BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking

Jiang Ming, Dongpeng Xu, Yufei Jiang, Dinghao Wu
This Talk is About

• A hybrid method to identify fine-grained relations between obfuscated binary code.
• Address “block-centric” limitation
• Detect “conditionally equivalent” behaviors
Security-relevant Software Analysis for Binary Code

Commercial-Off-The-Shelf

Firmware

Malware

Binary Code

010100101010100101010
010101010101010101010101
0101000100001001010101010
0100010010001001010101010
0100100100001001010101010
01010101001010010010101010
0101001010100101010101010
1001010010101001010101010
10101010101010010101010101
1110100101010101010101010
0101010101010101010101010
01010101010101010101010101010
Binary Difference Analysis (Binary Differing)

- MS12-005: Object Packager Patch Differing
Security Applications of Binary Diffing

• “1-day exploit” generation

• Malware analysis

Ransomware Family Count Surpasses 200

Source: http://www.bankinfosecurity.com/blogs/now-ransomware-goes-to-200-p-2287
Challenge: Code Obfuscation

- Packer, polymorphism, metamorphism, and code virtualization
Compare Runtime Execution Behavior

• Measure the similarities of program behavior features
  - E.g., system call sequences and system call dependency graphs
  - Limitation: many binary code differences do not reflect on the behavior change; neglect *conditionally equivalent* behaviors

```plaintext
// modify registry key
1: GetLocalTime(&systime);
2: if ( systime.Day < 20)
3: {
4:   RegOpenKeyEx(...);
5:   RegSetValueEx(...);
6:   RegCloseKey (...);
7: }
```

(a) Zbot.a  (b) Zbot.b

Conditional equivalent behaviors between Trojan-Spy.Win32.Zbot variants
Basic Block Semantics Modeling

- Represent the input-output relations as symbolic formulas
  - Constraint solver, random sampling, and hashing
- Resilient to moderate obfuscation within a basic block
  - Register swapping, instruction reordering/substitution, and junk code
- “Block-centric” limitation

STP constraint solver
Simple Theorem Prover SMT solver

z3
Constraint Solver

Input
Fox -> Hash function -> DFCD3454
The red fox runs across the ice -> Hash function -> 52ED879E
The red fox walks across the ice -> Hash function -> 46042841

Random sampling
Hashing
### Binary block 1:

- `mov edx, ecx`
- `mov ebx, 0x000A`
- `add edx, ebx`
- `mov ebx, ecx`
- `sub ecx, eax`
- `mov eax, ecx`
- `dec eax`
- `and ecx, 0`
- `jmp 0x401922`

### Binary block 2:

- `lea eax, [ebx]`
- `mov edx, 0x000A`
- `nop`
- `nop`
- `add eax, edx`
- `xchg eax, eax`
- `not ecx`
- `add ecx, ebx`
- `lea edx, [ebx]`
- `xor ebx, ebx`
- `jmp 0x401A22`

Are they semantically equivalent?
Symbolic inputs to basic block 1:
ecx_0 = symbol1;
eax_0 = symbol2;

mov edx, ecx
mov ebx, 0x000A
add edx, ebx
mov ebx, ecx
sub ecx, eax
mov eax, ecx
dec eax
and ecx, 0
jmp 0x401922

Outputs:
eax_2 = symbol1-symbol2-1;
ebx_1 = symbol1;
ecx_2 = 0x0;
edx_1 = symbol1 + 0xA

Symbolic inputs to basic block 2:
ebx_0 = symbol3;
ecx_0 = symbol4;

lea eax, [ebx]
mov edx, 0x000A
nop
nop
add eax, edx
xchg eax, eax
not ecx
add ecx, ebx
lea edx, [ebx]
xor ebx, ebx
jmp 0x401A22

Outputs:
ecx_2 = symbol3-symbol4-1;
edx_1 = symbol3;
ebx_1 = 0x0;
eax_3 = symbol4 + 0xA
Figure 3: Semantic equivalence spreads across basic blocks.
Block-centric binary diffing methods fail to match these three cases
Block-Centric Limitation (2)

• Compiler optimizations
  − loop unrolling and function inline

• Return-oriented programming obfuscation
  − The chain of ROP gadgets will result in a set of small basic blocks

• Different implementations of the same algorithm
  − More examples in Hacker’s Delight

• Control flow obfuscation
  − opaque predicates and control flow flattening

• Virtualization obfuscation
  − The decode-dispatch loop generates a sequence of basic blocks to interpret one x86 instruction.
BinSim Overview

- System Call Sliced Segments + Equivalence Checking

 syscall (arg1)

Equivalence Checking

 syscall (arg1)
BinSim Advantages

- Whether two matched system calls are *conditional equivalent*
- System call sliced segments bypass the boundary of a basic block
  - more likely to address “Block-centric” limitation
BinSim Core Method

1. System call alignment
2. Dynamic slicing and WP calculation
3. Cryptographic function detection
4. Equivalence checking
5. Cryptographic function approximate matching
BinSim Architecture

Figure 6: Schematic overview of BinSim. The output for each processing: (1) unpacked code, instruction log, memory log, and system call sequences; (2) IL traces and initial matched system call pairs; (3) weakest preconditions of system call sliced segments; (4) identified cryptographic functions.
System Call Alignment

• We rely on MalGene [CCS’15], an advanced bioinformatics-inspired approach to perform system call sequence alignment

• Remove system call noises
  - multi-tag taint analysis
  - consider the parameter semantics
  - fake dependency: xor eax, eax; NtClose(eax)
Dynamic Slicing Binary Code (1)

- Dynamic slicing on the obfuscated binaries is never a textbook problem
  - Indirect memory access: \texttt{mov ebx, [4*eax+4]}
  - Implicit control transfers: \texttt{COMVcc, SETcc, REP, LOOP, CMPXCHG}
  - Decode-dispatch loop of virtualization obfuscation

Source: SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic reverse engineering of malware emulators. (S&P’09)
Dynamic Slicing Binary Code (2)

- Our solution: split data dependencies and control dependencies tracking
  - index and value based slicing to only trace data flow
  - tracking explicit and implicit control dependencies: eflags bit
  - exception: \texttt{jecxz} jumps if register \texttt{ecx} is zero
  - remove the fake control dependencies caused by virtualization obfuscation code dispatcher: \texttt{5} $\rightarrow$ \texttt{3,163} $\rightarrow$ \texttt{109}
Handling Cryptographic Functions

- Cryptographic functions have been a known challenge to symbolic execution
- Almost no interaction with system calls
- Inspired by Caballero et al.’s work (CCS'10), we do a “stitched symbolic execution”
Table 1: Different obfuscation types and their examples.

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Intra-basic-block</td>
<td>Register swapping, junk code, instructions substitution and reorder</td>
</tr>
<tr>
<td>2  Control flow</td>
<td>Loop unrolling, opaque predicates, control flow flatten, function inline</td>
</tr>
<tr>
<td>3  ROP</td>
<td>Synthetic benchmarks collected from the reference [79]</td>
</tr>
<tr>
<td>4  Different implementations</td>
<td>BitCount (Figure 3)</td>
</tr>
<tr>
<td></td>
<td>isPowerOfTwo (Appendix Figure 12)</td>
</tr>
<tr>
<td></td>
<td>flp2 (Appendix Figure 13)</td>
</tr>
<tr>
<td>5  Covert computation[59]</td>
<td>Synthetic benchmarks</td>
</tr>
<tr>
<td>6  Single-level virtualization</td>
<td>VMProtect [69]</td>
</tr>
<tr>
<td>7  Multi-level virtualization</td>
<td>Synthetic benchmarks collected from the reference [79]</td>
</tr>
</tbody>
</table>

- Similarity scores change from right pairs to wrong pairs.
Table 3: Absolute difference values of similarity scores under different obfuscation schemes and combinations.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Obfuscation type</th>
<th>BinDiff</th>
<th>DarunGrim</th>
<th>iBinHunt</th>
<th>CoP</th>
<th>Syscall alignment</th>
<th>Feature set</th>
<th>BinSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>BullMoose</td>
<td>6</td>
<td>0.58</td>
<td>0.56</td>
<td>0.39</td>
<td>0.61</td>
<td>0.08</td>
<td>0.10</td>
<td>0.08</td>
</tr>
<tr>
<td>Clibo</td>
<td>1+6</td>
<td>0.57</td>
<td>0.64</td>
<td>0.41</td>
<td>0.62</td>
<td>0.10</td>
<td>0.12</td>
<td>0.10</td>
</tr>
<tr>
<td>Branko</td>
<td>1+2+6</td>
<td>0.63</td>
<td>0.62</td>
<td>0.35</td>
<td>0.68</td>
<td>0.10</td>
<td>0.15</td>
<td>0.12</td>
</tr>
<tr>
<td>Hunatcha</td>
<td>2</td>
<td>0.40</td>
<td>0.42</td>
<td>0.19</td>
<td>0.30</td>
<td>0.12</td>
<td>0.17</td>
<td>0.12</td>
</tr>
<tr>
<td>WormLabs</td>
<td>1</td>
<td>0.10</td>
<td>0.12</td>
<td>0.03</td>
<td>0.03</td>
<td>0.08</td>
<td>0.12</td>
<td>0.05</td>
</tr>
<tr>
<td>KeyLogger</td>
<td>2</td>
<td>0.38</td>
<td>0.39</td>
<td>0.12</td>
<td>0.26</td>
<td>0.09</td>
<td>0.15</td>
<td>0.09</td>
</tr>
<tr>
<td>Sasser</td>
<td>1+2+6</td>
<td>0.62</td>
<td>0.62</td>
<td>0.42</td>
<td>0.58</td>
<td>0.12</td>
<td>0.18</td>
<td>0.10</td>
</tr>
<tr>
<td>Mydoom</td>
<td>1+2</td>
<td>0.42</td>
<td>0.38</td>
<td>0.10</td>
<td>0.38</td>
<td>0.10</td>
<td>0.15</td>
<td>0.05</td>
</tr>
<tr>
<td>ROP</td>
<td>3</td>
<td>0.63</td>
<td>0.54</td>
<td>0.49</td>
<td>0.52</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>Different implementations</td>
<td>4</td>
<td>0.48</td>
<td>0.39</td>
<td>0.48</td>
<td>0.52</td>
<td>0.05</td>
<td>0.10</td>
<td>0.05</td>
</tr>
<tr>
<td>Covert computation</td>
<td>5</td>
<td>0.45</td>
<td>0.36</td>
<td>0.44</td>
<td>0.45</td>
<td>0.05</td>
<td>0.10</td>
<td>0.05</td>
</tr>
<tr>
<td>Multi-level virtualization</td>
<td>7</td>
<td>0.68</td>
<td>0.71</td>
<td>0.59</td>
<td>0.69</td>
<td>0.15</td>
<td>0.18</td>
<td>0.16</td>
</tr>
</tbody>
</table>

**Average**

“Right pairs” vs. “Obfuscation pairs” 0.50 0.48 0.34 0.46 0.10 0.15 0.09
“Right pairs” vs. “Wrong pairs” 0.65 0.65 0.66 0.55 0.71 0.63 0.76
Analyzing Wild Malware Variants

- Collect 1,050 active malware samples from VirusShare on February 2017, 112 families
  - Perform intra-family pairwise comparison
  - BinSim are able to find small distances among intra-family samples.
  - 11% of malware variants are "conditionally equivalent"
  - Example: A variant of CryptoWall terminate execution if infected machine’s UI languages are former Soviet Union country languages
BinSim Limitations

• BinSim only detects the similarities/differences exhibiting during execution.
  – incomplete path coverage + environment-sensitive malware

• Attack BinSim’s enhanced slicing algorithm
  – slice size explosion

• Custom cryptographic algorithm
  – evade cryptographic function approximate matching
Q & A