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

```
mov eax, 0x30
mov esi, ecx
mov ebx, 0x45
add ecx, 0x78
sub ebx, 0x22
inc ecx
dec eax
mov ecx, eax
and ebx, 0x99
sub eax, 0x23
xor esi, esi
jz $_l0

_l0:
shl ecx, 1
add eax, ebx
xor edx, edx
inc ebx
jmp $_l1
nop

_l1:
shr ecx, 1
sub ecx, 0x11
inc eax
and esp, -0x10
dec eax
```

```
mov edx, 0x0
mov esi, 0x0
mov ebx, 0x2
and esp, -0x10
mov eax, 0xd
mov ecx, 0x1e
```

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

Obfuscation method	Deobfuscation method
Insert dead code	?
Substitute with equivalent instructions	?
Reorder instructions	?

OptiCode solutions

Method 1: Code optimization

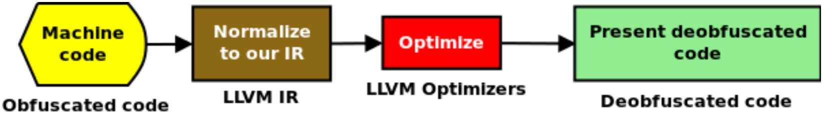
- Using compiler technique to "reverse" the obfuscation techniques
- Normalize code + optimize code
- Fully automatic

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



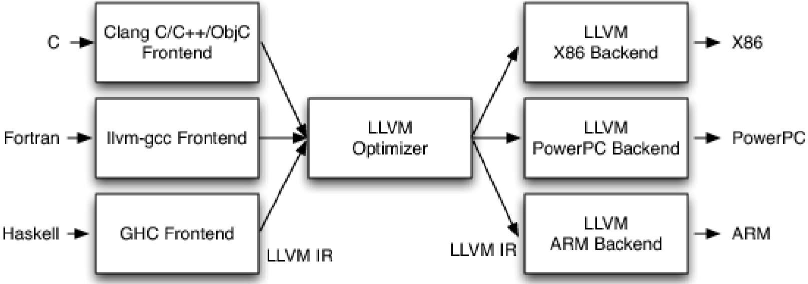
Introduction on LLVM

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

Compiler 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
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}
```

```llvm
define i32 @add2(i32 %a, i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse

recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

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 bitcode resulted from the step of translating machine code $\rightarrow$ 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

```
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
```

Example of translating x86 code to LLVM IR

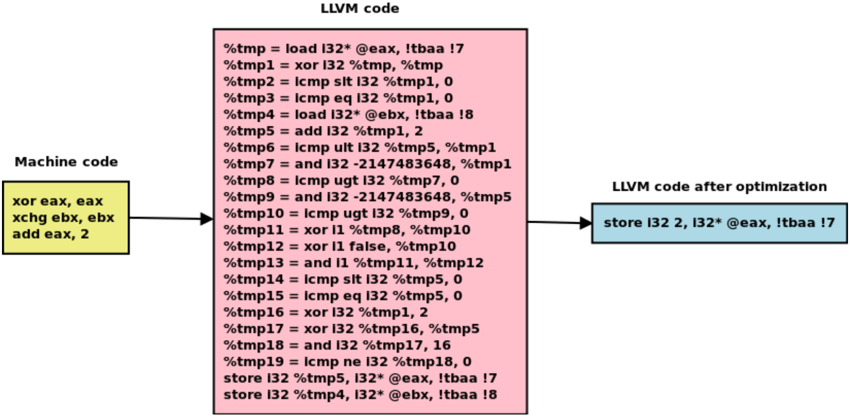
Optimize LLVM bitcode

Interesting LLVM passes that can deobfuscate LLVM bitcode

- Constant propagation (-constprop)
  - (x = 14; y = x + 8) ⇒ (x = 14; y = 22)
- Eliminate dead store instructions (-dse)
  - (y = 3; ...; y = x + 1) ⇒ (...; y = x + 1)
- Combine instructions (-instcombine)
  - (y = x + 1; z = y + 2) ⇒ (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

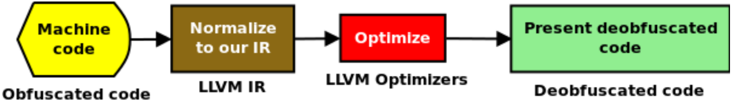
Deobfuscate techniques

Obfuscation method	Deobfuscation method
Insert dead code	LLVM -dse, -simplifycfg
Substitute with equivalent instructions	LLVM -constprop, -instcombine
Reorder instructions	LLVM -instcombine


Machine code
xor eax, eax
xchg ebx, ebx
add eax, 2
LLVM code
%tmp = load i32* @eax, !tbaa !7
%tmp1 = xor i32 %tmp, %tmp
%tmp2 = icmp slt i32 %tmp1, 0
%tmp3 = icmp eq i32 %tmp1, 0
%tmp4 = load i32* @ebx, !tbaa !8
%tmp5 = add i32 %tmp1, 2
%tmp6 = icmp ult i32 %tmp5, %tmp1
%tmp7 = and i32 -2147483648, %tmp1
%tmp8 = icmp ugt i32 %tmp7, 0
%tmp9 = and i32 -2147483648, %tmp5
%tmp10 = icmp ugt i32 %tmp9, 0
%tmp11 = xor i1 %tmp8, %tmp10
%tmp12 = xor i1 false, %tmp10
%tmp13 = and i1 %tmp11, %tmp12
%tmp14 = icmp slt i32 %tmp5, 0
%tmp15 = icmp eq i32 %tmp5, 0
%tmp16 = xor i32 %tmp1, 2
%tmp17 = xor i32 %tmp16, %tmp5
%tmp18 = and i32 %tmp17, 16
%tmp19 = icmp ne i32 %tmp18, 0
store i32 %tmp5, i32* @eax, !tbaa !7
store i32 %tmp4, i32* @ebx, !tbaa !8
LLVM code after optimization
store i32 2, i32* @eax, !tbaa !7

Present deobfuscated code

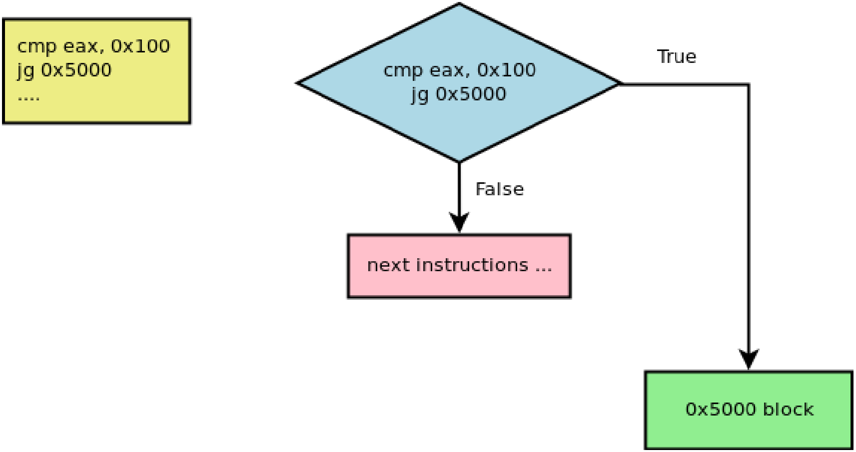
- LLVM IR as output from optimization phase is already deobfuscated
- Need to present that as result: LLVM IR or assembly?
- Assembly seems good enough
  - Compile LLVM IR back to object file containing machine code
  - Disassemble machine code back to assembly, then present it
  - Original machine registers 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
....
cmp eax, 0x100
jg 0x5000
True
False
next instructions ...
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 *Exist* (∃) & *ForAll* (∀)

Create logical formula

Encode arithmetic and moving data instructions

Malware code

```
mov esi, 0x48
mov edx, 0x2007
```

Logical formula

(esi == 0x48) and (edx == 0x2007)

Encode control flow

Malware code

```
cmp eax, 0x32
je $_label
xor esi, esi
...
_label:
mov ecx, edx
```

Logical formula

(eax == 0x32 and ecx == edx) or
(eax != 0x32 and esi == 0)

Create logical formula - 2

NOTE: watch out for potential conflict in logical formula

Malware code

```
mov esi, 0x48
...
mov esi, 0x2007
```

Logical formula

(esi == 0x48) and (esi == 0x2007)

Create logical formula - 2

NOTE: watch out for potential conflict in logical formula

Malware code

mov esi, 0x48

...

mov esi, 0x2007

Logical formula

(esi == 0x48) and (esi == 0x2007)

Malware code

mov esi, 0x48

...

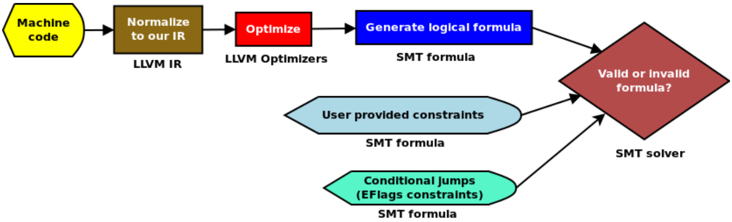
mov esi, 0x2007

Logical formula with SSA

(esi == 0x48) and (esi1 == 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

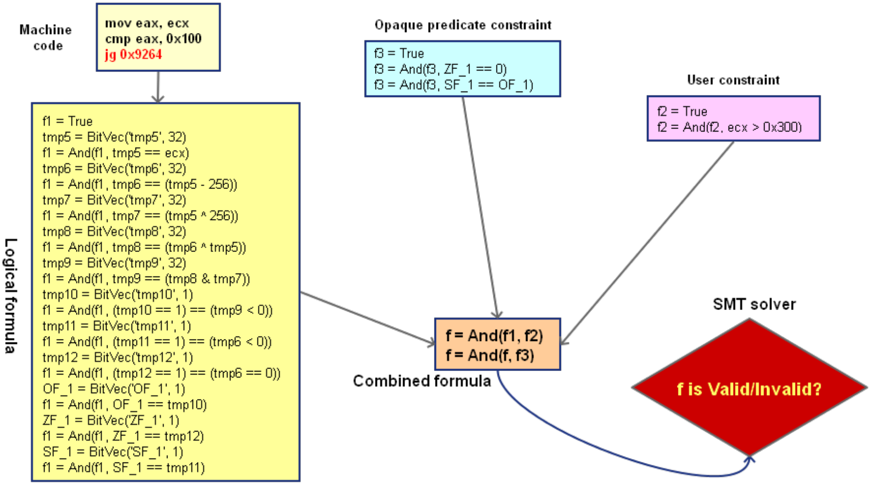


Generate logical formula from LLVM IR

- 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 infeasile ($\rightarrow$ 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


Machine code
mov eax, ecx
cmp eax, 0x100
jg 0x9264
Logical formula
f1 = True
tmp5 = BitVec('tmp5', 32)
f1 = And(f1, tmp5 == ecx)
tmp6 = BitVec('tmp6', 32)
f1 = And(f1, tmp6 == (tmp5 - 256))
tmp7 = BitVec('tmp7', 32)
f1 = And(f1, tmp7 == (tmp5 ^ 256))
tmp8 = BitVec('tmp8', 32)
f1 = And(f1, tmp8 == (tmp6 ^ tmp5))
tmp9 = BitVec('tmp9', 32)
f1 = And(f1, tmp9 == (tmp8 & tmp7))
tmp10 = BitVec('tmp10', 1)
f1 = And(f1, (tmp10 == 1) == (tmp9 < 0))
tmp11 = BitVec('tmp11', 1)
f1 = And(f1, (tmp11 == 1) == (tmp6 < 0))
tmp12 = BitVec('tmp12', 1)
f1 = And(f1, (tmp12 == 1) == (tmp6 == 0))
OF_1 = BitVec('OF_1', 1)
f1 = And(f1, OF_1 == tmp10)
ZF_1 = BitVec('ZF_1', 1)
f1 = And(f1, ZF_1 == tmp12)
SF_1 = BitVec('SF_1', 1)
f1 = And(f1, SF_1 == tmp11)
Opaque predicate constraint
f3 = True
f3 = And(f3, ZF_1 == 0)
f3 = And(f3, SF_1 == OF_1)
User constraint
f2 = True
f2 = And(f2, ecx > 0x300)
f = And(f1, f2)
f = And(f, f3)
Combined formula
SMT solver
f is Valid/Invalid?

Demo

OptiCode

- Web interface + IDA plugin
- Framework to translate x86 code to LLVM IR
- Framework to generate SMT formula from LLVM bitcode
- 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

Original code

```
...
mov edx, 0x5555
```

Obfuscated code

```
mov edx, 0x30
...
mov edx, 0x5555
```

Insert NOP semantic code

Original code

```
mov edx, 0x2013
```

Obfuscated code

```
mov edi, edi
mov edx, 0x2013
xchg cx, cx
```

Insert unreachable code

Original code

```
mov esi, 0x0
and eax, ebx
```

Obfuscated code

```
mov esi, 0x0
jmp $_label
junk code ...
_label:
and eax, ebx
```

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
```