Static Analysis of Binaries for Malicious Code Detection

Mihai Christodorescu, Somesh Jha
{mihai, jha}@cs.wisc.edu
University of Wisconsin, Madison
Arms Race

- Vanilla virus
- Register renaming
- Packing/encryption
- Code reordering
- Code integration

Signatures
- Regex signatures
- Emulation/heuristics
- ?
- ?
## Dismal State of the Art

**Commercial antivirus tools vs. morphed versions of known viruses**

<table>
<thead>
<tr>
<th></th>
<th>Norton AntiVirus</th>
<th>McAfee VirusScan</th>
<th>COMMAND AntiVirus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chernobyl-1.4</td>
<td>× Not detected</td>
<td>× Not detected</td>
<td>× Not detected</td>
</tr>
<tr>
<td>fOsf0r0</td>
<td>× Not detected</td>
<td>× Not detected</td>
<td>× Not detected</td>
</tr>
<tr>
<td>Hare</td>
<td>× Not detected</td>
<td>× Not detected</td>
<td>× Not detected</td>
</tr>
<tr>
<td>z0mbie-6.b</td>
<td>× Not detected</td>
<td>× Not detected</td>
<td>× Not detected</td>
</tr>
</tbody>
</table>

**Obfuscations used in morphing:** NOP insertion, code reordering
Worst-Case Scenario

• Each infection generates a worm morphed differently
Clear Danger

• Unlimited variants can be cheaply generated
  - Practically undetectable
• Obfuscations: part of the virus propagation step

◆ Threat of highly mobile, highly morphing malicious code
Obfuscation Example

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
  pop    ecx
  jecxz SFModMark
  mov    esi, ecx
  mov    eax, 0d601h
  pop    edx
  pop    ecx
  call   edi
  jmp    Loop
```

Morphed Virus Code:

```
Loop:
  pop    ecx
  nop
  jecxz SFModMark
  xor    ebx, ebx
  beqz   N1

N1:
  mov    esi, ecx
  nop
  mov    eax, 0d601h
  pop    edx
  pop    ecx
  nop
  call   edi
  xor    ebx, ebx
  beqz   N2

N2:
  jmp    Loop
```
Obfuscation Example

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
  pop  ecx
  jecxz SFModMark
  mov  esi, ecx
  mov  eax, 0d601h
  pop  edx
  pop  ecx
  call edi
  jmp Loop
```

Morphed Virus Code:

```
Loop:
  pop  ecx
  pop  edx
  mov  eax, 0d601h
  pop  ecx
  pop  edx
  call edi
  jmp Loop

N2:
  nop
  mov  eax, 0d601h
  pop  edx
  pop  ecx
  nop
  jecxz SFModMark
  xor  ebx, ebx
  beqz N1
  jmp Loop

N1:
  mov  esi, ecx
```
Obfuscation Example

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
  pop      ecx
  jecxz    SFModMark
  mov      esi, ecx
  mov      eax, 0d601h
  pop      edx
  pop      ecx
  call     edi
  jmp      Loop
```

Morphed Virus Code:
```
Loop:
  pop      ecx
  nop
  jmp      L1
L3:
  call     edi
  xor      ebx, ebx
  beqz     N2
N2:
  jmp      L4
L2:
  nop
  mov      eax, 0d601h
  pop      edx
  pop      ecx
  nop
  jmp      L3
L1:
  jecxz    SFModMark
  xor      ebx, ebx
  beqz     N1
N1:
  mov      esi, ecx
  jmp      L2
L4:
```

---

August 7, 2003

Mihai Christodorescu  mihai@cs.wisc.edu
WiSA Project  http://www.cs.wisc.edu/wisa
Code Integration

• Integration of virus and program
Our Solution

Better virus scanner:

- Analyze the program semantic structure
  - Control flow
  - Data flow

- Build on existing static analyses
Overview

- Threats
- Current detection limitations
- Detector design and architecture
- Sample detection
- Performance
- Future work and conclusions
Design Goals

• Static analysis
  - Provides safe results: identifies possible malicious sequences
  - Immune to anti-emulation techniques

• Identify malicious intent
  - Same behavior can be achieved through many implementations
Static Analysis of Binaries

- Detection is as good as the static analyses available
  - More predicates \(\rightarrow\) better detection
  - Better predicates \(\rightarrow\) fewer false alarms

Example: pointer analysis (P.A.)
- No P.A.: it is safe to assume all pointers point to all memory locations
- With P.A.: reduced cost to attain safety
Malicious Code Blueprint

Program to analyze

Pattern Library

Annotator

Annotated Program

Detector

Yes/No

Program infected with obfuscated virus

IDA Pro + CodeSurfer + custom analysis code

Custom-built
Infection:

Program

Vanilla Virus
Detection: 1) Virus Blueprint

Vanilla Virus

Virus Specification
Detection: 2) Deobfuscation

1. Detect code reordering
Detection: 2) Deobfuscation

1. Detect code reordering
2. Detect register renaming
Detection: 2) Deobfuscation

Program

1. Detect code reordering
2. Detect register renaming
3. Detect irrelevant code
What is irrelevant code?

• Code does not change program behavior:
  - NOPs
  - Jumps/branches that do not change the control flow
  - Code that modifies dead registers
  - Code that do not modify the program state
    • e.g.: add ebx, 1
      sub ebx, 1

• Theorem provers can be used to find irrelevant code
Detection: 3) Matching

Annotated Program

Virus Specification

\[ \approx \]
Detection in Theory

oodles General detection problem is undecidable:
Chess, White An undetectable computer virus (VBC’00)

Static analysis is undecidable as well:
Landi Undecidability of static analysis (LOPLAS’92)

(Computationally-bound) obfuscation is impossible
Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, Yang
On the (im)possibility of obfuscating programs (CRYPTO’01)
Detection in Practice

• Our approach is geared to common obfuscations in the wild

• Detection algorithm is matched against current obfuscation threats
  - Can handle more variants than signatures
Building block: **Patterns**

Two components:

1. sequence of instructions
2. predicate controlling pattern application

- Predicates use **static analysis results**
Defeating Garbage Insertion

Pattern:

\[
\begin{align*}
\text{instr } 1 \\
\ldots \\
\text{instr } N \\
\text{where} \\
\text{Delta( state pre } 1, \text{ state post } N) = 0
\end{align*}
\]
Defeating Register Renaming

- Use uninterpreted symbols

Program 1:
```
mov ebp, [ebx]
nop
mov bp, [ebx-04h]
test ebx
beqz next
next: lea esi, MyHook - @1[ecx]
```

Program 2:
```
mov eax, [ecx]
nop
mov ax, [ecx-04h]
test edx
beqz next
next: lea ebi, MyHook - @1[ebx]
```

Virus Spec with Uninterpreted Symbols:
```
mov X, [Y]ebx
```

Matches both Programs 1 and 2
Defeating Code Reordering

\[
\begin{align*}
\text{<instruction A>} \\
\text{<instruction B>}
\end{align*}
\]

\[
\begin{align*}
\text{L}_2: & \quad \text{<instruction B>} \\
\text{L}_1: & \quad \text{<instruction A>} \\
\text{L}_3: & \quad \text{...}
\end{align*}
\]

\[
\begin{align*}
\text{jmp L}_1 \\
\text{jmp L}_3 \\
\text{jmp L}_2
\end{align*}
\]
Defeating Code Reordering

Construct CFG:

```
jmp L_1
instruction A
jmp L_2
instruction B
jmp L_3
```

Input:
```
jmp L_1
L_2: <instruction B>
jmp L_3
L_1: <instruction A>
jmp L_2
L_3: ...
```
Defeating Code Reordering

Pattern:

\[
\text{jmp \ TARGET}
\]
\[\text{where}
\]
\[
\text{Count( CFGPredecessors( TARGET ) )} \neq 1
\]
Prototype Implementation

• The detection tool can handle:
  ✓ NOP-insertion
  ✓ Code reordering (irrelevant jumps and branches)
  ✓ Register renaming

• Work in progress to detect:
  - Malicious code split across procedures
    (need inter-procedural analysis)
  - Obfuscations using complex data structures
    (need integration with pointer analyses)
Testing Setup

Goals:

- **Measure true negatives and false positives**
  - Scan a representative collection of benign programs

- **Measure true positives and false negatives**
  - Scan a set of viruses obfuscated with various parameters

- **Measure performance**
Results

Effectiveness:

**False positive rate**: 0
All benign programs passed the scans.

**False negative rate**: 0
All obfuscated viruses were detected.

**But** there are obfuscations we cannot yet detect.
Performance

Detector avg. | Annotator avg.

- tiffdither (6656) | 7.363 | 1
- winmine (12120) | 17.950 | 1
- spyxx (307200) | 224.584 | 1
- QuickTimePlayer (499712) | 959.913 | 1
Performance Implications

- Combine with other techniques to amortize cost

E.g.: Secure checksum database
Performance Implications

• Combine with other techniques to amortize cost

E.g.: Secure checksum database
Future Directions

• New languages
  – Scripts: Visual Basic (in progress), ASP, JavaScript
  – Multi-language malicious code

• Attack diversity
  – Beyond virus patterns: worms, trojans

• Irrelevant sequence detection
  – Decision procedures
  – Theorem provers
Conclusions

Viruses can self-modify as they propagate.

Current virus scanners cannot detect such malware.

Our semantic analysis can defeat obfuscations and detect viruses.
Related Work

• **Metaccompilation:**
  Ashcraft, Engler *Using programmer-written compiler extensions to catch security holes* (Oakland'02)

• **Theorem proving for security properties:**
  Chess *Improving computer security using extended static checking* (Oakland'02)

• **Model checking programs for security properties:**
  Chen, Wagner *MOPS: an infrastructure for examining security properties of software* (CCS'02)

• **Malicious code filter:**

• **Inline reference monitors**
  Erlingsson, Schneider *IRM enforcement of Java stack inspection* (Oakland'00)
Static Analysis of Binaries for Malicious Code Detection

Mihai Christodorescu, Somesh Jha
{mihai,jha}@cs.wisc.edu
University of Wisconsin, Madison

WiSA Project
http://www.cs.wisc.edu/wisa