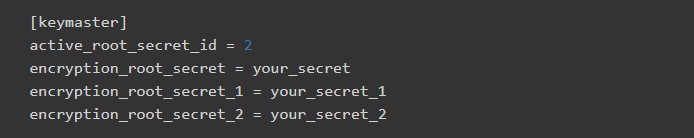
云系统上的安全存储

## Introduction

现今，云存储系统越来越受到公司和用户的偏爱，但是很多云系统的存储安全性不高，如何实现安全和便捷的平衡仍是一个问题。在本报告中，我们将展示一种云系统部署作为解决这个问题的可能措施。我们将针对[OpenStack Swift](Swift1.pdf)来测试我们的部署效果。OpenStack是一个开源的云计算管理平台项目，Swift是其下一个使用对象存储的云存储组件。我们将在Swift的架构中加入一个数据库层，实现加解密密钥的生成和存储。另外，我们修改了服务端和客户端的运行逻辑，赋予整个系统更大的权限控制粒度，让客户端有更大的自由选择加密的级别。

## Scope

Swift本身自带一个原始的加密方案，在这个加密方案中，使用AES加密算法，加密密钥被以明文形式写在代理服务器（Proxy server）的一个脚本文件中（xx.conf）。每一个代理服务器上，一次只允许一个密钥作为使用的密钥。这样的方案有非常大的缺陷，第一，如果当前



使用的key丢失（conf文件损坏）或者被篡改，那么使用该密钥加密的所有文件都无法被解密。第二，由于一个代理服务器只能允许一个密钥作为使用密钥，那么经过该代理服务器上传的文件，加密使用的密钥都将是相同的，这将会带来很大的安全风险，如果想更换新密钥，则每次都要进入代理服务器的conf文件中进行修改，这对系统的管理要求很高。第三，从客户端来看，这样的加密方案十分僵硬，客户端可能有对不同文件实行不同级别加密的需求，如同一组文件使用同一密钥，某些机密文件使用单独的密钥，而这个方案不能满足这些需求。第四，代理服务器本身也有传递文件的工作，把密钥文件直接放在代理服务器上，会增加代理服务器被攻击的风险，增加安全维护难度。综上，我们需要一个提升版的加密部署方案，增强Swift的安全性。接下来我们将展示一个新的加密方案，它将提升这个云存储系统的安全性，同时赋予更多的加密级别选择，整个系统将更加灵活。

## Contributions

**1.提升加密系统权限控制粒度和容错程度：**

Swift存储结构为account、container和object，原始加密方案对于用一个代理服务器以上都使用同一个密钥加密，加密粒度远远低于本身存储结构的粒度。在我们的方案中，加密提供三个级别加密，account、container和object和原始存储方案相对应。用户在每次上传都可以自由的选择对当前文件想使用的加密级别。另外，多个密钥可以增加系统容错度，降低由于单个密钥丢失等问题导致整个系统所有文件无法解密的风险。、

**2.增加了整个系统安全程度降低管理难度**

新的方案将密钥单独放置在一个数据库服务器中，对于一个攻击者，如果他要获得一个明文的文件，他必须同时攻破数据库服务器和存储服务器。通过将风险分散，我们可以更好的注重如何保护数据库服务器（防火墙或者安全模式控制等），而没有了密钥保护负担的代理服务器效率将得到提升。

## Design

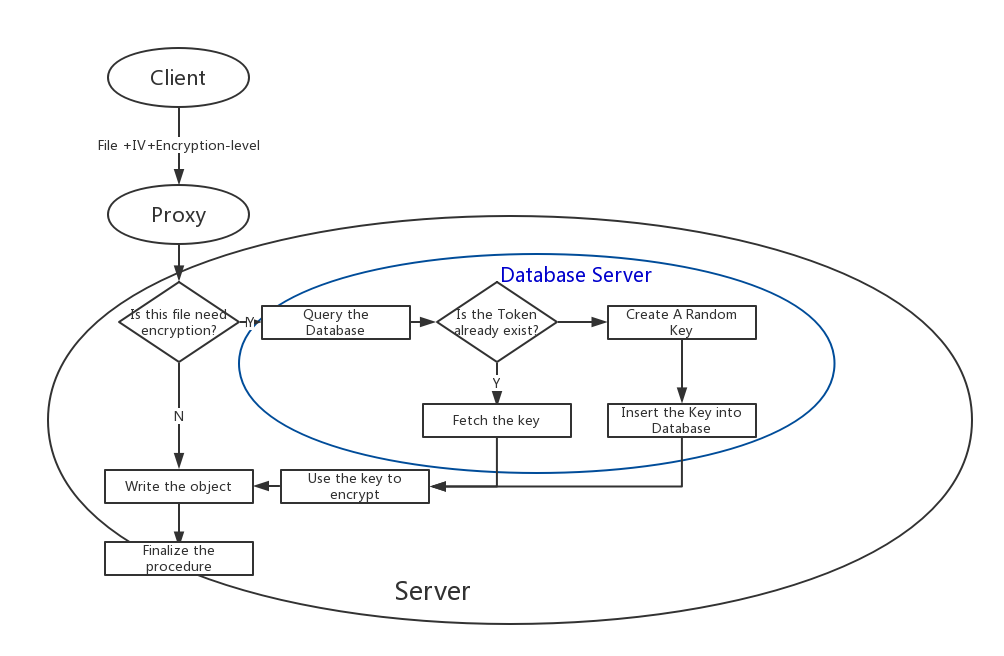


Fig 1: 上传文件时系统处理的整体概况

我们在整个系统中加入一个数据库层来存储和更新密钥信息，存储节点不再会使用来自代理节点的密钥，而是使用数据库中的密钥来加解密，以下是具体的加解密步骤。

当上传文件时：

1.客户端发送请求，请求的头部会有加密信息选项，选项中指定了需要加密的等级，并且把请求转给代理节点。

2.代理节点接受请求，并且查找头部是否有加密选项，如果选项中指定了加密的等级，代理节点将生成一个初始向量IV，并且把它加入到头部中，将新的请求转交给存储节点。如果不存在选项，则代理节点不做关于加密选项的处理，直接转交。

3.存储节点接受请求并且查看头部加密选项。如果存在，查找IV是否存在，如果不存在则自己直接生成一个（Swift中存在客户端直接与后端存储节点沟通的可能）。根据文件的标识信息（这里我们使用的是名称的md5），向数据库发送查询请求。

4.数据库查询结果，如果存在，则返回对应的密钥。

5.存储节点如果获得了密钥，则使用对应密钥加密输入块，并写入。如果没有获得密钥，则生成一个新密钥加密输入块写入，并把新的密钥条目写入数据库中保存。最后，存储节点把相关的加密信息，写入云上文件的Metadata之中，方便后期维护和查询。

当下载文件时：

1.存储节点获得了下载文件请求。节点将读取云文件的Body部分，并且根据Metadata中的信息向数据库提交一个查询请求。

2.数据库返回对应请求结果。

3.存储节点将body重新切分成块（为了解决对齐问题，后文会有更多阐述），使用获得的密钥解密解密块，并将块发送给客户端。

Class key\_generate(object):

#Create this engine while encrypting

def \_\_init\_\_(self,level,token):

result=query\_database(DBconn,level,token)

if result is not null:

key=result

else:

key=random(32)

insert\_database(DBconn,level,token,key)

self.keytext=key

Fig 2: 上传时获得加密密钥的伪代码

Class PUT\_object:

……

If ‘X-Encryption-level’ not in headers:

swift.writer(chunks)

else:

keye=keygenerate(headers[‘X-Encryption-level’],md5(token))

encrykey=keye.keytext

chunks.encrypt(encrykey,iv)

swift.writer(chunks)

……

metadata[‘Encryption-level’]= headers[‘X-Encryption-level’]

metadata[‘Encryption-token’]=token

metadata[‘IV’]=iv

put(metadata)

Class GET\_object:

If ‘Encryption-level’ not in metadata:

Swift.send(body)

Else:

chunks=split\_body(body)

keyd=keyfetch(metadata[‘Encryption-level’],md5(token))

decrykey=keyd.keytext

chunks.decrypt(decrykey,metadata[‘IV’])

body=join\_body(chunks)

Swift.send(chunks)

Fig 3 上传时加密文件步骤的伪代码

create database keyencry

create table keytable(group\_type varchar(10),group\_token char(32),keytext char(32))

alter table keytable add primary key(group\_type,group\_token)

Fig4 数据库字段

## Deployment

我们在一个4节点的集群中测试了部署方案。4节点为客户端、代理服务端、存储服务端和数据库服务端。

|  |  |  |
| --- | --- | --- |
| IP | 节点类型 | 服务 |
| 192.168.0.1 | 存储 | account-server, container-server and object-server |
| 192.168.0.5 | 代理 | proxy-server |
| 192.168.0.7 | 数据库 | MYSQL |
| 192.168.0.9 | 客户 | Python-swiftclient |

Table 1: 集群的部署

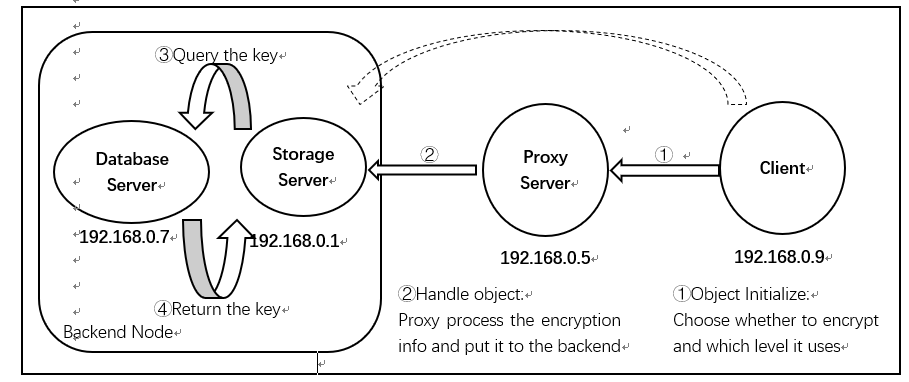
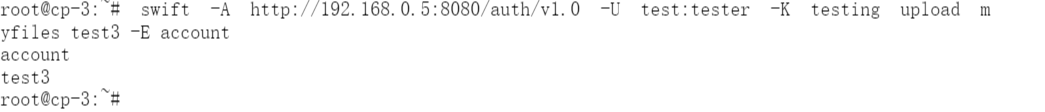


Fig 5 : Overview of the deployment architecture

在cloudlab上我们运行了集群测试，[部分测试结果截屏](Swift6.docx)。





## Challenges

**1.**[**上传和下载的对齐问题**](Swift4.docx)**：**

Swift在上传文件时是以块的形式写入（64K per chunk），而下载时则是全部读出然后以流形式传出，这将给对大型文件加解密造成一定的麻烦。AES是以明文分组形式进行加解密，因为swift上传下载传输方式不一致，可能会出现AES加密、解密分组不同的问题，如分组顺序、分组大小不一致。经讨论我们最后的解决方案是，对下载时读出的文件进行重新切割分块，块大小与上传时一致（64K），对块解密后重新连接，再流传输出去，经过检验这个方式解决了对齐问题。

**2.**[**md5的不一致问题**](Swift3.docx)

Swift原始方案中会对上传时的文件md5进行检查，这一步发生在所有块全部写入之后，但是由于我们的方案是在每个块写入之前对块进行加密，这样最后的md5将会是加密后文件的md5，必然与先前明文文件的md5不一致，会导致检查服务报错。解决方案是现在同时计算两个md5，明文的和密文的，在写文件完毕后，明文的Md5将会被交给检查服务检验，密文的md5将会被写入metadata保存。另外，为了防止存储中出错，也可以将两个md5都写入metadata以便日后核验。

## Further

当前方案是在两个假设前提下成立的，一是用户和云服务供应商是相互信任的，二是从客户端到存储服务端信道安全，在这两个条件存在时，方案安全效果最佳。但是我们也可以通过对方案进一步完善来解决实际中潜在的问题。

**1.客户端的加密方案**

现行方案中，客户端只起到选择加密级别的作用，在传输过程中文件可能被窃取，以及云服务商可能会查看用户文件造成隐私泄露。因此，我们可以在客户端就执行一次加密，这个加密的密钥存储在客户端本地，然后把加密的文件传输给服务端，这样传输过程中是密文形式，并且云服务商无法查看内容，但是这样的方案会和Deduplication产生一定的冲突。

**2.Deduplication**

有了数据库层的存在，对deduplication的解决可以更方便。比如识别到两个来自不同用户的相同文件，我们可以在数据库中对这个文件进行标识，使用用一个key加密文件，并且只存储一份，减少存储冗余。但是这个方案还是有很多漏洞，比如只使用一个key会带来密钥泄露风险，以及和1中所述的客户端加密方案冲突，因为不同用户传输的相同文件现在被不同密钥加密了，服务端无法识别他们是否为相同文件，这样传输中就必须携带原始的明文md5，可能会增加隐私泄露的风险。

## Conclusion

我们的加密方案为swift增加了一个数据库层，使云存储系统对加解密密钥管理更加灵活安全。对于客户端，他们可以自由的选择加密的等级。并且，这个方案给未来的deduplication以及客户端加密方案的解决提供了一定支持。