A Semantics-based Approach to Malware Detection

Mila Dalla Preda – University of Verona, Italy
Mihai Christodorescu, Somesh Jha – University of Wisconsin, USA
Saumya Debray – University of Arizona, USA

17-19 Jan, POPL'07, Nice

A Few Basic Definitions

Malware represents malicious software.

Malware detector is a program $\mathcal{D}$ that determines whether another program P is infected with a malware M.

$$\mathcal{D}(P, M) = \begin{cases} \text{True} & \text{if } \mathcal{D} \text{ determines that } P \text{ is infected with } M \\ \text{False} & \text{otherwise} \end{cases}$$

A Few Basic Definitions

Malware represents malicious software.

Malware detector is a program $\mathcal{D}$ that determines whether another program P is infected with a malware M.

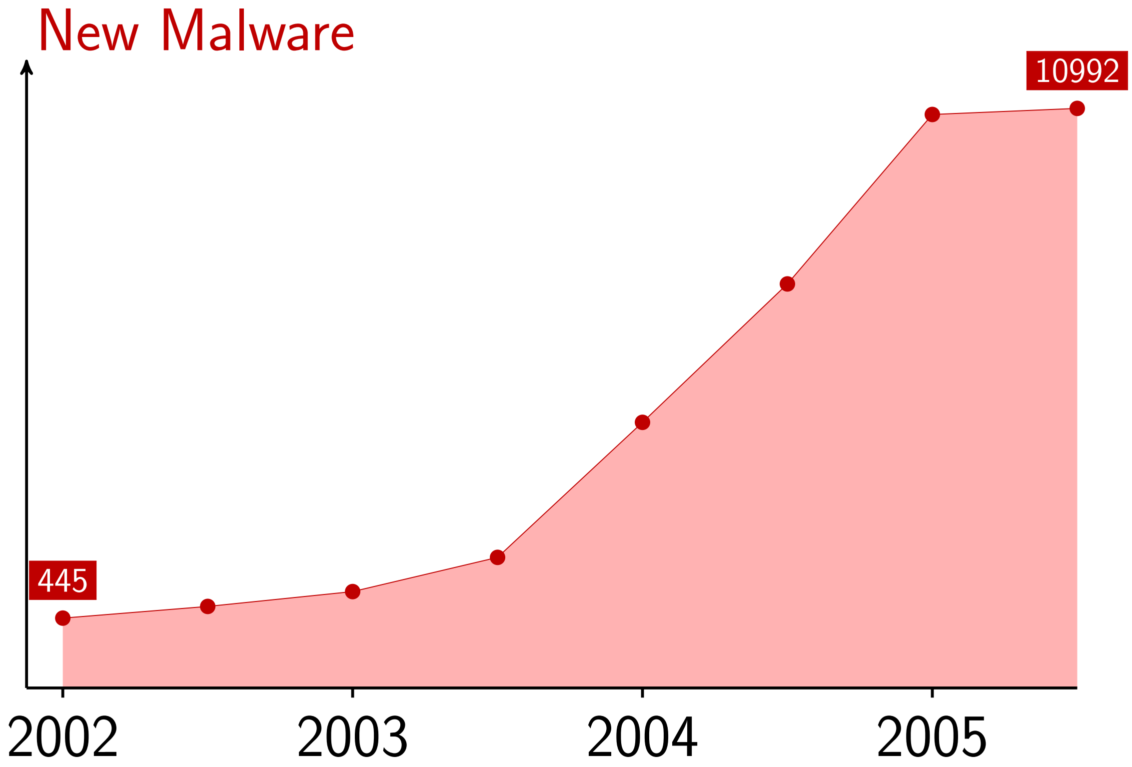
$$\mathcal{D}(P, M) = \begin{cases} \text{True} & \text{if } \mathcal{D} \text{ determines that } P \text{ is infected with } M \\ \text{False} & \text{otherwise} \end{cases}$$

An ideal malware detector detects all and only the programs infected with M, i.e., it is sound and complete.

- Sound = no false positives (no false alarms)
- Complete = no false negatives (no missed alarms)

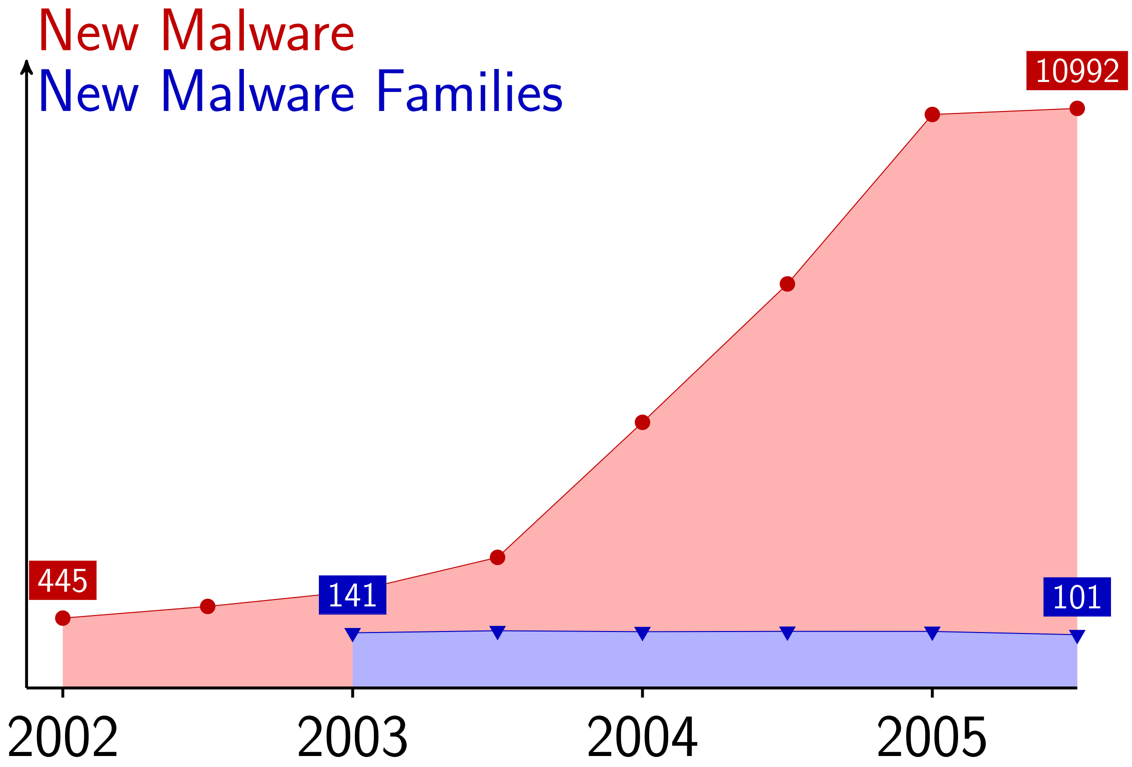
Malware Trends

There is more malware every year.



Malware Trends

There is more malware every year.



But the number of malware families has almost no variation.

Beagle family has 197 variants (as of Nov. 30).
Warezov family has 218 variants (as on Nov. 27).

The Malware Threat

Current detectors are signature-based:

$$P \text{ matches byte-signature } sig \quad \Rightarrow \quad P \text{ is infected}$$

Signature-based detectors, when sound, are not complete.

Malware writers use obfuscation to evade current detectors.

The Malware Threat

Current detectors are signature-based:

$$P \text{ matches byte-signature } sig \quad \Rightarrow \quad P \text{ is infected}$$

Signature-based detectors, when sound, are not complete.

Malware writers use obfuscation to evade current detectors.

Virus–antivirus “coevolution”

1. Malware writers create new, undetected malware.
2. Antimalware tools are updated to catch the new malware.
3. Repeat...

Common Obfuscations

- Nop insertion
- Register renaming
- Junk insertion
- Code reordering
- Encryption
- Reordering of independent statements
- Reversing of branch conditions
- Equivalent instruction substitution
- Opaque predicate insertion
- ... and many others...

Common Obfuscations

- Nop insertion
- Register renaming
- Junk insertion
- Code reordering
- Encryption
- Reordering of independent statements
- Reversing of branch conditions
- Equivalent instruction substitution
- Opaque predicate insertion
- ... and many others...

Obfuscation Example

(Pseudo-)Code:

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

Obfuscation Example

(Pseudo-)Code:

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

Obfuscated code (junk):

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

Obfuscation Example

(Pseudo-)Code:

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

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
```

Solutions?

Recent developments based on deep static analysis:

- Detecting Malicious Code by Model Checking [Kinder et al. 2005]
- Semantics-Aware Malware Detection [Christodorescu et al. 2005]
- Behavior-based Spyware Detection [Kirda et al. 2006]

Solutions?

Recent developments based on deep static analysis:

- Detecting Malicious Code by Model Checking [Kinder et al. 2005]
- Semantics-Aware Malware Detection [Christodorescu et al. 2005]
- Behavior-based Spyware Detection [Kirda et al. 2006]

Lack of a formal framework for assessing these techniques.

Our Contributions

Challenges:

- Many different obfuscations
- Obfuscations are usually combined
- Detection schemes usually rely on static/dynamic analyses

Our Contributions

Challenges:

- Many different obfuscations
- Obfuscations are usually combined
- Detection schemes usually rely on static/dynamic analyses

A framework for assessing the resilience to obfuscation of malware detectors.

- Obfuscation as transformation of trace semantics
- Malware detection as abstract interpretation of trace semantics
- Composing obfuscations vs. composing detectors

Two Worlds of Malware Detectors

Malware detector
on finite semantic structure

- Disassembler
- CFG construction
- Other analyses

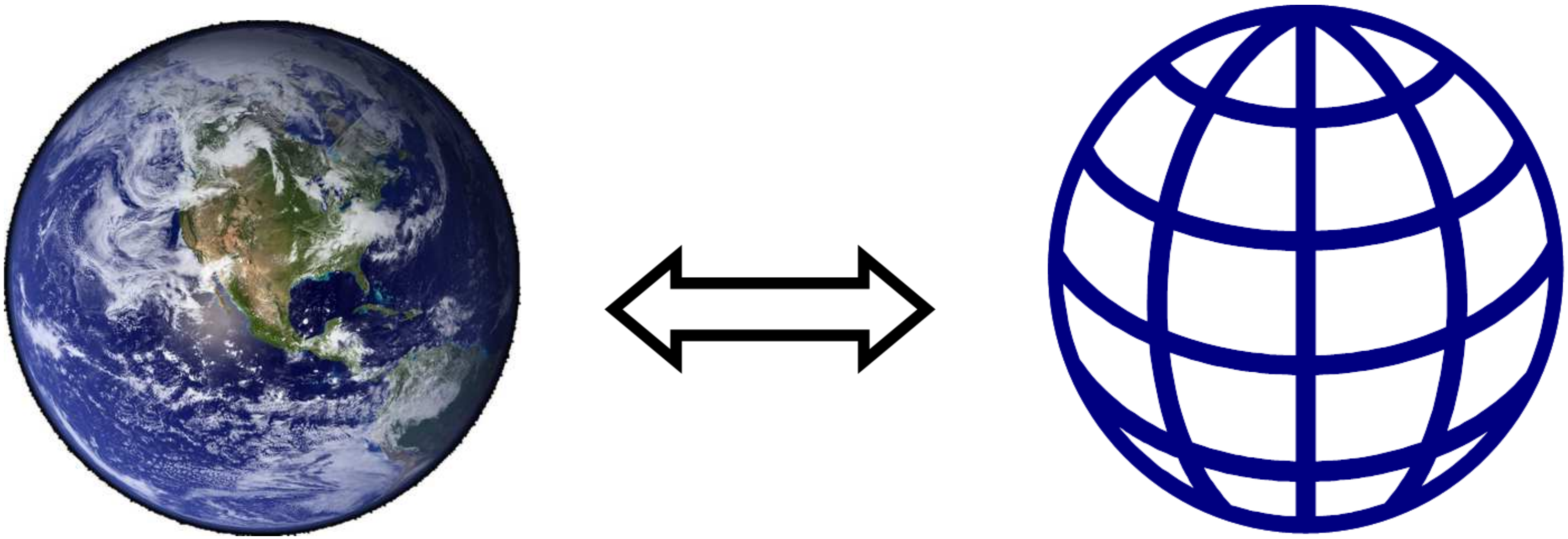
Two Worlds of Malware Detectors

Malware detector
on finite semantic structure

- Disassembler
- CFG construction
- Other analyses

Malware detector
on trace semantics

Two Worlds of Malware Detectors



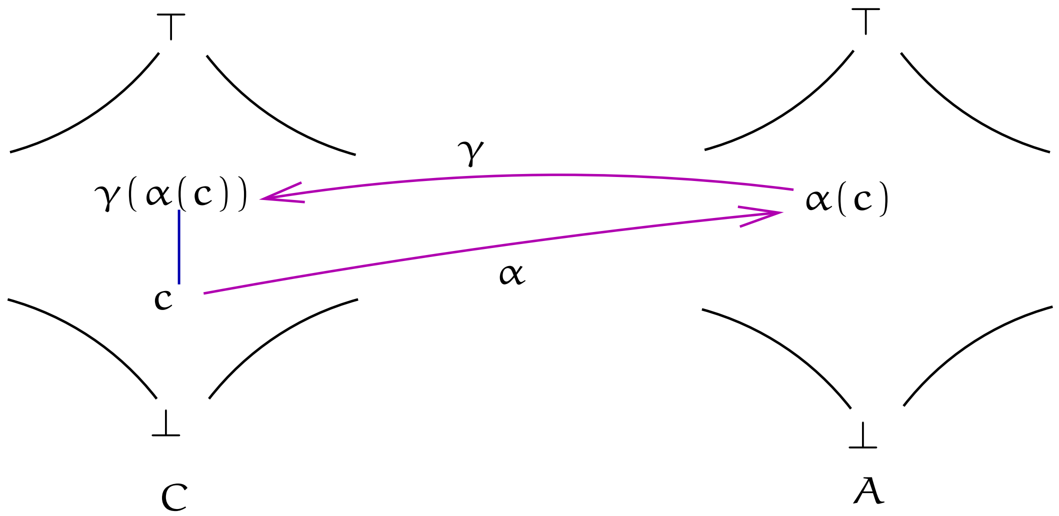
Malware detector
on finite semantic structure

- Disassembler
- CFG construction
- Other analyses

Malware detector
on trace semantics

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.

$\langle \mathit{Abs}(C), \sqsubseteq \rangle$ set of all possible abstract domains,

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

Outline

- Semantic Malware Detector
- Soundness and Completeness

- Classifying Obfuscations
- Composing Obfuscations

- Proving Soundness and Completeness

Semantic Malware Detector

A program P is infected by malware M, denoted $M \hookrightarrow P$
if (a part) of P execution is similar to that of M:

Semantic Malware Detector

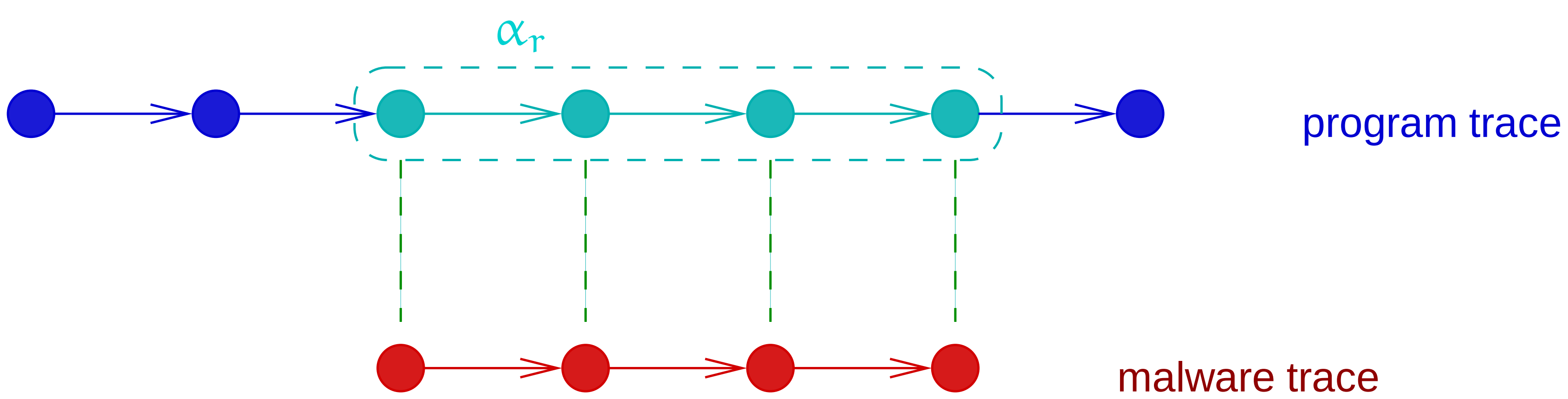
A program P is infected by malware M, denoted $M \hookrightarrow P$
if (a part) of P execution is similar to that of M:

$$S[\![M]\!] \quad \subseteq \quad S[\![P]\!]$$

Semantic Malware Detector

A program P is infected by malware M, denoted $M \hookrightarrow P$ if (a part) of P execution is similar to that of M:

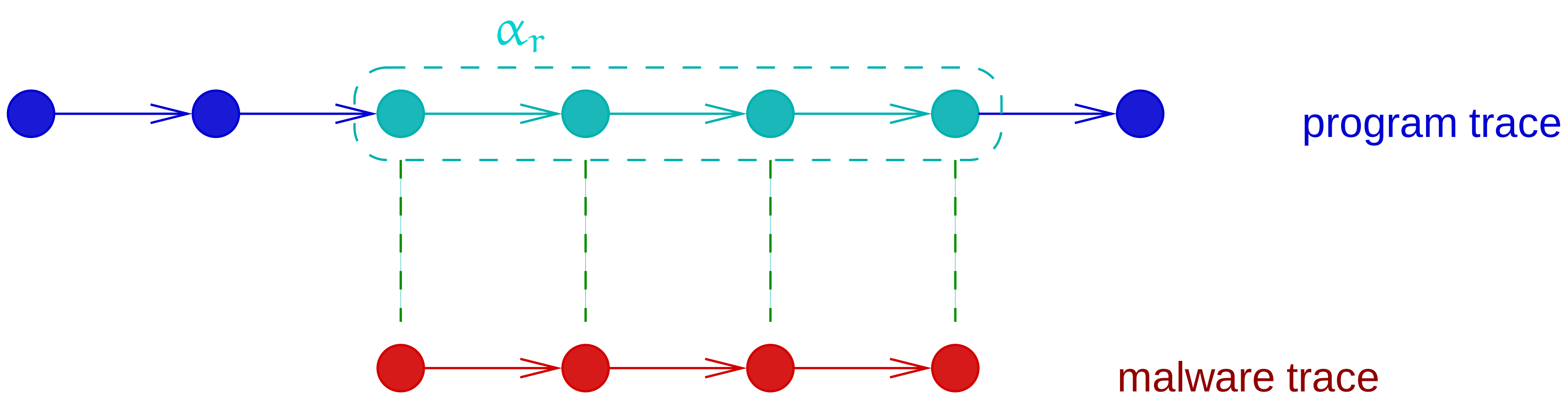
$$\exists \text{ restriction } r : S[\![M]\!] \subseteq \alpha_r(S[\![P]\!])$$



Semantic Malware Detector

A program P is infected by malware M, denoted $M \hookrightarrow P$ if (a part) of P execution is similar to that of M:

$$\exists \text{ restriction } r : S[\![M]\!] \subseteq \alpha_r(S[\![P]\!])$$



Vanilla Malware i.e. not obfuscated malware

Obfuscated Malware

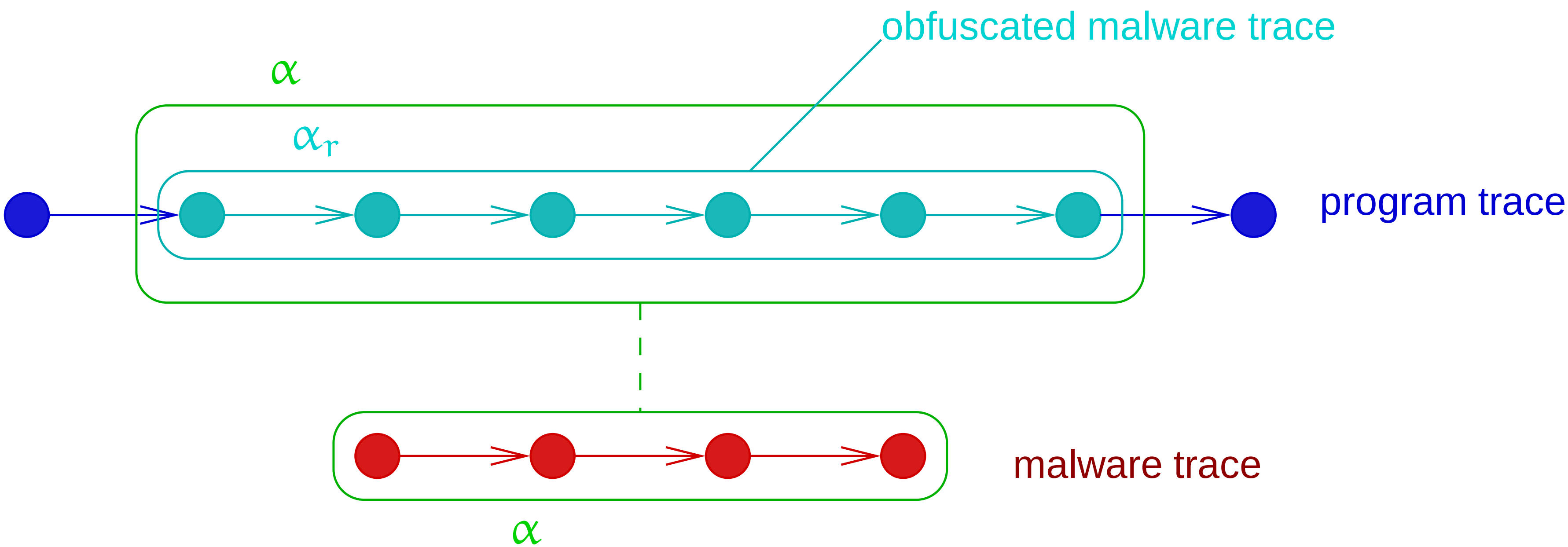
- $\mathcal{O} : \mathbb{P} \longrightarrow \mathbb{P}$ obfuscating transformation
- $\alpha : \mathit{Sem} \longrightarrow A$ abstraction that discards the details changed by the obfuscation while preserving maliciousness

$$\exists \text{ restriction } r : \alpha\left(S[\![M]\!]\right) \subseteq \alpha\left(\alpha_r(S[\![P]\!])\right)$$

Obfuscated Malware

- $\mathcal{O} : \mathbb{P} \to \mathbb{P}$ obfuscating transformation
- $\alpha : \mathit{Sem} \to A$ abstraction that discards the details changed by the obfuscation while preserving maliciousness

$$\exists \text{ restriction } r : \alpha\left(S[\![M]\!]\right) \subseteq \alpha\left(\alpha_r(S[\![P]\!])\right)$$



Sound vs. Complete

- Precision of the Semantic Malware Detector (SMD) depends on α

Sound vs. Complete

- Precision of the Semantic Malware Detector (SMD) depends on α

- A SMD on α is complete w.r.t. a set $\mathbb{O}$ of transformations if $\forall \mathcal{O} \in \mathbb{O}$:

$$\mathcal{O}(M) \hookrightarrow P \Rightarrow \begin{cases} \exists \text{ restriction } r : \\ \alpha(S[\![M]\!]) \subseteq \alpha(\alpha_r(S[\![P]\!])) \end{cases}$$

always detects programs that are infected (no false negatives)

Sound vs. Complete

- Precision of the Semantic Malware Detector (SMD) depends on α

- A SMD on α is complete w.r.t. a set $\mathbb{O}$ of transformations if $\forall \mathcal{O} \in \mathbb{O}$:

$$\mathcal{O}(M) \hookrightarrow P \Rightarrow \begin{cases} \exists \text{ restriction } r : \\ \alpha(S[\![M]\!]) \subseteq \alpha(\alpha_r(S[\![P]\!])) \end{cases}$$

  always detects programs that are infected (no false negatives)

- If α is preserved by $\mathcal{O}$ then the SMD on α is complete w.r.t. $\mathcal{O}$.

Sound vs. Complete

- Precision of the Semantic Malware Detector (SMD) depends on α

- A SMD on α is sound w.r.t. a set $\mathbb{O}$ of transformations if:

$$\left.\begin{array}{l}\exists \text{ restriction } r: \\ \alpha(S[\![M]\!]) \subseteq \alpha(\alpha_r(S[\![P]\!]))\end{array}\right\} \Rightarrow \exists \mathcal{O} \in \mathbb{O} : \mathcal{O}(M) \hookrightarrow P$$

never erroneously claims a program is infected (no false positives)

Outline

- Semantic Malware Detector
- Soundness and Completeness

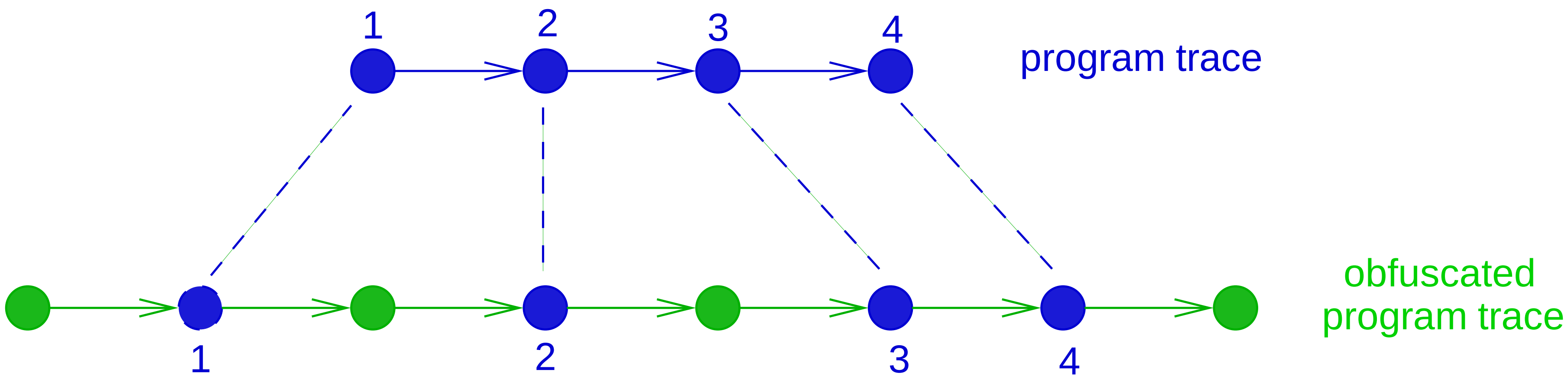
- Classifying Obfuscations
- Composing Obfuscations

- Proving Soundness and Completeness

Classifying Obfuscations

- $\mathcal{O} : \mathbb{P} \rightarrow \mathbb{P}$ is a conservative obfuscation if

  $\forall$ trace1 $\in S[\![P]\!]$, $\exists$ trace2 $\in S[\![\mathcal{O}[\![P]\!]]\!]$: trace1 is sub-sequence of trace2



Conservative Obfuscations

Abstraction α_c handles conservative obfuscations:

$$\alpha_c[X](Y) = X \cap \mathit{SubSequences}(Y)$$

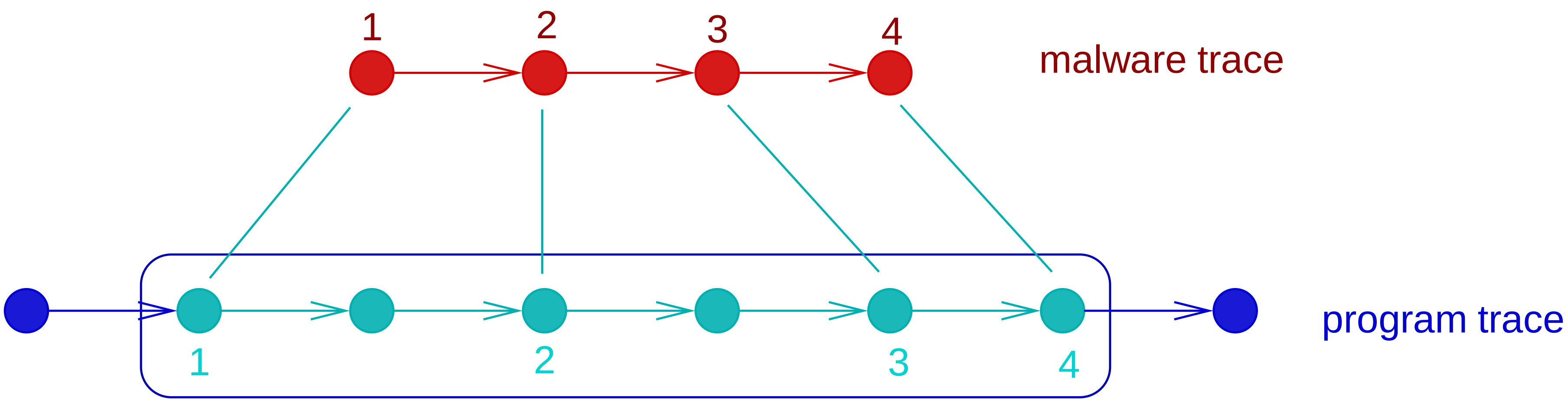
- The SMD on α_c is sound and complete w.r.t. conservative obfuscations

Conservative Obfuscations

Abstraction α_c handles conservative obfuscations:

$$\alpha_c[X](Y) = X \cap \mathit{SubSequences}(Y)$$

- The SMD on α_c is sound and complete w.r.t. conservative obfuscations

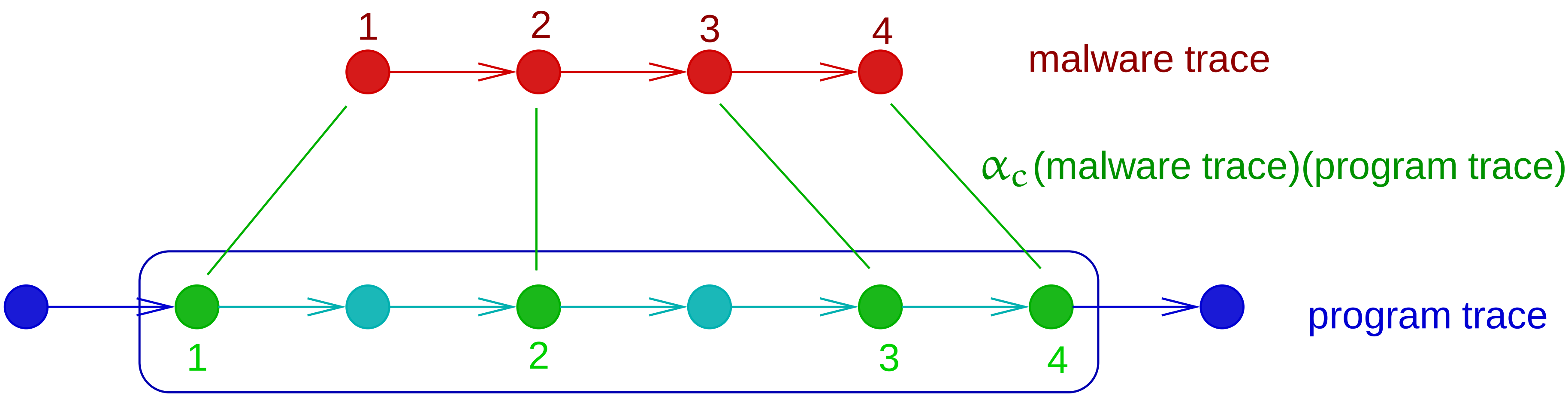


Conservative Obfuscations

Abstraction α_c handles conservative obfuscations:

$$\alpha_c[X](Y) = X \cap \mathit{SubSequences}(Y)$$

- The SMD on α_c is sound and complete w.r.t. conservative obfuscations



Conservative Obfuscations

Abstraction α_c handles conservative obfuscations:

$$\alpha_c[X](Y) = X \cap \mathit{SubSequences}(Y)$$

- The SMD on α_c is sound and complete w.r.t. conservative obfuscations

Abstraction α_c returns the set of malware traces
that are subsequences of some program trace

Classifying Common Obfuscations

- Nop insertion
- Register renaming
- Junk insertion
- Code reordering
- Encryption
- Reordering of independent statements
- Reversing of branch conditions
- Equivalent instruction substitution
- Opaque predicate insertion

Conservative Obfuscation Example

(Pseudo-)Code:

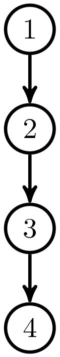
```
mov eax, [edx+0Ch]
push ebx
push [eax]
call ReleaseLock
```

Obfuscated code (junk + reordering):

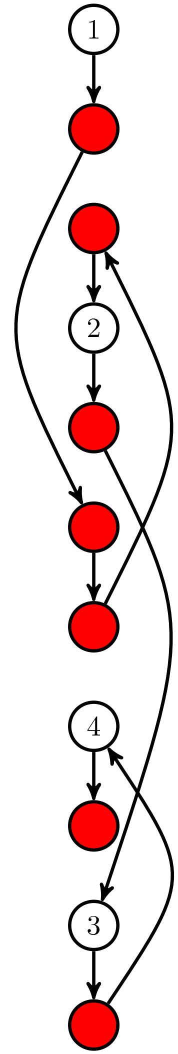
```
mov eax, [edx+0Ch]
jmp +3
push ebx
dec eax
jmp +4
inc eax
jmp -3
call ReleaseLock
jmp +2
push [eax]
jmp -2
```

Conservative Obfuscation Example

(Pseudo-)Code:

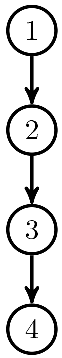


Obfuscated code (junk + reordering):

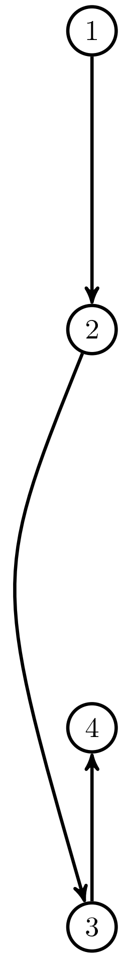


Conservative Obfuscation Example

(Pseudo-)Code:



Obfuscated code (junk + reordering):



Non-Conservative

Approach 1: Find a canonical transformation

Non-Conservative

Approach 1: Find a canonical transformation

(Pseudo-)Code:	Obfuscated Code (Renaming):
`mov eax, [edx+0Ch]`	`mov edi, [eax+0Ch]`
`push ebx`	`push ecx`
`push [eax]`	`push [edi]`
`call ReleaseLock`	`call ReleaseLock`

Non-Conservative

Approach 1: Find a canonical transformation

(Pseudo-)Code:

```
mov R1, [R2+0Ch]
push R3
push [R1]
call ReleaseLock
```

Obfuscated Code (Renaming):

```
mov R1, [R2+0Ch]
push R3
push [R1]
call ReleaseLock
```

Non-Conservative

- Program infection: $M \hookrightarrow P$ if $\exists$ restriction $r : S[\![M]\!] \subseteq \alpha_r(S[\![P]\!])$

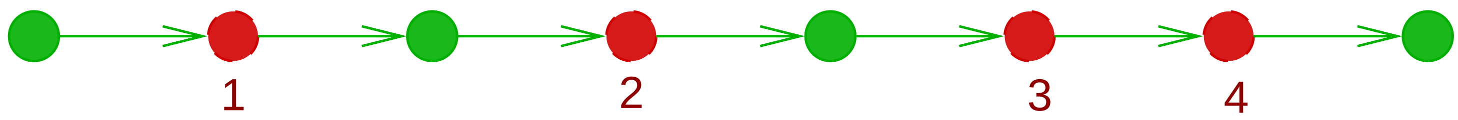
Non-Conservative

- Program infection: $M \hookrightarrow P$ if $\exists$ restriction $r : S[\![M]\!] \subseteq \alpha_r(S[\![P]\!])$

Approach 2: Futher abstractions

- Interesting Malware States: $I \subseteq \mathit{States}[\![M]\!]$:

$$M \hookrightarrow P \text{ if } \exists r : \alpha_I\ (S[\![M]\!]) \subseteq \alpha_I\ (\alpha_r(S[\![P]\!]))$$



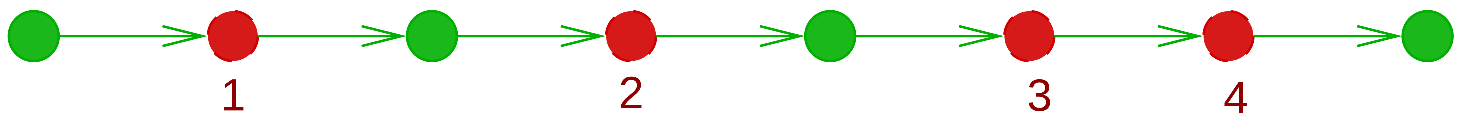
Non-Conservative

- Program infection: $M \hookrightarrow P$ if $\exists$ restriction $r : S[\![M]\!] \subseteq \alpha_r(S[\![P]\!])$

Approach 2: Futher abstractions

- Interesting Malware States: $I \subseteq \mathit{States}[\![M]\!]$:

$$M \hookrightarrow P \text{ if } \exists r : \alpha_I\,(S[\![M]\!]) \subseteq \alpha_I\,(\alpha_r(S[\![P]\!]))$$



- Interesting Malware Traces: $X \subseteq S[\![M]\!]$

$$M \hookrightarrow P \text{ if } \exists r : X \subseteq \alpha_r(S[\![P]\!])$$

Composition

- Malware writers combine different obfuscations to avoid detection

- The property of being conservative is preserved by composition
  $\Rightarrow$ abstraction α_c

- Under certain assumptions we can handle the composition of non-conservative obfuscations

Outline

- Semantic Malware Detector
- Soundness and Completeness
- Classifying Obfuscations
- Composing Obfuscations
- Proving Soundness and Completeness

Proving Soundness/Completeness of MD

- Identifying the class of obfuscators to which a malware detector is resilient can be a complex and error-prone task.
- Obfuscators and detectors can be expressed on executions traces.

A detector is resilient to an obfuscator if
it can "abstract away" the obfuscator's effect on the program.

- Case study: Semantics-Aware Malware Detection Algorithm proposed by [Christodorescu et al. 2005].
  - Complete for code reordering
  - Complete for junk insertion
  - Complete for variable renaming

Conclusions

- Malware detection as abstraction of program semantics
  vs.
  Obfuscation as transformation of program semantics
- We can now determine:
  - Whether a detector is resilient to a set of obfuscations
  - How complex a detector has to be to handle a given obfuscation
- Open Problems:
  - Can we handle some interesting classes of non-conservative obfuscations?
  - How does one design a semantic detector based on trace semantics?
  - Connecting cryptographic and program analysis views of obfuscation

Thank you!