VOT4CS: A Virtualization Obfuscation Tool for C#

Sebastian Banescu, Ciprian Lucaci, Benjamin Kraemer, Alexander Pretschner

Technical University of Munich, Germany

2nd International Workshop on Software Protection
28 Oct 2016 – Vienna, Austria
Introduction

Problems for programs written in C#:
- Intellectual property (IP) theft
- Code lifting attacks

Solution:
- Use multiple obfuscation to raise the bar against attackers
  - Problem: No free obfuscation tool with virtualization obfuscation as a feature

Contributions:
- Design and implementation of virtualization obfuscator for C# programs
- An open source alternative to commercial obfuscators
- Survey and implementation of attacks
  - Survey of popular attacks against virtualization obfuscation
  - Implemented automated dynamic analysis attack
- Evaluation based on a case-study
  - Performance
  - Security (resilience against implemented attack)
Overview of Virtualization Obfuscation

**Input:** Program P (C# source code)
1. Generate a random new language L
2. Translate P to the new language
3. Synthesize an interpreter for interpreting L

**Output:** Obfuscated program P’ (C# source code) with same functionality as P
VOT4CS consists of:

- Refactoring Phase (bring code in canonical form)
- Virtualization Phase (code translation and program generation)
Refactoring Phase

1. Refactoring if-statements and switch statements

\[
\text{if } (a > b) \quad \rightarrow \quad \text{cond} = a > b; \\
\text{if } (\text{cond})
\]

2. Refactoring loops: same as if-statements + jump back if \text{cond} is TRUE

3. Refactoring unary and binary operators

\[
a += b; \quad \rightarrow \quad a = a + b;
\]

4. Refactoring statements with multiple operands (tunable)

\[
a = b + c + d; \quad \rightarrow \quad \text{tmp} = b + c; \\
a = \text{tmp} + d;
\]

5. Refactoring statements with multiple method invocations (tunable)

\[
a.b().c() \quad \rightarrow \quad x = a.b(); \\
x.c();
\]
Virtualization Phase

1. Map all data items to **data** array
   - variables (uninitialized → initialize w. random value)
   - constants (shared)
   - method parameters

2. Map all instructions to **code** array
   - generate new random ISA
   - translate each statement in code to new ISA
   - inject random values in code array between: instructions, opcodes & operands

3. Create interpreter

```java
void obfuscated_method() {
    object[] data = ... //variables, constants
    int[] code = ... //bytecode
    int vpc = 0; //virtual program counter
    while (true) { //interpreter
        switch (code[vpc]) {
            case 1023: // assignment opcode
                data[code[vpc + 2]] = data[code[vpc + 3]];}
```
Raising the Bar for Attackers

• Software diversification options
  – un-initialized variables are initialized with random values
  – random junk inserted in code array
  – order of opcode and operands
  – size of each instruction

• Most frequent opcode is assigned to default branch of switch
  – opcode value in code array is replaced with random (non-opcode) values
  – harder to identify this instruction in the code array

• Interpreter level
  – method level
  – class level (multiple methods share the same interpreter)
MATE Attacks on Virtualization Obfuscation

<table>
<thead>
<tr>
<th>Authors</th>
<th>Attack Type</th>
<th>Attacker Goal</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [Rolles2009]</td>
<td>manual static analysis</td>
<td>extract original code</td>
<td>time consuming, not scalable</td>
</tr>
<tr>
<td>2 [Sharif2009]</td>
<td>automated static &amp; dynamic analysis</td>
<td>control flow graph</td>
<td>strong assumptions on interpreter structure</td>
</tr>
<tr>
<td>3 [Coogan2011]</td>
<td>automated dynamic analysis</td>
<td>approximation of original code</td>
<td>difficult to process large traces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>significant trace</td>
<td></td>
</tr>
<tr>
<td>4 [Kinder2012]</td>
<td>automated static analysis</td>
<td>approximated data values</td>
<td>strong assumptions on interpreter structure</td>
</tr>
<tr>
<td>5 [Yadegari2015]</td>
<td>automated dynamic analysis</td>
<td>control flow graph</td>
<td>large input space leads to many traces</td>
</tr>
</tbody>
</table>
Implemented Attack

Assumption: attacker fully aware of VOT4CS design & implementation

1. Trace the program at CIL level
   - implemented CIL traced by instrumenting .NET assembly
   - logs value of current opcode and VPC value

2. Simplify the trace
   - filter out instructions belonging to switch or if-else-if statements
   - filter out instructions that increment the VPC
   - replace instructions accessing data array, with variable accesses
     (new variable names based on index in data array)
Attack Example

**Obfuscated CIL**

```
96#_IL_01BD# ldarg data
97#_IL_01C1# ldarg code
98#_IL_01C5# ldarg vpc
99#_IL_01C9# ldc.i4 -17
100#_IL_01CE# add
101#_IL_01CF# ldelem.i4
102#_IL_01D0# ldarg data
103#_IL_01D4# ldarg code
104#_IL_01D8# ldarg vpc
105#_IL_01DC# ldc.i4 14
106#_IL_01E1# add
107#_IL_01E2# ldelem.i4
108#_IL_01E3# ldelem.i4
109#_IL_01E4# castclass System.String
110#_IL_01E9# ldarg data
111#_IL_01ED# ldarg code
112#_IL_01F1# ldarg vpc
113#_IL_01F5# ldc.i4 3
114#_IL_01FA# add
115#_IL_01FB# ldelem.i4
116#_IL_01FC# ldelem.ref
117#_IL_01FD# unbox.any System.Int32
118#_IL_0202# box System.Int32
119#_IL_0207# call System.String
System.String::Concat(System.Object,System.Object)
120#_IL_020C# stelem.ref
```

**Original C#**

```
string sum = "" + 3 + 4 + "";
```

**Simplified CIL**

```
1 ldloc var 3410
2 ldloc var 3245
3 call System.String System.String::Concat(System.Object,System.Object)
4 stloc var 2165
```
Evaluation: Resilience Against Attack

Original Code:

```csharp
private string f(int b) {
    string sum = "" + 3 + 4 + "";
    string r = "";
    string[] dst = new string[b];
    for (int i = 0; i < b; i++) { // b iterations
        sum += "_" + i + "_";
        sum += "~";
        r += sum + "#";
        var p1 = car.GetEngine().GetPistons().First().GetSize();
        r += "[" + p1 + "]";
        sum += r.Length;
        dst[i] = sum;
    }
    sum += "#" + dst.Length;
    return sum;
}
```
Evaluation: Resilience Against Attack

1. Obfuscated toy program with various configurations of VOT4CS
2. Recorded traces using our CIL tracer tool
3. Simplified traces

<table>
<thead>
<tr>
<th>VOT4CS settings</th>
<th>Original</th>
<th>Refactored-only</th>
<th>Refactored &amp; Virtualized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>op2 in1</td>
<td>op3 in1</td>
</tr>
<tr>
<td>Recorded trace</td>
<td>5</td>
<td>289</td>
<td>284</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>1149</td>
<td>1104</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>5449</td>
<td>5204</td>
</tr>
<tr>
<td>Simplified trace</td>
<td>5</td>
<td>280</td>
<td>275</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>1120</td>
<td>1075</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>5320</td>
<td>5075</td>
</tr>
</tbody>
</table>

Observation: Some simplified traces shorter than original → missing instructions
Levenshtein Distance

\[
\text{lev}_{a,b}(i, j) = \begin{cases} 
\max(i, j) & \text{if } \min(i, j) = 0, \\
\min \begin{cases} 
\text{lev}_{a,b}(i - 1, j) + 1 \\
\text{lev}_{a,b}(i, j - 1) + 1 \\
\text{lev}_{a,b}(i - 1, j - 1) + 1(a_i \neq b_j)
\end{cases}
\end{cases}
\]

Example:

\( a = "\text{potato}" \)
\( b = "\text{tomato}" \)
\( \text{lev}_{a,b}(|a|, |b|) = 2 \)

How to Compute Levenshtein Distance on Traces

• Problems comparing original & obfuscated traces:
  – Different variable, argument and constant names
  – Functions have a different number of arguments (due to refactoring)

• Solution: Abstract traces
  – Loading a variable, argument or constant considered the same
  – Storing a variable, argument or constant considered the same
  – Only function names are compared (not arguments)
Evaluation: Resilience Against Attack (#operands)
Evaluation: Resilience Against Attack (#invocations)
Evaluation: Run-time overhead (Quick Sort)
Evaluation: Run-time overhead (Binary Search)
Evaluation: File Size

- Small increase (e.g. <3%) for large programs or few functions
- Large increase (e.g. >90%) for small programs or many functions

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Obfuscated</th>
<th>Relative Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Sort</td>
<td>479,232</td>
<td>491,520</td>
<td>2.57%</td>
</tr>
<tr>
<td>Binary Search</td>
<td>483,328</td>
<td>491,520</td>
<td>1.70%</td>
</tr>
<tr>
<td>ResourceLib</td>
<td>75,264</td>
<td>145,920</td>
<td>93.87%</td>
</tr>
</tbody>
</table>
Conclusions

• Design and implementation of VOT4CS
• Implementation of CIL tracer and dynamic analysis attack
• Security evaluation of VOT4CS resilience against attack
  – Measured security using Levenshtein (edit) distance
  – Lower number of operands gives more security
  – Lower number of invocations gives more security
• Performance evaluation of VOT4CS
  – Iterative methods overall faster in VOT4CS than ConfuserEx CFO
  – Recursive methods much slower in VOT4CS than ConfuserEx CFO
• Future work
  – Add more features to VOT4CS
  – Automated equivalence checking of VOT4CS input and output
Thank you for your attention!

Questions?

Source code: https://github.com/tum-i22/vot4cs

Contact:
Sebastian Banescu
banescu@in.tum.de