Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master of Science in Computer Science	Spring semester, 2015 Open access
Writer: Morten Stangeland Salte	Morten S. Salte (Morten Stangeland Salte)
Faculty supervisors: Chunming Rong Chunlei Li	
Thesis title: **Secure Sharing System with Proxy Re-Encryption**	
Credits (ECTS): 30	
Keywords: File sharing, BitTorrent, security, bilinear maps, cryptography, proxy re-encryption	Pages: 91 Stavanger, June 12th 2015

Front page for master thesis
Faculty of Science and Technology
Decision made by the Dean October 30th 2009

Secure Sharing System with Proxy Re-Encryption

Morten Stangeland Salte

June 12, 2015

MASTER'S THESIS

Department of Electrical Engineering and Computer Science

University of Stavanger

Supervisor 1: Professor Chunming Rong

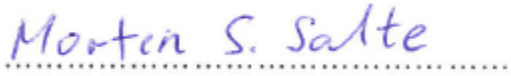
Supervisor 2: Postdoctoral Researcher Chunlei Li

Preface

This paper entails a master's thesis carried out at the Department of Electrical Engineering and Computer Science at the University of Stavanger. The work spread across the entire spring semester in 2015. The work introduces a new sharing system with high focus on security aspects. The sharing system is sufficient for sharing all types of files, and its implementation allows for easy incorporation into existing systems/applications.

The paper assumes that the reader has a background in computer science, but topics that are important to the system's implementation while being above average in complexity will contain a rather elaborate introduction.

Stavanger, June 12, 2015



Morten Stangeland Salte

Acknowledgments

I would like to thank Professor Chunming Rong and Postdoctoral Researcher Chunlei Li for their impressive enthusiasm and helpfulness throughout the semester.

The excellent knowledge of security topics provided by Chunming Rong was the deciding factor when debating whether to start this particular project in the fall of 2014.

With limited experience concerning mathematical proofs and complex algebraic structures initially, the help of Chunlei Li proved invaluable when trying to grasp the entirety of certain more complicated security aspects of this work.

M.S.S.

Abstract

This work introduces a new and secure mechanism for sharing files. In providing a complete implementation of a relatively recent cryptographic primitive known as proxy re-encryption, the thesis sharing system design enables file owners to store their private files in an arbitrary location while delegating access to others through the BitTorrent protocol.

A proxy re-encryption scheme involves, as the name implies, re-encryption of already encrypted content known as *ciphertexts*. To clarify, a file owner may initially encrypt a private file before uploading it to an arbitrary location on the Internet. Subsequently, the file owner may provide an untrusted third party known as the *proxy instance* with a special key. In possession of said special key, known as the *re-encryption key*, the proxy instance (the sharing system) is capable of re-encrypting the ciphertext originally intended for party A (the source) such that party B (the destination) can decrypt it. After re-encryption, Party B may decrypt the ciphertext with his or her secret decryption key, all while the sharing system (proxy) never need to access to the underlying plaintext.

The genius of the proxy re-encryption primitive entails that while a proxy instance is capable of delegating access to files that the respective file owner has permitted, a faulty or compromised proxy instance is unable to perform any hazardous actions without the file owner's say so. This is because the proxy instance has no means to access the underlying files. The proxy instance only has access to the original ciphertext and the re-encryption keys, both of which the file owner generates and may be publicly available without security concern. The thesis sharing system design inherits these very attractive properties.

As mentioned, this work also relies on the BitTorrent protocol. This protocol is responsible for providing the sharing system's file transfer capability. More specifically, when users delegate access to their files, they actually delegate access to encrypted metadata files known as *torrent* files.

To clarify, if Bob wants to share a picture with Alice, he does so by letting the sharing system generate and encrypt a torrent file corresponding to the picture. Moreover, Bob lets the system generate a re-encryption key that enables re-encryption of ciphertexts intended for Bob for decryption by Alice's secret decryption key. Bob then provides the system with the encrypted

torrent and the re-encryption key. When Alice makes an inquiry about the picture, the system will re-encrypt the picture's encrypted torrent file so that Alice may decrypt it. Once Alice has decrypted the torrent file, she can download the picture through a BitTorrent client.

For this to work in a secure manner, the thesis sharing system design makes some modifications to the BitTorrent protocol. More specifically, it provides a customized embedded BitTorrent tracker, which is responsible for coordinating file transfer between its users. To ensure secure file transfer, the BitTorrent tracker implements some cryptographic protocols while simultaneously enforcing its clients to do so as well.

Finally, to ensure easy incorporation of the sharing system into existing applications, all user interaction is available through a standardized web service interface.

Contents

Chapter 1

Introduction

This chapter contains sections introducing the thesis work in a structured manner. First, a review of the thesis background will take place. This includes a description of the problem that this work tries to solve, as well as a mention of why it is necessary to make such an attempt. Subsequently, the chapter presents an overview of the thesis objectives, its primary contribution, its limitations and its approach. Finally, the chapter rounds off with a paper roadmap, including what to expect from its remaining chapters.

1.1 Background

This work tries to solve the problem of sharing files in a secure and convenient manner over the Internet. In recent years, an increasing number of user's files move from being stored locally on personal computers to reside in large data centers accessible in the cloud. While this is very convenient and saves users from having to invest in large hard drives and other expensive hardware, it introduces the problem that the owner of a file is not in complete control of the file anymore. In fact, in many cases, the files are actually located on foreign servers where there are legal prejudices severely differentiating from those enforced by the file owner's own domestic laws and regulations.

When researching the practices of today's most popular cloud storage providers (CSPs), it quickly became clear that many of them are severely lacking in security areas [11]. Firstly, most providers does not offer any form of client side encryption. This means that if there is encryption

present at all, this happens on the server side as the files arrive. This is concerning as it means that the providers themselves are the key holders and thereby capable of decryption. In fact, several providers openly admit to these capabilities, but they all promise that they would never decrypt any data without a good reason. Moreover, it is a common claim that only a handful of their engineers have the necessary credentials to access the decryption functionality. However, from a user's point of view, these promises might not be satisfactory. They are probably not, especially in light of the recent exposure of private content covered by the media. Moreover, who decides what a good reason for allowing decryption of files is? It is a known fact that laws and regulations of many countries lets government officials inquire about the content of files serviced by domestic CSPs.

When users of CSPs that only provide server side encryption systems wants to share files with other users, security is often more or less abandoned. Amazingly enough, upon sharing, the CSPs will in some cases simply decrypt the files permanently and let the recipient user(s) access them as well.

While it is true that some CSPs do offer client side encryption, this approach also contain weaknesses once users try to share their files with others [16]. Client side encryption means that the data encryption occurs prior to upload. When users share files in client side encrypted cloud environments, the CSPs still manage to decrypt the respective files before encrypting them with the destination's public key. This means that at some point during this process, the CSPs handle the files as plaintext.

These problems alone solely keep the discussion open about whether these systems are good enough. Moreover, they sparked the idea of this thesis work, which tries to offer a new system for sharing files between users without providing yet another CSP.

The thesis work provides a sharing system whose encryption algorithm is a particular proxy re-encryption scheme. In a proxy re-encryption scheme, an untrusted third party referred to as the *proxy instance* is able to convert ciphertexts encrypted for party A into ciphertexts that party B can decrypt. This conversion works without the necessity to access the plaintext. By using such a scheme, this work's (untrusted) sharing system is able to provide its users with a secure sharing system while remaining ignorant as to the contents of its users' files. There is no way for the sharing system, which acts as the proxy instance, to access the file content. All the infor-

mation at the disposal of the sharing system may be publicly available to anyone. The sharing system acts as an application that provides its users with a proxy re-encryption implementation. Additionally, the sharing system manages its user's publicly available cryptographic keys in such a way that it knows which of its users have access to what content.

The intention of the sharing system is for its incorporation into existing systems to be seamless, without introducing yet another registration schema for the users.

The following literature encompasses the main portion of the survey for this work:

- *Secure Cloud Storage Sharing Through Proxy Re-Encryption* [11]
- *Divertible Protocols and Atomic Proxy Cryptography* [3]
- *Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage* [7]
- *"To Share or not to Share" in Client-Side Encrypted Clouds* [16]
- *On the Implementation of Pairing-based Cryptography* [9]
- *Incentives Build Robustness in BitTorrent* [6]

1.2 Contribution

The following will clarify what the thesis work's primary contribution is. Firstly, the following shows the primary objectives of the thesis work in no particular order. Design and develop a file sharing system that:

1. Satisfies today's security requirements
2. Handles arbitrary file types and sizes
3. Enables file owners to host their files on arbitrary locations
4. Does not require file owners to remain online indefinitely
5. Can easily be incorporated into existing applications
6. Works on all platforms

From the above, it is apparent that the core contribution of the thesis work is a file sharing system *design*. Additionally, as we shall see, a huge part of the sharing system's strengths relates to its underlying cryptographic emphasis on a proxy re-encryption scheme. The thesis sharing system implements the AFGH [7] algorithm to its fullest, which is one of many proxy re-encryption schemes. This particular scheme fits very well for a sharing system and is a scheme whose security is not compromised by obvious weaknesses.

When talking about limitations of the thesis work, it makes sense to pinpoint the fact that its primary focus is on providing a secure *sharing* system. The security regarding the underlying storage of the files is outside its scope. In other words, while the sharing system is able to provide satisfactory security during the sharing of files, the security of the files when they reside on the user's hard drives will remain unchanged. This becomes a problem if the users are hosting their files on certain cloud storage providers' (CSP) servers.

While this may seem like a rather huge limitation, it is easy to see that the sharing system is able to provide exceptional security improvements if compared to what goes on with the files when they process through many of the popular CSPs own sharing systems and practices.

1.3 Approach

The following elaborates briefly on the approach used to meet the requirements of the thesis objectives. The design and implementation chapter (Chapter 3) will contain a more complete walkthrough, including the why's and how's.

When starting development of a file sharing system that needed to be secure, it was initially required to do some research to find out which of today's popular cryptographic algorithms that are considered secure. In addition to answering the initial question about secure algorithms, this process found that proxy re-encryption would be a perfect cryptographic primitive for the thesis sharing system. This is because such a scheme would eliminate the primary deficiency of many popular sharing systems, namely the need for decryption of files that others should be able to access. In proxy re-encryption schemes, re-encryption can occur without the need for decryption. However, a proxy re-encryption scheme would only be satisfactory if using the correct algorithm. In other words, the respective algorithm implementation would need to satisfy

specific properties. It turns out that some of the first proxy re-encryption algorithms contain some rather problematic issues. A more thorough elaboration on this topic follows later in the paper.

In addition to being cryptographically secure, the sharing system's file transfer process itself would need to be efficient while still working for arbitrary file types and sizes. The BitTorrent protocol ended up satisfying these initial requirements very well. However, the security of BitTorrent was questionable. This led to some additional research into how to go about securing the BitTorrent protocol and what performance disadvantages that would potentially cause. In the end, enabling encryption capabilities in the BitTorrent protocol did not impediment the performance too badly, especially for this particular application area. The additional overhead was negligible.

While the BitTorrent protocol solved the primary objective of supporting arbitrary file types and sizes, it also fit well with meeting other objectives. More specifically, with utilizing the BitTorrent protocol, file owners are capable of hosting their files on arbitrary locations while simultaneously not being required to remain online for the complete sharing process. By utilizing this protocol, users can enter and leave the distribution of a file as they please, including the original source. Moreover, if a file is shared with multiple recipients, it is sufficient that at least one user in possession of the complete file is available at any given time. A more complete description of the BitTorrent protocol follows in Chapter 2.

Finally, since the intention for the sharing system was for easy incorporation into existing applications in a platform independent fashion, an examination of different exposure practices followed. The most applicable ended up being to provide a user interface through the HTTP protocol. This way it would be possible to ensure security by enabling SSL/TLS capabilities on the corresponding web server. Moreover, all platforms and programming languages have standardized methodologies for invoking HTTP requests and thus easy incorporation is a fact. Providing an application-programming interface (API) through the HTTP protocol is often synonymous with a *web service*.

1.4 Roadmap

The rest of the paper consists of the following. Chapter 2 introduces the reader to the scientific background of the thesis work. Chapter 3 goes into detail on its design and implementation phase. Chapter 4 goes through the various experimental results, including example use cases and the system's corresponding output. Chapter 5 contains the paper's discussion while Chapter 6 provides some concluding remarks.

Chapter 2

Scientific Background

As this work builds upon a handful of academic areas, this chapter will introduce them in turn. Areas that are of higher relevance to this work receives a more thorough elaboration than other areas. Each of the areas' domains consists of a wide variety of terminology, many of which have a high frequency of use throughout the paper. To allow for this, this chapter seeks to bring the reader to some extent of understanding for the most important topics, if he or she is not already there.

2.1 An Overview of Cryptographic Algorithms and Protocols

When talking about information security and cryptography, it is common to divide the field into four main categories [12]. These categories are *symmetric encryption, asymmetric encryption, data integrity algorithms* and *authentication protocols*.

Symmetric and asymmetric encryption schemes are similar in that they both provide a mechanism for concealing the contents of a message so that unwanted parties cannot access it. However, they are rather different in their respective underlying implementations.

2.1.1 Symmetric/Secret Key Encryption

Symmetric encryption schemes lets two parties communicate secretly by being possession of the same secret key that only they know. Both parties will use the same secret key to encrypt and decrypt the messages exchanged between them. This process is very challenging to initiate,

as the two parties must be able to exchange the secret key between them, without a third party learning about it. However, after the key exchange, symmetric encryption schemes are very effective.

2.1.2 Asymmetric/Public Key Encryption

Because public key cryptosystems lays the fundament for the cryptographic primitive known as proxy re-encryption, which is highly used in this work, the paper will pay more attention to detail and elaborate more thoroughly when discussing this topic compared to the other cryptographic algorithms and protocols.

Asymmetric encryption schemes, synonymous with *public key encryption schemes*, are able to solve the key exchange problem of symmetric encryption. Specifically, in asymmetric encryption, both parties possess their own unique key pair. The key pair consists of one secret key that only they know and a public key that they can publish for anyone to see. A very important property of this key pair is that the public key generation function takes the corresponding secret key as its input parameter. In other words, there is a mathematical relationship between a party's public key and its corresponding secret key. However, if the scheme is secure, there is no practical way to reverse engineer the secret key by only knowing the public key.

When the sender wants to encrypt a message, it does so by using the recipient's public key. Because the recipient's public key evolves from his or her secret key, the recipient is the only party capable of decryption. Since asymmetric encryption solves the key exchange problem, it is very attractive. However, because its security typically bases on computations on exceptionally large prime numbers, over 300 digits long, its performance suffers compared to symmetric encryption. Because of this, it is not uncommon to use an asymmetric scheme to exchange keys between two parties, before subsequently switching to a symmetric scheme for the following communications.

Below is an overview of what most public key cryptosystems relies on for security. As we shall see, the computationally hard problems making public key cryptosystems secure are either *RSA-based* or based on the *discrete logarithm problem* (DLP). Moreover, DLP-based cryptosystems are either *ElGamal-based* or based on the *elliptic curve* algebraic structure.

- RSA-based
- DLP-based
  - ElGamal-based
  - Elliptic curve-based

To understand these terms, we need to look closer at the history of public key cryptography.

The RSA algorithm and the Integer Factorization Problem

The term *RSA-based* origins from the well-known RSA public key cryptosystem developed by Rivest, Shamir and Adleman in 1978. RSA has since then been known as the most widely accepted public key cryptosystem [12]. The hardness of RSA relies on the tediousness of factoring large integers, more specifically the product of two exceptionally large primes. To explain this further, the following describes the RSA algorithm.

Let p and q be two exceptionally large prime numbers. Continue by computing $n = pq$. Compute $\phi(n) = (p-1)(q-1)$. Select e such that e is relatively prime to and less than $\phi(n)$. Moreover, select d such that $d \equiv e^{-1} \bmod \phi(n)$. Both parties must know the value of n. The sender knows the value of e while the recipient knows the value of d. In other words, the public key is represented as $PK = (e, n)$ while the secret key is represented as $SK = (d, n)$.

To encrypt a message using the RSA scheme, given the plaintext $M < n$, the ciphertext becomes $C = M^e \bmod n$. Similarly, in decryption of the ciphertext C, the recovered message becomes $M = C^d \bmod n$.

As we can see, the security of the RSA cryptosystem relies on the difficulty of recovering the two prime numbers q and p by only knowing the publicly available value n. This is a very hard problem. However, with computing power growing exceptionally in recent years, the length of n in bits, known as the *key size*, must be large (preferably several thousand bits). By increasing the key size to increase the security, we must do so at the cost of computational efficiency (introducing more overhead).

Diffie-Hellman, ElGamal and the Discrete Logarithm Problem

The *discrete logarithm problem* refers to the difficulty of computing x given only g and g^x. As mentioned, it is possible to divide the discrete logarithm problem into two categories, namely ElGamal-based and elliptic curve-based. The term ElGamal-based origins from the ElGamal public key cryptosystem, which bases on the Diffie-Hellman key exchange protocol.

The Diffie-Hellman key exchange protocol is a brilliant technique enabling two parties to agree secretly on an integer value. Its introduction in 1976 by Diffie and Hellman solved one of the major problems with symmetric/secret key cryptosystems, namely how to transfer the shared secret key between the communicating parties in a secure fashion. The protocol works by letting two parties A and B start by publicly sharing a prime number q and an integer p such that $p < q$ and p being a primitive root of q. Followed by this, they each select a secret value $S_A < q$ and $S_B < q$ respectively. Moreover, they publicly send the values $P_A = p^{S_A} \bmod q$ and $P_B = p^{S_B} \bmod q$ to the other party. Once this is done, each party can compute the shared value $K_A = P_B^{S_A} \bmod q = K_B = P_A^{S_B} \bmod q$. The key exchange is complete. As we can see, the security of the Diffie-Hellman key exchange protocol relates to the difficulty of computing the secret value S_A or S_B given only P_A or P_B together with p. As mentioned before, this problem is the *discrete logarithm problem*.

Additionally, there are two more hard problems related to the Diffie-Hellman key exchange, namely the *computational* and *decisional* Diffie-Hellman problems. The computational Diffie-Hellman problem is that given g, g^x, g^y, compute g^{xy}. The decisional Diffie-Hellman problem is that given g, g^x, g^y, g^z, determine if $xy = z$. However, no matter how hard these two additional problems are, the discrete logarithm problem is more important. If you can solve the discrete logarithm problem, you can solve the computational and the decisional Diffie-Hellman problems as well [9].

The ElGamal cryptosystem arrived in 1984 when Taher Elgamal announced it as a public key scheme basing on the discrete logarithm problem. Similar to the Diffie-Hellman key exchange protocol, the global elements of the ElGamal scheme are a prime number q and an integer $a < q$ and a being a primitive root of q. From this, key generation works by selecting a random integer $SK < q - 1$ as the secret key. Moreover, the public key is computed as $PK = a^{SK} \bmod q$ and published together with q and a as the tuple (q, a, PK).

Encrypting a message $M < q$ works by first selecting a random integer $k < q$ and computing $K = PK^k \bmod q$. The ciphertext can then be encrypted as a tuple $C = (C_1, C_2)$ where $C_1 = a^k \bmod q$ and $C_2 = KM \bmod q$. Note that k must be randomly generated for each encryption; it should never be re-used for subsequent encryptions. To recover the plaintext, the recipient computes $K = C_1^{SK} \bmod q$ before recovering the message $M = (C_2 K^{-1}) \bmod q$.

Elliptic Curve Cryptography

Elliptic curve cryptography refers to the application of the algebraic structure of elliptic curves over finite fields. The introduction of elliptic curves in cryptography is relatively recent, and is attractive because it is able to provide RSA-like security with far smaller key sizes. Smaller keys are both more practical and more computationally efficient as it reduces overhead. However, since elliptic curve theory is harder to explain and understand, it has the side effect of being significantly more difficult to cryptanalyze. Cryptanalysis refers to the careful examination of cryptographic algorithms. If this process is easy, it means that more people will be able to perform it in a satisfactory level. More people performing the cryptanalysis will increase the probability of discovering weaknesses before unwelcome parties can attack them.

The following will try to explain elliptic curve cryptography to the extent where the reader is able to understand its basic principles and purpose. Since elliptic curves is a rather complex mathematical subject, a complete review is outside the scope of this paper.

To keep it simple, an elliptic curve is a set of points that satisfy a specific mathematical equation [12, 15]. More specifically, the equation for an elliptic curve is of the form $y^2 = x^3 + ax + b$. Figure 2.1 shows the plot of $y^2 = x^3 - x + 1$.

When plotting such an equation, several properties appear that are interesting to cryptographers. One of these properties is the horizontal symmetry, which enables the reflection of any point on the curve over the x-axis to remain the same curve. Another interesting property is that any non-vertical line will intersect the curve at a maximum of three points. In other words, by taking any two points on the curve and drawing a line through them, the line will intersect at the curve at exactly one more point. When arriving at this point, it is possible to make a dotted line straight up if working above the x-axis or straight down if working below the x-axis. This way the dotted line will inevitably hit the curve's reflection on the other side of the x-axis. The procedure

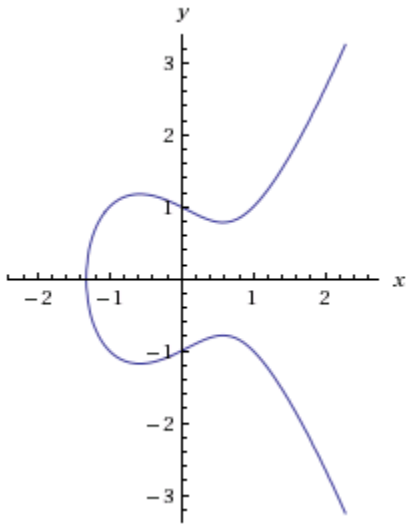


Figure 2.1: A plot of the elliptic curve $y^2 = x^3 - x + 1$

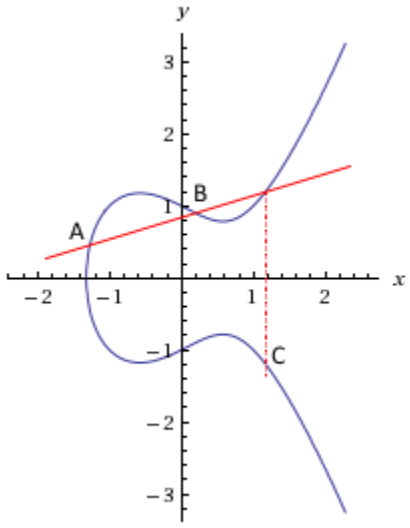


Figure 2.2: A plot of the *dotted* elliptic curve $y^2 = x^3 - x + 1$

of making dotted lines from the destination point goes by the term “dotting” and is illustrated in Figure 2.2.

It is possible to dot any two points on the curve to get a new point. It is also possible to string dots together in a sequence of dotting procedures. It turns out, if we have two points A and B, an initial point dotted with itself n times to arrive at a final point and the first point is a *hard problem*. Cryptographers love hard problems, especially the ones like this that are easy to do but hard to undo. This is the basis for the security of elliptic curve cryptography.

To summarize, an elliptic curve cryptosystem works by picking a large prime as the maximum, a curve equation and a public point on the curve. A secret key is a random number k while the public key is the public point dotted with itself k times. In elliptic curve cryptography, computing the secret key k with only knowing the public point on the curve goes by term “the elliptic curve discrete logarithm problem”.

2.1.3 Data Integrity Algorithms

When talking about data integrity algorithms, we are looking for a way to protect messages from alteration. A very common mechanism used for this purpose are hash functions. Hash functions accept messages or arbitrary length and produces a fixed sized output. Good hash functions will ensure a good *avalanche effect*, that is, changing a single bit of the input should produce an output with at least half of its bits changed. If a message source appends the hash value of the message to the message itself, it is easy for the recipient to ensure integrity of the message. Running the message through a hash function and comparing the output with the hash value appended to the original message will reveal foul play with very high probability.

2.1.4 Authentication Protocols

Authentication protocols exist to verify the identity of entities. Message Authentication Protocols (MAC) lets recipient entities verify that the source of the message, i.e. the sender, is authentic. User authentication protocols however, is the building block of the vast majority of access control protocols, and entails the process of providing credentials to a system that implies authenticity. Typical examples include passwords, PIN numbers, fingerprints, voice etc.

2.2 SSL/TLS and HTTPS

In the early years of the Web, people used protocols such as Telnet and the File Transfer Protocol (FTP) to access information. Moreover, the combination of the Hypertext Transfer Protocol (HTTP) and HTML provided users with a graphical user interface. These protocols often needed to transfer sensitive data such as usernames and passwords to the underlying web servers. As long as these communications was running over HTTP, there was in practice no security present.

The HTTPS protocol, or HTTP Secure, is a protocol that combines the HTTP protocol with the SSL/TLS protocol to provide a secure communication over HTTP. The following will elaborate more thoroughly on what SSL and TLS is. However, the explanations is still be kept rather simplified.

2.2.1 Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

The SSL protocol, or the Secure Sockets Layer protocol, is actually two layers of protocols designed to use the Transmission Control Protocol (TCP) to provide a secure and reliable end-to-end communication. Netscape first developed SSL version 1.0 in 1994. SSL is one of the most widely deployed security protocols on the Internet [17]. The protocol is able to provide authentication, integrity and confidentiality for two communicating parties.

TLS is a standardization initiative with the goal of providing a standard implementation of SSL. Because of this, SSL and TLS are often synonymous. TLS version 1.0 took over after SSL version 3.0 in 1999. As of 2015, TLS 1.2 is the established version, with version 1.3 on the horizon [14].

The most important protocol in the layers provided by SSL/TLS is the Handshake Protocol. This protocol allows the server and client to authenticate each other and to agree on an encryption algorithm and a Message Authentication Code (MAC)-algorithm [12].

Before any data transmission happens between a client and a server, SSL/TLS initializes the Handshake Protocol, which is comprised of the following four phases.

Establish Security Capabilities Initiated by the client establishing security capabilities including protocol version, cipher suite (set of supported cryptographic algorithms), compression method and initial random numbers.

Server Authentication and Key Exchange Server sends one or more X.509 certificates for authentication, performs the key exchange and requests client certificate.

Client Authentication and Key Exchange Client sends a certificate reply and participates in the key exchange.

Finish Finalizes the secure connection setup.

It is worth noting that the key exchange process of SSL/TLS supports either RSA or a variation of the Diffie-Hellman key exchange algorithm.

After the Handshake Protocol, SSL/TLS initializes the Record Protocol. The Record Protocol is responsible for securing the application data with the keys established in the Handshake Protocol. The supported encryption algorithms are symmetric in respect to the established keys. They can be both stream ciphers or block ciphers. Stream ciphers are algorithms that encrypt a digital data stream one bit or one byte at a time. On the other hand, block ciphers encrypt blocks of plaintext producing a ciphertext block of equal length. The following shows an overview of cryptographic algorithms supported by TLS. It is worth noting that as of February 2015, the Internet Engineering Task Force [30] (IETF) published a so-called Request for Comment (RFC 7465) where they state that they have reached consensus in the decision of prohibiting use of the RC4 stream cipher in TLS [29].

Block Ciphers DES, 3DES, DES40, IDEA, RC2

Stream Ciphers RC4

2.3 Proxy Re-Encryption

Proxy re-encryption is a relatively new cryptographic primitive. While regular symmetric or asymmetric encryption schemes are able to provide two parties with a secure communication channel, proxy re-encryption takes the idea a step further. Specifically, proxy re-encryption is a technique for letting a proxy instance re-encrypt an already encrypted message (i.e. ciphertext), so that another secret key can decrypt it. This may sound far-fetched and insecure, but do not worry. The re-encryption process will not work without the original sender's say so, as

the sender itself generates the re-encryption key by combining its own secret key and the recipient's public key. Because of these properties, it is common to categorize proxy re-encryption cryptosystems as extended cases of a public key cryptosystem.

To explain the significance of a proxy re-encryption scheme, the following example is often used [7, 5, 3]. Suppose that Alice and Bob works for the same company and that Bob is Alice's subordinate. Many people send e-mails to Alice containing important company information. When they do this, they of course encrypt the e-mails with Alice's public encryption key. This will ensure that only Alice can read the messages intended for her. However, if Alice is out sick or on vacation, messages for Alice will pile up in her inbox without anyone able to access the important information contained in them. To avoid this, Alice may grant Bob temporary access to read her e-mails while she is unavailable. To achieve this without Alice having to reveal her secret key, a proxy re-encryption scheme fits like hand in glove. All Alice has to do is to generate a re-encryption key by using her own secret key and Bob's public key. By providing Bob (or the e-mail server) with the re-encryption key, Bob (or the e-mail server) is capable of re-encrypting the encrypted messages intended for Alice so that Bob can decrypt them with his own secret key.

In theory, there are other workarounds for the above e-mail scenario as well. By having Alice supply the e-mail server with her secret key, the server will be able to decrypt her incoming e-mails before encrypting them with Bob's public key. This is a huge security concern though. In this approach, there must be complete trust between Alice and the e-mail server. Alice must be able to trust the e-mail server to keep her secret key secret. Relying on trusted parties is never a good idea in cryptography, so abandoning this approach for its naivety is eminent. Another possibility would be for Alice to remain online at all times, where she would manually decrypt her incoming e-mails with her secret decryption key, before encrypting them with Bob's public key. Subsequently, Alice would forward the e-mails to Bob. The problem with this approach is its lacking practicality.

The concept of proxy re-encryption origins from 1998 when Blaze, Bleumer and Strauss (BBS) published their paper titled Divertible Protocols and Atomic Proxy Cryptography [3]. Their scheme bases on the ElGamal public key cryptosystem and is perfectly capable of providing the desired re-encryption function. However, their scheme contains some rather problematic weak-

nesses for practical use. More specifically, it is *bidirectional*, its delegation function is *transitive* and it is prone to *collusion* between the proxy and either of the parties [5].

To elaborate on this, the scheme's bidirectional property means that a single re-encryption key generated for re-encrypting messages intended for Alice so that Bob can decrypt them, also enables re-encrypting of messages intended for Bob so that Alice can decrypt them. In other words, the re-encryption key works both ways. In practice, this means that if Alice wants to let Bob decrypt her incoming e-mails while she is unavailable, Bob must agree to revoke some of his own privacy in the process. By granting Bob the privilege to read Alice's e-mails, in theory, Alice will be able to reverse the process and to read Bob's e-mails. In conclusion, the scheme is only viable if there is a mutual trust relationship between Alice and Bob. That is rarely the case.

The scheme's delegation function being transitive means that the proxy instance is capable of providing delegations between two parties that have never agreed on it. In other words, the proxy instance is capable of re-encrypting Alice's ciphertexts not only for Bob, but also for a third party Charlie (without Alice's consent).

Finally, the collusion issue present in the scheme enables a fraudulent proxy instance to cooperate with either Alice or Bob to learn the other party's secret key. This is alarming, as secret keys should as the name implies, always be kept secret.

Since BBS introduced their scheme, several improved proxy re-encryption schemes have surfaced. There is no perfect scheme, but there are different schemes solving different challenges. Therefore, when deciding on which scheme to use, one should do so based on the specific application's requirements.

As we have seen, the BBS scheme contains three major weaknesses. To reiterate, these weaknesses are its bidirectionality, its transitivity and its weakness against collusion. The thesis sharing system implements a proxy re-encryption scheme called the AFGH algorithm. Similar to the BBS scheme, AFGH bases on the ElGamal public key cryptosystem. However, it also bases on the algebraic setting known as *bilinear maps*. As we shall see, the AFGH scheme is able to remedy all of the weaknesses of BBS. The following section describes the AFGH scheme in detail.

2.4 Bilinear Maps and the AFGH Proxy Re-Encryption Scheme

In the previous section, a brief discussion of proxy re-encryption in general took place. While proxy re-encryption schemes clearly have their use, the aforementioned BBS scheme is not suited for many applications because of its weaknesses. In 2005, Ateniese, Fu, Green and Hohenberger (AFGH) published their improved proxy re-encryption scheme [7]. Their scheme bases on the ElGamal cryptosystem (like BBS) and *bilinear maps*. The AFGH scheme is very attractive as it remedies all the major weaknesses of BBS.

2.4.1 Pairing-Based Cryptography and Bilinear Maps

To be able to understand the AFGH algorithm, we first need to make sure we understand what bilinear maps are. The following will introduce pairing-based cryptography. A pairing, or a bilinear map, is able to provide cyclic groups with additional properties [8, 9]. To understand this and grasp the significance, we need to know what groups and cyclic groups in particular are.

Groups

In discrete mathematics, a *group* $(\mathbb{G}, \cdot)$ is a set of elements $\mathbb{G}$ together with an abstract binary operation '$\cdot$' satisfying the following conditions.

Closure For all $a, b \in \mathbb{G}$, $a \cdot b \in \mathbb{G}$

Associativity For all $a, b, c \in \mathbb{G}$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

Identity Element There exists $e \in \mathbb{G}$, such that for all $a \in \mathbb{G}$, $a \cdot e = a = e \cdot a$

Inverse Element For any $a \in \mathbb{G}$, there exists $a^{-1} \in \mathbb{G}$, such that $a \cdot a^{-1} = e = a^{-1} \cdot a$

A group's binary operation could be anything from addition, multiplication, mapping etc. The definition of groups does not specify the operation.

To give a couple of examples of groups, the following shows two groups $(\mathbb{Z}_n, +)$ and $(\mathbb{Z}_n^*, \bullet)$ for $n = 8$. Note that n is the modulus in both of these groups.

$$(\mathbb{Z}_n, +) \qquad \mathbb{Z}_8 = \{0,1,2,3,\ldots,7\} \tag{2.1}$$

$$(\mathbb{Z}_n^*, \bullet) \qquad \mathbb{Z}_8^* = \{1,3,5,7\} \tag{2.2}$$

Note that in group 2.1, the operation is the addition modulus 8 while in group 2.2 the operation is the multiplication modulus 8.

A *subgroup* is a group $(\mathbb{H}, \cdot) \subseteq (\mathbb{G}, \cdot)$ satisfying the following conditions.

Closure For all $a, b \,\epsilon\, \mathbb{H}$, $a \cdot b \,\epsilon\, \mathbb{H}$

Inverse Element For all $a \,\epsilon\, \mathbb{H}$, $a^{-1} \,\epsilon\, \mathbb{H}$

A group is *cyclic* if the following conditions apply.

- Denote g^k as $(g \cdot g \cdot g \cdot \ldots \cdot g)$ (k times, where k is an integer)
- Every element in $\mathbb{G}$ is a power g^k of a fixed element g in $\mathbb{G}$. The element g is the *generator* of $\mathbb{G}$, or $< g > = \mathbb{G}$
- Given the *order* (number of elements) of the group is $|\mathbb{G}| = m$, then $\mathbb{G} = \{e, g, g^2, g^3, \ldots, g^{m-1}\}$ where e is the identify element.

The following shows an example of a cyclic group.

$$< 2, \bullet > = \mathbb{Z}_{11}^* = \{2, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}\} \bmod 11 = \{1,2,3,4,5,6,7,8,9,10\} \tag{2.3}$$

Bilinear Maps

Let $\mathbb{G}_1$ and $\mathbb{G}_T$ be cyclic groups of prime order q. Let g be a generator of $\mathbb{G}_1$. A bilinear map e is then an efficiently computable function $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ satisfying the following.

Bilinearity For all $g_1 \,\epsilon\, \mathbb{G}_1$, $g_2 \,\epsilon\, \mathbb{G}_2$, $a, b \,\epsilon\, \mathbb{Z}_q$, then $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$

Non-degeneracy If $\mathbb{G}_1 = < g_1 >$, $\mathbb{G}_2 = < g_2 >$, then $\mathbb{G}_T = < e(g_1, g_2) >$

Computability The computation of e must be efficient

The following shows a simplified example of a bilinear map e.

For $n = 11$, $e : \mathbb{Z}_{11}^* \times \mathbb{Z}_{10}^+ \rightarrow \mathbb{Z}_{11}^*$

$(g,k) \rightarrow e(g,k) = g^k mod 11$

$e(g^a, r \cdot k) = (g^a)^{r \cdot k} = (g^k)^{a \cdot r}$

$\mathbb{G}_1 = \mathbb{Z}_{11}^* = <2, \bullet>$, $\mathbb{G}_2 = \mathbb{Z}_{10}^+ = <1, +>$

More particularly:

$\mathbb{G}_1 = <4, \bullet> = \{4,5,9,3,1\} \leq \mathbb{Z}_{11}^*$

$\mathbb{G}_2 < \mathbb{Z}_5^+ = \{0,1,2,3,4\}$

$e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$

$e(g,k) = g^k mod 11$

Note that in the above example, groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are all of prime order $q = 5$.

2.4.2 The AFGH Algorithm

Now that we have basic understanding of bilinear maps, it is time to examine the AFGH algorithm in detail. The following describes the algorithm and its components.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be cyclic groups of prime order q. Let e be the bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$. The system parameters are random generators $g \, \epsilon \, \mathbb{G}_1$ and $Z = e(g,g) \, \epsilon \, \mathbb{G}_2$.

Key Generation Alice selects her secret key as a random element $SK = a \, \epsilon \, \mathbb{Z}_q$ and computes her corresponding public key as $PK = g^a$.

Re-Encryption Key Generation Alice delegates to Bob by publishing the re-encryption key $RK_{A \rightarrow B} = g^{b/a} \, \epsilon \, \mathbb{G}_1$. Note that a refers to Alice's secret key while b refers to Bob's public key.

Encryption Level 1 To encrypt a message $m \, \epsilon \, \mathbb{G}_2$ under the public key PK_A such that it can only be decrypted by the secret key holder SK_A, compute the ciphertext tuple $C = (mZ^k, Z^{ak})$. Note that k is a randomly selected element in $\mathbb{Z}_q$ and that a in this case refers to the public key PK_A.

Encryption Level 2 To encrypt a message $m \epsilon \mathbb{G}_2$ under the public key PK_A such that it can be decrypted by A and its delegatee, compute the ciphertext tuple $C = (mZ^k, g^{ak})$. Note that k is a randomly selected element in $\mathbb{Z}_q$ and that a in this case refers to the public key PK_A.

Decryption Level 1 To decrypt a level 1 ciphertext $C = (\alpha, \beta)$ with secret key $SK = a$, compute $m = \alpha / \beta^{1/a}$.

Decryption Level 2 To decrypt a level 2 ciphertext $C = (\alpha, \beta)$ with secret key $SK = a$, compute $m = \alpha / e(\beta, g^{1/a})$.

Re-Encryption To re-encrypt a level 2 ciphertext for A into a level 1 ciphertext for B, obtain the re-encryption key $RK_{A \to B} = g^{b/a}$. Continue by changing the ciphertext tuple $C_A = (mZ^k, g^{ak})$ into $C_B = (mZ^k, Z^{bk})$ where $Z^{bk} = e(g^{ak}, g^{b/a})$. Note that k is a randomly selected element in $\mathbb{Z}_q$. Moreover, note that $g^{ak} = C_A[2]$ and that $g^{b/a} = RK_{A \to B}$.

Comparing the AFGH scheme to the BBS scheme, we see that it is *unidirectional* under the *inverse exponent assumption*. This means that the proxy cannot compute $g^{b/a}$ by knowing $g^{a/b}$. Additionally, the scheme is both *non-transitive* and *collusion resistant*, meaning that the proxy cannot extract a or b from $g^{a/b}$ with help from either party. These three properties are the opposite of those that are part of the BBS scheme. In fact, as we saw in the previous section, the BBS scheme is *bidirectional*, prone to *collusion* and its delegation function is *transitive*. To conclude, AFGH is able to remedy the three major weaknesses of BBS.

If we examine the AFGH algorithm closely, we see that the ciphertexts have two different shapes. One "regular" shape and one re-encrypted shape, or one first level ciphertext and one second level ciphertext. In a first level ciphertext $C_{Level1} = (mZ^k, Z^{ak})$, the second element is an element of group $\mathbb{G}_1$. In a second level ciphertext $C_{Level2} = (mZ^k, g^{ak})$, the second element is an element of group $\mathbb{G}_2$. Without going into too much detail, this leads to the property that the AFGH scheme only allows for a single re-encryption of ciphertexts. This conclusion makes sense in correspondence to the above description of the algorithm. It specifically states that re-encryption requires a second level ciphertext and that the output of re-encryption is a first level ciphertext.

There are both pros and cons to the singular re-encryption property. The obvious benefit is that it guarantees that Bob is not able to further delegate messages that Alice delegated to him.

However, for some applications such behavior could be desirable. In the end, the application of the algorithm will decide.

2.5 The BitTorrent Protocol

The BitTorrent protocol is a peer-to-peer protocol commonly used for sharing files over the Internet [2]. To introduce the BitTorrent protocol, this chapter will first briefly look at a more traditional way of sharing files before diving into the peer-to-peer approach. Subsequently, an examination of the most apparent benefits of such a scheme follows, particularly with respect to the BitTorrent protocol.

2.5.1 The Traditional Client/Server Architecture

In the past, file sharing almost exclusively took place by using a client/server architecture. Such an architecture traditionally entailed storing all data/files on a single centralized server machine's hard drive. When the clients wanted to access a file, they had to establish a connection directly to the server and download the file continuously over the FTP or HTTP protocol. Consequently, the server would necessarily have to consist of extraordinary hardware specs and network connectivity to handle the high upload demand of its clients.

By utilizing a client/server architecture for file sharing, some problems are prone to arise. The most obvious problem is regarding the traffic load applied to the server's hardware as well as its network, particularly when handling multiple upload sessions simultaneously. If a single server contains large files of high interest, multiple clients downloading them simultaneously may cause reduced performance and reliability on the server. In some cases, high demand of particular files can even cause the server's hardware or network interface to fail. If that happens, clients lose their respective connections to the server, and the establishment of new connections must wait until operational engineers have intervened to fix the problem. Such failures will also cause lost progress and frustration for the users. In the worst cases, hard drive malfunctions may occur because of other various hardware failures, causing sensitive files to be permanently lost.

These are huge concerns, and the traditional approach of remedying them was to invest

even more money into hardware and state of the art network capabilities. This was to increase the server's upper limits so that it would be less likely to kneel. However, this only postponed the issues.

In recent years, improvements like load balancing and failover systems have appeared, making the problem less of a concern in some application areas. These systems work in that multiple server instances run in parallel, collaborating with handling the client requests. The idea behind parallelization of server instances is primarily to achieve increased redundancy. With a redundant server environment, a load balancing system examines the incoming client requests and determines to which of the server instances it should forward them. This deterministic operation uses the respective server instance's current traffic load as a parameter. Instances with low traffic load at any given time will more likely receive new client requests than instances with high traffic load. This ensures a balanced load of traffic among the redundant server instances, significantly decreasing the probability that the servers will crash/kneel.

In the unlikely case of a server instance crashing, a failover system will be able to make sure that the clients does not suffer from this issue. This entails the forwarding of current connections as well as new client requests to another redundant server instance.

The client/server architecture is still highly in use, especially in applications in which a peer-to-peer approach is insufficient. A highly relevant example of such an application area today is streaming applications such as Netflix and Twitch.tv. This is because these applications typically require the data to arrive to its clients continuously and in a specific order. However, in application areas where the arrival of data can happen in an arbitrary order, a peer-to-peer approach will likely outperform a client/server architecture, especially on low-end commodity hardware.

2.5.2 The Peer-to-Peer Architecture and the BitTorrent Protocol

The peer-to-peer architecture differs from the traditional client/server architecture in several areas. The decentralization property of a peer-to-peer architecture is one of the primary differences compared to the centralized client/server architecture. In a peer-to-peer architecture, the files are stored on the computers of the respective participants (peers), and file sharing takes place exclusively between peers within the swarm (collection of peers). This is opposed to the files residing on the centralized server's hard drive, where file sharing takes place exclusively be-

tween a client and the server. Because of the nature of the peer-to-peer architecture, the sharing of hardware resources between peers eliminates the imminent need for state of the art server hardware. Consequently, the peer-to-peer architecture immediately increases the reliability and redundancy of the file distributions. At the same time, it reduces the need of introducing technologies such as load balancing and failover systems.

The first generation of peer-to-peer architectures origins from 1999 when the Napster software revolutionized audio file sharing over the Internet. In this first generation however, the peer-to-peer architecture was in its infancy, provably containing some rather problematic properties. While it is true that peers exchanged content between themselves in a decentralized fashion, there was a centralized server entity present as well. This entity primarily coordinated the establishment of connections between peers. However, it also indexed all of the files in possession by the participating peers, and let peers execute search queries to locate other peers with files of interest. Because of this hybrid property of both centralization and decentralization, Napster ended up with a conviction in the infringement of copyrighted material [1].

After the first generation of peer-to-peer architectures, several improved architectures followed in the subsequent years. However, the BitTorrent protocol proposed by Bram Cohen in 2001 got the most attention [6]. This protocol was a very special peer-to-peer protocol, and after its introduction things seemed to fall into place. The BitTorrent protocol remains the preferred peer-to-peer protocol even today.

Many people in the public, especially those without background in computer science related disciplines, today associate the BitTorrent protocol with illegal piracy. While it is true that a lot of illegal piracy is distributed using the BitTorrent protocol, its use is widespread in almost all areas where the distribution of large files is necessary. While it makes sense to credit this fact to the protocol being open source, it is also apparent that the protocol's capability of performing at extraordinary levels running on commodity hardware is essential in its success. One of the primary reasons for its excellent performance is its ability to download from multiple peers simultaneously. This means that a single peer's poor Internet connection will not be a bottleneck for a file's transfer rates, but rather that the transfer rates will grow increasingly depending on how many peers are offering the file.

Since the protocol is open source, anyone who wishes to utilize it in his or her applica-

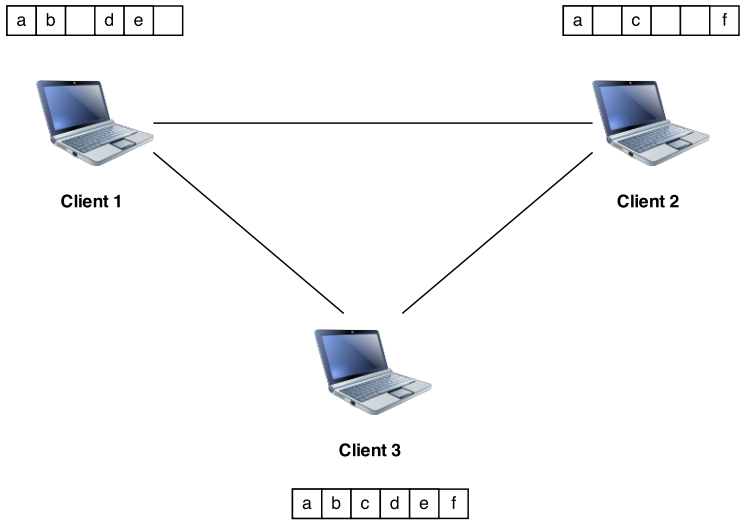


Figure 2.3: Clients with different file piece availability

tions may do so free of cost. Since the arrival of the BitTorrent protocol, file distribution has changed dramatically. The following will elaborate more thoroughly on how the BitTorrent protocol works and how it is able to outperform the client/server architecture.

How It Works

A file distributed over the BitTorrent protocol is an array of pieces, each piece representing a small portion of the file. A file piece is typically 256KB in size and the BitTorrent protocol allows peers to download files piece by piece. If a single peer is in possession of the entire file's array of pieces, the terminology of the BitTorrent protocol classifies that peer as a seed. In other words, a seed is either a peer that is the source of a file, or a peer that has been able to download all pieces of a file and is still participating in the swarm. In the process of sharing a new file, there is typically a single seed (the source) initially. Over time, as more and more peers acquire the complete file, the swarm may contain multiple seeds, i.e. the swarm becomes *healthy*.

The technique of splitting files into pieces enables peers to acquire an arbitrary amount of pieces over one download session, before returning in a later download session to continue the process. This property is very convenient when talking about files of excessive size. Figure 2.3 illustrates the differentiating availability of pieces between 3 participating clients. At this point in time, Client 3 is classified as the swarm's seed.

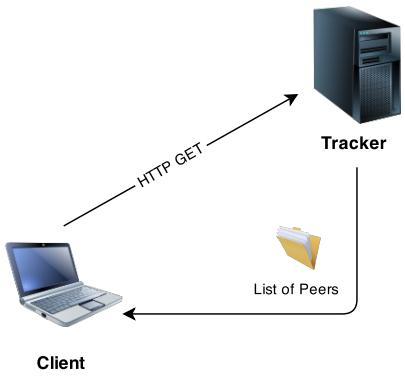


Figure 2.4: The initial client request to the tracker

To enable peers to find each other in a file distribution, the BitTorrent protocol consists of an entity called the *tracker*. The tracker is an application running on a web server responsible for coordinating the communication between peers. When a peer is interested in a file, it typically establishes a HTTP connection to the tracker to find out which of the other peers are in possession of that particular file's pieces. This is illustrated in Figure 2.4. This connection also serves the purpose of letting peers inform the tracker of their respective states. The peers send requests known as *announce requests* to the tracker continuously at specific time intervals. Announce requests includes letting the tracker know how much a peer has been able to download so far, how much is remaining, which file pieces is in its possession etc.

Once a peer has established a HTTP connection to the tracker and started sending announce requests, the peer can use the information provided by the tracker to establish TCP connections to other peers in the swarm and to begin the file transfer. This is illustrated in Figure 2.5.

Considering the above, like Napster, the BitTorrent protocol contains a centralized entity as well. However, unlike Napster's centralized entity, the BitTorrent tracker does not know anything about the contents of the files it is tracking. The tracker contains a list of all peers in the system, state information for every peer and knows which of the peers are in possession of a file's pieces. Since the file transfer sessions goes on between peers, the bandwidth requirement of the tracker and its corresponding web server is very low. On the other hand, it is worth noting that the BitTorrent tracker is arguably the BitTorrent protocol's foremost weakness. The tracker is a single point of failure and its potential failure will interrupt the distribution of its tracked files [10].

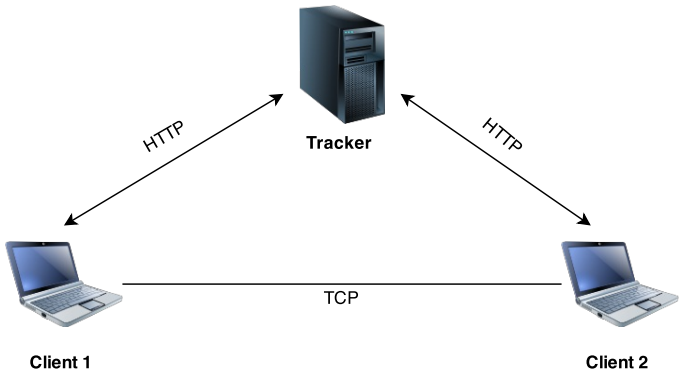


Figure 2.5: The relationship between the tracker and its peers

Like in a client/server architecture, the commonly used remedy for this issue is to run multiple trackers in parallel and to introduce load balancing and failover systems accordingly.

Every file's distribution in the BitTorrent protocol must have a corresponding meta-info file called the *torrent* file. The torrent file has the .torrent extension and typically resides in an arbitrary location on the Internet. Users that are interested in acquiring a file must obtain its corresponding torrent file. The torrent file contains information about the tracker, to which the user must connect to be able to locate the file's peers. In addition to the tracker information, the torrent file contains various other information, such as the file name, file size, piece length, hash values for each of the file's pieces etc. The purpose of the hash values is for the peers to ensure data integrity [6] at all times. Listing 2.1 shows a simplified illustration of a torrent file.

```
1 announce: 'http://some.tracker.url'
2 created by: 'creator'
3 creation date: 1427187561
4 info {
5         name: 'picture.jpg'
6         piece length: 256000
7         pieces: 'abcd' 'efgh' 'ijkl'
8 }
```

Listing 2.1: BitTorrent metadata/torrent file

As illustrated, most of the information in the torrent file that is concerning the underlying

file resides in a dictionary/map commonly referred to as the *info dictionary*. When clients/peers send announce requests (HTTP GET) to the tracker, they include a 20 byte SHA-1 hash value of the info dictionary appended to the `info_hash` parameter. This parameter serves as an identifier for the particular torrent on the tracker. Other parameters include `ip`, `port`, `downloaded` etc. Listing 2.2 shows an example announce request.

```
1 "http://tracker.url:port/announce?info_hash=abcdefg12345&peer_id=1&ip=192.168.1.1
2 &port=6881&downloaded=1234&left=4567&event=stopped"
```

Listing 2.2: BitTorrent announce request (HTTP GET)

Note that the torrent file illustrated in Listing 2.1 is heavily simplified. In reality, the BitTorrent protocol encodes the torrent files in a relatively uncommon binary format, namely the Bencode format (pronounced B-encode). This format is a data language that serves for storing data in a platform independent way. Some might argue that the Bencode format has many similarities with the XML and JSON formats in that respect. However, the Bencode format is a binary format and provides some unique properties that neither XML nor JSON has. Specifically, for every possible value, there is a one to one relationship between the particular plaintext value and its bencoded value. This property has the advantage of enabling applications to compare encoded values directly, instead of decoding/re-encoding for every comparison.

Data	Type	Bencode
hello	String	5:hello
6	Integer	i6e
{"one", 2, "three"}	List	l3:onei2e5:threee
{"year", 2015}	Dictionary	d4:yeari2015ee

Table 2.1: Bencoding Examples

As seen in Table 2.1, all strings in the Bencode format is length prefixed, meaning that to store the string "hello", it would need to be stored as "5:hello", indicating that it is five characters long. To store integers, one must prepend a prefix "i" as well as a suffix "e". To store the number "6", it would need to be stored as "i6e". To encode a list it would need to look like "l<contents>e", while a dictionary's encoding would need to look like "d<contents>e". The contents of a dictionary must appear as alternations between keys and their corresponding value.

Since the underlying file transfer between peers utilize the TCP protocol, BitTorrent enforces a pipelining structure to limit the delay between pieces. To clarify, as we already know, files splits into an array of pieces. However, the BitTorrent protocol actually continues to split a file's pieces into even smaller chunks known as sub-pieces [6]. Sub-pieces are typically 16KB in size. This allows the protocol to keep several requests pending at any given time to increase the performance of the underlying TCP protocol. Every time a sub-piece arrives, a peer sends a new request.

When sending requests for pieces it is very important to request the pieces in a clever order. The BitTorrent protocol implements several piece selection algorithms to ensure of this. During a file's distribution, the peers continuously report to all of its fellow peers about which pieces it possesses. Note that a peer will not report about a piece before it has checked its hash value with the hash value present in the respective torrent file. A bad piece selection algorithm can cause a peer to be in possession of all the pieces its fellow peers can provide at a given time, resulting in poor progression. Similarly, it can also cause a peer to be unable to provide its fellow peers with pieces that they do not already have.

When determining which pieces to request, the BitTorrent protocol enforces the so-called *strict* policy. This means that once a peer has obtained a sub-piece of a particular piece, subsequent requests will always entail the remaining sub-pieces of that same piece. This will ensure that peers acquire complete pieces as soon as possible, rather than becoming stuck with a bunch of sub-pieces unable to offer anything to other peers.

When a peer starts downloading for the first time, it will select a piece to download at random. This is because the peer will be unable to upload anything, and the BitTorrent protocol wants it to acquire a single complete piece as soon as possible, so that it can start contributing to the file's distribution. Once a peer has obtained a complete piece, the strategy changes from random to selecting the rarest pieces first. The rarest first approach increases the chance that a peer will be in possession of pieces that other peers are missing. By postponing the requests for more common pieces, it also increases the chance that the peers in possession of the common pieces will still be offering them later.

The Download Process

Now that we understand the BitTorrent protocol to some extent, the following will describe the process of downloading a file, from a regular user's point of view.

1. Obtain the torrent file containing the metadata about the desired file and open it in any BitTorrent client. The torrent file may reside anywhere. For instance, the user may browse the web, stumble upon the torrent and download it through their browser as they would with any other file.

2. After opening the torrent file in a BitTorrent client, the client will be able to parse the Bencoded file and to locate the URL of the tracker. The client will subsequently connect to the tracker.

3. If the BitTorrent client successfully connected to the tracker, the tracker will report information about the current peers participating in the distribution of the file back to the client. The client will make corresponding connections to these peers.

4. The client will begin downloading the file's pieces from the peers to which it is connected.

5. Eventually, the client will have downloaded all the pieces and the entire file will be at the user's disposal. At this time, the client can safely leave the swarm, or remain to continue the contribution of the file's distribution.

Security Concerns

While the BitTorrent protocol is very convenient and efficient, there are some concerns regarding its security and privacy capabilities. To be fair, the BitTorrent protocol's design does not involve security or privacy at all, so the fact that it is lacking in these areas is rather obvious. However, some application areas require some security, and there does exist some ways to achieve it, at least to some extent. However, introducing security and privacy capabilities to the protocol means that there will be a performance tradeoff.

The most obvious security concern with the BitTorrent protocol is the HTTP GET requests happening between the peers and the tracker. These requests are happening in plaintext, and as such, any eavesdropper is able to see all the traffic going on between peers and the tracker.

Additionally, the TCP connections established between the peers after they have determined which of the other peers in the swarm they will request the file's pieces from, is also completely open.

To remedy these concerns, it is possible to force the client requests between the peers and the tracker to convey over a secure channel by using HTTPS (SSL/TLS). However, this means that both the tracker and the clients must both support this, which is not always the case. Additionally, it is possible to force the clients to encrypt the traffic between peers as well, typically with a stream cipher such as RC4. Again, this is only possible if the respective BitTorrent clients support this.

Chapter 3

Design and Implementation

The following sections will elaborate on the design and implementation of the thesis sharing system. It will attempt to answer questions such as what the general idea behind the system is, what its purpose is and how it differs from other work out there.

3.1 Motivation

The primary motivating factor behind the thesis work is to provide a secure mechanism for sharing files that are stored on the Internet. Many of today's cloud storage providers (CSPs) are lacking when it comes to protecting their users' data. A common denominator for several providers out there is that they appear to invest heavily in making sure their users' files are stored in a secure manner. This typically entails that the files are satisfactory encrypted in accordance to today's recommendations and standards. However, at the same time several CSPs openly admit that they are capable of decrypting their users' files. This relates to the fact that the encryption algorithms in use are typically symmetric, and the cloud storage providers themselves are in possession of all the keys. This means that they have complete control of their users' files and if required by laws and regulations, are able to provide government representatives with the same access. Of course, the CSPs claim that only a handful of engineers have the necessary credentials to access the decryption keys and that the users need not to worry. However, given the incidents featured in the media in the recent years where private files have ended up in the wrong hands, such claims might not be sufficient.

3.2 System Analogy

To some extent, the sharing system leaves the idea of directly sharing files between parties behind. Instead, it introduces a sort of *map* or *location* analogy. This analogy becomes possible by utilizing so-called metadata files, which is part of the BitTorrent protocol domain. When users share files between themselves using the thesis sharing system, they actually share the metadata/location of the files between them instead. In other words, when Alice wants to share a private file with Bob, she shares the encrypted metadata of the file with Bob instead. This way, when Bob decrypts the metadata, he can access the file location.

3.3 Design

While it is convenient to think of the thesis sharing system as sharing the location of files between each other, it is not that simple in practice. To elaborate, when Alice shares the location of her private file with Bob, she actually shares the private file's corresponding metadata file, as ciphertext. The torrent files part of the BitTorrent protocol domain is exactly that, metadata files. Because of this, the BitTorrent protocol fits like hand in glove for the thesis sharing system. Upon receiving the encrypted torrent file (metadata), Bob is capable of learning the private file's location by decrypting it and examining its contents. Upon decryption, Bob learns the location of the file by accessing the tracker URL present in the torrent file. From the decrypted torrent file, Bob is able to obtain the underlying file through an arbitrary BitTorrent client.

In using the thesis sharing system, all interaction must go through a web service API. The functions exposed by the API is reachable through HTTPS calls and is capable of performing all the necessary actions to share files between users. An elaboration on exactly how the system implements this follows later in this chapter.

3.3.1 Framework/System Architecture

To elaborate on the sharing system's architecture, the following will walk through a sample scenario illustrating how the sharing system can work in practice. Assume that Alice is the professor of a cryptography course at a university. Bob and Charlie are both participating in the

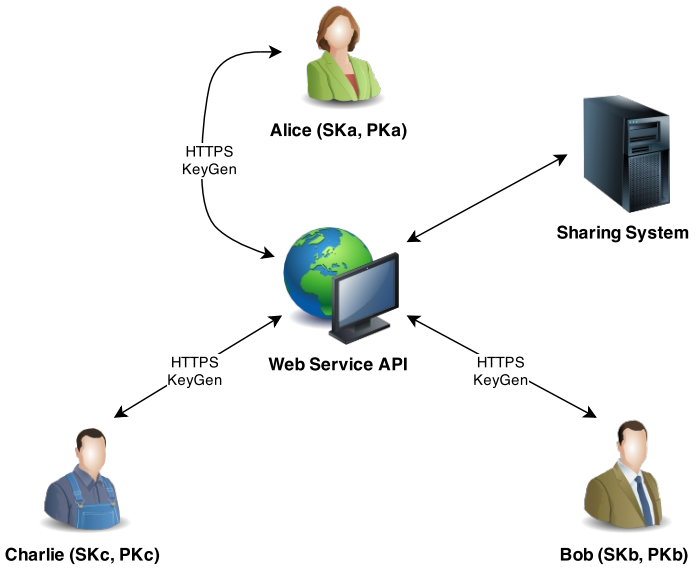


Figure 3.1: All parties generate their respective key pair separately

Figure 3.2: Alice generates a torrent/metadata file for the course syllabus document

cryptography course. Alice wants to be able to share the course syllabus document with all of her students. Moreover, she wants to achieve this in a secure and convenient manner. As we shall see, the thesis sharing system will meet this requirement.

To start using the sharing system, all parties need to obtain their own respective key pair, one secret key and one public key. The process of obtaining key pairs is illustrated in Figure 3.1. Note that the respective calls to *KeyGen* is in practice two different calls, one call to *newSecretKey* with no parameter and one call to *newPublicKey* with the respective secret key as its parameter. The figure illustrates this as a single call to a non-existent function *KeyGen* for simplicity.

To initialize the file sharing process, Alice starts by generating a torrent/metadata file for the syllabus document by calling the *newTorrent* API function, as illustrated in Figure 3.2.

Now that Alice has generated a metadata file for the course syllabus, she uploads the newly

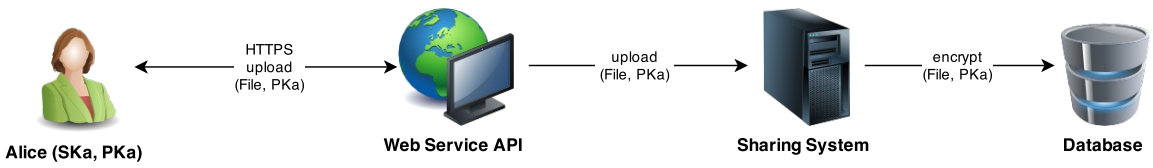


Figure 3.3: Alice uploads the torrent to the sharing system

Figure 3.4: Alice generates the re-encryption key $RK_{A \to C}$

generated torrent to the sharing system by calling the *upload* function. Alice supplies her own public key as one of two parameters to this function. This function call will make the sharing system generate a unique id (file name) for the torrent before announcing it on its BitTorrent tracker. Successively, it encrypts the torrent with Alice's public key and the resulting ciphertext is stored in a database. Figure 3.3 illustrates this procedure.

To finalize the initialization process, Alice starts seeding the torrent by opening it in a BitTorrent client while also making a note of the system generated id.

Upon student inquiry about the course syllabus, Alice generates a re-encryption key with her own secret key and the student's public key. For instance, if Charlie is the inquirer, Alice generates $RK_{A \to C}$. Figure 3.4 shows an illustration of the re-encryption key generation.

In possession of $RK_{A \to C}$, Alice initiates the sharing system's *share* function, supplying the torrent *id*, Charlie's public key PK_C and the newly generated re-encryption key $RK_{A \to C}$ as parameters. This will let the sharing system know that Alice has granted Charlie access to the course syllabus torrent. The sharing system will store these publicly available parameters in its database. Figure 3.5 illustrates this procedure. Note that all this information is public and as such, storing it as plaintext in the system's database does not cause security concerns.

Followed by this, Alice must inform Charlie of the torrent's id. Figure 3.6 illustrates this in a simple manner. Note that there is no specification on how this should take place. Regardless of

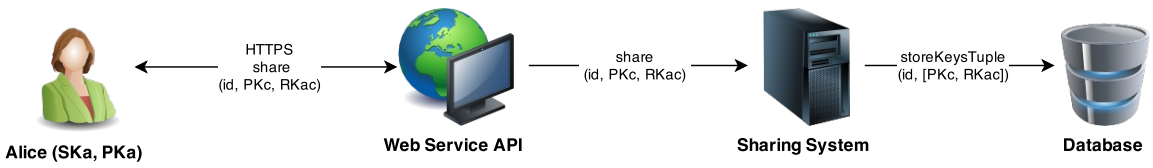


Figure 3.5: Alice shares the course syllabus torrent with Charlie

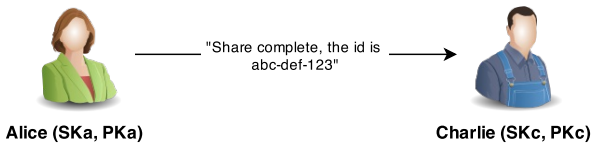


Figure 3.6: Alice informs Charlie of the torrent id

how the torrent id travels from Alice to Charlie, the reality is that Charlie is required to know the torrent id to proceed.

It is also worth noting that Alice must call the *share* function and generate $RK_{A \rightarrow X}$ once for every student to which she wants to share the syllabus. In other words, Alice must repeat the sharing procedure if she is to delegate syllabus access to Bob or any of the other students participating in the course later.

In possession of the torrent's *id* and a notification from Alice about the newly acquired access, Charlie initiates the sharing system's *download* function, supplying the torrent id and his own public key PK_C as parameters. Figure 3.7 illustrates this procedure. With these parameters, the sharing system looks up the torrent matching the parameter given id in its database and verifies whether there is a record containing Charlie's public key present. If Alice invoked the sharing function correctly, this should be the case. If so, there is also a re-encryption key present in the same record, which the sharing system uses to re-encrypt the torrent ciphertext before returning it to Charlie.

In possession of the new ciphertext, Charlie is capable of decryption with his own secret key. Upon decryption, the torrent/metadata file is in the hands of Charlie. At this point, Charlie can open the torrent in a BitTorrent client and obtain the course syllabus. Figure 3.8 illustrates the decryption call.

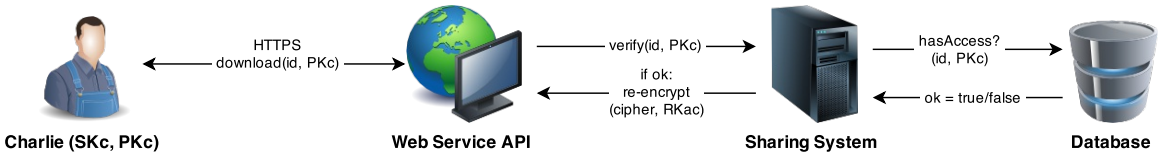


Figure 3.7: Charlie downloads the torrent file (ciphertext)

Figure 3.8: Charlie decrypts the torrent file (ciphertext)

If we compare the above procedure to many of today's sharing practices, one major difference comes to mind. When relying on third parties to re-encrypt content in more traditional public key cryptosystems, said third party must be fully trusted. This is because it is necessary to decrypt ciphertext into plaintext before re-encrypting it to new ciphertext compatible with another key. Because of the strength of the proxy re-encryption primitive, this is not the case in the above scenario. Alice, Bob and Charlie are all depending on the sharing system as a third party proxy instance. However, the sharing system only interacts with publicly available information/keys, and thus need not be trusted. Some sensitive information *does* travel between entities. However, this is protected by the HTTPS protocol.

3.4 Implementation

The implementation of the thesis sharing system consists of multiple components. The primary components are as follows, all of whose respective implementations are separate projects in the underlying source code workspace.

- Common
- Cryptography

- BitTorrent
- Sharing System
- Web Service

These five projects have dependencies between themselves. All projects have a reference to the `common` project, which mostly consist of static variables such as the path to different files and directories in use by several different parts of the system.

Throughout the development, the source code management took place with use of Git [27] as a version control system. The purpose of version control systems is primarily to help manage source code changes between several developers working on the same source code. There are many different version control systems available out there. Git differs from many of them in that it is decentralized, meaning that developers have their own copies of the repository on their own machines, instead of accessing the repository directly on a single server placed on a centralized location.

During the thesis development, the Git repository was set up as a GitHub [28] *private* repository. GitHub is a web-based hosting service for Git repositories. Private repositories are repositories that only people with the appropriate credentials can access and requires a small subscription fee. On the other hand, public repositories are available to anyone browsing the web. Given that only one developer worked on the thesis, GitHub primarily functioned as a graphical user interface for browsing past source code changes as well as a backup tool. By committing source code changes to the local repository first, before pushing these changes to a remote location (GitHub), backup capabilities became a side effect of using Git as a version control system. At the time of submission, the Git repository consists of 139 different commits where 90.2% is Java code, 5.5% is HTML and 4.3% is JavaScript (See Figure 3.9).

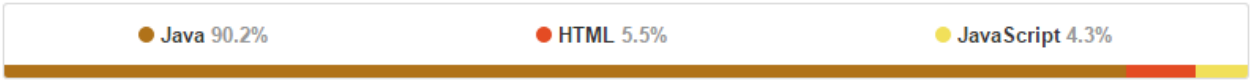


Figure 3.9: GitHub Repository Language Distribution

These numbers are representing *new* code exclusively, meaning no modified code is included here.

Note that most source code present in the five projects is Java code. Moreover, all projects are Maven [18] projects and thereby follow Maven-like practices. Maven is part of the Apache Software Foundation and is a project management tool. An essential part of Maven projects is that they all contain their own Project Object Model (POM) file. The POM file is an xml file that contains information about the project's configuration, dependencies and more. In other words, the dependency configuration between the five projects reside in the respective project's `pom.xml`.

Project Name	Description
common	Static functions and variables used by the other projects
crypto	Implements the AFGH proxy re-encryption scheme
bit-torrent	All BitTorrent related functionality
secure-share	Implements the thesis sharing system
web-service	Exposes the thesis sharing system as a web service API

Table 3.1: Implementation Overview - Project Descriptions

Project Name	Package	Dependencies
common	no.uis.msalte.thesis.common.*	-
crypto	no.uis.msalte.thesis.crypto.*	common
bit-torrent	no.uis.msalte.thesis.bit_torrent.*	common
secure-share	no.uis.msalte.thesis.secure_share.*	common, crypto, bit-torrent
web-service	no.uis.msalte.thesis.web_service.*	common, secure-share

Table 3.2: Implementation Overview - Packages and Dependencies

The source code of all projects obviously reside inside their own unique Java packages, but they all utilize a common package prefix `no.uis.msalte.thesis.*`. For instance, the `common` project resides in `no.uis.msalte.thesis.common.*`. Table 3.1 shows an overview of the respective project's descriptions while Table 3.2 shows an overview of the project's packages and dependencies. The `web-service` project is in bold because it represents the sharing system's entry point.

3.4.1 Cryptography

The cryptography project resides in `no.uis.msalte.thesis.crypto.*` and implements the AFGH proxy re-encryption scheme. As mentioned previously in the paper, such schemes are

a recent addition to the crypto world. Proxy re-encryption schemes are asymmetric cryptosystems with several convenient properties for a sharing system. The primary benefit is its capability of delegating decryption rights to ciphertexts to additional parties without the necessity to access the plaintext first. As mentioned previously in the paper, the AFGH proxy re-encryption scheme bases upon the ElGamal cryptosystem as well as pairing based cryptography, also known as bilinear maps. The implementation residing in this project uses a third party library for generating the bilinear maps and other scheme parameters, namely the Java Pairing Based Cryptography (JPBC 2.0.0) library [4]. This library is a Java port of the C library published as Pairing Based Cryptography (PBC). Ben Lynn, the author of "On the Implementation of Pairing-Based Cryptosystems" [9], is the primary developer behind the PBC library.

```
1  public interface ProxyReEncryptionScheme {
2          public String newSecretKey();
3          public String newPublicKey(String secretKey);
4          public String newReEncryptionKey(String srcSecretKey, String destPublicKey);
5          public String encrypt(String message, String destPublicKey);
6          public String decrypt(String cipher, String destSecretKey);
7          public String reEncrypt(String cipher, String reEncryptionKey);
8          public String encryptReEncryptable(String message, String destPublicKey);
9          public String decryptReEncryptable(String cipher, String destSecretKey);
10 }
```

Listing 3.1: AFGH PRE Interface - ProxyReEncryptionScheme.java

This portion of the thesis sharing system, in addition to its general design, is arguably in highest regard when it comes to novelty work. The authors of the AFGH proxy re-encryption scheme has implemented their algorithms in a C++ library known as the JHU-MIT Proxy Re-cryptography Library [33]. There appears to be an attempted Java implementation of AFGH out there as well [31], but it seems incomplete and abandoned [32]. Consequently, and since the thesis sharing system is primarily Java-based, it seemed like a good idea to include a complete implementation of the AFGH scheme as part of the thesis work. It also seemed very rewarding academically as it would ensure a correct understanding of the algorithm. Moreover, it also seemed beneficial in regards to compatibility and lack of additional overhead caused by refer-

encing a C++ library from Java source code.

Listing 3.1 shows a Java interface that defines the primary methods implemented in this project. As shown, all methods operate on `String` parameters. This is primarily for convenience and compatibility. In practice, unless these strings represent plaintext, they are Base64 encoded. Base64 is an encoding format that encodes binary data to textual data, which is very convenient in this case. The underlying implementations convert the `String` objects to more specific objects before performing the mathematical operations on them etc. To elaborate, the objects on which the crypto project operates are primarily part of the JPBC library, specifically in the `it.unisa.dia.gas.jpbc` package. To mention some of the most important ones, the `Field` interface represents a generic algebraic structure. The `Pairing` interface gives access to common pairing functions. Finally, the `Element` interface represents an element of a group or a field.

In the `crypto` project's source code, the class `no.uis.msalte.thesis.crypto.scheme.ProxyReEncryptionParams` is responsible for generating the parameters for the AFGH scheme. Moreover, `no.uis.msalte.thesis.crypto.scheme.ProxyReEncryptionSchemeImpl` implements the interface in Listing 3.1. The `ProxyReEncryptionSchemeImpl` class instantiates the `ProxyReEncryptionParams` class in its constructor.

3.4.2 BitTorrent

The BitTorrent projects resides in `no.uis.msalte.thesis.bit_torrent.*` and primarily provide the sharing system with a custom BitTorrent tracker implementation. The project is highly dependent upon a modified version of a library known as ttorrent [19]. The ttorrent library is an open-source implementation of the BitTorrent protocol written in Java. Because of this work's necessity for increased security capabilities, significant modifications to this project was imperative. Note that the ttorrent library is licensed under the Apache Software License version 2.0.

The modifications to ttorrent includes the enforcement of SSL/TLS on all HTTP requests made to the tracker. This enforcement is the result of a `javax.net.ssl.SSLContext` instance. The `SSLContext` object builds from a certificate generated by the Java `keytool` feature [35]. The `keytool` result is a Java KeyStore file (.jks), which functions as a repository of SSL/TLS certificates. In possession of the generated Java KeyStore, the source code builds the `SSLContext`

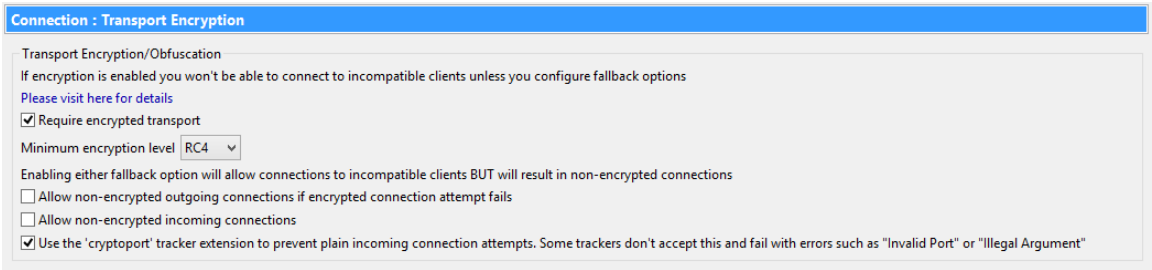


Figure 3.10: Vuze Client Encryption Options

object and the server is capable of providing secure connections.

Additionally, the tracker will abandon all clients that does not implement encryption between themselves and other peers. Most of today's popular BitTorrent clients have the capability of enabling the RC4 stream cipher for communication with other peers. However, most clients disable this option by default. In other words, users of the modified tracker must enable the encryption option manually.

To elaborate on this behavior, the tracker expects incoming *announce* requests to include an additional parameter `requirescrypto=1` to let it know that a client is indeed capable of encryption. If clients query the tracker without this parameter in the HTTP GET request, the tracker will abandon the request and return a HTTP 406 error message, meaning that the request was unacceptable.

The `requirescrypto` flag is part of a tracker extension specification provided by the Azureus (now Vuze) MSE (Message Stream Encryption) documentation. Because of this, many BitTorrent clients are not providing this flag even though they are capable of encryption. However, popular clients such as Vuze and Transmission does supply this flag in their requests when enabling encryption. Figure 3.10 shows the encryption options available in the Vuze client.

To summarize, the modifications to the BitTorrent tracker makes the thesis sharing system discriminate slightly on BitTorrent clients. Only clients supplying the `requirescrypto` flag in their HTTP GET requests is accepted. While this property is an obvious weakness, it was decidedly better to support only a handful of the most popular clients than it would have been to support none of them. The alternative would involve developing a new BitTorrent client as part of the thesis, but this would significantly reduce the sharing system's appeal.

In addition to providing the sharing system with a customized BitTorrent tracker, this project

also provides the sharing system with torrent generation functionality. In other words, by supplying the static function `BitTorrentUtils.createBase64()` with an arbitrary file such as a document or a picture, it will return a Bencoded torrent/metadata file as a Base64 encoded `String` object. In addition to default behavior, this function will append two additional fields to the torrent's info dictionary. The first field is the `private=1` field, which is an optional field part of the BitTorrent protocol. The *private* field serves the purpose of preventing clients from sharing torrents with clients that does not have access to the tracker. In other words, clients will only announce to the tracker and not between themselves.

The second field appended to the torrent's info dictionary is a custom field `atoken=<random unique value>`. This field contains a randomly generated universally unique identifier (UUID) and serves the purpose of forcing the torrent's info hash to change. This is to ensure that it is impossible to reverse engineer the torrent by looking at server logs. The sharing system's other BitTorrent functionality never access the *atoken* field and thus its only purpose is obfuscation. The curious reader might wonder why the field's name is "atoken" instead of "token". The answer is that a torrent's info dictionary always appear alphabetically sorted and that the info hash is a 20 byte SHA-1 value. By ensuring that the field occurs in the beginning of the dictionary, one can be certain that the field is included in the first 20 bytes and thus affects the resulting hash value.

3.4.3 Sharing System

The sharing system project resides in `no.uis.msalte.thesis.secure_share` and implements the sharing logic itself. As illustrated in Table 3.2, this project is dependent upon the *common*, *bit-torrent* and *crypto* projects. Through its dependencies, this project implements the complete sharing system. With a reference to this project as a library/dependency, the complete thesis sharing system's implementation is available to other projects.

```java
public interface SecureShare {
        public String newTorrent(File file);
        public String decrypt(String ciphertext, String secretKey);
        public String newSecretKey();
        public String newPublicKey(String secretKey);
        public String newReEncryptionKey(String srcSecretKey, String destPublicKey);
        public String upload(File file, String publicKey);
        public boolean share(String fileName, String publicKey, String reEncryptionKey);
        public String download(String fileName, String publicKey);
        public String announce(File file);
}
```

Listing 3.2: Sharing System Interface - SecureShare.java

Listing 3.2 shows a Java interface that defines the primary methods exposed by this project. As shown, this interface defines methods primarily for sharing functionality, but also for key generation in the referenced cryptography scheme.

The sharing system project implements a simple and lightweight database layer that is responsible for storing information about existing share operations. The implementation of the database/persistence layer is specifically two key-value stores (NoSQL databases), one for active share relationships known as *shares* and one for encrypted torrent content known as *torrents*. The database layer runs on the MapDB database engine [20], which is an in-memory database engine with high focus on performance and simplicity. In-memory databases load the complete database into memory during runtime and makes backups of the database as a dump file on the file system on application exit. This behavior ensures exceptional performance and removes the necessity of a database server instance, such as the case is with the more traditional relational databases.

Key	Value
545c728b-cef4-44d7-b261-6690bf3b07ce.torrent	hkSQsuAlgas7yhrlATMo5bSChPD8PsWidLefGlQlC67D6H...
66b297b6-ff8a-447a-b365-831edc76c427.torrent	A6aZ1CQgulpJ+WTrQTuUABl2oYtj8ISAKtX4Hb53xPLez...

Table 3.3: Persistence Layer - Torrents

As shown in Table 3.3, the *torrents* database is a key-value store/map of file names and encrypted file content. This particular example contains two records/keys. The sharing system generates the torrent file names upon torrent generation as a universally unique identifier (UUID). This ensures that it is impossible for two torrents to have the same name/id in the system. Because the file names function as the key field in the database map, the guarantee of unique file names is a necessity.

Key	Value
545c728b-cef4-44d7-b261-6690bf3b07ce.torrent	[{"publicKey":"pk1","reEncryptionKey":"rk1"}, {"publicKey":"pk2","reEncryptionKey":"rk2"}, {"publicKey":"pk3","reEncryptionKey":"rk3"}]

Table 3.4: Persistence Layer - Shares

Similarly, as shown in Table 3.4, the *shares* database is a key-value store/map of file names and a corresponding JSON object. JSON stands for JavaScript Object Notation and derive from the syntax used for representing objects in the JavaScript language. The JSON object functioning as the database map's value field is an array of key tuples. Every key tuple contains a public key and a corresponding re-encryption key. The array of key tuples functions as the sharing system's access control system for every torrent file. From the data stored in the *shares* database, the sharing system is able to validate access to stored torrents upon user inquiry. If the user supplies the sharing system with a file name and its own public key, the sharing system can look up the file name in the *shares* database and subsequently determine whether the supplied public key exists in the corresponding JSON object. If both these tests are successful, the sharing system can determine that the user has valid access to the torrent file. From this, the sharing system looks up the corresponding re-encryption key in the respective JSON object and uses it to re-encrypt the encrypted content stored in the *torrents* database under the same key (file name). The result is new ciphertext representing the user inquired torrent. The user is capable of recovering the plaintext/torrent with his secret key upon download.

In the source code, all database actions are reachable from the `Persist.java` singleton class. As with all classes implementing the singleton pattern, one does not create a new instance of the class through its constructor, rather by calling its static `Persist.getInstance()` method. To enforce this, a singleton class typically defines its constructor with the *private* mod-

ifier, meaning it will not be visible outside its own class. Using the singleton pattern for the `Persist` class ensures that the application is limited to a single static reference to the persistence layer, meaning that conflicting database operations will not occur.

3.4.4 Web Service (System Entry Point)

The web service project resides in `no.uis.msalte.thesis.web_service.*` and implements the user interface of the sharing system. This project serves as the entry point of the entire sharing system, as it exposes an API through the HTTP protocol. By calling the functions exposed by the web service API, it is possible to incorporate the entire thesis sharing system into existing applications. As with the sharing system's embedded BitTorrent tracker, the web server providing the web service API also enforce its HTTP requests to go over SSL/TLS.

The web service runs on a framework known as Spark [21], which is a lightweight Java-based web application framework. By default, the Spark framework runs on an embedded web server on top of Jetty [22]. The primary reason for choosing the Spark framework was its simplistic approach and almost absent necessity for configuration. When using Maven, specifying the Spark framework as a dependency in the project's `pom.xml` was sufficient configuration. From this point, exposing the sharing system API through both HTTP GET and HTTP POST functions was very easy.

```
1 import static spark.Spark.*;
2 public class HelloWorld {
3     public static void main(String[] args) {
4         get("/get_hello", (req, res) -> "GET: Hello World");
5         post("/post_hello", (req, res) -> "POST: Hello World");
6     }
7 }
```

Listing 3.3: Simple Spark Framework Example

Listing 3.3 shows a simple example of a Spark web service. On line 4 we can see the specification on how the web server should react to a HTTP GET request on the `/get_hello` path. The second parameter to this function is a so-called lambda expression, which is a recent intro-

duction to Java, specifically to Java 8. This particular lambda expression returns a simple string ‘‘`GET: Hello World`’’. Similarly, on line `5` we can see a specification on how the web server should request to a HTTP POST request on the `/post_hello` path. To enable HTTPS on Spark's default Jetty web server, it is necessary to call `Spark.secure()` and supply a Java KeyStore file as one of the parameters. Amazingly enough, the code snippet in Listing 3.3 is all that is required to run a simple web service on the Spark framework. However, to support the functionality present in the thesis implementation the required code is necessarily slightly more complex. The basic idea is still the same.

API Path	Description
/api	Returns an overview of the API
/newSecretKey	Generates a new secret key in the underlying PRE scheme
/newTorrent	Generates a new torrent file from a parameter given file
/upload	Uploads a parameter given torrent file and encrypts it before storing it
/newPublicKey	Generates a new public key in the underlying PRE scheme
/newReEncryptionKey	Generates a new re-encryption key in the underlying PRE scheme
/decrypt	Decrypts the parameter given ciphertext
/share	Shares a parameter given torrnet file with another user
/download	Downloads a torrent file matching the parameter given file name
/announce	Manually announces a torrent file on the system's tracker

Table 3.5: Web Service API Description

API Path	Implementation	Extends	Method
/api	ApiGetRoute.java	WebServiceRoute	GET
/newSecretKey	NewSecretKeyGetRoute.java	WebServiceRoute	GET
/newTorrent	NewTorrentPostRoute.java	WebServiceRoute	POST
/upload	UploadPostRoute.java	WebServiceRoute	POST
/newPublicKey	NewPublicKeyPostRoute.java	WebServiceRoute	POST
/newReEncryptionKey	NewReEncryptionKeyPostRoute.java	WebServiceRoute	POST
/decrypt	DecryptPostRoute.java	WebServiceRoute	POST
/share	SharePostRoute.java	WebServiceRoute	POST
/download	DownloadPostRoute.java	WebServiceRoute	POST
/announce	AnnouncePostRoute.java	WebServiceRoute	POST

Table 3.6: Web Service API Implementation

In the thesis source code, the class `no.uis.msalte.thesis.web_service.server.Server` is responsible for setting up the Spark web service. Moreover, all the API functions have their

own separate class in the `no.uis.msalte.thesis.web_service.routes` package. These classes all extend the abstract class `no.uis.msalte.thesis.web_service.routes.WebServiceRoute`, which in turn extends the Spark framework abstract class known `spark.RouteImpl`. This class provides its descendants with constant variables as well as some code that is common for all routes (See Listing 3.4). The primary purpose of the `RouteImpl` class is its `path` variable and its declaration of the `handle()` method. Because `WebServiceRoute` extends `RouteImpl`, that class must reflect these properties as well. The Spark framework uses the `path` variable for URL matching of incoming requests while also making sure that any request matching its respective path variable is handled by the correct and corresponding `handle()` method. As illustrated in Listing 3.4, the `handle()` method contains approximately ten lines of code. This code is common for all routes, and make up the basis upon which the specific route implementations build. To clarify, the descendants of `WebServiceRoute` start their own respective `handle()` implementations by calling `super.handle()` to fetch the initial state to which they make modifications.

Table 3.5 shows an overview of the functions exposed by the web service API. Moreover, Table 3.6 shows an overview of the classes responsible for handling the different HTTP requests, in other words the classes that extend the `WebServiceRoute` abstract class. Finally, Table 3.7 shows an overview of the parameters, arguments and return values in respect to each of the API functions. Note that the web service treats all HTTP POST requests as multipart/form-data, which is an encoding method used by HTML forms consisting of the “file” input type element.

```
1  public abstract class WebServiceRoute extends RouteImpl {
2
3          public static final String PARAM_FILE = "file";
4          public static final String PARAM_SECRET_KEY = "secretKey";
5          public static final String PARAM_PUBLIC_KEY = "publicKey";
6          public static final String PARAM_FILE_NAME = "fileName";
7          public static final String PARAM_RE_ENCRYPTION_KEY = "reEncryptionKey";
8          public static final String PARAM_CIPHERTEXT = "ciphertext";
9
10         public static final String FUNC_API = "api";
11         public static final String FUNC_DECRYPT = "decrypt";
12         public static final String FUNC_NEW_TORRENT = "newTorrent";
```

```
13          public static final String FUNC_NEW_SECRET_KEY = "newSecretKey";
14          public static final String FUNC_NEW_PUBLIC_KEY = "newPublicKey";
15          public static final String FUNC_NEW_RE_ENCRYPTION_KEY = "newReEncryptionKey";
16          public static final String FUNC_SHARE = "share";
17          public static final String FUNC_UPLOAD = "upload";
18          public static final String FUNC_DOWNLOAD = "download";
19          public static final String FUNC_ANNOUNCE = "announce";
20
21          private boolean isPostRoute;
22
23          public WebServiceRoute(String path, boolean isPostRoute) {
24                  super(path);
25
26                  this.isPostRoute = isPostRoute;
27          }
28
29          @Override
30          public Object handle(Request request, Response response) throws Exception {
31                  final WebServiceResponse r = new WebServiceResponse();
32
33                  // by default, treat as BAD_REQUEST
34                  r.setStatus(HttpURLConnection.HTTP_BAD_REQUEST);
35                  r.setMessage("Invalid parameter(s)");
36                  r.setContent(null);
37
38                  if(isPostRoute) {
39                          // treating all post requests as multipart/form-data
40                          request.raw().setAttribute("org.eclipse.multipartConfig",
41                                          WebServiceUtils.MULTIPART_CONFIG);
42                  }
43
44                  return r;
45          }
46  }
```

Listing 3.4: Abstract Class for a Web Service Route - WebServiceRoute.java

API Path	Parameters	Arguments	Returns
/api	-	-	List
/newSecretKey	-	-	Base64
/newTorrent	file	Binary	Base64
/upload	file, publicKey	Binary, Base64	String
/newPublicKey	secretKey	Base64	Base64
/newReEncryptionKey	secretKey, publicKey	Base64, Base64	Base64
/decrypt	ciphertext, secretKey	Base64, Base64	String
/share	fileName, publicKey, reEncryptionKey	String, Base64, Base64	Boolean
/download	fileName, publicKey	String, Base64	Base64
/announce	file	Binary	String

Table 3.7: Web Service API Parameters, Arguments and Return Values

All the API functions return a JSON formatted result, more specifically a JSON representation of a `no.uis.msalte.thesis.web_service.model.WebServiceResponse` object. It is very common for web service APIs to return JSON because of its platform independency and relative human readability.

Sample JavaScript Client

The web service project contains a sample JavaScript client. This client is available at `https://hostname:port/jsclient.html` while the web service is running. This client uses the JavaScript framework known as AngularJS and is built exclusively on top of the JSON object returned by the web service function `/api`. In addition to AngularJS, the client uses HTML together with cascading style sheets (CSS) and JavaScript provided by the Bootstrap framework [26].

To see an example of the JavaScript client, see Figure 3.11. To elaborate, the figure shows an example of the JavaScript client when the selected function is “newPublicKey”. At the top of the page, there is a dropdown menu for selecting the appropriate API function to test. For any selected function, the client will responsively produce an overview of the particular function’s details. Such details are parameters, arguments, return values and more. Moreover, the client also produces a simple HTML form for testing the function. Finally, upon entering valid data to said HTML form and pressing the submit button, the client will responsively produce the function’s result as a JSON object. In the top-right corner of the page, there is a conversion function capable of converting a Base64 encoded string to a torrent file.

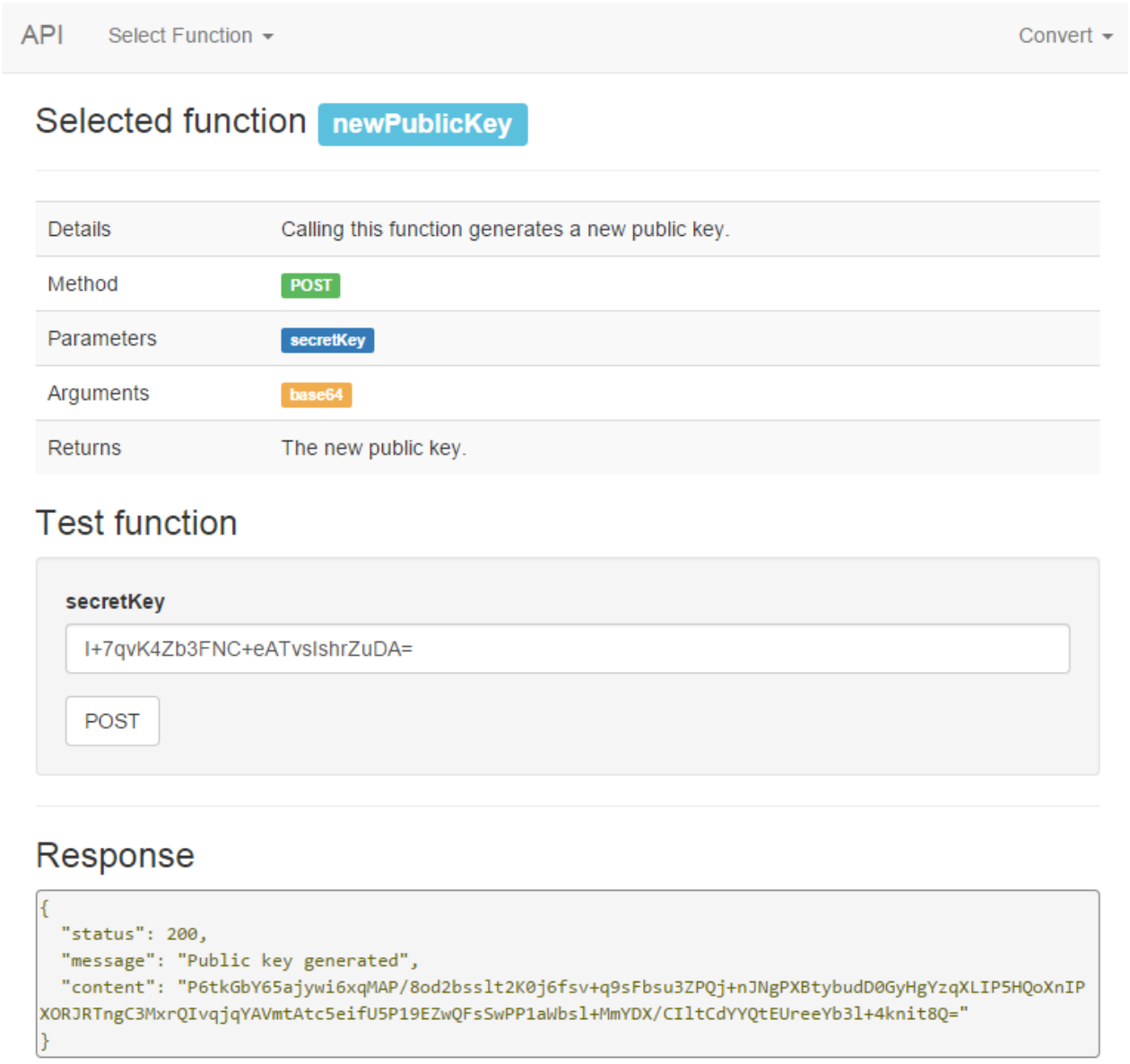


Figure 3.11: JavaScript Client - Selected Function: newPublicKey

Again, understanding that the JavaScript client is exclusively built on top of the JSON object returned by the `/api` function is essential (See Listing 4.2). Moreover, it is paramount to understand that the intention of the JavaScript client is not to function as a primary client for the thesis sharing system, but only as a tool for illustration/testing purposes. While the respective Java projects in the sharing system's implementation contain their own unit tests to make sure everything is working as intended, the JavaScript client introduces another dimension of testing with more focus on visibility. By providing a JavaScript client such as illustrated, it becomes

easier to show people outside the computer science domain that the sharing system is indeed working as intended.

Chapter 4

Experimental Results

The following sections will demonstrate the use of the thesis sharing system to verify that it is working as intended. While the sharing system consists of multiple components, it makes the most sense to divide the testing into two primary categories, namely the cryptography tests and the web service tests. In the case of the cryptography testing, it makes sense because it is such an essential part of the sharing system. If any crypto parts are incorrect, the entire sharing system is incorrect. On the other hand, by testing the web service, which is the primary entry point for the sharing system, all components of the sharing system is consequently tested. This is because the web service is dependent upon all the other components to work properly.

4.1 Cryptography Tests

The following will test the functionality related to the cryptography part of the thesis sharing system. In other words, these tests' intention is to make sure that the implementation of the AFGH proxy re-encryption scheme is correct.

To reiterate from the previous chapter, Listing 3.1 shows an overview of the primary methods implemented in the cryptography project. It makes sense to divide the cryptography tests into three groups, namely *key generation*, *encryption/re-encryption* and *decryption*.

4.1.1 Key Generation

There are three different key generation functions; `newSecretKey()`, `newPublicKey()` and `newReEncryptionKey()`. The secret key generation function takes no parameters. This is the entry point of the key generation process. To elaborate, both the public key generation function and the re-encryption key generation function takes a secret key parameter. This means that every user will need to generate a secret key before they can proceed to generate any of the other two key types. The public key generation function generates a public key corresponding to the parameter given secret key. This is similar to how traditional asymmetric cryptosystems work. However, the re-encryption key generation differs slightly; it takes the *source* secret key and the *destination* public key as its parameters. This makes sense in that its purpose is to re-encrypt ciphertext that originally resulted from encryption with the source's public key.

SK_A

```
M0Aef3i5CCubSLJp5ed5lnzR+vY=
```

SK_B

```
D7rQ4RZbdLOn9LKLy73CIBqKwkY=
```

The above shows examples of secret keys generated by the AFGH algorithm. As shown, these are rather short strings encoded in the Base64 format. By using SK_A and SK_B respectively as arguments to the public key generation function, it produces the following two corresponding public keys.

PK_A

```
e20WEYxKEN+NfJaDmTr0cwub4m85jpWmqOBgJfk9avme9XD5i7TFsho6o5flLbTODKKRdSQuLKI6g
OLHyzAuwX0JqcEH60FqclyU6jHTaN6IvMCMeHyK5xbRc0as8lrOA0tdGTrW6CeLmL+UGdVtFfHRmL
wfIG8iwZl2uBkuW08=
```

PK_B

```
WJtMhYXIzH+H41xXyPtsXrD/Vr31HXvztvklZTXTP2b/eLTCeFUYe45D/j5SP1/KDz1IWFUFUrzIj
WzWhbWzkSf33LDm5pfgWFmroYpWIiEJAvSnzTn6PM9krppp4uW/eTw20fxW4aYoc2O6nFeRLsw+oS
cy/AkonefeDtUvOsk=
```

These are also Base64 encoded strings, but as shown, they are rather long. By using SK_A and PK_B as arguments to the re-encryption key generation function, it produces the following re-encryption key.

$RK_{A \to B}$

```
XbQn4Dy0V0EIgkpkHjvLXE9LHPVpNS+myL86VoIT8Gw/03Yj9zCcGJTO3XWQzul4aFg+itv6iwLbP
uN4Tz5dZaWGNTBVGUt8X2ecsBEk4QbhnzvVC0KKV7pCm2+CZADuIUogzL+KsqIEuvpzIuXIPv8Qyj
HfF0wYx0y3Mx6wGls=
```

Again, this is a rather long Base64 encoded string. This encoding enables the system to work with strings exclusively, without having to depend on more particular parameter types. When all parameters are of the `String` type, compatibility across different projects and platforms is a fact.

4.1.2 Encryption/Re-Encryption

When it comes to encryption/re-encryption capabilities, there are three different functions. These are `encrypt()`, `encryptReEncryptable()` and `reEncrypt()`. In Chapter 2, when introducing the AFGH scheme, it was illustrated how there are two different levels of encryption present in the scheme. Messages encrypted in the first level resembles more traditional asymmetric encryption in that it is only possible to decrypt the ciphertext if in possession of the secret key corresponding to the public key used during encryption. The `encrypt()` function encrypts a message in the first level.

Messages encrypted in the second level, by the `encryptReEncryptable()` function, is slightly different. Second level ciphertexts supports re-encryption into first level ciphertexts. In other words, while it is possible to decrypt second level ciphertexts as is by the source's secret key, it is also possible to re-encrypt it into a first level ciphertext, which supports decryption by an arbitrary destination's secret key. The `reEncrypt()` function is capable of re-encryption, i.e. converting a second level ciphertext for Alice into a first level ciphertext for Bob. In practice, a proxy instance typically perform the re-encryption operation after user interaction, but the proxy instance has no means to reveal the underlying plaintext in the process. As mentioned several times in the paper by now, this is the genius of proxy re-encryption.

Note that when the following text references the keys SK_A, SK_B, PK_A, PK_B and $RK_{A \to B}$, these are in fact references to the keys generated in the previous section on *key generation*.

M

```
Meet me at the docks at 5 PM
```

The above shows an example message M that Alice wants to keep encrypted somewhere on the Internet. Alice may choose two different ways of encrypting the message, depending on whether she wants to share the message with someone else down the line. If Alice wants to keep the message for herself, first level encryption will suffice. However, if Alice wants to share the message with others through a proxy instance, second level encryption is required. The latter choice makes more sense when examining the message contents.

C_{A1}

```
h0fOdw77InT42YtSjyk9BxOhztbp4WrQR8SiNiAA880rXqu9RYI7bb05bD1RFUf2jDiwgOOBnuw74
DYQ0WoZdQKXY7QPDJmpXLJ6mhMJWwjK5rEKWWXUChQ5jUxXRjHsSpYun2XKyWL4ye19zyZcSTz1om
jaPewk5VQbCRRsq/hsxdXeHTyiTXgExQJMyg9pi3YgIAGrp7LU0NDBmp/aB+iaOQdeVVXvjd4A83A
C8IgYYn9qU/zo38qVd38g94CYRirWYwUvkbg1U3P/7BW/dgJefAWGgy9HIDqnuo5xWBAKPPTov2JB
HNeUefopI/2xWfZrzvXul1VZ42jPZA5qgA==
```

The above shows the first level ciphertext that results from inputting the message M and PK_A into the `encrypt()` function. On the other hand, the second level ciphertext resulting from inputting the same two parameters into the `encryptReEncryptable()` function follows below.

C_{A2}

```
opX5TBoVfU4LbaHeqDwixSObGn359t5Un7ezDwR2jl3gJArm82/Wy+GPqnrxVzsBs7JVtCTHgRoLd
PIp3i64eTSaIdu0/XVIa2Fz5EaC6gbGRL/MsLVAEjSVG4oP3XlUPClmLZnnKo76nQTpdUOlFh1ARy
2w/JCfos60t+1pP4qCtT77a4tHacKRh8WpmpdASItLrcf/Dn/88Ukp07T/wL44Axf9uhCLwTyt+wU
3fqVGlRXZu3/oOwUPqVEKxU/KoxTllbOV8F52DkwevZ/XvJs7K1byQc+Xb5TdhdR8wKYvUxoX+ooL
NxrJsQXGRh6Ujel51vdGborib8PTOYtjIQ==
```

At this point, only Alice is able to decrypt the ciphertexts with her own secret key. However, a proxy instance may now input C_{A2} and the re-encryption key $RK_{A\rightarrow B}$ into the `reEncrypt()` function and arrive at an entirely new ciphertext as shown below.

C_{B1}

```
opX5TBoVfU4LbaHeqDwixSObGn359t5Un7ezDwR2jl3gJArm82/Wy+GPqnrxVzsBs7JVtCTHgRoLd
PIp3i64eTSaIdu0/XVIa2Fz5EaC6gbGRL/MsLVAEjSVG4oP3XlUPClmLZnnKo76nQTpdUOlFh1ARy
2w/JCfos60t+1pP4ojrwDHY1pUJTuuRJCIylZVP1aLemdE+DFuq+xPik6JdKoR+G1wqWYGzfIBKK3
iwe/WaRqb/FzZXrWlaoRYhlHKMvcHAVqmdyMn6s0gHNu0jSS3S1fyQ1YK8/nt46j+KNK/i/x0mIDc
Xs8IzFgUsyQwAmzRVJZaWpw0j/TgPUUbJA==
```

Note that the new ciphertext is actually a first level ciphertext now, transformed from a second level ciphertext. In the hands of Bob and his secret key, this ciphertext will reveal the message to him upon *decryption*. A proof of this follows in the next section. It is essential to note that the re-encryption key used by the proxy instance origins from Alice. In other words, the proxy instance is not able to delegate decryption rights without the necessary re-encryption key that origins from the original message source.

4.1.3 Decryption

In the scheme's decryption category, there are two different functions. These are `decrypt()` and `decryptReEncryptable()`. The `decrypt()` function is capable of decrypting first level ciphertexts while the `decryptReEncryptable()` is capable of decrypting second level ciphertexts.

From the previous section, we have three different ciphertexts C_{A1}, C_{A2} and C_{B1}. As indicated, two of these ciphertexts are first level ciphertexts while one is a second level ciphertext. If everything is working as intended and in accordance to the AFGH proxy re-encryption scheme, it should be possible for Alice to decrypt ciphertexts C_{A1} and C_{A2} with her secret key SK_A. Moreover, Bob should simultaneously be able to decrypt C_{B1} with his secret key SK_B.

The following shows the result when inputting C_{A1} and SK_A into the `decrypt()` function. If successful, the result should be human readable plaintext.

$P_{A1} = M$

```
Meet me at the docks at 5 PM
```

As shown, the decryption reveals the plaintext and the message is "Meet me at the docks at 5 PM". Now that Alice's first level decryption was successful, it is time to attempt decryption when inputting C_{A2} and SK_A into the `decryptReEncryptable()` function. The result follows below.

$P_{A2} = M$

```
Meet me at the docks at 5 PM
```

As shown, the decryption reveals the plaintext again, and the message is still the same. Now that Alice's second level decryption was successful, it is time for Bob to attempt decryption of

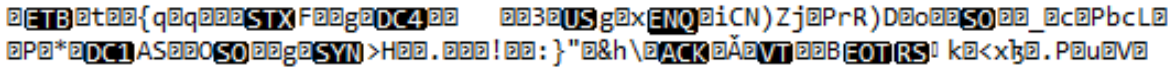


Figure 4.1: Decryption output when supplying incorrect secret key

the re-encrypted ciphertext. In other words, what happens when inputting C_{B1} and SK_B into the `decrypt()` function. The result follows below.

$P_{B1} = M$

```
Meet me at the docks at 5 PM
```

Again, the decryption is successful and the decryption reveals the plaintext to Bob. For completeness' sake, Figure 4.1 shows the decryption output when inputting C_{B1} and SK_C = `ckRPohDmX2BHDn/bQ4USK0ZtsXE=` into the `decrypt()` function.

As shown, the decryption did not work with the incorrect decryption key. From this, it is safe to assume that the AFGH proxy re-encryption implementation is correct. Please note that while the above tests applies to a very short message, the implementation is in fact applicable to arbitrary length messages, including file content.

4.2 Web Service Tests

The following will test the web service API, more specifically make sure that all the functions exposed by the web service is working as intended. Note that the underlying implementations of these functions are dependent upon the entire thesis sharing system's components. Because of this, the following tests will not only test the web service itself, but also all of its referenced components.

The web service tests will be performed in a per function manner, meaning that every function exposed by the web service API will be tested separately. As explained in Chapter 3, all web service functions return a JSON representation of a `WebServiceResponse` object. Listing 4.1 shows the skeleton of the `WebServiceResponse` class. The `status` variable will represent a HTTP status code while the `message` variable will contain a short message describing the outcome of the function call. Finally, the `content` variable will contain the result of the function call, and does not belong to a particular datatype or class.

```java
public class WebServiceResponse {
        private int status;
        private String message;
        private Object content;
}
```

Listing 4.1: WebServiceResponse.java

It makes sense to divide the API functions into two groups, namely HTTP GET functions and HTTP POST functions. The functions exposed as HTTP GET functions are those that take no parameters. On the other hand, functions that *do* take parameters are reachable through HTTP POST requests.

While the API exposes ten functions overall, only two of these are HTTP GET functions, more specifically `/api` and `/newSecretKey`. In other words, the remaining eight functions all take parameters to be able to produce the result.

4.2.1 HTTP GET Functions

By calling the `/api` function as a HTTP GET request, the JSON object shown in Listing 4.2 is returned by the web service. Admittedly, this object is rather large as it contains information about all the functions exposed by the web service API.

```json
{
  "status": 200,
  "message": "The API – All valid function calls return a JSON object",
  "content": [
    {
      "function": "announce",
      "method": "POST",
      "details": "Call this function to manually announce a torrent on the
                  system's tracker.",
      "params": [
        "file"
      ],
```

```
13        "args": [
14          "binary"
15        ],
16        "returns": "The torrent's new unique file name."
17      },
18      {
19        "function": "api",
20        "method": "GET",
21        "details": "Calling this function gives an overview of the API."
22      },
23      {
24        "function": "decrypt",
25        "method": "POST",
26        "details": "Calling this function attempts to decrypt the given ciphertext
27                    with the given secret key.",
28        "params": [
29          "ciphertext",
30          "secretKey"
31        ],
32        "args": [
33          "base64",
34          "base64"
35        ],
36        "returns": "The resulting plaintext."
37      },
38      {
39        "function": "download",
40        "method": "POST",
41        "details": "Call this function to download an encrypted torrent. The public key
42                    refers to the destination's public key and must reflect a
43                    corresponding call to the share function.",
44        "params": [
45          "fileName",
46          "publicKey"
47        ],
48        "args": [
```

```
49          "string",
50          "base64"
51        ],
52        "returns": "The torrent file encrypted under the given public key and
53                    encoded as a Base64 string."
54      },
55      {
56        "function": "newPublicKey",
57        "method": "POST",
58        "details": "Calling this function generates a new public key.",
59        "params": [
60          "secretKey"
61        ],
62        "args": [
63          "base64"
64        ],
65        "returns": "The new public key."
66      },
67      {
68        "function": "newReEncryptionKey",
69        "method": "POST",
70        "details": "Calling this function generates a new re-encryption key.",
71        "params": [
72          "secretKey",
73          "publicKey"
74        ],
75        "args": [
76          "base64",
77          "base64"
78        ],
79        "returns": "The new re-encryption key."
80      },
81      {
82        "function": "newSecretKey",
83        "method": "GET",
84        "details": "Calling this function generates a new secret key.",
```

```
85        "returns": "The new secret key."
86      },
87      {
88        "function": "newTorrent",
89        "method": "POST",
90        "details": "Call this function to generate a new torrent for the given file.
91                    (Note: nothing is stored on the server)",
92        "params": [
93          "file"
94        ],
95        "args": [
96          "binary"
97        ],
98        "returns": "A new torrent file encoded as a Base64 string."
99      },
100     {
101       "function": "share",
102       "method": "POST",
103       "details": "Call this function to share a torrent with the given public key
104                   holder. The corresponding re-encryption key should derive from the
105                   source's secret key and the destination's public key.",
106       "params": [
107         "fileName",
108         "publicKey",
109         "reEncryptionKey"
110       ],
111       "args": [
112         "string",
113         "base64",
114         "base64"
115       ],
116       "returns": "True if share was successful, false otherwise."
117     },
118     {
119       "function": "upload",
120       "method": "POST",
```

```
121       "details": "Call this function to upload a torrent file. The torrent
122                   will be encrypted with the given public key.",
123       "params": [
124         "file",
125         "publicKey"
126       ],
127       "args": [
128         "binary",
129         "base64"
130       ],
131       "returns": "The uploaded torrent's new unique file name."
132     }
133   ]
134 }
```

Listing 4.2: JSON Object Response - /api

Note that the `content` variable in the above JSON object result is in fact an array of `ApiItem` objects, as shown in Listing 4.3. While some might argue the result from calling `/api` is cumbersome and excessive, the truth is that because it is a JSON object it is in fact exceptionally easy to handle by the invoker. There exists very good tools for parsing JSON objects out there, so managing the object in Listing 4.2 is no problem.

```
1 public class ApiItem implements Comparable<ApiItem> {
2         private String function;
3         private String method;
4         private String details;
5         private String[] params;
6         private String[] args;
7         private String returns;
8 }
```

Listing 4.3: ApiItem.java

Similarly, by calling the `/newSecretKey` function as a HTTP GET request, the JSON object shown in Listing 4.4 is returned by the web service. This object is much smaller than the object

returned from `/api`, as its `content` variable simply represents a system generated secret key.

```
{
  "status": 200,
  "message": "Secret key generated",
  "content": "fyI+J+LgyTwxHEwrYvDEnMyZrlM="
}
```

Listing 4.4: JSON Object Response - /newSecretKey

4.2.2 HTTP POST Functions

The following tests work by invoking HTTP POST requests to the web service as multipart/form-data. This encoding type is required when sending files as parameters. Note that all the HTTP POST requests are assumed multipart/form-data requests in the web service's source code.

Note that some of the following API function tests take cryptographic parameters such as secret keys, public keys, ciphertext etc. Because of this and in the spirit of reducing paper redundancy, the following references to SK_A, PK_A, SK_B, PK_B, $RK_{A\rightarrow B}$ and similar variables all refer to the values with the same name introduced/generated in the Cryptography Tests section.

By calling the `/newTorrent` function with the parameter `file` set to an arbitrary file such as a document or picture, the JSON object shown in Listing 4.5 is returned by the web service.

```
{
  "status": 200,
  "message": "New torrent generated",
  "content": "ZDg6YW5ub3VuY2UzMTpodHRwczovLzEwLjAuNS45MDo2OTY5L2Fubm91bmNlMTA6Y3JlYX
              RlZCBieTM3Om5vLnVpcy5tc2FsdGUudGhlc2lzLmJpdF90b3JyZW50LnV0aWwxMzpjcmVh
              dGlvbiBkYXRlaTE0MzEwODA0MzRlNDppbmZvZDY6YXRva2VuMTQ6c2VjdXJpdHlfdG9rZW
              42Omxlbmd0aGkyMTAyNmU0Om5hbWU5OjdGZUl3LmpwZzEyOnBpZWNlIGxlbmd0aGkyNTYw
              MDBlNjpwaWVjZXMyMDqsZpgwyocXSlB75FPdwiAKokYCDc6cHJpdmF0ZWkxZWVl"
}
```

Listing 4.5: JSON Object Response - /newTorrent

The `content` variable in the above result is a Base64 encoding of a torrent/metadata file corresponding to the contents of the `file` parameter set in the HTTP request. Any programming language can easily convert this encoded string to a .torrent file for further use. For convenience, Listing 4.6 shows the decoded version of this string. We can see that this is a valid Bencoded torrent.

```
"d8:announce31:https://10.0.5.90:6969/announce10:created by37:no.uis.msalte.thesis.
bit_torrent.util13:creationdatei1430756428e4:infod6:atoken14:security_token6:
lengthi76e4:name8:test.txt12:piece lengthi256000e6:pieces20:<snip>:7:privatei1eee"
```

Listing 4.6: System Generated Torrent File

By calling the `/announce` function with the parameter `file` set to a valid torrent file, the JSON object shown in Listing 4.7 is returned by the web service.

```
{
  "status": 200,
  "message": "Torrent announced on the tracker",
  "content": "e3e7f325-31f7-4d05-97b2-b8fb5f574ad1.torrent"
}
```

Listing 4.7: JSON Object Response - /announce

As the `message` variable in the above result indicates, this function call resulted in the sharing system's BitTorrent tracker to start announcing this torrent. The `content` variable represents the torrent's id, which is equal to the torrent's file name. Note that this function only tells the tracker to start announcing the torrent. Other than that, it does not interact with the underlying sharing system in any capacity. The API's upload function is the go-to function for introducing a new torrent to the sharing system. The purpose of the announce function is to manually tell the tracker to start announcing a torrent. This will be necessary after application restart.

By calling the `/newPublicKey` function with the parameter `secretKey` set to SK_A, the JSON object shown in Listing 4.8 is returned by the web service.

```
{
  "status": 200,
  "message": "Public key generated",
  "content": "YGuRNXMkslbnb8+TMR2ZKlT16Gx0vdSVDnRk52RJ4QrzjJ2RuztQFlBQc1rEIHf6mm
              EsWKR3bpHmoWStbWPpgPVj4Iq946gyA7pJQX4UfVYwlJbYOPI6upbMUcwxxqWljNSp
              v/2F2CW6N25q7M9U9NenAEcnNpCDZOQZk8A38="
}
```

Listing 4.8: JSON Object Response - /newPublicKey

The `message` variable in the above result explains the outcome of this function call rather well. The `content` variable contains the public key corresponding to the parameter given secret key. Note that the `content` variable is equal to PK_A. This makes sense, as there is always a single public key corresponding to a single secret key.

By calling the `/newReEncryptionKey` function with the parameter `secretKey` set to SK_A and the parameter `publicKey` set to PK_B, the JSON object shown in Listing 4.9 is returned by the web service.

```
{
  "status": 200,
  "message": "Re-encryption key generated",
  "content": "P6PrrTuwAqnfovO5Zf12Iolj8z4OxLe5IZkX1y0p3lTuPzMK04NJcNqmft35nYp7EK4m6CwaW
              FC6RWfkNf3fZg2rdbZEnYwhdNEnoSxLvfkkydF1lcSk5mce0WNlGqY43nFgPO6crpsg/BJZdx
              TC+Ju/QWp0jZAzbQbyvg8d/Y="
}
```

Listing 4.9: JSON Object Response - /newReEncryptionKey

Again, the `message` variable in the above result explains the outcome of this function call rather well. The `content` variable contains the re-encryption key corresponding to source SK_A and destination PK_B. Note that the `content` variable is equal to $RK_{A \rightarrow B}$. Again, this makes sense, as there is always a single re-encryption key applicable for re-encryption from party A to party B.

By calling the `/upload` function with the parameter `file` set to a valid torrent file and the

parameter `publicKey` set to PK_A, the JSON object shown in Listing 4.10 is returned by the web service.

```
{
  "status": 200,
  "message": "Torrent uploaded successfuly",
  "content": "e3e7f325-31f7-4d05-97b2-b8fb5f574ad1.torrent"
}
```

Listing 4.10: JSON Object Response - /upload

Again, the `message` variable in the above result explains the outcome of this function call rather well, the upload was successful. The `content` variable reflects the uniquely generated torrent id in the underlying sharing system. Note that this function call initiates a rather complex procedure. Firstly, the sharing system announces the torrent on its embedded BitTorrent tracker before it encrypts the torrent with the PK_A parameter. The resulting ciphertext is then stored in the system's key-value store/database *torrents* where the torrent id is the *key* and the ciphertext is the *value*. It is also worth noting that the caller will need to write down the generated torrent id somewhere, as it is the only way of identifying this particular torrent in the sharing system. As we shall see, the next function test (`/share`) will take the torrent id as one of its parameters.

By calling the `/share` function with the parameter `fileName` set to `"e3e7f325-31f7-4d05-97b2-b8fb5f574ad1.torrent"`, the parameter `publicKey` set to PK_B and the parameter `reEncryptionKey` set to $RK_{A \rightarrow B}$, the JSON object shown in Listing 4.11 is returned by the web service.

```
{
  "status": 200,
  "message": "Torrent e3e7f325-31f7-4d05-97b2-b8fb5f574ad1.torrent shared",
  "content": "true"
}
```

Listing 4.11: JSON Object Response - /share

Again, the `message` variable in the above result explains the outcome of this function call

rather well. Note that this function call will share the torrent with id equal to the `fileName` parameter with the public key holder of PK_B. The $RK_{A \to B}$ parameter is used by the sharing system to grant this access request. This is achieved by storing a record in the key-value store/database *shares* where the torrent id is the *key* and the key tuple (PK_B, $RK_{A \to B}$) is the *value*. If a torrent matching this particular id is already present in the database, the procedure will simply append the key tuple to the current value, indicating that more than one user is currently on the receiving end of this torrent.

By calling the `/download` function with the parameter `fileName` set to `"e3e7f325-31f7-4d05-97b2-b8fb5f574ad1.torrent"` and the parameter `publicKey` set to PK_B, the JSON object shown in Listing 4.12 is returned by the web service.


```
{
  "status": 200,
  "message": "Download granted",
  "content": "BexCIhmyW5v9C5iiUQ6ESSn42gP/0JhGMADFawR6SqiDLN076Y1iUmHt85p7vXeevOQdp3sM
              mDTZKC0gyauDZ0ZzSbuCkk7dxHlqf7er+FaDDkFdeYYg9A5cZNOpS2DU0foNIri4mNnCjsQ2
              JkpspqBtaTlEJn+56jjngAroyTITVy/KuqdfCzDOIxIGm6MTEX5AoDoaSE3Jy8GEJae7Sdgf
              wqQ6um8KCb1TGKoQymOQoINxzitMWuIvL+u8g/CGOC52JiOmfb0alJdHBEXlVCutAYy1+HZ8
              5Rho4zcy+DgAKvNZ90spMBUscNKoqX38cagyNOyqOnX04+tGGLwbbTVXpnosIjZR1PemPqSa
              FM6Ejq8zRjsGr9cBGlNLxp5ly1otNlHozY8KcReRSTnmDIM6wQxwla8gSbpLAugTqSkTxGHx
              pF/69oI/EpwDbw9NJB4hzOUnBrlgsdIwe0dPxePsaTOM5NMIqM2a6w1FHZk0/JwbKBD6dtHm
              NAg5yvPpdkHKr/ikRJrfCfwp4Drg0vh1mmG4WqjcSrQCrLcCqfCk8j15EpBhFrGU0zrZLU83
              /ONQExK7IQ1HKC4blpleQzdNMGcqirmkyH9mbK4CFNU6XannXOaHrYhOoEi99T2nEKzRhgcq
              na31RE+uKNBirn/neUZbdHc8gVXnPprgvbk="
}
```


Listing 4.12: JSON Object Response - /download

Again, the `message` variable in the above result explains the outcome of this function rather well. The `content` variable however, represents the re-encrypted torrent ciphertext. To clarify, this ciphertext is only compatible for decryption with the secret key SK_B. This is the result of the sharing system's proxy re-encryption capabilities. Upon calling `/download`, the sharing system looked up the parameter given torrent id in its database and found the re-encryption key $RK_{A \to B}$ in the key tuple identified by PK_B.

```
{
  "status": 400,
  "message": "No access or file does not exist"
}
```

Listing 4.13: JSON Object Response - /download - Invalid Parameters

Listing 4.13 shows the JSON object returned by the web service when calling `/download` and the sharing system does not find a database record corresponding to either of the parameters.

By calling the `/decrypt` function with the parameter `ciphertext` set to the `content` variable from Listing 4.12 and the parameter `secretKey` set to SK_B, the JSON object shown in Listing 4.14 is returned by the web service.

```
{
  "status": 200,
  "message": "Decryption successful",
  "content": "ZDg6YW5ub3VuY2UzMTpodHRwczovLzEwLjAuNS45MDo2OTY5L2Fubm91bmNlMTA6Y3JlYXR
              lZCBieTM3Om5vLnVpcy5tc2FsdGUudGhlc2lzLmJpdF90b3JyZW50LnV0aWwxMzpjcmVhdG
              lvbiBkYXRlaTE0MzA3NTY0MjhlNDppbmZvZDY6YXRva2VuMTQ6c2VjdXJpdHlfdG9rZW42O
              mxlbmd0aGk3NmU0Om5hbWU4OnRlc3QudHh0MTI6cGllY2UgbGVuZ3RoaTI1NjAwMGU2OnBp
              ZWNlczIwOpvaUx2lW9otjUUQjJNJzSbIbvk6Nzpwcml2YXRlaTFlZWU="
}
```

Listing 4.14: JSON Object Response - /decrypt

The `message` variable indicates that the decryption was successful. This means that the `content` variable now reflects the initial torrent file as Base64 encoded plaintext. Decoding this string reveals the torrent shown in Listing 4.15. As we can see, this is the exact same Bencoded torrent as the one in Listing 4.6.

```
"d8:announce31:https://10.0.5.90:6969/announce10:created by37:no.uis.msalte.thesis.
bit_torrent.util13:creationdatei1430756428e4:infod6:atoken14:security_token6:
lengthi76e4:name8:test.txt12:piece lengthi256000e6:pieces20:<snip>:7:privatei1eee"
```

Listing 4.15: Decrypted Ciphertext

To conclude the above tests, it is apparent that the sharing system is working as intended. Party A is able to share a file with party B exclusively by calling the functions exposed by the web service API. This goes to show how easy it is to incorporate the sharing system into existing applications. Any programming language is perfectly capable of invoking these function calls and parsing/interpreting the resulting JSON objects.

Chapter 5

Discussion

The following section will primarily make a short analysis over what the test results from the previous chapter mean. There will also be a brief comparison of the thesis sharing system against other sharing practices in use today. Finally, the text will present a short discussion of the thesis sharing system's limitations and shortcomings.

As demonstrated, the thesis sharing system is fully capable of functioning as a file sharing system. As we have seen, the system's proxy re-cryptography capabilities is able to provide the rest of the system with a secure encryption mechanism. Moreover, the web service API exposes enough functionality such that existing applications may easily incorporate the system to provide their users with secure sharing.

When comparing the sharing system with other practices for sharing that is commonly in use today, this work is able to provide a major advantage. This advantage is specifically the system's ability to re-encrypt ciphertexts without accessing the underlying plaintext. Because of the system's implementation of proxy re-encryption, the sharing system is able to modify encrypted ciphertexts so that they are compatible for decryption by other keys. All this can happen without security concern, as the system exclusively operate on publicly available information. To elaborate on the reasoning for the above claim, we can see that the thesis sharing system eliminates the primary concern of many other sharing systems in use today where re-encryption must follow a preceding decryption to plaintext. Decrypting content before re-encryption is a huge security concern that the thesis sharing system is able to avoid.

Moreover, as the thesis sharing system uses the proxy re-encryption primitive, it only needs

to store a single ciphertext per file. This is opposed to other sharing systems where it is necessary to store one ciphertext per user on the receiving end of a file sharing operation, i.e. one ciphertext per decryption key. In the thesis sharing system, the underlying database only contain the original ciphertext encrypted by the file's source. Upon sharing, the sharing system will modify this single ciphertext for compatibility with the destination's decryption key. This conversion is accomplished by applying the publicly available re-encryption key to the ciphertext. There is no necessity for the sharing system to store the resulting ciphertext anywhere, just deliver it to the recipient. Exclusively storing the original ciphertext is satisfactory. It is also worth noting that the thesis sharing system is able to delegate access while the original file source is offline, as long as it has previously been supplied with the required re-encryption key, i.e. if the file source invoked the `share()` function at some point.

On the other hand, while the thesis sharing system is able to introduce an improved sharing mechanism, it will not be able to eliminate all security concerns for sharing files on the Internet. If the users are hosting their files on compromised locations, whether it is on cloud storage services or otherwise, the security algorithms in use on these locations will obviously affect the security of the files taking part in the sharing system. However, as mentioned, the thesis sharing system makes significant improvements on many of today's sharing practices and is thereby a big step in the right direction.

Additionally, the thesis sharing system is not compatible with all available BitTorrent clients. Compatible clients must as mentioned implement the Vuze specification for Message Stream Encryption (MSE).

When it comes to key management, the sharing system does not offer anything in this area. When users have generated their respective key pairs for use in the sharing system, these keys will remain their keys indefinitely. There is no system in place for key renewal. However, as this was not in the thesis' scope and because there are several such systems available as open source out there, this particular limitation is rather negligible.

While considering the sharing system's limitations, it is still able to make significant security improvements compared with many of today's practices. It *does* eliminate some of the most obvious weaknesses, but not all.

Chapter 6

Conclusion

The following will primarily compare the thesis' original objectives with the results and discussion. Moreover, it will describe some recommendations for future work.

To reiterate the thesis objectives from Chapter 1.2, the following shows them again in no particular order. Develop a file sharing system that:

1. Satisfies today's security requirements
2. Handles arbitrary file types and sizes
3. Enables file owners to host their files on arbitrary locations
4. Does not require file owners to remain online indefinitely
5. Can easily be incorporated into existing applications
6. Works on all platforms

From the above objectives, and when comparing them to the resulting sharing system's capabilities, it can be argued that all the initial objectives has been met. The thesis sharing system is able to provide a secure sharing mechanism that handles arbitrary file types and sizes. It also enables file owners to host their files on arbitrary locations, albeit this initially attractive property also has the side effect of being a potential security concern, depending on the file owner's particular file system location.

Since the thesis sharing system relies on the (modified) BitTorrent protocol for file transfer capability, the sharing system does not require file owners to remain online. Users can join and leave the system as they please, conveniently continuing their progress from where they left off.

Because the sharing system is exposed through a standardized web service API, it can easily be incorporated into existing applications while also remaining platform independent. All platforms have easily available tools for invoking HTTP requests and parsing JSON responses.

One of the primary weaknesses of the thesis sharing system is the fact that is *only* that, a *sharing* system. The overall system's security is heavily dependent on the underlying security of the files taking part in the sharing processes. While the thesis sharing system *does* provide secure sharing, it does not interfere with the underlying file systems. This means that if file owners host their files on insecure locations, the fact that he or she is able to share files securely becomes second rate. Because of this, if a future extension of the thesis sharing system included its own storage entity, it might be possible to provide a completely secure cloud storage system.

Because of the sharing system's nature, an incorporation into a cloud storage service, like any other application, is arguably one of its primary intentional use cases. As such, incorporating the sharing system in a cloud storage service seems like an obvious suggestion for future work. As the paper mentions frequently, we are seeing a huge increase in users moving their private files from their own expensive hardware to inexpensive cloud storage services. Again, while this is very convenient, there are rather big security questions remaining unanswered in many of these cases.

Other areas of the thesis sharing system that would benefit from future work is its key management capability and the BitTorrent client support. Currently, there is no key renewal system in place, which some might argue is rather inconvenient. While this was never part of the thesis scope, it might be a nice convenience to have in the future. When talking about BitTorrent client support, there are two ways to go. On one hand, it would be nice to be able to support arbitrary BitTorrent clients, not only those implementing the Vuze MSE specification, as is the case now. However, on the other hand, by implementing a custom BitTorrent client as an exclusive part of the sharing system, one would gain significantly more control over client behavior. This would of course lead to a more narrow system, in which the users must use a specific BitTorrent client to be able to partake. In any case, a close examination of this tradeoff would be necessary.

Finally, the sharing system would benefit greatly in the convenience department if it were to support identity-based cryptography and/or group sharing. Identity-based cryptosystems are systems in which its users are able to use personal information such as e-mail address or IP address as their public key component. Whether this is possible in the current proxy re-encryption scheme would need to be found out. Group sharing would potentially eliminate the current necessity to invoke the sharing procedure once per recipient user, and instead grant the ability to share with a large group of users at a time.

Appendix A

Acronyms

API Application Programming Interface

BT BitTorrent

CSP Cloud Storage Provider

CSSP Cloud Storage Service Provider

DLP Discrete Logarithm Problem

ECC Elliptic Curve Cryptography

HTTP Hypertext Transfer Protocol

ISP Internet Service Provider

JSON JavaScript Object Notation

P2P Peer to Peer

PBC Pairing Based Cryptography

PRE Proxy Re-Encryption

SSL Secure Sockets Layer

TLS Transport Layer Security

Appendix B

Modifications to ttorrent

The following will briefly summarize the changes made to the ttorrent library. Again, note that the ttorrent library is licensed under the Apache Software License version 2.0. Every modified Java class will have its complete location (including package) shown here. That way, it will be easy to track down modified code.

com.turn.ttorrent.common.Torrent -

- Added an additional method signature for `create()` that takes a security token parameter.
- Modified the private method `create()` to support the aforementioned security token parameter. This method will append the security token to a torrent's info dictionary. Moreover, it will also set its private flag to '1'.

com.turn.ttorrent.tracker.Tracker -

- The method `getAnnounceUrl()` will append 'https' to the returned string, instead of 'http'.

com.turn.ttorrent.tracker.Tracker.TrackerThread -

- This inner class' call to `connection.connect()` includes a reference to `no.uis.msalte.thesis.common.security.SecurityConstants.SSL_CONTEXT`. This is a static reference to a `javax.net.ssl.SSLContext` object.

com.turn.ttorrent.tracker.TrackerService -

- The method `process()` makes sure incoming HTTP GET requests include the 'requirecrypto' flag and that it is set to '1'. If this is not the case, HTTP status 406 Not Acceptable is returned.

com.turn.ttorrent.client.announce.HTTPTrackerClient -

- Note that this particular modification does not affect the thesis sharing system directly. The reason for the following change was primarily for testing purposes (to be able to examine BitTorrent client behavior during debugging).
- Modified the `announce()` method to support HTTPS. Now uses classes `HttpClient` and `GetMethod` from the Apache Commons library.

  HttpClient provides full support for HTTP over Secure Sockets Layer (SSL) or IETF Transport Layer Security (TLS) protocols by leveraging the Java Secure Socket Extension (JSSE) [34].

com.turn.ttorrent.common.protocol.HTTPAnnounceRequestMessage -

- The method `parse()` is modified to include parsing of the 'requirescrypto' and 'cryptoport' parameters.

Appendix C

Java Projects Setup

The following will explain how to setup the respective thesis sharing system Java projects in the Eclipse IDE on a Windows environment.

As described in the paper, the thesis sharing system consists of several Java projects. These are primarily the five projects `common`, `crypto`, `bit-torrent`, `secure-share` and `web-service`. However, the submission source code contains an additional sixth project named `ttorrent`. This is the modified version of the ttorrent library that the paper mentions frequently.

In the compressed file attached to the submission PDF file, the following folder structure is available.

app-dir/ Directory

bit-torrent/ Java Maven Project

common/ Java Maven Project

crypto/ Java Maven Project

secure-share/ Java Maven Project

ttorrent/ Java Maven Project

web-service/ Java Maven Project

As shown in the above, there are seven folders present in the compressed file. These folders represent the *main application directory* known as **app-dir/** as well as the six aforementioned

Java projects. The **app-dir/** directory contains all the files and subdirectories in use by the sharing system.

By unpacking the compressed file, it is easy to import the Java Maven projects into Eclipse. Start by selecting **File → Import... → Existing Maven Projects**. In the dialog window that appears, select the **Browse...** button next to the *Root Directory* input box. From the subsequent dialog window, find the root directory of the particular project to be imported (one of the six folders above) before pressing **Finish**. Repeat this procedure for every project successively. Because these projects are Maven projects referencing each other in their respective `pom.xml` files, the dependencies between them should primarily be working automatically. However, there is one exception. The `crypto` project needs to know about the JPBC library. The respective jar files for the JPBC library reside in the **app-dir/jars/** directory. To add these jars to the project's build path, **Right-click the crypto project → Build Path → Add External Archives...**. In the following dialog window, browse to **app-dir/jars/**, select both **jpbc-api-2.0.0.jar** and **jpbc-plaf-2.0.0.jar**, before pressing **Open**.

After importing the six projects by following the above, it is time to make some modifications. Make sure that the projects are compiling using a Java Runtime Environment (JRE) version 6 or later. Unless this is the case, their use of the `@Override` annotation will produce compilation errors.

Moreover, make sure that the `JAVA_HOME` *environment variable* is set in the operating system. This variable should point to the parent directory containing the Java Runtime Environment (JRE). A sample location for the JRE on a Windows system is "C:\Program Files\Java\jre1.8.0_20". After setting the `JAVA_HOME` environment variable, append the bin directory at "<JAVA_HOME>/bin" to the `PATH` environment variable.

After setting/updating the environment variables, it is time to generate a *server certificate* to enable the sharing system's HTTPS capability. To do this, we will use a tool known as the Java *keytool* [35]. This is an application located at "<JAVA_HOME>/bin/keytool.exe" on Windows systems.

Open a command line in *administrator mode* and execute the following commands. Remember to substitute "<pw>" with your desired password and "<JAVA_HOME>" with your `JAVA_HOME` environment variable.

```
<JAVA_HOME>\bin\keytool -genkey -alias secure-share -keyalg RSA -keypass <pw>
                        -storepass <pw> -keystore secure-share.jks

<JAVA_HOME>\bin\keytool -export -alias secure-share -storepass <pw>
                        -file secure-share.cer -keystore secure-share.jks

<JAVA_HOME>\bin\keytool -import -v -trustcacerts -alias secure-share
                        -file secure-share.cer
                        -keystore "<JAVA_HOME>\lib\security\cacerts"
                        -keypass <pw> -storepass changeit
```

After following the instructions presented by the above procedure, the resulting output files should be **secure-share.jks** and **secure-share.cer**. Move these two files into the **app-dir/tls/** directory. Edit the file **keystore.pw** present on this location to reflect the same keystore password as used in the certificate generation procedure (<pw>).

Now that we have generated the server certificate, navigate to the class `no.uis.msalte.thesis.common.AppConstants.java` in the `common` project. This class consists of several static string variables, all of which has had some documentation attached to explain their significance. The primary objective in modifying this source file is to make sure that the `DIR_APP` variable is pointing to the file system location of **app-dir/** and that the `TRACKER_HOSTNAME` variable contains the server's hostname. Other than this, it is perfectly fine to keep the default values.

When the above is complete, it is possible to test whether everything is working correctly by running the unit tests in the `web-service` project. Navigate to `no.uis.msalte.thesis.web_service.tests.AllTests.java` and select **Run → Run As → JUnit Test**. If all three tests pass, everything should be working as intended.

Now that the project setup is complete, start the web service through the `main()` method in `no.uis.msalte.thesis.web-service.App.java`. When the program is running, access the web service API through https://hostname:port/api. The default port is 9090.

Bibliography

[1] S. Androutsellis-Theotokis and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologiess," *ACM Computing Surveys*, 2004.

[2] S. V. L and A. Basu, "Modified BitTorrent Protocol and its Application in Cloud Computing Environment," *International Journal of Computer Science, Engineering and Applications*, 2012.

[3] M. Blaze, G. Bleumer, and M. Strauss, "Divertible Protocols and Atomic Proxy Cryptography," 1998.

[4] A. D. Caro and V. Iovino, "jPBC: Java Pairing Based Cryptography," 2011.

[5] S. S. Chow, J. Weng, Y. Yang, and R. H. Deng, "Efficient Unidirectional Proxy Re-Encryption," 2009.

[6] B. Cohen, "Incentives Build Robustness in BitTorrent," 2003.

[7] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage," 2005.

[8] D. Boneh, "A Brief Look at Pairings Based Cryptography," *IEEE Computer Society*, 2007.

[9] B. Lynn, *On the Implementation of Pairing-based Cryptography*. PhD thesis, Stanford University, 2007.

[10] G. Neglia, G. Reina, H. Zhang, D. Towsley, A. Venkataramani, and J. Danaher, "Availability in BitTorrent Systems," 2007.

[11] M. Salte, "Secure Cloud Storage Sharing Through Proxy Re-Encryption," 2014.

[12] W. Stallings, *Cryptography and Network Security*. Pearson, sixth ed., 2014.

[13] C. Cachin, R. Guerraoui, and L. Rodgigues, *Reliable and Secure Distributed Programming*. Springer, second ed., 2011.

[14] S. Parkinson, "Secure Crypto: TLS 1.3 – A New Beginning." https://blogs.rsa.com/secure-crypto-tls-1-3-new-beginning. Accessed: 2015-03-20.

[15] N. Sullivan, "A Primer on Elliptic Curve Cryptography." http://arstechnica.com/security/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/1/. Accessed: 2015-04-12.

[16] D. C. Wilson and G. Ateniese, ""To Share or not to Share" in Client-Side Encrypted Clouds," 2014.

[17] S. Turner, "Transport Layer Security," *Internet Computing Magazine*, 2014.

[18] T. A. S. Foundation, "Apache Maven." http://maven.apache.org. Accessed: 2015-04-11.

[19] M. Petazzoni, "Ttorrent, BitTorrent library in Java." http://mpetazzoni.github.io/ttorrent. Accessed: 2015-02-11.

[20] J. Kotek, "MapDB." http://www.mapdb.org. Accessed: 2015-03-18.

[21] P. Wendel, "Spark Framework." https://www.sparkjava.com. Accessed: 2015-03-07.

[22] T. E. Foundation, "Jetty - Servlet Engine and Http Server." http://eclipse.org/jetty. Accessed: 2015-04-07.

[23] C. Rong and E. Cayirci, *Security in Wireless Ad Hoc and Sensor Networks*. Wiley, 2009.

[24] J. F. Kurose and K. W. Ross, *Computer Networking A Top-Down Approach*. Pearson, fifth ed., 2010.

[25] R. P. Grimaldi, *Discrete and Combinatorial Mathematics An Applied Introduction*. Pearson, fifth ed., 2004.

[26] Twitter, "Bootstrap - The world's most popular mobile-first and responsive front-end framework." http://getbootstrap.com. Accessed: 2015-02-17.

[27] "Git." http://git-scm.com. Accessed: 2015-01-08.

[28] "GitHub." https://github.com. Accessed: 2015-01-08.

[29] "RFC 7465 - Prohibiting RC4 Cipher Suites." https://tools.ietf.org/html/rfc7465. Accessed: 2015-05-18.

[30] "Internet Engineering Task Force (IETF)." https://www.ietf.org/. Accessed: 2015-05-18.

[31] "NICS Cryptography Library." https://www.nics.uma.es/dnunez/nics-crypto. Accessed: 2015-01-03.

[32] "GitHub Repository: NICS Cryptography Library." https://github.com/cygnusv/nics-crypto. Accessed: 2015-01-03.

[33] "The JHU-MIT Proxy Re-cryptography Library." http://spar.isi.jhu.edu/~mgreen/prl. Accessed: 2015-01-02.

[34] "HttpClient - HttpClient SSL Guide." http://hc.apache.org/httpclient-3.x/sslguide.html. Accessed: 2015-04-02.

[35] "keytool - Key and Certificate Management Tool." https://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html. Accessed: 2015-04-09.