POLITECHNICA UNIVERSITY OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Data Anonymity in BlobSeer

Thesis supervisors:
Prof. Dr. Ing. Cristea Valentin
As. Dr. Ing. Leordeanu Cătălin

Author:
Bogdănescu I. Andrei

BUCHAREST

2012

Table of Contents

Abstract

The problem of data anonymity is one of the most important aspects of nowadays' data management systems. The main concern of users that want to store data over a cloud or over any distributed system from the internet is that their privacy needs to be guaranteed as well as their data's security. This thesis addresses the problem of data anonymity in BlobSeer, a large-scale distributed storage system. The easiest way to hide data from other users is by encrypting the data we want to store. We could additionally split the data and make from it a padding inside the BLOB we are using. The step of splitting the data and scattering the pieces inside a blob is very useful because an attacker will not be able to find where the data is written as its location is encrypted as well as the data itself. In this way the data we want to store via a BLOB will be hard to be tracked down and decrypted by other users.

Keywords

Blob, encryption, padding, data anonymity, distributed systems

1. Introduction

1.1 Context

The use of cloud computing is rapidly growing especially because people prefer to borrow hardware to deploy their software applications instead of buying it. One of the main reasons to use the cloud is because it is cheaper than buying the hardware that is necessary for the use of software. Also, it is very important nowadays to be able to use any application you need from any computer that only has an internet connection. In this context, the cloud computing domain has suffered a large number of transformations and it, now, represents one of the most important branches from the world of computers.

These being said, a team from INRIA, France, developed a system called BlobSeer, a service made for large-scale data storage of distributed applications, a project in which the Politechnica University of Bucharest is also involved with a team.

BlobSeer works with a large number of modules and services and it reached a point of development when the anonymity of the clients and the privacy of data is a must.

1.2 Motivation

In this thesis it will be discussed about how the concept of data anonymity is a big step towards the assurance of the privacy of the users of BlobSeer.

The data anonymity implementation addressed in this work is made for the BlobSeer clients that want to keep their data safe from being tracked down by any other users. The privacy of data is a concept researched intensively in the last four years of cloud computing. We implemented it as a client for the BlobSeer system, increasing its mobility and making it very is to be imported on other binary large objects management systems.

1.3 Structure

This thesis is composed of seven chapters in which we talk about some of the most known data management systems from which we started our investigation, about data anonymity, about BlobSeer system in general and then about the anonymity in BlobSeer and the solution that we came along, ending with some experimental results and the conclusions reached during the implementation.

In the first chapter we describe some of the most important and used data management systems in cloud computing. Firstly, we will discuss about Hadoop Distributed File System (HDFS), continuing with Google FS and ending with Amazon S3 service.

In the second chapter the main subject represents a description about the data anonymity concepts. First of all, we will point out the general characteristics of the both ideas and then some of the researches already made about this subject.

The third chapter is dedicated to BlobSeer, the system on which we implement our solution. The discussion will be about the BlobSeer's architecture, the concept of BlobSeer client and also about the communication protocol.

The fourth one is used for a description of data anonymity in BlobSeer. In this chapter we will develop the issue of data anonymity specifically for the BlobSeer's system and how this problem is resolved with our solution.

The fifth chapter contains the implementation details provided along with our solution: the client's architecture, the classes used, input and output format and how the module interacts with other components used.

The sixth chapter is dedicated to the experimental results: comparison of runs with and without our solution, performance analysis along with examples of normal runs of the client implementation. The last chapter contains the conclusions we reached at after the complete analysis and implementation of our solution.

2. Data Management Systems

2.1 Hadoop Distributed File System

Hadoop is an Apache project and all its components are available under an open source license. One of the main systems provided by Hadoop is a distributed file system along with a framework used for the management of very large data sets. This file system is called Hadoop Distributed File System (HDFS) and it is specifically designed to run on commodity hardware. Generally speaking, HDFS has many similar aspects with other distributed file systems but it highlights itself with some significant differences. First of all, the main characteristic is that HDFS is designed to be deployed on low-cost hardware; this leads to the fact that it is also very highly fault-tolerant. Secondly, HDFS is known to have an advantage regarding the applications with large data sets and at the same time, enables streaming access to the data.

In an HDFS instance, the hardware failure is seen as an usual thing to happen. This is caused by the fact that it may consist of thousands of servers which automatically leads to the fact that there are always some components of the file system that are not functioning.

HDFS is designed for applications that need streaming access to the data sets. This means that it is not a general purpose file system as most of the other distributed file systems are, and the accent is put on the high throughput instead of low latency data access. Thus, HDFS scales to hundreds of nodes in a single cluster so the bandwidth should aggregate at a high rate. This is needed so the data access could be made without any problems as HDFS needs to support files with a size of terabytes order.

One of the side effects of the fact that HDFS work with very large data sets is that once a very large file is written, it is closed at the same time as it should not be changed afterwards. In this way there is no need for algorithms that resolve the data coherency issues and so the appending-writes to the same file are not supported.

The second side effect is the fact that it is necessary to have the computation made as close to the data as possible. This will minimize the network transfers and so it increases the overall throughput of the system.

HDFS has an architecture of master-slave type. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files.

Files and directories are represented on the NameNode by *inodes*, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks and each block of the file is independently replicated at multiple DataNodes. The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes.

The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

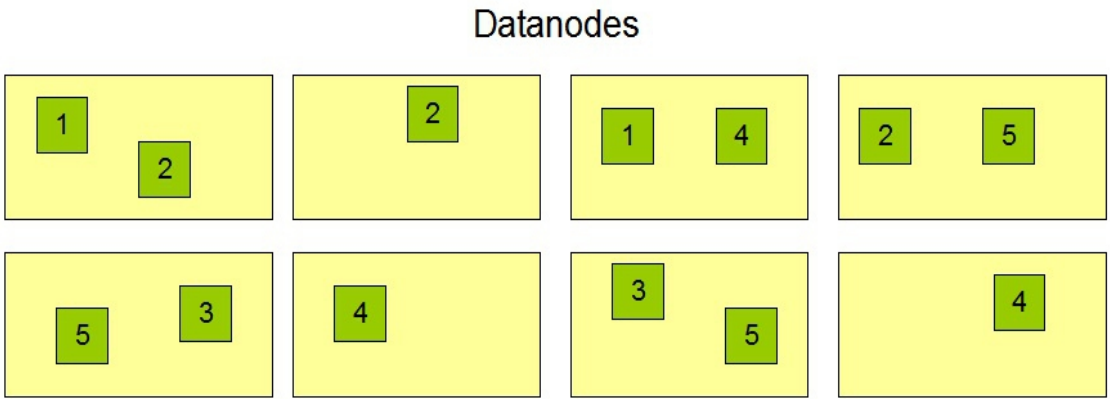
The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the

Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software.

In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. A replica stored on a DataNode may become corrupted because of faults in memory, disk, or network.

The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test any policy that's optimal for their applications.

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under or over-replicated when a block report from a DataNode arrives.



When a block becomes over replicated, the NameNode chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk space. The goal is to balance storage utilization across DataNodes without reducing the block's availability.

When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas.

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the ClientProtocol with the NameNode. The DataNodes talk to the NameNode using the DataNode Protocol. A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNode Protocol. By design, the NameNode never initiates any RPCs. Instead, it only responds to RPC requests issued by DataNodes or clients.

2.2 Google FS

As the Google's internet search engine became the market-leading one, the demands for data processing needs grew exponentially. As this was becoming a problem because of the lack of hardware and software meant to process large amounts of data, a team from Google implemented a

distributed file system to meet these requirements, named Google File System (GFS). GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability.

As these goals need to be kept while at the same time the system should be able to process large data sets, some differences appeared in functionality as well as in the architecture of the Google File System.

As well as in the Hadoop Distributed File System, the component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. There have been seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Another similarity to the HDFS is that the usual files stored are very large. Each file typically contains many application objects such as web documents. When the applications are regularly working with fast growing data sets of many TBs comprising billions of objects, it is almost impossible to manage billions of approximately KB-sized files. Also, most files are managed by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics: they may constitute large repositories that data analysis programs scan through, they may be data streams continuously generated by running applications, archival data or intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

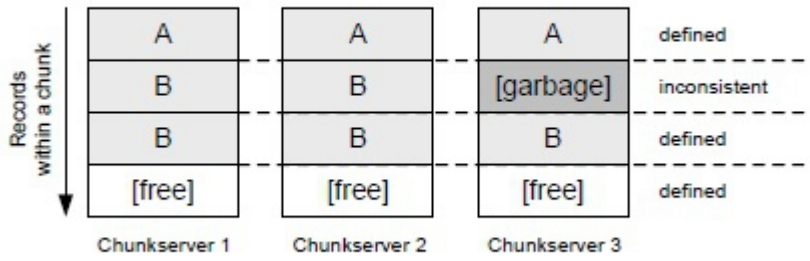
As indicated before, GFS is a distributed system to be run on clusters. Similar to HDFS, the architecture relies on a master/slave pattern. Whereas the master is primarily in charge of managing and monitoring the cluster, all data is stored on the slave machines, which are referred to as “chunkservers”. In order to provide sufficient data safety, all data is replicated to a number of “chunkservers”. While the exact replication algorithms are not fully documented, the system additionally attempts to use machines located in different racks or even on different networks for storing the same piece of data. This way, the risk of losing data in the event of a failure of an entire rack or even network is eliminated.

To be able to make it as distributive as possible, GFS came with a new concept which was used afterwards in the HDFS implementation as well, but under a different name. The main idea that helped the Google team to successfully implement this file system and at the same time keep the scalability and performance of an usual distributed file system, is the one that files are divided into chunks, each having a fixed size of 64 MB. A file always consists of at least one chunk, although chunks are allowed to contain less data than 64 MB. New chunks are automatically allocated in the case of the file growing. Chunks, not only that they are the unit of management and their role can thus be roughly compared to blocks in ordinary file systems, they also are the unit of distribution. Although client code deals with files, they are actually an abstraction provided by GFS in the way that a file refers to a sequence of chunks. This abstraction is primarily supported by the master, which manages the mapping between files and chunks as part of its metadata. Thus, *chunkservers* exclusively deal with chunks, which are identified by unique numbers. Based on this separation between files and chunks, GFS has the flexibility of implementing file replication solely on the basis of replicating chunks.

As said before, the *master server* holds the metadata and manages the file distribution. This means that his instance is needed whenever chunks are to be read, modified or deleted. Also, the metadata managed by the master has to contain information about each individual chunk. The size of a chunk (and thus the total number of chunks) is the main characteristic that influences the amount of data and interactions the master has to handle. Choosing 64 MB as chunk size can be considered a tradeoff between trying to limit resource usage and master interactions on the one hand and accepting an increased degree of internal fragmentation on the other hand.

To verify the integrity of the data, the chunks have a checksum which is calculated every time when a read operation needs to be made. This practice is used in many file systems and it is needed to be sure that there won't be any disk corruptions during the operation made. To limit the amount of data required to be re-read in order to recalculate a checksum, if parts of a chunk have been modified during a write operation, these checksums are not created on chunk-granularity but on block-granularity. Blocks, which do not correspond to file system blocks, are 64 KB in size and are a logical subcomponent of a chunk.

It could happen to have an append operation that failed. This would most likely lead to a successful retry but this would also have an impact on the consistency of the data stored. In fact, if a record append operation succeeds on all but one replica and is then successfully retried, the chunks on all servers where the operation has succeeded initially will now contain a duplicate record. Similarly, the chunk on the server that initially was unable to perform the modification now contains one record that has to be considered garbage, followed by a successfully written new record. In order to not influence, because of this scenario, the output generated by any application, Google FS created a consistency model, named Relaxed Consistency Model, which is based on three states of consistency: *consistent*, *defined* and *inconsistent*.



The relaxed nature of the consistency model used by GFS and the requirement that client code has to cooperate emphasizes the fact that GFS is indeed a highly specialized file system and is not intended as immediately applicable for general use outside Google.

2.3 Amazon S3

As data intensive user applications started producing large amounts of data, Amazon.com came with a revolutionary idea to add a new component to its Amazon Web Services. This new service they would provide is one that would store data in exchange of money, being named as Simple Storage Service (Amazon S3).

Amazon S3 is supported by a large number of computer systems distributed across multiple data centers in the United States and Europe and is expected to offer low data access latency, infinite data durability, and 99.99% availability. Since its launch, S3 has acquired a large user

base ranging from home users and small businesses to large business enterprises. Currently, S3 stores more than a trillion of user objects and handles over 900 million user requests a day.

Data stored in S3 is organized over a two-level namespace. At the top level are buckets (similar to folders or containers) which have a unique global name and serve several purposes: they allow users to organize their data, they identify the user to be charged for storage and data transfers, and they serve as the unit of aggregation for audit reports. Each Amazon Web Services (AWS) account may have up to one hundred of S3 buckets.

Each bucket can store an unlimited number of data objects. Each object has a name, a blob of data (of up to 5GB), and metadata consisting of a small set of predefined entries and up to 4KB of user-specified name/value pairs.

Users can create, modify and read objects from buckets, things that make the security problems to appear. This issue leads to the implementation of algorithms to manage the access control restrictions over the blobs. Renaming an object or moving it to a different bucket requires downloading the entire object under one name and writing it back to S3 with the new name. Search is limited to queries based on the object's name and to a single bucket. No metadata or content-based search capabilities are provided.

When users register with Amazon's Web Services, they are assigned an identity and a private key. Both keys are permanently stored at Amazon and can be downloaded from the Amazon's Web Services website. Clients authenticate using a public/private key scheme and keyed-hash message authentication code. Because the private key is made by Amazon and downloaded from the website, the security provided by S3 is equivalent to security provided by a simple password which can be reset by anyone who can receive email at a registered email address. Each S3 account is linked to a credit card used for account billing.

Currently, S3 supports three data access protocols: SOAP, REST and BitTorrent. Because objects are accessible by unmodified HTTP clients, S3 can be used to replace significant existing (static) webhosting infrastructure. The Amazon AWS Authentication mechanism allows the bucket owner to create an authenticated URL with time-bounded validity. That is, someone can construct a URL that can be handed off to a third-party for access for a period such as the next 30 minutes, or the next 24 hours.

BitTorrent is a popular file-sharing protocol that enables efficient cooperative data distribution: data is initially distributed at one or more seed sites that are pointed to by a tracker. As clients begin to download a BitTorrent file, those clients register themselves with the tracker and make portions that they have downloaded available to other clients. S3 can provide both tracker and seed functionality, allowing for substantial bandwidth savings if multiple concurrent clients demand the same set of objects.

3. Data Anonymity

Today's technical and legal landscape presents formidable challenges to personal data privacy. First, our increasing reliance on Web services causes personal data to be cached, copied, and archived by third parties, often without our knowledge or control. Second, the disclosure of private data has become commonplace due to carelessness, theft, or legal actions.

In this direction, the fields of cryptography and steganography have provided the possibility for two parties to communicate without fear of a third party intercepting their transaction. However, one can imagine a scenario where an attacker captures both the data and one of the parties. The victim cannot claim the non-existence of hidden data if straight-up encryption is used, and the attacker may be able to determine the existence of hidden data in a steganographically hidden message. In order to obtain the information the attacker may use any form of persuasion to force the captured party to reveal decoding keys. In order to eliminate situations like this it would be desirable to have a communication system that provides plausible data hiding in case one of the communicating parties is compelled to reveal a secret key.

3.1 Self-Destructing Data

The main goal for this kind of approach is to create data that self-destructs or vanishes automatically after it is no longer needed. Moreover, it should do so without any explicit action by the users or ay party storing or archiving that data, in such a manner that all copies of the data self-destructs from all storage locations.

An application example that maps very well to this kind of data hiding is an email client. Emails are frequently cached, stored or archived by email providers or ISPs. Such emails may stop having any value neither for the sender nor for the receiver and thus, especially if the respective data needs to be private, a self-destructing characteristic of the email would be the perfect security resolution to have. The automatically vanishing of emails is already implemented in the case of email clients as some sort of virus, attached to the email that will automatically destruct it after a period of time. This kind of solution meets some problems, thought, when an anti-virus is installed on the host machine.

More generally, self-destructing data is broadly applicable in today's Web-centered world, where users' sensitive data can persist in the cloud indefinitely (sometimes even after the user's account termination). With self-destructing data, users can regain control over the lifetimes of their Web objects, such as private messages on Facebook, documents on Google Docs, or private photos on Flickr.

As seen before, the need for a self-destructing data kind of security is increasing at the same time with the applications used via web services or any other application network-based. There are several problematic concepts when it comes to implement this solution. One of them is if an attacker can obtain any copy of the data, and other relevant cryptographic keys, before the timeout that was set for its the self-destruction.

The second problem is if this situation, of automatically make the data to vanish, can be made without any explicit delete action made by the user or any other parties that are storing the data. Also, this should be made without needing to modify any of the stored or archived copies of the data.

Another issue that could be a problem in implementing the self-destruction algorithm is that it needs to be made without the use of any secure hardware (this should be able to be used by

any user with any kind of hardware available) or without relying on the introduction of any new external services that would need to be deployed.

3.2 Deniable Encryption and Anonymity

There are two types of deniable encryptions. A *static* deniable encryption scheme does not allow a user to modify any encrypted data after encrypting it without extracting all hidden messages. A *dynamic* scheme allows at least one of the encoded messages to be modified without knowledge of all hidden messages.

Interleaved encryption divides each message into a collection of blocks of a known size, encrypts them, and then combines them together in a non-contiguous order and stores them in an encapsulating file or device. *Hidden volume* encryption allocates a contiguous block for each hidden message. An *on-the-fly* system allows the user to encrypt data without directly running a third party program.

3.2.1 Steganography and Cryptography

Even thought a tempting solution when trying to hide data from a potential attacker, is to encapsulate sensitive data inside a normal file, this technique has several flaws. The data that we want to make it unreachable is not actually hidden, just that it is only placed in a location that is unlikely to be searched. Also, if the data is found, nobody can deny its presence or that anyone tried to hide it. All this type of security is about is to not be found by anybody. The data is not encrypted and if it is found it becomes public from that moment. This kind of hiding the data from the attackers is called steganography.

On the other hand, the technique called cryptography is more complex and is emphasized by three types of encryption: hash functions, symmetric key encryption and public key encryption. A hash function creates a fixed length output for any given message. In cryptography, a hash function *H* needs to have two important properties. The first one is that if we have a hash (h = H (m)) then the message used to be calculated (m) needs to be impossible to be found. The second one refers to the fact that we cannot have two different messages (*m1* and *m2*) so that their hash to be the same (H(m1) = H(m2)).

The symmetric key encryption method is based on the idea that the same key is used to encrypt and to decrypt a message. For example, the user X wants to send a message to user Y. To do so, X will firstly encrypt the message m in a cipher text c using the shared key k. When Y receives the text, the message can be decrypted with the same key.

Finally, in a public key encryption, each user has a public key and a private key. Anything encrypted with the public key can only be decrypted with the private key. If X wants to send a message to Y, she will encrypt the secret using Y's public key. Only Y can now decrypt the cipher text using its private key. Even X can't decrypt the message. To answer to X's message, Y will encrypt its message using X's public key.

The biggest drawback of public key encryption is that it is considerably slower than symmetric key encryption. In everyday life, public key encryption is generally only used to agree upon a shared secret key to switch to symmetric encryption.

3.2.2 Deniable Encryption

The deniable encryption got its roots from the idea that from the same encrypted data, depending on what cipher is used, different plain text data will be decrypted. There is no way to tell how many ciphers were used on the encrypted data.

All modern encryption systems are very hard to be broken by a simple passive eavesdropper from the network. Even if the third party will get the encrypted message is very unlikely to be able to decrypt it. Although, there are other methods to get the cipher of an encrypted message and one of them is by force. This is where deniable encryption comes and its aim is to resist at this kind of attacks.

In the work of Cynthia Dwork, Moni Naor, Ran Canetti, and Rafail Ostrovsky called "Deniable Cryptography" there is given an example of a deniable encryption situation:

A one-time-pad is a shared-key deniable encryption scheme. Assume that the sender and the receiver share a sufficiently long random string, and each message m is encrypted by bitwise xoring it with the next unused |m| bits of the key. Let k denote the part of the random key used to encrypt m, and let c = (m xor k) denote the corresponding cipher text. Then, in order to claim that c is an encryption of a message m' ≠ m, the parties claim that the shared key is k' = (c xor m'). It is easy to verify that this trivial scheme satisfies Definition 4. Here the message m' can be chosen as late as at time of attack.

"Deniable Cryptography"

This means that for any given message *m* you can say that you meant to send any *m'* where *m'* is of the same length as *m* by creating the key associated to *m'* according to *m'*.

3.2.3 Anonymity

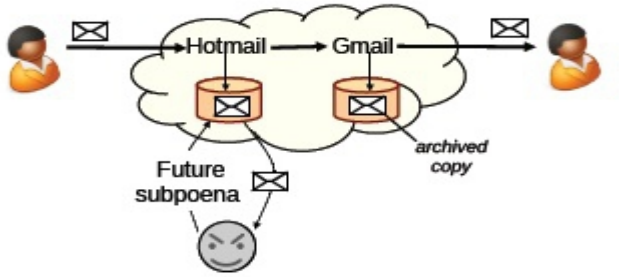
When studying cryptography one comes across the non-repudiation propriety a lot: an encryption protocol that guarantees non-repudiation prevents a user from denying having sent a message. If X sends a message to Y using such a system, Y or a trusted third party will be able to prove that the message was actually sent by X. This leads to the data anonymity problem. To be able to make a data have the property of anonymity we need to stop any proof that someone can bring stating that the respective text was written by X.

Not all encryption systems focus on the property of non-repudiation so the anonymity of the data could be made. If someone wants to use and encryption tool but it also need a high level of privacy, then the tool it uses should guarantee repudiation. This system has to guarantee that no proof of what has effectively been said between two parties can be collected.

3.3 Encryption Tools

3.3.1 Self-Destructing Data Tools

Vanish is a tool implemented by a team from the University of Washington. Their idea grew up from the problem met for the private emails, when a email that is supposed to be private sent from a party to another is recovered by the third party from the email provider's servers. The email had to self-destruct after it lost its value.



The key insight behind Vanish's approach is to leverage the services provided by decentralized, global-scale Peer-to-Peer infrastructures and, in particular, Distributed Hash Tables (DHTs). As the name implies, DHTs are designed to implement a robust index-value database on a collection of P2P nodes. Intuitively, Vanish encrypts a user's data locally with a random encryption key not known to the user, destroys the local copy of the key, and then sprinkles bits of the key across random indices (thus random nodes) in the DHT.

3.3.2 Deniable Encryption Tools

TrueCrypt is an open source program used to encrypt volumes on your computer. The volume can be a file (a virtual disk) or a disk partition, and is mountable in Mac OS, Linux and Windows. It is based on a dynamic deniable encryption scheme that utilizes a hidden volume to provide deniability. It provides on-the-fly encryption with a Windows driver that interfaces the encryption backend with a removable disk that may be dynamically mounted. The encrypted volume simply appears as another hard drive and is assigned a drive letter.

Using TrueCrypt you can create a hidden volume inside a normal TrueCrypt volume's free space. When mounting a volume using TrueCrypt, if the first 512 bytes are not decrypted by the given pass phrase, the program will try the same operation on the 512 bytes located at 1536 bytes from the end of the volume. This is the unique location for a hidden volume.

Elettra is a tool written using plausible deniability: it allows you to add many files with different ciphers in an archive. If someone comes and asks for the cipher, one of the used ones can be given while denying that there is any other one. Elettra is available for MacOS, Linux and Windows.

When you add a file to Elettra, it stores a gzip archived and encrypted version somewhere in the archive. In the archive there is a reserved space for storing header information regarding encrypted data. The header contains information such as the position of the encrypted data in the archive, the size of it, the size of the existing extra padding. Using a hash of the pass phrase, the location of the header is decrypted. Only after this is done, Elettra has the necessary information needed to extract the plain text data corresponding to the given pass phrase.

3.3.3 Data Anonymity Tools

Freenet is a decentralized and anonym peer-to-peer network and this leads to the fact that every user in Freenet is seen as a node and at its turn every node only knows its closest neighbors. Each file stored on Freenet is associated with a unique key allowing it to be found accordingly, thus, Freenet can be seen as a distributed storage system.

The principle behind a peer-to-peer network is that you can reach every node in a network just by being aware of a few nodes in the network. The link to any node from the network is made

by propagating the road from a neighbor node to another. The only problem with that kind of structure is how to route requests. At the starting point of the network, routing requests are made randomly so it is not as efficient as it will get afterwards, because the more it is used the more organized it gets.

Off-The-Record Messaging (OTR) is an open source project developed by Ian Goldberg, Nikita Borisov and Chris Alexander, who worked on implementing the tools needed to provide the same privacy found in a real life conversation, in online text messaging on applications like Adium or Pidgin. Three of the main rules that characterize a conversation from the real life to be private, in case a person whispers, as an example, to another person something, are that no one else will hear the conversation, they both know the identity of the other and no one can claim what the first person that was whispering actually said. Even if the second person would try to explain what it was told to it no one can tell that it is really the truth.

To be able to apply in practice this system it is needed to have a short-live encryption key at the disposal of the both parties. In this direction will be used the Diffie-Hellman key exchange protocol at the authentication process as well as during conversation as often as could be. The so called short-lived encryption key, that has nothing to do with the previous key used for authentication or any encryption added to the previous messages, is used for encryption of the messages until a new key is generated. If a third-party comes in the meantime and convinces one of the two parties that communicate to each other to give away the encryption key that would be useless in decrypting the entire conversation. Once the conversation is over no one, not even the two parties in action, can recover the conversation data.

4. BlobSeer

As the data-intensive applications grew in number over the past years, a team composed by members from INRIA and University of Rennes, developed a distributed data management system for Cloud environments meant to store large amounts of data. This service is made for any user who wants to store the data on a cloud via BLOBs using three simple operations: read, write and append.

The system focuses on heavy access concurrency where data is huge, mutable and potentially accessed by a very large number of concurrent, distributed processes. To be able to work with very large data BLOBs, each BLOB is cut into fixed-size *pages*, which are distributed among data providers.

BLOB Metadata facilitates access with the help of an *offset/size* pair for any existing version of a BLOB snapshot, by associating the corresponding pair with the physical nodes where the corresponding pages are located. Metadata is kept across the system using a Distributed Hash Table (DHT) and it is organized as a segment-tree structure.

The versioning system is introduced so that BlobSeer could deal with mutable data. This allows the users to have concurrent access over the BLOB, as the system enables the access to multiple versions of the same BLOB.

4.1 Main Features

BlobSeer stores huge amounts of data and at the same time executes data-intensive applications. This leads to the need of a compact and well implemented architecture which would contain the most important features of a distributive data management system: massive unstructured data, scalable architecture, high throughput under heavy access concurrency, data striping and distributed metadata management.

Massive unstructured data refers to the data organized as a set of large, unstructured sequence of bytes, named BLOBs (Binary Large Objects). These types of objects are the ones that BlobSeer was developed around. Each BLOB has a unique and globally shared identifier with which it can be found in the system. This characteristic also provides data transparency because the data can be accessed by any user only by knowing its identifier.

BlobSeer needs to be able to scale along with the addition of new storage or computing resources to the data centers or supercomputers centers. In cloud computing, the data management systems need to be able to manage up to thousands of nodes, which leads to the conclusion that to be able to assure these characteristics, BlobSeer has to have a **scalable architecture**.

Another feature that BlobSeer emphasizes is the fact that it assures **high throughput under heavy access concurrency**. Traditionally, the distributed management systems only allow the concurrent read, the write and append operations not being able to be made after the BLOB was already created. It is known that in cloud computing the most important aspect is that the processing of data is distributed and the parallelism is exploited at its most. In the context of data-intensive applications this translates to massively parallel data access that has to be managed efficiently by the system. At the same time, the data-intensive applications spend the most time on the I/O operations. These being said, one of the most important property of the system is to assure a high throughput in spite of the heavy access concurrency because this could heavily impact the execution time of the operations.

Data striping is a known technique that helps the system to increase the performance of the data access. In BlobSeer, each BLOB is split into equally-sized chunks which are distributed across

multiple storage nodes. The size of each BLOB is specified by the user, so that it can be fine-tuned according to the needs of the applications. This aspect allows the system to make the data access requests to be executed by multiple machines in parallel, which enables reaching high performance. As a side-effect, if the data chunks are too small, the overhead of identifying and transferring multiple chunks to a single process may be too large. BlobSeer is able to achieve high aggregate transfer rates due to *a configurable chunk distribution strategy* and *a dynamically adjustable chunk sizes*.

One of the key aspects that BlobSeer is developed on is the implementation of the **distributed metadata management** property. Since each massive BLOB is striped over a large number of storage space providers, additional metadata is needed to map subsequences of the BLOB defined by *offset* and *size* to the corresponding chunks. Such information is stored on specifically designed nodes and is employed by the users to discover the location of each chunk that has to be retrieved. Distributing metadata also has an additional advantage, namely it can eliminate *single points of failure* when the metadata is replicated across multiple servers.

4.2 Architecture

BlobSeer consists of a set of distributed entities that communicate with each other using asynchronous remote procedure calls (RPC) as illustrated in Figure 4.2.

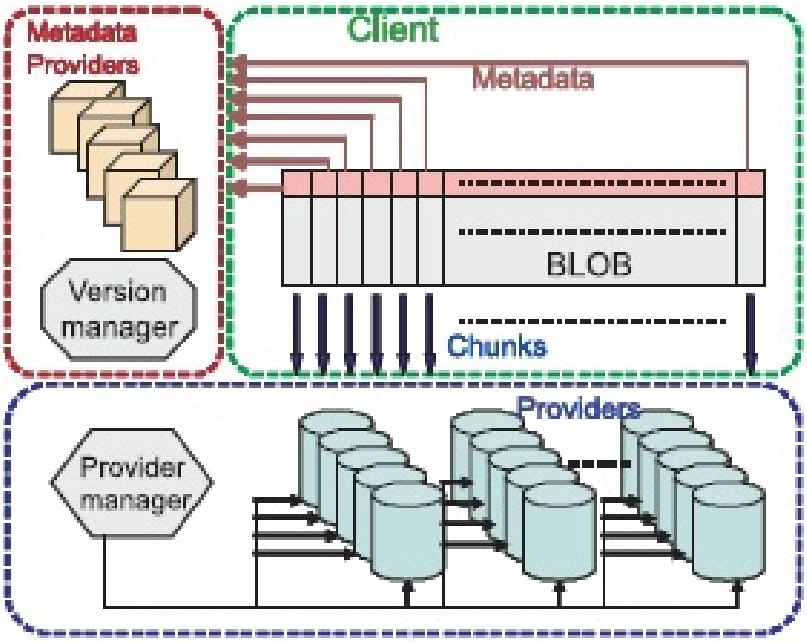


Fig. 4.2 BlobSeer Architecture

35 17 **Data (storage) providers** - BlobSeer relies on *data providers* to physically host the data chunks generated when writing or appending new data to a blob. This helps BlobSeer provide a scalable and efficient storage service. New data providers may dynamically join and leave the system.

35 17 **Provider manager** – The *provider manager* is responsible for assigning data providers to WRITE requests made by users. It keeps information about the available storage space and it employs a configurable chunk distribution strategy to

maximize the data distribution benefits with respect to the needs of the application.

- **Metadata (storage) providers** – To keep track of the chunks distribution across data providers, each BLOB is associated with a set of metadata. They are used to physically store the metadata that allows the system to identify the chunks that make up a snapshot version. For each BLOB, the metadata is organized as a *distributed segment tree*, where each node corresponds to a version and to a chunk range within that version. The metadata trees are stored on the *metadata providers*, which are processes organized as a Distributed Hash Table. Their implementation is similar to that of the *data providers*, metadata tree nodes being stored in a key-value cache.
- **Version manager** – The version manager is in charge of assigning new snapshot version numbers to users that make WRITE and APPEND operations and to reveal these new snapshots to users that make READ operations. Its goal is to create the illusion of instant version generation and to guarantee *atomic generation* of new snapshots each time the BLOB gets concurrently updated.
- **Client** – The clients can CREATE, READ, WRITE and APPEND data to or from the BLOBs from the system. The BlobSeer system is designed to handle many concurrent client operations accessing the same BLOB or different BLOBS.

4.3 Data Access Operations

BlobSeer provides to the client several primitives via its access interface. This user interface needs to be asynchronous and versioning-based and this is enabled by returning the control to the application immediately after the invocation of these primitives, rather than waiting for the operations to complete. When an operation completes, a callback function is called with the result of the operation as its parameters.

4.3.1 The CREATE Operation – CREATE (callback (id))

The CREATE primitive only involves a request for the *version manager*. The primitive creates a new empty BLOB of size 0. The BLOB will be identified by its *id*, which is guaranteed to be globally unique and the callback function receives this id as its only parameter.

In BlobSeer each data chunk is replicated on several data providers. The number of replicas, which sets the *replication degree*, has to be specified when the BLOB is created and therefore is constant for all data chunks belonging to the same BLOB.

4.3.2 The WRITE/APPEND Operation

– WRITE (id, buffer, offset, size, callback (v))

– APPEND (id, buffer, size, callback (v))

The client may update the BLOB by invoking the corresponding WRITE or APPEND primitive. The initiated operation copies size bytes from a local buffer into the BLOB identified by id, either at the specified offset (in case of write), or at the end of the BLOB (in case of append).

The WRITE operation guarantees *liveness*, as for each successful operation the corresponding snapshot is generated in a finite amount of time, *total version ordering*, because in the case of a successful write the snapshot having the returned version reflect reflects the successive

application of all updates numbered, and *atomicity*, which means that the snapshot appears to be generated instantaneously between the invocation of the WRITE primitive and the moment it is revealed to the readers by the system.

Write the content of a local buffer into the blob

```
1: procedure WRITE(buffer, offset, size, callback)
2:     K ← SPLIT(size)
3:     P ← get |K| providers from provider manager
4:     D ← ∅
5:     for all 0 ≤ i < |K| in parallel do
6:         cid ← uniquely generated chunk id
7:         roffset ← Σ_{j=0}^{i-1} K[j]
8:         store buffer[roffset .. roffset + K[i]] as chunk cid on provider P[i]
9:         if store operation failed then
10:            abort other store operations
11:            invoke callback(−1)
12:        end if
13:        D ← D ∪ {(cid, 0, K[i], roffset)}
14:        P_global ← P_global ∪ {(cid, P[i])}
15:    end for
16:    i_D ← uniquely generated id
17:    D_global ← D_global ∪ (i_D, D)
18:    (v_a, v_g) ← invoke remotely on version manager ASSIGN_WRITE(offset, size, i_D)
19:    BUILD_METADATA(v_a, v_g, D)
20:    invoke remotely on version manager COMPLETE(v_a)
21:    invoke callback(v_a)
22: end procedure
```

Fig. 4.3.2a WRITE algorithm [1]

To write the data chunks on the *data providers*, the client firstly contact the *provider manager* and asks for a number of *data providers* equal to the number of chunks that need to be written. After this step, the client uploads, in parallel, the data chunks to the *data providers* received.

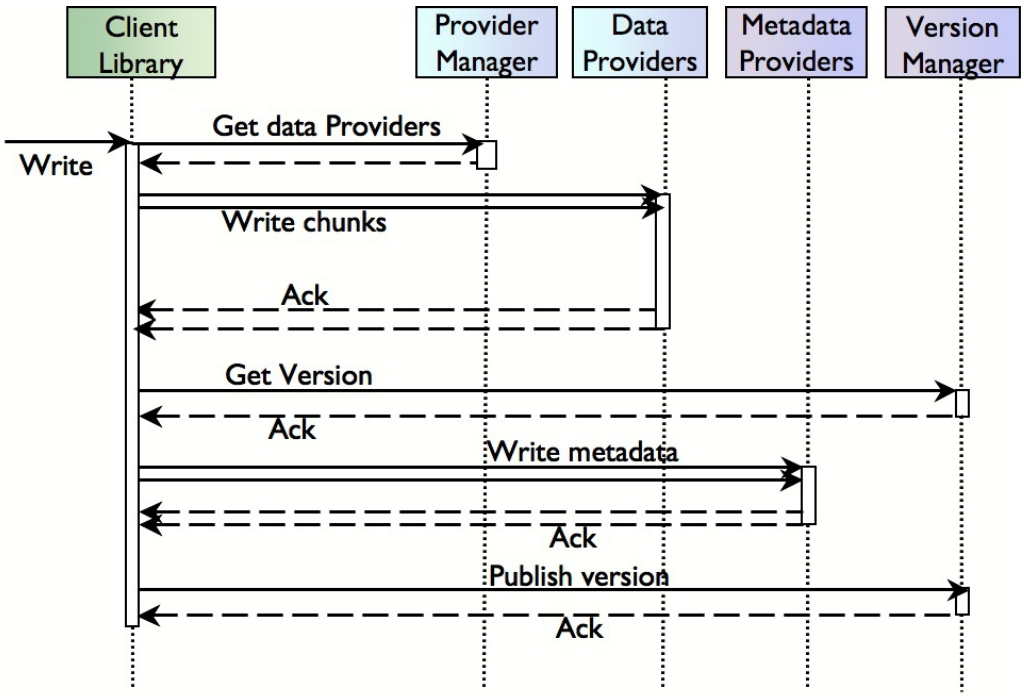


Fig. 4.3.2b WRITE operation in BlobSeer[2]

In order to make the uploaded data chunks available to the users, the client contacts the *version manager*, which assigns a version number and adds the WRITE operation into a queue of in-progress writes and after that the client constructs a metadata tree associated with the new version, so that the leaves corresponding to the written chunk range store the location of the *data providers*. Finally, the *version manager* is notified that the metadata associated with the new version is ready and it serializes all the concurrent writes before publishing the new version.

The APPEND operation is similar to the WRITE one, the only difference being that the client does not have to specify the offset where to write the data. The data will be automatically appended to the corresponding offset at the end of the data from the BLOB.

The algorithm describing the two operations is presented in the Figure 4.3.2a[1].

4.3.3 The READ Operation
– READ (id, v, buffer, offset, size, callback (result))

The READ primitive is invoked to read from a specified version of a BLOB. This primitive results in replacing the contents of the local buffer with *size* bytes from the snapshot version *v* of BLOB *id*, starting at offset, if *v* has already been generated. A Boolean value that indicates whether the read succeeded or failed is sent as a parameter to the callback function. The READ operation fails if the version requested has not been generated yet or if the total size of the snapshot, having the corresponding version, is smaller than the *offset + size*.

Read a subsequence of snapshot version v into the local buffer

```
1: procedure READ(v, buffer, offset, size, callback)
2:     v_g ← invoke remotely on version manager RECENT
3:     if v > v_g then
4:         invoke callback(false)
5:     end if
6:     (t, –, –, –) ← H_global[v]                ▷ t gets the total size of the snapshot v
7:     if offset + size > t then
8:         invoke callback(false)
9:     end if
10:    D ← GET_DESCRIPTORS(v, t, offset, size)
11:    for all (cid, co, csize, ro) ∈ D in parallel do
12:        buffer[ro .. ro + csize] ← get chunk cid[co .. co + csize] from P_global[cid]
13:        if get operation failed then
14:            abort other get operations
15:            invoke callback(false)
16:        end if
17:    end for
18:    invoke callback(true)
19: end procedure
```

Fig. 4.3.3a READ algorithm [1]

To make the READ operation, in a first step, the client will retrieve from the *version manager* the root of the metadata tree corresponding to the specified BLOB id and BLOB version. After the needed information is retrieved, the client scans the metadata tree in order to get the

location of the *data providers* that store the data chunks needed. Finally, the client efficiently downloads the data chunks in parallel from the *data providers*.

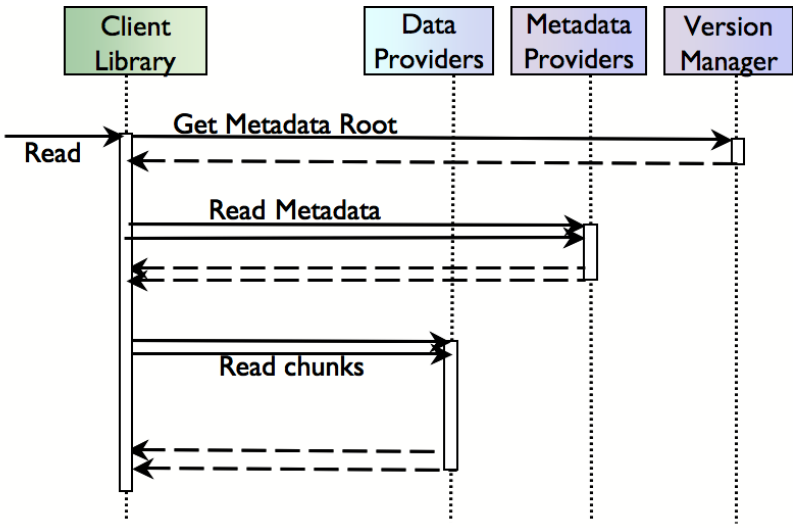


Fig. 4.3.3b READ operation in BlobSeer[2]

4.3.4 The GET_RECENT/GET_SIZE Operations

– GET_RECENT (id, callback (v, size))

– GET_SIZE (id, v, callback (size))

The GET_RECENT primitive queries the system for a recent snapshot of the bloc id. The result of the query is the version number *v* which is passed to the *callback* function and the size of the associated snapshot. A positive value for *v* indicates success, while any negative value indicates failure. The primitive is intended to just provide information regarding the latest version of a snapshot and it does not block any creation of new snapshots.

The GET SIZE primitive is used to _nd out the total size of the snapshot version *v* for BLOB id. This size is passed to the *callback* function once the operation has successfully completed.

4.3.4 The SUBSCRIBE /UNSUBSCRIBE Operations

– SUBSCRIBE (id, callback (v, size))

– UNSUBSCRIBE (id)

Some scenarios require the application to react to updates as soon as possible after they happen. In order to avoid continuously calling of the GET_RECENT primitive, the SUBSCRIBE and UNSUBSCRIBE primitives are created.

Invoking the SUBSCRIBE primitive registers the interest of a process to receive a notification each time a new snapshot of the BLOB *id* is generated. The notification is performed by calling the callback function with two parameters: the snapshot version *v* of the newly generated snapshot and its total size. The same guarantees are offered for the version as with the GET RECENT primitive. Invoking the UNSUBSCRIBE the client will not receive any notification about new snapshot versions for a given BLOB id.

5. Blobseer Data Anonymity

We will, now, present our solution regarding the data anonymity aspect from BlobSeer data management system. This chapter consists of all the research made related to this subject but strictly from a BlobSeer client's point of view. We take a look at the extra feature that our solution comes with, what technologies and algorithms were used and how this implementation can solve the problems of data anonymity in BlobSeer.

5.1 Overview

It is known that, from the client's point of view, BlobSeer has no module specially designed for data security and data privacy. For example, if client X makes a BLOB and writes data, that should be private, to it, another client Y could come and read any BLOB already stored on the respective *data provider* including the one with X's private data. This is possible because any user can have access to the BlobSeer system and implicitly to any BLOB that it is managed by it.

As stated in the fourth chapter of this work, the BlobSeer's architecture is based on the interaction of five entities: data provider, provider manager, metadata provider, version manager and clients. To be able to make the data private, action from the client side is needed. Thus, our solution is based on developing the *client* entity of the BlobSeer's data management system architecture.

The main idea of this approach is to create BLOBs and write the data to them that we want to be private in such a way that in case an attacker wants to get track of our data, it won't be able to do it even if it will know the BLOB's id. As this is not yet possible by restricting the access over the BLOBs created by us via the BlobSeer's interface primitives, we will analyze other possibilities and find a solution from the client's point of view.

First of all, we will take into consideration the self-destruct data method as it is the best approach to ensure that the data will not be found by any attacker or by someone that should not be able to have access to it. As the data could be kept only as long as it is needed and then it vanishes, it is impossible to be tracked down by an attacker. The problem comes from the fact that we need to keep that data and not to make it unavailable especially for the clients that created the BLOB and written the data.

Another solution to make the data private is the encryption of it every time it written to the BLOB. This solution is a viable one because today's cryptography reached a level of encryption that for any attacker it is a real challenge to try and decrypt the data without any other ways of persuasions. Still the problem appears when an attacker can get the cipher key directly from the possessor of the encrypted data by any possible ways. Also another weakness of this solution is that no data is hidden from anybody. All users can read the data from the respective BLOB but it is just encrypted and the attacker knows that and the only problem, until getting the correct data, remains the one of finding out the cipher to decrypt the it.

The solution that we came with was to encrypt the data and at the same time add another way of hiding it inside a BLOB. This aspect was inspired by the way Elettra is making the data it manages to be deniable. Besides the encryption added to any data written in the blob, it is, at the same time, split and scattered randomly inside a BLOB. In this way an attacker not only that will have to decrypt the data but its pieces have to be found in order to reconstruct the initial set of bytes. Thus, the attacker will not know where the data is actually set in the BLOB and how much it has to reconstruct.

By implementing this solution to any data written by the client using the BlobSeer's primitives, it is nearly impossible for any other client to track down the data and the initial user that has written the respective information. In this way, both the privacy of data and its anonymity properties are acquired.

5.2 Solution for BlobSeer

As all the possibilities were firstly analyzed to see which one is the most suited for being implemented for BlobSeer, we decided that the best one, from the BlobSeer implementation point of view and especially from the most suited data anonymity solution point of view, is an encryption by adding one of the cryptography's algorithm over the data and then make a padding of some randomly generated pieces of it inside the BLOB.

As we see, the processes of writing and reading the data from the blob will consist of two steps. For example for the write operation, the first step will consist of encrypting the data with a cipher and then apply an algorithm of randomly splitting and scattering the data around the BLOB at different offsets with different sizes of the chunks (note that the terms of *offset* and *chunk* are different from the ones that are a part of the BlobSeer's architecture). Also, the gaps created between these chunks of encrypted data need to be filled with any unimportant data so the algorithm of reconstructing the information is slightly more complex.

Two similar steps are added to the read operation as well but in the reverse order: we firstly have to reconstruct the data and then decrypt it. One of the issues that was having this implementation, was that we had to localize all the chunks' offsets in order to be able to put all the pieces of data together before trying to make de decryption of it.

To solve this problem we added another encrypted piece data that will be set at the beginning of the BLOB. This data will have the functionality of an header and it will contain the information needed in order to localize the chunks inside the blob and reconstruct the initial data.

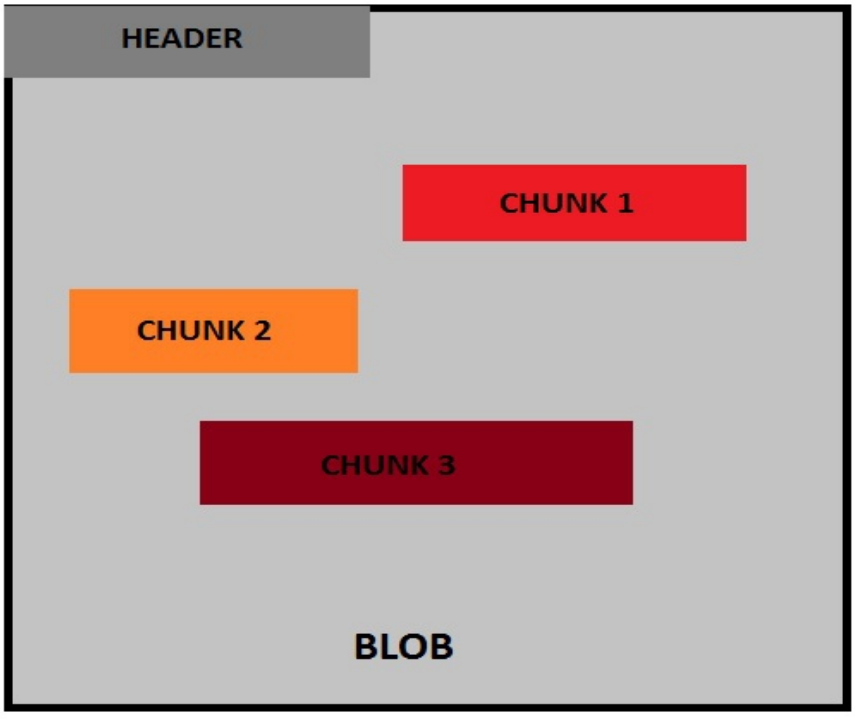


Fig. 5.2 BLOB's structure after the anonymity encryption is added

An example of the structure of a BLOB after the algorithm is added, is shown in Figure 5.2 having three chunks scattered randomly inside the BLOB after it was filled with random and unimportant data (the grey color describes the data that is randomly filled).

We will further discuss every part of this algorithm: the encryption of initial data, the splitting of the data and randomly dispersing it and finally, about the protocol on which the header is based.

5.2.1 Encryption of the Data

It is widely known that the most used technique to make any data safe from the attackers is by encrypting it. Although this kind of security alone is almost impossible to break, nowadays' security systems apply algorithms of hiding the data that are more complex. All of these systems include the traditional step of encryption as well.

In our case, just an encryption would not have been enough to achieve the data anonymity but it is a necessary part of our algorithm. Before splitting the data we will add a stage of encryption to it that would add a challenge to the attacker even if the data will be reconstructed.

In our implementation we chose to use a Password Based Encryption (PBE) as an example. This stage does not depend on a single encryption type as it could be used any encryption available or even add more cryptography algorithms at the same time.

A PBE algorithm is based on the technique of generating a secret key from a user-generated passphrase. Usually, this kind of security practice is not recommended because the threat of brute-force attacks has greatly increased in the last years.

BlobSeer's architecture does not have a security entity implemented yet and all the clients have access to all the managed BLOBs. This being said, the disadvantage of using the PBE encryption is not applying on our situation. First of all, a BLOB is organized as a long string of bytes that could be encrypted or not. Nobody knows what kind of information is kept into a BLOB as it could store both structured on unstructured data: pictures, text, music files format, video files format etc. In conclusion, in case of using a PBE encryption, brute-force attacks are not a threat for our solution as the attacker will not know when to stop from searching passwords and correctly decrypt data.

After the encryption is made, the next step towards providing anonymity to the data written into a BLOB is to split the data by a random algorithm and disperse the chunks inside the BLOB.

5.2.2 Padding the Data Chunks

As stated before, the encryption of the data alone would not provide the anonymity of the data. In order to have this property, our solution is to create a padding of the data inside the BLOB in order to hide the location of it from any attacker.

After the encryption was added to the data that we want to make it anonym, we will split it in a random number of pieces of random sizes. Every time the algorithm is applied on the same data the number of chunks will be different than the one from the previous run. Same applies to the size of the chunks, as even if the number of the chunks would be the same, the size of them will not coincide with the previous sizes.

When the splitting stage is finished, the data chunks created before, will be taken one by one and randomly scattered around the BLOB. We have pay attention here to the fact that the offsets need to be chosen outside the location where other data chunks should fit in order to not

overwrite our own data. In case the selected offsets of the data chunks overwrite the data of other chunks we will add a supplementary algorithm to the one of random selection. One of the solutions for this issue could be to add the size of the overwritten chunk to the newly selected offset or to lower the offset's position with the cumulated sizes of both of the chunks affected. This is showed in Figure 5.2.2.

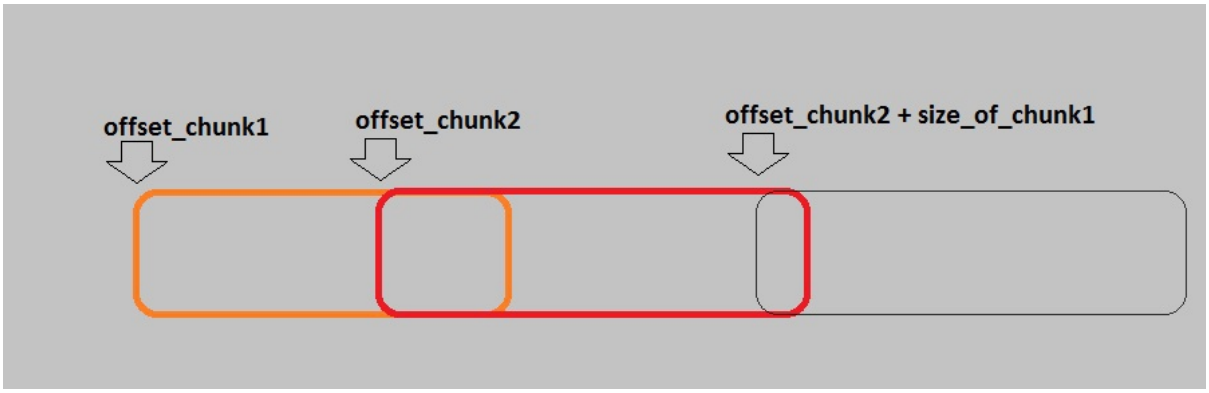


Fig. 5.2.2 Recalculation of an offset selected inside another data chunk

After all the offsets and sizes of the data chunks are calculated, we will write them one by one using the generic WRITE primitive. As the padding is finished we need to be sure that the gaps created, after the data chunks were scattered, are filled with unimportant data for us. This stage could be made before writing the data chunks, even if the overhead will be quite big.

For a more secure solution another encryption could be added to the data chunks separately. This makes the algorithm more complex and, even if it contributes to the overhead of writing or reading data into/from a BLOB, the data is more secure.

5.2.3 Header

In order to keep a track of the locations, sizes and number of data chunks we need to implement a *header*, at the beginning of the BLOB, which will hold this information. We have to add this functionality because this information is vital in reconstructing the initial data that needs to be decrypted afterwards.

We will add an encryption to the data hold in the header, as well, so that no other client could have directly access to it. For the encryption of the *header* it is recommended to use more complex algorithms of cryptography as it is the most important piece of the solution.

The information that the *header* stores are related to the data chunks created to be dispersed around the BLOB. The number of data chunks is the best information to start with. Depending on the number of the pieces, the size of the header will be determined. The size of this field will be set to four bytes in order to be in concordance with the size of an integer.

Further on, we need to add pairs composed by the offset and size of the data chunks for every chunk created. In this situation, the number of these pairs that are found is given by the number, which was previously set in the first field of the *header*, of the data chunks. To make the algorithm simpler the sizes of one pair was chosen to be of eight bytes: four bytes for the offset and the other four bytes for the size of the respective data chunk.

In the figure 5.2.3 it is presented the structure of the *header* with the number of chunks being set as N and respectively the number of pairs, of type offset/size, will be, as well, N.

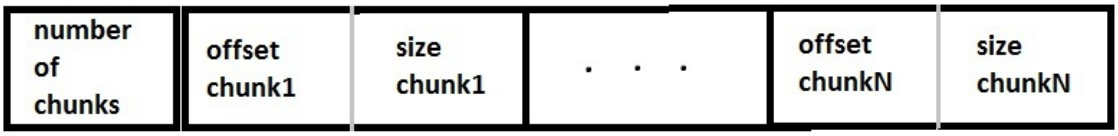


Fig. 5.2.3 Structure of the added *header*

We decided that the size of the BLOB's pages to be set as 64KB and this lead to the dimension of the header that we chose by default. First of all, the number of the data chunks resulted from splitting the initial data could not be larger than a number of type integer and that is why the first field of the header cannot be larger than four bytes of memory. By similar judgment, a pair of type offset/size cannot occupy more than eight bytes of memory.

5.3 Data Access Operations Modified

In order to be able to implement the suggested solution, several modifications are needed regarding how the operations of READ and WRITE work. It is not an actual modification of the primitives themselves, but more of a different way to use them when we want to write or read data to/from the BLOB.

As the proposed solution is developed as a BlobSeer client, the system's architecture and functionality will not be affected. Both the WRITE and READ primitives will have the same parameters and do the same atomic operations as before, while they are used by the client application developed according to the solution presented previously.

5.3.1 Write Operation

We will further present how the WRITE operation is modified from the primitive implemented for the client to use. As stated before, the primitive implemented in BlobSeer's user interface will not be changed, and it will be used, in the new operation of writing the data in the BLOB.

In order to start writing the data, the input parameters that are necessary are the buffer that contains the corresponding data and the id of the BLOB that should contain the respective data. As we will manage the offsets of the data pieces, the *offset* parameter will not be needed.

The first step from the writing process represents the encryption of data from the buffer. The running time of this step differs and it depends on the encryption algorithm that is used. It is known that for a more complex and secure type of encryption, the overhead time of the process increases accordingly.

After the data has successfully been encrypted, the next action that needs to be made by the operation is to split the data into the chunks that will be dispersed inside the BLOB. The splitting will be made in a random way so the chunks will have different sizes. This aspect greatly increases the chances of an attacker to not be able to successfully find the chunks and reproduce the initial data. On the other side, the chunks having different sizes will make the algorithm more complex when it comes to calculating the offsets of the data pieces.

The main issue that needs to be taken care of regarding the write operation is that the offsets of the chunks need to be chosen in such a way that the data contained in some pieces will not be overwritten by the data from other pieces. In case the randomly selected offset for a data chunk will be inside a chunk whose size and offset were already chosen, then the offset of the current chunk will be recalculated according to the formula presented in the chapter 5.2. The same logic applies in case the offset is not inside an already set chunk but the data of them overlaps.

After the pair containing the offset and size of a chunk was created for any of the data chunks available, we can proceed to the step of creating the data contained in the header as we have all the information that we need: the number of chunks and all the pairs of the type offset/size for every chunk. This data will be contiguous and set to be written at the beginning of the BLOB (the *offset* will be set as 0) only after it was encrypted. It is recommended that the encryption of the data contained in the header to be made with a different method than the one of the main data.

After all the encryptions, data splitting and calculations are made, the WRITE primitive is called successively until all the pieces are written at the corresponding offset. First of all, the encrypted data contained in the header will be written at the beginning of the BLOB and then, all the data chunks will be written at the correspondent offsets from the pairs defined before in the header. At this point the process of writing is finished. The steps are shown in figure 5.3.1.

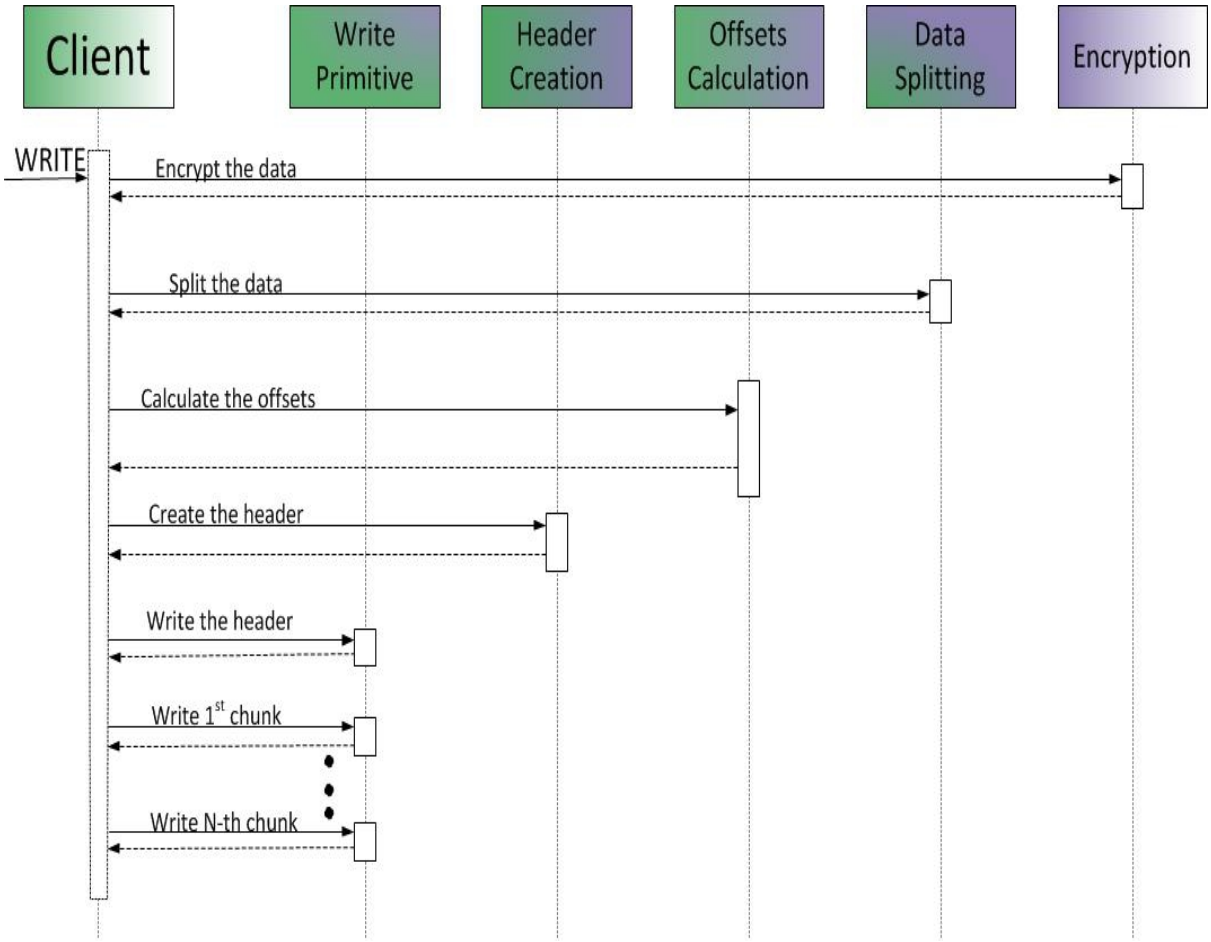


Fig. 5.3.1 WRITE process

5.3.2 Read Operation

The READ operation needs to be in concordance with the modifications added to the process of writing the data to the BLOB. Thus, even if, similar to the WRITE operation, the structure of the primitive implemented in the BlobSeer's user interface was not modified and just used for the READ operation, some modifications are made in the way a client has to read the data that was hidden inside the BLOB.

As the data is dispersed inside the BLOB, the *offset* and the *size* to be read are not needed as the input parameters. The input information that is important is composed of the *id* of the BLOB and the *version* of it.

The first stage from the operation of READ is represented by a call of the READ primitive to get the first portion of the BLOB where the header is kept. To be able to successfully take the entire header, we will have to read as much bytes as needed. This situation depends of the implementation made for the WRITE operation. As this will be a flexible solution, we cannot force any structure. We will suppose that the maximum length of the header is of 8 KB, a value that is valid in case we will have a maximum of one thousand data chunks.

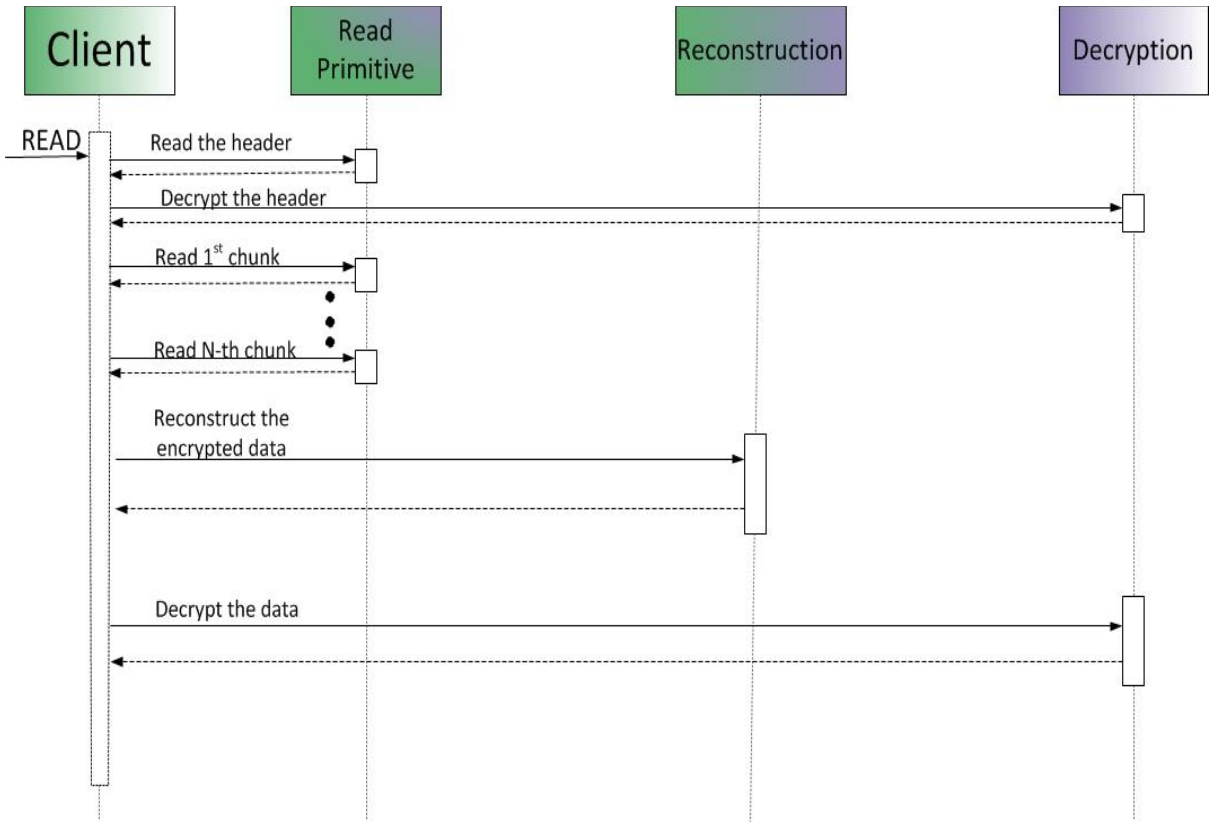


Fig. 5.3.2 READ process

The second step of the READ process consists of decrypting the header and extracting its information that is needed to have access to the data chunks from the BLOB. First of all, the first byte will be decrypted in order to find out the number of chunks that were created. After this information is found out, we are able to decrypt the rest of the data from the header as the size of it is known.

After all the information from the header is decrypted, we can proceed with reading all the data chunks that are inside the BLOB. These chunks can be found according to the offsets and sizes from every pair extracted from the header. Thus, every data chunk will be read one by one and added to a temporary buffer. After we have successfully read all the pieces of data, the reconstruction step can begin. All the read chunks will be taken one by one and added to the reconstructed data buffer in the order that they were read until no chunk remains unprocessed.

We need to keep in mind that the data that was reconstructed in the previous step is encrypted and it needs an additional action of decryption. After the data is decrypted and the entire information extracted, the read process reaches the end point and the returning buffer will contain all the data that was made anonym and private from the other unwanted clients. All the steps are presented in figure 5.3.2.

The main issue with the solution proposed by us is the impact that our algorithm has over the versioning system of BlobSeer. This problem is caused by the fact that at one write operation we actually increment the version more times than it should by using the WRITE primitive successively for the header and then for every data chunk. This could influence our functionality as well, because we have to know exactly what version to set as a parameter when we want to call the READ primitive. As the versions will be greatly increased this could become a bit challenging even if, after all, it is the same principle as before: we need to memorize the last version returned by the WRITE primitive.

6. Implementation Details

The previous chapter was dedicated to explain the functionality introduced by our solution and how the client's READ and WRITE operations are going to assure the anonymity of data as proposed. In this chapter we will talk about how this functionality is implemented.

6.1 Implemented Classes

The entire system developed consists of three main modules: the read module, the write module and a manager module that uses the first two ones in order to make the solution fully functional. Before presenting these modules we will talk about the utility classes used for the implementation of them.

The **Encryption** class is an abstract base class for any method of encryption that needs to be implemented. It consists of several pure virtual methods that are to be implemented by the extending class for every type of encryption implemented. From these functions the most important are *encrypt()* and *decrypt()* ones. Also, there are declared two main parameters that will keep the data as well as its value after the encryption is made: _data and respectively _encrytedData. As this is an abstract class, any other class can be implemented accordingly and used in the write and read implementations.

In our solution, we extend the Encryption class in order to implement the encryption that we used for the data to be hidden. The **PBE_Encryption** class represents the actual implementation of all the pure virtual functions defined in the parent class in order to make the encryption of data possible. The hierarchy of the classes implemented for the encryption module is presented in the 6.1a figure.

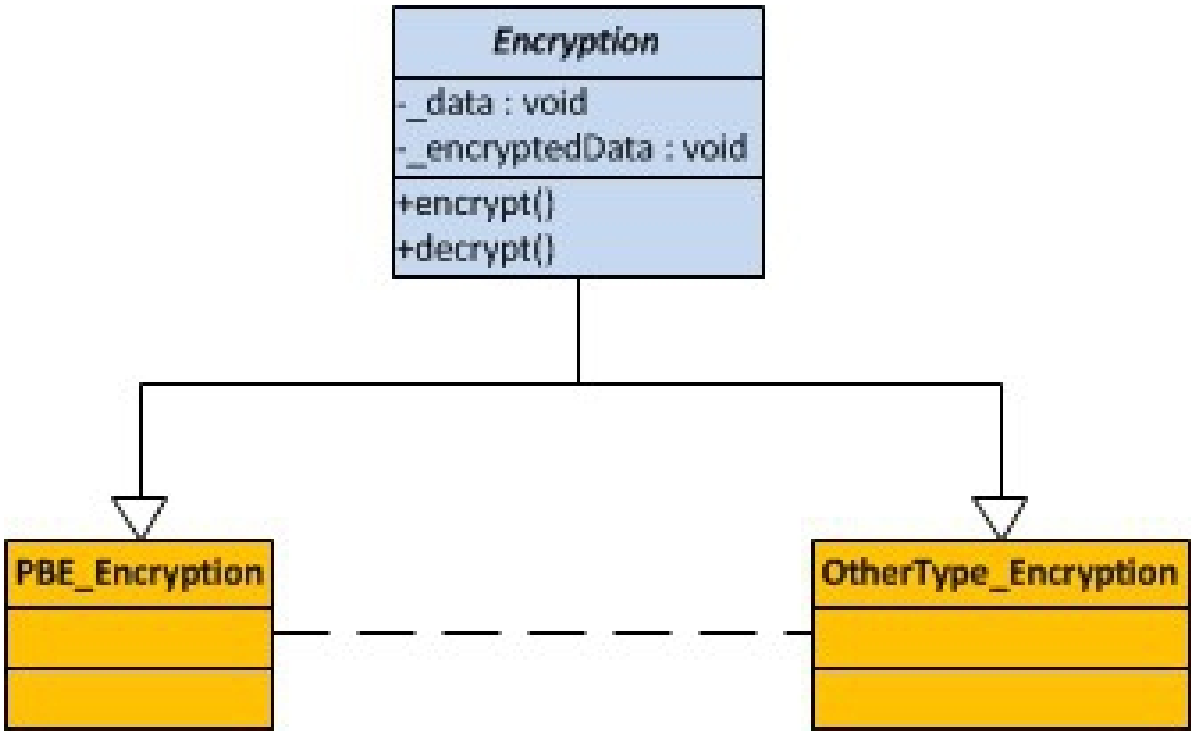


Fig. 6.1a The Encryption classes' hierarchy

Another class meant to be extended is the **Random** class. This class will contain the *random()* function that is initial implemented with the provided algorithm by the *C++*'s *std* library. The defined parameter that it will be inherited is the *_data* one, which represents the data intended to be further split and written. In order to make a more complex algorithm and more correct calculations of the offsets and sizes of the data chunks, this function will be extended, and it will be declared as virtual in order to have one instance of it, by the *OffsetRandom* and *DataSizeRandom* classes. Also, the Random class has the parameter *_numberOfChunks* defined, a parameter that represents the number of the data chunks that will be created. Another two predefined parameters needed are the arrays containing the offsets and the sizes of the data chunks: *_offsetsArray* and respectively *_chunkSizesArray*.

The **DataSizeRandom** class will extend the functionality of the *random()* function from the parent class so that the randomly created sizes of the chunks to correspond to the actually size of the entire data. An array, of *_numberOfChunks* length, will be created that will have applied at its every index the size of the corresponding data chunk. Similar to this class, the **OffsetRandom** class will extend the functionality of the *random()* function in order to calculate and set the correct offsets of the data chunks in concordance with their size. These two classes will be further extended by a manager class called **RandomManager**, which will make all the necessary actions to provide the randomly created sizes and offsets. The classes' interaction is presented in figure 6.1b.

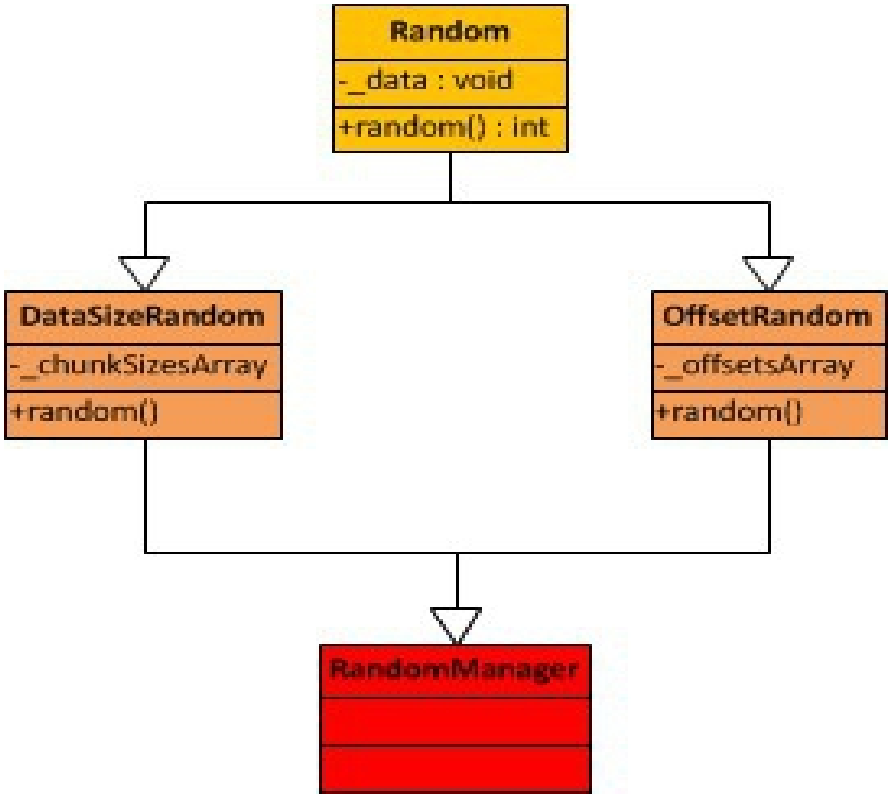


Fig 6.1b Random classes' hierarchy

Another utility class implemented is the one that will make the data splitting action according to the previous selected data chunks' offsets and sizes. This class will be named as **DataSplit** and it will have as a final result an array containing objects of type *DataChunk*, which is a structure that is a representation of a data chunk and contains all the necessary information. Also,

the class named **DataConstructor** will have the functionality of header construction, according to the *BlobHeader* structure, and the functionality of data reconstruction by implementing the *dataReconstruct* and *headerConstruct* methods.

All these classes presented will be the basis into implementing the **Read** and **Write** classes. These two modules will use all of the tools previously discussed and the WRITE and READ operations defined into the object handler that is implemented in the BlobSeer system. These two modules, at their turn, will be incorporated into the **Manager** module, which will take the decisions of reading or writing the data depending on the received parameters.

The input parameters to the application that will be run will have to define the operation that will be made: -R parameter will be used for read and -W for write operation. Also, the BLOB's id is needed and in case we are doing the read operation, the version of the BLOB has to be provided. In case of using the write operation, the file containing the data will have to be also provided and as a final parameter the password should be set in case it is used the PBE encryption. All the parameters could be modified by changing the implementation of the Manager class.

6.2 Data Structures Defined

To be able to implement all the classes, we had to define more data structures which would represent some used types of object. The two main structures are the *DataChunk* one and the one which represents the header of the BLOB named as *BlobHeader*.

The **DataChunk** data structure is defined by the data contained in the chunk, its size and its offset. The types of the three parameters will be set as a *void pointer* for the data and as *integers* variables for the offset and size parameters.

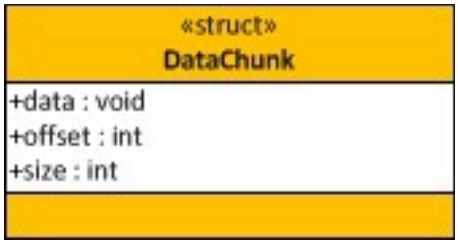


Fig. 6.2a DataChunk structure representation

The **BlobHeader** data structure is defined by the number of data chunks and by an array of pairs of type *ChunkPair* containing the information of a chunk's offset and size. The **ChunkPair** type, as explained, will be a structure which will contain the offset and the size of a chunk, both of *integer* type.

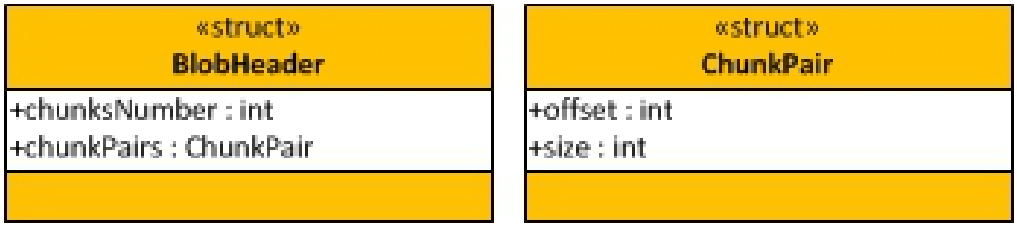


Fig. 6.2b BlobHeader and ChunkPair structures representation

7. Experimental Results

The system used for testing is composed of an Intel Core i3 CPU of 2.53 GHz and with 4 GB of RAM. The operating system installed and on which the tests were run is an Ubuntu 11 distribution with a 2.6 version of the Linux kernel.

The tests made consisted on making successive reads and writes of different sizes, in the first part without our implementation and in the second part of the test with our solution's implementation. There will be two sets of results, one for the WRITE operation and the second for the READ operation. We will make a comparison of the running times between the runs with and the runs without our implementation. The tested steps are for operations made for 256 MB, 512 MB and 1 GB of data.

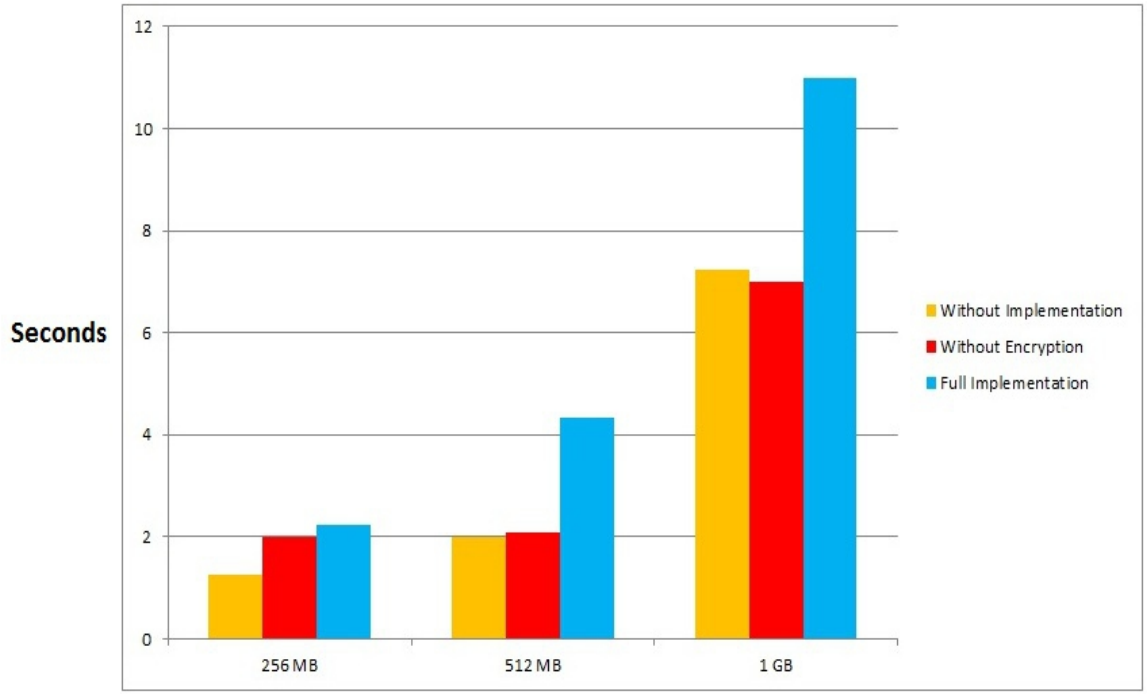


Fig 7a Running times of WRITE operation

In figure 7a yellow bar represent the speed of data write without any implementation, the blue one represents the speed of write with the full implementation and the red one represents the speed of write when writing only the data chunks without adding any other encryption algorithm.

Additionally to this information, we need to point out that there were used pages of 64 KB and the system was set to run without data replication. The configuration used for BlobSeer was made of a version manager, a data provider, a provider manager and a metadata provider.

Particularly to our implementation, the number of data chunks used for the tests were chosen to be set as four because the tests are made to point out the performance's change and not the functionality. Any overhead generated could be scaled in case of using more data chunks.

The results of the tests are shown in the diagrams represented in the figures 7a for the WRITE operation and 7b for the READ operation.

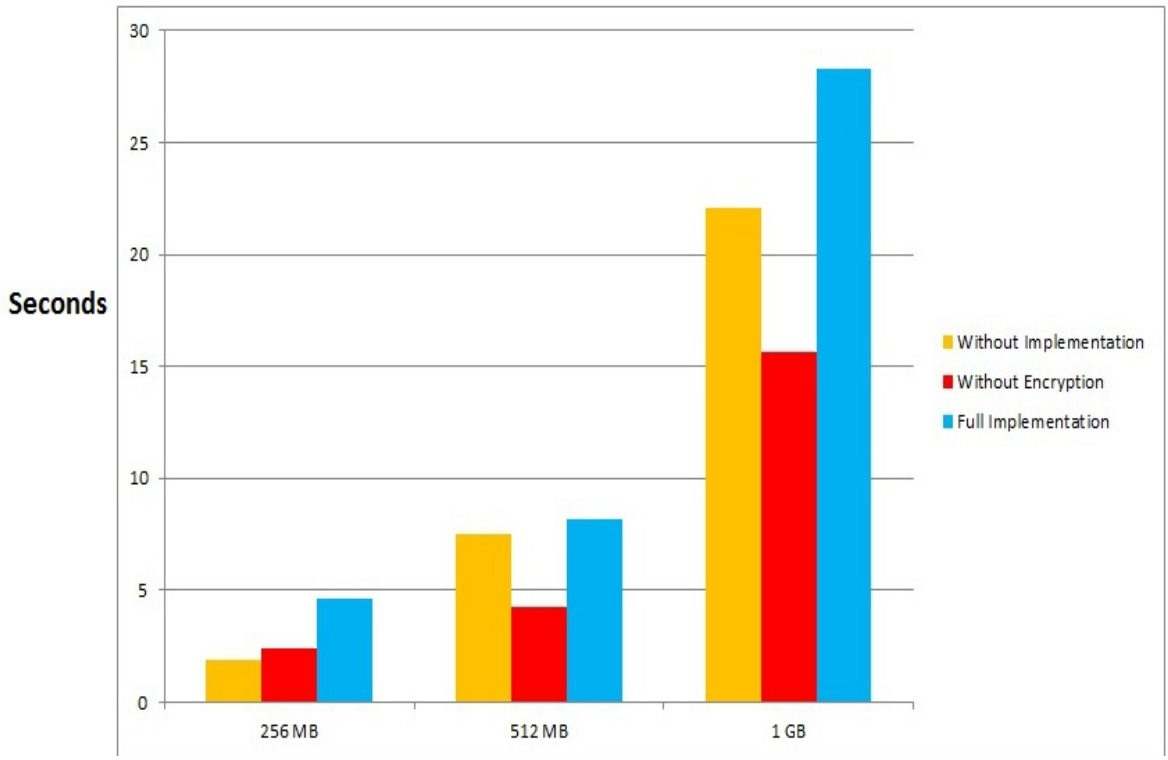


Fig 7b Running times of READ operation

The colors are chosen similar to the WRITE running time tests for the corresponding algorithms.

8. Conclusions

In this thesis we have presented our solution for one of nowadays' most focused issues from the domain of data management systems, the problem of data anonymity. More and more clients of the distributed storage systems available online want to be sure that their data is private and that no one else can have access to it or track it down. We consider that we have succeeded into implementing a good solution for achieving the anonymity of data written using the BlobSeer system's operations.

Even if the tests made with the implementation generated some overhead, our opinion is that this is an acceptable trade off with the advantage of having gained the anonymity of data from the BLOBs.

Currently, the implementation was tested with a low number of data chunks and it is expected to appear more problems when the number of data pieces increases. That should be the main issue to focus on for the future works. The system needs to be flexible for any data chunk sizes and numbers. Also, the header's structure and its security should be another aspect to be taken care of.

BIBLIOGRAPHY

[1] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *J. Parallel Distrib. Comput.*, vol. 71, pp. 169-184, 2011.

[2] PhD: Alexandra Carpen-Amarie, BlobSeer *as a data-storage facility for Clouds: self-adaptation, integration, evaluation*, ENS Cachan/Rennes, defended on December 15, 2011.

[3] V.-T. Tran, G. Antoniu, B. Nicolae, and L. Bougé, "Towards A Grid File System Based On A Large-Scale BLOB Management Service," in *Proc. Euro-Par '09: CoreGRID ERCIM Working Group on Grids, P2P and Service computing*, Delft, The Netherlands, 2009.

[4] Al Bento, "Cloud Computing: A New Phase in Information Technology Management", Journal of Information Technology Management.

[5] "HDFS User Guide", http://hadoop.apache.org/. Retrieved at 10.06.2012.

[6] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler. *The Hadoop Distributed File System,* In Proceedings of MSST2010, May 2010.

[7] S. Ghemawat, H. Gobioff, S. Leung. "The Google file system," In Proc. of ACM Symposium on Operating Systems Principles, Lake George, NY, Oct 2003, pp 29–43.

[8] Johannes Passing, "The Google File System and its application in MapReduce", Hasso-Plattner-Institute for Software Engineering, D-14482 Potsdam

[9] "Amazon S3 FAQs", http://aws.amazon.com/s3/faqs/. Retrieved at 13.06.2012.

[10] Mayur Palankar, Adriana Iamnitchi, Matei Ripeanu, Simson Garfinkel, "Amazon S3 for Science Grids: a Viable Solution?", Proceeding DADC '08 Proceedings of the 2008 international workshop on Data-aware distributed computing ACM New York, NY, USA 2008.

[11] Wikipedia. Deniable encryption. http://en.wikipedia.org/wiki/Deniable_encryption. Retrieved at 21.04.2012.

[12] Karstens, N L (2006), "Deniable Encryption", http://www.karstens.us/DeniableEncryption.pdf), pp 1 – 10. Retrieved at 17.02.2012.

[13] Jason Croft, Robert Signorile, "A Self-Destructing File Distribution System with Feedback for Peer-to-Peer Networks", 9th WSEAS International Conference on Applied Computer Science, Genoa, Italy, Oct. 17-19, 2009

[14] William CALDWELL, Pierre DUTEIL, "Data Hiding Under Mobile Storage"

[15] Roxana Geambasu, Tadayoshi Kohno, Amit Levy, Henry M. Levy. "Vanish: Increasing Data Privacy with Self-Destructing Data". In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.

[16] Martın Abadi, Bogdan Warinschi, "Password-Based Encryption Analyzed", Computer Science Department, University of California, Santa Cruz