UNIVERSITY OF LJUBLJANA
FACULTY OF COMPUTER AND INFORMATION SCIENCE

Bisera Milosheska

ON DATA INTEGRITY IN CLOUD STORAGE

DIPLOMA THESIS

UNIVERSITY STUDY PROGRAMME
COMPUTER AND INFORMATION SCIENCE

Mentor: dr. Andrej Brodnik

Ljubljana, 2012

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Bisera Milosheska

O CELOVITOSTI PODATKOV V OBLAČNI SHRAMBI

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: dr. Andrej Brodnik

Ljubljana, 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00049/2012
Datum: 15.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **BISERA MILOSHESKA**

Naslov: **O CELOVITOSTI PODATKOV V OBLAČNI SHRAMBI**
ON DATA INTEGRITY IN CLOUD STORAGE

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Že dolgo največji segment gospodarske dejavnosti predstavlja storitvena dejavnost. Slednje se dogaja tudi na področju IKT. Ena od storitev, ki jo ponujajo uporabniki, je shranjevanje podatkov in paradigma, ki jo pri tem uporabljajo je storitev shranjevanja podatkov v oblaku.

Uporabnik preprosto preko ponujene storitve oddaja svoje podatke v oblak in jih od tam pobira, pri čemer je količina podatkov takorekoč poljubna. Pri tem naleti na osnovno težavo, da mora verjeti, da je podatke, ki jih je odložil v oblak, dejansko v celoti dobil nazaj. Pri tem si ne more privoščiti, da bi podatke vedno v celoti prenesel k sebi in tako preverjal njihovo celovitost. Preučite možnosti in načine preverjanja celovitosti podatkov v oblačni shrambi.

Mentor:
doc. dr. Andrej Brodnik

Dekan:
prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisana **Bisera Milosheska,**

z vpisno številko **63090346**,

sem avtor diplomskega dela z naslovom:

Data integrity in cloud storage

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno, pod mentorstvom **dr. Andreja Brodnika**;
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.), ter ključne beside (slov., angl.) identični s tiskano obliko diplomskega dela;
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI"

V Ljubljani, dne 24.09.2012 Podpis avtorja:

Zahvaljujem se mentorju dr. Andreju Brodniku, ter asistentu Andreju Toliču za nasvete in pomoč pri izdelavi dilomske naloge. Iskrena zahvala gre tudi sorodnikom in prijateljem za vso podporo v času študija.

I give my acknowledgement to my mentor dr. Andrej Brodnik and assistant Andrej Tolič, for helping me in the process of writing the diploma thesis. My deepest appreciation goes to my relatives and friends, for all of the support during my studies.

Table of Contents

Povzetek

Diplomsko delo obravnava problem integritete in dostopnosti podatkov shranjenih v oblaku. Na samem začetku, bralec je spoznan z osnovami problema integritete oddaljenih datotek in razlogi zakaj prav ta problem zasluži našo pozornost.

Po kratkim uvodom sledi podrobnejša analiza najsodobnejšega stanja tehnološkega razvoja, oziroma "računalništvo v oblaku" (cloud computing). Da bi bralec pridobil boljšo percepcijo pomena tega pojma in njegove vloge v današnji tehnološki vseprisotnosti, podamo kratek opis zgodovine, kateremu sledi še analiza njegovih značilnosti, storitvene in izvedbene modele. Omenjene so najbolj pomembne karakteristike koncepta računalništva v oblaku, kot so: samopostrežba na zahtevo, širok mrežni dostop, združevanje sredstev, visoka elastičnost in plačilo na osnovi uporabe. Poznamo tri storitvene modele katerih definicij so podane v tem poglavju, kot so: infrastruktura kot storitev, platform kot storitev in programska oprema kot storitev. Bolj podrobno so razložene tudi vse izvedbene modele računalništva v oblaku: zasebni oblaki, javni oblaki, oblaki skupnosti in hibridni oblaki. Čeprav se sliši zelo tipično, ne moremo izostaviti pomembnost prednostih in slabostih tega koncepta. V nadaljevanju razlagamo bistvo značaja shranjevanja podatkov v oblaku, posebej za podjetja, ter pozitivne in negativne posledice takšne oblike shranjevanja podatkov. Med pozitivnimi posledicami je dejstvo da ni potrebe po investicij v osebno infrastrukturo, stranki je omogočena dinamična nadgradljivost in zagarantirana dostopnost do podatkov kadarkoli in kjerkoli, preko interneta. Med negativnimi posledicami spada slaba kontrola nad podatkov, ter zanesljivost ponudnikov storitev v oblaku. Ta oblika shranjevanja podatkov je lahko vzrok za varnostne težave ki vplivajo na dostopnosti in integriteti podatkov. Zaradi tega smo posebej obrnili pozornost na naravo takšnih varnostnih težav.

Namen tretjega poglavja je poudariti vprašanje za integritete datotek na splošno in podati podrobnejšo razlago o vzrokih za nedoslednosti podatkov, kot strojne in programske napake,

zlonamerni napadi in uporabnikove osebne napake. Za tem sledi še ideja za rešitev ki bi se lahko uporabljala za datoteke shranjene na oddaljenih strežnikih, oziroma v oblaku. Rešitev bo preverjala integriteto in doslednost oddaljene datoteke v dveh fazah: namestitvena in verifikacijska. V namestitveni fazi določimo katere bite(meta-bite) se bodo uporabljale za izvedbe verifikacije in njihove pozicije shranimo pri odjemalcu. Njihove vrednosti pa enkriptiramo in jih dodamo k originalni verziji datoteke. Po razširitev, datoteko lahko pošljemo zunanjemu ponudniku oblačne shrambe. V verifikacijski fazi odjemalec zahteva vrednost določenega bita, ter enkrpitiran blok v kateri ta bit spada. Po primerjavi njihovih vrednosti, odjemalec dobi odgovor ali je integriteta datoteke potrjena.

Naslednje poglavje je namenjeno programski implementaciji ponujene rešitve in ocenjevanja rezultatov pridobljenih pri testiranju programa. Programska implementacija je narejena v programskem jeziku Java. Za cilje tega programa, sta narejena dva posebna razreda *DIclient* in *DIserver*. Prvi razred opravi vse funkcije s strani uporabnika(*setup, verify*), kot generiranje meta-bitov, njihovo enkripcijo, dodajanje enkriptiranih blokov k origilani datoteki, ter verifikacijo vrednosti meta-bitov. Drugi razred opravlja funkcije s strani ponudnika oblačne storitve(*metaBit, metaBlock*), oziroma vrača zahtevane vrednosti bitov in blokov. Za izbrane ključne parametre, kot velikost bloka, število meta-bitov v en blok in število meta-bitov ki jih preverjamo v eni verifikaciji, rezultati testiranja so potrdili veliko verjetnost ugotovitve deleža pokvarjenih bitov.

Evaluacija služi kot osnova za naš zaključek v zadnjem poglavju in za predloge izboljševanja te rešitve. Med podanih nedoseženih idej ki bi bile lahko realizirane v prihodnosti je prilagoditev rešitve za dinamičnih datotek, oziroma datoteke katera vsebina se pogosto spreminja v oblaku. Druga omejitev naše rešitve je vnaprej določeno število izvedb verifikacije, kar omejuje življenski cikel programa. Tudi poleg omenjene slabosti naše implementacije, problem integritete podatkov je zalo dobro in podrobno elaboriran, ter ponujena je kakovostna rešitev ki lahko pride v poštev tako za posameznike, kot za večja podjetja.

Ključne besede: računalništvo v oblaku, shranjevanje podatkov v oblaku, integriteta podatkov, oblačne storitve

Abstract

The diploma thesis elaborates the problem of integrity and availability of data saved in cloud storage. At first, the reader is introduced with the basics of the studied problem and the reasons why precisely this theme deserves the scope of our attention. The brief introduction is followed by a deeper analysis of the state of the art of the technological evolution, i.e. cloud computing. In order to get a better perception off its meaning and role in today's technological omnipresence, we present a short description of its history followed by an analysis of the concept's characteristics, service models and deployment models. As typical as it may sound, we could not disregard the importance of the concept's advantages and disadvantages. Further, we explain the significance of cloud storage especially for businesses, as well as its positive and negative consequences. As this form of data storage may provoke security issues related to the availability and integrity of data, we pay special attention to the nature of those issues. The third chapter is intended to emphasize the problem of data integrity in general and offer an idea for a solution that may be applied on files stored in cloud storage. The solution will be specialized on checking a remote file's integrity and consistency. The next chapter presents the programming implementation of the suggested solution and evaluates the testing results of the program. This evaluation is used as a ground for laying our conclusions in the final chapter, as well as our propositions for some unaccomplished ideas that may be realized in a future work.

Key words: cloud computing, cloud storage, data integrity, cloud computing services

1 Introduction (motivation)

The number of software solutions that choose to exploit the cloud for storage purposes progresses exponentially. Even though the benefits of this choice are admittedly convincing, there are some serious security issues that need to be addressed.

When an individual/company decides to store its data off-premises, the data owner must be aware of the consequences the loss of control may provoke. When data leaves the confines of an individual/company's control facilities and falls in the hands of another owner, such as the cloud owner, the availability and integrity of that data are threatened. As no proper solution to address these threats has been developed, we decided to study a solution that would be acceptable both for individuals and for companies, and attempted to implement it in real life setting.

Our primary goal was to develop a solution that will enable the client to check the integrity of its remote data at any time. As we carried on with the development of our initial idea, we realized that the solution must not be computationally too demanding. The efficiency gains were part of the cloud computing benefits to persuade us on outsourcing data and services to a third party, at the first place. Thus, we may conclude that the solution must not, under any circumstance, download the outsourced file or read it entirely.

In this diploma thesis, we develop a solution that will access only a part of a file and will still be able to determine the consistency of that specific file (i.e. whether that file has been corrupted or not), with high probability.

2 Cloud

A regular, non-IT educated person is often confused by the use of the term “cloud” in the computer science world, as it is hard to make a parallel between something as intangible as a cloud and something as defined as technology. In reality, the term represents an abstraction of the latest revolution in computing. Starting from the mainframe, later replaced by the wide use of PCs, shifting to the client/server architectures, leading to the Web… The evolution of computing plausibly moves further away from traditional IT, approaching the customer with an offer to leverage the benefits of technology at a much lower cost level, which is especially attractive for businesses. As the latest computing frame distance itself from the traditional IT, leaning toward the end-user by means of the Web services, we are starting to visualize the complete scene where computing actually happens, i.e. in the cloud.

Following the predictions posed by computer scientist John McCarthy back in the sixties, we are now convinced in the ingenuity of the idea that computation should be delivered as a public utility. Unfortunately, as much as this idea was theoretically valid at the time, the Internet did not offer significant bandwidth to support its realization, until the nineties. The revolution started with the first one to embrace the concept of delivering applications via websites, Salesforce.com. Following the same fashion, Amazon released a couple of solutions offering storage, software and even human intelligence services (Amazon Web Services offer Amazon Elastic Compute cloud (EC2) and Amazon Simple Storage Service (S3)). The revolution expands even more as Google launches the browser-based applications (Google Docs), to inevitably reach the level of tech companies following the trend, while adopters would only amplify.

Although no concrete definition of cloud computing is fully accepted in all of its parts, I would personally give my appreciation to the broadly approved paper[9], written by Peter Mell and Tim Grance of the National Institute of Science and Technology (NIST), which offers an in-depth explanation of cloud computing. The concept, referred to as the “evolving paradigm” is described as:

> “… a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.”

In simple terms, third-party owners of computing resources offer customers those resources for rent under very favorable conditions. The regular maintenance costs imposed to a single customer, such as expensive skills and inefficient repeated effort, are now suppressed by the process of automation. In order to alleviate the investment risk, the infrastructure must be efficiently organized and structured. Therefore, withdrawing from the scenery of traditional usage of IT, the goal of organizing the infrastructure is best acquired by automating the process and therefore, cutting costs. That implies less expense associated with maintenance, data centers, servers, security, configuration management, bandwidth, etc., as the customers have the chance to rent the IT resources of their choice in quantities defined by their real needs, without capital commitments or traditional build overhead. “Using a public cloud, anyone with a laptop and credit card will be able to prototype and deliver services at an unprecedented scale.”([1]) Thus, the control goes in the hands of the customer.

Now, that statement would arise the following question: ”To which extent does the customer’s power over the resources he rents really amount?”. Considering the fact that data centers are entrusted to third parties and services are bought replacing the internal maintenance of applications, there is no doubt that the question poses a wide topic for further debate that would be discussed in greater detail in a later chapter.

Nevertheless, one thing is for certain. As more and more customers welcome the idea behind the concept of cloud computing, it slowly makes its path through the technological evolution, stressing its own revolutionary nature. In that honor, we would like to introduce

the core components of the concept (five essential characteristics, three service models, and four deployment models as defined in [9]) in greater detail.

2.1 Specifications, characteristics and qualities

Numerous characteristics justify the adoption of the cloud computing concept, several of which, according to the NIST definition in [9], deserve special attention.

On-demand self-service. The resources offered by the clouds are more and more used for real, vital tasks and serious businesses are becoming dependent on the resource providers' competences. That is a significant reason why the providers must guarantee clients access to the resources at any time, from any location, without the need of interaction with anyone from the provider's company.

Broad network access. The most fascinating quality of cloud computing, which is worthy of its smooth adoption and great success, is the network-based approach. This allows customers to gain access to another computer, which is in possession of the cloud, from any computer, and work on it as if they are working on their own.

Resource pooling. The computing resources provided by cloud owners are not intended for the use of one user only, but multiple consumers simultaneously. Resources such as storage, servers, security, configuration management, bandwidth, virtual machines etc. are shared to multiple tenants who can afford to pay access to a particular infrastructure. Managing a server farm requires a lot of work and maintenance, since resources must constantly be dynamically reassigned.

Rapid elasticity. The resource providers are capable of coordinating their computing capacities in a way that would cushion the demand for their services and reduce the outages and service interruptions.

Measured Service. As the word itself points out, services used by clients should be measured by the provider, not only in order to properly bill them, but to keep their rented hardware and software up to date, as well.

2.2 Service models

The following three service models are well known in cloud computing:

Cloud Software-as-a-Service (SaaS). It would probably not be a mistake to assume that every user of the Internet is familiar with the commonly used SaaS implementations, among which we can single out Gmail, Google Docs, as well as a great portion of the mobile applications. These services represent applications that do not defer significantly from the classical desktop applications, accept for the fact that the application as well as the data related are stored in the hands of the service provider. That does give the client less freedom and control over the underlying infrastructure, but the efficiency gains and cost reductions outweigh the limitations' consequences. We access these applications through a web browser, which has lately become the first tool a regular computer user searches for after turning on the computer. Thus, there are practically no outward differences of how we experience desktop and Web applications.

Cloud Platform-as-a-Service (PaaS). The PaaS model allows developers to build and deploy their own web applications on a hosted infrastructure (network, servers, operating systems, or storage), where it should be noted that the tools and languages used for creating the application must be supported by the provider. In this way, developers make use of the infinite compute resources of a cloud infrastructure, without having to take care of configuring, securing and maintaining the system. These circumstances are very mitigating, even though the developers' control is only limited to their own applications.

Cloud Infrastructure-as-a-Service (IaaS). The IaaS offers developers, as system and network administrators, resources such as virtual servers, data storage, and databases into one platform for deploying and running their software. The consumer is relieved from planning, configuring, setting up and maintaining the system environment, on the expense of having limited control which expands over his applications, storage and operating systems.

These three cloud computing service models are referred to as the SPI model.

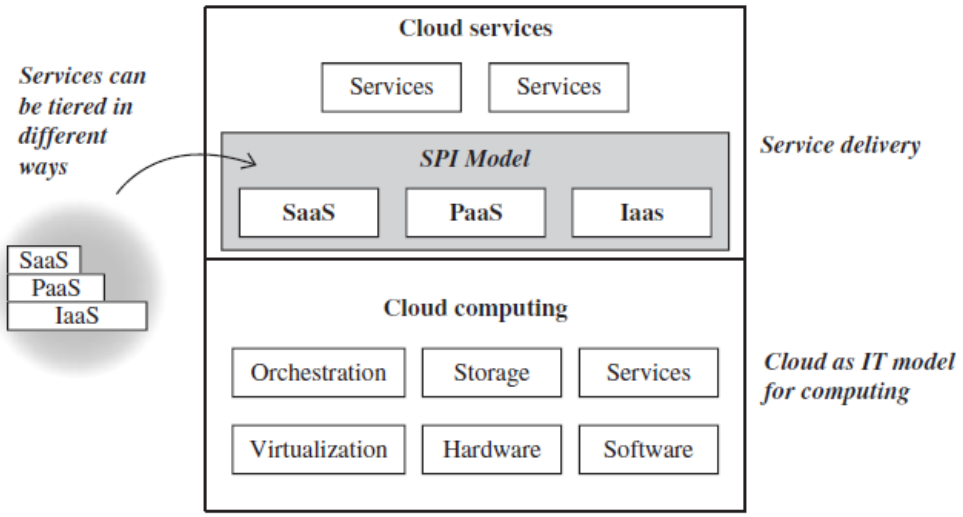

Figure 2.1 *The SPI model: software, platform, and infrastructure as a service.* ([7])

2.3 Deployment models

There are four alternatives that clouds offer as deployment models:

Private cloud. The most significant value the concept of this model adds to the cloud is security. Control over data and the overall system are especially stressed, however at the expense of cost inefficiency. Another reason why an organization may choose this model would be to automate and systemize the use of virtual infrastructure, if the organization already uses one. The model of the private cloud cuts the benefits offered by the public cloud, and consumers are now responsible for buying, building and managing their own infrastructure which implies significantly higher investment costs.
The cloud infrastructure is intended for the use of only one specific organization.

Community cloud. In this model, several organizations share the cloud infrastructure. Most of the time, these organizations share similar interests, requirements, mission, policy and compliance considerations. It would be inevitable and totally correct to assume that besides all of their common characteristics, organizations using this cloud share the expanses for its establishment, as well. Thus, costs per organization are significantly lower compared to the model of the private cloud.

This model should be useful for government departments that work with the same data related to the population or the infrastructure of one country.

Public cloud. The cloud infrastructure is made public by the cloud service provider and is provided to various clients. The use of the public cloud is far less costly than any of the other deployment models. This model is most suitable for developing and managing applications intended for the use of many users, in order to cut the investment costs in infrastructure.

Hybrid cloud. The cloud infrastructure is composed of a number of clouds, which differ by type, but still have the ability to inter-communicate, i.e. are bounded by technology that enables data and applications portability. This model allows customers to take advantage of the best qualities of each of the consisting models, depending on our requirements. Sensitive data and part of the applications that require stronger security measures can take advantage of the control over the resources allowed by the private cloud. On the other hand, the costs imposed by the use of the private cloud can be balanced by sharing the rest of the data and applications on the public cloud.

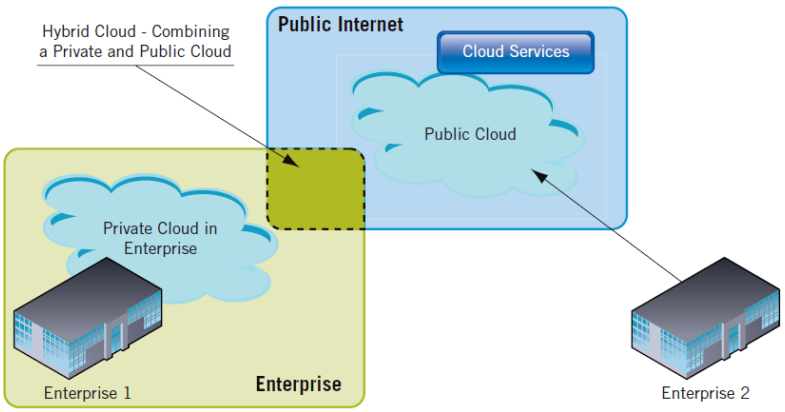

Figure 2.2 *Public, Private and Hybrid Cloud Deployment Example* ([12])

2.4 Roles and components

The use of cloud computing resources is supported by several key roles. Their contribution is worthwhile a more detailed explanation.

Cloud provider: a person or a company, which provides and promotes cloud computing services. For the establishment and management of the system, a lot of effort and knowledge are required.

Cloud user: a person or a company, which rents cloud computing services. The inputs and knowledge for performing this role are minimal.

Cloud client: hardware/software that is completely dependent on the cloud computing services for application delivery, is especially intended for the use of these services, and is useless without them. Ex.: Web browsers, thin clients, mobile platforms.

The following components represent the resources that may be provided by a cloud owner:

Cloud services: real time services delivered and used via the web, intended to end-users and connections to other cloud computing components.

Cloud application: an application offered as a service, often without the need of configuring or running on a local computer.

Cloud platform: platform as a service enables users to deploy their own applications, being spared of investments in compatible infrastructure.

Cloud storage: data storage in the cloud, as a service.

Cloud infrastructure: typically virtualized infrastructure (servers), as a service.

2.5 Advantages

There is no doubt that the burst of cloud computing owes its glory to the many qualities the concept offers. In light of what we now understand as the founding components of the concept of cloud computing, we are about to revise its crucial advantages.

There is no need for investment in personal infrastructure establishment! The complete infrastructure and maintenance services can be rented in exact quantities. That means that the customer is relieved from enormous expenses on maintenance, data centers, servers, security, backup and recovery, configuration management, bandwidth, as it pays only for as much as he consumes. In other words, the customer has the opportunity to choose not "to invest in additional servers that only run at 70% capacity two or three times in the year, the rest of the time sitting at 7-10% load"([2]). As we are aware of the fact that technology never seizes to evolve, it is very important to stress that the customer is relieved from software updates/upgrades and hardware replacements, as well. In addition to that, there will be no need even for personnel typically in charge of planning, configuring and maintaining the infrastructure. Therefore, the customer enjoys great efficiency gains, accompanied by even greater cost reductions.

The customer is guaranteed dynamic scalability! When a company launches its application on the Web, three scenarios are possible. The first possibility is that we underestimate the demand for our services and thus, the need for computing capacity. The second scenario would be the optimal case of an exact estimate for needed capacities. And the third scenario, which is as unpleasant as the first one, would be to overestimate our potential and invest in powerful software and hardware, which would never return our investment costs. As the optimal scenario is very rare, the cloud computing concept takes into consideration every possible situation and the customer is able to increase/decrease its demand for cloud computing services at any time and at very low costs. The customer is even able to withdraw from the scenery, dismiss the contract and save himself from further expenses.

The rented cloud computing resources are available at any time and any place, as long as the access point has Internet connection. The resource providers guarantee a nonstop availability of resources within the reach of the customer alone. The resources' consumer must always be able to access his data, or use the provider's services in a "self-service" fashion, without the need of additional help coming from a third-party.

2.6 Disadvantages

As the cloud reveals itself, we find out there is more to the nicely wrapped package of great opportunities. The responsibilities and care related to the cloud resources fall completely in the hands of the resource provider, which imposes a whole new set of risks and challenges.

The fact that data doesn't physically exist within an organization's premises is very alerting in terms of security, i.e. privacy and integrity of data. More and above, it is often the case that the owner of the data may not even have control of where the data is placed. Having the problem of the global market diverse privacy policies and protection rights, this would be a tough issue to manage. Since resource allocation and scheduling are one of the crucial trumps of the cloud's success and therefore cost reductions, the customer must settle for the offered conditions. Otherwise, he would have to take the mission of safeguarding the data in the midst of untrusted processes himself.

Another security challenge with clouds is the trustworthiness of cloud providers. In principle, cloud service providers must take care of data in accordance with the law. However, can we be completely positive that all of the employees would act accordingly? And what would happen if the cloud owner decides to change its policies and share the data with others? Moreover, it is not even clearly distinguished who is the real owner of the data stored in the cloud, whether the cloud user or the cloud provider.

Nevertheless, the cloud user must be very careful when estimating the trustworthiness of a cloud provider. In order to analyze a provider's reliability, many features must be taken into consideration, as its abilities, commitment, credibility… If the consequences that might be caused by the addressed security issues outweigh the benefits of the cloud implementation, than users would probably chose to think twice before making a decision.

2.7 Cloud storage

Ever since computers are present in our everyday lives, there has been the need for storing documents, music, photos, videos and other forms of information on external storage devices. In the beginning, data storage devices were intended for the purposes of creating

backups of data or transferring the data among computers. Among the oldest storage devices, the floppy disk would be undoubtedly the one to distinguish itself. Its scant reliability, accompanied by the evolution of technology, led to the invention of optical drives (CD-ROM, DVD-ROM, CD-R/W, DVD-R/W). These devices were theoretically capable of storing data safely for a longer period of time, i.e. 100 years as promised. However, the time estimations turned out incorrect, as in reality optical drives lasted for no longer than a couple of years in the best case. The next generation of storage devices is represented by the USB sticks, which showed no better reliability and no greater time span than its predecessors.

However, the Internet revolution opened new possibilities for the users and ever since, they have been enabled to exchange limited amounts of data via specific applications on the Web. These limitations loosen up with the burst of cloud computing. Cloud owners offer their servers for storage purposes at very favorable conditions and low prices. Great amounts of data (maybe even more than sufficient for a regular user) can be stored on third-party servers, even for free. Compared to the traditional way of storing information, cloud storage has the great advantage of enabling its users to access the database, which stores their data at any time and any place, where Internet connection can be established.

Cloud storage systems are housed in facilities called data centers. They need only one data server connected to the Internet, which the client communicates with. The client sends the data he wants to store to the data server and after the server records the data, the client is able to access it through a Web-based interface. The client can either retrieve, or work with the data on the server.

An on-demand, just-in-time access to the database that stores the client's information is enabled due to the basic concept of cloud computing, i.e. redundancy. Namely, the same data is stored on multiple machines, on servers that use different power supplies, on different geographic locations etc. In this way, the user is guaranteed access to its data even when one of the computers is being repaired, or one power supply fails.

Cloud storage providers are all over the Internet and they are more and more present each day. Among the many providers, we can single out the following: Dropbox, Google Docs, Gmail, Hotmail, Yahoo! Mail, Flickr, Picasa, Youtube, Facebook etc.

2.8 Cloud storage integrity issues

Until this point, we walked through the process of saving data to cloud storage performed and we can agree on its extreme simplicity. However comfortable the client may be with the use of it, there are some issues that need to be addressed. The fact that data doesn't physically exist within the control scope of the organization, raises serious security concerns. Ironically, the data may not exist on the premises of the organization, but on the other hand, it physically exists on multiple machines. As much as this contributes to the reliability and availability of our resources, clients must be aware of its harmful nature concerning the data's security.

For instance, a criminal may try and succeed in steeling one of the physical machines on which the data is stored. If one has the intention of modifying or even destroying some specific data, one would not chose the means to do it. Even if the physical machines are properly protected, hackers can manage to reach the data through an electronic back door. After they access the data, it will be easily subjected to any kinds of modifications.

The villain is not necessarily some accidental hacker who only wants to show the scope of his capabilities. It is the unsatisfied employee that may provoke the worst case scenario. The authenticated user name and password are in possession of the employees, therefore they are empowered to modify the data whenever they want to.

Having in mind these security issues, we may conclude that storing sensitive data on a third-party server is not the first best choice in all possible scenarios. Even when the benefits of cloud storage are highly attractive, the client must be aware of the data integrity issues and the seriousness of their consequences. In this case, it would be smart that the client builds a solution himself. For instance, the idea behind it may be to constantly check the integrity of its data and send the client the appropriate information.

3 Data integrity of a file

According to FISMA (Federal Information Security Management Act), information security is defined by the following definition:

> ‘‘The term ‘information security’ means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide—
> (A) **integrity, which means guarding against improper information modification or destruction, and includes ensuring information non repudiation and authenticity;**
> (B) confidentiality, which means preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information; and
> (C) availability, which means ensuring timely and reliable access to and use of information.”([3])

Since a computer file represents a “block of arbitrary information, or resource for storing information”([4]), it follows that the FISMA definition of information security applies to the information contained in a file, as well. Thus, in order to protect a specific file, first we need to protect the integrity, confidentiality and availability of the information contained in it.

Although all three components of the security triad are equally important, our attention focuses on the problem of preserving the integrity of information contained in a computer file. That information may be subjected to modification, addition, or even deletion and in that way may cause corruption of the file. A file’s integrity may be compromised due to hardware or software malfunctions such as a faulty storage media, an error in moving or copying the file, or malicious intrusions by hackers with the intention of altering the file’s information.

3.1 Causes of data integrity disruptions

We focus primarily on the integrity of static data contained in a remote file stored in cloud storage. However, there is no radical difference between protecting a regular file on-premises and protecting it off premises, as the security risks related to a remote file's integrity do not differ much from the integrity concerns that apply to locally stored files. Nevertheless, we must not forget the potential for added risk as the information integrity falls under the control of the cloud provider.

Several common security risks must be mentioned when addressing a file's information integrity, listed in [10]:

- hardware and software malfunctions,
- malicious attacks, phishing being the most widespread form (sometimes even caused by a malpractice of cloud provider employees' privileged access),
- a user's own slip/error.

Hardware and software malfunctions. Hardware and software malfunctions may compromise the information integrity of a file in the process of transmission from a local storage device to the cloud storage across a network, or while at rest in the cloud storage itself.

An example of hardware malfunction would be a bit-flip in reading a file system's inode bitmap that may result in serious alteration of data, i.e. cause the file system to overwrite important files.

It may happen that a software malfunction be provoked by a hardware error. However, the ever more complex structure of disks makes it almost impossible for developers of storage software to stay on track with all of the possible gaps in the hardware's design and thus, they usually fail to bypass the possible damages caused by disk errors.

Disk errors are not the only one to be blamed as the root cause for software misbehavior. Faulty device drivers may be also compromising to a file's information integrity, resulting in serious damage of data while being read from storage or asynchronously written/overwritten to a storage.

An example of a cloud provider's software malfunction would be the 2008 incident where data stored at Amazon S3 was silently corrupted. In March 2009, an unauthorized access to data occurred in Google Docs, again as a result of the provider's software malfunction ([5]).

As already mentioned, a file corruption may also happen in the process of its transmission from a local storage device to the cloud storage across an unreliable network. This would be an additional reason, why prompt checks of integrity must be performed.

Malicious attacks. Such attacks are usually performed by hackers who succeed in bypassing the security measures imposed on computer systems. After the intrusion they may choose to modify files stored on that system manually or infect them with a viral malicious program, resulting in serious corruption or even loss of data. For instance, the cloud provider's personnel may be fitted in this group of potential attackers, as they have privileged access to the data's storing machines and do not even need to search for back doors.

A malicious attack may be also performed by a network's eavesdropper who is capable of modifying information contained in a file while transmitting the file to a remote computer. By encrypting the file we create an additional security layer that impedes the eavesdropper's intentions.

Recently, a technological journalist Mat Honan happened to be the subject of a distasteful joke of anonymous hackers. The hackers first attacked the journalist's twitter account, but this was obviously not fun enough. In order to make sure that he would not be able to access it again, they managed to delete his iPhone, iPad and MacBook Pro. In order to do all of this, they only needed to reset his password. Yes, it was that simple ([6]). And that proves how unreliable and easy to access the cloud storage providers are.

A user's own slip/error. Integrity violations may also be caused by a user's inadvertence. The user must be properly acquainted with any application that accesses the file stored on a remote machine. If a user misuses the application and imprudently provokes an error action, the user itself will cause corruption of the remote file. Therefore, the data integrity may sometimes be compromised by the user itself.

The level of guaranteed information integrity in cloud storage is also dependent on the type of deployment and service model we choose to use for our purposes. On one hand, the

user's control over his data from the deployment model of a private cloud to the model of a public cloud declines, while the need for integrity assurance appreciates. On the other hand, the cloud provider's security responsibilities from the service model of *software-as-a-service* to the model of *infrastructure-as-a-service* rise. Therefore, the user must be very careful at choosing the combination of cloud deployment and service model and adjust the efficiency gains and cost benefits that combination of models provides to the level of the information sensitivity. If the combination does not appear to provide the specific desired level of information integrity the user would have to satisfy the need for integrity assurance himself, for in the lack of integrity assurance mechanisms alterations of data may go undetected, causing further loss of data.

3.2 Data integrity proof

After plainly exposing the ways in which a file's integrity in cloud storage may be threatened, we concluded that a detection mechanism of integrity violations is of crucial importance for keeping the user acquainted with any modifications made in a specific file in order to be able to prevent any further loss of data. The detection mechanism should serve the purpose of timely preventing the archiver/cloud storage provider to modify greater portions of a file without the approval of the file's owner, since it would be insufficient that data alterations are detected during a regular access to it. For this purpose, the file should be accessed on a more frequent basis with the intention of keeping the primary owner of the file confident in the file's integrity. Otherwise, serious data corruptions or even loss of data may be caused during longer time intervals, due to the malicious or unreliable nature of the provider. This kind of preventive mechanisms, keep the user informed about the trustworthiness of the cloud storage provider which should be in mutual favor under normal circumstances.

The idea behind such a detection mechanism is to implement an algorithm that will be capable of verifying the integrity of a computer file, by using frequent checks on storage archives. However, there are limitations to the provider's and client's resources. Consequently, there must be limitations to the frequency of checking a remote file's integrity, as well. Frequent accesses to an entire file stored on a third-party server can be very expensive in I/O costs to the storage provider for larger files. Another option is transmitting the file to our local storage, which is very consuming in terms of bandwidth.

Considering these difficulties, both of the scenarios should get out of option since an integrity check must be performed quickly and efficiently. If the solution should not consider accessing an entire file or retrieving it, than the only option left would be to design a solution that would perform integrity checks on the basis of processing only a portion of the observed file.

An idea for an algorithm that is supposed to access only a portion of a remote file has been proposed in Juels and Kaliski's scheme depicted in [7]. Their basic "*proof of retrievability*"(POR) protocol is founded on the concept of embedding a "set of randomly-valued check blocks called *sentinels*" ([7]). These sentinels are indistinguishable from the other file blocks, as the entire file is encrypted after the embedding. The process of verification is the following:

> "The verifier challenges the prover by specifying the positions of a collection of sentinels and asking the prover to return the associated sentinel values. If the prover has modified or deleted a substantial portion of F, then with high probability it will also have suppressed a number of sentinels. It is therefore unlikely to respond correctly to the verifier."([7])

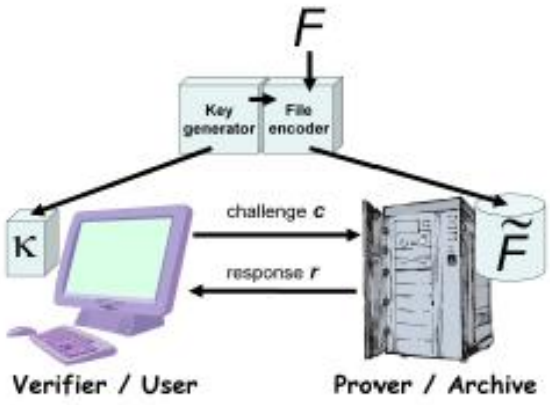

Figure 3.1 *Schematic of a POR system. An encoding algorithm transforms a raw file F into an encoded file* $\tilde{F}$ *to be stored with the prover/archive. A key generation algorithm produces a key k stored by the verifier and used in encoding. The verifier performs a challenge-response protocol with the prover to check that the verifier can retrieve F.*([7])

There is an additional step before the encryption of the file, i.e. implementation of an error-correcting code. However, it is meant to serve the purpose of protecting the file from

corruption of small portions. Since our scheme focuses only on checking the integrity of the file and not on its retrievability, this additional step is taken as irrelevant.

Juels and Kaliski's scheme does give an idea of how to resolve the problem of integrity checks' frequency, since it accesses only a small portion of the file. However, the idea cannot be accepted as the best possible solution, since every integrity check requires decryption of the entire file, which is again computationally burdensome.

This omission is taken into consideration by Kumar and Saxena's scheme in [8] which offers a more practical solution. The owner of the data first specifies the number of meta-bits (checking values) per block with respect to the size of the file. Then, the client randomly generates the positions of that many bits multiplied by the number of blocks in the processed file. These bits are referred to as meta-bits. After their positions in the file are specified, the meta-bits' values for each block are encrypted separately and the processed meta-data is appended to the end of the file. In this way, a client would not have to decrypt the whole file in the process of verification and not even a whole portion of the file determined by the number of meta-bits. The client would only have to decrypt the encrypted meta-data specified in the block whose integrity wants to check. Therefore, before sending the file to the archive the setup phase should be completed.

3.2.1 The setup phase

The setup phase consists of three steps:

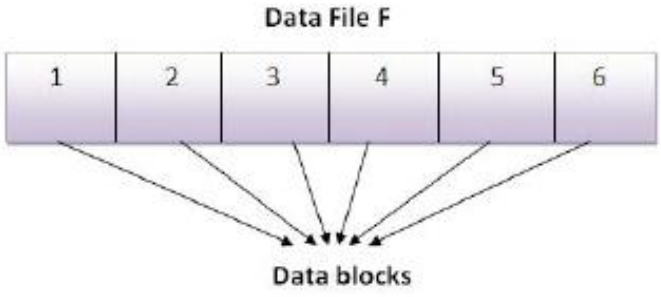

Figure 3.2 *A data file F with 6 data blocks* ([8])

1) **Generation of meta data**. If *n* is the number of bytes in a block and *k* is the number of meta-bits in a block, than *k* positions in each block should be generated to point out the meta bits in it. If *N* is the size of the file (in bytes), the meta bit positions may be specified by the following function:

$$g(i,j) -> \{1 \dots n * 8\}, i \in \{1..N/n\}, j \in \{1..k\}$$ [8]

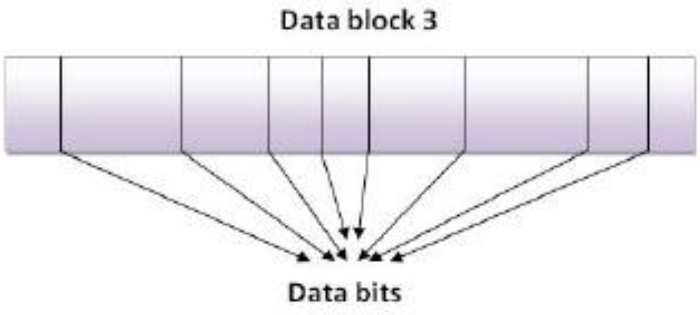

Figure 3.3 *A data block of the file F with random bits selected in it* ([8])

The portion of the file that is intended for detection purposes may be computed as a relation of the size of the block *n* (in bytes) and the number of meta bits in a block *k*, regardless of the size of the file *N (*in bytes*)*.

$$f(n,k) = \frac{k}{8 * n}$$

The positions and the values of these meta bits should be known only to the owner of the data, i.e. the verifier in the process of integrity verification.

The size of the meta data per file also determines the expansion of the file. As individual meta bits are one-time verifiable, the expansion of the file is inversely proportional to the number of verifications the detection mechanism will be able to perform. The more meta data attached to a file, the greater are the storage needs, but so is the number of possible verifications. The greater the expansion of the file, the greater is the algorithm's life cycle as well.

2) **Encrypting the meta data.** The meta data in each block is encrypted with a key which is regenerated for each block. This implies that all of the blocks are encrypted with a different key. Consequently, a new set of modified meta data is created for each block.
   The encryption may be done in a variety of ways. The scheme in [8] suggests an XOR operation on the block and the corresponding generated key. According to the referred scheme, the generation of the key for a particular block is defined by the following function:

$$h(i) -> \alpha_i, \alpha_i \in \{0 \dots 2^n\}$$

3) **Appending of meta-data**. The encrypted meta-blocks are first concatenated and then appended to the end of the file. The number of meta-bits per block must be carefully specified, as they determine the expansion of the original file. After the expansions, the setup phase is concluded and the file is ready to be sent to the archive.

Figure 3.4 *The encrypted file* $\widetilde{\boldsymbol{F}}$ *which will be stored in the cloud* ([8])

3.2.2 The *verification* phase

In the verification phase we can check the integrity of a file. The process may be repeated arbitrary number of times. As the setup phase concludes, the verification phase may begin. The verification is based on the challenge-response principle. A verifier, who wants to verify the integrity of a file, throws a challenge to the archiver/prover. The challenge consists of the position of a particular meta-bit (known only to the verifier) in the file and the position of the encrypted meta-block that corresponds to the specified meta-bit. The prover sends the encrypted meta-block and the value of the meta-bit back to the verifier. The verifier

decrypts the meta-block, reads the certain bit in it and compares it to the value of the meta-bit sent from the archiver/prover. The challenge's response is positive if these two values match. Otherwise, a mismatch would mean a loss of integrity of the client's data at the cloud storage.

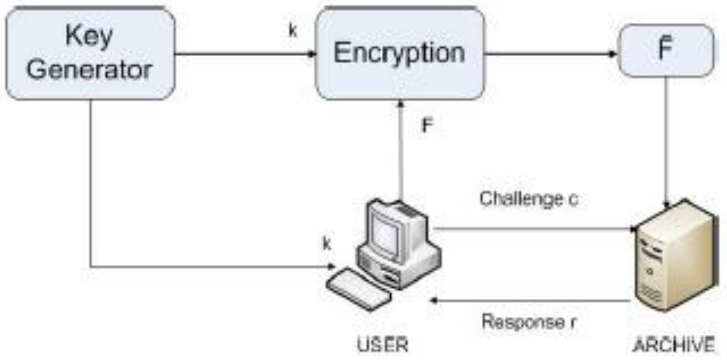

Figure 3.5 *Schematic view of a proof of retrievability based on inserting random sentinels in the data file F* ([8])

Kumar and Saxena's scheme described in [8] turns out to be very practical, comprehensible and relatively undemanding in terms of resources. According to their idea, the client should store a one-way function that generates the positions of the meta-bits in each block and a function that generates a key for each meta-block separately and encrypts that block by computing an XOR operation on the key and the meta-block. When decrypting an encrypted meta-block, the client should be able to generate the same key for the same meta-block in order to manage to decrypt it. However undemanding in terms of resources this may sound, the use of these functions evokes high security risks. If the client does not design new functions for each file stored in the cloud, these two functions will **always** have same outputs. If an intruder finds out the functions' principles, he will be able to generate the positions of the meta-bits for every file stored in the cloud storage that has been preprocessed in this way. A solution to this problem may be the use of a hash-based message authentication code (HMAC), which will provide a different function for generation of meta-data or generation of keys for encryption of blocks, by generating a new key for every file. However, there are still some security issues related to the generation and the storage of such MACs. Therefore, the implementation of the two functions mentioned earlier, still represents a great drawback to this scheme, which we attempt to improve in our implementation of the integrity verification algorithm.

In all other parts, the scheme really seems to be a good fit for the needs of individuals and companies, as well. Only a small portion of the file is encrypted, which minimizes the need for computational power at the client. The verification process is simple and computationally undemanding. The server needs to send only a small portion of the data to the client, which minimizes the network bandwidth. All of these qualities of the scheme are convincing enough to attempt its implementation.

4 Implementation and evaluation

4.1 Implementation

For our implementation we decided to use the *Java* programming language. The source code of the program is available in the Appendix. It must be noted that this decision is made for no particular computational benefits especially, but simply because the writer happens to be most familiar with it.

Two classes are implemented: DIclient and DIserver. These two classes correspond to the two roles that play the main parts in the process of the verification, i.e. the verifier (client) and the prover (server). The class DIclient contains two methods: *setup*(String fileName, **int** n, **int** k) and *verify*(String filename, **int** blockNum, **int** posInBlock, **int** n, **int** k, **int** bytesToSkip, **int** posInTable). The client's role is to prepare the file for outsourcing which means to perform the setup phase of the data integrity algorithm. During the setup phase, the verifier generates the meta data, encrypts it and appends it to the original file. The *setup* method in the DIclient class is comprised of these three steps and executes them all. Another role of the client is to verify whether a meta bit requested from the server matches the value of the bit in the corresponding meta block (also requested by the server) position. This is performed by the method *verify.* On the other hand, the server must be able to respond to the client's requests and this is performed by the two methods in the DIserver class: *metaBit*(String fileCor, **int** numBlock, **int** posInBlock, **int** n) and *metaBlock*(String fileCor, **int** bytesToSkip, **int** k). The *metaBit* method returns the value of the given bit's position(in *numBlock,* at *posInBlock*) and the *metaBlock* method returns the encrypted string of meta bits in the specified block.

The classes and their methods may be represented by the following UML diagrams.

Figure 4.1 *UML representation of the implemented classes DIclient and DIserver*

4.1.1 *Setup* phase

At first, the file to be processed is opened(*file, inStream*) and the file that will save the encrypted meta blocks is also opened(*fileEnc, outStreamEnc*). All of the additional files related to the original file that will be created in the setup phase, will be stored in a separate directory named by the original file(*dir*).

The setup phase starts with computing the total number of blocks(*numBlocks*) in the file and creating an int[][][] table that will be storing the values of the meta bits' positions in the file and the meta bits' values(*m*). After the initialization of the key variables, follows the initialization of the key generator for AES encryption.

The setup phase is consisted of three main steps: generation of meta-data, encryption of meta-data and appending of meta-data.

1) **Generation of meta data**

We initialize an iterator(*i*) that will serve to point the number of the block which is being processed at the moment. Then we start processing the blocks in the file, while there are unread bytes in the file.

The generation of meta-data is not performed by a previously designed function because of the security risks that particular approach imposes. Instead, the process of generating the meta-bits' positions is randomized using a hash set and an array list, named *numbers* and *meta,* respectively. A for loop adds to the hash set *numbers k* positions in a block, smaller than *n**8, using the *nextInt(int n)* method from the Java Random class. When the loop is completed, the set is transformed to the array list *meta*. Thus, an array list of *k* random positions in an *n**8 bit block is generated(*meta*) and this is how or implementation of the $g(i,j)$ function is realized.

As the expansion of the file, as well as the life cycle of the program are determined by the size of the meta data per file, we chose optimal values for the number of meta bits in a block (*k*=256) and the size of a block (*n*=768). The portion of the file that will be used as meta-data is the following:

$$f(n,k) = \frac{256}{8 * 768} = 0.0417 = 4.17\%$$

Considering the fact that only one meta bit per block is checked during a verification, this percentage of meta data enables that the program is used for almost one year(256 days, to be more precise), before new meta data for the file is generated. This portion of meta-data causes a 12.5% expansion of the file, which is also very acceptable.

As the positions and the values of these meta-bits should be known only to the verifier, every meta-bit in the block is read and its position and value are saved in the table *m*.

2) **Encrypting the meta data**

After the iteration through the block has been completed, we write the values of the meta bits in the block to a byte array(*newByte*). Further, an AES key is regenerated for each block separately and then saved to a file that contains the number of the block in its name. At last, the meta-block is encrypted using the AES encryption algorithm and a new set of modified meta-data is created for each block, which is appended to the end of the file *fileEnc*.

After all of the blocks have been processed, the streams *inStream* and *outStream* are closed and the number of processed blocks is saved in the attribute *numBlocks*.

3) **Appending of meta data**:

The encrypted meta-blocks are read from the file where they were saved and are simultaneously written to the file that is being prepared for storing to a remote storage.

After the expansion, a *DIclient* object is created which contains information about the meta bits, the total number of bytes and the total number of blocks in the unexpanded file. The object is returned, when the processing of a particular file has been demanded.

4.1.2 *Verification* phase

As previously mentioned, the verification is based on the challenge-response principle. A verifier throws a challenge to the archive, by calling the function *verify*. The input arguments in this function are: the name of the original file whose verification of integrity is in process(*file*), the length of a block in bytes(*n*), the number of metabits in a block(*k*), the block number of the meta bit to be verified(*blockNum*), the bit‘s exact position in that block(*posInBlock*), the number of bytes that should be skipped to get to the corresponding meta block(*bytesToSkip*) and the bit’s position in the table m(*posInTable*), which actually represents the position of the bit in the meta block. A boolean variable intended for returning the result of the function, is initialized.

The block number of the meta bit to be verified(*blockNum*) and the bit‘s exact position in that block(*posInBlock*) are sent to the prover, in order to get the value of that bit in response(*metaBit*). The response is returned by the *metaBit* method.

The number of bytes that should be skipped to get to the corresponding meta block(*bytesToSkip*) are used as input attribute when calling the prover to return the desired meta block(*metaBlockEnc*). The meta-block is returned by the *metaBlock* method.

The verifier opens the file that contains the key for decrypting the meta-block and decrypts the meta-block. Next, the verifier reads the specified bit in the meta-block and compares it to the value of the meta-bit received from the archiver/prover. If the values match, the integrity of the file is verified. If there are corrupted blocks detected, their numbers are

written to the file “incorrect.txt”. If the file size is changed, a loss of integrity of the client’s data at the cloud storage is instantly proclaimed.

In the process of verification, the prover should read and return the value of the meta-bit, which is done by the *metaBit* method in the *DIserver* class.

The prover should also read and return the value of the corresponding meta-block, which is done by the *metaBlock* method in the *DIserver* class.

In our implementation, the verification function is called for one random meta-bit in each block, which means that the function is executed that many times as there are number of blocks. Therefore, the probability of corruption detection in a file is determined by the correlation between the fraction of the number of meta-bits really corrupted and the number of meta-bits checked during the process of verification, with respect to the number of bytes really corrupted and the total number of bytes in the file.

> “If the archive deletes or modifies a substantial e-fraction of F, it will with high probability also change roughly an e-fraction of sentinels. Provided that the verifier V requests and verifies enough sentinels, V can detect whether the archive has erased or altered a substantial fraction of F. (Individual sentinels are, however, only one-time verifiable.)”([7])

Therefore, we estimate the efficiency of our implementation by the following probability distribution:

$$P(n, cBit, cByte, N) = \frac{\frac{corrupted\ meta\ bits\ detected}{meta\ bits\ checked}}{\frac{number\ of\ corrupted\ bytes}{size\ of\ the\ file(in\ bytes)}} = \frac{\frac{cBit}{\frac{N}{n}}}{\frac{cByte}{N}} = \frac{cBit * n}{cByte}$$

$$P(n, cBit, cByte) = \frac{cBit * n}{cByte}$$

cBit – corrupted meta bits detected
cByte – number of bytes corrupted

The Java program is meant to be run by the command prompt. We navigate to the "src" folder contained in the folder where our application "DataIntegrity" is saved. Then we set the path to the Java JDK programs. For instance,

```
set path=%path%;C:\Program Files\Java\jdk1.7.0_01\bin.
```

Next, we run the command that runs the javac.exe compiler and creates the class file.

```
javac DIclient.java
```

Next, we run the command that runs the Java interpreter and executes the program.

```
java DIclient setup (name of the file that should be processed)
java DIclient setup example.txt                    or

java DIclient verify (name of the file that has been processed)
java DIclient verify example.txt
```

For testing purposes, we have developed a method that corrupts a portion of a given file and stores the corrupted file under the original file's name, while saving the original file with the extension ".original". Therefore, at this point we are able to run the following command as well:

```
java DIclient corrupt (name of the file that has been processed) (number of bytes that should be corrupted)
java DIclient corrupt example.txt 20000
```

The last command actually executes the code contained in the main method. The block length is specified in the variable *n* and the number of meta-bits in a block is specified in the variable *k*.

If the first argument args[0] is "setup", then the *setup* method from the *DIclient* class is called and a *DIclient* object is returned. The verifier calls the function *setup* which performs all of the three steps required by the setup phase to process the file and returns a *DIclient* object that contains information about the meta bits(table *m*), the total number of bytes(*totalBytes*) and the total number of blocks(*numBlocks*) in the unexpanded file. The

redundant information in the table *m*, i.e. the values of the meta-bits are removed, but their positions are left. After modifying the information in table *m*, the attributes of the *DIclient* object are saved as objects in files that are placed in the archiving folder named after the original file name.

If the first argument args[0] is “verify”, the files associated with the current file, which contain information such as the positions of the meta bits, the total number of bytes in the original file and the number of blocks in it, are read. Then, *checkBits*(we have chosen 1 as a default value for this parameter) number of meta-bits in each block are chosen to be verified. After all of the blocks are processed, the verification phase is concluded.

If no additional file is created and no exception error has been thrown, the file’s integrity is verified and this is noted in the “statistics.txt” file. If a file “incorrect.txt” is created, it will contain the numbers of the detected corrupted blocks. If the exception error message “The file is corrupted. Portion of the file has been deleted." appears, it means that a portion of the file has been deleted or the file has been deleted entirely. In both cases, loss of data integrity is proclaimed.

The file “statistics.txt” contains some basic statistics about the process, as the length of the original file, the length of the expanded file, the number of bits checked during a verification, the storage memory used at the client, etc.

4.2 Evaluation

The setup phase is intended for the purpose of creating an archive on a special file *F*. Actually, two archives are created. One is stored at the client’s storage and is intended to serve the execution of the verification process. It contains the keys for the encrypted meta-blocks and other information that should be known only to the verifier, such as the positions of the meta-bits in the file. The other archive is stored at the cloud storage and represents the file itself, as well as some additional meta-data attached to serve the execution of the verification process. Hence, it would be very optimistic to state that this corruption detection mechanism will cause no computational or storage overheads. The client would still have to reserve a portion of his storage for the file, even though the file will be stored on

a remote machine. The size of the remote file will be greater than the size of the original file, as the meta-data attached to the original file causes expansion of the file. Therefore, the need for cloud storage space will be more demanding than in the case of outsourcing a regular file to a third-party server. However, these overheads must not attain the overhead level of reading and decrypting an entire file, or its transmission to the local storage.

In order to achieve valuable results, i.e. high probability corruption detection that will not provoke huge computational or storage overheads, we have chosen the appropriate optimal values for the input parameters in the corruption detection mechanism. The size of a block and the number of meta-bits in a block are the key input parameters that were set to 768B and 256 bits, respectively. These two parameters determine the rate of expansion of the original file(12.5%), as well as the level of the mechanism's reliability. The greater the portion of meta-bits in a file, the greater the number of verifications to be performed, the greater the expansion of the original file, the greater the storage overhead. The greater the portion of meta-bits checked during the verification phase, the greater the computational time required, the greater reliability assured. Shortly, this is how the input parameters influence the reliability and computational cost of the integrity verification mechanism, in practice.

During the verification phase, a set of challenges receives corresponding response about the integrity of a file. The probability that the right portion of corruption will be detected is calculated by the following distribution:

$$P(n, cBit, cByte) = \frac{cBit * n}{cByte}$$

n – size of a block (in bytes)
cBit – corrupted meta bits detected
cByte – number of bytes corrupted

The program was tested on a file of size 1084262 B. The encryption process of the meta bits is very efficient, even when tested for significantly larger files. The verification process is somewhat more demanding in terms of execution time.

The chosen parameters produced the following output in the statistics file:

Block size: 768 bytes.
Meta bits in a block: 256 bits.
Portion of meta bits with respect to the original file size: 4,17%.
The length of the original file was: 1084262 bytes.
The length of the expanded file is: 1219814 bytes.
The rate of the expansion is: 12,5 %.
Storage memory used at the client: 22592 bytes.

1412 bits checked, which is 0,02% of the original file size.
The integrity of the file is verified.

1000 bytes are corrupted, or approximately 1-2 blocks. Approximate portion of the file corrupted: 0,09%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 1
Portion of corrupted bits detected with respect to the number of bits checked: 0,07%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 1
Portion of corrupted bits detected with respect to the number of bits checked: 0,07%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 1
Portion of corrupted bits detected with respect to the number of bits checked: 0,07%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 1
Portion of corrupted bits detected with respect to the number of bits checked: 0,07%.

1412 bits checked, which is 0,02% of the original file size.

Number of corrupted bits detected: 2

Portion of corrupted bits detected with respect to the number of bits checked: 0,14%.

$$P(n, cBit, cByte) = \mathbf{768} * \frac{\frac{(\mathbf{1}+\mathbf{1}+\mathbf{1}+\mathbf{1}+\mathbf{2})}{\mathbf{5}}}{\mathbf{1000}} = \mathbf{92,16\%}$$ **of the corruption detected**

20000 bytes are corrupted, or approximately 26-27 blocks. Approximate portion of the file corrupted: 1,84%.

1412 bits checked, which is 0,02% of the original file size.

Number of corrupted bits detected: 26

Portion of corrupted bits detected with respect to the number of bits checked: 1,84%.

1412 bits checked, which is 0,02% of the original file size.

Number of corrupted bits detected: 26

Portion of corrupted bits detected with respect to the number of bits checked: 1,84%.

1412 bits checked, which is 0,02% of the original file size.

Number of corrupted bits detected: 26

Portion of corrupted bits detected with respect to the number of bits checked: 1,84%.

1412 bits checked, which is 0,02% of the original file size.

Number of corrupted bits detected: 26

Portion of corrupted bits detected with respect to the number of bits checked: 1,84%.

1412 bits checked, which is 0,02% of the original file size.

Number of corrupted bits detected: 26

Portion of corrupted bits detected with respect to the number of bits checked: 1,84%.

$P(n, cBit, cByte) = 768 * \frac{\frac{(26+26+26+26+26)}{5}}{20000} = 99,84\%$ **of the corruption detected**

100000 bytes are corrupted, or approximately 130-131 blocks. Approximate portion of the file corrupted: 9,22%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 130
Portion of corrupted bits detected with respect to the number of bits checked: 9,21%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 130
Portion of corrupted bits detected with respect to the number of bits checked: 9,21%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 130
Portion of corrupted bits detected with respect to the number of bits checked: 9,21%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 130
Portion of corrupted bits detected with respect to the number of bits checked: 9,21%.

$P(n, cBit, cByte) = 768 * \frac{\frac{(130+130+130+130+130)}{5}}{100000} = 99,84\%$ **of the corruption detected**

600000 bytes are corrupted, or approximately 781-782 blocks. Approximate portion of the file corrupted: 55,34%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 782
Portion of corrupted bits detected with respect to the number of bits checked: 55,38%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 781
Portion of corrupted bits detected with respect to the number of bits checked: 55,31%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 781
Portion of corrupted bits detected with respect to the number of bits checked: 55,31%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 781
Portion of corrupted bits detected with respect to the number of bits checked: 55,31%.

1412 bits checked, which is 0,02% of the original file size.
Number of corrupted bits detected: 781
Portion of corrupted bits detected with respect to the number of bits checked: 55,31%.

$$P(n, cBit, cByte) = 768 * \frac{\frac{(782+781+781+781+781)}{5}}{600000} = 99,99\%$$ **of the corruption detected**

$$P(n, cBit, cByte) = (92,16 + 99,84 + 99,84 + 99,99)/4 = 97,96\%$$

During the testing of the program, corruption was detected in 100% of the attempts. It must be noted however, that the program was tested for corruptions greater than 0.09% of the file, i.e. greater than one block. According to the probability distribution $P(n, cBit, cByte)$, we calculated that the average probability of corruption detection is 97,96%.

With block size is 768 B, portion of meta bits with respect to the original file size of 4,17%, which provokes file expansion of 12,5% and takes up storage memory at the client of 22592B

bytes(2.08% of the original file), our implementation of the integrity verification algorithm represents an acceptable and solid integrity proof, at low file expansion and client memory usage, i.e. communication and computational costs.

5 Conclusion and future work

In this thesis, we have studied the problem of loss of integrity of data stored on third-party servers. For that purpose, we have developed an algorithm that checks whether a file has been corrupted, or its integrity is preserved, on the basis of the knowledge of only a portion of the file. Thus, by accessing a small portion of a remote file, the program will detect what portion of the file has really been corrupted, with high probability.

A drawback to the solution we studied in this paper would be its static nature. The solution only applies to files that have a non-changing content, at least, for as long as the duration of the detection mechanism's life cycle. A potential starting point for a deeper analysis on the cloud storage security issue, may be the study of an algorithm that will also apply to files with dynamic content.

Another potential idea for further development of the studied solution, may be the development of an integrity verification solution that will not be limited by a predetermined number of executions. In our solution, the number of verifications the program will be able to perform is determined by the portion of the file we choose to use as meta-data. As explained in [7], meta-bits are only one-time verifiable. When we request the value of a particular bit, the archive may remember its position and be careful in the future not to change its value. Hence, this approach enables us to perform only a limited number of integrity verifications, until all of the meta-bits are verified. Therefore, it would be a good idea to develop a solution that will not have this kind of limitations.

In conclusion, although the solution studied in this paper has its disadvantages that may be the grounds for future studies on this topic, a solid, quality solution, offering very reliable results has been provided. The implemented solution may be a good fit for individuals or organizations, not only because of the simplicity of its use, but also because of the many advantages we mentioned several times throughout this paper, as low computational,

storage and network bandwidth requirements. The solution's integration into system software and its transparency to the end user may also be considered as an additional future challenge.

Appendix

Source code

DIclient.java

```
import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;

import javax.xml.bind.DatatypeConverter;

/**
   * This program generates a AES key, retrieves its raw bytes, and
   * then reinstantiates a AES key from the key bytes.
   * The reinstantiated key is used to initialize a AES cipher for
   * encryption and decryption.
   */

   public class DIclient {

     /**
     * Turns array of bytes into string
     *
     * @param buf  Array of bytes to convert to hex string
     * @return     Generated hex string
     */


         private int[][][][] m;
         private int totalByte;
         private int numBlocks;


         public DIclient(int[][][][] m, int totalByte, int numBlocks) {
                  this.m = m;
                  this.totalByte = totalByte;
                  this.numBlocks = numBlocks;
         }


         public static boolean verify
                  (String filename, int blockNum, int posInBlock,
                                    int n, int k, int bytesToSkip, int posInTable)
                                             throws IOException, NoSuchAlgorithmException, NoSuchPaddingException,
                                             InvalidKeyException, IllegalBlockSizeException, BadPaddingException {
```

```
		boolean res = false;

		int metaBit = DIserver.metaBit(filename, blockNum, posInBlock, n);
		String metaBlockEnc = DIserver.metaBlock(filename, bytesToSkip, k);

		FileInputStream keyInFile;
		String keyFile = "../" + filename + "_local/key" + blockNum + ".txt";
		try{
			keyInFile = new FileInputStream(keyFile);
			byte[] secretKeyBytes = new byte[16];
			keyInFile.read(secretKeyBytes);
			SecretKey  secretKey = new SecretKeySpec(secretKeyBytes, "AES");

			keyInFile.close();
			try{
				Cipher cipher = Cipher.getInstance("AES");
				cipher.init(Cipher.DECRYPT_MODE, secretKey);
				byte[] b = DatatypeConverter.parseHexBinary(metaBlockEnc);
				byte[] original = cipher.doFinal(b);

				byte metaB = original[posInTable/8];
				int bit = posInTable%8;
				int c = metaB & (1 << bit);
				if(c != 0){
					c=1;
				}
				if (metaBit == c){
					res = true;
				}

				if (res==false){
					FileWriter fwr = new FileWriter(new File(new
					File(System.getProperty("user.dir")).getParent() + "/" + filename +
					"_local", "incorrect.txt"), true);
					fwr.append(blockNum + ", ");
					fwr.close();
				}

			}
			catch(Exception e){
				System.out.println("The file is corrupted. Portion of the file has been
				deleted.");
				System.exit(0);
			}
		}
		catch(Exception e){
			System.out.println("The name of the file is not correct.");
			System.exit(0);
		}

		return res;
	}


	public static DIclient setup(String fileName, int n, int k)
		throws IOException, NoSuchAlgorithmException, NoSuchPaddingException,
			InvalidKeyException, IllegalBlockSizeException, BadPaddingException{

		File file = new File(new File(System.getProperty("user.dir")).getParent(), fileName);
		FileInputStream inStream = new FileInputStream(file);
		File dir = new File(new File(System.getProperty("user.dir")).getParent(), fileName + "_local");
		//We delete the directory if it exists.
		if (dir.exists()){
		    if (dir.isDirectory()) {
		        File[] children = dir.listFiles();
		        for (int i=0; i<children.length; i++) {
		            children[i].delete();
		        }
		    }
		}
		//We create a directory that will archive the files that
		//will serve the execution of the verification process.
		dir.mkdir();
		File fileEnc = new File(new File(System.getProperty("user.dir")).getParent(), fileName + "_local/"
		+ "out");
		FileWriter outStreamEnc = new FileWriter(fileEnc, true);

		File f = new File(new File(System.getProperty("user.dir")).getParent(), fileName +
		"_local/statistics.txt");
		FileWriter stat = new FileWriter(f);
		stat.write("Block size: " + n + " bytes." + System.getProperty( "line.separator" ));
		stat.write("Meta bits in a block: " + k + " bits." + System.getProperty( "line.separator" ));
		stat.flush();
```

```
long totalByte = inStream.available();
//total number of blocks in the file
int numBlocks = (int)totalByte/n;
if ((int)totalByte % n != 0){
        numBlocks = numBlocks+1;
}
int[][][] m = new int[numBlocks][2][k];

DecimalFormat df = new DecimalFormat("#.##");
stat.write("Portion of meta bits with respect to the original file size: "
+ df.format((double)(numBlocks*k*100)/(double)(inStream.available()*8))
+ "%." + System.getProperty( "line.separator" ));
stat.flush();

// Get the KeyGenerator
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);

//Iterates through all blocks of the file.
int i = 0;
while (inStream.available() > 0){

//The g function: generates a set of k bit positions for each data block

Set<Integer> numbers = new HashSet<Integer>();

//There are n*8 bits in a block.
//We make a set of k positions in the block, smaller than n*8.
while(numbers.size() < k){
        Random r = new Random();
        int num = r.nextInt(n*8);
        numbers.add(num);
}

ArrayList<Integer> meta = new ArrayList<Integer>(numbers);

//The k random positions are sorted again,
//so that they can be checked by order.
Collections.sort(meta);

//Iterates through all meta bits in the current block.
int count = 0;
inStream.skip(meta.get(count)/8);
//While the number of the current meta bit in the block
//is not exceeding the number of meta bits in a block
//and there are still available bytes to read, do:
while (count < k && inStream.available() > 0){
        byte b = (byte) inStream.read();
        //the position of the meta bit in the i-th block
        m[i][0][count] = meta.get(count);
        //the value of that bit
        int pos = meta.get(count)%8;
        if ((b & (1 << pos)) == 0){
                m[i][1][count] = 0;
        }
        else{
                m[i][1][count] = 1;
        }
        //In case more bits are in the same byte
        while(count < k-1 && meta.get(count+1)/8 == meta.get(count)/8){
                count = count + 1;
                //the position of the meta bit in the i-th block
                m[i][0][count] = meta.get(count);
                //the value of that bit
                pos = meta.get(count)%8;
                if ((b & (1 << pos)) == 0){
                        m[i][1][count] = 0;
                }
                else{
                        m[i][1][count] = 1;
                }
        }
        count++;
        if (count < k){
                inStream.skip((int)(meta.get(count)/8 - meta.get(count-1)/8 - 1));
                }
                else{
                        inStream.skip((int)(n - meta.get(count-1)/8 - 1));
                }
        }

        //If the number of the current meta bit in the block
```

```
                    //exceeds the number of meta bits in a block
                    //or there are no more bytes to read:
                    if (count >= k || inStream.available() <= 0) {
                            //We write the values of all meta bits to a byte array, that is later //encoded.
                            byte[] newByte = new byte[k/8];
                            for(int j = 0; j < m[i][1].length;){
                                    for(int pos = 0; pos < k/8; pos++){
                                            byte b = 0;
                                            for(int bit = 0; bit < 8; bit++){
                                                    if(m[i][1][j] == 1){
                                                            b |= 1 << bit;
                                                    }
                                                    else if (m[i][1][j] == 0){
                                                            b &= ~(1 << bit);
                                                    }
                                                    j++;
                                            }
                                            newByte[pos] = b;
                                    }
                            }

                            SecretKey skey = kgen.generateKey();
                            byte[] raw = skey.getEncoded();
                            String keyFile =  "../" + fileName + "_local/key" + i + ".txt";
                            FileOutputStream keyOutFile;
                            try {
                                    //Write the generated key for the meta bits in the i-th block //to a
                                    //file.
                                    keyOutFile = new FileOutputStream(keyFile);
                                    keyOutFile.write(raw);
                                    keyOutFile.close();
                            } catch (FileNotFoundException e) {
                                    // TODO Auto-generated catch block
                                    e.printStackTrace();
                            }
                            SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");

                            Cipher cipher = Cipher.getInstance("AES");
                            cipher.init(Cipher.ENCRYPT_MODE, skeySpec);

                            byte[] encrypted = cipher.doFinal(newByte);
                            // Convert encrypted bytes to hex.
                            String bytesAsString = DatatypeConverter.printHexBinary(encrypted);
                            //Write that hex value to a file.
                            outStreamEnc.write(bytesAsString);
                            outStreamEnc.flush();

                            //If the value of totalByte is not a multiple of n,
                            //change the value of n(number of bytes in the block)
                            //for the iteration through the last block.
                            if (i==numBlocks-2){
                                    n = (int)totalByte % n;
                            }
                            i++;

                    }

            }
            inStream.close();
            outStreamEnc.close();
            numBlocks = i;
            FileOutputStream outStream = new FileOutputStream(file, true);
            FileInputStream inStreamEnc = new FileInputStream(fileEnc);
            int bytesEnc = (int) fileEnc.length();
            byte[] bytes = new byte[k/8];
            //Concatenate the original file with the file containing
            //the encrypted meta bits for each block separately.
            while (inStreamEnc.available() > 0){
                    inStreamEnc.read(bytes);
                    outStream.write(bytes);
                    outStream.flush();
            }
            outStream.close();
            inStreamEnc.close();
            fileEnc.delete();
            System.out.println("More detailed statistics information about the encryption are contaied in the
            file with path: " + f.getAbsolutePath());
    stat.write("The length of the original file was: " + totalByte + " bytes." + System.getProperty(
    "line.separator" ));
    stat.write("The length of the expanded file is: " + (int)(totalByte + bytesEnc) + " bytes." +
    System.getProperty( "line.separator" ));
    stat.write("The rate of the expansion is: " + df.format(100*((double)(totalByte +
    bytesEnc)/(double)totalByte - 1)) + " %." + System.getProperty( "line.separator" ));
```

```
        stat.close();

        DIclient client = new DIclient(m, (int)totalByte, numBlocks);
        return client;
}

public static void main(String[] args) throws Exception {

        int n = 768;//block size in bytes(B) 128B=1024bits  768B=6144bits
        int k = 256;//number of meta bits in a block

        if (args[0].equals("setup")){
                DIclient client = setup(args[1], n, k);
                int[][] m = new int[client.numBlocks][client.m[0][0].length]; //m[block][position of the bit in the
                                                                                                     table]
                                                                              //- position of the bit in the block
                for (int i=0; i<client.numBlocks; i++){
                        for (int j=0; j<client.m[i][0].length; j++){
                                m[i][j] = client.m[i][0][j];
                        }
                }
                new ObjectOutputStream(new FileOutputStream(new File(new
                File(System.getProperty("user.dir")).getParent() + "/" + args[1] + "_local", "m"))).writeObject(m);
                new ObjectOutputStream(new FileOutputStream(new File(new
                File(System.getProperty("user.dir")).getParent() + "/" + args[1] + "_local",
                "totalByte"))).writeObject(client.totalByte);
                new ObjectOutputStream(new FileOutputStream(new File(new
                File(System.getProperty("user.dir")).getParent() + "/" + args[1] + "_local",
                "numBlocks"))).writeObject(client.numBlocks);

                FileWriter fwr = new FileWriter(new File(new File(System.getProperty("user.dir")).getParent() + "/"
                + args[1] + "_local", "statistics.txt"), true);
                fwr.append("Storage memory used at the client: " + (int)(new File(new
                File(System.getProperty("user.dir")).getParent() + args[1] + "_local", "m").length()
                + new File(new File(System.getProperty("user.dir")).getParent() + args[1] + "_local",
                "totalByte").length() + new File(new File(System.getProperty("user.dir")).getParent() + args[1] +
                "_local", "numBlocks").length()
                + 16*client.numBlocks) + " bytes.");
                fwr.close();
        }


        else if (args[0].equals("verify")){
                int numBlocks;
                int[][] m = null;
                int totalByte;
                int checkBits = 1;
                List<Integer> list = new ArrayList<Integer>();
                File f = new File(new File(System.getProperty("user.dir")).getParent() + args[1] + "_local",
                "incorrect.txt");
                if (f.exists())             f.delete();

                m = (int[][]) new ObjectInputStream(new FileInputStream(new File(new
                File(System.getProperty("user.dir")).getParent() + "/" + args[1] + "_local", "m"))).readObject();
                totalByte = (int) new ObjectInputStream(new FileInputStream(new File(new
                File(System.getProperty("user.dir")).getParent() + "/" +  args[1] + "_local",
                "totalByte"))).readObject();
                numBlocks = (int) new ObjectInputStream(new FileInputStream(new File(new
                File(System.getProperty("user.dir")).getParent() + "/" +  args[1] + "_local",
                "numBlocks"))).readObject();

                        for (int i=0; i<numBlocks; i++){

                                ArrayList<Integer> numbers = new ArrayList<Integer>();

                                for(int s = 0; s < m[i].length; s++){
                                        numbers.add(s);
                                }
                                Collections.shuffle(numbers);

                                for(int t = 0; t < checkBits; t++){
                                        int bytesToSkip = (int) (totalByte + i*(k/4 + 32));
                                        boolean res = verify(args[1], i, m[i][numbers.get(t)], n, k,
                                        bytesToSkip, numbers.get(t));
                                        if (res==false) list.add(i);
                                }
                        }


                        File fstat = new File(new File(System.getProperty("user.dir")).getParent() + "/" +
                        args[1] + "_local", "statistics.txt");
                        FileWriter fwr = new FileWriter(fstat, true);
                        DecimalFormat df = new DecimalFormat("#.##");
```

```
                        fwr.append(System.getProperty( "line.separator") + System.getProperty( "line.separator")
                        + numBlocks*checkBits + " bits checked, which is " +
                        df.format((double)(numBlocks*checkBits*100)/(double)(totalByte*8)) +"% of the original
                        file size." + System.getProperty( "line.separator" ));

                        if( list.size() > 0){
                                fwr.append("Number of corrupted bits detected: " + list.size() +
                                System.getProperty( "line.separator" ));
                                fwr.append("Portion of corrupted bits detected with respect to the number of
                                bits checked: " +
                                df.format((double)(list.size()*100)/(double)(numBlocks*checkBits)) + "%." +
                                System.getProperty( "line.separator" ));
                        }
                        else{
                                fwr.append("The integrity of the file is verified.");
                        }
                        fwr.close();
                        System.out.println("More detailed statistics information about the verification process
                        are contaied in the file with path: " + fstat.getAbsolutePath());
                }

                if (args[0].equals("corrupt")){

                        File in = new File(new File(System.getProperty("user.dir")).getParent(), args[1]);
                        File renameIn = new File(new File(System.getProperty("user.dir")).getParent(), args[1] +
                        ".original");
                        if(renameIn.exists()){
                                renameIn.delete();
                                renameIn = new File(new File(System.getProperty("user.dir")).getParent(),
                                args[1] + ".original");
                        }
                        in.renameTo(renameIn);
                        FileInputStream inStream = new FileInputStream(renameIn);

                        File out = new File(new File(System.getProperty("user.dir")).getParent(), "out");
                        FileOutputStream outStream = new FileOutputStream(out);
                        byte[] bytes = new byte[inStream.available()];
                        while (inStream.available() > 0){
                                inStream.read(bytes);
                                outStream.write(bytes);
                                outStream.flush();
                        }
                        outStream.close();
                        inStream.close();
                        RandomAccessFile file = new RandomAccessFile(out, "rw");
                        int bytesToCorrupt = Integer.parseInt(args[2]);
                        for(int i = 0; i < bytesToCorrupt; i++){
                                long offset = file.getFilePointer();
                                byte b = (byte) file.read();
                                for (int j = 0; j < 8; j++){
                                        b ^= 1 << j;
                                }
                                file.seek(offset);
                                file.write(b);
                        }
                        file.close();
                        File fstat = new File(new File(System.getProperty("user.dir")).getParent() + "/" +
                        args[1] + "_local", "statistics.txt");
                        FileWriter stat = new FileWriter(fstat, true);
                        int blocksCor = (int)bytesToCorrupt/n + 1;
                        stat.write(System.getProperty( "line.separator" ) + System.getProperty( "line.separator"
                        ) + bytesToCorrupt + " bytes are corrupted, or approximately " + bytesToCorrupt/n + "-" +
                        blocksCor + " blocks.");
                        int totalByte = (int) new ObjectInputStream(new FileInputStream(new File(new
                        File(System.getProperty("user.dir")).getParent() + "/" + args[1] + "_local",
                        "totalByte"))).readObject();
                        DecimalFormat df = new DecimalFormat("#.##");
                        stat.write("Approximate portion of the file corrupted: " +
                        df.format((double)bytesToCorrupt*100/(double)totalByte) + "%.");
                        stat.close();
                        System.out.println("More detailed statistics information about the corruption of the file
                        are contaied in the file with path: " + fstat.getAbsolutePath());
                        in.delete();
                        out.renameTo(new File(new File(System.getProperty("user.dir")).getParent(), args[1]));
                }
        }
}
```

DIserver.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;

public class DIserver {


	public static void main(String[] args) {

	}

	public static int metaBit(String filename, int numBlock, int posInBlock, int n) throws IOException{

		File file = new File("../" + filename);
		FileInputStream inStream = new FileInputStream(file);

		inStream.skip((int)(numBlock*n + posInBlock/8));
		byte b = (byte) inStream.read();
		int pos = posInBlock%8;
		int c = b & (1 << pos);
		if(c != 0){
			c=1;
		}

		inStream.close();
		return c;
	}

	public static String metaBlock(String filename, int bytesToSkip, int k) throws IOException{

		File file = new File("../" + filename);
		FileReader ir = new FileReader(file);
		ir.skip(bytesToSkip);
		byte data[] = new byte[k/4 + 32];
		for (int i=0; i < (k/4 + 32); i++){
			data[i] = (byte) ir.read();
		}
		String enc = new String(data);
		ir.close();
		return enc;
	}

}
```

Figures

Bibliography

[1] Winkler, V. (2011). *Securing the Cloud.* 225 Wyman Street, Waltham, MA 02451, USA: Elsevier Inc.

[2] Orlando, D. (2011 , January 25). *Cloud computing service models, Part 1: Infrastructure as a Service*. Retrieved August 15, 2012 from IBM - United States: http://www.ibm.com/developerworks/cloud/library/cl-cloudservices1iaas/

[3] (*csrc.nist.gov/drivers/documents/FISMA-final.pdf*. (n.d.). Retrieved August 29, 2012, from NIST.gov - Computer Security Division - Computer Security Resource Center: http://csrc.nist.gov/drivers/documents/FISMA-final.pdf

[4] *Computer File - Wikipedia, the free encyclopedia*. (2012). Retrieved August 28, 2012, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Computer_file

[5] Cachin, C., Keidar, I., & Alexander, S. (2009, June). Trusting the Cloud.

[6] Klančar, M. (2012, September). *Mnenja - Za varnost poskrbimo sami!* Retrieved September 10, 2012, from Monitor - revija za računalništvo: http://www.monitor.si/clanek/za-varnost-poskrbimo-sami/

[7] Juels, A., & Kaliski Jr., B. S. (2007). Pors: proofs of retrievability for large files. *Proceedings of the 14th ACM conference on Computer and communications security* (pp. 584-597). New York, NY, USA: ACM.

[8] Sravan Kumar, R., & Ashutosh, S. (2011). Data Integrity Proofs in Cloud Storage. *Third International Conference on Communication Systems and Networks (COMSNETS).* Bangalore.

[9] *http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf*. (2011,September). Retrieved September 1, 2012, from NIST.gov - Computer Security Division - Computer Security Resource Center: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[10] Sivathanu, G., Wright, C. P., & Zadok, E. (2005). Ensuring Data Integrity in Storage: Techniques and Applications. *Proceedings of the first ACM workshop on Storage security and survivability.*

[11] *The 4 Primary Cloud Deployment Models | CloudTweaks*. (2012, July 2). Retrieved 31 August, 2012, from CloudTweaks.com - Cloud Computing Community: http://www.cloudtweaks.com/2012/07/the-4-primary-cloud-deployment-models/

[12] *http://www.dialogic.com/solutions/cloud-communications/build/~/media/products/docs/whitepapers/12023-cloud-computing-wp.pdf*. (2010, July). Retrieved August 2012, from Bandwidth Optimization | Session Border Controllers | Softswitch | Dialogic: http://www.dialogic.com/solutions/cloud-communications/build/~/media/products/docs/whitepapers/12023-cloud-computing-wp.pdf

[13] *What is Cloud Computing?* (n.d.). Retrieved August 2012, from Princeton University - Welcome: http://www.princeton.edu/~ddix/cloud-computing.html

[14] *Internet Storage - Wikipedia, the free encyclopedia*. (2012). Retrieved August 2012, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Cloud_storage

[15] Meece, M. (2012, May 16). *A Computer User's Guide to Cloud Storage - NYTimes.com*. Retrieved August 28, 2012, from The New York Times - Breaking News, World News & Multimedia: http://www.nytimes.com/2012/05/17/technology/personaltech/a-computer-users-guide-to-cloud-storage.html?_r=1&pagewanted=all

[16] Strickland, J. (2011). *HowStuffWorks "Examples of Cloud Storage"*. Retrieved August 29, 2012, from HowStuffWorks "Computer": http://computer.howstuffworks.com/cloud-computing/cloud-storage2.htm

[17] Terrence V. Lillard, C. P. (2010). *Digital Forensics for Network, Internet, and Cloud Computing.* Burlington: Elsevier Inc.