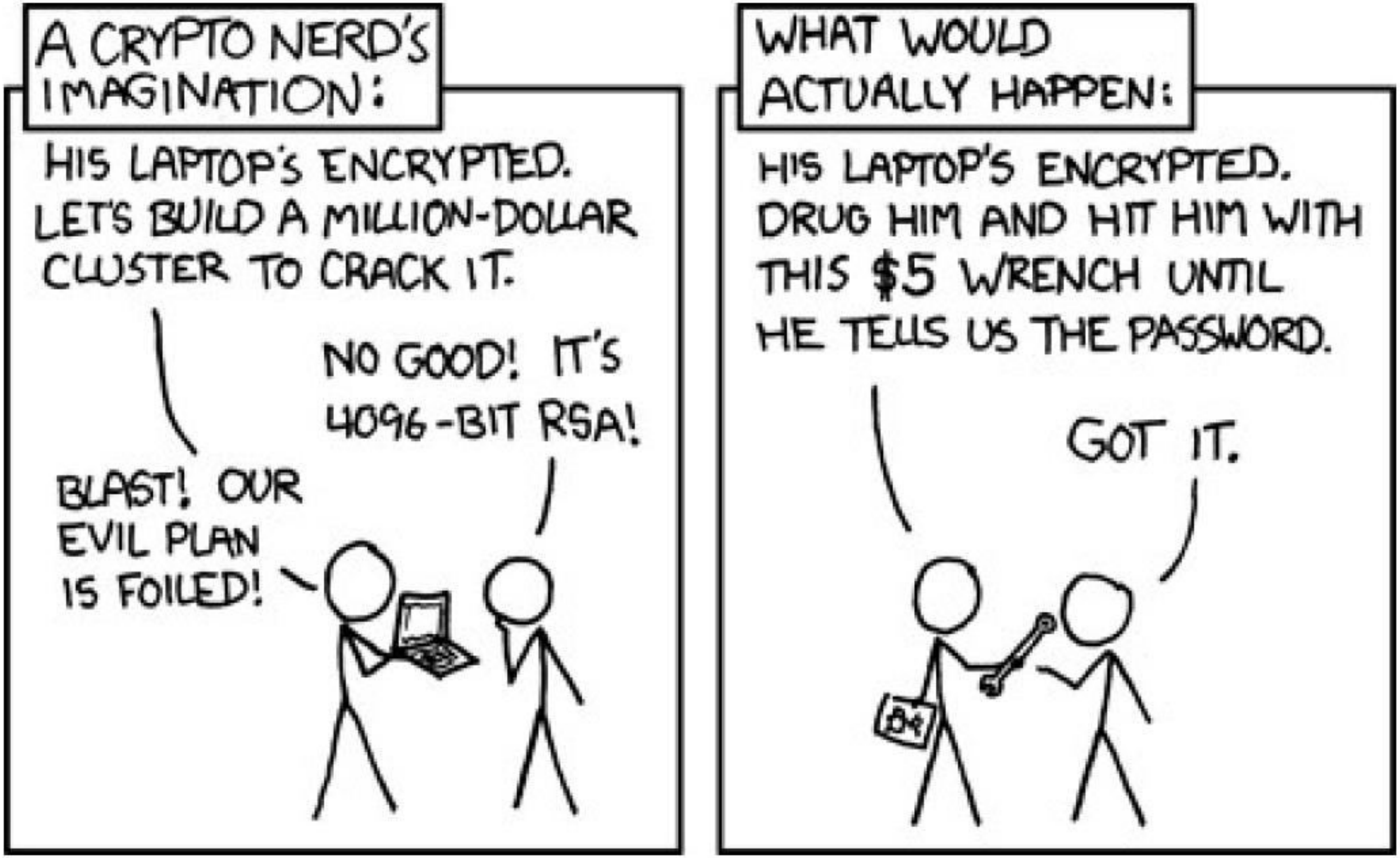
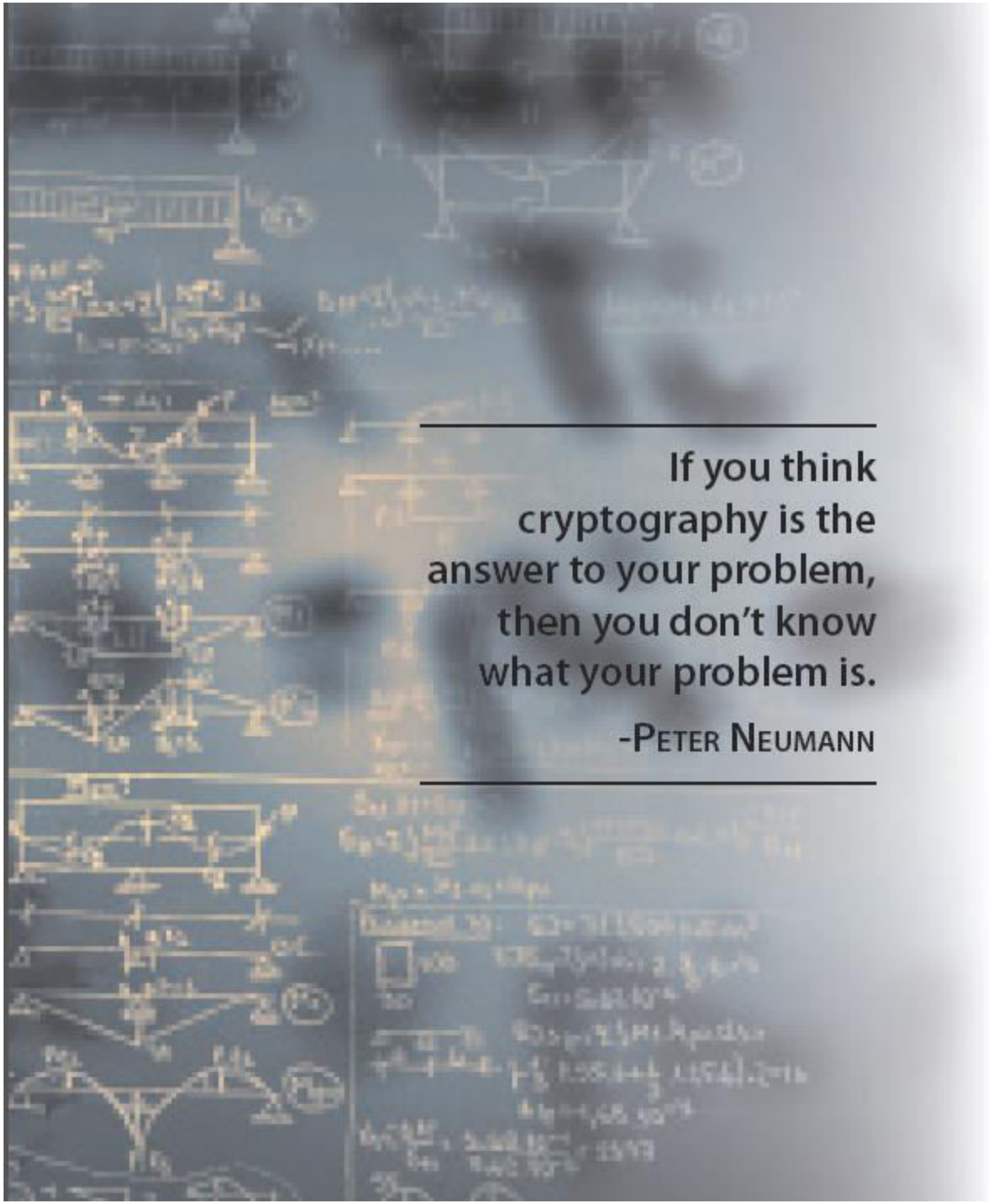
Malware Analysis, Detection

Three Lectures


A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.



If you think cryptography is the answer to your problem, then you don't know what your problem is.

-Peter Neumann

Organization

- Malware Threat and Impact
  - Difficulty of detecting Malware
- Malware Detection
  - Syntactic, Semantic, Behavioural (Static and Dynamic)
  - Behavioural Characterization: Metamorphic
  - Anti virus detection Issues
- Dimensions
  - Side Channel Attacks
- Protecting SCADA
- Threat Landscape
- Cyber War
- Summary

Malware

- As computers and networked systems have become an integral part of our daily lives, securing information from misuse, unauthorized access and modification has become very important
  - willful: tampering by the user
  - unintentional: execution of a malicious application
- Malware is software designed to infiltrate a computer system without the owner's informed consent and cause damage

Malware

Malicious Code

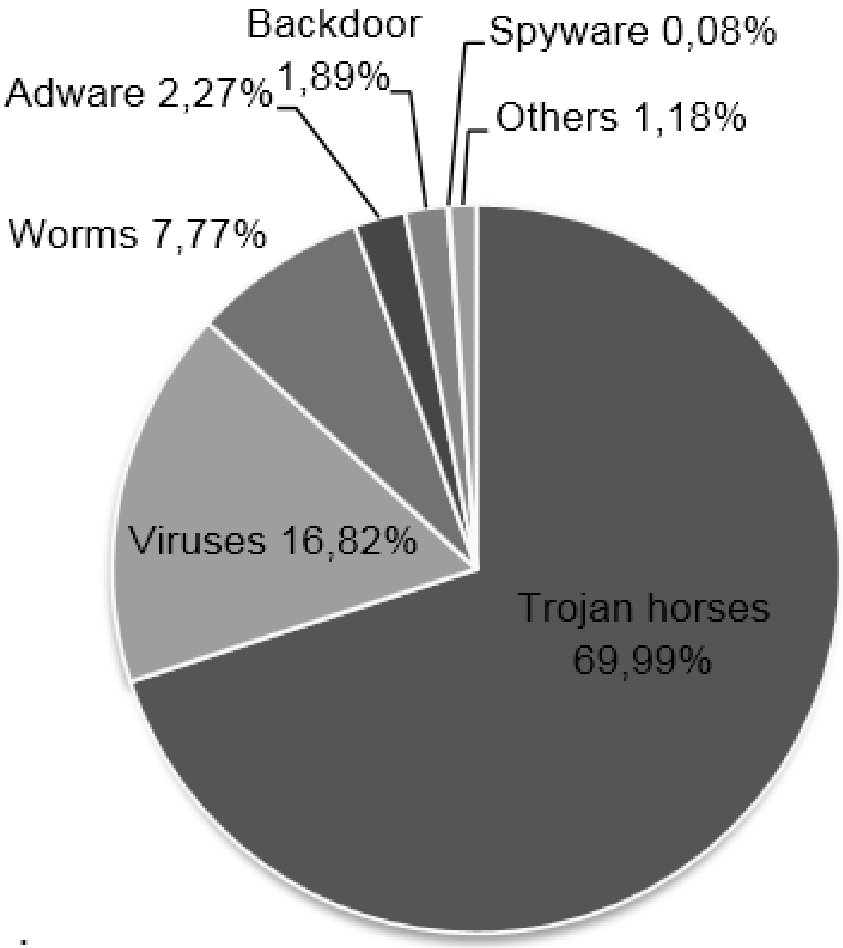
- Any code that has been modified with the intention of harming its usage or the user.

Primary Categories

- Virus - Propagates by infecting a host file.
- Worm - Self-propagates through e-mail, network shares, removable drives, file sharing or instant messaging applications.
- Backdoor - Provides functionality for a remote attacker to log on and/or execute arbitrary commands on the affected system.

Primary Categories (contd)

- Trojan - Performs a variety of malicious functions such as spying, stealing information, logging key strokes and downloading additional malware - several further sub categories follow such as infostealer, downloader,dropper,rootkit etc.
- Potentially Unwanted Programs (PUP) - Programs which the user may consent on being installed but may affect the security posture of the system or may be used for malicious purposes. Examples are Adware, Dialers and Hacktools/"hacker tools" (which includes sniffers, port scanners, malware constructor kits, etc.)
- Other - Unclassified malicious programs not falling within the other primary categories.



Malware by categories

March 16, 2011

Need to combat Malware

- There is an acute need for detecting and controlling the spread of malware
  - The direct damages incurred in 2006 due to malware attacks is USD 13 Billion [computereconomics.com]
  - The amount of suspicious obfuscated content has doubled from Q1 to Q2 of 2009 [*IBM X-force threat report*]
  - The time gap between a malware outbreak and the malware carrying out its intended damage is much smaller than the time taken by human experts to extract signature and deploy it for protection

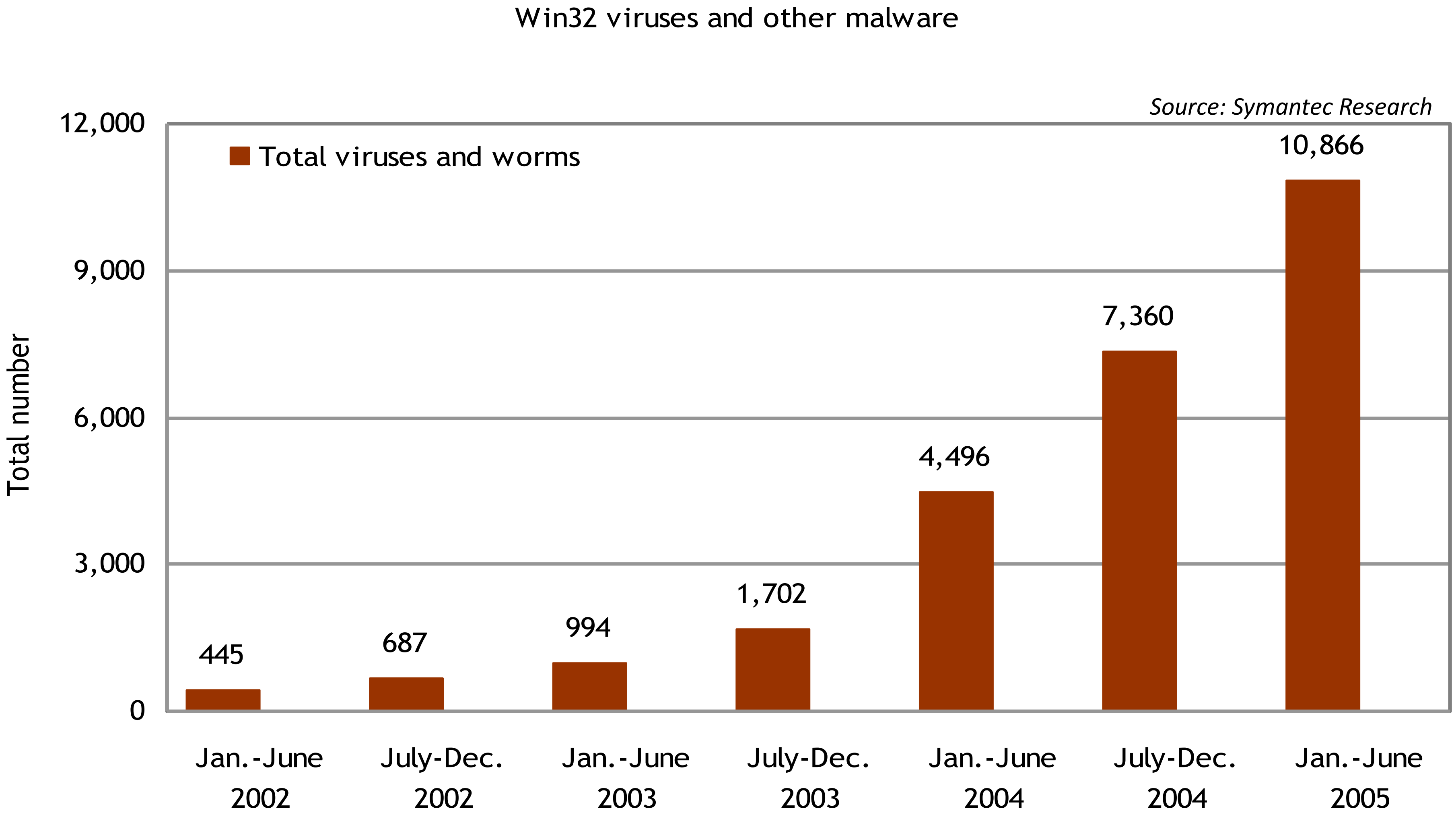
The Malware Problem

Host-based malicious-code detection:

- New program arrives an end-host system.
- Need to identify whether the program is malicious or not.

Viruses, trojans, backdoors, bots, adware, spyware, ...

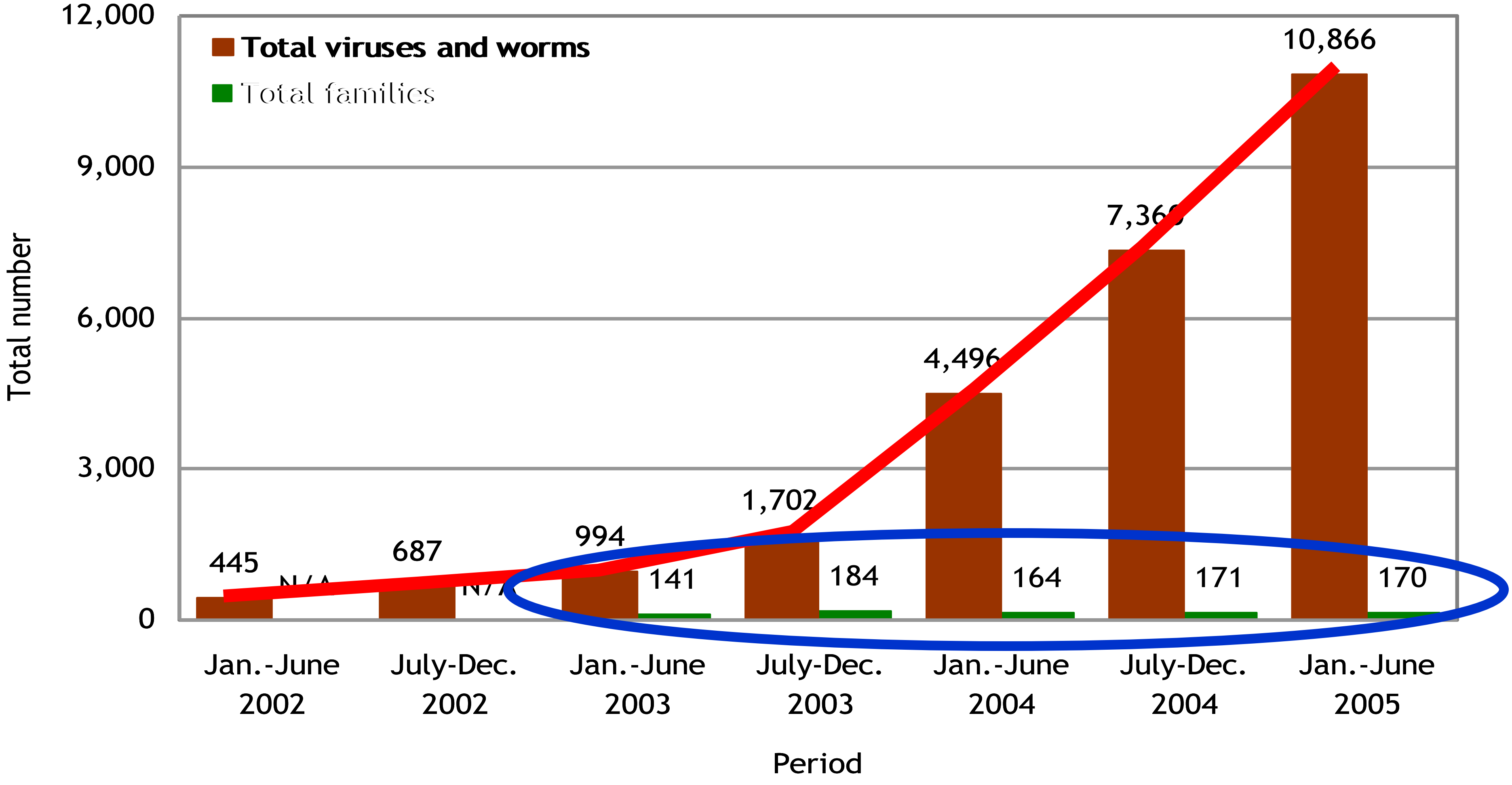
Malware: A Threat Assessment



Malware: A Threat Assessment

New Win32 virus and worm variants 2002-2005

Source: Symantec Research



Symantec Threat Report 2010

- Highlights from the report

- See
  - http://www.symantec.com/en/uk/business/theme.jsp?themeid=threatreport

Demographics

- *Where do attacks emerge?*
- US is still top on the list
  - 19% in 2009 (23% in 2008)
- Emergence of other countries in the top 10 list
  - Brazil and India
  - Emergence of these new countries related to increased internet connectivity in these countries

Attack Targets

- *Who are the attackers targeting?*
- Old news
  - Spam, identity theft, …
  - Still important factors
- New Trend
  - It looks like hackers are now targeting enterprises and government organizations
  - The goal seems to theft of sensitive data or espionage
  - *Stuxnet* is most sophisticated example of this attack

Vulnerabilities Exploited

- *What vulnerabilities are attackers exploiting?*
- It seems like web-based attacks are the most popular
  - Mozilla Firefox seems to be the most vulnerable
- The most common Web-based attack in 2009 was related to malicious PDF activity
  - Exploits vulnerabilities in “plug ins” that read the attached PDF file

Malware Trends

- *What types of malware were most prevalent?*
- Trojans rule!
  - Out of 10 malware families detected 6 were Trojans (2 worms, 1 back door, and 1 virus)
- Tool kits for creating malware and variants have matured
  - *Popular kits*: SpyEye, Fragus, Zues, …
  - In 2009 Symantec encountered 90,000 variants of malware variants created by the Zues toolkit

Take Aways

- Demographics of attack origins is expanding
- Web is the major vector for attack
- Trojans are the most prevalent form of malware
- Creating malware variants is easy because the toolkits have matured
- Enterprises and organizations are going to be increasingly targeted

Market Trends

- Security market will have a rapid growth in other countries (e.g., Brazil and India)
  - Reason: Demographics of attack origin
- Enterprise market will expand
  - Reason: Enterprises are being targeted by the attackers
- Other technologies for detection and remediation will become important

Modelling Malware

- First formal definition of a virus was given by Fred Cohen (student of Adleman)
  - A computer virus is a program that can infect other programs, when executed in a suitable environment, by modifying them to include a possibly evolved copy of itself

What is a virus?

- Virus (F Cohen): A sequence of symbols which when interpreted in a suitable environment modify other sequences of symbols in that environment by including a possibly evolved copy of itself
- A virus in some PL for some given OS may no longer be a virus for another OS

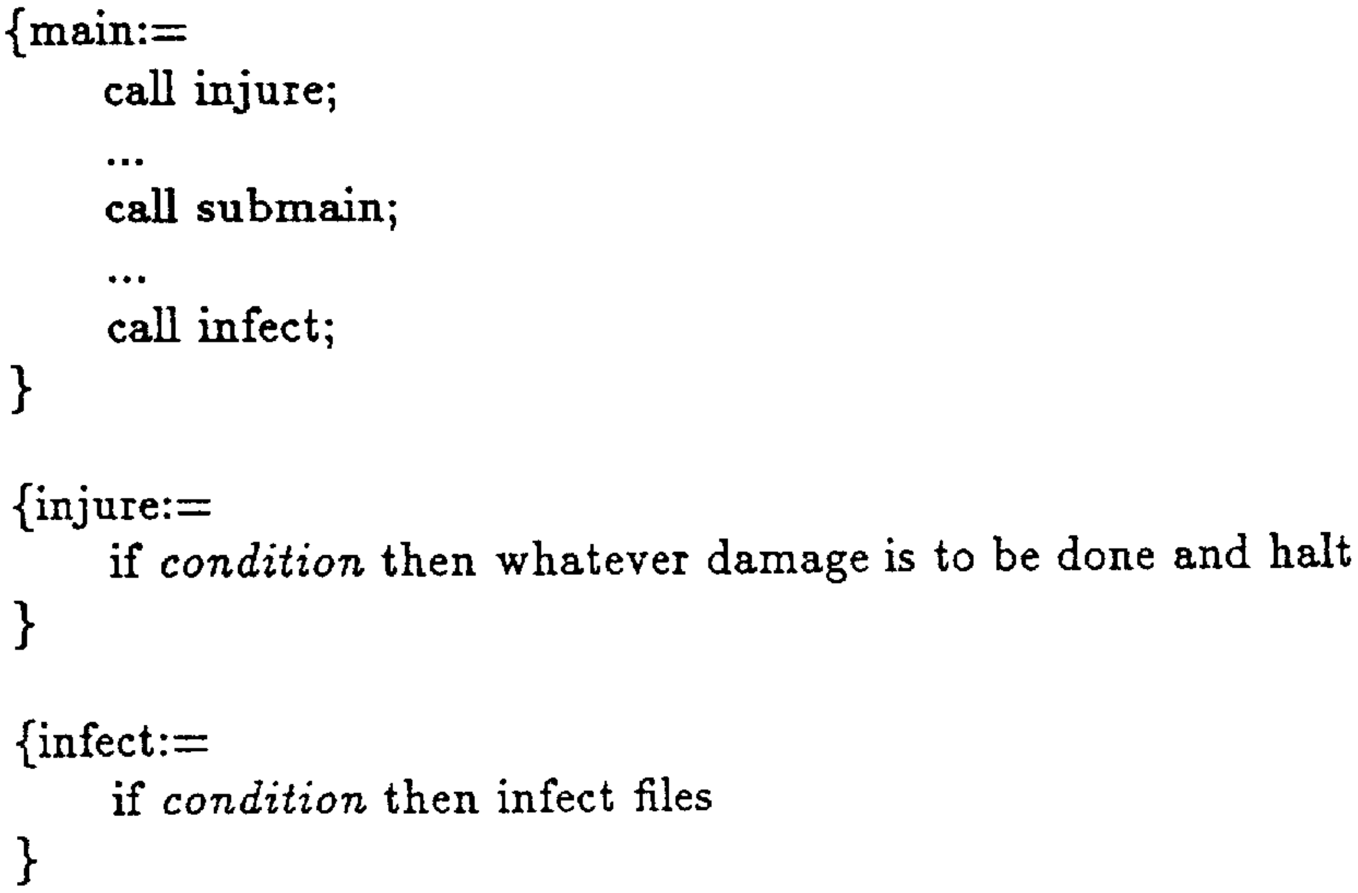
An example virus

```
program virus:=
{1234567;
subroutine infect-executable:=
    {loop: file = get-random-executable-file;
    if first-line-of-file = 1234567 then goto loop;
    prepend virus to file;
    }
subroutine do-damage:=
    {whatever damage is to be done}
subroutine trigger-pulled:=
    {return true if some condition holds}
main-program:=
    {infect-executable;
    If trigger-pulled then do-damage;
    goto next;}
    next:
}
```

Some remarks

- Ability to do damage is not considered a vital characteristic of a virus
- Possibility of a virus infection is based on the theory of self-reproducing automata
- Infected programs can also act as viruses, thus spreading to the transitive closure of information sharing

Adleman’s model of a virus

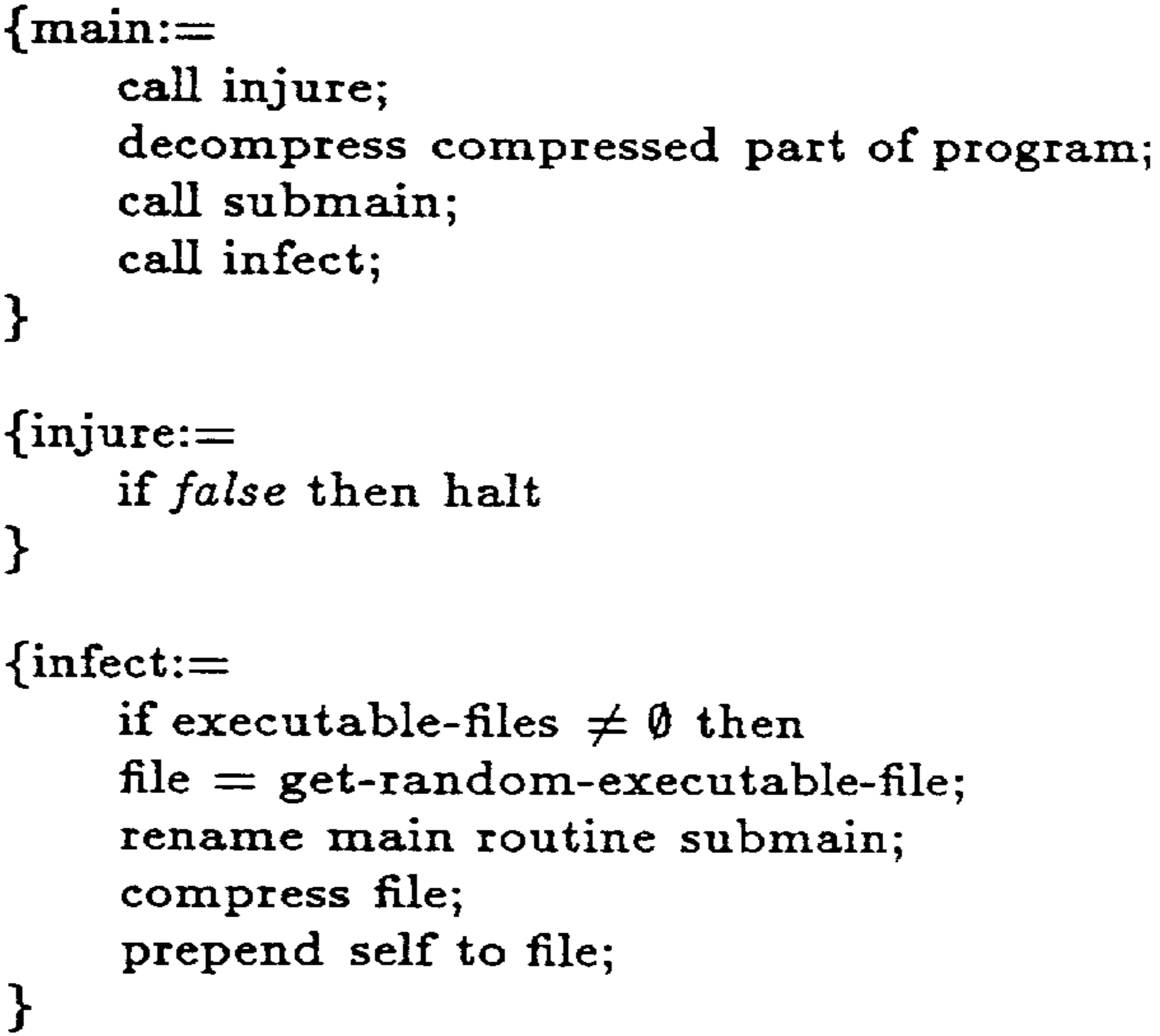


```
{main:=
     call injure;
     ...
     call submain;
     ...
     call infect;
}

{injure:=
     if condition then whatever damage is to be done and halt
}

{infect:=
     if condition then infect files
}
```

Compression Virus



```
{main:=
    call injure;
    decompress compressed part of program;
    call submain;
    call infect;
}

{injure:=
    if false then halt
}

{infect:=
    if executable-files $\neq \emptyset$ then
    file = get-random-executable-file;
    rename main routine submain;
    compress file;
    prepend self to file;
}
```

Formal characterization

- A program *v*, that always terminates, is called a virus iff for all states *s* either
  - Injure: all programs infected by *v* result in the same state when executed in *s*
  - Infect or Imitate: for every program *p*, the state resulting when *p* infected by *v* is executed in *s* is the same as the state resulting when *p* is executed in *s* possibly followed by an infection

Remarks

- Adleman's definition of a virus *v* characterizes the relationship between a program *p* and the program obtained by *v* infecting *p*
- There is no quantification of injury and infection
- Gives rise to a taxonomy of virus classes
  - *benign, Epeian, disseminating and malicious*

- Benign viruses never injure the system nor infect programs e.g.,

  compression virus

- Epeian viruses cause damage in certain conditions but never infect e.g.,

  Trojan horse Graybird

  - hides its presence on the compromised computer
  - downloads files from remote Web sites
  - gives its creator unauthorized access to the compromised machine

- Disseminating viruses spread by infecting other programs but never injure the system e.g., Internet worms like Netsky
  - sent as an e-mail attachment
  - scans computer for e-mail addresses
  - e-mails itself to all the addresses found
- Malicious viruses infect under some conditions and injure under some conditions e.g., CIH (Chernobyl)
  - corrupts the system BIOS on April 26
  - spreads by infecting portable executable files in Windows
  - inserts itself into the inter-section gaps of the target (hence, the infected file does not grow in size)

Basic results

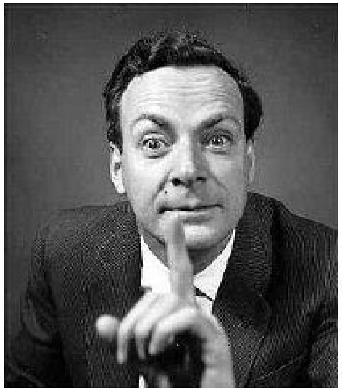
- Theorem: The set of viruses of a program is undecidable

- *No defense is perfect*: for every defense mechanism there is a virus which escapes it
- *Every virus can be caught*: for every virus there exists a defense mechanism which detects it

Process of Science

We never are definitely right;
we can only be sure when we are wrong.

Richard Feynman
Lectures on the Character of Physical Law



Viral detection

- ContradictoryVirus CV()

```
{ … main ()
    {  if not virusdetect(CV) then
       {  infection();
              if  trigger-value “true” then payload()
       }
       endif
       goto next;
       }
 }
```

Questions & Challenges

- Can we detect Computer Viruses?
  - What is the injury/infection caused by the virus?
- Can we disinfect infected programs?
  - Does quarantine help?
- Is it possible to protect?
  - Is isolation a protection strategy?
- How do we protect?
  - Can we certify a program to be free of virus?

Analogy: Biological Vs Computer viruses

Biological Viruses	Computer Viruses	Example
Attack on specific cells	Attack on specific file formats	Chameleon: polymorphic virus that infects COM files
Infected cells produce new viral offsprings	Infected programs produce new viral codes	
Modification of cell's genome	Modification of program's functions	
Viral interactions	Combined or anti-viruses viruses	Core wars game: 2 or more battle programs compete for complete control of a virtual simulator
Viruses replicate only in living cells	Execution is required to spread	
Already infected cells are not infected again	Use infection marker to prevent overinfeciton	Cohen's virus definition (checks for marker 1234567 at the beginning to prevent overinfection)
Retrovirus	Specifically bypasses given anti-virus software	AV Killer disables many AV software programs, such as McAfee, NOD32, Symantec Anti-Virus software etc.
Viral Mutation	Viral polymorphism	Chameleon: first known polymorphic virus
Antigens	Infection markers-signatures	CIH v1.2 contains string: CIH v1.2 TTIT

S Forrest (Univ of New Mexico)

Defenses

- Simple measures
  - Having policies in an enterprise can go a long way
  - For example, don't open a PDF attachment if you don't recognize the sender
- Signature-based detection is not enough
  - In 2009 Symantec created 2,895,000 signatures
  - In 2008 they created 1,691,323 signatures
  - These detectors need to be complemented with other types of detection

Defenses

- Complementing technologies
  - *Behavior-based* and *reputation-based* detection can complement signature-based detection
  - These complementing defenses can keep the number of signatures in check
  - These two technologies are mentioned throughout the report
- Data breaches
  - Keep confidential data secure even if an enterprise gets compromised
  - There are several solutions in the market
  - Remediation solutions will also gain traction

Key Definitions

Variants : New strains of viruses that borrow code, to varying degrees, directly from other known viruses.

Source: Symantec Security Response Glossary

Family: a set of variants with a common code base.

Beagle family has 197 variants (as of Nov. 30).

Warezov family has 218 variants (as on Nov. 27).

The Malware Problem

- Malware writers use any and all techniques to evade detection.
  - Obfuscation / packing / encryption
  - Remote code updates
  - Rootkit-based hiding

- Detectors use technology from 15 years ago: signature-based detection.

Signature-Based Detection

```
lea     eax, [ebp+Data]
push    offset aServices_exe
push    eax
call    _strcat
pop     ecx
lea     eax, [ebp+Data]
pop     ecx
push    edi
push    eax
lea     eax, [ebp+ExistingFileName]
push    eax
call    ds:CopyFileA
```

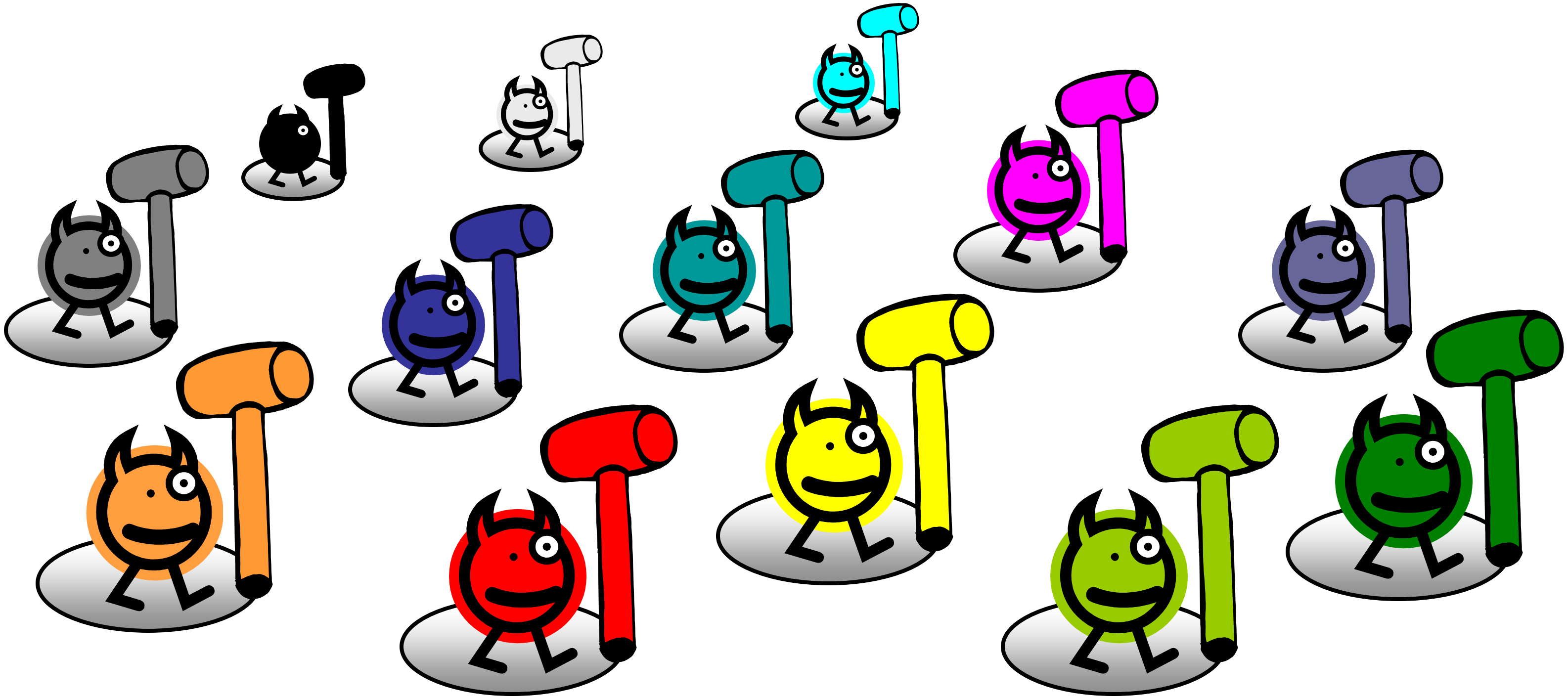
```
8D 85 D8 FE FF FF
68 78 8E 40 00
50
E8 69 06 00 00
59
8D 85 D8 FE FF FF
59
57
50
8D 85 D4 FD FF FF
50
FF 15 C0 60 40 00
```

Signature

- *Signatures (aka scan-strings)* are the most common malware detection mechanism.

Signature Detection Does Not Scale

One signature for one malware instance.



Current Signature Management

McAfee: release daily updates

- Trying to move to hourly "beta" updates

DAT File #	Date	Threats Detected	New Threats Added	Threats Updated
4578	Sep. 09	147,382	22	188
4579	Sep. 12	147,828	27	231
4580	Sep. 13	148,000	11	236
4581	Sep. 14	148,368	42	140
4582	Sep. 15	148,721	16	203
4583	Sep. 16	149,050	18	117

Source: McAfee DAT Readme

Huge Signature Databases

- Recently, McAfee announced the addition of the 200,000th signature.
  - More signatures than files on a standard Windows machine (approx. 100k).

- McAfee notes that:

  "Good family detection becomes crucial for a less worrisome experience on the Internet."

  Source: McAfee Avert Labs

Roadmap to Better Detection

- Make the malware writer’s job as hard as possible.

- Detect malware families,
  not individual malware instances.

- Catch behavior,
  not syntactic artifacts.

Threat Model

- Malware writers craft their programs so to avoid detection.

Two common evasion techniques:

- Program Obfuscation
  (Preserves malicious behavior)

- Program Evolution
  (Enhances malicious behavior)

Obfuscations for Evasion

Nop insertion
Register renaming
Junk insertion
Instruction reordering
Encryption
Compression
Reversing of branch conditions
Equivalent instruction substitution
Basic block reordering
…

Evasion Through Junk Insertion

```
lea     eax, [ebp+Data]
nop
push    offset aServices_exe
nop
nop
push    eax
call    _strcat
nop
nop
nop
pop     ecx
lea     eax, [ebp+Data]
pop     ecx
push    edi
push    eax
nop
lea     eax, [ebp+ExistingFileName]
push    eax
call    ds:CopyFileA
```

```
8D 85 D8 FE FF FF
68 78 8E 40 00
50
E8 69 06 00 00
59
8D 85 D8 FE FF FF
59
57
50
8D 85 D4 FD FF FF
50
FF 15 C0 60 40 00
```

Signature

Evasion Through Reordering

```
       lea      eax, [ebp+Data]
       jmp label_one

label_two:
       lea      eax, [ebp+Data]
       ...
       push     eax
       call     ds:CopyFileA
       jmp label_three

label_one:
       ...
       call     _strcat
       ...
       jmp label_two

label_three: ...
```

```
8D 85 D8 FE FF FF
90*
68 78 8E 40 00
90*
50
90*
E8 69 06 00 00
90*
59
90*
.
.
.
90*
50
90*
FF 15 C0 60 40 00
```

Regex Signature

Evasion Through Encryption

```
       lea      esi, data_area
       mov      ecx, 37
again:
       xor byte ptr [esi+ecx], 0x01
       loop     again
       jmp      data_area
       .
       .
       .
data_area:
       db       8C 84 D9 FF ...
       .
       .
       .
       db       FE 14 C1 61 ...
```

✗

```
8D 85 D8 FE FF FF
90*
68 78 8E 40 00
90*
50
90*
E8 69 06 00 00
90*
59
90*
.
.
.
90*
50
90*
FF 15 C0 60 40 00
```

Regex Signature

Evasion Through Evolution

- Malware writers are good at software engineering:
  - Modular designs
  - High-level languages
  - Sharing of exploits, payloads, and evasion techniques

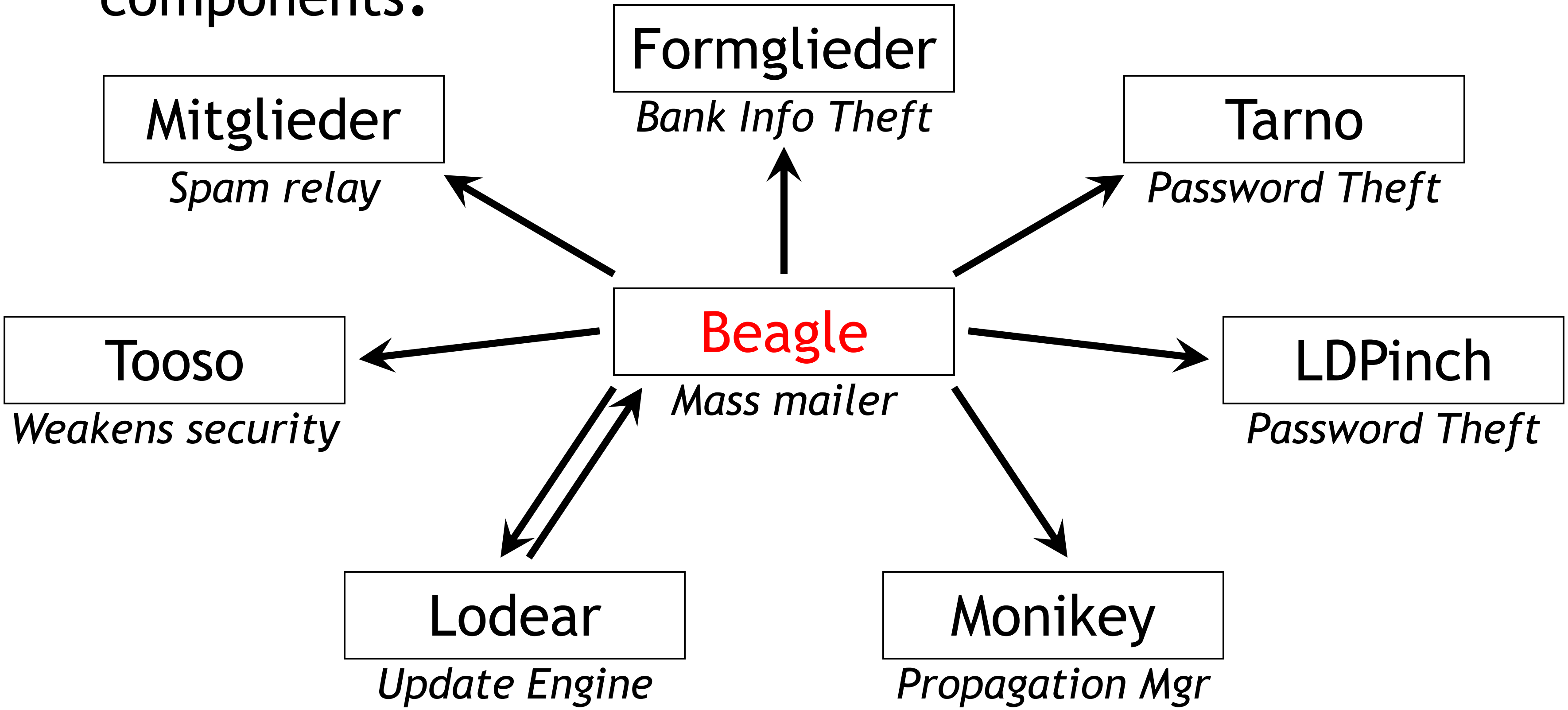
Example:

Beagle e-mail virus gained additional functionality with each version.

Beagle Evolution

Source: J. Gordon, infectionvectors.com

- More than 100 variants, not counting associated components.



Empirical Study
[Christodorescu & Jha, ISSTA 2004]

- Start with a set of known viruses.
- Create obfuscated versions:
  - Reordering
  - Register/variable renaming
  - Encryption

- Measure resilience to obfuscation (detection rate of obfuscated versions)

Evaluation Goal: **Resilience**

Question 1:

- How resistant is a virus scanner to obfuscations or variants of known worms?

Question 2:

- Using the limitations of a virus scanner, can a blackhat determine its detection algorithm?

High Level Specs

- A high-level definition can be very concise, but quite imprecise. This is because it has a lot of underlying assumptions. Any description that is to be automatically checked by a machine should be made more precise.
- We can make this description more precise by adding information about the protocols involved in this behavior.
- We also need to clarify what “mass” means: in this case, it is a rate of propagation, e.g., messages sent per hour.
- Finally, we explain what a “virus” is: a program that propagates itself.

Describing Malicious Behavior

[Christodorescu et al., Oakland 2005]

- Informal description:
  "Mass-mailing virus"

- A more precision description:
  "A program that:
  sends messages containing copies of itself,
  using the SMTP protocol,
  in a large number over a short period of time."

Malspec

- A specification of behavior.

```
push     10h
push     eax
push     edi
call     connect
...      ; compose SMTP
         ; command "HELO ..."
push     eax
push     ecx
push     edi
call     send
```

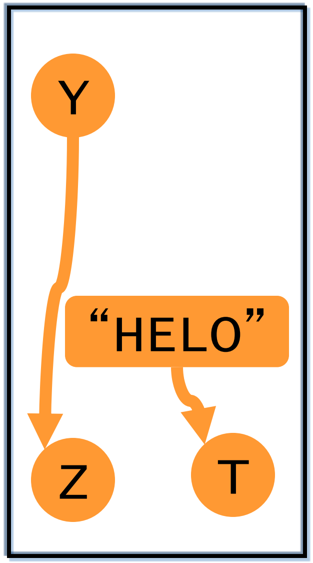
Malware Instance
(Netsky.B)

=

```
connect(Y);


send(Z,T);
```

+

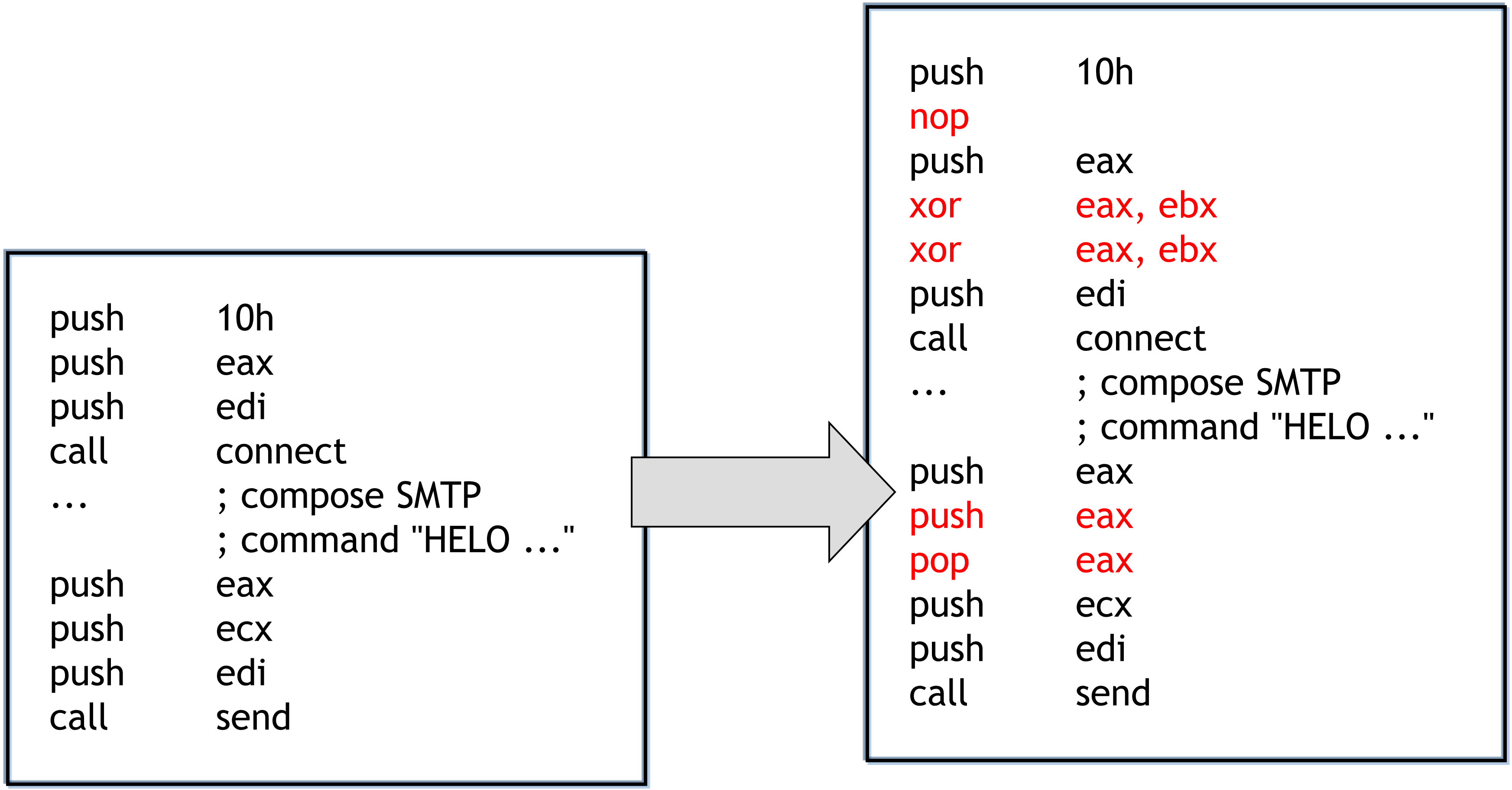


Syntactic info

Semantic info

Malspec

Obfuscation Preserves Behavior



```
push    10h
push    eax
push    edi
call    connect
...     ; compose SMTP
        ; command "HELO ..."
push    eax
push    ecx
push    edi
call    send
```

```
push    10h
nop
push    eax
xor     eax, ebx
xor     eax, ebx
push    edi
call    connect
...     ; compose SMTP
        ; command "HELO ..."
push    eax
push    eax
pop     eax
push    ecx
push    edi
call    send
```

- Junk insertion + code reordering.

Obfuscation Preserves Behavior

```
push    10h
push    eax
push    edi
call    connect
...     ; compose SMTP
        ; command "HELO ..."
push    eax
push    ecx
push    edi
call    send
```

```
push    10h
nop
push    eax
jmp     L1
L4: push ecx
push    edi
jmp     L5
L2: xor eax, ebx
push    edi
call    connect
...     ; compose SMTP
        ; command "HELO ..."
push    eax
push    eax
jmp     L3
L1: xor eax, ebx
jmp     L2
L3: pop eax
jmp     L4
L5: call send
```

- Junk insertion + code reordering.

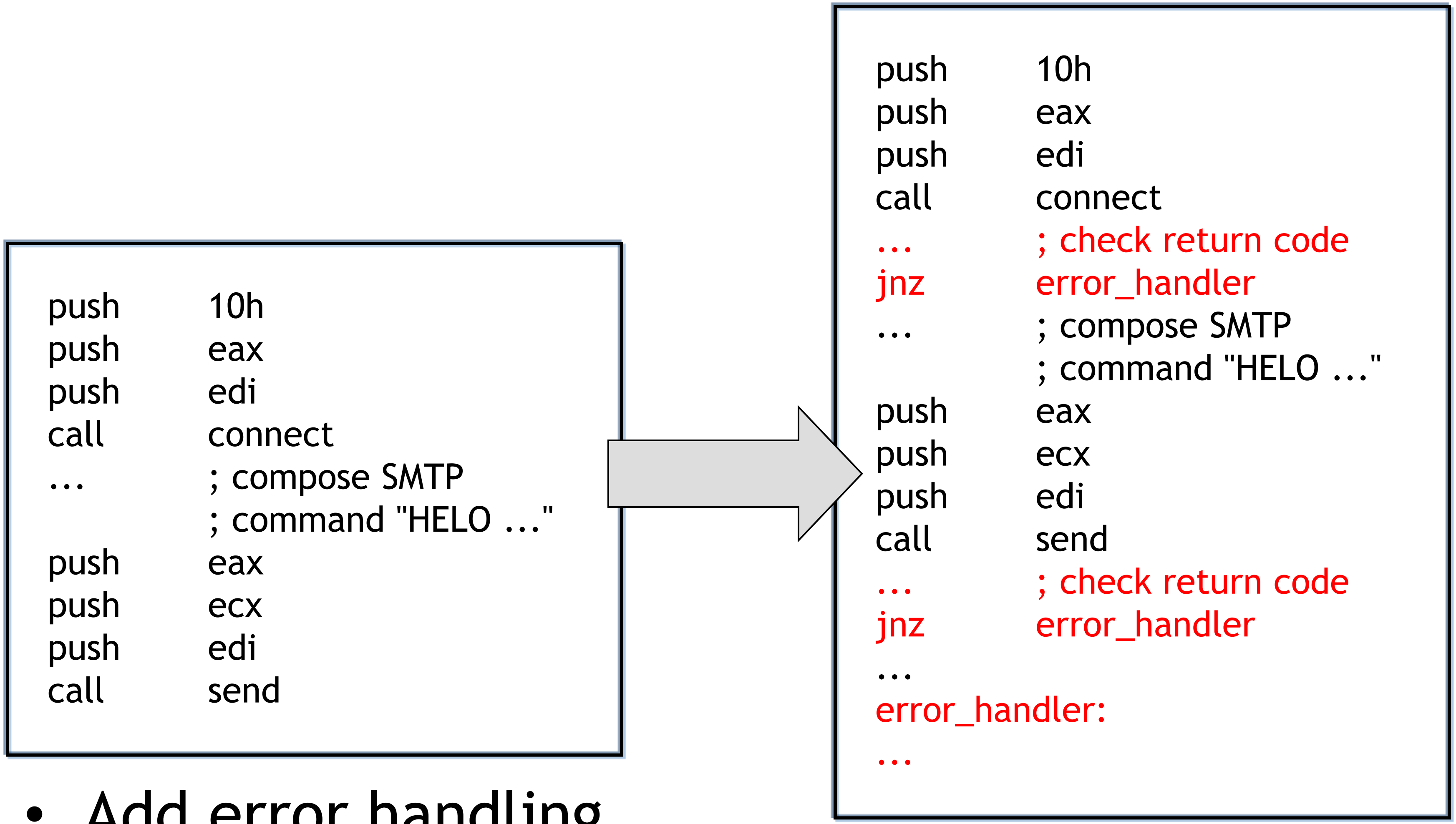
Obfuscation Preserves Behavior

```
push    10h
push    eax
push    edi
call    connect
...     ; compose SMTP
        ; command "HELO ..."
push    eax
push    ecx
push    edi
call    send
```

```
push      10h
nop
push      eax
jmp       L1
L4: push  ecx
push      edi
jmp       L5
L2: xor   eax, ebx
push      edi
call      connect
...       ; compose SMTP
          ; command "HELO ..."
push      eax
push      eax
jmp       L3
L1: xor   eax, ebx
jmp       L2
L3: pop   eax
jmp       L4
L5: call  send
```

- Junk insertion + code reordering.

Evolution Preserves Behavior

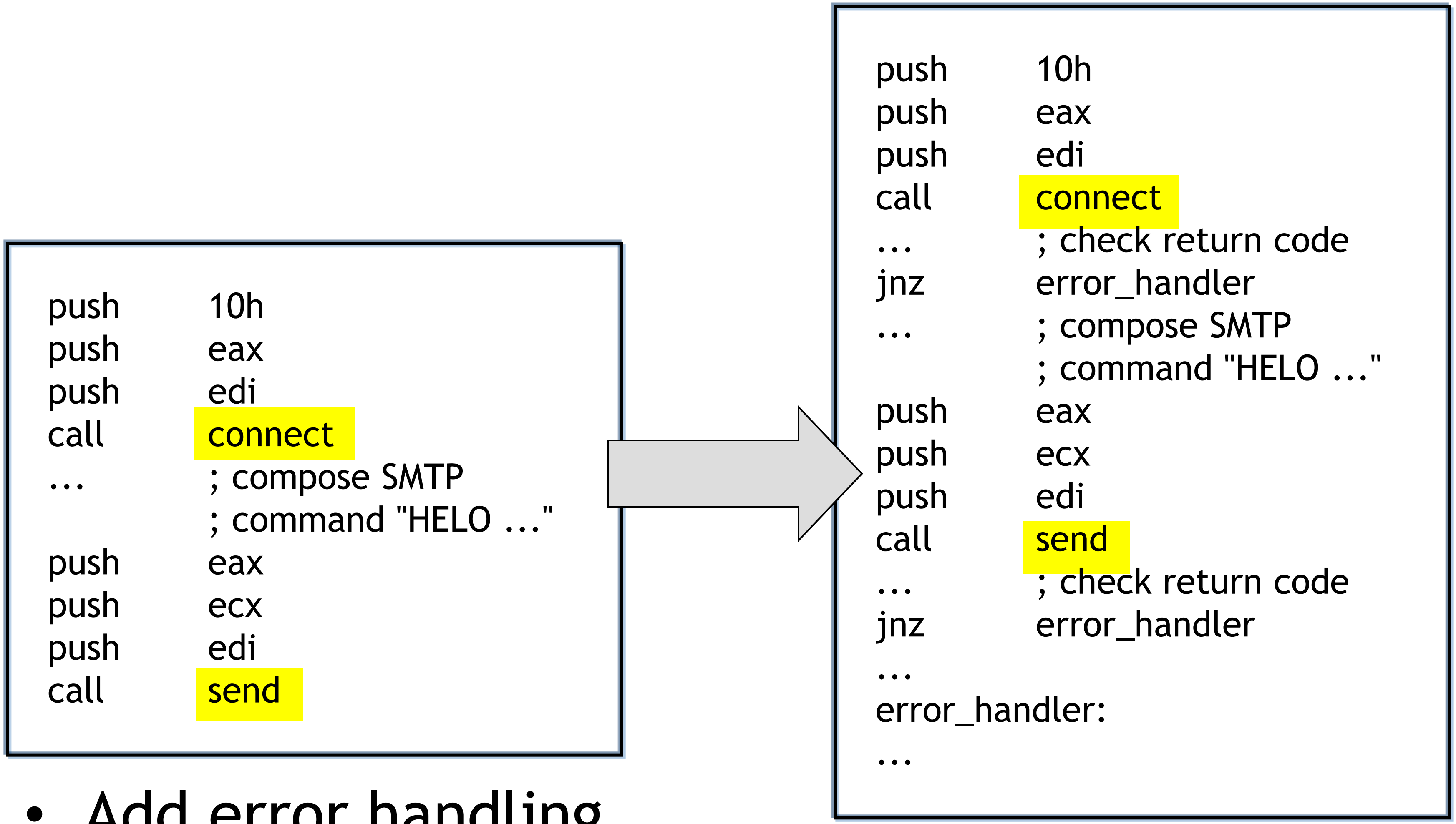


```
push      10h
push      eax
push      edi
call      connect
...       ; compose SMTP
          ; command "HELO ..."
push      eax
push      ecx
push      edi
call      send
```

```
push      10h
push      eax
push      edi
call      connect
...       ; check return code
jnz       error_handler
...       ; compose SMTP
          ; command "HELO ..."
push      eax
push      ecx
push      edi
call      send
...       ; check return code
jnz       error_handler
...
error_handler:
...
```

- Add error handling.

Evolution Preserves Behavior

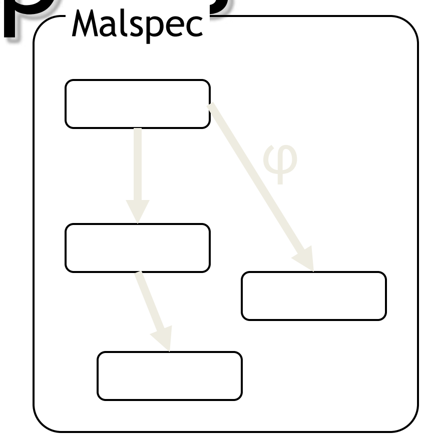


```
push      10h
push      eax
push      edi
call      connect
...       ; compose SMTP
          ; command "HELO ..."
push      eax
push      ecx
push      edi
call      send
```

```
push      10h
push      eax
push      edi
call      connect
...       ; check return code
jnz       error_handler
...       ; compose SMTP
          ; command "HELO ..."
push      eax
push      ecx
push      edi
call      send
...       ; check return code
jnz       error_handler
...
error_handler:
...
```

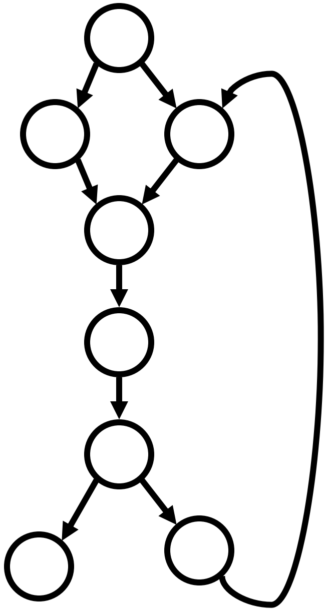
- Add error handling.

Detection Using Malspecs



Static detection:

Given an executable binary, check whether it satisfies the malspec.



Just like model checking, but...

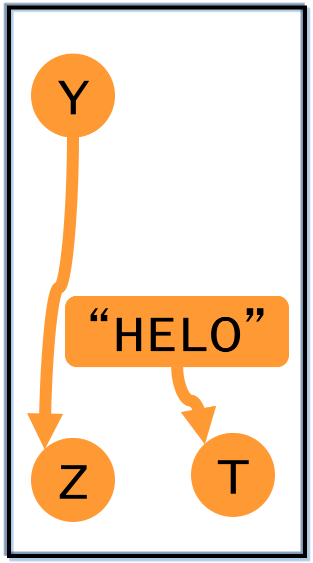
- Malicious code allows no assumptions to be made
- Real-time constraints

A Behavior-Based Detector

- Match the syntactic constructs, then check the semantic information.

```
connect(Y);


send(Z,T);
```



Syntactic info *Semantic info*

Malspec

Check the Semantic Info

Program (Netsky.O):

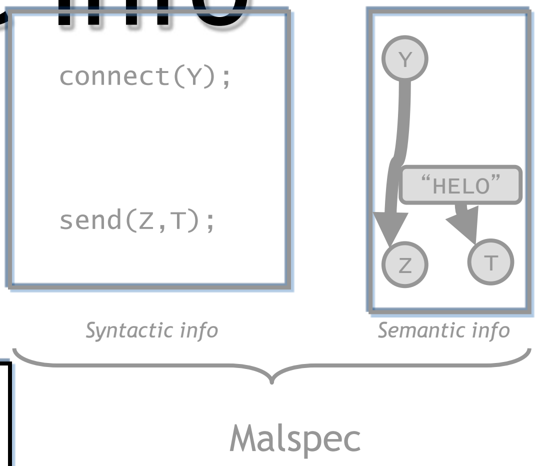
```
push      10h
push      eax
push      [ebp+s]
call      connect
…
push      ebx
lea       eax, [ebp+s]
push      eax
call      send_email
```

send_email()

```
…              ; compose SMTP
               ; command "HELO …"
lea            eax, [ebp+arg1]
push           eax
lea            eax, [ebp+buffer]
push           eax
call           SMTP_send_and_rcv
```

SMTP_send_and_rcv()

```
push      eax
push      [ebp+arg1]
mov       eax, [ebp+arg2]
push      [eax]
call      send
```



Consider another variant of Netsky (variant O). This one differs from the previous one in the code for sending email is split across several functions, and each function performs error checking

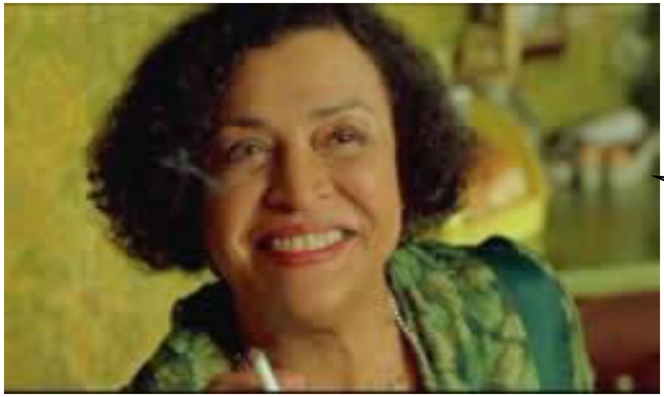
Check with the Oracle

- Assume we have an oracle that can validate value predicates.

Does
eax before == ebx after for
the code sequence:

```
push eax
call foo
mov ebx, [ebp+4]
```

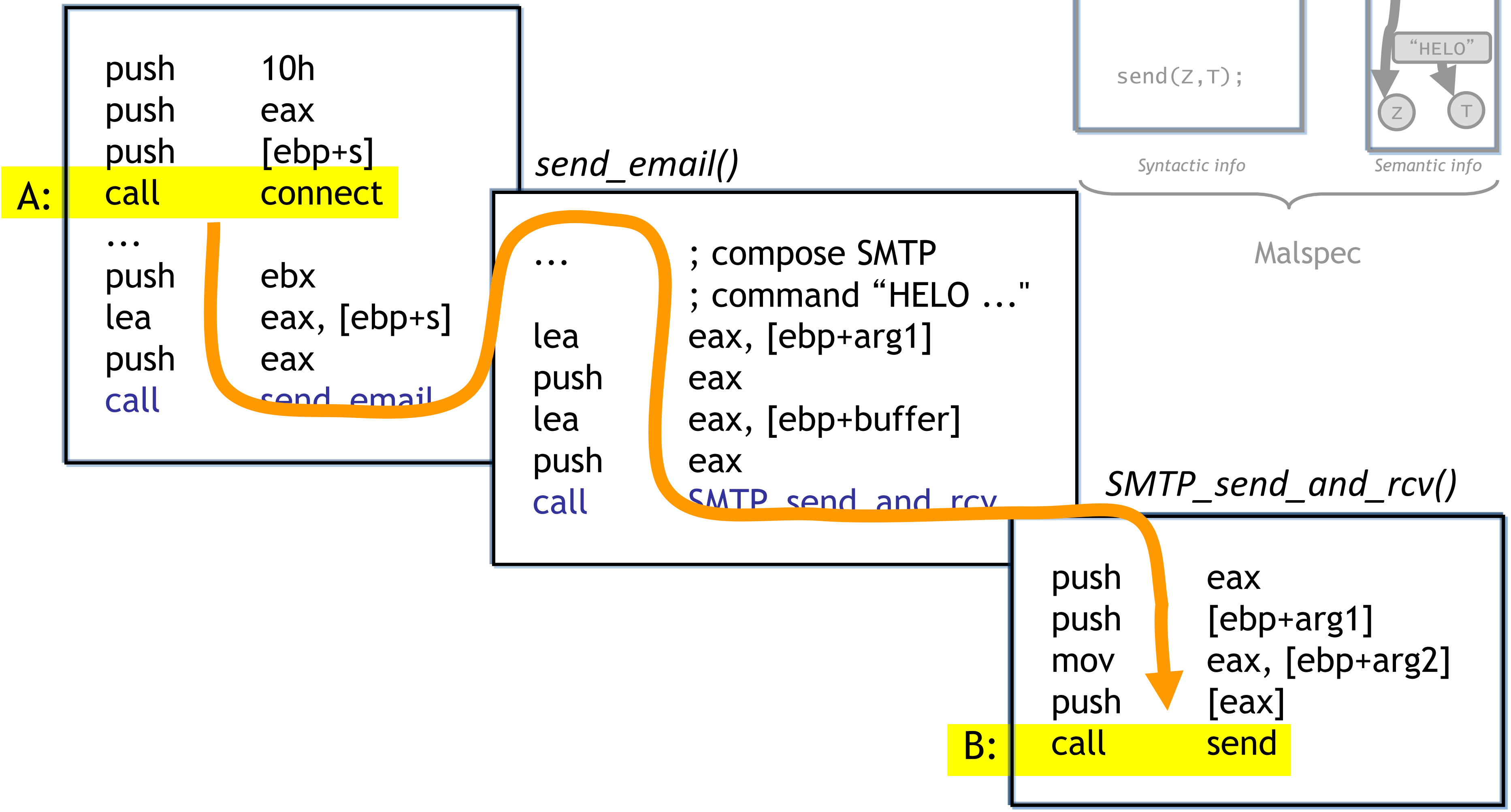
?



Yes.

Check the Semantic Info

Program (Netsky.O):



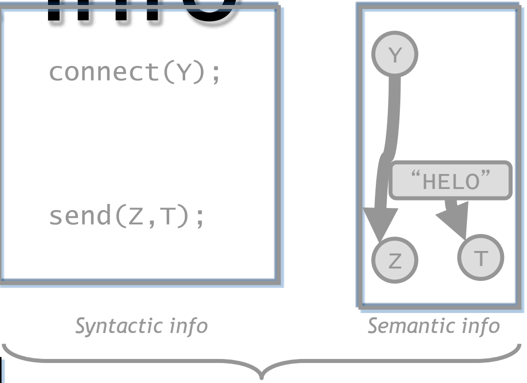
```
    push    10h
    push    eax
    push    [ebp+s]
A:  call    connect
    …
    push    ebx
    lea     eax, [ebp+s]
    push    eax
    call    send_email
```

send_email()

```
…       ; compose SMTP
        ; command "HELO …"
lea     eax, [ebp+arg1]
push    eax
lea     eax, [ebp+buffer]
push    eax
call    SMTP_send_and_rcv
```

SMTP_send_and_rcv()

```
    push    eax
    push    [ebp+arg1]
    mov     eax, [ebp+arg2]
    push    [eax]
B:  call    send
```



```
connect(Y);

send(Z,T);
```

Malspec

Query the Oracle

Program (Netsky.O):

```
    push    10h
    push    eax
    push    [ebp+s]
A:  call    connect
    …
    push    ebx
    lea     eax, [ebp+s
    push    eax
    call    send_email
```

send_email()

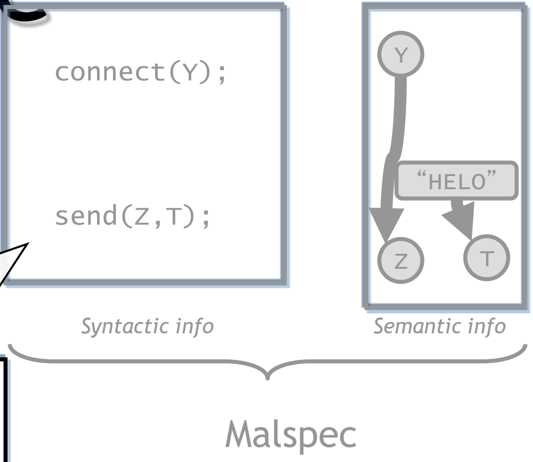
```
    ; parse SMTP
    ; command "HELO …"
    lea     eax, [ebp+arg1]
    push    eax
    lea     eax, [ebp+buffer]
    push    eax
    call    SMTP_send_and_rcv
```

Does memory[ebp@A+4] == memory[ebp@B+4] hold for the code sequence between A and B?

Yes.

SMTP_send_and_rcv()

```
    push    eax
    push    [ebp+arg1]
    mov     eax, [ebp+arg2]
    push    [eax]
B:  call    send
```



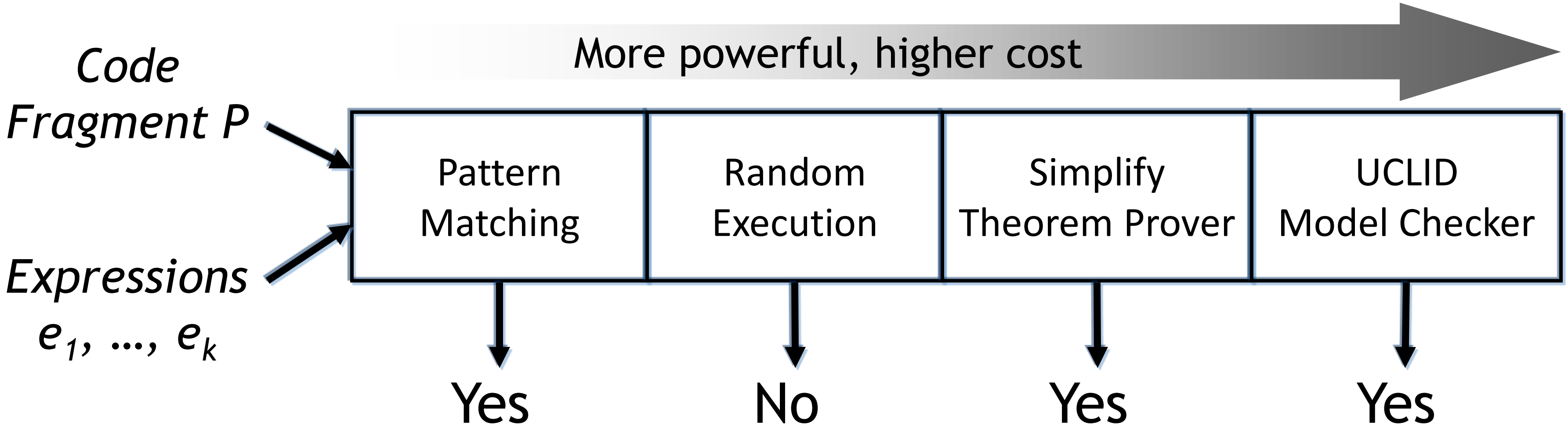
```
connect(Y);

send(Z,T);
```

A Recipe for an Oracle

- Instance of program verification problem:

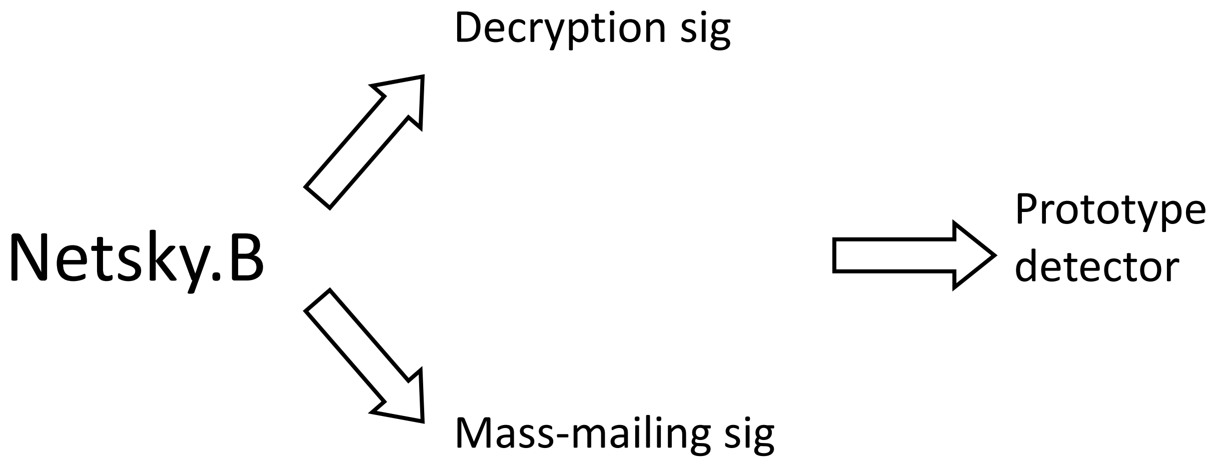
  Does program P respect property ϕ *?*



A Behavior-Based Prototype

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

Evaluation of Malspecs



Netsky.B

Decryption sig

Mass-mailing sig

Prototype detector

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

McAfee uses individual signatures for each worm.

Malspecs provide forward detection.

Performance

- Prototype is slower than commercial anti-virus tools.

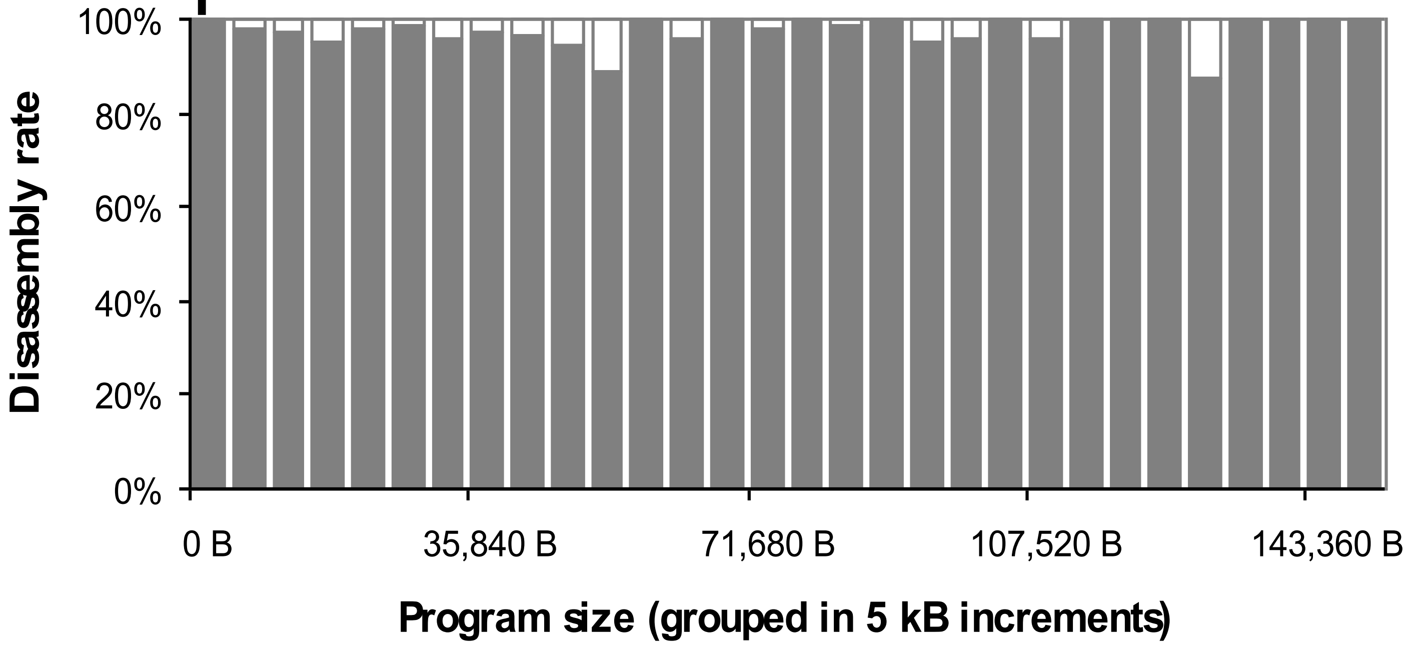
Malware Family	*Running Time*	
	Average	*Std. Deviation*
Netsky	99.57 s	41.01 s
Beagle	56.41 s	40.72 s

- Plenty of room for improvement.

  e.g. disassembler: 25% of time.

Evaluation: False Positive Rate

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



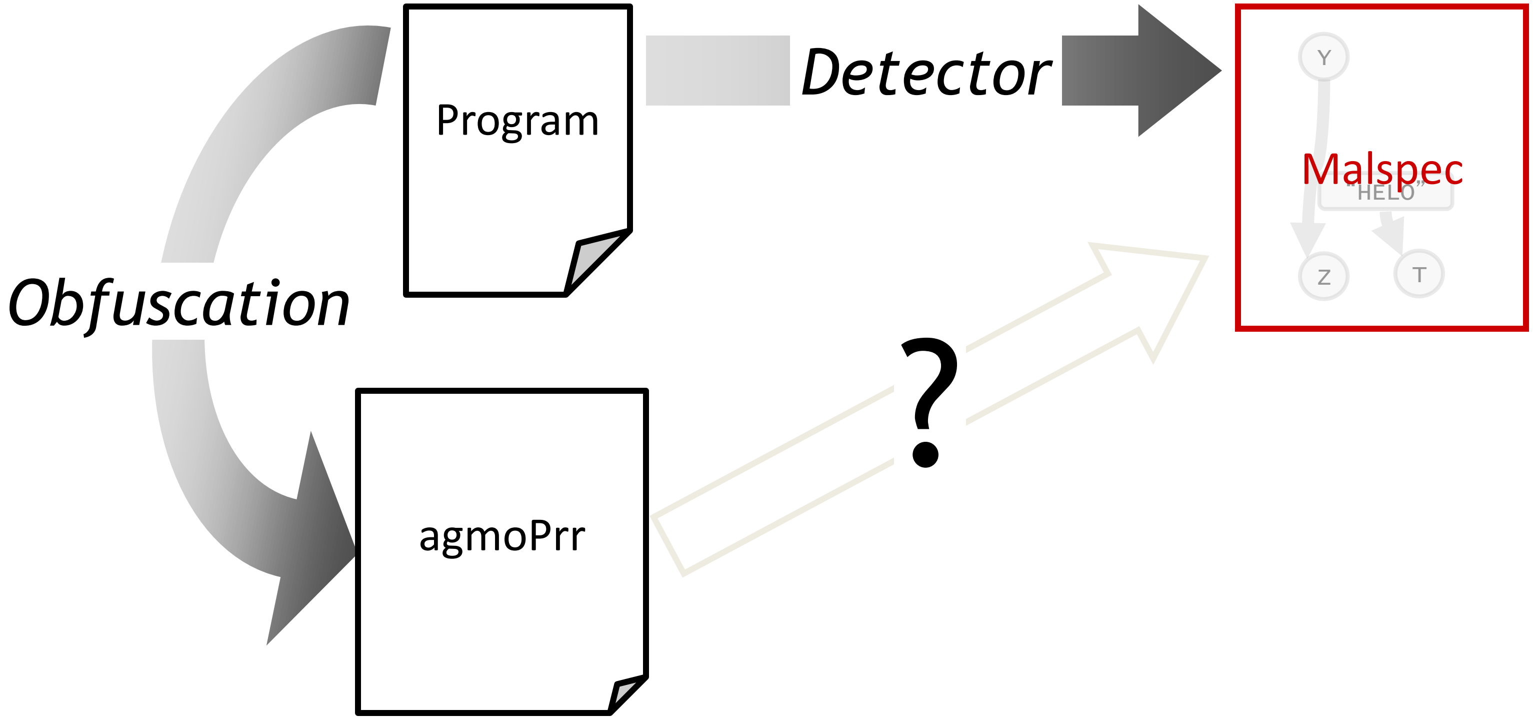
Evaluation: Obfuscation Resilience

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

Obfuscation Type	***Behavior-Based Detection***		*McAfee*
	Average Time	***Detection Rate***	
Nop insertion	74.81 s	**100%**	75%
Stack op. insertion	159.10 s	**100%**	25%
Math op. insertion	186.50 s	**95%**	5%

Formally Assessing Resilience
[POPL 2007]

- Soundness (no false positives)
- Completeness (no false negatives)



Program

Detector

Malspec

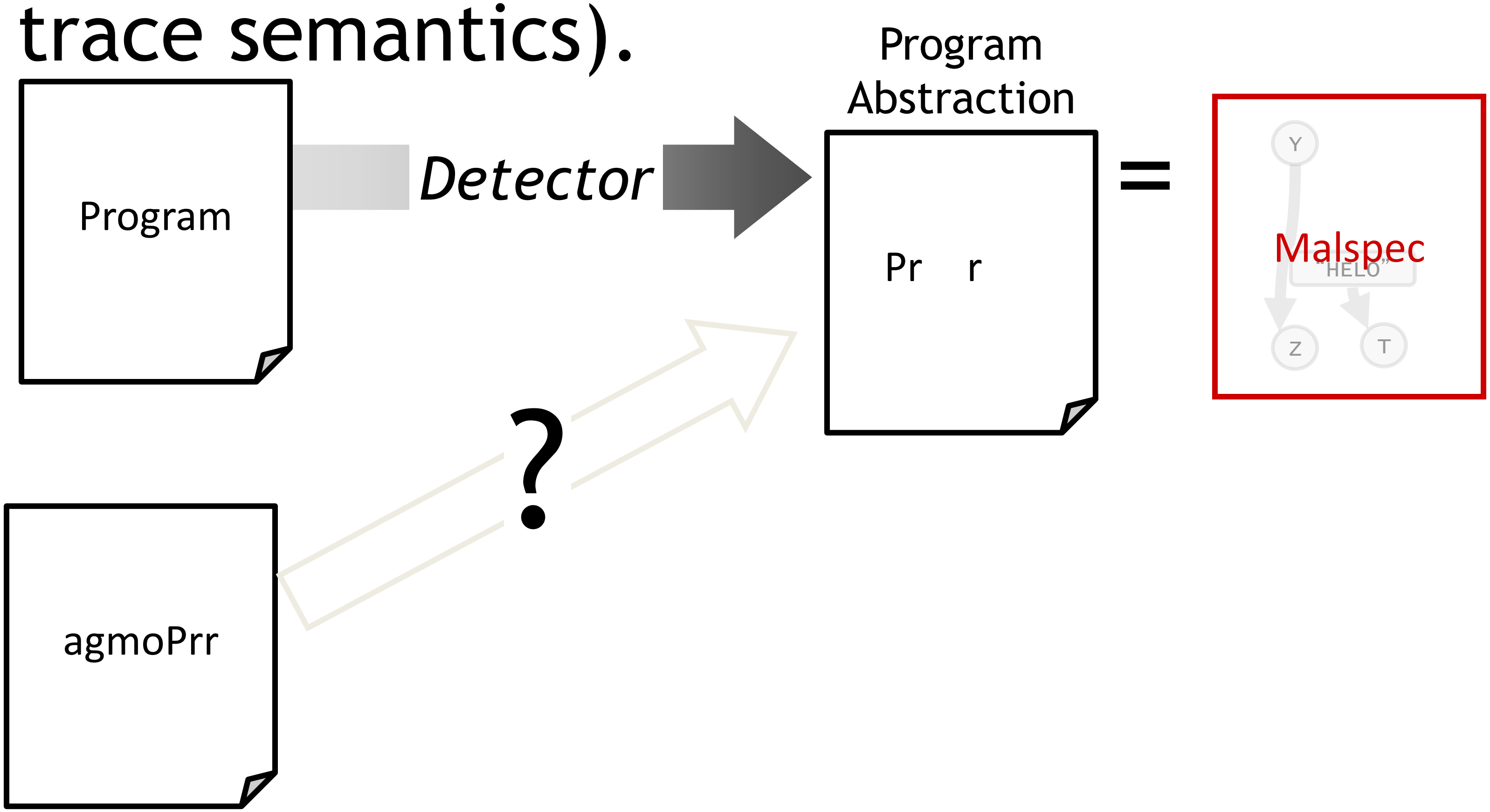
Obfuscation

agmoPrr

?

Approach to Assessing Resilience

- Detector “filters out” irrelevant aspects of the program (described in terms of trace semantics).



References

- Papers
  - M. Christodorescu and S. Jha, Testing Malware Detectors, *International Sympoisum on Testing and Analysis (ISSTA)*, 2004
  - M. Christodorescu, S. Seshia, S. Jha, D. Song, and R. Bryant, *Semantics-Aware Malware Detection, IEEE Symposium on Security and Privacy (Oakland)*, 2005.
  - M. Dalla Preda, M. Christodorescu, S. Debray and S. Jha, *A Semantics-Based Approach to Malware Detection, Symposium on Principles of Programming Languages (POPL), January 2007.*
- Website
  - http://www.cs.wisc.edu/~jha/

MINING

Detection: Textual Patterns

- Check for syntactic signatures that attempt to capture the machine level byte sequence of the malware spread across single packets to series of packets.
    - Pure-Text:

      Complexity of detecting a known fixed virus pattern of length M in a program of length N is harnessed by the Boyer-Moore string-searching algorithm which never uses more than N+M steps and under many circumstances (a small pattern and a large alphabet) can use about N/M steps.

- Virus:
    - Textual patterns are not any more the trend

Metamorphic virus

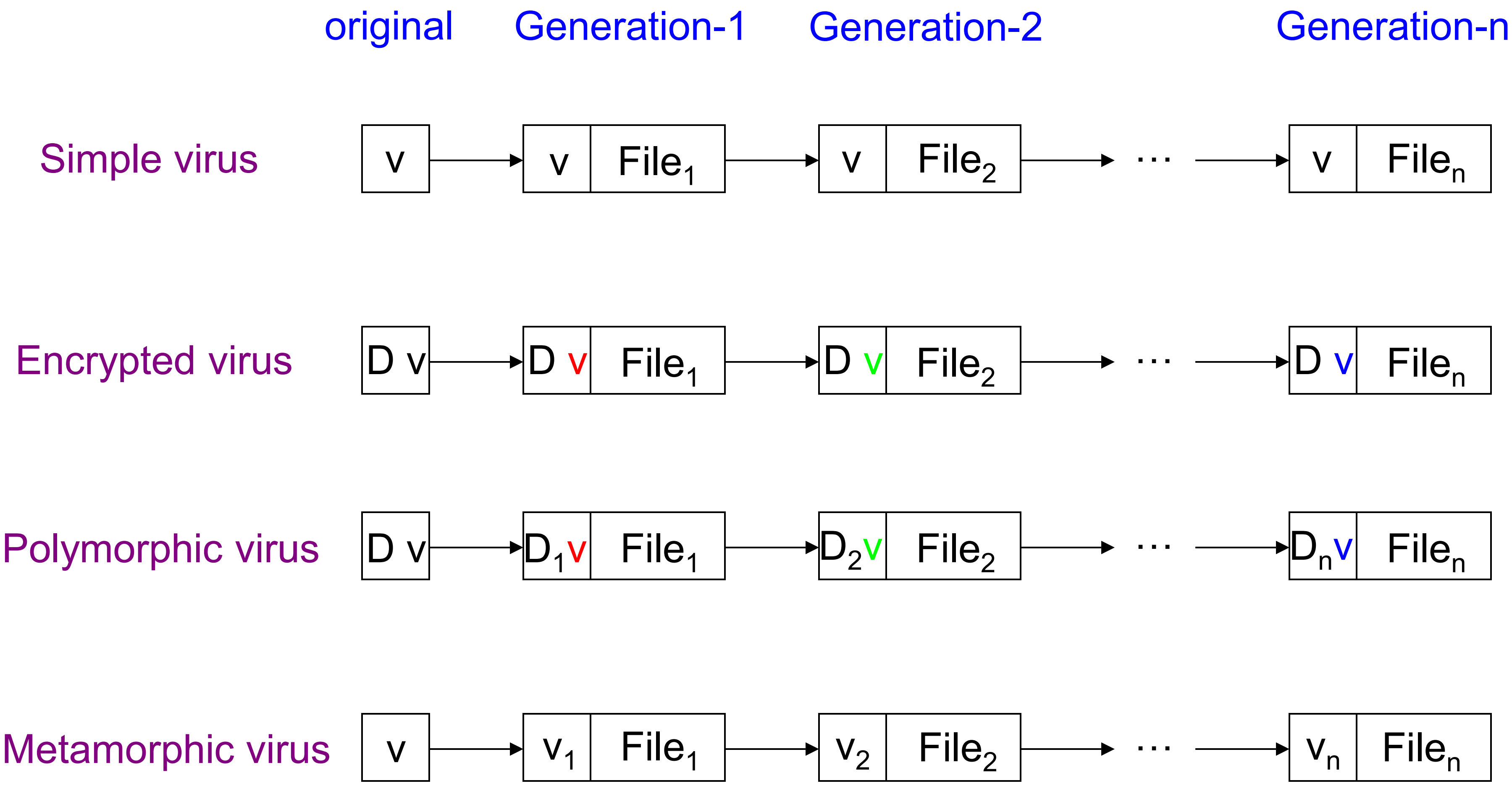
(Metaphoric Permutating High-Obfuscating assembler)

- Re-write themselves completely each time they have to infect a file
- Heavily use obfuscation techniques
- Usually require a large, complex metamorphic engine
- e.g., W32/Simile
  - over 14000 lines of assembly code
  - metamorphic engine comprises about 90%

Polymorphic Virus

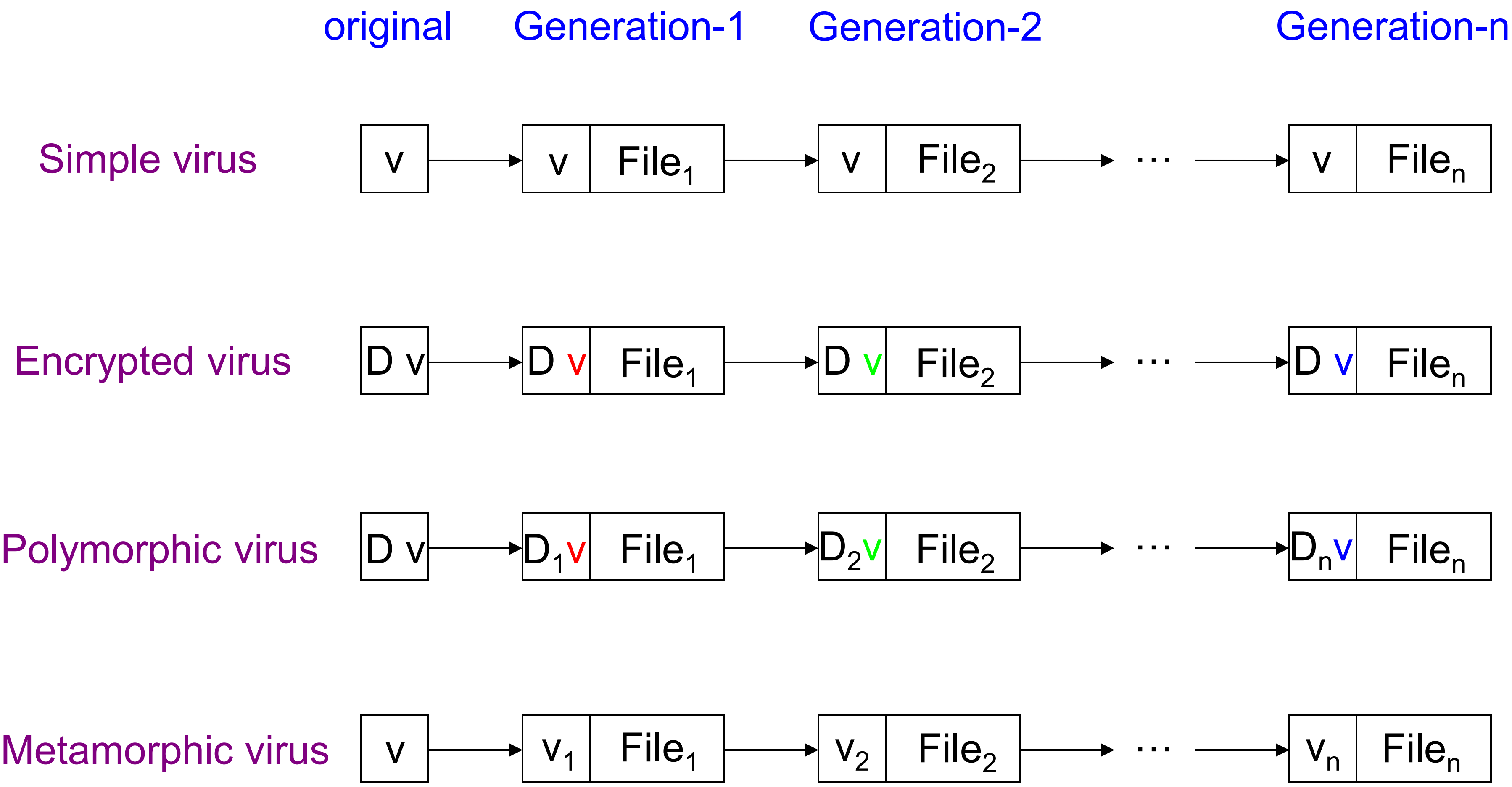
- Infects files with an encrypted copy of itself
- Encryption is modified on each infection
- Thus, polymorphic viruses have no parts that are common between infections
- e.g., Chameleon
  - Infects COM files in its directory
  - Changes its signature every time it infects a new file

Virus replication



	original		Generation-1		Generation-2			Generation-n
Simple virus	v	→	v File$_1$	→	v File$_2$	→	⋯ →	v File$_n$
Encrypted virus	D v	→	D v File$_1$	→	D v File$_2$	→	⋯ →	D v File$_n$
Polymorphic virus	D v	→	D$_1$v File$_1$	→	D$_2$v File$_2$	→	⋯ →	D$_n$v File$_n$
Metamorphic virus	v	→	v$_1$ File$_1$	→	v$_2$ File$_2$	→	⋯ →	v$_n$ File$_n$

Virus replication



	original	Generation-1	Generation-2		Generation-n
Simple virus	v	v File$_1$	v File$_2$	⋯	v File$_n$
Encrypted virus	D v	D v File$_1$	D v File$_2$	⋯	D v File$_n$
Polymorphic virus	D v	D$_1$v File$_1$	D$_2$v File$_2$	⋯	D$_n$v File$_n$
Metamorphic virus	v	v$_1$ File$_1$	v$_2$ File$_2$	⋯	v$_n$ File$_n$

Detection of Metamorphic Virus: pattern match

- Pure Text patterns are not any more a viable virus detection method
  - Problem of reliably identifying a bounded-length mutating virus is NP-complete (Spinellis IEEE Tn Inf Theory 2003)
    - Proof is based on showing that a virus detector D for a certain virus strain V can be used to solve the satisfiability problem

Code Obfuscation: Use and Misuse

- Obfuscation: modify the code in such a way that it becomes difficult to understand / analyse but the functionality remains the same
- Objective: evade detection by hiding the intent of malware (reverse engg/independent design)
- Common techniques: variable renaming, garbage insertion, code re-ordering, instruction substitution, data and code encapsulation etc.
  - A graph approach to quantitative analysis of control-flow obfuscating transformations, Tsai, Hsin-Yi and Huang, Yu-Lun and Wagner, David, IEEE Trans. Info. For. Sec., 2009
- e.g., JavaScript obfuscation
  - obfuscated JavaScript or HTML code in spam messages
  - when displayed by an HTML-capable e-mail client, appears as a reasonably normal message
  - may exhibit obnoxious JavaScript behaviors such as spawning pop-up windows

Detection: Analyze obfuscations

- Infinite
  - Predict through classical program transformations
  - Model checking via trace behaviour (compare to a known virus pattern) (Seshia, Jha, …)
- Mining Malware behaviour specifications
  - Collection of behaviour traces for known malware and their abstract models and compare (Jha et al.) with benign programs
    - Searching issues (splicing/assembling …)
  - Cannot handle concurrent fragments working together

Detection via Behaviour Abstractions

- <u>Our Method of Behavioural Traces</u>
- <u>For Detecting Metamorphic</u> Virus

Behaviour Based Detection

- Detection Methods based on Semantical Behaviour
    - Program + environment
- Narendra kumar & Shyamasundar EICAR 2010, ICSE 2010, AVAR 2010, EICAR 2011

from	Zdeněk Breitenbacher <Zdenek.Breitenbacher@avg.com>
to	Naren N <naren.nelabhotla@gmail.com>
date	Mon, Jul 26, 2010 at 1:56 PM
subject	Re[6]: EICAR 2010
mailed-by	avg.com

Hello Naren

How are you? Have you got any news about ZMist? How is your research going on?

Currently, AVG starting from build 9.0.0.851 detects all Etap samples which you sent to me, so thank you very much for the cooperation; you helped AVG a lot! According to www.virustotal.com service many other AV vendors still don't detect them.

I am looking forward to news from you.

Best regards,
Zdenek B.

Zdenek Breitenbacher, AVG Technologies
Algoritmic Detection Team Leader
Phone +420549524066, Fax +420549524073
Email zdenek.breitenbacher@avg.com
Pages www.avg.com, www.avg.cz

Benchmarking Program Behaviour for Infection Detection

- Problem Definition: Arrive at techniques for detecting malware and programs infected by malware

Benchmarking Program Behaviour for Infection Detection

- Existing approaches and Shortcomings:
  - Signature-based detection
    - String scanning using a database of known patterns
  - Static analysis of binaries
    - Uses abstraction patterns over CFGs provided by experts
  - Semantics-based detection
    - Templates (instruction sequences with variables and symbolic constants) for each transformation have to be provided by experts
    - Sequences of system calls

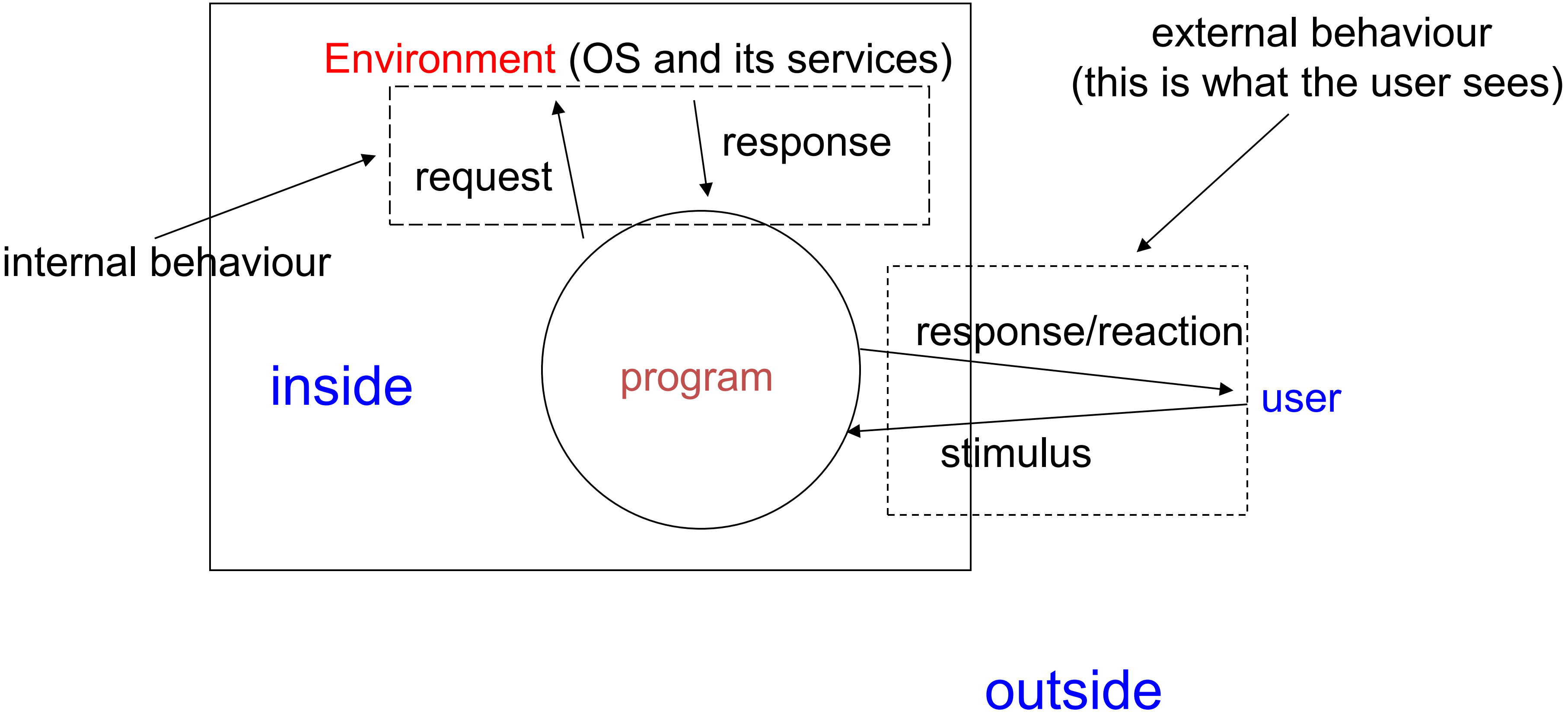
Benchmarking Program Behaviour for Infection Detection

- Existing approaches and Shortcomings:
  - Simple program transformation techniques like introducing *nop* instructions; jumbling the instruction sequence and inserting appropriate jump instructions to retain the control flow; dead-code insertion etc. can easily defeat the existing approaches
  - Rise of techniques for anti-emulation, anti-debugging and anti-disassembly like packing and (poly/meta)morphism make it extremely difficult to analyze binaries

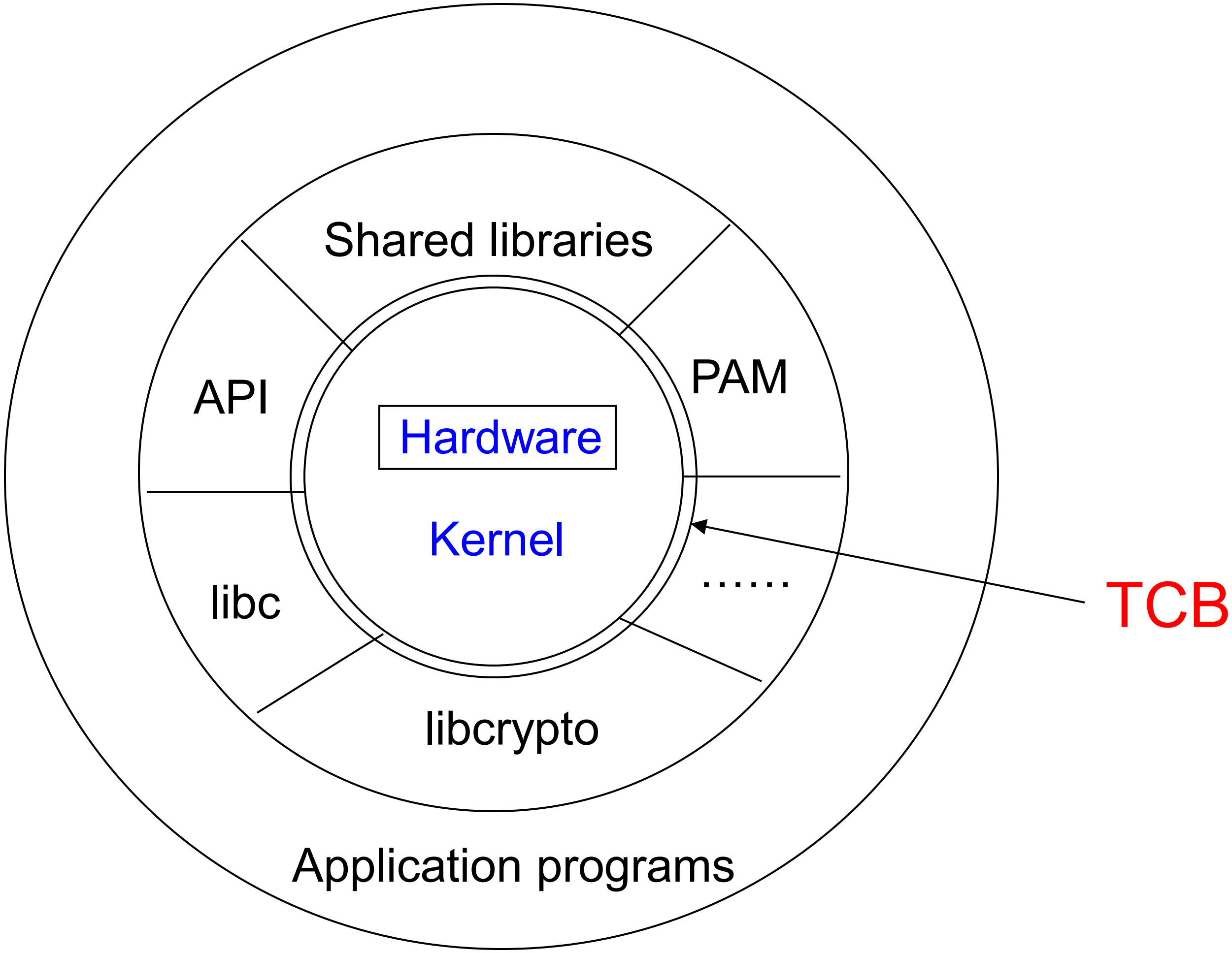
Benchmarking Program Behaviour for Infection Detection

- Our approach: We observe that malware activities are carried out without the consent of the user
  - *always* happen in the background (not observable by the user)
  - an infected browser functions the same as far as the user can see its effects, while it quietly carries out malicious activities
  - make user aware of the background activities and require explicit authorization for suspicious/sensitive operations (eg. Windows Vista OS)

Reactive programs



Logical Layers of OS



Ideas

- Let us restrict ourselves to viruses (parasitic, spread by infecting trusted programs)
- We know the intended behaviour for trusted programs!!!
- Assume that the functionality is left unmodified (else the user can suspect it)
- Run-time monitor the internal behaviour of trusted programs to discover anomaly in behaviour

Behaviour modelling

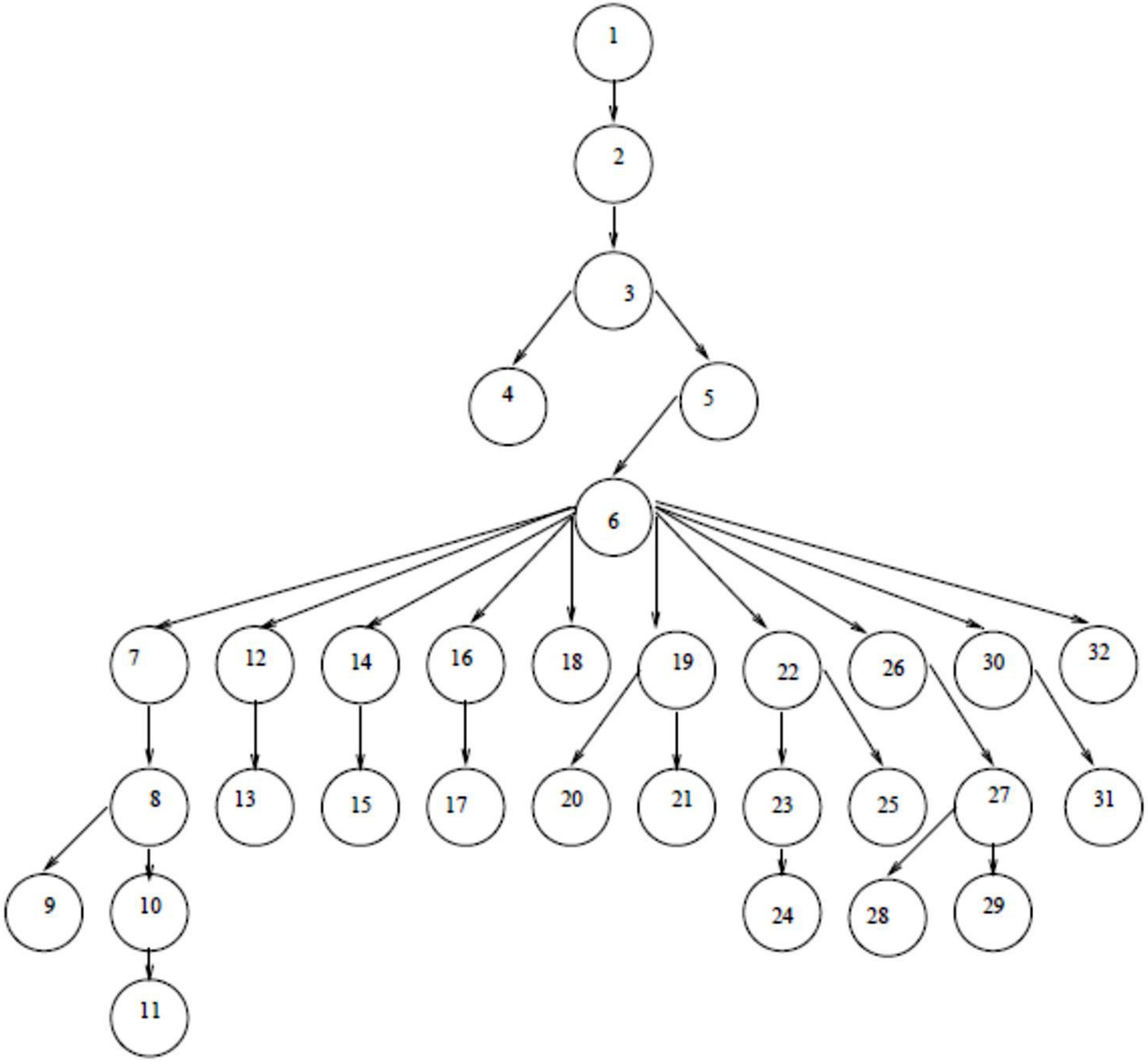
- Informally, a reactive program can be interpreted as follows: the program reacts to stimuli and can be treated as a non-terminating program that provides a finite response in a finite time for a given stimulus
- The external behaviour of such a program can be captured through its interfaces and its responses
- For example, for a vending machine its external behaviour can be given as
  - *place-coin ^ choose-item ^ receive-item*

Behaviour modelling

- During execution of a program *p* with external behaviour *t*, the main process may spawn child processes internally (not necessarily observable to the user) for modularly achieving/computing the final result
- Thus, the total (internal + external) behaviour can be denoted by a tree with processes, data operations etc denoted as nodes and directed edges
- Each node in the tree corresponds to a process and there is a directed edge from node *r* to node *s* if process *s* is the child of process *r*

Behaviour modelling

- We call this the *process tree* of program *p* with external behaviour *t* and in the environment *env*
- We define the system behaviour / internal behaviour of a program as the process tree generated during execution together with the sequence of system calls made by each process (vertex/node) in the tree



process tree generated by *ssh*

Automatic Extraction of Program Behaviour

1. *Collect execution traces*: execute the program and trace the sequence of system calls made by the process and all its children recursively
   - *strace* for Linux and *ProcMon* for Windows
2. *Construct process tree*: using *clone*, *fork* calls made by a process
3. *Label the nodes*: of the process tree with set of files read, written and executed

Comparing behaviours

- Once we have the database D_B of program behaviour benchmarks, we can monitor the execution of programs and validate their observed behaviour
- Steps
  - find a one-to-one correspondence between the process trees of the benchmark and the observed behaviour
  - verify that the set of input and output files of corresponding nodes of the process trees are *similar*
- If either of the above steps fail, we say that there is a strong case for the program being infected

Tree Isomorphism

- For comparing behaviour of a program execution with its benchmark we need to find an isomorphism between the trees denoting their behaviours
  - in practice, we observed that we can find isomorphism by considering these trees as rooted, directed and ordered

Policies for Comparing Corresponding Processes

- Depending on the security required for the system, we could have varying policies which govern the comparison of behaviours of corresponding process
  - for example, if integrity is more of a concern, then we can have the policy stating that set of files written by the current execution should be a subset of those written by the benchmark

Defining Infection

- Behaviour B_2 complies with behaviour B_1 w.r.t. input policy P_i and output policy P_o iff
  - process trees of B_1 and B_2 are isomorphic (*h*)
  - for every process *p* in B_1 and its corresponding process *h(p)* in B_2
    - files read by *h(p)* satisfy policy P_i w.r.t files read by *p*
    - files written by *h(p)* satisfy policy P_o w.r.t files written by *p*

Validating program behaviour

- Validating an observed program behaviour can be done *obtrusively* or *unobtrusively*
- In unobtrusive validation
  - construct the program behaviour as it executes
  - when the program terminates, validate the observed behaviour against the benchmark
- When we suspect a program to be infected / tampered, we can execute it in a quarantined environment and validate its behaviour unobtrusively

Validating program behaviour

- In obtrusive validation
  - stop the program execution whenever it makes a sensitive system call
  - if it's arguments are in compliance with the policy, let it continue execution
  - if not, either prompt the user to authorize this action or terminate the program
  - alternately, either suppress the system call or modify its arguments/return values according to the policy
  - gives the full power of edit automata

Experiments (Malware Detection)

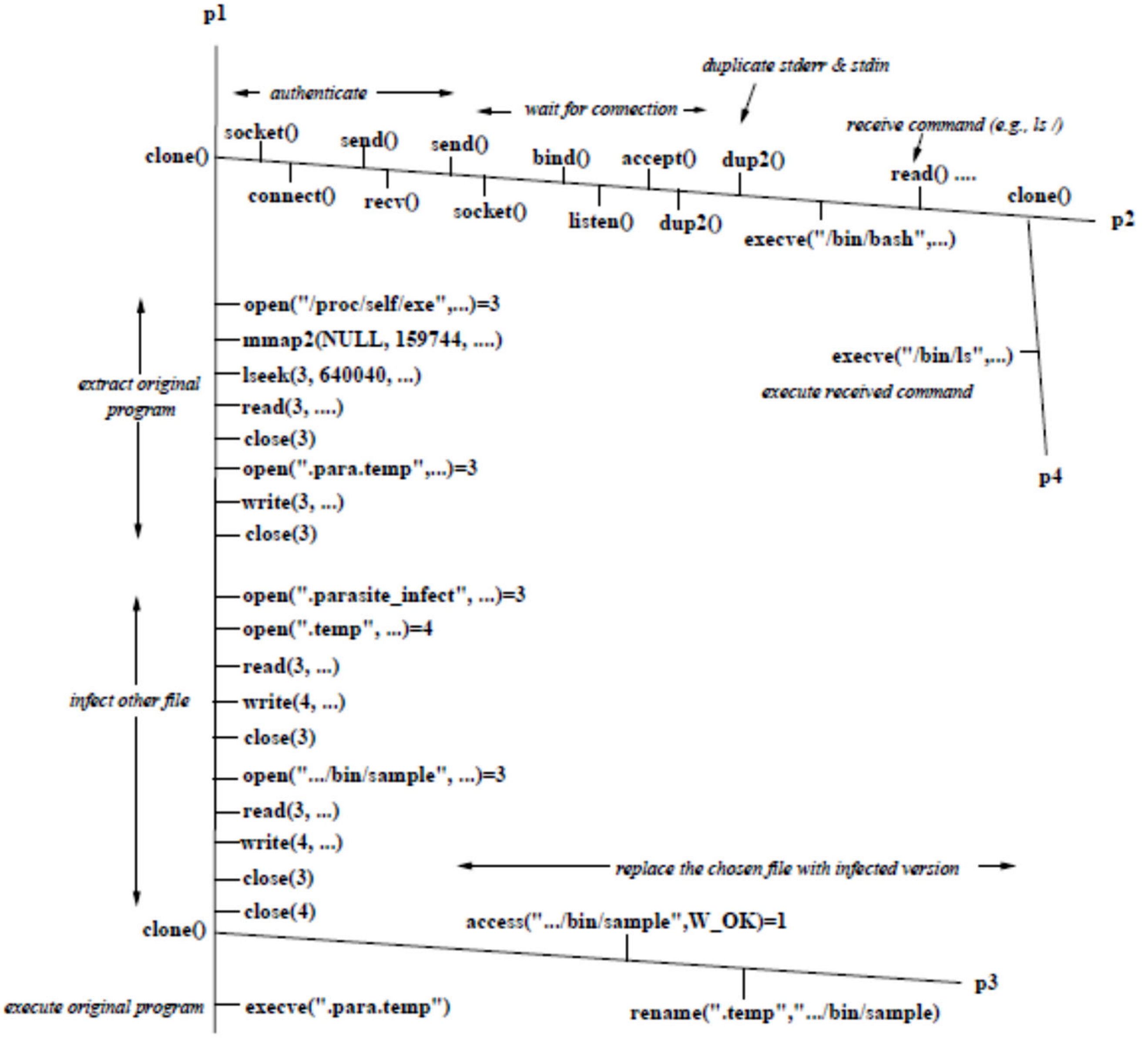
- We benchmarked the behaviour of
  - a text editor (*nano*),
  - a web browser (firefox) and
  - a ssh server
- We then executed the infected versions of these programs and tried to detect the infections using our framework

Experimental Setup

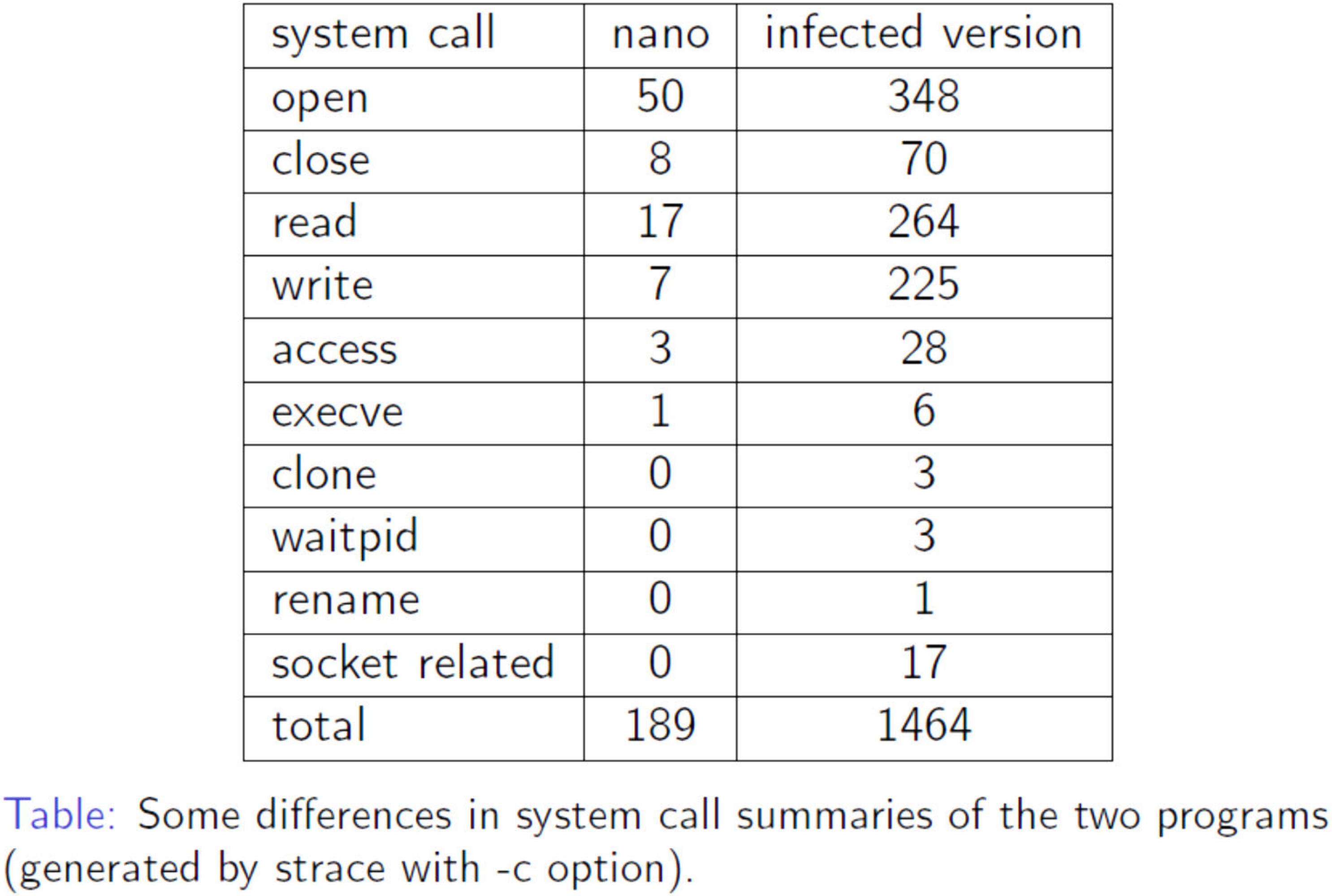
- For *nano* and *firefox* experiments, we took a virus and modified its code to infect a particular file at a particular location
- We pre-pended the virus to *nano* and *firefox* binaries thus infecting them
- When infected *nano* is executed, the virus starts executing (carrying out its payload) and clones a child process to extract and execute *nano* from self

Experiment 1: Infected editor Nano

Process tree and the relevant system calls



nano experiment



system call	nano	infected version
open	50	348
close	8	70
read	17	264
write	7	225
access	3	28
execve	1	6
clone	0	3
waitpid	0	3
rename	0	1
socket related	0	17
total	189	1464

Table: Some differences in system call summaries of the two programs (generated by strace with -c option).

nano experiment

- Some important differences in behaviour
  - original program made 18 different system calls whereas the infected version made 48
  - infected program made network related system calls like *socket*, *connect* etc. whereas the original program did not
  - infected program spawned 3 processes whereas the original program did not spawn any process
  - there is a huge difference in the number of *read* and *write* system calls

nano experiment

- Timing related observations
  - original program spent around 88% on *execve* system call and 12% on *stat64* (*note time is used to check whether memory has been tampered – Seshadri, Khosla .. CMU*)
  - infected version spent 74.17% on *waitpid*, 10.98% on *write*, 6.28% on *read*, 4.27% on *execve* and negligible time on *stat64*
- This indicates that the infected program spent more time waiting on children than in execution
- The increased percentage of time spent on writing and reading by infected program indicates malfunction

Part of trace illustrating the differences

- Extract the original file into a temp file *.para.temp*
  - open("/proc/self/exe", …) = 3
  - lseek(3, …
  - read(3, …
  - open(".para.tmp", …) = 3
  - write(3, …

Part of trace illustrating the differences

- Infecting another file
  - open(“…/.parasiteinfect", …) = 3
  - open(".temp", …) = 4
  - read(3, …
  - write(4, …
  - open(“…/nano", …) = 3
  - read(3, …
  - write(4, …

Part of trace illustrating the differences

- Network activity
  - socket(…) = 3
  - connect(3, …
  - send(3, …
  - recv(3, …
  - socket(…) = 4
  - bind(4, …
  - listen(4, …
  - accept(4, …

Part of trace illustrating the differences

- Receiving *ls* command over network
  - read(0, "l", …
  - read(0, "s", …
  - read(0, " ", …
  - read(0, "/", …
  - read(0, "", …
  - execve("/bin/ls", …

Experiment 2: Infected SSH

ssh experiment

- Behaviour of genuine *ssh* at a high level can be described in the following steps
  - start sshd service
  - wait for a connection and accept a connection
  - authenticate the user
  - prepare and provide a console with appropriate environment
  - manage user interaction and logout
  - stop sshd service

ssh experiment

- Infected version of *ssh* can be used in normal mode or Trojan mode
- When used in normal mode it is supposed to behave exactly like the genuine *ssh*
- When used in Trojan mode it allows a user to login with any valid user-id using a predefined *magic password* in addition to logging every access (user-id and password) in which a user logs in with his genuine password

ssh experiment

- We executed *ssh* 4 times on machine A
  - Run 1: genuine *ssh*
  - Run 2: infected *ssh* in normal mode
  - Run 3: infected *ssh* in trojan mode using *genuine password*
  - Run 4: infected *ssh* in trojan mode using *magic password*

ssh experiment

- We observed the following differences between the genuine ssh program and the infected ssh program
  - When starting the sshd service genuine ssh program uses *kerberos*, *keyutilities* and *pam libraries* which the infected ssh does not use
  - During authentication genuine ssh program uses *kerberos*, *crypto utilities* and *pam libraries* which the infected ssh does not use

Part of trace illustrating the differences

- open("/usr/lib/libgssapikrb5.so.2",…
- open("/usr/lib/libkrb5.so.3", …
- open("/usr/lib/libkrb5support.so.0", …
- open("/lib/libkeyutils.so.1", …
- open("/usr/lib/i686/cmov/libcrypto.so.0.9.8", …
- open("/usr/lib/libk5crypto.so.3", …
- open("/lib/libpam.so.0", …
- open("/etc/pam.d/sshd", …
- open("/lib/security/pamenv.so", …

Part of trace illustrating the differences

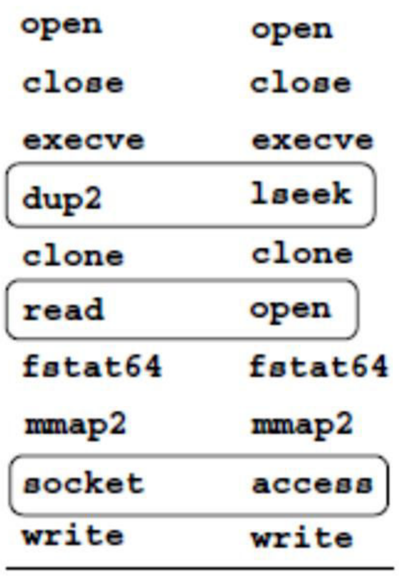
- open("/etc/pam.d/common-auth", …
- open("/lib/security/pamunix.so", …
- open("/lib/security/pamnologin.so", …
- open("/etc/pam.d/common-account", …
- open("/etc/pam.d/common-session", …
- open("/lib/security/pammotd.so", …
- open("/lib/security/pammail.so", …
- open("/lib/security/pamlimits.so", …
- open("/etc/pam.d/common-password", …
- open("/etc/pam.d/other", …

Experiment 3: Firefox web Browser

firefox experiment

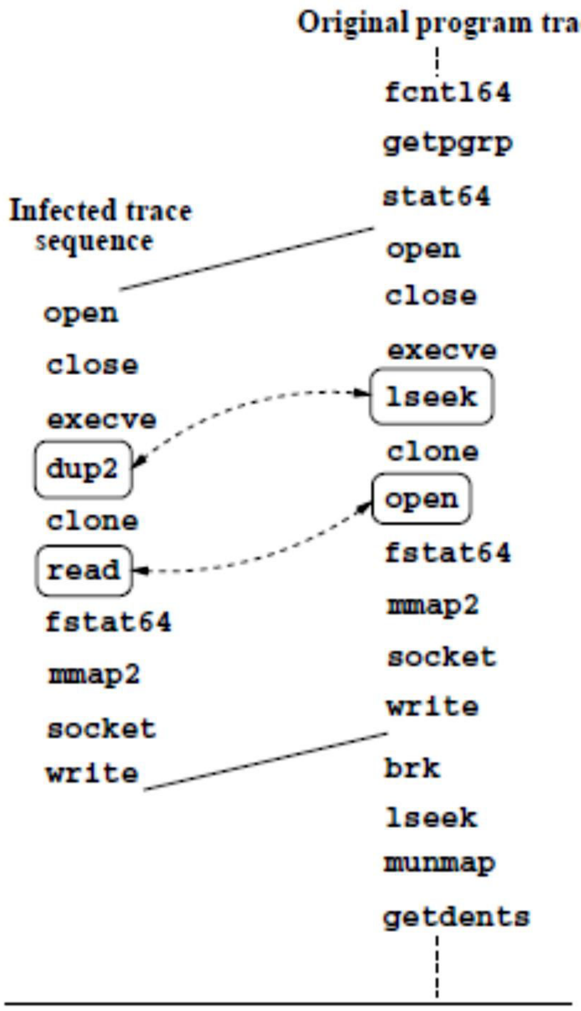
- huge trace file
- we chose to monitor only the main process
- we broke down the entire trace into sequences of 10 consecutive system calls
- for each such sequence in the infected version, we tried to find out the closest sequence in the original program using the notion of minimum hamming distance

Minimum Hamming distance



open	open
close	close
execve	execve
dup2	lseek
clone	clone
read	open
fstat64	fstat64
mmap2	mmap2
socket	access
write	write

Hamming distance = 3



Original program trace

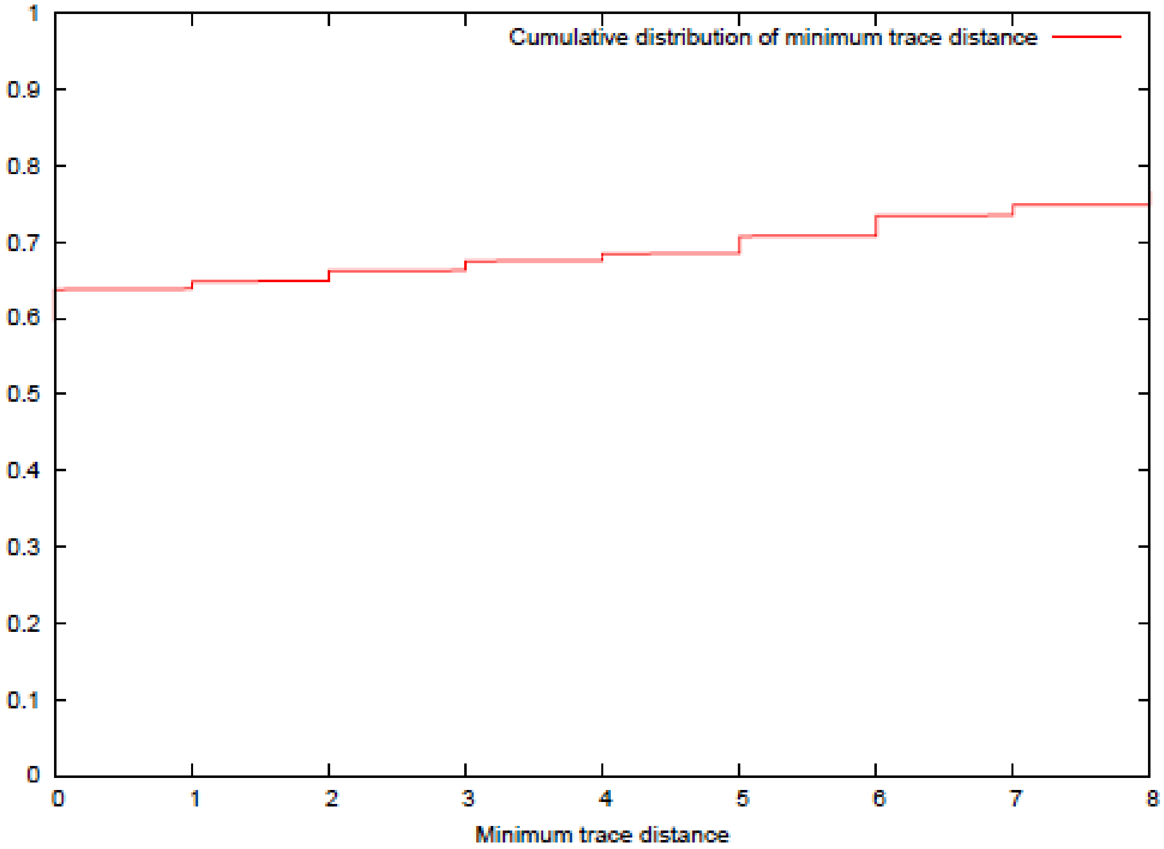
fcntl64
getpgrp
stat64
open
close
execve
lseek
clone
open
fstat64
mmap2
socket
write
brk
lseek
munmap
getdents

Infected trace sequence

open
close
execve
dup2
clone
read
fstat64
mmap2
socket
write

Min. Hamming distance = 2

Observations based on Hamming distance



- Less than 2/3 of the trace exactly matched the original program
- About 1/4 of the trace differed from the original by 80%

Observations based on system call trace

- original version makes 801 system calls whereas the infected version makes 1225 system calls
- infected version calls *read* and *write* a total of 360 times whereas the original version makes 37 *read* system calls and no call to *write*
- original version spawns 10 children whereas the infected version spawns 12
- original version makes 33 different system calls whereas the infected version uses 39 different system calls

Summary of Results

- These experiments demonstrate that our model of program behaviour and matching algorithms are very useful for detecting infections to trusted programs
- In our experiments we found that the size of benchmarks is very small (typically tens of kilobytes)
- Slow down in overall execution time due to monitoring is very negligible (crucially depends on the policy)

Resilience to Transformations

- Demonstrated the usefulness of our approach to detect infection
- Now we will show that our model of behaviour is resilient to the syntactic transformations used to evade current detection techniques
  - Compiler optimization transformations
  - Program obfuscation transformations

Compiler optimization transformations

- We compiled *ssh* using the five optimization levels supported by *gcc* – O0,O1,O2,O3 and Os
- We executed the resulting *ssh* programs and collected their behaviours in similar environments and for the same user interaction
- We observed that the behaviour (process tree with input/output files) of all these versions is exactly the same

Compiler optimization transformations

- When we considered their behaviour (process tree with sequence of system calls) we found the following minor differences
  - number of *time*() calls
  - size of chunks in which data is read

Program Obfuscation Transformations

- We obfuscated *nano*-virus using CObf, a state-of-the-art C-source obfuscation tool
- We collected behaviours of both the virus and its obfuscated version in similar environments and the same user interaction
- We observed that the behaviour (process tree with system call trace) of both the programs is exactly the same

- We can further refine our approach by having a database of known malicious behaviours D_m
  - for example, a sequence of system calls characterizing self-replication/reflection
- When the observed behaviour of a program differs from its benchmark, we can utilize the database D_m to check if the additional behaviour represents a malicious intent
  - possibility of incremental validation and localizing the errors for disinfection

Metamorphic Virus: Characterization as a Regular Expression (a signature)

Semantic Signature Extraction

- Problem Definition: malware industry has to analyze approximately 30,000 to 40,000 suspected samples every day, which necessitates a framework for automatic analysis of programs to classify them and also to aid the human experts to arrive at **algorithmic** ways of signature extraction
  - Virus scanners can be overcome by simple obfuscations (EICAR 2010/1 – Filiol et al.)

Semantic Signature Extraction

- Our approach: since polymorphic and metamorphic code have the capability to change shape across infections, it becomes necessary to have semantic signatures which capture the essential behaviour of the malware
  - We present an algorithmic approach for extracting the semantic signature of malware as a regular expression over API calls, based on algorithms for learning regular expressions

Semantic Signature Extraction

- Merge the abstracted activities of all the threads of all the processes into one single file, sort them according to their time stamps and forget the time stamps
- The resulting file describes the sequence of high-level activities performed by the sample
- Repeat the above steps for some set of variants of a malware and use their sequences to learn a regular expression (under the supervision of a human expert) that forms the semantic signature of the malware

Semantic Signature Extraction

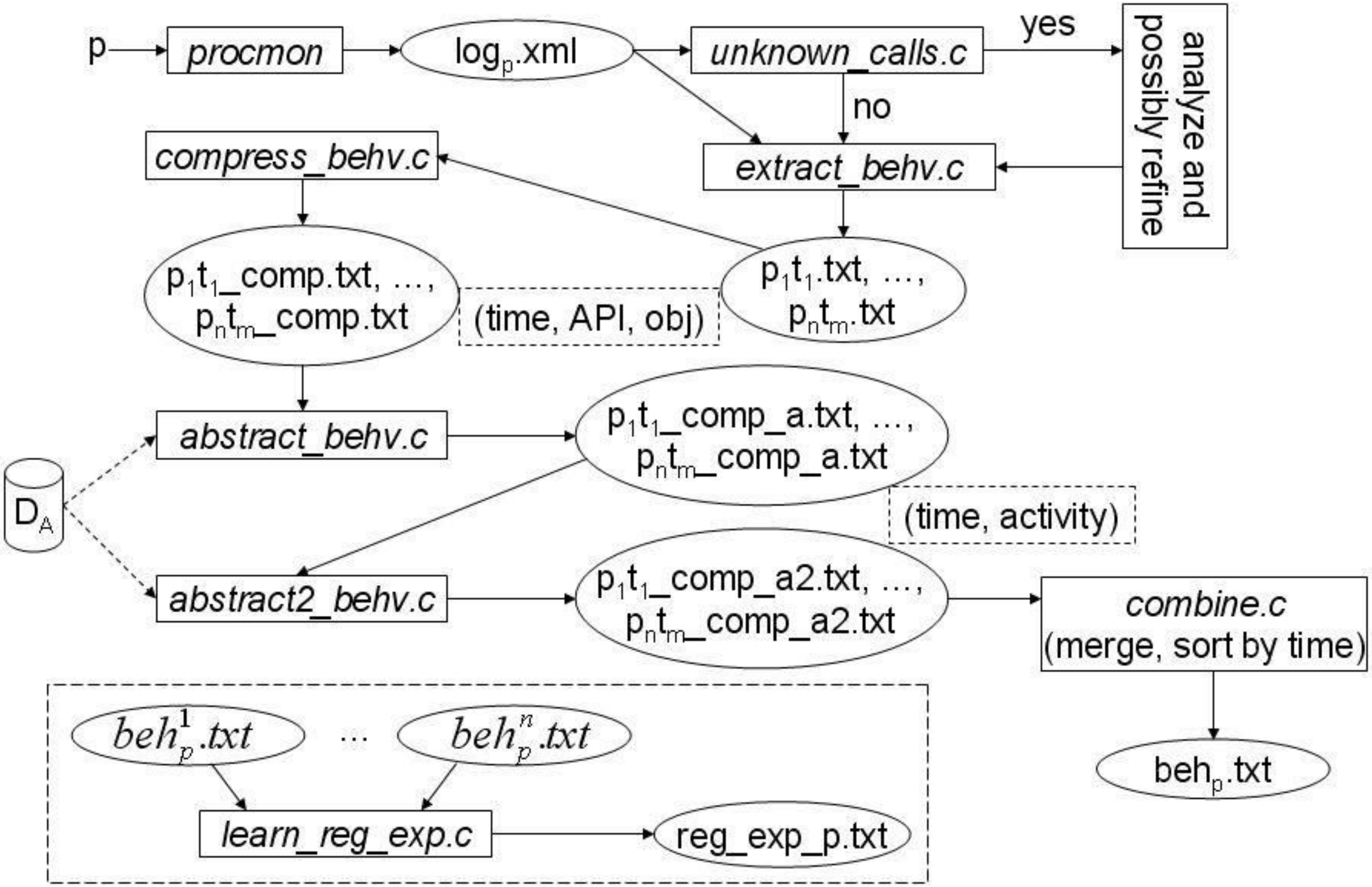
Symbol	Action
A	Read Registry Key/Value
B	Set Registry Value
C	Create Registry Key
D	Delete Registry Key/Value
E	Read File Metadata
F	Read File
G	Write File Metadata
H	Write File
I	Query Directory
L	TCP/UDP Send
M	TCP/UDP Receive
N	TCP/UDP Reconnect

Classification of Security Relevant API calls

Semantic Signature Extraction Algorithm

- Construct malware behaviour from its execution trace
- Filter to keep only security sensitive parts of the trace corresponding to each node in the tree
- Abstract the trace into sequences of high-level activities
- Using time-stamps interleave the abstracted activities of sub-processes to get a sequence of high-level activities
- Repeat the above steps for several instances / versions of the same malware and learn a regular expression that denotes its semantic signature

Implementation Architecture


p
procmon
log_p.xml
unknown_calls.c
yes
analyze and possibly refine
no
extract_behv.c
compress_behv.c
p_1t_1.txt, …, p_nt_m.txt
p_1t_1_comp.txt, …, p_nt_m_comp.txt
(time, API, obj)
D_A
abstract_behv.c
p_1t_1_comp_a.txt, …, p_nt_m_comp_a.txt
(time, activity)
abstract2_behv.c
p_1t_1_comp_a2.txt, …, p_nt_m_comp_a2.txt
combine.c (merge, sort by time)
beh_p.txt
beh_p^1.txt
…
beh_p^n.txt
learn_reg_exp.c
reg_exp_p.txt

Experimental Evaluation

- Successfully extracted the signature and used it to detect (also to predict) variants of in-the-wild malware – *Sality*, *Etap*, *Netsky*, *Beagle*, *MyDoom*
- Checked 38 infected (by Etap) files against 4 popular commercial anti-virus products (with latest updates) and observed the following

Antivirus Product	No.of inf files detected
Norton Antivirus 2009	38
Kaspersky Internet Security 2010	38
AVG Internet Security Business Edition 9.0	25*
Avast Free Antivirus 5.0	14
Our signature	**38**

from	Zdeněk Breitenbacher <Zdenek.Breitenbacher@avg.com>
to	Naren N <naren.nelabhotla@gmail.com>
date	Mon, Jul 26, 2010 at 1:56 PM
subject	Re[6]: EICAR 2010
mailed-by	avg.com

Hello Naren

How are you? Have you got any news about ZMist? How is your research going on?

- **Currently, AVG starting from build 9.0.0.851 detects all Etap samples which you sent to me, so thank you very much for the cooperation; you helped AVG a lot! According to www.virustotal.com service many other AV vendors still don't detect them.**

I am looking forward to news from you.

Best regards,
Zdenek B.

Zdenek Breitenbacher, AVG Technologies
Algoritmic Detection Team Leader
Phone +420549524066, Fax +420549524073
Email zdenek.breitenbacher@avg.com
Pages www.avg.com, www.avg.cz

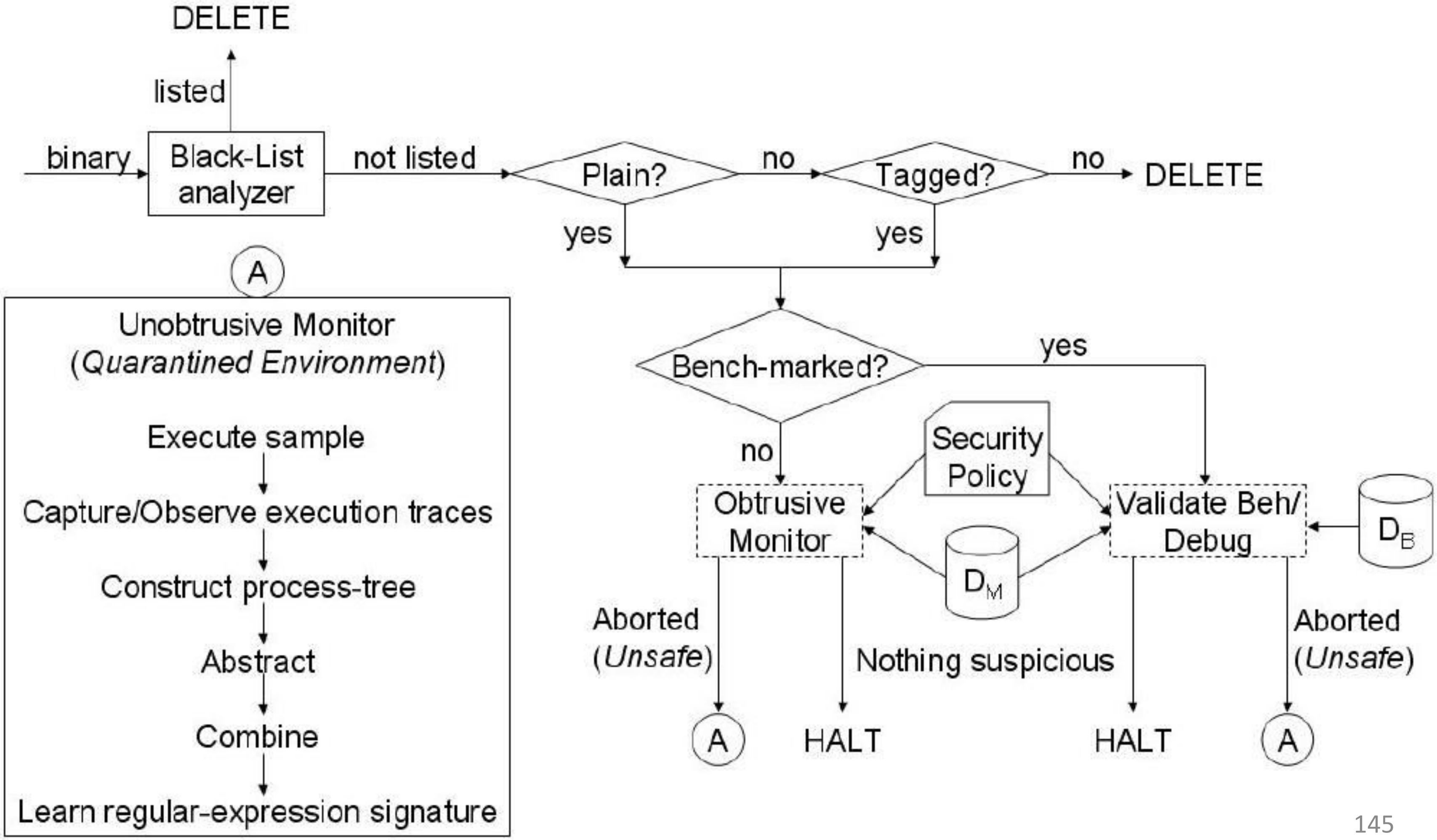
Summary of our Contributions

- Developed an algorithm to almost automatically extract the semantic signature (as regular expression over API calls) of a malware sample
- Demonstrated the usefulness of our semantic signatures for detecting several known variants of a malware and predicting possible future variants
  - analyzed in-the-wild metamorphic viruses Sality and Etap; and email-worms Netsky, MyDoom and Beagle

Results

- We have learning algorithm approach to extract the semantic signatures of malware with the following advantages
  - leads to algorithmic detection
  - signature extracted subsumes variants of viruses
  - makes proactive detection possible i.e. predicts new variants
  - extracted signatures nicely correspond to the descriptions given by industry experts

Architecture of Monitoring



DELETE

listed

binary

Black-List analyzer

not listed

Plain?

no

Tagged?

no

DELETE

yes

yes

A

Unobtrusive Monitor (*Quarantined Environment*)

Execute sample

Capture/Observe execution traces

Construct process-tree

Abstract

Combine

Learn regular-expression signature

Bench-marked?

yes

no

Security Policy

Obtrusive Monitor

Validate Beh/ Debug

D_B

D_M

Aborted (*Unsafe*)

Nothing suspicious

Aborted (*Unsafe*)

A

HALT

HALT

A

Summary (Contd)

  - effort needed to evade being detected by our signatures is much higher than evading traditional signatures
- Built prototypes for automating several steps involved in the process
- Provided experimental evidence for the efficacy of the approach in detecting and predicting variants of metamorphic viruses

Challenges and Future Work

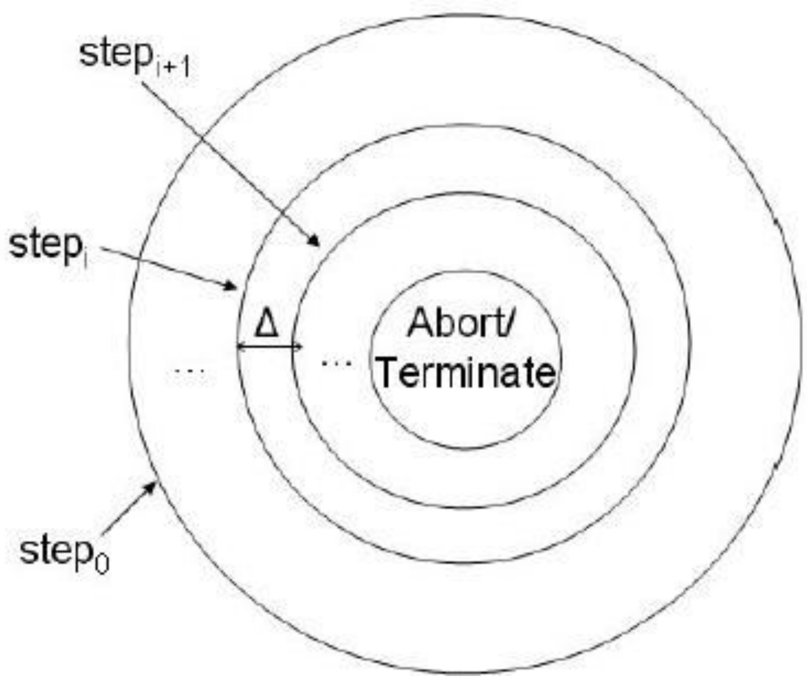
- For the proposed approach to be useful in practice (i.e. for better performance) we need to translate these semantic signatures into syntactic signatures (static analysis of binaries by over-approximation)
- Develop notions of supervised learning to automate the learning process which is a crucial step for signature extraction

Semantic Signature Extraction

- Can we capture performance-centric signature from the regular expression abstraction?

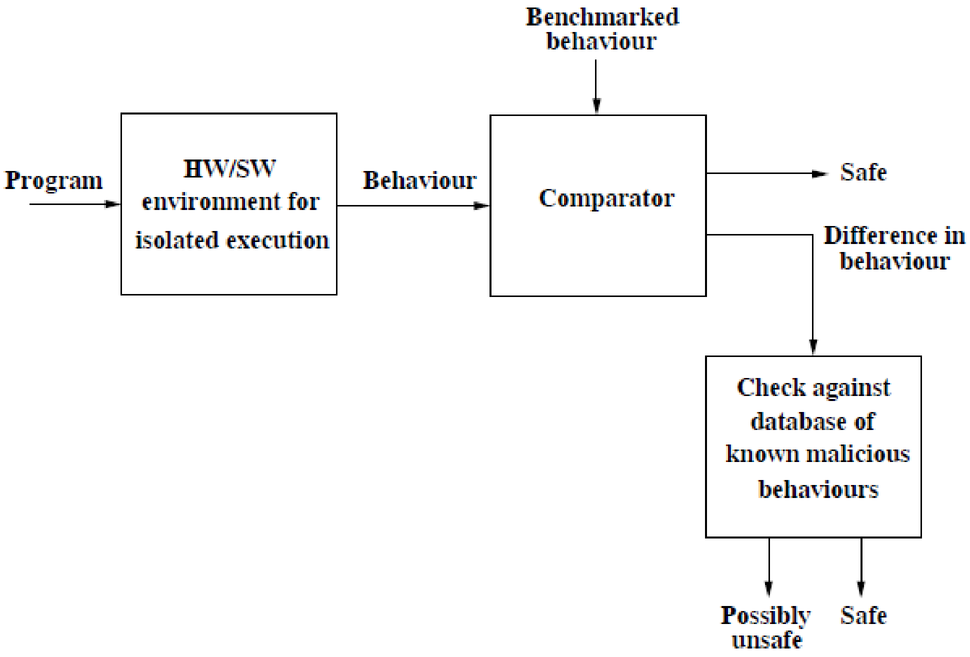
Debugging Environment

Zeller (Delta Debugging)



a. Arrive at textual abstractions

b. Forensics Tool


Benchmarked behaviour
Program
HW/SW environment for isolated execution
Behaviour
Comparator
Safe
Difference in behaviour
Check against database of known malicious behaviours
Possibly unsafe
Safe