General Purpose
Binary Rewriting

Mihai Christodorescu
mihai@cs.wisc.edu

WiSA  http://www.cs.wisc.edu/wisa
University of Wisconsin, Madison
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting
Binary Rewriting

- Transform a binary executable based on input rules
  - Add / remove / edit code
Binary Rewriting

• **No source code is needed**
  - Commercial component software
  - Independent of programming language
    • Treats multi-language systems consistently

• **Complete access to the binary**
  - Not affected by optimizations
Binary Rewriting

- Works regardless of program design
  - Similar to AOP
  - Apply transforms across the program

- Extremely powerful
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting
Binary Rewriting Applications

• Historically:
  - Program optimization
    • Based on profiling data
  - Performance instrumentation
  - Virus / worm infection
    • Using obfuscation
Program Obfuscation

• Used by viruses
  - Add code to make detection harder
  - No change in program behavior

• Competition:
  - Antivirus tools try to deobfuscate
  - Virus writers try to obfuscate
Program Obfuscation

• Semantic NOP Insertion
  - Add code fragment that does not modify program behavior

• Variable / Register Renaming
  - Change the name of program variables

• Instruction Reordering
  - Change the order of instructions without modifying program behavior

• Encode Program
### Obfuscation Example for x86

#### Virus Code
*(from Chernobyl CIH 1.4):*

<table>
<thead>
<tr>
<th>Loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop ecx</td>
</tr>
<tr>
<td>jecxz SFModMark</td>
</tr>
<tr>
<td>mov esi, ecx</td>
</tr>
<tr>
<td>mov eax, 0d601h</td>
</tr>
<tr>
<td>pop edx</td>
</tr>
<tr>
<td>pop ecx</td>
</tr>
<tr>
<td>call edi</td>
</tr>
<tr>
<td>jmp Loop</td>
</tr>
</tbody>
</table>

#### Morphed Virus Code
*(based on Chernobyl CIH 1.4):*

<table>
<thead>
<tr>
<th>Loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop ecx</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>jecxz SFModMark</td>
</tr>
<tr>
<td>xor ebx, ebx</td>
</tr>
<tr>
<td>beqz N1</td>
</tr>
<tr>
<td>N1:</td>
</tr>
<tr>
<td>mov esi, ecx</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>mov eax, 0d601h</td>
</tr>
<tr>
<td>pop edx</td>
</tr>
<tr>
<td>pop ecx</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>call edi</td>
</tr>
<tr>
<td>xor ebx, ebx</td>
</tr>
<tr>
<td>beqz N2</td>
</tr>
<tr>
<td>N2:</td>
</tr>
<tr>
<td>jmp Loop</td>
</tr>
</tbody>
</table>
# Obfuscation Example for x86

**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**  
(based on Chernobyl CIH 1.4):

```
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 for x86

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
(based on Chernobyl CIH 1.4):

Loop:
    pop    ecx
    jmp    L1
L1:
    jecxz  SFModMark
    xor    ebx, ebx
    beqz   N2
    jmp    Loop
L2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L4
L3:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L4:
    xor    ebx, ebx
    beqz   N1
    jmp    L2
N1:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
N2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L3:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L4:
    xor    ebx, ebx
    beqz   N1
    jmp    L2
N2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L3:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L4:
    xor    ebx, ebx
    beqz   N1
    jmp    L2
N2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L3:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L4:
    xor    ebx, ebx
    beqz   N1
    jmp    L2
N2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L3:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L4:
    xor    ebx, ebx
    beqz   N1
    jmp    L2
N2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L3:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L4:
    xor    ebx, ebx
    beqz   N1
    jmp    L2
N2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L3:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L4:
    xor    ebx, ebx
    beqz   N1
    jmp    L2
N2:
    mov    esi, ecx
    mov    eax, 0d601h
    pop    edx
    pop    ecx
    call   edi
    jmp    L3
L3:
Obfuscation Example for x86

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
(based on Chernobyl CIH 1.4):

```
Loop:
    pop    ecx
    jmp    L1
L4:
    xor    ebx, ebx
    beqz   N2
N2:
    jmp    Loop
L1:
    nop
    jecxz  SFModMark
    xor    ebx, ebx
    jmp    L2
L3:
    pop    ecx
    nop
    call   edi
    jmp    L4
L2:
    beqz   N1
N1:
    mov    esi, ecx
    nop
    mov    eax, 0d601h
    pop    edx
    jmp    L3
```
## Obfuscation Example for x86

### Virus Code
(from Chernobyl CIH 1.4):

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td>ecx</td>
</tr>
<tr>
<td>jecxz</td>
<td>SFModMark</td>
</tr>
<tr>
<td>mov</td>
<td>esi, ecx</td>
</tr>
<tr>
<td>mov</td>
<td>eax, 0d601h</td>
</tr>
<tr>
<td>pop</td>
<td>edx</td>
</tr>
<tr>
<td>pop</td>
<td>ecx</td>
</tr>
<tr>
<td>call</td>
<td>edi</td>
</tr>
<tr>
<td>jmp</td>
<td>Loop</td>
</tr>
</tbody>
</table>

### Morphed Virus Code
(based on Chernobyl CIH 1.4):

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td>eax</td>
</tr>
<tr>
<td>jmp</td>
<td>L1</td>
</tr>
<tr>
<td>L4:</td>
<td>xor</td>
</tr>
<tr>
<td></td>
<td>ebx, ebx</td>
</tr>
<tr>
<td>beqz</td>
<td>N2</td>
</tr>
<tr>
<td>N2:</td>
<td>jmp</td>
</tr>
<tr>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>L1:</td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td>jecxz</td>
</tr>
<tr>
<td></td>
<td>SFModMark</td>
</tr>
<tr>
<td>xor</td>
<td>ebx, ebx</td>
</tr>
<tr>
<td>jmp</td>
<td>L2</td>
</tr>
<tr>
<td>L3:</td>
<td>pop</td>
</tr>
<tr>
<td></td>
<td>eax</td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>edi</td>
</tr>
<tr>
<td>jmp</td>
<td>L4</td>
</tr>
<tr>
<td>L2:</td>
<td>beqz</td>
</tr>
<tr>
<td>N1:</td>
<td>mov</td>
</tr>
<tr>
<td></td>
<td>esi, eax</td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>ecx, 0d601h</td>
</tr>
<tr>
<td>pop</td>
<td>edx</td>
</tr>
<tr>
<td>jmp</td>
<td>L3</td>
</tr>
</tbody>
</table>
Obfuscation Example for x86

Virus Code
(from Chernobyl CIH 1.4):

<table>
<thead>
<tr>
<th>Loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop ecx</td>
</tr>
<tr>
<td>jecxz SFModMark</td>
</tr>
<tr>
<td>mov esi, ecx</td>
</tr>
<tr>
<td>mov eax, 0d601h</td>
</tr>
<tr>
<td>pop edx</td>
</tr>
</tbody>
</table>

Morphed Virus Code
(based on Chernobyl CIH 1.4):

<table>
<thead>
<tr>
<th>Loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop eax</td>
</tr>
<tr>
<td>jmp L1</td>
</tr>
<tr>
<td>L4: xor ebx, ebx</td>
</tr>
<tr>
<td>beqz N2</td>
</tr>
<tr>
<td>N2: jmp Loop</td>
</tr>
<tr>
<td>L1: nop</td>
</tr>
</tbody>
</table>

Morphed Virus Code
(based on Chernobyl CIH 1.4):

| L3: pop eax |
| nop |
| call edi |
| jmp L4 |

Morphed Virus Code
(based on Chernobyl CIH 1.4):

| L2: beqz N1 |
| N1: mov esi, eax |
| nop |
| mov ecx, 0d601h |
| pop edx |
| jmp L3 |
Obfuscation Example for VB

Worm Code
(based on AnnaKournikova worm):

```
On Error Resume Next
...
Set outlookObj = CreateObject("Outlook.Application")
...
For Each addressObj In addressBookObj
    ...
    newMsgObj.Send
    ...
Next
...
'Vbswg 1.50b
```
Obfuscation Example for VB

Worm Code
(based on AnnaKournikova worm):

Execute "On Error Resume Next\n...\nSet outlookObj = CreateObject("Outlook.Application")\n...\nFor Each addressObj In addressBookObj\n...\n    newMsgObj.Send\n...\nNext\n...\n'Vbswg 1.50b"
Obfuscation Example for VB

Worm Code
(based on AnnaKournikova worm):

Execute F( "X)udQ0Vpgjn... udiy3^Q70d2" )

Function F(S)
    For I = 1 To Len(S) Step 2
        X = Mid(S, I, 1)
        ...
        If Asc(X) = 15 Then
            ...
        End If
        ...
    Next
End Function
Program Obfuscation

• Obfuscation is a technique used by virus writers

=> Virus detection tools have to handle obfuscations

• Then, to test our virus detection tool, we need to rewrite infected binaries to add obfuscations
Obfuscating...

Commercial antivirus tools vs. morphed versions of known viruses

<table>
<thead>
<tr>
<th></th>
<th>Norton AntiVirus</th>
<th>Command</th>
<th>McAfee VirusScan</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>f0sf0r0</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>
Program Obfuscation

• The obfuscation war is not lost
  - We can effectively deobfuscate many idioms
  - More later

• Benefit of obfuscation:
  - IP protection (harder to reverse engineer)
 Bounds Checking

- \(C, \ C++\) do not perform bounds checking on array accesses
  - Possible buffer overflow due to programming errors

- Patch all string buffer and array accesses to check for length

=> No more buffer overflows
Sandboxing

• Restrict untrusted program’s access to OS interface:
  - Contain disk access & memory usage
  - Allow / deny network connections
  - No OS modification
Security Policy Enforcement

• Similar to Engler’s metaccompilation work

• Enforce rules not supported by the standard OS security mechanisms:
  - Sanitize untrusted input
  - Do not release sensitive data to users
  - Check custom permissions before doing operation X
Security Policy Enforcement

• Powerful to check at runtime:
  - Programming errors that can lead to security flaws
  - Security violations

• Technique:
  - Add code that at runtime enforces desired rules
Program Monitoring

- Monitor a running program to prevent malicious modification

- A monitor process will:
  - Trace the events produced by a running program
  - Make sure the events match what is expected
Program Monitoring

- Monitoring can be remote or local
- Flexible policy rules
Program Monitoring

• Problem:
  Certain event sequences are ambiguous

• Solution:
  Modify program to eliminate ambiguity
  (as much as possible)
  - Insert code to send special events
  - See Jon Giffin’s work on IDS
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting
Existing Binary Rewriters

- Limited in functionality or scope
- Some still in prototype mode
Existing Binary Rewriters

- **EEL (Executable Editing Library)**
  (U. Wisconsin)
  - Works only on SPARC binaries

- **Etch Binary Rewriter**
  (U. Washington)
  - Works only on x86 Windows
  - Not available as a separate library
Existing Binary Rewriters

• **Byte Code Engineering Library**  
  *(Open source, Apache Foundation)*  
  - Java specific

• **OM / ATOM**  
  *(DEC WRL/Compaq WRL/HP?)*  
  - proprietary
Existing Binary Rewriters

- **DynInst**
  (U. Wisconsin, U. Maryland)
  - Geared towards instrumenting running programs
  + Can handle multiple types of binaries
The Problem

- WiSA project relies on several tools:
  - EEL, IDA Pro, CodeSurfer + custom code

- Incomplete solution:
  - Platform specific (not cross platform)
  - Missing features, yet not easily extensible
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting
Goal

• General purpose binary rewriter
  - Cross platform
    • Windows, Linux, Solaris, ...
  - Multiple architectures
    • IA-32 (x86), IA-64, SPARC, ...
  - Extensible
  - Flexible
  - Useful
Binary Rewriter Architecture

- **x86 binary**
  - x86 FE
  - x86 BE
  - x86 binary

- **SPARC bin**
  - SPARC FE
  - SPARC BE
  - SPARC bin

- **program**
  - front-end
  - new CFG
  - new CFG

- **VB app**
  - VB FE
  - VB BE
  - new program

- **Java class**
  - Java FE
  - Java BE
  - Java class

CFG transformer

Transformation parameters

[[Image of the diagram]]
Seems easy enough...

- Different architectures
- Different execution environments
- Different languages

=> Bringing all of them into one data structure is a challenge!
Seems easy...

• Each architecture has features non-existent elsewhere
  - SPARC has *register windows*
  - IA-32 (x86) has a *hardware stack*
  - ...

• Did we mention it has to be flexible, customizable, and extensible?
Binary Rewriter Design

• Binary rewriter interface — multiple levels of abstraction:
  - Hide architecture-specific differences when needed
  - Provide low level details when necessary
Binary Rewriter

• A project worth undertaking:
  - The benefits are extraordinary

• The infrastructure can be reused...
Binary Rewriter Architecture

x86 binary  SPARC bin  program  VB app  Java class

x86 FE  SPARC FE  front-end  VB FE  Java FE

CFG + other info

Static analyzer 1

program info

CFG transformer

new CFG

new program

Static analyzer 2

SPARC BE

x86 binary  SPARC bin  new program  VB app  Java class
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting
Status

• Early stage
  - Gathering requirements
    • From WiSA subprojects
    • From external sources
  - Assessing tools used by WiSA
    • Some tools to be integrated in the new infrastructure
    • Some tools have good interfaces
Status

• **Current tools**
  - **IDA Pro**
    - Supports multiple architectures
    - Can act as a front-end only
    - Front-end for x86 - good progress
  - **CodeSurfer**
    - Multiple, complex static analyses
  - **EEL**
    - Good interface
    - Support for SPARC rewriting
Status

• **CFG transformer**
  - Some transformations have ad-hoc implementations
  - Specification and design in progress

=> The work done up to now can be integrated and reused
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting
Steps Forward

1. Architecture and design
   - Documented & reviewed

2. Define interfaces
   - Based on existing tools
   - Focus on integration with existing tools

3. Prototype implemented
   - Support several key architectures
   - Test and get feedback on interface design issues
More Steps Forward

4. Review design
   - Based on internal feedback

5. Implementation
   - Complete front-ends and back-ends
   - Several transformations ported to the new infrastructure
   - Static analyses ported to the new infrastructure

6. Release
Future Work

• Support many static analyses
  - Incremental precision gains
  - Enhance infrastructure info

• Add transformations of increasing complexity

• Add more architectures / languages
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting
Seeing Through the Obfuscations

Smart Virus Scanner

Program to analyze → Annotator → Pattern Library → Annotated Program → Malicious Code Blueprint → Model Checker

IDA Pro + CodeSurfer + custom analysis code

Yes/No

Custom-built

Annotator

Smart Virus Scanner

Program to analyze

Pattern Library

Annotated Program

Malicious Code Blueprint

Model Checker

Yes/No
Detection Example

Virus Code:

```
push    eax
sidt    [esp-02h]
pop     ebx
add     ebx, HookNo * 08h + 04h
cli
mov     ebp, [ebx]
mov     bp, [ebx-04h]
lea     esi, MyHook - @1[ecx]
push    esi
mov     [ebx-04h], si
shr     esi, 16
mov     [ebx+02h], si
pop     esi
```

(from Chernobyl CIH 1.4 virus)
Detection Example

Program to be checked:

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

Annotated program:

```
mov ebp, [ebx]
```
```
Irrelevant
```
```
mov bp, [ebx-04h]
```
```
Irrelevant
```
```
Irrelevant
```
```
lea esi, MyHook - @1[ecx]
```
Detection Example

Virus Automaton:

- mov X, Y
- mov X₁, Z
- lea A, B
- Irrelevant

Program model (annotated program):

- mov ebp, [ebx]
- mov bp, [ebx-04h]
- lea esi, MyHook - @1[ecx]

X = ebp
Y = [ebx]
Z = [ebx - 04h]
A = esi
B = MyHook - @1[ecx]
Smart Virus Scanner

• What are irrelevant instructions?
  - NOPs
  - Control flow instructions that do not change the control flow
    • e.g.: jumps/branches to the next instructions
  - Instructions that modify dead registers
  - Sequences of instructions that do not modify architectural state
    • e.g.:
    
    ```
    add ebx, 1
    sub ebx, 1
    ```
Uninterpreted Symbols

• What happens when the registers are changed?

Program 1:

<table>
<thead>
<tr>
<th>mov ebp, [ebx]</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
</tr>
<tr>
<td>mov bp, [ebx-04h]</td>
</tr>
<tr>
<td>test ebx</td>
</tr>
<tr>
<td>bnez next</td>
</tr>
</tbody>
</table>
next: lea esi, MyHook - @1[ecx] |

Program 2:

<table>
<thead>
<tr>
<th>mov eax, [ecx]</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
</tr>
<tr>
<td>mov ax, [ecx-04h]</td>
</tr>
<tr>
<td>test edx</td>
</tr>
<tr>
<td>bnez next</td>
</tr>
</tbody>
</table>
next: lea ebi, MyHook - @1[ebx] |

Virus Spec:

| mov ebp, [ebx] |

=> No match with Program 2

Virus Spec with *Uninterpreted Symbols*:

| mov X, Y |

=> Matches both Programs 1 and 2
Program Obfuscations

• Semantic NOPs
• Instruction Reordering
• Variable Renaming
  – Handled through static analysis

• Encoded Program Fragment
  – Partial evaluation
General Purpose Binary Rewriting

Mihai Christodorescu
mihai@cs.wisc.edu

WiSA     http://www.cs.wisc.edu/wisa
University of Wisconsin, Madison