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

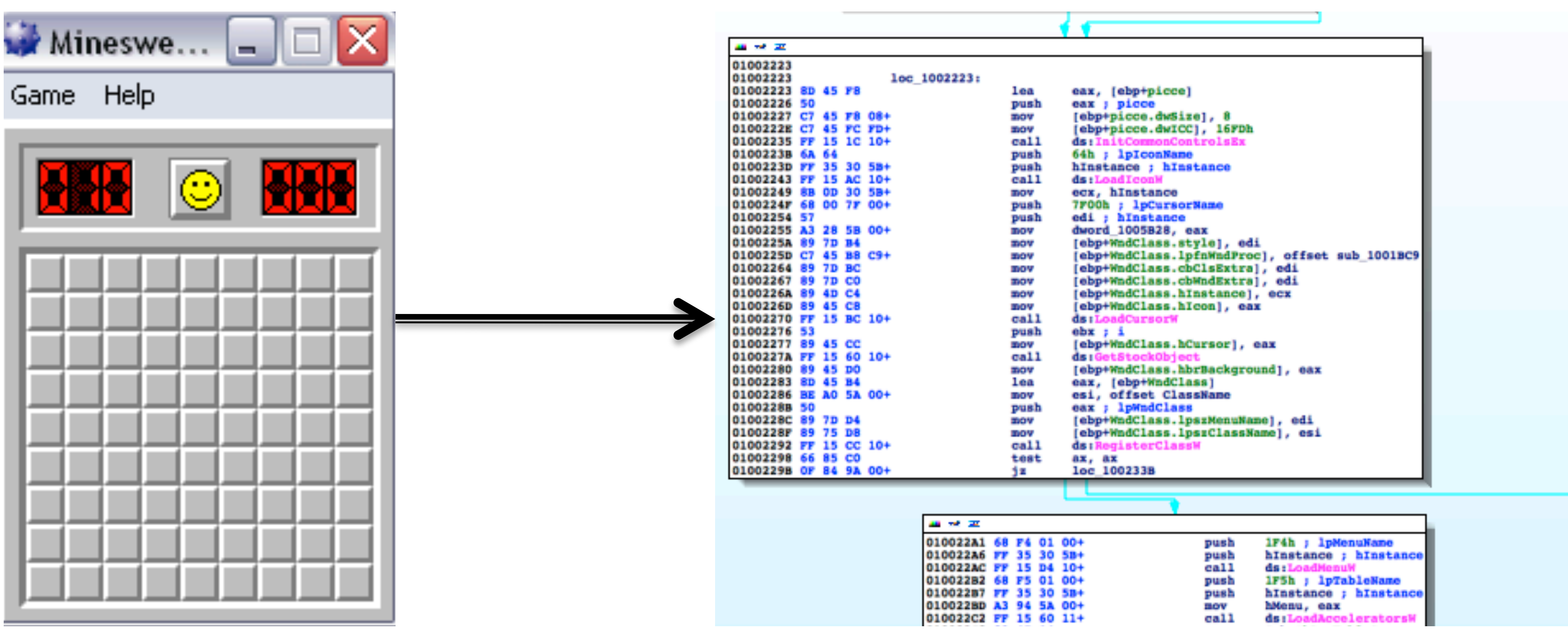
Code Obfuscation

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

    } Reverse Engineering
- A need to “complicate” software
  - While retaining semantics

Code Obfuscation

- Sample disassembly (IDA Pro Disassembler)



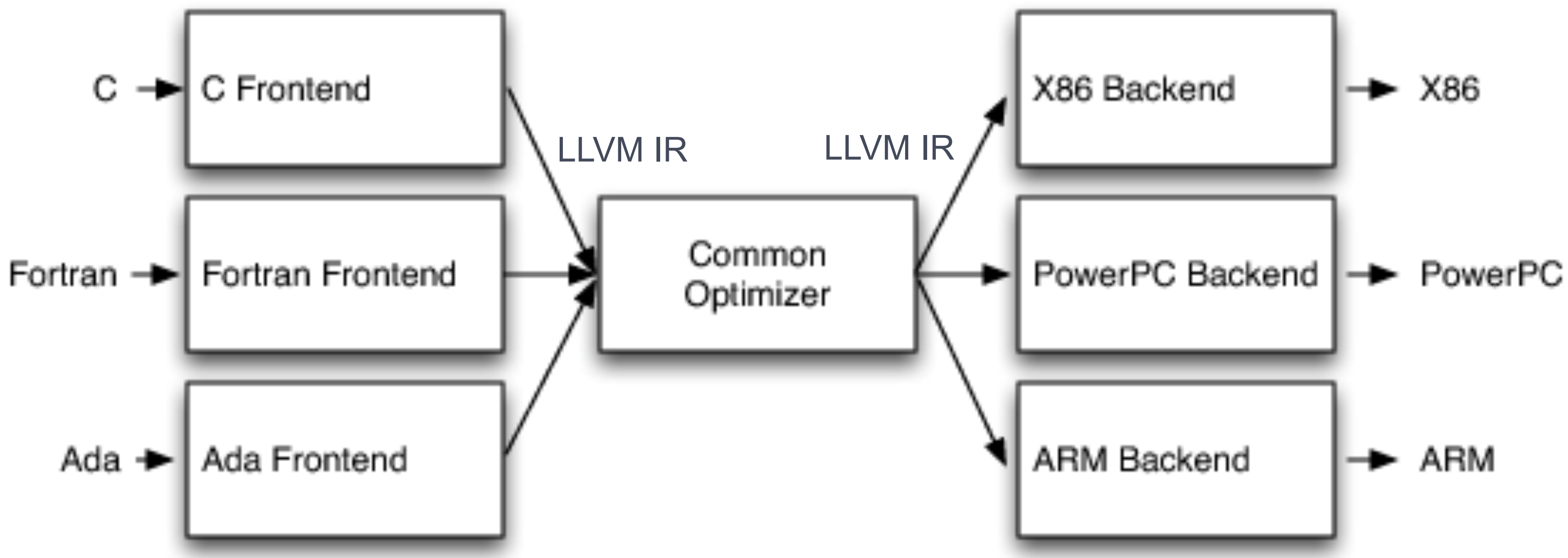
Code Obfuscation

- 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


C
C Frontend
Fortran
Fortran Frontend
Ada
Ada Frontend
LLVM IR
Common Optimizer
LLVM IR
X86 Backend
PowerPC Backend
ARM Backend
X86
PowerPC
ARM

LLVM

- 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

HelloWorld.ll

```
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

String Transformation

```
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?

String Transformation

```
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

Junk Code Insertion

- Sample transformation
  - Add additional instructions
  - Insert before store instructions

```
int func1(int x)
{
    x++;
    return x;
}
```

```
[======before insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 0, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %inc = add nsw i32 %0, 1
[instruction  6]:   store i32 %inc, i32* %x, align 4
[instruction  7]:   ret i32 0
```

```
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 0, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %inc = add nsw i32 %0, 1
[instruction  6]:   %temp = add i32 %inc, 5
[instruction  7]:   %temp1 = mul i32 %temp, 4
[instruction  8]:   %temp2 = sub i32 %temp1, 20
[instruction  9]:   %temp3 = sdiv i32 %temp2, 4
[instruction  10]:   store i32 %temp3, i32* %x, align 4
[instruction  11]:   ret i32 0
```

Junk Code Insertion

- Different methods of transformation – <u>Method 1</u>
  - Preserve value of *x* after insertion.

$$
\begin{aligned}
&1.\ (b(x+a)-ab)/b \\
&= (bx+ba-ab)/b \\
&= bx/b \\
&= x
\end{aligned}
$$

3-address code

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

Instruction

```
%temp = add i32 %inc, 5
%temp1 = mul i32 %temp, 4
%temp2 = sub i32 %temp1, 20
%temp3 = sdiv i32 %temp2, 4
```

Junk Code Insertion

- Different methods of transformation – Method 2
  - Preserve value of *x* after insertion.

$$
\begin{aligned}
2.\ & ((ax+b)-(cx+b))/(a-c) \\
&= (a-c)*x/(a-c) \\
&= x
\end{aligned}
$$

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
```

Junk Code Insertion

- Randomly apply transformation methods
  - Method 1 : +4 instructions
  - Method 2: +6 instructions

```c
int main()
{
    int x = 5;
    x = func1(x);
    x = func2(x);
    return x;
}
```

```
main
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 5, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %call = call i32 @func1(i32 %0)
```

2

```
[instruction  6]:   %temp = mul i32 %call, 8
[instruction  7]:   %temp1 = add i32 %temp, 8
[instruction  8]:   %temp2 = mul i32 %call, 1
[instruction  9]:   %temp3 = add i32 %temp2, 8
[instruction 10]:   %temp4 = sub i32 %temp1, %temp3
[instruction 11]:   %temp5 = sdiv i32 %temp4, 7
[instruction 12]:   store i32 %temp5, i32* %x, align 4
[instruction 13]:   %1 = load i32* %x, align 4
[instruction 14]:   %call1 = call i32 @func2(i32 %1)
```

1

```
[instruction 15]:   %temp6 = add i32 %call1, 9
[instruction 16]:   %temp7 = mul i32 %temp6, 2
[instruction 17]:   %temp8 = sub i32 %temp7, 18
[instruction 18]:   %temp9 = sdiv i32 %temp8, 2
[instruction 19]:   store i32 %temp9, i32* %x, align 4
[instruction 20]:   %2 = load i32* %x, align 4
[instruction 21]:   ret i32 %2
```

Junk Code Insertion

- Values inserted are random
  - a = 9, b = 8
  - a= 5, b = 7

```
main
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 5, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %call = call i32 @func1(i32 %0)
[instruction  6]:   %temp = add i32 %call, 9
[instruction  7]:   %temp1 = mul i32 %temp, 8
[instruction  8]:   %temp2 = sub i32 %temp1, 72
[instruction  9]:   %temp3 = sdiv i32 %temp2, 8
[instruction  10]:   store i32 %temp3, i32* %x, align 4
[instruction  11]:   %1 = load i32* %x, align 4
[instruction  12]:   %call1 = call i32 @func2(i32 %1)
[instruction  13]:   %temp4 = add i32 %call1, 5
[instruction  14]:   %temp5 = mul i32 %temp4, 7
[instruction  15]:   %temp6 = sub i32 %temp5, 35
[instruction  16]:   %temp7 = sdiv i32 %temp6, 7
[instruction  17]:   store i32 %temp7, i32* %x, align 4
[instruction  18]:   %2 = load i32* %x, align 4
[instruction  19]:   ret i32 %2
```

```
main
[======before insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 5, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %call = call i32 @func1(i32 %0)
[instruction  6]:   store i32 %call, i32* %x, align 4
[instruction  7]:   %1 = load i32* %x, align 4
[instruction  8]:   %call1 = call i32 @func2(i32 %1)
[instruction  9]:   store i32 %call1, i32* %x, align 4
[instruction  10]:   %2 = load i32* %x, align 4
[instruction  11]:   ret i32 %2
[Instruction]  store i32 %call, i32* %x, align 4
```

Junk Code Insertion

- Chain transformation together multiple times

```c
int func2(int x)
{
    x--;
    return x;
}
```

```
[======after insertion======]
[instruction  0]:   %x.addr = alloca i32, align 4
[instruction  1]:   store i32 %x, i32* %x.addr, align 4
[instruction  2]:   %0 = load i32* %x.addr, align 4
[instruction  3]:   %dec = add nsw i32 %0, -1
[instruction  4]:   %temp = add i32 %dec, 5
[instruction  5]:   %temp1 = mul i32 %temp, 8
[instruction  6]:   %temp2 = sub i32 %temp1, 40
[instruction  7]:   %temp3 = sdiv i32 %temp2, 8
[instruction  8]:   %temp4 = add i32 %temp3, 5
[instruction  9]:   %temp5 = mul i32 %temp4, 5
[instruction  10]:   %temp6 = sub i32 %temp5, 25
[instruction  11]:   %temp7 = sdiv i32 %temp6, 5
[instruction  12]:   store i32 %temp7, i32* %x.addr, align 4
[instruction  13]:   %1 = load i32* %x.addr, align 4
[instruction  14]:   ret i32 %1
```

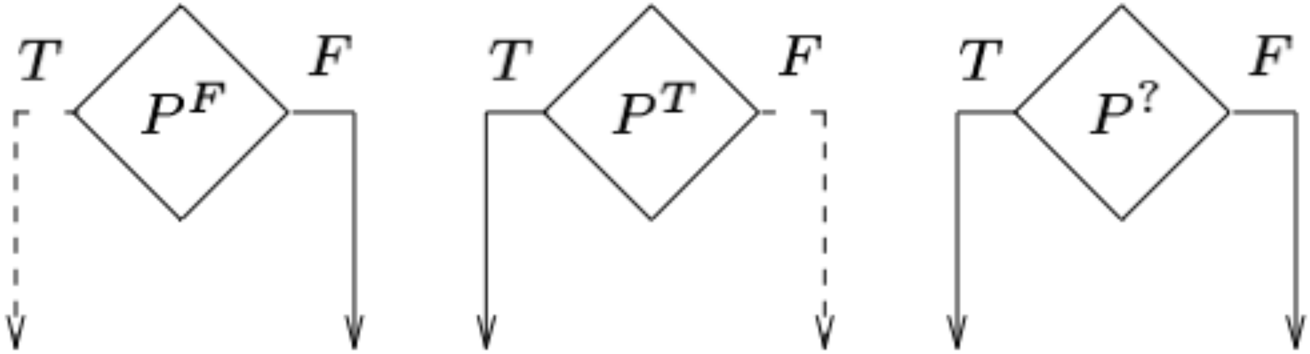
```
func2
[======before insertion======]
[instruction  0]:   %x.addr = alloca i32, align 4
[instruction  1]:   store i32 %x, i32* %x.addr, align 4
[instruction  2]:   %0 = load i32* %x.addr, align 4
[instruction  3]:   %dec = add nsw i32 %0, -1
[instruction  4]:   store i32 %dec, i32* %x.addr, align 4
[instruction  5]:   %1 = load i32* %x.addr, align 4
[instruction  6]:   ret i32 %1
[Instruction]  store i32 %dec, i32* %x.addr, align 4
```

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.

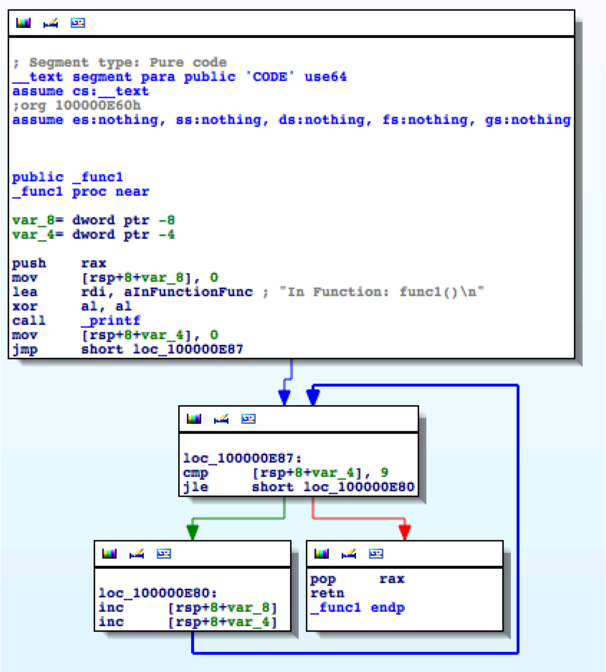


Opaque Predicates

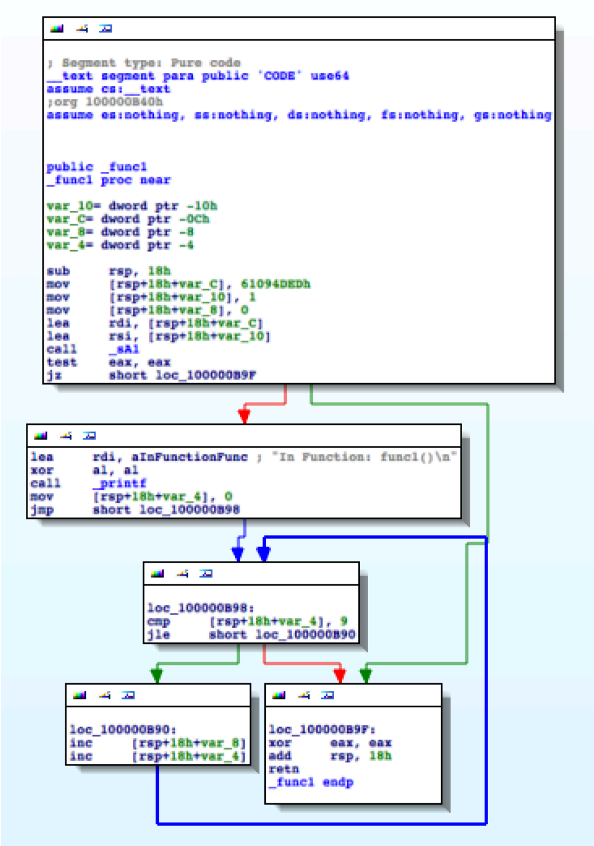
- How to ensure hardness against automatic solvers?
- Relies on mathematical identities
  - e.g. $(x^3-x) \bmod 3 = 0$
- Cyclomatic Complexity increases, causing growth in truth conditions of solvers.

Opaque Predicates

Original



Obfuscated



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

Testing

- 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

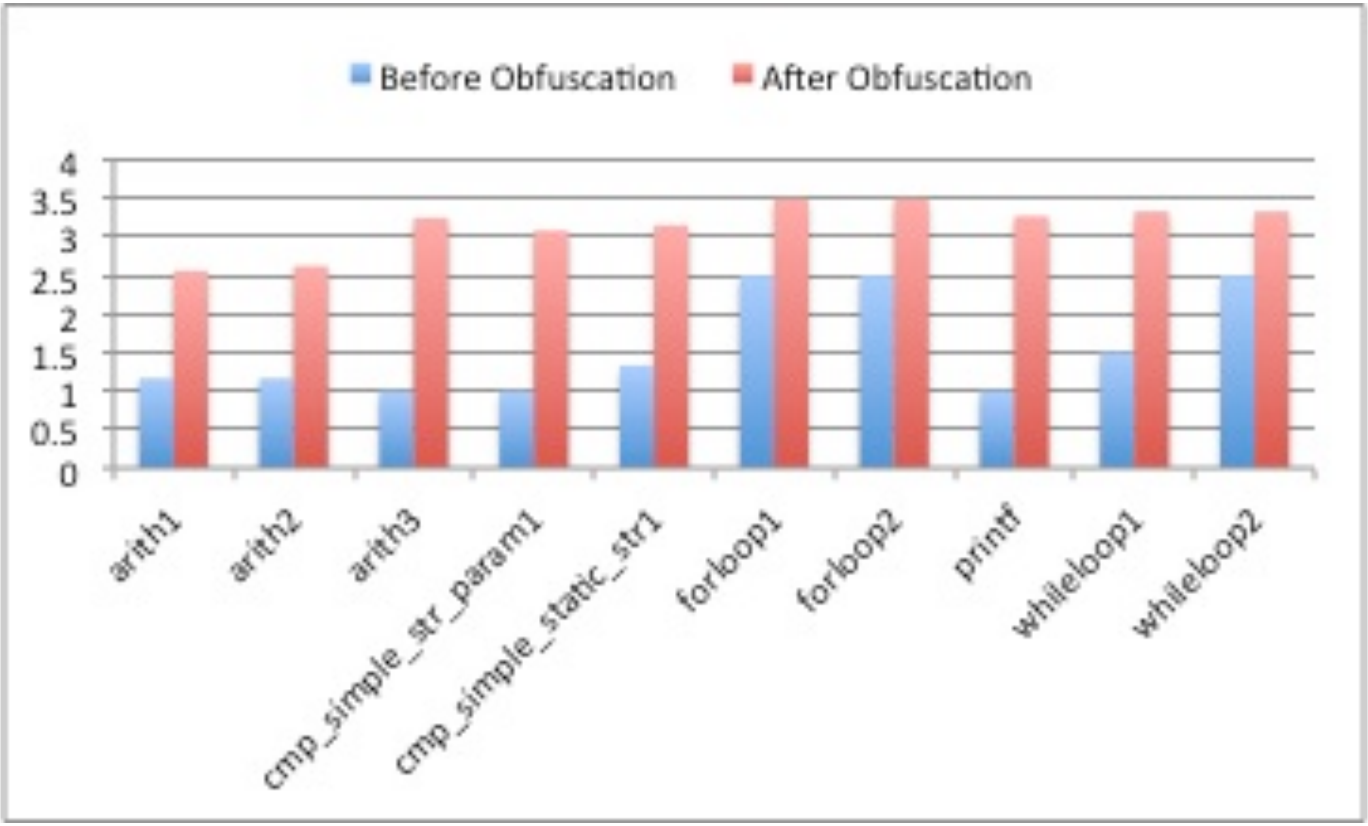
```
// RUN: clang -emit-llvm %s -c -o -| opt -load lib.dylib -objunk > %t1
// RUN: lli %t1 > %t2
// RUN: diff %t2 %s.out
#include <stdio.h>
void main () { … }
```

Testing

- 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



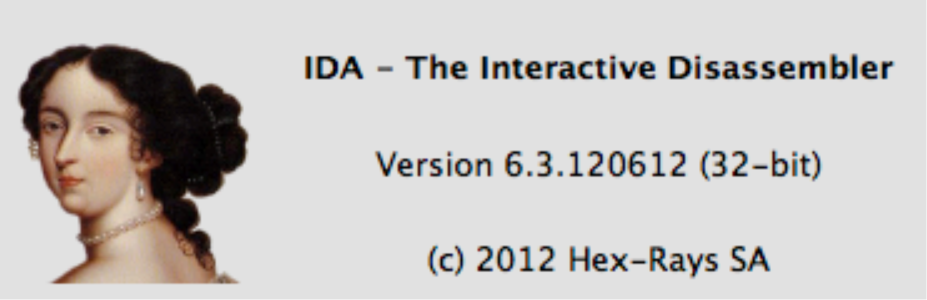
DEMO

Environment

Bitbucket

LLVM
COMPILER INFRASTRUCTURE

Vim



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

```
define i32 @main() #0 {
entry:
  %call = tail call i32 @rand() #2
  %rem = srem i32 %call, 3
  %temp = shl i32 %rem, 3
  %temp3 = sdiv i32 %temp, 8
  %call1 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([8 x i8
]* @.str, i64 0, i64 0), i32 %temp3) #2
  ret i32 %temp3
}
```

```
define i32 @main() #0 {
entry:
  %call = tail call i32 @rand() #2
  %rem = srem i32 %call, 3
  %temp4 = mul i32 %rem, 7
  %temp5 = sdiv i32 %temp4, 7
  %call1 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([8 x i8
]* @.str, i64 0, i64 0), i32 %temp5) #2
  ret i32 %temp5
}
```

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;
}
```

```
main
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   %call = call i32 @rand() #3
[instruction  4]:   %rem = srem i32 %call, 3
[instruction  5]:   %temp = mul i32 %rem, 6
[instruction  6]:   %temp1 = add i32 %temp, 8
[instruction  7]:   %temp2 = mul i32 %rem, 5
[instruction  8]:   %temp3 = add i32 %temp2, 8
[instruction  9]:   %temp4 = sub i32 %temp1, %temp3
[instruction  10]:   %temp5 = sub
[instruction  11]:   store i32 %r
[instruction  12]:   %0 = load i
[instruction  13]:   %call1 = cal
ounds ([8 x i8]* @.str, i32 0, i3
[instruction  14]:   %1 = load i3
[instruction  15]:   ret i32 %1
```

```
define i32 @main() #0 {
entry:
  %call = tail call i32 @rand() #2
  %rem = srem i32 %call, 3
  %call1 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([8 x i8
]* @.str, i64 0, i64 0), i32 %rem) #2
  ret i32 %rem
}
```

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;
}
```

```
main
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   %call = call i32 @rand() #3
[instruction  4]:   %rem = srem i32 %call, 3
[instruction  5]:   %temp = mul i32 %rem, 5
[instruction  6]:   %temp1 = sdiv i32 %temp, 5
[instruction  7]:   %temp2 = ad
[instruction  8]:   %temp3 = su
[instruction  9]:   store i32 %
[instruction  10]:   %0 = load
[instruction  11]:   %call1 = ca
ounds ([8 x i8]* @.str, i32 0,
[instruction  12]:   %1 = load
[instruction  13]:   ret i32 %1
```

```
define i32 @main() #0 {
entry:
  %call = tail call i32 @rand() #2
  %rem = srem i32 %call, 3
  %call1 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([8 x i8
]* @.str, i64 0, i64 0), i32 %rem) #2
  ret i32 %rem
}
```