摘要课题介绍

## 课题背景

## 研究意义

## 研究内容

# 相关基础知识

随着网络的发展，数据传输的速度也越来越快。密码学的重要性也越来越重要，消息认证是密码学的一个重要内容。本章详细的对密码学级密码学中消息认证的基础知识做出详细的介绍。具体包括：困难问题、消息认证的基础、Hash函数、以及典型的Hash算法SHA和MD5

## 困难问题

大整数因数分解问题

给定两个素数p，q，计算比较容易；

给定大整数n，求n的素因数p，q使非常困难。

例子：RSA算法是基于大整数因数分解问题提出的一种算法。分解的难度保证了RSA算法的可靠性，简单来说就是密钥长度决定了RSA算法的安全性。1999年，RSA-155（512bits）成功被破解。2009年RSA-768（1024bits）成功被破解。现在的RSA算法长度大多约为2048bits。

离散对数问题

已知有限循环群及其生成元g和阶。

给定整数a，计算元素很容易；

给定元素h，计算整数x，0≤x≤n,使得非常困难。

ElGamal算法

椭圆曲线离散对数问题（diffie-hellman）

格问题

## 认证的基本概念

定义

消息认证是指通过对消息或者消息有关的信息进行加密或签名变换进行的认证，目的是为了防止传输和存储的消息被有意无意的篡改，包括消息内容认证、消息的源和宿认证、及消息的序号和操作时间认证等。

认证和加密的区别在于：

加密用以确保数据的保密性，阻止对手的被动攻击，如截取，窃听等；

而认证用以确保报文发送者和接收者的真实性以及报文的完整性，阻止对手的主动攻击，如冒充、篡改、重播等。

认证系统常用的参数有：

口令、标识符、密钥、信物、智能卡、指纹、视网纹等。

对消息内容的攻击方式：

伪造消息

篡改消息内容

改变消息顺序

消息重放和延迟

### 认证系统的模型

Hash函数

MD5

RSA

三 云存储中的数据认证

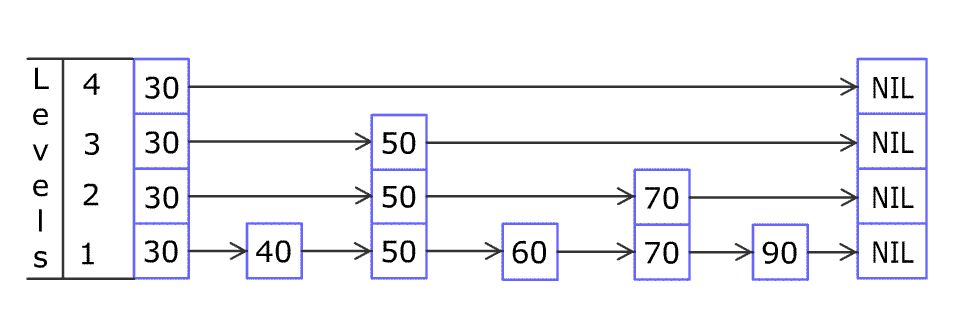
云存储作为一种线上的存储方式，我们肯定会怀疑存储的数据是否安全，是否被别人修改，数据认证成为了云存储中的一个重点研究对象。本章简述了云存储中数据认证的俩个重要的数据结构，以及典型的数据认证算法。

云存储中常用的两种数据结构

跳表

跳表全称跳跃表，是一个分层的链表结构，在链表原有的基础上增加了多层索引。一般来说，我们把最顶层最左边的节点称为起始节点。

跳表结构



Hash Tree

Hash Tree是基于Hash函数为基础构建的一个数据结构。是一种典型的以空间换取时间的算法之一

认证的设计模型

数据完整性验证机制的核心思想是：采用成员 问询策略，该策略最初被提出的动机是为了解决公 钥体制中的证书撤销问题 为基础的。所谓三方，是指云存储用户、不可信服 务器和可信数据源。在三方模型中，用户将数据外 包到外部服务器上，包括一个可信数据源和很多不 可信服务器。其中，可信数据源是完整而准确地保 存着用户的数据，它将数据再发布到各个不可信服 务器。用户需要使用外包数据时，只需访问其中一 个或几个不可信服务器即可。由于用户取回数据时 所访问的服务器是不可信的，这时就需要数据完整 性验证机制来确保用户所取回的数据是未被篡改和 伪造的原始数据。 [2-5] 成员问询可抽象为如下数学过程：将外包的数 据抽象成一个数据集 。这种策略将存储数据 的哈希表与一个具有可认证性的数据结构相结合， 即认证字典，使得成员问询的回答具有可验证性， 借此用户可验证所访问的外包数据是否完整。 1 2 (, , , )n S ee e = Λ ，其中 i e 表示 一个大小固定的数据块。验证某一数据块是否完整 的过程可抽象地看作为证明元素 x 是否属于数据集 S 的过程，其中 x 代表验证的数据块。具体实现过 程步骤如下： 1) 用户给不可信服务器发送一个问询请求，询 问元素 x 是否在数据集 S 中； 2) 不可信服务器在本地执行相应的查询操作， 将得出的答案Yes或No返回给用户，同时还要返回查 询过程中生成的“证据”； 930 电 子 科 技 大 学 学 报 第 43 卷 3) 用户根据不可信服务器返回的“证据”，进 行一系列计算，重新得出数据的特征值； 4) 用户将上一步骤中得到的特征值与本地的 数据特征值(由可信数据中心事先计算得出并传递 给用户，用户将其预存在本地)进行对比，两者相同 则用户接受不可信服务器返回来的答案，反之，则 拒绝。

云存储中典型的数据认证算法

1、基于MHT的数据完整性验证

默克尔哈希树是一种树型的数据结构，其叶子结点是可独立验证的数据块（或者间接数据）的Hash，任一内部结点是其孩子结点数据联接后计算得到的 hash 值。适用于快速验证某一数据块或少数数据块的完整性。通过节点i的值和snop算得根节点的hash值（snop：节点i与MHT根节点之间路径上每个节点的兄弟节点。在图中h3的snop为h4、n3、n2）则根节点hash值为root‘=hash(n2|hash(n3|hash(h3|h4)))，比较root’与原来的root节点的hash值即可。相对于传统的完整性检查，Merkel哈希树的时间复杂度大为降低，从0(n)降到为0(logn)。

2.基于跳表的数据完整性验证

跳表是从二叉树扩展出的一种数据结构，它是一种有序的多层次链表，如下图所示

在跳表中查找元素x，首先从左上角的头结点s开始，头结点s即为跳表中最左上角的节点，设v为查找到的当前节点，初始时，v=s，查找过程中只需要用到两个方式，右移和下移（跳表中的元素都是有序的），右移为在本行中向右查找直到找到第一个小于或者等于x的最大元素，下移也是同样的道理，那么在该图中查找元素14的过程如下：

本图中有S0-S4五个链，其中我们定义停滞元素为存在于Si-1而不存在与Si中的元素，高塔元素为即存在于Si-1又存在于Si中的元素，另外定义elem(v)为存在在节点v的元素，down(v)Si-1中节点v下面的结点；right(v)为Si中v结点右边的结点，跳表上的验证过程设计如下;

标签值f(v)的计算方法如下： 定义w=right(v),u=down(v),当right(v)=null时，定义f(v)=0。那么：

当u=null时，即v在So上：

（a）当w为高塔节点时，f(v) = h(elem(v),elem(w));

(b) 当 w 停滞节点时，f(v) = h(elem(v), f(w))。

当u不等于null时：

（a）当w为高塔节点， f(v) = f(u)；

（b）当w为停滞节点时，f(v) = h(f(u),f(w))

四 跳表

跳表产生的背景

平衡树可以用于表示抽象的数据类型如字典和有序链表，它通过树旋转（Tree Rotation）操作强制使树结构保持平衡来保证节点搜索的效率。在数据为随机插入的情况下，平衡树性能表现良好；但数据为顺序插入或者需要删除节点的情况下，平衡树的性能就会有些糟糕。

跳表可以作为平衡树的一种替代选择。它使用随机的平衡策略取代平衡树严格的强制的树平衡策略。因此它具有更简单有效的插入/删除方法以及更快的搜索速度。

跳表的是由链表构成

认证采用跳表这个数据结构是考虑到了很多因素，

1. 节点重排，以及顺序性的访问方式
2. 灵活的增删节点
3. 链表查询的效率低下

链表满足以上的前两的点，但由于查询的效率问题，从而对链表进行了改进。

单向链表越靠后末端的节点查询速度会越慢，双向越靠近中间会越慢。由于我们要在云存储上存储的文件越多，认证的结果就越有效率问题。

由于链表的查询效率的问题，对链表进行改进，从而衍生出了跳表。

跳表的相关操作：

查询节点：（如果当前节点存在，则返回当前节点。如果不存在则返回离此节点最近的）

定义左边界为head 右边界为 tail

令startNode = head  
1. 当 startNode.value < node.value 时 startNode.forward

2. 直到 startNode.value>=node.value ,startNode.down  
3. 重复1，2，直到startNode.down==null

4 return startNode;

public SkipListNode find(Integer value) {

int i = level-1;

SkipListNode node = head;

while(i!=-1) {

while(node.getForword()[i].getValue()<=value) {

node = node.getForword()[i];

}

i--;

}

return node;

}

删除节点

1通过查询 判断节点是否存在

2 存在 则把currentNode的表示前一个节点的值，赋给下一个节点

**public** **void** delete\_NodeByValue(Integer value) {

**if**(!isExsit(value)) {

**return**;

}

SkipListNode node = find(value);

**for**(**int** i =0;i<3;i++) {

SkipListNode pre = node.getBack()[i];

SkipListNode next = node.getForword()[i];

pre.getForword()[i] = next;

next.getBack()[i] = pre;

}

}

增加节点

**private** **void** add\_Node(Integer value) {

**if**(isExsit(value)) {

**return**;

}

SkipListNode addNode = **new** SkipListNode(value,level);

SkipListNode node = find(value);

**for**(**int** i = 0;i<=level-1;i++) {

SkipListNode next = node.getForword()[i];

SkipListNode pre = next.getBack()[i];

addNode.getForword()[i] = next;

addNode.getBack()[i] = pre;

}

**for**(**int** i = 0;i<=level-1;i++) {

SkipListNode next = node.getForword()[i];

SkipListNode pre = next.getBack()[i];

**double** num = rand.nextDouble();

**if**(i==0) {

pre.getForword()[i] = addNode;

next.getBack()[i] = addNode;

}

**else** **if**(i>0&&num>0.5) {

pre.getForword()[i] = addNode;

next.getBack()[i] = addNode;

}

**else** {

**break**;

}

}

}

修改节点

**public** **void** change\_Value(Integer value1,Integer value2) {

**if**(!isExsit(value1)) {

**return**;

}

SkipListNode node = find(value1);

node.setValue(value2);

}

跳表的性能

跳表的特点：

(1) 每个跳表由很多层结构组成；

(2) 每一层都是一个有序链表，且第一个节点是头节点；

(3) 最底层的有序链表包含所有节点；

(4) 每个节点可能有多个指针，这与节点所包含的层数有关；

(5) 跳跃表的查找、插入、删除的时间复杂度均为O(log N)。

跳表的实际应用：

应用场景：  
节点增加和更新比较少，查询频次较多的情况。

1、Lucene, elasticSearch

2、Redis：

Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的 是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

认证跳表