Software Protection:
How to Crack Programs, and Defend Against Cracking
Lecture 4: Code Obfuscation
Moscow State University, Spring 2014

Christian Collberg
University of Arizona

www.cs.arizona.edu/~collberg
© March 19, 2014 Christian Collberg
Last week’s lecture

Who needs program analysis?
Last week’s lecture

- Who needs program analysis?
- What is the result of a control flow analysis?
Last week’s lecture

- Who needs program analysis?
- What is the result of a **control flow analysis**?
- Give two algorithms for disassembly!
Last week’s lecture

- Who needs program analysis?
- What is the result of a control flow analysis?
- Give two algorithms for disassembly!
- Give some reasons why disassembly is hard!
Last week’s lecture

- Who needs program analysis?
- What is the result of a **control flow analysis**?
- Give two algorithms for disassembly!
- Give some reasons why disassembly is hard!
- What is a **script kiddie**?
Today’s lecture

1. Static obfuscation algorithms
2. Computer viruses
3. Code Diversity
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** $\Rightarrow$ obfuscated programs that remain fixed at runtime.
  - tries to thwart static analysis
  - attacked by dynamic techniques (debugging, emulation, tracing).

- **dynamic obfuscators** $\Rightarrow$ 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:

  \[ \begin{array}{c}
  \begin{array}{ccc}
  \text{true} & P^T & \text{false} \\
  \downarrow & \downarrow & \downarrow \\
  \text{true} & P^F & \text{false} \\
  \downarrow & \downarrow & \downarrow \\
  \text{true} & P? & \text{false} \\
  \downarrow & \downarrow & \downarrow \\
  \end{array}
  \end{array} \]

- Building blocks for many obfuscations.
Opaque Expressions

An opaquely true predicate:

\[ 2(x^2 + x)^T \]
Opaque Expressions

- An opaquely true predicate:

- An opaquely indeterminate predicate:
Inserting bogus control-flow

- Insert *bogus* control-flow into a function:
  - 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.

```
true  P?  false
```

```
[Diagram showing control flow with conditional block and two parallel blocks]
```
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:
```c
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;
}
```
```c
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; break;
        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;
        }
    }
```
switch(next)

B0: k=0
    s=1
    next=1

B1: if (k<w)
      next=2
    else
      next=6

B2: if (x[k]==1)
      next=5
    else
      next=4

B3: R=(s*y)%n
    next=5

B4: R=s
    next=5

B5: S=R*R%n
    L=R
    K++
    next=1

B6: return L
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!

```c
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; break;
            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:

```
ENTER
X := 20;
if x >= 10 goto B4
X := X - 1;
A[X] := 10;
if X <> 4 goto B6
X := X - 2;
goto B2
if x >= 10 goto B4
Y := X + 5;
B1
B2
B3
B4
B5
B6
EXIT
``` 

2. Give the source code for the flattened graph above.
Constructing Opaque Predicates
Opaque values from array aliasing

Invariants:
1. every third cell (in pink), starting will cell 0, is $\equiv 1 \mod 5$;
2. cells 2 and 5 (green) hold the values 1 and 5, respectively;
3. every third cell (in blue), starting will cell 1, is $\equiv 2 \mod 7$;
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 \( \equiv 1 \mod 5! \)
- That is, make any changes you want, while maintaining the invariant.
- This will make static analysis harder for the attacker.
```c
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):
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,
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!

```
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

\[ \ldots \]
\[ x_1 \leftarrow \cdots; \]
\[ x_2 \leftarrow \cdots; \]
\[ \ldots \]
\[ b \leftarrow f(x_1, x_2, \ldots); \]
\[ \text{if } b \text{ goto } \ldots \]

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 \( x_1, \ldots; \)
4. compute the outcome of \( f \) for all input values;
5. kill the branch if \( f \equiv \text{true}. \)
Breaking opaque predicates

```java
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** \( (x \text{ and } y) \),
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 $\forall x \in \mathbb{Z} : n \mid p(x)$

- 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, ... instructions until an opaque predicate (or
Breaking $\forall x \in \mathbb{Z} : n \mid p(x)$

- 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, ... instructions until an opaque predicate (or
Breaking $\forall x \in \mathbb{Z} : 2 \mid (x^2 + x)$

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

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

```plaintext
x = ...;
y = x*x;
y = y + x;
y = y % 2;
b = y==0;
if b ...
```
Breaking $\forall x \in \mathbb{Z} : 2 \mid (x^2 + x)$

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

(1) $x = \ldots$; $y = x \times x$; $y = y + x$; $y = y \% 2$; $b = y == 0$; if $b$ ...

(2) $x = \ldots$; $y = x \times x$; $y = y + x$; $y = y \% 2$; $b = y == 0$; if $b$ ...

(3) $x = \ldots$; $y = x \times x$; $y = y + x$; $y = y \% 2$; $b = y == 0$; if $b$ ...

(4) $x = \ldots$; $y = x \times x$; $y = y + x$; $y = y \% 2$; $b = y == 0$; if $b$ ...

$41 / 82$
Breaking $\forall x \in \mathbb{Z} : 2 \mid (x^2 + x)$

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x = ...;</td>
<td>x = ...;</td>
<td>x = ...;</td>
<td>x = ...;</td>
</tr>
<tr>
<td>y = x*x;</td>
<td>y = x*x;</td>
<td>y = x*x;</td>
<td>y = x*x;</td>
</tr>
<tr>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>y = y + x;</td>
</tr>
<tr>
<td>y = y % 2;</td>
<td>y = y % 2;</td>
<td>y = y % 2;</td>
<td>y = y % 2;</td>
</tr>
<tr>
<td>b = y==0;</td>
<td>b = y==0;</td>
<td>b = y==0;</td>
<td>b = y==0;</td>
</tr>
<tr>
<td>if b ...</td>
<td>if b ...</td>
<td>if b ...</td>
<td>if b ...</td>
</tr>
</tbody>
</table>

(1) (2) (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) $x = \ldots$;
   $y = x \times x$;
   $y = y + x$;
   $y = y \% 2$;
   $b = y == 0$;
   if $b$ ...

(2) $x = \ldots$;
   $y = x \times x$;
   $y = y + x$;
   $y = y \% 2$;
   $b = y == 0$;
   if $b$ ...

(3) $x = \ldots$;
   $y = x \times x$;
   $y = y + x$;
   $y = y \% 2$;
   $b = y == 0$;
   if $b$ ...

(4) $x = \ldots$;
   $y = x \times x$;
   $y = y + x$;
   $y = y \% 2$;
   $b = y == 0$;
   if $b$ ...
Breaking $\forall x \in \mathbb{Z} : 2 \mid (x^2 + x)$

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

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>...;</td>
<td>...;</td>
<td>...;</td>
<td>...;</td>
</tr>
<tr>
<td>y</td>
<td>$x \times x$;</td>
<td>$x \times x$;</td>
<td>$x \times x$;</td>
<td>$x \times x$;</td>
</tr>
<tr>
<td>y</td>
<td>$y + x$;</td>
<td>$y + x$;</td>
<td>$y + x$;</td>
<td>$y + x$;</td>
</tr>
<tr>
<td>y</td>
<td>$y % 2$;</td>
<td>$y % 2$;</td>
<td>$y % 2$;</td>
<td>$y % 2$;</td>
</tr>
<tr>
<td>b</td>
<td>$y == 0$;</td>
<td>$y == 0$;</td>
<td>$y == 0$;</td>
<td>$y == 0$;</td>
</tr>
<tr>
<td>if</td>
<td>$b \ldots$</td>
<td>$b \ldots$</td>
<td>$b \ldots$</td>
<td>$b \ldots$</td>
</tr>
</tbody>
</table>
Using Abstract Interpretation

Consider the case when \( x \) is an even number:

\[
\begin{align*}
x &= \text{even number}; \\
y &= x \times x; \\
y &= y + x; \\
z &= y \mod 2; \\
b &= z == 0; \\
\textbf{if} \ b \ldots
\end{align*}
\]
Using Abstract Interpretation

Consider the case when \( x \) starts out being odd:

\[
\begin{align*}
x &= \text{odd number}; \\
y &= x \times x; \\
y &= y + x; \\
z &= y \mod 2; \\
b &= z==0; \\
\text{if } b ... 
\end{align*}
\]

Regardless of whether \( x \)'s initial value is even or odd, \( b \) is true!
Breaking $\forall x \in \mathbb{Z} : n \mid p(x)$

- 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, $\triangledown$ 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, $\mathsf{b}$ is true!
- You’ve broken the opaque predicate, efficiently!!
- By constructing different abstract domains, Algorithm \texttt{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 $\forall x \in \mathbb{Z} : 2 \vert (2x + 4)$:

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

   Or, in simpler operations:

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

2. Play we’re an attacker!
Do a symbolic evaluation, using these rules:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \times_a y$</th>
<th>$x$</th>
<th>$y$</th>
<th>$x +_a y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>even</td>
<td>even</td>
<td>even</td>
<td>even</td>
<td>even</td>
<td>even</td>
</tr>
<tr>
<td>even</td>
<td>odd</td>
<td>even</td>
<td>even</td>
<td>odd</td>
<td>odd</td>
</tr>
<tr>
<td>odd</td>
<td>even</td>
<td>even</td>
<td>odd</td>
<td>even</td>
<td>odd</td>
</tr>
<tr>
<td>odd</td>
<td>odd</td>
<td>odd</td>
<td>odd</td>
<td>odd</td>
<td>even</td>
</tr>
</tbody>
</table>

$x \mod_a 2$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x \mod_a 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>even</td>
<td>0</td>
</tr>
<tr>
<td>odd</td>
<td>1</td>
</tr>
</tbody>
</table>
First, let’s assume that $x$ is even.

\[
\begin{align*}
x & = \text{even} \\
y & = 2 \times x \\
y & = y + 4 \\
z & = y \mod 2 \\
b & = z == 0 \\
\text{if } b & \ldots
\end{align*}
\]

\[
\begin{align*}
x & = \text{even} \\
y & = 2 \times a \\
y & = y + a \\
z & = y \mod a \\
b & = z == 0 \\
\text{if } b & \ldots
\end{align*}
\]
Now, let’s assume that $x$ is odd.

$x = \textit{odd};$

$y = x \times x;$

$y = y + x;$

$z = y \% 2;$

$b = z==0;$

\textbf{if} $b$ ...

\[
x = \textit{odd};
\]

\[
y = 2 \times a \times x =
\]

\[
y = y + a \times 4 =
\]

\[
z = y \% a \times 2 =
\]

\[
b = z==0;
\]

\textbf{if} $b$ ...

\[
b = z==0;
\]

\textbf{if} $b$ ...
Computer Viruses
Computer Viruses

Viruses

1. are **self-replicating**;
2. attach themselves to other files;
3. requires user assistance 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.

- **Document/Macro virus:**
Virus Types

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

**Document/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.

- **Document/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.

- **Document/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.

- Document/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.

- **Document/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.

- **Document/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.

- **Document/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

![Diagram showing the propagation of a virus in a program file. The original program file has a file header and the virus is inserted between the file header and the original program.]
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 distinguishing 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...
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 a 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.
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

I := Get next instruction.

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

Perform the function of the opcode.

"Hello!"

Memory

Stack

Heap

Instruction stream

add
store
mul
....
....

"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.
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:

```c
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:

```c
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:

```plaintext
switch (Program[PC]) {
    ADD: ...
    SUB: ...
}
PC++;
JUMP ...
```

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!).

```
switch (Program[PC]) {
  ADD: ...
  SUB: ...
}

PC1++; PC2++; PC3++;
JUMP ...
```
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
Next week’s lecture

1. Dynamic obfuscation algorithms
2. Tamperproofing algorithms
3. Please check the website for important announcements:

www.cs.arizona.edu/~collberg/
Teaching/mgu/2014