OptiCode: Machine Code Deobfuscation for Malware Analysis

NGUYEN Anh Quynh, COSEINC
<aquynh -at- gmail.com>

CONFidence, Krakow - Poland 2013, May 28th
Agenda

1. Obfuscation problem in malware analysis
2. Deobfuscation 1: Code optimization
3. Deobfuscation 2: Opaque predicate
4. Demo
5. Conclusions
Malware analysis

- Analyze malware to understand what it does internally
- Focus on static analysis in this presentation

Malware analysis problems

- Understanding machine code (assembly) is hard and time-consuming
- Malware always try to make the code harder to understand for analysts
  - Use a lot of obfuscation techniques to make asm code like spaghetti
  - Even have tricks to fool reverse and analysis tools
The obfuscated machine code versus code after deobfuscation
Understanding obfuscated code

- Understand the semantics of every single instruction is always hard
- No good tool available to help analysts :-(
  - Emulator does not handle code having multiple paths, and sometimes output depends on context
  - Decompiler might eliminate some context, or even impossible
  - Nothing supports 64-bit code (x86-64 platform)
Malware obfuscation techniques

Using different tricks to obfuscate machine code

- Insert dead code
- Substitute instruction with equivalent instructions
- Reorder instructions
- Combine all of above methods
### Deobfuscate spaghetti code

<table>
<thead>
<tr>
<th>Obfuscation method</th>
<th>Deobfuscation method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert dead code</td>
<td>?</td>
</tr>
<tr>
<td>Substitute with equivalent instructions</td>
<td>?</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>?</td>
</tr>
</tbody>
</table>
OptiCode solutions

Method 1: Code optimization
- Using compiler technique to "reverse" the obfuscation techniques
- Normalize code + optimize code
- Fully automatic

Method 2: Handle opaque predicate
- Furthermore remove dead code (dead branches)
- Use theorem prover (SMT solver) to decide which execution paths are infeasible
- Analyst tell OptiCode what he knows (user input required)
- Semi-automatic
Method 1: Code optimization

- Using compiler technique to "reverse" the obfuscation techniques
- Normalize machine code
  - Explicitly express the semantics of machine code
  - Require an Intermediate Representation language (IR)
- Optimize the normalized code
  - Simplify, thus deobfuscate spaghetti code
- Present optimized code as deobfuscated output
Introduction on LLVM

LLVM project

- Open source project to build compiler: http://www.llvm.org
- A set of frameworks to build compiler components
- LLVM Intermediate Representation (IR) with lots of optimization module ready to use
LLVM model

LLVM model: separate Frontend - Optimization - Backend
LLVM IR

- Independent of target architecture
- RISC-like, three addresses code
- Register-based machine, with infinite number of virtual registers
- Registers having type like high-level programming language
  - void, float, integer with arbitrary number of bits (i1, i32, i64)
- Pointers having type (to use with Load/Store)
- Support Single-Static-Assignment (SSA) by nature
- Basic blocks having single entry and single exit
- Compile from source to LLVM IR: LLVM bitcode
LLVM instructions

- 31 opcode designed to be simple, non-overlap
  - Arithmetic operations on integer and float
    - *add*, *sub*, *mul*, *div*, *rem*, ...
  - Bit-wise operations
    - *and*, *or*, *xor*, *shl*, *lshr*, *ashr*
  - Branch instructions
    - Low-level control flow is unstructured, similar to assembly
    - Branch target must be explicit :-(
    - *ret*, *br*, *switch*, ...
  - Memory access instructions: *load*, *store*
  - Others
    - *icmp*, *phi*, *select*, *call*, ...
Example of LLVM IR

C code - LLVM IR code
Optimize LLVM bitcode

- The core components of the LLVM architecture
- Optimize performed on the bitcode (LLVM Pass) with combined/selected LLVM passes
  - Optimization to collect/visualize information
  - Optimization to transform the bitcode
  - Others
- 182 passes ready to use in LLVM 3.2
Why LLVM IR?

- Good IR for normalization phase: simple + no side-effects + close to assembly language
- Only use subset of LLVM instructions
  - Ignore instructions about high-level information from source code
- Handy frameworks to process the output IR
- Possible to optimize the LLVM bytecode resulted from the step of translating machine code → LLVM IR
  - Use suitable LLVM passes to "reverse" the obfuscation
Translate machine code to LLVM IR

- Similar to building compiler frontend for "machine code language"
- Tough due to the unstructured characteristics of machine code
  - Target of indirect branches
  - Self-modified code
- From machine code, build the Control Flow Graph (CFG) consisting of basic blocks (BB)
- Translate all instructions in each BB to LLVM IR
  - Reference the ISA manual of corresponding platforms (e.x: Intel/AMD manual)
Translate x86 code to LLVM IR

Example of translating x86 code to LLVM IR

```c
and eax, ebx

%tmp = load i32* @_eax, align 4, !tbaa !1
%tmp1 = load i32* @_ebx, align 4, !tbaa !2
%tmp2 = and i32 %tmp1, %tmp
%tmp3 = icmp slt i32 %tmp2, 0
%tmp4 = icmp eq i32 %tmp2, 0
store i1 false, i1* @_AF, align 1, !tbaa !15
store i1 false, i1* @_OF, align 1, !tbaa !13
store i1 %tmp4, i1* @_ZF, align 1, !tbaa !9
store i1 false, i1* @_CF, align 1, !tbaa !10
store i32 %tmp2, i32* @_eax, align 4, !tbaa !1
store i1 %tmp3, i1* @_SF, align 1, !tbaa !11
```
Optimize LLVM bitcode

Interesting LLVM passes that can deobfuscate LLVM bitcode

- **Constant propagation (-constprop)**
  - $(x = 14; y = x + 8) \Rightarrow (x = 14; y = 22)$

- **Eliminate dead store instructions (-dse)**
  - $(y = 3; \ldots; y = x + 1) \Rightarrow (\ldots; y = x + 1)$

- **Combine instructions (-instcombine)**
  - $(y = x + 1; z = y + 2) \Rightarrow (z = x + 3)$

- **Simplify CFG (-simplifycfg)**
  - Remove isolated BB
  - Merges a BB into its predecessor if there is only one and the predecessor only has one successor
  - Merge a BB that only contains an unconditional branch
Deobfuscate techniques

<table>
<thead>
<tr>
<th>Obfuscation method</th>
<th>Deobfuscation method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert dead code</td>
<td>LLVM</td>
</tr>
<tr>
<td></td>
<td>-dse, -simplifycfg</td>
</tr>
<tr>
<td>Substitute with equivalent instructions</td>
<td>LLVM</td>
</tr>
<tr>
<td></td>
<td>-constprop, -instcombine</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>LLVM</td>
</tr>
<tr>
<td></td>
<td>-instcombine</td>
</tr>
</tbody>
</table>
Present deobfuscated code

- LLVM IR as output from optimization phase is already deobfuscated
- Need to present that as result: LLVM IR or assembly?
- Assembly seems good enough
  - Compile LLVM IR back to object file containing machine code
  - Disassemble machine code back to assembly, then present it
  - Original machine registers expressed as LLVM variables → presented as memory variable
Demo
Opaque predicate

- Logical conditions cannot be evaluated with static analysis
- Typically impact conditional jumps
- A popular obfuscated technique of malware
- Analyst can help to decide dead branches, get them removed for further deobfuscation
- Semi-automatic
```
cmp eax, 0x100
jg 0x5000
```

**Flowchart:**
- **Top:** `cmp eax, 0x100
jg 0x5000` with
  - **Branching:**
    - **True**:
      - **Diamond: False**:
      - **Next Instructions**
    - **False**:
      - **Diamond: True**
      - **Green Box:** `0x5000 block`

Handle opaque predicate

- Create (first-order) logical formula from machine code
  - Make the cuts at conditional jumps
- Combine with constraints ("knowledge") provided from outside by malware analyst
- Feed the combined formula to **Theorem Prover** to decide if desired condition is always True/False, or both
  - Always True: only follow target branch
  - Always False: only follow fall-through branch (next instructions)
  - Either True or False: follow both branches
Satisfiability Modulo Theories (SMT) solver

- Theorem prover based on decision procedure
- Work with logical formulas of different theories
- Prove the satisfiability/validity of a logical formula
- Suitable to express the behaviour of computer programs
- Can generate the model if satisfiable
Z3 SMT solver

- Tools & frameworks to build applications using Z3
  - Open source project: http://z3.codeplex.com
  - Support Linux & Windows
  - C++, Python binding
- Support BitVector theory
  - Model arithmetic & logic operations
- Support Array theory
  - Model memory access
- Support quantifier $\exists$ & $\forall$
Create logical formula

Encode arithmetic and moving data instructions

<table>
<thead>
<tr>
<th>Malware code</th>
<th>Logical formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov esi, 0x48</td>
<td>(esi == 0x48) and (edx == 0x2007)</td>
</tr>
<tr>
<td>mov edx, 0x2007</td>
<td></td>
</tr>
</tbody>
</table>

Encode control flow

<table>
<thead>
<tr>
<th>Malware code</th>
<th>Logical formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp eax, 0x32</td>
<td>(eax == 0x32 and ecx == edx) or</td>
</tr>
<tr>
<td>je $_label</td>
<td>(eax != 0x32 and esi == 0)</td>
</tr>
<tr>
<td>xor esi, esi</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>_label:</td>
<td></td>
</tr>
<tr>
<td>mov ecx, edx</td>
<td></td>
</tr>
</tbody>
</table>
Create logical formula - 2

NOTE: watch out for potential conflict in logical formula

<table>
<thead>
<tr>
<th>Malware code</th>
<th>Logical formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov esi, 0x48</td>
<td>(esi == 0x48) and (esi == 0x2007)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>mov esi, 0x2007</td>
<td></td>
</tr>
</tbody>
</table>
Create logical formula - 2

NOTE: watch out for potential conflict in logical formula

<table>
<thead>
<tr>
<th>Malware code</th>
<th>Logical formula</th>
<th>Logical formula with SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov esi, 0x48</td>
<td>(esi == 0x48) and (esi == 0x2007)</td>
<td>(esi == 0x48) and (esi1 == 0x2007)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov esi, 0x2007</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Malware code:

```assembly
mov esi, 0x48
...
mov esi, 0x2007
```
Steps to create logical formula

- Normalize machine code to LLVM IR
  - LLVM IR is simple, no overlap, no side effect (semantic explicitly)
  - By design, LLVM IR supports SSA for the step of generating logical formula
- Translate machine code to LLVM IR
- Optimize resulted LLVM bitcode
- Generate logical formula from LLVM bitcode
Generate logical formula from LLVM IR

- Wrote a LLVM pass to translate bitcode to SMT logical formula
- Go through the CFG, performing block-by-block on the LLVM bitcode
- Generate formula on instruction-by-instruction, translating each instruction to SMT formula
  - Use theory of BitVector or Array, depending on instruction
    - BitVector to model all arithmetic and logic operations
    - Array to model memory accesses
Logical formula for opaque predicate

- At conditional jump sites, evaluate EFlags status to find infeasible (dead) branches
- Example: JG depends on ZF is clear, and SF == OF
  ▶ Combine with constraint (ZF == 0 && SF == OF)
  ▶ Lastly, combine with constraint provided by malware analyst
- Feed final formula to SMT solver to see if it is always True, or False?
  ▶ Always True \( \rightarrow \) drop fall-through branch (dead code)
  ▶ Always False \( \rightarrow \) drop target branch (dead code)
  ▶ Neither \( \rightarrow \) take both branches as usual
Demo
OptiCode

- Web interface + IDA plugin
- Framework to translate x86 code to LLVM IR
- Framework to generate SMT formula from LLVM bytecode
- Support neutral disassembly engine to disassemble machine code (normalization phase)
  - BeaEngine & Distorm are supported now
- Support 32-bit and 64-bit Intel
- Use Z3 solver to process logical formulas (opaque predicate)
- Implemented in Python & C++
Limitation & future works

Limitation

- Deobfuscated code can be even more complicated
- Self-modified code
- Indirect branches in machine code
- The efficiency of SMT solver depends on the complexity of the machine code

Future works

- Solve limitations
- Support other hardware platforms: ARM (anything else?)
- Deployed as an independent toolset for malware analysts
Conclusions

- OptiCode is useful to deobfuscate machine code
  - LLVM is useful as IR to normalize and optimize code, so obfuscated code is "reversed"
  - Dead code can be removed thanks to SMT solver (opaque predicate) with the helps from malware analysts
- Will be available to public at http://opticode.coseinc.com
References

- LLVM project: http://llvm.org
- LLVM passes: http://www.llvm.org/docs/Passes.html
- Z3 project: http://z3.codeplex.com
- Weakest-precondition of unstructured programs, Mike Barnett and K. Rustan M. Leino, PASTE 2005
- The Case for Semantics-Based Methods in Reverse Engineering, Rolf Rolles, Recon 2012
# Questions and answers

**OptiCode: Machine Code Deobfuscation for Malware Analysis**

Nguyen Anh Quynh <aquynh-at-gmail.com>
Insert dead code

- Insert dead instruction
- Insert NOP semantic instructions
- Insert unreachable code
- Insert branch insn to next insn
Substitute instruction with equivalent instructions

**Original code**

- `mov esi, 0x0`
- `mov edx, 0x12340000`

**Obfuscated code**

- `mov esi, 0x1`
- `dec esi`
- `mov edx, 0x12347891`
- `xor dx, dx`
Insert dead instruction

<table>
<thead>
<tr>
<th>Original code</th>
<th>Obfuscated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>mov edx, 0x30</td>
</tr>
<tr>
<td>mov edx, 0x5555</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>mov edx, 0x5555</td>
</tr>
</tbody>
</table>
Insert NOP semantic code

Original code
mov edx, 0x2013

Obfuscated code
mov edi, edi
mov edx, 0x2013
xchg cx, cx

NGUYEN Anh Quynh, COSEINC <aquynh -at- gmail.com>
OptiCode: Machine Code Deobfuscation for Malware Analysis
**Insert unreachable code**

<table>
<thead>
<tr>
<th>Original code</th>
<th>Obfuscated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov esi, 0x0</td>
<td>mov esi, 0x0</td>
</tr>
<tr>
<td>and eax, ebx</td>
<td>jmp $._label</td>
</tr>
<tr>
<td></td>
<td>junk code ...</td>
</tr>
<tr>
<td></td>
<td>_label:</td>
</tr>
<tr>
<td></td>
<td>and eax, ebx</td>
</tr>
</tbody>
</table>
Insert branch insn to next insn

Original code
mov esi, 0x0
mov edx, 0x12340000

Obfuscated code
mov esi, 0x0
jmp $_label
_label:
mov edx, 0x12340000
Reorder instructions

Original code
mov esi, 0x1
dec esi
mov edx, 0x12347891

Obfuscated code
mov esi, 0x1
mov edx, 0x12347891
dec esi