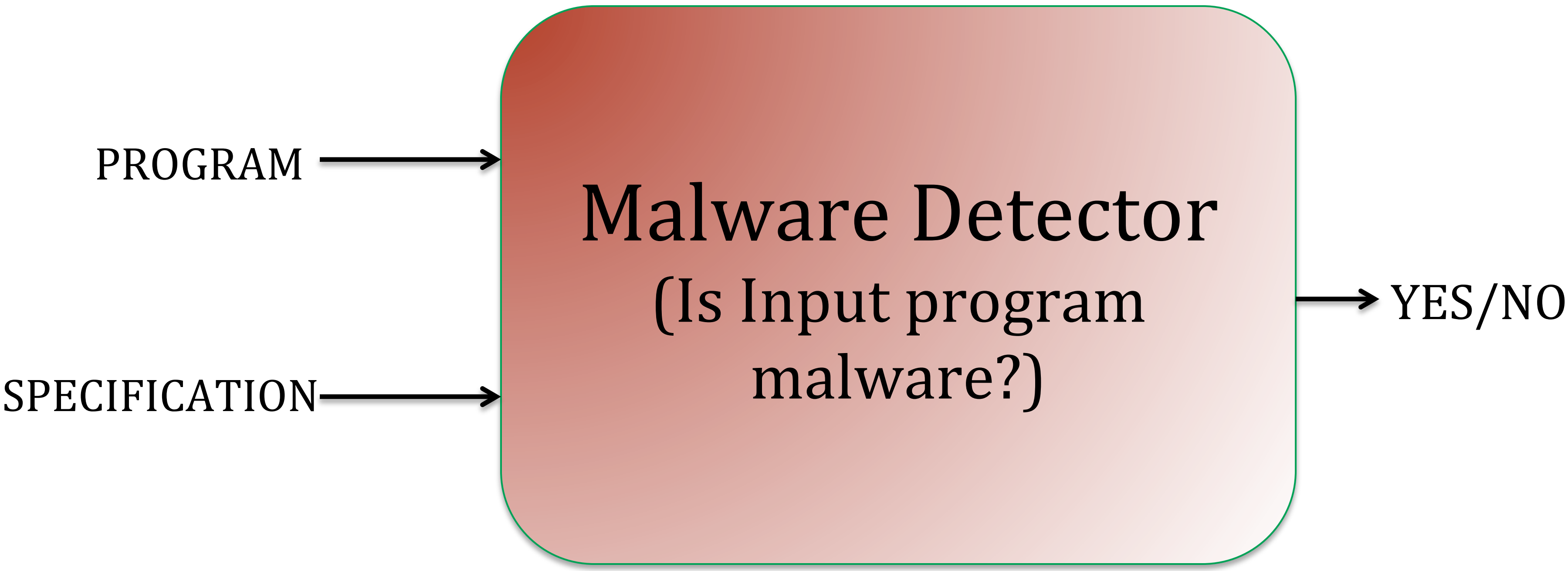
- **Previously we saw**
  - Ways of preventing exploitation
    - Use bug-finding and verification to get rid of errors
    - Hard
  - Managing attacks
    - Randomization (e.g., ASLR)
    - Works some times
- **Can we detect Malware?**
  - Static techniques
    - Analyze code to detect if it is malware
  - Dynamic techniques
    - Observe runtime behavior and flag suspicious activity
  - Hybrid

What is Malware?

- ***Virus***
  - malicious code which replicates by inserting itself into other programs
- ***Worm***
  - malicious code which replicates itself by itself
- ***Root kits***
  - malicious code designed to hide other programs and maintain control of system
- ***Trojans***
  - malicious code embedded by its designer in an application or system
- …

Many slides courtesy David Brumley (CMU)

Malware Detection



Malware Detection


Good Guy

Develop function $\boldsymbol{f}$ such that for a program P:

$\boldsymbol{f}(P = \text{malware}) = \text{true}$

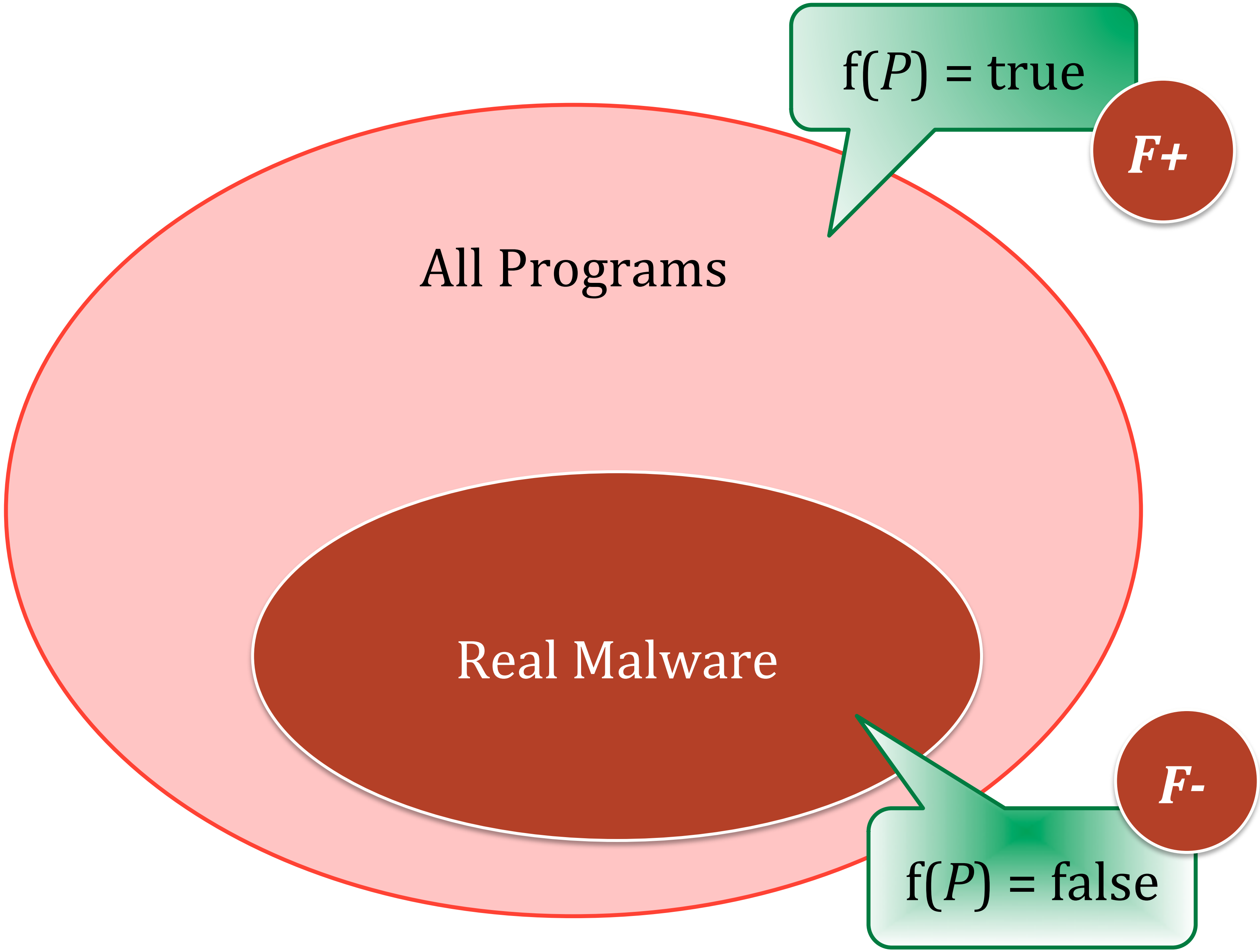
$\boldsymbol{f}(P = \text{safe}) = \text{false}$

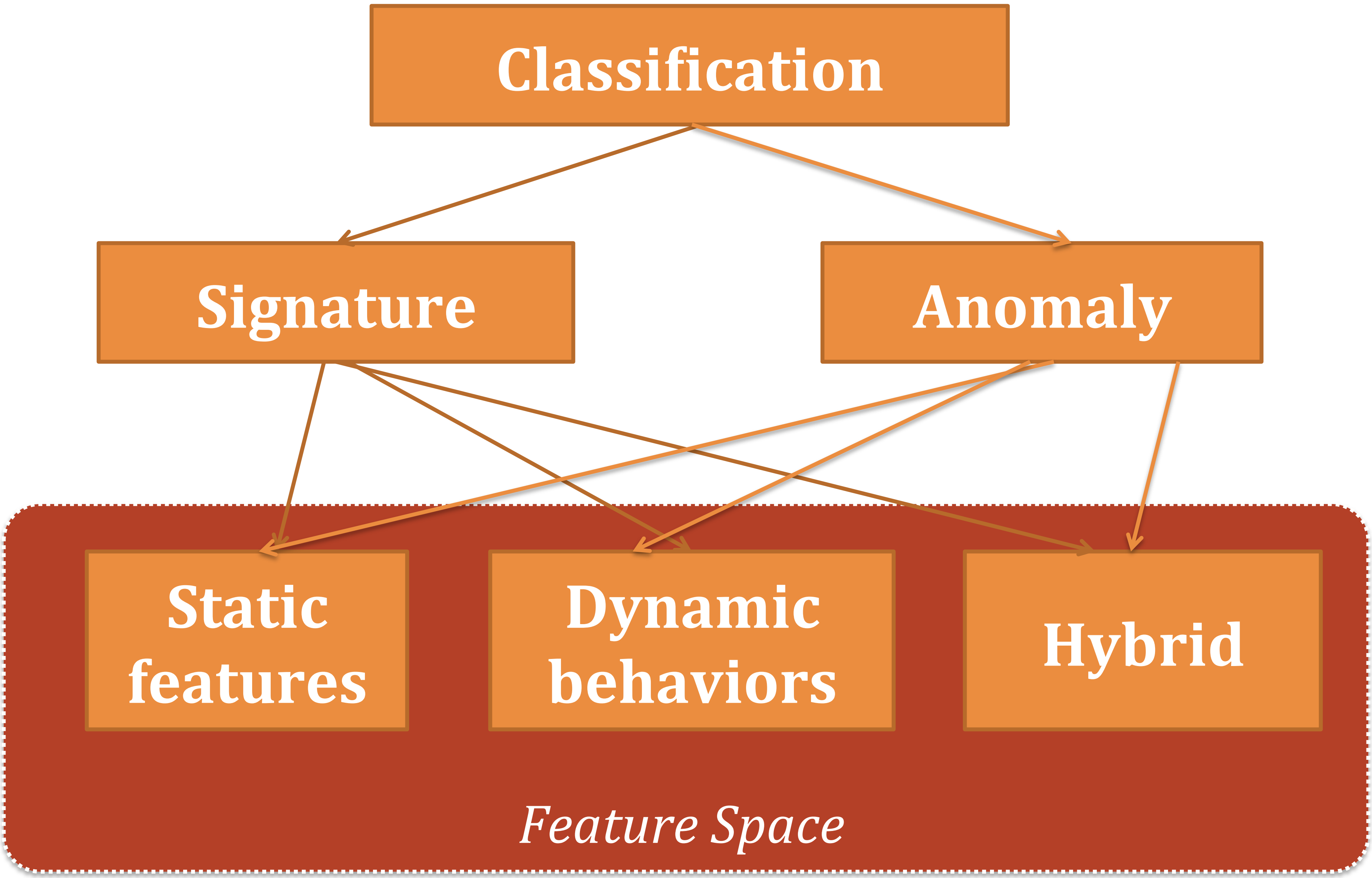

Bad Guy

Develop malware $\boldsymbol{m}$ such that $\boldsymbol{f}$ performs poorly.

Malware Detection is hard
Why?

- A malware is a program
- Program analysis is undecidable in general (Rice's Theorem) Reason: loops
- Infeasible in practice to detect properties if enough evasion mechanism is built-in
- Even so, we can make a lot of progress. How?
- Trade-offs: soundness, completeness, precision


f(P) = true
F+
All Programs
Real Malware
F-
f(P) = false


Classification
Signature
Anomaly
Static features
Dynamic behaviors
Hybrid
Feature Space

Static Features: Abstractions

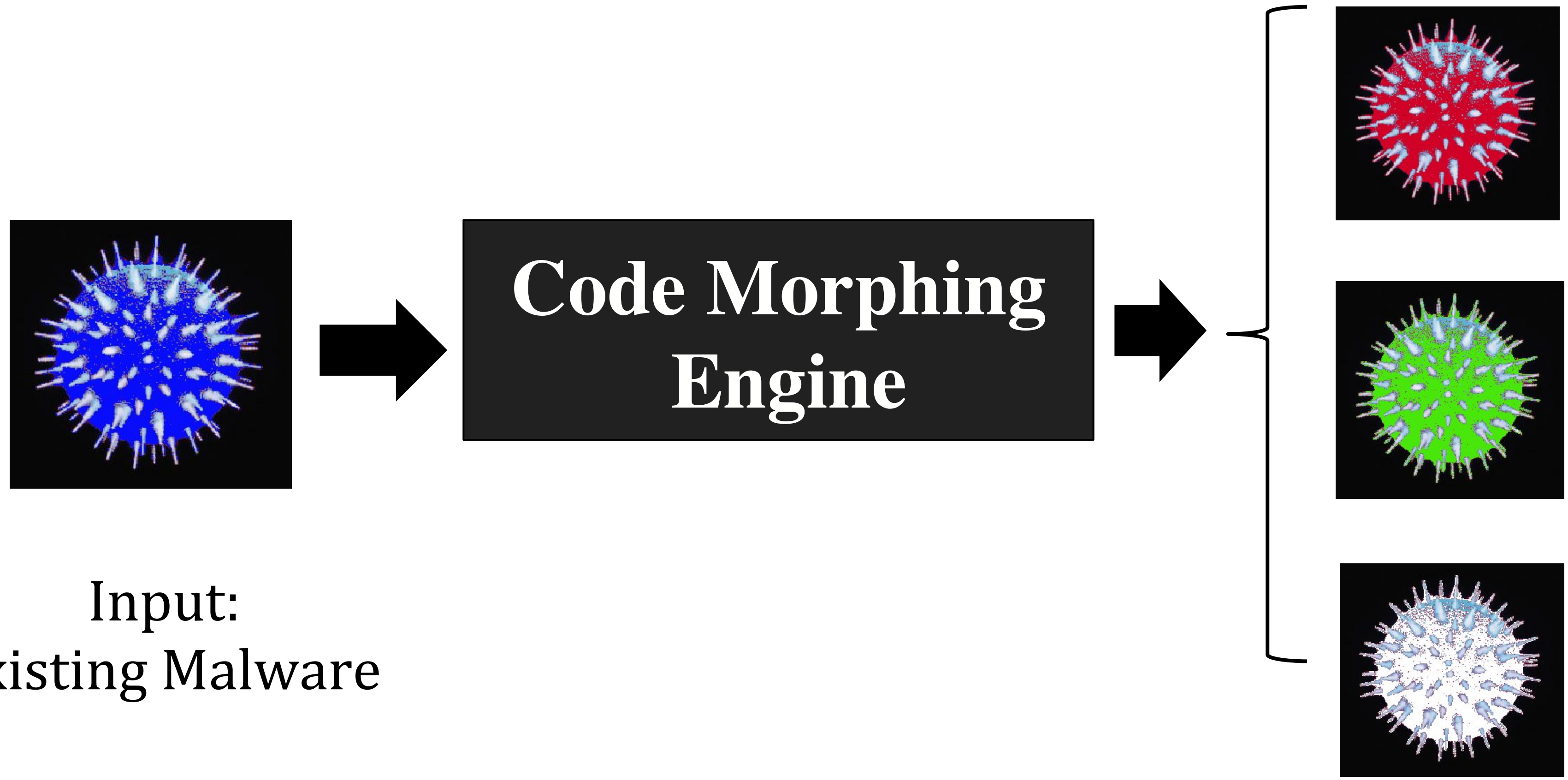
- Hash(P) == known malware hash
- Bytes-At(P,offset) == known malware string
- Instrs-At(P,offset) == known malware instr
- CFG (P) == known malware CFG
- …

Static Approaches

Some challenges and Pitfalls

- Metamorphism
  - Ex: Instruction substitution
- Polymorphism
  - Packing
    - “Unpack” code to memory and transfer control
    - Just change packer to come up with new malware
  - Encryption
    - Self decryption loop

Morphing Malware



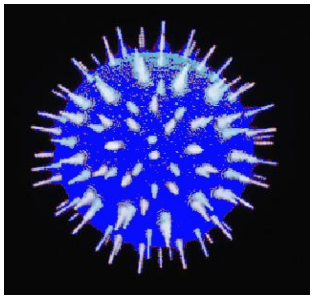
Input:
Existing Malware

Output:
New Variants

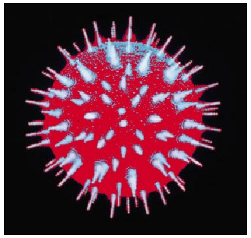
Metamorphism

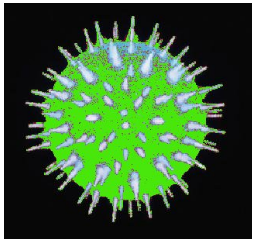
A metamorphic engine is a function ***E*** that:

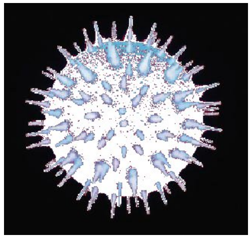
1. takes in a malware program ***P***: X->Y represented as a list of instructions
2. ***E***(***P***) outputs a list of instructions ***P'*** s.t. for all ***x*** in ***X***, ***P'(x) = P(x)***
3. ***P*** and ***P'*** differ on at least one instruction



Metamorphic Engine







Input:
Existing Malware

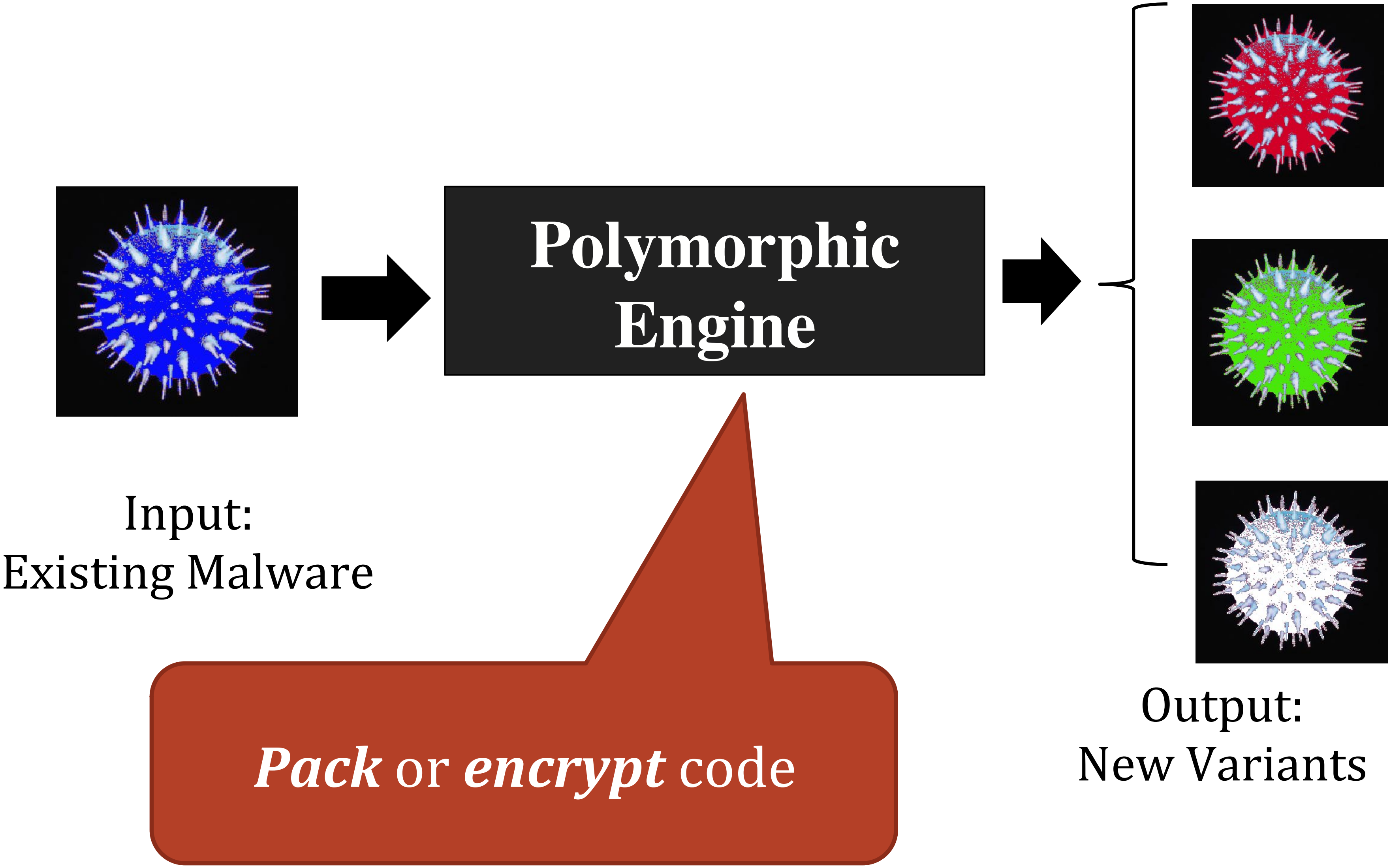
Replace ***left shifts*** by ***multiplication by power of two***, and *multiplication by a power of two* with *a left shift*

Output:
New Variants

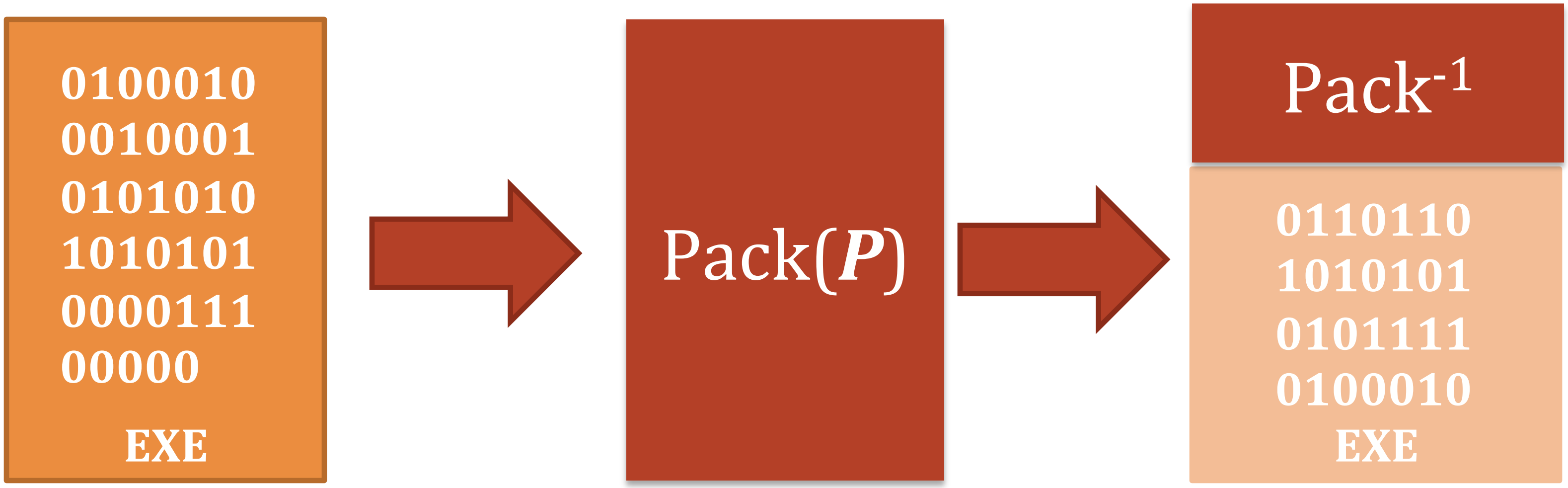
Polymorphism

A ~~metamorphic~~ polymorphic engine is a function ***M*** that:

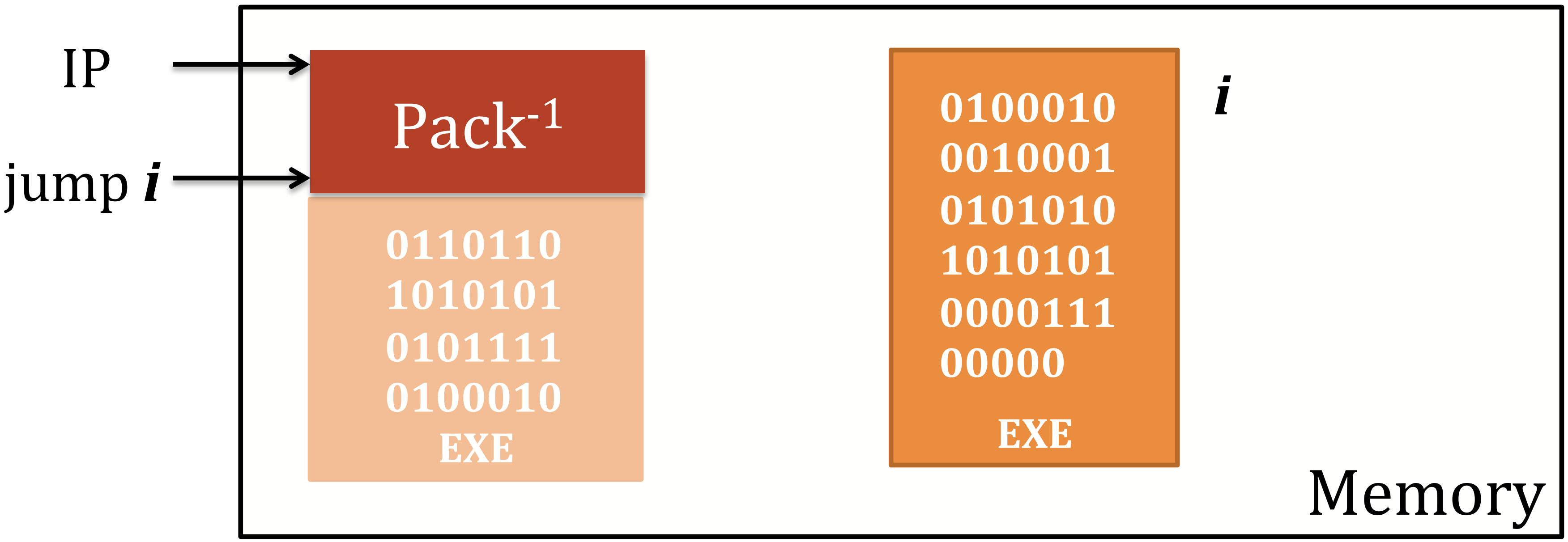
1. takes in a malware program ***P***: X->Y represented as a ~~list of instructions~~ string
2. ***M***(***P***) outputs a ~~list of instructions~~ string ***P'*** s.t. for all ***x*** in ***X***, ***P'(x) = P(x)***
3. ***P*** and ***P'*** differ on at least one ~~instruction~~ byte

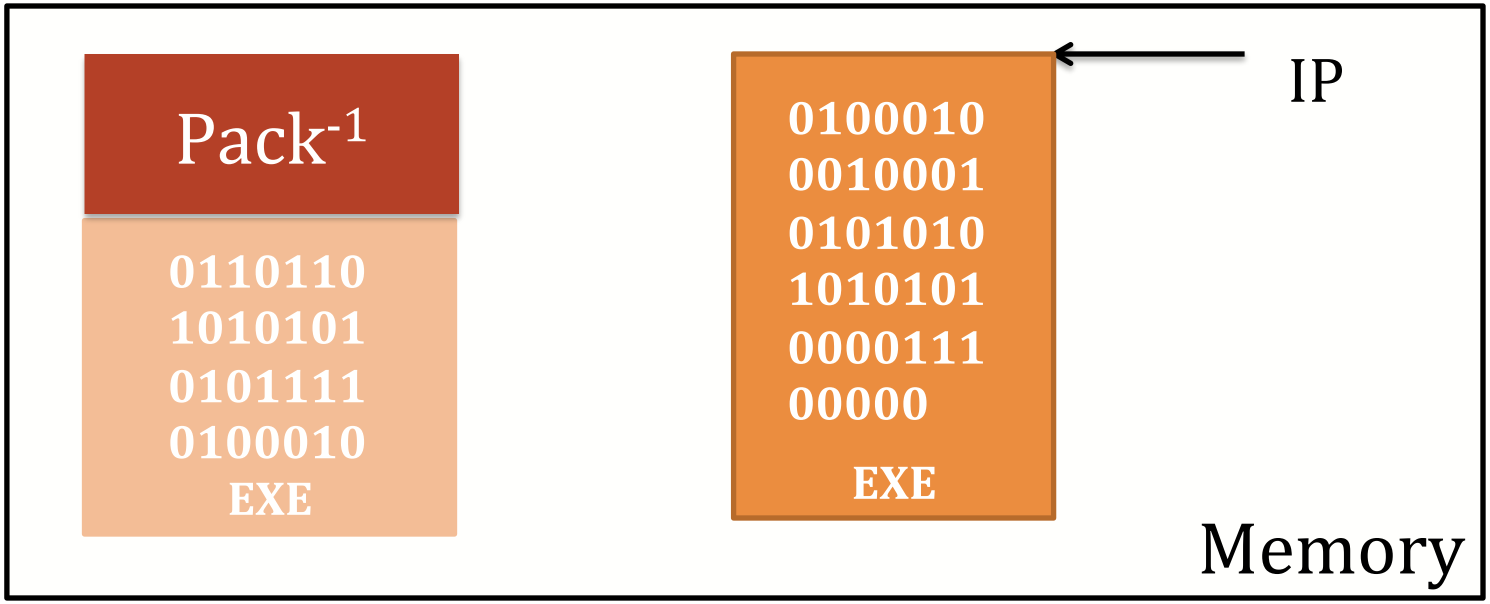

Polymorphic Engine
Input:
Existing Malware
Output:
New Variants
Pack or *encrypt* code

Code Packing: A pair of functions

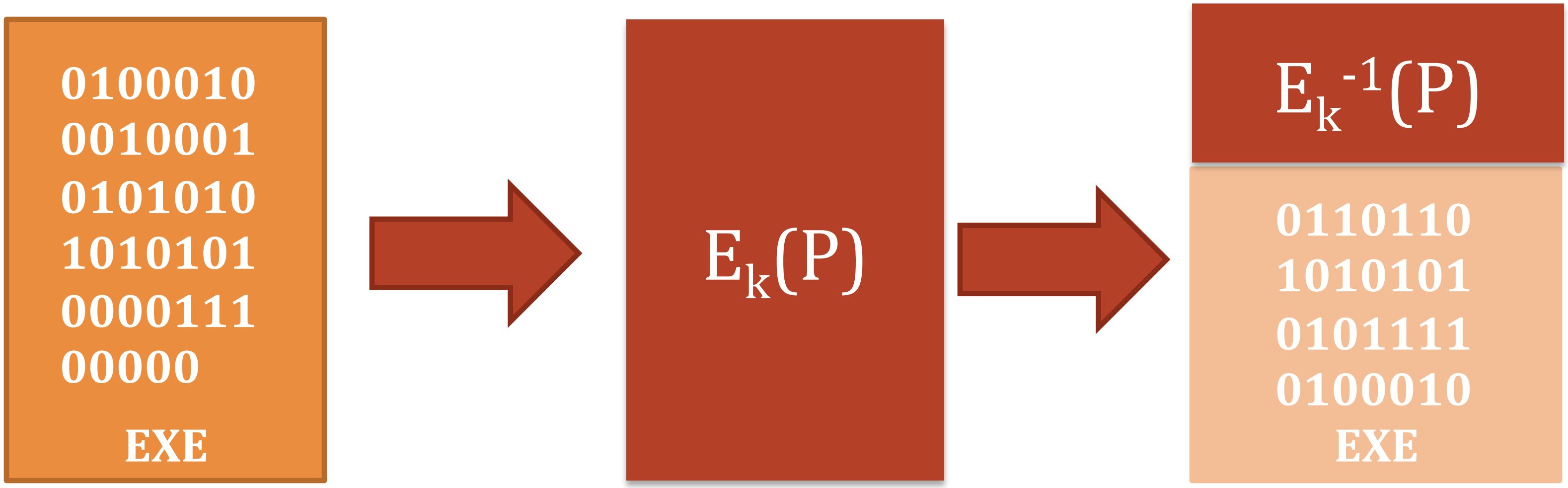


Pack and Pack^{-1} are inverse functions


IP
jump *i*
Pack-1
0110110
1010101
0101111
0100010
EXE
0100010
0010001
0101010
1010101
0000111
00000
EXE
i
Memory

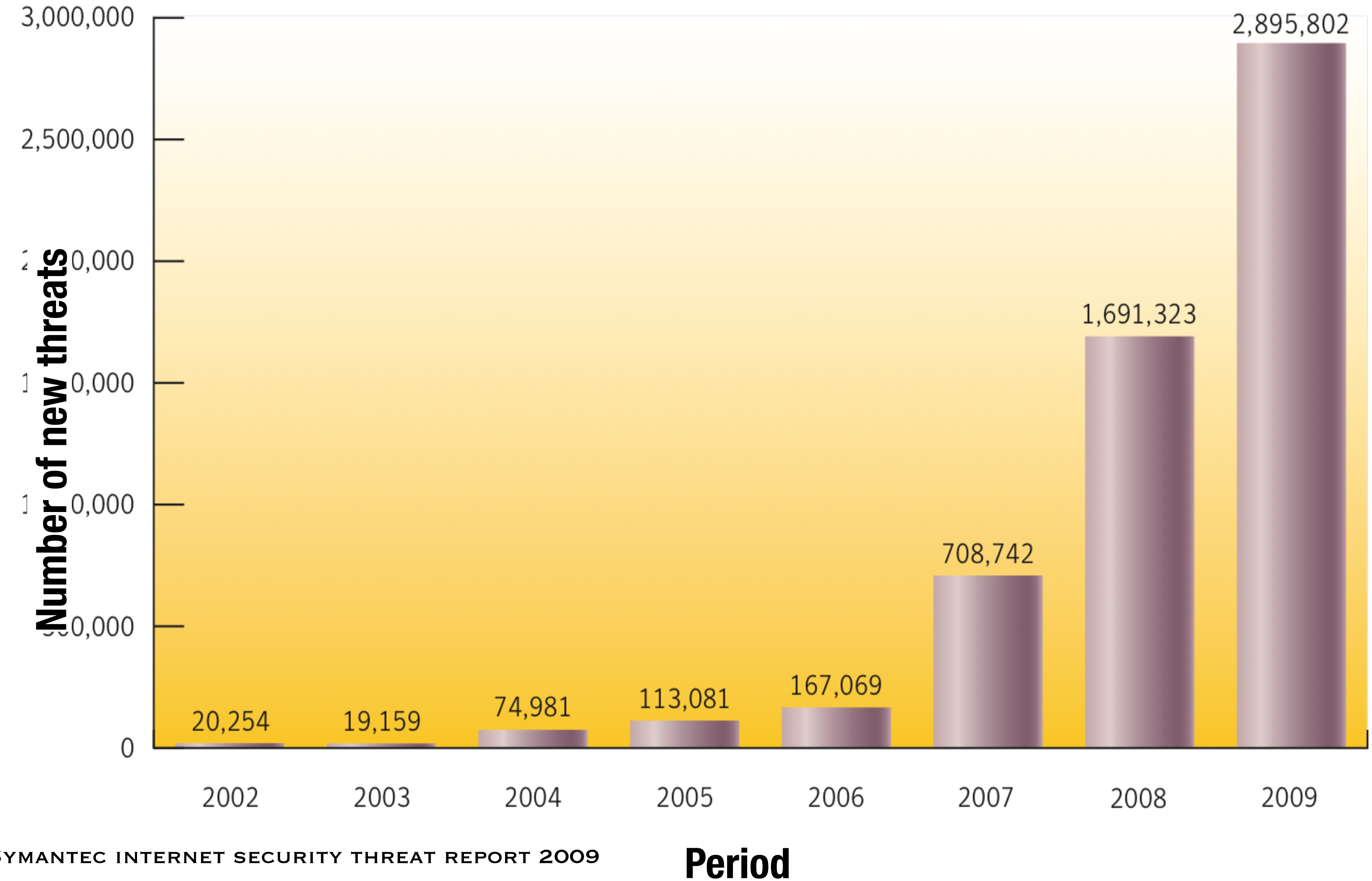

Pack^{-1}
0110110
1010101
0101111
0100010
EXE
0100010
0010001
0101010
1010101
0000111
00000
EXE
IP
Memory

Code Encryption: A pair of functions



E and E^{-1} are inverse functions

Mighty Malware Morphing



Symantec internet security threat report 2009

Malware Detection
Semantic-aware Static Analysis

- Hard for attacker to hide malicious behavior from a semantic detector
- Polymorphism and metamorphism are no match for a good semantic detector
- Why?

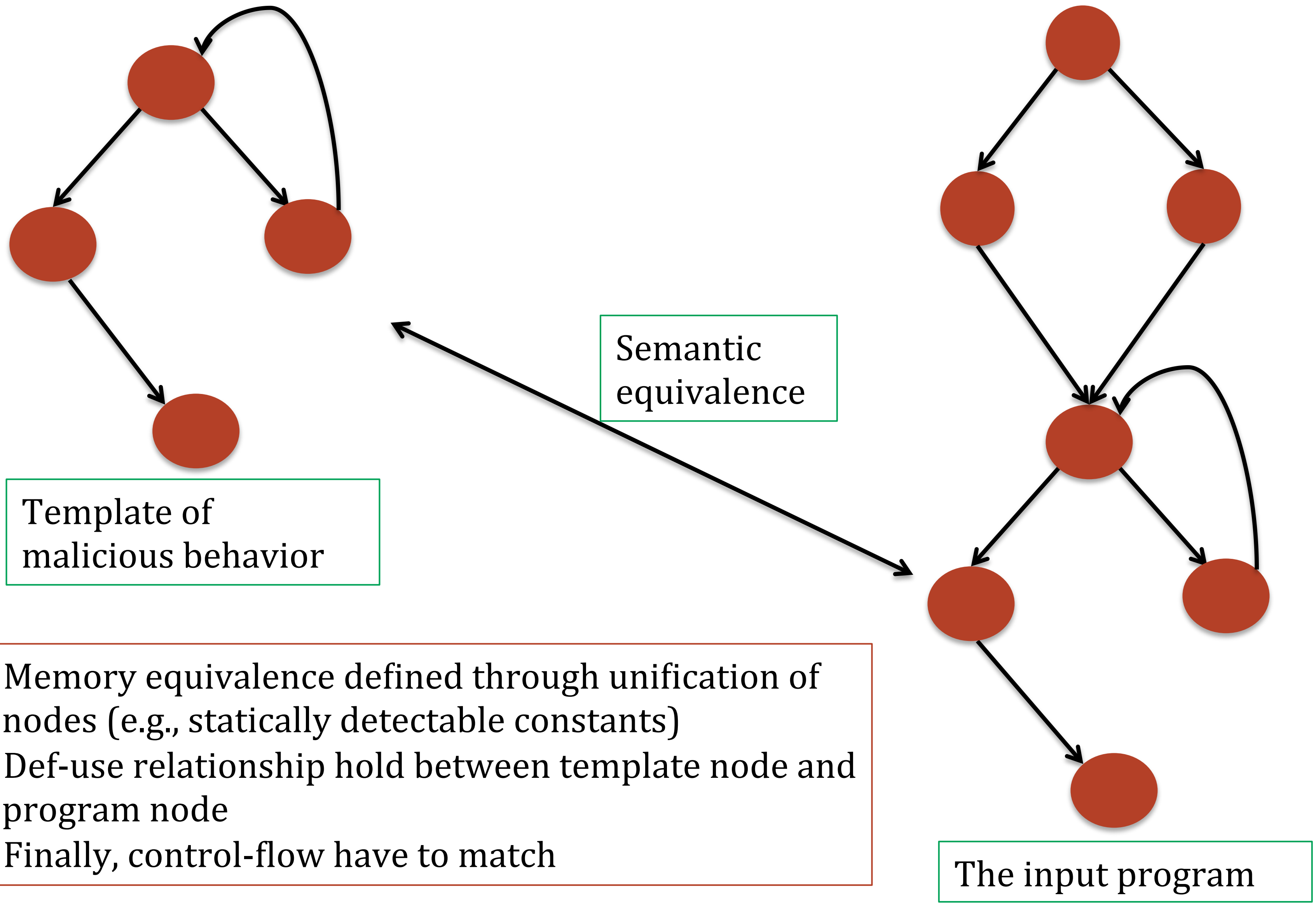
Semantic-aware Malware Detectors

- Template of malicious behavior
  - Initial memory
  - Control-flow
  - Final memory state

- Idea for semantic-aware detector
  - If your code start from the same initial memory as template
  - The two end up in the same memory state after a certain control-flow
  - Then your code is possibly malware

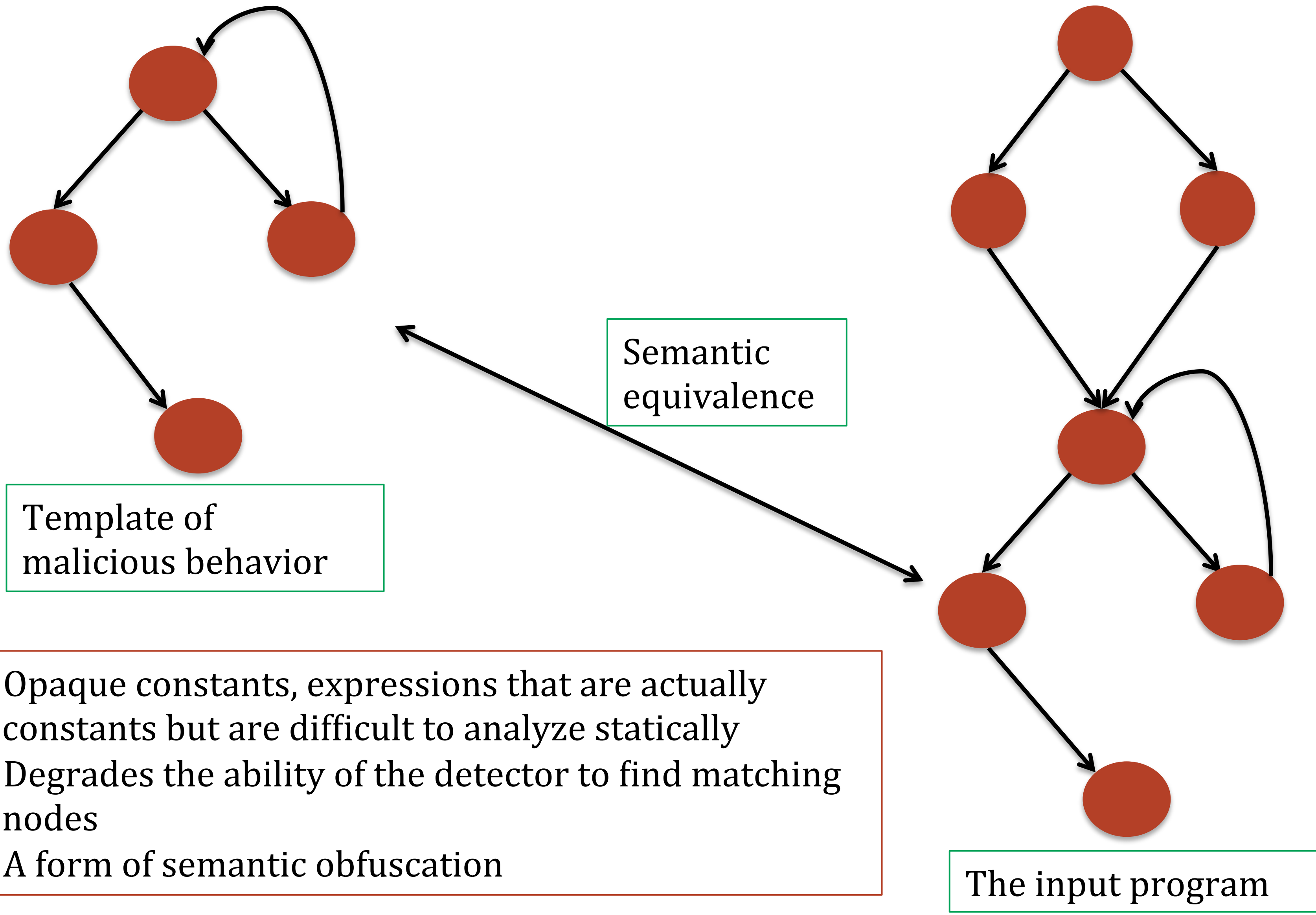
- Syntactic obfuscation, program transformation cannot change behavior

Semantic-aware Malware Detectors



- Memory equivalence defined through unification of nodes (e.g., statically detectable constants)
- Def-use relationship hold between template node and program node
- Finally, control-flow have to match

Defeating Semantic-aware Malware Detectors



- Opaque constants, expressions that are actually constants but are difficult to analyze statically
- Degrades the ability of the detector to find matching nodes
- A form of semantic obfuscation

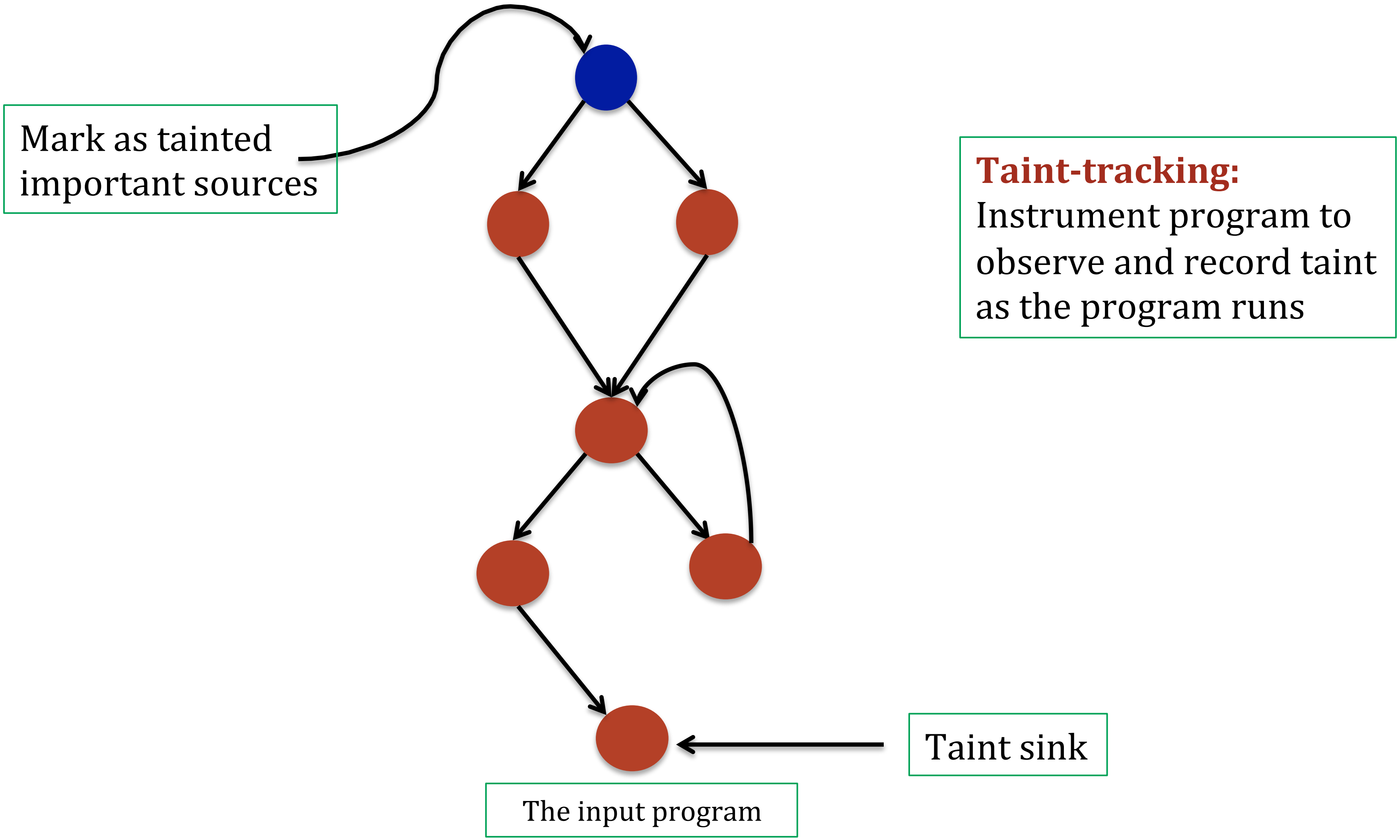
How do you defeat obfuscation?

- Detectors used signatures to detect malware
  - Attackers used meta and polymorphism to evade signature-detection
- Okay, so detectors used semantic-aware analysis
  - Attackers used semantic obfuscation, e.g., opaque constants
- Now what to do?
  - If obfuscated, flag as malware
  - Use dynamic behavior to detect suspicious activity
  - Use hybrid approaches that combine static and dynamic detection techniques

Look for Suspicious Dynamic Behavior

- Write to the registry where/what you shouldn't
- Delete sensitive files
- Send sensitive information over network without permissions
- Packing
  - Is this always a good indicator?

Information-flow Tracking



TaintDroid
Information-flow based Privacy Monitoring

Idea:

- Instrument for taint-tracking all of Android
- Variable-level, method-level, and file-level tracking
- Monitor at run-time, how apps handle user's private information
- Track using taint to see if they send sensitive information to "unsafe" sinks
- If yes, the App is malicious

TaintDroid
Information-flow based Privacy Monitoring

Issues:

- Slow-down (claim only 14% performance overhead)
- Implicit vs. explicit information-flow
- Does not track implicit information-flow
- Apps can game TaintDroid through implicit taint

Explicit information-flow

```
y = input + z;
```

Implicit information flow

```
Func (Nibble input)
If(input == val) y = 5 + z;
```

Information-flow Tracking

Implicit vs. Explicit Flows

Explicit information-flow

```
f(char sensitive)
{
   y = sensitive + z;
}
```

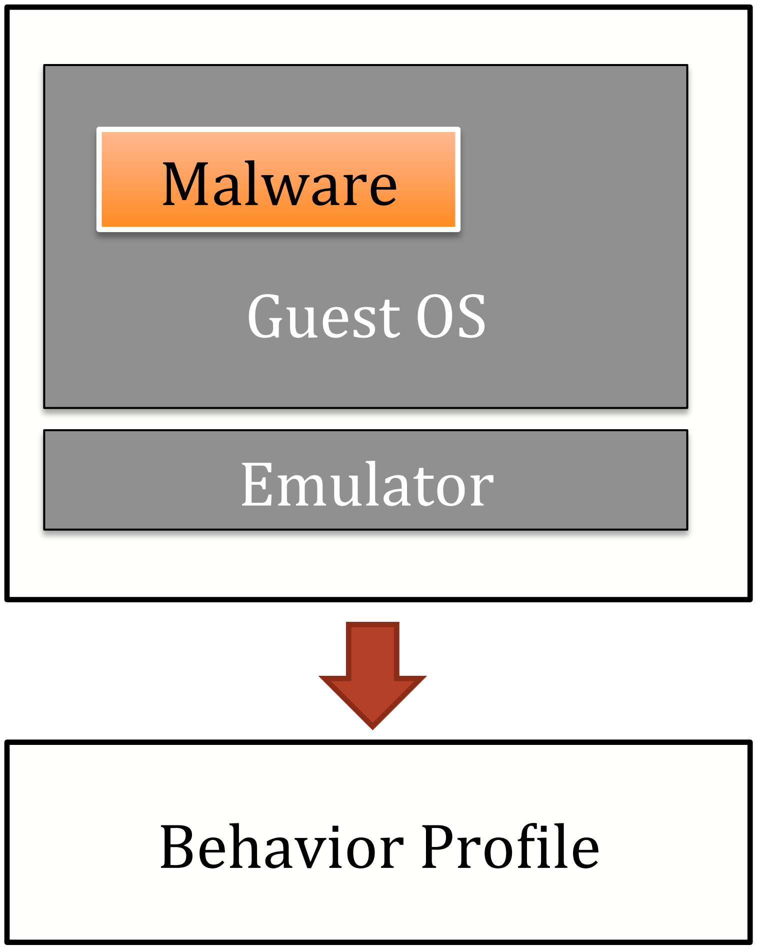
Implicit information flow

```
f(char sensitive)
{
  switch (sensitive) {

   case 0: y = 0 + z;
   case 1: y = 1 + z;
    ...
  }
}
```

- Dynamic implicit information flow tracking is more expensive. Why?
- We need to track information flowing through untaken path

Anubis



Behavior Profile

1. Tainted API calls
2. “Critical” control flow

Dynamic Approaches:
Some challenges and Pitfalls

- Coverage: dynamic analysis sees only one execution
- Trigger-based behavior: malicious behavior only triggered on certain inputs
- Speed:
  - Emulation-based analysis provides the most information, but is very slow (e.g., 5 minutes per sample)

Fallacies

- Dynamic analysis is best because you see the actual malware behaviors
- Static analysis is impossible on malware

There are no easy answers…

Putting it Together: Static and Dynamic Malware Detectors

- Detectors used signatures to detect malware (Static)
  - Attackers used meta and polymorphism to evade signature-detection
- Detectors used semantic-aware analysis (Static)
  - Attackers used semantic obfuscation, e.g., opaque constants
  - False positives
- Detectors use the signs of obfuscation (syntactic and semantic) as sign of malware
- Detector use information-flow to monitor runtime behavior (dynamic)
  - Can be expensive, esp. because of implicit flow
  - Implicit flow allows malware to evade detection
- Can we detect attempts to create implicit information flows?