可证明安全

一个密码学协议的提出，在没有可证明安全的概念以前，只能经过时间才能得出是否安全。

可证明安全是一种规约方式。如果没有敌手能实现其目标则证明该协议是安全的。

过程为：

描述协议

描述敌手的目标（越低则方案越安全）

描述敌手的能力（越强则方案越安全）

构造安全模型（利用困难问题构造）

归约证明（归约到困难问题不可解上）

例子RSA签名

敌手的目标：

Total Break（完全攻破）

Universal Forgery（对任意签名伪造）

Selective Forgery（可选择对签名伪造）

Existential Forgery（对存在的签名伪造）

Strong Existential Forgery（对多次存在的签名伪造）

敌手的能力：

No Message（没有信息）

Known Message（知道信息）

Generic Chosen Message（知道通用信息）

Directed Chosen Message（知道定向信息）

Adaptive Chosen Message（可以选择性知道信息）

RSA加密

敌手的目标：

Total Break（完全攻破）

Decrypting a given Ciphertext（解密给定密文）

Obtaining useful information from Ciphertext（从密文中获取信息）

敌手的能力：

Key Only Attack（唯密钥攻击）

Chosen Plaintext（选择明文攻击）

Generic Chosen Ciphertext（选择密文攻击）

Adaptive Chosen Ciphertext 1（自适应的选择密文攻击）

Adaptive Chosen Ciphertext 2（自适应的选择密文攻击2）

在Adaptive Chosen Message情况下，可以抵抗Existential Forgery。说明此协议是安全的

开发环境是eclipse+jkd1.8

认证跳表的实现过程：

准备算法：

Hash函数——MD5码

定义四个初始变量：

**static** **final** **long** ***A***=0x67452301L;

**static** **final** **long** ***B***=0xefcdab89L;

**static** **final** **long** ***C***=0x98badcfeL;

**static** **final** **long** ***D***=0x10325476L;

一个初始函数:

g(i,x,y,z);其中g(0,x,y,z)=F(x,y,z)=(x&y)|((~x)&z);

g(1,x,y,z)=G(x,y,z)=(x&z)|((~z)&y);g(2,x,y,z)=H(x,y,z)=x^y^z;

g(3,x,y,z)=I(x,y,z)=y^(x|(~z));

移位数组s = {{7,12,17,22},{5,9,14,20},{4,11,16,23},{6,10,15,21}};

结果集result={A,B,C,D}

四种常用操作（)）

1. 其中a=b+((a+F(b,c,d)++)<<<s)
2. 其中a=b+((a+G(b,c,d)++)<<<s)
3. 其中a=b+((a+H(b,c,d)++)<<<s)
4. 其中a=b+((a+I(b,c,d)++)<<<s)

具体算法

数据填充fileFunction(String message)：

1：将message转化为字节数组byte，然后利用字节数组byte的长度算出需要填充的字节数x。

2：添加1和若干0直到长度等于x，计算此时的在计算message的二进制的低64位，此时再将此填充到byte数组中。

3：将处理好的byte数组按512长度分为若干组，并把每个组按64字节转化为long型，记每一组为。

数据处理dataHanding(long[] data)：

输入长度为16的,对其进行4轮操作。第一轮进行16次

第二轮进行16次…。

其中是代表第数据块的16进制数。

最后将16进制数转化为字符串进行拼接。

其中关键算法dataHanding(long[] data)的代码如下：

a = result[0], b = result[1], c = result[2], d = result[3]

for i=0:3 step 1

for j=0:15 step 1

result[0] += g(i,result[1], result[2], result[3])

+data[K[i][j]]+)

result[0] = result[1]

+ (result[0]<< ***S***[n][i] | (result[0] >>> (32 - ***S***[n][i])));

end

change(result[0],result[1], result[2] result[3])

end

result[0] += a;

result[1] += b;

result[2] += c;

result[3] += d;

测试：

分别对”hello”和”认证”两个字符串进行加密：

java程序：

hello：5D41402ABC4B2A76B9719D911017C592

认证: B7158A42CA1C2845B3C08111982B0494

在线MD5码加密（[https://md5jiami.51240.com](https://md5jiami.51240.com/)）结果：

hello：5D41402ABC4B2A76B9719D911017C592

认证: B7158A42CA1C2845B3C08111982B494

认证跳表实现

建立跳表结构：

跳表节点的结构如下

{

Long key;//节点的key

Object value;//节点原始的Hash值

SkipListNode[] forword;//用于存储前节点

SkipListNode[] back;//用于存储后节点

}

实现跳表的常用操作：算法见第四章

init();//初始化跳表

add\_Node();//增加节点

find();

delete\_Node();

change\_Node();

然后将跳表加工成认证跳表,

认证路径和Hash值的计算：

node = find(key);

initValue = node.value

while node.value – MAX !=0

node = node.next;

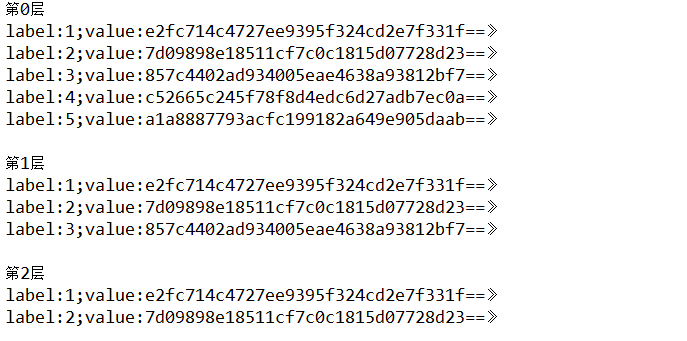
hash = culHash(initValue,node.value);//计算Hash值或运算

nodeArray.add(node);//认证路径的数组

end

return nodeArray hash

我将X:\\hash下的文件利用MD5消息摘要算出Hash值。跳表结构如下



图（5） 跳表结构的Java结果

每个节点存储的Hash值分别对应的是5个文件。任意文件的修改都会改变level=0的节点的Hash值从而影响到最后生成的Hsah。

我都修改key=3所对应的文件：

|  |  |
| --- | --- |
| key=1的Hsah值 | 修改前fffefdffffffffbf95f324cd2e7f331f |
| 修改后e7fefdffdfffffff95f324cd2e7f331f |
| key=2的Hsah值 | 修改前fdfefdf7ffffffbddb8f609c4f667f02 |
| 修改后e7fefdffdffffffdb55ae004c240ae60 |
| key=3的Hsah值 | 修改前e5feedf7ffffff9deae4638a93812bf7 |
| 修改后e5fefdffdffffffd56cc59b19ed215d2 |
| key=4的Hsah值 | 修改前e5aeedf7d7ffff9d4edc6d27adb7ec0a |
| 修改后e5aeedf7d7ffff9d4edc6d27adb7ec0a |
| key=5的Hsah值 | 修改前a1a8887793acfc199182a649e905daab |
| 修改后a1a8887793acfc199182a649e905daab |

从表格中可以看出：认证跳表结构确实能检验出数据的完整性，但是当我改变key=3所对应的文件是，key=4的认证路径没有经过被修改的文件，所以它认为此数据源没有被修改。如果想要确认整个文件的完整性，则必须从最小的节点开始，但是这要就要耗费比前一个节点更高的计算消耗。