Code Obfuscation

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?

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

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!
- 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

Obfuscating Transformations

p. 203

Obfuscation — The Early Years!

- Fred Cohen: *Operating system protection through program evolution*

Obfuscation — The Early Years!

- Fred Cohen: *Operating system protection through program evolution*

- Diversity of programs: ways to generate syntactically different but semantically identical versions of the same program.

Obfuscation — The Early Years!

- Fred Cohen: *Operating system protection through program evolution*

- Diversity of programs: ways to generate syntactically different but semantically identical versions of the same program.
- Make an installation of a program different from all other installations $\Rightarrow$ harder for the malware writer to write their code generically enough to work on all versions.

Algorithm
OBFCF

p. 203

Diversifying transformations

Obfuscating Transformations: Expression equivalence

- Compilers optimize for the fastest sequence of instructions.
- You can optimize for confusion instead!

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

⇨

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

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{reorder}}$: Reordering Code and Data

- Programmers put related pieces of code close together.

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{reorder}}$: Reordering Code and Data

- Programmers put related pieces of code close together.
- Locality can help a reverse engineer to see what pieces of code belong together.

Algorithm $\text{OBFCF}_{\text{reorder}}$: Reordering Code and Data

- Programmers put related pieces of code close together.
- Locality can help a reverse engineer to see what pieces of code belong together.
- ⇒ Randomize the placement of
  - modules within a program,
  - functions within a module,
  - statements within a function, and
  - instructions within a statement.

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{inoutline}}$: Splitting and Merging Functions

- As programmers we use *abstraction* to manage the complexity of larger programs.

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{inoutline}}$: Splitting and Merging Functions

- As programmers we use *abstraction* to manage the complexity of larger programs.
- Function inlining breaks the abstraction boundary.

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{inoutline}}$: Splitting and Merging Functions

- As programmers we use *abstraction* to manage the complexity of larger programs.
- Function inlining breaks the abstraction boundary.
- Function outlining inserts a bogus abstraction.

Algorithm $\text{OBFCF}_{\text{inoutline}}$: Splitting and Merging Functions

```
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;
}
```

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{inoutline}}$: Splitting and Merging Functions

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

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

Algorithm $\text{OBFCF}_{\text{inoutline}}$: Splitting and Merging Functions

```
float foo[100];

void f(int a,float b) {
    foo[a] = b;
}

float g(float c,char d) {
    return c*(float)d;
}

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

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{inoutline}}$: Splitting and Merging Functions

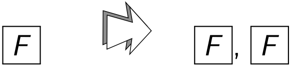
```
float foo[100];

float fg(int a,float bc,
         char d,int which) {
   if (which==1)
      foo[a] = bc;
   return bc*(float)d;
}

int main() {
   fg(42,42.0,'b',1);
   float v=fg(99,6.0,'a',2);
}
```

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{copy}}$: Copying code

- Make the program larger by cloning pieces of it:



Algorithm OBFCF$_{\text{copy}}$: Copying code

- Make the program larger by cloning pieces of it:

$$\boxed{F} \quad \Rightarrow \quad \boxed{F}, \boxed{F}$$

- Make the copied code look different from the original:

$$\boxed{F} \quad \Rightarrow \quad \boxed{F}, \boxed{F'}$$

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

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{copy}}$: Copying code

```
float foo[100];
void f(int a, float b) {
   foo[a] = b;
}
int main() {
   f(42, 42.0);
   f(6, 7.0);
}
```

- `f` is called twice

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{copy}}$: Copying code

```
float foo[100];
void f(int a, float b) {
    foo[a] = b;
}
float bogus;
void f1(int a, float b) {
    *(foo + a*sizeof(float)) = b;
    b += a*2;
    bogus += b+a;
}
int main() {
    f(42, 42.0);
    f1(6, 7.0);
}
```

- `f` and `f1` do the same thing.

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{interp}}$: Interpretation

- Add a level of interpretation:
  1. Define your own instruction set
  2. Translate your program to this instruction set
  3. Write an interpreter for the instruction set

Algorithm $\textsc{obf}\mathrm{CF}_{\mathrm{interp}}$: Interpretation

- Add a level of interpretation:
  1. Define your own instruction set
  2. Translate your program to this instruction set
  3. Write an interpreter for the instruction set
- Your program: 10-100x slower than before.

Algorithm $\textsc{obf}CF_{\text{interp}}$: Interpretation

```
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) {
   void* prog[]={...};
   int R, L, k = 0, s = 1;
   int Stack[10]; int sp=0;
   void** pc = (void**) &prog;
   goto **pc++;
   inc_k:     k++; goto **pc++;
   pusha:     Stack[sp++]=(int)*pc; pc++; goto **pc++;
   pushv:     Stack[sp++]=*(int*)*pc; pc++; goto **pc++;
   store:     *((int*)Stack[sp-2])=Stack[sp-1]; sp-=2;
              goto **pc++;
   x_k_ne_1:  if (x[k] != 1) pc=*pc; else pc++; goto **pc++;
   k_ge_w:    if (k >= w) return L; goto **pc++;
   add:       Stack[sp-2] += Stack[sp-1]; sp--; goto **pc++;
   mul:       Stack[sp-2] *= Stack[sp-1]; sp--; goto **pc++;
   mod:       Stack[sp-2] %= Stack[sp-1]; sp--; goto **pc++;
   jump:      pc=*pc; goto **pc++;
}
```

```
void* prog[]={
   // if (k >= w) return L
   &&k_ge_w,
   // if (x[k] == 1)
   &&x_k_ne_1,&prog[16],
   // R = (s*y) % n;
   &&pusha,&R,&&pushv,&s,&&pushv,&y,&&mul,&&pushv,&n,&&mod,
   // Jump after if-statement
   &&jump,&prog[21],
   // R = s;
   &&pusha,&R,&&pushv,&s,&&store,
   // s = R*R % n;
   &&pusha,&s,&&pushv,&R,&&pushv,&R,&&mul,&&pushv,&n,&&mod,
   // L = R;
   &&pusha,&L,&&pushv,&R,&&store,
   // k++
   &&inc_k,
  // Jump to top of loop
  &&jump,&prog[0]
};
```

Complicating control flow

p. 225

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

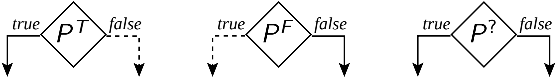
- Notation:
  - P^T for an *opaquely true* predicate
  - P^F for an *opaquely false* predicate
  - $P^?$ for an *opaquely indeterminate* predicate
  - $E^{=v}$ for an *opaque* expression of value v

Opaque Expressions

- 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

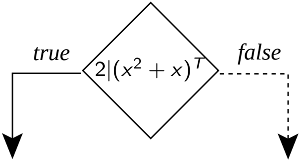
- Notation:
  - P^T for an *opaquely true* predicate
  - P^F for an *opaquely false* predicate
  - $P^?$ for an *opaquely indeterminate* predicate
  - $E^{=v}$ for an *opaque* expression of value v
- Graphical notation:



- Building blocks for many obfuscations.

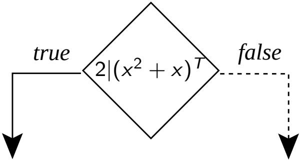
Opaque Expressions

- An opaquely true predicate:

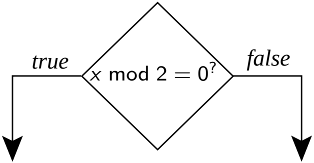


Opaque Expressions

- An opaquely true predicate:



- An opaquely indeterminate predicate:



Simple Opaque Predicates

- Look in number theory text books, in the *problems* sections:

  "Show that $\forall x, y \in \mathbb{Z} : p(x, y)$"

Simple Opaque Predicates

- Look in number theory text books, in the *problems* sections:

  "Show that $\forall x, y \in \mathbb{Z} : p(x, y)$"

- $\forall x, y \in \mathbb{Z} : x^2 - 34y^2 \neq 1$

Simple Opaque Predicates

- Look in number theory text books, in the *problems* sections:

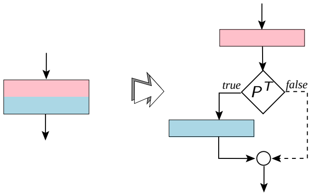
  $$\text{“Show that } \forall x, y \in \mathbb{Z} : p(x, y)\text{”}$$

- $\forall x, y \in \mathbb{Z} : x^2 - 34y^2 \neq 1$
- $\forall x \in \mathbb{Z} : 2|x^2 + x$
- ...

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{bogus}}$

p. 235

Inserting bogus control-flow



Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{bogus}}$: Inserting bogus control-flow

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

Algorithm $\text{OBFCTJ}_{\text{bogus}}$: 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

Algorithm $\text{OBFCTJ}_{\text{bogus}}$: Inserting bogus control-flow

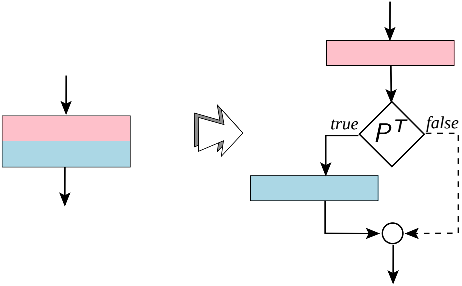
- Insert *bogus* control-flow into a function:
  1. dead branches which will never be taken
  2. superfluous branches which will *always* be taken
  3. branches which will sometimes be taken and sometimes not, but where this doesn't matter

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{bogus}}$: 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.

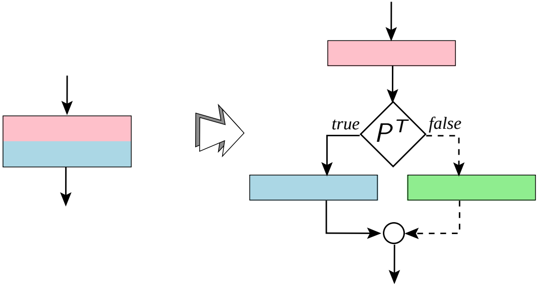
Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{bogus}}$: Inserting bogus control-flow

- It seems that the blue block is only sometimes executed:



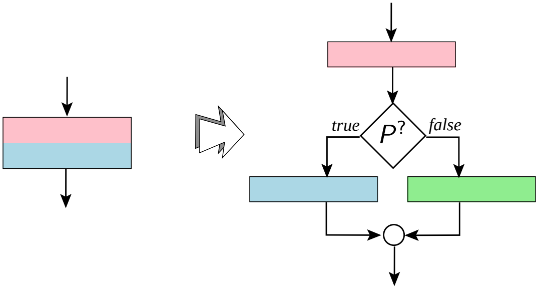
Algorithm OBFCTJ$_{\text{bogus}}$: Inserting bogus control-flow

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



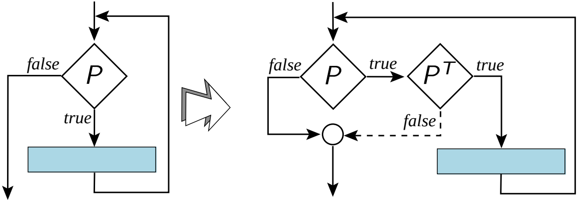
Algorithm $\text{OBFCTJ}_{\text{bogus}}$: Inserting bogus control-flow

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



Algorithm OBFCTJ$_{\text{bogus}}$: Inserting bogus control-flow

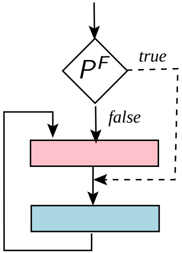
- Extend a loop condition P by conjoining it with an opaquely true predicate P^T:



Irreducible graphs

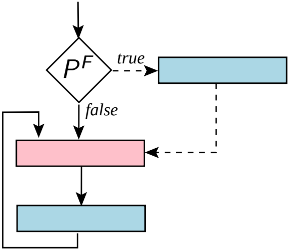
- Build your code out of nested if-, for-, while-, repeat-statements, etc., ⇒ the CFG will be reducible.
- Static analysis of reducible CFGs is straight-forward, and efficient.
- Jump into the middle of a loop ⇒ CFG is irreducible.
- Static analysis over irreducible CFGs is complicated.

```
if (P^F) goto b;
while (1) {
    x = y+10; return x;
    b: y = 20;
}
```



Irreducible graphs

```
if (P^F)
    y = 20;
while (1) {
   x = y+10; return x;
   y = 20;
}
```



- Before further analyzing the CFG, deobfuscate it, make it reducible.
- Here we used a nodesplitting deobfuscation.
- A really complex CFG with n nodes $\Rightarrow$ the deobfuscated reducible graph will have 2^{n-1} nodes!

Irreducible graph

- Unfortunately, there are other ways of deobfuscating:

```
int firsttime=1;
while (1) {
   if ((!firsttime) || (!P^F)) {
      x = y+10; return x;
   }
   y = 20;
   firsttime=0;
}
```

- It's not known whether this construction also causes exponential blowup.

Complicating dynamic analaysis

- Make opaque predicates *interdependent*.
- An attacker cannot simply remove one predicate at a time, rerun the program to see if it still works, remove another predicate, etc.
- Instead, he has to remove *all* interdependent opaque predicates at the same time (or a divide-by-zero will be raised):

```
int x=0, y=2,t;
while (1) {
    if (t=x*(x-1)%y==0,y-=2,x+=2,t)^T ...
    if (t=y*(y-1)%x==0,y+=2,x-=2,t)^T ...
}
```

Problem

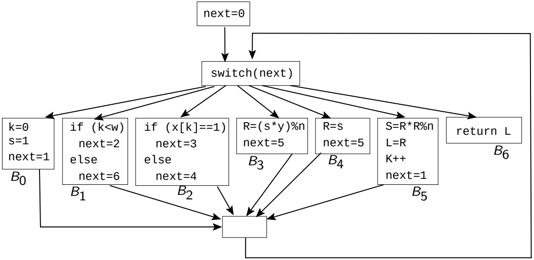
Problem

The above construction is admittedly lame, but, it's late, and it's all we could come up with. Can you think of a way to generate less conspicuous mutually dependent opaque predicates?

Algorithm OBFWHKD

p. 226

Control-flow flattening



Algorithm OBFWHKD: Control-flow flattening

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

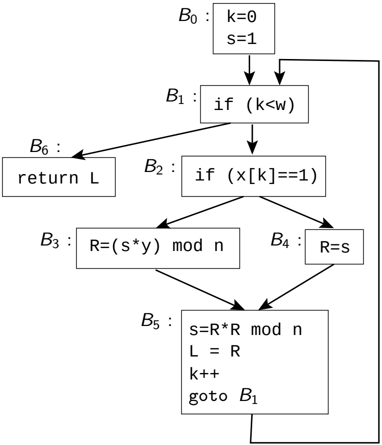
Algorithm OBFWHKD: Control-flow flattening

- 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.

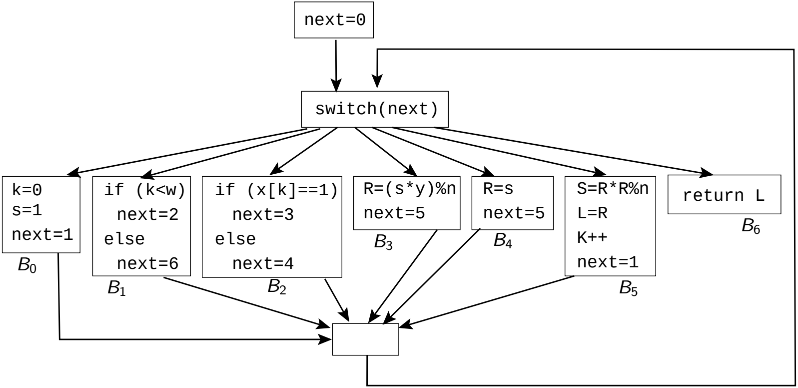
Algorithm OBFWHKD: Control-flow flattening

- 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.
- Known as *chenxify*, *chenxification*, after Chenxi Wang:

```
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; 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;
      }
}
```


next=0
switch(next)
k=0
s=1
next=1
B0
if (k<w)
next=2
else
next=6
B1
if (x[k]==1)
next=3
else
next=4
B2
R=(s*y)%n
next=5
B3
R=s
next=5
B4
S=R*R%n
L=R
K++
next=1
B5
return L
B6

Exercise: Chenxify a control-flow graph

- Consider again the control-flow graph for this 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;
    }
}
```

- Flatten the graph using Chenxification.

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.

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?

Use GCC's labels-as-values

```
int modexp(int y, int x[], int w, int n) {
    int R, L, k, s;
    char* jtab[]={&&case0,&&case1,&&case2,
                  &&case3,&&case4,&&case5,&&case6};
    goto *jtab[0];
    case0: k=0; s=1; goto *jtab[1];
    case1: if (k<w) goto *jtab[2]; else goto *jtab[6];
    case2: if (x[k]==1) goto *jtab[3]; else goto *jtab[4];
    case3: R=(s*y)%n; goto *jtab[5];
    case4: R=s; goto *jtab[5];
    case5: s=R*R%n; L=R; k++; goto *jtab[1];
    case6: return L;
}
```

Algorithm $\textsc{obf}\mathrm{WHKD}_{\mathrm{alias}}$: Control-flow flattening

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

Algorithm $\textsc{obf}\mathrm{WHKD}_{\mathrm{alias}}$: Control-flow flattening

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

Algorithm $\textsc{obf}\mathrm{WHKD}_{\mathrm{alias}}$: Control-flow flattening

- Attack against Chenxification:
  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

Algorithm OBF$\mathrm{WHKD}_{\mathrm{alias}}$: Control-flow flattening

- Attack against Chenxification:
  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

Compute 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}; 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;
        }
}
```

```
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;
      }
}
```

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

Thwarting inter-procedural analyses

- Make it hard for the adversary to build a call graph.

Thwarting inter-procedural analyses

- Make it hard for the adversary to build a call graph.
- Replace every function call with an indirect call through a pointer:

Thwarting inter-procedural analyses

- Make it hard for the adversary to build a call graph.
- Replace every function call with an indirect call through a pointer:
  1. Make every function have the same signature,

Thwarting inter-procedural analyses

- Make it hard for the adversary to build a call graph.
- Replace every function call with an indirect call through a pointer:
    1. Make every function have the same signature,
    2. create function pointer variables

Thwarting inter-procedural analyses

- Make it hard for the adversary to build a call graph.
- Replace every function call with an indirect call through a pointer:
  1. Make every function have the same signature,
  2. create function pointer variables
  3. initialize them with the addresses of functions.

Thwarting inter-procedural analyses

- Make it hard for the adversary to build a call graph.
- Replace every function call with an indirect call through a pointer:
  1. Make every function have the same signature,
  2. create function pointer variables
  3. initialize them with the addresses of functions.
  4. replace the static call with an indirect one through pointer.

Thwarting inter-procedural analyses

- Make it hard for the adversary to build a call graph.
- Replace every function call with an indirect call through a pointer:
  1. Make every function have the same signature,
  2. create function pointer variables
  3. initialize them with the addresses of functions.
  4. replace the static call with an indirect one through pointer.
- add bogus function pointers; add code that appears to call a function through a pointer, use pointer arithmetic to construct function pointers.

Algorithm

OBFWHKD$_{\text{opaque}}$

p. 250

Opaque values from array aliasing

$\textsc{obf}\mathrm{WHKD}_{\mathrm{opaque}}$: Opaque values from array aliasing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
36	58	1	46	23	5	16	65	2	41	2	7	1	37	0	11	16	2

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.

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 $\equiv 1 \bmod 5$!

```
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!

Introducing aliasing

p. 229

Introducing aliasing

- If you want to confuse static analysis — introduce spurious aliases into your program!

Introducing aliasing

- If you want to confuse static analysis — introduce spurious aliases into your program!
- Aliasing confuses both humans and analysis when performed by static analysis tools.

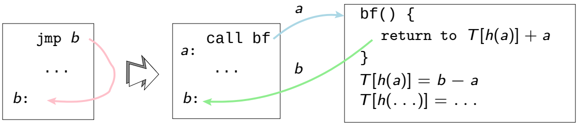
Introducing aliasing

- If you want to confuse static analysis — introduce spurious aliases into your program!
- Aliasing confuses both humans and analysis when performed by static analysis tools.
- Aliasing occurs in
  - two pointers can refer to the same memory location,
  - two reference parameters can also alias each other
  - a reference parameter and a global variable
  - two array elements indexed by different variables.

Algorithm OBFLDK

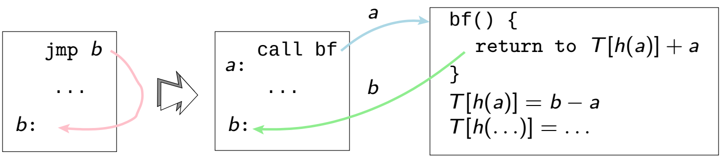
p. 239

Jumps through branch functions



OBFLDK: 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!



OBFLDK: Make branches explicit

```
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;
}
```

OBFLDK: Jumps through branch functions

- A table T stores
$$T[h(a_i)] = b_i - a_i.$$
- Code in pink updated the return address!
- The branch function:

```
char* T[2];
void bf() {
   char* old;
   asm volatile("movl 4(%%ebp),%0\n\t" : "=r" (old));
   char* new = (char*)((int)T[h(old)] + (int)old);
   asm volatile("movl %0,4(%%ebp)\n\t" : : "r" (new));
}
```

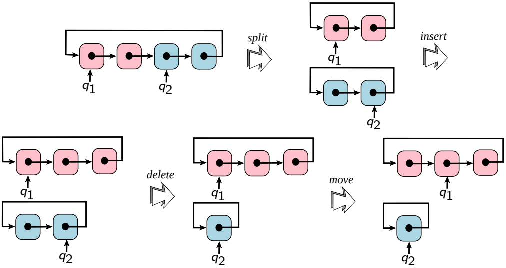
```
int modexp(int y, int x[], int w, int n) {
   int R, L; int k = 0; int s = 1;
   T[h(&&retaddr1)]=(char*)(&&endif-&&retaddr1);
   T[h(&&retaddr2)]=(char*)(&&beginloop-&&retaddr2);
   beginloop:
      if (k >= w) goto endloop;
      if (x[k] != 1) goto elsepart;
         R = (s*y) % n;
         bf();        // goto endif;
         retaddr1:
         asm volatile(".ascii \"bogus\"\n\t");
      elsepart:
         R = s;
      endif:
      s = R*R % n;
      L = R;
      k++;
      bf();          // goto beginloop;
      retaddr2:
   endloop:
   return L;
}
```

OBFLDK: 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%.

Opaque Predicates

p. 246



Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$: Opaque predicates from pointer aliasing

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

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$: 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.

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$: 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!

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$

- Creates opaque predicates from pointer analysis problems.

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$

- Creates opaque predicates from pointer analysis problems.
- The algorithm tries to go beyond the capabilities of known analysis algorithms:

  > *Despite a great deal of work on both flow-sensitive and context-sensitive algorithms [. . .], none has been shown to scale to programs with millions of lines of code, and most have difficulty scaling to 100,000 lines of code.*

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$

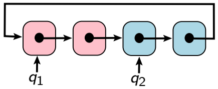
- Creates opaque predicates from pointer analysis problems.
- The algorithm tries to go beyond the capabilities of known analysis algorithms:

  > *Despite a great deal of work on both flow-sensitive and context-sensitive algorithms [. . .], none has been shown to scale to programs with millions of lines of code, and most have difficulty scaling to 100,000 lines of code.*

- Alias analysis algorithms are designed to perform well on "normal code" written by humans!

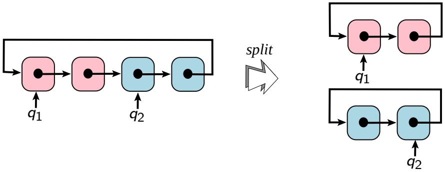
Algorithm $\text{OBFCTJ}_{\text{alias}}$

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



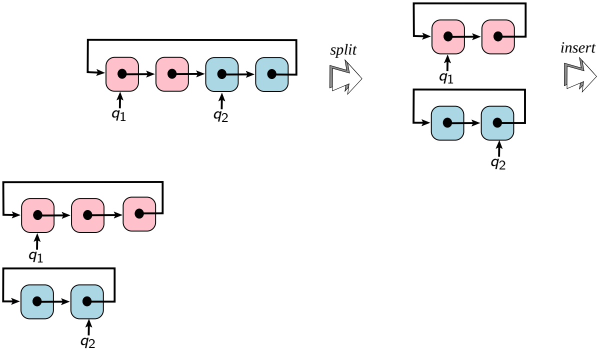
Algorithm $\text{OBFCTJ}_{\text{alias}}$

- 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):



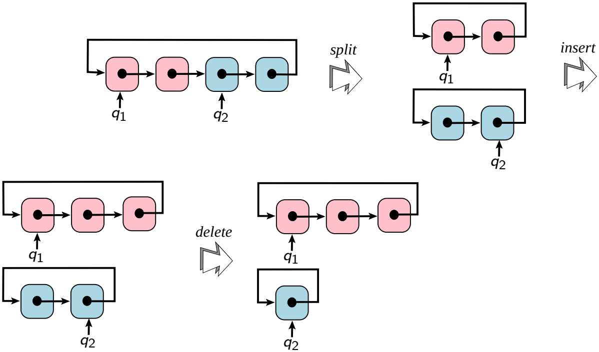
Algorithm OBFCTJ$_{\text{alias}}$

- 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):



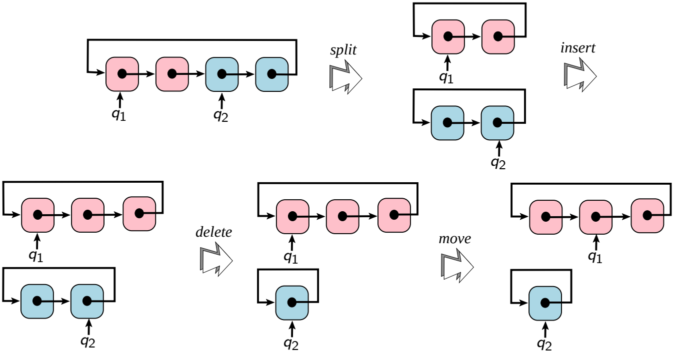
Algorithm OBFCTJ_{alias}

- 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):



Algorithm OBFCTJ$_{\text{alias}}$

- 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):



Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$

- 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."

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{alias}}$

- 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,

Algorithm $\text{OBF}\mathrm{CTJ}_{\mathrm{alias}}$

- 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.

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{pointer}}$: Opaque predicates from concurrency

- Concurrent programs are difficult to analyze statically: n statements in a parallel region can execute in $n!$ different orders.

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{pointer}}$: Opaque predicates from concurrency

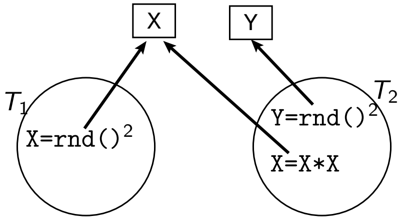
- Concurrent programs are difficult to analyze statically: n statements in a parallel region can execute in $n!$ different orders.
- Construct opaque predicates based on the difficulty of analyzing the threading behavior of programs!

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{pointer}}$: Opaque predicates from concurrency

- Concurrent programs are difficult to analyze statically: n statements in a parallel region can execute in $n!$ different orders.
- Construct opaque predicates based on the difficulty of analyzing the threading behavior of programs!
- Keep a global data structure G with a certain set of invariants I, to concurrently update G while maintaining I, and use I to construct opaque predicates over G

Algorithm $\textsc{obf}\mathrm{CTJ}_{\mathrm{pointer}}$: Opaque predicates from concurrency

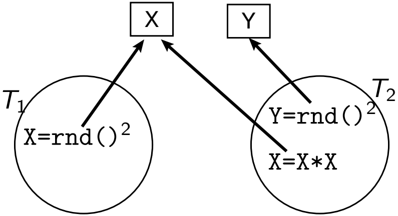
Threads T_1 and T_2 concurrently bang on two integer variables `X` and `Y`, with complete disregard for data races:



- Maintain the invariants that both `X` and `Y` will always be the square of some value.

Algorithm OBFCTJ$_{\text{pointer}}$: Opaque predicates from concurrency

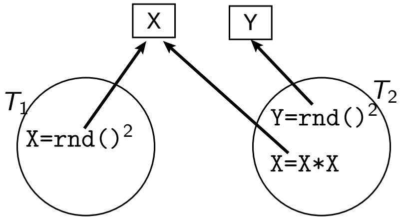
Threads T_1 and T_2 concurrently bang on two integer variables `X` and `Y`, with complete disregard for data races:



- Maintain the invariants that both `X` and `Y` will always be the square of some value.
- Construct an opaque predicate `(X−34*Y==−1)`^{F}.

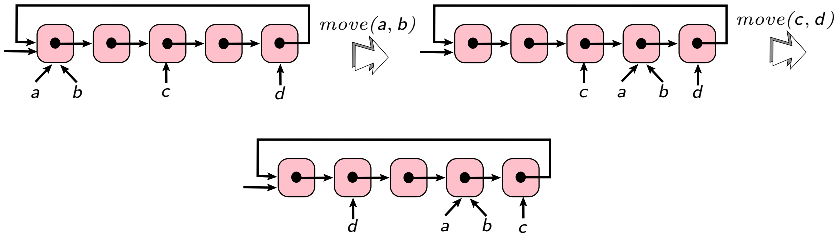
Algorithm $\text{OBFCTJ}_{\text{pointer}}$: Opaque predicates from concurrency

Threads T_1 and T_2 concurrently bang on two integer variables `X` and `Y`, with complete disregard for data races:



- Maintain the invariants that both `X` and `Y` will always be the square of some value.
- Construct an opaque predicate `(X-34*Y==-1)`^{F}.
- $\forall x, y \in \mathbb{Z} : x^2 - 34y^2 \neq -1$.

Opaque predicates from concurrency



Opaque predicates from concurrency

- Thread T_1 updates a and b, such that each time a is updated to point to its next node in the cycle, b is also updated to point to its next node in the cycle.

Opaque predicates from concurrency

- Thread T_1 updates a and b, such that each time a is updated to point to its next node in the cycle, b is also updated to point to its next node in the cycle.
- Thread T_2 updates c and d.

Opaque predicates from concurrency

- Thread T_1 updates a and b, such that each time a is updated to point to its next node in the cycle, b is also updated to point to its next node in the cycle.
- Thread T_2 updates c and d.
- Opaquely true predicate $(a = b)^T$ is statically indistinguishable from an opaquely false predicate $(c = d)^F$!

Breaking opaque predicates

Breaking opaque predicates

```
...
x₁ ← ···;
x₂ ← ···;
...
b ← f(x₁, x₂, ...);
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 $x_1, \ldots$;
4. compute the outcome of f for all input values;
5. kill the branch if $f \equiv \mathit{true}$.

Breaking opaque predicates

How can you make attacker's task more difficult?

- make it harder to locate the instructions that make up $f(x_1, x_2, \ldots)$;
- make it harder to determine what are the inputs $x_1, x_2, \ldots$ to f;
- make it harder to determine the actual ranges $R_1, R_2, \ldots$ of $x_1, x_2, \ldots$; or
- make it harder to determine the outcome of f for all possible argument values.

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 opaque predicates

How to make attacker's task more difficult? Make it harder to

- find $f(x_1, x_2, \ldots)$;
- find the inputs $x_1, x_2, \ldots$ to f;
- find the ranges $R_1, R_2, \ldots$ of $x_1, x_2, \ldots$; or
- determine the outcome of f for all argument values.

Algorithm
rePMBG

p. 256

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

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

- Mila Dalla Preda:

- Attack opaque predicates confined to a single basic block.

Algorithm REPMBG: 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.

Algorithm REPMBG: 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 beginning of basic block) is found.

Algorithm REPMBG: 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 beginning of basic block) is found.
- Brute force evaluate, or use abstract interpretation.

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

```
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 number:

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

⇨

$$
\begin{aligned}
&\texttt{x} && = \mathit{even}\,; \\
&\texttt{y = x } *_a \texttt{ x} && = \mathit{even} *_a \mathit{even} = \mathit{even}\,; \\
&\texttt{y = y } +_a \texttt{ x} && = \mathit{even} +_a \mathit{even} = \mathit{even}\,; \\
&\texttt{z = y } \%_a \texttt{ 2} && = \mathit{even} \bmod 2 = 0\,; \\
&\texttt{b = z==0;} && = \texttt{true} \\
&\texttt{if b ...} &&
\end{aligned}
$$

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`	=	$even \bmod 2 = 0$;
`b = z==0;`	=	`true`
`if b ...`		

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

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

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

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

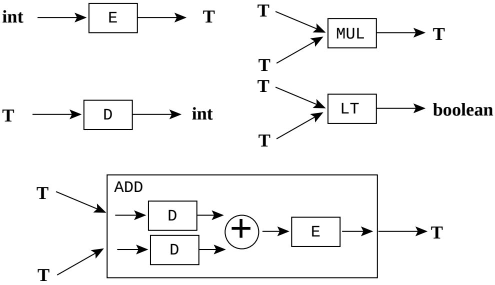
- Regardless of whether `x`'s initial value is even or odd, `b` is true!
- You've broken the opaque predicate, efficiently!!

Algorithm REPMBG: 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.

Data encodings

p. 258



Data encodings

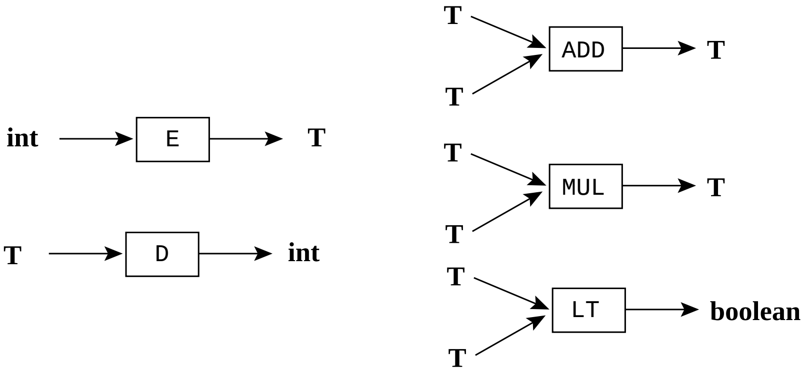
Obfuscating an Abstract Datatype

$$\left\{\begin{array}{lcl} \texttt{type}\ T & = & \cdots \\ \oplus_T : T \times T & \rightarrow & T \\ \otimes_T : T \times T & \rightarrow & T \end{array}\right.$$

$$\Downarrow$$

$$\left\{\begin{array}{lcl} \texttt{type}\ T' & = & \cdots \\ E_{T'} : T & \rightarrow & T' \\ D_{T'} : T' & \rightarrow & T \\ \oplus_{T'} : T' \times T' & \rightarrow & T' \\ \otimes_{T'} : T' \times T' & \rightarrow & T' \end{array}\right.$$

Obfuscating an Abstract Datatype



Obfuscating an ADT — Simplistic method

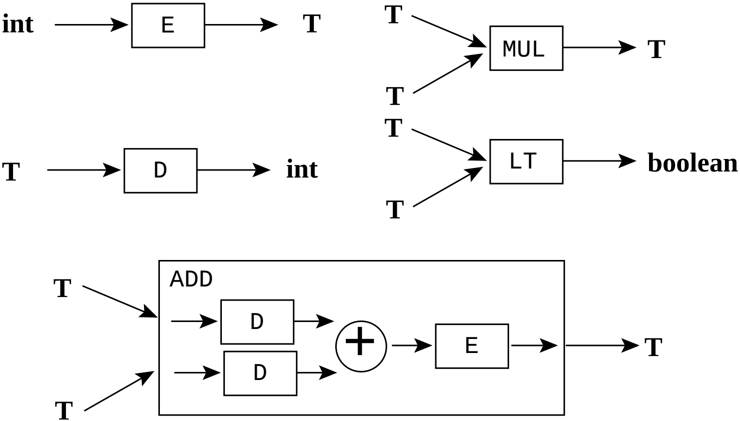
$$\left\{\begin{array}{lcl} \texttt{type}\ T & = & \cdots \\ \oplus_T : T \times T & \rightarrow & T \\ \otimes_T : T \times T & \rightarrow & T \end{array}\right.$$

$$\Downarrow$$

$$\left\{\begin{array}{lcl} x \oplus_{T'} y & = & E_{T'}(D_{T'}(x) \oplus_T D_{T'}(y)) \\ x \otimes_{T'} y & = & E_{T'}(D_{T'}(x) \otimes_T D_{T'}(y)) \end{array}\right.$$

Better if every operation is performed on the obfuscated representation directly!

Obfuscating an ADT — Simplistic method



Obfuscating an ADT — Parameterize family

- To prevent pattern matching attacks you want the obfuscated representation to be parameterized:

$$\begin{cases} \texttt{type } T'_p & = & \cdots \\ E^p_{T'} : T & \rightarrow & T'_p \\ D^p_{T'} : T'_p & \rightarrow & T \\ \oplus^p_{T'_p} : T'_p \times T'_p & \rightarrow & T'_p \\ \otimes^p_{T'_p} : T'_p \times T'_p & \rightarrow & T'_p \end{cases}$$

Multiple different representations

- The original program has three integer variables `a`, `x`, and `y`.
- You obfuscate `a` and `x` to be of type `T1` and `y` to be of type `T2`:

```
int  a  =  ···;
int  x  =  ···;
int  y  =  ···;
x  =  ···  a  ···;
y  =  ···  x  ···;
```

⇒

```
T1  a  =  ···;
T1  x  =  ···;
T2  y  =  ···;
x  =  ···  a  ···;
y  =  ···  E_T2(D_T1(x))  ···;
```

where $E_{T2}(D_{T1}(x))$

Data obfuscations

We're going to look at ways to obfuscate

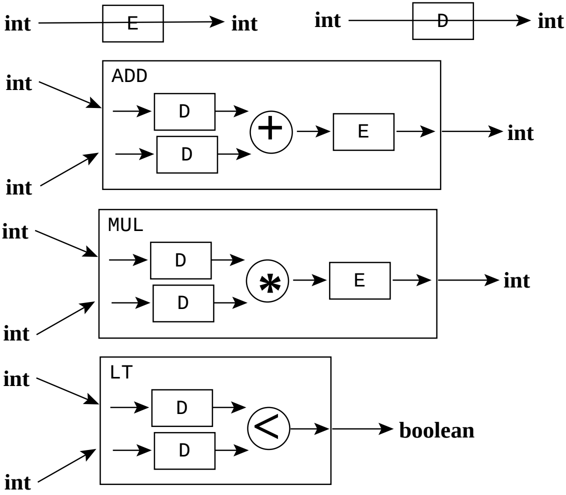
- Integers,
- Booleans,
- Strings, and
- Arrays

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);}
```

- `T1` is the data type of the obfuscated representation,
- `E1` is a function that transforms from cleartext integers into the obfuscated representation,
- `D1` transforms obfuscated integers into cleartext,
- `ADD1`, `MUL1`, and `LT1` define how to add, multiply, and compare two obfuscated integers.

Transforming Integers — The identity transformation



Transforming Integers — The identity transformation

- Add these definitions to your program and transform the code on the left into the code on the right:

```
int v = 7;
v = v * 5;
v = v + 7;
while (v<50) v++;
```

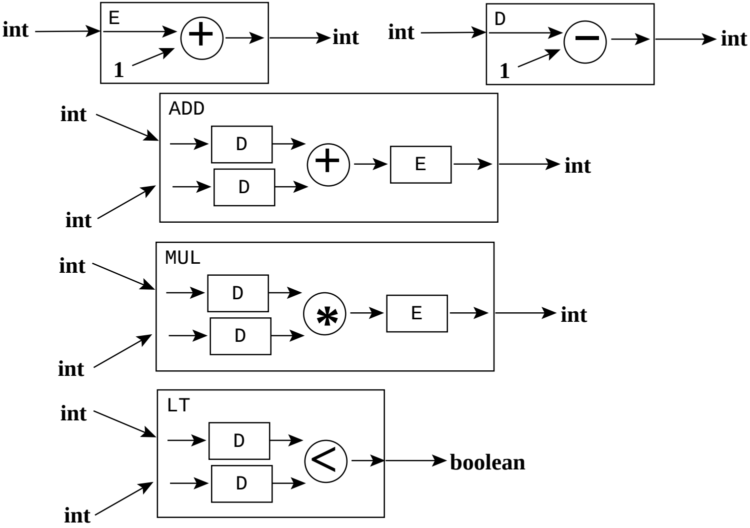
```
T1 v = E1(7);
v = MUL1(v,E1(5));
v = ADD1(v,E1(7));
while (LT1(v,E1(50)))
   v=ADD1(v,E1(1));
```

+1 transformation with deobfuscation

```
typedef int T2;
T2 E2(int e) {return e+1;}
int D2(T2 e) {return e-1;}
T2 ADD2(T2 a, T2 b) {return E2(D2(a)+D2(b));}
T2 MUL2(T2 a, T2 b) {return E2(D2(a)*D2(b));}
BOOL LT2(T2 a, T2 b) {return D2(a)<D2(b);}
```

- Bad implementation of addition and multiplication: before applying the operations we first convert to deobfuscated space.
- Watch out for overflow!

+1 transformation with deobfuscation

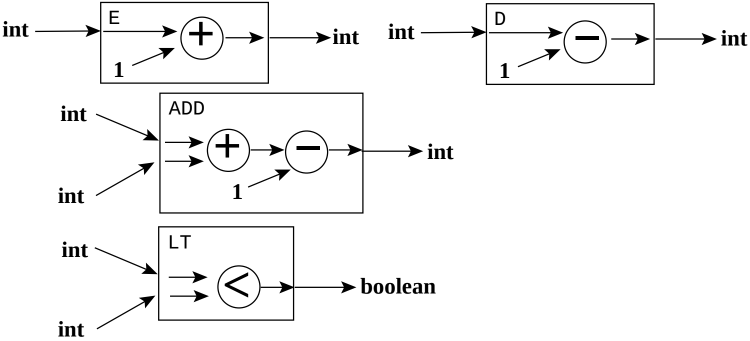


+1 transformation without deobfuscation

```
typedef int T3;
T3 E3(int e) {return e+1;}
int D3(T3 e) {return e-1;}
T3 ADD3(T3 a, T3 b) {return a+b-1;}
T3 MUL3(T3 a, T3 b) {return a*b-a-b+2;}
BOOL LT3(T3 a, T3 b) {return a<b;}
```

- Perform arithmetic operations directly on the obfuscated values.
- For $x + y$, adjust by subtracting 1, since $x + y$ in obfuscated space is $(x + 1) + (y + 1) = x + y + 2$.

+1 transformation without deobfuscation



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?

Algorithm $\textsc{obf}\mathrm{BDKMRV}_{\mathrm{num}}$: Number-theoretic tricks

```
typedef int T4;
#define N4 (53*59)
T4 E4(int e,int p) {return p*N4+e;}
int D4(T4 e) {return e%N4;}
T4 ADD4(T4 a, T4 b) {return a+b;}
T4 MUL4(T4 a, T4 b) {return a*b;}
BOOL LT4(T4 a, T4 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 of representation by choosing different values for p.

Operating on differently obfuscated integers

```
int x = 7;
int v = 6;
v = x * v;
x = v + x;
printf("%i\n",x);
```

```
T3 x = E3(7);
T4 v = E4(6,3);
v = MUL4(E4(D3(x),5),v);
x = ADD3(E3(D4(v)),E3(8));
printf("%i\n",D3(x));
```

- If two differently obfuscated integers need to be operated on, then one needs to be first deobfuscated and then re-obfuscated to the correct representation.

Algorithm OBFBDKMRV$_{crypto}$: Encrypting integers

```
DES_key_schedule ks;
DES_cblock key ={0x12,0x34,0x56,0x78...};

typedef struct {int x; int y;} T7;
T7 E7(int e) {
   T7 block = (T7){e,0};
   DES_ecb_encrypt((DES_cblock*)&block,
                   (DES_cblock*)&block,&ks,DES_ENCRYPT);
   return block;}
int D7(T7 e) {
   DES_ecb_encrypt((DES_cblock*)&e,
                   (DES_cblock*)&e,&ks,DES_DECRYPT);
   return e.x;}
T7 ADD7(T7 a,T7 b) {return E7(D7(a)+D7(b));}
T7 MUL7(T7 a,T7 b) {return E7(D7(a)*D7(b));}
BOOL LT7(T7 a,T7 b) {return D7(a)<D7(b);}
```

Algorithm $\text{OBF}\mathrm{BDKMRV}_{\mathrm{crypto}}$: Encrypting integers

- You can't perform arithmetic operations on values encrypted by DES directly!

Algorithm $\textsc{obf}\mathrm{BDKMRV}_{\mathrm{crypto}}$: Encrypting integers

- You can't perform arithmetic operations on values encrypted by DES directly!
- Decrypt the operands, perform arithmetic, re-encrypt the result $\Rightarrow$ bad.

Algorithm $\textsc{obf}\mathrm{BDKMRV}_{\mathrm{crypto}}$: Encrypting integers

- You can't perform arithmetic operations on values encrypted by DES directly!
- Decrypt the operands, perform arithmetic, re-encrypt the result $\Rightarrow$ bad.
- Overhead!

Counted loops using encryption

```
typedef struct {int x; int y;} T8;
T8 E8(T8 e) {
   DES_ecb_encrypt((DES_cblock*)&e,
                   (DES_cblock*)&e,&ks,DES_ENCRYPT);
   return e;}
T8 D8(T8 e) {
   DES_ecb_encrypt((DES_cblock*)&e,
                   (DES_cblock*)&e,&ks,DES_DECRYPT);
   return e;}
BOOL NE8(T8 a,T8 b) {return memcmp(&a,&b,sizeof(T8))!=0;}
```

- This representation only supports the *not-equal* comparison on encrypted values.

Counted loops using encryption

```
for(i=0;i<5;i++)
   printf("HERE\n");
```

```
T8 v = E8(E8(E8(E8((T8){42,42}))));
while (NE8(v,(T8){42,42})) {
   printf("HERE\n");
   v = D8(v);
}
```

- Allows you to construct simple counted loops inside the encrypted domain.
- The code in pink can be computed at obfuscation time.

RSA is *homomorphic* in multiplication

- To multiply two values that have been encrypted with RSA you just multiply the encrypted values:

$$E(x \cdot y) = E(x) \cdot E(y).$$

RSA is *homomorphic* in multiplication

- To multiply two values that have been encrypted with RSA you just multiply the encrypted values:

$$E(x \cdot y) = E(x) \cdot E(y).$$

- RSA definition:

$$\begin{aligned} C &= M^e \bmod n \\ M &= C^d \bmod p \\ n &= pq \\ ed &= 1 \bmod (p-1)(q-1) \end{aligned}$$

RSA is *homomorphic* in multiplication

- To multiply two values that have been encrypted with RSA you just multiply the encrypted values:

$$E(x \cdot y) = E(x) \cdot E(y).$$

- RSA definition:

$$\begin{aligned} C &= M^e \bmod n \\ M &= C^d \bmod p \\ n &= pq \\ ed &= 1 \bmod (p-1)(q-1) \end{aligned}$$

- M is the cleartext message, C the cryptotext, e is the *public modulus*, d is the *private modulus*, p and q are primes.

RSA is *homomorphic* in multiplication

- To multiply two values that have been encrypted with RSA you just multiply the encrypted values:

$$E(x \cdot y) = E(x) \cdot E(y).$$

- RSA definition:

$$\begin{aligned} C &= M^e \bmod n \\ M &= C^d \bmod p \\ n &= pq \\ ed &= 1 \bmod (p-1)(q-1) \end{aligned}$$

- M is the cleartext message, C the cryptotext, e is the *public modulus*, d is the *private modulus*, p and q are primes.
- RSA is homomorphic in multiplication:

$$(M_1^e \bmod n) \cdot (M_2^e \bmod n) = (M_1^e \cdot M_2^e) \bmod n = (M_1 \cdot M_2)^e \bmod n.$$

RSA is *homomorphic* in multiplication

```
typedef int T9;
#define M 33
T9 E9(int e) {return (((e*e)%M)*e)%M;}
int D9(T9 e) {int t=(((e*e)%M)*e)%M;
              return (((t*t)%M)*e)%M;}
T9 ADD9(T9 a,T9 b) {return E9((D9(a)+D9(b)));}
T9 MUL9(T9 a,T9 b) {return (a*b)%M;}
BOOL LT9(T9 a,T9 b) {return D9(a)<D9(b);}
```

- The modulus `M` is 33, the public exponent 3, the private exponent 7.
- You can only represent numbers smaller than the modulus.
- RSA is *not* homomorphic in addition!

Transform addition in counted loops to use multiplication.

```
for(i=1;i<6;i++)
   printf("HERE\n");
```

```
for(i=E9(1);i!=E9(32);i=MUL9(i,E9(2)))
    printf("HERE\n");
```

- It's easy to transform simple counted for-loops to use multiplication
- You never have to deobfuscate the loop variable, unless its value is used inside the loop.

Encoding literal data

- Literal data often carries much semantic information:
  - `"Please enter your password:"`
  - `0xA17BC97A7E5F...FF67` (maybe a cryptographic key???)

Encoding literal data

- Literal data often carries much semantic information:
  - `"Please enter your password:"`
  - `0xA17BC97A7E5F...FF67` (maybe a cryptographic key???)
- Split up in pieces.

Encoding literal data

- Literal data often carries much semantic information:
  - `"Please enter your password:"`
  - `0xA17BC97A7E5F...FF67` (maybe a cryptographic key???)
- Split up in pieces.
- Xor with a constant.

Encoding literal data

- Literal data often carries much semantic information:
  - `"Please enter your password:"`
  - `0xA17BC97A7E5F...FF67` (maybe a cryptographic key???)
- Split up in pieces.
- Xor with a constant.
- Avoid ever reconstituting the literal in cleartext! (What about `printf`?)

Encoding literal data

- Literal data often carries much semantic information:
  - `"Please enter your password:"`
  - `0xA17BC97A7E5F...FF67` (maybe a cryptographic key???)
- Split up in pieces.
- Xor with a constant.
- Avoid ever reconstituting the literal in cleartext! (What about `printf`?)
- Print each character one at a time?

Convert literals to code — Mealy machine

- Encode the strings `"MIMI"` and `"MILA"` in a finite state transducer (a *Mealy machine*)

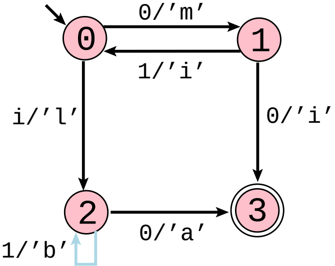
Convert literals to code — Mealy machine

- Encode the strings `"MIMI"` and `"MILA"` in a finite state transducer (a *Mealy machine*)
- The machine takes a bitstring and a state transition table as input and and generates a string as output.

Convert literals to code — Mealy machine

- Encode the strings `"MIMI"` and `"MILA"` in a finite state transducer (a *Mealy machine*)
- The machine takes a bitstring and a state transition table as input and and generates a string as output.
- `Mealy(`10_2`)` produces `"MIMI"`.
- `Mealy(`110_2`)` produces `"MILA"`.

Convert literals to code — Mealy machine




```
int next[][2] =
      {{1,2},
       {3,0},
       {3,2}};
char out[][2] =
      {{'m','l'},
       {'i','i'},
       {'a','b'}};
```

- $s_0 \xrightarrow{i/o} s_1$ means in state s_0 on input i transfer to state s_1 and produce an o.
- next[*state*][*input*]=*next state*
- out[*state*][*input*]=*output*

Mealy machine — table driven

```
char* mealy(int v) {
   char* str=(char*)malloc(10);
   int state=0,len=0;
   while (state!=3) {
      int input = 1&v; v >>= 1;
      str[len++]=out[state][input];
      state = next[state][input];
   }
   str[len]='\0';
   return str;
}
```

Mealy machine — hardcoded

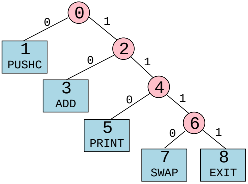
```
char* mealy(int v) {
   char* str=(char*)malloc(10);
   int state=0,len=0;
   while (1) {
      int input = 1&v; v >>= 1;
      switch (state) {
          case 0: state=(input==0)?1:2;
                  str[len++]=(input==0)?'m':'l'; break;
          case 1: state=(input==0)?3:0;
                  str[len++]='i'; break;
          case 2: state=(input==0)?3:2;
                  str[len++]=(input==0)?'a':'b'; break;
          case 3: str[len]='\0'; return str;
      }
   }
}
```

Breaking abstractions

Algorithm OBFAJV

p. 293

Modifying instruction encodings



Algorithm OBFAJV: Modifying instruction encodings

- Obfuscate the instruction set architecture!

Algorithm OBFAJV: Modifying instruction encodings

- Obfuscate the instruction set architecture!
- Now, can't run on the bare metal.

Algorithm OBFAJV: Modifying instruction encodings

- Obfuscate the instruction set architecture!
- Now, can't run on the bare metal.
- But, we can run on a virtual machine!

Algorithm OBFAJV: Modifying instruction encodings

- Produce diverse programs by generating a unique interpreter and a unique instruction set for every distributed copy of a program.

Algorithm OBFAJV: Modifying instruction encodings

- Produce diverse programs by generating a unique interpreter and a unique instruction set for every distributed copy of a program.
- Every program should encode instructions differently, and for this encoding to change at runtime.

Algorithm OBFAJV: Modifying instruction encodings

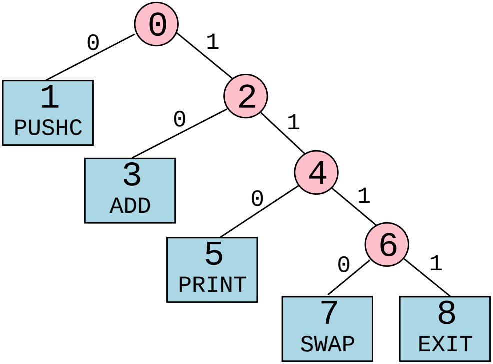
- Produce diverse programs by generating a unique interpreter and a unique instruction set for every distributed copy of a program.
- Every program should encode instructions differently, and for this encoding to change at runtime.
- The attacker will find that the instruction encoding change as he chooses different execution paths!

Example code

```
PUSHC 2
PUSHC 5
ADD
PRINT
SWAP 0
PUSHC 2
PUSHC 5
ADD
PRINT
EXIT
```

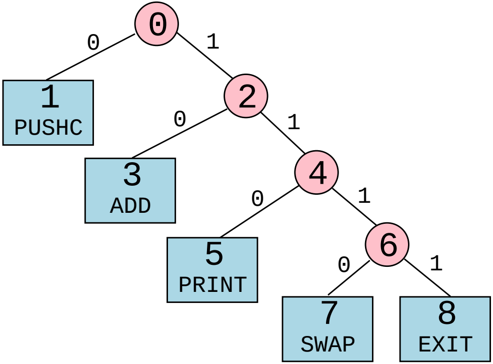
- PUSHC c pushes the 3-bit constant c,
- ADD adds the two top elements on the stack,
- PRINT prints and pops the top element on the stack, and
- EXIT stops execution.
- SWAP n mean "from here on, the instruction set changes." n is the node in the *instruction decoding tree*.

Instruction decoding tree



- Internal nodes (pink) point to left and right subtrees,
- Leaves (blue) contain references to the code that implements the opcode semantics.

Instruction decoding tree



- Internal nodes (pink) point to left and right subtrees,
- Leaves (blue) contain references to the code that implements the opcode semantics.
- $\langle 0, 0, 1, 1, 1, 1, 0 \rangle \Rightarrow$ PUSHC $\langle 0, 1, 1 \rangle$, PRINT.

How to diversify?

- To diversify:
  1. generate a decoding three

How to diversify?

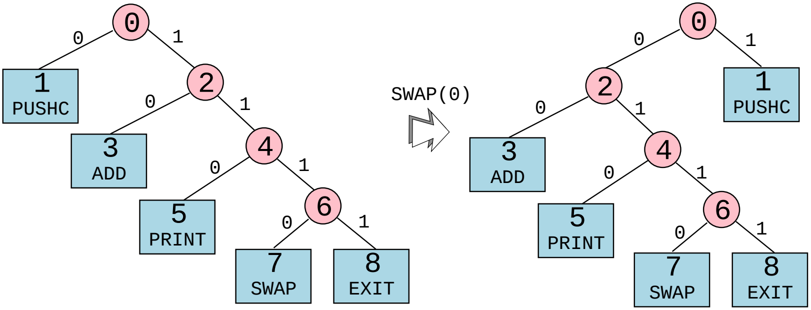
- To diversify:
  1. generate a decoding three
  2. translate each instruction from the original program into the new encoding.

How to diversify?

- To diversify:
  1. generate a decoding three
  2. translate each instruction from the original program into the new encoding.
- Resulting encoding is variable length!

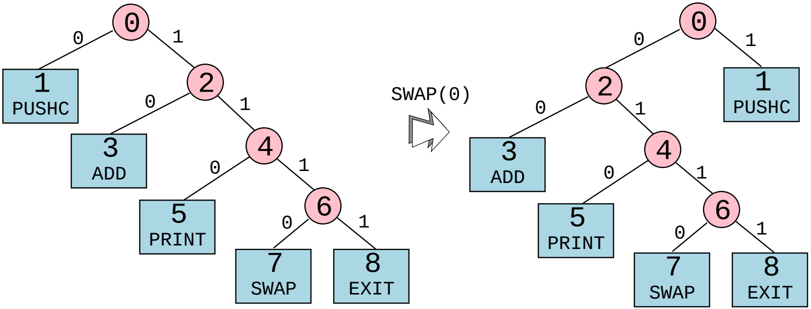
Changing encoding at runtime!

- The SWAP instruction changes the instruction encoding on the fly, at runtime!



Changing encoding at runtime!

- The SWAP instruction changes the instruction encoding on the fly, at runtime!
- SWAP *n* just swaps the children of node *n*.



$$\overbrace{0,}^{\texttt{PUSHC}}\ \overbrace{0,\ 1,\ 0,}^{\texttt{2}}\ \overbrace{0,}^{\texttt{PUSHC}}\ \overbrace{1,\ 0,\ 1,}^{\texttt{5}}\ \overbrace{1,\ 0,}^{\texttt{ADD}}\ \overbrace{1,\ 1,\ 0,}^{\texttt{PRINT}}$$

$$\overbrace{1,\ 1,\ 1,\ 0,}^{\texttt{SWAP}}\ \overbrace{0,\ 0,\ 0}^{\texttt{0}}$$

$$\overbrace{1,}^{\texttt{PUSHC}}\ \overbrace{0,\ 1,\ 0,}^{\texttt{2}}\ \overbrace{1,}^{\texttt{PUSHC}}\ \overbrace{1,\ 0,\ 1,}^{\texttt{5}}$$

$$\overbrace{0,\ 0,}^{\texttt{ADD}}\ \overbrace{0,\ 1,\ 0,}^{\texttt{PRINT}}\ \overbrace{0,\ 1,\ 1,\ 1}^{\texttt{EXIT}}$$

```
class Node{}
class Internal extends Node{
   public Node left,right;
   public Internal(Node left, Node right) {
      this.left = left; this.right=right;
   }
   public void swap() {
      Node tmp = left; left=right; right=tmp;
   }
}
class Leaf extends Node {
   public int operator;
   public Leaf(int operator) {
      this.operator=operator;
   }
}
```

```
static Node[] tree = new Node[9];
static {
   tree[1]=new Leaf(0);     // PUSHC
   tree[3]=new Leaf(1);     // ADD
   tree[5]=new Leaf(2);     // PRINT
   tree[7]=new Leaf(3);     // SWAP
   tree[8]=new Leaf(4);     // EXIT
   tree[6]=new Internal(tree[7],tree[8]);
   tree[4]=new Internal(tree[5],tree[6]);
   tree[2]=new Internal(tree[3],tree[4]);
   tree[0]=new Internal(tree[1],tree[2]);
}
```

```
static int prog[]={0,0,1,0,0,1,0,1,1,0,1,1,0,
                   1,1,1,0,0,0,0,
                   1,0,1,0,1,1,0,1,0,0,0,1,0,0,1,1,1};
static int pc = 0;
```

```
static int decode() {
   Node t = tree[0];
   while (t instanceof Internal)
      t = (prog[pc++]==0)?((Internal)t).left
                         :((Internal)t).right;
   return ((Leaf)t).operator;
}
static int operand() {
   return 4*prog[pc++]+2*prog[pc++]+prog[pc++];
}
```

```
static void interpret() {
   int stack[] = new int[10]; int sp = -1;
   while (true) {
      switch (decode()) {
         case 0 : stack[++sp]=operand();
                  break; // PUSHC
         case 1 : stack[sp-1]+=stack[sp]; sp--;
                  break; // ADD
         case 2 : System.out.println(stack[sp--]);
                  break; // PRINT
         case 3 : ((Internal)tree[operand()]).swap();
                  break; // SWAP
         case 4 : return; // EXIT
      }
   }
}
```

Algorithm obfAJV: Modifying instruction encodings

- Attack: find the interpreter, ignore the changes to encodings!

Algorithm OBFAJV: Modifying instruction encodings

- Attack: find the interpreter, ignore the changes to encodings!
- Must make sure that every instruction semantics is different.

Algorithm OBFAJV: Modifying instruction encodings

- Attack: find the interpreter, ignore the changes to encodings!
- Must make sure that every instruction semantics is different.
- Merge several instructions into new ones with unique and unknown semantics.

Algorithm OBFAJV: Modifying instruction encodings

- Attack: find the interpreter, ignore the changes to encodings!
- Must make sure that every instruction semantics is different.
- Merge several instructions into new ones with unique and unknown semantics.
- The authors of report slowdown factors of between 50 and 3500.