

# 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



Reverse Engineering

# Code Obfuscation

- Sample disassembly (IDA Pro Disassembler)

The image shows a screenshot of the Minesweeper game window on the left and the corresponding assembly code in the IDA Pro disassembler on the right. A large black arrow points from the game window to the assembly code.

**Minesweeper Game Window:**

- Title bar: Mineswe... (partially visible)
- Menu bar: Game Help
- Score: 0:0
- Difficulty: Easy
- Grid: A 16x16 grid of empty cells.

**IDB Pro Disassembly View:**

Assembly code for the Minesweeper application, showing the creation of window classes and menus.

```
01002223 loc_1002223:  
01002223 8D 45 F8 lea eax, [ebp+piccce]  
01002223 50 push eax  
01002227 C7 45 F8 08+ mov [ebp+piccce.dwStyle], 8  
01002228 C7 45 FC FD+ mov [ebp+piccce.dwCC], 16F0h  
01002235 FF 15 1C 10+ call ds:InitCommonControlsEx  
0100223B 6A 64 push 64h ; lpIconName  
0100223D FF 35 30 5B+ push hInstance ; hInstance  
01002243 FF 15 AC 10+ call ds:LoadIconW  
01002249 BB 0D 30 5B+ mov ecx, hInstance  
0100224F 58 00 7F 00+ push 7F00h ; lpCursorName  
01002254 57 push edi ; hInstance  
01002255 A3 28 5B 00+ mov dwrd 1005B28, eax  
0100225A 89 7D B4 mov [ebp+WndClass.style], edi  
0100225D C7 45 B8 C9+ mov [ebp+WndClass.lpfnWndProc], offset sub_1001BC9  
01002264 89 7D BC mov [ebp+WndClass.cbClsExtra], edi  
01002267 89 D0 C0 mov [ebp+WndClass.cbWndExtra], edi  
0100226A 89 4D C4 mov [ebp+WndClass.hInstance], ecx  
0100226D 89 45 C8 mov [ebp+WndClass.hIcon], eax  
01002270 FF 15 BC 10+ call ds:LoadCursorW  
01002276 53 push ebx ; i  
01002277 89 45 CC mov [ebp+WndClass.hCursor], eax  
0100227A FF 15 60 10+ call ds:GetStockObject  
01002280 89 45 D0 mov [ebp+WndClass.hbrBackground], eax  
01002283 8D 45 B4 lea eax, [ebp+WndClass]  
01002286 BE A0 5A 00+ mov esi, offset ClassName  
01002288 50 push eax ; lpWndClass  
0100228C 89 7D D4 mov [ebp+WndClass.lpszMenuName], edi  
0100228F 89 75 D8 mov [ebp+WndClass.lpszClassName], esi  
01002292 FF 15 CC 10+ call ds:RegisterClassW  
01002298 66 85 C0 test ax, ax  
0100229B OF 84 9A 00+ jz loc 1002338  
  
010022A1 6B F4 01 00+ push 1F4h ; lpMenuName  
010022A6 FF 35 30 5B+ push hInstance ; hInstance  
010022AC FF 15 D4 10+ call ds:LoadMenuW  
010022B2 6B F5 01 00+ push 1F5h ; lpTableName  
010022B7 FF 35 30 5B+ push hInstance ; hInstance  
010022BD A3 94 5A 00+ mov hMenu, eax  
010022C1 FF 15 60 11+ call ds:LoadAcceleratorsW
```

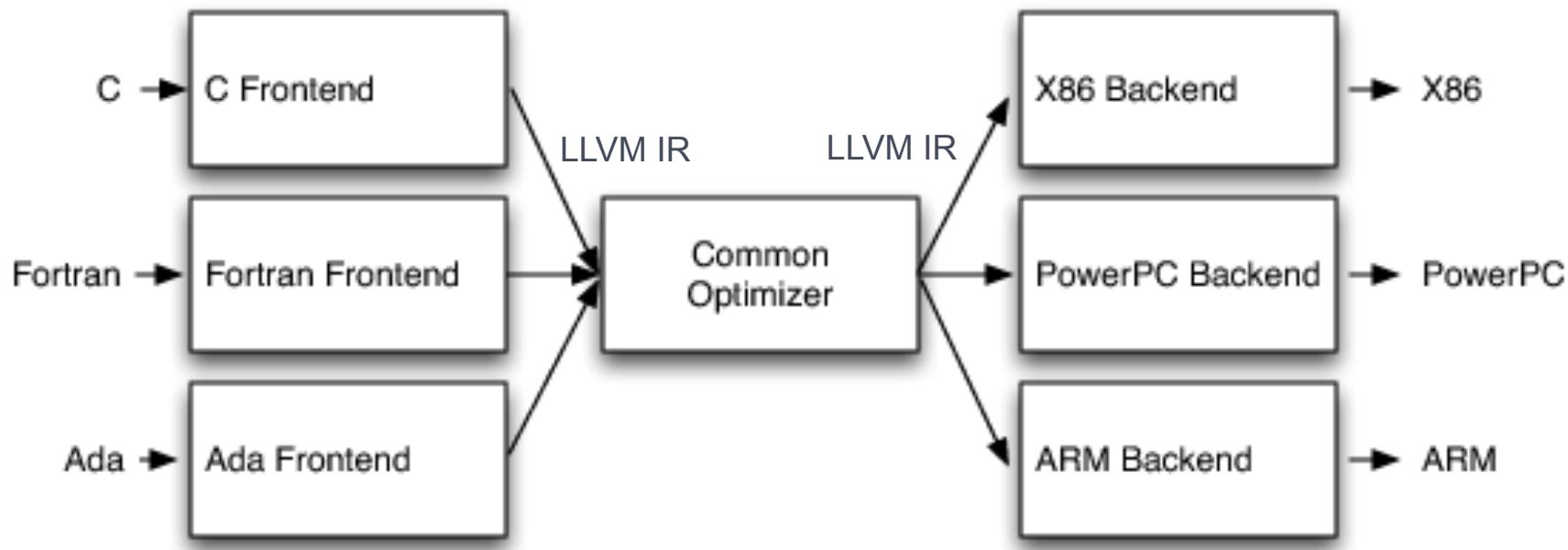
# Code Obfuscation

- 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



# 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 – Method 1
  - Preserve value of  $x$  after insertion.

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

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

```
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
[=====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
```

```
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
```

# Junk Code Insertion

- Chain transformation together multiple times

```
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
```

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

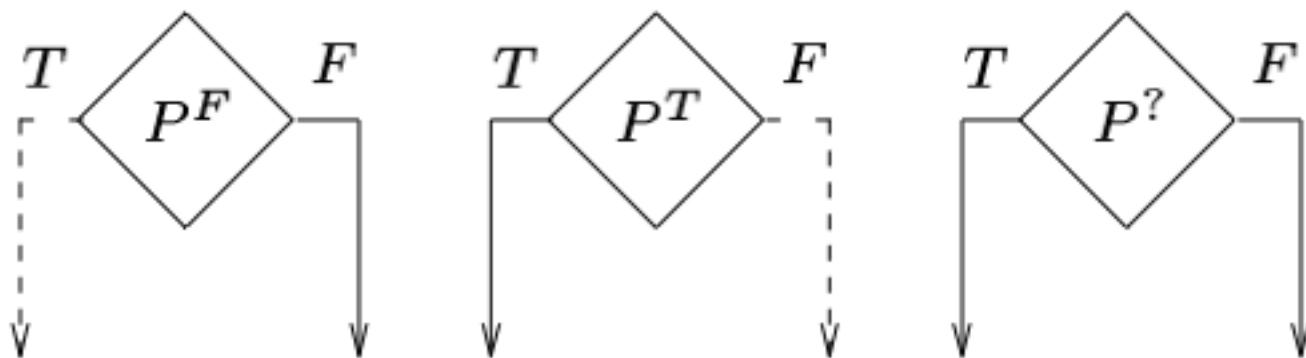
```
[=====after insertion=====]
[instruction 0]: %x.addr = alloca i32, align 4
[instruction 1]: store i32 %x, i32* %x.addr, align 4
[instruction 2]: %0 = load i32* %x.addr, align 4
[instruction 3]: %dec = add nsw i32 %0, -1
[instruction 4]: %temp = add i32 %dec, 5
[instruction 5]: %temp1 = mul i32 %temp, 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
```

# 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

```
; Segment type: Pure code
; text segment para public 'CODE' use64
assume cs: text
;org 100000E80h
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing

public _func1
_func1 proc near

var_8= dword ptr -8
var_4= dword ptr -4

push    rax
mov     [rsp+8+var_8], 0
lea     rdi, aInFunctionFunc ; "In Function: func1()\n"
xor    al, al
call    _printf
mov     [rsp+8+var_4], 0
short loc_100000E87

jmp
```

```
loc_100000E87:
cmp    [rsp+8+var_4], 9
jle    short loc_100000E80

loc_100000E80:
inc    [rsp+8+var_8]
inc    [rsp+8+var_4]

_func1 endp
```

Obfuscated

```
; Segment type: Pure code
; text segment para public 'CODE' use64
assume cs: text
;org 100000B40h
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing

public _func1
_func1 proc near

var_10= dword ptr -10h
var_8= dword ptr -8h
var_4= dword ptr -4h
var_0= dword ptr -0Ch

sub    rsp, 18h
mov    [rsp+18h+var_C], 61094DEDh
mov    [rsp+18h+var_10], 1
mov    [rsp+18h+var_8], 0
lea    rdi, [rsp+18h+var_C]
lea    rsi, [rsp+18h+var_10]
call    _SAI
test   eax, eax
jz    short loc_100000B9F

lea    rdi, aInFunctionFunc ; "In Function: func1()\n"
xor    al, al
call    _printf
mov    [rsp+18h+var_4], 0
jmp
```

```
loc_100000B98:
cmp    [rsp+18h+var_4], 9
jle    short loc_100000B90

loc_100000B90:
inc    [rsp+18h+var_8]
inc    [rsp+18h+var_4]

loc_100000B9F:
xor    eax, eax
add    rsp, 18h
ret
_func1 endp
```

# 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

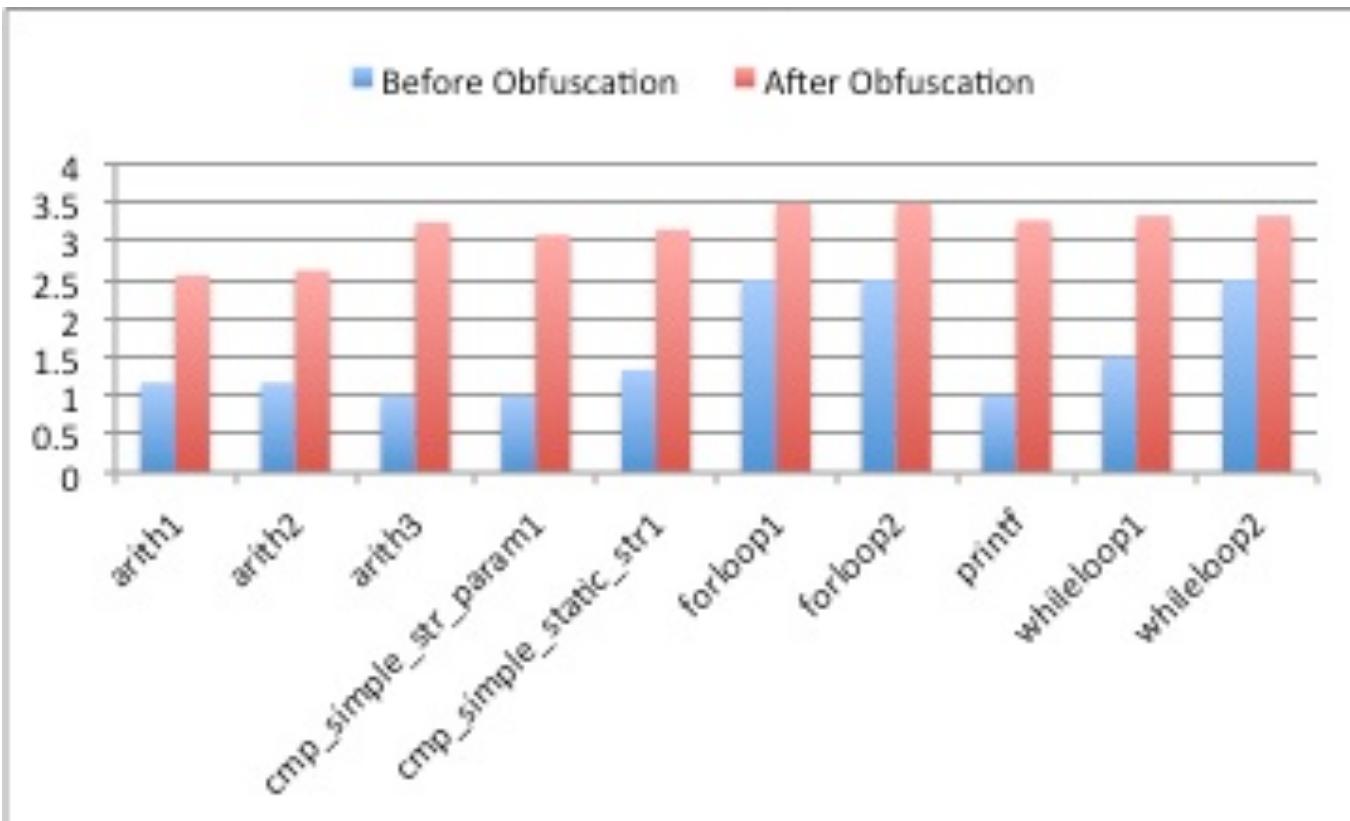
```
// RUN: clang -emit-llvm %s -c -o -| opt -load lib.dylib -objjunk > %t1
// RUN: lli %t1 > %t2
// RUN: diff %t2 %s.out
#include <stdio.h>
void main () { ... }
```

# 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

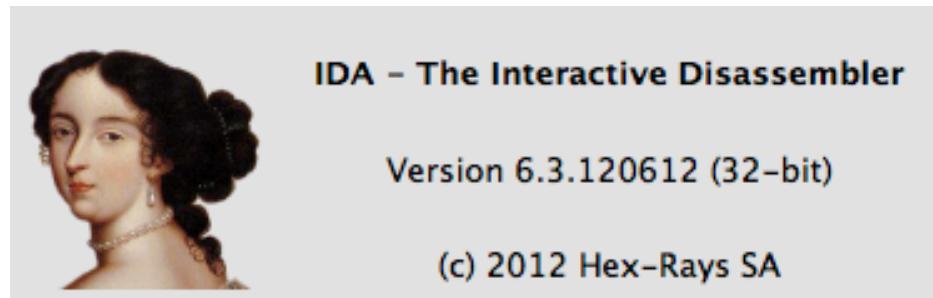
# Testing

## Evaluation of Cyclomatic Complexity



# DEMO

# Environment



# 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

```
main
[=====after insertion=====]
[instruction  0]:  %retval = alloca i32, align 4
[instruction  1]:  %x = alloca i32, align 4
[instruction  2]:  store i32 %x, 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 = srl i32 %temp4, 2
[instruction 11]:  store i32 %temp5, i32* define i32 @main() #0 {
[instruction 12]:  %0 = load i32 @entry:
[instruction 13]:  %call1 = cal %call = tail call i32 @rand() #2
ounds ([8 x i8]* @.str, i32 0, i3 %rem = srem i32 %call, 3
[instruction 14]:  %1 = load i3 %call1 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([8 x i8
[instruction 15]:  ret i32 %1 ]* @.str, i64 0, i64 0), i32 %rem) #2
                           ret i32 %rem
}
```

```
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

```
main
[=====after insertion=====]
[instruction  0]:  %retval = alloca i32, align 4
[instruction  1]:  %x = alloca i32, align 4
[instruction  2]:  store i32 %x, 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 = adddefine i32 @main() #0 {
[instruction  8]:  %temp3 = sumentry:
[instruction  9]:  store i32 %temp3, i32* %call = tail call i32 @rand() #2
[instruction 10]:  %0 = load i32 %rem = srem i32 %call, 3
[instruction 11]:  %call1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([8 x i8] @.str, i32 0, ]* @.str, i64 0, i64 0), i32 %rem) #2
[instruction 12]:  %l = load i32 %rem
[instruction 13]:  ret i32 %l}
```

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