SSPREW 2017 Training

Breaking Obfuscated Programs with Symbolic Execution

Sebastian Banescu
Technical University of Munich, Germany
Chair of Software Engineering
Prof. Alexander Pretschner

Outline

Assumptions and Tools for Hands-on Tutorial

- We assume you have Docker installed and have basic user knowledge
- Docker installation instructions
  `https://docs.docker.com/engine/installation/`
- Docker image based on Ubuntu contains: Tigress, KLEE, STP, Z3, SatGraf, etc.

```
$ docker pull banescusebi/obfuscation-symex
$ docker run -it banescusebi/obfuscation-symex
```

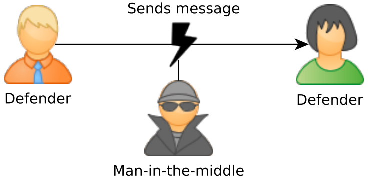
- For instructions on how to start the Docker image read description at `https://hub.docker.com/r/banescusebi/obfuscation-symex/`
- Instructions for running GUI apps available for Ubuntu and Mac OS (not mandatory, only needed for SatGraf)

Expected Outcome

After this training you will have a better understanding of the following:

- The theory and practice of obfuscation and software diversity
- Practical static & dynamic obfuscation transformations
- Tools for applying symbolic execution on obfuscated programs
- Which obfuscation transformations help against symbolic execution

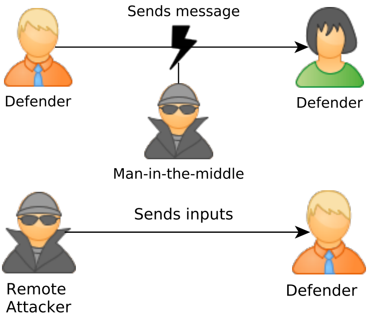
1. **Man-in-the-middle (MITM)** attacks communication channels



Attackers in Computer Security

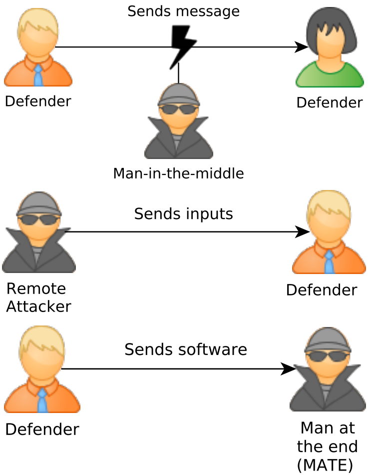
1. **Man-in-the-middle (MITM)** attacks communication channels

2. **Remote attacker** exploits vulnerabilities (e.g. buffer overflows)



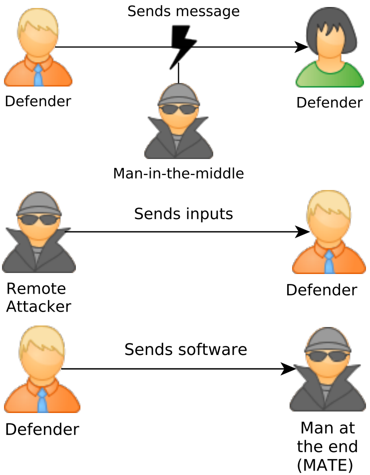
Attackers in Computer Security

1. **Man-in-the-middle (MITM)** attacks communication channels
2. **Remote attacker** exploits vulnerabilities (e.g. buffer overflows)
3. **Man-at-the-end (MATE)** reverse engineers software



Attackers in Computer Security

1. **Man-in-the-middle (MITM)** attacks communication channels
2. **Remote attacker** exploits vulnerabilities (e.g. buffer overflows)
3. **Man-at-the-end (MATE)** reverse engineers software



We focus on **MATE** during this tutorial.

Introduction

Informal Definition of Obfuscation:

To obfuscate a program P means to transform it into a executable program P' from which it is harder to extract information than from P.

Introduction

Informal Definition of Obfuscation:

To obfuscate a program P means to transform it into a executable program P' from which it is harder to extract information than from P.

Motivation:

- Obfuscation is last layer of software defense against attackers (e.g. after attacker bypasses OS authentication)
- Obfuscation raises the bar for reverse engineering

Introduction

Informal Definition of Obfuscation:

To obfuscate a program P means to transform it into a executable program P' from which it is harder to extract information than from P.

Motivation:

- Obfuscation is last layer of software defense against attackers (e.g. after attacker bypasses OS authentication)
- Obfuscation raises the bar for reverse engineering

Popular questions:

- Wasn't obfuscation proved to be impossible back in 2001?
- Isn't obfuscation the same as security by obscurity?

Outline

Black Box Obfuscation

Formal Definition of Black-Box Obfuscation:

A probabilistic algorithm O is an obfuscator if the following conditions hold:

- For every program P, the obfuscated program $O(P)$ has the same functionality (e.g. input-output behavior)

Black Box Obfuscation

Formal Definition of Black-Box Obfuscation:

A probabilistic algorithm O is an obfuscator if the following conditions hold:

- For every program P, the obfuscated program $O(P)$ has the same functionality (e.g. input-output behavior)
- The memory size increase and execution slowdown of $O(P)$ w.r.t. P are less than polynomial

Black Box Obfuscation

Formal Definition of Black-Box Obfuscation:

A probabilistic algorithm O is an obfuscator if the following conditions hold:

- For every program P, the obfuscated program $O(P)$ has the same functionality (e.g. input-output behavior)
- The memory size increase and execution slowdown of $O(P)$ w.r.t. P are less than polynomial
- Any probabilistic polynomial time attacker only has a negligible probability of guessing any bit of information about P, given $O(P)$

Obfuscated Program X

vs.

Black-Box version of Program X

Black Box Obfuscation

- In 2001 Barak et al. [3]:
  - Proved there exists no **general obfuscator** applicable to **all** programs

Black Box Obfuscation

- In 2001 Barak et al. [3]:
  - Proved there exists no **general obfuscator** applicable to **all** programs
  - Proof performed by giving a counter-example using 2 programs:

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

Black Box Obfuscation

- In 2001 Barak et al. [3]:
  - Proved there exists no **general obfuscator** applicable to **all** programs
  - Proof performed by giving a counter-example using 2 programs:

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

  - Attacker is defined as $A(C, D) = D(C)$. If $A(C, D) = 1$ then the attacker is successful

Black Box Obfuscation

- In 2001 Barak et al. [3]:
  - Proved there exists no **general obfuscator** applicable to **all** programs
  - Proof performed by giving a counter-example using 2 programs:

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

  - Attacker is defined as $A(C, D) = D(C)$. If $A(C, D) = 1$ then the attacker is successful
  - If the attacker is given $O(C)$ and $O(D)$ then $A(O(C), O(D)) = 1$ with 100% probability

Black Box Obfuscation

- In 2001 Barak et al. [3]:
  - Proved there exists no **general obfuscator** applicable to **all** programs
  - Proof performed by giving a counter-example using 2 programs:

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

  - Attacker is defined as $A(C, D) = D(C)$. If $A(C, D) = 1$ then the attacker is successful
  - If the attacker is given $O(C)$ and $O(D)$ then $A(O(C), O(D)) = 1$ with 100% probability
  - If the attacker only has black box access to C and D, then the probability of a successful attack is negligible

Black Box Obfuscation

- In 2001 Barak et al. [3]:
  - Proved there exists no **general obfuscator** applicable to **all** programs
  - Proof performed by giving a counter-example using 2 programs:

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

  - Attacker is defined as $A(C, D) = D(C)$. If $A(C, D) = 1$ then the attacker is successful
  - If the attacker is given $O(C)$ and $O(D)$ then $A(O(C), O(D)) = 1$ with 100% probability
  - If the attacker only has black box access to C and D, then the probability of a successful attack is negligible
- This proof does not imply that every obfuscator fails on a **subset** of programs

Black Box Obfuscation

- In 2001 Barak et al. [3]:
  - Proved there exists no **general obfuscator** applicable to **all** programs
  - Proof performed by giving a counter-example using 2 programs:

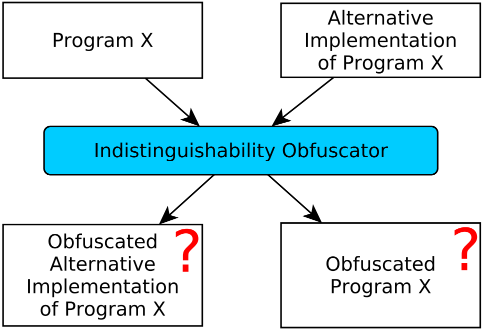
$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

  - Attacker is defined as $A(C, D) = D(C)$. If $A(C, D) = 1$ then the attacker is successful
  - If the attacker is given $O(C)$ and $O(D)$ then $A(O(C), O(D)) = 1$ with 100% probability
  - If the attacker only has black box access to C and D, then the probability of a successful attack is negligible
- This proof does not imply that every obfuscator fails on a **subset** of programs
- There may exist non-black-box obfuscators for some programs that leak bits of information, but are "good enough"

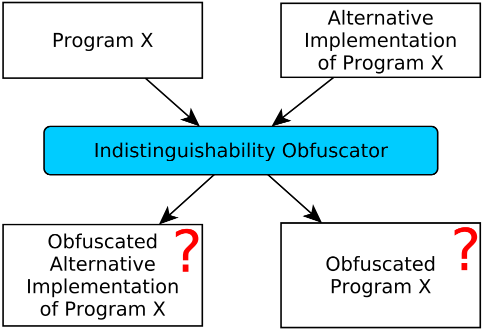
Indistinguishability Obfuscation

- In 2013 Garg et al. [8] proposed indistinguishability obfuscation:
  - The obfuscated versions of 2 semantically equivalent programs cannot be distinguished



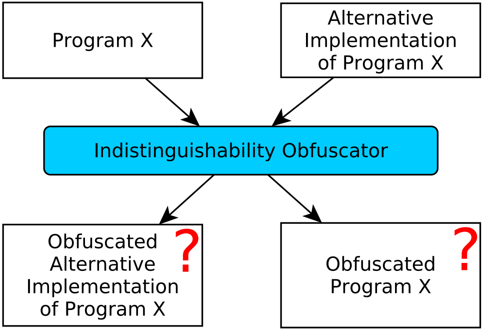
Indistinguishability Obfuscation

- In 2013 Garg et al. [8] proposed indistinguishability obfuscation:
  - The obfuscated versions of 2 semantically equivalent programs cannot be distinguished
  - [9] proved this obfuscation is the best-possible obfuscation



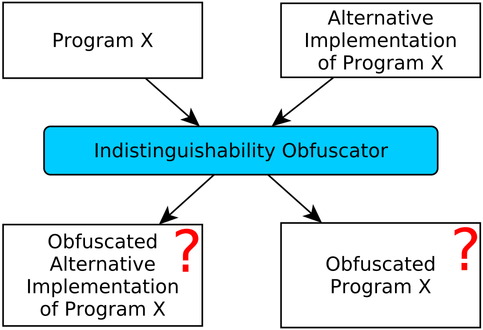
Indistinguishability Obfuscation

- In 2013 Garg et al. [8] proposed indistinguishability obfuscation:
  - The obfuscated versions of 2 semantically equivalent programs cannot be distinguished
  - [9] proved this obfuscation is the best-possible obfuscation
  - Implementations still far from being practical: [1, 2]



Indistinguishability Obfuscation

- In 2013 Garg et al. [8] proposed indistinguishability obfuscation:
  - The obfuscated versions of 2 semantically equivalent programs cannot be distinguished
  - [9] proved this obfuscation is the best-possible obfuscation
  - Implementations still far from being practical: [1, 2]



Cool stuff, but let's look at obfuscation we can use in practice.

Outline

Categorization of Obfuscation

What types of obfuscation transformations exist?

- **Static Vs. Dynamic**
  - **Static obfuscation**:
    - Obfuscated programs remain fixed at runtime
    - Raises the bar against static analysis
    - Can be attacked through dynamic techniques (debugging, emulation, tracing)

Categorization of Obfuscation

What types of obfuscation transformations exist?

- **Static Vs. Dynamic**
  - **Static obfuscation**:
    - Obfuscated programs remain fixed at runtime
    - Raises the bar against static analysis
    - Can be attacked through dynamic techniques (debugging, emulation, tracing)
  - **Dynamic obfuscation**:
    - Programs change during load-/run-time
    - Raises the bar against dynamic analysis
    - Different executions of the same program with the same input may produce different execution traces

Categorization of Obfuscation

What types of obfuscation transformations exist?

- **Static Vs. Dynamic**
  - **Static obfuscation**:
    - Obfuscated programs remain fixed at runtime
    - Raises the bar against static analysis
    - Can be attacked through dynamic techniques (debugging, emulation, tracing)
  - **Dynamic obfuscation**:
    - Programs change during load-/run-time
    - Raises the bar against dynamic analysis
    - Different executions of the same program with the same input may produce different execution traces
- **Point of insertion**:
  - Source code
  - Intermediate representation
  - Machine code

Categorization of Obfuscation

What types of obfuscation transformations exist?

- **Static Vs. Dynamic**
  - **Static obfuscation**:
    - Obfuscated programs remain fixed at runtime
    - Raises the bar against static analysis
    - Can be attacked through dynamic techniques (debugging, emulation, tracing)
  - **Dynamic obfuscation**:
    - Programs change during load-/run-time
    - Raises the bar against dynamic analysis
    - Different executions of the same program with the same input may produce different execution traces
- **Point of insertion**:
  - Source code
  - Intermediate representation
  - Machine code
- **Transformation targets**:
  - **Layout** → scramble identifiers and code layout
  - **Data** → obfuscate data (structures) embedded in code
  - **Control flow** → obfuscate secret algorithms

Outline

Compiler Optimizations

Compiler Optimizations

Compiler Optimizations

- In-lining function bodies
- Loop unrolling
- Loop-invariant code motion
- Common sub-expression elimination
- Constant folding and propagation
- Dead code elimination
- Strength reduction
- more @ `http://en.wikipedia.org/wiki/Optimizing_compiler`

In-lining function bodies

Replace function call by function body

Before

```c
int foo(int a, int b) {
  return a + b;
}
...
c = foo(a, b+1);
```

After

```c
...
c = a + b + 1;
```

Loop unrolling

Remove "end-of-loop" test overhead

Before

```
...
for (i = 0; i < 2; i++)
{
  a[i] = 0;
}
```

After

```
...
a[0] = 0;
a[1] = 0;
```

Loop invariant code-motion

Extract operations whose results are independent of loop execution

Before

```
...
for (i = 0; i < 2; i++)
{
  a[i] = p + q;
}
```

After

```
...
temp = p + q;
for (i = 0; i < 2; i++)
{
  a[i] = temp;
}
```

Common subexpression elimination

Replace re-occurring identical (sub-)expressions by a single variable holding the result

Before

```
...
a = b + (z + 1);
p = q + (z + 1);
```

After

```
...
temp = z + 1;
a = b + temp;
p = q + temp;
```


Constant folding and propagation

Simplify constant expressions and substitute the values of known constants in expressions

Before

```
...
a = 3 + 5;
b = a + 1;
func(a, b);
```

After

```
...
func(8, 9);
```

Dead code elimination

Remove code which does not affect program results: unreachable code and code that affects variables that are irrelevant for the program

Before

```
...
a = 1;
if (a < 0)
{
  printf("This should never be printed!");
}
```

After

```
...
a = 1;
```


Strength reduction

Replace expensive operations with equivalent cheap ones

Before

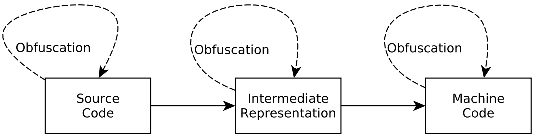
```
...
y = x / 8;
p = q * 15;
```

After

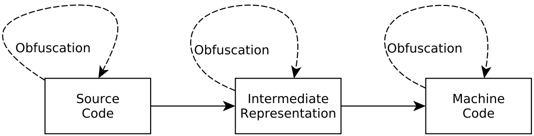
```
...
y = x >> 3;
p = (q << 4) - x;
```


Outline

Automated Code Obfuscation



Automated Code Obfuscation



Obfuscation Techniques:

- Scramble identifiers
- Instruction substitution
- Garbage code insertion
- Merging and splitting functions
- Encode Literals
- Encode Arithmetic
- Opaque predicates
- Control-flow flattening
- Virtualization obfuscation
- White-box cryptography

Scramble identifiers

Replace identifier names with random strings

Before

```
...
sum = 0;
for (i = 0; i < arr_len; i++)
  sum += arr[i];
average /= arr_len;
```

After

```
...
za82b547bcb = 0;
for (z1c0ab7cf0c = 0; z1c0ab7cf0c < za862d19cbc;
     z1c0ab7cf0c++)
  za82b547bcb += zc1c28ca67f[z1c0ab7cf0c];
z8c8f7c7867 /= za862d19cbc
```

This layout obfuscation has high potency, but low resilience

Instruction substitution

Replace binary operation (e.g. +, -, AND, OR, XOR, etc.) by functionally equivalent, but more complicated computations

Before

```
...
a = b + c
```

After

```
...
r = rand();
a = b + r;
a = a + c;
a = a - r;
```

Garbage code insertion

Insert code that executes, but does not affect the IO behavior

Before

```
...
sum = 0;
for (i = 0; i < arr_len; i++)
  sum += arr[i];
average = sum / arr_len;
```

After

```
...
sum = 0; prod = 1;
for (i = 0; i < arr_len; i++) {
  sum += arr[i];
  prod *= arr[i];
}
average = sqrt(prod);
average = sum / arr_len;
```

Merging and splitting functions

- Merging implies combining the code of two or more functions into a single function
- Splitting implies dividing the code one function into two or more functions

Split

```
func1(int a, int b) {
  x = 4;
  if (a < 3)
    x = x + 6;
  x *= b;
}

func2(int a, int c) {
  y = a + 12;
  y = y/c;
}
```

Merged

```
func3(int a, int b, int c) {
  if (c % 2 == 0) {
    x = 4;
    if (a < 3)
      x = x + 6;
    x *= b;
  } else {
    y = a + 12;
    y = y/b;
  }
}
```

Encode Literals

- Literals are constant (hard-coded) strings and numeric values
- Literals can be encoded in numerous ways using encoder functions
- At runtime they are decoded back to their original value

Before

```c
main(int ac, char* av[]) {
  printf("hello\n");
  return 0;
}
```

After

```c
gen_str(char str[]) {
  int i = 0;
  str[i++] = 'h';
  str[i++] = 'e';
  str[i++] = 'l';
  str[i++] = 'l';
  str[i++] = 'o';
  str[i++] = '\n';
  str[i] = '\000';
}

main(int ac, char* av[]) {
  char str[7];
  gen_str(str);
  printf(str);
  return 0;
}
```

Encode Arithmetic (Mixed-Boolean Arithmetic)

- Replace arithmetic or Boolean expressions with more complex ones
- Complex expressions contain both arithmetic and boolean operators
- Transformation can be applied recursively to increase strength

Before

```
func(int x, int y) {
 return x + y;
}
```

After (Version 1)

```
func(int x, int y) {
 return 2*(x | y) - (x ^ y);
}
```

After (Version 2)

```
func(int x, int y) {
 return (x | y) + (x & y);
}
```


Opaque predicates and opaque expressions [6]

Informal Definition:

An expression whose value is known to the defender (at obfuscation time), but which is difficult for an attacker to figure out statically.

Opaque predicates and opaque expressions [6]

Informal Definition:

An expression whose value is known to the defender (at obfuscation time), but which is difficult for an attacker to figure out statically.

Notation:

- P^T for an opaquely true predicate
- P^F for an opaquely false predicate
- $P^?$ for an opaquely intermediate predicate (range divider)
- $E^{=v}$ for an opaque expression of value v

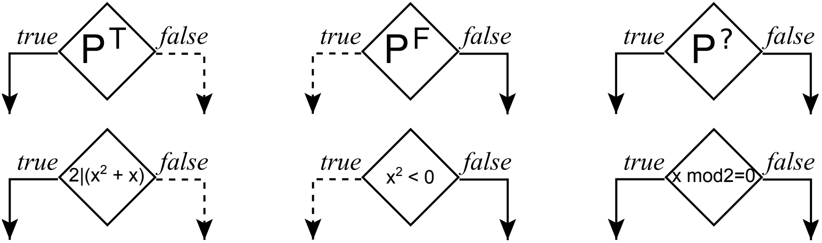
Opaque predicates and opaque expressions [6]

Informal Definition:

An expression whose value is known to the defender (at obfuscation time), but which is difficult for an attacker to figure out statically.

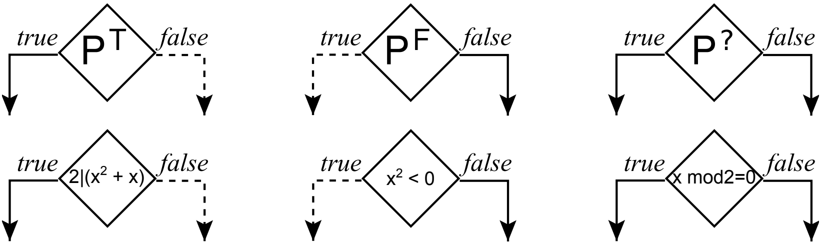
Notation:

- P^T for an opaquely true predicate
- P^F for an opaquely false predicate
- $P^?$ for an opaquely intermediate predicate (range divider)
- $E^{=v}$ for an opaque expression of value v



Bogus control-flow via opaque predicates

- Opaque predicates facilitate insertion of bogus control-flow:
  - Does not affect I/O behavior
  - Attacker does not know which code is bogus
  - Increases attack / analysis time
- Resilience of bogus control-flow reduced to resilience of opaque predicates



Bogus control-flow via opaque predicates

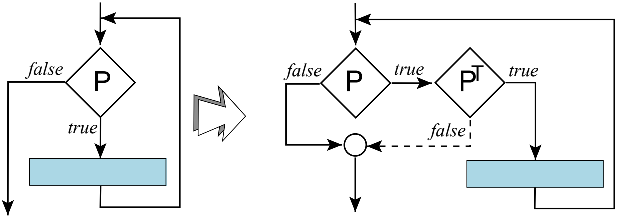
E.g. add an opaquely true predicate (P^T) to a *while* loop condition (P)

Before

```
i = 1;
while (i < 100) {
  dostuff(i);
  i++;
}
```

After

```
i = 1; j = 100;
while ((i < 100) &&
  (j*j*(j+1)*(j+1)%4 == 0)) {
  dostuff(i);
  i++;
  j = j*i+3;
}
```



Breaking opaque predicates via abstract interp.

- Dalla Preda et al. [7] used abstract interpretation to break opaque predicates:
  - Opaque predicates are confined in a single basic block
  - Only opaque predicate of the following form: $n|p(x), \forall x \in \mathbb{Z}$, where $p(x)$ is a polynomial in x and $n \in \mathbb{N}$

Breaking opaque predicates via abstract interp.

- Dalla Preda et al. [7] used abstract interpretation to break opaque predicates:
  - Opaque predicates are confined in a single basic block
  - Only opaque predicate of the following form: $n|p(x), \forall x \in \mathbb{Z}$, where $p(x)$ is a polynomial in x and $n \in \mathbb{N}$
- E.g. opaquely true predicate: $2|(x^2 + x), \forall x \in \mathbb{Z}$

Breaking opaque predicates via abstract interp.

- Dalla Preda et al. [7] used abstract interpretation to break opaque predicates:
  - Opaque predicates are confined in a single basic block
  - Only opaque predicate of the following form: $n|p(x), \forall x \in \mathbb{Z}$, where $p(x)$ is a polynomial in x and $n \in \mathbb{N}$
- E.g. opaquely true predicate: $2|(x^2 + x), \forall x \in \mathbb{Z}$

x is odd

```
x = ...; // any odd number
y = x * x; // odd
y = y + x; // even
if ( y % 2 == 0) // always
...              // true
else // dead branch
...
```

Breaking opaque predicates via abstract interp.

- Dalla Preda et al. [7] used abstract interpretation to break opaque predicates:
  - Opaque predicates are confined in a single basic block
  - Only opaque predicate of the following form: $n|p(x), \forall x \in \mathbb{Z}$, where $p(x)$ is a polynomial in x and $n \in \mathbb{N}$
- E.g. opaquely true predicate: $2|(x^2 + x), \forall x \in \mathbb{Z}$

x is odd

```
x = ...; // any odd number
y = x * x; // odd
y = y + x; // even
if ( y % 2 == 0) // always
...              // true
else // dead branch
...
```

x is even

```
x = ...; // any even number
y = x * x; // even
y = y + x; // even
if ( y % 2 == 0) // always
...              // true
else // dead branch
...
```

Breaking opaque predicates via abstract interp.

- Dalla Preda et al. [7] used abstract interpretation to break opaque predicates:
  - Opaque predicates are confined in a single basic block
  - Only opaque predicate of the following form: $n|p(x), \forall x \in \mathbb{Z}$, where $p(x)$ is a polynomial in x and $n \in \mathbb{N}$
- E.g. opaquely true predicate: $2|(x^2 + x), \forall x \in \mathbb{Z}$

x is odd

```
x = ...; // any odd number
y = x * x; // odd
y = y + x; // even
if ( y % 2 == 0) // always
...              // true
else // dead branch
...
```

x is even

```
x = ...; // any even number
y = x * x; // even
y = y + x; // even
if ( y % 2 == 0) // always
...              // true
else // dead branch
...
```

- Abstract interpretation is able to infer that regardless of x's value, the *IF* condition is always true

Opaquely intermediate predicate (Range divider)

- Range dividers can lead to different paths in the code
- No dead code
- All branches have the same behavior but different syntax

Before

```c
int main(int ac, char* av[]){
  char *str = av[1];
  int hash = 0;
  for(int i = 0;
      i < strlen(str);
      str++, i++) {
    hash = (hash<<7)^(*str);
  }
  if (hash == 809267)
    printf("You win!");
  return 0;
}
```

After

```c
int main(int ac, char* av[]){
  char *str = av[1];
  int hash = 0;
  for(int i = 0;
      i < strlen(str);
      str++, i++) {
    char chr = *str;
    if (chr > 42) {
      hash = (hash << 7) ^ chr;
    } else {
      hash = (hash * 128) ^ chr;
    }
  }
  if (hash == 809267)
    printf("You win!");
  return 0;
}
```

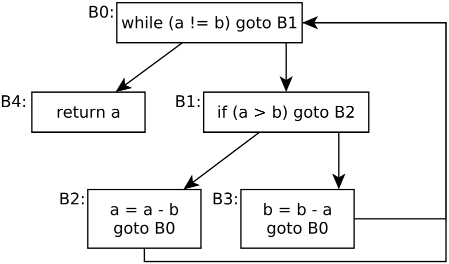
Control-flow flattening

- Remove the control-flow structure of functions:
  1. Put each basic block as a case inside a switch statement
  2. Wrap the switch inside an infinite loop

Control-flow flattening

- Remove the control-flow structure of functions:
  1. Put each basic block as a case inside a switch statement
  2. Wrap the switch inside an infinite loop
- Let's take one function, e.g. GCD

```c
int gcd(int a, int b){
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```



Control-flow flattening GCD example

Before

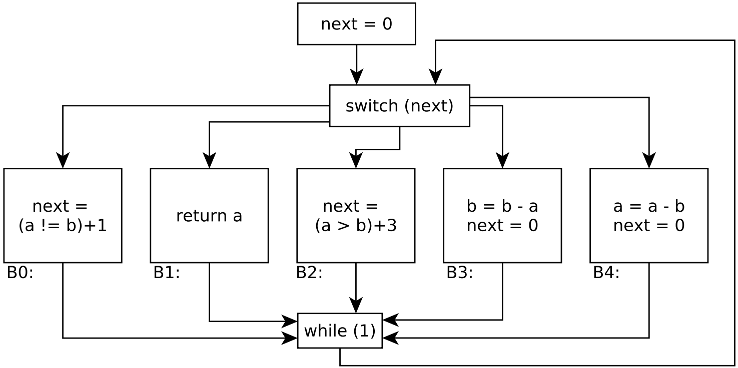
```
int gcd(int a, int b){
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

After

```
int gcd(int a, int b){
int next = 0;
while(1) {
  switch(next) {
    case 0:
      next = (int)(a != b) + 1;
      break;
    case 1: return a;
      break;
    case 2:
      next = (a > b) + 3;
      break;
    case 3: b = b - a;
      next = 0;
      break;
    case 4: a = a - b;
      next = 0;
      break;
    default:
      break;
}}}
```


Control-flow flattening GCD example

The CFG of the resulting code:



Control-flow flattening discussion

Performance penalty:

- For 3 SPEC programs: $4\times$ slowdown, $2\times$ size
- Reasons:
  - The wrapper loop condition check, plus jump
  - The switch `next` value check, plus indirect jump
- How to optimize:
  - Keep tight loops as one switch entry (don't split)
  - Use gcc's *labels-as-values* $\rightarrow$ allows jumping to next basic block

Control-flow flattening discussion

Performance penalty:

- For 3 SPEC programs: $4\times$ slowdown, $2\times$ size
- Reasons:
  - The wrapper loop condition check, plus jump
  - The switch `next` value check, plus indirect jump
- How to optimize:
  - Keep tight loops as one switch entry (don't split)
  - Use gcc's *labels-as-values* $\rightarrow$ allows jumping to next basic block

Attack on control-flow flattening:

1. Find next block of every basic block
2. Rebuild original CFG

Control-flow flattening discussion

Performance penalty:

- For 3 SPEC programs: $4\times$ slowdown, $2\times$ size
- Reasons:
  - The wrapper loop condition check, plus jump
  - The switch `next` value check, plus indirect jump
- How to optimize:
  - Keep tight loops as one switch entry (don't split)
  - Use gcc's *labels-as-values* $\rightarrow$ allows jumping to next basic block

Attack on control-flow flattening:

1. Find next block of every basic block
2. Rebuild original CFG

Mitigation: assign opaque expressions ($E^{=v}$) to `next`
Question: How do we build such opaque expressions?

Opaque expressions from array aliasing

1. A statically initialized **array with seemingly random values**:

   g =

10	5	13	3	27	5	24	38	0	73	115	3	66	60	17	31

2. The values are generated such that some **invariants** hold, e.g.:
   - Every 3rd cell starting from cell 0, contains a value $v \equiv 3 \mod 7$
   - Every 3rd cell starting from cell 1, contains a value $v \equiv 5 \mod 11$
   - Cells 2, 5, 8, 11 and 14 contain values 13, 5, 0, 3, respectively 17
3. **Update array cells** with values that respect invariants

Opaque expressions from array aliasing

1. A statically initialized **array with seemingly random values**:

   g =

10	5	13	3	27	5	24	38	0	73	115	3	66	60	17	31

2. The values are generated such that some **invariants** hold, e.g.:
   - Every 3rd cell starting from cell 0, contains a value $v \equiv 3 \mod 7$
   - Every 3rd cell starting from cell 1, contains a value $v \equiv 5 \mod 11$
   - Cells 2, 5, 8, 11 and 14 contain values 13, 5, 0, 3, respectively 17
3. **Update array cells** with values that respect invariants

Let's replace right-hand values $0, 1, 2, 3$ and 4 of assignments to `next` with $E^{=0}, E^{=1}, E^{=2}, E^{=3}$, respectively $E^{=4}$:

- `next = 0` → `next = g[3] % g[11] - g[8]`
- `next = 1` → `next = 3 * g[11] - 4 * (g[4] % g[5])`
- `next = 2` → `next = g[5] - g[3]`
- `next = 3` → `next = g[2] % g[1]`
- `next = 4` → `next = g[15] - g[4]`

Next slide shows how this looks in the flattened GCD example

Control-flow flattening + opaque expressions

```
int gcd(int a, int b){
int g[] = {10, 5, 13, 3, 27, 5, 24, 38, 0, 73, 115,
     3, 66, 60, 17, 31};
int next = g[3] % g[11] - g[11]; // 0
while(1) {
  switch(next) {
    case 0:
      if(a != b)
        next = 3 * g[11] - 4 * (g[4] % g[5]); // 1
      else next = g[5] - g[3]; // 2
      break;
    case 1:
      if (a > b) next = g[2] % g[1]; // 3
      else next = g[15] - g[4]; // 4
      break;
    case 2: return a;
      break;
    case 3: a = a - b;
      next = g[3] % g[11] - g[8]; // 0
      break;
    case 4: b = b - a;
      next = g[3] % g[11] - g[8]; // 0
      break;
    default:
      break;
}}}
```

Opaque expressions from pointer aliasing

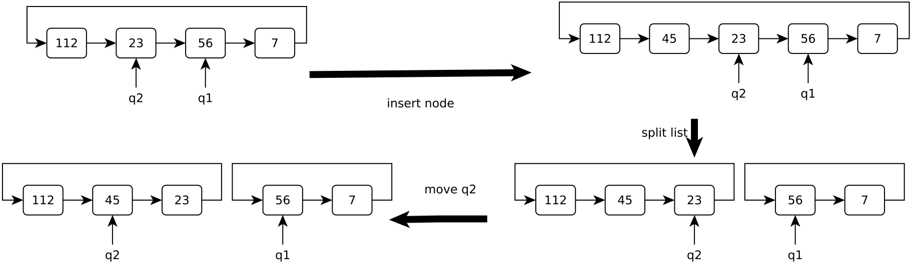
Assumption: pointer aliasing is a computationally hard static analysis problem

1. Construct one or more linked-lists
2. Set pointers into those linked-lists
3. Create opaque predicates by checking properties you know to be true / false, e.g.:
   - → q_1 points to a node with value $v > 53$
   - → q_2 points to a node with value $v \leq 53$

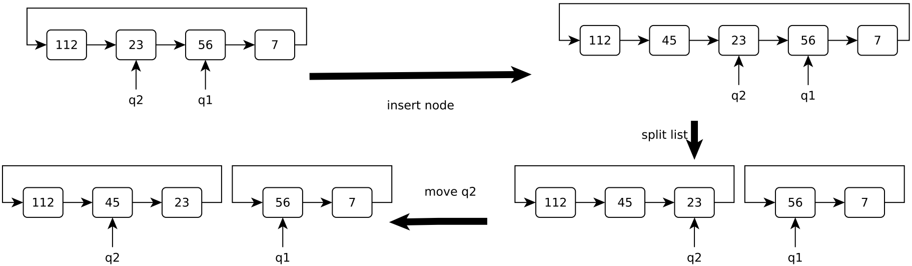
Opaque expressions from pointer aliasing

Assumption: pointer aliasing is a computationally hard static analysis problem

1. Construct one or more linked-lists
2. Set pointers into those linked-lists
3. Create opaque predicates by checking properties you know to be true / false, e.g.:
   - → q_1 points to a node with value $v > 53$
   - → q_2 points to a node with value $v \leq 53$



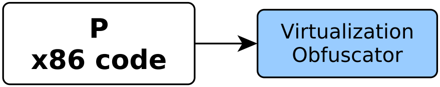
Opaque expressions from pointer aliasing



- One opaquely true predicate: $(*q_1 > *q_2)^T$
- After performing several operations to confuse alias analysis another opaque predicate is: $(q_1 \neq q_2)^T$

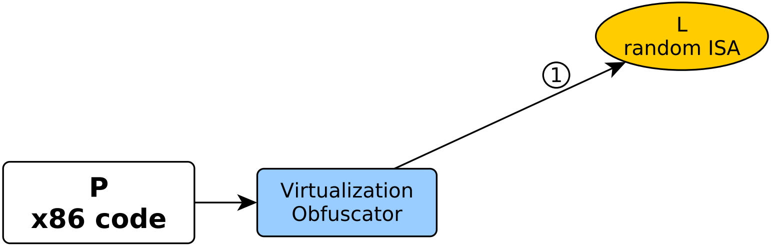
Virtualization Obfuscation

- **Goal:** Hide secret **algorithm** of program P



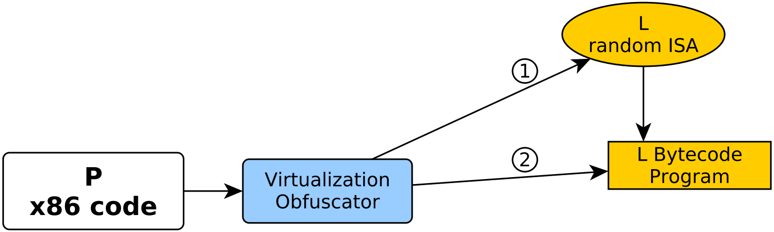
Virtualization Obfuscation

- **Goal:** Hide secret **algorithm** of program P
- **Obfuscation Procedure:**
  1. Generate random bytecode ISA (L) covering all instructions of P



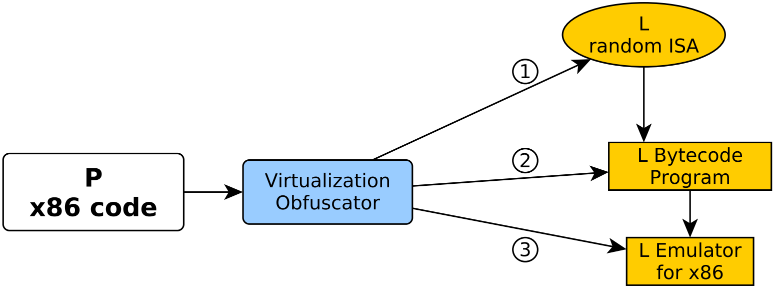
Virtualization Obfuscation

- **Goal:** Hide secret **algorithm** of program *P*
- **Obfuscation Procedure:**
  1. Generate random bytecode ISA (*L*) covering all instructions of *P*
  2. Translate *P* to *L* bytecode program



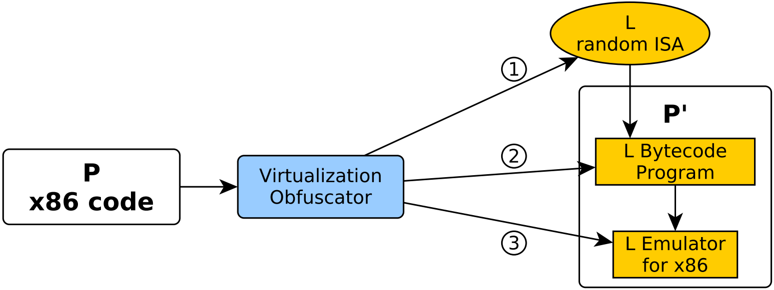
Virtualization Obfuscation

- **Goal:** Hide secret **algorithm** of program P
- **Obfuscation Procedure:**
  1. Generate random bytecode ISA (L) covering all instructions of P
  2. Translate P to L bytecode program
  3. Generate emulator to interpret L bytecode on x86 machine



Virtualization Obfuscation

- **Goal:** Hide secret **algorithm** of program P
- **Obfuscation Procedure:**
  1. Generate random bytecode ISA (L) covering all instructions of P
  2. Translate P to L bytecode program
  3. Generate emulator to interpret L bytecode on x86 machine
- Obfuscated program (P') consists of bytecode program and emulator



Virtualization Obfuscation Example [11]

We apply virtualization obfuscation to function `foo`, written in C

```c
1 void foo(int x){
2   int y = 10;
3   y++;
4   y++;
5   if (x > 0){
6       y++;
7   }
8   else {}
9   printf("%d\n",y);
10 }
```

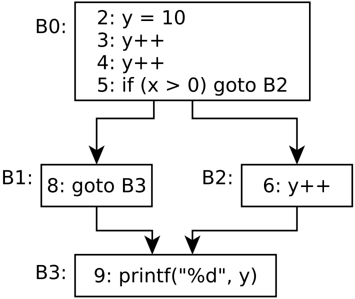


Figure : CFG of `foo`

Virtualization Obfuscation Example

Step 1: Generate random bytecode ISA covering all instructions of `foo`

```
1  void foo(int x){
2    int y = 10;
3    y++;
4    y++;
5    if (x > 0){
6        y++;
7    }
8    else {}
9    printf("%d\n",y);
10 }
```

Random ISA:

1. Integer assignment (line 2)
   $\xrightarrow{\textit{encode}}$ 52, LH_op, RH_op

Virtualization Obfuscation Example

Step 1: Generate random bytecode ISA covering all instructions of `foo`

```
1  void foo(int x){
2    int y = 10;
3    y++;
4    y++;
5    if (x > 0){
6        y++;
7    }
8    else {}
9    printf("%d\n",y);
10 }
```

Random ISA:

1. Integer assignment (line 2)
   $\xrightarrow{encode}$ 52, LH_op, RH_op
2. Integer increment (lines 3,4,6)
   $\xrightarrow{encode}$ 03, op

Virtualization Obfuscation Example

Step 1: Generate random bytecode ISA covering all instructions of `foo`

```c
1 void foo(int x){
2   int y = 10;
3   y++;
4   y++;
5   if (x > 0){
6      y++;
7   }
8   else {}
9   printf("%d\n",y);
10 }
```

Random ISA:

1. Integer assignment (line 2)
   $\xrightarrow{\textit{encode}}$ 52, LH_op, RH_op
2. Integer increment (lines 3,4,6)
   $\xrightarrow{\textit{encode}}$ 03, op
3. Branch if > 0 (line 5)
   $\xrightarrow{\textit{encode}}$ 08, tst_op, jmp_offset

Virtualization Obfuscation Example

Step 1: Generate random bytecode ISA covering all instructions of `foo`

```
1  void foo(int x){
2    int y = 10;
3    y++;
4    y++;
5    if (x > 0){
6       y++;
7    }
8    else {}
9    printf("%d\n",y);
10 }
```

Random ISA:

1. Integer assignment (line 2)
   $\xrightarrow{\textit{encode}}$ 52, LH_op, RH_op
2. Integer increment (lines 3,4,6)
   $\xrightarrow{\textit{encode}}$ 03, op
3. Branch if >0 (line 5)
   $\xrightarrow{\textit{encode}}$ 08, tst_op, jmp_offset
4. Call to `printf` (line 9)
   $\xrightarrow{\textit{encode}}$ 18, op

Virtualization Obfuscation Example

Variables and constants of bytecode program stored in array: `data`

- `data[0]` represents variable `x`
- `data[1]` represents variable `y`
- `data[2]` represents constant for initialization to 10 (line 2 of `foo`)
- `data[3]` represents constant jump offset of conditional branch

```
1  void foo(int x){
2    int y = 10;
3    y++;
4    y++;
5    if (x > 0){
6        y++;
7    }
8    else {}
9    printf("%d\n",y);
10 }
```

Random ISA:

1. Integer assignment (line 2)
   $\xrightarrow{\textit{encode}}$ 52, LH_op, RH_op
2. Integer increment (lines 3,4,6)
   $\xrightarrow{\textit{encode}}$ 03, op
3. Branch if > 0 (line 5)
   $\xrightarrow{\textit{encode}}$ 08, tst_op, jmp_offset
4. Call to `printf` (line 9)
   $\xrightarrow{\textit{encode}}$ 18, op

Virtualization Obfuscation Example

Step 2: Translate `foo` to bytecode program

- `data = {00,00,10,05} // {x, y, init_const, jmp_offset}`
- Bytecode: {52, 01, 02, 03, 01, 03, 01, 08, 00, 03, 03, 01, 18, 01, 00}

```
1  void foo(int x){// bytecode
2    int y = 10; // 52, 01, 02
3    y++;        // 03, 01
4    y++;        // 03, 01
5    if (x > 0){ // 08, 00, 03
6        y++;    // 03, 01
7    }
8    else {}
9    printf("%d",y); // 18, 01
10 }                 // 00
```

Random ISA:

1. Integer assignment (line 2) $\xrightarrow{\textit{encode}}$ 52, LH_op, RH_op
2. Integer increment (lines 3,4,6) $\xrightarrow{\textit{encode}}$ 03, op
3. Branch if > 0 (line 5) $\xrightarrow{\textit{encode}}$ 08, tst_op, jmp_offset
4. Call to `printf` (line 9) $\xrightarrow{\textit{encode}}$ 18, op

Virtualization Obfuscation Example

Step 3: Generate emulator to interpret bytecode on x86 machine

```
int data = {00,00,10,05};
{x, y, init_const, jmp_off}

int code = {
52, 01, 02,
03, 01,
03, 01,
08, 00, 03,
03, 01,
18, 01,
00};
```

Virtualization Obfuscation Example

Step 3: Generate emulator to interpret bytecode on x86 machine

```
int data = {00,00,10,05};
{x, y, init_const, jmp_off}

int code = {
52, 01, 02,
03, 01,
03, 01,
08, 00, 03,
03, 01,
18, 01,
00};
```

```
int vpc = 0, op1, op2;
while (true) {
  switch(code[vpc]) {
    case 03: // increment
      op1 = code[vpc + 1];
      data[op1]++;
      vpc += 2; break;
    case 08: // conditional jump
      op1 = code[vpc + 1];
      op2 = code[vpc + 2];
      if (data[op1] > 0)
        vpc += 3;
      else
        vpc += data[op2];
      break;
    case 18: // call printf
      op1 = code[vpc + 1];
      printf("%d\n", data[op1]);
      vpc += 2; break;
    case 52: // assignment
      op1 = code[vpc + 1];
      op2 = code[vpc + 2];
      data[op1] = data[op2];
      vpc += 3; break;
    default: return; // halt
  } // end switch
} // end while
```


Virtualization Obfuscation Example

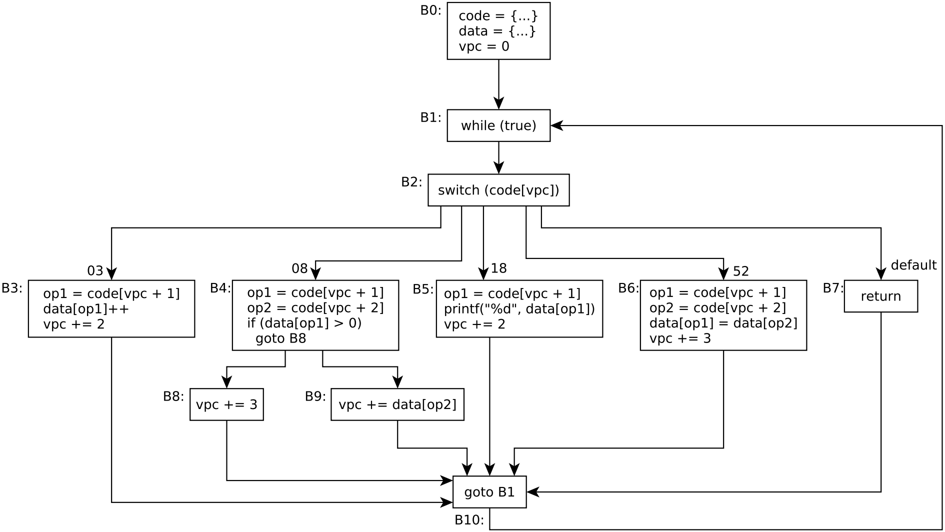


Figure : Control flow graph of obfuscated `foo`

Virtualization Obfuscation

- **Goal:** obfuscate secret algorithm (not secret data)

Virtualization Obfuscation

- **Goal:** obfuscate secret algorithm (not secret data)
- **Strengths:**
  - Random ISA, different for each instance

Virtualization Obfuscation

- **Goal:** obfuscate secret algorithm (not secret data)
- **Strengths:**
  - Random ISA, different for each instance
  - Control-flow flattening $\rightarrow$ Different locations in original program mapped to same case branch

Virtualization Obfuscation

- **Goal:** obfuscate secret algorithm (not secret data)
- **Strengths:**
  - Random ISA, different for each instance
  - Control-flow flattening → Different locations in original program mapped to same `case` branch
  - Many other obfuscation techniques can be added on top of:
    - Each `case` branch
    - The `switch` statement
    - The `data` and `code` arrays

Virtualization Obfuscation

- **Goal:** obfuscate secret algorithm (not secret data)
- **Strengths:**
  - Random ISA, different for each instance
  - Control-flow flattening → Different locations in original program mapped to same case branch
  - Many other obfuscation techniques can be added on top of:
    - Each `case` branch
    - The `switch` statement
    - The `data` and `code` arrays
- **Tools:**
  - CodeVirtualizer (59 € - 119 €)
  - VMProtect (69 € - 599 €)
  - Tigress, Diablo (Free research tools)

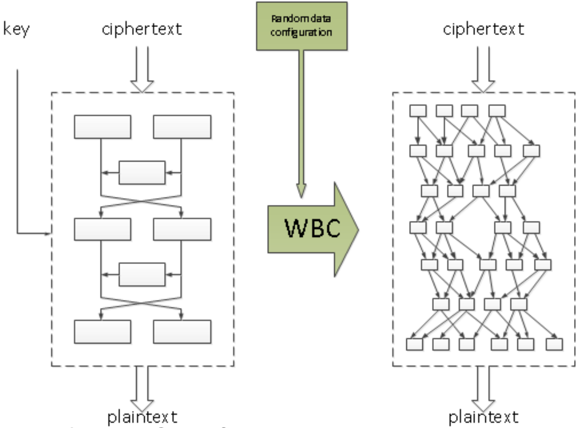
White-Box Cryptography (WBC)

- **Idea:**
  - Embed the key inside the cipher (e.g. in S-boxes)
  - Convert cipher computation into large network of look-up tables

White-Box Cryptography (WBC)

- **Idea:**
  - Embed the key inside the cipher (e.g. in S-boxes)
  - Convert cipher computation into large network of look-up tables

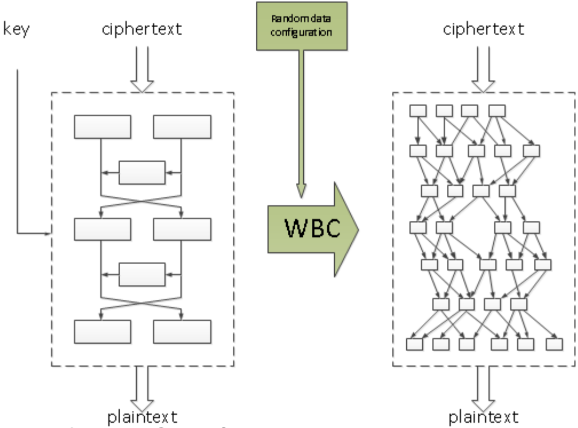
Source: `http://www.whiteboxcrypto.com/`



White-Box Cryptography (WBC)

- **Idea:**
  - Embed the key inside the cipher (e.g. in S-boxes)
  - Convert cipher computation into large network of look-up tables
- **Attack:** WB-AES and WB-DES broken (2^{22} operations)

Source: `http://www.whiteboxcrypto.com/`



Some Details on WBC

Before

```c
char xor(char inputs)
{
  char a = inputs & 0x0F;
  char b = inputs >> 4;
  return a ^ b;
}
```

After

```c
char lut[256] =
{
  0x00, 0x01, 0x02, ..., 0x0F,
  0x01, 0x00, 0x03, ..., 0x0E,
  0x02, 0x03, 0x00, ..., 0x0D,
   |                      |
  0x0F, 0x0E, 0x0D, ..., 0x00
};

char xor(char inputs)
{
  return lut[inputs];
}
```

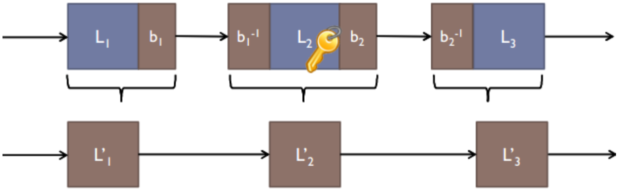
Some Details on WBC

Before

```c
char xor(char inputs)
{
  char a = inputs & 0x0F;
  char b = inputs >> 4;
  return a ^ b;
}
```

After

```c
char lut[256] =
{
  0x00, 0x01, 0x02, ..., 0x0F,
  0x01, 0x00, 0x03, ..., 0x0E,
  0x02, 0x03, 0x00, ..., 0x0D,
   |                      |
  0x0F, 0x0E, 0x0D, ..., 0x00
};

char xor(char inputs)
{
  return lut[inputs];
}
```

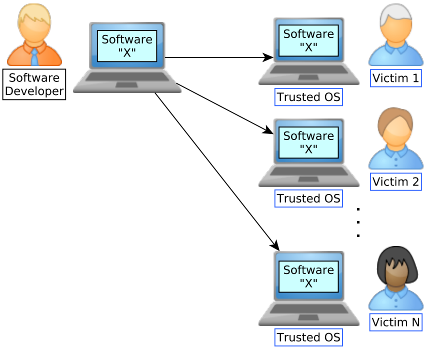


Source: PhD presentation of B. Wyseur

Outline

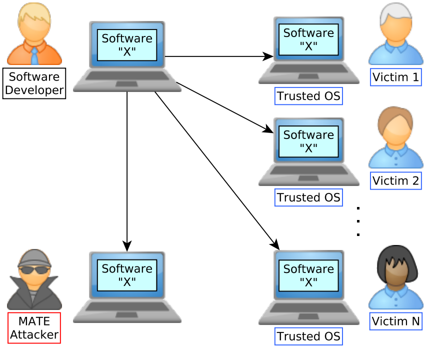
Software Diversity

1. Software developer distributes software X to all end-users



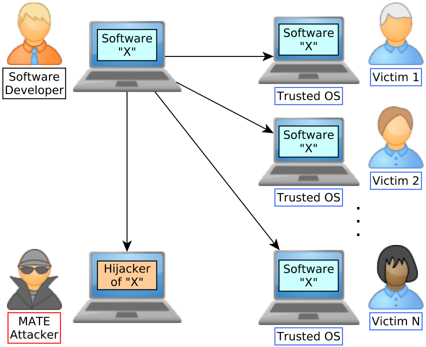
Software Diversity

1. Software developer distributes software X to all end-users
2. Some end-users are MATE attackers



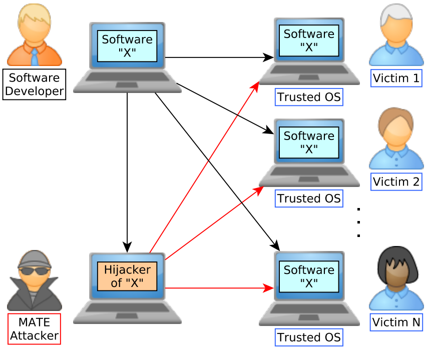
Software Diversity

1. Software developer distributes software X to all end-users
2. Some end-users are MATE attackers
3. MATE reverse engineers X and builds a hijacker of X



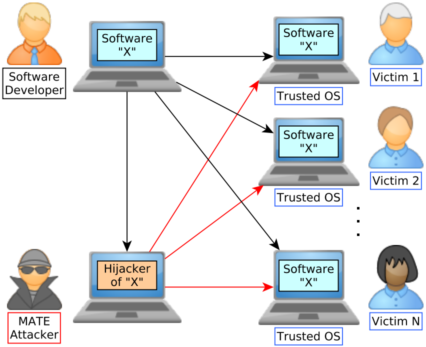
Software Diversity

1. Software developer distributes software X to all end-users
2. Some end-users are MATE attackers
3. MATE reverse engineers X and builds a hijacker of X
4. MATE distributes hijacker to other end-users of X



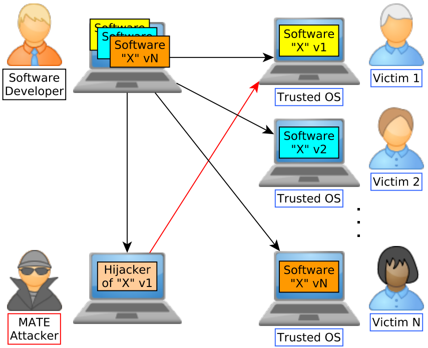
Software Diversity

- What can we do to protect victims?



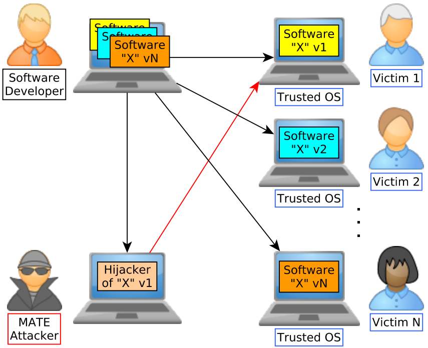
Software Diversity

- What can we do to protect victims?
- Give everyone a different version



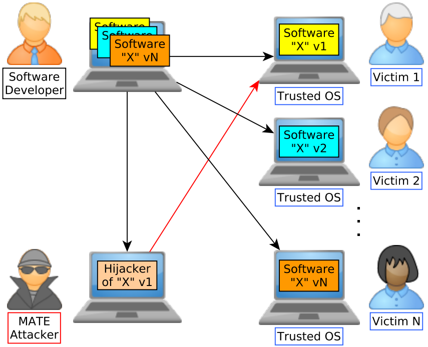
Software Diversity

- What can we do to protect victims?
- Give everyone a different version
- Generate 1000s of different versions using obfuscation



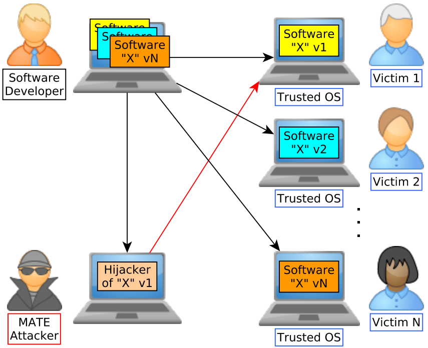
Software Diversity

- What can we do to protect victims?
- Give everyone a different version
- Generate 1000s of different versions using obfuscation
- **Assumption:** Same attack will not work on different binaries



Software Diversity

- What can we do to protect victims?
- Give everyone a different version
- Generate 1000s of different versions using obfuscation
- **Assumption:** Same attack will not work on different binaries
- **Issues:**
  - Analyzing crash-dumps
  - Incremental updates
  - Digitally signing all versions



Pre- vs. Post-Distribution Software Diversity

Pre-Distribution Software Diversity

- Obfuscation transformations involve randomly generated code & data
- Many different binaries generated by software developer
- Different binaries distributed to different end-users
- All previously presented techniques can be used for pre-distribution diversification

Pre- vs. Post-Distribution Software Diversity

Pre-Distribution Software Diversity

- Obfuscation transformations involve randomly generated code & data
- Many different binaries generated by software developer
- Different binaries distributed to different end-users
- All previously presented techniques can be used for pre-distribution diversification

Post-Distribution Software Diversity

- Software developer embeds self-modifying code in application
- All end-users may get the same binary from software developer
- Code may change differently for different users depending on inputs
- Next we present dynamic obfuscation techniques which can be used for post-distribution diversification

Outline

1 Introduction

2 Obfuscation in Theory

3 Obfuscation in Practice
- Static Obfuscation
  - Compiler Optimizations
  - Automated Code Obfuscation
- Software Diversity
- Dynamic Obfuscation

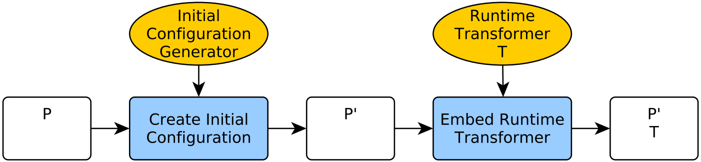
4 The Strength of Obfuscation

5 Hands-on Tutorial

6 Conclusions

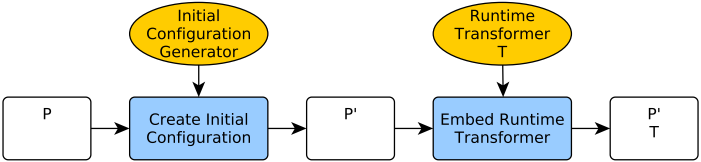
Dynamic Obfuscation

- A dynamic obfuscator runs in two phases:
  1. At **compile-time**:
     - Transform the program to an initial configuration
     - Add a runtime code-transformer

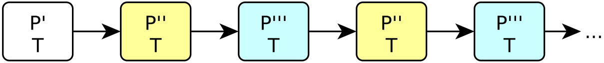


Dynamic Obfuscation

- A dynamic obfuscator runs in two phases:
  1. At **compile-time**:
     - Transform the program to an initial configuration
     - Add a runtime code-transformer



  2. At **runtime**:
     - Interleave execution of the program with calls to the transformer T
     - T changes the code segment content at runtime
     - Ideally a non-repeating series of configurations, **in practice** they repeat



Dynamic Obfuscation Techniques

General Techniques:

- **Build-and-execute**: generate code for a routine at runtime, and jump to it
- **Self-modification**: modify the executable code
- **Encryption**: the self-modification is decrypting the encrypted code before executing it
- **Move code**: every time the code executes, it is in a different location

Dynamic Obfuscation Techniques

General Techniques:

- **Build-and-execute**: generate code for a routine at runtime, and jump to it
- **Self-modification**: modify the executable code
- **Encryption**: the self-modification is decrypting the encrypted code before executing it
- **Move code**: every time the code executes, it is in a different location

These techniques can be applied at the following levels of **granularity**:

- File level
- Function level
- Basic block level
- Instruction level

Dynamic Obfuscation Techniques

General Techniques:

- **Build-and-execute**: generate code for a routine at runtime, and jump to it
- **Self-modification**: modify the executable code
- **Encryption**: the self-modification is decrypting the encrypted code before executing it
- **Move code**: every time the code executes, it is in a different location

These techniques can be applied at the following levels of **granularity**:

- File level
- Function level
- Basic block level
- Instruction level

The **attacker's goals** can be to:

- Recover the original code
- Modify the original code

Replacing instructions (Kanzaki [10])

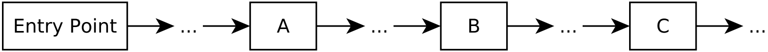
- **Motivation**: prevent code recovery via memory snapshot

Replacing instructions (Kanzaki [10])

- **Motivation**: prevent code recovery via memory snapshot
- **Algorithm idea**:
  1. Replace real instructions by bogus instructions
  2. Just before execution, replace bogus instruction with real instruction
  3. After execution, replace real instruction with bogus instruction

Replacing instructions (Kanzaki [10])

- **Motivation**: prevent code recovery via memory snapshot
- **Algorithm idea**:
  1. Replace real instructions by bogus instructions
  2. Just before execution, replace bogus instruction with real instruction
  3. After execution, replace real instruction with bogus instruction
- **Implementation** by choosing 3 points A, B and C in the CFG:
  1. All paths to B must flow through A and all paths from B must flow through C
  2. A replaces bogus instruction at B with real instruction
  3. C replaces real instruction at B with bogus instruction



Dynamic Code Merging (Madou [12])

- **Motivation**: keep code in constant flux

Dynamic Code Merging (Madou [12])

- **Motivation**: keep code in constant flux
- **Algorithm idea**:
  1. Have two or more **functions share the same location in memory**
  2. Create **templates for functions** that share same location
  3. Before a function is called, patch memory using **edit script** to load it

Dynamic Code Merging (Madou [12])

- **Motivation**: keep code in constant flux
- **Algorithm idea**:
  1. Have two or more **functions share the same location in memory**
  2. Create **templates for functions** that share same location
  3. Before a function is called, patch memory using **edit script** to load it
- **Implementation** example for program with 2 functions f_1 and f_2:
  1. Create template T with the same size as the largest function, i.e. f_1
  2. T contains values at memory offsets which are common for f_1 and f_2, i.e. offsets 1 and 2
  3. T contains wildcard values at all other memory offsets

f_1

memory offset:	0	1	2	3	4
memory value:	b7	48	a0	53	fa

f_2

memory offset:	0	1	2	3
memory value:	e9	48	a0	33

Dynamic Code Merging (Madou [12])

- **Implementation** example for program with 2 functions f_1 and f_2:
  1. Create template T with the same size as the largest function, i.e. f_1
  2. T contains values at memory offsets which are common for f_1 and f_2, i.e. offsets 1 and 2
  3. T contains wildcard values at all other memory offsets

f_1

memory offset:	0	1	2	3	4
memory value:	b7	48	a0	53	fa

f_2

	0	1	2	3
	e9	48	a0	33

T

memory offset:	0	1	2	3	4
memory value:	?	48	a0	?	?

Edit Scripts

$e_1 = [0 \to b7, 3 \to 53, 4 \to fa]$

$e_2 = [0 \to e9, 3 \to 33]$

Dynamic Code Merging (Madou [12])

- **Implementation** example for program with 2 functions f_1 and f_2:
  1. Create template T with the same size as the largest function, i.e. f_1
  2. T contains values at memory offsets which are common for f_1 and f_2, i.e. offsets 1 and 2
  3. T contains wildcard values at all other memory offsets
  4. Create **edit scripts** e_1 and e_2 which replace wildcards of T to load the functions f_1, respectively f_2

f_1

memory offset:	0	1	2	3	4
memory value:	b7	48	a0	53	fa

f_2

memory offset:	0	1	2	3
memory value:	e9	48	a0	33

T

memory offset:	0	1	2	3	4
memory value:	?	48	a0	?	?

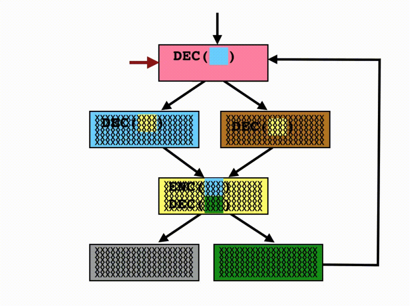
Edit Scripts

$e_1 = [0 \rightarrow b7, 3 \rightarrow 53, 4 \rightarrow fa]$

$e_2 = [0 \rightarrow e9, 3 \rightarrow 33]$

Dynamic Decryption and Re-encryption

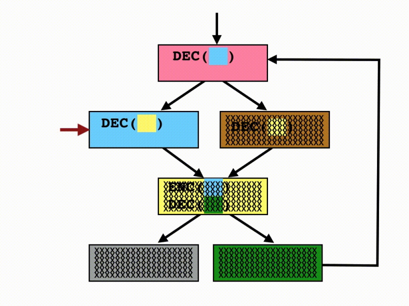
1. Execute current basic block
2. At some point in the current basic block decrypt the next basic block
3. Decryption key could be hash of some other basic block
4. Jump to decrypted basic block
5. Encrypt the previously executed basic block
6. Go to step 1



Source: `http://tigress.cs.arizona.edu/`

Dynamic Decryption and Re-encryption

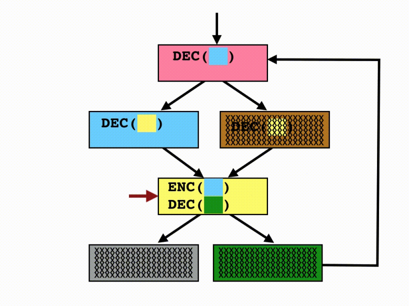
1. Execute current basic block
2. At some point in the current basic block decrypt the next basic block
3. Decryption key could be hash of some other basic block
4. Jump to decrypted basic block
5. Encrypt the previously executed basic block
6. Go to step 1



Source: `http://tigress.cs.arizona.edu/`

Dynamic Decryption and Re-encryption

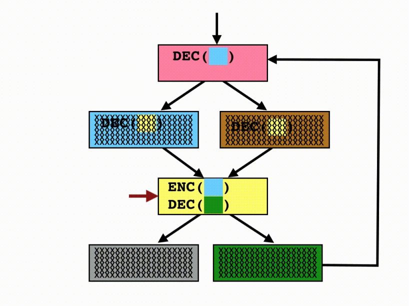
1. Execute current basic block
2. At some point in the current basic block decrypt the next basic block
3. Decryption key could be hash of some other basic block
4. Jump to decrypted basic block
5. Encrypt the previously executed basic block
6. Go to step 1



Source: `http://tigress.cs.arizona.edu/`

Dynamic Decryption and Re-encryption

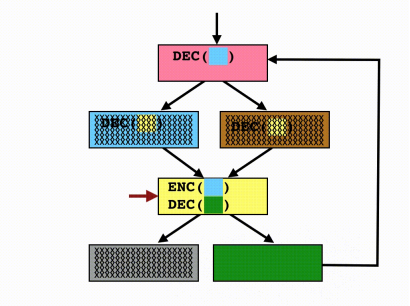
1. Execute current basic block
2. At some point in the current basic block decrypt the next basic block
3. Decryption key could be hash of some other basic block
4. Jump to decrypted basic block
5. Encrypt the previously executed basic block
6. Go to step 1



Source: `http://tigress.cs.arizona.edu/`

Dynamic Decryption and Re-encryption

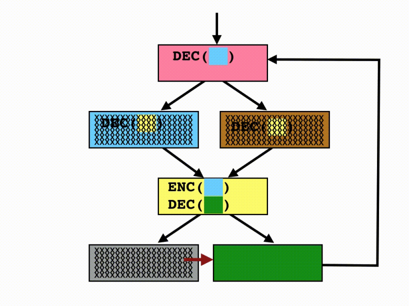
1. Execute current basic block
2. At some point in the current basic block decrypt the next basic block
3. Decryption key could be hash of some other basic block
4. Jump to decrypted basic block
5. Encrypt the previously executed basic block
6. Go to step 1



Source: `http://tigress.cs.arizona.edu/`

Dynamic Decryption and Re-encryption

1. Execute current basic block
2. At some point in the current basic block decrypt the next basic block
3. Decryption key could be hash of some other basic block
4. Jump to decrypted basic block
5. Encrypt the previously executed basic block
6. Go to step 1



Source: http://tigress.cs.arizona.edu/

Outline

Classical Obfuscation Strength Dimensions

According to Collberg et al. [5] we should keep the following 4 factors in mind when choosing obfuscation transformations:

Classical Obfuscation Strength Dimensions

According to Collberg et al. [5] we should keep the following 4 factors in mind when choosing obfuscation transformations:

- **Potency** $\rightarrow$ comprehensibility of code by humans
  - Time and skills needed to understand how the code works
  - Directly proportional with software complexity and size

Classical Obfuscation Strength Dimensions

According to Collberg et al. [5] we should keep the following 4 factors in mind when choosing obfuscation transformations:

- **Potency** $\rightarrow$ comprehensibility of code by humans
  - Time and skills needed to understand how the code works
  - Directly proportional with software complexity and size
- **Stealth** $\rightarrow$ identifiability of obfuscated code
  - Time and computation needed to identify the parts of the code which were obfuscated

Classical Obfuscation Strength Dimensions

According to Collberg et al. [5] we should keep the following 4 factors in mind when choosing obfuscation transformations:

- **Potency** $\rightarrow$ comprehensibility of code by humans
  - Time and skills needed to understand how the code works
  - Directly proportional with software complexity and size
- **Stealth** $\rightarrow$ identifiability of obfuscated code
  - Time and computation needed to identify the parts of the code which were obfuscated
- **Resilience** $\rightarrow$ resistance against automatic deobfuscation
  - Time and computation needed to deobfuscate the code with a software tool
  - Plus resources needed to build such a tool

Classical Obfuscation Strength Dimensions

According to Collberg et al. [5] we should keep the following 4 factors in mind when choosing obfuscation transformations:

- **Potency** → comprehensibility of code by humans
  - Time and skills needed to understand how the code works
  - Directly proportional with software complexity and size
- **Stealth** → identifiability of obfuscated code
  - Time and computation needed to identify the parts of the code which were obfuscated
- **Resilience** → resistance against automatic deobfuscation
  - Time and computation needed to deobfuscate the code with a software tool
  - Plus resources needed to build such a tool
- **Cost** → performance and resource overhead of obfuscation
  - Time overhead added to program execution after obfuscation
  - Size of the obfuscated binary relative to original binary

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software
  - What kind of attacker are you facing? (e.g. amateur hacker, professional organization, nation state)

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software
  - What kind of attacker are you facing? (e.g. amateur hacker, professional organization, nation state)
  - What is economically feasible for the attacker? (e.g. static attacks, dynamic attacks)

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software
  - What kind of attacker are you facing? (e.g. amateur hacker, professional organization, nation state)
  - What is economically feasible for the attacker? (e.g. static attacks, dynamic attacks)
  - What is the impact of protection on performance?

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software
  - What kind of attacker are you facing? (e.g. amateur hacker, professional organization, nation state)
  - What is economically feasible for the attacker? (e.g. static attacks, dynamic attacks)
  - What is the impact of protection on performance?
- **Example Scenario:** Protecting a hard-coded password / license key
  - Software: offline game / word processor, costs 50 USD

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software
  - What kind of attacker are you facing? (e.g. amateur hacker, professional organization, nation state)
  - What is economically feasible for the attacker? (e.g. static attacks, dynamic attacks)
  - What is the impact of protection on performance?
- **Example Scenario:** Protecting a hard-coded password / license key
  - Software: offline game / word processor, costs 50 USD
    - → **cost** (obfuscation overhead) $<$ 10% performance impact
    - → **stealth** less important for this scenario

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software
  - What kind of attacker are you facing? (e.g. amateur hacker, professional organization, nation state)
  - What is economically feasible for the attacker? (e.g. static attacks, dynamic attacks)
  - What is the impact of protection on performance?
- **Example Scenario:** Protecting a hard-coded password / license key
  - Software: offline game / word processor, costs 50 USD
    - $\rightarrow$ **cost** (obfuscation overhead) $<$ 10% performance impact
    - $\rightarrow$ **stealth** less important for this scenario
  - Each user needs a different password / license key
    - $\rightarrow$ **potency** $>$ 50 USD (exclude: professional org., nation state)

The Strength of Obfuscation

- **Question:** Which obfuscation transformation is better?
- **Answer:** It depends ...
  - What do you want to protect?:
    - intellectual property
    - keys
    - integrity of software
  - What kind of attacker are you facing? (e.g. amateur hacker, professional organization, nation state)
  - What is economically feasible for the attacker? (e.g. static attacks, dynamic attacks)
  - What is the impact of protection on performance?
- **Example Scenario:** Protecting a hard-coded password / license key
  - Software: offline game / word processor, costs 50 USD
    - $\rightarrow$ **cost** (obfuscation overhead) $<$ 10% performance impact
    - $\rightarrow$ **stealth** less important for this scenario
  - Each user needs a different password / license key
    - $\rightarrow$ **potency** $>$ 50 USD (exclude: professional org., nation state)
  - Attacker develops automated crack based on symbolic execution
    - $\rightarrow$ **resilience** $>$ 50 USD $\times$ Nr. of users not willing to pay

Symbolic Execution in a Nutshell

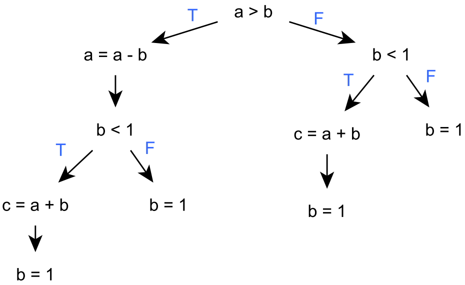
- Interpret program using symbolic values instead of concrete ones

```c
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```

Symbolic Execution in a Nutshell

- Interpret program using symbolic values instead of concrete ones
- Fork execution on each branch dependent on symbolic values

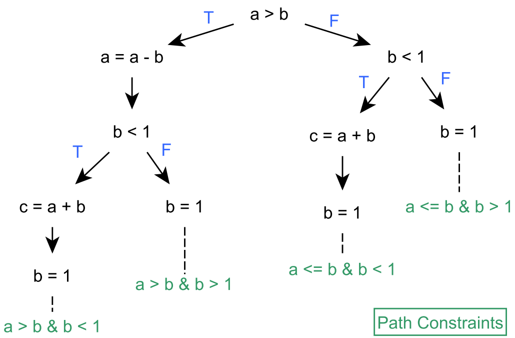
```c
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```



Symbolic Execution in a Nutshell

- Interpret program using symbolic values instead of concrete ones
- Fork execution on each branch dependent on symbolic values
- Collect *path constraints* for each execution path

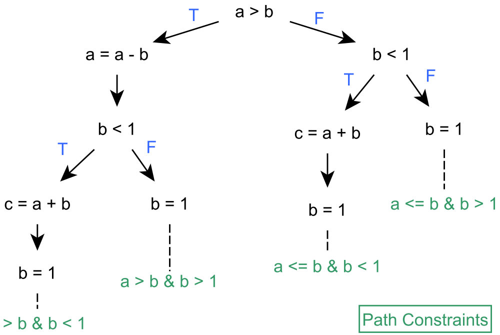
```c
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```



Symbolic Execution in a Nutshell

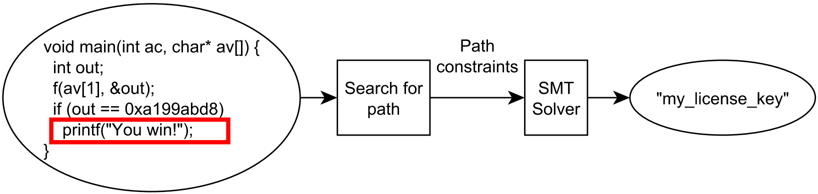
- Interpret program using symbolic values instead of concrete ones
- Fork execution on each branch dependent on symbolic values
- Collect *path constraints* for each execution path
- Get concrete input values from path conditions using SMT solver

```c
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```



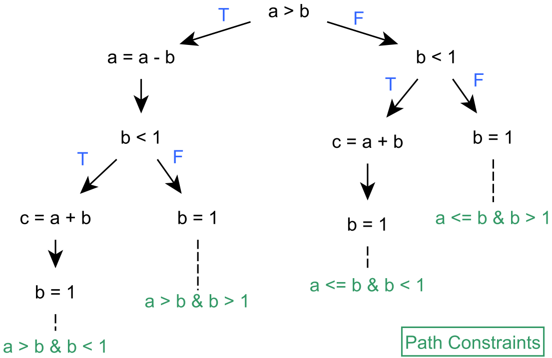
Bypassing License Checks via Symbolic Execution

1. Make license input symbolic
2. Indicate distinct statement executed when license key is correct
3. Explore paths until desired instruction (sequence) is found
4. Solve path constraints on paths that lead to desired instruction via SMT solver
5. Find satisfiable path constraints → concrete inputs to bypass check



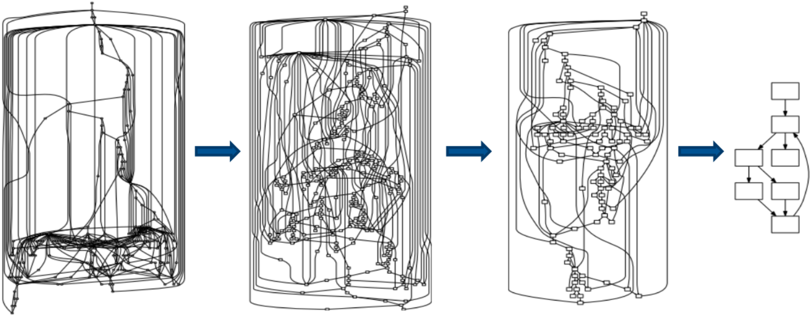
Are other Attacks Possible via Symbolic Execution?

- Yes! Symbolic execution is a prerequisite for several attacks:
  - Simplify control-flow graph
  - Identify & disable tamper-proofing checks
  - Bypass authentication checks / trigger conditions



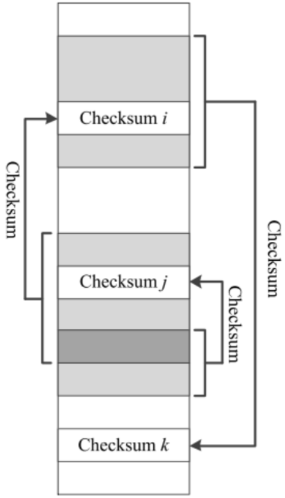
Simplifying the CFG (Yadegari et al. [14])

1. Explore paths such that all code is covered
2. Simplify traces using compiler optimization tricks
3. Reconstruct CFG from traces



Identify & Disable Checks (Qiu et al. [13])

1. Taint code segment
2. Explore paths until enough self-checks disabled (cyclic checks → explore all code)
3. Disable self-checking instructions

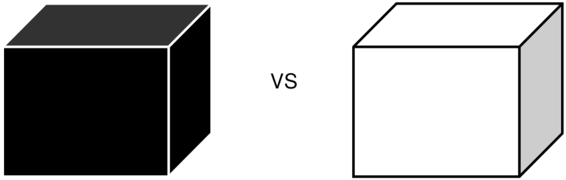


A Common Sub-Problem of Deobfuscation Attacks

- **Common sub-problem:** path exploration
- How do we explore paths of a given program?

A Common Sub-Problem of Deobfuscation Attacks

- **Common sub-problem:** path exploration
- How do we explore paths of a given program?
- Generate test cases:
  - **Black-box test generation:** Fuzzing, Random testing
  - **White-box test generation:** Symbolic/Concolic execution



Measuring Obfuscation Strength

- **Strength of obfuscation:** increase in test case generation time

Measuring Obfuscation Strength

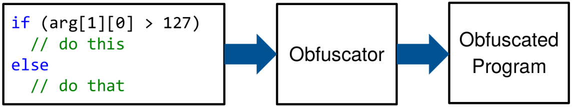
- **Strength of obfuscation:** increase in test case generation time
- **Observation:** Generally, obfuscation does not change input-output behavior

  → No increase in black-box test case generation time

Measuring Obfuscation Strength

- **Strength of obfuscation:** increase in test case generation time
- **Observation:** Generally, obfuscation does not change input-output behavior

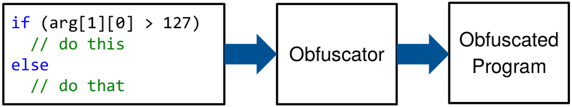
  → No increase in black-box test case generation time
- **Example:**



Measuring Obfuscation Strength

- **Strength of obfuscation:** increase in test case generation time
- **Observation:** Generally, obfuscation does not change input-output behavior

  $\rightarrow$ No increase in black-box test case generation time
- **Example:**

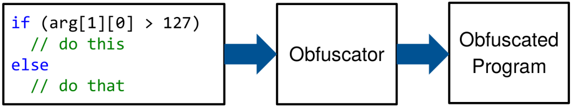


- **Observation:** Could be faster to use black-box test generator than white-box

Measuring Obfuscation Strength

- **Strength of obfuscation:** increase in test case generation time
- **Observation:** Generally, obfuscation does not change input-output behavior

  → No increase in black-box test case generation time
- **Example:**



- **Observation:** Could be faster to use black-box test generator than white-box
- **Conclusion:** Apply obfuscation transformations until white-box slower than black-box test case generation

Code Tampering Attacks

- **Question:** Why do we need code obfuscation? Just use cryptographic hash

Code Tampering Attacks

- **Question:** Why do we need code obfuscation? Just use cryptographic hash
- **Example:**

```
if (SHA256(arg[1]) == 0xa49…3793)
  // do this
else
  // do that
```

Code Tampering Attacks

- **Question:** Why do we need code obfuscation? Just use cryptographic hash
- **Example:**

```
if (SHA256(arg[1]) == 0xa49…3793)
  // do this
else
  // do that
```

- Hard for symbolic execution (SMT solver) to break crypto hash functions

Code Tampering Attacks

- **Question:** Why do we need code obfuscation? Just use cryptographic hash
- **Example:**

```
if (SHA256(arg[1]) == 0xa49…3793)
  // do this
else
  // do that
```

- Hard for symbolic execution (SMT solver) to break crypto hash functions
- **Answer:**
  - Test case generation is non-invasive attack, i.e. code is read, not changed
  - Obfuscation aims to defend against MATE attacker (can tamper with code)
  - Easy to find and patch-out crypto hash functions

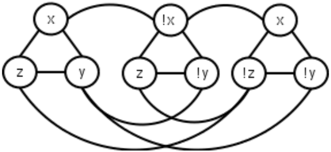
Metrics for Measuring Obfuscation Strength

- Number of lines of code
- McCabe cyclomatic complexity
- Nesting complexity
- Data flow complexity
- Object oriented design metrics
- Data structure complexity
- ...

Metrics for Measuring Obfuscation Strength

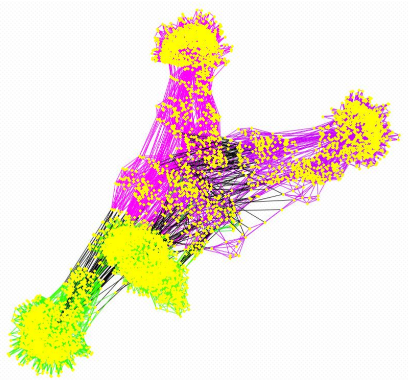
- Number of lines of code
- McCabe cyclomatic complexity
- Nesting complexity
- Data flow complexity
- Object oriented design metrics
- Data structure complexity
- ...
- SAT metrics: Graph metrics on a SAT formula represented as a graph

  $(x+y+z)\cdot(!x+!y+z)\cdot(x+!y+!z)$



SAT Instance Before & After Obfuscation

Figure : Before Obfuscation (7.5 sec)



```c
unsigned int SDBMHash(char* str, unsigned int len)
{
   unsigned int hash = 0;
   unsigned int i    = 0;
   for(i = 0; i < len; str++, i++)
      hash = (*str) + (hash << 6)
             + (hash << 16) - hash;
   return hash;
}

int main(int argc, char* argv[]) {
  unsigned char *str = argv[1];
  unsigned int hash = SDBMHash(str, strlen(str));

  if (hash == 0x89dcd66e)
    printf("You win!\n");
  return 0;
}
```

SAT Instance Before & After Obfuscation

Figure : Before Obfuscation (7.5 sec)

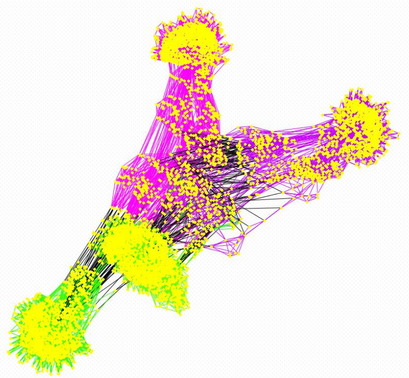
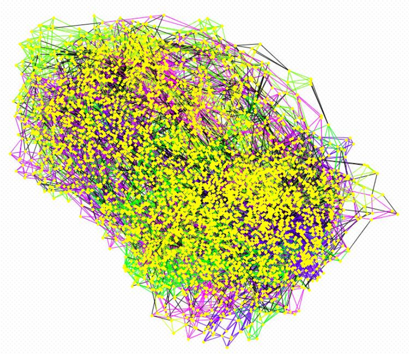


Figure : After Obfuscation (438 sec)



Strong obfuscation transformations destroy community structures

Outline

Hands-on Tutorial Overview

Example of bypassing password check in C programs

1. Start with plain unobfuscated program (break with static analysis)
2. Discuss the option of using hash functions (break with patching)
3. Add one obfuscation layer (break with symbolic execution)
4. Add more obfuscation layers (harder to break)

Hands-on Tutorial Overview

Example of bypassing password check in C programs

1. Start with plain unobfuscated program (break with static analysis)
2. Discuss the option of using hash functions (break with patching)
3. Add one obfuscation layer (break with symbolic execution)
4. Add more obfuscation layers (harder to break)

- Tools we are going to use:
  - We assume you have Docker installed and have basic user knowledge
  - Docker image based on Ubuntu contains: Tigress, KLEE, STP, Z3, SatGraf, etc.

    ```
    $ docker pull banescusebi/obfuscation-symex
    $ docker run -it banescusebi/obfuscation-symex
    ```
  - For instructions on how to start the Docker image read description at `https://hub.docker.com/r/banescusebi/obfuscation-symex/`
  - Instructions for running GUI apps available for Ubuntu and Mac OS (not mandatory, only needed for SatGraf)

Bypassing Password Check in Unobfuscated Program

Example C Program with Hard-Coded Password

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
  if (strcmp(argv[1],
      "my_license_key") == 0)
    printf("You win!\n");
  return 0;
}
```

- Hard-coded password can be found via static analysis

```
$ ./nohash my_license_key
You win!
$ strings nohash | grep ''license''
my_license_key
```

Bypassing Password Check even when Using Hash

Example C Program where Hard-Coded Password Replaced with Hash

```c
unsigned int BPHash(char* str, unsigned int len)
{
  unsigned int hash = 0;
  unsigned int i    = 0;
  for(i = 0; i < len; str++, i++) {
    hash = hash << 7 ^ (*str);
  }
  return hash;
}

int main(int argc, char* argv[]) {
  unsigned char *str = argv[1];
  unsigned int hash = BPHash(str, strlen(str));
  if (hash == 0x5bfaf2f9)
    printf("You win!\n");
  return 0;
}
```

- Check can still be disabled using binary patching

$ `strings bphash | grep ''license''`

$ `objdump -D bphash`

- Use vi in hex editor mode (`:%!xxd`) to patch check (`jne`)

Bypassing Password Check after Obfuscation

- **Add one layer of obfuscation:**

```
$ tigress --Transform=EncodeArithmetic
  --Functions=DEKHash --out=dekhash-obf/dekhash-encA.c
  dekhash.c
$ tigress --Transform=Flatten --Functions=DEKHash
  --out=dekhash-obf/dekhash-flat.c dekhash.c
$ tigress --Transform=Virtualize --Functions=DEKHash
  --out=dekhash-obf/dekhash-virt.c dekhash.c
```

Bypassing Password Check after Obfuscation

- **Add one layer of obfuscation:**

```
$ tigress --Transform=EncodeArithmetic
  --Functions=DEKHash --out=dekhash-obf/dekhash-encA.c
  dekhash.c
```

```
$ tigress --Transform=Flatten --Functions=DEKHash
  --out=dekhash-obf/dekhash-flat.c dekhash.c
```

```
$ tigress --Transform=Virtualize --Functions=DEKHash
  --out=dekhash-obf/dekhash-virt.c dekhash.c
```

- **Add another layer of obfuscation:**

```
$ tigress --Transform=Flatten --Functions=DEKHash
  --Transform=EncodeArithmetic --Functions=DEKHash
  --out=dekhash-obf/dekhash-flat-encA.c dekhash.c
```

Bypassing Password Check after Obfuscation

- **Add one layer of obfuscation:**

```
$ tigress --Transform=EncodeArithmetic
  --Functions=DEKHash --out=dekhash-obf/dekhash-encA.c
  dekhash.c
$ tigress --Transform=Flatten --Functions=DEKHash
  --out=dekhash-obf/dekhash-flat.c dekhash.c
$ tigress --Transform=Virtualize --Functions=DEKHash
  --out=dekhash-obf/dekhash-virt.c dekhash.c
```

- **Add another layer of obfuscation:**

```
$ tigress --Transform=Flatten --Functions=DEKHash
  --Transform=EncodeArithmetic --Functions=DEKHash
  --out=dekhash-obf/dekhash-flat-encA.c dekhash.c
```

- **Apply symbolic execution** to each of these programs (see KLEE command on next slide)

```
$ ~/scripts/step05-klee-symex-until-win.sh . exe 15
```

The KLEE Symbolic Execution Tool

```
klee --optimize
  // Optimize before execution
  --emit-all-errors
  // Don't stop on first error
  --libc=uclibc
  // Choose libc version
  --posix-runtime
  // Link with POSIX runtime
  --only-output-states-covering-new
  // Only tests covering new code
  --max-time=3600
  // Halt after given nr. of sec.
  --write-smt2s
  // Write smt2 file per test
  --output-dir=klee-out-${file_name}
  // Output directory for tests
  ./${file_name}.bc
  // Bitcode file under test
  --sym-arg 5
  // Length of symbolic arg.
```

Viewing Results

- **See number of nanoseconds needed for KLEE to analyze a bitcode program**

```
$ cat klee-time-to-win.txt
```

- **Check differences between 1 and 2 layers of obfuscation**

Viewing Results

- **See number of nanoseconds needed for KLEE to analyze a bitcode program**

$ `cat klee-time-to-win.txt`

- **Check differences between 1 and 2 layers of obfuscation**
- **Convert the SMT2 instances generated by KLEE into CNF instances**

$ `~/scripts/step07-convert-smt-to-cnf.sh . obfuscated-cnf-instances.txt`

$ `ll ./obfuscated-cnf-instances`

Viewing Results

- **See number of nanoseconds needed for KLEE to analyze a bitcode program**

$ `cat klee-time-to-win.txt`

- **Check differences between 1 and 2 layers of obfuscation**
- **Convert the SMT2 instances generated by KLEE into CNF instances**

$ `~/scripts/step07-convert-smt-to-cnf.sh . obfuscated-cnf-instances.txt`

$ `ll ./obfuscated-cnf-instances`

- **View CNF instances in SatGraf:**

$ `java -jar ~/satgraf/dist/SatGraf.jar com -f obfuscated-cnf-instances/dekhash-virt.cnf`

- **Check differences between 1 and 2 layers of obfuscation**

Outline

Conclusions

- Theoretical obfuscation offers security guarantees, but is unpractical

Conclusions

- Theoretical obfuscation offers security guarantees, but is unpractical
- Practical obfuscation raises the bar against MATE attackers

Conclusions

- Theoretical obfuscation offers security guarantees, but is unpractical
- Practical obfuscation raises the bar against MATE attackers
- Static obfuscation transformations:
  - Applied on source code, IR or machine code
  - Obfuscated program fixed (doesn't change at runtime)
  - Makes static analysis harder
  - Many transformations available: compiler optimizations, scramble identifiers, instruction substitution, merging & splitting functions, opaque predicates & expressions, control flow flattening, virtualization obfuscation, white-box cryptography, etc.

Conclusions

- Theoretical obfuscation offers security guarantees, but is unpractical
- Practical obfuscation raises the bar against MATE attackers
- Static obfuscation transformations:
  - Applied on source code, IR or machine code
  - Obfuscated program fixed (doesn't change at runtime)
  - Makes static analysis harder
  - Many transformations available: compiler optimizations, scramble identifiers, instruction substitution, merging & splitting functions, opaque predicates & expressions, control flow flattening, virtualization obfuscation, white-box cryptography, etc.
- Software diversity improves potency and resilience against reverse engineering attacks

Conclusions

- Theoretical obfuscation offers security guarantees, but is unpractical
- Practical obfuscation raises the bar against MATE attackers
- Static obfuscation transformations:
  - Applied on source code, IR or machine code
  - Obfuscated program fixed (doesn't change at runtime)
  - Makes static analysis harder
  - Many transformations available: compiler optimizations, scramble identifiers, instruction substitution, merging & splitting functions, opaque predicates & expressions, control flow flattening, virtualization obfuscation, white-box cryptography, etc.
- Software diversity improves potency and resilience against reverse engineering attacks
- Dynamic obfuscation transformations:
  - Program changes during execution
  - Makes dynamic analysis harder
  - Several transformations available: replacing instructions, dynamic code merging, dynamic decryption and re-encryption.

Conclusions

- Theoretical obfuscation offers security guarantees, but is unpractical
- Practical obfuscation raises the bar against MATE attackers
- Static obfuscation transformations:
  - Applied on source code, IR or machine code
  - Obfuscated program fixed (doesn't change at runtime)
  - Makes static analysis harder
  - Many transformations available: compiler optimizations, scramble identifiers, instruction substitution, merging & splitting functions, opaque predicates & expressions, control flow flattening, virtualization obfuscation, white-box cryptography, etc.
- Software diversity improves potency and resilience against reverse engineering attacks
- Dynamic obfuscation transformations:
  - Program changes during execution
  - Makes dynamic analysis harder
  - Several transformations available: replacing instructions, dynamic code merging, dynamic decryption and re-encryption.
- Multiple transformations should be combined to improve strength

Thank you for your attention

Questions ?

References I

S. Banescu, M. Ochoa, A. Pretschner, and N. Kunze.
Benchmarking indistinguishability obfuscation - a candidate implementation.
In *Proc. of 7th International Symposium on ESSoS*, number 8978 in LNCS, 2015.

Boaz Barak.
Hopes, fears, and software obfuscation.
Communications of the ACM, 59(3):88–96, 2016.

Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang.
On the (im) possibility of obfuscating programs.
In *Advances in Cryptology CRYPTO 2001*, pages 1–18. Springer, 2001.

References II

Christian Collberg and Jasvir Nagra.
Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.
Addison-Wesley, 2009.

Christian Collberg, Clark Thomborson, and Douglas Low.
A taxonomy of obfuscating transformations.
Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

Christian Collberg, Clark Thomborson, and Douglas Low.
Manufacturing cheap, resilient, and stealthy opaque constructs.
In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

References III

M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi.
Opaque predicates detection by abstract interpretation.
In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology*, volume 4019 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin Heidelberg, 2006.

S. Garg, C. Gentry, S. Halevi, M. Raykova, A Sahai, and B. Waters.
Candidate indistinguishability obfuscation and functional encryption for all circuits.
In *Proc. of the 54th Annual Symp. on Foundations of Computer Science*, pages 40–49, 2013.

S. Goldwasser and G. N. Rothblum.
On best-possible obfuscation.
In *Theory of Cryptography*, pages 194–213. Springer, 2007.

References IV

Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto.
Exploiting self-modification mechanism for program protection.
In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 170–179, 2003.

J. Kinder.
Towards static analysis of virtualization-obfuscated binaries.
In *19th Working Conference on Reverse Engineering (WCRE)*, pages 61–70, Oct 2012.

Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere.
Software protection through dynamic code mutation.
In *Information Security Applications*, pages 194–206. Springer, 2006.

References V

Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su.
Identifying and understanding self-checksumming defenses in software.
In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218. ACM, 2015.

Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray.
A generic approach to automatic deobfuscation of executable code.
In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 674–691. IEEE, 2015.