

硕士研究生综合实践

|  |  |
| --- | --- |
| 题目： | 基于HDFS客户端的 |
|  | ORAM的设计与实现 |

|  |  |
| --- | --- |
| 姓 名： | 康雨城 |
| 学 号： | 1701210869 |
| 院 系： | 软件与微电子学院 |
| 专 业： | 软件工程 |
| 研究方向： | 网络与系统安全 |
| 导师姓名： | 沈晴霓 |

二〇一八 年 八 月

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。

# 摘要

加密是一种常见的保护数据隐私的方法，但是单纯的使用加密并不能抵抗所有的攻击，攻击者可以根据观察用户对数据的访问模式来推断隐私信息。不经意随机访问机是一种重要的保护访问模式的手段，它通过混淆每次访问过程，使其与随机访问不可区分，从而保护真实访问中所访问数据块的位置、具体的访问方式、对相同数据块的访问频率、数据块的请求顺序。本文将不经意随机访问运用到分布式存储系统中，通过修改HDFS客户端源代码，修改读写的流程，实现对文件读写的混淆，从而保护数据的访问模式。

关键词：HDFS，不经意随机访问，读写混淆

# ABSTRACT

Encryption is a common method of protecting data privacy, but purely using encryption does not protect against all attacks. An attacker can infer private information based on observing the user's access patterns to the data. Oblivious Random Access Machine is an important means of protecting access patterns. It confuses each access process and makes it indistinguishable from random access, thus protecting the location of the access data block in real access, the specific access method, the frequency of access to the same data block, and the order in which data blocks are requested. In this paper, Oblivious Random Access Machine is applied to the distributed storage system. By modifying the source code of the HDFS client, modifying the process of reading and writing, the confusion of file reading and writing is realized, thereby protecting the data access mode.

KEY WORDS: HDFS, Oblivious Random Access Machine, Read and write confusion

# 目录

[摘要 1](#_Toc523527053)

[ABSTRACT 1](#_Toc523527054)

[目录 2](#_Toc523527055)

[第一章 绪论 4](#_Toc523527056)

[1.1 攻击场景 4](#_Toc523527057)

[1.2 项目背景 4](#_Toc523527058)

[1.3 项目意义 4](#_Toc523527059)

[1.4 文章结构 5](#_Toc523527060)

[第二章 HDFS读写流程分析 6](#_Toc523527061)

[2.1 HDFS客户端命令入口 6](#_Toc523527062)

[2.2 HDFS Command类 6](#_Toc523527063)

[2.3 HDFS PathData类 6](#_Toc523527064)

[2.3 HDFS读写文件类的父类 7](#_Toc523527065)

[2.3.1 getLocalDestination()方法 7](#_Toc523527066)

[2.3.2 getRemoteDestination()方法 8](#_Toc523527067)

[2.3.3 processPath()方法 8](#_Toc523527068)

[2.3.4 IOUtils类copyBytes()方法 8](#_Toc523527069)

[2.4 HDFS客户端写流程 8](#_Toc523527070)

[2.5 HDFS客户端读流程 9](#_Toc523527071)

[第三章 应用ORAM的HDFS方案设计 11](#_Toc523527072)

[3.1 以块为单位的ORAM-HDFS客户端读写混淆设计 11](#_Toc523527073)

[3.1.1 以块为单位的ORAM-HDFS客户端写混淆设计 11](#_Toc523527074)

[3.1.2 以块为单位的ORAM-HDFS客户端读混淆设计 11](#_Toc523527075)

[3.1.3 方案评价 12](#_Toc523527076)

[3.2 以文件为单位的ORAM-HDFS客户端读写混淆设计 12](#_Toc523527077)

[3.2.1 以文件为单位的ORAM-HDFS客户端写混淆设计 12](#_Toc523527078)

[3.2.2 以文件为单位的ORAM-HDFS客户端读混淆设计 12](#_Toc523527079)

[3.2.3 方案评价 13](#_Toc523527080)

[3.3 改进的以文件为单位的ORAM-HDFS客户端读写混淆设计 13](#_Toc523527081)

[3.3.1 改进的以文件为单位的ORAM-HDFS客户端写混淆设计 13](#_Toc523527082)

[3.3.2 改进的以文件为单位的ORAM-HDFS客户端读混淆设计 13](#_Toc523527083)

[3.3.3 方案评价 14](#_Toc523527084)

[第四章 应用ORAM的HDFS系统实现 16](#_Toc523527085)

[4.1 客户端ORAM方式读写Shell命令实现 16](#_Toc523527086)

[4.2 以块为单位的ORAM-HDFS客户端读写混淆实现 16](#_Toc523527087)

[4.3 以文件为单位的ORAM-HDFS客户端读写混淆实现 18](#_Toc523527088)

[4.3.1 以文件为单位的ORAM-HDFS客户端写混淆实现 18](#_Toc523527089)

[4.3.2 以文件为单位的ORAM-HDFS客户端读混淆设计 18](#_Toc523527090)

[4.4 改进的以文件为单位的ORAM-HDFS客户端读写混淆实现 19](#_Toc523527091)

[4.4.1 processArgument()方法修改 19](#_Toc523527092)

[4.4.2 getAllDirec ()方法和getAllFile()方法的添加 20](#_Toc523527093)

[4.4.3 通过最关键的processPath()方法修改实现混淆逻辑 22](#_Toc523527094)

[4.4.4 冗余块添加与切割方法修改 27](#_Toc523527095)

[第五章 实验结果与数据分析 30](#_Toc523527096)

[5.1 源码编译 30](#_Toc523527097)

[5.2 Hadoop部署 31](#_Toc523527098)

[5.3 实验结果 32](#_Toc523527099)

[5.3.1 以文件为单位的ORAM-HDFS客户端读写混淆实验结果 32](#_Toc523527100)

[5.3.2 改进的以文件为单位的ORAM-HDFS客户端读写混淆实验结果 33](#_Toc523527101)

[5.4 实验结果分析 35](#_Toc523527102)

[第六章 项目分工与完成情况 36](#_Toc523527103)

[6.1 项目分工 36](#_Toc523527104)

[6.2 完成情况 36](#_Toc523527105)

[6.3 不足之处 36](#_Toc523527106)

[第七章 总结与展望 37](#_Toc523527107)

# 第一章 绪论

1.1 攻击场景

在分布式文件系统中，存在着以下攻击场景。

1.Bob经过长期的观察，掌握了Alice对于不同节点文件的访问频率，就可以推测出这些节点上的文件对于Alice的重要程度，之后就可以根据重要程度去攻击那些对于Alice最为重要的节点。

2.Bob经过长期观察，发现Alice在有规律的访问几个节点之后，都会进行一次重要操作，那么，当用户在后期操作中再次访问该序列的节点时，就可推断出用户接下来极有可能要进行一次相同的重要操作，造成用户行为的泄露。

3.Bob了解到Alice刚刚获得某个自己很感兴趣的文件f，以及f的大小，即分块数m，于是Bob通过类似抓包的手段来监控Alice的网络传输，果然发现在不长的时间之后，Alice进行了一次上传操作，并且文件分块数正为m，那么Bob就可以知道f的存储位置，下一步就可以集中资源来攻击文件f所在节点。

1.2 项目背景

ORAM是在不可信的第三方存储数据时使用的一种密码学技术。它保证了数据存储在不可信的第三方服务器，但不会向第三方服务器泄露任何关于数据存储位置的信息。执行ORAM的实体与不可信的第三方服务器之问的交互，可以看作是客户与服务器交互模式。不可信的第三方看作是服务器端，客户向服务器端发送读写操作请求，服务器端响应客户端的操作请求，因此服务器知道存储数据的内容信息，由于服务器是不可信的，因此客户不能直接将数据存储在服务器端，为了解决这个问题，就是在客户端将数据加密后上传到服务器端，然而这并不能满足数据的机密性要求，尽管服务器不能知道加密数据的内容，但是服务器知道数据访问序列，而数据访问序列会泄露一些信息。我们将客户端的数据访问序列称为数据访问模式。ORAM能防止服务器通过分析数据访问模式获取相关信息。

1.3 项目意义

ORAM的目标是客户向服务器存储或检索数据时，不会向服务器泄露任何访问模式的信息，也就是说ORAM能向不可信的服务器隐藏它的访问模式。ORAM将虚拟数据访问模式转化为物理数据访问模式。ORAM技术通过在服务器端以特殊的数据结构存储数据，或者添加额外的数据访问请求，来实现这一目标。这个额外的数据访问请求对于客户来说通常是不必要的，但是对服务器来说，不能区分这些额外的数据访问请求和客户真正的数据访问请求，这就保证了服务器不能学习到有关客户数据访问模式的信息。将ORAM应用到分布式文件系统是本次研究的目标。HDFS是当前最流行的分布文件系统中之一，根据分布式文件系统HDFS的特点设计框架，本次研究通过修改HDFS客户端源代码，修改读写的流程，将ORAM思想应用到HDFS中，在加密的基础上进一步保护了数据的访问模式。

1.4 文章结构

本文首先提出分布式文件系统所面临的攻击场景、项目背景、项目目标，接下来对HDFS文件系统进行分析，其中对文件读写流程进行重点分析，然后对HDFS源代码进行修改，将ORAM在HDFS上进行实现，最后对结果进行测试与分析。

第二章 HDFS读写流程分析

2.1 HDFS客户端命令入口

HDFS提供了文件系统Shell命令来执行文件系统的常见操作，文件系统Shell命令包括了许多类似于传统Shell的命令，这些命令会与HDFS进行交互，执行类似于读取文件、移动文件、创建目录等操作。客户端可以通过命令的方式实现文件读写操作。

FsShell类是HDFS中用于执行文件系统的Shell命令的类，这个类的入口方法是main()方法，是一个典型的基于ToolRunner实现的应用。FsShell.main()中的ToolRunner.run()方法最终会调用FsShell.run()方法，FsShell.run()方法会调用CommandFactory.getInstance()从参数中解析出命令对应的command对象。然后在命令对象上调用run()方法执行对应的操作。

2.2 HDFS Command类

Command类是命令类的父类，Command类会解析命令选项，扩展命令的参数，然后依次处理每个参数。Command.run()方法调用的序列如下。

|  |
| --- |
| |-> {@link #processOptions(LinkedList)}  \* \-> {@link #processRawArguments(LinkedList)}  \* |-> {@link #expandArguments(LinkedList)}  \* | \-> {@link #expandArgument(String)}\*  \* \-> {@link #processArguments(LinkedList)}  \* |-> {@link #processArgument(PathData)}\*  \* | |-> {@link #processPathArgument(PathData)}  \* | \-> {@link #processPaths(PathData, PathData...)}  \* | \-> {@link #processPath(PathData)}\*  \* \-> {@link #processNonexistentPath(PathData)} |

其中processArgument()方法会实现对源文件的检测，检测其是否存在。

2.3 HDFS PathData类

在Command类中，我们可以看到一些方法中的参数是PathData类对象。PathData类的主要作用是通过文件的路径，对文件的信息进行封装。包括文件的URI、Path、文件所在文件系统、文件的状态、文件是否存在标识等。PathData类是实现文件读写拷贝的关键类之一。

2.3 HDFS读写文件类的父类

在HDFS中读文件命令和写文件命令是属于带有目的地的命令，因为文件的读写在有具体的位置。因此读文件命令和写文件命令都是CommandWithDestination类的子类。CommandWithDestination类继承了FsCommand，FsCommand继承了Command。关于以上类之间的关系如图2.1所示。



图2.1 HDFS读写文件类关系

在读操作和写操作中，CommandFactory.getInstance()从参数中解析出命令对应的命令对象为分别为Get对象和Put对象，在Put类和Get类中并没有完全覆盖Command类和CommandWithDestination中的方法，在读写过程中会有一些操作执行的方法是Put类和Get类的父类方法。所以Get和Put类有一些公共的方法可以共用。

2.3.1 getLocalDestination()方法

从getLocalDestination()方法的命名可以看出该方法的作用，即获取本地目的位置，当用户读文件时，文件的源位置位于Datanode，文件的目的地址在客户端本地，因此，该方法可以获取到想要存储文件的相关信息

2.3.2 getRemoteDestination()方法

从getRemoteDestination ()方法的命名可以看出该方法的作用，即获取远方目的位置，当客户端写文件时，文件的源位置位于本地，文件的目的地址在Datanode，因此，该方法可以获取到HDFS上想要写入文件的相关信息。

2.3.3 processPath()方法

processPath(PathData src, PathData dst)方法提供了两个参数，第一个参数存储了源文件相关信息，第二个参数存储了目标文件的相关信息。方法会调用 copyFileToTarget()方法，将源文件拷贝到目标文件。

2.3.4 IOUtils类copyBytes()方法

在IOUtils类中，最关键的方法是copyBytes静态方法，该方法实现将输入流读取的数据缓存到byte数组中，然后将数据写入输出流。copyFileToTarget()方法最终会调用该方法实现文件的拷贝。

2.4 HDFS客户端写流程

客户端要向Datanode中写文件时，需要经过以下几个步骤。

第一步，创建文件。HDFS客户端写一个文件时，会首先调用底层CilentProtocol接口定义的create()方法通知NameNode执行对应的操作，Namenode会首先在文件系统目录树中的指定路径下添加一个新的文件，然后将创建的新文件的操作记录到editlog中。完成ClientProtocol.create()调用后，会返回一个输出流对象。

第二步，建立数据流管道。获取了DFSOutputStream对象后，HDFS客户端就可以调用DFSOutputStream.write()方法来写数据了。由于CilentProtocol.create()方法只是在文件系统的目录书中创建了一个空文件，并没有申请任何数据块，所以DFSOutputStream会首先调用ClientProto.addBlock()向Namenode申请一个新的空数据块，addBlock()方法会返回一个LocatedBlock对象，这个对象保存了存储这个数据块的所有数据节点的位置信息，获取了数据流管道中的所有数据节点的信息后，DFSOutputStream就可以建立数据流管道写数据块了。

第三步，通过数据流管道写入数据：成功建立数据流管道后，HDFS客户端就可以向数据管道写数据了。写入DFSOutputStream中的数据会先被缓存在数据流中，之后这些数据会被切分成一个个数据包,通过数据流管道发送到所有数据节点。当客户端写满一个数据块之后，会调用addBlock()申请一个新的数据块，然后循环执行上述操作。

第四步，关闭输入流并提交文件。当客户端完成了整个文件中所有数据块的写操作之后，就可以调用close()方法关闭输出流，并调用CilentProtocol.complete()方法通知Namenode提交这个文件中所有的数据块，也就完成了整个文件的写入流程。

HDFS客户端写流程如图2.2所示

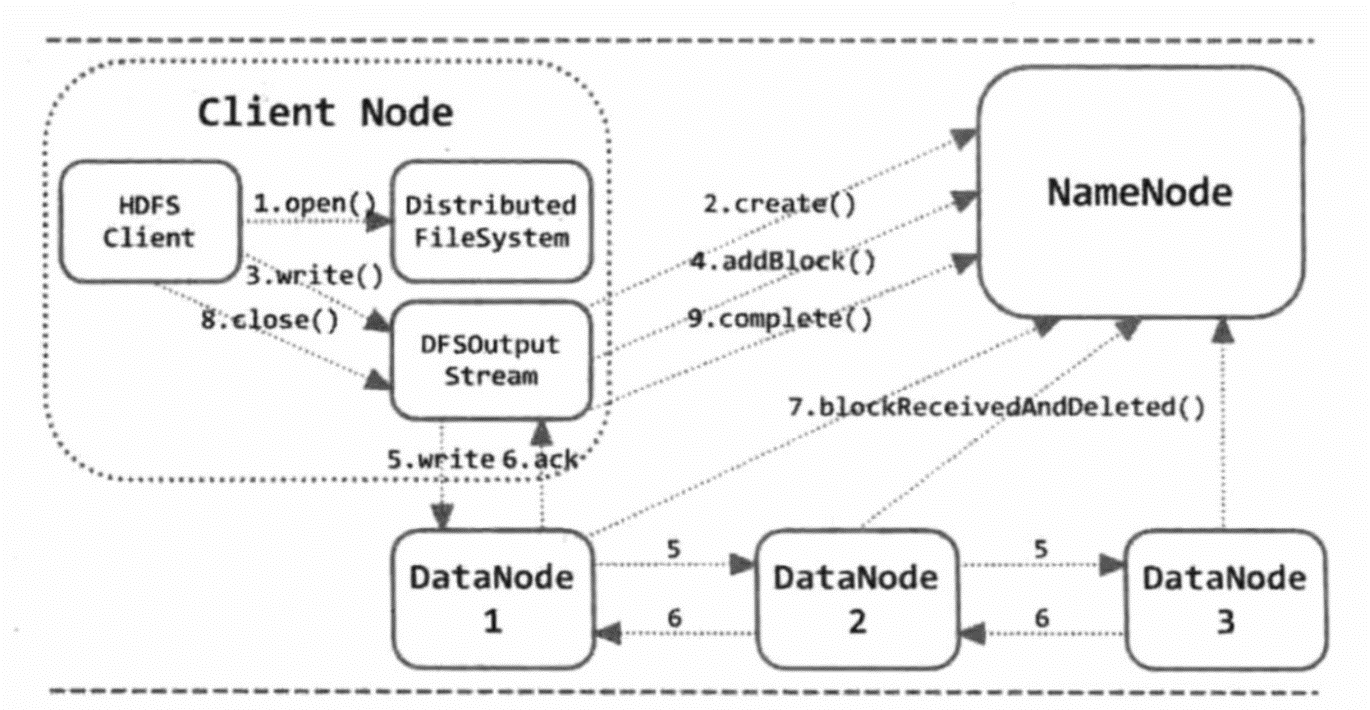


图2.2 HDFS客户端写流程

2.5 HDFS客户端读流程

HDFS客户端读取一个HDFS文件的流程可分为以下几个步骤。

第一步，打开HDFS文件：HDS客户端首先对用底层的ClientProtocol.open()方法，该方法会返回一个DFSInputStream对象。

第二步，从Namenode获取Datanode地址。在DFSInputSTream的构造方法中，会调用ClientProtocol.getBlockLocations()方法向名字节点获取该HDFS文件真实位置数据块的位置信息。Namenode返回的数据块的存储顺序是按照与客户端的距离远近排序的，所以DFSInputStream可以选择一个最优的Datanode节点，然后与这个节点建立数据连接读取数据块。

第三步，连接到Datanode读取数据块：HDFS客户端通过调用DFSInputStream.read()方法从这个最优的Datanode读取数据块，数据会以数据包为单位，从数据节点通过流式接口传送到客户端。当达到一个数据块的末尾时，DFSInputStream就会再次调用ClientProtocol.getBlockLocations()或失去下一个数据块的位置信息，并建立和这个新数据块的最优节点之间的连接，然HDFS客户的就可以继续读取数据块了。

第四步，关闭输入流：当客户端完成文件读取后，会通输入流的close()方法关闭输入流。

HDFS客户端读流程如图2.3所示

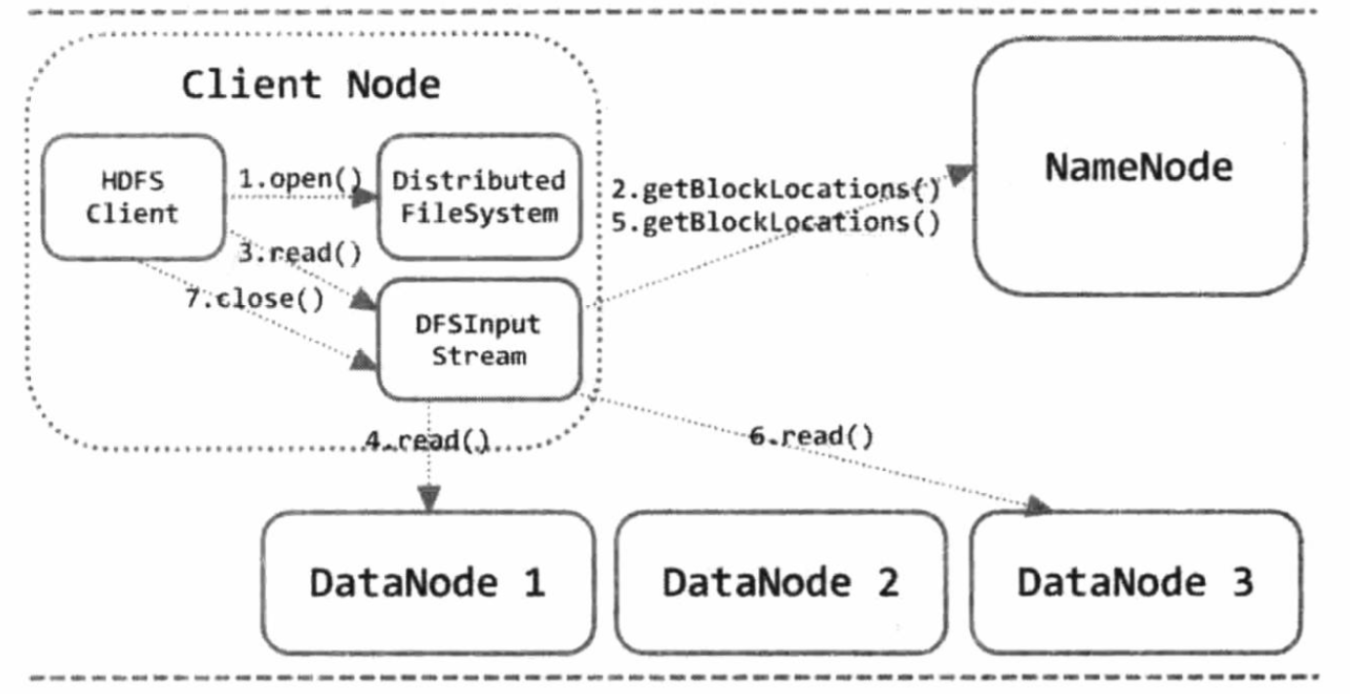


图2.3 HDFS客户端读流程

第三章 应用ORAM的HDFS方案设计

Partition ORAM是ORAM的一种类型。当客户请求IO操作时，Partition ORAM查找数据块对应的模块，将此IO操作转化为对应模块的ORAM算法，这样就向服务器暴露了该数据块在哪一个模块上，由于每个模块也是一个ORAM，所以服务器并不知道在模块内部发生了什么，但是当客户端对同一数据块请求多次访问操作时，如果这个数据块都在同一个模块中，那么就不能保证访问模式的茫然性。为了保证访问模式的茫然性，对数据块的每次IO操作之后，将数据块从原模块中删除，重新插入到一个随机模块。在客户端应用的ORAM都是Partition ORAM。

3.1 以块为单位的ORAM-HDFS客户端读写混淆设计

在HDFS中，文件以块为单位存储。早期HDFS块的默认单位是64MB，当前HDFS版本块的默认单位是128MB，这是默认设置可以在配置文件中改变。HDFS客户端写入一个新文件时，会按照将文件分成若干个块，按照一定的顺序写入HDFS，为了不让攻击者知道知道本次用户的操作是读操作还是写操作，需要在用户真正写文件之前进行一次假读取操作。HDFS读取一个文件时，会按照顺序从不同的Datanode上读取数据块，同样为了不让攻击者知道知道本次用户的操作是读操作还是写操作，需要在用户真正读文件之后进行一次写假块操作。

3.1.1 以块为单位的ORAM-HDFS客户端写混淆设计

在HDFS上初始化的时候，Client将一定数量的dummy block上传到每个Datanode中。一次写混淆的操作按照如下顺序进行。

1. 客户端向Namenode提交读请求。
2. Namenode查找内存中的元数据信息，将一个dummy block存储的Datanode地址以返回给客户端。
3. 客户端与Datanode建立连接，Datanode查询并找到此数据块。
4. Datanode将目标dummy block返回给客户端，Datanode删除此dummy block。
5. 客户端发送写请求给Namenode。
6. Namenode分配Datanode，并把信息返回给客户端。
7. Client访问相应的Datanode，将真实数据块写入Datanode。

3.1.2 以块为单位的ORAM-HDFS客户端读混淆设计

客户端进行一次读混淆操作按照如下顺序进行。

1. 客户端先查找本地的position map，目的是确定block是否存储在本地缓存，如果本地没有此数据块，客户端向Namenode提交读请求。
2. Namenode查找内存中的元数据信息，将Block存储的Datanode地址返回给客户端。
3. 客户端与Datanode建立连接，Datanode查询并找到此数据块。
4. 将目标block返回给客户端，删除Datanode上的此block。
5. 客户端发送写请求给Namenode
6. Namenode分配Datanode，并把这些信息返回给客户端
7. Client访问相应的Datanode，将数据块写入Datanode。

3.1.3 方案评价

该方案的优点在于能够较好地将ORAM应用到HDFS，且能够实现数据读写的混淆。

该方案的缺点在于，对于多客户端的情况下，不能保证文件的完整性。假如一个文件大小约600MB 共占五块，客户端的缓存区只有三块。当客户端读取文件之后，必然会有的块在缓存区里面，有的块在Datanode上，因此这个文件对于其他客户端来说，就变成了不完整的文件。此外，在HDFS源代码的实现上，按照以块为单位的操作难以实现。

3.2 以文件为单位的ORAM-HDFS客户端读写混淆设计

由于按照块的读写，会使多客户端的用户读取文件时发生错误，因此将混淆的单位提升为文件，即以文件为单位进行混淆。

3.2.1 以文件为单位的ORAM-HDFS客户端写混淆设计

文件写混淆操作按照以下顺序进行。

1. 客户端获取HDFS文件目录
2. 客户端匆匆获取到的目录中随机选择一个文件进行假读。
3. 读取文件后不进行保存。
4. 将用户真正要写入HDFS的文件按照原有的方式写入HDFS

3.2.2 以文件为单位的ORAM-HDFS客户端读混淆设计

文件读混淆操作按照以下顺序进行。

1. 将用户真正要读取的文件按照原有的读取方式读到客户端。
2. 在HDFS上将该文件删除。
3. 将该文件重新写回HDFS。

3.2.3 方案评价

该方案的优点是，由于HDFS提供了文件读写的接口，因此，操作文件的读写难度不大。对于多客户端来说，所有的文件都存储在Datanode上，不会存在文件的破坏与读文件错误。

该方案的缺点是，对于ORAM的实现并不完善，没有做到完全的混淆，文件访问顺序的特性依然会被攻击者寻找到规律。读混淆时，虽然客户端会先进行读，然后进行写，但是攻击者经观察依然会发现，读写两次数据传输的大小一样，这便可以使攻击者通过文件的大小来寻找规律，以便发起攻击。

3.3 改进的以文件为单位的ORAM-HDFS客户端读写混淆设计

3.3.1 改进的以文件为单位的ORAM-HDFS客户端写混淆设计

文件写混淆操作按照以下顺序进行。

1. 客户端获取HDFS文件目录。
2. 客户端从获取到的目录中随机选择一个文件进行假读。
3. 读取文件后不进行保存。
4. 在用户真正要写入HDFS的文件开头，添加随机数量的带有标识的数据块，并写入HDFS。
5. 文件写混淆操作流程如图所示。



图3.1 改进的以文件为单位的ORAM-HDFS客户端写混淆流程

3.3.2 改进的以文件为单位的ORAM-HDFS客户端读混淆设计

系统初始化时，客户端需要在本地建立dummy目录，用来存储dummy块文件。文件读混淆操作按照以下顺序进行。

1. 将用户真正要读取的文件，读到客户端，切掉冗余的混淆块，保存下来。
2. 复制该文件到dummy文件夹下面。
3. 在HDFS上将该文件删除。
4. 在dummy文件夹中随机选择一个文件。
5. 在文件的头部添加带有标识的数据块。
6. 将文件写回HDFS。

以文件为单位的读混淆流程如图3.2所示。



图3.2 改进的以文件为单位的ORAM-HDFS客户端读混淆流程

3.3.3 方案评价

该方案的优点是，由于HDFS提供了文件读写的接口，因此，操作文件的读写难度不大。对于多客户端来说，所有的文件都存储在Datanode上，不会存在文件的破坏与读文件错误。

该方案的缺点是，对于ORAM的实现并不完善，没有做到完全的混淆，文件访问顺序的特性依然会被攻击者寻找到规律。读混淆时，虽然客户端会先进行读，然后进行写，但是攻击者经观察依然会发现，读写两次数据传输的大小一样，这便可以使攻击者通过文件的大小来寻找规律，以便发起攻击。

第四章 应用ORAM的HDFS系统实现

4.1 客户端ORAM方式读写Shell命令实现

在HDFS文件系统的使用中，有的文件时较为重要的，需要通过ORAM的方式来防止攻击，而有的文件却不是十分重要的。通过ORAM的方式进行文件读写的混淆会增大系统的开销，因此在HDFS中，并不需要将所有的读写操作全部修改为ORAM操作，而是通过设置命令参数的办法，使ORAM读写操作变成用户的可选操作。

在CommandWithDestination类中设置标识，来判断用户需要进行的读写操作是否要以ORAM的方式进行。默认用户不进需要ORAM方式读写，其默认值设置为false，关键代码如下。

|  |
| --- |
| private boolean safeRead= false;  private boolean safeWrite = false;    protected void setSafeRead(boolean flag) {  safeRead = flag;  }  protected void setSafeWrite(boolean flag) {  safeWrite = flag;  } |

此外需在Get和Put类中添加处理参数的代码，在本项目中，设置ORAM方式读操作的参数为“-s”，ORAM方式写操作的参数为“-w”。当用户想要进行ORAM读操作时，需要在get命令后跟随“-s”参数，当用户想要进行ORAM写操作时，需要在put命令后跟随“-w”参数。

4.2 以块为单位的ORAM-HDFS客户端读写混淆实现

在以块为单位的读写混淆操作中，客户端文件读流程分为两部分，一部分是构建过程，一部分是传输过程。在构建过程中，客户端首先会根据本地目录和HDFS上的目录构建PathData对象，然后根据其所在的文件系统构建输入流和输出流。在传输过程中，通过copyFileToTarget方法和copyBytes方法传输数据，在底层会将数据缓存在Byte数组中，数组大小默认设置为4096。

在客户端上，HDFS的读文件的主要流程和方法如图4.1所示。



图4.1 HDFS的读文件的主要流程和方法

为了实现块的操作，客户端需要在底层进行块的切分，切分之后开始进行混淆操作，该过程如图4.2所示。

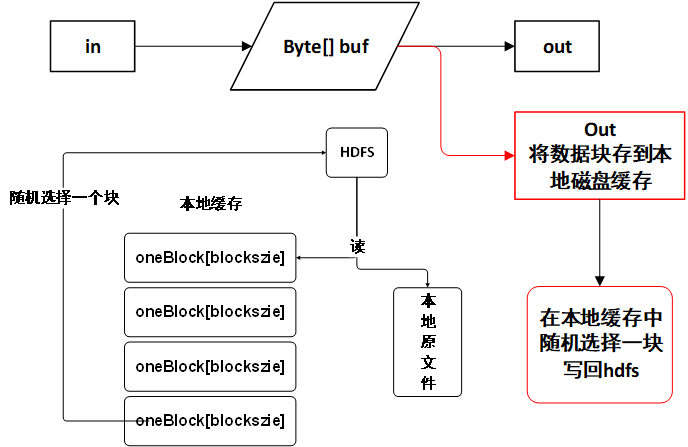


图4.2 块切分的混淆操作

在文件读入缓存数组的同时会在本地进行拷贝，拷贝之后再与本地已有的数据块进行一次混淆，随机选择一块写回HDFS。

由于块的混淆操作已经到达底层方法，编码过程中也并没有找到HDFS为客户端提供的以数据块为单位的读写方法，以块为单位的客户端读写混淆没有能够实现。

4.3 以文件为单位的ORAM-HDFS客户端读写混淆实现

4.3.1 以文件为单位的ORAM-HDFS客户端写混淆实现

除去原本客户端读操作需要构建的PathData对象和输入输出流之外，还需要构建额外的输入流和PathData对象，用来随机假读文件，需要注意的是用来假读的输入流并不需要与之对应的输出流，因为假读的数据，并不需要保存。以文件为单位的ORAM-HDFS客户端写混淆实现方法如图4.3所示。



图4.3 以文件为单位的ORAM-HDFS客户端写混淆实现方法

4.3.2 以文件为单位的ORAM-HDFS客户端读混淆设计

除去原本客户端读操作需要构建的PathData对象和输入输出流之外，还需要构建额外的输入输出流和PathData对象。用来将读取的文件写回HDFS。以文件为单位的ORAM-HDFS客户端读混淆设计流程与方法如图4.4所示



图4.4 以文件为单位的ORAM-HDFS客户端读混淆设计流程与方法

4.4 改进的以文件为单位的ORAM-HDFS客户端读写混淆实现

4.4.1 processArgument()方法修改

除去原本客户端读操作需要构

processArgument()方法在HDFS中的作用时检测文件是否存在，在改进的以文件为单位的客户端中，由于dummy文件夹与HDFS文件夹共同存储着文件，因此要保证相同的文件在HDFS和dummy文件夹下不能同时存在。例如文件A从HDFS中读到本地之后，要将其从HDFS中删掉，从本地随机选择dummy文件或其他文件写回，因此对于文件A来说，当客户端再次读取A时，A已经不存在于HDFS了，而是存在于本地Dummy文件夹，因此，当客户端读文件时在检查到文件不存在于HDFS后，需要同时检测文件是否存在于本地dummy文件夹，只有文件在两个文件夹都不存在的时候，才能断定文件不存在，这时要向客户端抛出异常，提示用户文件并不存在。修改后的processArgument()方法如下。

|  |
| --- |
| protected void processArgument(PathData item) throws IOException {  //add by kangyucheng this is to design for file exist in dummy;  if (item.exists){  // The file exist in HDFS  processPathArgument(item);  }  else{  String tem = item.toString();  String srcFilePath =DUMMY+tem;  PathData localFile =null;  try {  localFile = new PathData(new URI(srcFilePath), getConf());  if (localFile.exists){  localFile.existsInLocal=true;  }  } catch (URISyntaxException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  if (localFile.existsInLocal){  System.out.println("The file "+localFile+"exists in Local!!!!!!!");  processPathArgument(localFile);  }  else {  System.out.println("The file "+localFile+" do not exists ..... orz");  processNonexistentPath(item);  }  }  } |

为了完成以上修改需要在PathData类中添加变量existsInLocal来标志文件是否存在于本地。

4.4.2 getAllDirec ()方法和getAllFile()方法的添加

getAllDirec()方法的目的是获取HDFS中的所有文件，用于写混淆，该方法返回的是PathData数组，其代码如下。

|  |
| --- |
| /\*\*  \* get all filePath in HDFS  \* add by kangyucheng  \* @return PathData[]  \*/  public static PathData[] getAllDirec(Configuration con){  PathData safe\_src ;    PathData[] items,items2 = null;  try {  items = PathData.expandAsGlob(".", con);  switch (items.length) {  case 0:  throw new PathNotFoundException("Wrong write");  case 1:  safe\_src = items[0];  break;  default:  throw new PathNotFoundException("Wrong write1");  }    items2 = safe\_src.getDirectoryContents();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  return items2;  }  /\*\*  \* get all filePath Dummy filePath  \* add by kangyucheng  \* @return List<String>  \*/ |

getAllFile ()方法的目的是获取本地dummy文件夹中的所有文件，用于读混淆时随机选择文件，该方法返回的是存储了dummy文件夹中的所有文件路径的字符串List。其代码如下。

|  |
| --- |
| public static List<String> getAllFile(String directoryPath,boolean isAddDirectory) {  List<String> list = new ArrayList<String>();  File baseFile = new File(directoryPath);  if (baseFile.isFile() || !baseFile.exists()) {  return list;  }  File[] files = baseFile.listFiles();  for (File file : files) {  if (file.isDirectory()) {  if(isAddDirectory){  list.add(file.getAbsolutePath());  }  list.addAll(getAllFile(file.getAbsolutePath(),isAddDirectory));  } else {  list.add(file.getAbsolutePath());  }  }  return list;  } |

4.4.3 通过最关键的processPath()方法修改实现混淆逻辑

ProcessPath()方法中会对用户是否以ORAM的方式读写进行判断，如果safeRead标志值为true，就要进行安全读操作，如果safeWrite值为true，那么就要进行安全写操作。

在ORAM方式读操作中，我们要进行以下几个步骤的操作。第一步，检查文件是否存在于本地的标志值，如果文件存在于本地，那么我们要进行的操作就只有将本地的文件复制到目标路径即可。如果文件不存在于本地，那么该文件必然存在于HDFS。第二步，构建源文件PathData对象、目标文件PathData对象、dummy文件PathData对象。并完成文件的拷贝。第三步，FileSystem.delete()方法删除HDFS上源文件。第四步，从本地dummy文件夹中选择一个文件，构建其相应的PathData对象，执行相应的拷贝方法，将dummy文件写回HDFS。第五步，通过FileSystem.delete()方法将本地dummy文件删掉。

在ORAM方式写操作中，我们要进行以下几个步骤的操作。第一步，获取HDFS全部文件的路径，随机选择一个文件。第二步，将随机选择的文件读入输入流，不创建输出流，也不将文件写入本地。第三步，检测用户真正要上传的文件是否存在于本地dummy文件夹，如果文件存在于本地dummy文件夹，则不进行上传。如果本地dummy文件夹不存在该文件，那么就把该文件上传到HDFS。

processPath()方法的具体实现如下。

|  |
| --- |
| protected void processPath(PathData src, PathData dst) throws IOException {  if (src.stat.isSymlink()) {  // TODO: remove when FileContext is supported, this needs to either  // copy the symlink or deref the symlink  throw new PathOperationException(src.toString());  } else if (src.stat.isFile()) {  if (safeRead){    try {  //1. we should check if the file in local dummy filepath    String srcFilePath = "";  if (src.existsInLocal){  // the file in local,so the srcFilePath is just the src  // what we just need to do is copy it to usre's filePath  copyFileToTarget(src, dst);  }  else{  System.out.println("PLEASE WAIT ....");  //1.2 the file is in hdfs,so the srcFilePath  String filename = src.toString();  srcFilePath = DUMMY+filename;  //2.the source has been set, the destination are going to be set.  //One destination is dummy filePath, the other is user's filePath    PathData storageForSafeDst = new PathData(new URI(srcFilePath), getConf());  copyFileToTarget5forCutAddtionFile(src, dst);  copyFileToTarget5forCutAddtionFile(dst, storageForSafeDst);  //3.we should delete the file we have just read from hdfs  src.fs.delete(src.path,true);  //4.now we should choose a file in dummy filePath to write to hdfs  // the follow is to find dummy file to write to hdfs  // the src\_items is the list of dummy file  List<String> src\_items = getAllFile(DUMMY,false);  // k is a random integer  int k = (int) (Math.random()\* (src\_items.size()-1));  // choose a file at random  String srcpathString = src\_items.get(k);  System.out.println("The dummy file "+srcpathString+" is selected to write to HDFS");  // safe\_src is source safe\_dst is destination  PathData safe\_src = new PathData(new URI(srcpathString), getConf());  PathData safe\_dst ;  String[] dstpaths = srcpathString.split("/");  String dstpath =dstpaths[dstpaths.length-1];    PathData[] items = PathData.expandAsGlob(dstpath, getConf());  switch (items.length) {  case 0:  throw new PathNotFoundException("safe Wrong");  case 1:  safe\_dst = items[0];  break;  default:  throw new PathNotFoundException("safe Wrong");  }  // construct a integer at random to build dummy block  int num =(int)(Math.random()\*4);  // 5.write the dummy file we just chose  copyFileToTarget2ForAddBlocks(safe\_src,safe\_dst,num); // in this method the local file was deleted  // 6.delete the file from dummy filePath  safe\_src.fs.delete(safe\_src.path,false);  System.out.println("The file "+dstpath+" has been written to HDFS");  }    } catch (Exception e) {  // TODO Auto-generated catch block  System.out.println("WE DO NOT FINISH TO GET THE FILE --kangyucheng ");  //e.printStackTrace();  }    }  // this is for write  else if (safeWrite){  try {  System.out.println("PLEASE WAIT ....");  PathData safe\_src ;  //1. we will get all the filePath in HDFS  PathData[] items2 = getAllDirec(getConf());  //2.we chose one at random  int k = (int) ((items2.length-1)\*Math.random());  safe\_src=items2[k];  System.out.println("The file :"+safe\_src+" in HDFS was chose to read");  //3.we read it but we do not real write it  copyFileToTarget3readExceptWrite(safe\_src,dst);  // 4 .check if the file in dummy filePath  String tem = dst.toString();  String srcFilePath =DUMMY+tem;  PathData localFile =null;  try {  localFile = new PathData(new URI(srcFilePath), getConf());  if (localFile.exists){  localFile.existsInLocal=true;  }  //check local end  } catch (URISyntaxException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  if (localFile.existsInLocal){  System.out.println("The File has alread exist in Local,you sholud not put it!!!");  }  else{  int num =(int)(Math.random()\*4);  copyFileToTarget2ForAddBlocks(src,dst,num);  }  } catch (Exception e) {  // TODO Auto-generated catch block  System.out.println("WE DO NOT FINISH TO GET THE FILE --kangyucheng ");  e.printStackTrace();  }  }  else{  System.out.println("UNSAFE COMMOND --kangyucheng");  try {  copyFileToTarget5forCutAddtionFile(src, dst);  //check local end  } catch (Exception e) {  // TODO Auto-generated catch block  System.out.println("THERE IS SOMETHING WRONG ! --kangyucheng");  e.printStackTrace();  }    }  //add by kangyucheng end  } else if (src.stat.isDirectory() && !isRecursive()) {  throw new PathIsDirectoryException(src.toString());  }  } |

4.4.4 冗余块添加与切割方法修改

冗余块添加方法是指，客户端要向HDFS写入数据时，需要在文件的开始添加随机数量的块，用于文件大小的混淆，每块的大小为1024\*1024\*128byte，在每次缓存bufferSize大小的byte数组的起始添加26个字母由a到z，，用于读取文件时标识冗余块。冗余块添加方法代码修改如下。

|  |
| --- |
| public static void copyBytes2(InputStream in, int buffSize, OutputStream safe\_out,int num)  throws IOException {  PrintStream ps = safe\_out instanceof PrintStream ? (PrintStream)safe\_out : null;  byte buf[] = new byte[buffSize];  int i =0;  while (i<num\*1024\*1024\*128){  for (int t =0 ; t< 26 ;t++){  buf[t] = (byte) (t+'a');  }  for (int j=26;j<buffSize;j++){  buf[j] = (byte)(Math.random()\*127);  }  safe\_out.write(buf, 0, buffSize);  i+=buffSize;  }  int bytesRead = in.read(buf);  while (bytesRead >= 0) {  safe\_out.write(buf, 0, bytesRead);  if ((ps != null) && ps.checkError()) {  throw new IOException("Unable to write to output stream.");  }  bytesRead = in.read(buf);  }  } |

冗余块删除切割方法是指，客户端读文件时，需要将文件前面的dummy块去掉，读文件时，长度为buffSize的byte数据会缓存在byte数组中，对这个数组进行检测，如歌这个数组前26位是a-z的26个字母，则认为这个块是dummy块，这个块需要删除不需要写入文件，否则需要将其写入本地文件，完成正常的读写操作。冗余块删除切割方法修改代码如下。

|  |
| --- |
| public static void copyBytes5(InputStream in, OutputStream out, int buffSize) throws IOException {  PrintStream ps = out instanceof PrintStream ? (PrintStream)out : null;  byte buf[] = new byte[buffSize];  int bytesRead = in.read(buf);  while (bytesRead >= 0) {  boolean b = false;  for(int i = 0; i< 26 ;i++){  if (buf[i]!= (byte)(i+'a')){  b =true;  }  }  if (b){  out.write(buf, 0, bytesRead);  }  if ((ps != null) && ps.checkError()) {  throw new IOException("Unable to write to output stream.");  }  bytesRead = in.read(buf);  }  } |

# 第五章 实验结果与数据分析

5.1 源码编译

对于整个Hadoop项目可以一切从它的源代码开始，它的一切也是由源代码生成，包括我们部署的Hadoop项目也时由源码编译而来。我们在方案的实现中修改了源代码，因此必须要编译生成binary文件，从而能够部署Hadoop。

为了提高系统的性能，以及系统的管理，采用搭建两个虚拟机的方式，其中一个用来编译，另外一个用来部署。

在编译的虚拟机中，需要搭建编译环境，在Hadoop源代码根目录下有一个bulid.txt文件，该文件对编译需要的环境进行了说明。

对其关键命令的执行结果如图5.1所示。

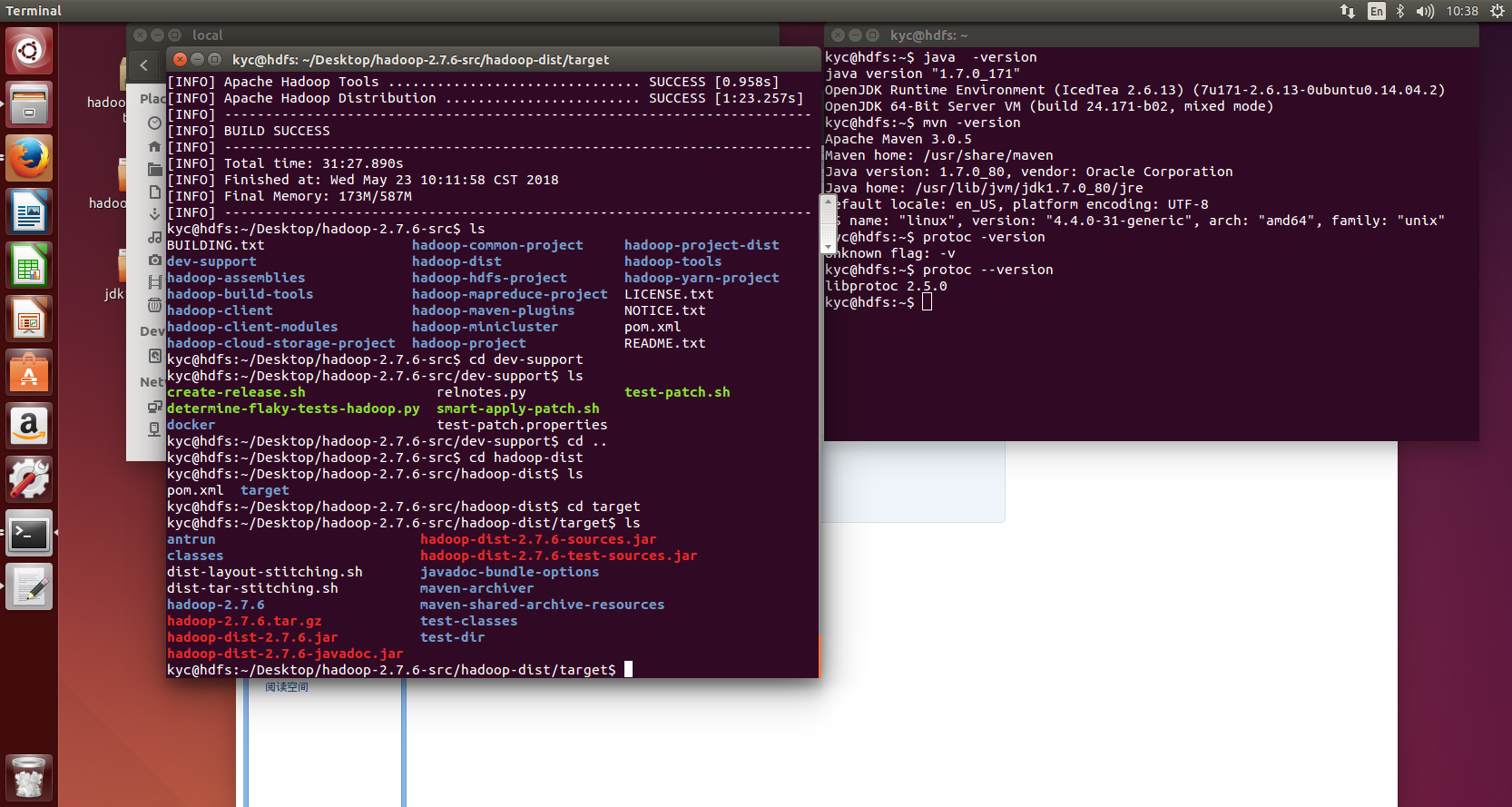


图5.1 Hadoop编译环境部署

采用mvn工具编译Hadoop源代码，编译成功后，系统会提示“BUILD SUCCESS”, 编译结果如图5.2所示。

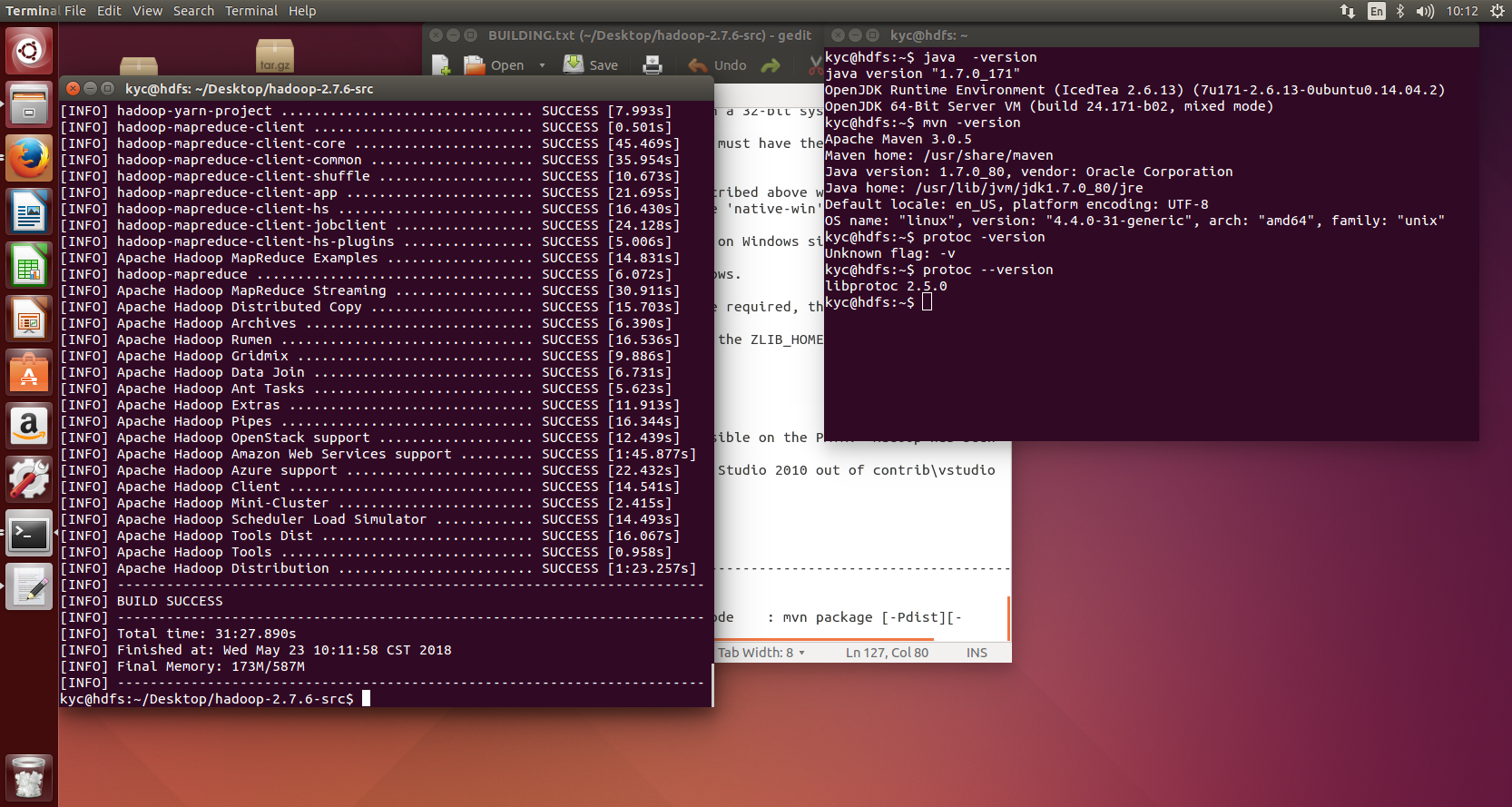


图5.2 Hadoop源代码编译成功

在hadoop-dist目录下会有编译成功的binary文件生成。这个文件就是我们需要在另一台虚拟机上部署的文件。binary文件位置如图5.3所示

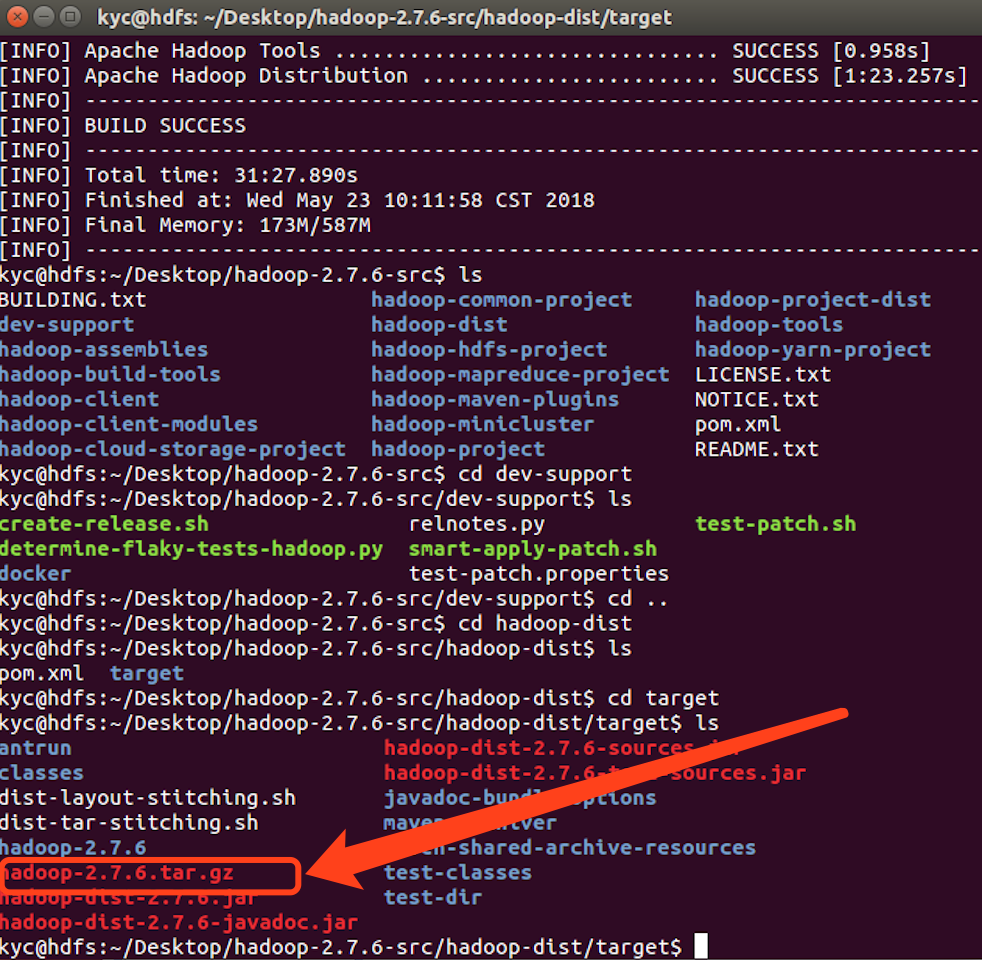


图5.3 binary文件位置

5.2 Hadoop部署

部署需要修改Hadoop配置文件，包括bashrc、core-site.xml、hadoop-env.sh、hdfs-site.xml等。可用jps命令查看Datanode与Namenode是否全部启动。改进的以文件为单位的客户端，需要在部署成功后，创建dummy文件夹，以及在HDFS和dummy文件夹下添加随机文件，初始化系统。初始化系统结果如图5.4所示。

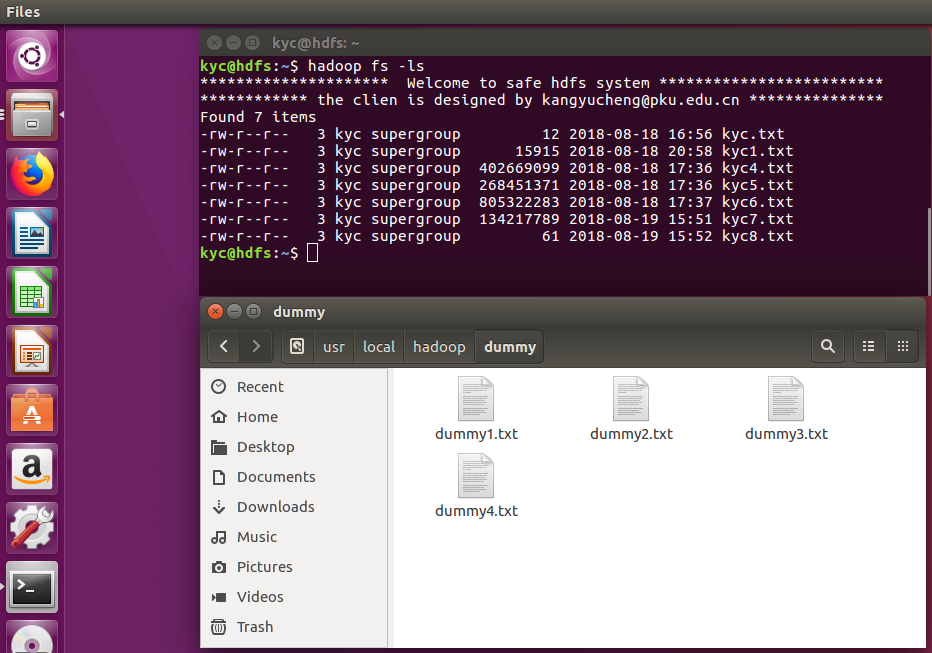


图5.4 binary初始化系统结果

可以发现在HDFS和dummy文件夹下都添加了一定数量的文件。

5.3 实验结果

5.3.1 以文件为单位的ORAM-HDFS客户端读写混淆实验结果

在以文件为单位的ORAM-HDFS客户端中，使用get命令读取文件，在get命令后面跟随“-s”参数，采用ORAM的方式进行读取文件，在控制台，将块的信息输出。实现结果如图5.5所示。

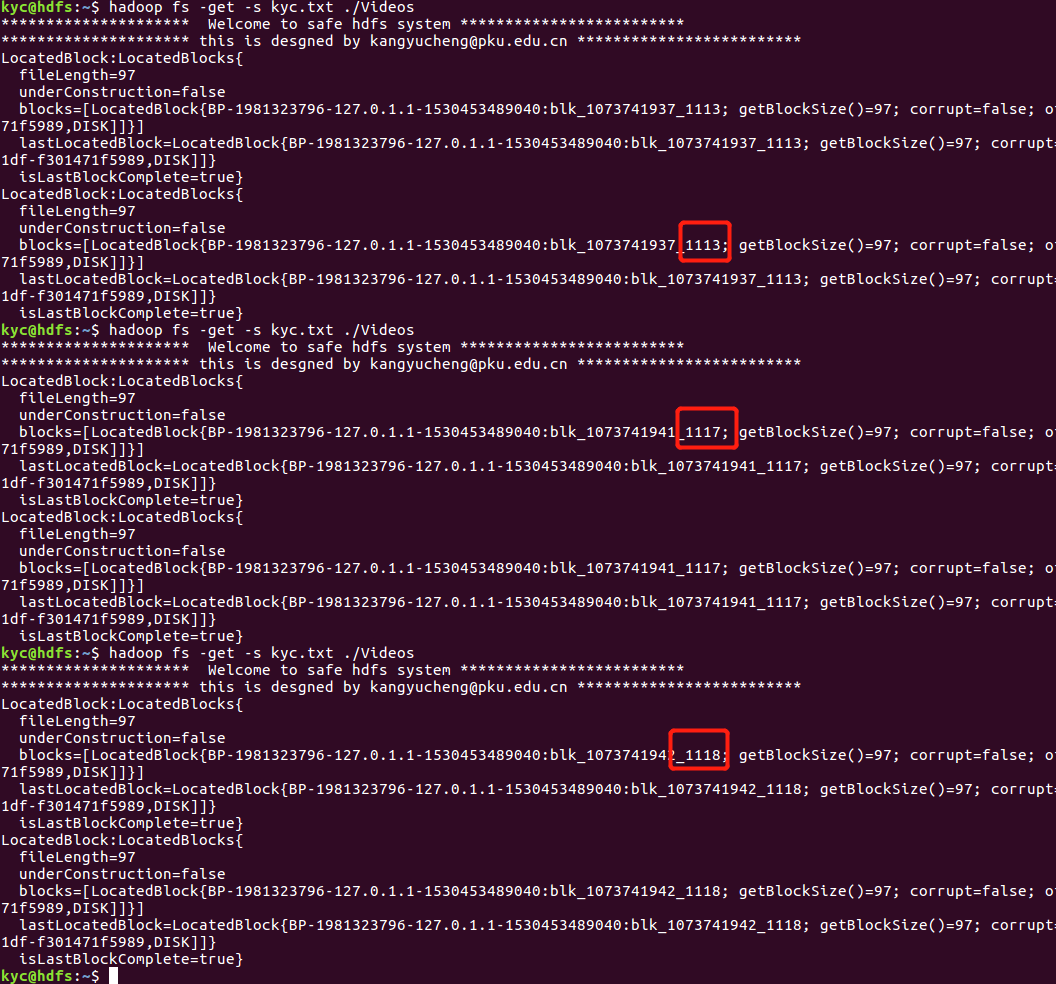


图5.5 以文件为单位的客户端读文件结果

通过输出信息可以明显发现，虽然每次读取的是相同文件，但是块的位置信息发生了改变。这个改变使得文件每次读取过后，都会改变位置。因此即使客户端频繁的访问同一个文件也不会被攻击者检测到。

在以文件为单位的ORAM-HDFS客户端中，使用put命令读取文件，并在put命令后跟随参数“-w”，表示以ORAM的方式写文件，在客户端写入一个文件时，会首先从HDFS中随机选择一个文件读取。实验结果如图5.6所示。

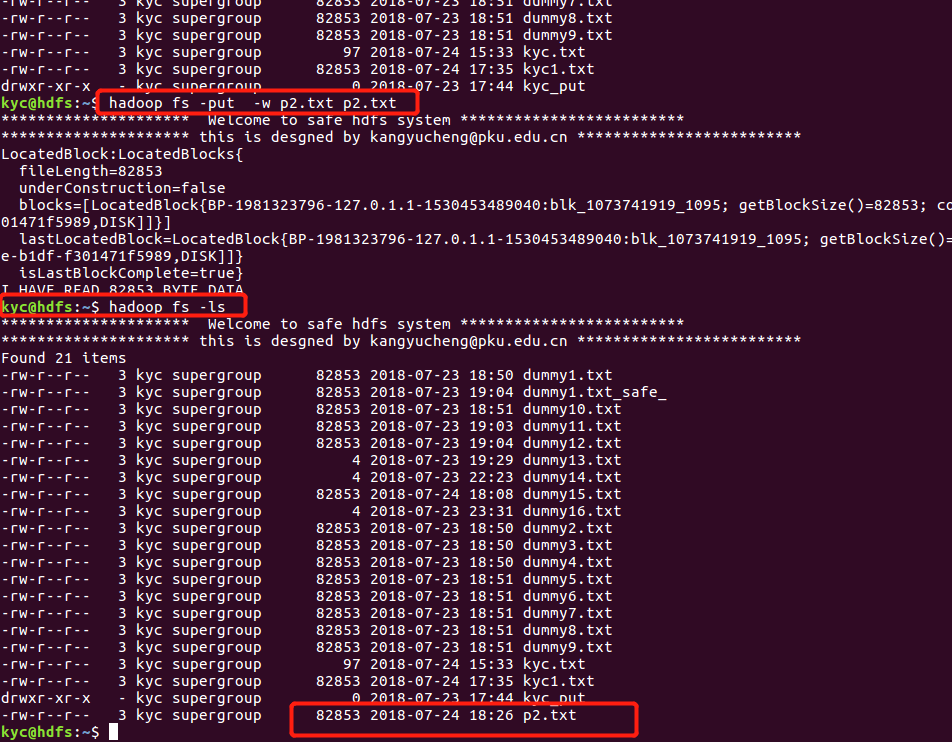


图5.6 以文件为单位的ORAM-HDFS客户端写文件结果

通过实验结果可以发现，在客户端对读取到的byte进行计数。并在读取完成时将读取到的byte数目输出。用户真正要写的文件也成功写入HDFS。

5.3.2 改进的以文件为单位的ORAM-HDFS客户端读写混淆实验结果

与以文件为单位的客户端不同的地方是，无论是否是ORAM方式的读写文件，均需要考虑冗余块的问题，因为在客户端，正常的写操作也会添加冗余块，相应读操作也会切割冗余块。

在以文件为单位的客户端中，使用put命令读取文件，并在put命令后跟随参数“-w”，表示以ORAM的方式写文件，在客户端写入一个文件时，会首先从HDFS中随机选择一个文件读取。在客户端对读取到的byte进行计数，并在读取完成时将读取到的byte数目输出到控制台。然后将用户要写入的文件写入HDFS，在写入的过程中，先把冗余的随机块写入文件，然后再把真正的文件内容写入文件。实验结果如图5.7所示。

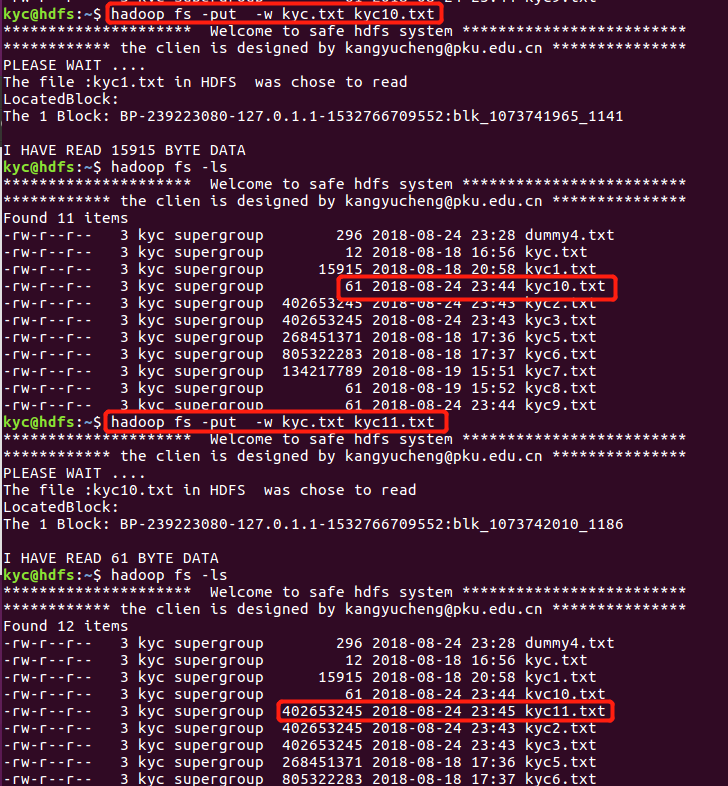


图5.7 改进的以文件为单位的客户端写文件结果

可以看到再两次客户端上传相同文件时，文件的内容一模一样，当时最终再HDFS上的显示却是不一样的，后者的大小发生改变，说明我们的冗余块添加成功。

在改进的以文件为单位的客户端中，使用get命令读取文件，在get命令后面跟随“-s”参数，采用ORAM的方式进行读取文件，在控制台，将块的信息输出。并将本地被选中的dummy文件、以及其被写入HDFS的形式显示出来。 实验结果如图5.8所示。

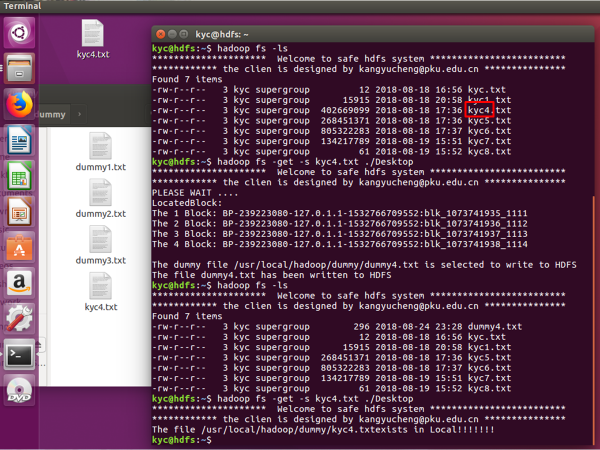


图5.8 改进的以文件为单位的客户端读文件结果

实现结果表明，当客户端读取的文件之后，文件的位置已经发生改变，原本的文件已经不再HDFS上，取而代之的是一个dummy文件，当客户端再次读取相同的文件时，该文件已经在本地的dummy文件夹下，这样降低了HDFS的访问频率。

由于对文件的内容进行了增添与切割，我们需要进一步验证读取到文件是否发生了变化。查看kyc4.txt的详细信息如图5.所示。

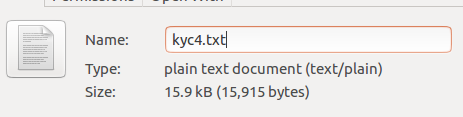


图5.9 kyc4.txt详细信息

查看结果表明，kyc4.txt并没有发生改变。文件的读写功能在混淆中保持完善。

5.4 实验结果分析

实验结果表明，ORAM-HDFS项目成功在分布式文件系统上应用了ORAM。

# 第六章 项目分工与完成情况

6.1 项目分工

秦嘉学长负责名字节点（Namenode）部分编写。

刘忠开负责数据节点（Datanode）部分编写。

康雨城负责客户端（Client）部分编写。

6.2 完成情况

项目均按照计划完成，并在Github上进行了开源，每当我们有新的想法与会进一步丰富和完善。地址为：<https://github.com/oram-hdfs/oramhdfs>。

6.3 不足之处

虽然项目均按照计划完成，但项目本身仍有一些不足之处。

第一，在改进的以文件为单位的ORAM-HDFS客户端中，对于多客户端的情况会存在个别文件无法读到的问题，即有的文件被用户读到本地后，该文件将会从HDFS中删除，导致其他用户无法读到，这是ORAM所带来的问题。但由于文件数量较少，因此不会造成太大影响。

第二，在以文件为单位的ORAM-HDFS客户端中，文件写回HDFS的随机性依靠的是HDFS负载均衡的策略，当读取一个文件回本地后将其写回时，其具体写回的节点存在写回原来节点的可能性，从而无法实现ORAM。

# 第七章 总结与展望

本项目是在沈晴霓老师的悉心指导下完成的。沈晴霓老师作为一名优秀的、经验丰富的教师，具有丰富的知识和经验，在整个工程设计和实现过程中，对我们进行了耐心的指导和帮助，提出严格要求，引导我们不断开阔思路，为我答疑解惑，鼓励我大胆创新，使我们在这一段宝贵的时光中，既增长了知识、开阔了视野、锻炼了心态，又培养了良好的实验习惯和科研精神。在此，向我们的指导老师表示最诚挚的谢意！