

# Software Security

## Obfuscation (2)

Guosheng Xu,  
guoshengxu@bupt.edu.cn

# 基本内容

- ◆ 1、简单的混淆转换
- ◆ 2、标识符重命名
- ◆ 3、控制流混淆
- ◆ 4、不透明谓词
- ◆ 5、结构混淆
- ◆ 6、替换指令

# 1、简单的混淆转换

# 1、简单的混淆转换（1）

## 1) 等价表达式

理论上，有无限多个方法把一个数学表达式分解成一系列基本算术操作。例如，编译器就通常会把乘法/除法指令分解成一些列更短的或执行起来更快的指令。

例子：如果被乘数是一个常数，编译器一般会把这个乘法指令分解成一串更不容易被看懂的加法和移位指令，如下所示：

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



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

# 1、简单的混淆转换（2）

## 2) 重新排列代码和数据

在逆向工程中，仅仅观察代码的排列顺序，破解者就很可能得到重要提示。因而随机摆放程序中的各个模块、函数、语句和指令在一定程度上可以对抗逆向分析。

重排各个模块和函数的位置是很简单的，但在重排语句和指令的顺序之前，代码混淆器必须先进行数据/控制依赖分析。只有在确认两条指令之间不存在任何依赖关系时，才能对它们进行重排。在Java这类安全语言中，同一作用域中变量声明语句间的顺序是可以任意改变的。但在C语言里，由于可以利用指针，而且数组越界访问不会被禁止，所以，即使是重排变量声明语句也要慎重。

例子：程序员无意中越界访问了数组a中的元素，但程序仍然能够工作

```
int main( ) {  
    int a[5];  
    int b[6];  
    a[5] = 42;  
    printf( "%i\\n", b[0] );  
}
```

如果使用的混淆器交换了数组a和b声明语句出现的位置，代码马上就会莫名地崩溃掉。

# 1、简单的混淆转换（3）

## 3) 分解和合并函数

程序员会使用抽象这个方法对付复杂而有巨大的程序。把它分解成若干个小函数，再把相关的函数组织在同一个模块或类中，并把相关的内组织在包里。在结构混淆中将介绍通过破坏这类程序内部结构，以达到代码混淆目的的技术。

函数**内联**是把被调用函数的代码直接嵌入到主调函数的函数体中。由于这种技术同时也抹除了由程序员创建的函数间的边界，因此也可以算是一种有效的代码混淆技术。

函数**外抽**是把函数中的某段代码抽取出来，单独封装成一个函数，并把原来的代码改成对新函数的调用。因为它往程序的内部组织结构中加入了一个原本不存在的、冗余的函数调用关系，所以也可以算是一种有效的代码混淆技术。

内联实际上就是把主调函数和被调函数合并在一起，而外抽则是把一个函数分解成两部分。如果被合并的是两个纯函数（不存在副作用的函数），只要合并两个函数的参数列表及函数中的代码，并把所有对原来两个函数的调用替换成对合并后新函数的调用即可。

# 1、简单的混淆转换（4）

例子：函数合并。f是有副作用的函数，g是纯函数。

合并后的新函数fg有一个额外参数which。这个参数是为了确保只有当程序调用原f函数时，原来属于f的代码才会被执行。

```
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' ) ;
}
```



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

# 1、简单的混淆转换（5）

例子：函数分解。模n取幂函数中被加上浅灰色底色的部分被单独提取出来，变成了一个新的函数f，原代码被改成了调用新函数f的代码。

```
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 ;
}
```



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

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 ;
}
```



# 1、简单的混淆转换（6）

## 4) 复制代码

复制代码是一种常用的代码混淆技术，它把被混淆程序撑大。如果能使被复制的代码看上去与原来的代码完全不同，就能在一定程度上对抗逆向分析。

例子：复制函数f，得到一个冗余函数f1。然后对f1进行混淆处理——改变了对数组中元素的访问方法，添加了一些不影响函数语义的冗余代码。

```
float foo[100];

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

int main ( ) {
    f ( 42, 42.0 ) ;
    f ( 6, 7.0 ) ;
}
```



```
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 ) ;
    f ( 6, 7.0 ) ;
}
```

# 1、简单的混淆转换（7）

## 5) 解释器

编译时，会把程序由高级语言的源码转换成物理机或虚拟机执行的二进制代码。在这一过程中，可以自己定义一套指令集，并提供能处理这个指令集代码的专用虚拟机，然后把源码编译成与这套指令集对应的代码。

实际上，**插入解释器**相当于给程序加虚拟机壳，有基于switch的、直接调用式索引、直接索引、间接索引几种虚拟机壳。

例子：将模n的取幂函数转化成直接索引型的代码

在转化之前，要根据程序定制指令集：其中pusha、pushv、add、mul、mod、jump、store是通用指令，可以用任意一个解释器解释运行，inc\_k、x\_k、ne\_1、k\_ge\_w为“超算符”，是专门为这个模n取幂函数定制的。

这些超算符的作用有两个：一是通过减少需要执行的指令的数目，以达到提高程序性能的目的；另一个是即使针对同一个函数，也能产生许多使用不同指令集的虚拟机代码。

# 1、简单的混淆转换（8）

转换后代码如下所示：

```
int modexp ( int y, int x[ ], int w, int n) {
    int R, L, k = 0, s = 1 ;
    int Stack[10] ; int sp = 0 ;
    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,
        &&store,
        // 跳转到if语句后
        &&jump, &prog[21] ,
        // R = s ;
        && pusha, &R, &&pushv, &s, &&store,
        // s = R * R%n ;
        &&pusha, &s, &&pushv, &R, &&pushv, &R, &&mul, &&pushv, &n, &&mod,
        &&store,
        // L = R ;
        && pusha, &L, &&pushv, &R, &&store,
```

# 1、简单的混淆转换（9）

```

    && pusha, &L, &&pushv, &R, &&store,
    // k++
    &&inc_k ,
    // 跳转到循环的顶部
    &&jump, &prog[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++ ;
}

```

## 2、标识符重命名

## 2、标识符重命名 (1)

对于Java这样很容易被反编译回源码的语言，一种对付反编译器的常用技巧是用Java源码中非法的，但在JVM虚拟机里却是合法的字符重命名标识符。这样抹除的信息是反编译器不可恢复的，而且不会给程序性能带来额外开销。

算法：在面向对象的语言中，利用重载功能，用最少的名字重命名所有的类、域和方法

- 1) 画出P的继承层次图；
- 2) 创建一个空的无向图G，并把程序中每个类、域以及所有没有重写的标准库中的方法都添加为G的一个结点；
- 3) 给每对类结点之间都添加一条边；
- 4) 给属于同一个类的每对域结点之间都添加一条边；
- 5) 合并任何一对相互重写的方法结点；
- 6) 给签名相同的方法对 $C_1m_1$ 和 $C_2m_2$ 之间加一条边，其中 $C_1=C_2$ ，或 $C_2$ 直接或间接地继承自 $C_1$ ；
- 7) 用最少的颜色对图G的结点进行着色，是尽可能多的结点颜色相同。

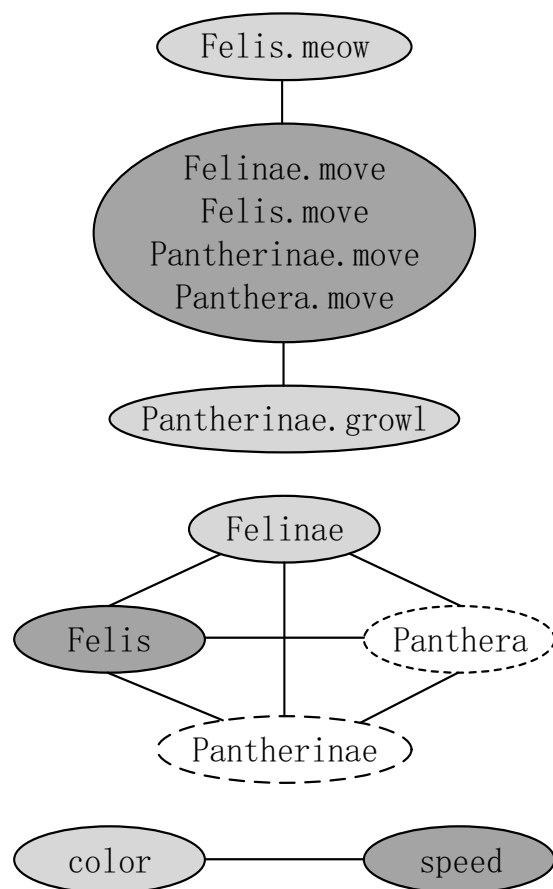
## 2、标识符重命名 (2)

例：利用算法对程序进行标识符重命名，原程序如下所示

```
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 ) {}  
}
```

## 2、标识符重命名 (3)

画出程序的继承图，重命名标识符后的代码如下所示：



```

class Light {
    int Light ;
    int Dark ;
    public void Dark( int x, int y ) {}
}
class Dark extends Light {
    public void Dark( int x, int y ) {}
    public void Light( int tone, int length ) {}
}
class Dashed extends Light {
    public void Dark( int x, int y ) {}
    public void Light( int tone, int length ) {}
}
class Dotted extends Dashed {
    public void Dark( int x, int y ) {}
}
  
```



# 3、控制流混淆

◆ 3.1、压扁控制流

◆ 3.2、插入多余的控制流

◆ 3.3、通过跳转函数执行无条件转移指令

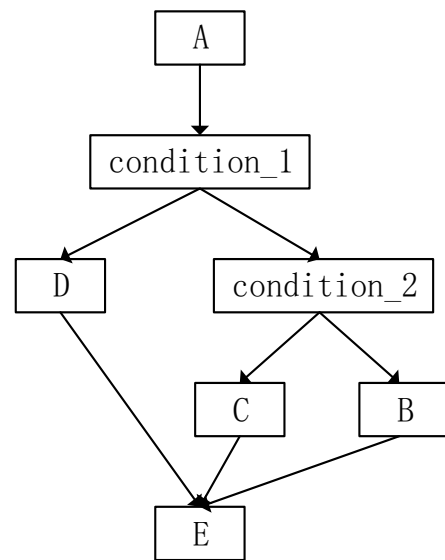
### 3、控制流混淆

控制混淆是一种基本的混淆方式，其原理是采用各种技术手段来隐藏或修改程序真正的控制流程，即通过改变程序的判断条件，对程序结构和执行路径的调整，或向程序中添加不透明谓词等方法来增加程序的复杂度，增加反编译程序的难度，从而阻止攻击者分析程序的控制流程。

一般的，控制流分析从**识别基本块**开始。基本块就是一个指令序列，中间没有跳转指令，最后一条是跳转指令，通过这条跳转指令跳转到其他的基本块。

控制流程图（Control Flow Graph）是由节点和有向边构成的，节点就是基本块，边表示了基本块之间可能的跳转。

```
basic block A
if(condition_1)
{
    if(condition_2)
        basic block B
    else
        basic block C
}
else
{
    basic block D
}
basic block E
```

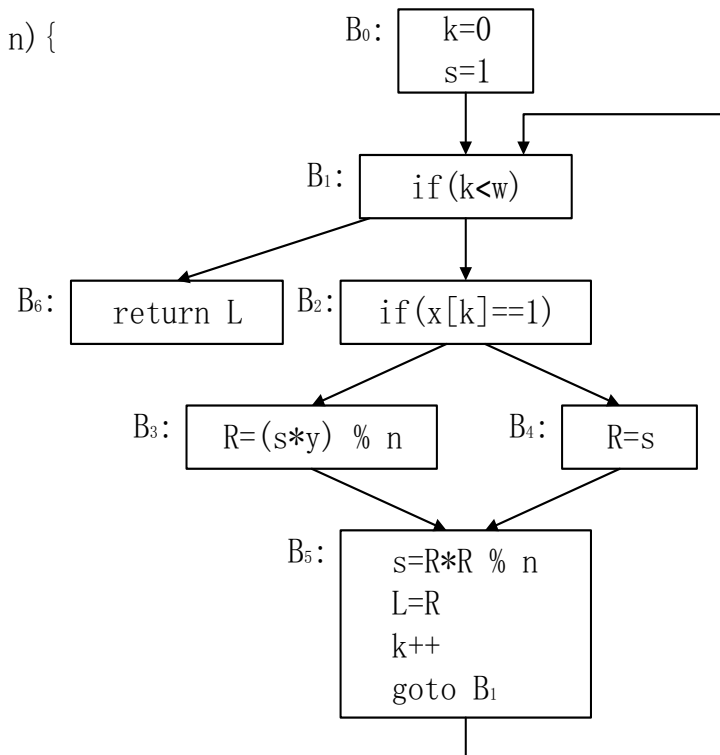


## 3.1、压扁控制流

压扁控制流是指通过“压扁”嵌套的循环和条件转移语句，重新组织程序中原有的控制流结构。这一过程还可以称为“chenxify”，源自这一算法的发明者Chenxi Wang。

例：将模n的取幂函数进行控制流压扁处理，代码及其控制流程图如下所示：

```
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
}
```



## 3.1、压扁控制流

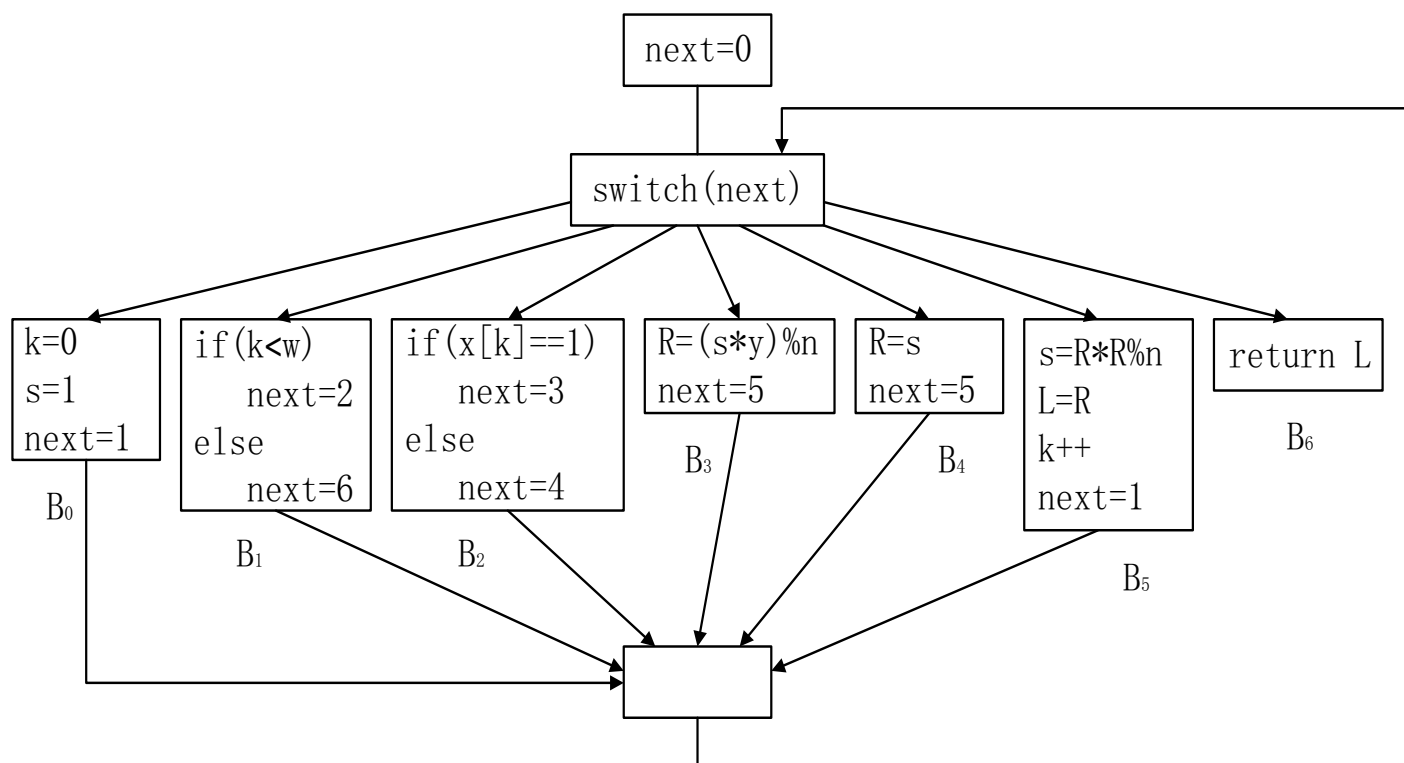
把控制流图中的各个基本块全部放到一个switch语句中，然后再把这个switch语句封装到一个死循环里，变换后的代码如下：

通过不断在各个基本块中更新变量next的值，维护正确的控制流结构。运行时控制流仍会以正确的顺序流经各个基本块，但控制流图的结构已被彻底地破坏掉了。

```
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;  
        }  
}
```

## 3.1、压扁控制流

变换后的控制流图如下所示。各个基本块中已经失去了明确记载接下来应该跳转到哪个基本块去执行的信息，破解者只能记录哪些基本块曾经被执行过，这无疑加重了破解的负担。



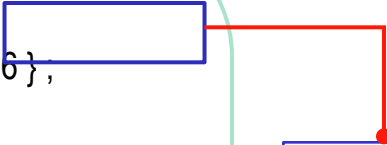
## 3.1、压扁控制流：性能评估

- 算法会带来较大的性能损失——for循环至少增加了一个跳转，switch语句中要增加一个对next变量的“多余”检查，以及一个用跳转表实现的间接跳转。
- 在实践中，出于程序性能的考虑，一般不会把整个控制流图全部压扁。
- 如果函数中有个循环会被频繁地执行，那么可以把这个循环归约为一个结点，然后再行压扁。这样，循环中的各个基本块仍能集中在同一个case语句中，保持原有结构和执行效率不变，而不会被打散到各个case语句块中，引发较大的性能开销。
- 根据所使用的编程语言，可以找出一个执行效率较高的for-switch实现方法。
- 在处理用Java这类支持异常处理的语言编写的代码时，“chenxify”会比较困难。因为这类代码中，所有可能抛出异常的语句（包括除数可能为零的表达式，或对一个可能是空指针的指针解引用）都会引出一条指向最近的catch块的边（如果函数中没有使用try-catch语句，那就指向函数的结尾处），因而它们的控制流图会比正常的控制流图多许多边。这样复杂的控制流图给压扁工作平添了不少麻烦。使用分块封装的“chenxify”结构可以解决这一问题——try和catch语句块会被分别放到两个case语句块中，并最终被放到for-switch块的相应位置中。

## 3.1、压扁控制流

例子：使用C语言时，可以如下优化。利用gcc中的标签变量，构造了一个能直接跳转到下一个基本块中去的跳转表。

```
int modexp( int y, int x[ ], int w, int n )
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 ;
}
```



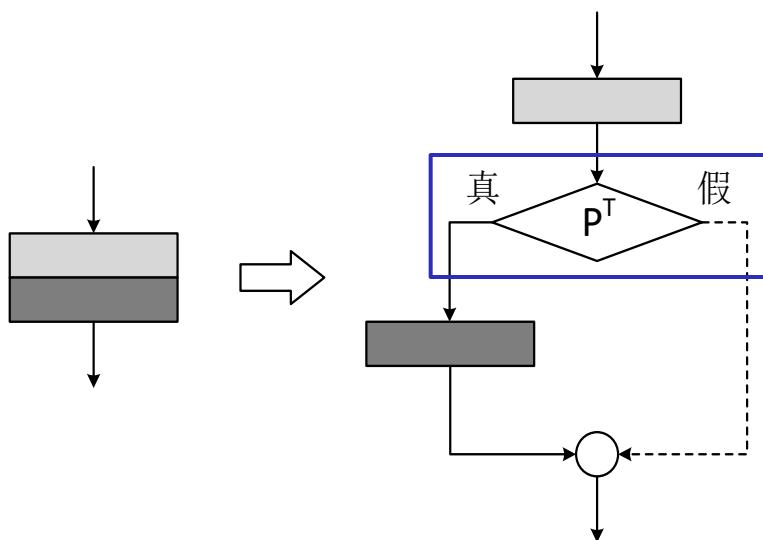
取当前函数  
中标号地址，  
得到的值的  
类型是  
“void\*”。

## 3.2、插入多余的控制流

插入多余的控制流是指通过分裂基本块、增加循环条件或增加冗余跳转等方式，使控制流结构复杂化，提高破解者重建原有控制流结构的难度。

(1) 分裂基本块有三种基本转换：

1) 插入总会被执行的冗余代码

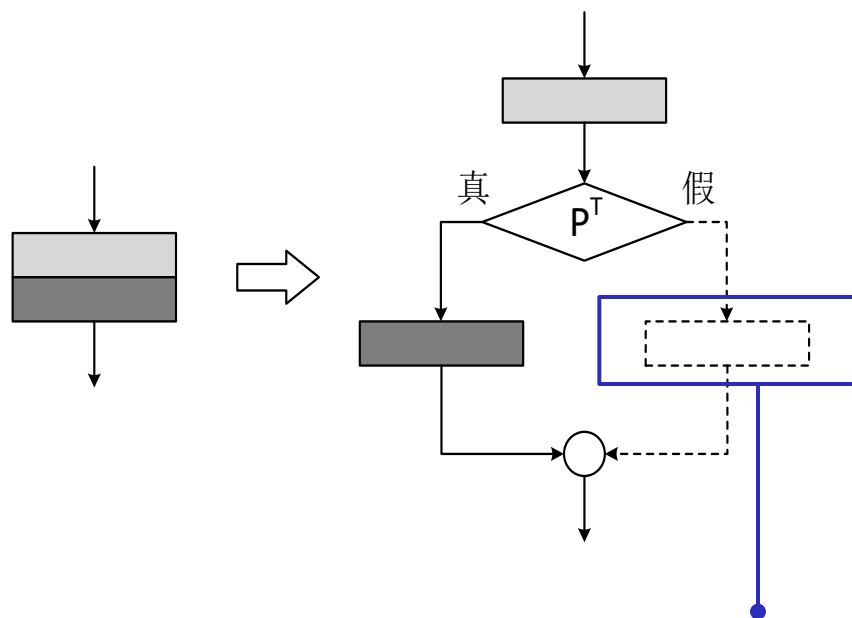


在控制流图中，左边的基本块看上去只是在某些情况下才会被执行到，而实际上它每次都会被执行。



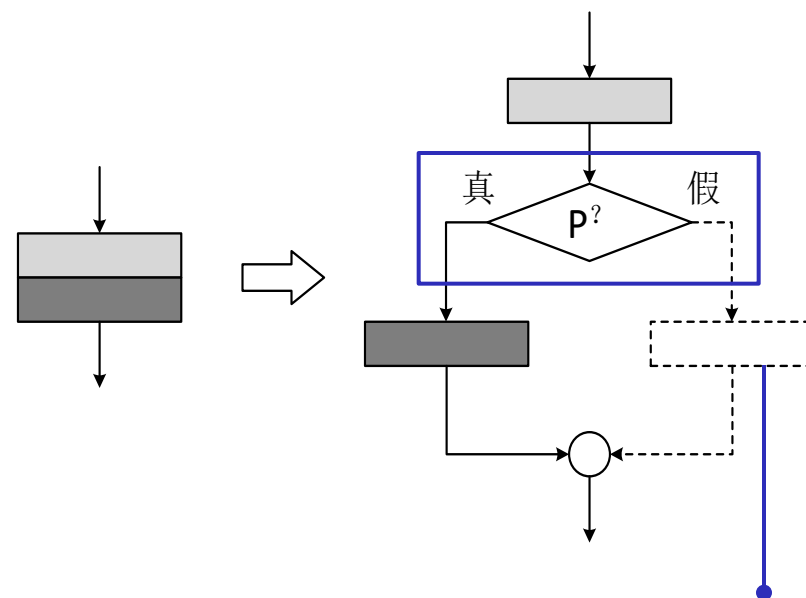
## 3.2、插入多余的控制流

2) 插入不会被执行的“死代码”



$P^T$  透明谓词。可以写任意代码，或者将左边基本块的代码复制过来做一些小的随机变动，能极大地保证混淆代码的隐蔽性。

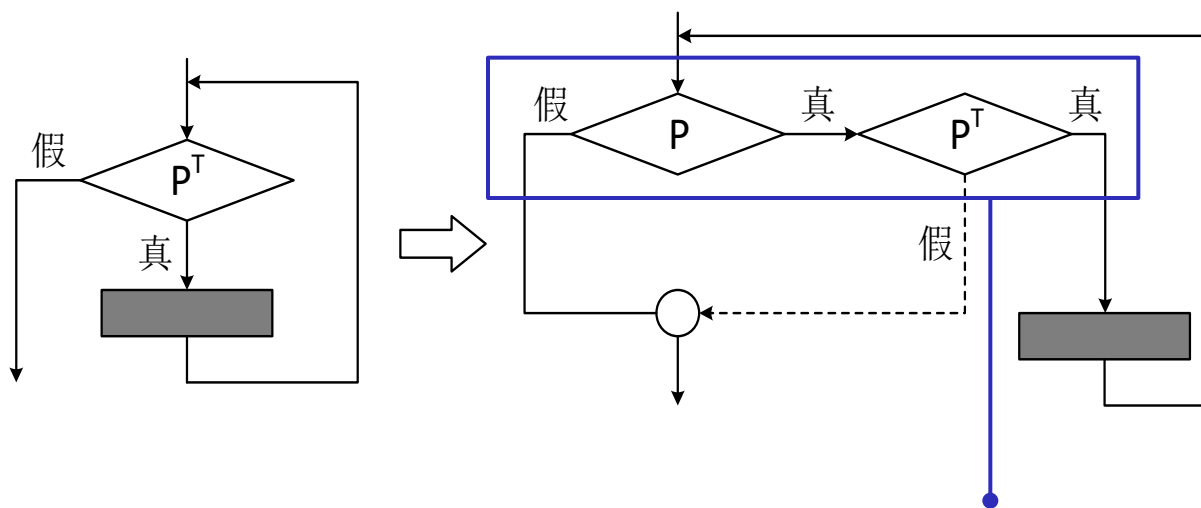
3) 插入不一定被执行的代码



与左边基本块的语义是等价的。可以直接把左边基本块的代码复制过来，然后对两个基本块各自使用不同的混淆方法进行混淆。

## 3.2、插入多余的控制流

2) 循环条件插入变换：在循环条件P中插入多余的条件



程序看上去必须在 $P \wedge P^T$ 为真时，才会继续运行，但实际上， $P^T$ 是个多余的条件。

也可以用类似的方法插入 $P^F$ ，使程序看上去必须在 $P \wedge P^F$ 为假时，才会退出。

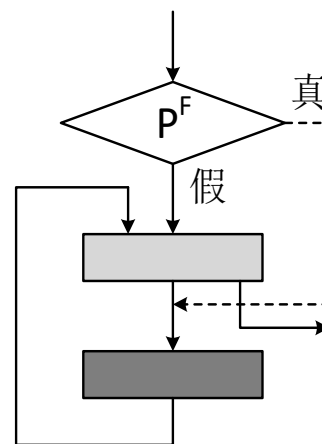
## 3.2、插入多余的控制流

3) 将可归约的控制流图转换成不可归约的控制流图：往程序中增加冗余的跳转，让它直接跳转到循环的中间去

如果按照结构化编程的规则，用嵌套的if-、for-、while-、repeat-、case-语句编程，程序的控制流图就是可归约的。对这类控制流图进行数据流分析是简单而有效的。如果在程序中加上跳转，让它直接调到某个循环的内部，这样循环就有了2个入口，这类控制流图是不可归约的。对这类控制流图进行数据流分析的过程就会变得非常复杂，而且通常要先把不可归约图转化成可归约图才能进行进一步的分析。

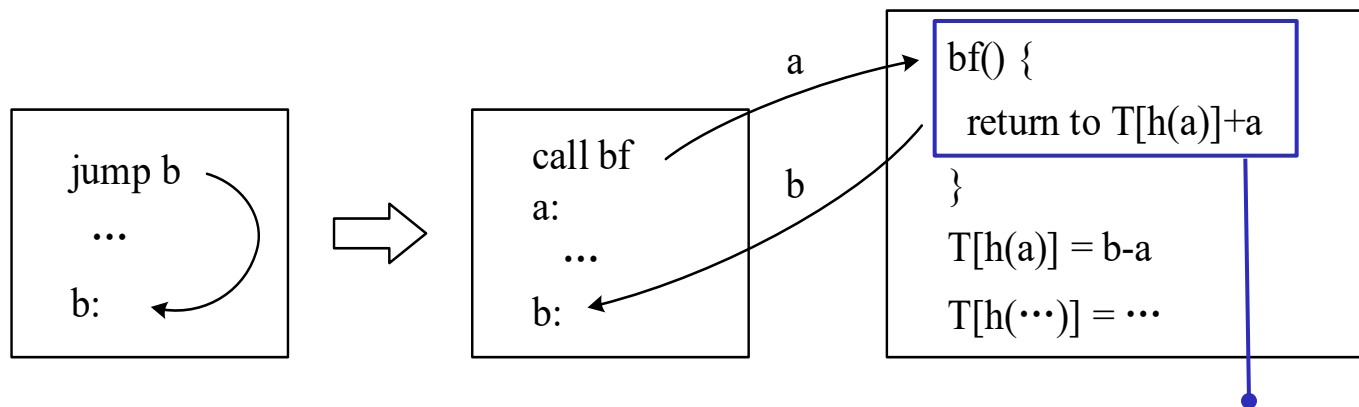
例子：增加冗余跳转后的循环及其控制流图。

```
if (PF) goto b ;
while (1) {
  x = y+10 ; return x ;
  b: y = 20 ;
```



### 3.3、通过跳转函数执行无条件转移

通过跳转函数执行无条件转移的思路就是把程序中的一个无条件转移指令替换成调用一个跳转函数指令。这样，函数在执行完毕后，会返回到原来的无条件转移指令的目标地址上去，利用call指令后面的代码实际上并不会执行的“漏洞”，可以在这里填上冗余指令。



函数bf被调用时，会记录自己的返回地址a。bf函数使用返回值a查表T，得到需要跳转的偏移量b-a，然后再加上a就得到了b，即原来的无条件转移指令的目标地址，再把自己的返回地址换成b，执行return指令。这样，就完成了跳转。

## 3.3、通过跳转函数执行无条件转移

例子：对模 $n$ 的取幂函数进行用跳转函数代替无条件转移指令处理。

1)、用label和goto代替掉while-、if-等高级控制语句

```
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 ;
    int k = 0 ;
    int s = 1 ;
    beginloop :
        if (k>=w) goto endloop ;
        if (x[k] != 1) goto elsepart ;
        R = (s*y)%n ;
        goto endif ;
    elsepart :
        R = s ;
    endif :
        s = R*R%n ;
        L = R ;
        k++ ;
        goto beginloop ;
    endloop :
    return L ;
}
```

## 3.3、通过跳转函数执行无条件转移

2)、给定表T，替换跳转函数的返回地址

对于每个地址对  $(a_i, b_i)$ ，其中  $a_i$  是跳转函数的“正常”返回地址， $b_i$  是想要跳转去的地址，表T中存放的是：

$$T[h(a_i)] = b_i - a_i$$

其中  $h()$  是一个hash函数，它的作用是把由组成的稀疏地址空间压缩到  $1 \cdots n$  这个范围内。给定表T后，跳转函数为：

```
char *T[2];
void bf() {
    char *old;
    asm volatile( "movl 4(%%ebp), %0\n\t" : "=r" (old)); 读出函数原有的返回地址
    char *new = (char *)((int)T[h(old)] + (int)old);
    asm volatile( "movl %0, 4(%%ebp)\n\t" :: "r" (new)); 保存新的返回地址
}
```

## 3.3、通过跳转函数执行无条件转移

### 3)、用跳转函数替代掉无条件转移指令

```

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 ;
        bf( ) ;
        retaddr1 : // goto endif ;
        asm volatile( ".ascii \ "bogus \ " \n\t" );
        ;

    elsepart :
        R = s ;
    endif :
        s = R * R % n ;
        L = R ;
        k++ ;
        bf( ) ; // goto beginloop ;
        retaddr2 :

    endloop :
    return L ;
}

```

计算表T中的各个偏移量。通常混淆时已计算完毕，不会在程序运行时计算。

不会被执行的代码。

用跳转函数替代的无条件转移指令。

## 4、不透明谓词

◆ 4.1、不透明谓词的定义

◆ 4.2、从别名中产生不透明谓词

◆ 4.3、从并发中产生不透明谓词

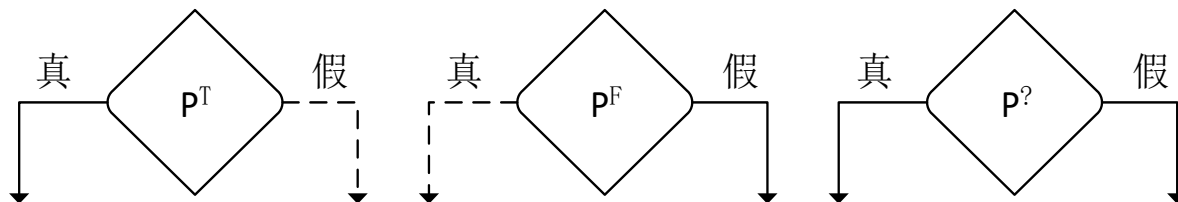


## 4.1、不透明谓词的定义

许多代码混淆技术都有赖于不透明表达式。在混淆时，如果一个表达式的值开发者已经知道了，但攻击者很难根据表达式本身推断它的值，那么它就能算是一个不透明表达式。

最常见的不透明表达式是不透明谓词，它是一种控制流混淆技术，对其比较完整的描述是：谓词 $P$ 在程序中的某一点 $p$ ，如果在混淆之后对于混淆者是可知的，而对于其他人是难以获知的，则称该谓词为不透明谓词，实际就是你知道它总是为真，或总是为假，或时真时假的布尔表达式。

用 $P^T$ 表示结果实际上总是为真的不透明谓词， $P^F$ 表示结果实际上总为假的不透明谓词， $P^?$ 表示结果真假不定的不透明谓词。虚线箭头表示程序执行时，实际上并不会经过的控制路径。



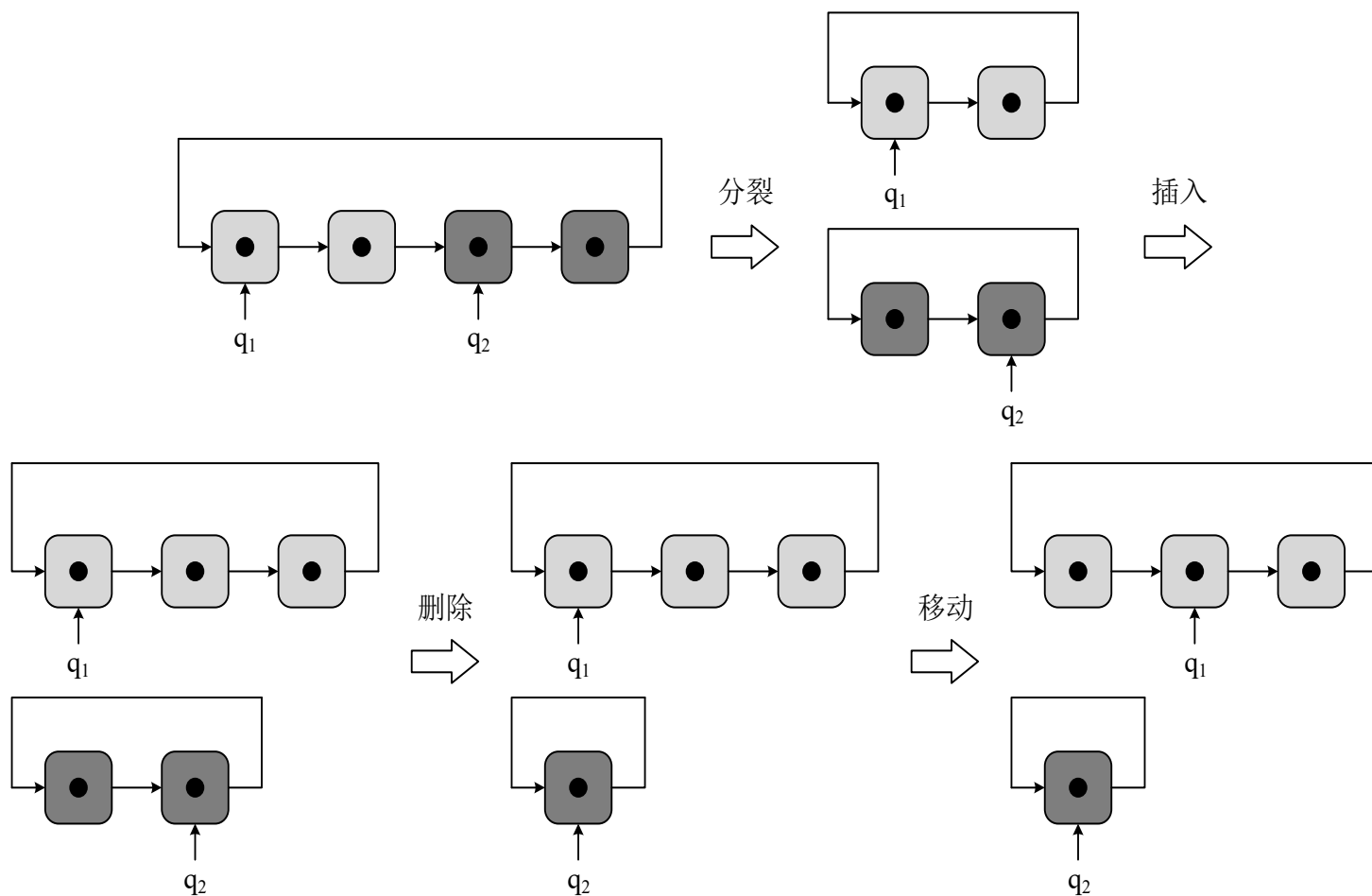
## 4.2、从指针别名产生不透明谓词

算法的基本思路是代码以动态分配的方式构造一个或多个图，并使一些指针一直指向图中的一些结点，然后通过检查指针的一些属性（已知这些属性的真伪）来构造不透明谓词。

- 1) 在程序P中添加一些代码，用这些代码以动态分配的方式构建多个图，并将这些图合并成一个集合 $G=\{G_1, G_2, \dots\}$ ;
- 2) 在P中添加一个指针集 $Q=\{q_1, q_2, \dots\}$ ，使Q中各个指针指向G中各个指针结构;
- 3) 创建G和Q之间的固定关系集 $I=\{I_1, I_2, \dots\}$ ，比如：
  - $q_i$ 总是指向 $G_j$ 中的结点;
  - $G_i$ 一定是强连通图。
- 4) 在P中再加上一些能在维持固定关系集I不变的前提下，对G中的图，或Q中的指针进行修改操作的代码;
- 5) 使用关系集I，制造一些基于Q的不透明谓词，比如：
  - 如果 $q_i$ 和 $q_j$ 是分别指向图 $G_m$ 和图 $G_n$ 的指针，那么有 $(q_i \neq q_j)^T$ ;
  - 如果指针 $q_i$ 所指的对象一直是在图 $G_k$ 中各个结点里移动，且图中还没有叶子节点，那么有 $(q_i \neq \text{null})^T$ ;
  - 如果 $q_i$ 和 $q_j$ 都是指向图 $G_k$ 的指针，那么有 $(q_i = q_j)^?$ 。

## 4.2、从指针别名产生不透明谓词

例子：创建两个指针 $q_1$ 和 $q_2$ ，使它们分别指向 $G_1$ （浅灰色）和 $G_2$ （深灰色）这两个图。



## 4.2、从指针别名产生不透明谓词

上例中所有的结点都是一个类型的，所以所有的指针也都是一个类型的。比如，所有结点都是java中class Node{Node next;}这个类型的实例。

- 1) 开始时，所有的结点都在一个循环链表中，
- 2) 然后循环链表被一分为二，成了两个图 $G_1$ 和 $G_2$ 。
- 3) 接着添加一些任意的操作——让指针 $q_1$ 和 $q_2$ 在各自的图中移动，随机地添加和删除图中的一些结点，但有两个固定关系不变，
  - “ $G_1$ 和 $G_2$ 都是循环链表”；
  - “ $q_1$ 指向 $G_1$ 中的某个结点， $q_2$ 指向 $G_2$ 中的某个结点”。
- 4) 当插入的冗余操作（针对别名分析算法）执行了足够多的次数之后，就可以在代码中加入 $(q_1 \neq q_2)^T$ 之类的不透明谓词了。

## 4.2、从数组别名产生不透明谓词

从数组别名中产生不透明谓词使用数组别名来产生不透明值，以此来隐藏对控制流进行编码的技术。

例子：整型数组如下所示

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
36	58	1	46	23	5	16	65	2	41	2	7	1	37	0	11	16	2	21	16

数组构造规则如下：

- 1) 从第0个元素起，每隔三个元素（浅灰色底色），它们的值都 $\equiv 1 \bmod 5$ ；
- 2) 下标为2和5的元素（条形阴影）的值分别为1和5；
- 3) 从第1个元素其起，每隔三个元素（中灰色底色），它们的值都 $\equiv 2 \bmod 7$ ；
- 4) 下标为8和11的元素（深灰色底色）的值分别是2和7。

其他的元素（白色底色）为垃圾数据。

## 4.2、从数组别名产生不透明谓词

代码如下所示：

```
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 ” ) ;
```

产生不透明谓词

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

在程序运行时实时地对g中数据进行更新。

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

产生一个不透明值42

## 4.2、从数组别名产生不透明谓词

使用从数组别名中产生不透明值的方法来加强压扁控制流算法。

通过对switch语句块中的next变量进行“使用-定值链”的常量传播分析，就可以重构出进行压扁控制流混淆后的程序的控制流图。针对这种攻击方法，可以用一个不透明表达式去计算next的值，使得数据流分析器无法确定next变量的变化情况。

```
int modexp( int y, int x[ ], int w, int n) {  
    int R, L, k, s ;  
    int next = E0;  
    for( ; ; )  
        switch(next) {  
            case 0: k = 0 ; s = 1 ; next = E1 ; break ;  
            case 1: if(k<w) next = E2 ; else next = E6 ; break ;  
            case 2: if(x[k]==1) next = E3 ; else next = E4 ; break ;  
            case 3: R = (s*y)%n ; next = E5 ; break ;  
            case 4: R = s ; next = E5 ; break ;  
            case 5: s = R*R%n ; L=R ; k++ ; next = E1 ; break ;  
            case 6: return L ;  
        }  
}
```

## 4.2、从数组别名产生不透明谓词

用很难进行别名分析的数组来实现这些不透明表达式。实际上，可以在这里使用任何一种不透明表达式的生成算法。

引入一个常量数组g，各个表达式则是根据g中的各个元素的值来计算变量next的值的。

```
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 ;
        }
}
```



## 4.2、从数组别名产生不透明谓词

为了使静态分析更复杂，可以让数组g中的值稳定的发生变化，这样只要能跟上数组变化的节奏，不透明表达式的值仍然可以是一个常量。

例如，利用如下所示的permute函数每次把数组中的元素往右旋转一位。

```
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 ;  
}
```

## 4.2、从数组别名产生不透明谓词

如下所示是用循环每进行一轮就旋转一下数组g的元素的方法加强的压扁控制流的代码。不管数组中的元素如何变化，g[m]的值永远是10。

```
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) ;
    }
}
```

## 4.2、从数组别名产生不透明谓词

为了进一步使静态分析更复杂，建议使用全局数组。

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

## 4.2、从数组别名产生不透明谓词

这样做的好处在于，可以在其他的地方对数组中的元素进行初始化，还可以让程序每次调用modexp函数时，数组g中的值都不一样。

```
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) ;
```

## 4.2、从数组别名产生不透明谓词

最后，为了进一步对抗别名分析器，可以在程序中添加指针变量和指针操作。

```
int modexp( int y, int x[ ], int w, int n) {
    int R, L, k, s ;
    int next = 0 ;
    int *g2 ; int *gr ; 5, 3, 42} ;
    int *g2 ; int *gr ; 规划吗,
    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 ;
        }
}
```

g中的第三个元素  
既可以通过数组  
下标g[2]来访问，  
也可以通过指针  
\*g2来访问。

## 4.3、从并发中产生不透明谓词

由于其交叉语义，并发程序很难被静态分析。如果在一个并发范围内有 $n$ 条语句，那么这些语句可以有 $n!$ 种不同的执行顺序。这样，基于分析程序中线程的行为非常困难这一事实，可以构造出能对抗自动分析攻击的不透明谓词。

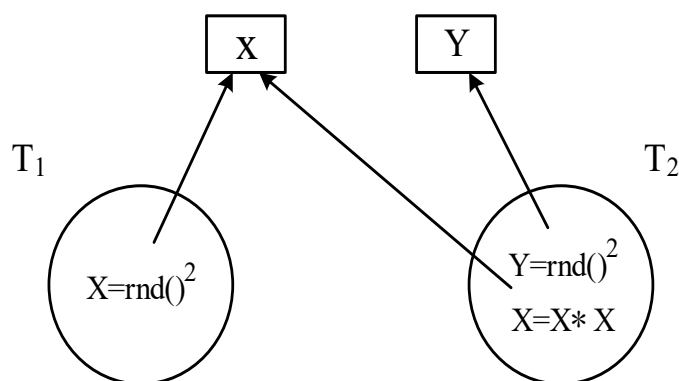
算法的基本思路是，产生一个满足固定关系集 $I$ 的全局数据结构 $G$ ，通过并发的方式，在保持 $I$ 不变的情况下，对 $G$ 中的数据进行更新操作，最终使用 $I$ 产生一个关于 $G$ 的不透明谓词。

算法：在程序 $P$ 中创建基于线程的不透明谓词

- 1) 在 $P$ 中添加一些代码，用这些代码创建一个满足固定关系集 $I=\{I_1, I_2, \dots\}$ 的全局数据结构 $G$ ；
- 2) 在 $P$ 中添加一些创建线程 $T_1, T_2, \dots$ 的代码，在新创建的线程中对 $G$ 中的数据进行更新，并使之继续满足固定关系集 $I$ ；
- 3) 使用固定关系集 $I$ ，产生一个关于 $G$ 的不透明谓词。

## 4.3、从并发中产生不透明谓词

例子：如图所示，两个线程 $T_1$ 和 $T_2$ 完全无视数据竞争条件，并发地对两个整型变量 $X$ 和 $Y$ 的值进行更新。



假设赋值语句是原子操作（所谓原子操作是指不会被线程调度机制打断的操作，**Java中对整型变量的操作就能保证这一点**），上图所示的代码保持的固定关系 $I$ 是变量 $X$ 和 $Y$ 的值都是某个数的平方。根据数论的相关知识， $\forall x, y \in \mathbb{Z}: x^2 - 34y^2 \neq -1$ ，就能创建不透明谓词 $(X - 34Y == -1)^F$ 。

## 4.3、从并发中产生不透明谓词

利用多线程结合从指针别名中产生不透明谓词来构造一个结果肯定为真的不透明谓词，且无法通过静态分析判断其真伪。基本思路如下：

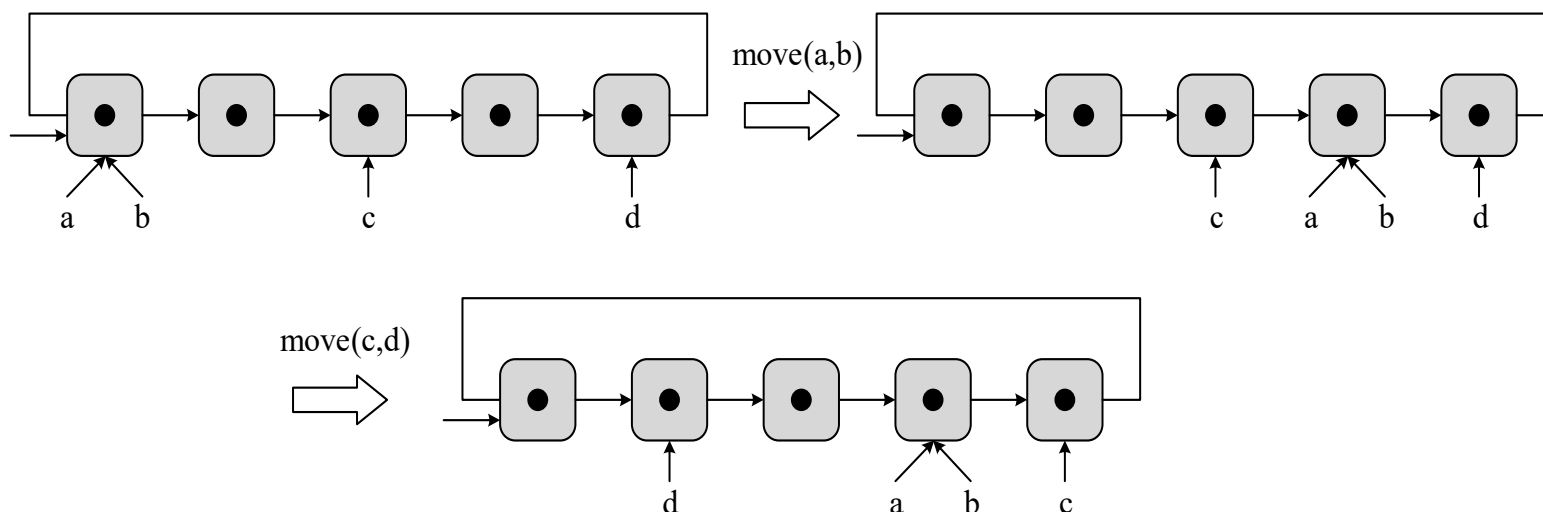
基本思路如下：

- 1) 构建一个固定的、拥有 $k$ 个结点的循环单链表 $G$ ；
- 2) 构造两个指针 $a$ 和 $b$ 。初始化时，这两个指针指向 $G$ 中同一个节点；
- 3) 再构造两个指针 $c$ 和 $d$ 。初始化时，这两个指针指向 $G$ 中不同的结点；
- 4) 创建一个线程 $T_1$ ，该线程异步地对 $a$ 和 $b$ 进行原子的更新操作，使每次更新后， $a$ 和 $b$ 都按 $G$ 循环的顺序指向下一个结点；
- 5) 类似4)，创建一个新线程 $T_2$ ，对 $c$ 和 $d$ 进行与类似 $T_1$ 的更新操作。



## 4.3、从并发中产生不透明谓词

这两个线程异步地沿着循环的方向移动指针，并保持固定关系  $a == b$  和  $c != d$  不变。这样，就构建了一个结果一定为真的不透明谓词  $(a = b)^T$ 。在静态分析时，如果事先不知道线程使之保持不变的固定关系，这个不透明谓词很难与另一个结果一定为假的不透明谓词  $(c = d)^F$  区别开来。



## 4.3、从并发中产生不透明谓词

实现代码如下：

```
class Race extends Thread {
    Node x, y ;
    public Race( Node x, Node y, Object lock) {
        this.x = x ;
        this.y = y ;
        this.lock = lock ;
        start( ) ;
    }
    public void run( ) {
        while(true) {
            synchronized(lock) {
                x = x.next ;
                y = y.next ;
            }
        }
    }
}
```

```
int size = getRandomBetween(2, 10) ;
```

返回[2,10]  
之间的一个  
随机整型数。

```
Node cycle = creatCycle(root, size) ;
```

```
int m = getRandomBetween(2, 10) ;
```

```
int n = m ;
```

```
While (m==n)
```

```
    n= getRandomBetween(2, 10) ;
```

```
Node a = getNth(root, m) ;
```

返回从结点  
root数起的  
第m个结点。

```
Node b = getNth(root, m) ;
```

```
Node c = getNth(root, m) ;
```

```
Node d = getNth(root, n) ;
```

```
Object lock = new Object( ) ;
```

```
Race race1 = new Race(a, b, lock) ;
```

```
Race race2 = new Race(c, d, lock) ;
```

```
synchronized(lock) {
```

```
    if (race1.x==race1.y)T...
```

```
    if (race2.x==race2.y)F...
```

```
}
```

# 5、结构混淆

## 5.1、结构混淆概念



## 5.2、合并函数签名



## 5.3、分解和合并类



## 5.4、破坏Java的高级结构



## 5.5、修改指令的编码方式



## 5.1、结构混淆

程序存在的各类组织结构是个很重要的漏洞，这其中有一些结构是强加的（例如，运行程序的机器使用的指令集体系结构，操作系统的系统调用接口，程序中使用的Java库中的类等），而另一些则是编程是创建的。

程序分解成包、模块、类和方法的方式反映了程序的设计思想，会给攻击者留下大量的线索。例如，基本上可以肯定在同一个类中声明的两个函数之间一定有着某种联系，而类则很可能是其超类的一种特例。

结构混淆将介绍破坏或隐藏这类结构的方法，包括合并程序中函数的签名，从而使攻击者无法从函数签名中获得任何有用的信息；如何分解和合并类从而隐藏类与类之间的继承层次关系；摧毁Java程序中所有类的层次关系，并把它们全部编码成普通的数据，以及通过在程序中插入一个解释器，破坏指令集体系结构的不变形。

## 5.2、合并函数签名

函数的签名会泄露不少语义信息。例如，可以很有把握的根据函数签名推断下面这两个函数（foo和bar）的语义是完全不同的。

```
int foo( int, int ) {…}  
void bar( Windows, String ) {…}
```

算法的思路是尽可能把程序中所有函数的签名都改成一样。但在实践中，这一做法的代价过于高昂，多数情况下，只是把程序中的函数分为几个等价类，并使每个等价类中的所有函数都拥有同一种签名。

在转换时要注意防止给程序中引入新bug。例如，根据C语言标准，函数参数列表中各个参数的出现顺序是无关紧要的，可以任意排列，但在Java中，参数却需要严格按照从左到右的顺序给出。

## 5.2、合并函数签名

- 对于Java，合并函数签名的最简单方法就是把程序中所有方法的签名全部改成 `Object foo(Object[ ])` 这种。但是，这需要在调用方法之前把所有的参数全部封装在 `Object[ ]` 数组中，然后再在方法的入口处从 `Object[ ]` 数组把各个参数都解析出来。此外，程序中还要分配给 `Object[ ]` 数组必要的内存空间。
- 另一个解决方案是通过
  - a) 把所有被引用的数据类型都变成 `Object` 类型和
  - b) 插入多余的参数，是两个或者更多的方法拥有同样的签名。

算法就对C使用了类似的策略，把函数参数中所有较大的参数（数组或者结构体）都变成 `void *`，然后在函数的参数列表中插入了相应的冗余参数。

注意：如果使用这一技术对发布版本的程序可执行文件（例如Java字节码）进行混淆，修改的只是方法的签名，各个参数的类型在代码中其实还是可见的。

## 5.2、合并函数签名

例子：合并函数签名，代码如下

```
typedef struct {int waldo ;} thud_t ;
void foo(int i, int j) {}
void bar(int i, float f, int j, int k) {}
void baz(char* s, thud_t t) {
    int x = t.waldo ;}
void fred(char* s, char* t) {}
void quux( int i ) {}
void qux(char* s, int i, int j) {}
```

```
int main ( ) {
    int i ;
    thud_t corge ;
    foo( 1, 2 ) ;
    bar( 3, 4, 2, 8, 9 ) ;
    baz( "dude", "corge" ) ;
    fred( "yo", "da" ) ;
    for( i = 0 ; i<10000000 ; i++ )
        qux( "luke", 1, 2 )
    quux(5) ;
}
```

函数foo、bar和quux比较相像，可以把它们的签名修改为同一个签名void( int, float, int, int )。同样，baz和fred函数也很相像，把它们的签名合并为void( char\*, void\* )。

## 5.2、合并函数签名

经过合并，代码如下：

```
typedef struct {int waldo ; } thud_t ;

void foo(int i, float bogus1, int j, int bogus2 ) { }
void bar(int i, float f, int j, int k) { }
void quux( int l, float bogus1, int bogus2, int bogus3 ) { }
void baz(char* s, void* t) {
    int x = ((thud_t*)t)->waldo ;
}
void fred(char* s, void* t) { }
void qux(char* s, int i, int j ) { }
```

```
int main ( ) {
    int i, bogus1, bogus2 ;
    float bogus3 ;
    thud_t corge, corge_cp ;
    foo( 1, bogus3, 2, bogus1 ) ;
    bar( 3, 4, 2, 8, 9 ) ;
    memcpy(&corge_cp, &corge, sizeof(corge)) ;
    baz( "dude", &corge_cp ) ;
    fred( "yo", "da" ) ;
    for( i = 0 ; i<10000000 ; i++ ) qux( "luke", 1, 2 )
    quux( 5, bogus3, bogus1, bogus2 ) ;
}
```

由于C语言是通过传值的方法把结构体传递给函数的，所以在main函数中增加了一个thud\_t型局部变量corge\_cp，复制corge后传递给baz函数。在上例中，增加了一些冗余变量来充当函数的冗余参数。



## 5.3、分解和合并类

在典型的基于类的面向对象语言中，类是由变量实例、虚方法以及构造函数组成的。类之间还可以继承，因此程序中的类就构成了一个继承关系树，如果该语言支持多重继承，这棵树就变成了一个有向无环图。

本算法给出了3中混淆这类继承关系的方法，即

- 1) 把一个类分解成两个类，
- 2) 把两个类合并为一个类以及
- 3) 在继承关系中插入多余的类。

混淆类的3个基本操作是MOVEUP、INSERT-EMPTY和DELETE-EMPTY。

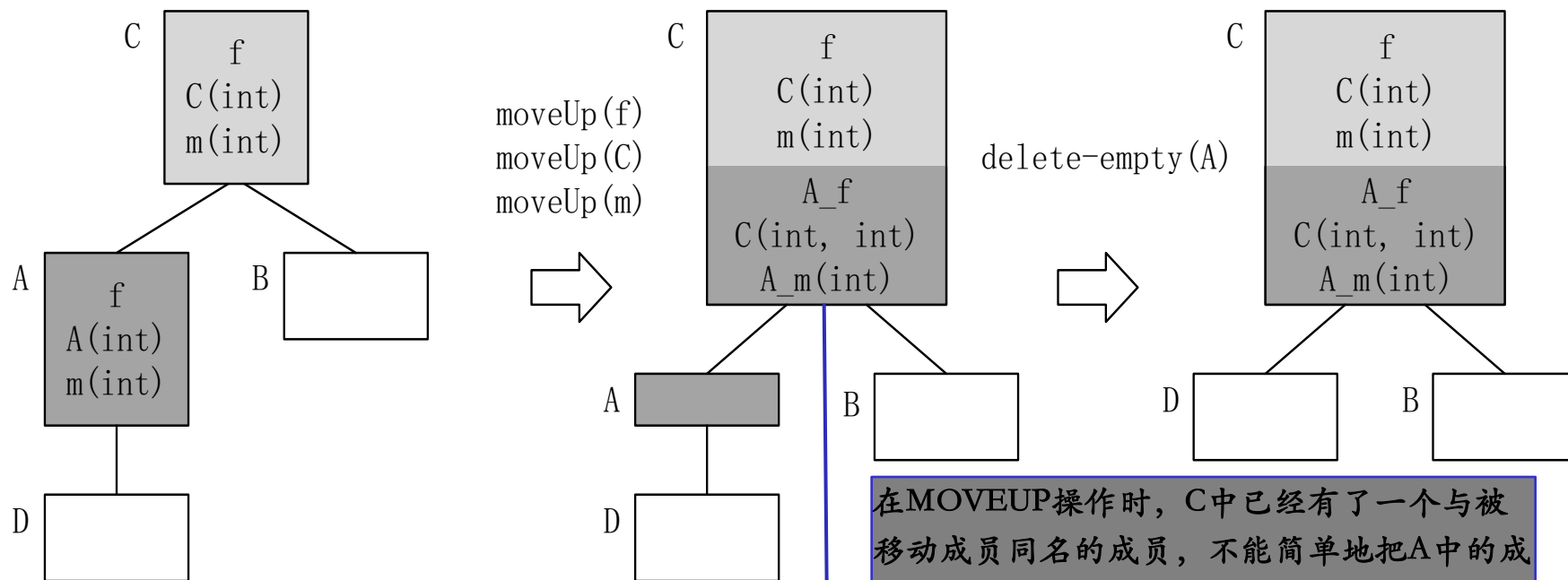
- 基本操作MOVEUP是把类中声明的一个名字（数据成员、方法或构造函数）移动到它的父类中去；
- 基本操作INSERT-EMPTY是往继承关系图中插入一个新的空类；
- 基本操作DELETE-EMPTY是把继承关系图中的一个空类删除掉。

由于这3中基本操作的具体实现细节有程序所使用的语言决定，这里只讨论一种简单的情况——不带接口的Java。

## 5.3、分解和合并类

例子：类的合并

左边的图包含4个类。若要A合并到C中去，首先要执行MOVEUP(F)、MOVEUP(M)、MOVEUP(A)操作，然后在执行DELETE-EMPTY(A)操作。

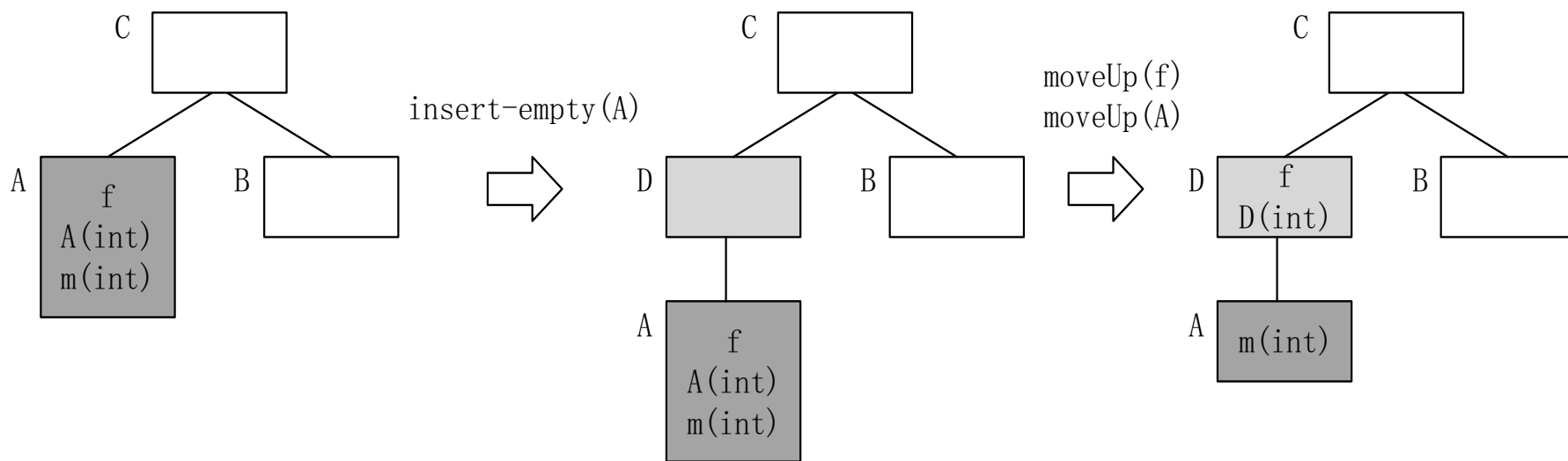


在MOVEUP操作时，C中已经有了一个与被移动成员同名的成员，不能简单地把A中的成员移到C中去。由于A中的`m()`方法实际上覆盖了C中的`m()`方法，所以仅仅重命名为`A_m`还是不够的。

## 5.3、分解和合并类

例子：类的分解

类的分解就是类合并的逆过程。在例子中，先添加一个空白的超类D，然后再把A的数据成员f和构造方法A移动到D中，重命名为D，这样就完成了类的分解。



遗憾的是，并不能随心所欲地MOVEUP类中的成员。在MOVEUP方法或构造函数时，必须把所有该方法/构造函数可能使用到的成员都一起MOVEUP到目标类中去。

## 5.3、分解和合并类

算法中用来执行类的分解和合并的MOVEUP操作的定义如下所示：

### ■ MOVEUP (数据成员f)

- 1) 给f一个唯一的名字f'；
- 2) 把所有使用f的地方全部改成使用f'；
- 3) 把f'移动到父类中去。

### ■ MOVEUP (方法m)

- 1) 给m一个唯一的名字m'；
- 2) 把所有使用m的地方全部改成使用m'；
- 3) 把m'移动到父类P中去；
- 4) 如果P已经有了一个方法P.m，该方法的名字或签名与m'相同，则把P.m改写为：  

```
method P.m (...) {
    if (kind=="P")
        P.m的原有代码；
    else
        m'(...);
}
```

- 5) 对所有m中将会使用到的成员执行MOVEUP操作。

## 5.3、分解和合并类

### ■ MOVEUP (构造方法C)

- 1) 不断地给C加上冗余的形参，直到父类P中没有一个构造函数的签名与C一样为止；
- 2) 给所有调用该构造方法的语句中加入冗余的实参，使其与步骤1) 中添加的冗余形参相匹配；
- 3) 在P中添加一个数据成员kind（如果P中当前没有这样一个数据成员），并添加代码，使得当kind="P"时，调用P自己的构造函数；当kind="C"时，调用C；
- 4) 把C重命名成P，并把它移动到P中；
- 5) 对所有运行时的类型检查函数进行改造，使之能够根据数据成员kind的值判断对象的类型；
- 6) 对所有C中将会使用到的成员执行MOVEUP操作。

## 5.3、分解和合并类

例子：将类A（深灰色底色）合并到类C（浅灰色底色）中去

对类的继承关系图进行转换

```
class C {
    int f ;
    public C( int x ) {
        f = x ;
    }
    int m( int x ) {
        return ( f += x ) ;
    }
}

class A extends C {
    int f ;
    public A( int x ) {
        super (x) ;
        f = x +1 ;
    }
    public int m( int x ) {
        return ( f -= x ) ;
    }
}
```

```
class B extends C {
    public B( int x ) {
        super (x) ;
    }
}

class D extends A {
    public D( int x ) {
        super (x) ;
    }
}
```



## 5.3、分解和合并类



```
class C {
    public String kind ;
    int f ;
    public C( ) { }
    public C( int x ) {
        f = x ;
        kind = " C " ; }
    int m( int x ) {
        if (kind.equals(" C "))
            return ( f + = x ) ;
        else
            return A_m(x) ; }
}
```

```
int A_f ;
public C(int x, int dummy) {
    this(x) ;
    kind = "A" ;
    A_f = x +1 ; }
int A_m( int x ) {
    return ( A_f - = x ) ; }
}
```

```
class D extends C {
    public D( int x ) {
        super (x, 42) ; }
}
Class B extends C {
    public B( int x ) {
        super (x) ; }
}
```

## 5.3、分解和合并类

转换使用了相关类的代码：

```
C c = new C(0) ;  
A a = new A(0) ;  
a = (A) c ;
```



```
C c = new C(0) ;  
C a = new C(0, 42) ;  
  
If ( ! ( ( c instanceof D )  
||( c.kind.equals("A") ) ) )  
    throw new ClassCastException("C") ;  
a = c ;
```



## 5.4、破坏Java中的高级结构

算法的基本思路是尽可能地去掉类型化的Java字节码中的高级语言结构。

在Java虚拟机中，类、成员、方法以及异常都是有类型的，而这些类型可能会提示攻击者程序中各部分都是干什么的。同样，由于Java字节码的强类型属性，以及其代码和数据不能混合的特点，使得攻击者可以方便的重建出程序的控制结构。

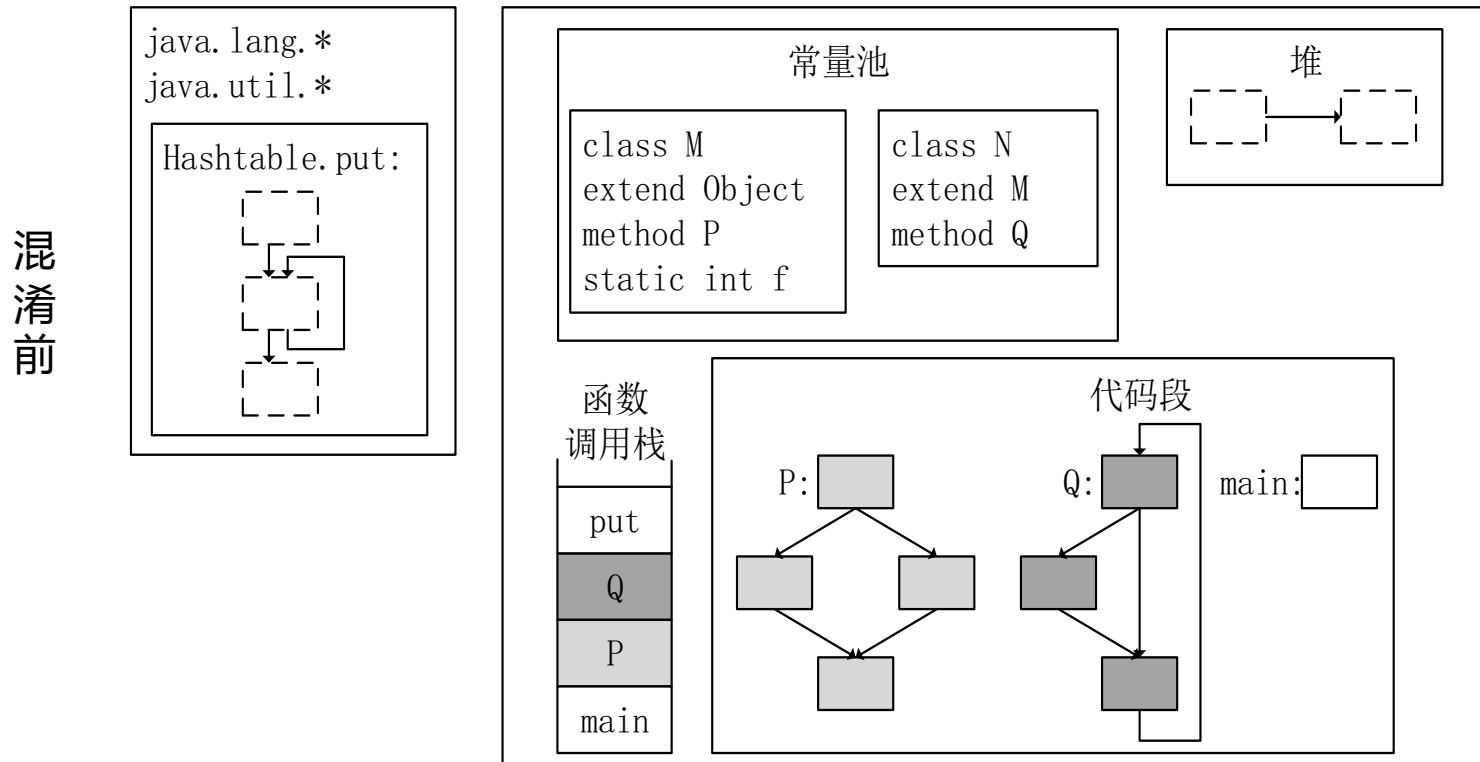
如果想要写一个Java源码到机器码的编译器，就必须把所有的高级结构（类、方法分配、异常处理、动态内存分配和回收，等等）都转换成由硬件或操作系统支持的底层操作。对Java而言，在字节码的范围进行这类转换，即原来的程序和混淆后的程序都是由Java.class文件组成的，在混淆后的程序中，**没有一个类会在常量池中（因为每个类都用一个互不重复的整型数表示了）；虚方法分配则是通过查虚方法表，而不是Java字节码invokevirtual指令来实现；所有的内存都是以一个平坦数组成的形式存在的（包括方法调用栈），处理异常也不会去用Java内置的机制等。**

## 5.4、破坏Java中的高级结构

去掉类型化的Java字节码中的高级语言结构：

第一步：对控制流图进行chenxify转换（使得重构源码级的控制结构变得相当困难），把类型化的数据转换成（几乎是）非类型化的数据。

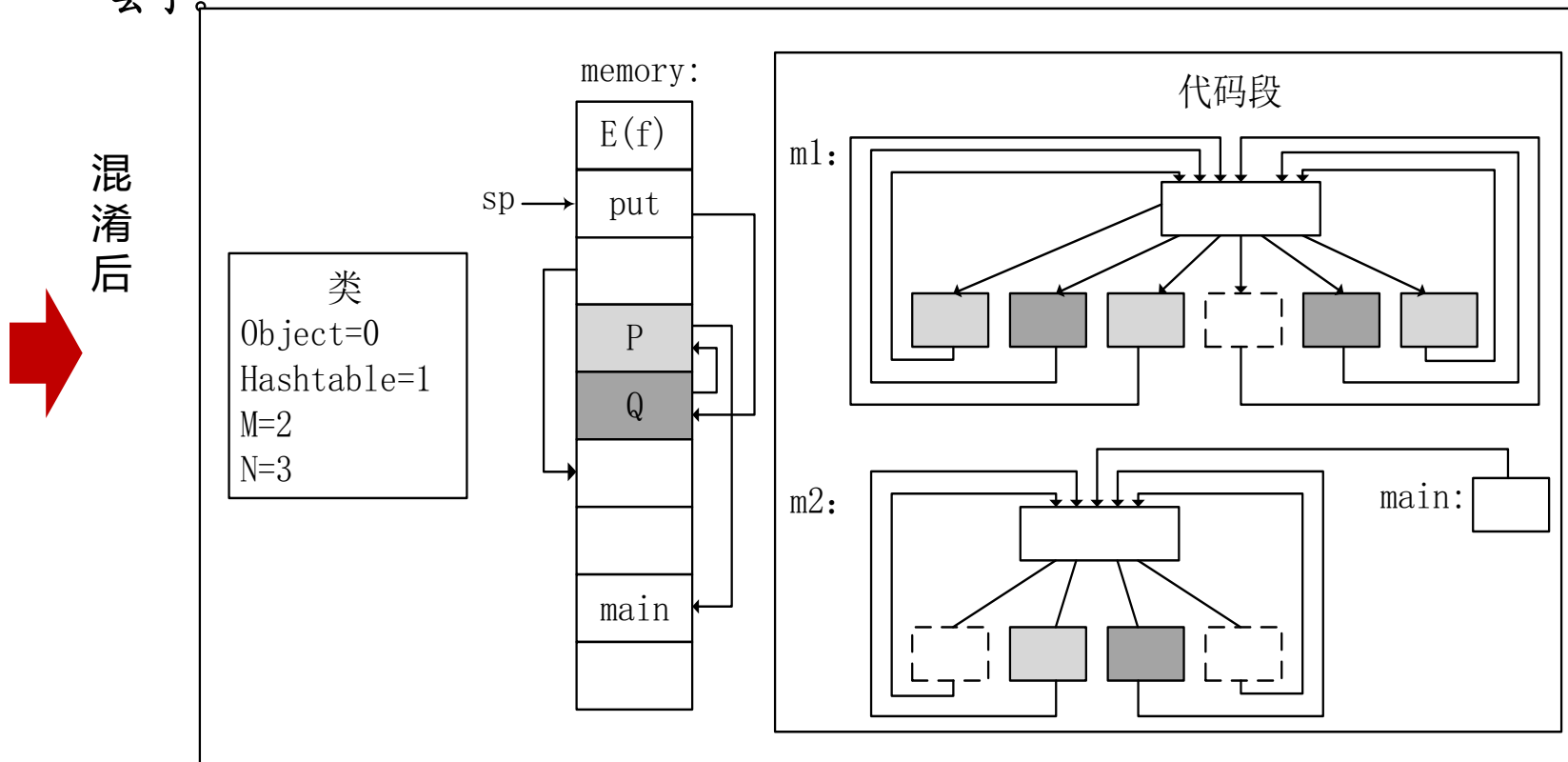
例子：程序由两个类（M和N）组成，M中包含方法P和静态的整型变量f，N中包含方法Q，这两个方法的控制流图将被合并，并进行“chenxify”处理。



## 5.4、破坏Java中的高级结构

由于Java虚拟机中对方法的大小有限制，所以在混淆后的程序中，各个基本块可能分散在几个方法中。

混淆后，方法P、Q、main以及Hashtable.put中的基本块，被放在两个方法（m1和m2）中，所有的数据（方法调用栈、堆和静态数据）都被合并到数组memory中去了。



## 5.4、破坏Java中的高级结构

第二步：合并各个方法（去掉过程抽象），用不重复的整型数表示各个类，用自己的虚方法表来实现虚方法的分配，使用自己的方法调用栈以及为每个Java库中的常用类提供一个自己的实现（防止因为调用库函数而泄露信息）。

例子：如下所示是一个简单的Java程序，它含有两个类A和B，其中B是A的子类。在main方法中创建了一个B的实例，并调用了B中的虚方法n。

```
class A{
    int i;
    int m(){return i;}
    void n(int i){}
}
class B extends A{
    String s;
    public B(int i,String s)
        this.i=i;this.s=s;
    }
    void n(int v){
        int x=v*m();
        System.out.println(s+x);
    }
    public static void main(String[] args){
        B b=new B(6, "42" );
        b.n(7);
    }
}
```

## 5.4、破坏Java中的高级结构

原来程序中的每个数据对象都由混淆后程序中的Memory对象表示。在Memory对象中，每种基本数据类型都有一个对应的数组，此外，还有一个类型为Memory的数组M——通过这个数组，就能把所有的对象相互连接起来，能很方便构成任意形式的对象关系图。

```
class Memory{  
    public int[] I;  
    public Object[] L;  
    public long[] J;  
    public float[] F;  
    public double[] D;  
    public Memory[] M;  
}
```

经过混淆后，原来程序中的各个数据片段（包括虚方法表、方法调用栈以及类的实例）就都被映射到了Memory对象中。甚至可以通过把所有数据全部当成二进制byte数组，进一步去掉大部分的类型信息，但这样一来，程序的性能损失必然非常大。此外，由于Java虚拟机指令本身就是类型化的，它会泄露执行时所需的内存地址上存放的数据的类型，所以即使这样，估计也不能给攻击者制造多大的麻烦。

## 5.4、破坏Java中的高级结构

在开始执行之前，还需要为每一个类创建一个虚方法表，类A的虚方法表由2和3两个数字组成。它们分别对应的是A的方法m和n的第一个基本块chenxify后的函数里的switch语句中的case标号。因为B继承了A中的m方法，所以在它的虚方法表中也有数字2。

```
public class Danna {
    static Memory[] mem=new Memory[10];
    static int mc=-1;
    static Memory SP;

    static{//初始化memory数组和虚方法表
        for(int i=0;i<mem.length;i++)
            mem[i]=new Memory();
        mem[++mc].l=new int[]{2,3};    //{A.m,A.n}
        mem[++mc].l=new int[]{2,5};    //{B.m,A.n}
    }
}
```

虚方法表在memory数组中的索引号（A的是0，B的是1）也就充当了类的标识符。因此if (x instanceof A) …这样的语句就会被转换成if((x.vtab==0)|| ( x.vtab==1))…。当然，也可以使用其他任何一种用在标准编译器中，时间复杂度为O(1)的类型测试技巧来完成这一任务。

## 5.4、破坏Java中的高级结构

由于演示程序很小，可以把程序中所有的基本块放在一个方法(m1)中，这个方法会直接被main方法的函数桩调用：

SP是方法调用栈的指针，AR (activation record, 活动记录) 实际上就是一个由SP的M成员引用的Memory对象数组。

```
static void m1() {
    mem[++mc].M=new Memory[2];
    SP=mem[mc];           // 分配main方法的AR

    int PC=0;
    while(ture) {
        switch(PC) {
            ...           // 其他基本块
            case1:{       // 原main函数的结束部分代码
                SP=SP.M[0]; // 解引用n的AR
                return;     // 返回到stub的main函数中去
            }
        }
    }

    public static void main(String args[]) {
        m1();              // 调用m1()方法
    }
}
```

## 5.4、破坏Java中的高级结构

原来的main方法要做的第一件事就是要创建一个B的实例。这就要分配相应的内存空间（在这个例子中就是要获得一个新的Memory对象，并在其中为对象的成员i和s分配空间，即创建对象的各个成员，同时还要在新对象中填上相应的虚方法表指针），然后再调用B的构造函数，对对象中的各个成员进行初始化：

把AR（也就是下一个基本块的case标号）设为6，就是让构造函数直接返回到标号为6的case语句块上去，这个语句块是main方法的第二个基本块。

例中没有使用引入伪别名这种隐藏各个基本块之间关系的技术。所以很容易就能发现，B的构造函数位于标号为4的case语句块中。

```
case 0: {
    Memory b=mem[++mc];           // 原main方法的第一个基本块
                                   // 给新对象分配内存空间
    b.I=new int[1];                // 创建对象成员“i”
    b.L=new Object[1];             // 创建对象成员“s”
    b.M=new Memory[] {mem[1]};    // 填入b的虚方法表指针
    SP.M[1]=b;                     // b=new B(6, " 42" )
    mem[++mc].M=new Memory[] {SP, b}; // B的构造函数AR
    SP=mem[mc];
    SP.I=new int[] {6, 6};          // RA=6
    SP.L=new ObfStr[] {new ObfStr(new char[] { '4' , '2' } )};
    PC=4; break;                  // 调用B的构造方法
}
```



## 5.4、破坏Java中的高级结构

构造函数将会得到新对象的指针（也就是传入this指针），并把传入的参数值写入对象的数据成员中：

```
case 4: {                                     // B的构造方法
    PC=SP. I[0];                             // 记录返回地址
    Memory this_=(Memory) SP. M[1];          // 获取this指针
    this_I[0]=SP. I[1];                      // this.i=i
    this_L[0]=(ObfStr) SP. L[0];             // this.s=s
    break;                                   // 返回
}
```

## 5.4、破坏Java中的高级结构

构造函数执行完毕后，他将要返回到main方法的第二个基本块上。要执行语句b.(n)7，先要确定应该调用哪个方法（因为根据对象b的类型，被调用的可能是A.n方法，也可能是B.n方法）。可以循着b中的虚方法表指针，找到应该被调用的方法：

```
case 6: {                                     // main函数中调用的b.n(7)的代码
    SP=SP.M[0];                             // 解引有构造函数的AR
    Memory b=SP.M[1];                       // main的局部变量b
    mem[++mc].M=new Memory[] {SP, b};      // 给方法n的AR分配空间
    SP=mem[mc];
    SP.I=new int[] {1, 7, 0};               // {返回值, 参数7, 执行方法n所需要的空间x}
    SP.L=new Object[2];                    // 方法n的求值栈
    PC=b.M[0].I[1];                        // b.n所在的case语句块的标号;
    break;                                 // 调用b.n
}
```

## 5.4、破坏Java中的高级结构

在这个例子中，b中虚方法表指针指向的是B对象，所以应该调用的是b.n方法。查一下B的虚方法表，就能知道方法b.n的第一个基本块在标号为5的case语句块中。着这个基本块中又要去调用另一个虚方法表m，所以又得去查虚方法表的设置活动记录：

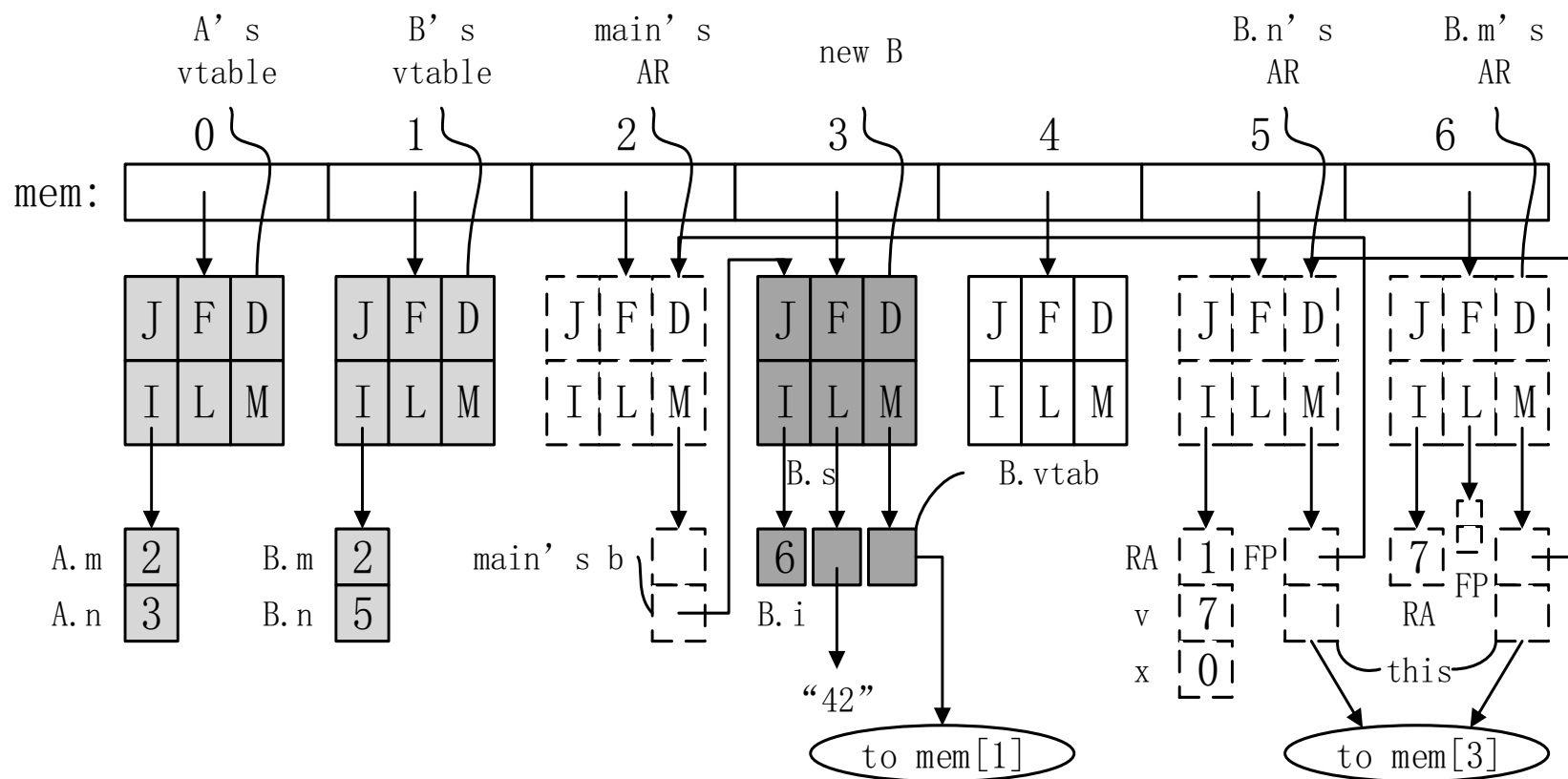
```
case 5: {                                     // B. n方法的方法体
    Memory this_=SP.M[1];                   // 获取this指针
    mem[++mc].M=new Memory[] {SP, this_};
    SP=mem[mc];                             // 给m方法的AR分配空间
    SP.I=new int[] {7};                     // 返回值参数7
    PC=this_.M[0].I[0];                     // this.m所在case块的标号
    break;                                  // 调用B.m() 方法
}
```

方法m只是返回实例变量v的值，而只要通过下面的this指针就能读出v的值了：

```
case 2: {                                     // A. m方法的方法体
    PC=SP.I[0];                             // 记录返回值
    Memory this_=SP.M[1];                   // 获取this指针
                                           // <<<内存dump>>>
    SP.I=new int[] {this_.I[0]};            // 返回this.i的值
    break;                                  // 返回
}
```

## 5.4、破坏Java中的高级结构

上述对该例的方法调用链进行了详细描述，程序执行到现在，内存中的情况如图所示。这是A.m()方法返回的瞬间，内存中的情况。其中浅灰色的块的虚方法表，深灰色的块是活动记录（AR），用虚线作边的块是类的实例，而白色的块则需要系统回收的垃圾（调用B的构造函数后，解引用构造函数的AR时形成的）。



## 5.4、破坏Java中的高级结构

最后，当方法m返回之后，还要执行n方法的第二个基本块：

```
case 7: {                                     // 方法n的第二个基本块
    int rm=SP.I[0];                          // rm是m的返回值
    SP=SP.M[0];
    Memory this_=SP.M[1];                   // 解引用AR
    PC=SP.I[0];                             // 获取this指针
    int v=SP.I[1];
    SP.I[2]=rm*v;                           // x=rm*v
    ObfStr s=((ObfStr)this_.L[0]);          //
    ObfStr vs=new ObfStr(SP.I[2]);          // vs=new String(x)
    System.out.println((new ObfStr(s)).append(vs).toJavaString());
    break;                                  // 返回到main方法中
}
```

## 5.5、修改指令编码方式

算法的基本思想：攻击者通过反汇编或者在调试器中单步执行各条指令来获取有关信息，需要对程序中所使用的指令集体系结构有所了解。如果在机器和代码之间插入一个解释器，那就能随心所欲地构造出任意一种体系结构的指令集，以增加程序破解难度。

算法的设计目标：给每个程序的每份副本都配备一个互不相同的解释器和一套互不相同的指令集，使它们变得各不相同（因而也就具有更强的抗修改能力）。

算法是通过破坏指令编码这层抽象来做到这一点的，它使程序的每份副本都使用不同的指令编码方式，而且在程序运行时还会不断改变指令的编码方式。

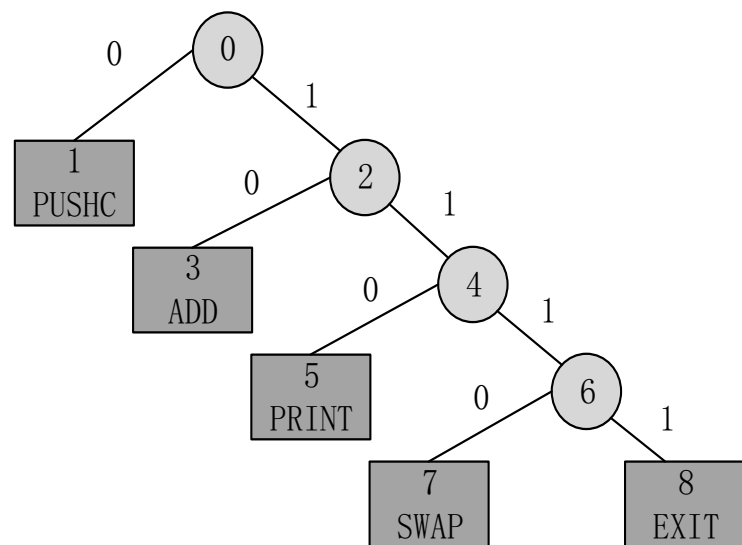
因此，当破解者攻击这类程序，分析指令的各个bit位都是做什么用的时，他会发现当他选择按不同的执行路径运行程序时，指令的编码方式也会随之发生变化。

## 5.5、修改指令编码方式

下例中，指令PUSHC c就是把一个占3个bit位的常量c压入栈中；指令ADD就是把栈顶上的两个元素相加；指令PRINT就是打印并弹出栈顶上的那个元素；指令EXIT则是停止执行的意思。

这个程序会打印数字7两次。SWAP n指令是个元指令，它的意思是“从这里开始，指令集发生了变化了”，而SWAP指令的参数n则是指指令解码树上某个结点。指令解码树中每个结点都带有一个标号。内部结点（浅灰色）拥有两个分别指向左右子树的指针，而叶子结点（深灰色）中则含有实现相关操作的代码。

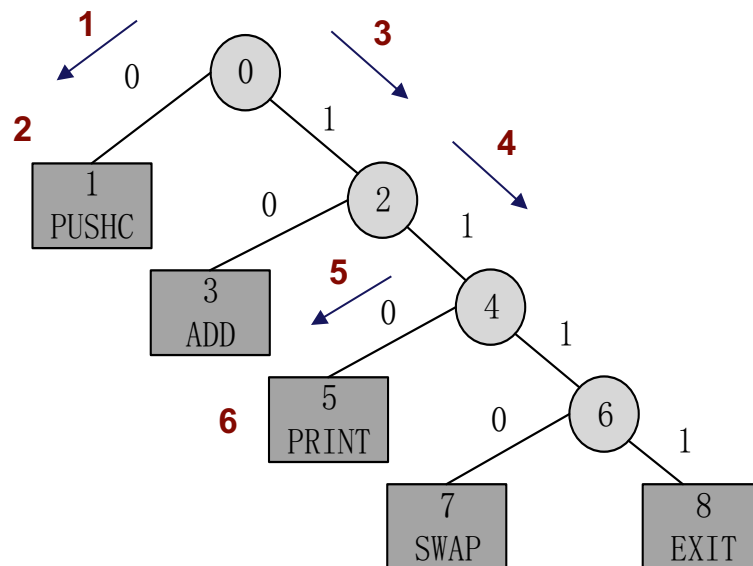
```
PUSHC 2
PUSHC 5
ADD
PRINT
SWAP 0
PUSHC 2
PUSHC 5
ADD
PRINT
EXIT
```



## 5.5、修改指令编码方式

解码：例如，对<0, 0, 1, 1, 1, 1, 0>这样一串机器码进行解码。

- 1. 先从解码树的根结点往左走（因为这串机器码的第一个bit位是0），来到结点1，执行PUSHC指令；
- 2. PUSHC指令的操作数是一个占3个bit位的整型数，所以机器码中接下来的3个字节<0, 1, 1>就是PUSHC指令的操作数。也就是说把常量3压入了栈中；
- 3. 剩下的bit串<1, 1, 0>，接着从解码树的根结点开始，往右走（因为第一个bit是1）来到结点2；
- 4. 接着在往右走（因为接下来的bit位还是1）来到结点4；
- 5. 最后再转向左（因为最后一个bit是0），执行PRINT指令；
- 6. PRINT指令的作用是弹出栈顶元素（就是刚才压入的3），并把它打印出来。



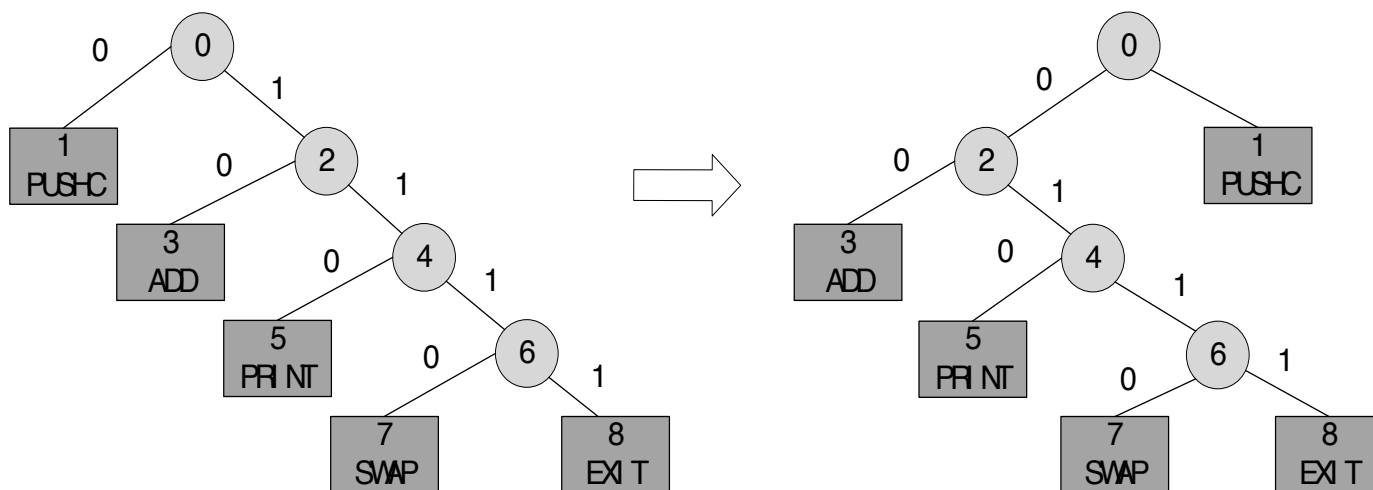


## 5.5、修改指令编码方式

混淆程序所要做的就是产生新的解码树，把原来程序中的各条指令转换成用新的编码方式编码的指令。

算法增加了一个SWAP指令，它能在运行时临时更改指令编码方法。其中，对SWAP指令做了简化处理，它只是交换解码树中的第n号结点的两个子树的位置。在算法中，这个SWAP指令实际上能对整棵解码树进行任意修改。

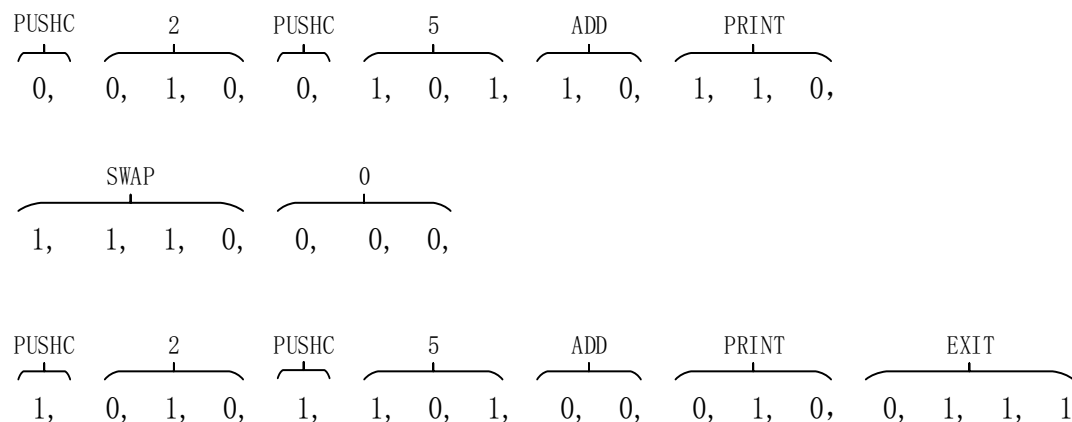
如图所示为执行了SWAP 0指令之后解码树发生的变化。



## 5.5、修改指令编码方式

把示例程序解码出来:

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



注意SWAP指令执行前后指令编码方式的变化。例如，在SWAP指令执行之前，ADD指令的编码是 $\langle 1, 0 \rangle$ ，SWAP指令执行后，ADD指令的编码就变成了 $\langle 0, 0 \rangle$ 。

## 5.5、修改指令编码方式

实现这样一个执行引擎并不困难，下面给出的就是解码树中各个结点的实现代码以及解码树是如何构成的：

```
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]);
}
```

## 5.5、修改指令编码方式

这个程序仅是一个bit数组，程序计数（PC）就相当于这个数组的索引号：

```
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 ;
```

此外，还需要一个decode( )函数，用它在运行时读取作为prog[ ]数组中的数据。遍历解码树，找到与其对应的叶子结点。而函数operand( )的作用则是从prog[ ]数组中读出一个占3个bit位的整型数来。

```
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++];  
}
```

## 5.5、修改指令编码方式

最后，是实现解释器本身，用它调用decode( )和operand( )函数对指令流进行解码，以及调用swap( )函数修改解码树：

```
static void interpret() {  
    int stack[]=new int[10];int sp=-1;  
    while (true) {  
        switch (decode()) {  
            case 0:stack[++sp]=operand();break;           //PLISHC  
            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; //SVTAP  
            case 4:return;                                  //EXIT  
        }  
    }  
}
```

## 6、替换指令

## 6、替换指令

替换指令是一种代码迁徙的方法，只是把某个指令替换成别的指令，即在程序运行到即将执行该替换指令时，把原指令写回去，指令执行完毕后，再把它替换成原来的替换指令。

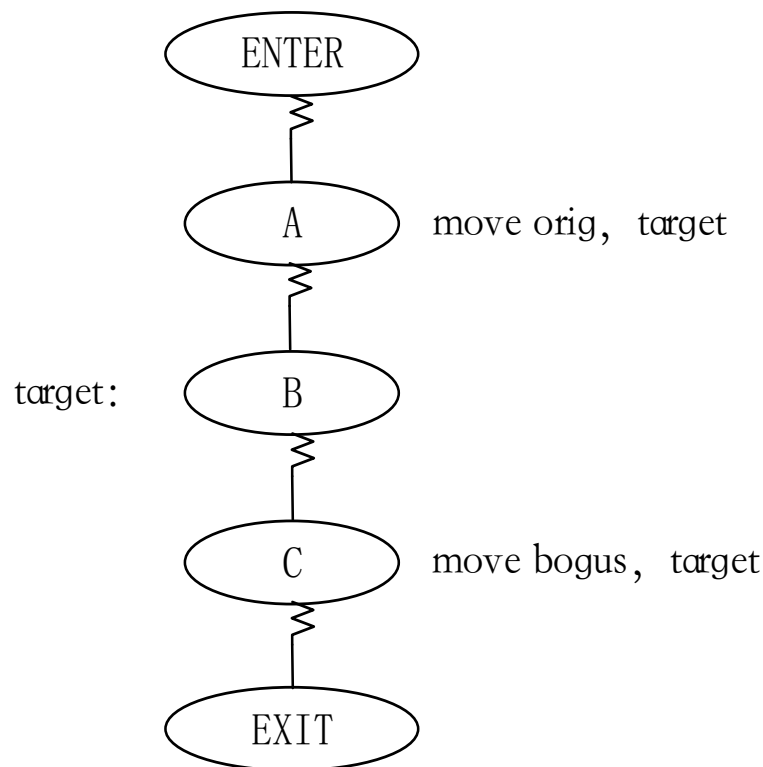
算法：替换指令，其中P表示被混淆的程序

- 1) 在程序P中选择3个点A、B和C，它们必须满足：
  - (a) A是B的严格必经结点；
  - (b) C是B的严格后向必经结点；
  - (c) 任何一条指令从B流向A的控制流路径都会经过C。
- 2) 令B上指令为orig；
- 3) 选择另一条与orig长度相同的指令bogus；
- 4) 把B上的orig指令替换成bogus指令；
- 5) 在点A插入指令move orig,  $v=B$ ，其中v是一个表示B地址的不透明表达式；
- 6) 在点C插入一条相应的指令move bogus,  $v=B$ ；

## 6、替换指令

算法的核心思想是在控制流图中找出3个点A, B, C, 如图所示:

- 在A这个位置需要插入一条指令, 把目标指令还原成其原始值orig, 令其恢复原样;
- 在C这个位置同样要插入一条指令, 把目标指令变成一个错误的值;
- A、B、C这3个位置必须满足条件: 每条流向B的控制流必然经过A, 且每条从B出来的控制流必然经过C。





## 6、替换指令

例子：简单的DRM多媒体播放器

```
int player_main (int argc, char *argv[]) {
    char orig = (*(caddr_t)&&target);
    (*(caddr_t)&&target) = 0 ;
    int digital_media[] = {10, 102} ;
    int len = 2 ;
    int player_key = 0xbabeca75 ;
    int user_key = 0xca7ca115 ;
    int key = user_key ^ player_key ;

    int i ;
    for(i = 0 ; i<len ; i++) {
        (*(caddr_t)&&target) = orig ;
    }
```

把调用printf函数的第一条指令替换成一个字节0。

```
int decrypted = digital_media[i] ^ key ;
float decoded = (float)decrypted ;
target:
printf( "%f\n" , decoded) ;
    (*(caddr_t)&&target) = 0 ;
}
```

```
int main(int argc, char *argv[]) {
    makeCodeWritable(...) ;
    player_main(argc, argv) ;
}
```

执行指令之前，把正确的指令写回来。

printf函数返回之后，再次改为替换指令。

## 6、替换指令

对这一算法，最容易被想到的攻击方法是使用模拟器，监视对指令流的写操作。下面这次攻击中，攻击者调用了mprotect，把相关的内存页设为可读和可执行，但不可写：

```
(gdb) call (int)mprotect(0x2000, 0x3000, 5)
```

```
(gdb) cont
```

```
Program received Signal EXC_BAD_ACCESS, Could not access memory.
```

```
Reason:KERN_PROTECTION_FAILURE at address:0x00002934
```

```
0x000028c0 in player_main(argc=1, argv=0xbffff31c) at kanzaki.c:30
```

```
30          (*(caddr_t)&&target) = orig ;
```

```
(gdb) x/I $pc
```

```
0x28c0 <player_main+220>:          stb      r0, 0(r2)
```

```
(gdb) print (char)$r0
```

```
$7 = -64
```

```
(gdb) print/x (int)$r2
```

```
$10 = 0x2934
```

当程序试图向代码中写入数据时，操作子系统抛出一个异常，发现是位于0x28c0的指令引发的异常。

这条指令试图把r0寄存器中的数值 (-64) 写入r2寄存器所指向的地址 (0x2934)。只要把0x28c0位置上的指令改为nop指令，再把地址0x2934上的数据改成-64即可。

# 小结

- 本讲内容介绍了5种静态混淆，1种动态混淆
- 代码混淆算法的应用非常广泛。
  - 最初，它被用来产生同一个程序的许多不同的变种。
  - 混淆算法的第二个应用领域是把程序变得足够复杂，使得逆向分析人员几乎无法分析出程序中使用的算法。
  - 混淆算法的第三个应用领域是隐藏秘密的数据（比如加密密钥）。
  - 最后，攻击者们也会利用混淆技术。例如，在非法复制大量的盗版软件之前，盗版者可以先对软件进行混淆转换，这会使得很难再从盗版软件中提取出水印来。
- 混淆是否可能是由对混淆的定义有多严格，想要保护什么东西以及所要保护的程序是属于哪种类型的程序等因素决定的。
- 绝对的混淆是不可能的，即确实有一些程序是无法被混淆的。