

18732: Secure Software Systems

Malware Detection

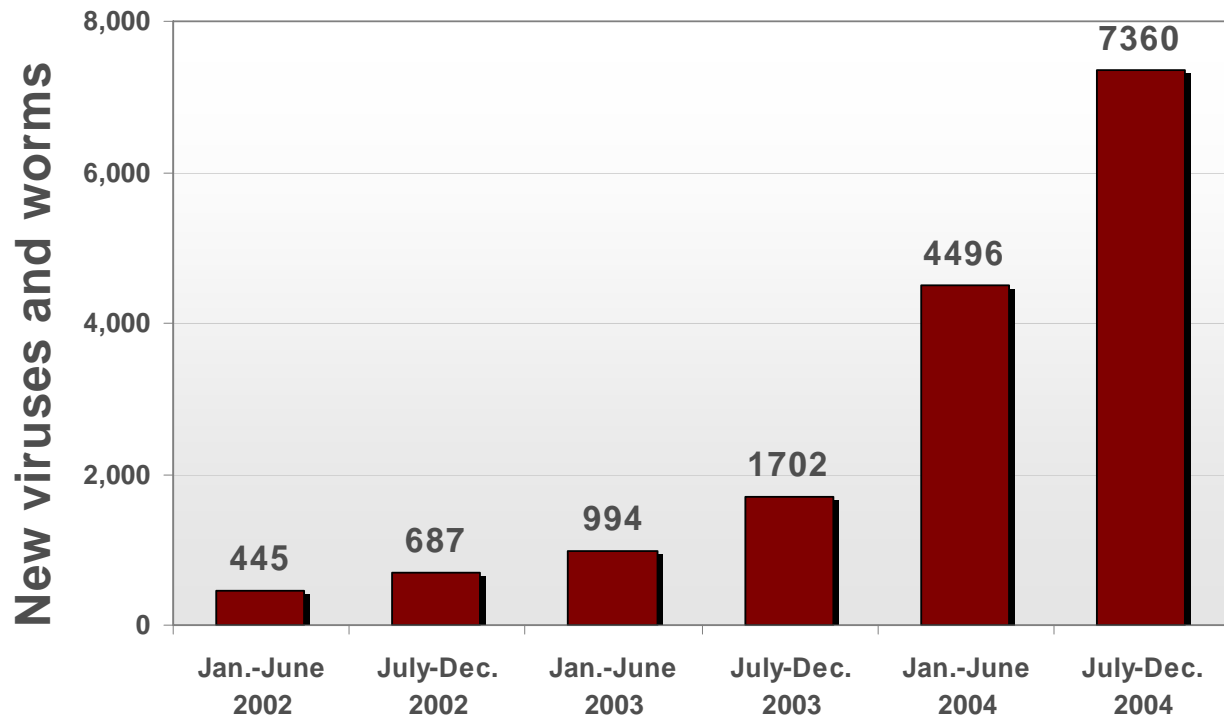
Anupam Datta
CMU

Fall 2010

Malicious Code Problem

Malware is everywhere.

Source: Symantec Internet Security Threat Report (vol. VII)



- Large malware families.

Malware types

- Viruses, internet worms, spyware/adware, botnets, keyloggers, network sniffers, mass mailers, rootkits,...
- Commercial detection tools from companies like Symantec, McAfee

Evasion Techniques

- Obfuscations applied to malicious code make evasion very easy.
- Attacker's goal:
Preserve (subset of) behavior.
 - Transformations of code and data.
 - Addition of new code and data.
- Easy to create variants in large numbers.

An Example

Goal: detect any mass-mailing virus.

1. Detect email capabilities.
2. Detect self-propagation.

```
s = socket( ... );  
connect( s );  
...  
sprintf( buf, "EHLO %s", dnsname );  
send( s, buf );
```

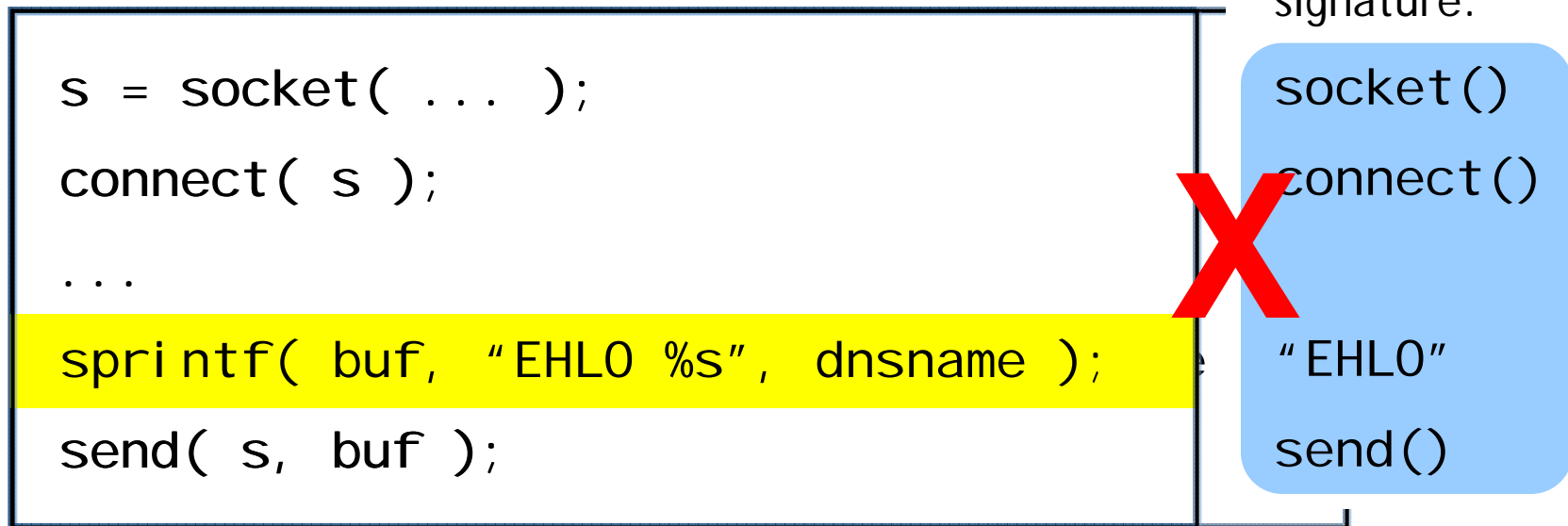
Possible syntactic signature:

```
socket()  
connect()  
  
"EHLO"  
send()
```

Adapted from MyDoom.Q

Variant 1: String Manipulation

- Hide known constants from virus scanner.
- Syntactic signature does not match.



Adapted from MyDoom.L

Variant 2: String Obfuscation

- Hide known constants using simple encryption techniques (e.g. ROT13, XOR).
- Syntactic signature does not match.

```
s = socket( ... );  
connect( s );  
...  
send(buf, buf13("EHLO", s, dnsname));  
send(s, buf);
```

Syntactic
signature:

socket()

connect()

"EHLO"

send()



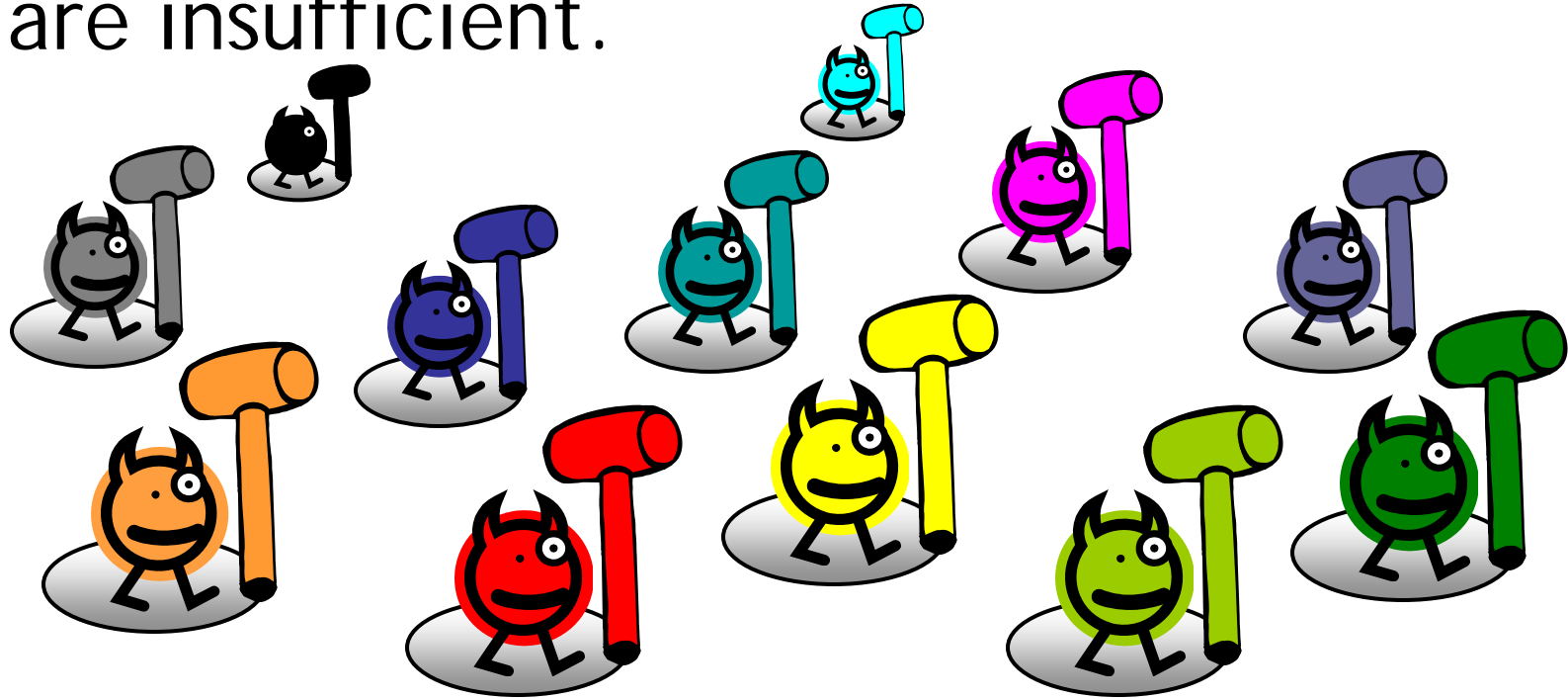
Adapted from MyDoom.G

Other common obfuscations

- Instruction reordering
- Register renaming
- Garbage (nop) insertion
- Substitution of equivalent instructions

The Current Solution

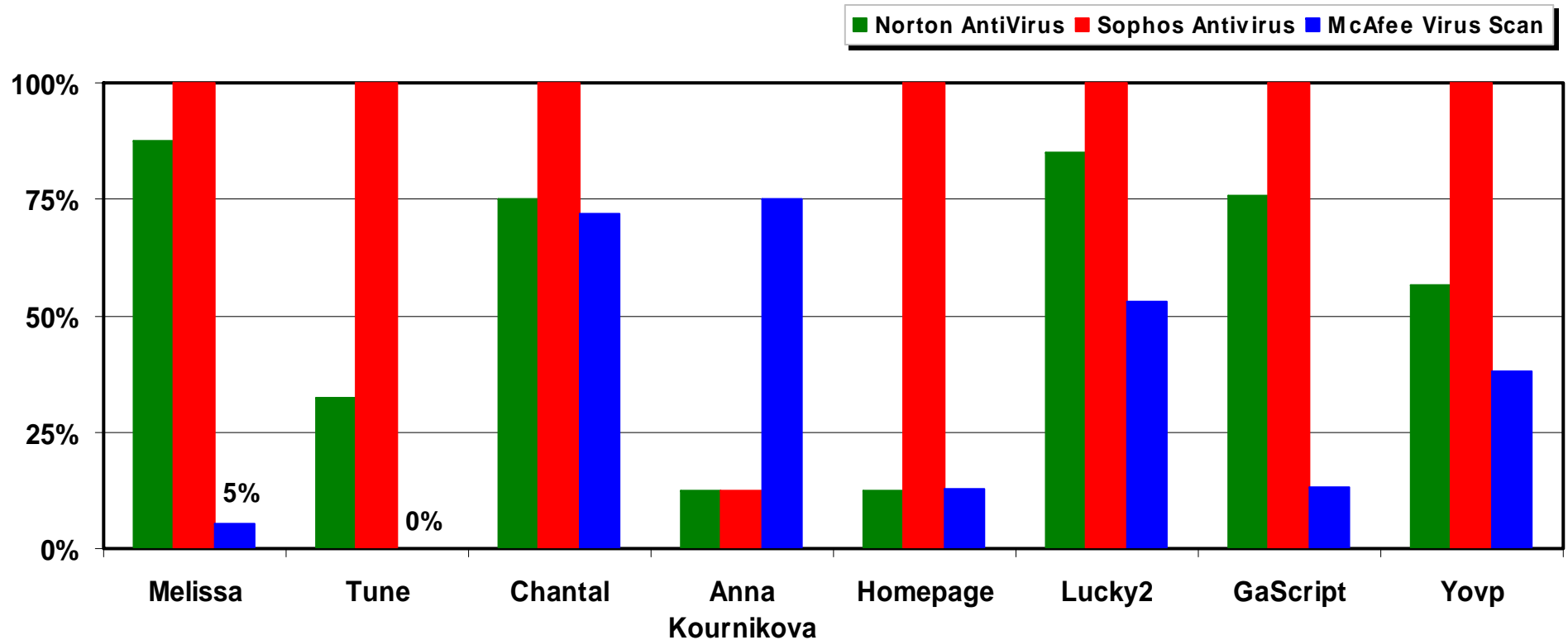
- Syntactic signatures (regular expressions) are insufficient.



- Overfitting, easy to evade.

No Resilience to Obfuscations

False Negative Rate for Obfuscated Worms



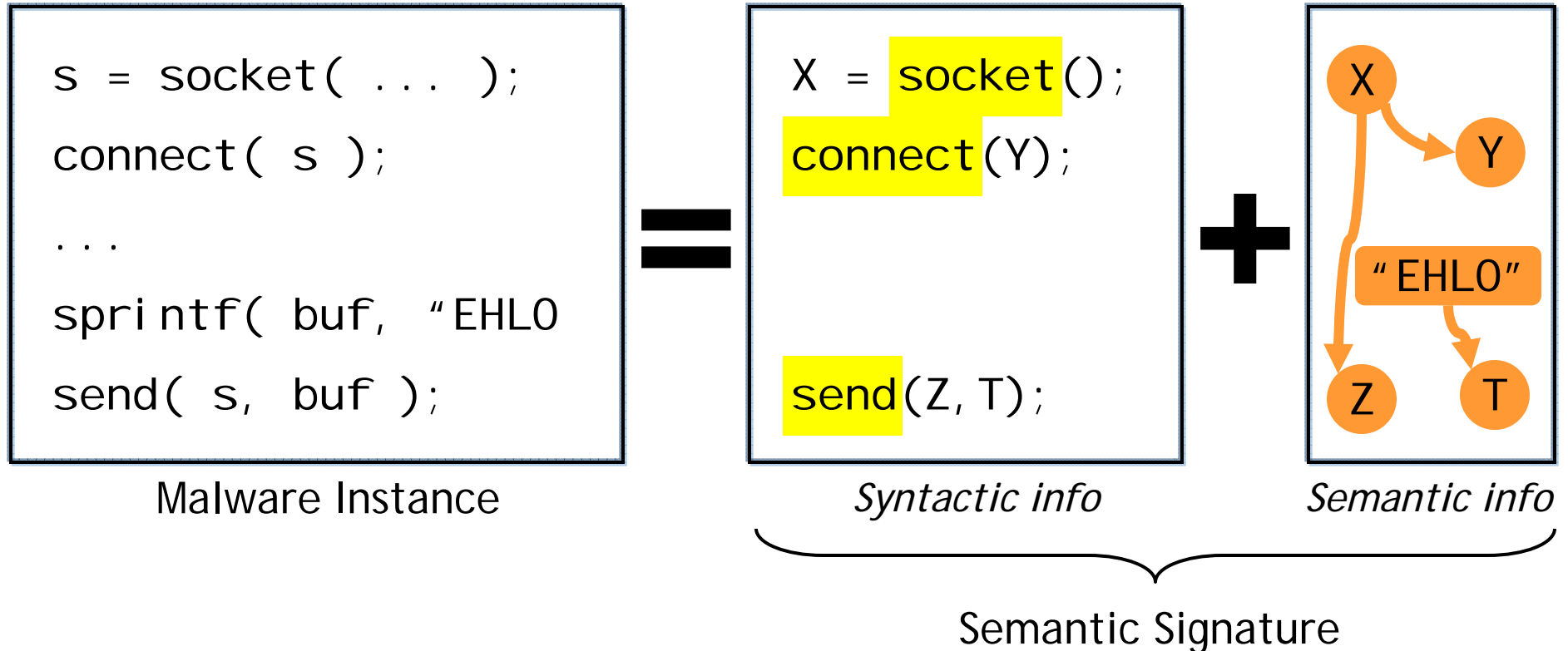
Source: "Testing Malware Detectors" (ISSTA 2004)

Contributions

- Introduce **semantic signatures** that combine syntactic and semantic information.
- Develop a prototype for malware detection using semantic signatures.
- Show, empirically, that **one semantic signature can detect a malware family**.

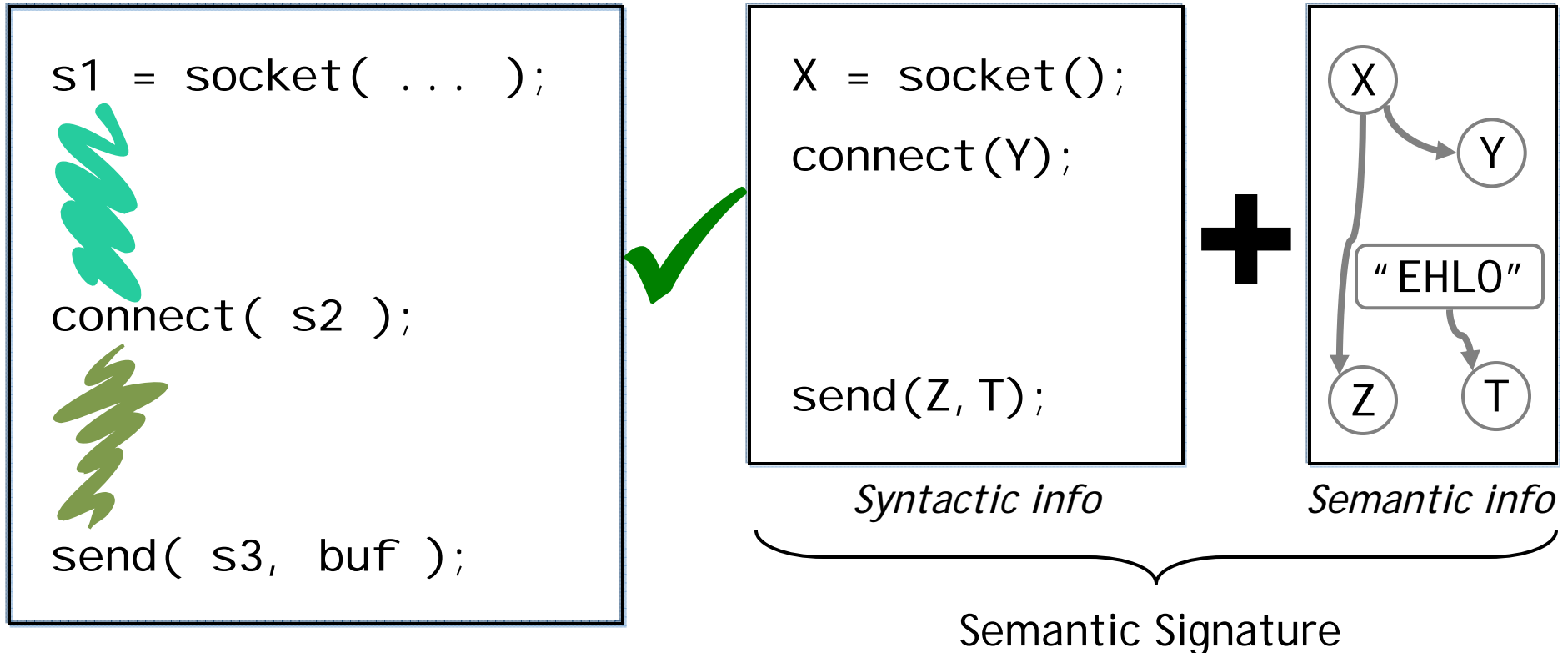
Semantic Signatures

- Goal of attacker: same behavior in different form.



Power of Semantic Signatures

- Detect any variant that uses the same sequence of instructions.



Semantics-Aware Detection

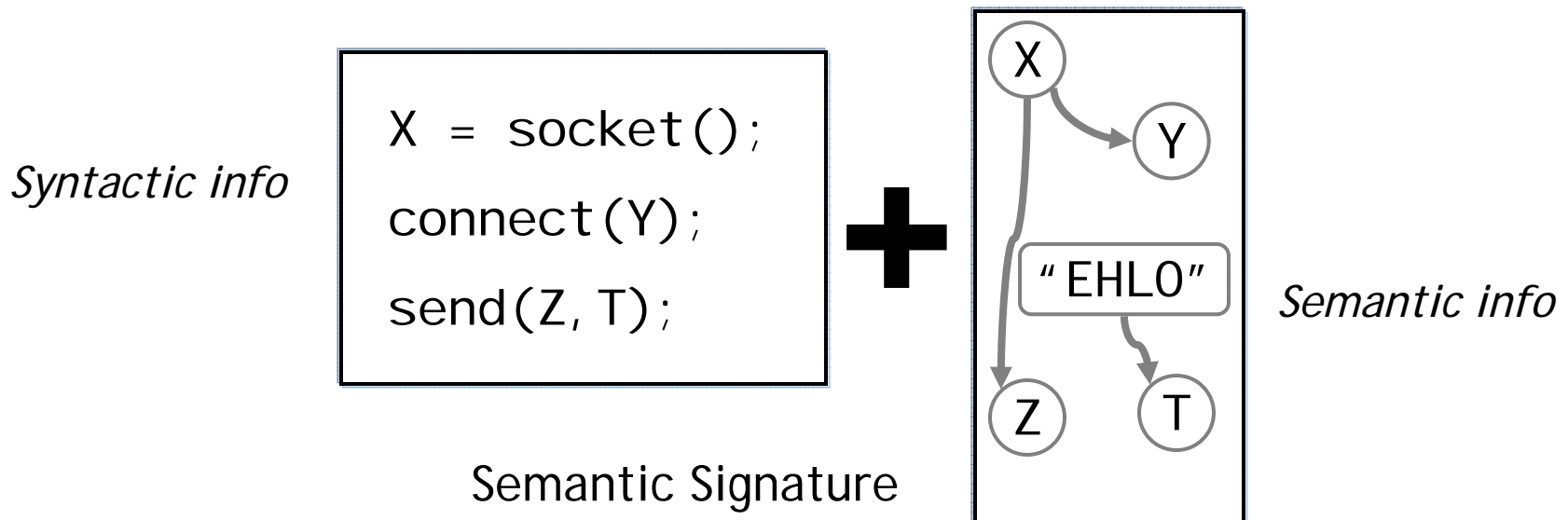
- We have a new detection model:
Semantic signatures combine syntactic and semantic information.

1. Can we build it?

2. Does it work?

A Semantics-Aware Detector

- Match the syntactic constructs
 - template nodes to program nodes
- Check for semantic information
 - Value preservation on def-use paths

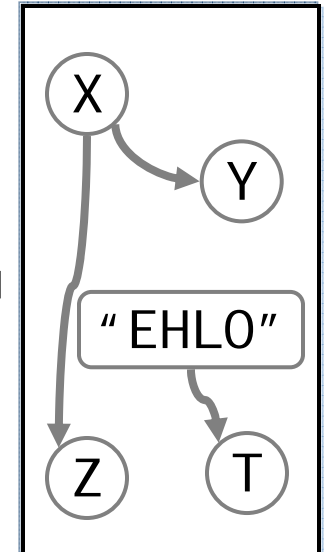
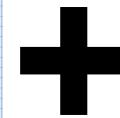


Checking for Semantic Info

Program:

```
1  s1 = socket( ... );  
2  a[ i ++ ] = s1;  
3  s2 = a[ i - 1 ];  
4  connect( s2 );  
...
```

```
X = socket();  
connect(Y);  
...
```



- Check that "s2 has the same value as s1"
or check the **value predicate**:
 $\text{value}(s1 \text{ after line } 1) == \text{value}(s2 \text{ before line } 4)$

Checking a Value Predicate

Program:

```
1  s1 = socket( ... );  
2  a[ i ++ ] = s1;  
3  s2 = a[ i - 1 ];  
4  connect( s2 );  
   ...
```

Value predicate:

```
value(s1 after line 1)  
    ==  
value(s2 before line 4)
```

Equivalent condition:

Lines 2 and 3 are a **semantic nop** with respect to the value predicate.

Value Predicates

Template

```
1 s = socket( ... );  
2 connect( s );  
3 send( s, buf );
```

Conditions

value(s@1) = value(s@2)
value(s@1) = value(s@3)
value(buf[0..4]@3) = "EHLO "

Program

```
1 x = socket( ... );  
2 connect( x );  
3 ...  
4 sprintf( tmp,  
    "E%s %s", "HLO",  
    dnsname );  
5 send( x, tmp );
```

Conditions

value(x@1) = value(x@2)
value(x@1) = value(x@5)
value(tmp[0..4]@5) = "EHLO "

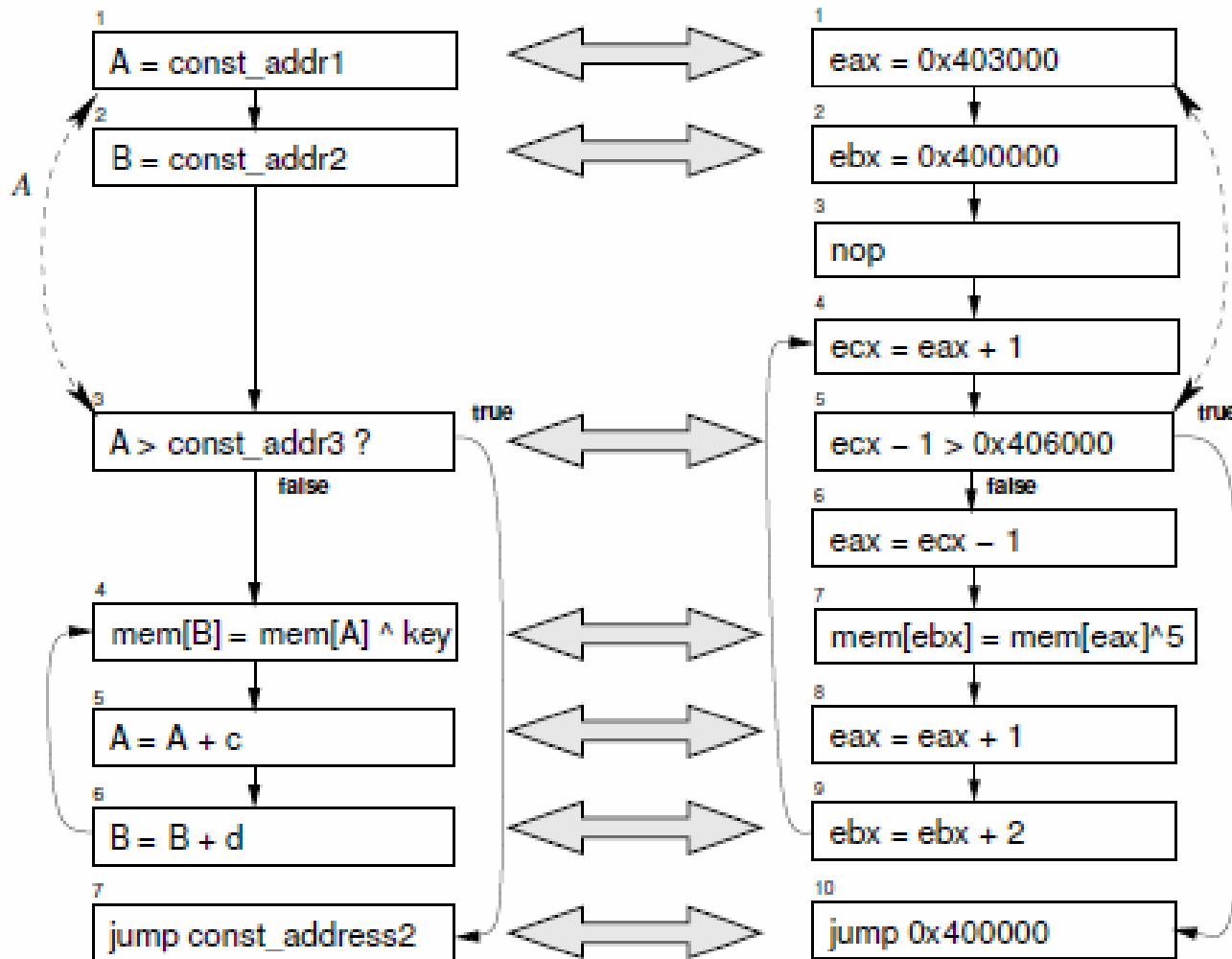
If conditions are satisfied, then program exhibits malicious behavior specified by the template

Another Example

Two steps:

1. Matching template nodes to program nodes
2. Value preservation on def-use chains

Example Template & Program



Template

Program

Example Matching Substitutions

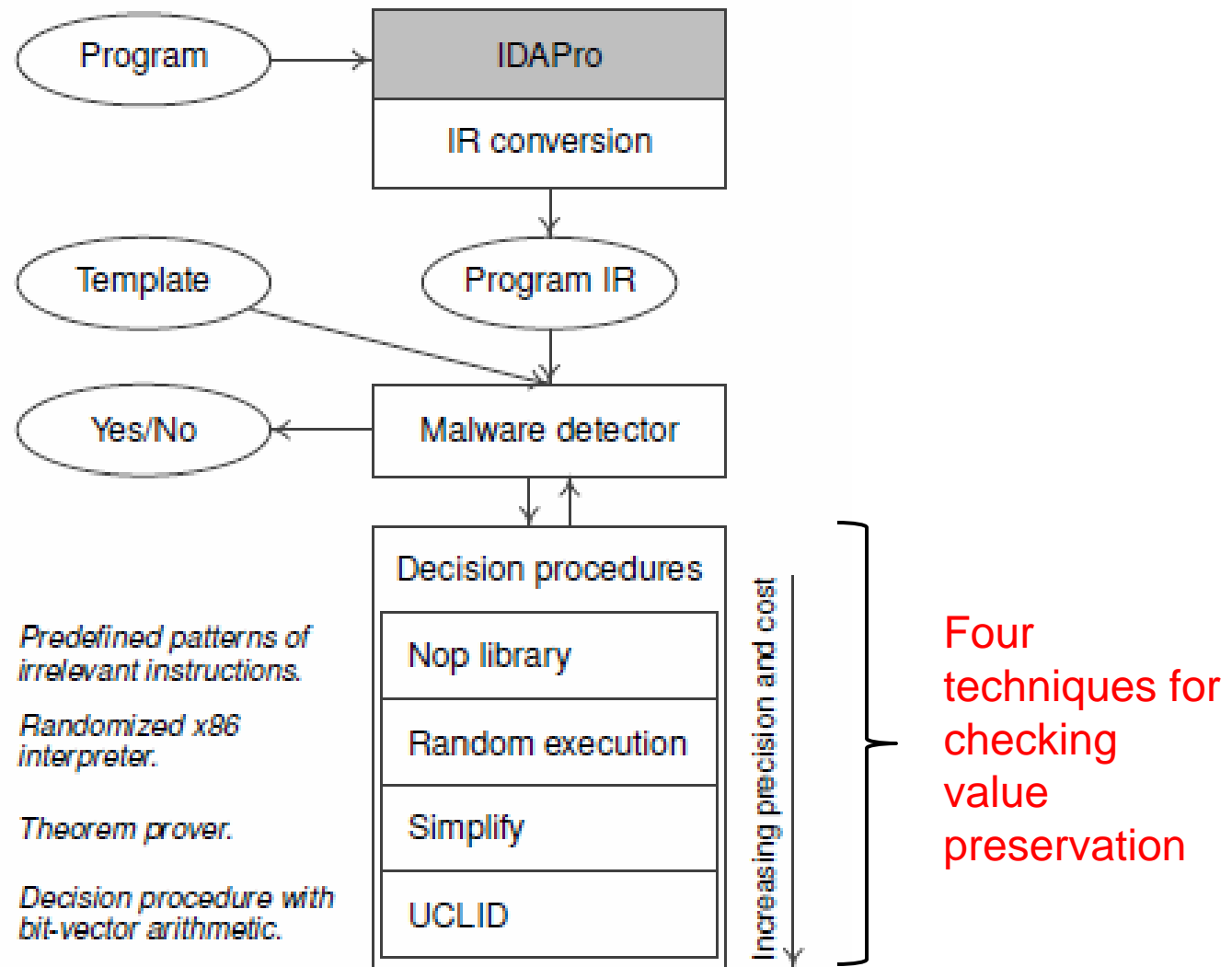
<i>Unified nodes</i>		<i>Bindings</i>
T1	P1	A ← eax const_addr1 ← 0x403000
T2	P2	B ← ebx const_addr2 ← 0x400000
T3	P5	A ← ecx - 1 const_addr3 ← 0x406000

Value Predicate for Def-Use Chain

<i>Def-use chain and value predicate</i>	<i>Program fragment</i>
$T1 \xrightarrow{A} T3$ $val_{post(\Gamma)}(ecx - 1)$ $= val_{pre(\Gamma)}(eax)$	2: <code>ebx = 0x400000</code> 3: <code>nop</code> 4: <code>ecx = eax + 1</code>

Value preservation on def-use chains can be checked using a number of techniques

Implementation



Checking Value Predicate using Theorem Prover



Evaluation of Prototype

- Developed signatures for several families of worms.
- No false positives.
- Improved resilience to common obfuscations.

Evaluation of Semantic Signatures

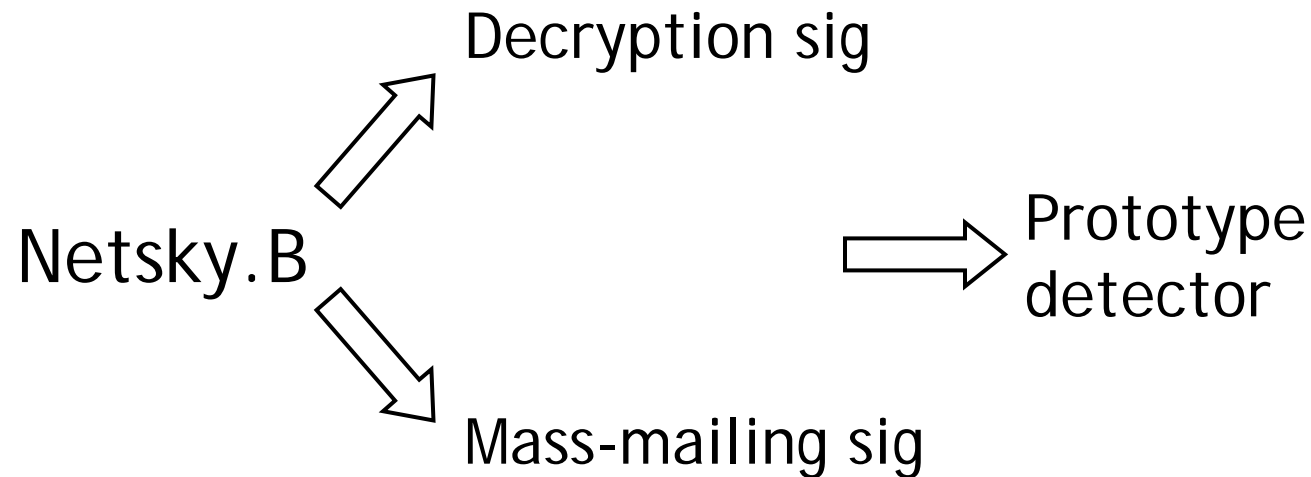
Created 2 templates describing decryption and mass-mailing features.

- Netsky B, C, D, O, P, T, W
- Beagle I, J, N, O, P, R, Y
- Sober A, C, D, E, F, G, I

Measured false positive rate.

Measured resilience to obfuscation.

Evaluation of Semantic Signatures



Netsky.C	✓
Netsky.D	✓
Netsky.O	✓
Netsky.P	✓
Netsky.T	✓
Netsky.W	✓

McAfee uses individual signatures for each worm.

Semantic signatures provide forward detection.

Performance

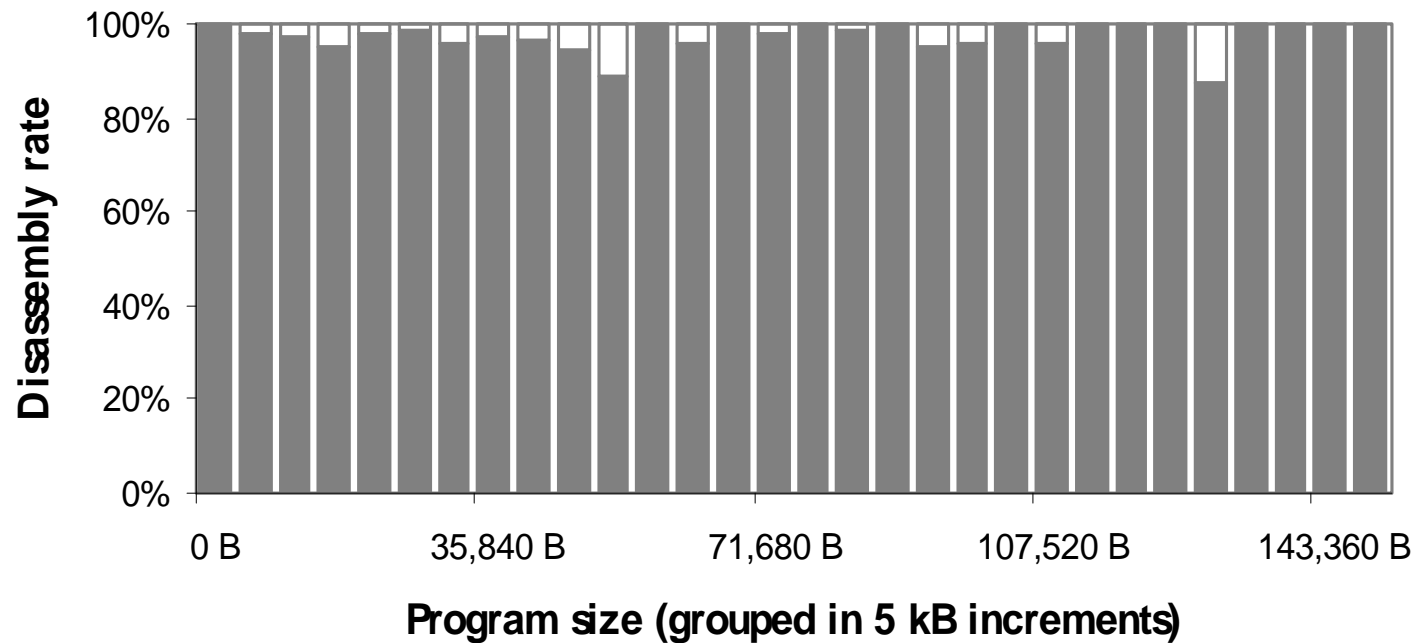
- Prototype is slower than commercial anti-virus tools.

<i>Malware Family</i>	<i>Running Time</i>	
	<i>Average</i>	<i>Std. Deviation</i>
Netsky	99.57 s	41.01 s
Beagle	56.41 s	40.72 s

- Plenty of room for improvement.
e.g. disassembler: 25% of time.

Evaluation: False Positive Rate

- Tested the semantic signatures on 2,000 benign Windows binaries.
- False positive rate: **0%**



Evaluation: Obfuscation Resilience

- Different types garbage insertion applied to Beagle.Y to obtain more variants.

<i>Obfuscation Type</i>	<i>Semantics-Aware Detection</i>		<i>McAfee</i>
	<i>Average Time</i>	<i>Detection Rate</i>	
Nop insertion	74.81 s	100%	75%
Stack op. insertion	159.10 s	100%	25%
Math op. insertion	186.50 s	95%	5%

Strengths

- Detection succeeds in the presence of following obfuscations.
 - Code reordering
 - Detected because CFG captures execution order
 - Register renaming
 - Detected because template variables can be unified with different register names
 - Garbage insertion
 - Detected because value predicates on def-use chains are still the same

Limitations

- Limited support for equivalent code sequences.

`a = b * 2`

X

`a = b << 1`

- In the semantic signature, order of instructions is significant.

`a = b + 1`

`c = c + 1`

X

`c = c + 1`

`a = b + 1`

Where do we go from here?

Up to now: **Syntactic signature detection**

This work: **Semantics-aware detection**

Future: Equivalent code sequences
 Detection of self-replication
 Better performance

Question: Will we ever win the arms race?

THANKS! QUESTIONS?

Sources

- Mihai Christodorescu, Sanjit Seshia, Somesh Jha, Dawn Song, Randal E. Bryant, Semantics-Aware Malware Detection, *IEEE Symposium on Security and Privacy, Oakland, California, May 2005.*
- Mihai Christodorescu provided many slides used in this lecture