Frankenstein
Stitching Malware from Benign Binaries

Vishwath Mohan and Kevin W Hamlen
University of Texas at Dallas

Supported in part by AFOSR
Agenda

● (Drawbacks in) current approaches to obfuscation

● Our approach

● Implementation

● Results
The motivation

- Extend the malware obfuscation arsenal
The motivation

● The current state of detection -
  ○ Largely static and feature-based
  ○ Complex analysis only for suspicious files

● The current state of obfuscation -
  ○ Encryption
  ○ Virtualization
  ○ Metamorphism
The motivation

- The current state of detection -
  - Largely static and feature-based
  - Complex analysis only for suspicious files

- The current state of obfuscation -
  - Encryption, or "I'm going to look suspicious"
  - Virtualization
  - Metamorphism
The motivation

● The current state of detection -
  ○ Largely static and feature-based
  ○ Complex analysis only for suspicious files

● The current state of obfuscation -
  ○ Encryption, or "I'm going to look suspicious"
  ○ Virtualization, or "I'm going to carry a virtual machine around"
  ○ Metamorphism
The motivation

- The current state of detection -
  - Largely static and feature-based
  - Complex analysis only for suspicious files

- The current state of obfuscation -
  - Encryption, or "I'm going to look suspicious"
  - Virtualization, or "I'm going to carry a virtual machine around"
  - Metamorphism, or "I'm going to change my code each time (but only in simple, detectable ways)"
The motivation

- Gadgets are very interesting
  - Turing-complete functionality
  - Found everywhere
  - Classifiably benign!

- So why not use gadgets to compose your malware

**Gadget:**
A sequence of bytes representing a valid instruction sequence, and ending in return, that can be used to perform a semantically specific task.

Eg:
```
mov eax, ebx
ret
```
is a gadget that moves a value from one register (ebx) to another (eax)
The idea - Or Mary Shelley's recipe

- Consult a high-level blueprint of the human body malware
- Harvest organs gadgets from cadavers benign programs
- Stitch them together to (re)create Frankenstein the malware
Overview

Frankenstein

Gadget Discovery -> Gadget Arrangement -> Gadget Assignment

Code Generation

Semantic Blueprint

PE Template

Code Injector

Executable Synthesis

Benign Binaries

Obfuscated Copies
Malware functionality expressed as a set of logical predicates.
Semantic Blueprint

- Eg: semantic blueprint to find the slope of a line given two points \((x1,y1)\) and \((x2,y2)\)

\[
evil\_slope := \\
\text{sub}(L1, y1, y2), \quad // L1 = y1 - y2 \\
\text{sub}(L2, x1, x2), \quad // L2 = x1 - x2 \\
\text{div}(L3, L1, L2). \quad // L3 = (y1-y2) / (x1-x2)
\]
Semantic Blueprint

- Logical predicates represent actions
  - Predicates can be layered for abstraction
  - Lowest layer predicates match multiple gadgets
  - The goal is to encode *what* rather than *how*

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Semantic Definition</th>
<th>Suitable Gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>noop</td>
<td>—</td>
<td>NoOp</td>
</tr>
<tr>
<td>move($L_1, L_2$)</td>
<td>$L_1 \leftarrow L_2$</td>
<td>All Loads/Stores</td>
</tr>
<tr>
<td>add($L_1, L_2, L_3$)</td>
<td>$L_1 \leftarrow L_2 + L_3$</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>sub($L_1, L_2, L_3$)</td>
<td>$L_1 \leftarrow L_2 - L_3$</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>jump($n, Why$)</td>
<td>Jump $n$ blueprint steps if $Why$ holds</td>
<td>DirectBranch, ConditionalBranch</td>
</tr>
</tbody>
</table>
Overview

Frankenstein

Executable Synthesis

Code Generation

Gadget Discovery
Gadget Arrangement
Gadget Assignment

Semantic Blueprint

PE Template

Code Injector

Executable Synthesis

Benign Binaries

Obfuscated Copies
Overview

Builds a database of gadgets from the benign binaries.
Abstract Evaluator

- **Input**: Instruction sequence, abstract state
- **Output**: All gadget types that it can match
- **Algorithm**:

```plaintext
Algorithm 1 Gadget discovery
Input: $\sigma_0$ (initial symbolic machine state), and $[i_1, \ldots, i_n]$ (instruction sequence)
Output: $G \subseteq T \times \Phi$ (matching gadget types)

for $j = 1$ to $n$ do
    $\sigma_j \leftarrow \mathcal{E}[i_j] \sigma_{j-1}$
end for

$G \leftarrow \emptyset$

for all $t \in T$ do
    if $U(t, \sigma_n)$ is defined then
        $\phi \leftarrow U(t, \sigma_n)$
        $G \leftarrow G \cup \{(t, \phi)\}$
    end if
end for

return $G$
```
Abstract Evaluator

- **Input:** Instruction sequence, abstract state
- **Output:** All gadget types that it can match
- **Algorithm:**

A (contrived) example:

**Input:**
add eax, ebx
mov ecx, edx

**Matched gadget types:**
1) Arithmetic(eax $\leftarrow$ eax + ebx)
clobbers[ecx]

2) LoadReg(ecx $\leftarrow$ edx)
clobbers[eax]

3) Nop
clobbers[eax, ecx]

---

**Algorithm 1 Gadget discovery**

**Input:** $\sigma_0$ (initial symbolic machine state), and $[i_1, \ldots, i_n]$ (instruction sequence)

**Output:** $G \subseteq T \times \Phi$ (matching gadget types)

```
for $j = 1$ to $n$ do
    $\sigma_j \leftarrow E[i_j][\sigma_{j-1}]
end for

G \leftarrow \emptyset

for all $t \in T$ do
    if $U(t, \sigma_n)$ is defined then
        $\phi \leftarrow U(t, \sigma_n)
        G \leftarrow G \cup \{(t, \phi)\}
    end if
end for
return G
```
Overview

Benign Binaries -> Gadget Discovery -> Gadget Arrangement -> Gadget Assignment

Semantic Blueprint -> PE Template

Executable Synthesis

Code Injector

Frankenstein

Obfuscated Copies
Overview

Express blueprint as potential sequences of gadget types.

Find gadgets to satisfy each identified arrangement.

Create binaries for each assignment by injecting into a template file.

Multiple variants of Frankenstein
The implementation

- Combination of
  - Python - gadget discovery, executable synthesis
  - Prolog - gadget assignment

- Small abstract evaluator
  - Only identifies 8 non-branch instructions
  - Still more than sufficient for good results

- Not self-propagating (yet!)
The results

- **Gadget discovery**
  - Tested with files from `windows/system32`
  - Limited gadget size to 2-6 instructions

<table>
<thead>
<tr>
<th>Binary Name</th>
<th>File Size (KB)</th>
<th>Without Sliding Window</th>
<th>With Sliding Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc.exe</td>
<td>1327</td>
<td>82885</td>
<td>97163</td>
</tr>
<tr>
<td>calc.exe</td>
<td>758</td>
<td>41914</td>
<td>60390</td>
</tr>
<tr>
<td>explorer.exe</td>
<td>2555</td>
<td>89617</td>
<td>127859</td>
</tr>
<tr>
<td>cmd.exe</td>
<td>295</td>
<td>17514</td>
<td>25008</td>
</tr>
<tr>
<td>notepad.exe</td>
<td>175</td>
<td>4512</td>
<td>6974</td>
</tr>
</tbody>
</table>

- **Verdict:**
  - 46 gadgets per KB of code
  - 2300 gadgets per second
  - Encouraging!
The results

- Generated working mutants
  - Gadgets only from explorer.exe
  - Insertion Sort
  - XOR-based olgimorphism

- Found over 10,000 viable gadget assignments each!

- Size increase a little less than double on average
The results

- Compositionally benign?

- Analyzed fresh n-grams across 20 mutants
  - 2% common across 3 or more
  - 0.3% common across 5 or more
  - 0% common across 7 or more

- i.e: No defining n-grams across 35% or more
Conclusion

• New way to obfuscate malware
• Principled approach to metamorphism
• Early results show high potential for non-distinguishability from benign programs
• Definitely worth developing further
Questions?

Vishwath Mohan
(vishwath.mohan@gmail.com)
Thank You

Vishwath Mohan
(vishwath.mohan@gmail.com)
Gadgets

- Less constrained notion of a gadget
- More varied (but less than micro-gadgets?)

<table>
<thead>
<tr>
<th>Gadget Type (t)</th>
<th>Input (ℓ)</th>
<th>Parameters (p)</th>
<th>Semantic Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoOp</td>
<td>—</td>
<td>—</td>
<td>No change to memory or registers</td>
</tr>
<tr>
<td>DirectBranch</td>
<td>Offset</td>
<td>—</td>
<td>EIP ← EIP + Offset</td>
</tr>
<tr>
<td>DirectConditionalBranch</td>
<td>Offset</td>
<td>△_comp, Reg₁, Reg₂</td>
<td>EIP ← EIP + Offset if Reg₁ △_comp Reg₂</td>
</tr>
<tr>
<td>LoadReg</td>
<td>OutReg, InReg</td>
<td>—</td>
<td>OutReg ← InReg</td>
</tr>
<tr>
<td>LoadConst</td>
<td>OutReg, Value</td>
<td>—</td>
<td>OutReg ← Value</td>
</tr>
<tr>
<td>LoadMemAddr</td>
<td>OutReg, Addr</td>
<td>—</td>
<td>OutReg ← [Addr]</td>
</tr>
<tr>
<td>LoadMemReg</td>
<td>OutReg, AddrReg</td>
<td>Scale, Disp</td>
<td>OutReg ← [AddrReg * Scale + Disp]</td>
</tr>
<tr>
<td>StoreMemAddr</td>
<td>InReg, Addr</td>
<td>—</td>
<td>[Addr] ← InReg</td>
</tr>
<tr>
<td>StoreMemReg</td>
<td>InReg, AddrReg</td>
<td>Scale, Disp</td>
<td>[AddrReg * Scale + Disp] ← InReg</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>OutReg, InReg₁, InReg₂</td>
<td>△₀op</td>
<td>OutReg ← InReg₁ △₀op InReg₂</td>
</tr>
</tbody>
</table>
Gadgets

- Less constrained notion of a gadget

- More varied (but less than micro-gadgets?)

<table>
<thead>
<tr>
<th>Gadget Type ( (t) )</th>
<th>Input ( (\ell) )</th>
<th>Parameters ( (p) )</th>
<th>Semantic Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoOp</td>
<td>—</td>
<td>—</td>
<td>No change to memory or registers</td>
</tr>
<tr>
<td>DirectBranch</td>
<td>Offset</td>
<td>—</td>
<td>( EIP \leftarrow EIP + \text{Offset} )</td>
</tr>
<tr>
<td>DirectConditionalBranch</td>
<td>Offset</td>
<td>( \bowtie_{\text{cmp}} \text{, Reg}_1, \text{Reg}_2 )</td>
<td>( EIP \leftarrow EIP + \text{Offset} \text{ if } \text{Reg}<em>1 \bowtie</em>{\text{cmp}} \text{Reg}_2 )</td>
</tr>
<tr>
<td>LoadReg</td>
<td>OutReg, InReg</td>
<td>—</td>
<td>OutReg \leftarrow \text{InReg}</td>
</tr>
<tr>
<td>LoadConst</td>
<td>OutReg, Value</td>
<td>—</td>
<td>OutReg \leftarrow \text{Value}</td>
</tr>
<tr>
<td>LoadMemAddr</td>
<td>OutReg, Addr</td>
<td>—</td>
<td>OutReg \leftarrow [\text{Addr}]</td>
</tr>
<tr>
<td>LoadMemReg</td>
<td>OutReg, AddrReg</td>
<td>Scale, Disp</td>
<td>OutReg \leftarrow [\text{AddrReg} \ast \text{Scale} + \text{Disp}]</td>
</tr>
<tr>
<td>StoreMemAddr</td>
<td>InReg, Addr</td>
<td>—</td>
<td>[\text{Addr}] \leftarrow \text{InReg}</td>
</tr>
<tr>
<td>StoreMemReg</td>
<td>InReg, AddrReg</td>
<td>Scale, Disp</td>
<td>[\text{AddrReg} \ast \text{Scale} + \text{Disp}] \leftarrow \text{InReg}</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>OutReg, InReg_1, InReg_2</td>
<td>( \diamond_{\text{aop}} )</td>
<td>OutReg \leftarrow \text{InReg}<em>1 \diamond</em>{\text{aop}} \text{InReg}_2</td>
</tr>
</tbody>
</table>