Hermod: A File Transfer Protocol Using Noise Protocol Framework

MARKUS ÅKESSON
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Hermod: A File Transfer Protocol Using Noise Protocol Framework

Markus Åkesson
dat14mak

Department of Electrical and Information Technology
Lund University

Supervisor: Paul Stankovski Wagner
Daniel Jankovic
Stefan Chevul

Examiner: Thomas Johansson

June 12, 2020

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Transferring files between computers and servers has been an area of interest since early days of the modern computer era. With the rise of distributed computing and data centres, the interest has only grown stronger. Todays solutions for transferring files often rely on Secure File Transfer Protocol (SFTP) and Secure Shell Protocol (SSH) for doing so securely. Any vulnerabilities in the protocols would lead to a huge security gap to fill.

The newly released Noise Protocol Framework provides a promising tool for designing a new secure file transfer protocol. The framework allows for the creation of Diffie-Hellman based protocols that provides confidentiality, integrity and authentication. After a review of the security provided by SSH for SFTP, this thesis provides a proof-of-concept for a new secure file transfer protocol developed from the Noise Protocol Framework. In addition to the specification a reference implementation is provided to allow further testing and benchmarking.

The proposed protocol, Hermod, eliminates the drawbacks in the configuration procedure of SSH and SFTP servers while still providing the same security properties. Hermod also comes with improved performance compared to SFTP, especially for smaller files.

Forewords

I would like to express my gratitude towards Advenica AB for allowing me to pursue and complete this thesis. I would also like to thank my parents Stefan and Karin and my sister Johanna for their support. A special thank you is directed to Johan Petersson and Karin for proofreading my thesis.

At last I would like to extend my gratitudes to my three supervisors Daniel Jankovic, Stefan Chevul and Paul Stankovski Wagner for their support through out the thesis.

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1
Introduction

The amount of data being produced and stored on servers increase and more devices are connected to the Internet. This creates a demand for secure, fast and efficient protocols for file transfer.

Today's alternatives for performing secure file transfers depend on protocols conceived twenty years ago, and even if the protocols have received numerous updates they still carry their old legacy. The legacy can be in the form of support for insecure cryptographic primitives or extensive backwards compatibility making the implementations larger, increasing the chances for bugs or flaws. This adds a lot of complexity and makes the current protocols large and harder to evaluate.

The most common option used today, SSH File Transfer Protocol (SFTP), relies on Secure Shell (SSH) to negotiate and secure the tunnel used for the transfer. If a critical flaw or vulnerability in SSH was to be found, that would result in our most popular file transfer protocol being useless, leaving our files vulnerable and insecure during transmission over e.g. Internet. In order to minimize potential damage from a vulnerability in SSH, developing an alternative protocol for secure file transfer is needed. The new protocol would need to be separated from the SSH ecosystem while offering, at least, the same security guarantees and performance.

In recent years a new framework for building secure cryptographic protocols has been released as an alternative to SSH, Transport Layer Security (TLS) and QUIC for securing communications tunnels, the Noise Protocol Framework (NPF) [18, 23]. The Noise Protocol Framework contains, among other features, support for mutual and optional authentication, identity hiding, forward secrecy and zero round-trip encryption. This makes Noise a promising alternative to use when securing the communication channel in a new file transfer protocol [23].

The main goal of this thesis was to provide a proof-of-concept for how a new secure protocol for file transfer could be designed and how it would perform. In order to ensure that the new protocol provides at least the same security guarantees and performance as today's alternatives a study of the security provided by SSH for SFTP was first performed. The proposed protocol will rely on the Noise Protocol Framework when it comes to establishing a secure communication tunnel. The secure tunnel will provide authentication, confidentiality and integrity protection for the transferred files. It also protects against eavesdropping and man-in-the-middle attacks.

1.1 Purpose and Goals

The goals of this thesis can be summarized as follows:

- Provide an overview of the security provided by SSH for SFTP.
- Develop a specification for a new secure file transfer protocol using the Noise Protocol Framework.
- Provide a reference implementation.

1.2 Scope

The new protocol will target users that want to protect their file transfers against an attacker that can eavesdrop on the communication as well as perform man-in-the-middle attacks. It will not protect against an attacker that has physical access to one of the endpoints.

The provided reference implementation targets the GNU/Linux operating system. The application might be able to run on other UNIX based operating systems, but it is beyond the scope of this thesis.

1.3 Related Work

The increase of data stored at data centres combined with the rise of the Internet of Things has lead to an increase in research about file transfer protocols. However, research mainly focuses on providing faster and more efficient alternatives and minimizing the bandwidth and power usage.

The original File Transfer Protocol (FTP) sent the files unencrypted between two endpoints. As the need for secure alternatives became larger, work began to use FTP over TLS/SSL (FTP/S); the developers behind SSH also started working on SFTP [14]. Other than that, there have not been much work on secure file transfer protocols. IBM developed a new protocol in the mid 2000s, Fast Adaptive and Secure Protocol (FASP) [44]. While FASP is an independent file transfer protocol, it still relies on SSH for authentication and key negotiation. Today FASP is the file transfer protocol used by IBM Aspera [1].

1.4 Thesis Outline

The thesis is structured as follows: Chapter 2 (Introduction to Cryptography and Cryptographic Protocols) introduces the necessary knowledge needed to understand how cryptographic protocols work and the cryptography used within them. Chapter 3 (SSH) presents an introduction to SSH and an overview of the security provided by SSH for SFTP. Chapter 4 (Noise Protocol Framework) provides an introduction to the Noise Protocol Framework and the security it provides. Chapter 5 (Hermod: A File Transfer Protocol using Noise) introduces the proposed proof-of-concept for the new protocol. Chapter 6 (Implementation) introduces and explains the implementation process of the protocol. Chapter 7 (Results)

presents the results of the new protocol regarding its security and performance. Chapter 8 (Discussion) provides a discussion around the specification and implementation while also presenting potential improvements and future work. Chapter 9 (Conclusions) presents conclusions drawn from the result and discussion.

NoisechapterIntroduction to Cryptography and Cryptographic Protocols In order to fully understand the content of this thesis, some essential concepts have to be introduced. This chapter starts by introducing vital concepts that are needed in order to reason about information security. Then an introduction of some cryptographic protocols and secure channels follows. The chapter then continues by introducing cryptographic primitives, before ending with an introduction to some common attacks on cryptographic protocols.

1.5 Information Security Concepts

When reasoning about information security protocols there are a few concepts that are used to describe some vital objectives or wanted features [29]. These concepts are confidentiality, data integrity, authentication and non-repudiation.

Confidentiality means that only those authorized to have access to some data should be able to access it.

Data integrity means protection against unauthorized data manipulation such as insertion, deletion and substitution.

Authentication means identification, both of identities and data. Authentication can therefore be divided into peer entity authentication and data origin authentication.

Non-repudiation means that an actor cannot deny having performed an operation, such as signing a contract.

1.6 Cryptographic Protocols

A cryptographic protocol is an algorithm or a series of steps describing a well-defined sequence of actions, often between two or more parties, that need to be performed in order to achieve a security objective [34, pp. 21–22]. The algorithm often contains multiple cryptographic primitives such as encryption schemes, digital signatures, random number generators and hash functions.

In addition to the cryptographic primitives, a cryptographic protocol needs: a transport layer, a message protocol, message identity and data representation.

The transport layer is responsible for transporting data between the involved endpoints. The transported data could be files, keys or data from some application, for example, the TLS protocol is used to securely tunnel application data between two endpoints. This layer is also responsible for supplying the transmitted data with confidentiality and integrity protection.

Message protocol and message identity mean that the packets or binary data that are transmitted through the protocol must follow a predefined specific scheme. A server must be able to translate the binary data into variables and data fields to be able to understand what is being sent and received. Message identity is used to identify that the message received actually is a message originating from the protocol and not just random data or noise. The message identity also allows the receiver to correctly parse the received data into the correct message type.

The data representation is used to translate the raw binary data into a predefined representation that allows the receiver to process or parse the received data,

act on it and perhaps create and transmit a response.

1.7 Secure Channel

A secure channel is a communication channel between two parties or devices over which the transmitted data is protected against manipulation. An attacker can learn nothing about the data that is being transmitted. In a perfect world an attacker would not be able to gain any knowledge whatsoever about any data being transmitted. However, it is hard to be protected against an attacker gaining knowledge about data sizes, end points or the timing of the transmissions through traffic analysis [13, p. 101].

In order for the transmitted data to be protected against an attacker, authentication and encryption is needed. Authentication often is in the form of a Message Authentication Code (MAC), Chapter 1.12, which protects the data against tampering and ensures that the receiver can verify the origin of the transmitted data. Encryption ensures that the data can only be viewed by the authorized receiver.

1.8 Symmetric Cryptography

Symmetric Cryptography uses a single shared key between the involved parties for both encryption and decryption [34, p. 4]. This enables both parties to both encrypt and decrypt the data using the same key, as seen in Figure 1.1.

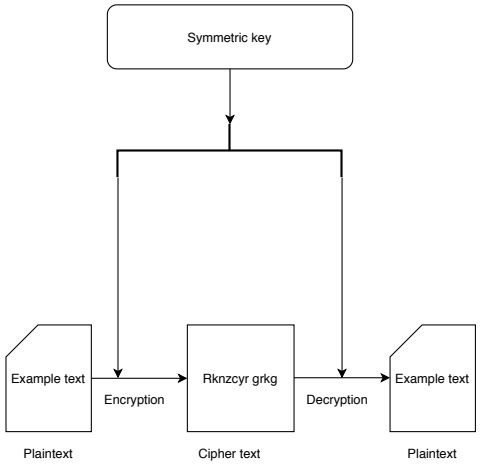

Figure 1.1: Symmetric encryption

As symmetric cryptography allows for high throughput and performance, it is well suited for usage in environments where performance is critical or when working with large data sizes. A disadvantage of using symmetric cryptography is that the same key is used for both encryption and decryption. It is therefore vital that the key is securely stored and shared between the parties. Another disadvantage is that for every new party a user wants to share information with

a new key is needed. This rapidly increases the number of keys that need to be distributed and stored, and creates a key management problem. A system with n users that can all communicate with each other needs to store $n(n-1)/2$ keys [6]. A system with a thousand users that all can communicate with each other needs to store close to half a million keys. This key management problem can be solved by using asymmetric cryptography.

Symmetric encryption can be divided into two main categories of algorithms, stream ciphers and block ciphers, based on how they operate on the input data.

1.8.1 Block ciphers

Block ciphers operates on a fixed length data block at a time [34, p. 4]. In order to encrypt data that exceeds the algorithm's block length there exist various modes of operation, so called *block cipher mode of operation* [13, pp. 63–77]. These modes describe how to apply the algorithm's single block operation on multiple blocks, how to pad shorter messages to the block length, and how the initialization vector (IV) and the encryption key should be used for each block.

1.8.2 Stream ciphers

Stream ciphers treats the input data as a stream of bits or bytes. The encryption key is used to create a key stream, together with an initialization vector. The two streams are then combined, creating an encrypted output stream. Since two separate streams are combined, the encryption operation is often based on *exclusive-or* (XOR).

As stream ciphers treat the input data as a data stream, they are easy to use when the input length is unknown as there is no need for padding the input data. Stream ciphers are often also very simple to implement in hardware and are often very fast; this makes them very popular for securing wireless connections and or for systems with constrained resources.

1.9 Asymmetric Cryptography

Asymmetric cryptography or public key cryptography utilizes two keys, one for encryption and one for decryption as seen in Figure 1.2. This can be compared to one key for both encryption and decryption in symmetric cryptography. Public key cryptography can be divided into three categories: key exchange or negotiation algorithms, encryption algorithms and digital signature algorithms.

A benefit of using public key cryptography is that it simplifies the key management needed in order to share encrypted information. Only the decryption key, the private key, needs to be stored securely.

Reusing the example from the section about symmetric encryption with 1000 users who all can communicate with each other, we now only need to store 1000 keys. The drastic decrease in stored keys comes from the fact that the users now only need to share their public key, the encryption key, with everyone. Two communicating parties can now encrypt messages using the other party's public key

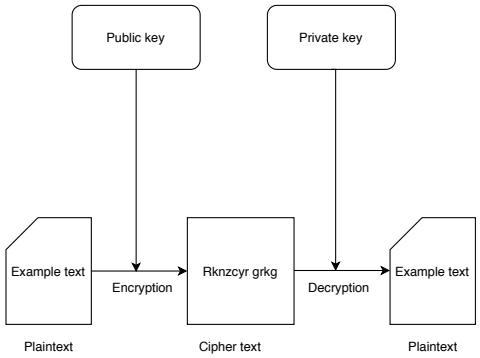

Figure 1.2: Asymmetric encryption

and decrypt received messages using their own private key. Asymmetric cryptography also removes the need for trusted third parties to share symmetric keys [6].

A disadvantage of using asymmetric cryptography instead of symmetric, is that asymmetric cryptography is a lot slower. Public key cryptography is therefore often used to negotiate symmetric encryption keys or for encrypting a symmetric key.

1.10 Hybrid Cryptosystem

A hybrid cryptosystem is a cryptosystem that combines symmetric and asymmetric cryptography. The hybrid cryptosystem can be viewed as a combination of an asymmetric cryptosystem for key transportation and a symmetric cryptosystem for data encapsulation [34, pp. 32–33]. The asymmetric cryptosystem is responsible for encrypting and sharing a newly generated symmetric key with the other party. The symmetric cryptosystem is then responsible for encrypting all transmitted data during the sessions. The hybrid cryptosystem therefore has all the advantages from both symmetric and asymmetric cryptography without any of their disadvantages [38].

Hybrid crypto systems allow us to build efficient public key protocols and can be found in most of today's protocols for public key cryptography.

1.11 Key Exchange

The purpose of a key exchange or a key negotiation scheme is to construct a key that can be used by the involved parties for the use within a protocol. The key exchange is often the first step in a cryptographic protocol as it is responsible for negotiating the key or keys that will be used for encryption.

1.11.1 Diffie-Hellman

The Diffie-Hellman protocol (DH) was one of the first asymmetric protocols, and is used to negotiate a shared secret [13, pp. 181–192]. What makes the Diffie-

Hellman protocol special is that it allows two parties to negotiate a shared secret without having any prior knowledge about each other [13, pp. 181–192]. The original algorithm uses multiplicative group of integers modulo p and the discrete logarithm problem and is still used today. Today there also exist versions of the original Diffie-Hellman protocol that use elliptic curves (ECDH) instead of the multiplicative group of integers modulo p.

$$A = g^a \bmod p \tag{1.1}$$

$$B = g^b \bmod p \tag{1.2}$$

$$K = B^a \bmod p \tag{1.3}$$

$$K = A^b \bmod p \tag{1.4}$$

Two communicating parties, commonly referred to as Alice and Bob, start the key exchange by agreeing on a modulus p and a generator g. Alice then chooses a secret integer number a and calculates A using Equation 1.1; this A is then sent to Bob. Bob then chooses a secret integer number b, calculates B using Equation 1.2 and sends it to Alice. Alice and Bob can now both calculate the shared secret, K, using Equation 1.3 for Alice and Equation 1.4 for Bob.

1.11.2 Forward Secrecy

Forward secrecy is a crucial feature for cryptographic protocols. It ensures that session keys are not compromised if the long-term private static key gets compromised by an attacker. This ensures that an attacker cannot decrypt previously recorded encrypted communication. As Forward Secrecy is such an important feature in cryptographic protocols, the feature can be found in close to all major cryptographic protocols such as TLS, SSH and Signal [28, 35, 43].

1.12 Message Authentication Code

A Message Authentication Code (MAC) is used to authenticate a message and to detect message tampering and manipulation [13, pp. 89–90]. A MAC takes a secret key K and a message of arbitrary length and produces a fixed size MAC output. The secret key is shared between the sender and the receiver and ensures that only they can calculate the correct output. The sender then calculates a MAC for the message it wants to protect and appends the MAC to the output and sends the concatenated message to the receiver. The receiver then recalculates the MAC of the message received minus the MAC and compares it to the MAC attached to the message. If they match, the message has not been tampered or manipulated with and it originates from the sender.

Since a MAC takes a secret key as parameter it is also suitable for authenticating that a message originates from the supposed sender. The receiver can be sure the message originates from the sender if the MACs match, as long as the secret key handled securely.

Hash functions are often used in the construction of MACs. The hash functions can map the message to a fixed length output that can then be used together with

the secret key to construct a MAC. This speed up the MAC calculation as only the hash needs to be signed and not the complete message.

1.13 Authenticated Encryption

Authenticated Encryption (AE) or Authenticated Encryption with Associated Data (AEAD) are forms of encryption that provide both confidentiality and authentication of the protected data. Authenticated encryption schemes provide the same properties that an encryption scheme and a message authentication code provide on their own. However, when using an AE or AEAD scheme these properties are provided under a single, simpler interface.

The input to the encryption function of an authenticated encryption scheme takes the data that will be encrypted and a key, and outputs a ciphertext and an authentication tag. For decrypting, the authentication tag is calculated and compared to the tag attached to the ciphertext. If the tags do not match, the cipthertext is discarded. If the tags match, the ciphertext is decrypted and the plaintext is returned. This ensures that only authenticated data is decrypted, and a minimum of resources are spent on unauthenticated messages or data.

An AEAD algorithm also provides the option to check the integrity of an encrypted message. This feature ensures that the message cannot be manipulated and that it originates from an authenticated sender. AEAD can therefore replace the more traditional approach of using separate algorithms for encryption and MAC.

1.14 Attacks on Cryptographic Protocols

1.14.1 Man-in-the-middle Attack

In a man-in-the-middle attack an attacker secretly inserts itself in between two communicating parties. The two parties believe they are communicating with each other directly when in fact a third party has access to the communication. The attacker can either passively eavesdrop and relay the communication or actively alter it. In order to prevent a man-in-the-middle attack most cryptographic protocols include some kind of authentication for the endpoint.

1.14.2 Replay Attacks

A replay attack is an attack where the attacker first eavesdrops on a communication channel and intercepts the communication between two parties before sending the intercepted packets to the receiver. If an attacker eavesdrops and intercepts, for example, a login attempt to a server, the attacker could try to gain access by replaying the handshake. As the attack only involves sending packets, it is an attack that is very simple to perform as there is no need, for example, to break a key exchange scheme to get access to the session key.

In order to be protected against a replay attack, session identifiers, nonces or timestamps can be used. The receiving party could, for example, only accept

messages within a certain time frame. By using a new unique session identifier for each session, the receiving party could discard messages if the provided identifier does not match the identifier associated with the current session. This makes it impossible for an attacker to replay messages from a different session.

Chapter 2
Secure Shell

Secure Shell (SSH) is a protocol and a software package that provides secure connections over insecure networks. SSH enables secure system administration, file transfer, remote command-line and remote command execution to take place over secured connections.

The protocol was first developed to provide a secure alternative to Telnet and the Berkley protocols for remote shell, rlogin, rsh and rcp [30, 27, 31]. Today the protocol is used in most data centers and larger enterprises [39]. SSH can also be found embedded in many solutions for file transfer and system management.

2.1 The SSH Protocol

The Secure Shell Protocol follows a client-server model where the connection is initiated by an SSH client connecting to an SSH server. The SSH client is responsible for establishing the connection and is the driving party during the connection and setup phase. During the setup phase the client authenticates the SSH server using asymmetric cryptography.

Once a connection has been established, the SSH protocol uses symmetric encryption and MAC to ensure that data exchanged between the client and the server remain confidential and integrity protected. Connection specific algorithms and parameters are negotiated during the setup phase.

The SSH protocol runs on top of TCP/IP and is built on top of three core protocols, with SFTP adding a fourth protocol [52]. Figure 2.1 provides an overview of how the different protocols and applications in SSH are connected.

The *SSH Transport Layer Protocol* sits at the bottom of the protocols and is responsible for the connection and for establishing a secure channel between the client and the server [53]. When a client initiates a connection, this protocol is responsible for authenticating the server, negotiating session keys and cryptographic algorithms to use. After the connection phase the Transport Layer Protocol provides the other SSH protocols with privacy, integrity protection and optional data compression.

The *SSH Authentication Protocol* is responsible for authenticating the connecting client to the server and operates on top of the Transport Layer Protocol [50].

The *SSH Connection Protocol* takes over after a successful client authentication, and just as the Authentication Protocol, it uses the secure channel provided

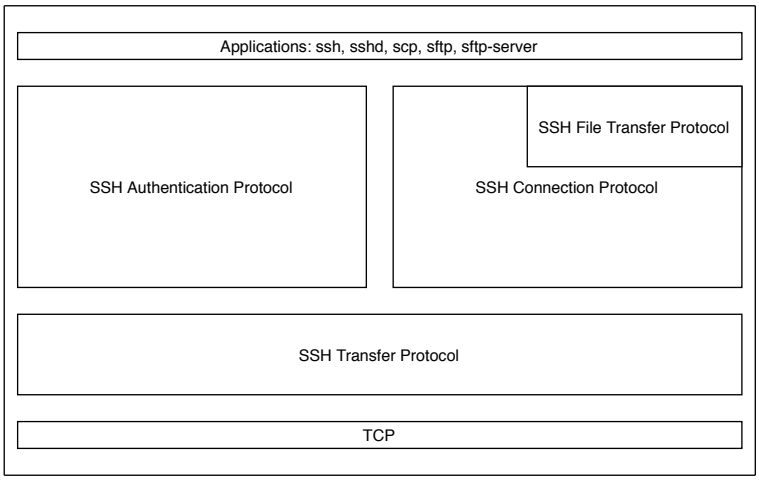

Figure 2.1: The relation between the SSH protocols.

by the Transport Layer Protocol [51]. The Connection Protocol is responsible for providing a channel for tunneling data from the requested remote service. It also enables multiplexing the secure tunnel into multiple channels.

The *SSH File Transfer Protocol* (SFTP) runs on top of the Connection protocol and allows for remote access to a file system [49]. SFTP will be further introduced in Section 2.3.

2.1.1 User Authentication

The SSH protocol supports multiple ways of user authentication, e.g. password, public key, challenge-response based, host based and GSSAPI [41]. This thesis analysis will focus on public key-based authentication, as it is the recommended way of doing user authentication in SSH [26].

Public key authentication was originally added to the SSH protocol to support secure automation, allowing for smooth integration into third-party applications or scripts. Since then it has been the recommended way of conducting authentication when using the SSH protocol as it provides a higher level of security compared to password-based authentication.

The public key authentication method is built around the fact that the client has a cryptographic key pair consisting of one public and one private key. The client's public key is then shared, either manually or through the *ssh-copy-id* application, with the server and stored in the server's *authorized_keys* file. When a client connects, the server grants access if the private key matches a public key the server has stored for that client. The client also maintains a file, *known_hosts*, that contains the public key of server's and hosts that the client trusts. If a server

or a client updates their key pair, a mismatch will happen during the authentication. In case there is a new key pair for the server the client will be notified and prompted to accept or discard the new key. If the client updated the key pair without sharing the new keys with the server, the authentication will fail.

2.1.2 Key Exchange

A key exchange is performed in order to establish session keys for encryption and authentication. The key exchange can be split into two stages. The first stage negotiates the algorithms and parameters to use and a final stage where the actual key exchange takes place. SSH also combines the key exchange with a signature to provide explicit server authentication.

The explicit server authentication is used so the client can be sure it is talking to the correct server [53]. By including the shared Diffie-Hellman secret, the client can verify that the server knows the correct shared secret and is capable of decrypting any data sent through the channel. The authentication is performed by signing a hash. Equation 2.1 shows how the hash is calculated.

$$\begin{aligned} hash = HASH(&Client\,identification\,string\,||\\ &Server\,identification\,string\,||\\ &Client\,message\,payload\,||\\ &Server\,message\,payload\,||\\ &Server\,public\,hostkey\,||\\ &DH\,value\,from\,client\,||\\ &DH\,value\,from\,server\,||\\ &DH\,shared\,secret) \end{aligned} \tag{2.1}$$

2.1.3 Confidentiality

During the handshake an encryption algorithm will be negotiated, and a key exchange takes place, establishing a shared key used for encryption. After the key exchange and algorithm negotiating is done the packet length, padding length and padding fields as well as the payload must be encrypted using the negotiated algorithm and key.

The benefit of negotiating new session keys for every new session is that if the host keys are compromised, an attacker will not be able to decrypt data from previous sessions.

2.1.4 Integrity

The SSH protocol uses a MAC tag that is computed from the unencrypted message payload, the sequence number of the packet and from a shared secret as seen in Equation 2.2.

$$mac = MAC(key, sequence\_number\,||\,message\,payload) \tag{2.2}$$

This ensures that messages cannot be tampered with and that the payload originates from the sender.

2.2 Supported Algorithms

The SSH protocol supports a wide range of algorithms for providing authentication, confidentiality and data integrity. In RFC4252 and RFC8332 the officially supported algorithms are presented [50, 4]. Depending on the SSH implementation, other algorithms could be supported through implementation specific extensions. The SSH protocol is more than twenty years old and it still provides support for outdated algorithms. This means that users can create, configure and expose vulnerable servers.

2.3 SSH File Transfer Protocol

The *SSH File Transfer Protocol* (SFTP) is a file transfer protocol and was added in SSH2 (an updated version of the original SSH protocol) as a secure alternative to FTP [49]. Initially SSH provided a secure version of the remote copy (rcp) tool from Berkley, in the form of the Secure Copy (scp) application [27]. However, the original *scp* application was very limited in features compared to FTP. When developing SSH2, the developers saw an increased need for secure file transfer and introduced SFTP as a more feature complete protocol for transferring files.

SFTP runs on top of the SSH Connection Protocol and has therefore all security and authentication features provided by the SSH Transport Layer and Authentication Protocols. Once a user is logged in and authenticated the SFTP protocol is initiated. Since the SSH protocol is available on most corporate networks, it has enabled SFTP to become widely adopted and it is now a standard part of the SSH protocol suite and implementations are available for most systems. With the introduction of SFTP, the insecure FTP has become obsolete and replaced. FTP/S (FTP over SSL/TLS) is quickly getting replaced by SFTP as well [36].

The SSH ecosystem comes with the *sftp* shell command for Linux systems, and is an interactive command line interface for using SFTP. With the addition of the SFTP protocol in SSH2, the original *scp* was rebuilt to utilize the new STFP protocol for the file transfers, providing two interfaces for the same protocol [11].

2.4 Security Considerations

2.4.1 Authentication

In the SSH protocol, it is up to the server to decide which method to use when performing user authentication. The authentication is only as strong as the method used for authentication. Configuring a server to accept a weak method for user authentication exposes the underlying machines and network to an attacker.

When using public key authentication, it is assumed that the client's host keys, and the server's private key are not compromised. Therefore it is recommended to store the private keys encrypted. However, this is hard to enforce in practice

and causes other problems when using *ssh* in scripts. If a clients private key is compromised, an attacker has full unrestricted access to the server.

2.4.2 Man-in-the-middle

In order for the protocol to offer protection against a man-in-the-middle attack the server's public key must be securely distributed to the client. Should the keys be insecurely distributed or stored, the client cannot verify that it is talking to the correct server.

If the server key is securely distributed to the client and an attacker tries to perform a man-in-the-middle attack the client will notice a mismatch between the server key and the hostname. A warning will be shown to the user combined with an option to abort the connection. The user should also be careful and not automatically accept a new or changed server key.

The protection against man-in-the-middle attacks is therefore only as strong as the key distribution is secure combined with the users' and administrators' ability to verify the key/hostname relation.

A huge problem with the man-in-the-middle protection in SSH is that it relies on 'Trust on First Use' for host keys [40]. When a user first connects to a new server, the server's key fingerprint is shown, and the connecting user is asked whether to trust the key and continue on with the connection or to abort it. Studies have shown that users tend to not verify the provided fingerprint and if they do, they rarely verify the full fingerprint [16]. This side steps or weakens SSH man-in-the-middle protection.

2.4.3 Forward Secrecy

As the SSH protocol uses Diffie-Hellman or elliptic curve Diffie-Hellman for every session to generate a new session key, the protocol provides perfect forward secrecy [35]. Should the private Diffie-Hellman parameters for either the client or the server be revealed, the session key will also be revealed. It is therefore vital for the implementation to securely discard these parameters when the key exchange is completed.

2.4.4 Data Integrity

The SSH protocol provides data integrity protection in the form of a MAC. As the SSH protocol uses a 32-bit MAC, information could start to leak after 2^{32} packets have been sent.

The MAC schemes used by the SSH protocol rely on SHA-1 or MD5, which are both deemed to be insecure [37]. In 2017 Google announced they had successfully performed a collision attack against SHA-1 [42]. MD5 has been vulnerable to collision and pre-image attacks for years. These two algorithms are no longer accepted by web browsers and warnings are issued when using them with TLS [47].

2.4.5 Replay and Key Reuse

In order to protect oneself against a replay attack, the SSH protocol derives the session key from pseudo-random data gathered from the key exchange process. The SSH Authentication Protocol uses this to ensure that signatures from previous sessions cannot be replayed. This also enables protocols running on top of SSH to bind data to a session, thus preventing an attacker from replaying messages from previous sessions.

Two sessions could theoretically have the same session key, but since the key originates from a hash the probability for this to happen is minimal. The risk can be further minimized by using a larger output string from the hash function.

2.4.6 Confidentiality

The ciphers used by SSH are the industry standards for encryption. The developers have chosen to support eight different algorithms with a variety of key lengths. Due to the age of the protocol, the recommended algorithms have changed and some of the algorithms included in the protocol are no longer regarded to be secure enough, e.g. 3DES. It is therefore up to the developers to suggest and recommend strong default algorithms and up to the users to actually follow the recommendations.

2.4.7 Cipher suite

SSH supports a high number of cryptographic algorithms. While this allows users to tailor the used algorithms for their own use case, it increases the code size and maintainability cost. More supported algorithms mean that there are more potential attack vectors, and the larger code size means that the risk for bugs increases.

2.4.8 Configuration

The greatest security threat to SSH is through misconfiguration, by choosing weak cryptographic primitives, or by using a compromised system. A user of the SSH protocol cannot be expected to have deep knowledge in information security so the potential for misconfiguration is high. The lack of knowledge in information security also means that the users lack the ability to properly review and understand the configuration and its security implications. Therefor, the users have to rely on that the developers provides a sane default configuration.

Since the SSH client and the SSH server negotiate the algorithms used for authentication, key exchange and encryption during the setup phase, it is possible for a party configured to use secure primitives to be forced to use weaker primitives due to the configuration of the other party.

The large number of supported primitives increase the risk of a vulnerability being found. It also increases the chances of exposing a less secure server to the Internet and it forces administrators and users to be up to date on the recommended algorithms to use.

2.4.9 Conclusions

The SSH Protocol offers a high level of security. The protocol, the family of applications and the protocol enable users to perform a wide range of tasks. However, SSH has grown quite a bit since its first release and now comes with a large overhead and complexity. SSH forces the users to configure the clients and servers correctly while also forcing the developers to recommend safe algorithms for the end users.

Within the SSH protocols all critical parameters are negotiated, e.g. the methods for server and user authentication and algorithms for exchanging session keys. This adds a lot of complexity to the protocols and introduces potential attack vectors.

For conducting file transfers using SFTP there is a lot of overhead and the process involves four different protocols. The SSH Transport Layer Protocol needs to agree on algorithms to use and perform server authentication. The SSH Authentication Layer Protocol then needs to perform user authentication before the SSH Connection Protocol can initiate a *sftp* channel and launch the *sftp* service. Once the *sftp* service is started, the actual SFTP protocol can start. This process is complex and can be streamlined.

Since SFTP runs on top of the other SSH protocols the whole SSH protocol suite is needed even if a user only wants to share files and has no interest in remote execution, proxy services or socket binding. It also means that the *sftp* service is open for vulnerabilities in the other services and protocols.

A new secure file transfer protocol should focus on only providing secure file transfer. This makes the specification and implementation smaller and easier to review, verify and maintain.

Chapter 3
Noise Protocol Framework

The Noise Protocol Framework (Noise) is a new framework for creating cryptographic protocols based around the Diffie-Hellman key exchange method [23, 25]. The framework primarily supports authentication, both mutual and optional identity hiding, forward secrecy and zero round-trip encryption.

Noise provides a lightweight framework that is not bound to any specific domain. As the framework relies on public and private keys for authentication there is no need for a trusted third party. This enables Noise to be a competitive framework for building protocols that utilizes the Diffie-Hellman protocol.

The features offered by Noise make it a promising alternative to TLS and SSH for establishing secure communication channels, e.g. WhatsApp has replaced TLS with a protocol based on Noise for mobile devices [48]. Noise can also be found in the VPN protocol WireGuard, the decentralized network Lightning and in the anonymous network I2P [12, 5, 24]. There is also a version of QUIC, that uses Noise instead of TLS for the secure channel [18, 17].

The developers behind Noise have chosen to provide a few but carefully chosen ciphers. The cryptographic primitives advocated by Noise have been chosen due to their security and performance, allowing for the creation of protocols offering competitive performance and security on all kinds of devices.

Another benefit with Noise is the notation introduced by the Noise framework to describe handshake patterns. This notation makes it very easy to formally verify the handshake patterns and in extension the created protocols. The ability to easily create formal verifications of the handshake patterns allows developers to prove that their protocol comes with the claimed security. Noise Explorer is a tool that is developed to help verify Noise Patterns, and provides an overview of the security properties the various patterns provide [22].

3.1 Noise State Machine

The core of the Noise Protocol Framework is a state machine that contains and maintains a predefined set of variables. During the handshake these variables are used by the state machine to perform the specific calculations needed by the handshake pattern. The state machine is advanced by processing tokens from the provided handshake pattern. The set of variables used and maintained by Noise are the following [25]:

- A local party's static key pair - s
- A local party's ephemeral key pair - e
- The remote party's static public key - rs
- The remote party's ephemeral public key - re
- A hash value that is calculated from the previous handshake data sent and received - h
- A chaining key based on all previous DH outputs, used to derive the encryption keys - ck
- Encryption key, recalculated every time ck is changed - k
- Counter based nonce - n

In order to begin the execution of a Noise protocol, you initialize a *HandshakeState* by calling the *Initialize* method and providing a pattern and specifying which algorithms should be used [25]. Depending on which pattern is specified, static keys are also provided to the state machine. After initialization, the *WriteMessage* and *ReadMessage* methods are called to process each handshake message. *WriteMessage* consumes a received message and the *ReadMessage* produces a message that can be sent to the remote party. These methods are also responsible for any calculations that are needed in order to fully process the current token. If any error is encountered the handshake has failed and the *HandshakeState* object is deleted.

When the handshake is completed, all tokens in the patterns have been processed and encryption keys have been negotiated, two *CipherState* objects are returned: one object for encrypting messages from the initiator and one object for decrypting messages that are received from the responder. The *HandshakeState* object is also deleted. At this point messages can be encrypted or decrypted using the *EncryptWithAd* and *DecryptWithAd* methods on the relevant *CipherState*. If an error is encountered at any point from here on the *CipherState* object is deleted and the session is terminated.

3.2 Noise Patterns

The main building block when working with the Noise Protocol Framework is the Noise handshake patterns, used to describe Diffie-Hellman based protocols. A handshake pattern is a sequence of messages, defined by tokens, that describes how information flows in the handshake. The tokens used in the patterns, map to the variables that are stored in the state machine. By using the token system, it is very easy to quickly understand what data is being sent and how it is used by the framework. The patterns and tokens also make it easy to both reason about and formally verify the security they intend to provide. In order for the two involved parties to have some predefined shared data, Noise supports so-called pre-message patterns that represent some shared data, such as public keys.

The set of tokens Noise patterns are made of are the following [25]:

- Ephemeral key pair generated by the sender, stored in the e variable - e
- The sender transmits the static public key. Is taken from the s variable and encrypted if k is not empty - s
- A Diffie-Hellman operation between the initiator's key pair. The first letter determines if the initiator's static or ephemeral key is used, likewise the second letter determines if the responser's static or ephemeral key is used. - *ee, es, ss, se*

The Noise patterns are named with one or two letters. The first letter refers to the static key of the initiator [25]:

- No static key for initiator - N
- Static key for initiator known to responder - K
- Static key for initiator transmitted to responder - X
- Static key for initiator immediately transmitted to responder - I

The second letter refers to the static key of the responder [25]:

- No static key for responder - N
- Static key for responder known to initiator - K
- Static key for responder transmitted to initiator - X

This thesis will follow the notation used in the Noise specification for the Noise patterns [25].

```
Noise NN pattern:
    Alice:              Bob:
          -> e
          <- e, ee
```

Listing 3.1: Noise handshake pattern.

```
Noise KK pattern:
    Alice:                  Bob:
          -> s
          <- s
           ...
          -> e, es, ss
          <- e, ee, ss
```

Listing 3.2: Noise handshake pattern with pre-message pattern.

The Noise pattern NN, show in Listing 3.1, describes a Noise pattern for performing an unauthenticated Diffie-Hellman handshake. The pattern has two messages: the sender starts by sending its ephemeral public key and the recipient responds by sending an ephemeral public key. The two ephemeral keys are then used to create ee by performing a Diffie-Hellman calculation.

The KK pattern, as show in Listing 3.2, describes a Noise handshake pattern involving a pre-message pattern. Before the handshake is initialized, the parties have at some point earlier in time securely exchanged static keys. The sender initiates the handshake by transmitting an ephemeral key and constructing es and ss. The recipient responds by sending an ephemeral key allowing for the creation of ee, and calculates ee and ss. When the initiator has received the response from the original recipient, it can calculate ee. The full list of patterns supported by Noise is available in the Noise specification [25].

3.2.1 Noise Algorithms

In order to initialize and use the Noise state machine a set of algorithms must be specified.

For the key exchange a Diffie-Hellman algorithm must be chosen. This algorithm will be used to generate the public and private key pairs. An encryption algorithm must also be chosen and will be used for encryption and decryption of packet data. A hashing algorithm must also be designated. The hashing algorithm will in addition to hashing be used for deriving keys, using a Hash-based Key Derivation Function (HKDF).

3.2.2 Cipher Suite

The Noise framework differs from most of the popular protocols for establishing a secure channel in that it comes with a small supported cipher suite. Noise comes with support for two elliptic-curve Diffie-Hellman key negotiation schemes, the Curve25519 and the Curve448. Two algorithms for authenticated encryption, AES-GCM and ChaChaPoly1305. Two hash algorithms with 256 bits output, SHA256 and BLAKE2s and finally two hash algorithms with 512 bits output, SHA512 and BLAKE2b.

3.2.3 Validity Rules

In order for a pattern to be valid, it must not only be syntactically correct, but it must also conform to four validity rules [25]. If the pattern described fulfill all four validity rules, the pattern should be executable and provide the security guarantees the pattern is specified to have. The validity rules are as follow:

1. Diffie-Hellman operations can only be performed using keys both involved parties possess.
2. A key cannot be sent more than once during a handshake.
3. Each Diffie-Hellman calculation may only be performed once.

4. After a Diffie-Hellman operation on a remote public key and a local static key the local party is not allowed to call the ENCRYPT method without first performing a Diffie-Hellman operation on its local ephemeral key and the remote key.

In order to initialize a Noise state machine an ASCII string is passed to the *Initialize* method. The string specifies the Noise pattern, the Diffie-Hellman function, the AEAD function and the hash function and starts off with "Noise_", Listing 3.3 shows how such a string could look.

```
Noise_KK_25519_ChaChaPoly_BLAKE2b
```

Listing 3.3: Noise protocol name.

The string specifies the KK pattern, the Curve25519 for Diffie-Hellman, ChaChaPoly1305 for authenticated encryption and BLAKE2b for hashing.

3.3 Security Properties

Protocols built using the Noise framework provide security tailored to the specific needs of the protocol thanks to the Noise patterns. Each Noise pattern comes with different source and destination properties; the properties are also tied to each message in the handshake. Choosing the correct pattern is therefore vital when creating a protocol using the Noise framework.

In order to make it easier to overview the specific security properties provided by a certain Noise pattern, Kobeissi, Nicolas, and Bhargavan created the Noise Explorer [20]. The Noise Explorer allows for automated modelling and verification of arbitrary Noise protocols and pattern analysis over the handshakes for each pattern. It is available as both a command line and a web application [22].

3.3.1 Source Properties

The source properties are a set of properties that handle the level of authentication a recipient is provided when receiving messages from a sender. There are three levels of authentication that can be provided by the Noise framework [25].

The first property is *no authentication*. This property provides no authentication which means that the payload could have been sent by anyone.

The second property is *sender authentication vulnerable to key-compromise impersonation (KCI)*. This property authenticates the sender using static-static Diffie-Hellman. When using a protocol with this property, it is vital that the long term key belonging to the recipient is stored securely. If this key gets compromised the authentication can be forged.

The third source property is *sender authentication resistant to key-compromise impersonation*. This property authenticates a sender using ephemeral-static Diffie-Hellman. The authentication is performed with the sender's static key pair and the recipient's ephemeral key pair and is resistant to forging assuming the private keys are not compromised.

3.3.2 Destination Properties

The destination properties are a set of properties that handle the level of confidentiality that the payload is provided to the sender. There are six levels of confidentiality that can be provided by the Noise framework [25].

No confidentiality offers no confidentiality and the payload is transmitted without any encryption to the receiver.

Encryption to an ephemeral recipient offers the payload forward secrecy as it is using *ephemeral-ephemeral* Diffie-Hellman. The payload can however be sent to anyone as the sender is not authenticating the receiver.

Encryption to a known recipient, forward secrecy for sender compromise only, vulnerable to replay. Protocols with this property encrypt the payload based only on the static key of the recipient. If the recipient's static keys are compromised the payload can be encrypted. The payload is also vulnerable to replay attacks as only static keys are used in the Diffie-Hellman operation.

Encryption to a known recipient, weak forward secrecy encrypts the payload based on an ephemeral-ephemeral and ephemeral-static Diffie-Hellman operation. As the sender has not verified the relation between the recipient's ephemeral key pair and the static key pair, the ephemeral key can be forged by an active attacker. If an attacker gains access to the recipient's private static key the payload can be encrypted.

Encryption to a known recipient, weak forward secrecy if the sender's private key has been compromised encrypts the payload based on an ephemeral-ephemeral Diffie-Hellman combined with an ephemeral-static Diffie-Hellman using the recipient's static key pair. Should the sender's static key be compromised, an attacker could compromise new sessions and decrypt their payloads. However, past sessions are still secure.

Encryption to a known recipient, strong forward secrecy encrypts the payload based on an ephemeral-ephemeral Diffie-Hellman combined with an ephemeral-static Diffie-Hellman using the recipient's static keys. As long as the private key of the ephemeral key pair stays secure and the recipient is not impersonated by an attacker the encrypted payload cannot be decrypted. This is the strongest destination property offered by Noise.

3.4 Security Considerations

When using a Noise pattern that uses static public keys, it is important to remember that Noise only verifies that the other party possesses the corresponding private key. The overlaying protocol or application still needs to decide if the remote party's key should be accepted or discarded.

As with all protocols that use nonces and ephemeral keys it is important that they do not get reused. Reusing a nonce with the same encryption key would render the provided confidentiality useless for the affected data.

If the overlaying protocol or application changes the Noise protocol, either the pattern or the algorithms, the keys must be recalculated. Reusing the keys with a new protocol could leak information about the keys, compromising the session.

Chapter 4

Hermod: A Secure File Transfer Protocol using Noise

Hermod is the proposed proof-of-concept protocol for conducting secure file transfer. The protocol aims to provide a secure protocol that should be easy to use for the end user. SFTP provides a secure protocol but gives the user the responsibility to ensure a secure configuration is used. Hermod aims to remove the need for the user to control options related security, enabling the user to focus fully on using the protocol.

Hermod will utilize the Noise Protocol Framework for creating a secure tunnel between the client and the server. By using the Noise Protocol Framework, Hermod will be provided with strong security. Noise will provide Hermod with Diffie-Hellman based static keys for authentication and strong authenticated encryption to provide confidentiality and data integrity to the data transmitted through the secure tunnel.

4.1 Security Goals

In order to provide a secure protocol and rival SFTP when it comes to secure file transfer a few properties must be supported. Forward secrecy must be provided to ensure that the transmitted data is secure should a device or party become compromised. It protects past sessions in case an involved party's static keys are leaked in the future. In order to provide forward secrecy Hermod will derive the session keys from ephemeral key pairs.

Mutual authentication is needed to ensure that the correct and expected parties are involved in the communication. Noise provides authentication based on pre-shared asymmetric public static keys and allows for both client and server authentication. The usage of authenticated encryption also ensures that not only the handshake provides authentication, but every transport packet sent. Noise tags every message with a 128-bits authentication tag to ensure packet authentication.

Data integrity ensures that messages cannot be tampered with without detection and is needed to ensure the files transmitted using Hermod arrive without outside manipulation. Hermod, being based on Noise, will provide Data integrity by using authenticated encryption with associated data.

Replay protection is needed to protect the protocol from replay attacks. The protection ensures that an attacker cannot replay messages or handshakes to establish a new connection or force the party to accept data or request once more.

Another goal of Hermod is to remove the need for the user to have a deep knowledge of information security, being up to date on cryptography and reading manuals to create a secure system. The protocol will provide no options for configuring security properties. Key sizes and algorithms will all be specified in the protocol and will not be configurable. The user can then focus on looking after authorized keys and known hosts.

The properties must also be provided in a small package. Keeping the protocol small, especially the operations regarding handshakes and confidentiality, makes the protocol easier to be implemented, reviewed, verified and maintained. This is important to ensure that potential attack vectors are kept at a minimum.

4.2 Specification

4.2.1 Design

Hermod aims to be a simple protocol with minimum overhead. To accomplish that the protocol offers minimal configuration and removes all of the negotiation about algorithms that can be found in most other protocols.

A Hermod session can be divided into two stages. The first stage is responsible for authentication of both the client and the server combined with negotiating a session key for encryption. The second stage handles the uploading or downloading of files to or from the server. Uploading or downloading is done by sending a request to the server which then responds accordingly, either by preparing to receive file content or by sending a file back to the client. An overview of how a Hermod session progresses can be seen in Figure 4.1.

Similar to SSH, Hermod uses text files for storing information about authorized clients, known servers, public and private keys. Hermod also borrows the concept of aliases from SSH, but instead of making them optional they are used by default in Hermod. When a client adds information about a new server it is given an alias. This alias is then used to reference the remote server. By using aliases, a client does not need to remember a specific host name or its IP. Hermod differs from SSH by forcing the use of aliases whereas in SSH they are optional.

4.2.2 Cryptographic Algorithms

Since Hermod utilizes the Noise Protocol Framework for authentication, encryption and integrity protection it is bound to use the algorithms supported by Noise.

For Authenticated Encryption with Associated Data, Hermod uses the ChaChaPoly1305 scheme. AES could offer better performance due its hardware support through the AES-NI extension to the x86 instruction set. However, ChaChaPoly1305 allows for high performance in software implementations and does not rely on the instructions provided by AES-NI for fast encryption and decryption.

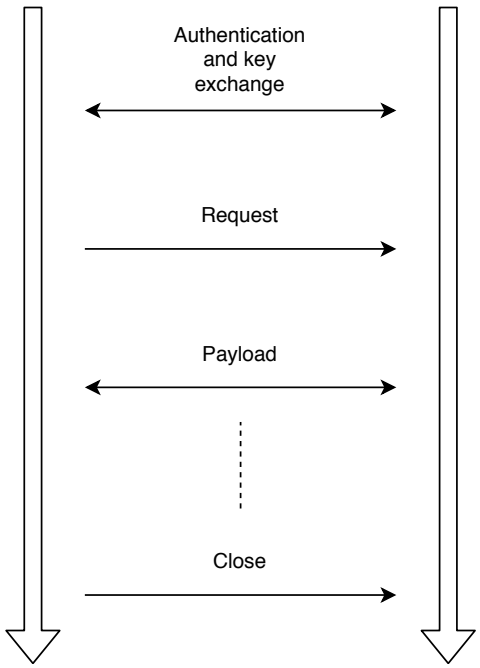

Figure 4.1: Overview over the Hermod protocol.

Hermod uses the BLAKE2s hash algorithm as a cryptographic hash function. BLAKE2s offers 256 bits of output and excellent performance on 32-bit architectures. Its big brother BLAKE2b is another algorithm that would suite the Hermod well; it offers 512 bits of output but comes with a decline in performance on non 64-bit architectures. One of the goals for Hermod is to not be bound to any specific domain and architecture thus making BLAKE2s a better choice, allowing for better performance on embedded systems with 32-bit CPU architectures.

For elliptic-curve Diffie-Hellman, Hermod uses the Curve25519 algorithm. The Curve488 algorithm is also supported by Noise and offers more bits of security by sacrificing some performance. However, Curve25519's level of security is still high and seems to be the recommended algorithm [21].

4.2.3 Noise Pattern

Hermod utilizes the KK Noise pattern consisting of a pre-shared static key and an ephemeral key for both the client and the server [25].

The Noise Explorer shows that after a successful handshake the KK pattern ensures that the data transmitted is provided with sender and receiver authentication, message secrecy and strong forward secrecy [19].

Authentication and Key Exchange Handshake

The handshake between the client and server is initialized by the client sending an initialization message containing its identification token and an ephemeral public key. The receiver then performs the first step in the Noise pattern and responds by sending a response message containing the server's ephemeral public key. Both the initiator and receiver can now complete the calculations required by the pattern for deriving a session key and move into the transport mode. An overview of the handshake can be seen in Figure 4.2.

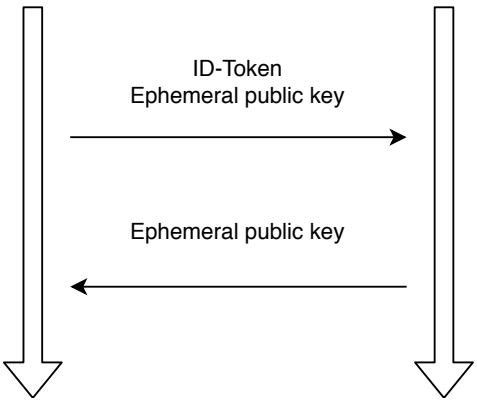

Figure 4.2: Overview over the Hermod authentication and key exchange.

4.2.4 Transport Layer

The data transmitted by the Hermod protocol after a successful handshake is protected by AEAD encryption from the ChaChaPoly1305 algorithm. By using Authenticated Encryption with Associated Data, the transmitted data are provided with confidentiality, data integrity and authentication. The communication can only be decrypted by an attacker with knowledge about the ephemeral private keys and that at the same time impersonates the responder.

In order to add another layer of protection and to minimize the impact of an attacker getting knowledge about the session key, a new session key for encryption is generated for each gigabyte of data sent.

Messages

Data is sent between the client and the server as messages. The messages consist of a type, a length field and a payload. This allows the receiver to ensure that it received expected data by just checking the message type. The messages sent by Hermod can be seen in Table 4.1 on Page 29.

Message Type	Sender	Description
Init	Client	Initialize a handshake
Response	Server	Respond to a handshake Init message
Request	Client	Request to upload or download a file
Payload	Both	The contained data is file content
Metadata	Server	Metadata about the file that will be transferred to the client
EOF	Both	Signals end of file
Error	Both	Signals an error
Close	Client	Close the connection
Okay	Both	Signal that all is ok
Rekey	Both	Signal that a new encryption key should be generated
Unknown	Internal	Used internally if the message type does not correspond to any of the above types

Table 4.1: Table over the messages sent by the Hermod client and server

4.2.5 File Transfer

As Hermod is a secure file transfer protocol, the central part of the protocol is the uploading and downloading of files. In order for a client to upload or download a file to or from the server, a request is sent to the server as seen in Figure 4.3. The request can either be an upload request, which means that the client wants to upload a file, or a download request in which case the client wants to download a file from the server. A Hermod session can consist of multiple requests, sent synchronously, in order to facilitate the need to upload or download multiple files.

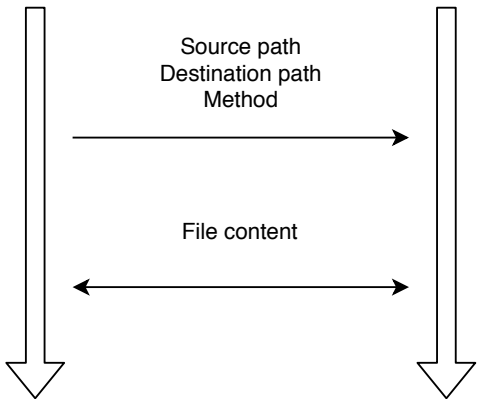

Figure 4.3: Overview over how requests are sent.

For uploads, the file content is sent directly to the server without any overhead.

For downloads a metadata message is sent before the file content. This allows the client to get knowledge about if the requested path is a directory or a file and its size. The format could easily be extended to support sending file permissions and other meta data of interest.

4.2.6 Client Data

The client maintains a directory, *known_servers*, that contains a file for each server it trusts. The file name is an alias for the server and is chosen by the user at creation. In the file the servers host name or IP address is found together with the client's static key pair for that server, its identification token and the server's public key. If the client tries to connect to a server, not previously known or if the information mismatches, the connection attempt will be stopped.

The id token serves as identification for the client when it connects to a server and allows the server to find the corresponding public static key. The token is a 32-bit random string that is generated at the same time as the public static key for the client. Both the identification token and the keys are stored Base64 encoded.

Listing 4.1 showcases what a server entry, stored in *known_servers*, can look like.

```
Hostname: 127.0.0.1:4444
PublicKey: 5EMaAeNeqX+o1lHEhL2LsJKWM/Ee9AWIH1gFhYrpX1A=
PrivateKey: RUjOZYhmJa2DMsY3ukFGNpqGAqI2PuRVb2bjuwXTOMo=
IdToken: wrmcv3lFUVQ=
ServerKey: hP6bUrXjogziTJJZ0TOFjLWdyFjwoNpjq+4JGI1CkHs=
```

Listing 4.1: Example of a client file.

4.2.7 Server Data

The server maintains a file, *authorized_clients*, that contains clients that are authorized to establish a connection. For each client, the client's id token and public static key is stored. In addition to the *authorized_clients* file the server maintains a file for its public static key, *server_key.pub* and one file for its private static key, *server_key*. All keys are encoded using Base64.

If a client connects and its id token it is not found in the *authorized_clients* file, the connection will be closed. In addition, the handshake will fail if the static key provided does not match the key stored together with the id token.

```
dea2b412:AAAAC3NzaC1lZDI1NTE5AAAAIB7U...
dea2b413:BAAAC3NzaC1lZDI1NTE5AAAAIB7U...
dea2b414:CAAAC3NzaC1lZDI1NTE5AAAAIB7U...
```

Listing 4.2: Example of a server file.

Listing 4.2 shows an example of how a server's *authorized_clients* file can look like. The file contains authorized clients. Each line corresponds to an identification token for a client and its public static key.

4.3 Key generation and Sharing

In order to securely generate an identification token for the client and the static key pair for both the server and the client, a command line interface is provided. For the server, the application will generate the static key pair and store the keys in the correct place. For the client, the application will generate a new static key pair and a new identification token. The keys and the identification token will then be stored in a file together with the hostname or IP address of the server. The file name will be the alias for the server. The user will then have to manually distribute the credentials to the server or the clients that will need access.

Chapter 5
Implementation

In addition to the specification of Hermod, this thesis provides a reference implementation. The implementation is written in the Rust programming language. By providing a reference implementation it allows for testing and benchmarking of the protocol, making a comparison to existing solutions easier.

The reference implementation allows for file transfer, key generation and key sharing. Key sharing is supported to enable users to share their keys with a server similar to *ssh-copy-key*, removing the need to do key management manually.

Rust

Rust is a recently released systems programming language, initially developed by the Mozilla Foundation [32, 33]. The Rust programing language allows developers to create fast and memory-efficient programs with no runtime and with no garbage collector. Rust comes with a rich type system and with an ownership model for memory allocations. The ownership model guarantees both memory-safety and thread-safety at compile time, eliminating multiple classes of bugs during compilation.

Both Microsoft and Google reports that more than 70% of all bugs in Windows 10 and Chrome are related to memory-safety [45, 7]. By using Rust (and its ownership model) for the implementation, memory-safety bugs should be scarce, ensuring the implementation is reliable and stable.

Rust also supports writing asynchronous applications, through its zero-cost *futures*, which comes in handy when writing network applications. For a short introduction to Rust *async-await* I recommend reading the blog post *Async-await on stable Rust* [2]. Our reference implementation relies on the *async-std* library for asynchronous I/O operations and for running *tasks* [3, 46].

5.1 Overview

5.1.1 Noise

Hermod uses the *snow* crate for the Noise implementation [10]. Snow supports the latest revision of the Noise Protocol Framework [25].

The *snow* crate provides us with a builder struct, allowing us to specify the Noise Protocol and which keys we should use. This struct can then build a *HandshakeState* object that will construct all the necessary messages our Noise pattern needs for its handshake. After a successful handshake the *HandshakeState* can be transformed into a *TransportState* that can encrypt and decrypt byte buffers.

After a TCP connection is established between the client and the server, an Endpoint object is created. The Endpoint object wraps the TCP connection and a *TransportState* object from *snow*. This gives us a simple API to use; we just need to call the *send* method on the Endpoint object and pass in a buffer to encrypt. The endpoint object will then encrypt the message using the *TransportState* object before sending the message through the TCP connection. Receive is just as simple: just call the *recv* method on the Endpoint which then reads from the TCP connection, decrypts the received bytes read and passes a message back to the caller.

5.1.2 Server

The server is written in an asynchronous fashion using Rust's built in support for *async-await* and supports both logging and running as a background daemon. The central part of the server is the main loop listening for new incoming TCP connections and spawning a new async task for each new connection. Each connection then performs a handshake, using Noise with the help of the *snow* crate, before entering its own listening loop that receives and handles incoming messages from the client.

5.1.3 Client

The client is just as the server written in an asynchronous fashion. It initiates a TCP connection to the server before authenticating itself to the server and establishing a secure tunnel using Noise. The client object is initiated with a config that contains a list of files, if it is an upload or download request and information about which server to connect to.

5.1.4 Requests

Requests are structs that contain a source path, a destination path and a request type, download or upload. The source path is the path of the file that is to be uploaded or downloaded and the destination path points out where to store the uploaded or downloaded file. The request type specifies if it is an upload or a download that should take place.

The request structs can be serialized using the *bincode* format [8]. This ensures that the structs take up minimal space when transmitted over the secure channel while also allowing the structs to be easily extended in the future.

File data is transferred in chunks and the data is sent and received in order.

Chapter 6
Result

This chapter presents the result from the thesis. Both security and performance related results are presented.

6.1 Security Overview of Hermod

This section provides an insight into the security provided by Hermod.

6.1.1 Authentication

When using Hermod the authentication is based on elliptic-curve Diffie-Hellman, which assumes that the private key is stored and handled securely. Should the secret key of any involved party become compromised the party's communication can be forged.

As Hermod uses the KK pattern the first handshake message is vulnerable to a key impersonation attack, since the initiator cannot be sure the message is delivered to the correct recipient. It is only when the initiator receives a response it can be reassured it is talking to the correct receiver.

Even though Noise can verify that a matching static key pair is used in the handshake, the Hermod server must still decide on whether or not the key should be allowed access. This is why a random 32-bits identification token is included in the handshake. The connecting client must provide a matching token and key pair to the server for a successful authentication. If the identification is unknown to the server, the client is not allowed access.

If the token and the key or the hostname and the key does not match the authentication must fail.

6.1.2 Confidentiality

The Hermod protocol provides confidentiality by using AEAD, more specifically is uses the ChaChaPoly1305 algorithm. This assures that packets arrive encrypted, integrity protected and authenticated.

A goal of the Noise Protocol Framework is to ensure that encrypted data being sent should be indistinguishable from random noise. An attacker should not be able to draw any conclusions about the plaintext sent through the secure tunnel

by eavesdropping on a file transfer. Due to the fact that messages are not padded when sent, a passive attacker could gain knowledge about message length.

After a successful handshake, the KK pattern ensures that the following encrypted data is provided *message secrecy* and *strong forward secrecy*. As long as the ephemeral private keys of both the client and the server stay private, the encrypted data cannot be decrypted.

6.1.3 Data Integrity

By using the ChaChaPoly1305 algorithm for AEAD we gain both encryption and data integrity in one step. The Poly1305 algorithm provides authentication and integrity protection by calculating a 128-bits tag over the ciphertext. When decrypting the received message, the message is first checked by the Poly1306 algorithm and if the tags do not match, the message can be discarded. This ensures that any message that gets tampered with or manipulated in any other way, will not be accepted.

6.1.4 Trust

As with all protocols using asymmetric cryptography a certain level of trust is needed. Both involved parties must take necessary precautions in order to ensure that their private keys stay secure.

In Hermod it is also vital that the client's token is stored securely and that the client's *known_ servers* and the server's *authorized_ clients* file is not tampered with. Any manipulation of this file would either allow an attack or render no one available to connect.

6.1.5 Replay and Key Reuse

Since the protocol uses ephemeral keys, packets from one session cannot be replayed in a later session. In order to replay a handshake, an attacker must have access to the responder's static private key.

If the ephemeral keys are reused this would come with catastrophic consequences. The patterns' validity rules do not allow for patterns that reuse ephemeral keys. Should keys be reused anyway during the communication an attacker could gain knowledge about the data that is being sent.

6.1.6 Identity Hiding

The identification token is sent in plain text in the handshake initiation message. This means an attacker eavesdropping on the communication gets full knowledge about a client connecting to a server. Since the client uses a unique identification token for every server, an attacker gets no knowledge if the same client talks to different servers.

6.1.7 Configuration

With Hermod there is little need for configuration to have a secure system. Hermod removes the need for disabling insecure algorithms and authentication methods.

As Hermod uses public static keys combined with an identification token for client authentication it is important that they are stored and distributed securely. Should an attacker get access to both the key and the token they get full access to the server.

For server authentication the server's public key is used combined with the hostname. Just as with the client's credentials it is important that the server's credentials are handled securely when it comes to storage and distribution.

6.2 Security comparison between Hermod and SSH/SFTP

This section aims to compare the security properties of Hermod with those of SSH/SFTP.

From the table below we can observer that Hermod and SSH/SFTP comes with similar security properties with some minor differences. The protocols differs in which algorithms are supported for ensuring Forward Secrecy and in how the protocols provide data confidentiality and integrity. They also differ in how new or unknown keys are handled and in the size of their cipher suite.

Hermod:

- Public key Authentication
- Forward Secrecy
  - ECDH
- Data Confidentiality & Integrity
  - AEAD
- No 'Trust on first use'
- Minimal cipher suite

SSH/SFTP:

- Public key Authentication
- Forward Secrecy
  - DH or ECDH
- Data Confidentiality & Integrity
  - Encryption + MAC
- No 'Trust on first use'
- Large cipher suite

6.3 Performance

This section will provide some insight into the performance of Hermod and how it relates to SFTP. Both transfer times and data sent by the protocols will be looked on.

6.3.1 Transfer time

Setup

The tests are done between two lxd containers running on an AMD Ryzen 2700x CPU. The times are measured using the *hyperfine* tool, which provides us with

statistical analysis over the execution time from multiple runs [9]. *Hyperfine* runs the benchmark a minimum of 10 times or until it can provide an accurate result from a statistical standpoint.

When testing the transfer times four scenarios are used. The four scenarios are presented below.

1. Sending a file with size 10GB
2. Sending a file with size 1GB
3. Sending a file with size 500MB
4. Sending a file with size 10KB
5. Sending the Hermod source directory, 18 files with sizes ranging from 331 bytes to 21K for a total of 108KB.

In order to get a better understanding of how Hermod compares to SFTP when it comes to performance it is benchmarked against both *scp* and *sftp*. Both *scp* and *sftp* uses the Secure File Transfer Protocol but provide two different interfaces for it, with *sftp* offering a more interactive interface. By including both *scp* and *sftp* a deeper understanding of how different interfaces can affect the performance of a protocol is reached. Both *scp* and *sftp* used the *chacha20-poly1305@openssh.com* extension for encryption.

Result

The data provided by *hyperfine* after running the benchmarks is presented in Table 6.1 on Page 39. The result can also be seen in Figure 6.1 on Page 39 for larger files and Figure 6.2 on Page 40 for smaller files.

When looking at the performance of Hermod relative to SFTP, we can quickly see that when transmitting small files Hermod outperforms SFTP with a factor close to 25 When we increase the file size to 500MB, 1GB and 10GB the gap between Hermod and *scp* closes somewhat, but Hermod is still comfortably ahead when looking at the average times.

The one result that stands out, is the result from *scp* for the 10GB file. The standard deviation for *scp* is much larger and so is its maximum time compared to Hermod and *sftp*. This can probably be attributed to some outer disturbance, especially as the minimum and the average time does not stand out compared to the other applications. SCP even has the best minimum time of the three applications, this further points to that its average time is skewed by a few round in which the system was affected by some outer disturbance.

Application	File	Mean	Standard deviation	Min	Max
Hermod	10G	41.94	0.94	40.45	43.49
Hermod	1G	4.34	1.09	3.94	7.44
Hermod	500M	1.94	0.01	1.92	1.96
Hermod	10K	0.01	0.0	0.01	0.02
Hermod	Hermod source	0.02	0.0	0.02	0.03
scp	10G	47.7	11.61	39.51	72.31
scp	1G	4.65	0.98	4.07	7.38
scp	500M	2.46	0.09	2.39	2.68
scp	10K	0.55	0.02	0.52	0.57
scp	Hermod source	0.54	0.02	0.52	0.59
sftp	10G	45.07	1.44	42.79	47.83
sftp	1G	5.2	0.72	4.88	7.22
sftp	500M	2.77	0.24	2.62	3.43
sftp	10K	0.54	0.03	0.52	0.58
sftp	Hermod source	0.55	0.02	0.53	0.58

Table 6.1: Average execution time for the benchmark, as well as the standard deviation, the minimum and maximum execution time, as reported by Hyperfine.

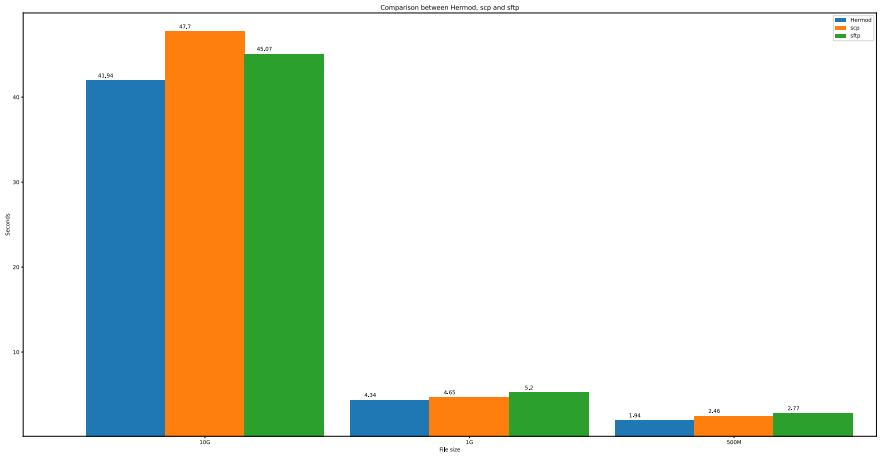

Figure 6.1: Comparison over average transfer times for large files in seconds.

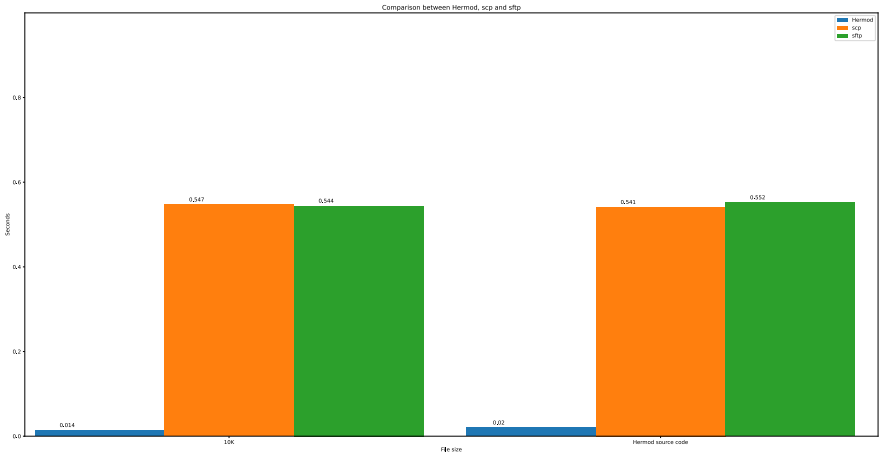

Figure 6.2: Comparison over average transfer times for small files in seconds.

6.3.2 Data sent

In Table 6.2 and Table 6.3 the number of packets and bytes sent when transferring two different files are shown. This gives a good view of the overhead added by Hermod and SFTP when it comes to transferring files. As for the transfer times benchmark, both the *scp* and the *sftp* applications are included when looking at SFTP.

The number of bytes and the number of packages sent are captured using tcpdump.

Program:	Hermod	scp	sftp
File size in bytes:	1000	1000	1000
Packets sent:	9	34	38
Packets received:	8	26	27
Packets total:	17	60	65
Bytes sent:	1648	6017	6401
Bytes received:	493	4485	4897
Bytes total:	2141	10502	11298

Table 6.2: Comparison of packets and bytes sent when uploading a file with 1000 bytes.

Program:	Hermod	scp	sftp
File size in bytes:	100000	100000	100000
Packets sent:	16	103	111
Packets received:	12	57	52
Packets total:	28	160	163
Bytes sent:	101031	108785	109753
Bytes received:	701	6133	6383
Bytes total:	101732	114918	116146

Table 6.3: Comparison of packets and bytes sent when uploading a file with 10000 bytes.

Chapter 7
Discussion

This chapter will evaluate and analyze the specification of the Hermod secure file transfer protocol. We will also discuss building protocols using Nosie and the reference implementation. The focus will be on how Hermod compares to SFTP when it comes to security and performance, and the benefits of using the Noise Protocol Framework.

7.1 Noise

The Noise Protocol Framework has proven itself to be very useful when designing Hermod. Instead of designing a state-machine for authentication and key derivation we have been able to focus on choosing a Noise pattern that fits Hermods security needs.

When using Noise it is important to remember that the authentication that is done when using a pattern with static keys only verifies that the party possesses the corresponding private key. For user authentication it is important for the protocol to include some other way of identification to ensure that the static key is allowed to connect.

The Noise patterns make it easy to find a pattern that fits your needs when creating a new protocol. As each pattern comes with its own set of payload security properties and identity hiding, it is important that you choose the correct pattern. It is also important to understand the implications of using the chosen pattern and what security properties it comes with. Choosing the wrong pattern can lead to devastating consequences, depending on the use case.

Due to the small and focused scope of Noise, it is very easy to incorporate it when creating a new protocol. Noise gives you a way of creating a secure channel and encrypts and decrypts the data provided. The developer is still in charge over implementing user authentication and for sending and receiving packets. The fact that Noise operates on raw byte buffers makes it "message type" agnostic, allowing the developers to decide what kind of message format should be used, e.g. JSON.

Another huge benefit of using Noise when creating new protocols is that all protocols that use Noises have the same process for authentication and key exchange. This speed up the review process of a new protocol as focus can be on ensuring that the correct Noise pattern is being used depending on the security

level wanted. Noise removes the need for understanding and reviewing a new process or state machine for authentication and key negotiation for each new protocol. The Noise Protocol Framework also facilitates the implementation of a protocol. If a verified Noise implementation is used, we can be sure that the authentication and key exchange is done securely and correctly.

7.2 Security

When comparing Hermod with SFTP, both protocols provide similar security properties. Both protocols provide confidentiality, data integrity and authentication for the data that is sent through the secure tunnel. In addition, both protocols generate session keys with strong or perfect forward secrecy, ensuring that past file transfer sessions stay secure if the private keys should be compromised.

The main difference when it comes to the security is the absence of configuration needed in order to get a secure Hermod client and server.

7.2.1 Credentials

Both the server and the client rely on the credentials of the other party being stored securely. Should an attacker gain access to the client's credentials the attacker would be able to fetch files from an unknowing server. This is a problem that can be found in SSH as well.

A potential solution to this problem would be to use some form of two-factor authentication using data that is not stored on the client's machine. This would however make the application harder to use in scripts and increase the complexity of the protocol.

As Hermod is using static keys in combination with an identification token from the client for authentication and key derivation we cannot completely get away from storing credentials for the clients and server. In SSH, the keys can be optionally encrypted for added protection. As this does not affect the protocol itself, it is better to leave the decision on how to securely store the keys to each act of implementation. By only specifying the format, developers are free to choose their own method of storing the credentials when implementing the protocol. This allows the credentials to be stored in clear text or encrypted on a disk, smart card or USB drive.

7.2.2 Configuration

A goal for Hermod was to remove the need for extensive configuration when setting up a new client or server. In comparison to SFTP, a user does not need to worry about enabling secure ciphers and options while disabling the insecure ones. Hermod provides no way for the user to (miss-) configure a system and aims to provide sane ciphers options by default.

In Hermod the only configuration needed for the server is to generate a static key pair and for the client to generate a static key pair and an identification token for each server it wants to conduct file transfers with. There is no need to specify which algorithms to use, what kind of authentication methods are as in SSH. Users

are not exposed to or forced to interact with any settings that affect the security of Hermod. Another benefit is that less background knowledge is needed to set up a secure system.

The decision to remove most of the configuration found in SSH and SFTP allows Hermod to support fewer ciphers, making the attack vector smaller. The support for fewer configuration options should also make the implementation simpler and by extension easier to review and verify. A protocol with lots of configuration options opens itself up to segregation as different implementations could support only part of all the options. This could make different implementations incompatible with each other. Creating a protocol that is easy to both implement and review allows for faster adoption as there are less edge cases to handle and implement.

When designing a new protocol or application there is a fine line between being configurable, customizable and hard to use. Providing the user with the option to configure and customize an application for its use case can be a good thing. Crossing the fine line towards being hard to use when designing secure protocols and applications can have devastating consequences. Users could for example configure the service to use insecure ciphers or allow access to unwanted parties. Increasing the number of options available for configuration and customization also increases the number of edge cases and potential bugs that need to be handled, thereby increasing the specification and implementation.

7.3 Implementation

The reference implementation relies on Rust's built in support for asynchronous programming. This should allow for a better hardware utilization at the cost of some added latency. In order to handle incoming connections not only asynchronously but also in parallel, a multi-threaded asynchronous runtime is used. Rust's support for asynchronous programming is still quite new which in return makes the surrounding ecosystem and libraries new. This means that there is still a lack of knowledge of when asynchronous programming is preferable over more traditional multi-threading. As we are implementing a file transfer protocol, we should be mostly IO-bound, which should mean that asynchronous programming should work just fine. It is also a trade-off between chasing an efficient implementation and minimizing the latency for the file transfers.

As our implementation targets Linux, there is new interesting work being done that adds support for high-performance asynchronous I/O to the Linux Kernel. The new interface, *io_uring*, allows for asynchronous buffers I/O and could offer an improvement over the current I/O API. The interface is however quite new, added in kernel 5.5 and 5.6, which means that we would either limit ourselves to users with the latest kernel or we would increase the size of the implementation and increase the code base. Due to the fact that *io_uring* is still new, there is still an uncertainty as to how big of an improvement the new interface will bring and what bugs and quirks exist. As Hermod is a file transfer protocol, any improvements to doing I/O operations would indirectly benefit our protocol.

7.4 Performance

The fact that the gap between Hermod and *scp/sftp* decreases as the file size grows, can probably be attributed to the fact that the overhead matters less the greater the file size is. As we can see in Tables 6.2 and 6.3, the overhead from Hermod is much lower compared to SFTP for smaller files. When the file size increases, the impact the protocols overhead has decreased. When the overhead constitutes smaller percentages of the transmitted data, the difference between the various protocols and applications should become smaller, which we can observe in Figure 6.1 and Figure 6.2.

As SFTP is an older protocol, it has had more time for optimizations compared to Hermod, which should mean that the performance of Hermod can be improved on. While the overhead from the handshake cannot be removed or optimized away, the file transfer and the reading and writing to files can probably be better optimized in Hermod. Analyzing the performance of Hermod and understanding where it spends time could lead to improvements in the performance moving forward.

7.5 Future Work

Since the latest release of the Noise Protocol Framework a new hash algorithm from the Blake family has been released. The new algorithm, Blake3 [15], offers increased performance and comparable security to the Blake2 algorithms [15]. A huge part of the increased performance comes from the fact that there is a significant reduction in the hashing rounds in Blake3 compared to Blake2. This is quite controversial and needs more research but Blake3 could be a good addition to Noise, especially as it could remove the need for users of Blake2 to choose between using a 32-bit or 64-bit version of Blake2.

Performance gains in cryptographic algorithms is always appreciated as it would lower the latency and speed up the file transfers, especially for larger files sizes. However, performance gains should not jeopardize the security the algorithm provides. Another benefit of using Blake3 instead of Blake2, is that in Blak3 the creators have been able to remove the need for a separate implementation for 32-bit or 64-bit platforms. This makes the algorithm easier to use in libraries and developers do not need to sacrifice one platform.

Hermod would also benefit from secure credentials distribution. Currently credentials are distributed manually; by extending the protocol with an API for distributing the credentials users could share the credentials with a trusted server without needing physical access or using a third-party application.

Chapter 8
Conclusions

In this thesis a proof-of-concept for a new protocol for secure file transfer was created. The new protocol tries to learn from and mitigate some of the drawbacks found in SSH and SFTP.

In order for us to create a new protocol for secure file transfer that rivals SFTP we first looked into how SSH works and what security it provides SFTP. We followed that up with an introduction of the Noise Protocol Framework and looked at the security Noise provides. After we had introduced Noise, we created a specification for our new protocol and created a reference implementation for testing.

With Hermod we have managed to provide a file transfer protocol that is small, simple, secure and comes with little overhead. The small overhead that Hermod comes with, allows it to really shine for smaller files while still providing an improvement over existing solutions for larger files.

By using the Noise Protocol Framework in Hermod, there is no need to create a new method for public key authentication and key exchange. This allowed us to spend more time focusing on the file transfer and the implementation which simplified and speeded up the process. Throughout this thesis we have managed to show that even though the Noise Protocol Framework is still young, it has great potential and provides a lot of benefits when working with or developing cryptographic protocols and secure tunnels.

Bibliography

[1] *Aspera*. Last Accessed 2020-04-30. URL: https://www.ibm.com/products/aspera.

[2] *Async-await on stable Rust!* Last Accessed 2020-04-29. URL: https://blog.rust-lang.org/2019/11/07/Async-await-stable.html.

[3] *async-std*. Last Accessed 2020-04-29. URL: https://async.rs/.

[4] D. Bider. *Use of RSA Keys with SHA-256 and SHA-512 in the Secure Shell (SSH) Protocol*. RFC 8332. RFC Editor, Mar. 2018.

[5] *BOLT 8: Encrypted And Authenticated Transport*. Last Accessed 2020-05-04. URL: https://github.com/lightningnetwork/lightning-rfc/blob/master/08-transport.md.

[6] *Chapter 6 Introduction to Public keycryptography*. Jan. 16, 2020. URL: https://www.eit.lth.se/fileadmin/eit/courses/edi051/lecture_notes/LN6a.pdf.

[7] Catalin Cimpanu. *Chrome: 70% of all security bugs are memory safety issues*. Last accessed 2020-05-27. URL: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[8] *Crate bincode*. Last Accessed 2020-04-29. URL: https://docs.rs/bincode/1.2.1/bincode/.

[9] *Crate hyperfine*. Last Accessed 2020-04-29. URL: https://crates.io/crates/hyperfine.

[10] *Crate snow*. Last Accessed 2020-04-29. URL: https://docs.rs/snow/0.7.0-alpha4/snow/index.html.

[11] Richard Silverman Daniel J.Barret. *SSH The Secure SHell: The Definitive Guile*. O'Reilly, 2001. ISBN: 0596000111.

[12] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnel*. Whitepaper. Wireguard. URL: https://www.wireguard.com/papers/wireguard.pdf.

[13] Niels Ferguson et al. *Cryptography engineering: design principles and practical applications*. Wiley, 2010. ISBN: 9780470474242.

[14] P. Ford-Hutchinson. *Securing FTP with TLS*. RFC 4217. http://www.rfc-editor.org/rfc/rfc4217.txt. RFC Editor, Oct. 2005. URL: http://www.rfc-editor.org/rfc/rfc4217.txt.

[15] *Github: Blake3*. Last Accessed 2020-05-11. URL: https://github.com/BLAKE3-team/BLAKE3/.

[16] Peter Gutmann. *Do Users Verify SSH Keys?* Tech. rep. Aug. 2011. URL: https://www.usenix.org/system/files/login/articles/105484-Gutmann.pdf.

[17] Mathias Hall-Andersen et al. "NQUIC: Noise-Based QUIC Packet Protection". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. EPIQ'18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 22–28. ISBN: 9781450360821. DOI: 10.1145/3284850.3284854. URL: https://doi.org/10.1145/3284850.3284854.

[18] M. Thomson J Iyengar. *QUIC: A UDP-BAsed Multiplexed and Scure Transport draft-ieft-quic-transport-27*. RFC. RFC Editor, Feb. 2020. URL: https://tools.ietf.org/html/draft-ietf-quic-transport-27.

[19] *KK Handshake Pattern Analysis*. Last Accessed 2020-05-07. URL: https://noiseexplorer.com/patterns/KK.

[20] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. "Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols." In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P), Security and Privacy (EuroS&P), 2019 IEEE European Symposium on* (2019), pp. 356–370. ISSN: 978-1-7281-1148-3. URL: http://ludwig.lub.lu.se/login?url=https://search-ebscohost-com.ludwig.lub.lu.se/login.aspx?direct=true&db=edseee&AN=edseee.8806757%5C&site=eds-live%5C&scope=site.

[21] A. Langley, M. Hamburg, and S. Turner. *Elliptic Curves for Security*. RFC 7748. RFC Editor, Jan. 2016.

[22] *Noise Explorer*. Last Accessed 2020-05-05. URL: https://noiseexplorer.com/.

[23] *Noise Protocol Framework*. Last Accessed 2020-01-16. URL: http://noiseprotocol.org/.

[24] *NTCP 2*. Last Accessed 2020-05-03. URL: https://geti2p.net/spec/ntcp2.

[25] Trevor Perrin. *The Noise Protocol Framework*. Reversion 34. July 2018. URL: http://noiseprotocol.org/noise.html.

[26] *Public Key authentication for SSH*. Last Accessed 2020-05-03. URL: https://www.ssh.com/ssh/public-key-authentication.

[27] *rcp*. Last Accessed 2020-05-03. URL: https://www.ssh.com/ssh/rsh.

[28] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018.

[29] E. Rescorla and B. Korver. *Guidelines for Writing RFC Text on Security Considerations*. BCP 72. http://www.rfc-editor.org/rfc/rfc3552.txt. RFC Editor, July 2003. URL: http://www.rfc-editor.org/rfc/rfc3552.txt.

[30] *rlogin*. Last Accessed 2020-05-03. URL: https://www.ssh.com/ssh/rlogin.

[31] *rsh*. Last Accessed 2020-05-03. URL: https://www.ssh.com/ssh/rcp.

[32] *Rust*. Last Accessed 2020-04-29. URL: www.rust-lang.org.

[33] *Rust language*. Last Accessed 2020-04-29. URL: https://research.mozilla.org/rust/.

[34] Bruce Schneier. *Applied cryptography : protocols, algorithms, and source code in C*. 5th ed. Wiley, 1996. ISBN: 0-8493-8523-7.

[35] *Session Key*. Last Accessed 2020-05-03. URL: https://www.ssh.com/ssh/session-key.

[36] *SFTP – SSH Secure File Transfer Protocol*. Last Accessed 2020-05-25. URL: https://www.ssh.com/ssh/sftp/.

[37] Nigel P. Smar. *Algorithms, Key Size and Protocols Report (2018)*. Tech. rep., 4g. URL: https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf.

[38] Nigel P. Smart. *Cryptography : an introduction*. 3rd ed. McGraw-Hill, 2003. ISBN: 0077099877.

[39] *SSH (Secure Shell)*. Last Accessed 2020-04-30. URL: https://www.ssh.com/ssh/.

[40] *SSH Keys*. Last Accessed 2020-05-25. URL: https://www.ssh.com/ssh/key/.

[41] *SSH(1)*. Last Accessed 2020-05-03. URL: https://man.openbsd.org/ssh.

[42] Marc Stevens et al. *Announcing the first SHA1 collision*. Feb. 2017. URL: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html.

[43] *The Double Ratchet Algorithm*. May 25, 2020. URL: https://signal.org/docs/specifications/doubleratchet/.

[44] *The fasp solution*. Last Accessed 2020-04-30. URL: https://downloads.asperasoft.com/en/technology/fasp_solution_3/the_fasp_solution_3.

[45] Liam Tung. *Microsoft: Here's how we're killing a class of memory security bugs in Windows 10*. Last accessed 2020-05-27. URL: https://www.zdnet.com/article/microsoft-heres-how-were-killing-a-class-of-memory-security-bugs-in-windows-10/.

[46] *Welcome to async-std*. Last Accessed 2020-04-29. URL: https://book.async.rs/overview/async-std.html.

[47] Andrew Whalley. *SHA-1 Certificates in Chrome*. Nov. 2016. URL: https://security.googleblog.com/2016/11/sha-1-certificates-in-chrome.html.

[48] *WhatsApp Encryption Overview*. Whitepaper. Wireguard, 2017. URL: https://scontent.whatsapp.net/v/t61.22868-34/68135620_760356657751682_6212997528851833559_n.pdf/WhatsApp-Security-Whitepaper.pdf?_nc_sid=41cc27&_nc_ohc=uyKLZq9R_nQAX8JcADy&_nc_ht=scontent.whatsapp.net&oh=4df2f77787eb45c6894233cc1d121b45&oe=5EB1D5D3.

[49] T. Ylonen and S. Lethinen. *SSH FIle Transfer Protocol draft-ietf-secsh-filexfer-02*. RFC. RFC Editor, Oct. 2001.

[50] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. http://www.rfc-editor.org/rfc/rfc4252.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4252.txt.

[51] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. http://www.rfc-editor.org/rfc/rfc4254.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4254.txt.

[52] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. http://www.rfc-editor.org/rfc/rfc4251.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4251.txt.

[53] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. `http://www.rfc-editor.org/rfc/rfc4253.txt`. RFC Editor, Jan. 2006. URL: `http://www.rfc-editor.org/rfc/rfc4253.txt`.

Appendix A

Source Code

The code for the reference implementation is hosted at github, `https://github.com/MarkusAkesson/Hermod`.

Printed by Tryckeriet i E-huset, Lund 2020

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2020-773
http://www.eit.lth.se