LLVM and Code Obfuscation

Pascal Junod

UCL, Louvain-la-Neuve (B) // 26-06-2013
Context
Who?

- HEIG-VD, part of the University of Applied Sciences Western Switzerland (HES-SO)
- Pascal Junod, Julien Rinaldini, Grégory Ruch + several bachelor and master students
What?

• Everything started with a HES-SO-funded project about software protection.

• Two directions
  • Source-level protection [HEIG-VD]
  • Binary protection [EIA-FR]
How Much?

- 100 hours (professors)
- 1000 hours (research assistants)
- Until today, the effort has been at least doubled thanks to external funding.
- Forecast until March 2014: 2 man-year
When?

- First phase: started 12/2010, to 12/2012
- Second phase: started 01/2013, until at least 03/2014
Why?

● Different types of adversaries:
«Black-Box»

Adversaries

- Play according to the rules (!)
- Interact with components according to the defined APIs
- Adversaries considered in most provably-secure schemes
«Grey-Box» Adversaries

- They are looking to exploit additional «side-channel» information
- Timing
- Various types of leakage
- Faults
«White-Box» Adversaries

- Most powerful types of adversaries
- (Almost) completely master SW/HW
- Can read every memory
- Can disturb every computation at will
«White-Box» Adversaries

- Examples in real-life:
  - DRM circumventing;
  - License management system cracking;
  - Rogue SW «reverse-engineering», IP stealing.
Just think about this:

```c
if (RSA_verify (signature) == RSA_VALID_SIGNATURE) {
    // Perform some critical operation
} else {
    return NOT_AUTHENTICATED
}
```
«White-Box» Adversaries

At the assembly level:

```asm
... 
cmp $0x0, %ebx 
jbe 0x64FE89A1 
... 
```

The whole RSA signature security relies on the fact that this instruction is properly executed or not.
Classical Counter-Measures

• Use of HW as a root of trust:
  • TPM
  • smartcard
  • USB dongle
  • ...

LLVM and Code Obfuscation / UCL, Louvain-la-Neuve (B) / 26-06-2013
Classical Counter-Measures

• Problem: a hardware root of trust is not always ...
  • there;
  • sufficiently cheap;
  • flexible;
  • etc.
Software Protection

Pictures credit: Collberg, Nagra, «Surreptitious Software», Addison-Wesley, 2009
«LLVM and Code Obfuscation» / UCL, Louvain-la-Neuve (B) / 26-06-2013
Software Protection
Rogue «Reverse Engineering»
DRM
Cloud Computing
Attack scenarios

- «Reverse Engineering», IP extraction, cryptographic secrets extractions.
- Code modification
- Code distribution
Software Protection

- Render code more difficult to understand
- Render code more difficult to modify
- Render code unique
Obfuscation Techniques
Software Protection

- According to Wikipedia, obfuscated software is *source* or *machine* code that is *hard* (or *expensive*) to understand.
Software Protection

On the (Im)possibility of Obfuscating Programs*

Boaz Barak† Oded Goldreich‡ Russell Impagliazzo§ Steven Rudich∗
Amit Sahai∥ Salil Vadhan∗∗ Ke Yang††

July 29, 2010

Abstract

Informally, an obfuscator $O$ is an (efficient, probabilistic) “compiler” that takes as input a program (or circuit) $P$ and produces a new program $O(P)$ that has the same functionality as $P$ yet is “unintelligible” in some sense. Obfuscators, if they exist, would have a wide variety of cryptographic and complexity-theoretic applications, ranging from software protection to homomorphic encryption to complexity-theoretic analogues of Rice’s theorem. Most of these applications are based on an interpretation of the “unintelligibility” condition in obfuscation as meaning that $O(P)$ is a “virtual black box,” in the sense that anything one can efficiently compute given $O(P)$, one could also efficiently compute given oracle access to $P$.

In this work, we initiate a theoretical investigation of obfuscation. Our main result is that, even under very weak formalizations of the above intuition, obfuscation is impossible. We prove this by constructing a family of efficient programs $P$ that are unobfuscatable in the sense that (a) given any efficient program $P'$ that computes the same function as a program $P \in P$, the “source code” $P$ can be efficiently reconstructed, yet (b) given oracle access to a (randomly selected) program $P \in P$, no efficient algorithm can reconstruct $P$ (or even distinguish a certain bit in the code from random) except with negligible probability.

We extend our impossibility result in a number of ways, including even obfuscators that (a) are not necessarily computable in polynomial time, (b) only approximately preserve the functionality, and (c) only need to work for very restricted models of computation ($\text{TC}^0$). We also rule out several potential applications of obfuscators, by constructing “unobfuscatable” signature schemes, encryption schemes, and pseudorandom function families.
Software Protection

● Even if it is theoretically impossible, let’s still try to do it in practice :-)  

● Weaker security assumptions ?  

● The goal consists in rendering the adversary’s task more costly, even if does not result in an unpractical effort...
Software Protection

\[
\text{return (int) } (((x - 2) * (x - 3) * (x - 4) * (x - 5) * (x - 6) *
    (x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
    (x - 12) * 31) /
    ((x - 2) * (x - 3) * (x - 4) * (x - 5) * (x - 6) *
    (x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
    (x - 12) + .00001)) +
    (((x - 3) * (x - 4) * (x - 5) * (x - 6) * (x - 7) *
    (x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) *
    (28 + z)) /
    ((x - 3) * (x - 4) * (x - 5) * (x - 6) * (x - 7) *
    (x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) +
    .00001)) +
    (((x - 4) * (x - 5) * (x - 6) * (x - 7) * (x - 8) *
    (x - 9) * (x - 10) * (x - 11) * (x - 12) * 31) /
    ((x - 4) * (x - 5) * (x - 6) * (x - 7) * (x - 8) *
    (x - 9) * (x - 10) * (x - 11) * (x - 12) + .00001)) +
    (((x - 5) * (x - 6) * (x - 7) * (x - 8) * (x - 9) *
    (x - 10) * (x - 11) * (x - 12) * 30) /
    ((x - 5) * (x - 6) * (x - 7) * (x - 8) * (x - 9) *
    (x - 10) * (x - 11) * (x - 12) + .00001)) +
    (((x - 6) * (x - 7) * (x - 8) * (x - 9) * (x - 10) *
    (x - 11) * (x - 12) * 31) /
    ((x - 6) * (x - 7) * (x - 8) * (x - 9) * (x - 10) *
    (x - 11) * (x - 12) * 31) +
    (((x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
    (x - 12) * 30) /
    ((x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
    (x - 12) + .00001)) +
    (((x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) * 31) /
    ((x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) +
    .00001)) +
    (((x - 9) * (x - 10) * (x - 11) * (x - 12) * 31) /
    ((x - 9) * (x - 10) * (x - 11) * (x - 12) + .00001)) +
    (((x - 10) * (x - 11) * (x - 12) * 30) /
    ((x - 10) * (x - 11) * (x - 12) + .00001)) +
    (((x - 11) * (x - 12) * 31) /
    ((x - 11) * (x - 12) + .00001)) +
    (((x - 12) * 30) / ((x - 12) + .00001)) + 31 + .1) - y;
\]

Software Protection

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU /
 lreP rehtona tsuJ";sub p{
 @p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=
 $f=!fork;map{$P=$P[$f^ord
 ($p{$_})&6];$p{$_}=/ ^$P/ix?$P:close$_}keys%p
 p;p;p;p;p;map{$p{$_}=~/^[P.]/&&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$=_
 $d[$q];sleep rand(2)if/\S/;print
```

This Perl snippet displays «Just another Perl/Unix hacker», several characters at a time, with delays.
void primes(int cap) {
    int i, j, composite;
    for(i = 2; i < cap; ++i) {
        composite = 0;
        for(j = 2; j * j <= i; ++j)
            composite += !(i % j);
        if(!composite)
            printf("%d	", i);
    }
}

int main(void) { primes(100); }

An Eratosthenes’ sieve

Software Protection
Software Protection

- Different scenarios
  - Source vs. binary code
  - Supported languages
    - .NET, C#, Java, Javascript, C/C++, (Fortran, Ada, Haskell, Python ?)
  - Costs
    - Size and speed of the resulting code
  - Resistance to reverse engineering
More Advanced Techniques
Packing

Once used for compressing executables, «packers» are often used for software protection purposes.
Code Flattening
Opaques Predicates

• An opaque predicate is a constant boolean expression such that:

  • The developer knows its resulting value;

  • The reverse engineer is supposedly forced to dynamically study its behavior for gaining the same information.

• Force to perform a dynamic analysis («debugging», emulation, ...)
Code Interleaving

• The idea consists in mixing several independent pieces of code.
• Adding junk code;
• Merging procedures;
• Re-splitting in different threads;
• Etc.
Custom Virtualization

- Translate software in a customized and individualized «byte-code», and execute it in a custom VM.

- You can iterate the concept, but be afraid of the resulting performances!

- Use Turing-complete and exotic architectures?
  - brainfuck (8 instructions, no operands)
  - subleq (1 instruction, 3 operands)
  - ...
Add a Time Dimension

- If a bidirectional and (somewhat) reliable network is available, then one can apply ideas similar to the following:
  - At random times, a server asks the client a checksum of machine code
  - Response delay: less than one second.
  - Allows to detect software tampering.
- Frequently used by the online gaming industry to fight cheating players.
Obfuscation with LLVM
Motivations

• As a matter of fact, there does not exist open-source SW allowing to obfuscate C or C++ code in an efficient way.

• «Reverse engineering is hard, protection against RE is even harder, so let’s face the challenge»!
Abandoned Ideas

- Source parsing, and regeneration of C/C++ code.
- Tool written in Python by Sébastien Bischof, and implementing «code flattening»
Abandoned Ideas

- Major disadvantages of this idea:
  - Very difficult to write a complete and robust parser
  - You have to re-do the job for every language
Abandoned Ideas

• Use an existing «front-end», and add obfuscation capabilities to it

• Work of Grégory Ruch, based on the APIs of Clang, a C/C++ «front-end» of LLVM
Abandoned Ideas

- Major disadvantages of this idea:
  - The APIs of Clang have not really been designed to modify the generated AST.
  - Supporting only C/C++/Objective C.
LLVM

• Complete compilation infrastructure
• Open source project initiated at the University of Illinois in 2000
• Since 2005, the main sponsor is Apple Inc., which hired Chris Lattner
• Very dynamic community
• State-of-the-art software architecture
LLVM

- Front-ends:
  - C, C++, Objective C, Fortran, Ada, Haskell, Python, Ruby, ...

- Back-ends:
  - x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, Sparc, Alpha, MIPS, MSP430, SystemZ, XCore
Listing 3.1 – Simple fonction faisant une addition en C

```c
int addition(int a, int b) {
    return a + b;
}
```

Listing 3.2 – Simple fonction faisant une addition en LLVM-IR

```llvm
define i32 @addition(i32 %a, i32 %b) nounwind readnone {
    entry:
    %1 = add i32 %a, %b
    ret i32 %1
}
```
LLVM and Obfuscation

● LLVM offers a very rich API that is well documented to play with IR code.

● The primary goal was to be able to write language-independent optimization passes in an easy way.

● This approach can be used to perform obfuscation, instead of optimization.
LLVM and Obfuscation

$ clang -O3 -o prog prog.c

$ clang -O3 -B3 -o prog prog.c
Code Substitution

- First pass written, to familiarize ourselves with the LLVM API
- Replace an instruction by an equivalent expression:
  - $A \land B = (A \& \sim B) \lor (\sim A \& B)$
  - $A + B = A - (-B)$
  - $A+B = (A+R) + (B+R) - 2*R$
  - ...
Fake Branches Insertion

- Bachelor thesis of Julie Michielin

- Base ideas:
  - Insert fake branches
  - Render the flow graph irreducible
  - Use opaque predicates
LLVM and Code Obfuscation / UCL, Louvain-la-Neuve (B) / 26-06-2013
Code Flattening

- By far the most complete working pass as of today
Code Flattening

• Case of IF-THEN-ELSE:

```
entry:
%retval = alloca i32, align 4
%argsc.addr = alloca i32, align 4
%argv.addr = alloca i8**, align 8
%a = alloca i32, align 4
%b = alloca i32, align 4
store i32 0, i32 * %retval
store i32 * %argsc, i32 * %argsc.addr, align 4
store i8** %argv, i8** * %argv.addr, align 8
%0 = load i8** * %argv.addr, align 8
%arrayidxs = getelementptr inbounds i8** %0, i64 1
%1 = load i8** %arrayidxs, align 8
%call = call i32 @atoi(i8** %1) nonwind readonly
store i32 %call, i32 * %a, align 4
%2 = load i32* %a, align 4
%cmp = icmp eq i32 %2, 1
br i1 %cmp, label %if.then, label %if.else
if.then:
  %3 = load i32* %a, align 4
store i32 %3, i32* %b, align 4
br label %if.end
if.else:
  store i32 0, i32* %b, align 4
br label %if.end
if.end:
  ret i32 0
```

CFG for 'main' function

```
def 0 1 2 3
switch %switchVar, label %switchDefault |
  i32 0, label %first
  i32 1, label %if.then
  i32 2, label %if.else
  i32 3, label %if.end |
```

CFG for 'main' function

```
```

switchDefault:
  br label %loopEntry
if.then:
  %4 = load i32* %a, align 4
  store i32 %4, i32* %b, align 4
  store i32 3, i32* %switchVar
  br label %loopEnd
if.else:
  store i32 0, i32* %b, align 4
  store i32 3, i32* %switchVar
  br label %loopEnd
if.end:
  ret i32 0
```

CFG for 'main' function
Code Flattening

Case of a FOR loop:

entry:
%reval = alloca i32, align 4
%argv.addr = alloca i8 * *, align 8
%a = alloca i32, align 4
%j = alloca i32, align 4
u = alloca i32, align 4
store i32 0, i32 * 1
store i32 %argv, i32 * %argv.addr, align 4
store i8 * %argv, i8 * 1 %argv.addr, align 8
store i32 0, i32 * %a, align 4
%0 = load i8 * %argv.addr, align 8
%arrayidx = getelementptr inbounds i8 * 1, i64 1
%1 = load i8 * %arrayidx, align 8
%call = call i32 %atoi(i8 * 1) nonnull readable
store i32 0, i32 * %a, align 4
store i32 0, i32 * %a, align 4
br label %forCond

for Cond:
%2 = load i32 * %i, align 4
%3 = load i32 * %a, align 4
%cmp = icmp sle i32 %2, %3
br i1 %cmp, label %forbody, label %forEnd

for Body:
%4 = load i32 * %i, align 4
%5 = load i32 * %a, align 4
%add = add nsw i32 %4, %5
store i32 %add, i32 * %j, align 4
br label %forInc

for Inc:
%6 = load i32 * %i, align 4
%inc = add nsw i32 %6, 1
store i32 %inc, i32 * %i, align 4
br label %forEnd

switchDefault:
br label %loopEnd

for End:
ret i32 0

loopBody:
switchVar1 = load i32 * %switchVar1
switch i32 %switchVar1, label %switchDefault [ i32 0, label %forCond
i32 1, label %forBody
i32 2, label %forInc
i32 3, label %forEnd
]}

for Body:
%5 = load i32 * %i, align 4
%6 = load i32 * %a, align 4
%add = add nsw i32 %3, %5
store i32 %add, i32 * %j, align 4
store i32 0, i32 * %switchVar
br label %loopEnd

for Inc:
%8 = load i32 * %i, align 4
%9 = load i32 * %a, align 4
%add = add nsw i32 %9, %8
store i32 %add, i32 * %j, align 4
store i32 0, i32 * %switchVar
br label %loopEnd

for End:
ret i32 0

loopEnd:
br label %loopEntry

def 0 1 2 3

CFG for 'main' function

loopEntry:
switchVar = load i32 * %switchVar
switch i32 %switchVar, label %switchDefault [ i32 0, label %forCond
i32 1, label %forBody
i32 2, label %forInc
i32 3, label %forEnd
]
Code Flattening

● Case of a SWITCH:

```
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  %b = alloca i32, align 4
  %a = alloca i32, align 4
store i32 0, i32* %retval
store i32 %argc, i32* %argc.addr, align 4
store i8** %argv, i8*** %argv.addr, align 8
store i32 0, i32* %b, align 4
%0 = load i8*** %argv.addr, align 8
%arrayidx = getelementptr inbounds i8**, %0, i64 1
%1 = load i8** %arrayidx, align 8
%call = call i32 @atoi(i8* %1) nounwind readonly
store i32 %call, i32* %a, align 4
%2 = load i32% %a, align 4
switch i32 %2, label %sw.default [ i32 1, label %sw.bb ]
```

```
sw.default:
  store i32 0, i32* %b, align 4
  br label %sw.epilog

sw.bb:
  store i32 1, i32* %b, align 4
  br label %sw.epilog

sw.epilog:
  ret i32 0
```

CFG for ‘main’ function
Code Flattening

- A more complex case
Code Flattening

● A really complex case:

● One single routine involving found in ImageMagick involving 6000+ lines of C
Code Flattening

- Main encountered difficulties
  - Complexity of LLVM and of its APIs
  - SSA philosophy («single static assignment»), and its «phi nodes»
  - «Debugging» of our code
Test Procedures

- Library libtomcrypt
- Library ImageMagick
- Test suite of MySQL
  - 2’729 unitary tests OK
  - 2 tests failing on 28
- Test suite of sqlite
  - 41 tests out of 119’350 failing
Performances

- Code flattening
  - Benchmark of libtomcrypt (-O0)
    - 30% less speed
    - 15% size increase
  - --flatten + -O3 faster than -O0
TODOs

- Identify the remaining bug in our flattening pass
- Flatten the invoke
  - Used for handling the try-catch
Work in Progress

- Procedures merging pass
- Working at the x86 back-end level
- Several technique to thwart faults
- Code spaghettization
- Anti-debugging tricks insertion
- Tamper-proofing
- Buffer encryption, code packing
- Custom virtualization using the LLVM
Go Open-Source!
Thank you!
Questions welcome!