NTNU – Trondheim
Norwegian University of
Science and Technology

Cloud Security without Trust

Ole Rasmus Gilberg

Master of Science in Communication Technology
Submission date: June 2014
Supervisor: Colin Alexander Boyd, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Cloud Security without Trust

Student: Ole Rasmus Gilberg

Problem description:

Businesses and individuals are increasingly storing important and private data on servers in the cloud outside their control. This has compelling benefits in terms of convenience and cost but introduces security issues which do not exist for local storage. Some of these issues arise because customers may be unwilling or unable to trust cloud providers. This project will focus on the issue of ensuring that data is stored in the manner agreed between the cloud server and the client.

In the last few years there has been considerable research effort in developing cryptographic schemes to allow users to efficiently check that data is stored correctly in a remote server. Schemes also exist for more exotic forms of checking, for example to show that data is stored redundantly on more than one disk, to show that data is stored in a specific geographical area, or to show that data is stored in encrypted form. Some of these schemes have been tested empirically but others have not. Some have restrictions on applicability, for example whether they apply to dynamic data or whether they allow data to be processed by the server. This project will examine what has been achieved in the research literature so far with focus on dynamic schemes. After that the project will continue with a theoretical and possibly a practical investigation where we will consider useful ways of extending the schemes and compare them using real-world cloud services.

Responsible professor: Colin Boyd, ITEM

Supervisor: Colin Boyd, ITEM

Abstract

The usage of cloud services is increasing for each day. This applies to private persons which store pictures and documents, as well as bigger corporations whom outsource parts of, or all, handling of their ICT infrastructure to cloud providers. Despite the continuous increase in application, there are still substantial security concerns among current and potential cloud users. Much of the concerns are due to lack of transparency to how the cloud providers maintain and process the user data. Motivated by this, multiple cryptographic schemes has been proposed to provide users with confidence that their data are maintained as agreed upon, without the necessity of changing the architecture of the cloud provider.

In this master thesis we have studied some of these cryptographic schemes, and performed a practical and economical analysis on one of them, the hourglass scheme. The hourglass scheme utilizes economical incentives to provide the cloud provider with reasons to act as agreed upon. Through implementation of hourglass functionality we identified that the validity of the scheme is dependent on resource pricing by the cloud provider, together with the actual implementation.

Based on the hourglass scheme and observations while studying the different schemes, we propose a new cryptographic scheme applying to deletion of data in the cloud. Remote deletion is a challenging task to prove, but we argue that our approach will deliver the user of a cloud service comfort that the actual data in the cloud has been deleted by the cloud provider.

Sammendrag

Bruken av skytjenester øker for hver dag. Dette gjelder alt fra privatpersoner som lagrer bilder og dokumenter, til større organisasjoner som outsourcer deler av eller hele håndtering av IKT infrastruktur til en skyleverandør. På tross av en stadig økning i anvendelse, er det fortsatt vesentlige sikkerhetsbekymringer blandt både eksisterende og potensielle brukere. Bekymringene grunner mye i mangel på innsikt i hvordan levrnadørene av skytjenester oppbevarer og behandler brukerenes data. Med dette som motivasjon, har flere kryptografiske systemer blitt foreslått for å gi brukere trygghet i at deres data blir oppbevart i henhold til avtale, uten å måtte forandre på skytjenestetilbyderenes struktur og arkitektur.

I denne masteroppgaven har vi utforsket noen av disse systemene, og utført en praktisk og økonomisk analyse av et av dem, kalt for timeglassløsningen. Timeglassløsningen benytter seg av økonomiske incentiver for å gi skytilbyderen grunn til å opptre som avtalt. Gjennom en implementajon av timeglassfunksjonalitet fant vi ut at dette systemets anvendelsesgrad er avhengig av resursprising hos skytilbyderen, samt selve implementasjonen.

Basert på timeglasssystemet og observasjoner gjort under studie av forskjellige kryptografiske systemer, foreslår vi et nytt kryptografisk system som henvender seg mot sletting av data i skyen. Avstandsbasert sletting av data er en utfordrende oppgave å begi seg ut på, men vi mener likevel at vi har klart å foreslå en metode som kan gi brukeren av en skytjenste tiltro til at det faktisk er ønsket data som er blitt slettet hos skytilbyderen.

Preface

This paper serves as a master thesis in the 10^{th} semester of my Master of Science degree in Communication Technology at the Norwegian University of Science and Technology.

I would like to thank my professor Colin Boyd for much valued guidance and discussions throughout the work. I would also like to thank postdoc George Petrides for his helpful inputs and ideas.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**
1.1 Outcome . . . 1
1.2 Report outline . . . 2

2 Background **3**
2.1 Cloud Computing . . . 3
2.2 Cryptographic schemes related to cloud computing . . . 5
2.2.1 Data possession and retrievability . . . 6
2.2.2 Proving distributed storage . . . 10
2.2.3 Deletion . . . 11
2.3 Hourglass scheme . . . 13
2.3.1 Challenges and solution . . . 14
2.3.2 Hourglass functions . . . 18
2.3.3 Security analysis . . . 21
2.3.4 Comparison of hourglass functions . . . 23
2.3.5 Economic incentives and arguments . . . 25

3 Practical analysis **27**
3.1 Implementation . . . 27
3.1.1 Original Hourglass paper implementation . . . 27
3.1.2 Our implementation . . . 28
3.2 Comparison of paper results and practical work . . . 29
3.3 Economical perspective . . . 32

4 Proposed Scheme **37**
4.1 Encoding and integrity checks . . . 37
4.1.1 Encoding - Random number generator . . . 38
4.1.2 Integrity checks . . . 39
4.2 The protocol . . . 41

4.2.1 Executing the overwrite - encoding (phase 1) 41
4.2.2 Encapsulate overwrite - hourglass (phase 2) 42
4.2.3 Prove the overwrite - challenge-response (phase 3) 43
4.3 Choice of hourglass function 44
4.4 Alternative to providing a seed to the cloud provider 44
4.5 Economic arguments 45

5 Practical challenges **47**
5.1 Hardware issues 47
5.1.1 Acquiring hardware information 47
5.1.2 How will SSDs affect the hourglass function? 48
5.2 Storage allocation 48
5.3 Alternative deletion approach 49
5.4 The price development of cloud storage and processing 50

6 Conclusion **53**
6.1 Future work 54

References **55**

Appendices

A Performance results **59**

B Source code **61**
B.1 Butterfly source code 61
B.1.1 Butterfly encapsulation 61
B.1.2 Butterfly decapsulation 63
B.2 Permutation source code 65
B.2.1 Permutations encapsulation and decapsulation 65

List of Figures

2.1 *Protocol for provable data possession, as presented in [1]* 7
2.2 *File transformation in POR before transfer to server* 9
2.3 Generic hourglass protocol as presented in [2] 15
2.4 *Naive approach* 18
2.5 *Butterfly algorithm as presented in [2]* 19
2.6 *Permutation function, inspired by [3]* 20
2.7 *RSA function, inspired by [3]* 21
2.8 *Overhead data s' for butterfly function* 24
2.9 *Overhead data s' permutation function* 24

3.1 *Operation w, mixing of two blocks* 29
3.2 *Performance results from hourglass paper (taken from [2])* 30
3.3 *Performance results of our butterfly implementation* 31
3.4 *Performance results of our permutation implementation* 32

4.1 *Merkle tree structure* 41
4.2 Deletion protocol 42

5.1 *Double pass overwrite* 50

List of Tables

2.1 Comparison of hourglass functions 25

3.1 Amazon EC2 instance prices 33
3.2 Hourglass performance and storage cost as presented in [2] 34
3.3 Updated hourglass performance and storage cost with current prices . . 35
3.4 Calculated hourglass performance and storage cost for the implementation in this thesis in Amazon environments 36

Chapter 1
Introduction

Adoption to and usage of cloud services are constantly increasing. If it is by private individuals, corporations or public services, increasing amounts of information are stored and processed in the cloud.

When transforming old systems to function in cloud environments, new security and trust issues arise. Compared with traditional systems, where individuals control their own hardware, the cloud provider often has sole control over the components when in cloud environment. Providing little transparency to the tenant about what is going on under the hood, tenants must simply rely on the cloud provider to keep the information maintained as agreed upon.

This lack of transparency makes the transition to the cloud, for many large corporations especially, challenging as decision makers are reluctant to put their trust in the hands of the cloud provider. Also privacy regulations, in terms of laws could constrain business from legally adopting to cloud environments. Thus, the transition from old systems are held back. This works as motivation for research communities in finding new methods of gaining the trust of current and potential cloud users. A lot of focus has been directed towards approaching the transparency issue, without being forced to invoke or change too much of infrastructure and implementations of the cloud architecture.

Through the research, new cryptographic schemes arise, under different categories. Some of them still on a theoretical level, while others are tested in real world environments. The aim of this thesis is to study and analyse some of these schemes.

1.1 Outcome

In this thesis we present some of the cryptographic schemes for cloud environments, with the purpose they serve, and what they accomplish. We also look deeper into one particular scheme, the hourglass scheme. First presented by Juels et al. [2], the

scheme provides tenants with proof of correct data encoding at the cloud provider for data at rest. The presented hourglass scheme is dependent on economical aspects, such as pricing of storage and computation by the cloud provider. Based on the hourglass scheme, we conduct an implementation and practical analysis on some of the functionality that they present. We compare our achieved implementation results up against results from the original paper. Based on pricing from Amazon cloud services, we update computation costs and results, validating the hourglass function with current costs.

Through the study of the cryptographic schemes we identified a possibility for improvement in the deletion category. This together with a thought of using the hourglass scheme, motivated us to propose a new scheme in the deletion category. Previously proposed schemes for deletion, rely on removal of encryption keys to do the job. We propose a way of proving deletion, through overwriting of data and using functionality of the hourglass scheme. We argue our solution to have advantages over the previously proposed schemes.

1.2 Report outline

Chapter 2 presents research already conducted within cryptography communities on different research areas. It also introduces the preliminaries needed for the development of the new scheme.

Chapter 3 presents the practical work of this thesis, and an analysis of it compared with the conducted work in the original hourglass scheme.

Chapter 4 presents the proposed scheme for this thesis

Chapter 5 highlights some challenges related to the original hourglass applications and proposed scheme in context of real world deployment.

Chapter 6 concludes with the findings in this thesis, and presents options for further research.

Chapter 2
Background

In this chapter we present the necessary background information related to this thesis. First we clarify the term cloud computing, and discuss how it is evolving and why many still are reluctant to take advantage of its benefits. Second we look into proposed cryptographic schemes which are relevant to this thesis. Third we present in more detail the hourglass paper by Juels et al. [2].

2.1 Cloud Computing

Much attention in security research communities has been directed to different areas within cloud computing and cryptography. The term cloud computing embraces very much, and is often used in a vague context. We first clarify the term cloud computing by looking into the definition from the National Institute of Standards and Technology (NIST) [4]:

> *Cloud computing is a model for enabling ubiquitous, 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 is composed of five essential characteristics, three service models, and four deployment models.*

A compressed description, based on the NIST definition, of the characteristics, service models and deployment models:

The five characteristics:

- *On demand self service* The tenant can scale computation capabilities such as processing time and storage, without interactions with humans.
- *Broad network access* The services are available over a network on standard devices such as laptops, mobile phones, tablets and workstations.

- *Resource pooling* The resources of the cloud provider are shared, where resources are dynamically assigned tenants on demand. Resources are location independent, but can be limited to geographically locations such as country, state or data center.
- *Rapid elasticity* Capabilities can be added and released on demand or automatically, for efficient and simple scaling of services.
- *Measured services* The cloud system monitor usage for optimizing resources.

The three service models:

- *Software as a service (SaaS)* The model provides the tenant with applications running on cloud infrastructure, while the tenant itself does not manage or control the cloud infrastructure. The application could be a web-based email service available on various devices.
- *Platform as a Service (PaaS)* The tenant controls a deployment environment for its application using supported programming language and libraries of the provider, but has not control over underlying cloud infrastructure such as network, servers, storage or operating system.
- *Infrastructure as a Service (IaaS)* The tenant does not have sole control over underlying cloud infrastructure, but control processing, storage, operating system, deployed applications and possibly limited selected network components such as firewalls.

The four deployment models:

- *Private cloud* The cloud infrastructure is dedicated for use by a single organisation, and may be owned by them or third parties, and located on or off premises of the organisation.
- *Community cloud* The cloud infrastructure is dedicated to a spesific community. May be owned by the community or third party, and located on or off the premises of one of the community participants.
- *Public cloud* The cloud infrastructure is for public use, not limited to a community or organisation. Infrastructure is located on the cloud provider premises.
- *Hybrid cloud* The cloud infrastructure is a combination of two of the tree mentioned deployment models.

The hourglass scheme which we take a better look at in section 2.3, is among others, dependent on how the cloud provider handles with resource pooling and rapid elasticity.

Cloud computing is and has been increasing in its use the last years/for some time now. It is estimated that it will surge with 25% in 2014 becoming a 100 billion dollar industry [5]. More and more individuals and corporations rely on cloud computing providers to handle their data. Compared with old systems, this gives rise to a new security questions and challenges. As the data is now handled by third party service providers, much of the risk and "security breach consequences" relies on trust between tenant and provider. By security breach consequences, we mean the responsibility the tenant has over the data if it should be leaked, compromised or in any way be handled in an undesirable manner.

As a survey conducted by Unisys and IDG Connect in 2013 indicates, one of the biggest concerns with thought to deploying businesses to cloud computing platforms is security. Over 70 percent categorized security as their biggest barrier to implementing cloud based solutions in their organisation [6]. Nevertheless, the same survey reveal that the willingness to deploy to the cloud, is increasing, despite significant security worries amongst the business decision makers [7]. Also, the survey reveals that businesses are more willing to take use of private cloud solutions over a hybrid or public cloud solution [8]. With a private cloud solution, the tenant has a higher degree of control over their infrastructure, thus it has better insight to how the data is actually managed.

Additionally, in light of last years events, referring to the disclosures made by Edward Snowden, the trust in outsourcing data to larger data handling corporations would naturally not be reinforced. Snowden revealed NSA's influence over large security companies and organisations such as RSA, NIST and telecom providers, allegedly providing back doors into standard crypto algorithms and hardware [9, 10, 11].

2.2 Cryptographic schemes related to cloud computing

As we have pointed out, research within cloud computing security is important, and has been for some years now. As the survey briefly discussed above indicates, this research area is important for the users of cloud services, to be able to put trust in the services offered.

Different types of cryptography schemes have been proposed in research communities, which propose methods of giving the tenants, or the "weak party" of the agreement, some proof or insurance that their data are being handled correctly and as agreed upon.

We have identified some of these schemes, and present briefly what purpose they serve and techniques they use to accomplish their goals. The schemes are categorised

in four different categories, namely provable data possession, proofs of retrieveability, redundancy and deletion.

2.2.1 Data possession and retrievability

Provable data possession

Ateniese et al. [1] present a type of scheme called Provable Data Possession (PDP). They propose a new PDP scheme, which gives the tenants a means to verify that their data, stored at an untrusted storage is intact and has not been tampered with, without requiring the tenant to download the actual data. The scheme takes advantage of a challenge response protocol, to achieve verification.

Tenants store only a minimum of metadata, which they use to verify responses from the storage provider. Figure 2.1 illustrates the protocol of provable data possession. The upper part regards the creation of metadata and exchange of the file for storage. Prior to uploading the file F to the storage provider, the tenant calculates metadata called homomorphic verifiable tags to each file block using its private/public key pair. It also calculates metadata M on the file for its own storage. The tags are sent to the server for storage, together with the file (denoted as F' in figure 2.1) and the tenants public key, and is what the storage provider uses to generate correct responses to challenges. The tags are of considerably smaller size than the file blocks themselves, leaving the overhead for the storage provider to a minimum. The tenant discards the file and its tags after they are delivered to the storage provider, and only keeps a public/private pair of keys and the metadata M. The storage provider simply stores the received data F', and the tenants public key.

The lower part of figure 2.1 illustrates the process of challenging the storage provider for prof of data possession. On producing the challenge C, the tenant selects random block indexes which it challenges the storage provider on. The storage provider uses the tags for the challenged blocks, the tenants public key, the challenge and the challenged blocks as input to a method for generating the correct response P. To check the response, the tenant inputs the response, the challenge and the public/private key pair to a verification method.

The PDP scheme proposed applies to static data, with the ability to append data. Meaning, if one wishes to insert, modify or delete parts of the data at the storage provider, this would be infeasible, as it would be necessary to set up the PDP scheme again from scratch. Erway et al. [12] propose a dynamic PDP (DPDP) scheme which builds upon the proposed scheme by Ateniese et al., and extends the functionality of it to support deletion, modification and insertion of data.

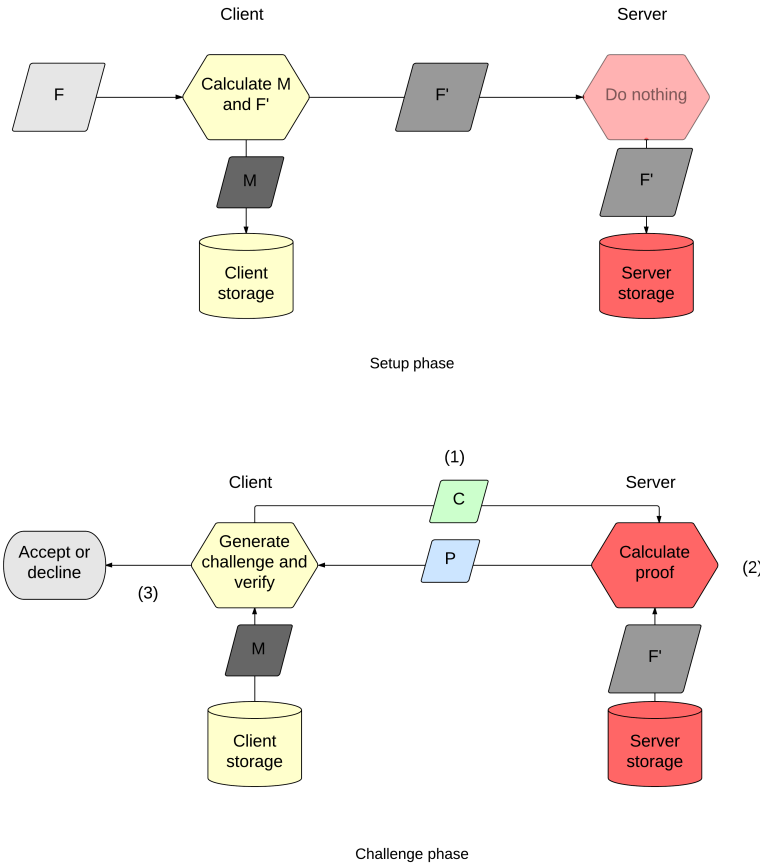


Figure 2.1: *Protocol for provable data possession, as presented in [1]*

Proofs of retrievability

A bit similar to the provable data possession schemes by Erway et al. and Ateniese et al., is the the proof of retrievability (POR) scheme by Juels et al. [13]. This scheme focus on means for the tenant of a cloud service to retrieve proofs of their data being stored without corruption and with the possibility of retrieval of the entire file even with small file corruptions. The scheme is focused on static storage for archival data, meaning data which are seldom updated.

In short the scheme encrypts the file F and applies blocks called sentinels within the encrypted file. The sentinels are random check blocks indistinguishable from

the other encrypted blocks of the file, which the tenant inserts before transferring the file. The tenant challenges the provider on these sentinels. If the provider has altered or deleted ϵ fraction of the file, then ϵ fraction of the sentinels are likely to also been altered or deleted. Error-correcting codes are also applied to the file before transfer. This is for revealing small corruptions to the file which could slip between sentinels. To clarify, the sentinels are there to verify that large portions of the file are not corrupted, however, if small parts of the file are corrupted(e.g. few bits) the sentinels might not reveal this. Then the error-correcting code, can still restore the original file. This implies that if large portions of the file are corrupted, sentinels will reveal it, but one would not be able to repair the file.

The embedded sentinels in the file and the error-correcting codes, will add some computational overhead and larger storage requirements at the cloud provider. Figure 2.2 illustrates the process preparing the file F to be transferred to the server. F is illustrated in a block wise representation, $F_1 \dots F_n$, where F_i represents the i^{th} block in the file. First the error-correct code is applied on each block, and the file takes the form F', $F'_1 \dots F'_n$. Then the file is encrypted through a block cipher, taking the form F'', $F''_1 \dots F''_n$. The final phase generates the sentinels, and appends them to the encrypted file, and permutes the blocks to distribute the sentinels over the result file F^*, $F^*_1 \dots F^*_n$. The sentinels are indicated in figure 2.2 with a blue color. This process is executed on the client side, before the result F^* is transferred to the server.

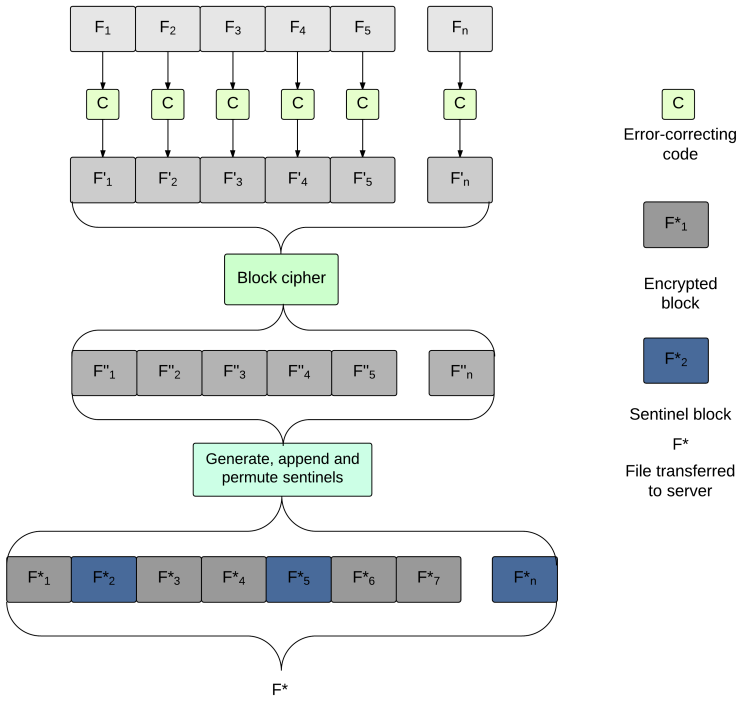


Figure 2.2: *File transformation in POR before transfer to server*

A drawback in the proposed POR scheme by Juels et al. [13] is that the sentinels can only be used for verification once, meaning that the number of embedded sentinels in the file puts a restriction on the lifetime of the scheme.

The scheme by Bowers et al. [14] is a new variant of the POR scheme by Juels et al. They use the proposed POR scheme by Juels et al. as a starting point, and improve it in terms of a higher acceptance for error rate ϵ, on the server while still being able to retrieve the original file. They also achieving lower overhead data on the uploaded file. As error-correcting method, they use an inner and outer error-correcting code. An outer error-correction code is embedded to the whole file before uploading to the server. On challenges from the client, the server computes an inner error-correction code on the response to the client. This implies that if any altering of bits occurred in transaction between server and client, the block sent from the server could be restored at the client. If the original blocks were altered at the server, the outer error-correcting code, could retrieve the original file block again.

The papers which presents the PDP and POR schemes, criticize each other. It is

highlighted by Bowers et al. that the PDP scheme provide means for the tenant to verify that the storage provider possesses the file, but gives no guarantee that the tenant can retrieve the file. On the other side, the PDP schemes by Atenise et al. and Erway et al. criticize the POR of Juels et al. for the limited appliance with only encrypted files and the limited number of challenges which can be executed on a file.

The mentioned schemes will in some manner provide the sufficient proof to the tenants that their data is intact or at least reveal if it is corrupted. But, they provide no insight into the encoded format in which the data is stored (proving correct encoding) or if they are stored with redundancy. The POR scheme by Juels et al. is static requiring encryption of the data. This sets some restrictions on the schemes, leaving the usage limited.

2.2.2 Proving distributed storage

Redundancy
Bowers et al. [15] present a scheme for verifying that your files can resist hard drive crashes in the cloud. This means, that your files should be stored in such a manner that if a drive crashes, it should be possible to retrieve the file again. For example Amazon claims to have stored copies of files in their S3 storage service, over three separate instances. But lack of transparency to the tenants, give them little comfort other than trusting the cloud providers to keep files in a crash resistant manner.

The presented scheme is called remote assessment of fault tolerance (RAFT). It is a timing based challenge-response protocol used to prove that blocks of files are stored over separate hard drives, and that the original file can be retrieved in face of a number t hard drive crashes. To be able to ensure recoverability of a hard drive crash, the files are stored with redundancy. This is realized by applying an erasure code on the file, before distributing the blocks to different drives. An agreement between the tenant and cloud provider maps the file blocks, well balanced, over the number of drives.

The idea is that the tenant challenges the provider for an amount of blocks of the file F. The number of hard drives which the file is distributed over, will determine how fast the provider is able to respond to the challenge. For example for a 100 block challenge which takes one second to respond to for storage on one hard drive, should be able to produce the correct response in a half second, if it stores the blocks on two hard drives. The time based protocol is dependent on the operation times of hard drives (i.e. seek time) which vary substantially from seek to seek on random reads. By requiring each drive to fetch a number of random blocks, this variation is smoothed out over the drives. Thus, similar to the hourglass scheme by Juels et al. [2], which we shall look into later, the RAFT scheme is dependent on resource performances at the cloud provider, mainly the performance of rotational hard drives.

Related to the RAFT scheme is the scheme presented by Wang et al. [16]. They propose what they call a "layout free" scheme, which removes the mapping agreements of block to hard drive between the tenant and provider in the RAFT scheme. They claim this mapping of blocks to physical hard drive addresses to be a difficult operation to execute, as cloud providers are reluctant to giving out storage layout information and implementation details. To overcome this challenge, and realize their "layout free" scheme, they take advantage of an evenness index, realized by the Shannon-Wiener Index, to measure the distribution of blocks over the different hard drives. An honest provider, which stores equal amount of file blocks on each drive, would get an evenness index of one. The dishonest provider would get a lower score, dependent on the degree of unbalance between drives.

2.2.3 Deletion

Deletion by key removal
A scheme which considers deletion assurance is presented by Tang et al. [17]. They present the file assured deletion scheme, FADE. The scheme operates with a definition of deletion as, if the encryption keys to the actual data is permanently removed or inaccessible, the data will be permanently unavailable - permanently encrypted.

Each file is associated with a policy for file access, where each policy is associated with a control key. Control keys are handled by a trusted third party, and are used for encryption of data keys. So when deletion is requested by the data owner, the policy for that file is revoked and the corresponding control key is discarded. Policies can be created for specific users, groups or scenarios. To access the files at the cloud provider, the data owner first needs to authenticate itself to the third party key manager, to prove that it satisfies the policies for the files. With the correct policy the data owner will be presented with a control key in which it can use to generate the correct data key.

Perlman [18] take a similar approach as Tang et al. with the FADE scheme, using a trusted third party key manager to handle control keys. But instead of binding the control key to a policy, the keys are time-based. This implies that, on creation of the keys, an expiration time is associated with the key. When the time expires, the key manager removes the key from storage, leaving encrypted data inaccessible.

The scheme presented by Geambasu et al. [19] called Vanish, address deletion of data without any particular user interaction. It is similar as Perlman in the manner that the keys has an expiration date, but the keys themselves are handled differently. They focus on sensitive data, e.g. data related to emails with sensitive information, which no longer serves a purpose after having been read. Thus, they present a self destruction deletion technique, which delete or makes all data unavailable after execution. The technique splits the data encryption key into multiple parts, and

distributes the parts over a peer-to-peer network, mapped with a distributed hash table. The nodes in the network, remove their part of the key, after 8 hours. (If one wish to have an expiration time exceeding 8 hours, the peer-to-peer network nodes must be updated.) In this manner, the data will be unavailable, even if it is not deleted by the storage provider. With their approach they avoid relying all trust in one single third party key manager, which they argue is in their favour. A third party key manager could be untrustworthy, or even if trustworthy, clients can be reluctant to trusting them.

All identified deletion schemes rely on encryption of the data and the corresponding keys to fulfil their purpose. This automatically removes the possibility of processing the data, unless the cloud provider is given the keys. Also the timing-based key schemes limit their application in terms of expiration dates on the keys.

Deletion in general In chapter 4 we present a new thought of a cryptographic scheme which will provide the tenants with proof of deletion by an economical rational cloud provider. As we shall see, the new scheme will take advantage of an hourglass function similar to the ones we present in section 2.3. We briefly introduce some thoughts around the deletion methods presented above and what one might have to consider when handling deletion in general.

The term deletion has been interpreted and defined in many different ways, however, Reardon et al. [20] define it as:

> *"data is securely deleted from a system if an adversary that is given some manner of access to the system is not able to recover the deleted data from the system"*

The identified schemes for cloud environments previously discussed, are not focused on the deletion of the actual data, rather the insurance that if the encryption keys are unavailable (deleted) the data will be irretrievable. At the present, for sufficient choice of encryption method, this is true - also according to the mentioned definition. However, as criticized before [21], this approach is only as strong as the underlying encryption method, and the future could develop into ways of undermine these encryption methods. The everyday increasing development in computation power, and presently unknown loopholes in encryption methods, leaves these schemes vulnerable for the future, thus one can argue them not safe.

In addition, as we mentioned the revealing of Snowden, the encrypted data may not even at current time be secured from organisations having knowledge about backdoors into standardized crypto algorithms. Thus, leaving data encrypted after a deletion, would not be preferable.

Also, one might consider the scenario where one does not wish to encrypt the data before outsourcing it to a cloud provider for storage. This could be the case, when the data is required to undergo processing in the cloud. To ensure deletion of this data, there are no encryption key to dispose of. This implies that these schemes are limited by the intended usage of the data.

As we shall see, the proposed scheme in chapter 4 applies overwriting of the intended data for deletion, instead of encryption key removal.

In most normal file systems, a deletion of a file will not actually remove the data. When executing a deletion, a reference to the actual file is removed from the register, and a bit in the file header is flipped [22]. The flipped bit indicates that the space which the file occupies on the disk, is now available for new data. This can potentially lead to that the intended data is never overwritten by new data, and will forever be available to retrieve for users with some degree of computer skills. A known way of ensuring that the data is no longer retrievable is to overwrite the data with random generated data. This by far makes the probability of a malicious user being able to recover the deleted data smaller.

It is discussed that to completely ensure no chance of recoverability, a certain number of overwrites must be executed (35 overwrites are advised by Gutmann [23]). This, however, has been criticised not to have as much importance as first thought. Wright and Kleiman [24] reveal that by a single overwrite, one would have only slightly better than 1/2 chance of predicting the original bit, which is close to simply guessing it. The best result measured was a 56% success ratio. This implies that to guess a character (one byte, 8-bit) correctly, there is a 9% $(0,56^8)$ chance of success. The probability for successfully restoring the original data at bit level, decreases as the number of sequential bit guesses increases. Based on these observations, we propose a scheme which takes advantage of a single overwrite of the intended data, to ensure the data unavailable for the future.

2.3 Hourglass scheme

In this section, we present in detail, the hourglass scheme proposed by Juels et al. [2].

The proposed cryptographic scheme is designed for providing the user of some cloud storage service an assurance that the stored data is kept as agreed upon, and to strongly motivate the cloud providers to fulfil its part of the agreement. This means that if the client and cloud provider have an agreement about storing data in an encoded form, the scheme provides measures for the client to challenge the cloud provider for proof of correct encoding. The scheme presented applies to solutions

where the cloud provider has access to the raw data of the client for processing purposes, meaning the encoding of the data is handled by the cloud provider itself. In cases where the cloud provider has such access, a cheating cloud provider can store the raw data instead or in addition to save resources by not having to decode data when raw data is needed. The proposed scheme however presents strong economical incentives for the economical rational cloud provider to actually store the clients data encoded, when the data is at rest. An economical rational cloud provider would not misbehave if the consequence of doing so inflicts a certain cost. The framework of the hourglass scheme contains the necessary means so the client can ensure such compliance.

The term at rest is used when we talk about data stored at the cloud provider, when the data is not needed for processing. In the case of the hourglass scheme, data should be under a defined encoding when at rest.

The encoding can be of the clients' choice, e.g. encrypted, watermarking or file binding, and is more or less independent of the actual hourglass transformation. Encryption encoding is the form they focus on in the original hourglass paper [2].

2.3.1 Challenges and solution

Imagine the client wants to store the file F in an encrypted form at the cloud provider, where F represents the file in a raw format. When encrypting F, the file takes the format G. Transformation from F to G is referred to as the encoding phase. Figure 2.3 displays the generic hourglass protocol as it is presented in the original paper, with the functions and interactions involved between the tenant and cloud provider when executing the hourglass scheme. This is constituted by three phases. Encoding phase is the first phase.

Sending challenges on parts of, or the entire encoding G, will not be sufficient proof that the cloud provider actually is storing the data encrypted. As symmetric key-encryption is fast, the encryption could be done on-the-fly by the cloud provider, when receiving the challenge. The encoding phase consists of the methods $keygen - enc$ and $encode$, which respectively generates a secret encoding key and encodes the raw data provided by the client, using the secret key.

To solve the challenge of encrypting on-the-fly, an additional transformation of the clients data is required, applied on the encrypted file G. This transformation, referred to as the hourglass encapsulation phase (Phase 2 in figure 2.3), encapsulates all blocks of G in a new format through use of the $hourglass$ function, and results in the hourglass format H. The idea is that the $hourglass$ function has some timing bound with an upper limit T, which means the cloud provider will not be able to execute the hourglass encapsulation in less than T time. The timing bound for

Client		**Cloud Provider**
Phase 1: Generate file encoding		
input: file $F = F_1 ... F_n$		
	$\xrightarrow{F}$	$\kappa \leftarrow keygen - enc$
		$G = encode(\kappa, F)$
	$\xleftarrow{G,\pi}$	$\pi \leftarrow Proof\ of\ correct\ encoding$
Phase 2: Hourglass encapsulation		
$H \leftarrow hourglass(G)$		$[H = hourglass(G)]$
Generate integrity checks I_H on H	$\xrightarrow{I_H[,H]}$	
Phase 3: Format checking		
$\{c_i \leftarrow challenge\}_{i=1}^{t}$		
Start timer	$\xrightarrow{{c_i}_{i=1}^{t}}$	
		$\{r_i = respond(H, c_i)\}_{i=1}^{t}$
	$\xleftarrow{{r_i}_{i=1}^{t}}$	
Stop timer τ : elapsed time		
Accept iff $\{verify(H, c_i, r_i) = true\}_{i=1}^{t}$		
(using I_H)[and $\tau \leq T$]		

Figure 2.3: Generic hourglass protocol as presented in [2]

hourglass functions, are associated with a resource bound at the cloud provider. Such resource bound could be on storage, computation or networking. For example a storage resource bound is the delay which is experienced by rotational hard drives when seeking for and reading data blocks. With knowledge about the characteristics of the resource bounds, one can take advantage of these to implement *hourglass* functions.

A challenge with the hourglass function is that the cloud provider could store the file or parts of it in the hourglass format, but additionally keep a raw copy of the file, which it use when processing is necessary, referred to as the double storage problem. However, this is where the economical incentives of the hourglass scheme take effect. In the work with the original paper, they show that the cost of storing the extra data, will exceed the cost of executing the *hourglass* function. In section 2.3.5 we go more into detail about this.

The original file F is constituted by n blocks of size l bits, where F_i denotes the i^{th} block of F. For the encoded file G and the hourglass encapsulated file H, similar notation applies, n is the number of blocks in the file, with a block length of l bits. G_i and H_i represents the i^{th} block of G and H, respectively.

Once the file is encapsulated in the hourglass format H, the client can challenge the cloud provider for proof of correct encoding. This phase is referred to as the format checking phase (Phase 3 in figure 2.3). The client selects a random block index i (c_i) and challenges the cloud provider to respond with the correct response, based on block H_i from H. Challenges can also be sent on multiple random selected blocks of H. An honest cloud provider simply looks up block H_i of H and responds with r_i. This is a quick operation including only a seek and read for a sequentially stored H. While, the cheating cloud provider which stores only the raw file F, must do the entire encoding (G) and encapsulation (H) of the file to be able to produce the correct response. The hourglass encapsulation alone is a process which on average takes T time. Upon verification, the client can check that the challenge is provided the correct response by applying the challenge c_i, response r_i and hourglass encapsulation H to the $verify$ function. If the elapsed response time τ is less then time bound T, the client can conclude that the cloud provider did store the data in an hourglass encapsulated format.

The three phases mentioned, namely file encoding, hourglass encapsulation and format checking, constitute the generic hourglass protocol.

In the original hourglass paper they assume, for simplicity, that the tenant has knowledge about the entire hourglass encapsulated file H. This basically implies that the tenant stores its own copy of H, and upon verification of responses, it checks H together with the response and challenge. This is impractical, and removes much of the incentives of outsourcing the data. However, they state that a simple MAC or Merkle tree method could be applied in a real world implementation, to reduce the storage of the tenant. In chapter 4 we will look into verification methods that can be applied.

Naive approach

In the original hourglass paper [2] a naive approach to achieve an hourglass function is presented to illustrate some challenges.

The suggested function mixes all blocks of the encoded file G in such a manner that any block H_i will be dependent on all blocks in G. This would be a time consuming process which a cheating server, storing only raw file, will not be able to perform in less then T time. Figure 2.4 illustrates the process.

To achieve this dependency the approach would use a pseudorandom permutation (PRP) of G by applying a block cipher in cipher block chaining (CBC) mode in two directions over the intended file. Running a CBC over a series of blocks, makes each block in the chain dependent on all preceding blocks in the chain [25]. In the

naive approach the block cipher is indicated by the notation enc_k, where k is the key. The key is known to the server, for reversal reasons. It first applies the CBC over every block in G, and stores the result for every round as an intermediate value A_i, where i is the ith round. The initial round takes an IV as input, denoted by IV_f, where f indicates it is for the forward round. Equation 2.1 defines the forward round. After the forward round every intermediate block A_i will be dependent on the previous intermediate block A_{i-1}. Then it applies the CBC one more time, in the reverse order with the intermediate blocks A_i and H_{i+1} as input for each round. This time the initial round takes IV_b (backward direction) and A_n as input. Equation 2.2 defines the backward round. The output of each round produces the hourglass encapsulated blocks, H_i. This makes every single block in H dependent on all blocks in G.

$$A_1 = enc_k(G_1 \oplus IV_f),\ A_i = enc_k(G_i \oplus A_{i-1}) \tag{2.1}$$

$$H_n = enc_k(A_n \oplus IV_b),\ H_i = enc_k(A_i \oplus H_{i+1}) \tag{2.2}$$

The problem with this solution is the possibility to store chosen values of the computation to be able to produce a response on-the-fly.

Lets say the server stores the raw file F, and additionally every 10^{th} intermediate block of A and encapsulated block H. When challenged on e.g. H_{14}, the cloud provider could on-the-fly encode F to G. Then, as it is in possession of the intermediate blocks it can compute the forward chain $A_{10} \dots A_{19}$ and retrieve H_{20}, and then compute the backward chain, using the computed blocks in the forward chain as input, until it reaches H_{14} ($H_{20}, H_{19} \dots H_{14}$). As this only requires a partial computation of the chain, it would be less time consuming, and could be executed on-the-fly in less then T time.

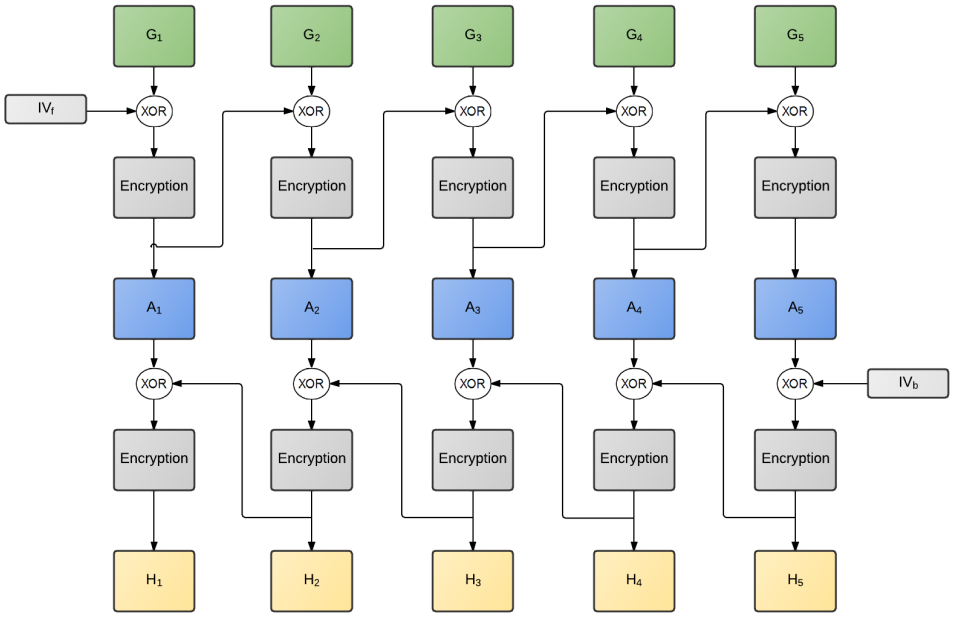


Figure 2.4: *Naive approach*

To solve the problem in the naive approach, three hourglass functions are proposed, namely the butterfly function, permutations function and RSA function.

2.3.2 Hourglass functions

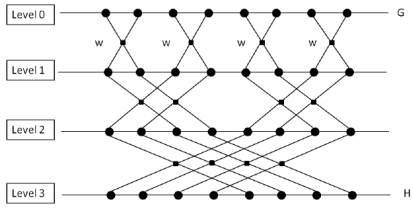
Butterfly function

The butterfly functions resource bound relies on an upper bound on the number of storage accesses which can be executed within a time bound. The function applies a cryptographic operation w on blocks two by two in a number of rounds d, a bit similar to the naive approach but with different permutation method. For each round, each block is combined with a new block, given by the butterfly hourglass algorithm, right side of figure 2.5. Each cryptographic operation makes the two output blocks dependent on both of the input blocks. By switching the index of input blocks for each round, a dependency between all original file blocks and each single hourglass encapsulated block will exist. The number of rounds d in the butterfly function is dependent on the number of blocks n in the file, $d = \log_2 n$. $G_j[i]$ represents the i^{th} block at the j^{th} level in the butterfly, where $1 \leq j \leq d$ and $1 \leq i \leq n$. Left side of

figure 2.5 shows an execution of the butterfly function on a file with $n = 8$ blocks.

In practice the cryptographic operation is typically a known key pseudo-random permutation (PRP), such as a block cipher. The operation is involved in a butterfly encapsulation $n \log_2 n$ times.

The butterfly function solves the problem from the naive approach through permuting the intermediate nodes instead of chaining them. This limits the possibility to use intermediate nodes on the computation from G to H, to deliver a large fraction of the encapsulated format H. We look more into this in section 2.3.3 and 2.3.4.



Algorithm 1 Butterfly hourglass algorithm

1: **for** k from 0 to $n/2^j - 1$ **do**
2: **for** i from 1 to 2^{j-1} **do**
3: $(G_j[i + k \cdot 2^j], G_j[i + k \cdot 2^j + 2^{j-1}]) \leftarrow w(G_{j-1}[i + k \cdot 2^j], G_{j-1}[i + k \cdot 2^j + 2^{j-1}])$;
4: **end for**
5: **end for**

Figure 2.5: *Butterfly algorithm as presented in [2]*

Permutation function

The permutation-based hourglass function provides good security without cryptography, making it extremely fast and simple. Its security and resource bound relies on the characteristics of random accessing data on rotational hard drives. Rotational hard drives are optimized for sequential file accesses and perform poorly on random data access. This method is taking advantage of these characteristics, by permuting the content of the encoded file G widely across H.

This implies that if an adversary cloud provider is challenged on a block H_i, but stores no or only partial information about H, it must collect elements of H_i which are widely spread over G, to produce the satisfying response to H_i. This would require a number of random accesses, since the elements of H_i is not located in a sequential order. As random access is a time consuming process it slows down the cloud providers response time.

The encoded file consists of n blocks. The permutation is executed on symbols, where m symbols constitute a block. Symbol are represented as a sequence of z bits. The notation $G_i[j]$ represents the j^{th} symbol of the i^{th} block of G. Thus, a file is divided into $n * m$ symbols, where $G[k]$ represents the k^{th} symbol in a symbol

wise representation of G, where k is defined within $0 \leq k \leq n * m - 1$. Figure 2.6 illustrates the symbol wise permutation of a file. In the first step the file is encoded in the chosen format. Second, the file is permuted following a defined permutation pattern.

The permutation pattern used has the property that, for the m symbols of the encoded block G_i, the symbols are uniformly distributed over the hourglass encapsulation H. The following permutation pattern were used in the original paper:

$$H[i] = G[ih \mod nm] \; and \; G[i] = H[ig \mod nm]$$

with $gcd(hg, nm) = 1$.

This gives $H[i] = G[ih \mod nm] = H[igh \mod nm]$, thus $gh = 1 \mod nm$. The parameters h and g are inverse of each other and $gh = 1 \mod nm$ has a solution only if $gcd(gh,\ nm) = 1$. The parameter g is defined as $g = \lceil \tau_s/\tau_r \rceil m + a$ for some $0 \leq a < m$, where τ_s and τ_r are the average seek and read time for rotational hard drives, respectively. Through this, all parameters needed to execute the hourglass function on a file can be calculated.

Example parametrization of the permutation function is provided in section 2.3.4.

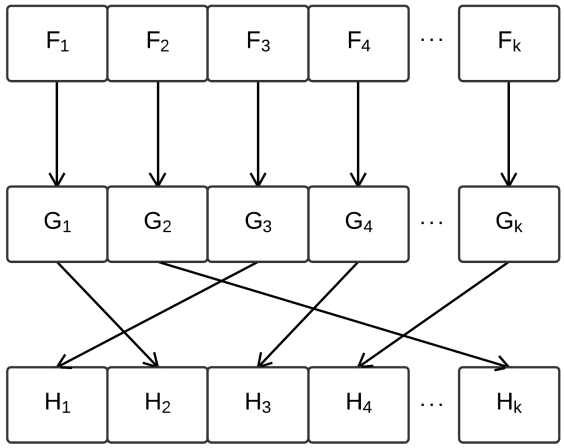


Figure 2.6: *Permutation function, inspired by [3]*

RSA function

The RSA-based hourglass function uses an RSA signing of G to encapsulate the encoded format to the hourglass format H. The protocol uses a function f with

the property of it being hard to invert. The function f has the requirement of being a trapdoor one way permutation, where the client has knowledge about the trapdoor K and the cloud provider has not. When performing the encapsulation, each block G_i is inverted through the trapdoor function f to produce the corresponding hourglass block H_i, $H_i = f^{-1}(G_i)$. Figure 2.7 illustrates the process. The hourglass encapsulation phase is executed at the tenant, and later H is sent over to the cloud provider for storage. The resource bound on the RSA function is solely on computation, rather than storage. In the original hourglass paper, the RSA scheme is presented, but no implementation or experiments was done with it, because of the strengths of RSA and its way of working is well documented.

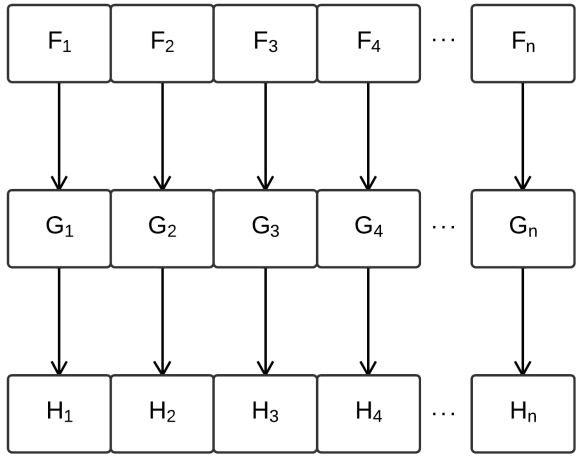


Figure 2.7: *RSA function, inspired by [3]*

2.3.3 Security analysis

The security analysis in the original paper provides a security theorem for the different hourglass functions. The analysis discusses how the adversary can store intermediate blocks of encoded and large fractions of raw data, to successfully respond to challenges from the clients, while omitting to store the file in an hourglass encapsulated format. The theorems provides a formula which gives a value to the extra storage the adversary cloud provider needs to produce the correct responses. We will here focus on the butterfly and permutations functions as these are the ones they implement in their experiments.

Butterfly function

The analysis under the butterfly function experiments with the thought of the

adversary sitting on intermediate values in the butterfly graph (blocks in the transformation from G to H) and blocks of the raw data, called black pebbles and red pebbles respectively. The adversaries storage H' starts with an initial configuration of red and black pebbles. Given a pair of black pebbles in storage H', the adversary can input these to the operation w, where the output will be two new black pebbles on the next level in the butterfly graph. The goal of the adversary is to have the best possible pebble configuration, so that it can successfully respond to challenges from a client, while still keeping a large number of red pebbles in its storage.

As the butterfly function is time-based, it is dependent on some time bound in the challenge response protocol which sets the limit of time the server can use to respond with the correct response. The time bound is denoted as T, and translates into an upper bound determined by a parameter $\epsilon < 1$ and n. ϵn sets an upper limit for the number of storage accesses the server can do.

Through their security analysis they prove the following theorem for the butterfly function:

Theorem 1 *Suppose A can successfully respond to t challenges on randomly selected blocks of size l bits of H with a probability α, using extra storage of size s' and up to ϵn timely block accesses. Then:*

$$s' = \min\{\alpha^{1/t} \cdot nl \cdot (1 - \epsilon, nl \cdot (1 - \epsilon) + \log_2 \alpha^{1/t}\} \tag{2.3}$$

We interpret the meaning of the theorem in section 2.3.4.

Permutation function

The security analysis for the permutation function relies on parameters in a timing model for rotational drives. Rotational drives are designed for optimal performance on sequential read of data. When blocks are scattered over a disk, a seek process must be performed for each block read which are not sequentially placed together. This is a time consuming process as the read head must be moved to correct location on the platter for each block. The seek time is dependent on the rotational speed and the time the read head needs to move to the correct track of the hard drive [26]. In the drive model for the rotational drives, a constant seek time is assumed, τ_s. Also, constant read time for a block is assumed to be τ_r. A sequential read of e blocks are thus defined as, $\tau_s + (e - 1)\tau_r$. A timing bound for an adversary server A is defined as T, which is the time bound for responses to a challenge block from a client.

In reality a server will likely store files in parallel, i.e. on multiple drives, making symbol lookup d times faster for d parallel storage units. However, this will not reduce the adversaries overhead storage s' with a factor of d. The read of a response block for an honest server to a challenge on H_i, will still be faster then collecting m symbols from d drives for the adversary, for $d < m$. Also, if the client can estimate the degree of parallelism d for the adversary, it could challenge for d number of blocks.

Through their security analysis they prove the following theorem for a challenge on a single block of H:

Theorem 2 [2, Theorem 3] *Let T, τ_s, τ_r, m, l and n be defined as above satisfying $2m \leq l(m - T/\tau_r)$. Suppose A can successfully respond to a challenge on a randomly selected block of H with probability $\alpha \geq 3/4$. Then the following bound holds for the extra storage used by A, where $k = \min\{\lfloor \tau_s/\tau_r \rfloor, 1 + \lfloor n/(2m^2 + fl/3) \rfloor\}$*

$$s' \geq (2\alpha - 1) \cdot nl \cdot \frac{m - T/(k \cdot \tau_r)}{m - 1} \tag{2.4}$$

2.3.4 Comparison of hourglass functions

Figure 2.8 and 2.9 display graphs for the overhead relationship for an adversary provider, with different values of α ranging from $0 < \alpha < 1$ with a 0.1 interval, for theorems 1 and 2, respectively. We used the same parameters for block and symbol size as they did in the original paper with their example parametrisation. The parameters used are $t = 1$ challenge (no other option with theorem 2), block size 128 bit and 4 KB for the butterfly and permutation function, respectively. We used a 64 bit symbol size, $\tau_s = 6ms$, $\tau_r = 0.03125ms$ and $T = 6ms$ for the permutation function. For the butterfly function the parameter ϵ was set to 0.05.

We experienced that the file size had no impact on the overhead relationship. The overhead relationship is defined as the overhead bit amount s', divided by the original file size. We observe that the overhead is linear, dependent on the success probability α.

As an example, the adversary for the butterfly function must store an additional $0.94n$ blocks in addition to the raw file F to successfully respond with 99% certainty on a challenge from the tenant. The adversary in the permutation case must store $0.89n$ additional blocks for the same requirement.

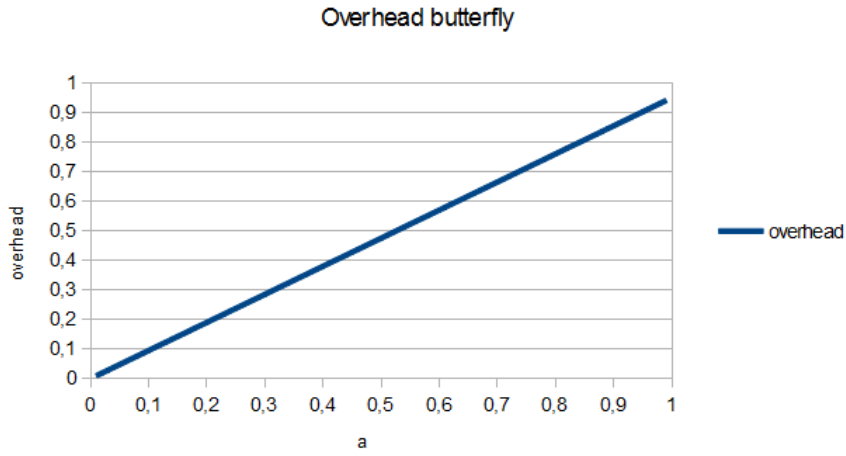


Figure 2.8: *Overhead data s' for butterfly function*

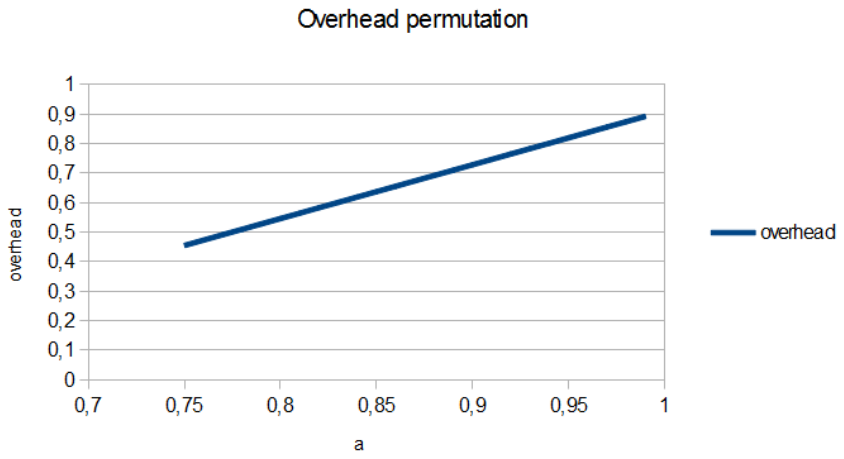


Figure 2.9: *Overhead data s' permutation function*

Table 2.1 summarize and compares the characteristics of the three hourglass functions presented in the original scheme. As we shall see in section 3.3, the functions depart in usage based on the characteristics of computation cost. The RSA function stand out from the other two, by not providing the server with the ability to compute the hourglass encapsulation and on resource bound. The permutation function is the simplest one, not relying on any cryptography to achieve its purpose.

	Butterfly	**Permutation**	**RSA**
Resource bound	Storage	Storage	Computation
Computation cost	Medium	Low	High
Cryptography	Yes	No	Yes
Server compute H	Yes	Yes	No
Timing based	Yes	Yes	No

Table 2.1: Comparison of hourglass functions

2.3.5 Economic incentives and arguments

As previously mentioned, the hourglass scheme gives the cloud provider economical incentives to store the tenants data as agreed upon. In this section we elaborate on how the use of the scheme would impose the providers with these incentives.

The obvious reason for the cloud provider to act as agreed upon would be that if caught cheating, this could impose great economical losses, for example in terms of reputation, loss of clients, law suits and so on.

However, the hourglass scheme presents a more sophisticated method to motivate the cloud providers to do as agreed upon. It is of such character that it is not economical reasonable for the cloud provider to store a raw copy of the data, in addition to the encapsulated format. This is realized by the fact that extra storage implies extra costs for the cloud provider, while the extra cost of executing the hourglass encapsulation is substantially lower. Dependent on how often the raw data needs to be accessed for processing, the cost will vary. To elaborate - as each execution of the hourglass function adds a certain cost, the sum of these costs must be lower then e.g. a monthly cost of storing the extra data. From the experiments in the original paper, they identify that the butterfly function is well suited for storage where access to the raw data is rarely needed, while the permutation function is better suited in cases with more frequent raw data access (in the hundreds per month). The numbers related to cost and storage are presented in section 3.3.

Chapter 3
Practical analysis

In this chapter the practical work conducted in the thesis is presented. We wanted to do an implementation of the experiments done in the original hourglass paper, to compare the results and get an insight in how different implementations and hardware specifications would affect the performance and economical arguments. The first section presents the implementation conducted in the original hourglass paper and the choices and details we did with our implementation. Second the results are compared and discussed up against each other. Third, an economic analysis will look into the pricing and performance to evaluate the economical arguments, also with current cloud provider prices.

3.1 Implementation

3.1.1 Original Hourglass paper implementation

In the work with the original hourglass paper, experiments with the butterfly and permutation hourglass function where conducted, both in an Amazon EC2 [27] cloud environment and on local hardware. The local machine had an i7 980X processor running 6 cores at 4 GHz, and hardware support for AES encryption.

The butterfly function was implemented in two versions, one single threaded and one multi-threaded version. Running the hourglass function as a single thread implies that only one core of the CPU is exploited at the same time, and parallelism is not available. This of course affects the time required for the encapsulation to finish. The cryptographic operation (w) used in the butterfly function was a AES block cipher, and thus the block size for the file was set to 128 bit, as this is the block size in which AES operates on. As mentioned their local machine has hardware support for AES, which is a benefit for the butterfly implementation, performance wise.

For the permutation function they used the construction mentioned in section 2.3.2 to permute the file and scatter the symbols widely over H. As parameters they defined the block size as $4KB$, and the symbol size z as 64 bits. This gives the number of symbols in a block, $m = l/z = 512$. They implemented the permutation function in a stream lined fashion. This means, as soon as a block has been read from the hard drive into memory, the symbols of the block are permuted to their respective placement in the hourglass result H. This will give performance advantages compared to reading the whole file into memory, before applying the permutation operation.

3.1.2 Our implementation

Like for the experiments conducted in the original paper, we also implemented the butterfly and permutation hourglass function, but we only did a local implementation. Not much detail about their implementation was reviled in the paper, with thought to pseudo code or type of programming language used. This caused us to make some assumptions and guesses along the way, resulting in code which probably could be done more efficient by a more experienced programmer.

Our machine ran an Intel Core i7 860 processor with four cores running at 2.80 GHz. With thought to the butterfly function implementation, the processor did not have hardware support for AES, so a software implementation was used instead. Python was used as programming language, and the choice was based on previous experience and simplicity of the language. The AES software implementation used was the one provided by the PyCrypto library for Python [28]. Based on this platform, we had a weaker basis for achieving efficient performance results.

From the original hourglass paper we used the information about their experiments as best as we could to produce our implementation. Thus, for the butterfly function we also chose an AES block cipher in the operation w and a 128 bit block length (l), and used the butterfly algorithm specified in section 2.3.3 to define the butterfly permutation. We used an electronic codebook (ECB) [29] mode for the AES encryption. For simplicity we chose to do a single thread implementation of the butterfly function.

The specified butterfly algorithm does not specify, on block level, how parts of two blocks are mixed to produce the sufficient dependency between them, i.e. for each invocation of the operation w, some mixing of the two input blocks must be performed before they are delivered to the AES encryption within w. Remember, the goal of the butterfly function is to make every single encapsulated block H_i, dependent on all encoded blocks in G. Figure 3.1 shows how our implementation of the w operation realized the mixing of two blocks. It takes two blocks of size 128 bit as input, and switches 64 bits of each block to the other. In this manner the output blocks will be dependent on both input blocks. Following the butterfly algorithm

which mixes the indexes of input blocks for each round, we achieve the necessary dependency. With this approach there is two invocations of the AES block cipher for each invocation of w, giving a total of $n \log_2 n$ invocations of AES for a n block file.

Our implementation of the permutation function used the same block and symbol size as the experiments conducted in the original paper, $l = 4KB$ and $z = 64$ bit, thus $m = l/z = 512$ symbols in each block. First the function reads the whole file into memory. Then the symbols of the file are places one by one in its correct location in the hourglass file H, according to the permutation pattern in section 2.3.2.

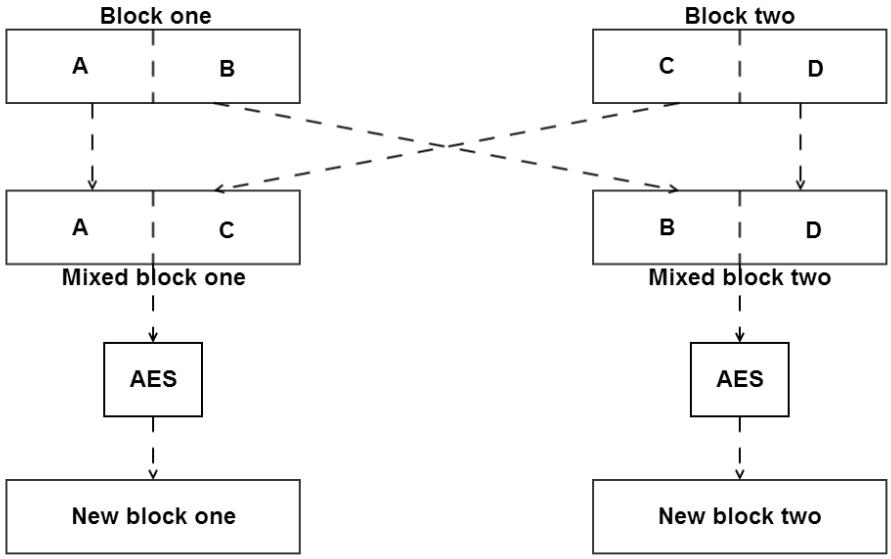


Figure 3.1: *Operation w, mixing of two blocks*

The source code for our implementation is provided in appendix B.

3.2 Comparison of paper results and practical work

In this section we present the results of our implementation and make comparisons with the result from the original hourglass paper. As mentioned, they implemented their solution both locally and on an Amazon EC2 instance for the butterfly and permutation function. The butterfly function they implemented both as a single-thread version and a multi-thread version.

With our work we have confirmed that an implementation of the hourglass functions can be achieved. The implementation provides a relative cost comparison of the butterfly and permutation functions.

Comparison of performance results between the original paper and our implementation, shows a more efficient implementation in their favour. Much because of hardware advantages (AES support and processor clock rate), but also likely because of software decisions and implementation details resulting in more efficient code. Figure 3.2 shows the performance results of their implementations, both on the local machine (left side of figure) and on the EC2 instance (right side of figure). All their tests were executed 5 times, and the figures display an average of these runs.

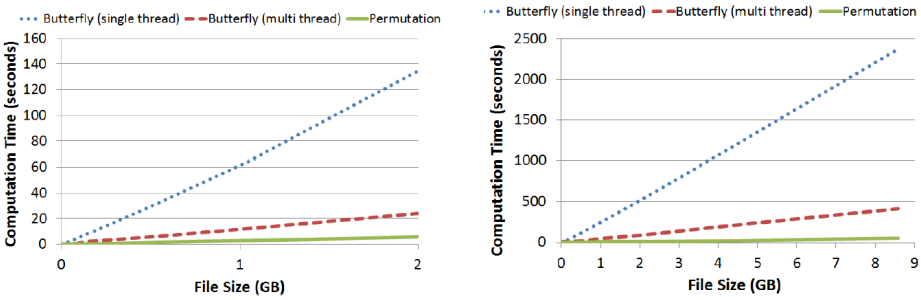


Figure 3.2: *Performance results from hourglass paper (taken from [2])*

Comparing their result shows that the multi-threaded butterfly implementation, used on large files, is a factor of 5 times faster then the single threaded, both locally and on the EC2 instance.

Comparing the timing results between the local and EC2 butterfly implementation (both single and multi-threaded), shows that the local version is a factor of 4 times faster then the EC2 version. This is mainly due to the hardware support for AES on the local machine.

The permutation implementation is a factor of 8 times faster then the multi-threaded implementation on the EC2 instance, and 4 times faster locally. Much because, the permutation function does not include any encryption operation.

Figure 3.3 and 3.4 shows our performance results for the butterfly and permutation function, respectively. As with the experiments in the original hourglass paper, the figures display an average over five runs with the hourglass functions. One quickly observes that our implementation is more time consuming. For the butterfly function, a 256 MB file requires a processing time of 1276 seconds on average. All timing

results are summarized in appendix A. To comparison, the same file would require about 14 seconds with the implementation done in the original paper.

The above estimate is made from the left side of figure 3.3, which indicates that the single thread implementation uses 60 seconds on a 1 GB file. As the performance is linear, we calculate a throughput of about 18 MB per second for their implementation, resulting in a 14 second execution time on the 256 MB file.

If we compare the achieved results of the implementation in this thesis up against each other, we find that our permutation function is a factor of 25 times faster than our single-thread butterfly implementation. This is comparable with the difference in the original implementation. They experience a performance difference between the same implementations, with a factor of about 20 locally. This shows that even though our performance times are weaker, the results relate in computational cost when times are compared internally.

Running the inverse hourglass function, i.e. decapsulation, we experienced the computation cost to be relatively close to the computation cost of the hourglass encapsulation for the same file. For example, the average cost of 5 runs over a 128 MB file for the permutation encapsulation takes 22.73 seconds, opposed to 21.32 seconds for inverse. Same comparison for the butterfly function gives a 613.25 second encapsulation time and 619.46 second inverse time (see appendix A).

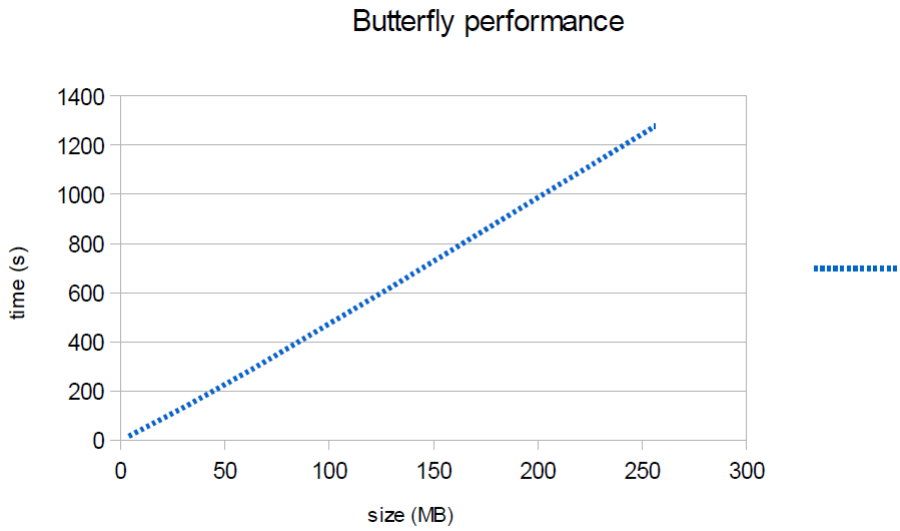


Figure 3.3: *Performance results of our butterfly implementation*

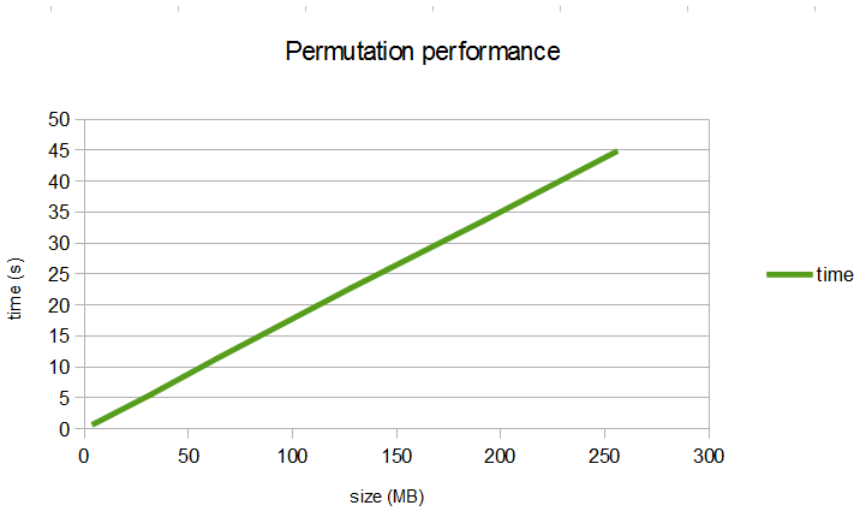


Figure 3.4: *Performance results of our permutation implementation*

3.3 Economical perspective

In this section we will look into the economical numbers which are presented in the original hourglass paper. As the incentives for the hourglass function to hold are much based on the economical arguments, we will discuss and update the price tables used in the paper with current prices, and also evaluate how the economical incentives would relate to our hourglass implementation.

As mentioned, the cloud instance used in the implementation of the original hourglass schemes was an Amazon instance of Elastic Compute Cloud (Amazon EC2) [27]. This is a cloud service which allows the tenant to scale up or down dependent on performance and storage needs for its service. Amazon provides the infrastructure and hardware, and the tenants rent virtual machines and resources which can run operating systems and software of the tenants choice. This is an IaaS deployment model, as presented in section 2.1. The pay plans for the tenants are dynamic, meaning they will only be charged for the resources that are used. As tenants have different needs, different pay plans exist and the best suited can be chosen [30].

The pay plan for an Amazon instance of the EC2 service platform depends on different aspects. One can choose instance after what requirements are needed for your service. One can choose to pay nothing up front and be charged by the hour for as long as the instance is up and running, so called on-demand instances, or one can pay a fee up front, and get discounts on the hourly prices, reserved instances.

It is not mentioned which pay plan is used for the EC2 instance in the experiments with the original paper, other than the hourly fee that was 68 cent per hour. It is reasonable to assume that they used a pay plan which there was no upfront payment, thus they were only charged by hourly use. For such services the current prices range from 7 cent and goes up to $4.60 per hour dependent on the instance of choice.

The different EC2 instances are categorized in different types, with different advantages depending on its usage. Namely, the types vary in focus on CPU, memory, storage and network capacity. In the original paper it is mentioned that they used a quadruple-extra-large high-memory (m2.4xlarge) instance [31] and an EBS storage. Table 3.1 displays the specification of this instance together with the options for current memory optimised instances [30].

	Model	vCPU	ECU	Memory (GB)	Storage(GB)	Price
Instance used in [2]	m2.4xlarge	8	26	68.4	1690(HDD)	68
Current instances	r3.large	2	6.5	15	32 (SSD)	17.5
	r3.xlarge	4	13	30.5	80(SSD)	35
	r3.2xlarge	8	26	61	160(SSD)	70
	r3.4xlarge	16	52	122	320(SSD)	140
	r3.8xlarge	32	104	244	2x320(SSD)	280

Table 3.1: Spesification for memory optimized Amazon EC2 instances. The price (in cents) is per hour.

The concrete performance data for EC2 instances are difficult to identify. The performance of different instances is dependent on the underlying commodity hardware, and computational power is shared among users subscribing to virtual CPU cores. However, Amazon measures the CPU performance in what they call ECU (EC2 Compute Unit). One ECU is defined as 1.0-1.2 GHz on a 2007 server processor [32]. Different EC2 instances are provided with an ECU calculation based on the underlying hardware.

Comparing the specifications for EC2 instance used in the original hourglass paper, with current EC2 instances indicate that the r3.xlarge instance model is the one closest to what they used (m2.4xlarge). It has the same number of vCPU (virtual CPU) and ECUs, and roughly the same amount of memory. This instance is priced at 70 cent per hour usage.

For storage and price references, they use the Elastic Block Store (EBS) from Amazon. EBS is a storage type designed to operate with an EC2 instance. It is basically storage volumes that can be directly attached to an EC2 instance, when needed. It allows for easy and quick scaling of storage.

At the time of their experiments, the price for EBS storage was 10 cent per GB per month. As of now, the price for a standard EBS volume is 5 cents per GB.

We observe that the provided storage capacity for the new EC2 instances are SSD drives, opposed to HDD drives for the old instance. If the hourglass encapsulation was to be stored and executed on a SSD drive, the performance parameters would have changed (discussed in chapter 5). However, we assume a persistent EBS storage connected to the EC2 instance, where encapsulated files and execution of hourglass function is done. HDD drives are still the standard storage hardware for Amazon EBS. We argue this to be a fair assumption, as the storage provided with the EC2 instance is non-persistent. This implies that the stored data will be removed as soon as the instance is taken down, as opposed to a persistent storage like EBS storage [33]. Thus, EBS storage is preferable to use when storage is to be obtained for the unseen future, as with the use of an hourglass function.

Table 3.2 shows the cost of executing the butterfly and permutation function, on the Amazon EC2 instance for different files at the time of the experiments in the original hourglass paper. The rightmost column shows the monthly cost of the EBS storage, for the same files. Comparing the cost of performing the hourglass functions (0.96 or 0.03 cent for 1 GB file) and storing a raw copy of the same file (10 cents per month for 1 GB file), shows that it would not be economical rational by a cloud provider to store an additional raw copy on a monthly basis. The butterfly function could roughly be executed 10 times per month (to retrieve the original data), before the computation cost exceeds the cost of extra storage. For the permutation function, this factor increases to roughly 270 executions per month.

File size	Butterfly		Permutation		EBS storage
	Time	Cost	Time	Cost	Cost (monthly)
1 GB	50.91	0.96	1.64	0.03	10
2 GB	103.96	1.96	3.24	0.06	20
4 GB	213.07	4.02	6.45	0.12	40
8 GB	432.91	8.17	12.73	0.24	80

Table 3.2: Performance (in seconds) and cost (in cents) for butterfly and permutation function, as presented in [2], from F to H. To the right, is the monthly cost (in cents) for storage of the same file on an EBS instance.

Table 3.3 displays performance results compared with current prices for an EC2 instance and EBS storage.

As we assume the same computation performance for the identified EC2 instance in table 3.1 and the instance used in the original paper (same number of ECUs), the timing of the functions are the same. This implies that the 2 cents increase in

the hourly fee of an EC2 instance (from 68 cent to 70 cents), affect computation cost minimally. However, as the EBS storage cost is halved per GB, this implies a noticeable decrease in the number of executions for the economical argument to hold. Still, one could execute the butterfly function roughly 5 times and the permutation function 135 times on a monthly basis. This illustrates that the hourglass functions are strongly dependent on the pricing at cloud providers.

File size	Butterfly		Permutation		EBS storage
	Time	Cost	Time	Cost	Cost (monthly)
1 GB	50.91	0.99	1.64	0.03	5
2 GB	103.96	2.02	3.24	0.06	10
4 GB	213.07	4.14	6.45	0.13	20
8 GB	432.91	8.42	12.73	0.25	40

Table 3.3: Performance (in seconds) and cost (in cents) for butterfly and permutation function with current Amazon pricing. The right column is the monthly EBS price (in cent)

We wanted to do a price estimate for deployment of our implementation in an Amazon EC2 environment.

The performance difference between the local machine we used and the EC2 instance they used are quite noticeable. So a cost comparison based on the prices which are charged for their EC2 instance and our results would be misleading. To make this comparison more realistic, we identified an EC2 instance with more matching performance specifications to the hardware we had available. Recall, our machine runs an Intel Core i7 processor running at 2.8Ghz and a 4 GB memory. This specifications are closest comparable with the m3.medium EC2 instance model [30]. This instance has a 3.75 GB memory, 1 vCPU and is classified with 3 ECU performance. The cost of this instance is 7 cent per hour.

Table 3.4 shows the expected cost, if the implementation done in the work with this thesis was to be applied with the EC2 instance identified above.

For the butterfly implementation we identify that the cost of executing the function exceeds the monthly cost of storing a raw copy in addition to an encapsulated format. Thus, despite the affordable EC2 instance, the performance is too weak, and the double storage problem is not solved with our butterfly implementation.

However, the permutation function seems to hold. Here the hourglass encapsulation could be executed roughly 14 times for the monthly cost of the same amount of storage.

File size	Butterfly		Permutation		EBS storage
	Time	Cost	Time	Cost	Cost (monthly)
4 MB	14.95	0.029	0.69	0.0013	0.02
32 MB	139.63	0.272	5.54	0.011	0.16
64 MB	292.31	0.568	11.42	0.022	0.31
128 MB	613.25	1.192	22.74	0.044	0.63
256 MB	1276.80	2.483	44.87	0.087	1.25

Table 3.4: Performance (in seconds) and cost (in cents) for butterfly and permutation function for the experiments conducted in this thesis on an Amazon EC2 instance. To the right, is the monthly cost (in cents) for storage of the same file on an EBS instance.

As this is a comparison of the results on our local machine and a suggested matching EC2 instance, we realise that the outcome could be somewhat different in a real world scenario. It is difficult to compare the performance of local hardware and the EC2 instances, even with the measurement of ECUs. The number of users sharing the underlying hardware could also affect the performance results of your instance, and vary between the different types of instances. If one rents a high performance instance, the probability of sharing the underlying hardware is reduced compared to a smaller instance [32]. Also, the price per ECU is reduced for high performance instances. In our case, this implies that the cost per ECU increases compared with the price paid in the original work, and the number of users sharing the underlying hardware could also have been increased, affecting the performance if one would have implemented the solution on this EC2 instance.

Chapter 4
Proposed Scheme

In this chapter we propose a new scheme which provides means for a tenant of a cloud provider to verify proper deletion of data. First section defines a new encoding to be applied for the deletion scheme. We also propose methods for the tenant to generate sufficient integrity checks, releasing it from storing the encapsulated data as assumed in the original hourglass paper. The second section presents the deletion protocol, and interactions between tenant and cloud provider, taking advantage of the new encoding, integrity checks and an hourglass function. Last, we discuss alternative approaches and validity of the proposed scheme.

As discussed in section 2.2, the identified cryptographic schemes for deletion in cloud computing applies to scenarios where the data is encrypted and keys are handled by a trusted third party. Their definition of deletion also relies on that removal of the encryption keys are sufficient method. We argued that removal of encryption keys is risky as the unforeseen future could break today's encryption standards. Also, one may not wish to encrypt the intended data. We propose the solution with our method, where the intended data is overwritten when deletion is executed, and proof of this overwriting is provided to the tenant. It is independent of the original encoding of the data. [1]

As for the presented hourglass scheme in chapter 2, the proposed scheme will consist of the same three phases. Namely, encoding phase, encapsulation phase and format checking phase.

4.1 Encoding and integrity checks

For our scheme, a new encoding is defined. We also propose a way of generating sufficient integrity checks, as this is not provided in the original hourglass paper.

[1] The approach in this thesis proofs that a single overwrite of the data has occurred, as the discussion in section 2.2.3 concludes that it is sufficient. An extension of our scheme, could however handle multiple overwrites.

4.1.1 Encoding - Random number generator

The encoding phase will consist of an overwrite of the data intended for deletion. We here present the proposed method of creating the overwrite data.

Before overwriting, the actual data applied to the overwrite has to be produced. This data must be known to the tenant, so he can verify upon receiving responses. A random number generator (RNG) can be taken advantage of in this case. As we shall discuss later, the data is preferable to be random, as opposed to just overwriting with 0's or any other predetermined string of data.

A deterministic or pseudorandom number generator (PRNG) is suitable for our use. In contrast to a non-deterministic RNG which is based on randomness of the physical environments/surroundings, the PRNG bases its randomness on cryptographic algorithms and associated keying material [34]. The cryptographic algorithms used for PRNG can be of different choices. In the NIST recommendation in [35], they discuss four different approaches to PRNG, namely based on Hash, HMAC, block ciphers and number theoretic problems.

Independent on choice of method for the PRNG, all PRNGs shall have a seed. The seed is used to initiate the PRNG and should be based on a entropy source with entropy equal to or greater than the security requirements of the intended application [35]. The entropy source in software implementations are often based on system date and time or other processor values.

The choice of method for the PRNG depends on its usage and security requirements of the intended application.

Why random

One could think that the data to use for the overwriting could just as simply have been all 0's or 1's, as it often is when overwriting is applied with deletion. However, in the case of proving that the overwriting has occurred, one quickly realise that this would not apply. First of all, the data that will be written over the original data cannot be predetermined. If this was the case, the prover could simply pre-compute the correct responses for the format checking phase. In addition, for the permutation function especially, if the data applied to the function is all 0's or 1's, the result of executing the permutation function would inflict no difference on the result. It would still all be a number of 0's or 1's, as it is only a permutation of the symbols. The same, more or less, applies to the butterfly function if a block cipher in ECB mode is used as the encryption method. The result would not be all 0's or 1's, but each block of the encapsulation would result in the same output, as the same key is used for the cipher.

4.1.2 Integrity checks

As mentioned in chapter 2, the original hourglass paper omits, for simplicity, to specify an integrity check method on blocks of H, and assumes tenants knowledge about the entire encapsulated file for verification on responses. We explain two methods which take advantage of HMAC and Merkle tree to free the tenant from storing entire H.

MACs

Message authentication codes (MAC) are used for integrity and authenticity checks of data. The usage of hash MACs generate a compact output based on the input of a secret key and an arbitrary length data. HMAC is one such way of generating integrity checks.

The HMAC definition is presented in equation 4.1, as described in the RFC [36].

$$HMAC(K, m) = h((K \oplus opad)|h((K \oplus ipad)|m)) \tag{4.1}$$

- h is the underlying cryptographic hash function
- K is a secret key
- m is the message to hash
- *opad* is an outer padding of the key
- *ipad* is an inner padding of the key
- $\oplus$ is an exclusive or operation
- $|$ is a concatenation

The output length of the HMAC is of the same length as the underlying hash function, h. h could be any approved hash function. Thus, in case of SHA-1 usage, the output will be of length 160 bit. In our presentation we refer to the message m as a block of the file intended for verification. The HMAC is applied to each block m of the file, consisting of n blocks. The integrity checks would require a storage of $160 * n$ bit, in case of SHA-1 usage. One could use a truncated version of HMAC to reduce the storage cost of the verification blocks. This simply implies that the t leftmost bits of the computed output is stored as the HMAC result [36]. One should however not truncate the output more than to a 32 bit output [37]. The degree of truncating is dependent on the security requirements of the intended application.

Upon verifying the original data, the verifier must be in possession of the secret key K, the original block m and the corresponding HMAC computation.

Merkle tree

We propose the use of a Merkle tree to create integrity checks for the encapsulated file H. Figure 4.1 shows the structure of a Merkle tree on a file with 8 blocks. The blocks in the file to be verified are denoted by $H_1 \dots H_8$. The leaf nodes in the Merkle tree is an execution of the hash function h for each of the file block, denoted by $h(H_1) \dots h(H_8)$. The next steps are executed with the same function h, where two sibling nodes are concatenated and hashed forming the parent node, i.e. $h(H_{1-4}) = h(h(H_{1-2})|h(H_{3-4}))$. Upon completion of the Merkle tree, the root node is the result of multiple hashes, with ancestry in the original file blocks.

Upon verification of one of the file blocks, the root node and the Merkle tree nodes associated with the file block to be verified must be available. The associated Merkle tree nodes are used to compute a "copy" of the root node. For example, on a request for verification of block H_2, the H_2 block itself and $h(H_1)$, $h(H_{3-4})$, and $h(H_{5-8})$ nodes must be provided to the verifier. Using function h and the intermediate Merkle tree nodes, the verifier can compute the copy root block:

$$h(H_{1-8}) = h(h(h(h(H_1)|h(H_2))|h(H_{3-4}))|h(H_{5-8}))$$

The copy and the originally computed root node should be identical for the verification to be successful.

The cost of computing the verification of a block in a Merkle tree is dependent on the number of leaf nodes. One invocation of h is executed on the challenged block itself. If N is the number of leaf nodes, h must additionally be applied $log_2 N$ times using the associated nodes of the block to be verified.

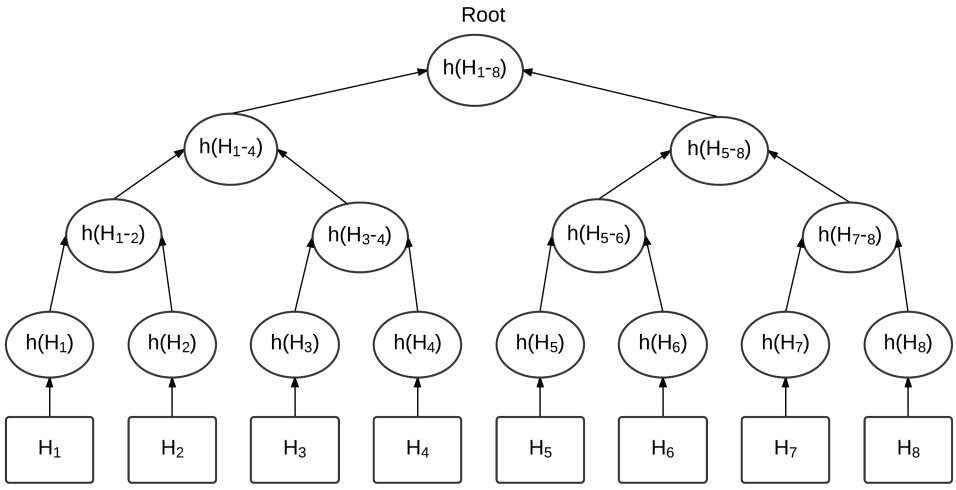


Figure 4.1: *Merkle tree structure*

4.2 The protocol

In this section we elaborate on the interactions between the tenant and cloud provider in execution of the deletion scheme. The three phases, namely encoding, encapsulation and verification, are presented in their respective order. Figure 4.2 shows the phases and the interaction between tenant and cloud provider.

4.2.1 Executing the overwrite - encoding (phase 1)

The encoding phase will consist of an overwrite of the data intended for deletion. We here present the usage of a PRNG, as presented in section 4.1.1, to produce and exchange the overwrite data between the tenant and cloud provider.

The necessary seed S to initiate the PRNG is obtained by the tenant. Further, the tenant transfers the seed together with a file indication F_{id} about which file to overwrite. Upon reception, the cloud provider initiates a $PRNG$ using S and F_{id} as input, generating the pseudorandom data, PRN. The length of the PRN is decided by the length of the indicated file. The *overwrite* function executes the actual overwriting, resulting in the encoding G. It takes as input the PRN and the file indication F_{id}. The same data is generated on the tenant side, preparing for the hourglass encapsulation in the next phase.

By sending the seed for the PRNG, communication cost is greatly reduced compared to generating the random data at the tenant and sending it to the provider.

Client		**Cloud Provider**
Phase 1		
$S \leftarrow generateSeed(entropy\ source)$	$\xrightarrow{S, F_{id}}$	
$PRN = PRNG(S, F_{id})$		$PRN = PRNG(S, F_{id})$
		$G = overwrite(PRN, F_{id})$
Phase 2		
$H \leftarrow hourglass(PRN)$		$H \leftarrow hourglass(G)$
$M \leftarrow merkle(H)$	$\xrightarrow{M}$	
Keep root M_r, discard H and M		
Phase 3		
$\{c_i \leftarrow challenge\}_{i=1}^{t}$		
Start timer	$\xrightarrow{{c_i}_{i=1}^{t}}$	
		$\{M_i = retreiveNodes(M, c_i)\}_{i=1}^{t}$
		$\{r_i = respond(H, c_i)\}_{i=1}^{t}$
	$\xleftarrow{\{r_i, M_i\}_{i=1}^{t}}$	
Stop timer		
$\{rootCopy_i = calculateMerkle(r_i, M_i)\}_{i=1}^{t}$		
$Accept\ iff \{verify(rootCopy_i, M_r) = true\}_{i=1}^{t}$, and $\tau \leq T$		

Figure 4.2: Deletion protocol

Use of an hourglass function ensures the provider does not calculate responses on-the-fly. However, in section 4.4 we look at the implications of actually sending the random data to the provider.

4.2.2 Encapsulate overwrite - hourglass (phase 2)

As the overwrite encoding G is executed at the cloud provider, the encapsulation phase is initiated. The tenant also generates the random data from the seed, for the reason that to compute the integrity checks, it must also possess the hourglass encapsulated data H at some point.

The hourglass function is executed on the encoded format G, for both the tenant (uses the pseudorandom number which is the same as G) and the cloud provider. As with the original hourglass scheme discussed in chapter 2, the *hourglass* function is applied which is a block-by-block or symbol-by-symbol encapsulation of encoding G. This results in the hourglass encapsulation H with n blocks, $H_1 \dots H_n$.

After this the integrity checks can be calculated on the tenant side. Figure 4.2 illustrates the scenario where a Merkle tree is used. The tenant calculates the Merkle tree M on the blocks of H, using the *merkle* function. The root M_r of M is kept in the tenants storage, and M is sent to the cloud provider. Now the tenant can discard H and M.

If one uses the HMAC mentioned in section 4.1.2 for verification method, the tenant must calculate the HMAC for each of the n blocks of H. The calculated results could then be transferred to the cloud provider for storage. As with the Merkle tree method, the tenant is now free of storing H, and can discard it and the HMAC calculations, while keeping the secret key, K. The cloud provider is not able to forge HMAC calculations, as it is not in possession of K.

4.2.3 Prove the overwrite - challenge-response (phase 3)

After G has been encapsulated, taking the form H, and integrity checks on H has been generated and exchanged, the tenant can start challenging the cloud provider for proof of overwriting. This is a process which can be executed multiple and at random times, opposed to the previous phases, which in the normal case are executed only once.

As the hourglass encapsulation H should be stored at the cloud provider, the tenant should expect to see acceptable responses within T time, as explained in chapter 2.

In figure 4.2 we see the tenant produces t challenges c, which could be random selected block indexes of H, thus $c \in \{1 \dots n\}$ [2]. As the t challenges are sent to the cloud provider, a timer is initiated. On the cloud provider side, the $retreiveNodes$ function is invoked for each challenge taking the challenge and Merkle tree M as input. Remember, M was provided to the cloud provider in the encapsulation phase. This produces M_i which is a list of the nodes in M associated with the challenged block of H. In addition, the *respond* method is applied in the same manner as the original hourglass scheme. Based on c_i it retrieves the challenged block from H, resulting in the response block r_i. So for the single challenge c_i, a response is constituted by M_i and r_i which is the challenged block of H. This is sent back to the tenant.

Upon retrieval, the tenant stops the timer, and checks the elapsed time τ. It also applies the $calculateMerkle$ function for each of the t responses, which takes the provided challenged block r_i and Merkle nodes M_i as input. The nodes of M_i and block from r_i are used to compute a Merkle tree root M_i'. Each computed M_i' is passed to the $verify$ function, where it is compared with the original Merkle tree root M_r. If all invocations of $verify$ are accepted and $\tau \leq T$, the challenge-response

is accepted, and the tenant can rely on that the cloud provider has overwritten the intended data.

In usage of a HMAC for verification, the challenge sent to the cloud provider would be on the same format, thus $c \in \{1 \dots n\}$. However, the calculated response from the cloud provider would instead of the Merkle tree nodes M_i, consist of the corresponding HMAC calculation for the challenged block in c_i. Remember, the cloud provider was provided with all the calculated HMACs for the encapsulated file H in the encapsulation phase, while the tenant is in possession of the secret key K. In addition to the HMAC block for each challenge, the encapsulated block is provided to the tenant within r_i. Upon receiving the responses from the cloud provider, the tenant calculates the HMAC on the received response blocks r_i using K. Each calculated HMAC is compared with the original HMAC provided in the response. If they are equal and $\tau \leq T$, the challenge-response is accepted.

4.3 Choice of hourglass function

On the question about which hourglass function to take advantage of in the deletion scenario, one must consider a few things. In the original work conducted with the hourglass schemes, arguments are presented through economic analysis related to the performance of the hourglass functions and the needs of the tenant, when choosing hourglass function. They state that the butterfly function is better suited for scenarios where the tenant wishes to store data for archival purposes. This is related to the cost of computing the butterfly function when access to the plain plain text is necessary. This cost is high compared to the cost of computing the permutation based function. With a high computation cost, the number of executed hourglass functions while the economical argument still holds, are decreased.

Relating this analysis to the deletion scheme, we can assume that as soon as the encapsulation of the overwritten data is executed, one would never apply the reverse hourglass function to retrieve the overwritten data again. Thus, both the butterfly and permutation function could be applied with no clear advantage in terms of the above arguments. However, using the characteristics identified in table 2.1, one could argue the permutation function better suited, based on its simplicity with no need for cryptography and the low computation cost.

4.4 Alternative to providing a seed to the cloud provider

Providing the cloud provider with the seed for generation of random data in the proposed scheme, saves the participating parts for significant communication costs. The alternative is to leave the random data generation to the tenant, and transfer the data to the cloud provider. Thus, the extra communication cost would be

proportional with the size of the intended data for deletion. However, with such an approach, we identify that the need for an hourglass function might not be necessary.

The point of applying the hourglass function is that the cloud provider should not be able to calculate the correct response on-the-fly when challenged. This imposes that in the case of providing the seed to the cloud provider, the hourglass function is needed, as it could generate the random number on the fly when in possession of the seed. However, providing the cloud provider with the actual random data intended for overwriting, gives the cloud provider no means of generating the random data. For this reason, the hourglass function could be redundant. Also random data can not be compressed [38], leading to that the cloud provider must store the data as it is, to be able to respond to challenges.

4.5 Economic arguments

Opposed to the scenario where data is to be stored for the unseen future, as is the case for the original hourglass scheme, when deletion is the scenario, the occupied hard drive space of the cloud provider is to be free as soon as possible. This reduces the economical arguments to depend solely on the cost of an additional copy in storage, rather than looking at the computation cost of the hourglass function compared with the storage cost.

If the tenants would no longer be obligated to pay storage cost after the hourglass encapsulation and challenge-response had taken place, the cloud provider would be the one that would be inflicted with additional costs if it stores a copy and does not release the storage space. However, one can inflict the cloud provider to keep the encapsulated data for some period of time, giving the tenant the opportunity to challenge the provider for at random for some time. The cost in this time period could be paid by the tenant, but at least it imposes a minimum cost on the cheating cloud provider, as it additionally must keep the copy for the time period, which gives no income.

Chapter 5
Practical challenges

As we in the above analyses have assumed some simplifications, we here highlight these and try to discuss implications this could have on real world implementation in a cloud environment. These implications apply to the original hourglass applications and our new proposal for deletion in chapter 4. We also discuss a different approach to how one could execute the proposed deletion scheme.

5.1 Hardware issues

We identify that the hourglass functions are dependent on the underlying hardware which they are executed on, in terms of computation capacity, type of storage medium and cloud pricing for the hardware resources. We spilt the hardware issues in two - acquiring information about the underlying hardware, and how the development in storage hardware can affect the validity of the hourglass function.

5.1.1 Acquiring hardware information

The permutation and butterfly hourglass functions are both time based functions with a resource bound on storage. To be able to determine the timing bound T, specifications on read and seek time must be obtained for the permutation function and the upper limit on ϵ (see section 2.3.3) for the butterfly function. To obtain such information, ideally some knowledge about the underlying storage hardware used by cloud providers is preferable. The process of obtaining such information seems difficult.

As far as we have identified, cloud providers are reluctant to reveal to much about their infrastructure. Again, this illustrates the lack of transparency by cloud providers, making it difficult to obtain trust relationships to tenants. In addition, if one manages to identify some of the underlying hardware, the same type of instance can run on different hardware, while achieving much the same performance results and purpose to tenants.

For the hourglass function however, the necessary parameters to apply could be obtained by approximating the necessary values. By using average read and seek values for enterprise storage hardware, one can assume sufficient parametrization, as done in the original paper. By applying multiple challenges on the encapsulated file H, one could smooth out the approximation variance. This would also smooth out the variation in network latencies.

5.1.2 How will SSDs affect the hourglass function?

As pointed out in section 2.3.3, the resource bound on the permutation function is strongly dependent of the inefficiency of random reads on ordinary rotational hard drives. As solid state drives (SSD) are increasing in usage, this could inflict new challenges to the usage of the hourglass function. SSDs are designed in a manner which among others makes them less dependent on reading data in a sequential order, with thought to performance. Input/output per second (IOPS) is a measurement of the number of operations a hard drive can achieve per second, such as random accesses. According to [39], the random read advantages of SSDs are in the magnitude of 100 over normal rotating hard drives (HDD: 100 IOPS, SSD 10,000 IOPS). However, experiments conducted with different SSDs[40] illustrate that a random read still exposes the SSD for some extra read time compared to a sequential read.

Adjusting for the performance difference with SSDs, we believe the hourglass function with resource bound on storage still would apply. The RSA function would not have been affected by SSD storage, as the resource bound is on computation.

5.2 Storage allocation

There is also a challenge in how the cloud provider allocates its storage resources to tenants. When storage is allocated, the question is if the tenants are assigned their own storage or partition on a hard drive, or if their data simply stored where it is best suitable for the cloud provider with no dedicated storage for the tenant. As with information about the underlying hardware, this is information that is challenging to get clear insight on.

Allocation of storage, could affect the performance of the hourglass function. Both when it comes to computing the actual encapsulation and on responding to challenges. In the extreme scenario, one could imagine that every block of the intended file is scattered out over different disk locations, or even disks. On producing the hourglass encapsulation, each of the scattered blocks must be gathered and processed to be able to execute the encapsulation. This implies, a seek and read operation for each block. This clearly inflicts a time and resource consuming process compared with the assumed scenario, where the data is stored in a sequential order.

As for when the file is in an encapsulated format, and the above scenario is applied, the response time on challenges and work of the cloud provider would increase.

The above scenario is not likely to occur in real world cloud services, as it would be inefficient for the cloud provider itself to manage storage like this. But even with a smaller degree of file scattering, some extra costs must be counted for. We can assume that larger files are more affected by such problems, as they necessarily require more storage allocation. With the lack of insight in how storage is given to tenants, it is worth to mention.

5.3 Alternative deletion approach

Double pass overwrite

The proposed scheme in chapter 4 inflicts the cloud provider with the work of executing the hourglass function on the overwritten data to provide proof to the tenant. The economical arguments would apply for the cloud provider with a single invocation of the function. However, one could require the cloud provider to do a double pass with the hourglass function, only partially overwriting the data in the first round.

In this manner, a cheating cloud provider would be inflicted an additional cost. In addition, the tenant would be provided with stronger assurance that the intended data is overwritten.

Figure 5.1 illustrates the double pass overwrite on a file, including interactions with the tenant. The data is first partially overwritten. This is illustrated with the blocks F_2, F_4 and F_6 taking the form O_2, O_4 and O_6 in the figure, respectively. Then an hourglass encapsulation is executed on this first overwrite. Challenges on the hourglass encapsulation are sent out by the tenant, and responses produced by the cloud provider. The responses will include data from both the overwriting and the original file.

On acceptance from the tenant, the next round is initiated. Here, the remaining data is overwritten ($F_1 \rightarrow O_1,\ F_3 \rightarrow O_3,\ F_5 \rightarrow O_5$ *and* $F_7 \rightarrow O_7$), and a new hourglass encapsulation is executed. The tenant challenges again on this encapsulation, which now solely holds information about the overwrite data.

The intermediate challenge-response exchange provides the tenant with an indication that the intended data, is the data that is actually being overwritten. The only way for the cloud provider to deliver sufficient proof, is to locate the intended data for overwriting and then executing it. As opposed to the single overwrite scenario, where

a cheating or "lazy" cloud provider, could execute the overwrite on any disk space best suited (e.g. the closest available free space on the time of being challenged). In addition to the extra cost the economical arguments still apply for the double pass overwrite approach.

The proof of the double overwrite could be an execution of the hourglass function on a copy of the original data, in the case of an adversary cloud provider. This however, inflicts the adversary with additional cost of copying and writing the data to a separate location, before overwriting the data.

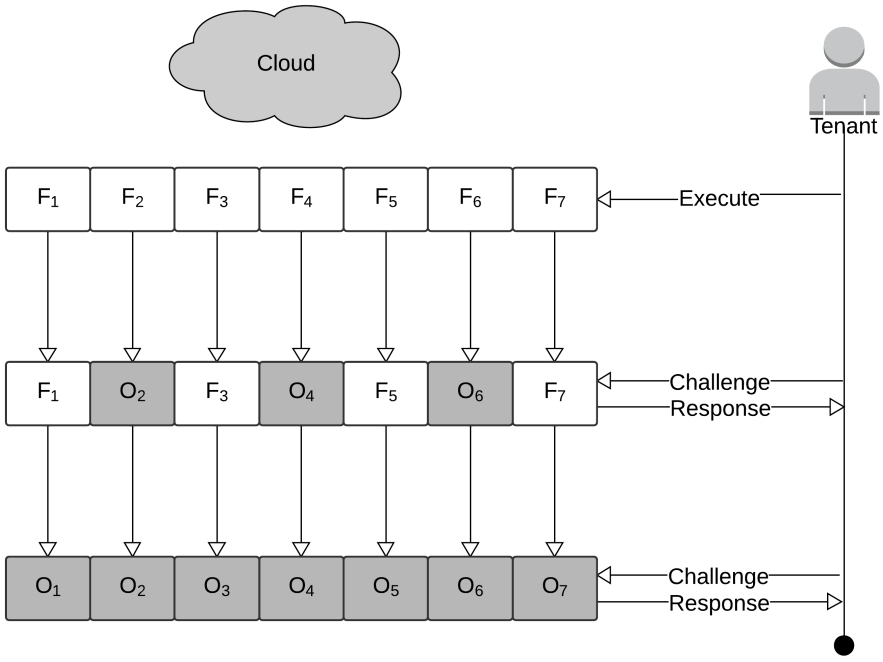


Figure 5.1: *Double pass overwrite*

5.4 The price development of cloud storage and processing

As the hourglass functions are dependent on pricing from the cloud providers, the price development is important for the sustainability of the hourglass based schemes. From the time of writing the original paper, we observed that the storage on cost is halved, while the cost of computation has changed little. We identified that this impacts the economical arguments in the original paper for using the hourglass function, in a negative manner (halved the number of hourglass function executions,

for the economical argument to still hold). Analyses reveal, that in general, the price reductions are accelerating on all services in cloud computing. With increasingly number of cloud providers, the competition provides better prices for the tenants. The storage and computation pricing are both decreasing, however, the price reduction on computation over all services (computation, storage, DB, bandwidth and others) for the strongest providers in the market, constituted 42% of the reductions in 2013. Storage constituted 23% [5] of the reductions. For the year 2012 the scenario was opposite, storage constituted 36%, while computation was on 23%. For the hourglass function, a balance between computation cost and storage cost is preferable. As long as the reduction does not accelerate too much on storage, the sustainability is promising.

Chapter 6
Conclusion

In this thesis we have looked at existing cryptographic schemes intended for cloud environments, and particularly directed focus on the hourglass scheme. We argue the necessity for cryptographic research within cloud computing, as businesses are reluctant to convert to cloud technology. This scepticism we believe, among others, are due to lack of transparency by the providers, leaving the cloud tenants with little knowledge of how their data is actually handled. The existing schemes apply to different categories and area of use.

We did a practical and economical analysis of the hourglass scheme, identifying that implementation details such as used hardware and software choices can drastically affect its validity, together with price development of cloud services. We used achieved implementation results to estimate performance cost in a cloud environment, and observed that the cost ratio between storage and execution would reduce the validity for our implementation, compared with achieved results in the original paper. Also the development in storage cost for Amazon, implied that the current validity of the hourglass functions were halved compared with the time of writing the original paper.

Through research of different cryptographic schemes, we identified deletion as a category with potential for improvements. This, together with the idea of using the hourglass functionality, was the motivation for proposing a new deletion scheme. Taking advantage of the economical arguments in the original hourglass paper, the proposed scheme takes a different approach on deletion then the identified existing schemes. Arguing that key removal might not be secure for the unseen future, we approach deletion by overwriting the intended data. Requiring an hourglass encapsulation on the overwritten data, the tenant applies a challenge response protocol to receive proof of overwriting through the encapsulated file.

6.1 Future work

In the work with this thesis we did not conduct practical work specifically devoted to the proposed scheme. An interesting task for future work could be to make an implementation of the proposed scheme in a cloud environment, to evaluate its validity in practice. At the same time, one could get better insight into the practical challenges related to transparency obstacles from the cloud providers. Documenting these challenges, based on real world observations could be valuable, not only for the purpose of the hourglass functions and proposed scheme, but for research regarding cloud computing in general.

As we mentioned in section 4.4, the alternative approach by transferring the random data to the cloud provider, could make the hourglass function redundant in the deletion scenario. An interesting study could be to identify other approaches to realize hourglass like capabilities, e.g. by usage of puzzles (proof-of-work functions).

References

[1] Ateniese Giuseppe, Burns Randal, Curtmola Reza, et al. Provable data possession at untrusted stores. *Proceedings of CCS*, 10:598–609, 2007.

[2] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass schemes: How to prove that cloud files are encrypted. 2012.

[3] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass schemes: How to prove that cloud files are encrypted, slides. http://www.arijuels.com/wp-content/uploads/2013/09/vDJOR+12slides.pptx, 2012.

[4] Peter Mell and Timothy Grance. The NIST definition of cloud computing (draft). *NIST special publication*, 800(145):7, 2011.

[5] Bret Clement. Cloud Price Reductions: A Definitive Analysis of 2013 Trends. http://www.rightscale.com/blog/cloud-industry-insights/cloud-price-reductions-definitive-analysis-2013-trends, March 2014.

[6] Unisys. Cio unisys 2014 benchmark tool refresh. Powerpoint received on mail by Jim Carr, Media contact at Unisys, 2014.

[7] Unisys. CIOs Are Moving More Organizational Information into the Cloud Even as Security Concerns Persist, Unisys Research Reveals. http://www.unisys.com/unisys/news/detail.jsp?id=1120000970028410151, March 2014.

[8] Larry Barrett. When it comes to the cloud, CIOs, CEOs prefer to keep it private. http://www.zdnet.com/when-it-comes-to-the-cloud-cios-ceos-prefer-to-keep-it-private-7000024943/, January 2014.

[9] Matthew Green. On the NSA. http://blog.cryptographyengineering.com/2013/09/on-nsa.html, September 2013.

[10] Matthew Green. A note on the NSA, the future and fixing mistakes. http://blog.cryptographyengineering.com/2013/09/a-note-on-nsa-future-and-fixing-mistakes.html, September 2013.

[11] Joseph Menn. On the NSA. http://www.reuters.com/article/2013/12/20/us-usa-security-rsa-idUSBRE9BJ1C220131220, December 2013.

[12] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 213–222. ACM, 2009.

[13] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. ACM, 2007.

[14] Kevin D Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54. ACM, 2009.

[15] Kevin D Bowers, Marten van Dijk, Ari Juels, Alina Oprea, and Ronald L Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 501–514. ACM, 2011.

[16] Zhan Wang, Kun Sun, Jiwu Jing, and Sushil Jajodia. Verification of data redundancy in cloud storage. In *Proceedings of the 2013 international workshop on Security in cloud computing*, pages 11–18. ACM, 2013.

[17] Yang Tang, Patrick PC Lee, John CS Lui, and Radia Perlman. Fade: Secure overlay cloud storage with file assured deletion. In *Security and Privacy in Communication Networks*, pages 380–397. Springer, 2010.

[18] Radia Perlman. File system design with assured delete. In *Security in Storage Workshop, 2005. SISW'05. Third IEEE International*, pages 6–pp. IEEE, 2005.

[19] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX Security Symposium*, pages 299–316, 2009.

[20] Joel Reardon, David Basin, and Srdjan Capkun. Sok: Secure data deletion. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 301–315. IEEE, 2013.

[21] Sarah M Diesburg and An-I Andy Wang. A survey of confidential data storage and deletion methods. *ACM Computing Surveys (CSUR)*, 43(1):2, 2010.

[22] Bill Nelson, Amelia Phillips, and Christopher Steuart. *Guide to computer forensics and investigations*, chapter Working with Windows and DOS systems. Cengage Learning, 2010.

[23] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, 1996.

[24] Craig Wright and Dave Kleiman. Overwriting hard drive data: The great wiping controversy. In *Information Systems Security*, pages 243–257. Springer, 2008.

[25] Wikipedia. Block cipher mode operation - Cipher-block chaining (CBC). http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher-block_chaining_.28CBC.29.

[26] Ian Atkin. Getting the hang of IOPS. http://www.symantec.com/connect/articles/getting-hang-iops, March 2011.

[27] Amazon. Amazon ec2. http://aws.amazon.com/ec2/, 2014.

[28] Andrew Kuchling and Dwayne C. Litzenberger. Pycrypto - the python cryptography toolkit. https://www.dlitz.net/software/pycrypto/, 2014.

[29] Wikipedia.org. Aes electronic codebook. http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Electronic_codebook_.28ECB.29, 2014.

[30] Amazon. Amazon instances and princing. http://aws.amazon.com/ec2/pricing/, 2014.

[31] Amazon. Amazon announcement. http://aws.amazon.com/about-aws/whats-new/2009/10/27/announcing-amazon-ec2-high-memory-instances/, 2009.

[32] Alexis Lê-Quôc. Are all AWS ECUs created equal? https://www.datadoghq.com/2013/08/are-all-aws-ecu-created-equal/, August 2013.

[33] Naor Weissmann. Amazon AWS EC2 storage types. http://unixsystems.blogspot.no/2012/02/amazon-aws-ec2-storage-types.html, February 2012.

[34] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management-part 1: General (revision 3). In *NIST special publication*. Citeseer, 2012.

[35] Elaine B Barker and John Michael Kelsey. *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2012.

[36] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. RFC 2104: HMAC: Keyed-hashing for message authentication, February 1997. *Status: INFORMATIONAL*, 2006.

[37] Quynh Dang. *Recommendation for applications using approved hash algorithms*. US Department of Commerce, National Institute of Standards and Technology, 2008.

[38] Matt Mahoney. Data compression explained. *mattmahoney. net, updated May*, 7, 2012.

[39] buildcomputers.net. SSD vs HDD - Sould You Buy A Solid State Drive or Hard Disk Drive. http://www.buildcomputers.net/ssd-vs-hdd.html.

[40] Anand Lal Shimpi. The Seagate 600 & 600 Pro SSD review. http://www.anandtech.com/show/6935/seagate-600-ssd-review/5, May 2013.

Appendix A

Performance results

The performance results (in seconds) over five executions of the permutation and butterfly hourglass functions for different file sizes.

File size	Permutation		Butterfly	
	Encapsulate	Inverse	Encapsulate	Inverse
4 MB	0,697394314	0,666209657	14,92680497	15,3380162
	0,691927431	0,652818672	14,96035727	15,2689192
	0,67665953	0,655059607	14,92381328	15,22056058
	0,695186738	0,647783993	14,99015246	15,33535333
	0,686200639	0,648388859	14,99136732	15,26003321
32 MB	5,544139772	5,402710896	139,3902335	140,0684302
	5,574588692	5,221452557	141,2436249	142,5586798
	5,485434843	5,228120372	139,4000689	141,5985227
	5,608553895	5,193892693	139,6030232	141,3280514
	5,524504744	5,209362215	138,522112	141,4515545
64 MB	11,40846555	10,94102197	291,2359402	296,2010653
	11,3335297	10,50232415	293,5572391	296,5845441
	11,67670169	10,87228251	292,0286903	296,4798467
	11,32803532	10,53423868	291,6837046	298,4865094
	11,3655487	10,58208719	293,0548996	296,5738065
128 MB	22,76962916	21,43372472	614,1207461	621,1951013
	22,82438634	21,56498233	612,3571768	616,8216064
	22,66354972	21,16572686	610,4074479	620,6524882
	22,60830498	21,2881868	614,71746	622,0291119
	22,83060875	21,15386711	614,6427055	616,5889941
256 MB	45,56441315	42,52006716	1266,254056	1286,345805
	44,90993075	42,5958884	1275,202061	1266,825122
	44,59273061	42,6580115	1281,218515	1262,750174
	44,88791912	41,77801524	1283,819445	1268,765749
	44,39368835	42,32068705	1277,476192	1261,263452

Appendix B
Source code

B.1 Butterfly source code

B.1.1 Butterfly encapsulation

```
1  '''
2  Created on 18. mars 2014
3
4  @author: olerasmu
5  '''
6
7  import math
8  import os
9  import timeit
10 import sys
11 from Crypto.Cipher import AES
12 from _hashlib import new
13
14
15
16 # Works with files which have a number of blocks n that is a
       power of 2
17 class Butterfly(object):
18
19     def __init__(self, filepath=None):
20         self.filepath = filepath
21
22     fileTab = []
23     butterflyTab = []
24
25     #Return the size of file in bytes
26     def fileSize(self):
27         tempFile = open(self.filepath, 'r')
28         tempFile.seek(0,2)
29         size = tempFile.tell()
30         tempFile.seek(0,0)
31         tempFile.close()
32         return size
33
34     #Slit the file into blocks of 128 bit (16*8 bytes)
35     def blockifyFile(self):
36         with open(self.filepath, 'rb') as newfile:
37             byte = newfile.read(16)
38             while byte:
39                 self.fileTab.append(byte)
40                 byte = newfile.read(16)
41
42
43
44     fileTabTest = []
```

```
45
46      def fileifyBlocks(self, filepath, butterflyTab):
47          with open(filepath, 'ab') as hourglass_file:
48              for byte in butterflyTab:
49                  hourglass_file.write(byte)
50
51      #Initialize the butterfly function. Variable j is
            controlled from here
52      def initiateButterfly(self, d, n):
53          self.butterflyTab = [None]*n
54          #print ("initiate")
55          for j in range(1, d+1):
56              #print "this is j: ", j
57              self.executeButterfly(j, n)
58
59      #Execute the butterfly algorithm
60      def executeButterfly(self, j, n):
61          for k in range(0, int((n/math.pow(2, j))-1)+1):
62              if j == 1:
63                  for i in range(1, int(math.pow(2, j-1))+1):
64                      indexOne = int(i+k*math.pow(2, j))-1
65                      indexTwo = int(i+k*math.pow(2, j)+math.pow
                            (2, j-1))-1
66                      #self.count += 1
67                      self.w(self.fileTab[indexOne], self.fileTab
                            [indexTwo], indexOne, indexTwo)
68
69              else:
70                  for i in range(1, int(math.pow(2, j-1))+1):
71                      indexOne = int(i+k*math.pow(2, j))-1
72                      indexTwo = int(i+k*math.pow(2, j)+math.pow
                            (2, j-1))-1
73                      self.w(self.butterflyTab[indexOne], self.
                            butterflyTab[indexTwo], indexOne,
                            indexTwo)
74
75
76      #Defines the cipher which should be used in the
            cryptographic operation
77      cipher_machine = AES.new(b'This is a key123', AES.MODE_ECB)
78
79      #Cryptographic operation w
80      def w(self, block_one, block_two, indexOne, indexTwo):
81
82          #This part is for splitting and combining blocks (
                alternative to interleaving)
83          new_block_one = block_one[len(block_one)/2:] +
                block_two[len(block_two)/2:]
84          new_block_two = block_one[:len(block_one)/2] +
                block_two[:len(block_two)/2]
85
86          new_block_one = self.cipher_machine.encrypt(
                new_block_one)
87          new_block_two = self.cipher_machine.encrypt(
                new_block_two)
88
89          self.butterflyTab[indexOne] = new_block_two
90          self.butterflyTab[indexTwo] = new_block_one
91
92      #Start the hourglass encapsulation based on the valid file
            path input
93      def start(self, input_filepath):
94          bf = Butterfly(filepath=input_filepath)
95
96          #Start the timer
97          start = timeit.default_timer()
98          bf.blockifyFile()
99                  #Defines the parameters based on the file
                        intended for execution
100         n = len(bf.fileTab)
101         d = int(math.log(n, 2))
```

```
102             bf.initiateButterfly(d, n)
103
104             #Uncomment this to write the bf table to file.
105             bf.fileifyBlocks('bf_hourglass.txt', bf.butterflyTab)
106
107                     #Stop the timer
108             stop = timeit.default_timer()
109
110             #Write results to 'resultfile.txt'
111             writeable = "File: " + str(self.filepath) + "size: " +
                      str(self.fileSize()) + "time: ", str(stop-start)
112             writeable = str(writeable)+"\n"
113             with open('resultfile.txt', 'a') as resultfile:
114                 resultfile.write(writeable)
115
116         #Read the input from console (filepath). Checks if input is
                a valid filepath.
117         def input(self, input_filepath):
118             try_again = ''
119             if os.path.isfile(input_filepath):
120                 bf.filepath = input_filepath
121                 bf.start(input_filepath)
122             elif not os.path.isfile(input_filepath):
123                 try_again = raw_input("Provided filepath did not
                          exist, try again or write quit to exit...")
124                 if try_again == 'quit':
125                     sys.exit('You choose to quit')
126                 else:
127                     self.input(try_again)
128
129
130
131     bf = Butterfly()
132
133     input_filepath = raw_input("Provide filepath...")
134
135     bf.input(input_filepath)
```

B.1.2 Butterfly decapsulation

```
1   '''
2   Created on 25. mars 2014
3
4   @author: olerasmu
5   '''
6   import math
7   import os
8   import timeit
9   from Crypto.Cipher import AES
10  class Bf_reverse(object):
11
12      def __init__(self, filepath = None):
13          self.filepath = filepath
14
15
16      fileTab = []
17      butterflyTab = []
18
19
20      def fileSize(self):
21          tempFile = open(self.filepath, 'r')
22          tempFile.seek(0,2)
23          size = tempFile.tell()
24          tempFile.seek(0,0)
25          tempFile.close()
26          return size
27
28      #Split the file into blocks of 128 bit (16*8 bytes)
29      def blockifyFile(self):
30
```

```
31          with open(self.filepath, 'rb') as newfile:
32
33              bytes = str(newfile.read(16))
34
35              while bytes:
36                  self.butterflyTab.append(bytes)
37
38                  bytes = str(newfile.read(16))
39
40
41
42      def fileifyBlocks(self):
43          with open('rev_hourglass2.txt', 'ab') as
                rev_hourglass_file:
44              for byte in self.fileTab:
45                  rev_hourglass_file.write(byte)
46
47      #Save blocks from fileTab to a string
48      def stringifyBlocks(self):
49          plain_text_string = ""
50
51          for byte in self.fileTab:
52              plain_text_string = plain_text_string + byte
53
54          return plain_text_string
55
56
57
58      #Initialize the butterfly function. Variable j is
            controlled from here
59      def initiateButterfly(self, d, n):
60
61          self.fileTab = [None]*n
62
63          for j in range(d, 0, -1):
64              self.executeButterfly(j, n, d)
65
66      #Execute the butterfly algorithm
67      def executeButterfly(self, j, n, d):
68
69          for k in range(int((n/math.pow(2, j))-1), -1, -1):
70              if j == d:
71                  for i in range(int(math.pow(2, j-1)), 0, -1):
72                      indexOne = int(i+k*math.pow(2, j))-1
73                      indexTwo = int(i+k*math.pow(2, j)+math.pow
                            (2, j-1))-1
74
75                      self.w(self.butterflyTab[indexOne], self.
                            butterflyTab[indexTwo], indexOne,
                            indexTwo)
76              else:
77                  for i in range(int(math.pow(2, j-1)), 0, -1):
78                      indexOne = int(i+k*math.pow(2, j))-1
79                      indexTwo = int(i+k*math.pow(2, j)+math.pow
                            (2, j-1))-1
80
81                      self.w(self.fileTab[indexOne], self.fileTab
                            [indexTwo], indexOne, indexTwo)
82
83      cipher_machine = AES.new(b'This is a key123', AES.MODE_ECB)
84      def w(self, blockOne, blockTwo, indexOne, indexTwo):
85
86          blockOne2 = self.cipher_machine.decrypt(blockOne)
87          blockTwo2 = self.cipher_machine.decrypt(blockTwo)
88
89          #This is for when encapsulation is done through split
                and combine
90          new_block_one = blockOne2[len(blockOne2)/2:] +
                blockTwo2[len(blockTwo2)/2:]
91          new_block_two = blockOne2[:len(blockOne2)/2] +
                blockTwo2[:len(blockTwo2)/2]
```

```
92
93
94              self.fileTab[indexOne] = new_block_two
95              self.fileTab[indexTwo] = new_block_one
96
97
98  bf_rev = Bf_reverse(filepath = 'C:\\Users\\olerasmu\\Documents
        \\workspace\\Butterfly\\src\\root\\nested\\bf_hourglass.
        txt')
99  start  = timeit.default_timer()
100
101 bf_rev.blockifyFile()
102
103 n = len(bf_rev.butterflyTab)
104
105 d = int(math.log(n, 2))
106
107 bf_rev.initiateButterfly(d, n)
108
109 bf_rev.fileifyBlocks()
110
111 stop = timeit.default_timer()
112
113
114
115 writeable = "Size:" + str(bf_rev.fileSize()) + "␣time:" + str(
        stop-start)
116 writeable = str(writeable) + "\n"
117
118 with open('resultfile.txt', 'a') as resultfile:
119             resultfile.write(writeable)
```

B.2 Permutation source code

B.2.1 Permutations encapsulation and decapsulation

```
1   '''
2   Created␣on␣4.␣apr.␣2014
3
4   @author:␣olerasmu
5   '''
6
7   import math
8   import timeit
9   from macpath import join
10  class Permutation(object):
11
12      def __init__(self, w=None, block_size=None, ts=None, tr=
            None, a=None, filepath=None):
13          self.w = w
14          self.filepath = filepath
15          self.ts = ts
16          self.tr = tr
17          self.a = a
18          self.block_size = block_size
19
20      n = 0
21      m = 0
22      g = 0
23      h = 0
24      block_size = 0
25      memory_read = 0
26      hg_tab = []
27      block_tab = []
28      symbol_tab = []
29      hourglass_tab = []
30      original_tab = []
31
```

```
32
33
34      def symbolifyFile(self):
35          start = timeit.default_timer()
36          with open(self.filepath, 'rb') as newfile:
37              symbol = newfile.read(self.w)
38              while symbol:
39                  self.symbol_tab.append(symbol)
40                  symbol = newfile.read(self.w)
41
42          stop = timeit.default_timer()
43          self.memory_read = stop - start
44          print "time for reading file into memory:", self.
                memory_read
45
46          self.m = self.block_size/self.w
47          self.n = len(self.symbol_tab)/self.m
48          return self.symbol_tab
49
50
51      def egcd(self, a, b):
52          if a == 0:
53              return (b, 0, 1)
54          else:
55              g, y, x = self.egcd(b % a, a)
56              return (g, x - (b // a) * y, y)
57
58      def modinv(self, a, m):
59          g, x, y = self.egcd(a, m)
60          if g != 1:
61              raise Exception('modular inverse does not exist')
62          else:
63              return x % m
64
65      def computeGandH(self, a):
66          self.g = math.ceil(self.ts/self.tr)*self.m + a
67
68          self.h = self.modinv(self.g, self.n*self.m) #g % self.n
                *self.m
69
70
71      def hourglass(self, i, h, n, m):
72          h_i = self.symbol_tab[int((i*h) % (n*m))]
73          self.hourglass_tab.append(h_i)
74
75          return h_i
76
77
78      def revHourglass(self, i, g, n, m):
79          g_i = self.hourglass_tab[int((i*g) % (n*m))]
80          self.original_tab.append(g_i)
81          return g_i
82
83      def encapsulate(self):
84          for i in range(0, len(self.symbol_tab)):
85              self.hourglass(i, self.h, self.n, self.m)
86
87
88      def decapsulate(self):
89          for i in range(0, len(self.hourglass_tab)):
90              self.revHourglass(i, self.g, self.n, self.m)
91
92
93  per = Permutation(w = 8, filepath = "C:\Users\olerasmu\
        Documents\\256mb_file.txt", block_size=4*1024, tr
        =0.0003125, ts=0.06)
94  per.symbolifyFile()
95
96  per.computeGandH(3)
97
98  start1 = timeit.default_timer()
```

```
 99
100 per.encapsulate()
101
102 stop1 = timeit.default_timer()
103
104 with open('encapsulated.txt', 'wb') as newfile:
105     for byte in per.hourglass_tab:
106         newfile.write(byte)
107
108 start2 = timeit.default_timer()
109
110 per.decapsulate()
111
112 stop2 = timeit.default_timer()
113
114
115 with open('decapsulated.txt', 'wb') as newfile:
116     for byte in per.original_tab:
117         newfile.write(byte)
118
119
120 with open("resultfile.txt", 'a') as newfile:
121     to_write = "File: " + per.filepath , " Memory read: " + str
            (per.memory_read), " Enc time:" + str(stop1-start1) ,
            " Dec time:" + str(stop2-start2) + "\n"
122     to_write = ';'.join(to_write)
123     print to_write
124     newfile.write(to_write)
```