CONFUSE
LLVM-based Code Obfuscator

PLT Course Project Demo
Spring 2013

Chih-Fan Chen
Theofilos Petsios
Marios Pomonis
Adrian Tang
Outline

1. Introduction To Code Obfuscation
2. LLVM
3. Obfuscation Techniques
   1. String Transformation
   2. Junk Code
   3. Opaque Predicates
4. Testing
5. Conclusion
Code Obfuscation

- Problem
  - Compiled binaries can be easily disassembled and “reversed”
  - Readily available tools
    - Disassemblers (e.g. IDA-Pro)
    - Decompilers (e.g. Hex-Rays)

- A need to “complicate” software
  - While retaining semantics
Sample disassembly (IDA Pro Disassembler)
Motivation

- To make binary programs more resistant to reverse-engineering efforts
- Protection of intellectual property
- Malware
Code Obfuscation

- Obfuscation can occur at virtually any point:
  - Front-end (source code)
  - Optimizer (intermediate representation)
  - Back-end (assembly)
  - After compilation (machine code)

- Our approach
  - Obfuscate C source files
  - Add optimization (obfuscation) passes for the LLVM optimizer
LLVM Design

The diagram illustrates the LLVM Design architecture, showing how different frontends (C, Fortran, Ada) translate to LLVM IR, which is then optimized by a common optimizer. The optimized LLVM IR is then used by backends (X86, PowerPC, ARM) to generate code for specific architectures.
Optimizer Design

- Multiple passes executed sequentially
- Passes written in C++
- 2 Types of passes
  - Analysis
  - Transform
- Transform passes are not using each other
- They might use analysis passes though
LLVM IR

- Low level, RISC-like
- 3-Address Code (Static Single Assignment)
- Infinite number of registers
- 3 Flavors of representation
  - In memory data structure
  - Bit-code
  - Textual
declare i32 @puts (i8*)
@global_str = constant [13 x i8] c"Hello World!\00"
define i32 @main() {
    %temp = getelementptr [13 x i8]* @global_str, i64 0, i64 0
call i32 @puts(i8* %temp)
    ret i32 0
}
Obfuscation Techniques

- 3 different techniques
  - String Transformation
  - Junk Code
  - Opaque Predicates
- Implemented as different (independent) passes
- Can be used separately or in conjunction
h = hash(x);
if (h == H1) {
    /* code for H1 */
}
else if (h == H2) {
    /* code for H2 */
}
else if (h == H3) {
    /* code for H3 */
}

- What does x represent?
if (strcmp(x, "open") == 0) {
    /* code for open */
}
else if (strcmp(x, "delete") == 0) {
    /* code for delete */
}
else if (strcmp(x, "edit") == 0) {
    /* code for edit */
}
Cryptographic Hash Functions

- Small changes in input
  - Totally different digest
- Uniform distribution
- Heavily used for authentication
  - Digital signatures
  - MAC
String Transformation

- Used SHA-1 cryptographic function
  - Has $2^{-52}$ chance of collision (latest attack)
- Remove string literals from file if their sole use is in comparisons
- Supports:
  - strcmp
  - strncmp
Junk Code Insertion

- **Approach:**
  - Transform variable assignment operations to semantically equivalent instructions

- **Challenge:**
  - Need to ensure transformations do not get “optimized” away

- **Key idea:**
  - Transform using arithmetic operations
Sample transformation
- Add additional instructions
- Insert before store instructions

```
int func1(int x)
{
    x++; 
    return x;
}
```
Different methods of transformation – **Method 1**
- Preserve value of \( x \) after insertion.

1. \( \frac{(b(x + a) - ab)}{b} = \frac{(bx + ba - ab)}{b} = \frac{bx}{b} = x \)

3-address code

```plaintext
\begin{align*}
& t1 = x + a; \\
& t2 = b \times t1; \\
& t3 = t2 - c; \\
& x = t3 / b;
\end{align*}
```

Instruction

```plaintext
%temp = add i32 %inc, 5 %temp1 = mul i32 %temp, 4 %temp2 = sub i32 %temp1, 20 %temp3 = sdiv i32 %temp2, 4
```
Different methods of transformation – **Method 2**
- Preserve value of $x$ after insertion.

2. \[\frac{(ax + b) - (cx + b)}{(a - c)} = \frac{(a - c) \times x}{(a - c)} = x\]

**3-address code**

```
t1 = a * x;
t2 = t1 + b;
t3 = c * x;
t4 = t3 + b;
t5 = t2 - t4;
x = t5 / d;
```

**Instruction**
```
%temp = mul i32 %inc, 2
%temp1 = add i32 %temp, 3
%temp2 = mul i32 %inc, 1
%temp3 = add i32 %temp2, 3
%temp4 = sub i32 %temp1, %temp3
%temp5 = sdiv i32 %temp4, 1
```
Randomly apply transformation methods

- Method 1: +4 instructions
- Method 2: +6 instructions
- Values inserted are random
  - $a = 9$, $b = 8$
  - $a = 5$, $b = 7$
Junk Code Insertion

- Chain transformation together multiple times
Opaque Predicates

- Control Flow Obfuscation Method
- Adds new branch to the CFG
- Predicate is the condition of the new branch
- “Opaque” since it is known at compilation time but not easy to infer at runtime
Opaque Predicates

- How to ensure preservation of semantics?
- Put valid instructions in the taken branch, junk code or nothing in the not-taken.
Opaque Predicates

- How to ensure hardness against automatic solvers?
- Relies on mathematical identities
  - e.g. \((x^3-x) \mod 3 = 0\)
- Cyclomatic Complexity increases, causing growth in truth conditions of solvers.
Opaque Predicates

Original

Obfuscated

Confuse
Testing

- Testing Architecture
  - Regression test suite
    - Individual obfuscation
    - Combined passes
  - Integrated Makefile to automate obfuscation
    - Outputs both LLVM IR files and binaries for diff-ing
  - IDAPython Script
    - Disassembles a program and evaluates cyclomatic complexity of programs
Regression Testing

- lit – LLVM Integrated Tester
  - Tool included in LLVM framework for executing LLVM and Clang test suites
- Test files are `.c` source files instrumented with testing scripts

```c
#include <stdio.h>

void main()
{
 ...
}

// RUN: clang -emit-llvm %s-c -o -| opt -load lib.dylib -objunk > %t1
// RUN: lli %t1 > %t2
// RUN: diff %t2 %s.out
```

#include `<stdio.h>

void main () { ... }
Test Design

- String obfuscation
  - Variants of string comparison functions
  - Using strings as actual parameters or as string buffers

- Junk code obfuscation
  - for / while loops with iterations
  - Arithmetic operations

- Opaque predicate obfuscation
  - Reuse the tests above
Testing

- Evaluation of Cyclomatic Complexity

![Chart depicting the evaluation of Cyclomatic Complexity before and after obfuscation.](chart.png)

Confuse
DEMO
Environment

Bitbucket

LLVM

Vim

IDA – The Interactive Disassembler
Version 6.3.120612 (32-bit)
(c) 2012 Hex-Rays SA
Conclusions

- (we passed the comp :P)
- Very interesting project
  - Going against what this course is teaching!
- An optimization problem
  - How to “de-optimize” without being optimized
  - Deg of obfuscation vs Overhead
Questions?
Backup slides
Junk Code Insertion

- Need to ensure transformations do not get “optimized” away
Junk Code Insertion

- Challenge faced -- eliminated by LLVM optimizer
  - irrelevant code

```c
int main()
{
    int x = rand() % 3;
    printf("x = %d\n", x);
    return x;
}
```
Junk Code Insertion

- Challenge faced -- eliminated by LLVM optimizer
  - Only mul and div
  - Only add and sub

```c
int main()
{
  int x = rand() & 3;
  printf("x = %d\n", x);
  return x;
}
```