Outline

Code obfuscation — It's elusive!

- Hard to pin down exactly what obfuscation is
- Hard to devise practically useful algorithms
- Hard to evaluate the quality of these algorithms.

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

Code obfuscation — Overview

1. Simple obfuscating transformations.
2. How to design an obfuscation tool.
3. Definitions.
4. Control-flow transformations.
5. Data transformations.
6. Abstraction transformations.
7. Constructing opaque predicates.
8. Dynamic obfuscating transformations.

Outline

Algorithm OBFTP: Identifier renaming

- Java released 1996:
  - decompilation is easy!
  - compiled code ⇔ source!
- Hans Peter Van Vliet
  1. released Crema a Java obfuscator.
  2. released Mocha Java decompiler.
  3. RIP
- It's an obfuscator/decompiler war!
  1. HoseMocha kills Mocha (add an instruction after `return`);
  2. Rename identifiers using characters that are legal in the JVM, but not in Java source.

Renaming Example

```
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 f1(
     int x1,int x2[],
      int x3,int x4) {
   int x5, x6;
   int x7 = 0;
   int x8 = 1;
   while (x7 < x3) {
      if (x2[x7] == 1)
         x5 = (x8*x1)%x4;
      else
         x5 = x8;
      x8 = x5*x5%x4;
      x6 = x5;
      x7++;
   }
   return x6;
}
```


Identifier renaming

- Historical interest.
- Decompiler can't recover information which has been removed!
- Identifier renaming ⇒ no performance overhead!

Algorithm obfTP

- In an object-oriented language:
  - Use overloading!
  - Give as many declarations as possible the same name!
- Algorithm by Paul Tyma:

  - Used in PreEmptive Solutions' Dash0 Java obfuscator.
  - Licensed by Microsoft for Visual Studio

Algorithm obfTP

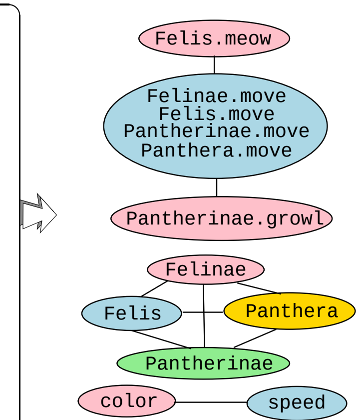
- Java naming rules:
  1. Class names should be globally unique,
  2. Field names should be unique within classes
  3. Methods with different signatures can have the same name.
- Algorithm
  1. Build a graph:
     - nodes are declarations
     - edges between nodes that cannot have the same name
  2. Merge methods that must have the same name (because they override each other) into super-nodes.
  3. Color the graph with the smallest number of colors (=names)!

Algorithm obfTP: Original program

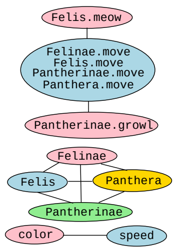
```
class Felinae {
   int color;
   int speed;
   public void move(int x,int y){}
}
class Felis extends Felinae {
   public void move(int x,int y){}
   public void meow(int tone,int length){}
}
class Pantherinae extends Felinae {
   public void move(int x,int y){}
   public void growl(int tone,int length){}
}
class Panthera extends Pantherinae {
   public void move(int x,int y){}
}
```


Algorithm obfTP: Interference graph

```
class Felinae {
   int color;
   int speed;
   void move(int x,int y)
}
class Felis extends Felinae {
   void move(int x,int y){}
   void meow(int tone,int len)
}
class Pantherinae extends Felinae{
   void move(int x,int y){}
   void growl(int tone,int len)
}
class Panthera extends Pantherinae{
   void move(int x,int y)
}
```



Algorithm OBFTP: Renamed program




```
class Pink {
   int Pink;
   int Blue;
   public void Blue(int x,int y){}
}
class Blue extends Pink {
   public void Blue(int x,int y){}
   public void Pink(int tone,int len){}
}
class Green extends Pink {
   public void Blue(int x,int y){}
   public void Pink(int tone,int len){}
}
class Yellow extends Green {
   public void Blue(int x,int y){}
}
```


Outline

Complicating control flow

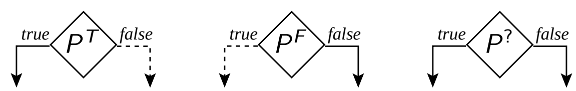
- Transformations that make it difficult for an adversary to analyze the flow-of-control:
  1. insert bogus control-flow,
  2. flatten the program
  3. hide the targets of branches to make it difficult for the adversary to build control-flow graphs
- None of these transformations are immune to attacks,

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*

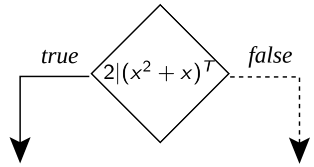
- Notation:
  - P^T for an *opaquely true* predicate
  - P^F for an *opaquely false* predicate
  - $P^?$ for an *opaquely indeterminate* predicate
  - $E^{=v}$ for an *opaque* expression of value v
- Graphical notation:



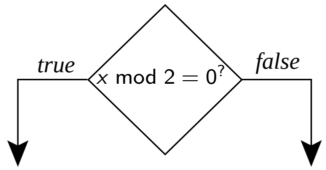
- Building blocks for many obfuscations.

Opaque Expressions

- An opaquely true predicate:



- An opaquely indeterminate predicate:



Simple Opaque Predicates

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

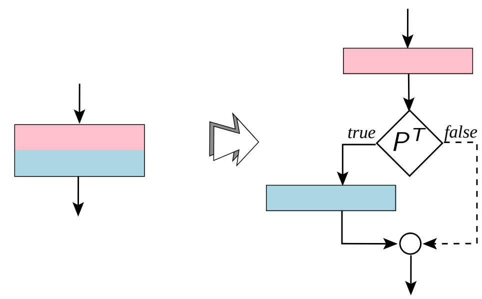
  "Show that $\forall x, y \in \mathbb{Z} : p(x, y)$"
- $\forall x, y \in \mathbb{Z} : x^2 - 34y^2 \neq 1$
- $\forall x \in \mathbb{Z} : 2|x^2 + x$
- ...

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
- The resilience reduces to the resilience of the opaque predicates.

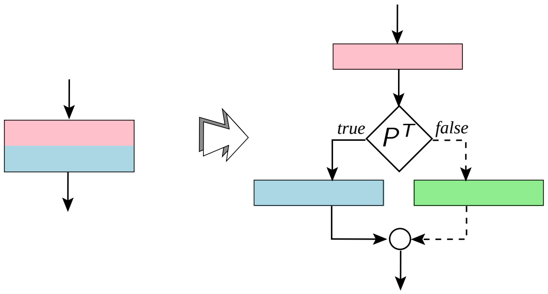
Algorithm $\text{OBFCTJ}_{\text{bogus}}$: Inserting bogus control-flow

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



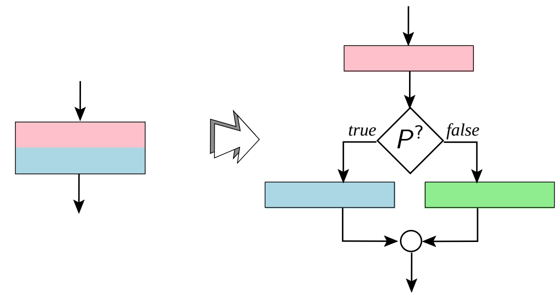
Algorithm OBFCTJ$_{bogus}$: Inserting bogus control-flow

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



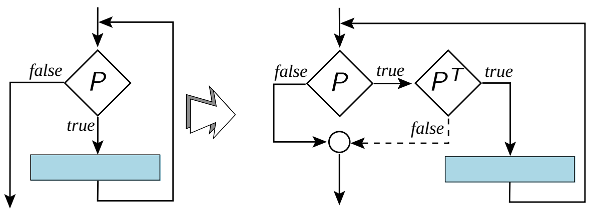
Algorithm OBFCTJ$_{bogus}$: Inserting bogus control-flow

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



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

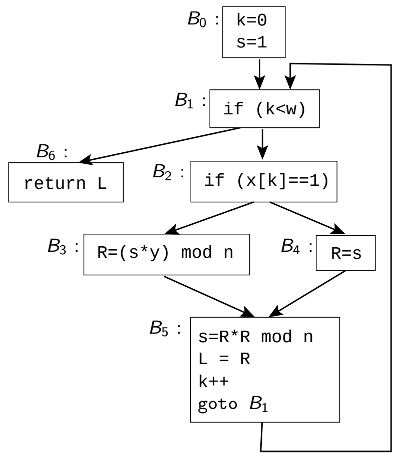
- Extend a loop condition P by conjoining it with an opaquely true predicate P^T:



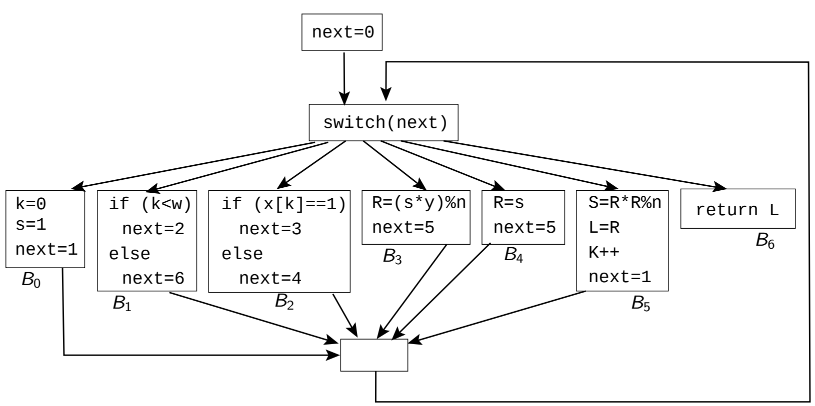
Algorithm OBFWHKD: Control-flow flattening

- 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;
      }
}
```



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.

Algorithm $\text{OBFWHKD}_{\text{alias}}$: 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;
      }
}
```

Modify the array at runtime!

A function that rotates an array one step right:

```
void permute(int g[], int n, int* m) {
   int i;
   int tmp=g[n-1];
   for(i=n-2; i>=0; i--) g[i+1] = g[i];
   g[0]=tmp;
   *m = ((*m)+1)%n;
}
```

- Make static array aliasing analysis harder for the attacker!
- Modify the array at runtime!

```
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);
   }
}
```

Make the array global!

```
int g[20]; int m;
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=g[m+0]%g[m+1]; break;
         case 1 : if (k<w) next=g[m+g[m+2]];
                  else next=g[m+0]-2*g[m+2]; break;
         case 2 : if (x[k]==1) next=g[m+3]-g[m+2];
                  else next=2*g[m+2]; break;
         case 3 : R = (s*y)%n; next=g[m+4]+g[m+2]; break;
         case 4 : R=s; next=g[m+0]-g[m+3]; break;
         case 5 : s = R*R%n; L=R; k++;
                  next=g[m+g[m+4]]%g[m+2]; break;
         case 6 : return L;
      }
}
```


With the array global you can initialize it differently at different call sites:

```
g[0]=10; g[1]=9; g[2]=2; g[3]=5; g[4]=3; m=0;
modexp(y, x, w, n);
...
g[5]=10; g[6]=9; g[7]=2; g[8]=5; g[9]=3; m=5;
modexp(y, x, w, n);
```

Sprinkle pointer variables (pink), pointer manipulations (blue), dead code (green) over the program:

```
int modexp(int y, int x[], int w, int n) {
   int R, L, k, s; int next=0;
   int g[] = {10,9,2,5,3,42}; int* g2; int* gr;
   for(;;)
      switch(next) {
      case 0 : k=0; g2=&g[2]; s=1; next=g[0]%g[1];
               gr=&g[5]; break;
      case 1 : if (k<w) next=g[*g2];
               else next=g[0]-2*g[2]; break;
      case 2 : if (x[k]==1) next=g[3]-*g2;
               else next=2**g2; break;
      case 3 : R=(s*y)%n; next=g[4]+*g2; break;
      case 4 : R=s; next=g[0]-g[3]; break;
      case 5 : s=R*R%n;L=R;k++; next=g[g[4]]%*g2;break;
      case 6 : return L;
      case 7 : *g2=666; next=*gr%2; gr=&g[*g2]; break;
      }
}
```

Algorithm OBFWHKD$_{alias}$

- Hopefully, because of the obfuscated manipulations the attacker's static analysis will conclude that nothing can be deduced about `next`.
- Not knowing `next`, he can't rebuild the CFG.
- Symbolic execution? We know next starts at 0...

OBFWHKD$_{opaque}$: Opaque values from array aliasing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
36	58	1	46	23	5	16	65	2	41	2	7	1	37	0	11	16	2	21

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 $\equiv 1 \bmod 5$!

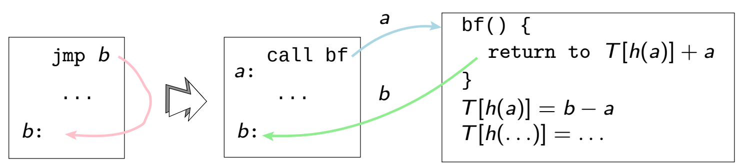

```
int g[] = {36,58,1,46,23,5,16,65,2,41,
           2,7,1,37,0,11,16,2,21,16};

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!

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%.

Outline

Constructing opaque predicates

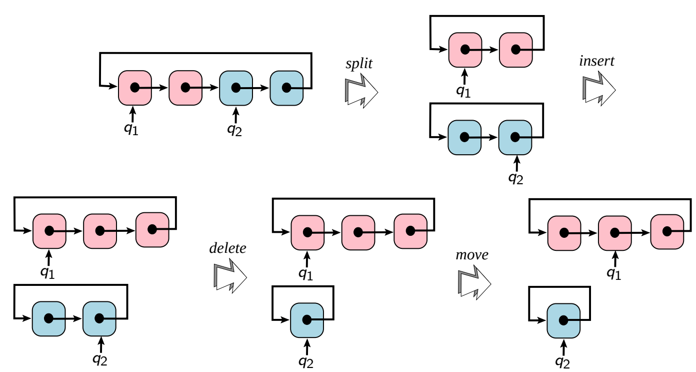
- Construct them based on
  - number theoretic results
    - $\forall x, y \in \mathbb{Z} : x^2 - 34y^2 \neq 1$
    - $\forall x \in \mathbb{Z} : 2 | x^2 + x$
  - the hardness of alias analysis
  - the hardness of concurrency analysis
- Protect them by
  - making them hard to find
  - making them hard to break
- If your obfuscator keeps a table of predicates, your adversary will too!

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

- 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):



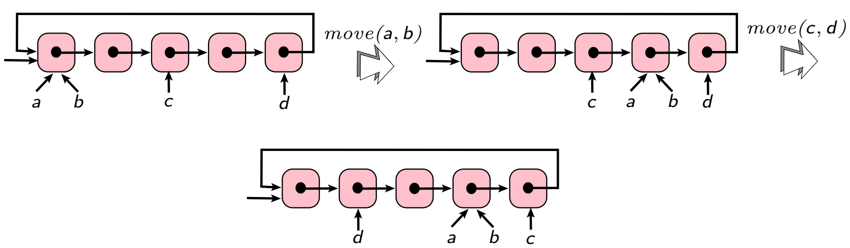
Algorithm $\mathrm{OBFCTJ}_{\mathrm{alias}}$

- 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 $\mathrm{OBFCTJ}_{\mathrm{pointer}}$: 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

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.
- Thread T_2 updates c and d.
- Opaquely true predicate $(a = b)^T$ is statically indistinguishable from an opaquely false predicate $(c = d)^F$!

Outline

1. Introduction
2. Identifier renaming
3. Complicating control flow
   - Inserting bogus control-flow
   - Control-flow flattening
   - Opaque values from array aliasing
   - Jumps through branch functions
4. Opaque Predicates
   - Opaque predicates from pointer aliasing
5. **Data encodings**
6. Dynamic Obfuscation
   - Self-Modifying State Machine
   - Code as key material
7. Discussion

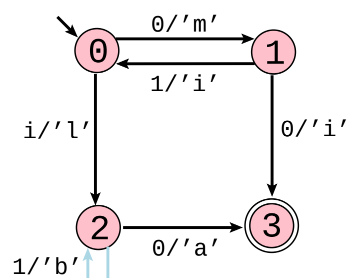
Encoding literal data

- 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*)
- 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;
      }
   }
}
```


Outline

Static vs. Dynamic obfuscation

- Static obfuscations transform the code prior to execution.
- Dynamic algorithms transform the program at runtime.
- Static obfuscation counter attacks by static analysis.
- Dynamic obfuscation counter attacks by dynamic analysis.

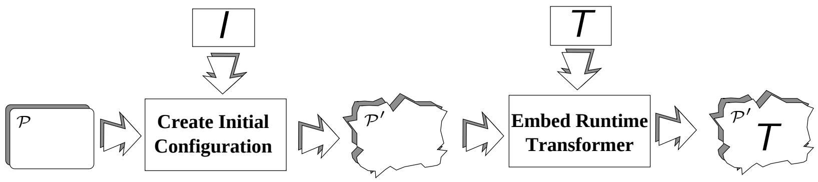
Static vs. Dynamic obfuscation

- Statically obfuscated code: the attacker sees *the same* mess every time.
- Dynamic obfuscated code: the execution path changes as the program runs.
- Some algorithms are "semi-dynamic" — they perform a small, constant number of transformations (often one) at runtime
- Some algorithms are *continuous*: the code is in constant flux.

Dynamic Obfuscation: Definitions

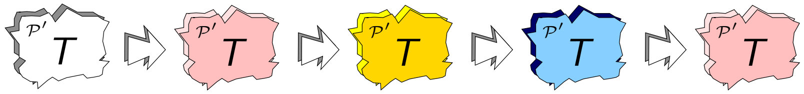
- A dynamic obfuscator runs in two phases:
  1. At compile-time transform the program to an initial configuration and add a runtime code-transformer.
  2. At runtime, intersperse the execution of the program with calls to the transformer.
- A dynamic obfuscator turns a "normal" program into a self-modifying one.

Modeling dynamic obfuscation — compile-time



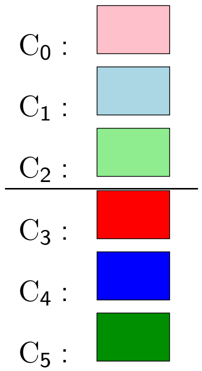
- Transformer I creates $\mathcal{P}$'s initial configuration.
- T is the runtime obfuscator, embedded in $\mathcal{P}'$.

Modeling dynamic obfuscation — runtime



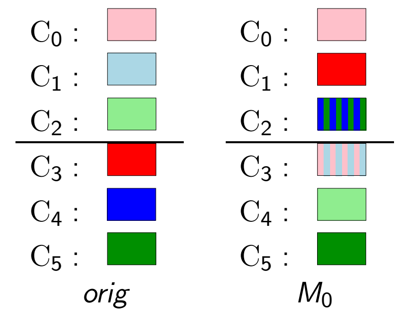
- Transformer T continuously modifies $\mathcal{P}'$ at runtime.
- We'd like an infinite, non-repeating series of configurations.
- In practice, the configurations repeat.

Dynamic obfuscation: Aucsmith's algorithm

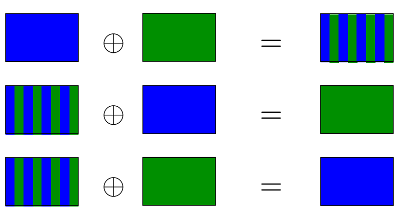


- A function is split into cells.
- The cells are divided into two regions in memory, upper and lower.

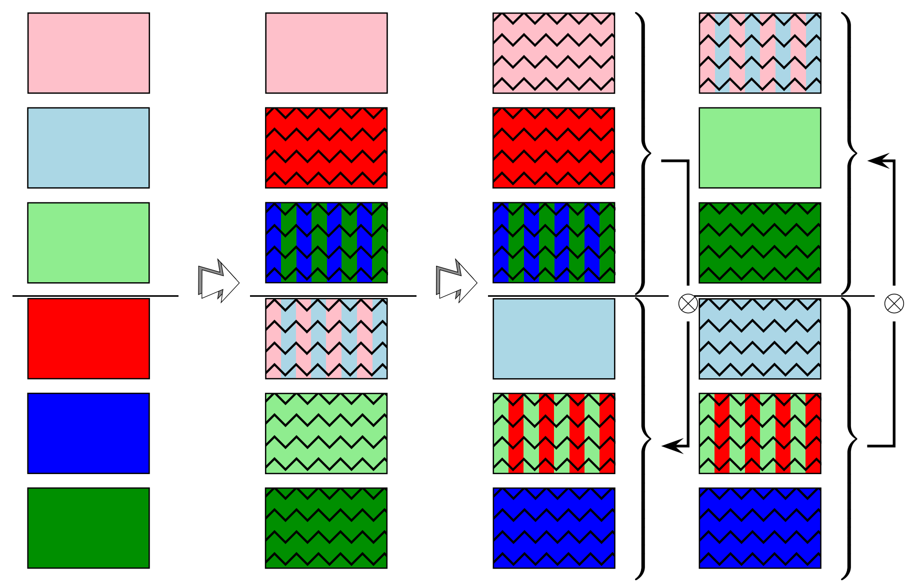
One step



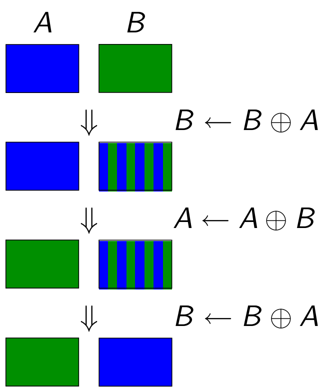
XOR!



The Dynamic Primitive — Aucsmith



Why does this work?



obfCKSP: Code as key material

- Encrypt the code to keep as little code as possible in the clear at any point in time during execution.
- Extremes:
  1. Decrypt the next instruction, execute it, re-encrypt it, ... ⇒ only one instruction is ever in the clear!
  2. Decrypt the entire program once, prior to execution, and leave it in cleartext. ⇒ easy for the adversary to capture the code.

obfCKSP: Code as key material

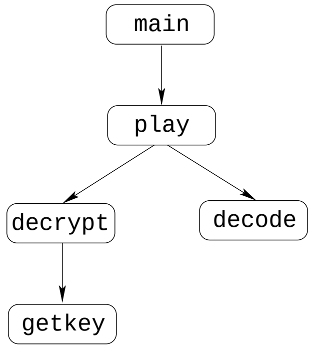
- The entire program is encrypted — except for `main`.
- Before you jump to a function you decrypt it.
- When the function returns you re-encrypt it.
- On entry, a function first encrypts its caller.
- Before returning, a function decrypts its caller.
- ⇒ At most two functions are ever in the clear!

obfCKSP: Code as key material

- What do we use as key? The code itself!
- What cipher do we use? Something simple!

OBFCKSP: Code as key material

- In the simplest case the call-graph is tree-shaped:



- Before and after every procedure cally you insert calls to a guard function that decrypts/re-encrypts the callee, using a hash of the cleartext of the caller as key.
- On entrance and exit of the callee you encrypt/decrypt the caller using a hash of the cleartext of the callee as key.

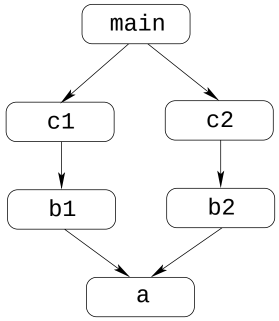

```
int player_main (int argc, char *argv[]) {
   int user_key = 0xca7ca115;
   int digital_media[] = {10,102};
   guard(play,playSIZE,player_main,player_mainSIZE);
   play(user_key,digital_media,2);
   guard(play,playSIZE,player_main,player_mainSIZE);
}
int getkey(int user_key) {
   guard(decrypt,decryptSIZE,getkey,getkeySIZE);
   int player_key = 0xbabeca75;
   int v = user_key ^ player_key;
   guard(decrypt,decryptSIZE,getkey,getkeySIZE);
   return v;
}
int decrypt(int user_key, int media) {
   guard(play,playSIZE,decrypt,decryptSIZE);
   guard(getkey,getkeySIZE,decrypt,decryptSIZE);
   int key = getkey(user_key);
   guard(getkey,getkeySIZE,decrypt,decryptSIZE);
   int v = media ^ key;
   guard(play,playSIZE,decrypt,decryptSIZE);
   return v;
}
```

```
float decode (int digital) {
   guard(play,playSIZE,decode,decodeSIZE);
   float v = (float)digital;
   guard(play,playSIZE,decode,decodeSIZE);
   return v;
}
void play(int user_key, int digital_media[], int len) {
   int i;
   guard(player_main,player_mainSIZE,play,playSIZE);
   for(i=0;i<len;i++) {
      guard(decrypt,decryptSIZE,play,playSIZE);
      int digital = decrypt(user_key,digital_media[i]);
      guard(decrypt,decryptSIZE,play,playSIZE);

      guard(decode,decodeSIZE,play,playSIZE);
      printf("%f\n",decode(digital));
      guard(decode,decodeSIZE,play,playSIZE);
   }
   guard(player_main,player_mainSIZE,play,playSIZE);
}
```

```
void crypto (waddr_t proc,uint32 key,int words) {
   int i;
   for(i=1; i<words; i++) {
      *proc ^= key;
      proc++;
   }
}

void guard (waddr_t proc,int proc_words,
            waddr_t key_proc,int key_words) {
   uint32 key = hash1(key_proc,key_words);
   crypto(proc,key,proc_words);
}
```

OBFCKSP: Code as key material

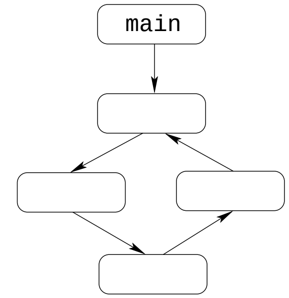
- So, what if the call-graph is shaped like a DAG, like this:



  What key to use to decrypt `a`?
- We can't use the cleartext of the caller as key, because now there are two callers!
- Let the callers' callers(`c1` and `c2`) do the decryption using a combination of the *ciphertexts* of `b1` and `b2`.

OBFCKSP: Code as key material

- What if the program is recursive?



- Keep the entire cycle in cleartext. . . .

Outline

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

Common Obfuscating Transformations

- Many obfuscating transformations are built on some simple general operations:
  - Splitting/Merging
  - Duplication
  - Reordering
  - Mapping
  - Indirection
- Apply these basic operations to
  - Control structures
  - Data structures
  - Abstractions

Static VS. Dynamic Obfuscation

- Static obfuscations confuse static analysis.
- Dynamic obfuscations confuse static and dynamic analysis.
  - the code segment is treated as code *and* data
- Dynamic algorithms generate self-modifying code. Bad for performance:
  1. flush instruction pipeline
  2. write data caches to memory
  3. invalidate instruction caches