Secure Software Design and Engineering
(CY-321)

Code Obfuscation

Dr. Zubair Ahmad

Informally, to obfuscate a program P means to transform it into a program P ′ that is still executable but for which it is hard to extract information.

“Hard?” ⇒ Harder than before!

static obfuscation ⇒ obfuscated programs that remain fixed at runtime.

dynamic obfuscators ⇒ transform programs continuously at runtime, keeping them in constant flux.

Obfuscating : Expression Equivalence

$x+0 \to x$ → $X*1 \to x$

$x+0 \to x$ → $(a-(-b))$

$a \times b$ → $e\ln(a)+\ln(b)$

Obfuscating : Expression Equivalence

```
y = x + 1
```

→

```
temp = x * 2 - x
y = temp + 1
```

```
y = (x + 5) - 3;
```

→

```
y = x + (5 - 3);
```

Obfuscating : Expression Equivalence

```
y = x * 42;
```

```
y = x << 5;
y += x << 3;
y += x << 1;
```

Obfuscating : Splitting and Merging

```
int compute(int x, int y) {
int result = (x * y) + (x - y);
return result;
}
```

Splitting

```
int multiply(int a, int b) {
return a * b;
}

int difference(int a, int b) {
return a - b;
}

int compute(int x, int y) {
int part1 = multiply(x, y);
int part2 = difference(x, y);
return part1 + part2;
}
```

Obfuscating : Splitting and Merging

```
int modexp ( int y , int x []
, int w , int n ) {
    int R , L ;
    int k = 0;
    int s = 1;
    while (k < w ) {
        f ( x [k ] , s , y , n
, & R );  // Call to `f`
        s = R * R % n ;  //
Square the result
        L = R ;
        k ++;
    }
    return L ;
}
```

Splitting

```
void f ( int xk , int s , int y , int
n , int * R) {
  if ( xk == 1)
    * R = ( s* y ) % n ;
  else
    * R = s ;
}
```

Obfuscating : Splitting and Merging

```
int modexp ( int y , int x []
, int w , int n ) {
    int R , L ;
    int k = 0;
    int s = 1;
    while (k < w ) {
        f ( x [k ] , s , y , n
, & R );  // Call to `f`
        s = R * R % n ;  //
Square the result
        L = R ;
        k ++;
    }
    return L ;
}
```

It takes four parameters:

- y: the base
- x[]: an array of binary exponent values
- w: the number of bits in the exponent
- n: the modulus

It initializes:

- k = 0: index for looping through x[]
- s = 1: stores intermediate results
- L: stores the final computed value

Inside the while loop:

- Calls f(x[k], s, y, n, &R), which modifies R
- Computes s = R * R % n
- Stores R in L
- Increments k
- When k == w, L holds the result and is returned.

Obfuscating : Splitting and Merging

The function f takes the following parameters:

- xk: A single bit from an exponent array (x[]) in the original modexp function.
- s: An intermediate computation value (used in modular exponentiation).
- y: The base of exponentiation.
- n: The modulus for modular exponentiation.
- R: A pointer to store the result.

```
void f ( int xk , int s , int y , int n , int * R)
{
   if ( xk == 1)
      * R = ( s* y ) % n ;
   else
      * R = s ;
}
```

Obfuscating : Splitting and Merging

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

int multiply(int a, int b) {
    return a * b;
}
```

```c
int compute(int a, int b, int operation) {
    if (operation == 0) {
        return a + b;
    } else if (operation == 1) {
        return a * b;
    }
    return -1; // Invalid operation
}
```

Merging

Obfuscating : Splitting and Merging

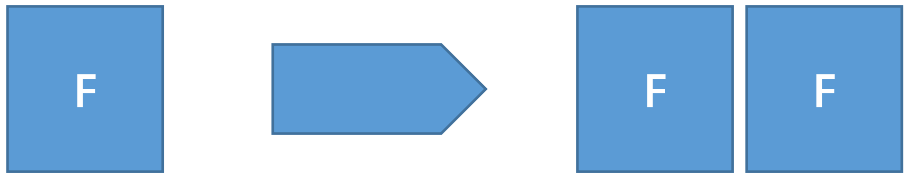
```
float foo[100];

void f(int a, float b) {
    foo[a] = b;  // Stores `b` at index `a` in the array
}

float g(float c, char d) {
    return c * (float)d;  // Multiplies `c` with `d` (converted to float)
}

int main() {
    f(42, 42.0);          // Calls `f()`
    float v = g(6.0, 'a');  // Calls `g()`
}
```

Merging

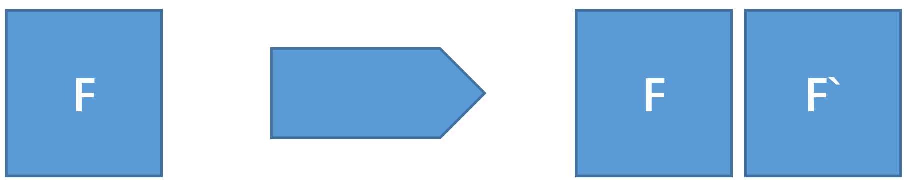
```
float foo[100];

float fg(int a, float bc, char d, int which) {
    if (which == 1)
        foo[a] = bc;  // If `which == 1`, perform `f()` logic (store in array)

    return bc * (float)d;  // Always return `bc * d`, mimicking `g()`
}

int main() {
    fg(42, 42.0, 'b', 1);   // Equivalent to calling `f(42, 42.0)`
    float v = fg(99, 6.0, 'a', 2);  // Equivalent to `g(6.0, 'a')`
}
```

Obfuscating : Copying Code

Make the program larger by cloning pieces of it:



Make the copied code look different from the original:



Now the attacker must examine all pairs of code blocks to see which ones are the same

Obfuscating : Copying Code

```
int compute(int x) {
    return (x * 2) + 5;
}
```

```
int compute(int x) {
    int temp = x * 2;
    int y = temp + 5;
    return y;
}

int compute_alternative(int x) {  // Same logic but slightly changed structure
    int y = (x * 2);
    return y + 5;
}
```

Makes it harder to recognize identical logic.

Code analysis tools may **fail to detect function similarity**.

Obfuscating : Interpretation

Add a level of interpretation

1 Define your own instruction set

2 Translate your program to this instruction set

3 Write an interpreter for the instruction set

Your program: 10-100x slower than before.

Obfuscating : Interpretation

```c
int compute(int x) {
    return (x * 2) + 5;
}
```

```c
int interpretBytecode(int *bytecode, int x) {
    int result = 0;
    for (int i = 0; i < 3; i++) {  // Loop over bytecode instructions
        if (bytecode[i] == 1) result = x * 2;   // Instruction 1 = Multiply by 2
        else if (bytecode[i] == 2) result += 5; // Instruction 2 = Add 5
    }
    return result;
}

int main() {
    int bytecode[] = {1, 2};  // Encoded sequence of operations
    int result = interpretBytecode(bytecode, 10);
    printf("%d\n", result);
    return 0;
}
```

Obfuscating : Opaque Values from Array Aliasing

Opaque values: These are values that appear unpredictable to an adversary but are known at compile or runtime.

Array aliasing: This refers to using multiple references (aliases) to the same memory location in an array, making it difficult to track actual variable values.

Obfuscating : Opaque Values from Array Aliasing

Aliasing occurs in two pointers can refer to the same memory location

Two reference parameters can also alias each other

A reference parameter and a global variable

Two array elements indexed by different variables

Obfuscating : Opaque Values from Array Aliasing

Two Pointers Referring to the Same Memory Location

```
#include <stdio.h>

int main() {
    int num = 42;
    int *ptr1 = &num;
    int *ptr2 = ptr1; // Both ptr1 and ptr2 point
to 'num'

    *ptr2 = 99; // Changing value using ptr2
    printf("%d\n", *ptr1); // Output: 99 (ptr1 also
sees the change)

    return 0;
}
```

Obfuscating : Opaque Values from Array Aliasing

Two Reference Parameters Can Also Alias Each Other

```
#include <iostream>

void modify(int &a, int &b) {
    a = 10;  // Modifies b as well if they alias
each other
    b = 20;
}

int main() {
    int x = 5;
    modify(x, x); // x is passed twice, creating
aliasing
    std::cout << x << std::endl; // Output: 20
(last modification applies)

    return 0;
}
```

Obfuscating : Opaque Values from Array Aliasing

A Reference Parameter and a Global Variable

```
#include <iostream>

int globalVar = 50;

void modifyGlobal(int &param) {
    param = 100; // Modifies globalVar because
param is an alias
}

int main() {
    modifyGlobal(globalVar);
    std::cout << globalVar << std::endl; // Output:
100

    return 0;
}
```

Obfuscating : Opaque Values from Array Aliasing

Two Array Elements Indexed by Different Variables

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr1 = &arr[2]; // Points to arr[2]
    int *ptr2 = arr + 2; // Also points to arr[2]
(same memory location)

    *ptr1 = 99; // Modify arr[2]
    printf("%d\n", *ptr2); // Output: 99

    return 0;
}
```

Obfuscating : Opaque Values from Array Aliasing

Two Array Elements Indexed by Different Variables

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    int i = 1, j = 2;  // Two different index
variables
    int *ptr1 = &arr[i];  // Points to arr[1]
    int *ptr2 = &arr[j - 1];  // Also points to
arr[1] (j - 1 = 1)

    *ptr1 = 99;  // Modify arr[1] using ptr1

    printf("%d\n", *ptr2); // Output: 99 (because
ptr2 also points to arr[1])

    return 0;
}
```

Obfuscating : Opaque Values from Array Aliasing

```
#include <stdio.h>

int main() {
    int arr[3] = {42, 99, 7};
    int *ptr = arr; // Pointer
aliasing the array

    int x = ptr[1]; // Opaque value
from aliasing
    if (x == 99) {
        printf("Secret branch!\n");
    } else {
        printf("Normal branch\n");
    }
}
```

Instead of directly accessing arr[1], we retrieve its value through a pointer alias (ptr[1]).

The value 99 (retrieved via aliasing) determines whether the conditional executes.

Reverse engineers trying to analyze the control flow statically may not recognize that x == 99 is always true.

Instead of writing if (x == 99), the program retrieves 99 dynamically from the array

Obfuscating : Opaque Values from Array Aliasing

```
#include <stdio.h>

int main() {
    int arr[3] = {10, 20, 30};
    int *alias1 = arr;
    int *alias2 = alias1 + 1; // Points to arr[1]

    int x = *alias2; // Opaque value from aliasing
    if (x == 20) {
        printf("Access Granted!\n");
    } else {
        printf("Access Denied!\n");
    }
}
```

Obfuscating : Renaming

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

int main() {
    int sum = add(5,
10);
    return sum;
}
```

```
int x1(int x2, int x3) {
    return x2 + x3;
}

int main() {
    int x4 = x1(5, 10);
    return x4;
}
```

Obfuscating : Code Flattening

```
void process(int x) {
    if (x > 10)

printf("High\n");
    else

printf("Low\n");
}
```

```
void process(int x) {
    int state = (x > 10) ? 1 : 0;

    switch (state) {
        case 1:
            printf("High\n");
            break;
        case 0:
            printf("Low\n");
            break;
    }
}
```

Obfuscating : Junk Code Insertion

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

```
int add(int a, int b) {
    int noise = 42;
    noise += 10;
    return a + b;
}
```

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby