ISSISP 2014
Code Obfuscation

Verona, Italy

Christian Collberg
University of Arizona
www.cs.arizona.edu/~collberg

© July 27, 2014 Christian Collberg

Overview

Code obfuscation — what is it?

- 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.

Code obfuscation — what is it?

- 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?" $\Rightarrow$ Harder than before!

Code obfuscation — what is it?

- static obfuscation ⇒ obfuscated programs that remain fixed at runtime.
  - tries to thwart static analysis
  - attacked by dynamic techniques (debugging, emulation, tracing).

Code obfuscation — what is it?

- static obfuscation ⇒ obfuscated programs that remain fixed at runtime.
  - tries to thwart static analysis
  - attacked by dynamic techniques (debugging, emulation, tracing).
- dynamic obfuscators ⇒ transform programs continuously at runtime, keeping them in constant flux.
  - tries to thwart dynamic analysis

Bogus Control Flow

Complicating control flow

- Transformations that make it difficult for an adversary to analyze the flow-of-control:
  1. insert bogus control-flow

Complicating control flow

- Transformations that make it difficult for an adversary to analyze the flow-of-control:
  1. insert bogus control-flow
  2. flatten the program

Complicating control flow

- Transformations that make it difficult for an adversary to analyze the flow-of-control:
    1. insert bogus control-flow
    2. flatten the program
    3. hide the targets of branches to make it difficult for the adversary to build control-flow graphs

Complicating control flow

- Transformations that make it difficult for an adversary to analyze the flow-of-control:
  1. insert bogus control-flow
  2. flatten the program
  3. hide the targets of branches to make it difficult for the adversary to build control-flow graphs
- None of these transformations are immune to attacks

Opaque Expressions

- Simply put:

  an expression whose value is known to you as the defender (at obfuscation time) but which is difficult for an attacker to figure out

Opaque Expressions

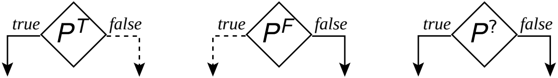
- Simply put:

  > *an expression whose value is known to you as the defender (at obfuscation time) but which is difficult for an attacker to figure out*

- Notation:
  - P^T for an *opaquely true* predicate
  - P^F for an *opaquely false* predicate
  - $P^?$ for an *opaquely indeterminate* predicate
  - $E^{=v}$ for an *opaque* expression of value v

Opaque Expressions

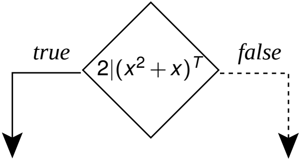
- Graphical notation:



- Building blocks for many obfuscations.

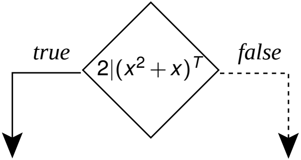
Opaque Expressions

- An opaquely true predicate:

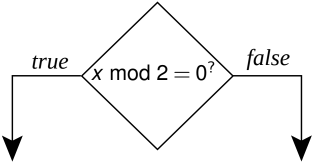


Opaque Expressions

- An opaquely true predicate:



- An opaquely indeterminate predicate:



Inserting bogus control-flow

- Insert *bogus* control-flow into a function:
    1. dead branches which will never be taken

Inserting bogus control-flow

- Insert *bogus* control-flow into a function:
  1. dead branches which will never be taken
  2. superfluous branches which will *always* be taken

Inserting bogus control-flow

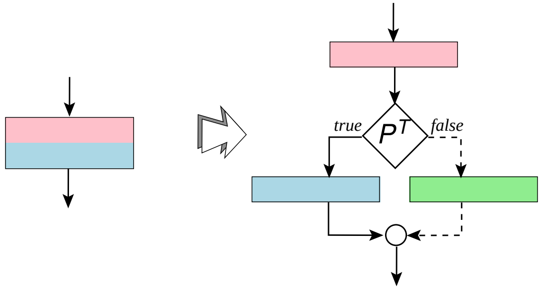
- Insert *bogus* control-flow into a function:
  1. dead branches which will never be taken
  2. superfluous branches which will *always* be taken
  3. branches which will sometimes be taken and sometimes not, but where this doesn't matter

Inserting bogus control-flow

- Insert *bogus* control-flow into a function:
  1. dead branches which will never be taken
  2. superfluous branches which will *always* be taken
  3. branches which will sometimes be taken and sometimes not, but where this doesn’t matter
- The resilience reduces to the resilience of the opaque predicates.

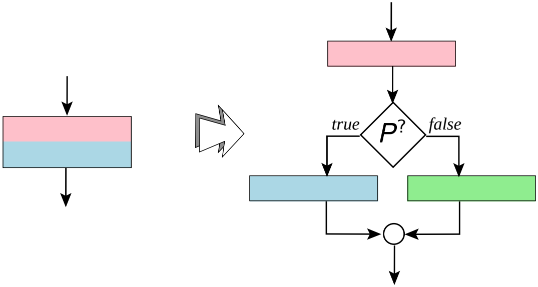
Inserting bogus control-flow

- A bogus block (green) appears as it might be executed while, in fact, it never will:



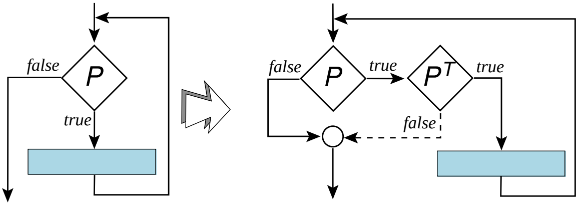
Inserting bogus control-flow

- Sometimes execute the blue block, sometimes the green block.
- The green and blue blocks should be semantically equivalent.



Inserting bogus control-flow

- Extend a loop condition *P* by conjoining it with an opaquely true predicate P^T:



Control Flow Flattening

Control-flow flattening

- Removes the control-flow *structure* of functions.

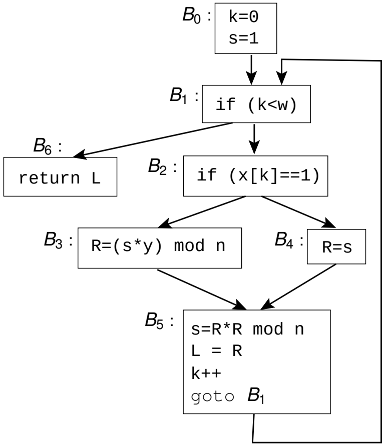
Control-flow flattening

- Removes the control-flow *structure* of functions.
- Put each basic block as a case inside a switch statement, and wrap the switch inside an infinite loop.

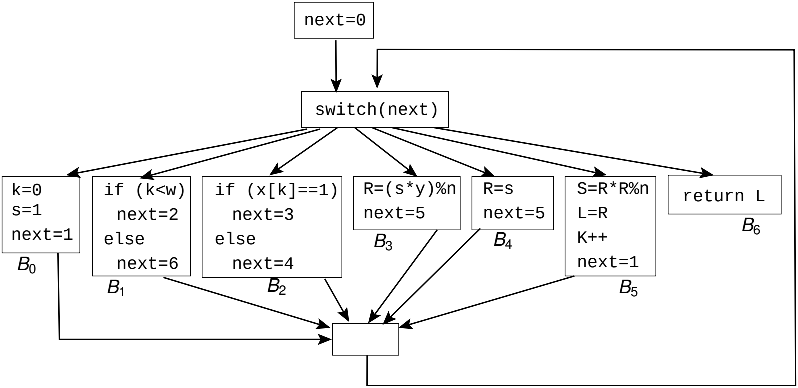
Control-flow flattening

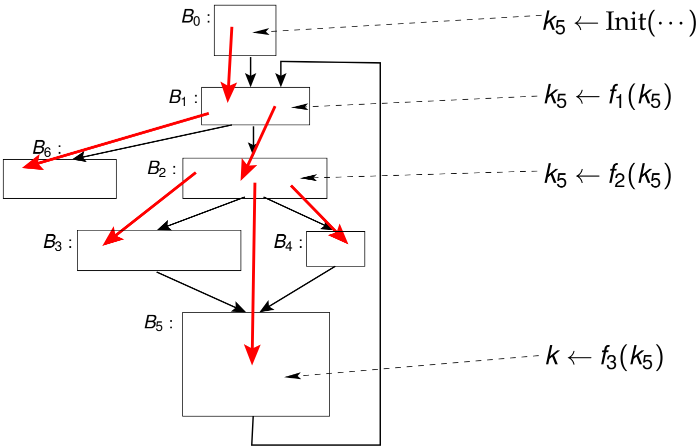
- Removes the control-flow *structure* of functions.
- Put each basic block as a case inside a switch statement, and wrap the switch inside an infinite loop.
- Chenxi Wang's PhD thesis:

```
int modexp(int y,int x[],
           int w,int n) {
   int R, L;
   int k = 0;
   int s = 1;
   while (k < w) {
      if (x[k] == 1)
         R = (s*y) % n;
      else
         R = s;
      s = R*R % n;
      L = R;
      k++;
   }
   return L;
}
```



```
int modexp(int y, int x[], int w, int n) {
   int R, L, k, s;
   int next=0;
   for(;;)
      switch(next) {
         case 0 : k=0; s=1; next=1; break;
         case 1 : if (k<w) next=2; else next=6; break;
         case 2 : if (x[k]==1) next=3; else next=4; brea
         case 3 : R=(s*y)%n; next=5; break;
         case 4 : R=s; next=5; break;
         case 5 : s=R*R%n; L=R; k++; next=1; break;
         case 6 : return L;
      }
}
```


next=0
switch(next)
k=0
s=1
next=1
B_0
if (k<w)
next=2
else
next=6
B_1
if (x[k]==1)
next=3
else
next=4
B_2
R=(s*y)%n
next=5
B_3
R=s
next=5
B_4
S=R*R%n
L=R
K++
next=1
B_5
return L
B_6



- Red lines form the dominator tree.
- We insert functions $\mathrm{Init}, f_1, f_2, f_3$ that, when B_5 is reached must have executed, and the new value for k has been evolved.

Performance penalty

- Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2.

Performance penalty

- Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2.
- Why?
    1. The for loop incurs one jump,

Performance penalty

- Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2.
- Why?
  1. The for loop incurs one jump,
  2. the switch incurs a bounds check the `next` variable,

Performance penalty

- Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2.
- Why?
    1. The for loop incurs one jump,
    2. the switch incurs a bounds check the `next` variable,
    3. the switch incurs an indirect jump through a jump table.

Performance penalty

- Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2.
- Why?
    1. The for loop incurs one jump,
    2. the switch incurs a bounds check the `next` variable,
    3. the switch incurs an indirect jump through a jump table.
- Optimize?

Performance penalty

- Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2.
- Why?
  1. The for loop incurs one jump,
  2. the switch incurs a bounds check the `next` variable,
  3. the switch incurs an indirect jump through a jump table.
- Optimize?
  1. Keep tight loops as one switch entry.

Performance penalty

- Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2.
- Why?
  1. The for loop incurs one jump,
  2. the switch incurs a bounds check the `next` variable,
  3. the switch incurs an indirect jump through a jump table.
- Optimize?
  1. Keep tight loops as one switch entry.
  2. Use `gcc`'s labels-as-values ⇒ a jump table lets you jump directly to the next basic block.

Attack against Control-flow flattening

- Attack:
    1. Work out what the next block of every block is.

Attack against Control-flow flattening

- Attack:
  1. Work out what the next block of every block is.
  2. Rebuild the original CFG!

Attack against Control-flow flattening

- Attack:
    1. Work out what the next block of every block is.
    2. Rebuild the original CFG!
- How does an attacker do this?
    1. use-def data-flow analysis

Attack against Control-flow flattening

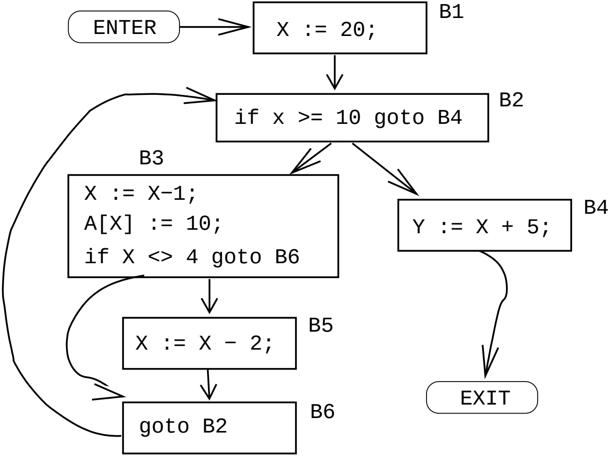
- Attack:
  1. Work out what the next block of every block is.
  2. Rebuild the original CFG!
- How does an attacker do this?
  1. use-def data-flow analysis
  2. constant-propagation data-flow analysis

next as an opaque predicate!

```
int modexp(int y, int x[], int w, int n) {
   int R, L, k, s;
   int next=$E^{=0}$;
   for(;;)
      switch(next) {
         case 0 : k=0; s=1; next=$E^{=1}$; break;
         case 1 : if (k<w) next=$E^{=2}$; else next=$E^{=6}$; brea
         case 2 : if (x[k]==1) next=$E^{=3}$; else next=$E^{=4}$;
                  break;
         case 3 : R=(s*y)%n; next=$E^{=5}$; break;
         case 4 : R=s; next=$E^{=5}$; break;
         case 5 : s=R*R%n; L=R; k++; next=$E^{=1}$; break;
         case 6 : return L;
      }
}
```

In-Class Exercise

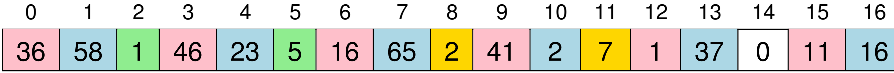
1. Flatten this CFG:



2. Give the source code for the flattened graph above

Constructing Opaque Predicates

Opaque values from array aliasing



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
36	58	1	46	23	5	16	65	2	41	2	7	1	37	0	11	16

Invariants:

1. every third cell (in pink), starting will cell 0, is $\equiv 1 \bmod 5$;
2. cells 2 and 5 (green) hold the values 1 and 5, respectively;
3. every third cell (in blue), starting will cell 1, is $\equiv 2 \bmod 7$;
4. cells 8 and 11 (yellow) hold the values 2 and 7, respectively.

Opaque values from array aliasing

- You can update a pink element as often as you want, with any value you want, as long as you ensure that the value is always $\equiv 1 \bmod 5$!
- That is, make any changes you want, while maintaining the invariant.
- This will make static analysis harder for the attacker.

```
int g[] = {36,58,1,46,23,5,16,65,2,41,
           2,7,1,37,0,11,16,2,21,16};

if ((g[3] % g[5])==g[2])
   printf("true!\n");

g[5] = (g[1]*g[4])%g[11] + g[6]%g[5];
g[14] = rand();
g[4] = rand()*g[11]+g[8];

int six = (g[4] + g[7] + g[10])%g[11];
int seven = six + g[3]%g[5];
int fortytwo = six * seven;
```

- pink: opaquely true predicate.
- blue: `g` is constantly changing at runtime.
- green: an opaque value 42.

Initialize `g` at runtime!

```
int modexp(int y, int x[], int w, int n) {
   int R, L, k, s;
   int next=0;
   int g[] = {10,9,2,5,3};
   for(;;)
      switch(next) {
         case 0 : k=0; s=1; next=g[0]%g[1]=1; break;
         case 1 : if (k<w) next=g[g[2]]=2;
                  else next=g[0]-2*g[2]=6; break;
         case 2 : if (x[k]==1) next=g[3]-g[2]=3;
                  else next=2*g[2]=4; break;
         case 3 : R=(s*y)%n; next=g[4]+g[2]=5; break;
         case 4 : R=s; next=g[0]-g[3]=5; break;
         case 5 : s=R*R%n; L=R; k++; next=g[g[4]]%g[2]=1
                  break;
         case 6 : return L;
      }
}
```

Opaque predicates from pointer aliasing

- Create an obfuscating transformation from a known computationally hard static analysis problem.

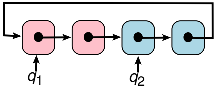
Opaque predicates from pointer aliasing

- Create an obfuscating transformation from a known computationally hard static analysis problem.
- We assume that
  1. the attacker will analyze the program statically, and
  2. we can force him to solve a particular static analysis problem to discover the secret he's after, and
  3. we can generate an actual hard instance of this problem for him to solve.

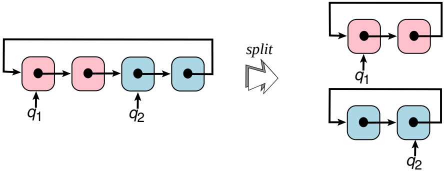
Opaque predicates from pointer aliasing

- Create an obfuscating transformation from a known computationally hard static analysis problem.
- We assume that
  1. the attacker will analyze the program statically, and
  2. we can force him to solve a particular static analysis problem to discover the secret he's after, and
  3. we can generate an actual hard instance of this problem for him to solve.
- Of course, these assumptions may be false!

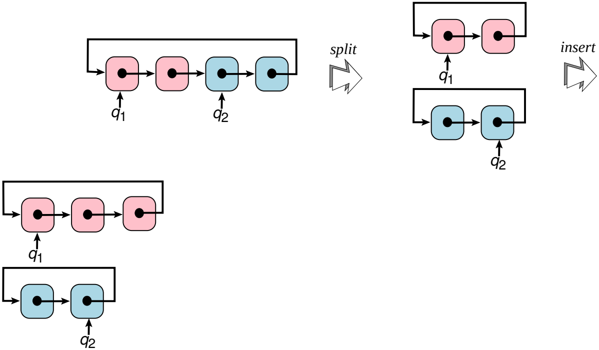
- Construct one or more heap-based graphs, keep pointers into those graphs, create opaque predicates by checking properties you know to be true.



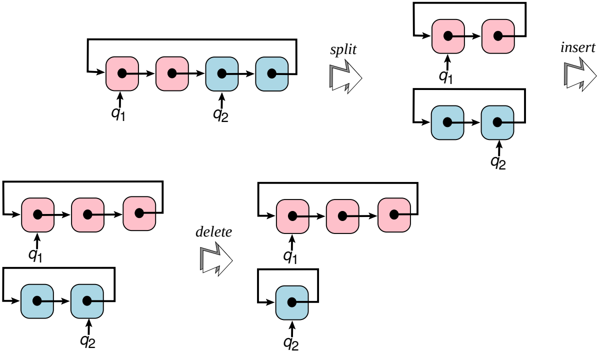
- Construct one or more heap-based graphs, keep pointers into those graphs, create opaque predicates by checking properties you know to be true.
- q_1 and q_2 point into two graphs G_1 (pink) and G_2 (blue):



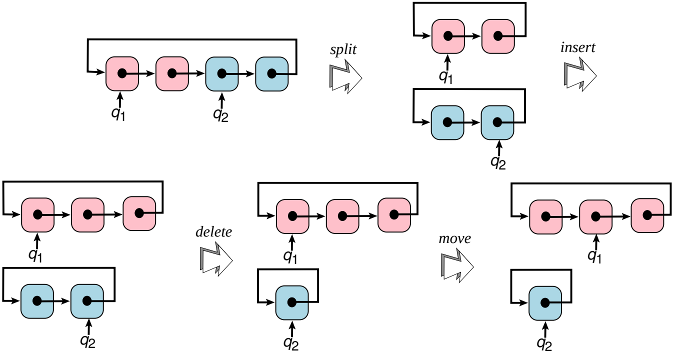
- Construct one or more heap-based graphs, keep pointers into those graphs, create opaque predicates by checking properties you know to be true.
- q_1 and q_2 point into two graphs G_1 (pink) and G_2 (blue):



- Construct one or more heap-based graphs, keep pointers into those graphs, create opaque predicates by checking properties you know to be true.
- q_1 and q_2 point into two graphs G_1 (pink) and G_2 (blue):



- Construct one or more heap-based graphs, keep pointers into those graphs, create opaque predicates by checking properties you know to be true.
- q_1 and q_2 point into two graphs G_1 (pink) and G_2 (blue):



Invariants

- Two invariants:
  - “G_1 and G_2 are circular linked lists”
  - “q_1 points to a node in G_1 and q_2 points to a node in G_2.”

Invariants

- Two invariants:
  - “G_1 and G_2 are circular linked lists”
  - “q_1 points to a node in G_1 and q_2 points to a node in G_2.”
- Perform enough operations to confuse even the most precise alias analysis algorithm,

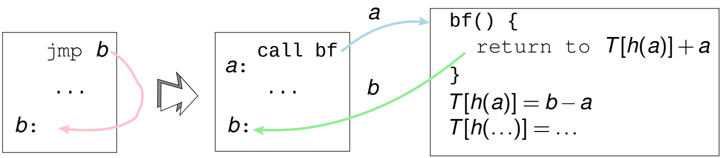
Invariants

- Two invariants:
  - "G_1 and G_2 are circular linked lists"
  - "q_1 points to a node in G_1 and q_2 points to a node in G_2."
- Perform enough operations to confuse even the most precise alias analysis algorithm,
- Insert opaque queries such as $(q_1 \neq q_2)^T$ into the code.

Branch Functions

Jumps through branch functions

- Replace unconditional jumps with a call to a branch function.
- Calls normally return to where they came from... But, a branch function returns to the target of the jump!



`jmp` b

...

b:

`call bf`

a:

...

b:

a

b

`bf() {`

`return to` $T[h(a)]+a$

`}`

$T[h(a)] = b - a$

$T[h(\ldots)] = \ldots$

Jumps through branch functions

- Designed to confuse disassembly.
- 39% of instructions are incorrectly assembled using a linear sweep disassembly.
- 25% for recursive disassembly.
- Execution penalty: 13%
- Increase in text segment size: 15%.

Breaking opaque predicates

Breaking opaque predicates

```
...
$x_1 \leftarrow \cdots$;
$x_2 \leftarrow \cdots$;
...
$b \leftarrow f(x_1, x_2, \ldots)$;
if $b$ goto ...
```

1. find the instructions that make up $f(x_1, x_2, \ldots)$;
2. find the inputs to f, i.e. $x_1, x_2 \ldots$;
3. find the range of values R_1 of x_1,. . . ;
4. compute the outcome of f for all input values;
5. kill the branch if $f \equiv \mathit{true}$.

Breaking opaque predicates

```
int x = some complicated
expression;
int y = 42;
z = ...
boolean b = (34*y*y-1)==x*x;
if b goto ...
```

1. Compute a backwards slice from `b`,
2. Find the inputs (`x` and `y`),
3. Find range of *x* and *y*,
4. Use number-theory/brute force to determine $b \equiv$ `false`.

Breaking $\forall x \in \mathbb{Z} : n | p(x)$

- Mila Dalla Preda:

- Attack opaque predicates confined to a single basic block.

Breaking $\forall x \in \mathbb{Z} : n | p(x)$

- Mila Dalla Preda:

- Attack opaque predicates confined to a single basic block.
- Assume that the instructions that make up the predicate are contiguous.

Breaking $\forall x \in \mathbb{Z} : n|p(x)$

- Mila Dalla Preda:

- Attack opaque predicates confined to a single basic block.
- Assume that the instructions that make up the predicate are contiguous.
- Start at a conditional jump instruction *j* and incrementally extend it with the $1, 2, \ldots$ instructions until an opaque predicate (or

Breaking $\forall x \in \mathbb{Z} : n | p(x)$

- Mila Dalla Preda:

- Attack opaque predicates confined to a single basic block.
- Assume that the instructions that make up the predicate are contiguous.
- Start at a conditional jump instruction *j* and incrementally extend it with the $1, 2, \ldots$ instructions until an opaque predicate (or

Breaking $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

Opaquely true predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$:

(1) (2) (3) (4)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

Breaking $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

Opaquely true predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$:

(1) (2) (3) (4)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

Breaking $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

Opaquely true predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$:

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(1)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(2)

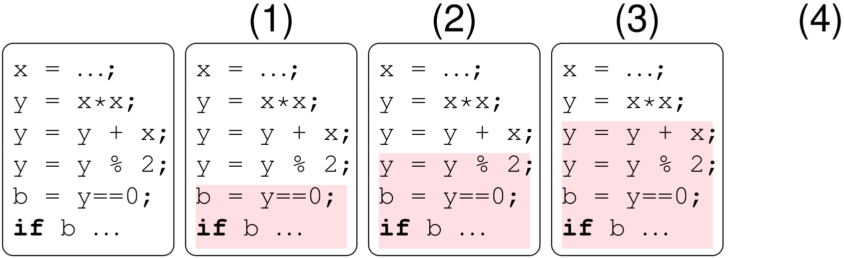
```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(3)

(4)

Breaking $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

Opaquely true predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$:



(1) (2) (3) (4)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(1)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(2)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

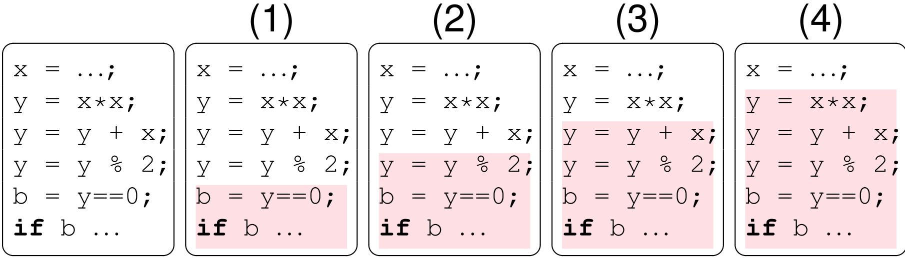
(3)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(4)

Breaking $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$

Opaquely true predicate $\forall x \in \mathbb{Z} : 2 | (x^2 + x)$:



```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(1)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(2)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(3)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

(4)

```
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```

Using Abstract Interpretation

Consider the case when `x` is an even

```
x = even number;
y = x * x;
y = y + x;
z = y % 2;
b = z==0;
if b ...
```

`x` = *even*;

`y = x` $*_a$ `x` = $\mathit{even} *_a \mathit{even} = \mathit{even}$;

`y = y` $+_a$ `x` = $\mathit{even} +_a \mathit{even} = \mathit{even}$;

`z = y` $\%_a$ `2` = $\mathit{even} \bmod 2 = 0$;

`b = z==0;` = **`true`**

`if` `b ...`

Using Abstract Interpretation

Consider the case when `x` starts out being odd:

```
x = odd number;
y = x * x;
y = y + x;
z = y % 2;
b = z==0;
if b ...
```

```
x              = odd;
y = x *_a x    = odd *_a odd = odd;
y = y +_a x    = odd +_a odd = even;
z = y %_a 2    = even mod 2 = 0;
b = z==0;      = true
if b ...
```

- Regardless of whether `x`'s initial value is even or odd, `b` is true!

Breaking $\forall x \in \mathbb{Z} : n | p(x)$

- Regardless of whether `x`'s initial value is even or odd, `b` is true!

Breaking $\forall x \in \mathbb{Z} : n | p(x)$

- Regardless of whether `x`'s initial value is even or odd, `b` is true!
- You've broken the opaque predicate, efficiently!!

Breaking $\forall x \in \mathbb{Z} : n|p(x)$

- Regardless of whether `x`'s initial value is even or odd, `b` is true!
- You've broken the opaque predicate, efficiently!!
- By constructing different abstract domains, Algorithm REPMBG is able to break all opaque predicates of the form $\forall x \in \mathbb{Z} : n|p(x)$ where $p(x)$ is a polynomial.

In-Class Exercise

1. An obfuscator has inserted the opaquely true predicate $\forall x \in \mathbb{Z} : 2|(2x+4)$:

```
x = ...;
if ((((2*x+4) % 2) == 0)^T) {
    some statement
}
```

   Or, in simpler operations:

```
x = ...;
y = 2 * x;
y = y + 4;
z = y % 2;
b = z==0;
if b ...
```

2. Play we're an attacker!

3. Do a symbolic evaluation, using these rules:

x	y	$x *_a y$
even	*even*	*even*
even	*odd*	*even*
odd	*even*	*even*
odd	*odd*	*odd*

x	y	$x +_a y$
even	*even*	*even*
even	*odd*	*odd*
odd	*even*	*odd*
odd	*odd*	*even*

x	$x \text{ mod }_a 2$
even	0
odd	1

④ First, let's assume that `x` is even.

```
x = even;
y = 2 * x;
y = y + 4;
z = y % 2;
b = z==0;
if b ...
```

$\mathtt{x} = \textit{even};$

$\mathtt{y = 2} \star_a \mathtt{x} =$

$\mathtt{y = y} +_a \mathtt{4} =$

$\mathtt{z = y} \%_a \mathtt{2} =$

$\mathtt{b = z==0;} =$

if b ...

⑤ Now, let’s assume that `x` is odd.

```
x = odd;
y = x * x;
y = y + x;
z = y % 2;
b = z==0;
if b ...
```

`x` = *odd*`;`

`y = 2` $\star_a$ `x =`

`y = y` $+_a$ `4 =`

`z = y` $\%_a$ `2 =`

`b = z==0; =`

`if` `b ...`

Integer Arithmetic

Encoding Integer Arithmetic

$$
\begin{aligned}
x+y &= x-\neg y-1\\
x+y &= (x\oplus y)+2\cdot(x\wedge y)\\
x+y &= (x\vee y)+(x\wedge y)\\
x+y &= 2\cdot(x\vee y)-(x\oplus y)
\end{aligned}
$$

- `www.hackersdelight.org`

Integer Arithmetic – Example

- One possible encoding of

```
z = x + y + w
```

  is

```
z = (((x ^ y) + ((x & y) << 1)) | w) +
    (((x ^ y) + ((x & y) << 1)) & w);
```

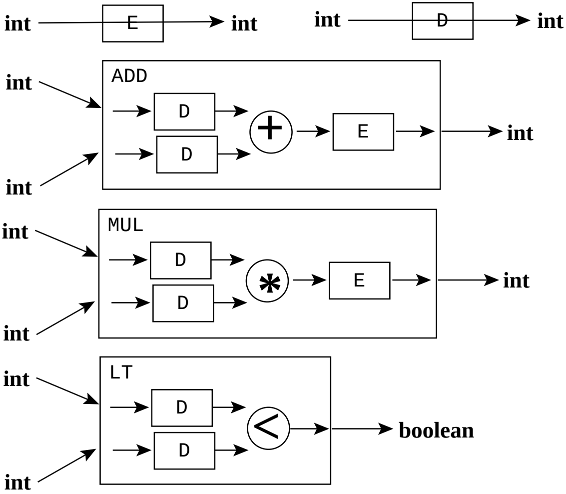
- Many others are possible, which is good for diversity.

Transforming Integers — The identity transformation

```
typedef int T1;
T1 E1(int e) {return e;}
int D1(T1 e) {return e;}
T1 ADD1(T1 a, T1 b) {return E1(D1(a)+D1(b));}
T1 MUL1(T1 a, T1 b) {return E1(D1(a)*D1(b));}
BOOL LT1(T1 a, T1 b) {return D1(a)<D1(b);}
```

- `E1` transforms cleartext integers into the obfuscated representation,
- `D1` transforms obfuscated integers into cleartext,
- `ADD1`, etc., perform operations in obfuscated space.

Transforming Integers — The identity transformation



Linear Transformation I

- We have 3 integer variables `x`, `y`, `z`, and we want to encode them with a linear transformation:

$$\begin{aligned} x' &= a \cdot x + b \\ y' &= a \cdot y + b \\ z' &= a \cdot z + b \end{aligned}$$

- Let a be an odd constant, and b a random constant.
- Let's pick $a = 7$, $b = 5$.

Linear Transformation II

```
int E(int e) {return a*e + b;}
int D(int e) {return ?;}
int ADD(int a, int b) {return ?;}
int MUL(int a, int b) {return ?;}
BOOL LT(int a, int b) {return a<b;}
```

- We need to solve for x:

$$\begin{aligned} x' &= a \cdot x + b \\ x &= a^{-1} \cdot x' - a^{-1} \cdot b \end{aligned}$$

Linear Transformation III

- Remember, all arithmetic is done mod 2^{32}!

$$\begin{aligned} x' &= a \cdot x + b \\ x &= a^{-1} \cdot x' - a^{-1} \cdot b \\ a &= 7 \\ a^{-1} &= 3067833783 \end{aligned}$$

- Why???

Linear Transformation IV

- Why??? Well, because

$$3067833783 \cdot 7 \bmod 2^{32} = 1$$

- Why??? Because Euclid's Extended Algorithm tells us

$$\gcd(7, 2^{32}) = 3067833783 \cdot 7 + 2 \cdot 2^{32} = 1$$

- And, since $2 \cdot 2^{32} \bmod 2^{32} = 0$, we get

$$3067833783 \cdot 7 = 1 \bmod 2^{32}$$

I.e., 3067833783 is the inverse of 7, mod $2^{3}2$.

Linear Transformation V

- We compute $a^{-1} \cdot b$

$$a^{-1} \cdot b \;=\; 3067833783 \cdot 5 \bmod 2^{32}$$

- And now we can encode and decode integers:

```
int E(int e) {return 7*e + 5;}
int D(int e) {return 3067833783*e - 2454267027;}
int ADD(int a, int b) {return ?;}
int MUL(int a, int b) {return ?;}
BOOL LT(int a, int b) {return a<b;}
```

Linear Transformation VI

- Let's try an example, 10:

$$\begin{aligned} E(10) &= (7 * 10 + 5) \bmod 2^{32} \\ &= 75 \\ D(75) &= (3067833783 \cdot 75 - 2454267027) \bmod 2^{32} \\ &= 1 \end{aligned}$$

- So, now we can encode and decode integers, using the linear formula $x' = a \cdot x + b$!

Linear Transformation VII (a)

What about addition in the encoded domain?

```
int E(int e) {return 7*e + 5;}
int D(int e) {return 3067833783*e - 2454267027;}
int ADD(int a, int b) {return ?;}
```

$$
\begin{aligned}
E(x)+E(y) &= E(D(E(x))+D(E(y))) \\
&= E((a^{-1}\cdot x-a^{-1}\cdot b)+ \\
&\quad (a^{-1}\cdot y-a^{-1}\cdot b)) \\
&= a\cdot(a^{-1}\cdot x-a^{-1}\cdot b)+ \\
&\quad (a^{-1}\cdot y-a^{-1}\cdot b)+b \\
&= x-b+y-b+b=x+y-b
\end{aligned}
$$

Linear Transformation VII (b)

- So, we get

```
int ADD(int a, int b) {
   return a + b - 2454267027;
}
```

Linear Transformation VIII

- Example:

```
int main () {
   int x = 10;
   int y = 12;
   int z = x + y;
   printf(z);
}
```

- We get:

```
int main () {
   int x = 7*10 + 5; // 75
   int y = 7*12 + 5; // 89
   int z = 75 + 89 - 5; // 159
   printf(3067833783*z - 2454267027); // 22!
}
```

Exercise: Integer encoding

- Consider again the GCD routine:

```
int gcd(int x, int y) {
   int temp;
   while (true) {
      boolean b = x%y == 0;
      if (b) break;
      temp = x%y;
      x = y;
      y = temp;
   }
}
```

- Use the *E*()/*D*() scheme above to encode the integer variables.
- What kind of encoding would work well here?

Another Number-theoretic trick

```
#define N4 (53*59)
int E4(int e,int p) {return p*N4+e;}
int D4(int e) {return e%N4;}
int ADD4(int a, int b) {return a+b;}
int MUL4(int a, int b) {return a*b;}
BOOL Lint(int a, int b) {return D4(a)<D4(b);}
```

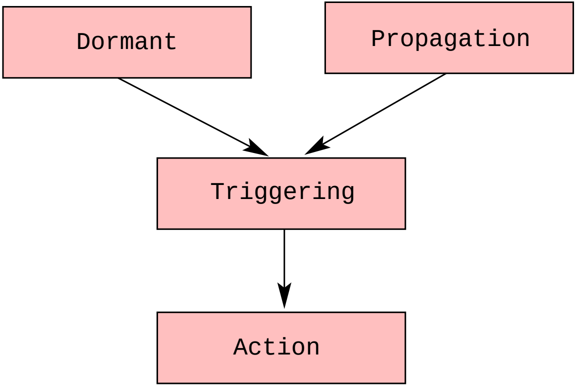
- An integer *y* is represented as $N * p + y$, where *N* is the product of two close primes, and *p* is a random value.
- Addition and multiplication are performed in obfuscated space.
- Comparisons require deobfuscation.
- Parameterized obfuscation: create a family

Computer Viruses

Computer Viruses

- Viruses
  1. are self-replicating;
  2. attach themselves to other files;
  3. requires user assistance to to replicate.
  4. use obfuscation to hide!

Computer Viruses: Phases



Dormant

Propagation

Triggering

Action

Computer Viruses: Phases. . .

- Dormant — lay low, avoid detection.
- Propagation — infect new files and systems.
- Triggering — decide to move to action phase
- Action — execute malicious actions, the payload.

Virus Types

- Program/File virus:

Virus Types

- Program/File virus:
  - Attaches to: program object code.

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.
- Doocument/Macro virus:

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.
- Doocument/Macro virus:
  - Attaches to: document (.doc,.pdf,. . .).

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.
- Doocument/Macro virus:
  - Attaches to: document (.doc,.pdf,. . .).
  - Run when: document is opened.

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.
- Doocument/Macro virus:
  - Attaches to: document (.doc,.pdf,. . .).
  - Run when: document is opened.
  - Propagates by: emailing documents.

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.
- Doocument/Macro virus:
  - Attaches to: document (.doc,.pdf,. . .).
  - Run when: document is opened.
  - Propagates by: emailing documents.
- Boot sector virus:

Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.
- Doocument/Macro virus:
  - Attaches to: document (.doc,.pdf,. . .).
  - Run when: document is opened.
  - Propagates by: emailing documents.
- Boot sector virus:
  - Attaches to: hard drive boot sector.

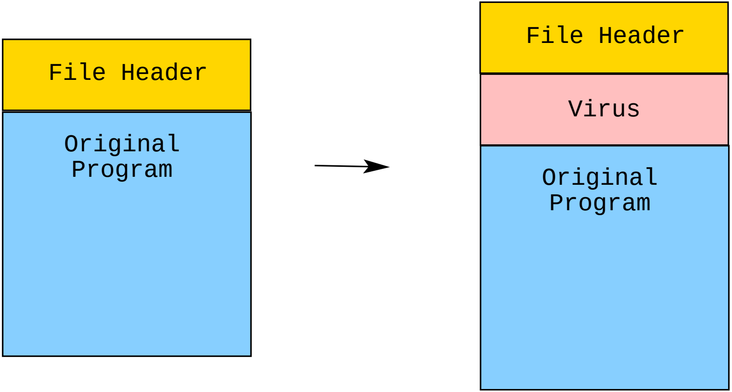
Virus Types

- Program/File virus:
  - Attaches to: program object code.
  - Run when: program executes.
  - Propagates by: program sharing.
- Doocument/Macro virus:
  - Attaches to: document (.doc,.pdf,. . .).
  - Run when: document is opened.
  - Propagates by: emailing documents.
- Boot sector virus:
  - Attaches to: hard drive boot sector.
  - Run when: computer boots.

Virus Types

- Program/File virus:
    - Attaches to: program object code.
    - Run when: program executes.
    - Propagates by: program sharing.
- Doocument/Macro virus:
    - Attaches to: document (.doc,.pdf,. . .).
    - Run when: document is opened.
    - Propagates by: emailing documents.
- Boot sector virus:
    - Attaches to: hard drive boot sector.
    - Run when: computer boots.
    - Propagates by: sharing floppy disks.

Computer Viruses: Propagation


File Header

Original Program

Virus

Virus Defenses

- Signatures: Regular expressions over the virus code used to detect if files have been infected.
- Checking can be done
  1. periodically over the entire filesystem;
  2. whenever a new file is downloaded.

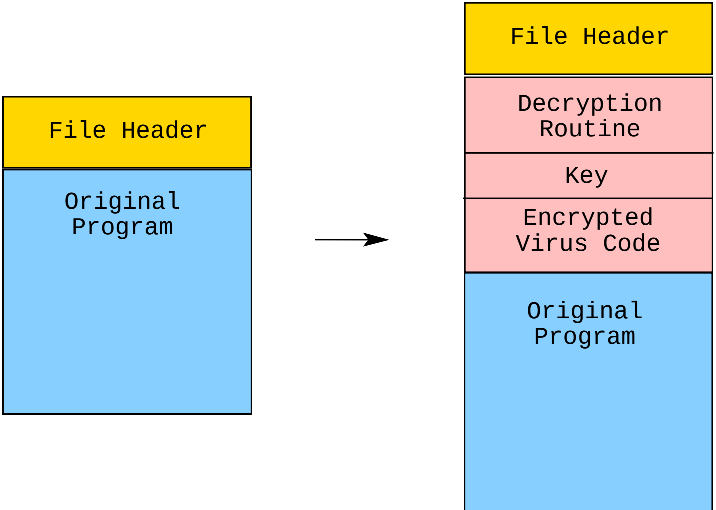
Virus Countermeasures

- Viruses need to protect themselves against detection.
- This means hiding any distringuishing features, making it hard to construct signatures.
- By encrypting its payload, the virus hides its distinguishing features.
- Encryption is often no more than xor with a constant.

Virus Countermeasures: Encryption

- By encrypting its payload, the virus hides its distinguishing features.
- The decryption routine itself, however, can be used to create a signature!

Computer Countermeasures: Encryption. . .



File Header

Original Program

File Header

Decryption Routine

Key

Encrypted Virus Code

Original Program

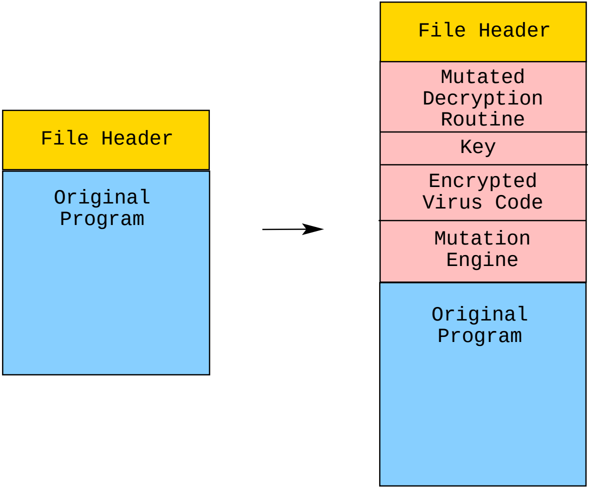
Virus Countermeasures: Polymorphism

- Each variant is encrypted with a different key.

Virus Countermeasures: Metamorphism

- To prevent easy creation of signatures for the decryption routine, metamorphic viruses will mutate the decryptor, for each infection.
- The virus contains a mutation engine which can modify the decryption code while maintaining its semantics.

Computer Countermeasures: Metamorphism…



Virus Countermeasures: Metamorphism. . .

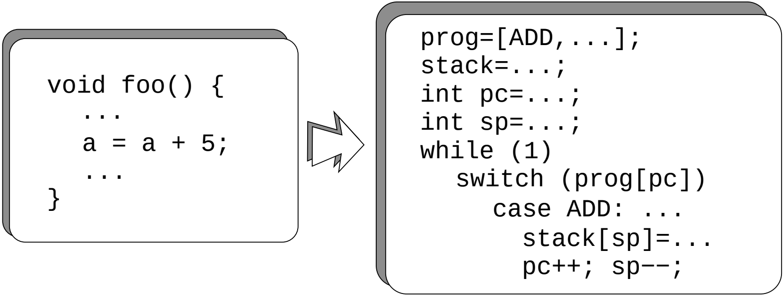
- To counter metamorphism, virus detectors can run the virus in an emulator.
- The emulator gathers a trace of the execution.
- A virus signature is then constructed over the trace.
- This makes it easier to ignore garbage instructions the mutation engine may have inserted.

Virtualization

Interpreters

- An interpreter is program that behaves like a CPU, but which has its own
  - instruction set,
  - program,
  - program counter
  - execution stack
- Many programming languages are implemented by constructing an interpreter for them, for example Java, Python, Perl, etc.

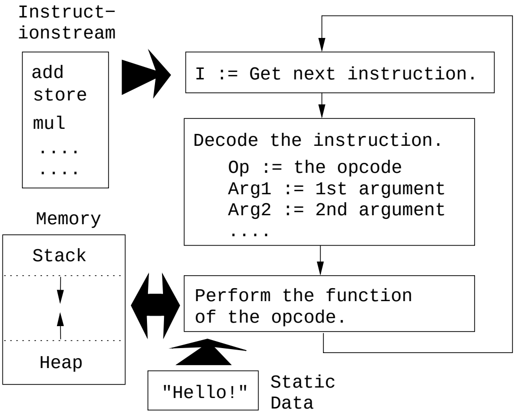
Interpreters for Obfuscation



```
void foo() {
   ...
   a = a + 5;
   ...
}
```

```
prog=[ADD,...];
stack=...;
int pc=...;
int sp=...;
while (1)
   switch (prog[pc])
      case ADD: ...
        stack[sp]=...
        pc++; sp−−;
```

Interpreter Engine



Instruct-
ionstream

add
store
mul
....
....

I := Get next instruction.

Decode the instruction.

```
   Op := the opcode
   Arg1 := 1st argument
   Arg2 := 2nd argument
   ....
```

Perform the function
of the opcode.

Memory

Stack

Heap

"Hello!"

Static
Data

Diversity

- Viruses want diversity in the code they generate.
- This means, every version of the virus should look different, so that they are hard for the virus detector to find.
- We want the same when we protect our programs!

Tigress Diversity

- `tigress.cs.arizona.edu`
- Interpreter diversity:
  1. 8 kinds of instruction dispatch: switch, direct, indirect, call, ifnest, linear, binary, interpolation
  2. 2 kinds of operands: stack, registers
  3. arbitrarily complex instructions
  4. operators are randomized
- Along with: flatten, merge functions, split functions, opaque predicates, etc.

Tigress Diversity

- Every input program generates a unique interpreter.
- A seed sets the random number generator that allows us to generate many different interpreters for the same input program.
- The split transformation can be used to break up the interpreter in pieces, to make it less easy to detect.

In-class Exercise

```
tigress --Transform=Virtualize --Functions=fib \
           --VirtualizeDispatch=switch \
           --out=v1.c test1.c
gcc -o v1 v1.c

tigress --Transform=Virtualize --Functions=fib \
           --VirtualizeDispatch=indirect \
           --out=v2.c test1.c
gcc -o v2 v2.c
```

In-class Exercise

```
tigress --Transform=Virtualize --Functions=fib \
           --VirtualizeDispatch=switch \
        --Transform=Virtualize --Functions=fib \
           --VirtualizeDispatch=indirect \
        --out=v3.c test1.c
gcc -o v3 v3.c

tigress --Transform=Virtualize --Functions=fib \
          --VirtualizeDispatch=switch \
          --VirtualizeSuperOpsRatio=2.0 \
          --VirtualizeMaxMergeLength=10 \
          --VirtualizeOptimizeBody=true \
          --out=v4.c test1.c
gcc  -o v4 v4.c
```

Attack 1

- Reverse engineer the instruction set!
- Look at the instruction handlers, and figure out what they do:

```
case o233:
   (pc) ++;
   s[sp - 1].i = s[sp - 1].i < s[sp].i;
   (sp) --;
   break;
```

- Then recreate the original program from the virtual one.

Counter Attack 1

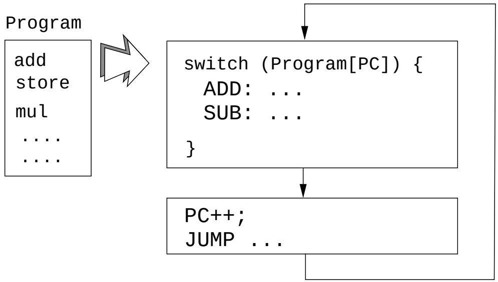
- Make instructions with complex semantics, using super operators:

```
case o98:
    (pc) ++;
    *((int *)s[sp + 0].v) = s[sp + -1].i;
    *((int *)((void *)(l + *((int *)(pc + 4))))) =
        *((int *)((void *)(l + *((int *)pc))));
    s[sp + -1].i = *((int *)((void *)(l + *((int *)(pc + 8)))))
        *((int *)(pc + 12));
    s[sp + 0].v = (void *)(l + *((int *)(pc + 16)));
    pc += 20;
    break;
```

- Then recreate the original program from the virtual one.

Attack 2

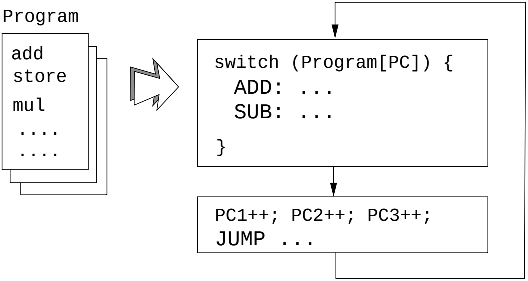
- Dynamic attack: run the program, collect all instructions, look for patterns that look like the virtual PC:



`Trace:switch,ADD,PC++,JUMP,switch,...`

Counter Attack 2

- Tigress can merge several programs, so they execute in tandem, making it harder to detect what is the PC (there are many PCs!).



Discussion

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks
- Code Privacy — make programs hard to understand to protect algorithms

Code Obfuscation — What’s it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks
- Code Privacy — make programs hard to understand to protect algorithms
- Data Privacy — make programs hard to understand to protect secret data (keys)

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks
- Code Privacy — make programs hard to understand to protect algorithms
- Data Privacy — make programs hard to understand to protect secret data (keys)
- Integrity — make programs hard to understand to make them hard to change