

# Software Security

Obfuscation

Guosheng Xu,  
guoshengxu@bupt.edu.cn

# 基本内容

## 1 、混淆概念介绍

## 2 、安全混淆理论

## 3 、安全混淆算法

## 4 、通用混淆器悖论

## 5 、安全混淆的出路

# 基本内容

## 1 、混淆概念介绍

## 2 、安全混淆理论

## 3 、安全混淆算法

## 4 、通用混淆器悖论

## 5 、安全混淆的出路

## 1 (1) 目的

通过把程序转换成另一种新的、需要保护的东西更难被推演出来的形式来保护程序里的某些秘密。

为了能给出这一需求形式化的定义，我们先给出一些基本的定义。

## 1 (2) 代码混淆定义

**混淆转换**：把  $T: P \rightarrow P_0$  记为一个程序到程序的转换。如果  $P$  和  $P_0 = T(P)$  **具有相同的可观测行为**，则称  $T$  为混淆转换。程序  $P$  和  $P_0$  应该满足如下的关系：

- 1) 如果  $P$  运行后无法中止或者以错误的状态中止，则  $P_0$  可中止也可不中止；
- 2) 否则  $P_0$  必须中止并且产生和  $P$  相同的输出结果。

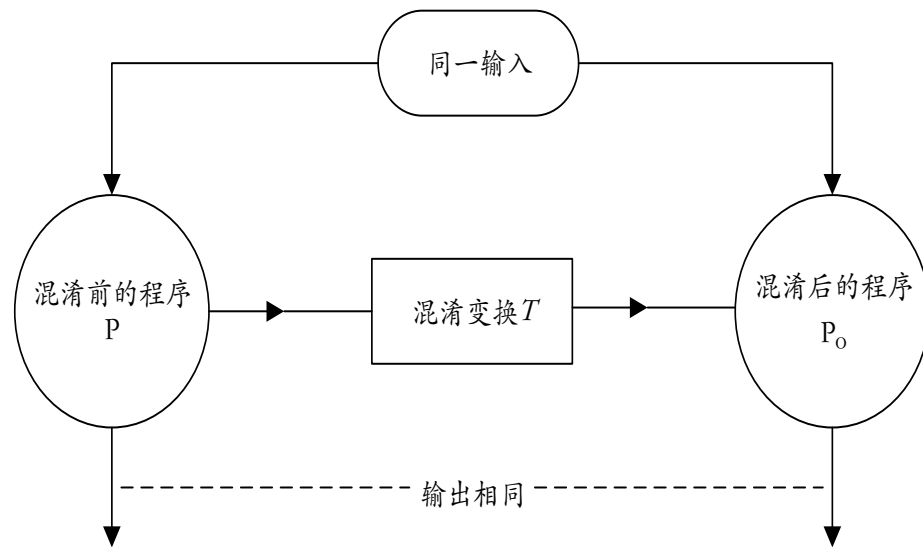
对于以上的定义，需要说明的是：

■ 混淆前后代码实现的功能是相同的。

■ “相同的可观测行为”是指软件使用者观察到的结果一样，但它们实际的运行过程肯定是不一样的。

■ 程序  $P_0$  也可以拥有其他的运行结果，也就是说程序  $P$  的运行结果应该是程序  $P_0$  的一个子集。

■ 对于程序  $P$  以及程序  $P_0$  的执行效率，定义中并没有提及。实际上，代码混淆技术都会或多或少给程序带来新的开销，通过代码混淆技术生成的代码的执行效率一般都低于原代码。



## 1 (2) 代码混淆定义

**有效的混淆转换**：令  $T$  是一个混淆转换， $P^\sigma$  是带有一个秘密属性  $\sigma$  的程序， $P^\sigma_o = T(P^\sigma)$  则是  $P^\sigma$  混淆后的版本。 $A$  是一个程序分析器，它能够分析程序，并从中找出  $\sigma$  ( $\sigma = A(P^\sigma)$ )，而  $\sigma_d$  则表示用  $A$  分析程序  $P^\sigma_o$  后得到的结果 ( $\sigma_d = A(P^\sigma_o)$ )。考虑  $P^\sigma$  和  $A$  的关系：

- 1) 如果  $\sigma_d \neq \sigma$ ，或者计算  $A(P^\sigma_o)$  要比计算  $A(P^\sigma)$  耗费更多的资源，则称  **$T$  是有效的**；
- 2) 如果  $\sigma_d \approx \sigma$ ，而且计算  $A(P^\sigma_o)$  和计算  $A(P^\sigma)$  耗费的资源一样多，则称  **$T$  是无效的**；
- 3) 如果  $\sigma_d \approx \sigma$ ，而且计算  $A(P^\sigma_o)$  比计算  $A(P^\sigma)$  耗费的资源还要少，则称  **$T$  是设计失败的**。

**可以实用的混淆转换**：令  $T$  是一个混淆转换， $P^\sigma$  是带有一个秘密属性  $\sigma$  的程序， $P^\sigma_o = T(P^\sigma)$  则是  $P^\sigma$  混淆后的版本。 $A = \{A_1, A_2, \dots, A_n\}$  则是一个程序分析器的集合。

对于  $P^\sigma$ ，如果有  $\exists A_i \in A$ ，根据  $A_i$ ， $T$  是有效的混淆算法，而对于其他  $\forall A_j \in A$ ，根据  $A_j$ ， $T$  不是失败的混淆算法，则称  **$T$  是可以实用的**。

## 1 (2) 代码混淆定义

混淆转换和攻击行为实际上都是抽象域中的抽象符号。这些域以格的形式存在，抽象域中的元素在格中的位置越靠近下界，它就越具体，越靠近格的上界，它就越模糊。如果代码混淆转换能够摧毁源程序中的某些属性，它就算是实用的。利用这一点，可以比较两个混淆算法的强度，即经过混淆转换后，保留了原程序中更多属性的算法比较差。

例子：为了形象地说明这个问题，比较以下两个混淆转换算法

$$T_1(x) = (2 \cdot x)$$

$$T_2(x) = (3 \cdot x)$$

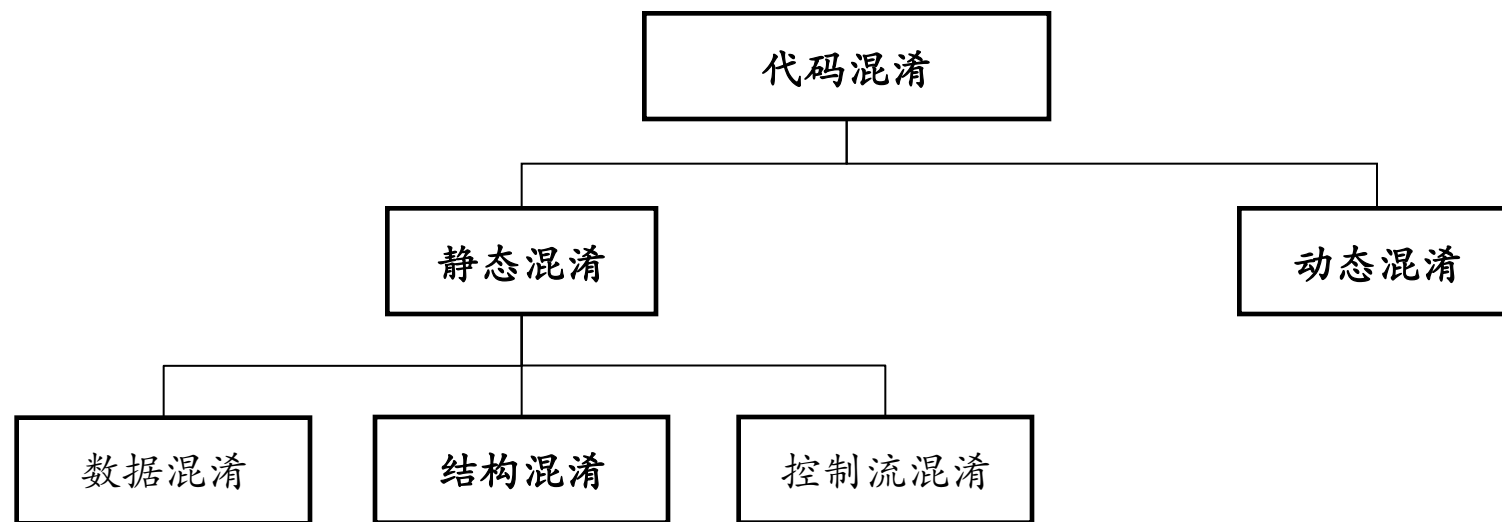
为了比较 $T_1$ 和 $T_2$ 两个算法的强度，现考察两个数据属性，符号和奇偶性。

$$\text{Sign}(x) = \begin{cases} -1 & \text{若 } x < 0 \\ 0 & \text{若 } x = 0 \\ 1 & \text{若 } x > 0 \end{cases} \quad \text{Parity}(x) = \begin{cases} 0 & \text{若偶数}(x) \\ 1 & \text{若奇数}(x) \end{cases}$$

由于 $\text{Sign}(x) = \text{Sign}(2 \cdot x) = \text{Sign}(3 \cdot x)$ ，所以 $T_1$ 和 $T_2$ 都能保留符号属性；但 $\text{Parity}(x) = \text{Parity}(3 \cdot x)$ ，而 $\text{Parity}(x) \neq \text{Parity}(2 \cdot x)$ ，所以经过 $T_1$ 转换后原来数据的奇偶性会被破坏，经过 $T_2$ 转换后奇偶性这个属性仍被保留下了。因此 $T_1$ 比 $T_2$ 保留的属性更少， $T_1$ 是比 $T_2$ 更好的混淆转换算法。

# 1 (3) 混淆技术的分类

根据混淆的原理与所需进行混淆处理的对象的不同，混淆技术可以分为4大类：**数据混淆**——混淆程序中所使用的数据、**结构混淆**——破坏程序的类、模块和函数（Java中称为方法）等结构，**控制流混淆**——隐藏程序执行时的控制流结构、以及**动态混淆**——混淆器通过在程序中插入一个解释器使得程序在运行时不断地改变自身的代码，其中数据混淆、结构混淆、控制流混淆属于静态混淆。本章将介绍这些静态混淆。





## 1 (4) 混淆技术的评估

代码混淆技术应该从**强度、适应性、开销以及隐蔽性**这4个方面进行评估。

1) 从**强度**对代码混淆技术进行评估是指根据混淆算法在程序中引入的混乱程度有多大来衡量混淆的强度。混乱程度可以用**软件复杂性度量**中的一系列用来评价软件的设计是否合理、结构是否良好的术语来评估进行代码混淆之后，程序原有的结构被破坏的程度。

2) **适应性**是指经过混淆后的代码能够在自动化的反混淆器的攻击下撑多长时间。它由两方面组成，有针对性编写一个自动化的反混淆器需要多长时间，以及使用这样一个反混淆器对代码进行反混淆需要多长时间。

3) 从代码混淆引起的**开销**来说，大多数的混淆转换会使 **$P_0$ 比 $P$ 更大，运行速度更慢，消耗更多内存**，而且有时候这3个副作用会同时出现。

在实践中，可以选用各种基准测试工具集（比如编译器开发人员常用的SPEC基准测试）来评估混淆转换引起的开销。用算法 $T$ 混淆程序之后，使用基准测试工具集提供的输入数据集分别运行原始程序和混淆后的程序，比较执行二者所需的时间和内存开销。

# 1 (4) 混淆技术的评估

4) 可以把**隐蔽性**问题粗略地理解为经过代码混淆的代码在那些未经过混淆的代码中到底显得有多突兀。把隐蔽性分为两类：位置隐蔽和整体隐蔽。

- 如果攻击者无法判断程序P是否经过了算法 $T$ 的转换，那么保护算法 $T$ 是整体隐蔽的；
- 如果攻击者无法找出程序P中哪段代码被算法 $T$ 转换过，那么保护算法 $T$ 是位置隐蔽的。

在讨论隐蔽性时，是在整个软件的所有程序组成的全域 $U$ 中进行讨论的。例如，某个算法会往程序中插入大量的xor指令，如果它保护的是一个与密码学或位映像图相关的程序，这个算法就是隐蔽的；如果它保护的是一个用于科学计算的程序，那它的隐蔽性就很差了。

## 1 (5) 理论相关概念

转换的正确性：被混淆了的程序的行为应该和它被混淆之前一模一样。

定义1.1 (正确性) 令 $I$ 是程序 $P$ 所有可能输入的集合 $Z$ 中的一个，当且仅当 $\forall I \in Z, T(P)(I) = P(I)$ 时，我们才认为对程序 $P$ 的混淆转换 $T$ 是正确的。

## 1 (5) 理论相关概念

定义1.2 (asset) 要保护的资产/秘密？

$\text{asset}(\cdot)$ 是指一个能从程序 $P$  及其输入集 $Z$ 中分析推导出的东西，例如  
 $\text{asset}(P, Z) = 1$ .

你想要隐藏的asset一般都有哪些呢？你可能想要保护程序中的数据，比如嵌在程序中的密钥、程序对数据的处理方式。你还可能想要保护程序中的某些功能，比如正则表达式、算法，或者其他一些东西的组合。

## 1 (5) 理论相关概念

例如，如果你想隐藏程序的真实大小和程序中所使用变量的数量，那么就可以定义

$$\text{asset}(P, Z) = (\text{length}(P), \text{count}(\text{variables}(P)))$$

例如，你可能只是想防止攻击者确定程序中是否有一个叫foo的变量。这时，`asset(.)`就可以写成

$$\text{asset}(P, Z) = \text{foo} \in \text{variables}(P)$$

有时你对`asset`的定义可能会更复杂些。例如你想隐藏的也许是程序P绝对不会输出某个特定的值x，这样一个事实。这时`asset`就可以这样写：

$$\text{asset}(P, Z) = \forall I \in Z: P(I) \neq x$$

## 1 (5) 理论相关概念

### 定义1.3 混淆转换

令 $P$ 是基于输入集 $Z$ 的程序,  $asset(P, Z)$ 为程序中你想要保护的东西, 令

$$m(P, asset(.)) \in [0, 1]$$

为分析出 $asset(P, Z)$ 的难度。  $T(P)$ 为对 $P$ 进行语义保留的转换。只有当

$$m(T(P), asset(.)) > m(P, asset(.))$$

时, 我们才认为 $T(P)$ 是对 $P$ 的混淆转换。

但是你怎么才能算出 $m$ 的值(推算出 $asset(.)$ 的难度)呢?

这当然是同 $asset(.)$ 到底是指什么紧密相关的, 例如把程序中指令的数量或者程序执行所需的时间作为一个 $m$ 的近似值。

## 1 (5) 理论相关概念

定义 1.4 (强效的混淆转换)

当  $m(P, \text{asset}(.))/m(T(P), \text{asset}(.)) < \varepsilon$

时，对P 进行的混淆转换T才是强效的。

我们真正想要的是强效的 (**strong**) 混淆转换，且使人难以对混淆后的程序进行逆向分析。换言之，强效的混淆是指推算出混淆后程序中的`asset(.)`所需的代价远大于推算出混淆前的程序中`asset(.)`所需的代价。

## 2、可被证明是安全的混淆:我们能做到吗

在本节中，我们将学习多种对代码进行混淆转换的方法。还将  
从理论角度检查代码混淆的安全性可证明性。并且会见到**2**个结  
论，这**2**个结论合在一起可以说明对于一个攻击者来说，**要得到程  
序中的某些asset是计算不可行的。**



阿兰·图灵告诉我们，即使是对于“未经混淆的”程序而言（也就是普通的程序），要检查其是否会停机也是不可能的。作为这个结论的直接推论，要计算某些程序中的其他一些属性也是不可能的。用上一节定义的混淆术语来讲就是，如果你需要保护的asset是关于程序是否会停机这个问题的，那么程序的原始状态就可以算是一种已经被混淆了的状态。

可是，在实践中，我们并不会凭空产生一个混淆了的程序，而是先产生一个正规的程序，然后使用混淆算法对其进行转换。我们将向你展示，Andrew Appel如何根据这一事实推导出了他的结论：在这种情况下中，攻击者找出隐藏的asset（比如上面讲的哪种关于图灵停机问题的asset）就不再是个不可计算问题了，它充其量至多也只是个NP困难问题。

## 2.1图灵停机问题

# 2.1图灵停机问题

■ 结论：要检查某些程序是否会停机也是不可能的。

证明思路：

1. 图灵首先假设存在这样一个能确定任意程序是否能够停机的布尔函数HALTS，HALTS函数的参数包括被测试的程序P和P的输入input。
2. 证明HALTS是无法正确判断这个新程序是否能够停机的。由于在一开始我们假设HALTS能够判断所有程序是否会停机，而现在我们又证明了HALTS不可能完成这一假设，于是就反证出了我们最初的假设——不存在这样一个HALTS函数，即要检查某些程序是否会停机也是不可能的。

## 2.1图灵停机问题

### 图灵停机问题的检测器

**COUNTEREXAMPLE**与**HALTS**的行为非常类似，它也是接受两个参数，一个代表程序的字符串和一个对程序的输入。

不过如果**HALTS**函数判定在输入数据**X**之后，程序**P**不会停机，**COUNTEREXAMPLE**就会让自己停机。反之，**HALTS**判断程序**P**会停机，**COUNTEREXAMPLE**就自己陷入一个死循环，不会停机。

```
public class COUNTEREXAMPLE {  
    HALTS detector = new HALTS();  
    static void dontHalt() {  
        while (true) {}  
    }  
    static void halt() {  
        System. exit 0 ;  
    }  
    public static void main(String args[]) {  
        String program = args[0];  
        String input args[l];  
  
        if (detector.halts(program, input))  
            dontHalt();  
        else  
            halt();  
    }  
}
```

## 2.1图灵停机问题

反证法的最后一步是把**COUNTEREXAMPLE**程序自己作为一个参数输入到**COUNTEREXAMPLE**程序中。根据代码，如果**HALTS**能够确定**COUNTEREXAMPLE**程序不会停机，那么**COUNTEREXAMPLE**就会自己停下来，于是判断失败；但是如果**HALTS**确定**COUNTEREXAMPLE**程序会停下来，**COUNTEREXAMPLE**程序却偏偏会陷入死循环，不能停机。

不管发生哪种情况，**HALTS**的判断结果总是错误的，即这样一个**HALTS**函数不可能存在。

■ 结论：要检查某些程序是否会停机也是不可能的。

## 图灵停机问题推论——程序等价的不可计算性

由停机问题的不可计算性还能导出其他一些不可计算的结果。例如，它可以证明程序等价问题（EQUIVALENCE问题就是证明两个程序完成同一功能）的不可计算性。

某些情况下，解决这个问题非常简单——比如，有个没有参数的程序，各自在屏幕上打印了“Hello World”和“Goodbye”后就退出了，很明显这两个函数是不同的。

要证明这个问题的不可计算性，需要构造出一对相互无法区分的程序。可以把EQUIVALENCE问题归为图灵停机问题一类的问题。具体来说就是，可以构造这样一对程序——如果能彼此区分就能解决图灵停机问题。当然是不可能的，所以也就证明了EQUIVALENCE问题的不可计算性。

下面给出程序：

## 图灵停机问题推论——程序等价的不可计算性

## 证明程序等价（EQUIVALENCE）问题的不可计算性

```

public class COUNTEREXAMPLE {
    HALTS detector = new HALTS();
    static void dontHalt() {
        while (true) {}
    }
    static void halt() {
        System.exit 0 ;
    }
    public static void main() {
        String program = "COUNTEREXAMPLE";
        if (detector.halts(program, ""/*没有输入*/)
            dontHalt();
        else
            halt();
    }
}

```

(a)

```

public class INFINITE {
    static void dontHalt() {
        while (true) {}
    }
    public void main() {
        dontHalt () ;
    }
}

```

(b)

很显然，程序（b）是不停机的，如果存在一个可以判断程序（a）和（b）是否相等的函数，那么这个函数就可以判断程序（a）是否停机，而这与上面的图灵停机问题矛盾。  
=>EQUIVALENCE问题的不可计算性

## 图灵停机问题推论——程序等价的不可计算性

如何把图灵停机应用到代码混淆中？

比如要混淆一个含有密钥的程序，这个密钥就是前面提到的asset。攻击者知道这个密钥是嵌入到程序中的，他想知道密钥的值到底是多少？要达到这个目标，攻击者可能使用的一个方法是，他拥有许多类似的程序，每个程序中都含有不同的密钥（输入程序集），他需要从这些程序中找出那个与被破解程序等价的程序。这时攻击者实际上就是要解决EQUIVALENCE问题，而这个问题总的来说是不可计算的。

乍一看，图灵关于停机问题的结论和EQUIVALENCE问题的不可计算性似乎说明了对程序中某些asset的攻击注定是不可能成功的。

利用这个推论，混淆就可以毕其功于一役么？这个推论无懈可击么？

实际上，攻击者的攻击手段多种多样，不能把攻击者手段限制为一种。攻击者可以利用其它方法得到混淆前的程序，请看下节。



## 2.2 对程序进行反混淆

所谓混淆就是对**程序进行会增加程序复杂度的转换**，而反混淆就是通过分析把**混淆给程序增加的复杂度去掉的转换**。可以把它认为是一种把程序变回混淆之前的样子的过程，即混淆的逆过程。

定义1.5 (反混淆转换)

对于给定的已知被混淆了的程序 $P'$ 和混淆转换 $T'$ ，如果转换 $T'(P', T)$ 满足：

$$m(T'(P', T), \text{asset}(.)) < m(P', \text{asset}(.))$$

则称 $T'$ 是对 $T'$ 的反混淆转换。

换言之，对于给定的被混淆程序和算法，反混淆就是要降低分析出asset的难度 $m$ 。

**2002年 Andrew Appel**已经提出，如果反混淆器知道混淆程序的算法，那么反混淆就不再是一个不可计算的问题。攻击者只要利用同样的混淆算法对猜的程序进行混淆，把结果进行对比。这样这个过程已经不再是不可计算的了。

### 3 可被证明是安全的混淆：有时我们能做到

根据上面提到的证明程序等价的不可能性，介绍几种可以被证明是安全的混淆方法。


### 3.1 (1) 混淆点函数

什么是点函数？

检查口令是否正确的函数，统称为点函数。因为这类函数只有在某个特殊的点上才会为真，故而得名。

例如，通常在登陆系统时，**login**函数首先会检查输入的口令与系统存放的用户口令是否一致：

```
boolean isValidPassword(String password){  
    if(password.equals("mypassword"))  
        then return true;  
    else return false;
```



而在实践中绝对不会用到这种程序，因为这个程序中口令是以明文形式嵌入在程序中的，很容易被逆向分析出来。为了防止这一点，口令绝对不能以明文形式存放——必须经过混淆。

常常使用**hash**保护口令检查功能。

### 3.1 (1) 混淆点函数

程序中不再记录口令明文，而是记录口令的**hash**，用户输入的口令在被计算了**hash**之后，再与程序中记录的口令**hash**相比较。

```
boolean isValidPassword(String password){  
    if(password.equals("642fb631ad5e946766bc9a25f15dc7c2"))  
        then return true;  
    else return false;
```



计算指定**hash**对应的口令是一个非常难以解决的密码学难题，所以口令检查函数现在已经被混淆了。

还有什么东西能代替这些**hash**函数来进行混淆，并证明混淆是不可破解的呢？首先需要证明这个东西是单向的，即使知道了特定的输出信息，无法推出造成这种输出的输入信息是什么。如果有一个全局访问的函数，而且还有一个不错的随机数发生器，可以试试下面的函数：

## 3.1 (1) 混淆点函数

```
class RandomHash{
    Hashtables hashes = new Hashtable();
    Integer randomHash(String value){
        if(hashes.contains(value))
            return (String)hashes.get(value);
        Integer newhash = new Integer(Math.random());
        hashes.put(value,newhash);
        return newhash;
    }
}
```

在这个**RandomHash**函数中，如果一个对象第一次出现，那么就会在一个巨大的集合中，随机选取一个数字，分配给他并存储起来，然后把这个数字返回给调用者。在这之后，如果这个对象再次出现，就把存储起来的数字直接返回给调用者。

许多银行就是用类似的系统制造借记卡的。银行卡上的**ID**号是随机分配给用户的。当一个用户申请开通一张借记卡时，银行会随机选取一个**ID**号（对应于**RandomHash**函数中的随机数）分配给他。一旦给用户分配**ID**后，银行就会把用户的账号和卡上的**ID**号关联起来存储妥当。这样攻击者根据银行卡**ID**不能获取账户信息。

### 3.1 (1) 混淆点函数

有一点非常重要，银行必须保证用户账户与银行卡**ID**号之间的对应关系的保密性，不然整个混淆计划就全部破产了。可以想象银行总行里有这一台专门跑着**RandomHash**函数的服务器，他每次只接受一个给定的**ID**号，然后返回一个对应的用户账户。在数学上，这样的程序被称为“随机预言模型”（**random oracle**）。

### 3.1 (1) 混淆点函数

可以把有限个单点函数组合在一起，构成“多点函数”。因此对单点函数的混淆可以被照搬到多点函数上来。下面我们把刚才那个最原始的单点login函数扩展成一个可以检查多用户登陆的多点login函数。

```
boolean isValidPassword(String password){  
    if(password.equals("1password01"))  
        then return true;  
    else if(password.equals("2mypassword02"))  
        then return true;  
    else if(password.equals("3mypassword03"))  
        then return true;  
    else return false;  
}
```



```
boolean isValidPassword(String password){  
    if(sha1(password).equals("ce5a522e817efcac33decd1b667e28c277b5b5f0"))  
        then return true;  
    else if(sha1(password).equals("dca12bfecff189a98a274d7aad43aa7b4ee4eed2"))  
        then return true;  
    else if(sha1(password).equals("095978404280e6c43dcde9db7903f7c72aac109c"))  
        then return true;  
    else return false;  
}
```

可以把这种多点函数混淆应用到其他函数的有效混淆中，比如混淆分级访问控制和正则表达式。



### 3.1 (2) 混淆访问控制

访问控制策略比单用口令保护某个文件的访问权限这个方法更通用些。他对程序的一系列属性（读、写、执行等权限）进行设置，从而规定了哪些用户可以以何种形式访问这些文件。

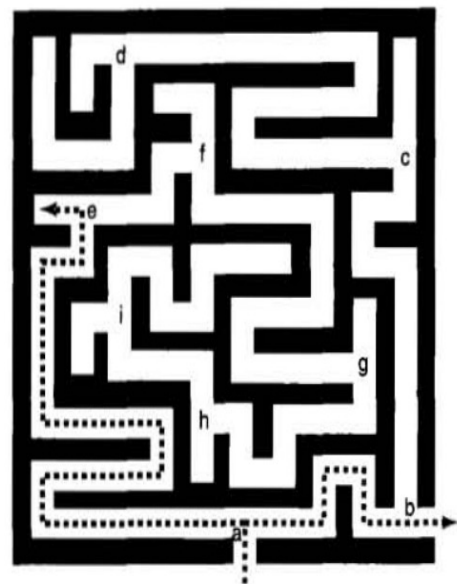
在实践中，由于这些实现访问控制的程序自身经常会运行在并非完全可信的主机上，所以经常会有混淆他们的需求。

例如，可能想使有些用户只能访问文件系统的制定子树，也就是说，如果用户有权访问文件系统的某个节点的话，他就有权访问该节点的所有子孙节点。

这样的系统可以用程序来实现：用户在遍历文件系统时，访问控制程序会要求他提供相应的权限信息，然后解析出他是否有权访问相关目录及其中的文件。为了防止他人窃取嵌入在访问控制程序中的秘密，这个访问控制程序就必须被混淆。

比喻为

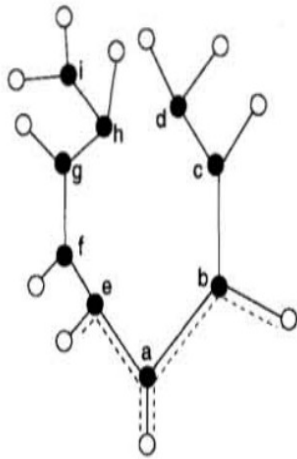
英雄的冒险经历：英雄往往被迫要搞定一个迷宫才能圆满完成一个任务，而在迷宫的每个岔路口，都必须解决一个难题。难题的解答为解决下一个难题提供基础，同时也显示下一步怎么走。



## 3.1 (2) 混淆访问控制

英雄的探险故事就相当于一个图的遍历问题。

迷宫转化为图遍历



图中每条边都有一个**password**控制着，英雄需要答对**password**才能通过这条边，每个节点都有一个名为**key**的随机长字符串，此外还有字符串变量**secret**和相邻节点的指针**neighbors**。相关代码如右边所示：

```
class Graph{
    Node[] nodeList;
    Node[] edgeList;
}

class Node{
    Node[] neighbors;
    String secret;
    Key key;
}

class Edge{
    Node source;
    Node sink;
    String password;
}

class Key extends String{}
```

## 3.1 (2) 混淆访问控制

英雄从起点u出发，越过*i*条边，到达节点v。英雄必须具有节点u的key，以及通过前往v的路径上的各条边的password才能完成任务——拥有正确的key说明英雄有权访问该节点，而拥有password说明英雄有权通过这条边前往下一个节点。通过了这条边后，用户也就到了下一个节点，同时也就得到了访问他的key。代码如下：

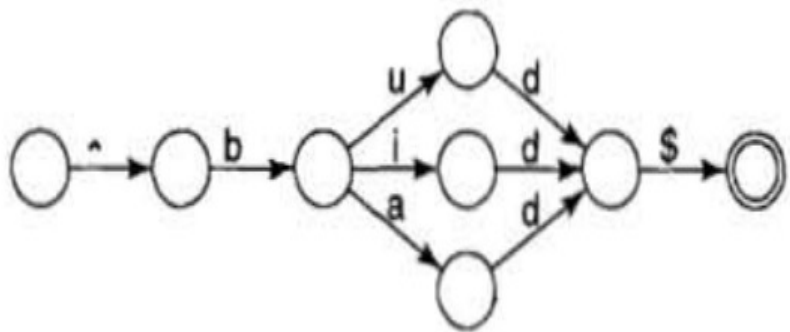
```
if (current.getKey().equals(key)) {  
    Node next = current.getNthNeighbor();  
    if (next.getPassword().equals(edgePassword)) {  
        return new Tuple<next, next.getKey()>;  
    }  
}
```

对每个节点的检查可以用一个多点函数表示——一个点用来检查key是否有权访问当前节点，另一个点检查是否有权通过下一个节点的边。

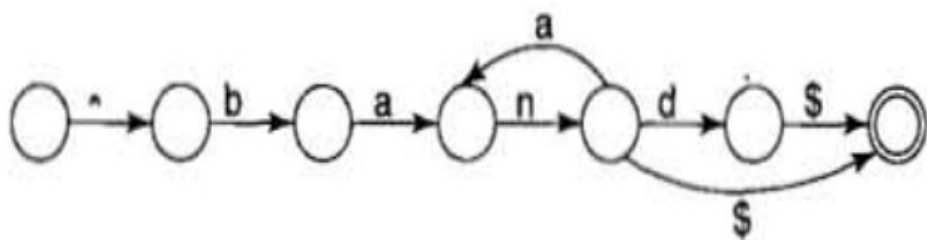
这两个是单点函数，可以用hash函数混淆它们了。

### 3.1 (3) 混淆正则表达式

正则表达式就是用有限自动机描述一组字符串的样式。这些样式包括连接（**ab**表示字符**a**后面必须是字符**b**），或（**a|b**表示下一个字符或者是**a**或者是**b**）以及重复（**a\***表示一串字符**a**，这一串**a**的数量可以是零个或无穷多个）。



这个有限自动机表示的是正则表达式  $^b(a|i|u)d\$$ 。表示任意以字母“b”打头的，后面接字母“a”、“i”或“u”，然后以字母“d”结尾的字符串。可能的组合是 {“bad”，“bid”，“bud”}



这个有限自动机表示的是正则表达式  $b(an)^+d?$$ 。表示任意以字母“b”打头，后接一个或者多个“an”，然后再接一个“d”结尾或者不加字母“d”直接结束的字符串。可能的组合是 {“ban”，“band”，“banand”，……}

能否对这种用正则表达式表达的字符串进行混淆？即能不能产生这样一个程序——它既能认出所有属于该集合的字符串，又不会暴露集合的具体内容？

### 3.1 (3) 混淆正则表达式

如果该正则表达式表示的只是有限个字符串（如`^b(a|i|u)d$`），那么答案当然可以。只要列出所有属于该集合的字符串，然后用多点函数进行混淆就行了。如下：

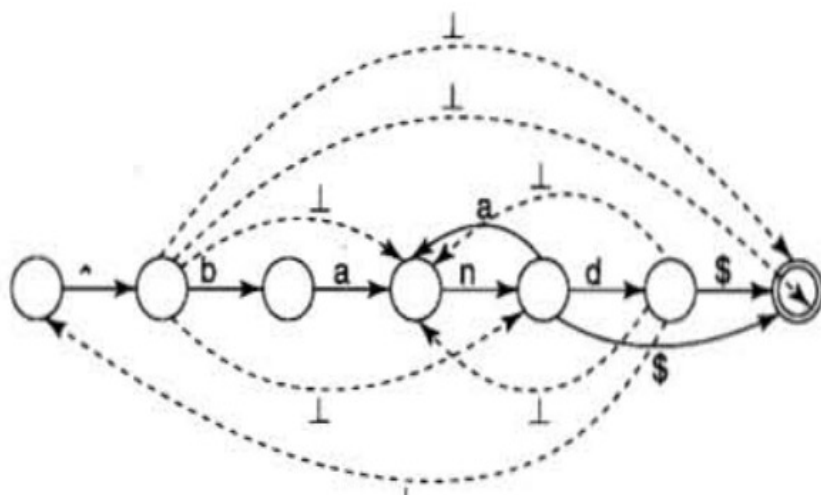
```
boolean matches(String test){  
    if(sha1(test).equals("ce5a522e817efcac33decd1b667e28c277b5b5f0"))  
        then return true;  
    else if(sha1(test).equals("dca12bfecff189a98a274d7aad43aa7b4ee4eed2"))  
        then return true;  
    else if(sha1(test).equals("095978404280e6c43dcde9db7903f7c72aac109c"))  
        then return true;  
    else return false;  
}
```

这个程序就是把正则表达式表示的所有字符串全都列出来了，然后再计算各个字符串的单向**hash**，并把计算结果放入程序中，要还原出程序的本来面目，攻击者就不得不对单向**hash**进行攻击。

如果正则表达式表达的字符串个数可以增长至无限多个，那么用多点函数进行混淆就不现实了。

### 3.1 (3) 混淆正则表达式

如前所述，正则表达式所表示的字符集可以用一个有限自动机来表示。我们可以把有限自动机表示正则表达式和多点函数两个技术结合起来，就能对类似访问控制系统的方法对正则表达式进行混淆。其核心思想就是把有限状态机中的**每条边代表的字符都用一个单向hash函数保护起来**。除此之外，还需要**隐藏有限状态机的结构**。要做到这一点，最简单的方法就是给有限状态机的每一对节点都添上一条同样由hash函数保护起来的却不代表任何字符的边。



原有的边代表的字符都用hash函数保护起来，标有“ $\perp$ ”符号的新边则没有对应的hash。此外还可以给状态机加上新节点，只要这些新节点绝对无法由正常的状态转换到达即可。这些新增加的边和节点有效地隐藏了自动状态机的结构

## 3.2 数据库混淆

数据库的保护在软件安全中非常重要，可以把混淆多点函数的方法用到混淆任意数据库中。一般对数据库的查询操作可以被当做是一个单点函数，用指定的条件对数据库进行检索，返回满足该条件的结果。我们可以利用点函数对数据库进行混淆。下面给出混淆算法。

OFUSCATEDATABASE(DB)

- (1)创建一个新混淆了的数据库DB；
- (2)对于原数据库DB的每一条由字段(key,value)组成的记录进行如下操作；
  - (a)生成两个随机数r1和r2；
  - (b)通过计算key和r1的hash，产生一个新的经过了混淆的主键字段；
  - (c)计算key和r2的hash，并把计算结果与字段value的值进行异或运算，得到一个新的经过混淆的数据字段；
  - (d)把混淆的key和value保存到DB中。

我们通过一个电话本为例来说明这一算法



### 3.2 数据库混淆

左图假设是我们常见的电话本数据库，数据库**Phonebook**的每一条数据都是由字段<key, value>组成的，比如我们要找Alice的电话号码，只要在数据库**name**字段中寻找“**Alice**”，然后返回相应的电话号码 **phone** 就可以了。对原始数据库中的每条记录<**name,number**>，在新的数据库中都对应下面这样一条记录：

$$(hash(concat(r1, name)), hash(concat(r2, name)) \oplus number, r1, r2)$$

混淆后的数据库如下所示：

#### Obfuscated Phonebook

Obfuscated name	Obfuscated phone	r1	r2
hash(15144Alice)	hash(15144Alice) $\oplus$ 555-1100	15144	16783
hash(11114Bob)	hash(11114Bob) $\oplus$ 555-3124	11114	87346
hash(13623Charles)	hash(13623Charles) $\oplus$ 555-1516	13623	46395
hash(12378David)	hash(12378David) $\oplus$ 555-9986	12378	35264
hash(11114Einstein)	hash(11114David) $\oplus$ 555-7764	11114	25234

其中**r1**和**r2**都是存放在新数据中的随机数，占了单独两列，在混淆后的数据库中寻找**Alice**的电话号码是这样的：先从数据库找到**Obfuscated name**的值等于**hash(concat(r1,"Alice"))**的那条记录。然后使用记录**r2**的值计算**hash(concat(r2,"Alice"))**，并把计算结果与**Obfuscated phone**异或，得到电话号码。



### 3.3 同态加密

在实践中，假设数据库的数据是以加密形式存在的，如果想对这些数据进行计算（加减乘除），必须要对这些数据解密么？能不能不经过解密直接计算呢？如果可以，这些运算操作有没有什么限制呢？

#### 同态加密（homomorphic encryption）

所谓“同态加密”就是这样一类加密算法——我们可以直接对加密后的数据进行一定运算，运算的结果一定等于我们把数据解密出来后进行一些数学运算（可能与直接对加密数据进行运算的运算类型不同）后，再将运算结果加密回去后得到的值。现有的不对称加密算法中有不少算法，比如RSA，对于某些运算操作都具有这一特性。

## RSA算法回顾

Alice用下列方式计算他的公钥，她先要选取两个大的素数 $p$ 和 $q$ ，并且计算 $n=pq$ 。同时，她也要计算 $n$ 的欧拉函数 $\varphi(n) = (p-1)(q-1)$ ，Alice接着选取一个满足 $\gcd(e, \varphi(n))=1$ 的整数 $e$ 。最后，她还要求出一个整数 $d$ ，使得 $ed \equiv 1 \pmod{\varphi(n)}$ ，那么，Alice的公钥就是 $(n, e)$ ，她的私钥就是 $(n, d)$ 。

当要加密一个数字 $m$ ，并把它发送给Alice时，只需计算 $E(m) = m^e \pmod{n}$ ，并把 $E(m)$ 发送出去就可以了。Alice在接收到 $E(m)$ 之后，只要计算 $E(m)^d$ 就能还原出明文 $m$ 。

现在我们假设Alice用她的私钥加密了两个数字 $x$ 和 $y$ ，得到的密文分别为 $E(x)$ 和 $E(y)$ ，如果只能拿到Alice的公钥，能只对 $E(x)$ 和 $E(y)$ 进行运算就得到等价与对 $x$ 和 $y$ 进行的运算的结果么？

### 3.3 同态加密

当把 $E(x)$ 和 $E(y)$ 相乘时，得到的将是

$$E(x) \times E(y) = (x^e \times y^e) \bmod n = E(x \times y \bmod n)$$

结果就是用Alice的私钥加密 $x \times y$ 的值

可惜**RSA**的同态只支持乘法算法，有没有一种同态加密算法可以支持加法和除法操作？

**Paillier**加密算法介绍：

这是一种类似**RSA**的不对称加密算法。如果还存在另一个整数 $y \in Z_{n^2}^*$ （符号 $x \in Z_{n^2}^*$ 表示 $x$ 为模 $N$ 的平方剩余），使得整数 $z$ 满足 $z = y^n \pmod{n^2}$ ，那么就称 $z$ 为模 $n^2$ 的 $n$ 次剩余。计算 $n$ 次剩余是个非常难的问题，这也就构成了**Paillier**算法。

Paillier加密算法生成密钥：

- (1) 随机选择两个大素数， $p$ 和 $q$ 。
- (2) 计算 $n = pq$ 和 $\text{lcm}(p-1, q-1)$ 。（ $\text{lcm}(a, b)$ 表示 $a$ 和 $b$ 的最小公倍数）
- (3) 随机选择一个满足 $g \in Z_{n^2}^*$ 的整数 $g$ 。
- (4) 如果存在下面这个模 $n$ 的乘法逆元： $\mu = (L(g^\lambda \pmod{n^2}))^{-1} \pmod{n}$ ，那么说明 $n$ 必然能被 $g$ 的秩整除，其中： $L(u) = \frac{u-1}{n}$ 。
- (5) 那么加密的公钥就是 $(n, g)$ ，解密的私钥就是 $(\lambda, \mu)$ 。

### 3.3 同态加密

**Paillier**加密算法加密和解密:

假设 $m$ 表示加密前消息,  $c$ 表示加密后消息

$ENCRYPT(m, (n, g))$

- ①令需要加密的消息为 $m$ , 且 $m \in Z_n$ 。
- ②选取一个满足 $r \in Z_n$ 的随机数。
- ③按照下面这个公式计算密文 $c$ :  $c = g^m r^n \pmod{n^2}$

$ENCRYPT(m, (n, g))$

- ①令密文 $c \in Z_{n^2}^*$ 。
- ②按照下面这个公式计算明文 $m$ :  $m = L(c^\lambda \pmod{n^2}) \cdot \mu \pmod{n}$ 。

对于给定的公钥以及消息, 算法Paillier具有很好的同态性:

1.  $D(E(m_1) \times E(m_2) \pmod{n^2}) = m_1 + m_2 \pmod{n}$
2.  $D(E(m_1)^k \pmod{n^2}) = km_1 \pmod{n}$
3.  $D(E(m_1) \times g^{m_2}) = m_1 + m_2 \pmod{n}$
4.  $D(E(m_1)^{E(m_2)} \pmod{n^2}) = D(E(m_2)^{E(m_1)} \pmod{n^2}) = m_1 m_2 \pmod{n}$

### 3.4 白盒加密

目前，有很多设备，比如手机，机顶盒等都掌握在攻击者手里，攻击者掌握设备的完全控制权，如果在一个设备中执行加解密算法，攻击者很可能使用静态分析工具、调试器、直接读取内存中的数据等方法 and 工具找到软件中的密钥，我们把这一类攻击成为“白盒攻击环境”。

能用纯软件的方式把解密的密钥隐藏在软件中，同时还能正常的解密么？对这个问题的相关研究统称为“白盒加密”（whitebox cryptography）。这类研究的目的是要实现即使攻击者能够完全得到加/解密程序的源码并运行的情况下，也不能从中找到密钥。

(1) 选取一种加密算法 $\mathcal{A}$ ，记其加密过程为 $E_k(m)$ ，解密过程为 $D_k(m)$ ，其中 $k$ 为对称加密/解密的密钥， $m$ 为消息明文。

(2) 选取一个密钥 $skey$ 。

(3) 针对选定的密钥 $skey$ 和解密过程 $D$ ，构造一个白盒解密器 $D^{wb}(m)$ ，使得 $D^{wb}(m) = D_{skey}(m)$ 。

(4) 对 $D^{wb}(m)$ 进行混淆，使 $skey$ 能安全地隐藏在 $D^{wb}(m)$ 中，成为它不可分割的一部分。

注意， $D^{wb}(m)$ 只要一个参数，而 $D^k(m)$ 需要两个！从本质上说， $D^{wb}(m)$ 实际上就是 $D^k(m)$ 针对某个特定密钥的专用版，即一个常量的位串。

### 3.4 白盒DES加密

传统的DES加密操作是由替换、移位、异或这3中基本操作组成的。这些基本操作单独看都简单明了，很难直接混淆。首先提出白盒加密的作者Chow提出，把这些基本操作转换成一组随机且与密钥紧密相关的网格化查询表（lookup table）。

由于任意一个有限函数（包括DES中使用的线性和非线性转换）都可以用一个查询表来表示，所以它是用来模糊换位、替换以及异或操作之间差别和界限的理想工具。

$A$		$B$		$C$		$A \circ B \circ C$	
□狠	□级	□狠	□级	□狠	□级	□狠	□级
00	10	00	0	0	1	00	0
01	01	01	1	1	0	01	0
10	11	10	1			10	1
11	00	11	0			11	1

在白盒加密所使用的攻击模型中，攻击者不但能控制程序的输入，同时还能有选择地单独执行程序的一部分，并分析其行为。为了防止攻击者采用各个击破的方式把各个基本操作一个一个地识别出来，可以把相关的查询表混合成一张大的查询表。例如，依次使用 $A$ 、 $B$ 、 $C$  3张表代表的操作对数输入的数据进行操作，这时就可以把这3张表混合成一张大表 $D$ ，使 $D = C \circ B \circ A$ 。合成后的 $D$ 如图5-12所示。

### 3.4 白盒DES加密

如果被合并的操作的数目非常大（比如DES的整轮转换）而且程序中只嵌入静态的查询表，攻击者就无法从这张大表中把各个基本操作分析出来，自然也无法把密钥信息提取出来。

可惜的是，查询表的大小会飞速膨胀。要制作一张有 $n$ 个比特的输入，并有 $m$ 个比特的输出的表。就会需要 $2^n m$ 比特的空间来存放这张表。由于DES的每个分组都有64比特，很显然查询表太大。为了解决这个问题，可以把输入的数据分成更小的分组，用一组网格化的查询表来处理他们。

比如，把64比特分成每8比特一组，每组结合给定的密钥再结合运算步骤合并成查找表，给定一个密钥，输入和输出的关系是固定的，把从输入到输出的运算步骤查找表代替，可以在一定程度上隐藏密钥。但是攻击者是可以查找表中得到密钥的。为了不让攻击者把密钥从股东查找表提取出来。可以对这些查找表进行混淆。

内部编码:为了对查找表进行混淆，可以对查找表的输出进行随机化处理，然后这些输出在被输入到另外一个查找表中之前对其进行相应的逆转换。这种做法通常称为“内部编码”。



### 3.4 白盒DES加密

#### 内部编码

例如，可以对上述3张查找表A、B、C进行如下转化：

令  $A \rightarrow A' = r_1 \circ A$ ; 令  $B \rightarrow B' = r_0 \circ B \circ r_1^{-1}$ ; 令  $C \rightarrow C' = A \circ r_2^{-1}$

其中， $r_1$ 和 $r_2$ 是对查询表输出u结果进行处理的随机化因子，而 $r_1^{-1}$ 和 $r_2^{-1}$ 是对应的逆操作。所以  $C' \circ B' \circ A' = C \circ r_2^{-1} \circ r_2 \circ B \circ r_1^{-1} \circ r_1 \circ A = C \circ B \circ A$ 。即使单独分析这3张查询表  $(C', B', A')$ ，也不会泄露密钥信息了。攻击者必须综合分析这3张表才能得出密钥信息，大大增加了攻击者付出的代价。

#### 外部编码

那么剩下一个问题就是，**第一张查找表的输入和最后一张查找表的输出（比如上面查询表A'和C'）还没有经过处理**。Chow等人建议使用外部编码。外部编码与内部编码类似，随机转换是应用在输入和输出上的。首先把随机因子结合到第一轮和最后一轮查找表中，在进行加密的时候，明文首先进行随机因子的逆操作，再查表，最后一轮也一样，密文经过最后一轮查找表后已经被插入的随机因子随机化了，他的值再经过逆操作即可。

准确的说，因为外部编码是放在解密器之外的人一部分进行的，位于解密器之外，所以是“外部”编码。

目前白盒加密算法的强度还不够，到目前为止还没有一种绝对安全的白盒加密算法。



## 4 绝对的混淆是不可能的

看了上面的成功案例，你现在是不是认为随着研究的深入，可以让这张可以混淆的 asset 的列表越来越长，直至涵盖程序中所有的属性呢？事实上，从最通用的角度讲，**混淆是不可能完全做到的。**

下面我们来介绍几个概念：

**黑盒程序：**即无法对其内部进行探究的程序。

再好的混淆也比不上一个黑盒程序，因为黑盒程序暴露的信息量是最少的。分析一个黑盒程序时，我们只能观察它的 I/O 行为（也许还包括运行它用掉的时间）。在实践中，不可能有真正的黑盒程序。相比之下，分析混淆后的程序时，我们可以观察程序使用的内存的情况、执行过的指令、调用过的函数以及使用其他方法。

如果能以这样一种方式访问程序，攻击者应该一定能够完成破解，找出程序隐藏的秘密。

下面我们来证明：**想把所有程序中的所有属性全部隐藏起来是不可能做到的。**

证明思路：我们构造一个程序——这个程序含有一个秘密，而且我们根本就无法通过混淆把这个秘密隐藏起来。

## 4 绝对的混淆是不可能的

通用混淆器：

能够隐藏程序中所有asset的混淆器。从某种意义上说，通用混淆器是能被设计出来的最终混淆方案，经过这般混淆的程序，除了能知道程序的输入和输出之间的关系外，将无法获得原始程序的其它任何属性。

黑盒访问模式：

对于一个程序，唯一能做的只是给它有限个输入，然后观察它的输出结果，我们把这种访问方式称为“黑盒访问模式”（**oracle access**）。

源码级访问：

对于一个程序，我们可以获得他的源码或者类似源码的访问。

## 4 绝对的混淆是不可能的

```
public class M{
    public int run(int curr){
        int steps = 0;
        while(curr <> 1) {
            if (curr%2 == 0) {
                curr = curr/2;
            }else{
                curr = 3*curr+1;
            }
            steps ++;
        }
        println(steps);
    }
}
```

(a)

```
public class ObfM{
    public int run(int curr){
        int steps = 0;
        while(curr <> 1) {
            curr = ++ steps ?
                (curr%2)*(3*curr+1)
                +((curr+1)%2*(curr/2)
                :steps*curr;
        }
        println(steps);
    }
}
```

(b)

```
> askoracle M 1
0
> askoracle M 10
5
```

(c)

在**a**中，对程序进行的是源码级访问；在**b**中，是对混淆后的程序**ObfM**进行源码级访问；**c**对**M**进行黑盒级访问。

## 4 绝对的混淆是不可能的

定义1（源码级访问中的可计算概率）对于给定的程序 $P'$ 以及一个分析程序中某些属性的源码级分析器 $A$ ，记 $\Pr[A(P') = 1]$ 为 $A$ 能分析出 $P'$ 中相关属性的概率。

定义2（黑盒访问中的可计算概率）对于给定的程序 $P'$ 以及其运行时间 $time(P')$ ，以及一个分析程序中的某些属性的禅宗级分析其 $R$ ，记 $\Pr[Sim^{P'}(1^{time(P')}) = 1]$ 为 $R$ 能分析出 $P'$ 中拥有属性的概率。

定义3（近似黑盒）对于混淆器 $O$ ，以及一个黑盒级的分析器 $R$ 和一个源码级分析器 $A$ 。如果对与一个属性全部被混淆了的程序 $P' = O(P)$ ，用源码级分析器所能分析出相关属性的概率就进似与黑盒级分析器所能分析出相关属性的概率，那么 $O$ 就可以认为是近似黑盒的 $\Pr[A(P') = 1] = \Pr[Sim^{P'}(1^{time(P')}) = 1]$ 。

定义4（大小增长可接受）如果对于任——一个程序 $P$ ， $O(P)$ 的大小最多是 $P$ 的大小的多项式倍数（多项式倍数增长已经是我们接受的极限了，对于指数增长我们认为是不接受的），则称 $O$ 是大小增长可接受的。

定义5（性能开销可接受）如果对于任意一个程序 $P$ ， $O(P)$ 的执行所需的时间最多 $P$ 执行所需的时间的多项式倍数，则称 $O$ 的性能开销可接受。

## 4 绝对的混淆是不可能的

### 定义 4.4 (混淆)

只有当满足下列条件时，才认为 $o(P)$ 程序是 $P$ 的混淆版本。

- $o$ 具有正确性。
- $o$ 的大小增长可接受。
- $o$ 的性能开销可接受。
- $o$ 是近似黑盒的。

定理4.1 ( 混淆是不可能完全做到的 ) 令 $\mathcal{P}$ 为所有程序组成的集合，对于任意一个给定的混淆算法 $O$ ，一定存在 $P \in \mathcal{P}$ ，使得 $O(P)$ 无法满足定义4.4的要求。

下面我们通过构造一段程序来证明定理4.1，我们只要证明两点即可。

- 1.即使程序经过了混淆，程序中的某个属性仍能通过源码级的访问方式被发现；
- 2.同时，该属性通过黑盒级的访问方法被推断出来的概率小到可以忽略不计。

## 4 绝对的混淆是不可能的

### 对混淆是不可能完全做到的证明：

下面给出一个对上述定理进行反证的程序Secret： Secret程序分为两个模式：点模式（point mode）和发现模式（spy mode），下面程序分别显示点模式和发现模式：

#### 点模式（point mode）

```
final int S=8544;
public int pointFunction(int x){
    if(x==S){
        return 1;
    }else{
        return 0;
    }
}
```

在点模式下，如果输入是密钥值的话，程序就输出1，否则，程序就输出0。在点模式下，仅仅对Secret进行黑盒级的访问，想要猜出密钥的概率是很低的。

#### 发现模式（spy mode）

```
public boolean behavesLikeMe(Program p){
    int testPoint = S;
    int tests = 100;
    boolean result = true;
    do{
        int Presult = p.run(testPoint);
        if(Presult == pointFunction(testPoint))
            return result;
        else if(Presult==Program.RAN_TOO_LONG)
            return false;
        testPoint =
            (int) Math.random(Interger.MAXINT);
        tests--;
    }until (tests==0);
    return result;
}
```

在发现模式下，程序的输入是另外一个程序，姑且把输入的另外的程序叫做Candidate。Secret会去检查Candidate的行为是否与自己的一致。如果是，Secret会把密钥输出出来

## 4 绝对的混淆是不可能的

### 对混淆是不可能完全做到的证明:

右边程序是对点模式和发现模式的调用方法。

●点模式时：程序判断输入值是否和密钥相等，想要猜出密钥的概率是很低的。

●发现模式：程序的输入值是一个Candidate程序，程序会判断这个程序是否和本程序的行为一致。如果行为一致，则打印秘密S。

```
final int POINT_MODE=0;
final int SPY_MODE =1;
public void main(String args){
    int mode = getMode(args);
    Program p = getProgram(args);
    int input = getInput(args);
    switch(mode){
        case POINT_MODE:
            System.out.println(pointFunction(input));
        case SPY_MODE:
            {
                if(behavesLikeMe(p)){
                    System.out.println("The secret is "+S);
                }else{
                    System.out.println("no secret for you.");
                }
            }
    }
}
```

如果能够以源码级的方式访问Secret，只要把他自己当做输入运行程序，不管它经过了何种混淆，还是能让他把秘密S输出出来。可是，如果只能以黑盒级的方式访问它（攻击者拿不到Secret 的源码，或无法把Secret 输入到其它程序中），就无法获得秘密S。这与混淆的定义中“近似黑盒”的要求是矛盾的，所以也就以反证法证明了程序Secret是不可能被混淆的了。



## 回顾一下

1. 我们首先证明了图灵停机问题——并不存在一个可以检查一个程序是否可以停机的方法。
2. 接下来根据图灵停机问题，我们推出了程序等价的不可计算性。
3. 利用程序等价的不可计算性，我们介绍了对单点函数和多点函数进行混淆、对访问控制和正则表达式进行混淆、数据库混淆、同态加密和白盒加密。
4. 随着越来越多的成功混淆案例，那么我们怀疑是否对于所有程序存在一个通用的混淆方法呢？针对这个问题我们证明了对所有程序的绝对混淆是不可能的。

我们如此热衷于寻找一个可证明的安全的混淆算法失败了。

### 5. 如何跳出不可能的阴霾？

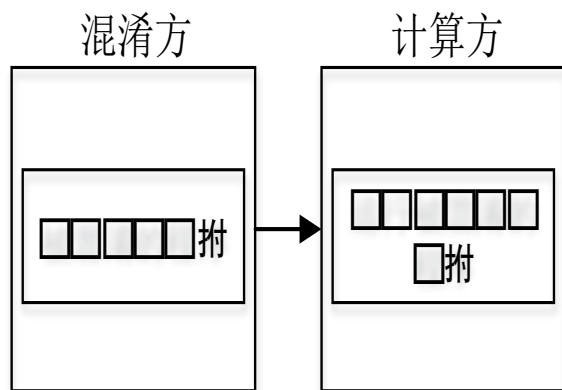
在对实践有指导意义的前提下，修改对混淆的定义，使定理4.1不成立。

- 1.构造交互式的混淆方法。
- 2.如果混淆不保留语义又当如何。

## 5.1 (1) 构造交互式的混淆方法

- 前文中对混淆的定义是基于这样一个假设的：当一个程序被混淆之后，它总是不会再和一个服务器端进行交互了。根据这一定义，关于混淆不可能完全做到证明就是在这基础上作出的。
- 如果允许一个被混淆了的程序执行分布式运算，这样攻击者就不能访问它的某些部分。这就相当于给混淆后的程序提供了一个黑盒。在黑盒里，程序可以安心地执行运算并存储数据。

那么，哪些数据和代码是可以不必放入黑盒的呢？这个问题的答案被称为“安全多方计算（SMC）”



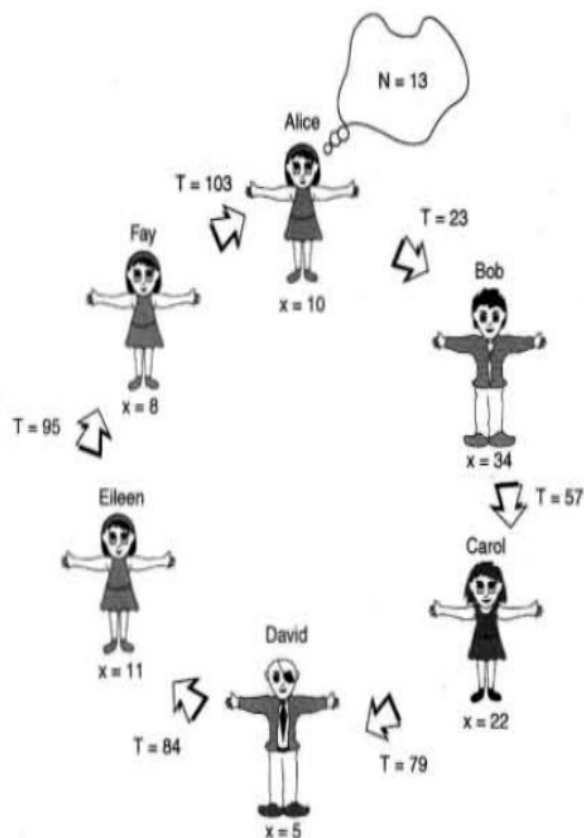
- 混淆方提供被混淆了的程序。
- 计算方希望运行被混淆后的程序。

混淆方提供asset，并且根据安全多方计算确保攻击者无法通过运行程序（参与到分布式计算中来）获得相应的asset。

## 5.1 (2) 小气鬼的问题

下面看一个多方安全计算的应用例子。

小气鬼问题：假设你和你的朋友想要算出你们所有人零用钱的平均值，但是你又不想让朋友们知道你具体有多少零用钱。能否设计一个既能算出平均值，又不会泄露每个人具体多少零用钱的方案呢？



先把所有参与计算的人排成一个圈，第一个人（比如说是Alice）想一个随机数 $N$ ，然后把 $N$ 与她所有的零花钱的总数相加，并把结果悄悄告诉Bob。Bob听到Alice告诉他的数字之后，再把他的零花钱总数加在上面，并把结果告诉下一个人，Carol。Carol再把她的零花钱加上去，并把结果告诉下一个人……最后的总数传递到Alice那儿之后，Alice再把结果减去 $N$ ，就得到所有人零花钱的总数，公布出来之后，大家把总数除以总人数就得到了零花钱的平均值。

- 怎么确保每个人说的是实话呢？

- 如果其中有人合作呢？

我们能做得更好么？能否对这个协议进行扩展，使之能算出更多东西而不光是零花钱的平均值吗？

## 5.1 (3) 百万富翁问题

百万富翁问题：假设你和你的朋友想要计算一下你们俩谁更富，同时你们不想让对方知道自己到底有多少钱。一个可行的方法是把你们俩的资产告诉一个可信第三方，由第三方对谁更富作出评判。另外一个方法可以使你无需第三方就能完成比富。假设Alice和Bob是两个百万富翁，如下是比富方案：

- (1) 令Alice的资产总额是 $a$ ，Bob的资产总额是 $b$ ，这两数均为1~5之间的整数，单位为百万。
- (2) Bob选择一个N-bit的随机数 $x$ ，用Alice的公钥进行加密运算得到 $c = E_{Alice_{pub}}(x)$ 并把 $C - b + 1$ 发送给Alice。
- (3) Alice用她的私钥解密出这样一组数字 $\langle Y_1, Y_2, Y_3, \dots, Y_5 \rangle$ ，其中 $Y_n = D_{Alice_{pri}}(C - b + n)$ 。尽管Alice不知道 $b$ 的值究竟是多少，但是他仍能完成这一操作。
- (4) Alice产生一个长度为N/2-bit的随机数 $p$ 。
- (5) Alice计算一组数字 $\langle Z_1, Z_2, Z_3, \dots, Z_5 \rangle$ ，其中 $Z_n = Y_n \pmod{p}$ 。
- (6) Alice把 $p$ 发送给Bob，同时还发送给Bob这样一组5个数字 $\langle Q_1, Q_2, Q_3, \dots, Q_5 \rangle$ ，其中，如果 $n < a$ ,  $Q_n = Z_n$ ，否则 $Q_n = Z_n + 1$ 。
- (7) Bob计算 $x \pmod{p}$ ，然后把结果和Alice发过来的 $\langle Q_1, Q_2, Q_3, \dots, Q_5 \rangle$ 这组数字中第 $b$ 个数字相比较，如果两个数字相等，则说明Alice比较富，否则就是Bob比较富。

## 5.2 如果混淆不保留语义又当如何

是否存在一种方法，这种方法不要求对混淆后的程序保持正确性，即混淆转换不必是语义保留的。如果混淆算法允许改变原有程序的输出结果——例如，为了阻止输出结果被攻击者利用，可以把程序的输出信息加密起来或者予以混淆处理。

而对程序的输出进行加密的操作放在远程执行是再适合不过了。

两种远程执行的方法

(1) 远程过程调用

(2) 白盒远程执行程序

## 5.2 (1) 远程过程调用

远程过程调用（RPC）协议是一种用以实现服务器-客户端式的分布式计算技术。它允许服务器启动并执行远程计算机（客户端）上的一个子线程（也被称为过程），并将运算结果返回给服务器。

●例如，使用固定密钥对加密消息进行解密的操作就非常适合用远程过程调用来实现。

下面我们介绍一种远程执行程序的方法，它的工作方式与远程过程调用非常相似，只不过，在这里会把可执行文件也编码成易于网络传输的形式，另外需要执行的程序还要被转换成可以在目标平台上运行的格式。

---

## 在服务器-客户端系统中的远程执行程序

---

### 服务器 : RPE\_SEND( $P, I$ )

(1) 把 $P$ 和 $I$ 编码成 $P'$ 和 $I'$ , 使之适于通过网络传给客户端。

(2) 把( $P', I'$ )发送给客户端, 并等待客户端返回结果 $O$ 。

(3) 对 $O$ 进行解码, 得到 $P(I)$ 。

### 客户端 : RPE\_RECEIVE( $P', I'$ )

(1) 对 $P'$ 和 $I'$ 进行解码, 分别得到 $P''$ 和 $I''$ 。

(2) 把 $I''$ 作为输入数据, 执行 $P''$ , 即

$O \leftarrow P''(I'')$ 。

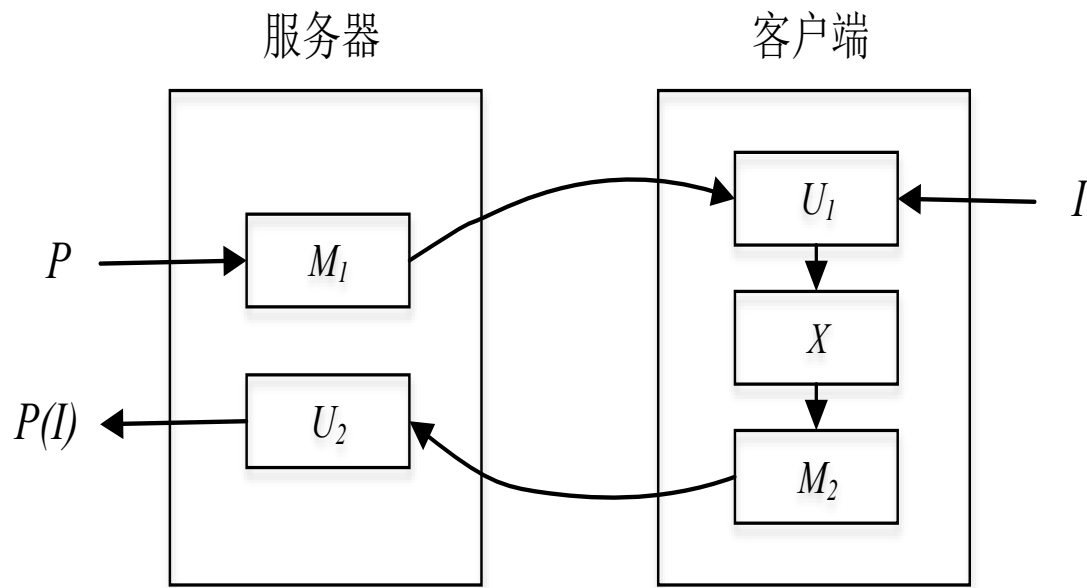
(3) 对结果 $O$ 进行编码, 并把编码后的 $O$ 传回服务器那里。

---



## 5.2 (2) 远程过程调用

服务器用转换器 $M_1$ 把程序 $P$ 编码成易于网络传输的形式，并把它发送到客户端。客户端收到它之后，用转换器 $U_1$ 把程序解码出来，并把它放到虚拟机 $X$ 中，用 $I$ 作为其输入予以执行，程序执行后的输出将在客户端用另一个编码器 $M_2$ 编码成易于网络传输的形式，并发回服务器。服务器收到它之后用转换器 $U_2$ 把信息解码出来。



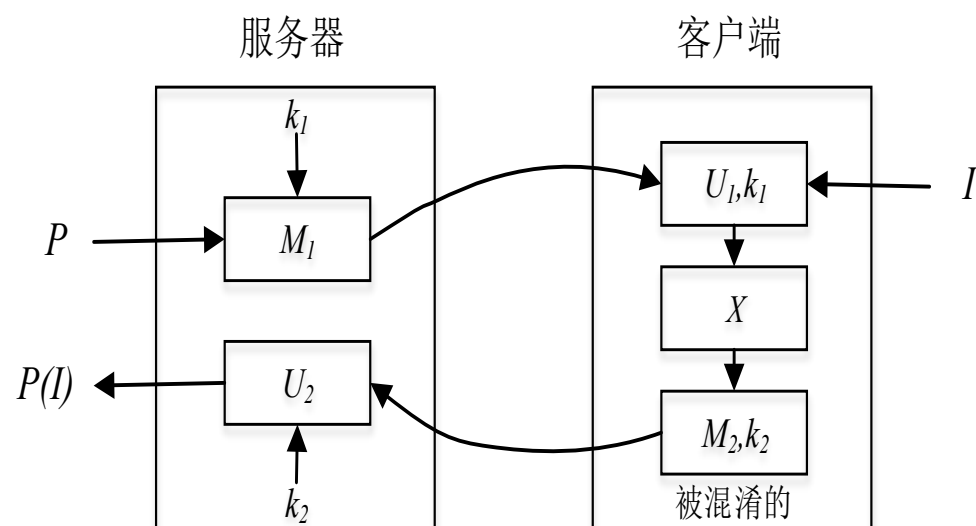
有两个缺点：

- (1)攻击者能够在服务器中获取程序，使用静态分析技术把需要保护的asset提取出来。
- (2)如果程序中需要保护的部分很大，运行所需开销就比较大。程序性能堪忧

## 5.2 (3) 白盒远程执行程序

接下来，我们介绍一种可证明安全的远程执行程序模型。在这一模型中，服务器并不会把程序编码成易于网络传输的形式，而是把它转换成能跑在某种特定虚拟机上的格式，并且还会对其进行加密。

整个客户端就是一个被混淆了的程序，由三部分组成：嵌有一个密钥的解密程序、一个虚拟机以及一个嵌有另一个密钥的加密程序。如下图：



服务器用转换器 $M_1$ 对需要执行的程序 $P$ 进行编码，并用 $k_1$ 对编码结果进行加密。然后服务器把结果打包发送给客户端，客户端首先用嵌有密钥 $k_1$ 的转换器 $U_1$ 对其进行解密，然后以 $I$ 为输入，在虚拟机 $X$ 中执行 $P$ 。 $P$ 的输出结果将在另一个转换器 $M_2$ 中，用密钥 $k_2$ 予以加密，加密结果将被发回服务器。服务器用 $U_2$ 解密它，并把解密结果返回给最终用户。在客户端中 $U_1$ 、 $X$ 和 $M_2$ 这三样东西是被放在一起进行混淆的，这就使得谁也无法单独把它们提取出来。当然要做到这点还是一个开放问题。

## 5.2 (3) 白盒远程执行程序

这个协议中有很多东西值得注意：

首先，加密解密的密钥是被保存在客户端里的，攻击者可能通过逆向工程找到它，所以需要一种能够把3个步骤混淆起来的技术，使得攻击者无法从中提取密钥信息。

其次，这个模型的安全性有赖于加密的输出不会泄露任何信息这一点，如果有信息泄露给了攻击者，他就能利用这一缺陷，构造一个Candidate程序再度证明绝对混淆的不可能性。

## 6 小结

我们首先介绍了图灵停机问题和程序等价问题，根据这些理论，我们如何鉴定哪些程序是可混淆的，哪些是不可混淆的。我们也看到了一个例子：用点函数去混淆访问控制策略、正则表达式以及构造混淆数据库。

同时，我们也讨论了加密算法，以及怎样利用加密算法中的一些特性直接对加密数据进行计算（同态加密）。但是，没有办法用同态加密的办法隐藏加解密本身，我们又介绍了白盒加密。

再见识了这些例子之后，我们推出了一个定理：绝对的混淆是不可能的——确实有一些程序是无法被混淆的。

在这一基础上，我们把程序分为各种不同类型，并分别对他它们进行混淆。我们对其中的两类程序进行了研究，他们分别是安全多方计算（其安全性源于多方参与的分布式计算）和白盒远程过程调用。