Eivind Nordal Gran

Secure data sharing in the cloud

Master's thesis in Communication Technology
Supervisor: Colin Alexander Boyd, Gareth Thomas Davies & Clementine Gritti
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Eivind Nordal Gran

Secure data sharing in the cloud

Master's thesis in Communication Technology
Supervisor: Colin Alexander Boyd, Gareth Thomas Davies &
Clementine Gritti
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

Problem description:

Data sharing using cloud platforms has become increasingly more popular over the last few years. With the increase in use comes a heightened demand for security and privacy. This project will conduct a thorough study of a key transport protocol developed at NTNU, which targets strong security as its preeminent property, including a form of forward secrecy. More specifically, it will investigate how this escalation in security level affects the performance and usability of the protocol. How will the new protocol designed with security as its primary concern compare against other already established schemes when it comes to efficiency and practicality?

Abstract

Cloud sharing security is an important topic in today's society. The majority of the most common cloud sharing solutions require that the user trust the Cloud Service Provider (CSP) to protect and conceal uploaded data. A new key sharing protocol called the Offline Assisted Group Key exchange (OAGK) for cloud use has been developed at the Norwegian University of Science and Technology (NTNU) to create an alternative to trusting the CSP. The OAGK protocol aims to have strong security capabilities while at the same time function well in a cloud environment. This thesis sets out to test the OAGK protocol in practice, to evaluate the efficiency, practicality and how well it compares to other known solutions. In order to do this, two other protocols been chosen for comparison, the Tree-based Group Key Agreement (TGDH) protocol which is used in an existing secure cloud solution and the basic protocol concept called Public Key Encryption (PKE). These two protocols in addition to the OAGK are realized and implemented in a Java environment. Tests have been conducted to determine the the efficiency and practicality of the protocol. Experiments were constructed to provide well defined sample data for a comparison of the protocols. Computational effort, storage required and several other features used to determine the practicality was used in order to juxtapose the three protocols. Both the OAGK and a version of PKE were implemented from scratch and the TGDH was done using existing code found on Github. Even though the tests were only run locally they do indicate that the OAGK is slower and less practical than the PKE and TGDH. However, the OAGK performs adequately and is in certain cases the preferred protocol of the three. The tests indicate that there could be a future for the OAGK protocol.

Sammendrag

Sikkerhet innen skydeling er et viktig tema i verden i dag. Sikkerhetssystemet i de vanligste løsningene for skydeling krever at brukeren stoler på leverandøren av skytjenesten for å beskytte den opplastede dataen. En ny nøkkel-delingsprotokol kalt OAGK, for skybruk er utviklet på NTNU for å omgå dette tillitsproblemet. OAGK-protokollen har som mål å ha sterke sikkerhetsfunksjoner samtidig som den skal fungerer i kombinasjon med skytjenester. Denne masteroppgaven tar for seg å teste OAGK protokollen i praksis, for å evaluere hvor effektiv og praktisk den er, og hvor godt den fungerer sammenlignet med andre kjente protokoller brukt i skyen. For å gjøre dette, har to andre protokoller blitt utvalgt for å danne et sammenligningsgrunnlag, TGDH protokollen som brukes i en eksisterende sikker skyløsning, og en elementær protokol basert på konseptet PKE. Disse to protokollene, i tillegg til OAGK, er derfor blitt realisert og implementert i et Java-miljø med hensikt å bestemme hvor effektiv og praktisk protokollen er. Disse forsøkene ble konstruert for å gi pålitelige og gode resultater for en sammenligning av protokollene. Tidsbruk, lagringsbehov og flere attributter ble brukt for å bestemme hvor funksjonelle og praktiske de tre protokollene er i forhold til hverandre. I TGDHder det ble gjenbrukt gammel kode funnet på Github, OAGK og PKE ble implementert fra grunnen av. Selv om testene bare ble kjørt lokalt, og dermed ikke representerer et ekte skymiljø, indikerer resultatene at OAGK er tregere og mindre praktisk enn PKE og TGDH. Imidlertid så opererer OAGK på en tilstrekkelig god måte og er i visse tilfeller den foretrukne protokollen av de tre. Dette kan bety at det er en fremtid for OAGK protokollen.

Preface

This thesis is submitted as the final part of my master's degree at the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology.

I would like to thank my Professor Colin Alexander Boyd for his thoughts and insights which have contributed to forming this thesis. I would also like to thank Gareth Thomas Davies and Clementine Gritti for guidance and support throughout this final semester.

Thank you to Olav Sortland Thoresen and Østein Sigholt who provided assistance and shared their knowledge on program development.

These 5 years at NTNU in Trondheim has flown by and I would like to thank all the fantastic people who have made this time a memorable experience for me.

Trondheim, 5th of June 2019

Eivind Nordal Gran

Contents

List of Figures

List of Tables

List of Acronyms

AES Advanced Encryption Standard.

CA Certificate Authority.

CIA Confidentiality, Integrity and Availability.

CSP Cloud Service Provider.

DH Diffie-Hellman.

DSA Digital Signature Algorithm.

DSS Digital Signature Standard.

ECC Elliptic Curve Cryptography.

ECIES Elliptic Curve Integrated Encryption Scheme.

IaaS Infrastructure as a Service.

IETF Internet Engineering Task Force.

KEM Key Encapsulation Mechanism.

KMS Key Management Service.

NIST National Institute of Standards and Technology.

NTNU Norwegian University of Science and Technology.

OAGK Offline Assisted Group Key exchange.

OSI Open Systems Interconnection.

PaaS Platform as a Service.

PGP Pretty Good Privacy.

PKC Public Key Cryptography.

PKE Public Key Encryption.

PKI Public Key Infrastructure.

RA Registration Authority.

RSA Rivest-Shamir-Adleman.

SaaS Software as a Service.

SSL Secure Sockets Layer.

TGDH Tree-based Group Key Agreement.

TLS Transport Layer Security.

VPN Virtual Private Network.

Chapter 1
Introduction

File sharing has been around since the birth of the Internet, and the cloud computing concept has been evolving for 40 years. The term *"Cloud"* emerged in the 1990s with the use of Virtual Private Network (VPN) becoming more popular for data transfer. It makes sense that this virtual computing environment with dynamic allocation is comparable to the cloud we have today. Recently the use of the cloud for data sharing has become increasingly more attractive. The Internet was not designed with security in mind and it was meant to be used by good will users. Unfortunately this is not the case with the Internet we have today. With a world that is evolving, providing new features and functions every day, a higher standard for security is required.

Cloud computing can now be divided into three different types of services, Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). PaaS is most commonly used for applications and makes the foundation for developers as a framework to build upon. This presents a service where developers have control over applications, but do not have to worry about servers, storage and networking. The service is available through the Internet and gives developers flexibility in the building of applications. Usual characteristics of the PaaS are scalability of resources, a large diversity in development services, designed to be access by multiple users on the same project and comes with integrated database and web services. IaaS is constructed with scalable and automated resources. It can be used for system and hardware monitoring as well as a resource for hire. Businesses can buy IaaS instead of buying expensive equipment. This service is identified by having resources as a service, it is dynamic, adjustable and scalalable. The cost of this service depends on consumption and gives control to the client. SaaS is what most people associate with the concept "cloud". It is the cloud application services which is normally distributed through the Internet and controlled by a third party. Many of these services are accessed via a web browser or can be easily installed on personal devices. SaaS is recognized as being managed by a central authority, hosted on a remote server, attainable through the Internet and users are not responsible for

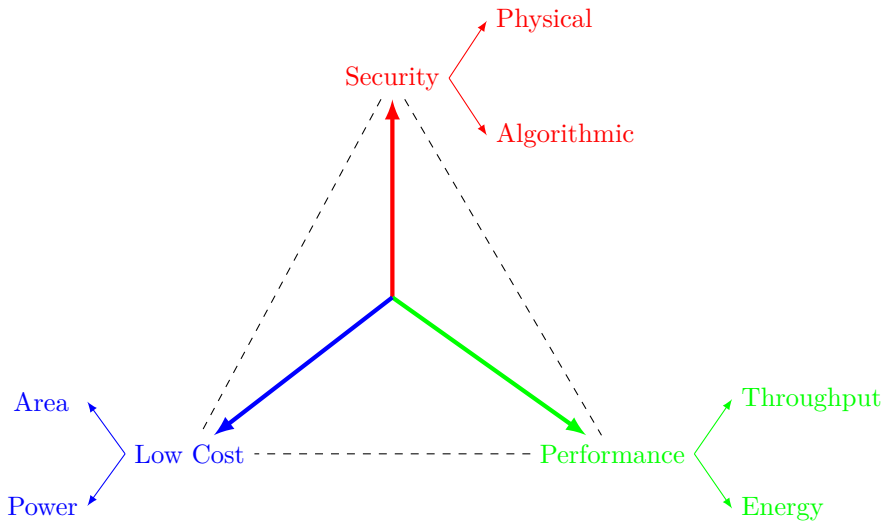

Figure 1.1: Performance [Jea16]

handling software and hardware updates.

A heightened security in cloud services is there to make sure that several features such as confidentiality, integrity, availability, authentication and non-repudiation are enforced. It is desired that shared data is confidential and only accessible for the designated parties. Unwanted changes in data should not occur, hence integrity is vital. Cloud services also depend on availability of the stored data, because there is no point of using the cloud of the data that is not available. Non-repudiation, assuring that performed actions can not be undone or denied by users. Together with authentication of data and users, does this help creating a more secure platform for cloud sharing. These security measures are not easy to combine and implement, and it is often a trade-off between the different features. As described in Figure 1.1 there are several different aspects to consider when designing a cloud system. Everything from power consumption to software configuration matter.

The public demand for more security in cloud sharing is somewhat contradictory to what cloud companies want, as they seek to make a profit. CSP wishes to keep the cost as low as possible, thus often opting for a lower security level than what the public would prefer. Some companies also make acquisition of data from its own clients and use this as a byproduct which contributes to a breach in privacy and security. Usually companies aggrandize themselves, posing as a secure and highly confidential cloud platform, but often this turns out not to be true as companies have access to all uploaded data. This thesis will conduct a study of a key transport

protocol developed at NTNU, which targets strong security as its prominent property, including a form of forward secrecy. More specifically, it will investigate how this escalation in security level affects the performance and usability of the protocol. How will the new protocol designed with security as its primary concern compare against other already established schemes when it comes to efficiency and practicality?

Key sharing in a cloud environment

Key exchange is a widely used method for establishing a secure channel between two points in a network. A secret key is exchanged and later used to encrypt data with a cryptographic algorithm before transmission, effectively hiding the original data for any eavesdroppers. This arrangement is performed quite often when establishing an online connection today. It is crucial that this technique is executed in a perfectly secure fashion using well established secure algorithms. The preferred method would be to include forward secrecy as this is the gold standard in key exchange algorithms. This method can be reassigned to make file sharing in a cloud environment secure. Where a secure channel between clients and a CSP is established, there are still a need for a mechanism that holds the information secret while stored at the CSP servers. Cloud services are usually owned and managed by a third party. Which means that the control of any uploaded information usually falls into their hands. Using the cloud to share data is a simple process, which is one of the reasons why it has become such a wide spread solution. There exist many different solutions to do cloud sharing, with quite contrasting security standards. Only a few companies provide a service with high security standard, featuring a functions such as encryption of given files before transmission. Cloud sharing should also on multiple platforms regarding both hardware and software, such as laptops/computers, mobile devices and the most common operating systems. This creates a challenge when implementing and designing software. Making everything compatible and integrating all security features necessary can be difficult. The need for a protocol which incorporates strong security features as well as being efficient, practical, cost effective and satisfies client demand is therefore extensive.

Let us imagine that a person wants to use the cloud to share a private document with only a few of his closest friends. Privacy is put in high regard for this person and he/she desires to keep a high security level throughout the entirety of the cloud sharing process. This means that the connection setup, transmission and storage of data must have strong security features. The first two are taken care of by standard implementations like Transport Layer Security (TLS). For the storage case one could trust the CSP, use a secure CSP or take use of a third party security mechanism. The simple option for storing files securely is to encrypt them before they are uploaded to the cloud. This will protect the files from all entities that have access to the given cloud server. Users must therefore choose whether to trust the

CSP or invoke some key agreement mechanism in order to achieve a desired security level. Implementing this encryption feature requires that a key distribution happens to the parties involved in the cloud sharing. A vast majority of users would not deem this solution practical, as the key management would cause inconvenience across multiple devices and platforms. This thesis will therefore focus on solutions which reduce the technical burden on the users and at the same time have strong security. This could be done with the OAGK protocol [BDGJ18], other protocols or existing solutions that already are in use in the world today. The question then becomes which one should be used. How the protocols compare against each other? Which one is the most practical and which is the fastest? Does file size of uploaded content matter and how often is a share created? How many participants are expected to join a cloud share? All of this must be taken into consideration when it comes to evaluating solutions.

The goals for this thesis are to answer the following questions:

- How efficient is the OAGK protocol compared to other known solutions?
- How practical is the OAGK protocol compared to other known solutions?
- How is the performance and usability of the OAGK in a real world setting?
- Is the heightened security level worth using in a cloud environment?

Previous work/Studied works

This work is based on the [BDGJ18] paper and the OAGK protocol introduced there by Boyd et al. A thorough description of the protocol and its security properties are presented and the reasoning for why it may be needed.

Furthermore the TGDH protocol implementation is based on the a Github repository[1]. The TGDH protocol is described and presented in [KPT04].

Thesis outline

Preliminaries: This section contains a set of security concepts followed by definitions of cryptographic primitives and algorithms that are pertinent for this thesis. A short overview of cloud services is then provided for the reader.

Existing solutions: This section describes existing solutions for cloud sharing, that contains various degree of security and functionality. A comparison of these protocols are displayed here.

[1] https://github.com/lemmy/de.rub.hgi.nds.tgdh (accessed on 06.05.2019)

Key exchange protocols: This part of the thesis consists of three chapters, namely *The OAGK protocol*, *Public Key Encryption* and *Tree-based Group Diffie-Hellman*. It describes in depth the protocols in question. Additionally, it gives an explanation of the motivation behind choosing the aforementioned protocols, how the protocols should function and an explanation on how the implementation works.

Testing: This section explains the method of testing, which includes the environment, how and why a test is conducted. This chapter should provide a through description in order to prove the validity of the results and also make the tests reproducible.

Results: This chapter contains the results from the previously described tests. It gives a description of what the results mean and how they affect the protocols.

Discussion: This is the evaluation of the *OAGK* protocol with relation to efficiency, practicality, usability and performance. I discuss what the results indicate and what limitations that are in this study.

Conclusion: This chapter will take a look back at the thesis goals and try to reflect upon the results with these in mind. The chapter is also giving a short section of some possible future work.

Chapter 2
Background

This chapter start with a brief introduction of some important security concepts within cloud security. Explaining what it means to have security and how to achieve such a state. Furthermore this chapter will go through a few essential cryptographic primitives in order to enhance the understanding and insight for the coming chapters. These basic cryptographic primitives will be followed by several common encryption methods and security features. A few existing solutions will be analysed and described in order to establish how the current situation is within secure cloud sharing.

2.1 Preliminaries

2.1.1 Security Concepts

Integrity

Integrity is as defined by the Oxford Dictionary: *"Internal consistency or lack of corruption in electronic data"*. Data integrity means that data can not be changed without you knowing about it. This ensures that data can be "trusted". As a part of the Confidentiality, Integrity and Availability (CIA) triad of information security it must be considered during implementation of a system. Mitigating or reducing the chance for a compromise of integrity is important in many actions performed recurrently in present-day society. A common solution for implementing integrity is to have a checksum or a hash of the given data [SWZ05]. The checksum or hash is a number computed based on the original data and can later be calculated again to verify that it is the same as before. This verification method can assure completeness of data and detect alterations both on data in transit and in storage.

Availability

Availability is as restated from the Oxford Dictionary: *"The quality of being able to be used or obtained"*.The availability of a system is vital for it to serve its purpose. Being able to access data when needed is important when regarding security. Loss of

availability can come from several sources such as power loss, hardware malfunction, software malfunction, network attacks, compromise of the system and more [And14]. In some systems availability can surpass integrity and confidentiality. For instance, the cloud becomes quite useless if one can not access it. Ensuring availability is hard because of all the factors that can play a part. Protecting everything that makes a system operational is quite demanding. Physical property, hardware, software and data in transit are elements that makes a system available, and needs to be protected in order to ensure availability. Maximum security comes with a huge cost and one must choose wisely where to implement different security measures.

Confidentiality

Confidentiality is according to the Oxford Dictionary *"The state of keeping or being kept secret or private"*. Confidentiality of data means that only authorized entities are able to observe the data [Knu98]. It is similar to privacy, but privacy is more of an ethical nature, and is regarded as keeping personal information a secret. Confidential data should be available for authorized users and at the same time not be revealed to unauthorized entities. Confidentiality can be achieved by encrypting data. This way ensures that only the ones that possess the correct decryption key can view the data. This can be done during transit and on stored data, but confidentiality also involves giving authorization to the correct entities, thus implementing confidentiality involves more than encryption and should be considered from the start when designing a system. Confidentiality is considered important in a cloud sharing environment, where only the correct entities should have access to certain data.

Non-repudiation

Repudiation is defined as: *"Refusal to fulfill or discharge an agreement, obligation, or debt"* by the Oxford Dictionary. Non-repudiation means that a party can not deny that a transaction has taken place [CS06]. This attribute is obtained by a proof of integrity and origin of data together with an trustworthy authentication. Non-repudiation is crucial in systems where individuals must be accounted for. Actions can then be traced to users in a system without a possibility for denial from the user. Two prevalent ways of implementing non-repudiation is through Message Authentication Code (MAC) or using a digital signature. Both solutions require that information has been shared previously. MAC require a pre-shared secret, and digital signature involves a central authority for distributing public keys.

Forward secrecy

A.J. Menezes et al. in [MVOV96] defines forward secrecy as follows: *"A protocol is said to have perfect forward secrecy if compromise of long-term keys does not compromise past session keys."* Having forward secrecy implies that a compromise

of established long term keys should not reveal any session keys established in the past. This means that encrypted data should not become easier to decrypt by an eavesdropper in the future, excluding compromise of the ciphersuit used and the development of quantum computers. Having forward secrecy is important to protect data that is being transmitted from later being disclosed. Previous sessions are sealed and locked in the past when the given session key is deleted, but forward secrecy does not protect an entity from an adversary that captures all traffic and then later breaks the ciphersuit on its own. Forward secrecy can be established using a Diffie-Hellman key exchange for each session established. Forward secrecy is highly recommended in IT security and is an essential feature to have if strong security is valued. One could therefore expect that cloud sharing solutions have this quality enforced.

Authenticity

Authenticity of an entity is vital in communication, especially in wireless networks. Verifying the identity of a user has therefore become important in secure systems. Authentication is applied to many technical solutions worldwide, like banking, user accounts, passports, etc. Proof of authentication can be achieved in three elementary ways:

-Something you know(Knowledge): e.g. Password

-Something you have(Ownership): e.g. Visa card, Passport, IP-address

-Something you are(Inherence): e.g. Biometrics

These stages can be used to provide authenticity of an entity. Digital authentication over a network is a bit more complicated and requires a trusted central authority that can verify users, distribute information about users and keep track of all users. If a strong Public Key Infrastructure (PKI) is in the foundation of a network, implementing authentication becomes straightforward. Already established schemes and protocols for authenticity confirmation can then be used in all kinds of transactions and interactions on a network.

2.1.2 Key Exchange

A key exchange in terms of cryptography is an arrangement where two parties exchange cryptographic keys, giving support to use a cryptographic algorithm. The goal of a key exchange is as defined by Diffie et al. in [DH76]: *"The goal is for two users, A and B, to securely exchange a key over an insecure channel. This key is then used by both users in a normal cryptosystem for both enciphering and deciphering"*. Traditionally this exchange of keys takes place in physical form of a

pre-agreed password or in a codebook like the one used for the Enigma machine[1]. In recent years this key exchange usually happens virtually, and if the key exchange is not secure then the parties involved will not be able to truly communicate securely in the following session. One can according to NIST [BBB+12] define two types of key establishment schemes: key agreement and key transport. Key agreement is when two or more entities work together to form a key dependent on information from all entities. Key transport is the distribution of an already defined key and is normally encrypted before it is transmitted to receiving parties.

2.2 Cryptographic primitives

2.2.1 Key Encapsulation Mechanism (KEM)

Key Encapsulation Mechanism (KEM) is an encryption technique for making transmission of a symmetric keys secure. KEM utilizes an asymmetric key algorithm, making this scheme suitable when public/private keys are established. This scheme is useful because using Public Key Cryptography (PKC) does not work that well in practice to transmit long messages. Using a KEM to transmit an encryption key is therefore a more common solution. The amount of transmitted data is smaller and a different encryption scheme can be used on the data while maintaining the same security level. KEM is thus suitable in environments where data is transmitted often and in large quantities. This scheme is therefore implemented in the OAGK protocol which is analyzed in this thesis.

2.2.2 Blinding

Blinding is a technique that can be used by an actor for making either input or output hidden. This hiding feature can be applied by adding a random part to data. For example encrypting with an added random bit will yield a completely different result than without. Combining a random component to data will blind it from anyone who does not know the blinding factor. This approach is used as a service when either the input, output value or both should be kept secret, as blinding prevents information leakage. As shown in Equation 2.1 the value C is applied a blinding factor b. The values $\bar{C}$ can be reconstructed back using the inverse of the blinding factor. This feature can be useful when information must pass through entities that should remain oblivious to what the data truly is. This blinding factor combined with the KEM is one of the key features in the OAGK protocol [BDGJ18].

$$\bar{C} = C^b \longrightarrow C = \bar{C}^{-b} \tag{2.1}$$

[1] https://www.cryptomuseum.com/crypto/codebook/index.htm (accesses on 25.05.2019)

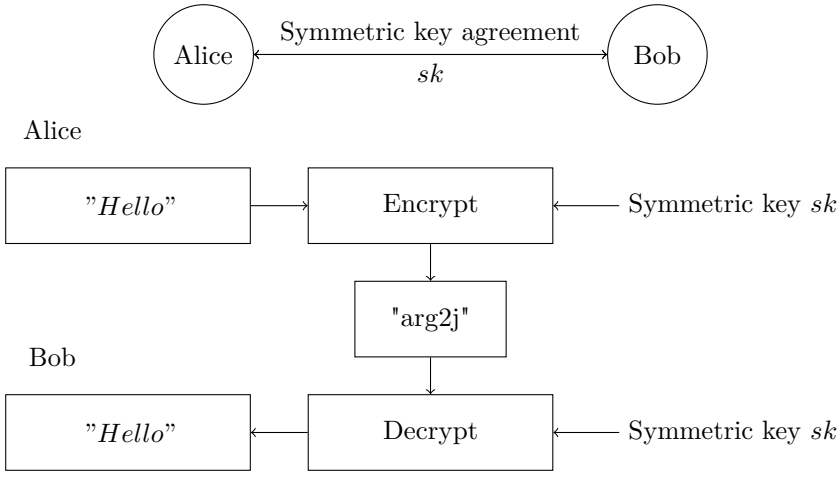

Figure 2.1: Symmetric key cryptography

2.2.3 Symmetric encryption

The concept of symmetric key encryption involves using the same cryptographic key for both encryption and decryption. This technique can be applied with several different ciphersuits and is therefore applicable in many situations due to its flexibility. The basics of symmetric encryption are shown in Figure 2.1, where the same key is used for both encryption and decryption. A precondition for using this scheme is that the key must be shared before the transmission of encrypted data. The need for adopting a key sharing protocol beforehand is therefore a consequence of this. How this key establishment is conducted may vary and can have a huge effect on the security accomplished in a transmission. Symmetric key encryption is normally quite fast compared to asymmetric encryption and usually requires a shorter key to achieve the same security level.

2.2.4 Asymmetric key encryption

Asymmetric encryption requires two keys that are generated with a mathematical connection [Knu98]. The creation of this key pair must make sure that the keys are dependent, but one can not be derived from the other. The key pair can then be divided into one secret key and one public key. The secret key must be held private, and the public key can be distributed in the network. Thus be used to encrypt messages and verify digital signatures that belong to this given key pair.

Asymmetric keys can be used not only for encryption, but also for authentication in the form of digital signatures. This works in a similar way to encryption with asymmetric keys. A hash based signature is made on the information you are

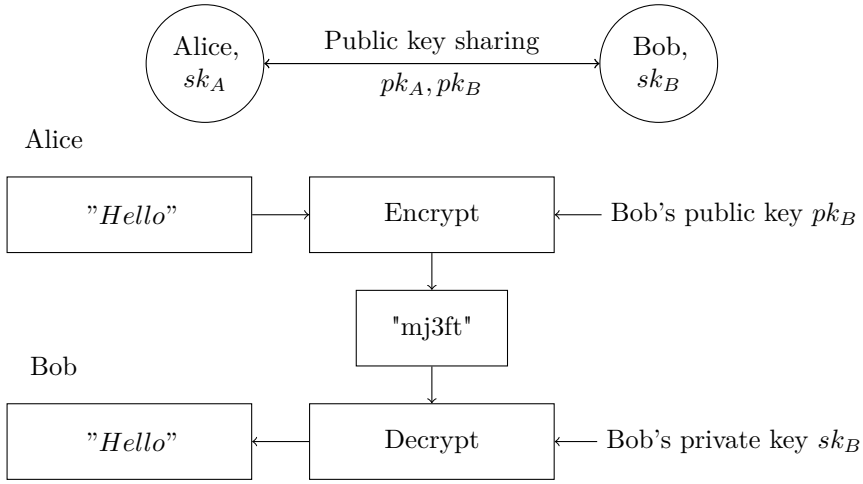

Figure 2.2: Asymmetric key cryptography

transmitting using the secret key. It can later then be confirmed using the public key of the signer that the information was indeed signed with the corresponding secret key of the signer. The basic idea behind asymmetric encryption is described in Figure 2.2 where pre-shared keys have been established, and public keys are used for encryption and the corresponding private key is used for decryption.

RSA

Rivest-Shamir-Adleman (RSA) is one of the earlier developed public cryptographic systems. It was introduced in 1978 [RSA78]. The protocol works by utilizing the factorization problem. The RSA algorithm is based on Equation 2.2, consisting of a message m that is to be encrypted. Two numbers are chosen to create the modulus n, $p * q = n$ where both p and q are two very large prime numbers. After this e is chosen to be a coprime of $(p-1) * (q-1) = \lambda(n)$ and then d can be computed as $d \equiv e^{-1}(mod\ \lambda(n))$. Where $\lambda(n)$ is the Carmichael function which is *"In algebraic terms, $\lambda(n)$ equals the exponent of the multiplicative group of integers modulo n"*[2]. Proven hard to break as restated from [Cal17]:

"This cryptosystem is hard to break, because if a person intercept the message, all they would know is the public keys: e and n. Theoretically, you could construct $\lambda(n)$ from n, but this would involve factoring n into p and q. This is hard if n is large, for

[2]https://en.wikipedia.org/wiki/Carmichael function (accessed on 14.05.2019)

as of yet there is no efficient way to factor really large numbers."

$$(m^e)^d \equiv m(mod\ n) \tag{2.2}$$

This provides a nice way of generating key pairs with one private and one public key, which can be securely used to transmit messages over an open network.

2.2.5 Diffie-Hellman Key Exchange

Diffie-Hellman (DH) key exchange is a scheme that is widely used in practice, it conditions for a secure agreement of cryptographic keys. Assuming that all adversaries see all transmitted traffic requires a clever solution [Mer78]. Cryptographic keys can not be sent in clear in an agreement. A solution that does not send any key material was therefore proposed. Utilizing the discrete logarithm problem (Definition 2.1 page 13), makes it possible to send key material without revealing any information about the derived agreed key. This makes sessions using keys derived with the Diffie-Hellman protocol forward secure [DH76]. Which is as explained previously an important feature to have in private conversation on a public network.

The discrete logarithm problem and the Diffie-Hellman problem as defined by [Gjø16]:

Definition 2.1. The discrete logarithm of x to the base g is the smallest non-negative integer a such that $x = g^a$. We write $\log_g x = a$. The *discrete logarithm problem* in a cyclic group G is to find the discrete logarithm of x to the base g, when x has been chosen uniformly at random from the group.

Definition 2.2. The Diffie-Hellman problem in a cyclic group G of order n with generator g is to find $z = g^{ab}$ given $x = g^a$ and $y = g^b$, when a and b has been chosen independently and uniformly at random from $\{0, 1, 2, ..., n-1\}$.

Alice and Bob have to agree upon some numbers before the DH protocol can be run. They must use the same prime number p for modulus and have the same base g, where $g \in G$ with G being a cyclic group of prime order p. The protocol can then begin and Alice and Bob each chooses a secret value. Alice choose a and Bob b. Alice then proceeds with transmitting $A = g^a$ to Bob, and Bob sends $B = g^b$ to Alice. Alice can then use the number received from Bob to derive a key by raising it to her secret number $B^a = (g^b)^a$. Likewise Bob can do the same calculation $A^b = (g^a)^b$. As we can see in Figure 2.3 they end up with the same product g^{ba} and they have now agreed and share a common secret key without leaking information about it to any eavesdropper.

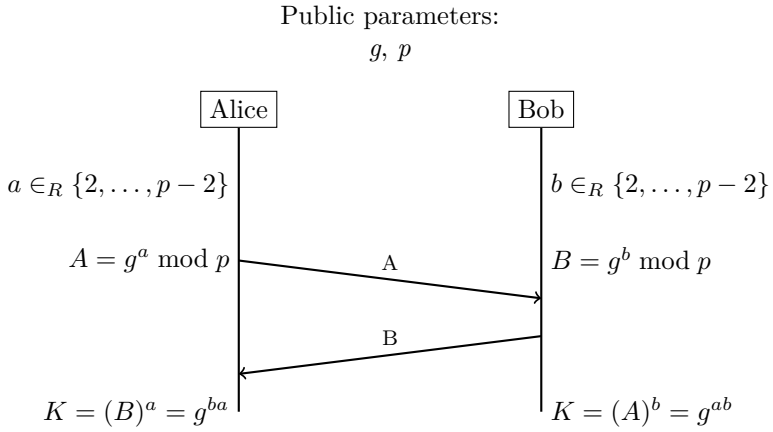

Figure 2.3: Diffie-Hellman Key Exchange, taken from [Jea16]

Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is another way of doing public key cryptography. An elliptic curve is an algebraic structure defined in a finite field. The most considerable reason to use ECC is that a smaller key can be used to achieve the same security level compared to other cryptographic solutions. This is because the elliptic curve is built upon the discrete logarithm problem (Definition 2.1 on page 13). Elliptic curve can be used in multiple cryptographic functions, for example Diffie-Hellman, Digital Signature Algorithm (DSA), Integrated Encryption Scheme, pseudorandom number generator and more. Elliptic curve cryptography is commonly used instead of RSA because you get reach the same level of security with a key that is approximately 10 times smaller[3]. This will in turn lead to less storage space needed, less transmitted data and the DH algorithm typically run faster than its adversaries [Bar16].

2.2.6 Digital signatures

Digital signatures are used to prove the authenticity of transmitted data. Which means that the entity of the creator or sender of data can be verified. Another contribution coming from digital signatures that also enhances security is integrity. A receiver can be assured that the transmitted signed data has not been altered in transit [IBM19]. Digital signatures take use of asymmetric cryptography that needs two connected keys to be deployed. This key pair comes in form of a public key and a private key. Signing a document or message requires the knowledge of a private key, which then together with a one-way hash function creates a signature. This signature can then be verified with the corresponding public key. It is important to

[3]https://www.ietf.org/rfc/rfc5349.txt (accessed on 14.05.2019)

recognize that digital signatures only accomplish their task if you can trust the issuer of the public keys. One must be confident that a public key actually corresponds to the said entity. Digital signatures are used in most network transmission, at least when establishing a connection in order to ensure authenticity of the entities.

2.2.7 Digital Signature Algorithm

The DSA is specified in the Digital Signature Standard (DSS) [Nat94] and is appropriate when a digital signature is needed instead of a normal written signature. The DSA signature is a pair of numbers that provides information such that the identity of the signer and integrity of data is verifiable. Non-repudiation is also supported in the DSA. The groundwork for the DSA comes from ElGamal signature scheme, which is a scheme based in the discrete logarithm problem (Definition 2.1 on page 13). Digital signatures using DSA are common, and can be found in most interaction and transmission over a network. This scheme enforces features that provide trust during interactions and on transmitted data.

2.2.8 Hash functions

Hash functions can be applied to data of a random size and output data of a fixed size. A hash function has therefore the one-way feature. Meaning that there is no feasible way of using the result of a hash function to construct the input material. A hash function usually has a large input space compared to the output space. The fact that hash function output is hard to reverse back to the original input makes them useful in cryptography. Hash functions can be applied for integrity checks and message authentication (digital signatures).

A good hash function should have the following properties:

- Deterministic: A given input should always give the same output.
- Fixed output range: Output should have a fixed size.
- Non-invertible: One should not be able to reconstruct the input based on the output.
- Collision resistant: It should be infeasible to find two inputs that produce the same output.
- Uniformity: Output values should spread uniformly over the entire output value space.

2.3 Cloud Service

Fundamental properties

The essential characteristics of cloud computing defined by National Institute of Standards and Technology (NIST) in [MG11] reiterated below in Definition 2.3.

Definition 2.3. 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 (On-demad self-service, Broad network access, Resource pooling, Rapid elasticity, Measured service), three service models (Software, Platform, Infrastructure), and four deployment models (Private, Community, Public, Hybrid).

2.3.1 Cloud Service Provider

The owner of a cloud service will depend on the kind of deployment model.

Deployment models defined by NIST [MG11] are:

- Private: Single organization.
- Community: Severel organizations or group of consumers that share interest.
- Public: Open cloud service to the public.
- Hybrid: A blend of all aforementioned models.

The owner of the cloud service will have a varying degree of control and access in each of the four models. The more admittance the owner has in a cloud the greater the trust in the CSP is needed. Which model is used will also affect how the cloud operates, what security level, how functional, efficient and practical it is. It could also influence each session setup and the distribution of data.

2.3.2 Storage

Cloud storage is a service that allows users to upload data of a vast variety to the Internet and have it stored on a remote server. This removes the need for having large amount of storage capacity on devices and therefore makes the use of multiple smaller devices possible. Because of this and the fact that the data is "airborne" and it spreads more easily means that having and using multiple devices are becoming increasingly more popular. Cloud services are virtualized service and often have an

easy accessible interface. It should be scalable and have potential for multiple users at the same time. The resources needed are normally distributed among several locations and devices, but they all function as one unit. One or multiple CSPs are in charge of these physical facilities. They have control of all data that passes through their equipment. That is why CSPs are responsible for the security of the data. The handling of the security issue executed in different ways by each individual CSP. Some have complete access to stored data and others have implemented features making the data only accessible by the allowed entities. One cloud also use software distributed by a third party to implement security measures to data before it is uploaded to the cloud. This will not change how data is stored at the CSP, but it will increase the security level.

2.4 Web Framework and Security Implementations

2.4.1 Public Key Infrastructure (PKI)

The purpose of PKI is to define a set of regulations and standards to handle digital certificates and public-key encryption. This makes it possible to trust authentication on the public network. PKI lays the foundation that makes public-key encryption achievable and is necessary for security in extensive networks [BKMM96]. The system consists of multiple parts and is based on certificates. A digital certificate is an electronic document that can validate the ownership of a public key. This document holds information about the public key, the owner as well as a digital signature of an advocate entity for verification. Several bodies are needed to manage, verify and uphold these certificates. A Certificate Authority (CA) has the responsibility of storing, promulgating and signing digital signatures. The verification of these digital signatures stored by the CA are done by a Registration Authority (RA). All keys are stored in a central directory and these keys can then be accessed or distributed using the certificate management system [Vac04]. This hierarchy is shown in Figure 2.4 where there is one root authority that issues certificates to higher tear CAs and they have the possibility of issuing more certificates, thus generating a hierarchy that must be trusted in order to use PKI.

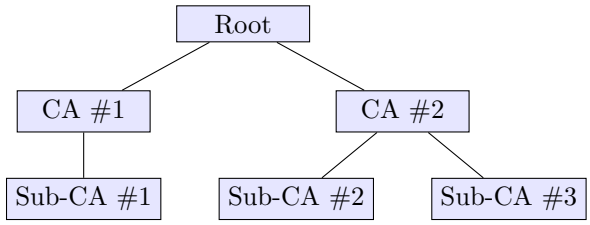

Figure 2.4: PKI hierarchy

2.4.2 Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) is a security scheme that administers authentication and cryptographic guarantees. The normal use cases for this scheme is with signing, encryption and decryption. PGP encryption and decryption operate in a distinct manner. It takes use of already established public and private keys, which are generally slow and expensive [Gar95]. That is why it is desirable to encrypt using a different system. The PGP system starts by generating a random encryption key (symmetric key) that is used to encrypt some data. This symmetric key is then encrypted using the time-consuming RSA scheme, but the key is relatively small compared to the amount of data and this is therefore not a problem. The encrypted data is then transmitted together with the encrypted symmetric key. This transmitted message can then only be decrypted by the entity in possession of the corresponding private RSA key, which can decrypt and use the symmetric key to decipher the encrypted data. An example of this kind of system is email, which is used frequently in our society. PGP is a non-interactive protocol, meaning that users do not have to be in direct contact and must use long term keys for encryption and decryption. This is shown in Figure 2.5 where Alice is transmitting a data to Bob. In the end does Alice send the the data encrypted with the symmetric key, which is also sent encrypted with Bobs public key. Due to the non-interactivity is forward secrecy not achieved, however by adding another layer of security could forward secrecy be implemented, e.g. using the TLS scheme explained in the next section.

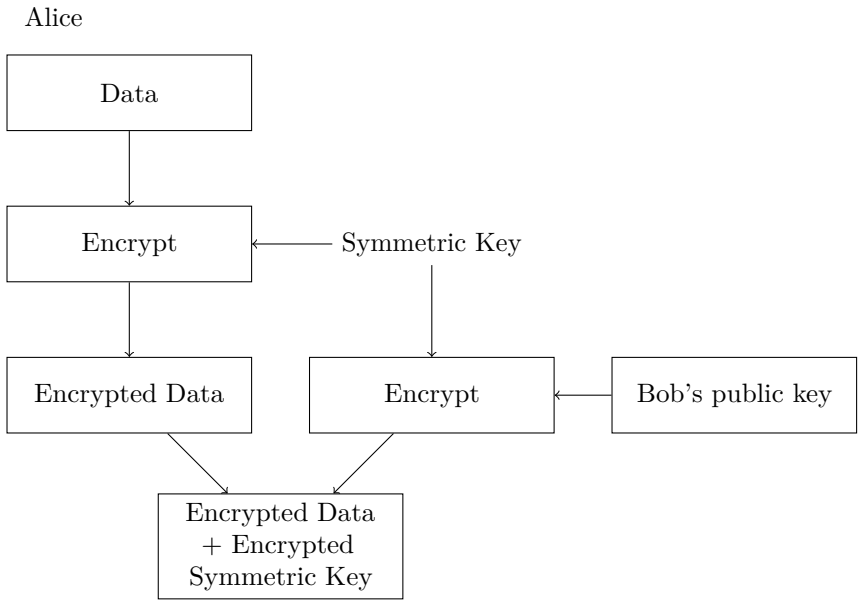

Figure 2.5: PGP

2.4.3 Transport Layer Security (TLS)

TLS as specified by the Internet Engineering Task Force (IETF) in [Res18] is a scheme that provides a secure tunnel between two communicating parties. The TLS protocol accommodate for three security properties in the form of authentication, confidentiality and integrity. In every connection established is the server side authenticated and the client side has optional authentication. The authentication can be done using asymmetric cryptography such as RSA or elliptic curve DSA. The transmitted traffic is encrypted which ensures the confidentiality of the data. Even padding can be added to obscure the data further, thus not revealing the length of the data. The traffic is therefore coincidental to the endpoints of the TLS channel. Integrity of the channels is established also by the encryption of data because if the data is altered during transit it will not decrypt correctly. The TLS protocol starts with a handshake between the parties involved and authenticates them. The cryptographic method is then negotiated and key material is shared. Each generated tunnel is separated and protected by including different parameters that define a session.

The TLS protocol history started by the creation of Secure Sockets Layer (SSL), which is a standard technology used to keep a connection between two entities secure. All created versions of the SSL protocol have today been deprecated. There have been several iterations of the TLS protocol and the first one was the TLS 1.0 that was an upgrade of the SSL 3.0. The evolution of the TLS has been drastic and it has become more and more secure. The latest version, the TLS 1.3 defined in [Res18] was released in 2018 and possesses a large amount of new security features. All of the TLS versions are still in use, but it is strongly suggested to mitigate from the earlier versions of the protocol because they have known weaknesses.

The TLS protocol that can be found in the Session layer in the Open Systems Interconnection (OSI) model, which is a model describing the characteristics and standards of telecommunication. This Session layer sits on top of the transport layer as shown in Figure 2.6 and is therefore something one can implement on top of the already existing transmission system. The TLS protocol is a system built upon trust and requires digital certificates. These certificates are usually issued by a trusted third party. This third party is then a part of the PKI as described in a previous paragraph. A compromise of this third party could allow adversaries to perform a man-in-the-middle attack. In addition to digital certificate the TLS protocol executes a key exchange ahead of any other actions. The key exchange algorithm is negotiated between the two parties as represented by the "ChangeCipherSpec" in Figure 2.6. The chosen protocol will affect the rest of the transmission. Some of the algorithms have stronger security properties than others, such like forward secrecy, integrity or authentication checks. The TLS protocol is typically used to establish a secure

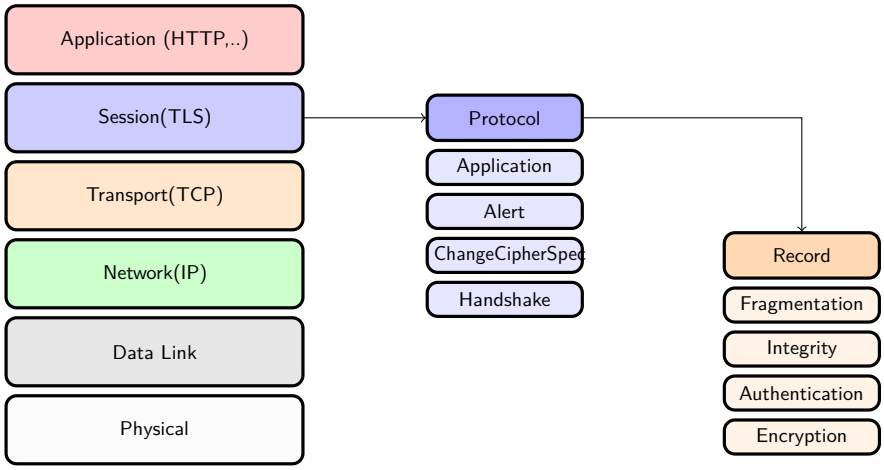

Figure 2.6: Transport Layer Security

channel between two end points and protect data during transit. Data is therefore protected between the endpoints, but not at the endpoints. Another system must therefore be incorporated to protect data at the endpoints, hence the investigation on protocols like the OAGK [BDGJ18].

2.5 Existing solutions

There are already some existing solutions that have been made and put in use today. The solutions described in the following sections function in different ways to provide a cloud sharing service and they have a varying degree of security, efficiency and practicality.

2.5.1 Tresorit

Tresorit[4] is a cloud sharing solution that claim to be superior to most other cloud sharing solutions [Tre19a]. With end-to-end encryption, cryptographic key sharing and client side integrity protection does Tresorit have a completely secure cloud sharing platform. The end-to-end encryption is handled by encrypting and decrypting at the client side, making the data only available in plain text for the client. This removes the need for trust in the CSP and the security of data is claimed to be as high as if it was stored on your own local machine.

Tresorit utilizes PKC together with PKI to ensure the security of their cloud sharing system. The PKC makes Tresorit "blind" regarding the shared data and the

[4]https://tresorit.com/ (accessed on 20.04.2019)

PKI that implements certificates providing authentication of the users in the system. Together with a structure of symmetric keys, does this make the whole scheme secure against unwanted access of data. The used encryption keys are stored and hidden using a master key. This master key is the users RSA public key, which makes this scheme not forward secure because all the keys are dependent on the master key.

This security system is cloud independent and can therefore be able to operate on several different clouds. Tresorit also have a basic way of sharing data by creating sharable links that can be sent via e-mail. Together with this sharing method does there exist a hierarchical structure for actions and permissions on shared data. The encryption scheme can also be applied to all kinds of data. All these facts indicates that Tresorit is a practical scheme and the ease of use is quite good. Tresorit claim as well to be en efficient and fast scheme, mostly limited by bandwidth when uploading and downloading data.

Technical solution

The technical implementation is described in Tresorits white paper [Tre19b] and also an overview is given in Figure 2.7. The encryption of data happens at the client side with an AES cipher in CFB mode, making it a continuous stream cipher. A 256-bit symmetric key is used in this encryption. HMAC-SHA-512 is used to provide integrity of data and the data is authenticated using a digital signature scheme signed by the uploader of the data. This data is then uploaded through a TLS tunnel to the cloud. For all this to work the PKC and PKI have to be in place beforehand. Tresorit have their own module for key sharing which either can be based of RSA or TGDH. This module contains all the master keys from each user, but each key is encrypted using each clients corresponding public RSA key.

2.5.2 Dropbox

Dropbox[5] is an interface that can be used for cloud sharing and collaboration described on their website [Dro19b]. It is made to make cloud sharing easy, fast and reliable. Dropbox have an infrastructure designed to reduce the complexity for the user and they claim to have strong security properties. Dropbox is devided in a 4 part working structure. Client side, block servers, storage servers and metadata servers. Unlike Tresorit there is no encryption happening at the client side. The client can only download and upload data. The encryption happens in the block servers, where Dropbox use a strong cipher suit for encryption. The encrypted data is then stored on the storage servers. The metadata servers stores information each and every user and work as an inventory index for each specific user. The metadata contains information concerning a users and its files. Together with the metadata

[5]https://www.dropbox.com (accessed on 20.04.2019)

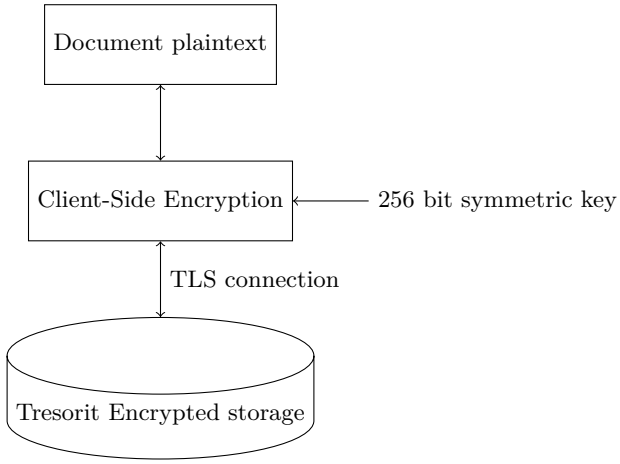

Figure 2.7: Description of encryption in Tresorit. Restated from Tresorit white paper [Tre19b]

server does Dropbox have a notification service that monitors changes to files and data. Because of this does files that are stored with Dropbox synchronize much more often with the cloud than in Tresorit.

Dropbox is a program that must be installed on the device that is using it. Thus making it a efficient solution because it resides within a local system. It synchronizes and operates automatically on its own. Additionally, adding or removing shares is also relatively effortless. Dropbox claim to have perfect forward secrecy by encrypting at the endpoints using a special SSl key that can not be used to decrypt past Internet traffic. The keys for encrypting files are stored on servers owned by Dropbox, hence giving Dropbox the control of them. This solution therefore holds the CSP trust issue.

Technical solution

The specifications and implementations of Dropbox are explained in this white-paper [Dro19a]. Dropbox utilize TLS to provide security for data in transit making a tunnel with 128-bit Advanced Encryption Standard (AES) encryption. Forward secrecy is provided by ensuring that no files are sent before the encrypted connection is fully established between Dropbox front end and the Dropbox servers. The files are actually in plain text when they arrive at Dropbox servers. Dropbox then protect received data by encrypting it with a 256-bit AES cipher. The keys used for encryption in Dropbox are distributed with a decentralizing processes, separating the key generation, exchange and storage. As mentioned previously, Dropbox handle the encryption of data on the internal system. This means that they encrypt data on

behalf of the clients and Dropbox have to secure these keys inside their systems. This system can only be accessed using unique SSH key pairs, which are protected in the Dropbox internal system. Dropbox takes use of certificate pinning to authenticate clients, and clients can also verify Dropbox servers this way.

2.5.3 MEGA

MEGA came to life in 2013 and is an end-to-end encrypted cloud storage and collaboration platform[6]. Described in both this case study [DDC17] and MEGAs white-paper [MEG18]. It is user-controlled, meaning that the users have full control of the applied encryption keys. MEGA has no access to clients data due to the end-to-end encryption. This is because the encryption keys are stored on the client side. MEGA also provide a communication suit integrated in the cloud sharing system. Every client have the opportunity to share with others just by encrypting the key used and transmitting it to others. MEGA has its source code out in the open for everyone to see.[7] The fact that anyone can view the code makes it more secure. The correctness of implementation and whether there exist backdoors or not can be checked by anyone. It also makes it possible for others to find unintended vulnerabilities.

MEGA is compatible with several operating systems and comes with a synchronization ability. They also claim to be the only end-to-end encryption scheme with browser access of data on the market. Choosing who you share with is quite simple, just sending the given sharing key. Removing access can be a bit more complicated, but MEGA have implemented a feature that makes it possible to put a time limit that expires, thus removing the share. There is no need to download a program to use MEGA and it is reasonably fast given that it works using javascript in the browser. It is fast enough for user interaction. The practicality of this scheme is therefore quite good.

Technical solution

For each file that is stored there is created a corresponding file key made from 128-bits and a nonce of 64-bits. Files are divided up into chunks and encrypted with AES with CCM and the nonce is incremented for each chunk that is encrypted. In order to provide integrity of data a MAC is calculated for each encrypted chunk and a "master" MAC is created from all the calculated MACs. The file key, the nonce and the "master" MAC is then obfuscated and encrypted using a master key and then uploaded to the API. The master key is derived using a random hash function with your password and a salt. For sharing folders and files a RSA key pair is used. Users

[6] https://mega.nz/ (accessed on 21.04.2019)
[7] https://github.com/meganz/sdk (accessed on 21.04.2019)

need to have an account to be able to use this system, but one can still share to persons that does not have an account. The authentication in this system comes in form of a login by the user.

2.5.4 Google Drive

Google Drive is a popular platform for storing, sharing and collaborating on data[8]. It works on a web interface and has support for multiple different devices and operating systems. This is why Google Drive is a widely used cloud sharing service. It is an easy, fast and reliable and practical solution because it is implemented to be used in the browser. Google apply security in a lot of stages in the cloud storing structure and they have an overview o their website [Goo19a]. They protect data in physical form in their data centers by applying normal security features like guards, fences etc. Furthermore do Google use hardware that is designed for the job which increase performance, security and reliability. This hardware is placed in a strong and robust infrastructure to provide uninterrupted service and heightened security. Lastly do Google claim that they encrypt in all stages involved in cloud computing. Data is encrypted in transit, when stored on disk, when it is stored on backup devices and in transit between datacenters.

Technical solution

The technical description for the security of the Google cloud is explained in their white-paper [Goo19b] and a simple overview is given in Figure 2.8. Data at rest will be stored on disk and backup media, both will be encrypted. Data on disk are divided into chunks and each chunk is then encrypted using a secret key. The encryption cipher used on each chunk will be AES with a key of 128-bit or larger. Google uses a Key Management Service (KMS) to store and distribute keys. Used keys are stored together with corresponding encrypted data. Encryption and decryption of the keys used on the chunks are done in the KMS. On request, the KMS will decrypt the key and return it to the storage system. Google rotate used keys to reduce the impact of a key being compromised. This reduces the vulnerability of the data. The KMS system only allows authenticated users to get access to keys. Data in transit between a device and Googles data centers are protected using either RSA with 2048-bit or P-256 ECDSA SSL certificates to ensure authentication. This encrypted tunnel (2) in 2.8 be established with Diffie-Hellman key exchange style protocol and therefore provide forward secrecy on the transmitted data.

[8]https://www.google.com/drive/ (accessed on 22.04.2019)

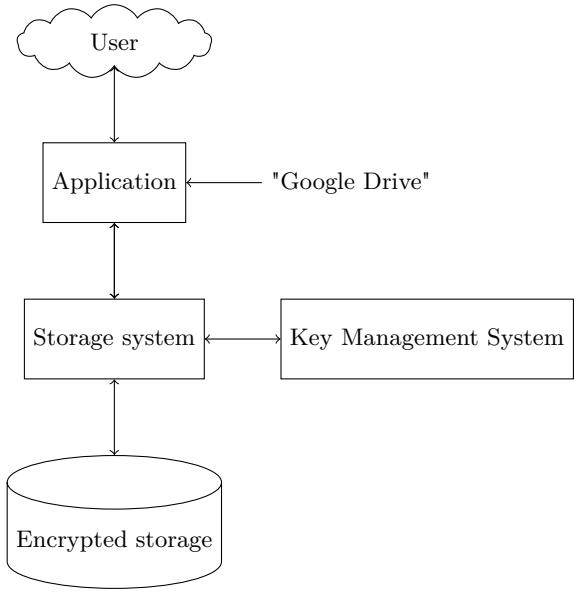

Figure 2.8: Description of encryption in the Google cloud. Figure restated from Google encryption white-paper [Goo19b]

2.5.5 Boxcryptor

Boxcryptor is a program that diverts a little bit from the aforementioned solutions[9]. This is because Boxcryptor only is a program designed to work on top of an already existing cloud solutions. It will work with providers such as Dropbox, Google Drive, iCloud Drive and many more. Boxcryptor is a downloadable program that encrypts data before upload and sharing. This protects data from end-to-end. It is claimed that the program is easy to use and should integrate well in normal work situations.

Technical solution

Boxcryptor describes their implementation thoroughly on their website[10]. Boxcryptor takes use of normal RSA key pair and also some AES keys for specific purposes. Every user has their own RSA-4096 bit key pair with OAEP padding which is used to generate a Master Key and Group keys. The AES keys are used in encryption of files and also making password keys. Boxcryptor uses servers to provide better service to all its clients. On these servers there are stored general data about each user as well as multiple keys and key material. Private keys will be encrypted by each user before it is stored on the server. They also have a function to keep track of groups and

[9]https://www.boxcryptor.com (accessed on 23.04.2019)
[10]https://www.boxcryptor.com/en/technical-overview/

their corresponding keys. Boxcryptor takes care of encrypted encryption keys and to retrieve those is a password needed. For authentication does Boxcryptor rely on accounts, which only can be accesses by providing the correct password. Encryption and decryption of files are done using 256 bit AES symmetric key cipher on CBC mode and PKCS7 padding. To share data with Boxcryptor requires that all parties have a user account, which makes it possible to use the RSA keys already established. Then a sharing happens in a PGP style of fashion. If one wants to share with a group the same thing happens only a group key is created and shared instead.

2.5.6 Comparison

Table 2.1: Comparison table of existing solutions. The ✔and ✖display whether the protocol has the attribute or not. The − indicates that the feature is a weakness, + indicates that the feature is a strength.

Solution	Dropbox	Google Drive	MEGA	Tresorit	Boxcryptor
Forward Secrecy	✔	✖	✔	✖	✔
Non-Interactive	✔	✔	✔	✔	✔
Efficient	+	−	−	+	+
Practical	−	+	+	−	−
Zero-knowledge[11]	✖	✖	✔	✔	✔

[11] In the sense that the encryption key is not stored on the server.

Chapter 3
Methodology

In order to study the efficiency and practicality of the OAGK protocol it will be useful to compare against other protocols. Two protocols have been chosen for this purpose. This chapter will go through all the three protocols OAGK, PKE and TGDH explaining why they have been chosen, how they work, key features and how they have been implemented in order to answer the research question. Several other key distribution schemes could have been chosen, but due to time restrictions and to reduce complexity where only the PKE and TGDH chosen. After the implementation description follows a section which describes the different tests and the chosen testing method.

3.1 The Offline Assisted Group Key Exchange protocol

The OAGK is a cloud-assisted protocol designed to provide higher security than common cloud sharing solutions. The goal of this thesis is to evaluate and analyse how the OAGK protocol operates compared to already established key sharing protocols.

3.1.1 Participants in the protocol

There are at least three different actors involved when the OAGK is invoked. The initiator can then only interact with one server. The responders involved are decided by the initiator, which interacts with the server.

Initiator

The initiator is the party that wish to share some data. The responsibility of choosing the responders that can partake belongs to the initiator. Only the entities that are chosen have the possibility of retrieving the shared session key. The initiator also enforce the key encapsulation mechanism in the protocol, using the key received by the server. The derivation of the cloud sharing key takes place at the initiator as well.

The final actions completed by this entity is the encryption of the computationally needed key information and the dispatching of this data to all the selected responders.

Server

The server works as a broker between the Initiator and the Responders. Although the server only interacts with each individual party one time. This makes the protocol non-interactive because there is no need for active communication between the initiator and the responders. All information that is needed for the key exchange to unfold is temporarily stored in the server and easily accessible for the responder. The server is responsible for the generation of the key pair used in the KEM. The decryption key is temporarily stored on the server while the encryption key is transmitted to the initiator. The second interaction which the server is involved in is with the responder. The server is used to decapsulate data sent by the responder and return the result. During this operation the server also verifies that the responder is indeed in the list of allowed partakers of the sharing.

Responder(s)

The responder is the entity that is at the receiving end of the key exchange, the reponder must perform a couple of operations to get the key. Information about the key sharing that is sent by the initiator is received and decrypted by the responder. This data is then used to execute the rest of the protocol. At this stage the responder can inspect the list of all entities involved and deem if this sharing is considered safe. To hide vital key information from the server a blinding factor is added by the responder. With the respond from the server can the sharing key be derived and also control checked for correctness.

3.1.2 Blind KEM

The most innovative component of the OAGK protocol is the Blind Key Encapsulation Mechanism. More specifically, the blind factor is what makes this protocol somewhat unique. This blinding element is what makes it possible to achieve forward secrecy with a server engaged in the key sharing. This server gains no knowledge about the shared key as proven in [BDGJ18]. The blinding factor prevent the server from observing or estimating the true value of the received key material.

3.1.3 How does it work

The OAGK protocol is a protocol developed and designed by (C. A. Boyd, G .T Davies, K. Gjøsteen & Y. Jiang, 2018) [BDGJ18]. It where made with security as its main property together with the ability to be non-interactive, requiring no established connection between participants.

A prerequisite for the protocol is that public and private key pairs must be generated for each user. Furthermore does the public key actually need a way to distributed among the participants. It has to be shared to other users before one can run the OAGK properly. This public/private key distribution should be handled by a trusted third party or central authority, which can be accessed by each entity. The protocol utilizes public key cryptography to attain integrity and confidentiality. Which takes use of digital signatures on transmissions and also PKC to encrypt some of the data in these transmissions. This is one of the properties that gives the OAGK a non-interactive feature. Meaning that the initiator and the responders do not need to be in direct contact at any point during the protocol run. Described in [BDGJ18] and displayed in Figure 3.1 on page 30 does the protocol start with the initiator generating a random nonce and choosing all the recipients. This data is then transmitted to the server, which then generates the KEM key pair together with the session identification. The encryption key in the KEM key pair is sent to the initiator. This key is then used in the encapsulation phase and the result from this encapsulation is used to derive the symmetric key that is going to be shared with all the recipients. Data that is needed to retrieve and recalculate the key is then encrypted using PGP style encryption and transmitted to every recipient. The recipients receive and decrypt all data and uses this together with an added blind to trigger the server. The server has the job of decapsulate received data and return the result to the responder, which in turn will be able to derive the common shared symmetric key. The responder is now able to take part in the cloud sharing because he has the symmetric key.

3.1.4 Main properties

The main goal for this protocol is to increase the security within a cloud sharing session. This is done by adding another layer of encryption to the data that is going to be shared. The data is then hidden from the cloud service provider, which is not a standard provided by common CSPs. The added encryption also has a forward secrecy feature, making the scheme even more robust and secure by protecting the session from potential future compromise of key material. Achieving this presents a couple of new problems. In this proposed solution a symmetric key has to be shared by all parties that should have access to the data. This is where the key sharing protocol comes into action. Sharing a key securely can be proven to be a difficult task and the computational cost highly depend on how many participants that are involved. This leads to a conclusion that a scalable scheme should be used since the environment is the cloud, with a potential for thousands of recipients. The distribution of the key starts from the initiator, but this initiator should not establish a connection to every other party. For the first, every recipient is not necessarily online at every given moment making contact impossible. Secondly contacting every recipient takes a lot of effort for the initiator and is not a scalable solution.

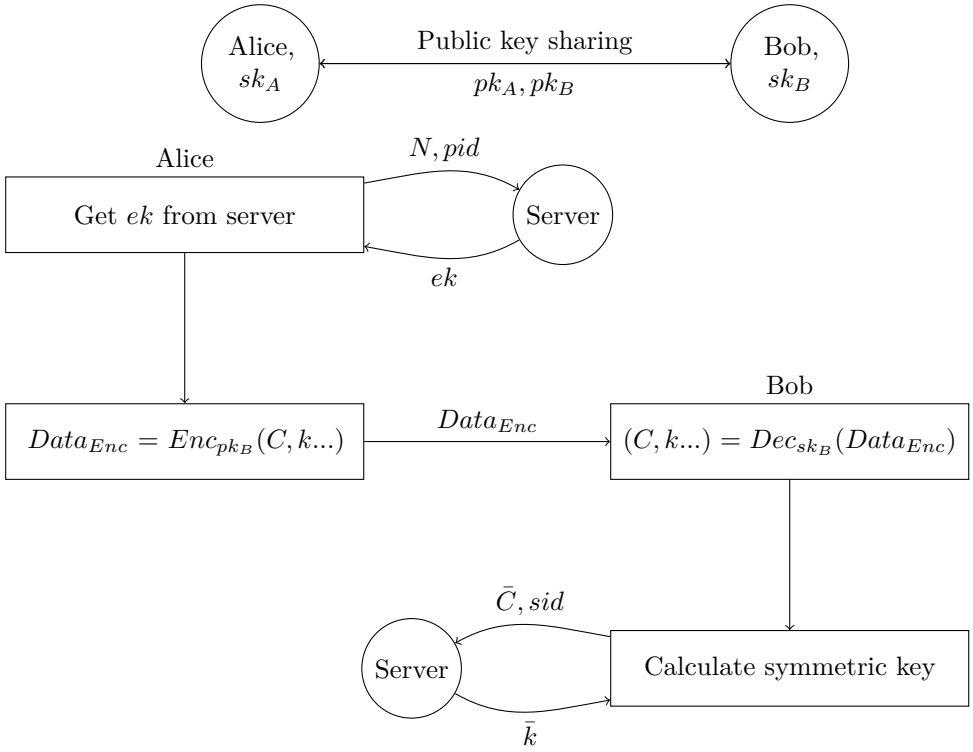

Figure 3.1: OAGK interactions [BDGJ18]

That is why the OAGK protocol is non-interactive, meaning that there is no direct interaction between the initiator and the responder. Instead, a server is introduced in the protocol and each involved recipient can contact the server to retrieve the shared key after the key sharing is set up between the initiator and the server and key material is distributed.

The OAGK is designed with forward secrecy and will have this property if implemented correctly. The requirements for the OAGK to function at desired security level are specified and discussed in [BDGJ18]. In the protocol both long term keys and short term ephemeral keys are used. As defined "short term", the ephemeral keys are important regarding security. It is proven that the protocol provides forward secrecy if there is a correct deletion of ephemeral data/key material. The server is there to calculate key material for the recipients. In order to keep the forward secrecy a blinding factor is appended by each responder to keep the data hidden from the server. The combination of long term keys and ephemeral keys gives the OAGK protocol a higher security level than other common solutions. Ephemeral

data must eventually be deleted by each participant to achieve the forward secrecy. If correctly executed, the protocol provides strong security guarantees.

3.1.5 Implementation

The implementation of the OAGK protocol happened in several stages as an iterative process. It began by analyzing the needs and features presented in [BDGJ18]. Which showed that an implementation could be made i Java, using a built in cryptographic library. A decision was made to reduce complexity by making the protocol only run locally. The system specifications are shown in Table 3.1. Simply constructing the protocol as a series of function calls that are triggered in succession. This way of implementing the protocol will therefore not take online connection into consideration. This may seem strange as this is a protocol supposed to work in the cloud. However, tests should still give valid results when it comes to efficiency and practicality. Removing the network variable will most likely improve accuracy of the test results. An analytical evaluation of how a real implementation would function will also be discussed by observing data sizes and amount of transmitted data. Implementing a prototype of the OAGK required the need of an external open source cryptographic library called Bouncycastle [1]. This library supported the creation of ECC in Java. Elliptic curve was chosen because it is the norm in modern day cryptography and requires less computation than its counterpart in RSA.

The final implementation is designed to be able to construct one initiator, one server and as many responders as desired. All entities are automatically provided a public/private key pair once created. The protocol then follows as shown in Figure 3.2.

Bouncycastle library	version: "1.60"
Runtime Environment	openjdk version "11.0.2"
Operating System	Ubuntu 18.04.1 LTS
CPU	Intel® Core™ i7-6700 CPU @ 3.40GHz × 8
RAM	31,3 GiB
Storage	502,0 GB

Table 3.1: Environment specifications

[1] https://www.bouncycastle.org/ (accessed on 01.06.2019)

The goal for this code

The main goal for this code is to make a simple prototype of the protocol, but at the same time implementing all the key features. Features such as the public/private key infrastructure. Making the protocol generically secure and choosing appropriate functions and variable sizes. This should yield a protocol that can be used as a good representative for a possible real life implementation in the future.

Challenges

There where some challenges during the implementation phase that must be acknowledged. The choice fell on Java as the programming language after a failed attempt in Python. Java is more complex but had more built in cryptography. During the implementation was there always a focus on simplicity as well as making the protocol as close to a real world implementation. Using only the paper on OAGK and advice from my supervisors I made the protocol. The paper describe quite thoroughly how the protocol should operate. It is assumed in the paper that everything needed to do PKC is already in place. This fact proved to be a small challenge, finding out how i should implement the creation and distribution of public/private keys. At first I started with a implementation using RSA keys and function for the whole protocol. I realized that it would be better and more alike the norm to use elliptic curves. This took some extra time to implement because everything i needed for the elliptic curve was not in the standard library in Java. A external library had to be implemented and eventually a functioning prototype with elliptic curve was created.

Sequence diagram and functions in the OAGK implementation

1. startServer() is initiated by the main function to start the whole protocol. All entities and their corresponding pivate/public key pairs are created at this point.
2. submitNonce(N, pid, S, this) trigger the server by sending the nonce, responder ids(pid), a digital signature and the initiator object. This triggers the server to create the BKEM key pair.
3. checkSid(ek, S, this) return the ek key to the initiator.
4. EncapAndCreateKey() Encapsulate and creates the symmetric key.
5. DecryptData(data, S, this, server) Trigger each responder by sending key the key materiel needed together with the server object and the initiator object.
6. BlindAndSign(C, sid, ek) The responder do a self call and run the function which appends a blinding factor to the key material.
7. Decapsulate(sid, blindC, S, this) Triggers the server to decapsulate the data received.
8. UnblindAndKDF(sid, blindk, S, server) The responder unblinds the data retrieved from the server.
9. ValidateKey(Tau) This function is a self call and the last operation done by the responder. Checking that the protocol ran as it should.

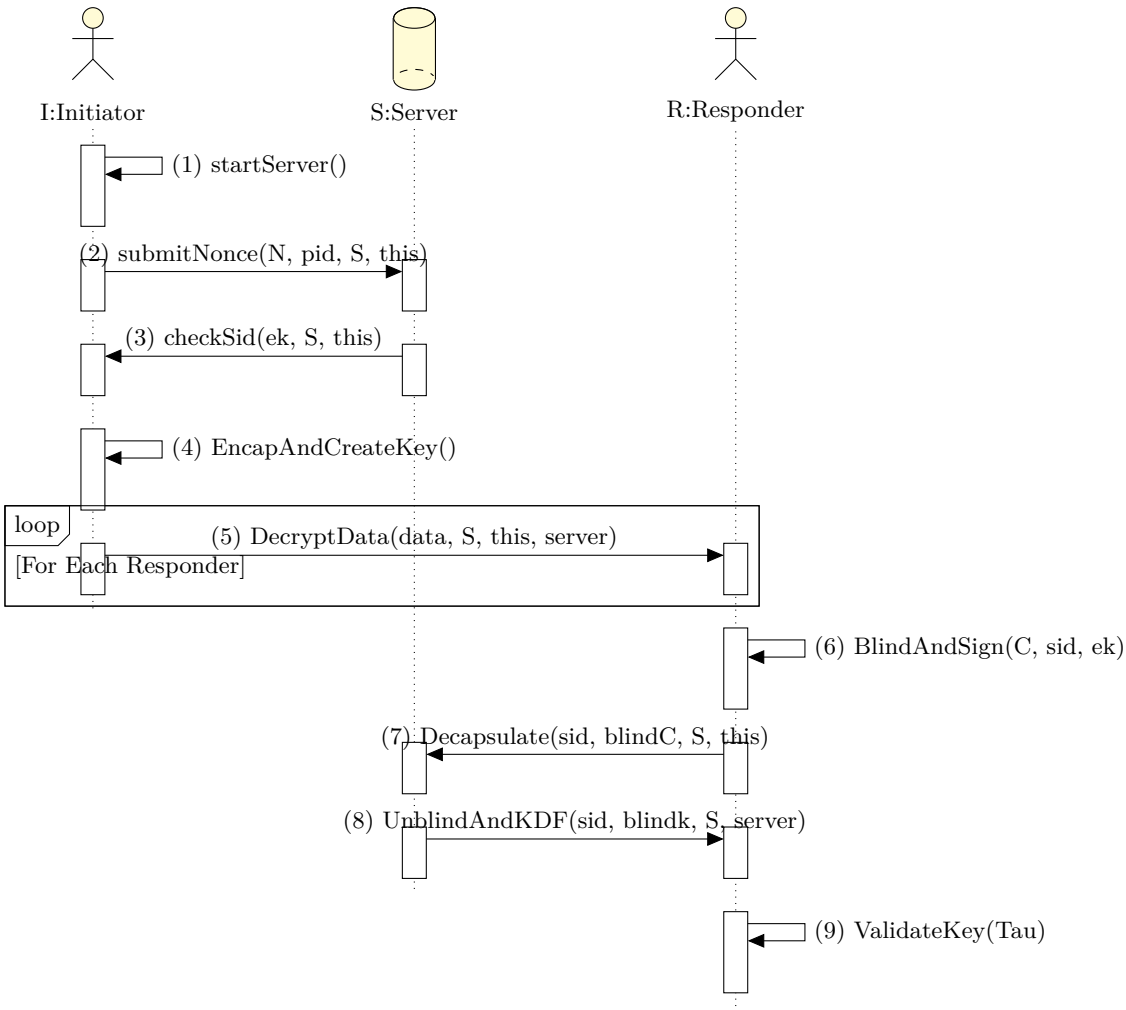

Figure 3.2: Sequence diagram of OAGK code. Showing the function calls in the implementation.

3.2 Public Key Encryption (PKE)

This is a very simple form of key transport. That is why this encryption scheme was chosen in this thesis as it will provide a lower boundary baseline. There exist many variations of PKE such like the RSA and Elliptic Curve Integrated Encryption Scheme (ECIES) and this thesis look into an simplified version of PKE with elliptic curve.

3.2.1 Participants in the protocol

Initiator

The initiator is the actor who wish to share some information. The involved parties in a specific cloud sharing is decided by the initiator. Only the recipients chosen will get the opportunity of taking part of this specific cloud sharing. The symmetric key used to encrypt and decrypt the data shared in the cloud will be generated by the initiator. Who then in turn can share it with all recipients chosen.

Recipients

The recipients in this elementary PKE does not have to perform an active action to be involved. They simply have to react when they receive the key from the initiator, decrypting the received message using their respective private key.

3.2.2 How does it work

As underlying described in [DH76] public and private keys must already be established to use this protocol. This requires a key distribution before anything can actually happen, exactly like in the OAGK protocol. When keys are established and shared the interactions displayed in Figure 3.3 can take place. Starting with the initiator creating a symmetric key using a random key generator. This key can then be distributed among the chosen recipients using each unique public key for encryption. Adopting this technique ensures that only the recipient having the private key corresponding to the public key used for encryption can retrieve the shared symmetric key. There is an authentication mechanism for the message sent by the initiator. This authentication comes in the form of a digital signature provided by the initiator which can be check by each recipient. This is a simple way of sending a symmetric key and does not provide forward secrecy and the shared data in the cloud is compromised if one of the transmitted messages are disclosed.

3.2.3 Main properties

The PKE is quite simple to use for the initiator, but can be a bit taxing if there are many recipients. For each recipient that is chosen a new encryption of the

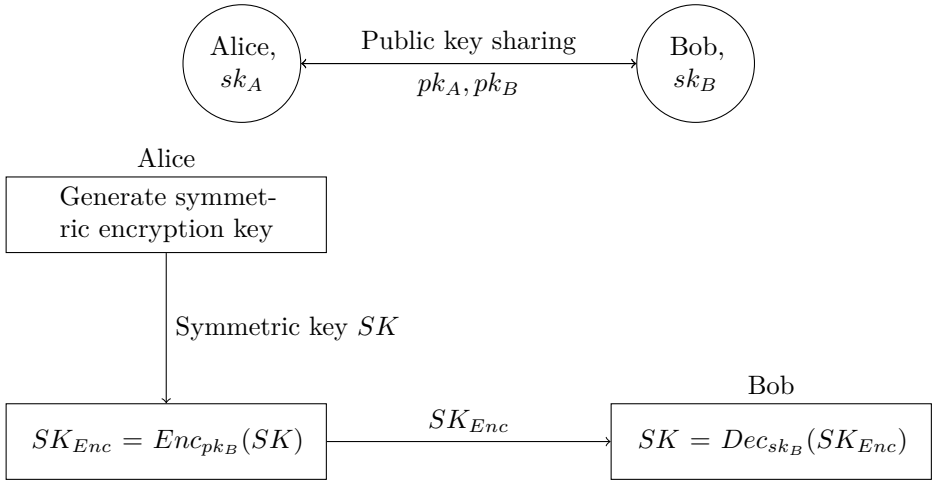

Figure 3.3: PKE interactions

symmetric key must be performed. This protocol does not scale very well, but has a low amount of computing so it may be used anyway. The initiator does not have to directly interact with each participant because the key is distributed in a PGP fashion, which makes this protocol non-interactive. Since this protocol only uses long term keys(public/private) it is vulnerable for attacks. If a private key is disclosed the symmetric key is revealed. This protocol does therefore not provide forward secrecy.

3.2.4 Implementation

The implementation of the PKE in this thesis is based on the paper [DH76] and [RSA78], freedom is taken where it was felt needed. The protocol is quite simplistic and did not need much. Elliptic curve was used and the public/private keypairs were set up the same way as in the OAGK protocol. The PKE protocol implementation was also constructed to be used on a connection less system. The same environment for implementation as in OAGK where used, showed in Table 3.1. This protocol does not have a server object and relies only the public/private keypairs for secure transmission. The protocol is designed with one initiator and can have as many responders as desired. Figure 3.4 is a sequence diagram describing the order of the function calls in the PKE implementation.

Challenges

Unlike the OAGK protocol was there no template for constructing the PKE protocol. So a thorough review of different papers where needed to establish a code which could resemble a real world implementation of the PKE. However when PKE protocol was

Figure 3.4: PKE sequence diagram

chosen where the challenges comparable to the once in the OAGK requiring almost interchangeable solutions.

Sequence diagram

1. EncryptAndSendKey() All entities and their corresponding pivate/public key pairs are created at this point. This is a function call from main, triggering the initiator to commence.
2. ConfirmEncryptionKey(signature,encryptedKey,this) Sends the encrypted key to every recipient with a digital signature. Triggering a decryption at the receiver end.
3. AccceptKey() This function confirms and accept the received key.

3.3 Tree-based Group Diffie-Hellman

The TGDH protocol utilizes the Diffie-Hellman problem shown in Definition 2.2 to generate a secure tree-structure of keys. This structure can then be adopted to be used as a key sharing protocol. It shows promise to be a secure and scalable protocol due to the tree-structure design. The TGDH protocol is chosen primarily because it is known to be used in solutions that are deployed in the world today [Tre19b]. It is more complicated and advanced than the PKE protocol. Which therefore improves the scalability of the protocol making it more suitable for a cloud environment. The protocol also has a higher level of security than the PKE, bringing it up to the level of the OAGK protocol. It can be argued that it has a form of forward secrecy and even more functionality than the OAGK. The behaviour should therefore be unlike the two other protocols making it suitable for a comparison.

3.3.1 Participants in the protocol

Initiator

The initiator is the entity that want to share some information using a cloud service. The recipients are chosen by the initiator and can be added both in the beginning of the share and after the initial share. The initiator is the only one with the authentication to include new responders. The TGDH tree is initialized by the initiator and will normally be stored on a server. The initialization starts by making the initiator the only node in the tree, having a private key, a public key, and the symmetric encryption key.

Server

The server is responsible for storing, updating and making the tree available for the entities involved. Each node in the tree is associated with a public and private key pair. The public keys needs to be accessible by both responders and the initiator for calculating the symmetric key. The tree needs to be updated every time a new leaf node is appended. A new leaf node will typically add one normal node that gets a brand new public and private key pair which creates an avalanche of key updates all the way to the top of the tree, producing a new symmetric key.

Recipients

The recipients will receive an invitation to join the key sharing tree of the TGDH protocol and only need to provide a public and private key pair to join the tree. Responders can get all the public keys in the tree from the server in order to calculate the symmetric key.

3.3.2 How does it work

The main idea for this protocol is that each participant can calculate the symmetric key used for encryption using their respective private key together with accessible public keys using a similar solution the Diffie-Hellman key exchange show in Subsection 2.2.5 on page 13, but using multiple participants. This way there is no need to send the master sharing key around. There is a downside to this solution and that comes in the form of a binary tree that must be kept up to date at all times. The tree changes every time a new participant(leaf node) is added. This means that for every new recipient update a new master sharing key that is generated. Old participants is therefore forced to calculate this key once more. The tree is a binary tree that consists of a root node, nodes and leaf nodes. The leaf nodes are the participants and are always at the bottom of the tree. Each node and leaf node has a public/private key pair associated with it. Leaf nodes have their own public/private key pair and the regular nodes have their private key generated by combining the keys of the 2

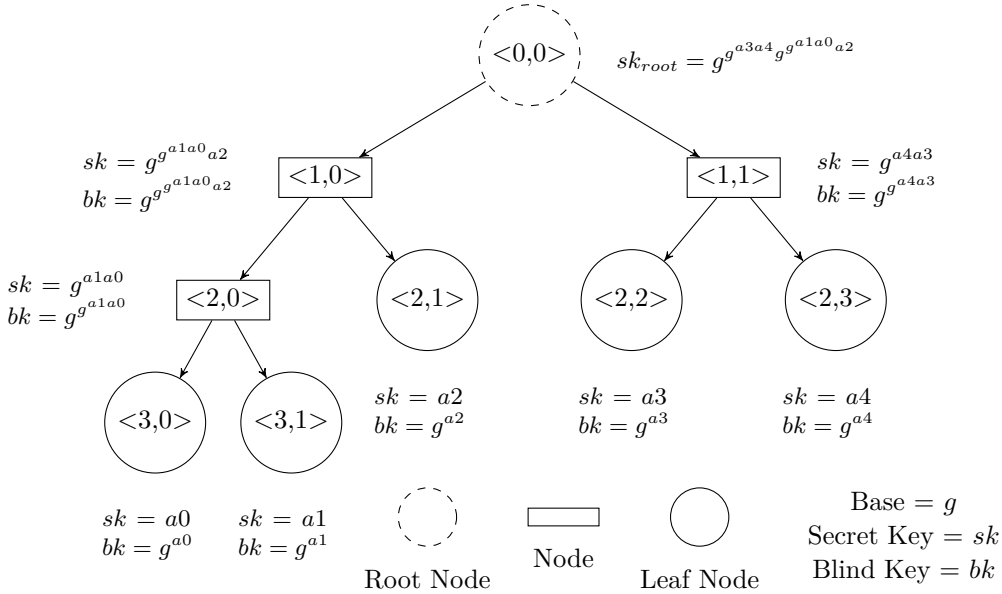

Figure 3.5: How keys are calculated in a TGDH key agreement. All nodes keep sk private and broadcast their bk. Making it possible for every leaf node (participant) to calculate upwards and end up with the same sk_{root} key.

child nodes with an exponential calculation. The computation starts at the bottom by one node taking its sibling nodes public key to the power of its own private key. This generates the parents private key. The parent nodes public key is then the base to the power of the newly generated private key. This way makes it possible for a leaf node that has access to all public keys to calculate the master sharing key as shown in Figure 3.5. The tree is as described updated each time a new leaf node is added by recalculating the the parent node keys which leads to an avalanche effect on the higher layers in the tree, recalculating keys of parent nodes until one reach the root node which gets a new master sharing key.

3.3.3 Main properties

The TGDH protocol is easy for the initiator to use and require only a small effort to create the tree and add responders. Distributing a key using the protocol removes the need for a centralized group key distribution[KPT04]. This protocol could be argued to have forward secrecy as long as the public/private key pair is generated for each session. The protocol could have forward secrecy if the initial tree is generated by the initiator after interacting with all responders. Removing and appending responders will however divert the TGDH protocol from the definition of forward secrecy as the

generated Diffie-Hellman keys for all entities must be stored during the entire session, hence not being ephemeral. The need for direct communication between the initiator and the responders can be omitted, hence the protocol can be non-interactive. The TGDH also have the ability to add or remove participants during a session. However this initiates a rekeying which affects all entities. The symmetric key is dependent on every participant, and every time the involved parties change so does the shared symmetric key. Interruptions of the cloud service could therefore occur frequently [ZR04]. However this rekeying stops when all participants have joined the cloud share. Once the initial set of recipients have joined the tree rekeying should happen infrequent.

Unlike the two previously explained protocols is the TGDH a fully decentralized protocol [LSB12]. Meaning that there is no one entity that calculates and decides the key. There is no trusted third party involved and the key computation is based on the Diffie-Hellamn problem for groups. This creates the rekeying issue as mentioned, but at the same time distributes and spreads the workload over all entities. The key freshness in this protocol is good because of the change in key with each append to the binary-tree. Ensuring that no parties other than the joined ones have the shared key. TGDH requires asynchronous (public/private) keys like the two other protocols to create an encrypted channel between the entities.

Four important cryptographic properties that the a normal group key exchange protocol and the TGDH should have is restated below from [KPT00]:

- **Group key Secrecy** guarantees that it is computationally infeasible for a passive adversary to discover any group key.
- **Forward Secrecy** guarantees that a passive adversary who knows a contiguous subset of old group keys cannot discover any subsequent group key.[2]
- **Backward Secrecy** guarantees that a passive adversary who knows a contiguous subset group keys cannot discover preceding group key.
- **Key Independence** guarantees that a passive adversary who knows a proper subset of group keys cannot discover any other group key.

3.3.4 Implementation

The implementation of the TGDH is mostly taken from [KPT02] with the code from a Github repository[3] and an website explaining how the code work[4]. This code was

[2]This is a different definition to what is used throughout this thesis.
[3]https://github.com/lemmy/de.rub.hgi.nds.tgdh (accessed on 02.06.2019)
[4]https://www.ruhr-uni-bochum.de/nds/forschung/gebiete/tgdh/ (accessed on 02.06.2019)

quite complex and hard to understand. It therefore took some extra time and effort to comprehend what was happening. The required libraries and packages where eventually added to the Java code and the TGDH protocol was running. Then it was simply the case of adding the different entities to the code in similar fashion to the two previous protocols. The implementation on Github had a lot of functionality meaning that there was little extra to implement when it was finally working. This Github solution could append, remove and calculate the symmetric keys for all entities. No server where implemented as there was no need in order to get the code to function properly. The binary tree was created as its own object and could therefore be called by all other entities.

Challenges

The source code was hard to get working, as this code was quite old and had some dependencies that was troublesome to implement. The source code was also complicated and with no or little description to help understand it. The creation of the binary tree was also a demanding task, it did not want to build like a balanced tree. This was because one had to determine the position of the next leafnode and that proved to be a difficult task. One unresolved challenge was to replace the DSA keys that where used in the source code. These keys where used to derive the secret and public keys in each node and could have been changed to make the code simpler, faster and more understandable.

3.4 Testing

Setup and equipment

The environment specifications which the tests are run in are shown in Table 3.1 on page 31, displaying both software and hardware used. The VisualVM [5] software is used to surveillance and monitor applications memory heap, classes loaded and thread activity in real-time and at the same time displaying the results. VisualVM can do this and still not have a large effect on the running program. All the tools used are publicly available.

Each test is designed to test a feature that is important for a working real-world implementation. The tests should go through many stages of scaling with a variable amount of recipients. This should provide graphs showing the development for increased variable sizes. The graphs could also show the progress of each protocol as a function of time.

[5] https://visualvm.github.io/ (accessed 11.05.2019)

The architecture of each test is designed with simplicity and repeatability in mind. Using built inn functions in addition to some supplementary third party software should provide adequate results which are stable with a low variance. The data collected should have a decently low standard deviation to be counted as valid. The testing environment differ from a real world environment, and everything runs on a local system. Connection speed and capacity is therefore taken out of the equation, reducing complexity and giving more stable results. However these protocols are supposed to be implemented in a cloud environment and are affected by those factors. Hence measuring transmitted data sizes and the amount of computation needed is crucial. These sizes should be manageable to compute and get by investigation the current offline implementation. Reproducibility and repeatability of the test is also managed by making the source code publicly available and describing each test thoroughly.

3.4.1 Time usage

How much computational effort and time does each entity use during a protocol run? How is this measure distributed over the distinct entities and the different function calls.

Why

The efficiency of a protocol is good to know in order to know how it operates and behave. This information can then be used to determine suitable areas of use. A vast majority of the efficiency a protocol has can be estimated by how fast it can operate. This is why the time a protocol uses to do certain operations can be a good indicator of how efficient the protocol is. The number of computations will by an extent influence the time usage, where more computations implies more time. A protocol with a low efficiency will be expected to need a large amount of time compared to an efficient solution. This test could also reveal how scalable each protocol is.

How

To actually get quantifiable results, there a need for making a standard test or at least make the tests comparable. To measure the efficiency of each protocol a timer is going to be used. Each protocol is made up by multiple stages and what each stage does varies. That is why it can be valuable to make multiple measures on different parts of the protocols. Producing a timestamp for each function of the code will provide the amount of time each one uses. Comparing results from tests with different parameters should then be possible to do.

Important phases in each protocol:

- Setup
  - OAGK, PKE and TGDH: Creating and distributing Public/Private key pairs.
- Initiator operations
  - OAGK, PKE: How much time and effort must the initiator use in total for a given key distribution. Creating the symmetric key, encrypting and distributing key material.
  - TGDH: Initialize the binary tree. Sending invites to the given recipients.
- Server operations
  - OAGK: Reacting to the initiator. Creating the BKEM key pair. Computing the session ID. Interaction with each recipient, decapsulating the received key material.
  - PKE: No server needed.
  - TGDH: A server should store and distribute the binary tree containing all public keys. Making it possible to calculate the symmetric key for the leaf nodes. Server also confirms that entities are allowed to join.
- Responder operations
  - OAGK: Receive and decrypting data from the initiator. Create a blinding factor and applying this to the key material. Receiving and unblinding data from the server. Use the KDF function to calculate the key, and verifying the result.
  - PKE: Receive and decrypting data from the initiator, and accepting this as a symmetric key.
  - TGDH: Join the tree be calculating the new public keys from bottom to the top of the new binary tree. Should be $log(n)$ operations, where n is the level where the new leafnode is appended.
- Everything combined
  - OAGK, PKE and TGDH: How long time does it take from the start of a key distribution(without the setup phase) to complete each protocol.
- Adding/removing participants
  - OAGK: Rerun the protocol with new responders, both for removing and adding.
  - PKE: Rerun the protocol with new responders, both for removing and adding.

    - TGDH: Used the implemented join or leave functions.

- Updating key
    - OAGK: Encrypt the data with a new key and rerun the protocol with the same responders.
    - PKE: Encrypt the data with a new key and rerun the protocol with the same responders
    - TGDH: Change the private key of one leafnode changes the symmetric key. Causing a cascading effect where every entity has to recalculate the key, but in order to achieve forward secrecy must all entities generate a new secret DH key.
- Uploading/downloading cloud content
    - OAGK, PKE and TGDH: How is this affected by each protocol.

To measure the runtime a time stamp is created just before starting the protocol and another timestamp is created after a complete run. The difference between each timestamp will be appended to a list, which later is used to retrieve the results. A loop will be created in order to run a specific part of the protocol and sample many data points. This process is reproducible and can be done numerous times. The results can then be estimated by using all the data points collected and then take the average. It is important to notice that in most cases it is beneficial to remove the warm up phase of the data sampling as this phase is not representative for the true result value.

System.nanotime(); was used and timestamps where created at different point in the code to record the used time of distinct sections. The procedure for sample collection was do 500 runs while creating timestamps throughout the code. The sample size of timestamps was therefore varying depending on the protocol and how many timestamps i created during one run. To do calculations on the data set where the first sets of data removed so that timestamps from the last 50 runs where left. Calculate average and standard deviation based on the timestamps from these 50 last runs and use this as the results. The reason for removing the first data is to get rid of any warm up time data that might affect the result. Normally java improves runtime over time and discarding the earlier data will therefore give more stable results.

3.4.2 Data size

How much space does each protocol take up in RAM, disk space and during transmission.

Why

The amount of transmitted data can determine whether the protocol is suitable for cloud sharing use. This data could be a limiter for the protocol to work properly and could severely restrict the functionality and efficiency of the protocol. The amount of bandwidth available must also be taken into considerations because this will affect the impact the size of transmitted data has. Reducing data signalling and data transfer is always valued in a real cloud environment setting. The size of data could also have an effect on the amount of storage needed. Which is also essential to consider when this effects the system the protocol is run on. The amount of storage that is necessary to occupy after and during the protocol run may also be different, even between each entities.

How

Measuring the size of data that is created, stored and transmitted during a code run. There are several ways of measuring the size of variables in each protocol. It can be important to distinguish between the size of the data and how much memory it uses on a system. The memory usage can be surveyed by a third party software and the actual physical size of data can be measured using built in functions in Java. An analytical approach can also be used to evaluate the sizes of data in storage.

VMMonitor can be used to get an overview of the RAM usage with increasing amount of entities involved in a cloud share.

The miscellaneous data in each protocol are displayed in Table 3.2, Table 3.3 and Table 3.4 with three different data types: Long term (L), Ephemeral (E) and Transmitted.

RAM usage

VisualVM can be used to monitor the running Java program. This can give a strong indication of the memory usage of each protocol. It can also be used to provide a snapshot of the heap, containing all the created data elements. What the smallest and largest amount of storage needed can also be calculated with an analytical approach.

3.4.3 Practicality

Practicality can be a subjective measure and the security level needed may dictate the practicality of a solution. It could also be a measure of how well it incorporates into already operational systems. Is there a downside to use a given protocol when evaluating from both a user perspective and for a CSP? Can the protocol

Table 3.2: OAGK storage space

Data	**Transmitted**	Type	**Entity**
Public/private Key Pair	✖	L	I/S/R
Nonce	✔	E	I/S
PID	✔	L	I/S/R
KeyEncryptionKey ek	✔	E	I/S
Shared Symmetric Key	✖	L	I/R
Symmetric Key IV	✖	L	I/R
SID	✔	L	I/S/R
BKEM key pair ek/dk	✖	E	S
BKEM encryption key (ek)	✔	E	I/S
τ	✔	E	I/R
k	✖	E	I/R
C	✔	E	I/R
uk	✖	E	R
$\bar{C}$	✔	E	R/S
$\bar{k}$	✔	E	R/S
Signature (σ)	✔	E	I/S/R
Encrypted Key material (c)	✔	E	I/R

Table 3.3: PKE storage space

Data	**Transmitted**	Type	**Entity**
Public/private Key Pair	✖	L	I/R
Shared Symmetric Key	✔	L	I/R
Symmetric Key IV	✔	L	I/R
PID	✖	E	I
Signatures (σ)	✔	E	I/R
Encrypted Key material (c)	✔	E	I/R

Table 3.4: TGDH storage space

Data	Size (bytes)	Type	Entity
Public/private Key Pair	✖	L	I/S/R
Shared Symmetric Key	✖	L	I/R
Symmetric Key IV	✖	L	I/R
Binary Tree	✔	L	I/S/R
PID	✖	L	I
Share invite	✔	E	I/R
One public key	✔	E	I/R

be implemented without to much effort or can it be moved from one software to another?

Why

How practical a protocol is can greatly influence the area of use. How practical a protocol is can be looked at from several perspectives. Is it practical to implement for a CSP? Is it easily implemented on different devices and on various operating systems? Is the code portable and manageable to transfer from one unit to another. Users might look different on how practical the protocol is and how they are willing to trade inconvenience for security. If the protocol fills need for a large enough group and they are willing to use it, then the protocol can be considered practical. Practicality can also be measured in cost. Where cost could be time, money, effort, storage needed, functionality, maintenance and installation difficulty.

How

Determine the practicality of each individual protocol can quickly become a subjective measure. The goal should be to find some objective criterion to evaluate each protocol. There are several aspects that can be looked at when estimating the practicality of a protocol. One has to do an assessment from the user and also from the CSP standpoint. Some of the measures have to be analytical and some practical.

Some key points that are important to analyze when comparing practicality:

- User standpoint
  - Ease of use
  - Is it worth the extra security?
  - Mobility - multiple devices and OS

- Availability
- Reliability
- Intractability
- Scalability
- Interoperability

– CSP standpoint

- Software
- Mobility of the code
- Easy to implement and maintain?
- Hardware requirements
- Scalability
- Cost/Reputation
- Legality

3.4.4 Scalability

"The capacity to be changed in size or scale." - Oxford Dicitonary. How well does each protocol cope with a large amount of involved entities. Are they suitable to handle large groups?

Why

Scalability is key in a cloud setting environment, where there can be millions of users that share data. A bad designed protocol can seriously hamper the functionality, practicality and efficiency of the protocol. If the protocol is supposed to be used in a cloud environment scalability is a necessity. Not all parts of the protocol has the same impact on the scalability, and bottlenecks can created when a lot of work is distributed to one node. The scalability has a huge impact on whether the protocol is a viable option to use in a real environment. Does the protocol scale with many users and does it also scale with multiple sessions.

How

To asses the scalability of a protocol one has to look at all parts of the protocol, but it is normally a weakest link that will limit the scalability. That is why every part of the protocol must be evaluated, even though some part are indicated through the analytical work beforehand to be more susceptible to scaling issues. The proportion of contribution by each entity may differ in each protocol. In order to evaluate scalability, tests with escalating number of users will be done. The procedure will

be similar to previous testing, with numerous runs where a large proportion of the first data will be thrown away and the rest will contribute to the average result. The storage space scalability will be discussed based on results from previous findings.

3.4.5 Security properties

No physical test will be conducted to determine the security properties of each protocol. When implemented properly, each protocol should have the same security level. Still an analytical comparison will be performed in order to establish differences between the security levels.

Choice of variables, algorithms and random functions

The choice algorithms, functions and variable sizes will have a great impact on how fast and secure a given protocol is. These features should therefore be thought through before a protocol is put into action. Different functions operate at various speeds, require different storage space and also required computational power. Creating a secure protocol requires focus on security in every step and layer of designing and implementation. One badly chosen random function could render all other security features useless. There only need to exist one error or fault to compromise the whole protocol. Each function and variable have various security quality. For instance, if one where to choose a hash function one has a large range of functions available. If security is important a hash function that has the features listed in Subsection 2.2.8 on page 15 should used. This thesis aims to test the functionality, practicality and efficiency of the chosen protocols. It is therefore meaningful to get the protocols designed as similar to what they would be in a real environment. Although it is not vital for a comparison to yield results. What is important regarding the security properties in each protocols is that the security is homogeneous throughout the protocol. This means that all the various security features have a similar degree of security. This removes the chance of a weakest link. The OAGK, PKE and TGDH should all share approximately the same level of security in their algorithms, functions and variables.

3.4.6 Mode of Use

Each case is meant to provide a deeper understanding and characterization of the handling of each of the chosen protocols. This will show how they behave and function under different circumstances. The cases are meant to be linked to real world scenarios and should therefore reflect some normal use cases that happens often. In each case there is only one variable that will change and that is the number of responders, as this is likely to change in the real world as well.

Case 1(Regularly sharing)

This is a case where one Initiator regularly establish cloud sharing with the same recipients. In this case the Initiator needs to deliver a new symmetric key each time. Making a more robust and secure cloud storage.

Case 2(Appending or removing recipient)

If a shared key is already established among a set of entities, are they affected by adding or removing a single entity? What is the difference in time, effort and practicality between each protocol and which one is the more appropriate choice in this scenario.

Case 3 (Rekeying)

A key has already been distributed between the entities. This key is getting old and has been in use for some time. In order to improve security a new key should be generated and replace the old one. This case exist when a cloud share is going to exist and be used for a long time.

Case 4 (Very basic receiver)

What happens when an Initiator want to share with one or more recipients that have a low capacity for computing. This scenario can occur if one tries to connect to small censors or devices. For example having measuring instruments placed out of physical access and they all connect to a cloud service. These devices must then partake in a key agreement scheme in order to upload securely.

Case 5 (One-shot)

An initiator sends one key to a number of receivers and then these parties never interact with each other again afterwards. This could simply be a distribution of a document or a data file.

Chapter 4
Results

This chapter contains the most important results found in this study. Firstly are the results of the efficiency test displayed, followed by the evaluation of data sizes. Then information about the practicality of the protocols are laid forth.

4.1 Time usage

The time each protocol uses can be divided into several parts as the various entities must do different tasks. Stages that will describe the efficiency and performance are the setup, the total runtime and each individual entity computational effort. These measures among others are studied in this section.

Setup time

Setup time is the amount of time needed to do the preparing work before the protocol can be run. This involves distributing public/private key pairs.

The setup for all three protocols is very similar. It is identical for the PKE and OAGK and similar in the TGDH. In order to use the TGDH protocol each receiver needs to compute a private and a public key which is the same as with the two other protocols, but the TGDH computes DSA keys and the OAGK and PKE generate keys designed for elliptic curves. The setup time for all protocols are dependent on the number of users and is only required to be done once. The results are quite similar and have approximately the same linear increase shown in Figure 4.1, but the TGDH line is higher than the OAGK and PKE.

Runtime

Runtime is the time it takes from the Initiator start distributing the key until all the recipients have calculated the key.

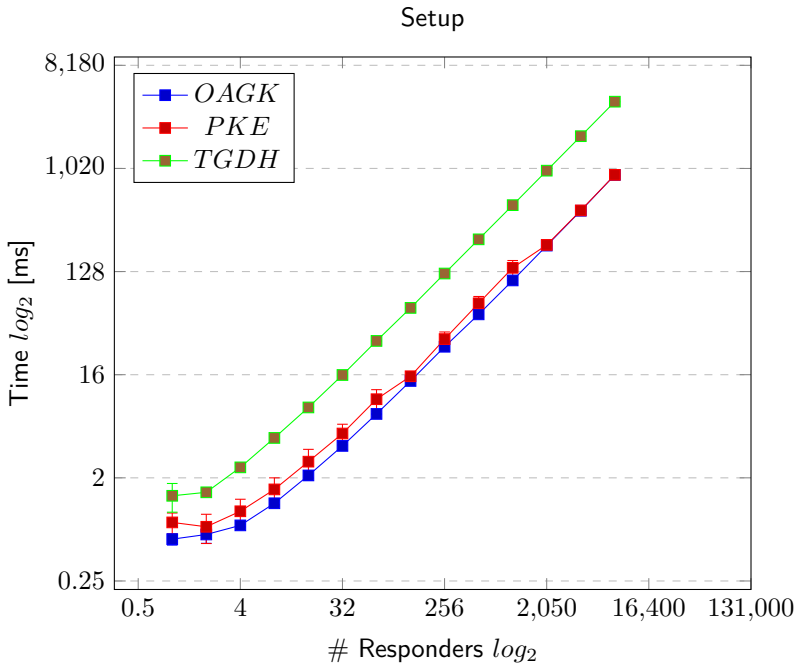

Figure 4.1: Setup Time Usage

Running tests on all the protocols produce a number of different results. Multiple timers gave an overview of all the different parts in each protocol. The total amount of time usage for a distribution of keys shown in Figure 4.2. This corresponds to the time it takes from the initiator to begin until the last recipient receives the symmetric key, meaning the time it takes to distribute a symmetric key. Here we can see that the OAGK and PKE have an almost linear runtime with a small time penalty as the number of responders increases. The OAGK time usage is higher than the PKE and is more affected by the number of users. The TGDH protocol is faster than the other protocols up until the highest tested user amount of 65536. An important consideration for the PKE and OAGK protocol is that the responders can work in parallel and the time to finish will therefore depend on the server ability compute the key material for each responder. The amount of time one responder uses in the PKE and OAGK be viewed in Table 4.3 and Table 4.2 where the different function runtimes are displayed. Responders can not operate in parallel in the TGDH, but the initial tree can be constructed by the initiator alone if all responder public keys are distributed, but each responder still has to calculate the shared symmetric key with the function runtimes displayed in Table 4.4.

The time that each distinct entity uses in each protocol varies. In both the OAGK

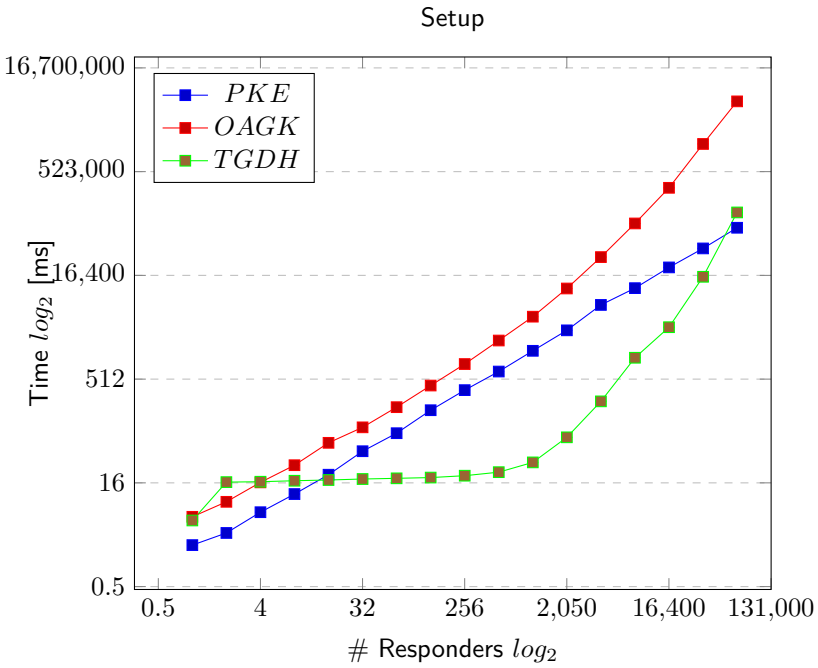

Figure 4.2: Total Time Usage

and PKE there is a linear relationship between the amount of time the initiator uses and the number of responders. This is shown in Figure 4.3 and Figure 4.4. Each recipient only has to do a small amount of work to receive the key. The Initiator has to do operations for each responder, thus use a lot of effort when the amount of receivers rises. The OAGK protocol also needs a server, which uses less time and effort than the initiator and reciever. The TGDH protocol is more complex as this needs a server to distribute the tree. The amount of work the server uses should be small as it is only to distribute the public keys in the tree when needed. The amount of work that has to be done by the initiator is small compared to the other protocols, however each entity needs to recalculate the shared key when there is a change. This recalculation requires $\log_2(n)$ computations, where n is the amount of entities involved in the share. This work is almost equivalent to what a new entity must do to join the tree. This makes the actual computation needed larger than the measured result. The TGDH Initiator does not have to do a lot of work compared to the other entities as shown in Figure 4.5.

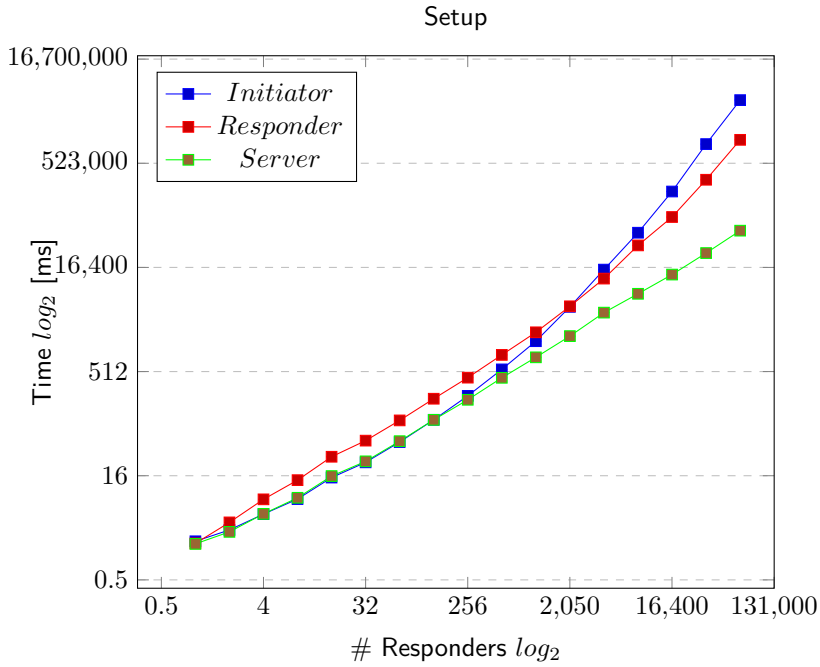

Figure 4.3: Time Usage OAGK

Table 4.1: Time usage table. Different parts of each protocol have a varying degree of time usage and Table 4.1 displays this by indicating with a + sign for being a strength, − for weakness and = if they are approximately equal.

	OAGK	PKE	TGDH
Setup time	+	+	−
Rekeying time	−	−	+
Ciphertext encryption time	=	=	=
Public key generation time	=	=	=
Secret key generation time	+	+	−
Key distribution time	−	−	+
Initiator time usage	−	−	+
Responder time usage	+	+	−
Server time usage	+	None	−

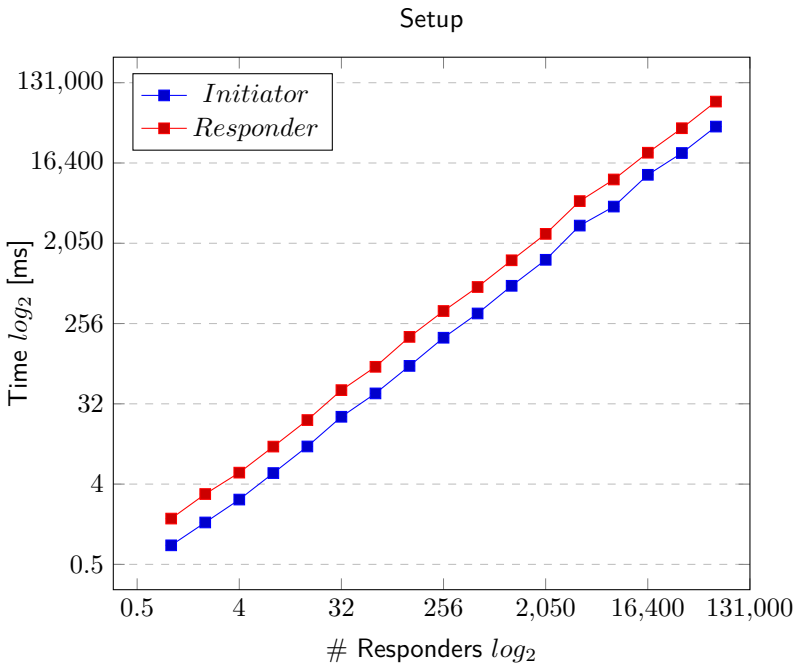

Figure 4.4: Time Usage PKE

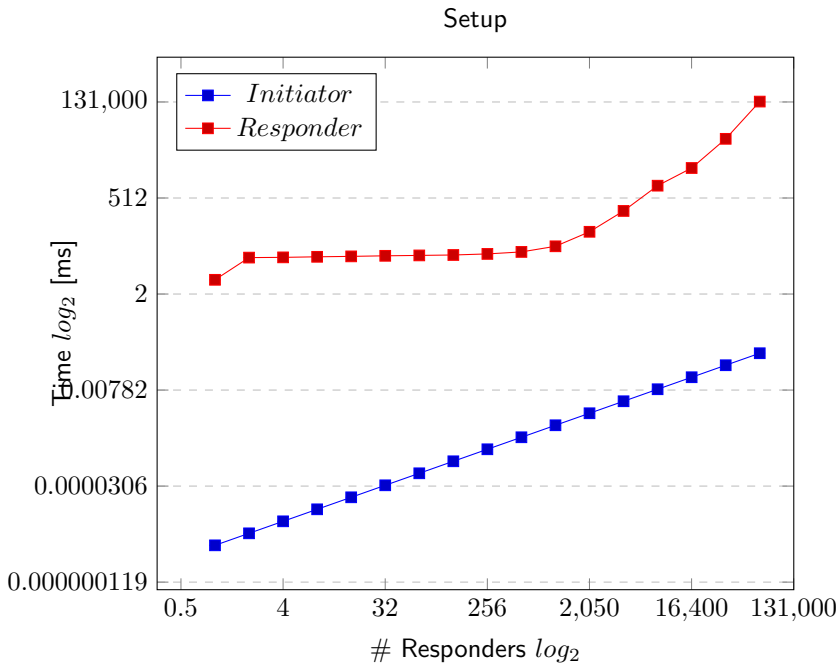

Figure 4.5: Time Usage TGDH

Function runtime

The tables below shows the estimated runtime for each part of each protocol, extrapolated from the results shown in Appendix A. The results are displayed as the total usage of time. This means that a function that uses $\mathcal{O}(1)$ amount of time but is run by n entities will in total have a runtime of $\mathcal{O}(n)$.

The OAGK protocol functions:

Table 4.2: OAGK functions.

Function	Behavior	Time usage[1] (ms)
startServer()	$\mathcal{O}(1)$	0.3360708
submitNonce()	$\mathcal{O}(1)$	0.78815004
checkSid()	$\mathcal{O}(1)$	0.25488204
EncapAndCreateKey()	$\mathcal{O}(1)$	0.44222338
Choosing responder and encrypting	$\mathcal{O}(n)$	0.43413842
Signing and sending encrypted data	$\mathcal{O}(n)$	0.33896396
DecryptData()	$\mathcal{O}(n)$	0.5991359
BlindAndSign()	$\mathcal{O}(n)$	0.58976368
Decapsulate()	$\mathcal{O}(n)$	0.8695125
UnblindAndKDF	$\mathcal{O}(n)$	0.49482956

One thing that slows down the runtime is the increasing size of the PID list containing the IDs of the involved entities. Sending and doing operations on this list require an increasing amount of time as it grows.

The PKE protocol functions:

Table 4.3: PKE functions.

Function	Behavior	Time usage[2] (ms)
CreateSymmKey	$\mathcal{O}(1)$	0.027496496
Encryption	$\mathcal{O}(n)$	0.79177713
Decryption	$\mathcal{O}(n)$	1.63788555

The PKE is quite simple and only the Initiator gets an increase in workload as the amount of receivers becomes larger.

[1] for 1 Initiator, 1 server and 1 responder
[2] for 1 Initiator and 1 responder

The TGDH protocol functions:

Table 4.4: TGDH functions.

Function	Behavior	Time usage[3] (ms)
InitilalizeTree()	$\mathcal{O}(1)$	0.11293326
ShareKey loop	$\mathcal{O}(n)$	0.00287673
joinShareTree()	$\mathcal{O}(n\,log_2(n))$	6.02021663
Recalculate key	$\mathcal{O}(log_2(n))$	6.02021663

The TGDH protocol shows promise for being more scalable with operations that require less computational power than the two other protocols, but it has one additional part which is the recalculation of keys every time a new node is appended to the binary tree. Which is not accounted for in the results and is and extra burden on all the involved parties.

4.2 Data size

The three important types of data that must be considered is long term, ephemeral and transmitted data. Evaluating the exact size of data in Java has proven to not be straight forward, but roughly estimating and making an analytical calculation is possible.

4.2.1 Transmitted data

I assume that all public/private key pairs have already been distributed. This should only be necessary to do once and therefore counts for a small amount of the data traffic in the long run. Otherwise all data that must be sent between entities are described in table Table 4.5, Table 4.6 and Table 4.7 for each respective protocol.

OAGK

The signature is a computed hash that is a part of every transmitted message and is always the same size. The encrypted key material consists of the Encap values C, ek, τ, SID, PID. So the size is dependent on these values. The value of PID is the only one that can change in this implementation since this is a list of all the involved entities in the key distribution. The encrypted key material also has to be sent to each responder and the true size of the transmitted data must therefore be multiplied by the number of responders. This also holds for the Blind C and the Blind k. In addition is the SID actually sent twice per responder. The total value is for a run with 1 Initiator, 1 responder and 1 server.

[3]for 1 Initiator, 1 server and 1 responder

Table 4.5: OAGK data size.

Data	Size (bytes)
Nonce	16
PID	Depends
Signature (σ)	32
BKEM encryption key (ek)	32
Encrypted Key material (c)	144 + PID
SID	32
Blind C	32
Blind k	32
Total	480 + 2*pid

The design of the PID varies and in my implementation it is simply just a list with integer IDs which take up 4 bytes each. This means that the space needed is $4 \cdot n$ where n is the number of responders. Another option is to send a list containing the public key of the recipients. This would mean that there would be 32 bytes per responder. Another option is to send a list with the same amount of bits as entities in the public key list, indicating with a 0 or 1 whether the entity is a recipient or not. This means that the list must be n bits long where n is the number of entities in the public key list.

PKE

Table 4.6: PKE data size.

Data	Size (bytes)
Signature (σ)	32
Encrypted Key material (c)	48
Total	80

The encrypted key material consist of a symmetric key and an IV. The total value is for a run with 1 Initiator and 1 responder.

TGDH

The transmitted data in this protocol depend on the size of the binary tree. In order to function properly there is a need for each entity to recive the changes if the symmetric key has been altered so they can recalculate it. I therefore expect that most of the traffic in this protocol will be between a server and the entities when information about the current tree is requested. Since only the changed data in the tree is necessary to send, this should be $\log(n)$ public keys as this is what is changed when an n-th responder joins the tree.

Table 4.7: TGDH data size.

Data	**Size (bytes)**
Share invite	$\mathcal{O}(1)$
One public key	32
Total Binary tree	$2 \cdot n \cdot 64$
Total	256

The binary tree should represent the size of the entire tree, but it is important to remember that the whole tree is not needed to calculate the shared symmetric key. If a tree consist of N nodes, meaning that there are N/2 leaf nodes, one leaf node only needs to know its own private key in addition to one public key at each level in the binary tree. Hence $\log(N)$ nodes must be known by a leaf node in order to calculate the symmetric key. The total is for a run with 1 Initiator, 1 responder and 1 server.

4.2.2 Long term and ephemeral data

The data displayed in Table 4.8, Table 4.9 and Table 4.10 are the theoretical sizes of each data element. Ephemeral data in the tables are data that must eventually be deleted in order to maintain the security.

OAGK

Table 4.8: OAGK storage space.

Data	**Size (bytes)**	Retention	**Entity**
Public/private Key Pair	64	Permanent	I/S/R
Nonce	16	Ephemeral	I/S
PID	Depends	Permanent	I/S/R
KeyEncryptionKey ek	32	Ephemeral	I/S
Shared Symmetric Key	32	Permanent	I/R
Symmetric Key IV	16	Permanent	I/R
SID	32	Permanent	I/S/R
BKEM key pair ek/dk	64	Ephemeral	S
τ	48	Ephemeral	I/R
k	32	Ephemeral	I/R
C	32	Ephemeral	I/R
uk	32	Ephemeral	R
$\bar{C}$	32	Ephemeral	R

The data stored in the OAGK implementation is quite large and the RAM usage is displayed in Figure 4.6. This figure shows the RAM usage for increasing amount of responders, 16384 being at the last high point in the graph. We can see that there is a linear growth in the amount of storage needed. This makes sense when looking at Table 4.8 where we can observe the amount of data needed for each of the corresponding entities.

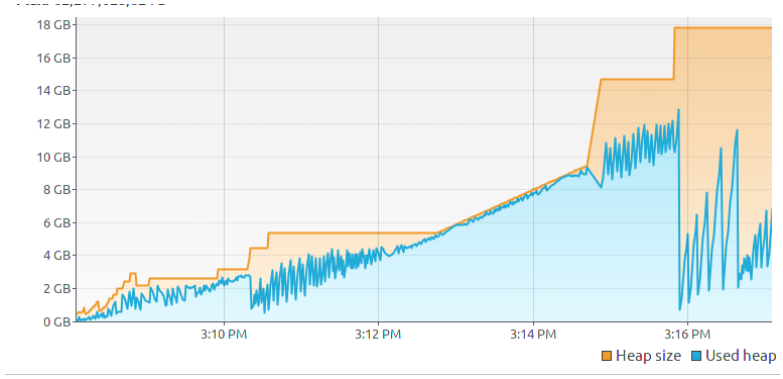

Figure 4.6: OAGK RAM usage

Table 4.9: PKE storage space.

Data	Size (bytes)	Long term (P)/ Ephemeral (E)	Entity
Public/private Key Pair	64	P	I/R
Shared Symmetric Key	32	P	I/R
Symmetric Key IV	16	P	I/R
PID	Depends	E	I

PKE

The PKE protocol has fewer data object as shown i Table 4.9 and thus less data that need to be stored. This can also be seen in the Figure 4.7, where we can se the the amount of storage needed for an increasing amount of recipients and the last section is for 65536.

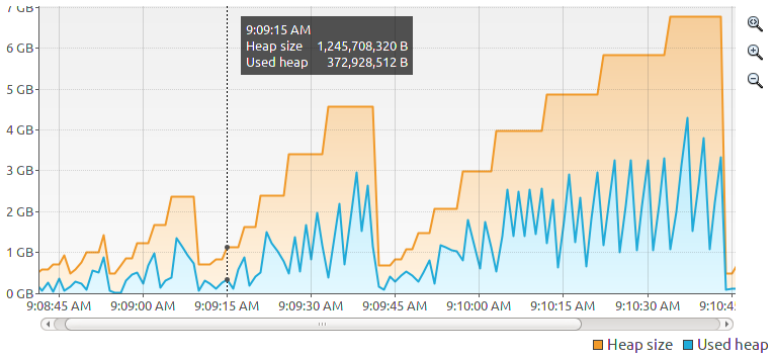

Figure 4.7: PKE RAM usage

TGDH

Table 4.10: TGDH storage space.

Data	Size (bytes)	Long term (P)/ Ephemeral (E)	Entity
Public/private Key Pair	64	P	I/S/R
Shared Symmetric Key	32	P	I/R
Symmetric Key IV	16	P	I/R
Binary Tree	$2 \cdot n \cdot 64$	P	I/S/R
PID	Depends	P	I

The TGDH protocol does not require much of the Initiator and the recipients when it comes to storage. However the server needs to store the whole binary tree. The size of data depends on the amount of responders and this is shown i Table 4.10

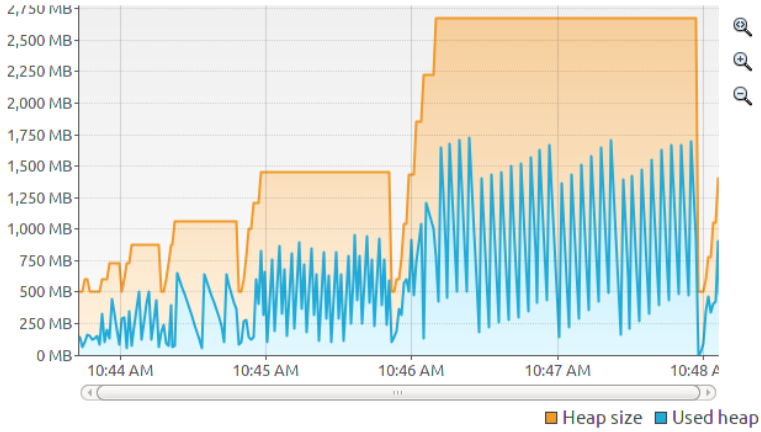

Figure 4.8: TGDH RAM usage

as a function of n. Since only the tree is stored at one entity does it not take up much space. What takes up the most space is the public/private key pairs which are not that large. One can see that the amount of storage space increases with an increase in the number of responders in Figure 4.8 where the last section is for computing with 65536 responders.

4.3 Practicality

In order to evaluate the practicality of a protocol must multiple different features be assessed from two contrasting standpoints. How practical is the protocol for the users and for the CSPs. All aspects must be weighted to determine the functionality and practicality of each protocol.

4.3.1 User standpoint

Each protocol could in theory be implemented with the same interface. Since they have the same functionality and should be equally easy to use. However, the TGDH has currently more functionality and is therefore easier to interact with. Each implementation is done in Java and should therefore run on all Android devices as well as equipment that supports Java. All the three protocols are non-interactive, meaning that no entity has to be in direct contact. The availability is affected by this. The PKE is not dependent on anything except for one transmission to each recipient. A server is needed in the OAGK protocol which could be a bottleneck or single point of failure and could therefore affect the availability. The TGDH requires that an update can be sent to all involved entities whenever the binary tree changes

or that each party can inquire for changes at the beginning of each cloud access. This reduces availability.

The coding complexity increases in PKE to OAGK and TGDH. This is an increase in both length of code and the comprehensibility. This could affect reliability and also more data is transmitted with the increasing complexity, giving a higher risk for something to go wrong.

Both the PKE and OAGK have a linear growth in time usage and the number of computations needed when the number of responders increases. Most of this work is done by the Initiator, making that entity a bottleneck. The PKE requires less computation and will be more scalable than the OAGK. The TGDH operates with a growth of $n \cdot log(n)$ where n is the number of entities involved in the protocol. Even though the slope of the time used curve is higher compared to the two other protocols. does the computations in the TGDH requires less effort and the TGDH is therefore faster up to a certain point. One can see in Figure 4.2 that the TGDH is catching up with the two other protocols with increasing amount of recipients. Albeit, the workload is spread much more evenly over all entities and the Initiator does considerably less work than in the other protocols.

Each of the three protocols are implemented as an add-on or third-party software to support a cloud service. It should therefore not be too complicated to implement versions that would support different CSP solutions, but the more complex the protocol is the harder it becomes to make it work and interact with a different systems.

The amount of storage space needed varies between the protocols and also between the entities within each protocol. All the protocols are required to have the public key of all entities. They also need space in order to temporary store data during computations and also space to save important data such as the symmetric key. In the PKE only the symmetric key is necessary to save. The OAGK requires saving a larger amount of data and the TGDH requires the savings of $log(n)$ public keys, where n is the number of nodes in the binary tree.

4.3.2 CSP standpoint

Code mobility is essential in order to spread the code to many devices and broaden the area of possible use. This together with how difficult and complex it is to install the code on different devices will affect the profit and cost of the protocol. Maintenance of the protocols software and platform hardware depends on how much it is used. Upgrades are also in required in order to be competitive or secure further increasing costs of complex protocols. The higher the complexity is the higher the cost is.

Table 4.11: Practicality table, + is good.

	OAGK	PKE	TGDH
User standpoint			
Ease of use	++	++	+
Mobility[4]	++	++	++
Availability	++	++	+
Reliability	++	++	+
Scalability	+	+	++
Interoperability[5]	++	+	+
Storage space	++	+	+
CSP standpoint			
Code mobility	+	+	+
Ease of Setup	++	+	+
Maintenance	++	+	+
Hardware requirements	++	+	+
Cost	++	++	+
Legality[6]	+	+	+
Storage space	++	+	++

For the CSPs earnings are important and hardware requirements have an effect on the profit. A more complex protocol with more computations will require better hardware. The storage space needed by the CSP is affected in the OAGK where the server needs to store the BKEM key pair and the TGDH needs to store the binary tree.

4.4 Security

All the implementations are programmed with the intention of being at the desired real world security standard level. This was done in order to get the code as similar as possible to the real world scenario. However, some functions and features may not be inherently secure and this code should not be used in a real environment. The intention for security in the chosen protocols was for them to have the same level of security within all functions and also similar security between each protocol. Each protocol also use TLS to establish a secure connection before transmitting data between each other. The protocols have therefore an additional layer of security

[4]Does it support multiple types of devices?
[5]Does it work with different cloud services?
[6]Is it allowed with encryption in a country?

Table 4.12: Security comparison table

Protocol	OAGK	PKE	TGDH
Forward secrecy	✔	✖	✔✖[7]
Zero-knowledge encryption[8]	✔	✔	✔
Symmetric Encryption algorithm	AES	AES	AES
Transmission scheme	TLS	TLS	TLS
Ephemeral data	✔	✖	✖[9]
Signature scheme	ECDSA	ECDSA	ECDSA
Key derivation scheme	HKDF[10]	RNG	SHA256
Private/Public Key size	256 EC	256 EC	256/2048 DSA

on top of the TLS. The security properties of each protocol are not alike and are displayed in Table 4.12.

OAGK

The OAGK protocol uses Elliptic curve secp256r1 [TBY+09] throughout. Random functions are supported by Java and the function SecureRandom() reduces the chance for a repeated output. Using 128 bits makes it computationally infeasible to break, but it uses a PRNG and should actually never be used in security critical applications. The private/public key pairs are of type elliptic curve cryptography and are of size 256 bit. PKC with the ECIES scheme with 256 bits key size is used for encrypting the AES key and IV used during transmission of key material. This AES key and IV are both 128 bit in size. The shared symmetric key is 256bit with an IV of 128 bit. The shared symmetric key size can be changed to mach the security of the other parts of the protocol. The BKEM keys *ek/dk* are generated using a 256 bit random integer. The encapsulation happens with a new 256 bit random integer. The blinding factor also uses a 256 bit random integer. The KDF is a built in Java function called HKDF and uses SHA256. This SHA256 is also used to generate the SID and also applied when creating signatures with ECDSA.

PKE

The PKE protocol uses Elliptic curve secp256r1 [TBY+09]. Random functions are supported by Java and the function SecureRandom() reduces the chance for a repeated output. Using 128 bits makes it is computationally infeasible to break, but

[7] Can have forward secrecy at the cost of non-interactivity.
[8] In the form that the no other party gains information from the interaction.
[9] One could say that TGDH have long term ephemeral data. Data must be stored as long as a session last.
[10] Chosen in this thesis, could be something else.

it uses a PRNG and should actually never be used in security critical applications. The private/public key pairs are of type elliptic curve cryptography and are of size 256 bit. PKC with the ECIES scheme with 256 bits key size is used for encrypting the AES key and IV used during transmission of key material. The shared symmetric key is 256bit with an IV of 128 bit and like in the OAGK could easily be changed. The signatures are made with ECDSA using SHA256.

TGDH

DSA keys are used in the tree to generate the private and public numbers. The TGDH key specifications are set to p=random 2048bit, q=random 256bit and g=random 128bit. Random functions are supported by Java and the function SecureRandom() reduces the chance for a number repeat. The DSA private key is generated to a 256 bit number and the public DSA key is 2048 bit. The shared symmetric key is 256bit with an IV of 128 bit and like in the OAGK could this easily be changed. The implementation in this thesis does not use digital signatures for this protocol, but when implemented in a real environment the TGDH could also use ECDSA for strong security.

4.5 Mode of Use

The following cases are chosen because they represent some different situations that can happen when using the cloud as a sharing platform.

Case 1(Regularly sharing)

The TGDH has more functionality than the OAGK and PKE. It is also quite efficient to initiate new sharing when the binary tree is already in place. Both the TGDH and OAGK has strong security features, as opposed to the PKE which does not have forward secrecy. All three protocols are non-interactive, but the TGDH requires more active interaction with the server. In order to establish a new sharing with the PKE and OAGK protocol do they have to be run again, but the TGDH can already use the established binary tree. This can be done by simply changing one number in a leaf node, thus changing the shared symmetric key. Although this action requires that each entity must recalculate the shared key.

Case 2(Appending or removing recipient)

In the case of OAGK and PKE the only way to append or remove a responder is to create a new symmetric key, re-encrypt the shared data and then distribute this new key to the desired recipients. When it comes to appending is it in fact the same as simply running the protocol one more time with the new responders, but the whole

protocol has to be run again when removing a responder. The OAGK and PKE could append a new responder by running the protocol with only the new responder. The TGDH protocol is simpler to append or remove responders, simply add or remove the desired leaf node, and the symmetric key will then change automatically.

Case 3 (Rekeying)

With the OAGK and PKE a new symmetric key must be calculated and distributed. This must de done with all the involved parties. The TGDH only needs to have one of the leaf nodes to change the private key to change the shared symmetric key. However, this compromise the forward secrecy in this protocol as the session keys must be stored. This means that if a private session key is compromised so are all the shared symmetric keys derived with this private key compromised. It would normally be the initiator that calculates new public/private key pair and use this to alter the tree. All the other involved parties must then recalculate the shared symmetric key in order to have the new correct one.

Case 4 (Very basic receiver)

With the case of a basic receiver the importance of security may not be as important. There may be low storage space available, little or no energy source, very limited computational power. The PKE is very simple and has low amount of computation required for the receiver. In addition is there no need to store anything else than the shared symmetric key with the PKE protocol. The OAGK requires more computation and effort by the receiver, forcing the receiver to interact with a server and calculating various numbers. There is also a need for larger available storage space. The TGDH symmetric key changes every time a new responder is added to the binary tree. This forces already established responders to recalculate the shared key, increasing the work that the responder must do.

Case 5 (One-shot)

An initiator sends one key to a number of receivers and then these parties never interact with each other again afterwards. Security, speed, reliability and availability is important in this case. The key distribution service must be available when it is going to be used. It must also be reliable so that it works the time it is used. The TGDH protocol has a dynamic shared key which changes for every new responder, making it hard for a responder that can only be contacted once to eventually have the correct symmetric key. The PKE could work because of the non-interactive feature, but it lacks strong security. The OAGK have both strong security and non-interactive properties.

Chapter 5
Discussion

This chapter is the evaluation of the *OAGK* protocol with relation to efficiency, practicality, usability and performance. I will discuss what the results indicate and what limitations that are in this study.

5.1 Summary

The goal for this thesis was to evaluate the efficiency and practicality of the OAGK protocol, as well as determine how an increase in security level will affect the usability of the protocol. With this in mind the results can be analyzed and the OAGK can be gauged compared to each of the other protocols. The results indicate that the OAGK protocol has some benefits and a few disadvantages. In sheer numerical data comes the OAGK protocol short in both the speed, computation and storage compared to the PKE and the TGDH. However the OAGK protocol walks a middle ground and has other stronger features such as high security and not to high complexity which can make the protocol practical in some cases. The findings also demonstrate a correlation between the number of users and a decrease in speed and increase in the amount of storage needed. Indicated in the Figure 4.2 the OAGK operate with a linearly increasing time consumption and is therefore quite predictable. The amount of work that has to be done by each recipient in the OAGK also makes it quite efficient from their standpoint, suggesting that the protocol is suitable in certain situations.

5.2 Interpretation

The results from the speed and computation test can be broken down into several parts. For each of the protocols one can look at the individual entities and evaluate how much effort is needed there. When looking at the data for the PKE and OAGK one can see that the amount of work done by the single initiator increases linearly with the amount of responders. This means that the initiator must do a lot of work

when the amount of recipients increases, and it could therefor become a bottleneck or high cost problem. In the case of the OAGK the server also gets more work when the amount of responders increases. This work is linearly correlated to how many responders there are. The responders in the OAGK however have to do the same work every time which makes this protocol suitable for recipients with a low amount of computing power like in Case 4 with a basic receiver in Section 4.5 on page 68. It is the same with the PKE protocol, which actually does not have a server, removing a lot of computation. The TGDH uses a binary tree for distribution of the symmetric key and the computations needed is therefore dependent on the size of this tree. The number of computations is therefore $n \cdot log(n)$, which is higher than the two other protocols, but we can see in the results that with a low amount of recipients the TGDH is faster.

The data size in each of the protocols correlates somewhat with the amount of computation needed, especially the load of data that is transferred over the network. The TGDH protocol has a low amount displayed in the results, which could be a misleading number since actually more traffic is needed there in order to update the key for each entity every time the binary tree is changed. This also requires a server to keep the tree on record and also respond to entities that want an update. The PKE has a low amount of data that needs to be transferred and could be a good starting point when using a channel with a low bandwidth or if transfer cost is a major issue for example when transferring to remote areas or when using a congested network during peak hours. Data sizes are larger in the OAGK, but this may be because of the implementation. The PID list can be quite extensive, but also be reduced by quite a lot as explained in the results, thus improving the OAGK protocol.

If strong security is imperative, for example when sharing confidential business documents, then it is clear that the PKE is not suitable, as this does not offer forward secrecy. However it does have the non-interactivity feature which makes it suitable for cloud use. The OAGK has both the forward secrecy property and the non-interactivity. It also has strong security properties as proven in [BDGJ18] and if implemented properly a high security level in practice. The TGDH features forward secrecy as well and could be regarded as non interactive as the involved parties do not have to actively interact with each other and could use a server to distribute the necessary data. Rekeying in the TGDH protocol is simple because only one of the leaf nodes has to change their key in order to change the shared key. Unfortunately does this causes a cascading effect where all the involved parties must recalculate the symmetric key in order to have the correct one. This case does not happen in the OAGK where the only way to rekey is to distribute a new key by running the protocol again, which is the same amount of effort required to start a completely new share.

When evaluating the practicality of the OAGK protocol one must consider what is practical. Since this can be a subjective measure, based on whether a high level of security is worth a small time or cost penalty. A time penalty may also be negligible if the protocol is still fast enough to satisfy the user, a large percentage in speed difference may not be noticeable in a real-world implementation. The computation and speed data seems to indicate that the OAGK is more expensive and slower than the two other protocols, it also needs more storage space, which is increasing the cost. However, the scalability of the OAGK is quite good if the initiator is willing to do a bit of work. All the protocols are easy to use when implemented as a third party software, but as of now the TGDH have more functionality. The reliability and availability of the protocols should be higher in both the PKE and OAGK since they are less complex and rely less on an independent entity such as a server. Intractability between the different protocols and various types of software should be very much alike and they should all be able to run on different kinds of systems.

5.3 Implications

In a world where security and privacy are becoming more and more important it makes sense to investigate and improve the viable options for enhanced security in cloud sharing. Removing the trust issue, where the CSPs can be left out and not gain or have access to user data in the cloud is an important step to take. As reviewed earlier in Section 2.5 there exist solutions that omit this issue already. Some of them are third party applications that run on top of an already existing cloud and some are cloud solutions with built in security measures. The OAGK protocol could be used with either one of these solutions and it is a matter of cost for the parties that want to use a system with higher security. Forward secrecy is as mentioned previously an important aspect to have when implementing security in a network environment as this reduces the impact for a future compromise of key material. The existing solution Tresorit uses the TGDH protocol and is a well functioning third party solution that works with multiple cloud environments. This protocol a god indicator of how well the OAGK is operating. The results contribute to an understanding that the OAGK shows promise as a functioning and secure protocol when implemented correctly. A seamless implementation of the protocol would be a viable option as this improves the ease of use for the users, but a third party solution would have a broader impact area and could easier be implemented with different clouds.

While the focus of the paper [BDGJ18] was to introduce and demonstrate a secure solution that omits the trust issue in cloud solutions, this thesis has showed that the solution can be implemented and be almost as efficient as a real-world existing solution with the same security level, proving that the OAGK protocol may be taken into consideration in certain cases. The results also make it clear that there are some

improvements to be made regarding the implementation, which should improve the efficiency of the OAGK protocol, affecting the cost and also usability by making it more practical.

5.4 Limitations

The result can be generalized to be used in a real world-environment, but it is not tested in a real-world setting which could have been useful. This could potentially have shown weaknesses and elements that are left out during tests conducted on a local system. Using Java as the implementation and testing environment also gave some challenges. With more time all the code would be more effective and less costly when considering computations. The implementation is also limited by the experience and skill of the programmer. This fact is no problem as long as the all the code is made by that person, but in this thesis some of the code was retrieved and used from Github. Even though the code was reviewed before use was it quite complicated and could affect the results in both directions. Java is also self improving, which makes testing more complicated. An attempt to omit this was done by running the code several times before capturing data. This could anyway affect the data and there is a trend in the TGDH result data that indicates this. The reliability of the result data can therefore be questioned, but since the same procedure is done for testing each of the protocols should not matter. The results data in numbers should therefore not be compared with other studies in raw numbers, but there is information to be gained about the behavior of the studied protocols. Efficiency may also not be the most important factor. The underlying platform may be so inefficient that it does not matter or an improvement may be too small to even be noticed by humans.

The true size of data was not recorded during this thesis, only the theoretical size. This should give a good indication to what storage space is needed and what size the transmitted data would be. However transmitted data would require some overhead to be sent which makes the results only a part of the true size. This could be tested in a real-world environment implementation test. The size of RAM needed captured in Figure 4.7, Figure 4.6 and Figure 4.8 shows the true size of RAM utilized by the testing environment. It shows the combined RAM usage of all users and is also not very detailed. One can therefore only draw conclusions on the maximum required RAM and average required RAM. Evaluating the storage uptake of objects in Java proved not to be as simple as expected, hence only theoretical sizes was provided in the results.

Although this thesis has touched upon the functionality of the different protocols there has not been a thorough evaluation on how this will affect the usability of the protocol. The functionality is something that should be a focus area in the future as this may have an impact on the practicality of the OAGK protocol. Whether this

functionality affects cost or profit has not been reviewed in depth here. Practicality is a subjective measure and I will naturally weight features and properties different than others. This fact could have an impact on the chosen assumptions and facts about the protocols. However the Case study section 4.5 on page (67) provides a variety of cases where each protocol is evaluated in a different setting.

Chapter 6
Conclusion

This chapter will bring back the research questions and make some concluding remarks based on the results found in this thesis. Followed by a short presentation of possible future work.

6.1 Answering the research questions

How efficient and practical is the OAGK protocol compared to other known solutions? In this thesis the OAGK protocol was compared to the PKE and the TGDH protocol. The OAGK performs quite similar to the PKE and is slower up to a certain point than the TGDH. However the OAGK shows promise by offering strong security while at the same time not being to inefficient. The difference between the protocols is not too large to deem the OAGK unfit to be used as a key establishment protocol for cloud sharing. The amount of storage space needed and the amount of transmitted data is greater in the OAGK protocol compared to the two other protocols. However these sizes are so small that under normal circumstances this should have no major effect on the efficiency of the protocol. The protocol can be implemented as any other third-party solutions or it can be implemented directly into a cloud solution. The OAGK lacks some functionality compared to other solutions, but if one is regarding security as an important feature then the OAGK could be practical in multiple cases.

How is the performance and usability of the OAGK in a real world setting? All the existing solutions described in Section 2.5 have been established by professionals and competent programmers. This means that they should be quite effective and functional. The solutions are spread worldwide and used every day by millions of people. It is therefore hard to determine if the performance of the OAGK can match the existing solutions. The usability of the OAGK should be quite good and the protocol is possible to implement to different kinds of cloud solutions. If the

implementation is made seamless and user friendly, I see no real downside to using the OAGK in order to improve security of an already existing cloud solution.

Is the heightened security level worth using in a real cloud environment? The importance of security is spreading and more solutions require a high level of security. Making a solution that can improve the security of any existing cloud sharing solution is therefore worth investing in. Even though there exist solutions that work with strong security capabilities the OAGK can be a nice addition to this group of protocols. The efficiency is not too bad compared to known solution and the practicality could be better but it is still a solutions that I could see implemented in a real cloud sharing solution.

6.2 Future work

In order to get an even more through evaluation of the OAGK protocol, it should be implemented in a real-world environment and tested with existing cloud sharing solutions. This would give a more precise and comparable evaluation on whether it can compete with the already existing solutions. This testing environment would be more complex and require more time in order to get a fully functional implemented OAGK protocol. More use case testing may also be needed in order to determine if the OAGK has some areas where it excels compared to other solutions. Improvements could also be made to the OAGK protocol in order to reduce the amount of work done by the Initiator. It could be interesting to investigate if attribute-based encryption could be used as PKC to reduce the workload.

References

[And14] Jason Andress. *The basics of information security - understanding the fundamentals of InfoSec in theory and practice, Second edition.* Syngress, 2014.

[Bar16] Elaine Barker. Guideline for using cryptographic standards in the federal government:cryptographic mechanisms. Standard 800-175B, National Institute of Standards and Technology, National Institute of Standards and Technology Attn: Computer Security Division, Information Technology Laboratory 100 Bureau Drive Gaithersburg, 2016.

[BBB+12] Elaine Barker, William Barker, William Burr, William Polk, Miles Smid, Patrick D. Gallagher, and Under Secretary For. Nist special publication 800-57 recommendation for key management – part 1: General, 2012.

[BDGJ18] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. Offline assisted group key exchange. *IACR Cryptology ePrint Archive*, 2018:114, 2018.

[BKMM96] Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors. *Computer Security - ESORICS 96, 4th European Symposium on Research in Computer Security, Rome, Italy, September 25-27, 1996, Proceedings*, volume 1146 of *Lecture Notes in Computer Science*. Springer, 1996.

[Cal17] Michael Calderbank. The rsa cryptosystem: History, algorithm, primes, August 2017. http://www.math.uchicago.edu/ may/VIGRE/VIGRE2007/REUPapers/FINALAPP/Calderbank.pdf.

[CS06] M. Cremonini and P. Samarati. Contingency planning management. In H. Bidgoli, editor, *Handbook of Information Security*. Wiley, 2006.

[DDC17] Farid Daryabar, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Cloud storage forensics: Mega as a case study. *Australian Journal of Forensic Sciences*, 49(3):344–357, 2017.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

[Dro19a] Dropbox. Dropbox business security. Technical Report 1, Dropbox, 3 2019. https://www.dropbox.com/static/business/resources/Security_Whitepaper.pdf.

[Dro19b] Dropbox. Under panseret: Oversikt over arkitektur, 2019. https://www.dropbox.com/business/trust/security/architecture.

[Gar95] Simson L. Garfinkel. *PGP - pretty good privacy: encryption for everyone (2. ed.)*. O'Reilly, 1995.

[Gjø16] Kristian Gjøsteen. Diffie-hellman and discrete logarithms, November 2016.

[Goo19a] Google. G suite-sikkerhet og tillit, 2019. https://gsuite.google.no/intl/no/security/?secure-by-design_activeEl=data-centers.

[Goo19b] Google. How google uses encryption to protect your data. Technical Report 1, Google, 2 2019. https://storage.googleapis.com/gfw-touched-accounts-pdfs/google-encryption-whitepaper-gsuite.pdf.

[IBM19] IBM. Digital signatures. https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.12/gtps7/s7dsign.html, February 2019.

[Jea16] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016.

[Knu98] Jonathan Knudsen. *Java cryptography*. Java series. O'Reilly, 1998.

[KPT00] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000.*, pages 235–244, 2000.

[KPT02] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. *IACR Cryptology ePrint Archive*, 2002:9, 2002.

[KPT04] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. *ACM Trans. Inf. Syst. Secur.*, 7(1):60–96, 2004.

[LSB12] István Lám, Szilveszter Szebeni, and Levente Buttyán. Invitation-oriented TGDH: key management for dynamic groups in an asynchronous communication model. In *41st International Conference on Parallel Processing Workshops, ICPPW 2012, Pittsburgh, PA, USA, September 10-13, 2012*, pages 269–276, 2012.

[MEG18] MEGA. Mega security white paper. Technical Report 1, MEGA, Level 21, Huawei Centre, 120 Albert Street, Auckland 1010, New Zealand, 12 2018. https://mega.nz/SecurityWhitepaper.pdf.

[Mer78] Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, 1978.

[MG11] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

[MVOV96] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of applied cryptography*. 01 1996.

[Nat94] National Institute of Standards and Technology. Digital signature standard (dss). FIPS Publication 186, May 1994.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[SWZ05] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring data integrity in storage: techniques and applications. In *Proceedings of the 2005 ACM Workshop On Storage Security And Survivability, StorageSS 2005, Fairfax, VA, USA, November 11, 2005*, pages 26–36, 2005.

[TBY+09] Sean Turner, Daniel R. L. Brown, Kelvin Yiu, Russ Housley, and Tim Polk. Elliptic curve cryptography subject public key information. *RFC*, 5480:1–20, 2009.

[Tre19a] Tresorit. Cloud storage + end-to-end encryption, 2019. https://tresorit.com/security/encryption.

[Tre19b] Tresorit. Tresorit white paper. Technical Report 2, Tresorit, 3 2019. https://tresorit.com/files/tresoritwhitepaper.pdf.

[Vac04] John R. Vacca. *Public Key Infrastructure: Building Trusted Applications and Web Services*. Auerbach Publications, Boston, MA, USA, 1st edition, 2004.

[ZR04] Xukai Zou and Byrav Ramamurthy. A block-free TGDH key agreement protocol for secure group communications. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, Innsbruck, Austria, February 17-19, 2004*, pages 288–293, 2004.

Appendix A
Test Results

Speedtest PKE

Table A.1: PKE functions 1 responder

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	21663.656	6285.1673175552
Encryption	729580.18	82867.5586072596
Decryption	1546351.61	167745.637632452

Table A.2: PKE functions 2 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	21663.656	6285.1673175552
Encryption	729580.18	82867.5586072596
Decryption	1546351.61	167745.637632452

Table A.3: PKE functions 4 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	13650.336	3853.67597173193
Encryption	666471.24	30952.3659877949
Decryption	1345603.82	62021.7262701354

Table A.6: PKE functions 32 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	10565.168	2854.21454200205
Encryption	714593.09	97133.9450998563
Decryption	1424666.79	202688.645998255

Table A.7: PKE functions 64 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	11261.272	4389.51062488929
Encryption	654338.7	16802.9706882444
Decryption	1303864.82	37099.9225081078

Table A.8: PKE functions 128 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	12755.24	4195.63936562713
Encryption	667484.64	22260.3345062557
Decryption	1415353.35	76540.4624683409

Table A.4: PKE functions 8 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	12220.4	3052.47195040347
Encryption	663673.66	23732.1030661928
Decryption	1320888.13	83509.4608638632

Table A.5: PKE functions 16 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	12272.968	7702.8913978438
Encryption	661781.23	28056.1210654128
Decryption	1312440.1	31531.9631721528

Table A.9: PKE functions 256 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	10743.52	3692.0019557958
Encryption	692201.65	140239.735336771
Decryption	1382247.73	272254.243699042

Table A.10: PKE functions 512 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	10723.344	3604.06015511173
Encryption	648694.29	13048.1522142371
Decryption	1287363.62	40443.228515236

Table A.11: PKE functions 1024 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	10034.52	2835.57009604771
Encryption	664194.25	176986.527459599
Decryption	1285918.59	31915.5142741254

Table A.12: PKE functions 2048 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	9593.312	2536.17479970447
Encryption	651178.59	35163.1377547838
Decryption	1274139.28	25895.0944428013

Table A.13: PKE functions 4096 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	10614.456	2999.80917660841
Encryption	789739.33	405467.228003597
Decryption	1488550.81	228663.435672768

Table A.14: PKE functions 8192 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	9622.136	2561.11771644804
Encryption	645979.44	6534.65987381134
Decryption	1303377.64	449216.384941656

Table A.15: PKE functions 16384 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	10067.64	3118.89443078794
Encryption	735636.78	842184.376750075
Decryption	1303099.75	61882.1851828739

Table A.16: PKE functions 32768 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	9261.864	2463.93138327836
Encryption	646555.95	9653.55919997904
Decryption	1228631.78	11632.7861534372

Table A.17: PKE functions 65536 responders

Function	Time usage(nanoseconds)	Standard deviation
CreateSymmKey	9147.016	3345.81985883042
Encryption	641132.79	9528.19465512224
Decryption	1225141.48	24631.827037181

Speedtest OAGK

Table A.18: OAGK functions 1-16 responders

Function	Time usage(nanoseconds)	Standard deviation
1		
startServer()	336070.8	0.3360708
submitNonce()	788150.04	0.78815004
checkSid()	254882.04	0.25488204
EncapAndCreateKey()	442223.38	0.44222338
Choosing responder and encrypting	434138.42	0.43413842
Signing and sending encrypted data	338963.96	0.33896396
DecryptData()	599135.9	0.5991359
BlindAndSign()	589763.68	0.58976368
Decapsulate()	869512.5	0.8695125
UnblindAndKDF	494829.56	0.49482956
Total time	5147670.28	5.14767028
2		0
startServer()	382057.56	0.38205756
submitNonce()	823284.6	0.8232846
checkSid()	258286.34	0.25828634
EncapAndCreateKey()	456862.76	0.45686276
Choosing responder and encrypting	425683.72	0.42568372
Signing and sending encrypted data	343375.08	0.34337508
DecryptData()	584559.08	0.58455908
BlindAndSign()	601419.92	0.60141992
Decapsulate()	824030.5	0.8240305
UnblindAndKDF	507457.14	0.50745714
Total time	5207016.7	5.2070167
4		0
startServer()	371622.08	0.37162208
submitNonce()	918655.96	0.91865596
checkSid()	287353.84	0.28735384
EncapAndCreateKey()	502230.28	0.50223028
Choosing responder and encrypting	452714.28	0.45271428
Signing and sending encrypted data	372295.74	0.37229574
DecryptData()	628944.76	0.62894476
BlindAndSign()	643225.7	0.6432257
Decapsulate()	894273.5	0.8942735
UnblindAndKDF	551193.78	0.55119378
Total time	5622509.92	5.62250992
8		0
startServer()	347745	0.347745
submitNonce()	813723.62	0.81372362
checkSid()	353423.1	0.3534231
EncapAndCreateKey()	462119.04	0.46211904
Choosing responder and encrypting	429176.68	0.42917668
Signing and sending encrypted data	345667.62	0.34566762
DecryptData()	585155.48	0.58515548
BlindAndSign()	627295.06	0.62729506
Decapsulate()	858110.52	0.85811052
UnblindAndKDF	512879.6	0.5128796
Total time	5335295.72	5.33529572
16		0
startServer()	383822.18	0.38382218
submitNonce()	907216.14	0.90721614
checkSid()	297874.92	0.29787492
EncapAndCreateKey()	527188.38	0.52718838
Choosing responder and encrypting	478809.42	0.47880942
Signing and sending encrypted data	386175.4	0.3861754
DecryptData()	643985.3	0.6439853
BlindAndSign()	668824.34	0.66882434
Decapsulate()	931128.56	0.93112856
UnblindAndKDF	566971.72	0.56697172
Total time	5791996.36	5.79199636

Table A.19: OAGK functions 32-512 responders

Function	Time usage(nanoseconds)	Standard deviation
32		0
startServer()	329872.86	0.32987286
submitNonce()	785675.3	0.7856753
checkSid()	262844.62	0.26284462
EncapAndCreateKey()	468597.14	0.46859714
Choosing responder and encrypting	409713.58	0.40971358
Signing and sending encrypted data	337204.68	0.33720468
DecryptData()	552377.56	0.55237756
BlindAndSign()	576469.94	0.57646994
Decapsulate()	782365.1	0.7823651
UnblindAndKDF	477751.82	0.47775182
Total time	4982872.6	4.9828726
64		0
startServer()	342676.64	0.34267664
submitNonce()	765217.1	0.7652171
checkSid()	249839.24	0.24983924
EncapAndCreateKey()	426713.22	0.42671322
Choosing responder and encrypting	415850.28	0.41585028
Signing and sending encrypted data	334811	0.334811
DecryptData()	543530.22	0.54353022
BlindAndSign()	560259.4	0.5602594
Decapsulate()	776886.96	0.77688696
UnblindAndKDF	470724.42	0.47072442
Total time	4886508.48	4.88650848
128		0
startServer()	341256.64	0.34125664
submitNonce()	848571.16	0.84857116
checkSid()	260587.86	0.26058786
EncapAndCreateKey()	434148.08	0.43414808
Choosing responder and encrypting	450331.52	0.45033152
Signing and sending encrypted data	345020.72	0.34502072
DecryptData()	561316.7	0.5613167
BlindAndSign()	575115.5	0.5751155
Decapsulate()	795897.66	0.79589766
UnblindAndKDF	482894.28	0.48289428
Total time	5095140.12	5.09514012
256		0
startServer()	348033.86	0.34803386
submitNonce()	812597.12	0.81259712
checkSid()	266957	0.266957
EncapAndCreateKey()	429667.58	0.42966758
Choosing responder and encrypting	508120.24	0.50812024
Signing and sending encrypted data	378606.9	0.3786069
DecryptData()	598430.64	0.59843064
BlindAndSign()	561963.84	0.56196384
Decapsulate()	778744.68	0.77874468
UnblindAndKDF	471987.94	0.47198794
Total time	5155109.8	5.1551098
512		0
startServer()	379457.9	0.3794579
submitNonce()	1045329.4	1.0453294
checkSid()	303843.48	0.30384348
EncapAndCreateKey()	447671.68	0.44767168
Choosing responder and encrypting	627934.88	0.62793488
Signing and sending encrypted data	444779.96	0.44477996
DecryptData()	664865.62	0.66486562
BlindAndSign()	593633.12	0.59363312
Decapsulate()	810133.4	0.8101334
UnblindAndKDF	481464.96	0.48146496
Total time	5799114.4	5.7991144

Table A.20: OAGK functions 1024-16384 responders

Function	Time usage(nanoseconds)	Standard deviation
1024		0
startServer()	436125.98	0.43612598
submitNonce()	924320.5	0.9243205
checkSid()	344816.24	0.34481624
EncapAndCreateKey()	438535.68	0.43853568
Choosing responder and encrypting	834023.08	0.83402308
Signing and sending encrypted data	537311.3	0.5373113
DecryptData()	768759.88	0.76875988
BlindAndSign()	598614.3	0.5986143
Decapsulate()	801653.3	0.8016533
UnblindAndKDF	478284.88	0.47828488
Total time	6162445.14	6.16244514
2048		0
startServer()	587265.04	0.58726504
submitNonce()	1107269.4	1.1072694
checkSid()	464592.22	0.46459222
EncapAndCreateKey()	467963.2	0.4679632
Choosing responder and encrypting	1369837.46	1.36983746
Signing and sending encrypted data	781316.7	0.7813167
DecryptData()	1023270.8	1.0232708
BlindAndSign()	598448.7	0.5984487
Decapsulate()	812282.66	0.81228266
UnblindAndKDF	569773.04	0.56977304
Total time	7782019.22	7.78201922
4096		0
startServer()	1078482.94	1.07848294
submitNonce()	1577874.48	1.57787448
checkSid()	679257.44	0.67925744
EncapAndCreateKey()	528955.28	0.52895528
Choosing responder and encrypting	2404340.44	2.40434044
Signing and sending encrypted data	1309613.18	1.30961318
DecryptData()	1566982.1	1.5669821
BlindAndSign()	643958.94	0.64395894
Decapsulate()	887183.16	0.88718316
UnblindAndKDF	542896.3	0.5428963
Total time	11219544.26	11.21954426
8192		0
startServer()	1848104.88	1.84810488
submitNonce()	2133045.48	2.13304548
checkSid()	1035006.28	1.03500628
EncapAndCreateKey()	435346.12	0.43534612
Choosing responder and encrypting	4065765.74	4.06576574
Signing and sending encrypted data	2268449.96	2.26844996
DecryptData()	2994027.82	2.99402782
BlindAndSign()	664064.7	0.6640647
Decapsulate()	830350.4	0.8303504
UnblindAndKDF	496952.12	0.49695212
Total time	16771113.5	16.7711135
16384		0
startServer()	3468810.04	3.46881004
submitNonce()	4892581.02	4.89258102
checkSid()	2757580.18	2.75758018
EncapAndCreateKey()	471363.4	0.4713634
Choosing responder and encrypting	8657824.62	8.65782462
Signing and sending encrypted data	3819861.94	3.81986194
DecryptData()	4273279.7	4.2732797
BlindAndSign()	592920.08	0.59292008
Decapsulate()	786954.9	0.7869549
UnblindAndKDF	477301.64	0.47730164
Total time	30198477.52	30.19847752

Table A.21: OAGK functions 32768-65536 responders

Function	Time usage(nanoseconds)	Standard deviation
32768		0
startServer()	7284802.98	7.28480298
submitNonce()	1.47E+07	14.73507482
checkSid()	9972569.06	9.97256906
EncapAndCreateKey()	630312.44	0.63031244
Choosing responder and encrypting	2.22E+07	22.23836946
Signing and sending encrypted data	7980636.62	7.98063662
DecryptData()	8147587.12	8.14758712
BlindAndSign()	596529.16	0.59652916
Decapsulate()	804429.92	0.80442992
UnblindAndKDF	482926.3	0.4829263
Total time	72873237.88	72.87323788
65536		0
startServer()	1.51E+07	15.11364838
submitNonce()	3.26E+07	32.56906474
checkSid()	2.13E+07	21.34136728
EncapAndCreateKey()	649513.52	0.64951352
Choosing responder and encrypting	5.01E+07	50.09371114
Signing and sending encrypted data	1.52E+07	15.15999656
DecryptData()	1.62E+07	16.24704886
BlindAndSign()	652090.24	0.65209024
Decapsulate()	847448.82	0.84744882
UnblindAndKDF	512563.06	0.51256306
Total time	153186452.6	153.1864526

Speedtest TGDH

Table A.22: OAGK functions 1-512 responders

Function	Time usage(nanoseconds)	Standard deviation
1		
InitilalizeTree()	2556.68	902.583900587641
ShareKey loop	417.58	550.475652140946
joinShareTree()	4547611.22	64917.255143541
Update node position	565.42	166.912922207958
2		
InitilalizeTree()	3378.74	2226.45324954287
ShareKey loop	321.78	434.610505625439
joinShareTree()	8193682.02	3713610.08123781
Update node position	434.33	289.495217749793
4		
InitilalizeTree()	3452.24	5893.22013693702
ShareKey loop	178.15	216.270079992587
joinShareTree()	4152370.06	4657915.15851107
Update node position	259.24	181.349641300996
8		
InitilalizeTree()	2555.22	1634.94366006906
ShareKey loop	214.26	784.530558614513
joinShareTree()	2144928.75	3876939.62676568
Update node position	231.76	291.201317648118
16		
InitilalizeTree()	2994.28	1208.69713394216
ShareKey loop	153.26125	211.717129676456
joinShareTree()	1099182.15	2916565.43478134
Update node position	260.29625	1746.92810341065
32		
InitilalizeTree()	2896.9	1354.20542385563
ShareKey loop	142.3875	221.327093560075
joinShareTree()	567898.89625	2148828.06101205
Update node position	159.825625	217.629091502858
64		
InitilalizeTree()	2645.58	1338.29567868988
ShareKey loop	136.755625	229.993846995653
joinShareTree()	290528.7028125	1533986.06170548
Update node position	147.2084375	232.814466080415
128		
InitilalizeTree()	2599.56	2609.60794112832
ShareKey loop	86.6084375	66.4044576162531
joinShareTree()	148900.68421875	1090919.68578231
Update node position	96.00375	79.8210204046359
256		
InitilalizeTree()	2095.22	1199.12566964435
ShareKey loop	93.9765625	65.4147064939039
joinShareTree()	79160.329296875	786920.529448709
Update node position	102.378984375	52.8148982789739
512		
InitilalizeTree()	2365.12	970.70351065606
ShareKey loop	95.6775390625	43.6295194135381
joinShareTree()	44428.190625	574122.944427136
Update node position	104.5111328125	110.230740031705

Table A.23: OAGK functions 1024-65536 responders

1024		
InitilalizeTree()	4766.5	1812.96708464329
ShareKey loop	95.8698046875	132.093732413515
joinShareTree()	30739.4204101562	442897.655957456
Update node position	108.9100390625	77.7511474543481
Function	Time usage(nanoseconds)	Standard deviation
2048		
InitilalizeTree()	3509.92	1893.64635388977
ShareKey loop	98.363662109375	96.3842768003189
joinShareTree()	35526.6207617187	447010.019581803
Update node position	126.89123046875	139.744424521368
4096		
InitilalizeTree()	2518.92	232.135205429939
ShareKey loop	99.19162109375	148.777506352866
joinShareTree()	59167.4192675781	669521.392393376
Update node position	148.841982421875	137.243569091633
8192		
InitilalizeTree()	3313.46	356.631305972989
ShareKey loop	105.370910644531	132.790829284963
joinShareTree()	127167.259782715	1959778.99287343
Update node position	185.393515625	181.937247600438
16384		
InitilalizeTree()	3720.32	498.876956373012
ShareKey loop	113.51001953125	153.208938488385
joinShareTree()	176414.97583252	3833344.95257454
Update node position	223.745520019531	5328.56973103857
32768		
InitilalizeTree()	3653.96	338.485979621018
ShareKey loop	170.016391601563	492.742758943839
joinShareTree()	475930.771730957	1.13E+07
Update node position	364.678930053711	317.417104110091
65536		
InitilalizeTree()	3596.82	299.059973249514
ShareKey loop	247.429142456055	2032.68749603162
joinShareTree()	2042845.76676971	8.60E+07
Update node position	758.975707702637	521.11678710556

Setup time

Table A.24: PKE Setup 1-8192 responders

Responders	Time	Standard Deviation
1	814329.06	169795.759407756
2	745795.82	213675.728415063
4	1019254.94	279657.506755275
8	1584193.3	415492.023552667
16	2771130.56	781163.36062289
32	4888243.5	1005094.97544629
64	9758338.74	2072435.49909071
128	1.55E+07	1481905.48254428
256	3.28E+07	5060212.66343553
512	6.75E+07	9936635.77599196
1024	1.38E+08	2.19E+07
2048	2.19E+08	3918940.45488366
4096	4.39E+08	1.01E+07
8192	8.99E+08	1.89E+07

Table A.25: OAGK Setup 1-8192 responders

Responders	Time	Standard Deviation
1	581097.52	65752.1353977922
2	638403.32	104149.59477222
4	767192.16	15442.0032629967
8	1197685.96	63506.4515125699
16	2098999.64	200053.868227011
32	3807615.46	302280.409660117
64	7253270.04	90363.14713465
128	1.41E+07	566862.591784196
256	2.81E+07	1240307.87120466
512	5.40E+07	367913.013029202
1024	1.07E+08	2.43E+05
2048	2.16E+08	964836.992961696
4096	4.36E+08	7.88E+06
8192	8.98E+08	1.97E+07

Table A.26: TGDH Setup 1-8192 responders

Responders	Time	Standard Deviation
1	1392805.22	392178.391689307
2	1494086.1	18365.0788533564
4	2470063.52	37715.6132572387
8	4471541.2	106720.917498118
16	8267029.84	106321.519000692
32	1.59E+07	279024.748032363
64	3.16E+07	369273.678581471
128	6.15E+07	229280.063588529
256	1.23E+08	1618542.45563652
512	2.45E+08	3892925.82178744
1024	4.88E+08	1.06E+06
2048	9.78E+08	3154293.2501629
4096	1.96E+09	8.28E+06
8192	3.93E+09	2.56E+07

Eivind Nordal Gran

Secure data sharing in the cloud

NTNU
Norwegian University of
Science and Technology