Static Analysis of Virtualization-Obfuscated Binaries

Johannes Kinder

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Virtualization Obfuscation

• Obfuscation
  - Make code hard to understand for humans and tools
  - Popular for protecting benign and malicious code

• Virtualization Obfuscation
  - Hide code inside self-contained VM
  - Considered one of the strongest obfuscation schemes
  - Commercial tools (CodeVirtualizer, VMProtect) and research obfuscator
Static Analysis vs. Virtualization

• Static Analysis
  • Computes program invariants
  • Safety proofs, bug finding, malware detection

What makes static analysis of virtualization-obfuscated code fundamentally hard?

Can we still make it work and use it for deobfuscation?
void foo (int x) {
    int y = 10;
    y++;
    y++;
    if (x > 0) {
        y++;
    }
    else {}
    apiCall(y);
}
Virtualization Obfuscation

```c
void foo (int x) {
    int y = 10;
    y++;  // code: 52, 01, 02, 03, 01, 03, 01, 08, 00, 03, 03, 01, 18, 01, 00
    y++;  // data: 00, 00, 10, 05
    if (x > 0) {
        y++;  // conditional jump distance
    }
    else {}  // x
    apiCall(y);  // y
}
```
```c
int vpc = 0, op1, op2;
while (true) {
    switch(code[vpc]) {
        case 03: // increment
            op1 = code[vpc + 1];
            data[op1]++;
            vpc += 2;
            break;
        case 08: // conditional jump
            op1 = code[vpc + 1];
            op2 = code[vpc + 2];
            if (data[op1] <= 0)
                vpc += data[op2];
            else
                vpc += 3;
            break;
        case 18: // call function
            op1 = code[vpc + 1];
            apiCall(data[op1]);
            vpc += 2;
            break;
        case 52: // assignment
            op1 = code[vpc + 1];
            op2 = code[vpc + 2];
            data[op1] = data[op2];
            vpc += 3;
            break;
        default: // halt
            return;
    }
} // end switch
} // end while
```

```
code = { 52, 01, 02, 03, 01, 03, 01, 08, 00, 03, 03, 01, 18, 01, 00 }
data = { 00, 00, 10, 05 }
```

conditional jump distance
Control Flow Graphs

Original CFG

Obfuscated CFG
code[vpc]==03
op1 = code[vpc+1]
data[op1]++
vpc += 2

code[vpc]==08
op1 = code[vpc+1]

data[op1]<=0

vpc += 2

code[vpc]==18
op1 = code[vpc+1]

data[op1]>0

vpc += 3

default

apiCall(data[op1])

vpc += 2

op1 = code[vpc+1]

vpc += 2

vpc += data[op2]

vpc += 3

vpc += 3
code = {52, 01, 02, 03, 01, 03, 01, 08, 00, 03, 03, 01, 18, 01, 00}
data = {00, 00, 10, 05}

data[01]++

vpc = 0

op1 = code[vpc+1]

vpc += 2

vpc ∈ [−∞; ∞]
d_y ∈ [0; 0]

vpc ∈ [0; 7]  d_c ∈ [10; ∞]
d_y ∈ [0; ∞]  d_j ∈ [5; ∞]

vpc ∈ [3; 5]  d_c ∈ [10; 11]
d_y ∈ [0; 12]  d_j ∈ [5; 6]

op1 ∈ [1; 3]

vpc ∈ [3; 5]  d_c ∈ [10; 12]
d_y ∈ [0; 13]  d_j ∈ [5; 7]

vpc ∈ [5; 7]  d_c ∈ [10; 12]
d_y ∈ [0; 13]  d_j ∈ [5; 7]

Upper bounds grow to infinity

Weak update

Constant imprecise

Jump distance imprecise
• 1 interpreter case = many original locations
• Interpreter loop head shared among all

Original CFG

Obfuscated CFG
Domain Flattening

Location Sensitive Analysis → Location Insensitive Analysis
VPC Lifting

• Virtualization flattens one dimension of location
• Idea: track VPC and use as additional dimension
  • Separate states with differing VPC values
    \[
    \text{int } y = 10; \quad y++; \\
    \begin{align*}
    vpc &\in [0; 0] \\
    dy &\in [0; 0]
    \end{align*} \quad \begin{align*}
    vpc &\in [3; 3] \\
    dy &\in [10; 10]
    \end{align*}
  \]

• Join states with equal VPC values
  \[
  \begin{align*}
  \text{if } (...) &\{ \} \quad \text{else } \{ \} \\
  vpc &\in [12; 12] \\
  dy &\in [12; 13]
  \end{align*} \quad \begin{align*}
  vpc &\in [12; 12] \\
  dy &\in [13; 13]
  \end{align*}
  \]
\[
y++
\]
\[
vpc \in [3; 3]
\]
\[
y++
\]
\[
[5; 5]
\]
\[
\{ y++ \}
\]
\[
[10; 10]
\]
\[
\text{apiCall}(y)
\]
\[
[12; 12]
\]

\[\text{vpc} = 0\]

\[\text{vpc} \in [3; 3]\]

\[\text{code}[\text{vpc}] = 03\]

\[\text{vpc} \in [5; 5]\]

\[\text{vpc} \in [10; 10]\]

\[\text{vpc} \in (-\infty; \infty)\]

\[\text{vpc} \in [12; 12]\]

\[\text{vpc} \in [12; 13]\]

\[\text{op1} = \text{code}[\text{vpc} + 1]\]

\[\text{vpc} \in [5; 5]\]

\[\text{vpc} \in [10; 10]\]

\[\text{vpc} \in [12; 12]\]

\[\text{vpc} \in [12; 13]\]

\[\text{data}[\text{op1}]++\]

\[\text{vpc} \in [10; 10]\]

\[\text{vpc} \in [12; 12]\]

\[\text{vpc} \in [12; 13]\]

\[\text{vpc} \in [13; 13]\]

\[\text{vpc} \in [13; 13]\]

\[\text{vpc} \in [13; 13]\]
VPC Values as Locations

• Location-sensitive analysis over domain $A$ has domain

$$L \rightarrow A$$

• VPC-sensitive analysis over domain $A$, for VPC domain $V$, has domain

$$L \rightarrow V \rightarrow A$$

“VPC Location”
Reconstructing CFGs

• A program CFG is the set of all feasible transitions between \textit{locations}

\[ CFG = \left\{ (l, l') \mid (l, a) \mapsto (l', a'), l, l' \in L, a, a' \in A \right\} \]

Abstract transition relation

• A \textit{VPC-CFG} is the set of all feasible transitions between \textit{VPC locations}

\[ CFG = \left\{ ((l, v), (l', v')) \mid (l, v, a) \mapsto (l', v', a'), \quad l, l' \in L, a, a' \in A, v, v' \in V \right\} \]
Each \( (pc,vpc) \) has unique pair

- \( pc = 0 \)
- \( vpc = 0 \)
- \( vpc += 2 \)
- \( vpc += 3 \)
- \( vpc += 3 \)
- \( vpc += 3 \)
- \( vpc += 3 \)
- \( vpc += 3 \)

- \( op1 = code[vpc+1] \)
- \( data[op1]++ \)
- \( data[op1]<=0 \)
- \( data[op1]>0 \)
- \( apiCall(data[op1]) \)
- \( data[op1]=data[op2] \)
switch(code[vpc]) {
  case 03:
    op1 = code[vpc + 1];
    data[op1]++;  
    vpc += 2;
    break;
}

- Constant propagation
- Dead code elimination
- Jump threading

Each node has unique pair \((pc, vpc)\)

Original CFG

\(y = 10\)
\(y++\)
\(y++\)
\(x \leq 0\)
\(x > 0\)
\(y++\)
\(\text{apiCall}(y)\)
Implementation

• Implemented in Jakstab [CAV’08]
  • Processes obfuscated binaries
  • Reconstructs CFGs in presence of indirect jumps

• Analysis
  • VPC-lifted Bounded Address Tracking [FMCAD’10]
  • Sets of memory states up to per-variable bound

\[
\left\{ \begin{array}{c}
 (x = 5, y = 20) \\
 (x = 6, y = 20) \\
 (x = 7, y = 24)
\end{array} \right\} \cup \left\{ \begin{array}{c}
 (x = 8, y = 24) \\
 (x = 9, y = 28)
\end{array} \right\}
\]

\[k = 3\]
VPC Discovery

• Intuition: VPC changes frequently
  • *Each instruction will have a separate VPC value*
  • *VPC is the busiest variable of the interpreter*

• Detect VPC on the fly
  • *The variable that hits the value bound per variable first is promoted to VPC*
  • *Heuristic – not guaranteed to identify it correctly*
Preliminary Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>Similarity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>tamperproof guard</td>
<td>0%</td>
<td>81%</td>
<td>13s</td>
</tr>
<tr>
<td>search tree</td>
<td>0%</td>
<td>100%</td>
<td>312s</td>
</tr>
<tr>
<td>matrix multiply</td>
<td>0%</td>
<td>100%</td>
<td>311s</td>
</tr>
<tr>
<td>stuxnet</td>
<td>0%</td>
<td>87%</td>
<td>319s</td>
</tr>
</tbody>
</table>

• Targets
  • *Created with research obfuscator (C. Collberg)*

• Similarity
  • *Graph edit distance using basic block markers*
  • *Baseline: similarity of obfuscated code to original*
Conclusion

• Virtualization causes domain flattening
  • Strips one level of location sensitivity

• VPC-lifting reintroduces location sensitivity

• CFG reconstruction by tracing VPC values

• Ongoing work
  • Improve VPC discovery
  • Apply to real world obfuscators

http://www.jakstab.org