Performance Evaluation of Security Protocols in LibP2P Connections

Bachelor Thesis

by

Luca Weist
3084294

in fulfillment of requirements for degree
Bachelor of Science (B.Sc.)

submitted to
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

in degree course
Computerscience (B.Sc.)

First Supervisor:	Prof. Dr. Peter Martini University of Bonn
Second Supervisor:	Prof. Dr. Michael Meier University of Bonn
Sponsor:	Daniel Baier University of Bonn

Bonn, July 15, 2022

Abstract

The peer-to-peer network model constitutes a widespread alternative to the classic client-server model of network structures. There are various frameworks and libraries for developing such networks. One of the more established ones is libp2p, on which various real-world peer-to-peer applications are based. It allows for modular combinations of utilised network protocols, such as transport protocols, content routing protocols and security protocols. As efficiency is of course always relevant in a computer network, it is of interest how the different protocols perform in comparison to each other. This thesis focuses on libp2p's security protocols, conducting a performance analysis of each officially implemented protocol, TLS, Noise and Secio, as well as an additional custom implementation of the Signal protocol, created specifically for this work. The protocols' time performance, space performance as well as the workload of the CPU and power draw are considered in the analysis, along with various other more minor aspects.

Contents

1 Introduction

With the digital world having taken over as the primary channel to communicate and exchange information, only gaining increasingly more prominence by the day, securing the confidentiality of data exchanged through this channel is more important than ever before. Even so, this goal is often not met to a satisfactory degree.

One possible source of breaches of confidentiality is the fact that once data is sent to a server, it is impossible to know how exactly the data will be treated. For example, a recent freedom-of-information request revealed the extent to which the FBI may legally access data sent over various Instant Messaging systems. In the case of WhatsApp, they are able to view source and target of each message sent, along with other various user data. [33]

One network model that can help alleviate a lot of these concerns is the peer-to-peer network model. In contrast to the more common client-server model, in which a server listens for requests from (up to a very large amount of) clients and reacts once queried, all participants in a peer-to-peer network assume an equal role [78]. The participants in such a network structure are called "peers" or "nodes" [22]. Each peer can send requests to others in the network, but also listens for incoming request from other peers, effectively taking both the role of a client and that of a server [78]. The amount of connections each individual peer may have within the network is, in theory, unlimited [26]. Possible use cases for this kind of network model are, among others, Instant Messaging systems or file sharing networks [5, 6]. In both of these examples, securing data confidentiality is inherently an important factor.

This decentralized structure of peer-to-peer networks bring various advantages with it. Primarily, since there is no central middleman that all sent data has to pass through, something like the WhatsApp example mentioned above would not be possible. Additionally, peer-to-peer networks come with a handful of other advantages over client-server networks such as cheaper and easier scalability and, depending on the scale of the network, stronger resistance against service failure [68].

With this in mind, it is easy to see why, in certain use-cases, peer-to-peer networks can offer themselves up as attractive alternatives to the client-server model. Of course, peer-to-peer networks come with their own set of disadvantages. A major one is the large performance loss that often comes with growth of the network, as it is likely that more connections need to be handled by the individual peers [68]. libp2p, an open-source framework and library that was originally part of the IPFS project [18], has already established itself as a tool for creating peer-to-peer networking applications [18], being used in the development of various blockchain applications [102, 25], including Ethereum 2.0 [24].

One of the most important decisions to make when developing a peer-to-peer network, especially in the context of attempting to optimize data confidentiality, is what cryptographic protocol to use. The cryptographic protocol, also referred to as security protocol, is mainly responsible for the encryption of the exchanged data [8]. Supplying the data with certificates of integrity and authenticity falls into the protocol's responsibility as well, as does the initial exchange of encryption keys [8]. libp2p offers various officially implemented security protocols as well as allowing for implementation of new ones using its API.

As mentioned, performance is a big concern for peer-to-peer networks. Since cryptographic processes are relevant for both time and space performance [21, 32], it is of interest to know how different cryptographic protocols compare in this regard.

This thesis aims to analyse and evaluate a set of implementations of cryptographic protocols for the Go [52] implementation of libp2p in consideration of their time- and space-performance as well as some other parameters. In order to achieve this, each of the three officially implemented security protocols [53, 34] are tested and benchmarked using Go's testing [64] framework alongside other various tools. In addition to the official libp2p implementations, the Signal protocol [93] is implemented for libp2p using a pre-existing library.

First, chapter Background introduces required background knowledge. This is followed by chapter Related Work which discusses work that has already been done by others relating to this thesis' subject. Afterwards, chapter Considered Protocols specifies what cryptographic protocols are analysed within this work. It presents the general protocols as well as the libp2p implementations specifically. Then, chapter Methodology defines the exact methodology that is used to gather the data points for the analysis, as well as giving a rough documentation of the implementation of the Signal protocol. Finally, the analysis results for each cryptographic protocol are presented in chapter Analysis.

2 Background

This chapter introduces necessary background knowledge. First, the main concepts utilized in cryptographic protocols are presented, including encryption, key exchange and authenticating mechanisms. Following that, the peer-to-peer network structure as well as its primary competitor, the client-server structure are described and compared to each other. Finally, the libp2p framework is introduced.

2.1 Cryptographic Protocols

In the field of computer science, protocols define a set of rules on how to handle and process data. Protocols are commonly used in networks, for which there are many different types of protocols, each serving a different purpose. For example, IP (Internet Protocol) is responsible for routing between different hosts and TCP (Transmission Control Protocol) guarantees that each packet of data is received correctly, whereas UDP (User Datagram Protocol), an alternative to TCP, delivers packets faster but much less reliably. [10]

Cryptographic protocols specifically are a subset of protocols which deal with cryptographic methods [8]. This can include a very wide range of functions, of which a few major and common ones, also referred to as cryptographic primitives [42], are explained below. Each cryptographic protocol analysed in this work fulfills all of these functions.

2.1.1 Encryption

The term encryption describes the process of obscuring a piece of data, which is referred to as plaintext, to the point where its original content is no longer discernible. This is commonly achieved using another piece of data, referred to as the encryption key. The encryption key is combined with the initial piece of data according to various set rules, defined by the specific kind of encryption used. This new piece of data is called the ciphertext. To decrypt the data, meaning to restore its original content, the decryption key is required. That key is once again applied according to the rules set by the encryption type. Encryption serves to ensure data confidentiality, where only the communication partners who possess the corresponding key can de- or encrypt messages sent between each other. [48]

Encryption strategies can be divided into two categories: symmetric and asymmetric encryption. Symmetric encryption is defined by the fact that the encryption and decryption process both make use of the same key, whereas asymmetric encryption utilizes a key pair made up of one key for encryption and one key for decryption. [90]

In the case of asymmetric encryption, each communicating party possesses a key pair made up of a public key and a private key. The public key is openly shared with the other party, allowing them to encrypt data using this key, which then in turn can only be decrypted again using the corresponding private key. Therefore, each party encrypts any data that it wants to send using the other party's public key and decrypts any received data using its own private key. Since the private keys are at no point sent over the network, and can not be efficiently derived from the public key, there is no direct way for an attacking third party to decrypt any of the messages. [39]

The opposite is the case for symmetric encryption. Here, the same key is used to encrypt and decrypt data and therefore both communication partners first have to agree on a shared key. This key, of course, should not be simply sent over the network, as it cannot be encrypted and therefore would have to be transmitted as plaintext and could be read by an observing third party. Instead, key exchange methods are used, described in the following section. [39]

The different processes for encrypting and decrypting with symmetric and asymmetric algorithms in a typical network situation, where packets of data are sent between two communicating parties, is illustrated in figure 1.

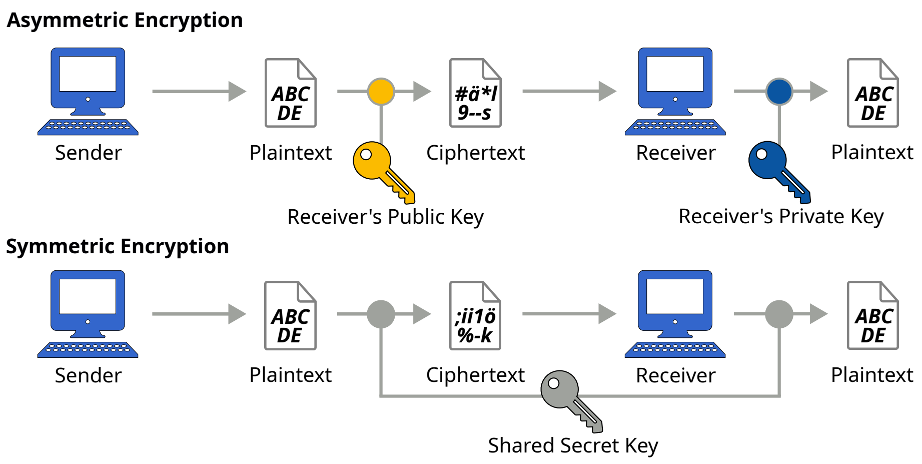

Figure 1: *Visualisation of both asymmetric and symmetric encryption [39]*

Generally, asymmetric encryption strategies are more cryptographically secure than symmetric ones [15]. On the other hand, symmetric encryption is usually easier to implement and encrypts data faster [15]. Each of these characteristics is however of course dependent on which specific encryption method and implementation is used.

2.1.2 Key-Exchange

When using symmetric encryption methods, all communication partners have to agree on a shared key in order to exchange and decrypt encrypted messages with each other. This key has to be confidential between the communicating parties and there cannot be simply exchanged as plaintext. The common technique for achieving this confidentiality, and also one of the earliest ones, is the Diffie-Hellman algorithm. [87]

The first step of the algorithm is for both parties to agree on two variables g, the so-called generator variable, and p, a very large prime number. These may be exchanged in plaintext over

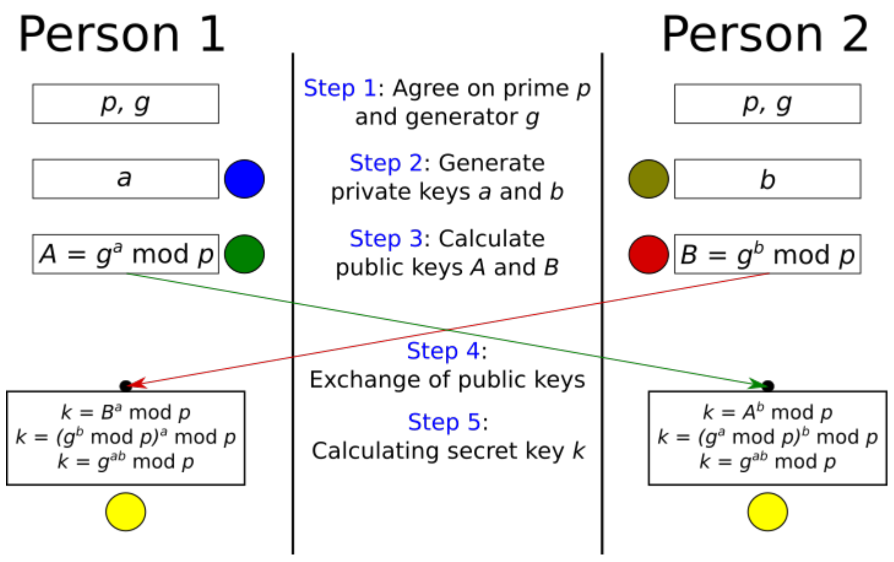
Person 1
Person 2
p, g
p, g
Step 1: Agree on prime p and generator g
a
b
Step 2: Generate private keys a and b
$A = g^a \bmod p$
$B = g^b \bmod p$
Step 3: Calculate public keys A and B
Step 4: Exchange of public keys
Step 5: Calculating secret key k
$k = B^a \bmod p$
$k = (g^b \bmod p)^a \bmod p$
$k = g^{ab} \bmod p$
$k = A^b \bmod p$
$k = (g^a \bmod p)^b \bmod p$
$k = g^{ab} \bmod p$

Figure 2: *Visualisation of the Diffie-Hellman algorithm [91]*

the network. Next, each party generates an asymmetric key pair. They exchange their respective public keys, once again as plaintext, and use their own private key together with the remote party's public key to generate a new key. This newly generate key is identical between both parties and serves as the shared key. As at least one of the private keys, neither of which has been sent over the network, is needed to generate the shared key, it is impossible for an observing third party to derive it using the exchanged data. The confidentiality of the shared key is therefore guaranteed. This process is illustrated in figure 2, along with more mathematical detail.

There are variations on the method presented here. A major one, which all of the protocols relevant to this thesis use exclusively (with the exception of TLS), is Elliptic-curve Diffie-Hellman [75]. The key-exchange process is very similar; the two parties each generate a key pair, send their public key over the network and calculate a shared key using their own private key and the received key. Mathematical details differ however, the symmetric key is generated using the formula $(a * G) * b = (b * G) * a$ which is a property of point on an elliptic curve [74]. Here, a and b are the two private keys and G is a so-called generator point, which is defined by the specific version of ECDH used. As $a * G$ is equivalent to the public key of the party possessing private key b, and vice versa, both parties can calculate the formula once public keys are exchanged. [75]

The key-exchange is part of the handshake the two communicating parties execute upon connecting with each other. In order for the handshake to be secure, it is also necessary to be able to authenticate the identity of the other party (for example to prevent a man-in-the-middle attack [67]). The method used by the relevant protocols to achieve this is described in the following section.

2.1.3 Authentication

One other major goal of cryptographic protocols is the ability to validate a given data packet's source as well as its integrity. Here, a received data packet is considered integrative if it is identical to the data packet that was sent. So, for example, a packet that has been altered by way of a third party listening in on and manipulating a connection would fail the integrity check. One way to ensure both authenticity and integrity is using message authentication codes, or MACs [77]. As MACs are the result of one-way hash-functions [107], the concept of hash functions is quickly introduced in the following paragraph, prior to an explanation of MACs specifically.

Hash-functions are functions that take an input of any length and project it on a fixed-length value, called the input's hash-value [92]. The defining property of one-way hash-functions is that, while hash-values are rather easy to compute, the inverse is not (efficiently) computable [106]. In addition to this, in order for a hash-function to be cryptographically viable, it needs to be collision-resistant [106]. A function is considered as such if it is impossible to efficiently determine two unequal inputs that are projected on the same hash-value [106]. The function should also be fully publicly known, to ensure it does fulfill these other conditions [82].

The usage of MACs requires the communicating parties to already be in possession of a shared key, assuming symmetric encryption is used. To calculate a MAC for any given message (which may, as mentioned prior, be of any length), the message along with the shared key are used as input for the used hash-function. The resulting hash-value is the MAC of the message. Along with the message itself, the MAC is sent to the communicating partner, who then may recalculate the MAC using the received message and the shared key. Assuming it matches the received MAC, the receiver will consider the authenticity of the message guaranteed, as the sender should be the only other party in possession of the shared key. Due to the collision-resistance of the hash-function, the integrity of the message is also confirmed. If it should be different from the received MAC, the message will be trashed. As it is purely dependent on the used key and the message itself, a MAC is also referred to as the "keyed checksum" of the message. This process is exemplified in figure 3. [107]

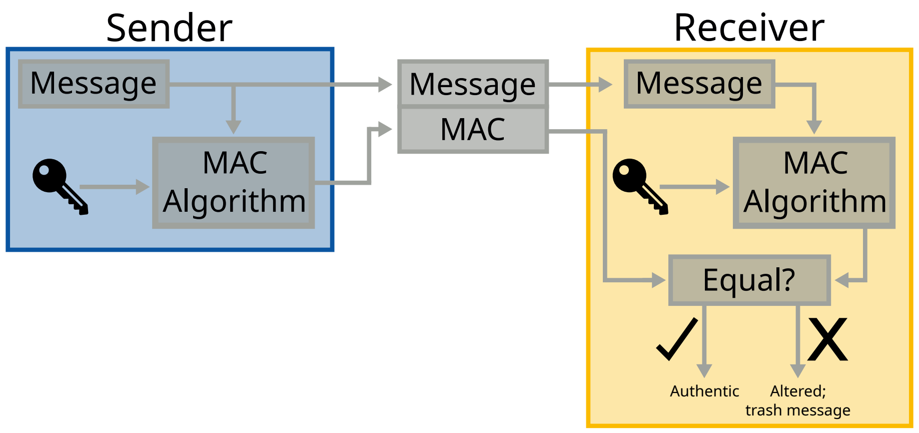

Figure 3: *Visualisation of a message authentication check using MACs and symmetric encryption [107]*

2.2 Network Models

Computer networks consist of two or more computers that are interconnected with each other, generally with the purpose of sharing resources or data.

Two often employed network structure models are the client-server model and the peer-to-peer model. The two following sections give a quick overview of each model, followed by a direct comparison of their respective advantages and disadvantages.

2.2.1 Client-Server Model

The client-server model is the most widespread network model, finding various uses on the internet as well [68].

Networks that apply the client-server model consist of the two name-giving entities; servers and their clients. These entities are strictly separated by their functions. In theory, servers are the sole provider of resources and data - clients do not share anything with other clients or servers. The server's responsibility is to listen for incoming connection requests from clients and to then to respond to received requests, assuming they are valid of course. The only action clients take is requesting said services from a given server. [89]

From this it also follows, at least in theory, that servers do not communicate with each other, nor do clients. In practice however there are often various types of servers, each with different purposes and acting on different layers, working together. For example, web servers are used to deliver web content to clients. Web servers in turn often rely on database servers to supply them with data. [2]

Below, figure 4 contains simple visual conceptualisations of both a client-server network as well as a peer-to-peer network. As shown in it, servers are designed to be able to handle large amounts of concurrent incoming (requests) and outgoing (responses) connections efficiently, often much more than the 16 displayed here [4]. Since the server is the sole provider of services and all connections are to it, the client-server model is also referred to as a centralized network model [43]. This also introduces the problem of the server acting as a single point of failure for the network. In practical applications, this problem is often alleviated by employing multiple servers providing identical services [47].

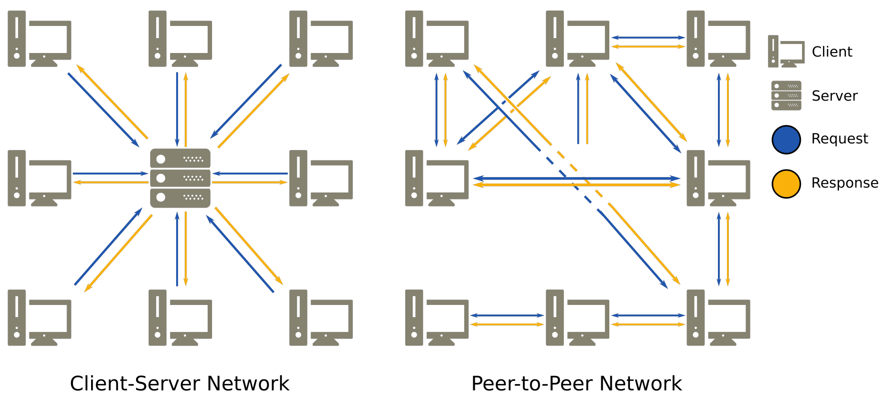

Figure 4: *Representation of theoretical client-server and peer-to-peer network [43]*

2.2.2 Peer-to-Peer Model

The peer-to-peer network model is, after the client-server model, the second most commonly employed paradigm for designing network structure [68]. It is used especially extensively for file-sharing software, but also for various other applications like chat systems and cryptocurrency [101, 76].

Unlike client-server networks, peer-to-peer networks do not make use of the concepts clients and servers, but instead of peers. Peers are also sometimes referred to as servents, a portmanteau of 'client' and 'server', as they take over the responsibilities and functionalities of both client and server in client-server networks. This means that, generally, every peer listens for incoming requests from other peers, while also being able to initiate and request connections from any given peer in the network. Peers are therefore, as the title suggests, 'equal' to each other in terms of functionality, in opposition to the more hierarchical relation of clients and servers. Because of this trait, peer-to-peer approaches are considered decentralized. [89, 68]

In his book "Peer-to-Peer Systems and Applications" [101], the author describes decentralized resource usage as well as decentralized self-organization as two of the main characteristics of peer-to-peer networks. "Decentralized resource usage" describes the equal distribution of desired resources, often data and processing power, analogous to the client-server network model, as well as the direct access peers have to these resources supplied by other peers. "Decentralized self-organization" refers to the fact that, in pure peer-to-peer systems, no centralized component is introduced to organize network traffic and connections between peers. [101]

It should however be mentioned that while generally peer-to-peer systems are "self-organizing [systems] of equal, autonomous entities" [101], there is not only this pure peer-to-peer approach, but also hybrid approaches. For these hybrid models, one or more servers are introduced into the network. Often, these are meant to handle peer discovery and aid traffic between peers. Peer discovery describes the process of learning a peer's identity, for example its IP address, in order to be able to start sending requests to it. [68]

This lack of centralization in peer-to-peer networks comes with various advantages and disadvantages over client-server systems. For one, as mentioned above in Client-Server Model, the server acts as single point of failure in client-server networks. Assuming there are multiple peers in the network that are in possession of, and sharing, a desired resource (for example, a specific file), this is not the case for peer-to-peer networks. The higher the amount of peers in a network sharing the desired resource, the lower the likelihood of a critical peer failure is. It follows that peer-to-peer networks are also generally more robust against denial-of-service attacks. They are also more easily scalable; adding another peer to an existing peer-to-peer network is trivial [68]. Installation and maintenance of a new peer is, generally, much cheaper than the same process is for a server too, as peers can be nearly any device with network access whereas servers use highly specialized, expensive hardware [68]. This comes with the infraction that a single average peer's performance is a lot lower than an average server's. However, in many cases peer-to-peer networks grow by online users voluntarily installing a device into the network, as with file-sharing networks, which would come at no direct cost for the operator of the network. As mentioned, peers can request connections from other peers in a shared network. Therefore, the amount of open connections each peer in a peer-to-peer network has at any point in time is arbitrary. This is reflected in figure 4.

While not a technical advantage in and of itself, the fact that data sent over peer-to-peer networks is usually less monitored and therefore harder to track is often presented and understood as one of the major advantages of peer-to-peer networks. But as stated, peer-to-peer systems also have direct disadvantages when compared to ones based on the client-server model. For one, as there is no central database, central malware controls are impossible, leading to a large amount of malicious peers in some networks [68]. This is an especially large issue in file-sharing networks, for which peer-to-peer networks are commonly employed, as mentioned above. The same lack of central database also leads to the inability of creating coherent backups of all resources being shared on the network [68]. Another problem is that effective peer-to-peer network are dependent on adaption rate; when a large amount of peers leave the network, certain resources will end up being unavailable. In contrast, servers are available as long as the responsible operator keeps it running [68]. Depending on network topology, the scalability of peer-to-peer networks can also lead to diminishing performance. For example, many peers may request a resource but do not keep sharing it themselves once they have received it. This is a common problem in file-sharing networks, where peers download files from other peers but leave the network once they are finished downloading it. A concise overview of these core advantages and disadvantages can be seen in figure 1.

Table 1: *Overview of some major advantages and disadvantages of the peer-to-peer network model when compared to the client-server model*

Advantages Peer-to-Peer Model	Disadvantages Peer-to-Peer Model
Generally more resistant against service failure	No central control →higher risk of malicious influence
Easily and cheaply scalable	No central backups
Flexible	Possible larger performance loss with growth of network
Network traffic harder to track	

2.3 libp2p

Originally beginning development as part of the IPFS (InterPlanetary File System) project by Protocol Labs, which was first released in 2015, libp2p is a modular, open-source protocol stack and library for the development of peer-to-peer networking applications [9, 18]. It has since been decoupled from the IPFS project and now is its own stand-alone module [18].

libp2p is in use in various real-world peer-to-peer applications. A framework for building blockchains, substrate, is built on the official rust implementation of libp2 [102], which in turn the cryptocurrency Polkadot is based on [83]. Also built on libp2p is the cryptocurrency Filecoin [25]. The current consensus specs for the planned Ethereum 2.0 also include libp2p as one of its basic building blocks [24].

The main goal of libp2p, as stated in an article published on Medium [104], is to horizontalize the commonly used OSI model in order to modularize the various network protocols and give developers a "frame in which all of these protocols can co-exist and co-operate" [104]. libp2p therefore offers, among others, transport protocols, identity protocols, content routing protocols and cryptographic protocols, more commonly referred to as security protocols within the official libp2p documentation [59], which are of course of the most interest for this thesis [55]. Depending on the implementation of choice, each of those protocol categories comes with one or multiple readily implemented protocols. Currently, there are official implementations of libp2p in Go, JavaScript, Node.js and rust, with versions for Java and Python in development [51]. All code written in the course of this thesis uses the Go implementation, which is one of the most feature-complete of the currently available versions of libp2p [51]. For security protocols, it includes official implementations of the Noise protocol as well as TLS [52]. In addition to these two, the now-deprecated security protocol Secio is also supplied for older version of libp2p. These are, as stated in the corresponding chapter, three of the protocols analysed in this work.

The current Go implementation of libp2p supports four various key types: RSA, with minimum 2048-bit key length, Ed25519, ECDSA and Secp256k1 [57]. An overview of other relevant API points is omitted here, as they are presented in conjunction with the implementation of the Signal protocol created for this thesis in section libp2p Signal Protocol Implementation.

3 Related Work

This chapter gives a quick overview of work that has been done by others in related fields. So far, to the extent it can be judged by the research conducted in preparation for this work, there are no other papers which conduct a comparative performance analysis of the cryptographic protocols discussed here, much less of course of their specific libp2p implementations. While there have been various efforts at comparing the performance of cryptographic protocols, they not only concern themselves with different protocols as this thesis, they are also generally conducted in different, specific contexts, such as a paper by Antonio De Rubertis et al. [2013] [88] which compares the resource performance of IPSec [11] with that of DTLS [30] within the context of the internet of things. Another example is a paper by P.G. Argyroudis et al. [2004] [3] analysing the performance of SSL, S/MIME [41] and IPSec specifically for handheld devices.

Despite this, there are still various published analyses directly related to this work. For one, there are many publications which analyse and compare the performance of cryptographic primitives (see section Cryptographic Protocols), some of which evaluate primitives finding use in the libp2p protocols presented here.

A relatively recent paper authored by Eduardo Anaya et al. [2020] [1] examines both time performance and power consumption of two of the encryption algorithms appearing in this paper, ChaCha20 and AES (although, like one of the previously mentioned papers, specifically for internet of things devices). It finds that while ChaCha20's power draw is around 3.8% higher than that of AES, both its encryption and decryption process is roughly four times faster. Figure 5 illustrates the paper's relevant time performance findings. [1]

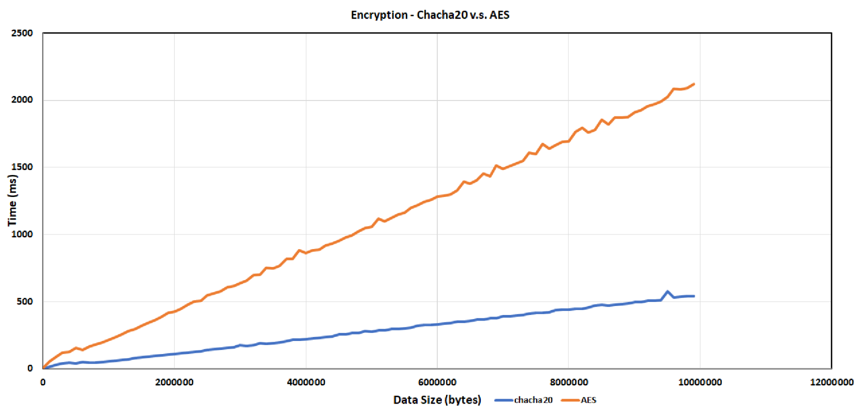

Figure 5: *Encryption time performance of ChaCha20 and (an unspecified version of) AES, as measured in paper "A Performance Study on Cryptographic Algorithms for IoT Devices" [1]*

Another relevant paper is one by Diaa Salama Abdul. Elminaam et al. [2008] [21] which, analyses and compares performance of six different symmetric encryption algorithms. Encryption and decryption time performance are considered. While only including one algorithm also relevant to this thesis in its analysis, it gives some insight into the performance difference between AES128 and AES256; using AES256 sees a 16% increase in both power and time consumption. [21]

While primarily a comparison of performance between different cryptographic keys, the article "RSA and ECDSA Performance" [46] also reveals performance of various versions of Diffie-Hellman algorithms, using the same or different key types. Namely, ECDHE-RSA, ECDHE-ECDSA and DHE-RSA are examined. [46]

Concerning itself with hashing functions, a paper by Junko Nakajima and Mitsuru Matsui [2002] [73] evaluates and compares the amount of micro-operations performed by a wide range of hashing algorithms, including SHA-256 and SHA-512. The paper finds that SHA512 performs around 26,7% more micro-operations per run. An article by Jackson Dunstan compared various hashing algorithms' runtime and also finds that SHA512 is 26.4% slower than SHA256 [20]. Graph 6, taken from the paper "Performance of Message Authentication Codes for Secure Ethernet" by Philipp Hagenlocher [32] in which various functions are tested for "per-packet MAC" [32], shows performance of various MAC hash-functions in relation to packet size.

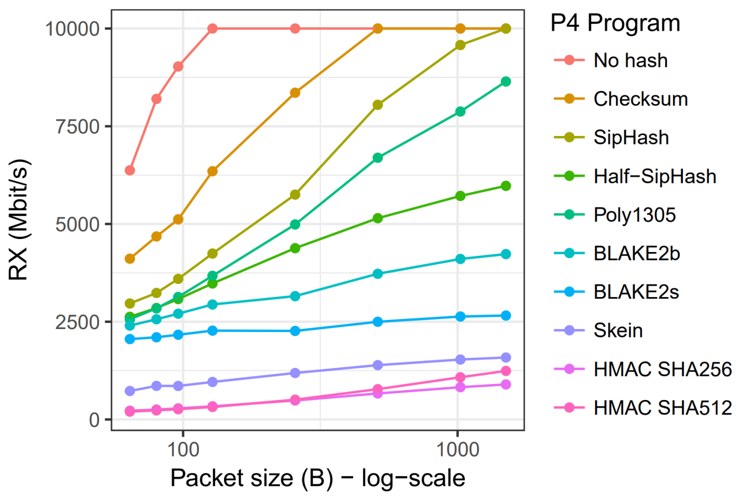

Figure 6: *Throughput comparison by packet size [32]*

There is of course also a large amount of similar papers with less relevance to this thesis, analysing a set of algorithms that include only one of the relevant primitives or none at all [80, 72, 44]

Beyond analyses of cryptographic primitives' performance, there are also papers which perform thorough performance analysis of a full cryptographic protocol (though not of the libp2p implementations). The paper "Performance Analysis of TLS Web Servers" [2002] [13] compares the CPU performance of TLS in various set-ups (different primitives and different hardware, among other aspects) in detail. The Noise protocol also was fully analysed, though not as directly as TLS. Instead, a paper by Son Ho et al. [2022] [35] presents various implementations of the Noise protocol and evaluates and compares their time performance with each other. As far as the research for this thesis has revealed, neither the Signal protocol nor Secio have been similarly analysed as TLS and Noise. In the same manner, it should also be noted here that libp2p has not received a full performance analysis of any of its security protocols either, nor of any other aspect. Most papers on libp2p so far have focused either on development using the libp2p API [103, 31] or on development of new features for libp2p [105].

4 Considered Protocols

Each protocol that is analysed in chapter Analysis is introduced. The cryptographic primitives supported by each are presented, followed by a quick discussion of the specifics of the libp2p implementation. An overview of each protocol's handshake is also given. In the tables present in this chapter, a primitive marked in blue indicates that it is also supported by the libp2p implementation, and a yellow marking that it is utilized in the analysis. Due to scope of this work only the primitives which are selected by default, at least on the system on which the analysis is conducted (see section System Specification), are considered in the analysis.

All the protocols considered in this thesis, with the exception of Secio, have been fully analysed in regards to their security and concluded to be safe [14, 19, 50]. This refers to each protocol in general, not its specific libp2p implementation. Due to topic and limited scope of this work, all implementations are assumed to be safe without further investigation.

TLS, Noise and Secio have been chosen for the analysis as they are the three officially implemented security protocols for libp2p. Prior to this work it was decided to additionally implement one new security protocol for libp2p. Signal was chosen due to its prominence [45, 69, 71].

4.1 TLS

The TLS (Transport Layer Security) protocol was first proposed in 1999 by the Internet Engineering Task Force (IETF) [12]. It is the successor to the SSL (Secure Sockets Layer) protocol, which itself was defined in 1994 and of which the final version (SSL 3.0) was deprecated in favor of TLS in 2015 [12, 40]. Largely due to its integration in the HTTPS (Hypertext Transfer Protocol Secure) protocol, originally proposed as "HTTP Over TLS" [29], it is one of the most widely utilized security protocol on the internet, with 99.7% of the 150000 most popular websites in the world supporting TLS1.2 [85]. 54.2% also support the latest version of TLS, TLS1.3 [85], which is also the specific version of TLS implemented for libp2p [56].

Compared with previous versions, TLS1.3 offers only a limited amount of cipher suites. Whereas TLS1.2 defined 37 different ones, TLS1.3 only has support for 5 cipher suites [7]. These are reflected in table 2, which lists all cryptographic primitives (as defined in Cryptographic Protocols) supported by TLS1.3.

libp2p TLS implementation

The libp2p implementation of TLS is one of as of now two officially implemented (non-deprecated) libp2p security protocols, and is a current candidate for recommendation [98, 97]. It is heavily based on, and consists primarily of wrapper functions for, the official Golang TLS implementation

Table 2: *Cryptographic primitives supported by TLS1.3 [7]*

Key Exchange	Encryption	Authentication
ECDHE Curve25519	AES128 GCM	HMAC SHA256
ECDHE Curve448	AES128 CCM	HMAC SHA384
DHE Curve25519	AES128 CCM-8	
DHE Curve448	CHACHA20-POLY1305	
	AES256 GCM	

[58]. While the Go implementation supports older versions of TLS as well [63], the libp2p TLS handshake may not negotiate any version lower than 1.3 [98].

The Go TLS package cuts various primitives, and these are therefore not supported by the libp2p implementation either. For one, only the cipher algorithms using GCM (Galois Counter Mode) as well as CHACHA20-POLY1305 are included, CCM algorithms are not. Further, both non-elliptic curve key exchanges are unsupported. [63]

Although TLS1.3 does not directly support key type Secp256k1 (see libp2p) [7], the libp2p implementation still allows for all key types supported by libp2p itself [56]. It should also be mentioned that the current libp2p implementation will always prefer CHACHA20-POLY1305 encryption over AES encryption unless both connecting parties have AES hardware support [58] (as is the case for the system on which the analysis in this thesis is performed).

TLS Handshake

The TLS 1.3 handshake, which has been simplified when compared to that of TLS 1.2 [7], consists of three messages. First, the initiating party sends a message containing a list of its supported cipher suites, which in TLS 1.3 only define the utilized encryption and hash algorithms. In addition to this, the message also includes a suggestion to the remote party on which key agreement algorithm to use. Its public key is included in this initial message as well. The remote party, once it received the initiating message, answers with a message defining the key agreement protocol which will actually be used. This message further includes the remote party's public key, a signed certificate authenticating the remote party, and a "Finished" message which is encrypted using the symmetric key the remote party can generate at this point. The initiator receives this packet and uses the remote public key along its own to also generate the symmetric key. It then checks the received certificate and "Finished" message using this key. If everything is in order, it sends its own "Finished" message and the handshake is complete. [38]

4.2 Noise

Noise itself is not a protocol, but rather a framework for creating cryptographic protocols [81]. While not as widespread as TLS, Noise is an established protocol as well, finding use in major applications such as the chat messenger WhatsApp [108]. Compared to TLS, it offers a small pool of cryptographic primitives, all of which are presented in table 3.

Table 3: *Cryptographic primitives supported by Noise [81]*

Key Exchange	Encryption	Authentication
ECDH Curve25519	CHACHA20-POLY1305	HMAC SHA256
ECDH Curve448	AES256 GCM	HMAC SHA512
		HMAC BLAKE2s
		HMAC BLAKE2b

It being a framework, Noise requires many other decisions to be made when implementing a protocol based on it. For example, there are 15 different handshake patterns defined by the official Noise specifications, more if the optional pre-shared symmetric key mode is considered. [81]

libp2p Noise implementation

libp2p's Noise implementation is currently the only officially recommended security protocol for libp2p [99].

The current libp2p protocol based on the Noise framework (which will be referred to as libp2p's implementation of Noise from here on) defines only one valid cipher suite, consisting of the Curve25519 key exchange algorithm, CHACHA20-POLY1305 as encryption / decryption method and SHA256 as the hash function. All libp2p sessions using the Noise implementation as security protocol must use these specific primitives, else the handshake will fail. This decision was made in order to ease implementation and slightly boost performance of the libp2p-Noise handshake, as this way no suite must be negotiated. Along with this, the decision to include only one of the mentioned 15+ handshake patterns was made, once again to limit complexity of implementation. [99]

Noise Handshake

Only the one mentioned and supported handshake pattern, the XX pattern, is explained here. In the case that the XX pattern is utilised, each party possesses an ephemeral key pair and a persistent static key pair. The XX pattern handshake consists of three messages. The initiating party sends a message containing their ephemeral public key, in plaintext. Once it received the initiating message, the remote party responds with its own ephemeral public key, also in cleartext, and its static public key, encrypted using the ephemeral key sent by the initiator. In addition to this, the message also includes the result of a Diffie-Hellman operation between both of the public ephemeral keys, as well as the result of a Diffie-Hellman operation between the public ephemeral key of the initiator and the remote party's public static key. Both of these are also encrypted using the remote party's public ephemeral key. In a Noise handshake, each message is assumed to have a payload in addition to the "standard" content of the message (in this case this would be the ephemeral key), although this may also be of length zero. As mentioned above, the libp2p implementation only supports one specific cipher suite and therefore has no need to further negotiate any primitives. If this was not the case, this first message would contain a payload string defining the used handshake pattern, the curve utilised in the elliptic-curve Diffie-Hellman algorithm, the encryption algorithm and the hash function, in this order. For the cipher suite supported by the libp2p implementation, this string would be "Noise_XX_25519_ChaChaPoly_SHA256". The initiator receives this message and decrypts it using its private ephemeral key. It performs the same two Diffie-Hellman operations and compares the results. If they are equal, everything succeeded so far. Additionally, it performs a Diffie-Hellman operation on its own public static key and the remote ephemeral public key. The result of this, along with the initiator's public static key is sent, encrypted using the remote party's public ephemeral key. The remote party decrypts this message using its private ephemeral key, also performing the same operation and comparing the results. If they are equal, the handshake has concluded successfully. Both parties derive the shared key by hashing the results of all three Diffie-Hellman operations. [81]

4.3 Signal

Originally developed by the now-dissolved software development group Open Whisper Systems as Axolotl [23, 70], the Signal protocol was created to serve as the cryptographic protocol for the messenger app Signal (originally called TextSecure) [23]. It has since been used in various other chat applications as well, including major ones such as Skype [45] and Facebook's proprietary Messenger app [69], as well as WhatsApp [71] which has an active userbase of roughly 2 billion people [17], making Signal one of the most widely utilized security protocols along with TLS.

As with the two protocols already introduced, the cryptographic primitives used by the Signal protocol can be seen in table 4. For signing, Signal also uses modified versions of EdDSA (XEdDSA and VSEdDSA) [93]. These have been omitted.

Table 4: *Cryptographic primitives supported by Signal [93]*

Key Exchange	Encryption	Authentication
X3DH Curve25519	AES256 CBC	HMAC SHA256
X3DH Curve448		HMAC SHA512

libp2p Signal implementation

The libp2p Signal implementation used in the analysis has been written specifically for and in the course of this thesis. It is based on a modified version of an inofficial Signal library for Go, which can be found in this work's git repository [1], created by Radical App LLC, a software development company [66, 65]. For the reasoning behind this decision, see the first paragraph of section libp2p Signal Protocol Implementation The utilized library implements the Curve25519 key exchange, AES256 encryption and SHA256 as hash function [86].

[1]The git repository contains the already modified version. See the readme included in the repository for the original library

The following section quickly introduces the Signal handshake. Slight possible variations and mathematical detail defined in the official specs [94] are omitted.

Signal Handshake

In a Signal handshake, each party possesses a public identity key. Each party can also generate so-called prekeys. A prekey is "essentially [a] protocol message" [93]. In order for the dialer to be able to initiate a handshake, it is required for the listener to have previously pushed various prekeys they generated on a server accessible to the dialer. Because of this, a pure peer-to-peer implementation of the Signal protocol is infeasible without deviating from the official specifications. The keys pushed to the server by the listener include, per bundle, one signed prekey, one one-time prekey (one-time prekeys serve to initiate exactly one run of the X3DH protocol) as well as their public identity key. The dialing party requests all of these keys from the server and, once retrieved, uses them along with a locally generated ephemeral key [74] and their own public identity key as input to four runs of the Diffie-Hellman function. The four resulting values are in turn used as input for the HKDF algorithm, a key derivation function based on HMAC [49], which returns what will be the shared key between the two parties once the handshake is concluded. The various calculations performed at this step can be gathered from figure 7.

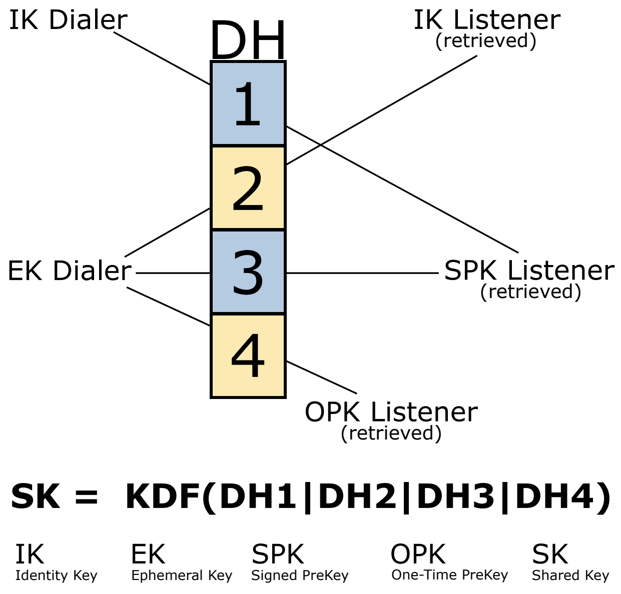

Figure 7: *Visualisation of the DH calculations performed after retrieving the other party's prekey bundle from the server [94]*

The dialing party then sends an initial message containing a ciphertext encrypted with the result of the HKDF function along with their identity key and ephemeral key in plaintext. The packet also includes a plaintext identifying parameter, telling the listener which of their one-time prekeys was used. Once the listener receives this initial message, they possess all of the parameters the dialer used to calculate the key and can therefore chain the same Diffie-Hellman functions to derive SK. If the ciphertext included in the received message can be correctly decrypted using the shared key, the listener deletes its used one-time prekey and the handshake is complete. [94]

4.4 Secio

Secio (secure input/output) was developed in 2014, and is the only one of the 4 chosen security protocols to have been conceptualized and developed specifically for libp2p. As TLS1.2, the TLS standard at the time, required Certificate Authorities, Secio was created to serve as a protocol in the interim. Therefore, when TLS1.3 support as well as Noise support for libp2p had established, Secio was deprecated in 2020. [34]

Despite its deprecated status Secio is still included in the analysis for the sake of completeness as it is one of only three officially implemented security protocols (discounting the transport protocol QUIC, which uses TLS1.3 internally) [34].

The cryptographic primitives utilized by Secio are again presented below in table 5. While the official specifications of Secio do not specify what version of AES is to be used, the actual implementation uses GCM. This is also reflected in table 6.

Table 5: *Cryptographic primitives supported by Secio [59]*

Key Exchange	Encryption	Authentication
ECDH Curve P-256	AES128	HMAC SHA256
ECDH Curve P-384	AES256	HMAC SHA512
ECDH Curve P-521		

Secio Handshake

At the beginning of a Secio handshake, both parties generate a message containing their own public key and a 16-byte, randomly generated nonce. In addition to this, this initial message also includes three lists of supported cryptographic primitives, one for encryption algorithms, one for supported hash functions and one for supported key-exchanges. Both parties send this message to the remote party. Once received, each of the parties calculate whose primitive preferences will be prioritised. This is done by hashing the remote public key in concatenation with the local nonce, and vice versa. Whichever result is larger, the preferences of the party whose nonce was used in the generation of it will be preferred in the next step. If both results are equal, it is assumed that the peer is communicating with itself and the handshake is interrupted. Next, the preferred lists of primitives are iterated over. The first primitive of each list that is also supported by the other party is chosen. This concludes the proposal phase of the handshake. Both peers then generate an ephemeral key pair as well as a new message. This new message consists of a concatenation of their own, local proposal message from earlier, the remote proposal message and their own public ephemeral public key. Each then signs this message with their static private key. This signature is then in turn concatenated with their ephemeral public key. Both parties send this new message to each other, both recreating the received message using the known proposal messages and the just received remote public key. Using the remote persistent public key received in the proposal process, this signature may be validated. If it fails to validate, the handshake is interrupted. If it does validate, both peers then generate a shared secret using their private ephemeral key and the remote ephemeral public key as input to a Diffie-Hellman function. This shared secret is then used to derive, using key stretching, the details of which are omitted here, to generate a shared secret key. [100]

4.5 Overview

Below, table 6 gives an overview of the primitives supported by each protocol. While authentication and encryption primitives are fully reflected, only the key exchange functions which are actually utilized in the protocol's respective libp2p implementations are included to improve readability. The color code is identical to the previous tables in this chapter.

Table 6: *Comparison of cryptographic primitives supported by each protocol*

	TLS	Noise	Signal	Secio
Key Exchange				
ECDHE Curve 25519	×			
ECDH Curve 25519		×		
X3DH Curve 25519			×	
ECDH Curve P-256				×
Encryption				
CHACHA20-POLY1305	×	×		
AES128 GCM	×			×
AES128 CCM	×			
AES128 CCM-8	×			
AES256 GCM	×	×		×
AES256 CBC			×	
Authentication				
HMAC SHA256	×	×	×	×
HMAC SHA384	×			
HMAC SHA512		×	×	×
HMAC BLAKE2b		×		
HMAC BLAKE2s		×		

5 Methodology

This chapter gives an overview of the methodology used in gathering the data points for the analysis following in chapter Analysis. First, the implementation of Signal for libp2p is presented. Then, the various considered aspects for the analysis are introduced along with the methods used to gather the necessary data. Finally, the analysis methods are presented.

5.1 libp2p Signal Protocol Implementation

The implementation of the Signal protocol developed for this work is based on an altered version of an inofficial Go Signal library, developed by software development company RadicalApp [66, 65]. This choice was made as there is no official and no other major implementation of Signal for the Go programming language (as far as the research conducted for this work revealed), and the other considered possibility, to use the official C library [96] and call into it from Go, was deemed as having too little effect on the quality of the analysis to justify the increased difficulty of implementation, as there would be overhead generated from calling into a different language [28]. It should be mentioned here however that, as the analysis will show, due to the performance of the chosen library it would likely have been preferable to simply adjust the results for the overhead had it not been for the limited timeframe of this thesis. This is acknowledged and discussed in section Future Work as well. As the original library has been unmaintained since 2017 [86], various fixes and updates have been adopted from the most actively maintained fork [16] as well as various minor custom modifications done for this work. This modified version of the library can be found in the git repository of this thesis, as well as a reference to the original library and the mentioned fork. Major differences from the original library are pointed out in the respective files.

The following paragraph gives a quick overview of the relevant libp2p API aspects and the corresponding Signal implementations. Since the implementation-specific documentation of libp2p is not entirely thorough, some of the following points are assumptions made and developed by working with and researching the API for several months.

In the Go implementation of libp2p, security protocols must implement two interfaces: the SecureTransport interface, which creates instances of the other interface required of security protocols, SecureSession [61]. In addition to the following explanation of these two interfaces in the context of the Signal implementation, figure 8 also shows a general visualisation of these two interfaces' roles, exemplified by a connection establishment (followed by a write operation and the infinite read-loop).

SecureTransport's implementing struct typically holds information about the local peer, such as its ID and the local private key. The Signal implementation additionally contains storage structs holding the various prekeys. These are required to build secure sessions by the Signal library [86].

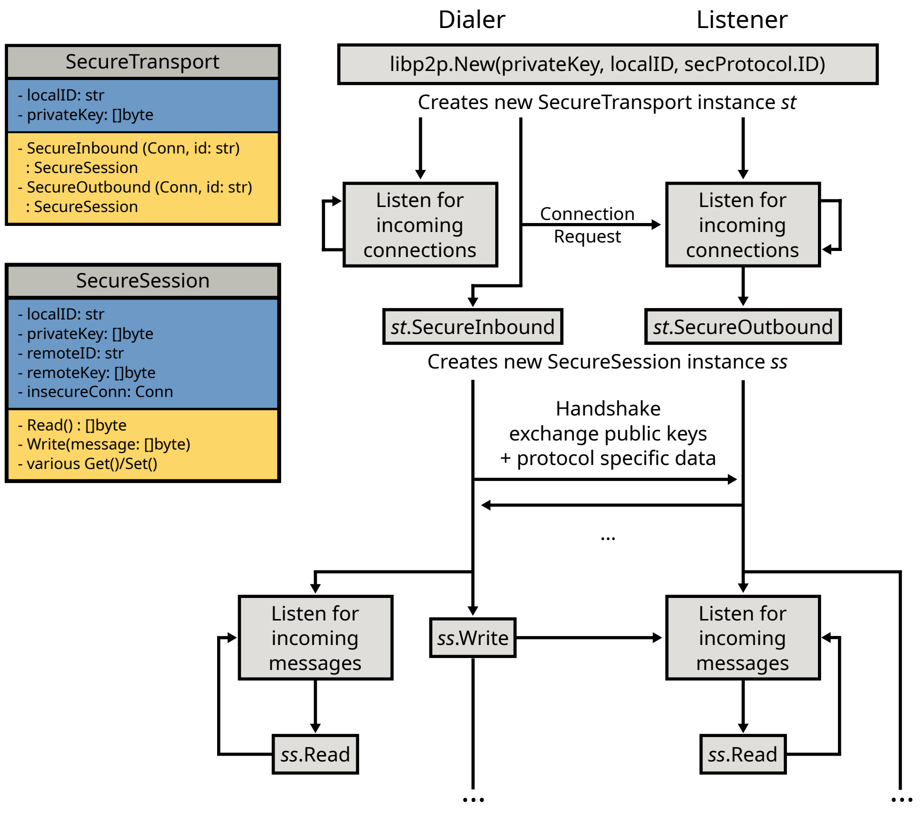
Dialer
Listener
SecureTransport
- localID: str
- privateKey: []byte
- SecureInbound (Conn, id: str) : SecureSession
- SecureOutbound (Conn, id: str) : SecureSession
SecureSession
- localID: str
- privateKey: []byte
- remoteID: str
- remoteKey: []byte
- insecureConn: Conn
- Read() : []byte
- Write(message: []byte)
- various Get()/Set()
libp2p.New(privateKey, localID, secProtocol.ID)
Creates new SecureTransport instance st
Listen for incoming connections
Connection Request
Listen for incoming connections
st.SecureInbound
st.SecureOutbound
Creates new SecureSession instance ss
Handshake exchange public keys + protocol specific data
...
Listen for incoming messages
ss.Write
Listen for incoming messages
ss.Read
ss.Read

Figure 8: *Visualisation of SecureSession's and SecureTransport's role in a libp2p connection establishment*

Its interface functions are called whenever an attempt at an incoming or outgoing connection is detected and receive an insecure connection struct and an ID identifying the other peer as input. Primarily, these serve as wrappers for the protocol-specific handshake, creating an instance of SecureSession and calling the handshake on it. In the Signal implementation, this is also where the peers generate any necessary prekeys. In the case of the peer who would normally push their prekeys on a server, the prekeys are read from a local file. If the handshake succeeds, the new SecureSession is returned. Otherwise, an error is thrown.

While a private function and therefore not part of the interface, the handshake is typically implemented as a method on the SecureSession struct. As every peer in libp2p is identified both by its ID and its key pair [53], the libp2p section of the handshake consists of exchanging and verifying ID and public key of the remote peer. This requires at least one message by each peer. Therefore, while the majority of the handshake is implemented as described in section Signal, an additional message is sent by the receiver of the initial message. A quick walkthrough of the implemented handshake is given in the next paragraph.

Upon entering the handshake function, each peer, whether dialer or listener, generates a payload containing their own public key appended to a concatenation of the prefix string "signal-libp2p-static-key:" and their public key, signed with their corresponding private key. The dialer then reads the listeners prekey bundle from drive, simulating retrieving them from a server. The "retrieved" bundle is used as input to the ProcessBundle function, equivalent to the chaining of functions presented in figure 7 of section Signal. Using the derived key, the previously generated payload is encrypted, generating a PreKeySignalMessage, containing both the ciphertext payload and the utilized prekeys. The PreKeySignalMessage is sent to the listener, the dialer now waiting for a response. The listener receives the packet and processes the packet the same way the dialer did, afterwards decrypting the payload, verifying the signature with the included plaintext key. The listener's payload is encrypted and sent to the dialer, who verifies the signature in the same manner. If none of these steps fail, the handshake has concluded successfully.

An alternative implementation of the handshake was considered in which the bundle stored on drive would also include the listener's generated payload. This was tested but ultimately decided against, as performance impact is minimal and the current implementation is closer to the official specifications of the Signal handshake.

SecureSession's implementing struct contains all information pertaining to the secured connection. It holds a reference to the insecure connection struct, identifiers of both local and remote peer, their IDs and private / public key respectively, as well as the shared key. Its interface functions consist mainly of Getters for these variables, in addition to Read/Write functions for receiving data from and sending data to the remote peer.

Verification

In order to be able to test and verify the Signal implementation is working as intended, developing a simple libp2p application is necessary. The application has to be able to connect two libp2p peers with each other, which includes performing the handshake implemented by the respective security protocols. Once connected, both peers need to be able to send data packets of variable size to each other. These packets have to be fully encrypted and authenticated. This way, all cryptographic primitives defined in 2.1 are utilized and can therefore be evaluated using this application.

The official Git repository of the Go implementation of Noise [53] contains a test application designed to work with Go's native testing tool [64]. This testing application fits the needs described above perfectly; upon running the application, libp2p peers are created and connected to each other. Various tests are performed on the two connected peers, including tests for successful encryption, decryption and handshake. Beyond testing the primitives, it is also verified whether necessary, libp2p-specific information is exchanged (including each other's public key and peer ID). Along with these major ones go various minor tests. Should any of the tests fail, the whole program fails. Using a modified version of this application, the Signal implementation was fully verified. All modifications are limited to changes necessary to work with the implementation of Signal, such as variable name changes.

5.2 Benchmarking the Security Protocols

Aspects considered in the process of data gathering are presented. Afterwards, methods utilized to gather said data are explained.

Time- and Space-Performance

Time- and Space- performance represent the primary analysis target in this work. Time-performance here straightforwardly describes the duration of the protocol's execution whereas space-performance consists of the amount of memory space levied by the protocol during execution. Even in cases where both of these values are low, which as the analysis will show do exist, minor differences in performance are relevant in practice as well considering the amount and scale of operations that may be required in a peer-to-peer network. For example, file sharing networks may require large amounts of data to be sent or received continuously.

The protocols are benchmarked in regard to both these targets using a test program, uniform for all tested protocols except for minor API-dependent changes, all of which are assumed to have near-zero performance impact. The test program, utilizing the benchmarking functionality built-in to Go's official testing package [64], consists of two major parts; the BenchmarkHandshake function and the BenchmarkTransfer functions. BenchmarkHandshake creates two peers and performs a full handshake between them. Each of the BenchmarkTransfer functions also generates two peers and connects them by executing a handshake (though the resources levied by this handshake are not considered part of the benchmark), and, once the handshake has concluded successfully, has each peer sending/receiving a previously specified amount of data to/from the other. Upon execution of the test file, BenchmarkHandshake is performed as well as each of the BenchmarkTransfer functions, once for 1, 10, 100, and 1000 kilobytes of data. All of these benchmarks output the amount of time the specified function took, as well as the amount and size of memory allocations performed, along with various other information. If multiple runs of each benchmark are executed using the *benchtime* flag, the average values are output. If the custom flag [62] *indTimes* is set, the values for each iteration are output alongside the averages.

Beside *indTimes*, also defined for this test are the custom flags *keytype* and *keylen*, which specify which key type and length are used to generate the peer's libp2p identity key pair, respectively. Also defined is the *packetsize* flag, which, if set, executes an additional BenchmarkTransfer run using a data packet of the size specified by *packetsize*. The test program can be found in this thesis' git repository.

CPU Performance

Another analysed performance measure are the various protocol's load on the CPU. Considering the prevalence mobile devices have achieved [27], there is also an interest in minimizing power consumption of any given software. Therefore, the CPU's power draw is considered alongside its workload.

Both of these values are gathered using the Windows-specific tool HWiNFO [36] and a small test program. The same program is used to test all four of the protocols, disregarding very minor changes due to the protocols' implementations. As with the program written for Time- and Space-Performance, all differences can be safely presumed to have zero effective performance impact. This test program consists of two test functions, handshake and rw. handshake creates two peers and performs a full handshake between them, terminating execution if it fails. The other function, rw, receives two already connected peers as input. One peer then sends the other 100 packets ranging in size from 10 kilobytes to 1 megabyte (the size of the packet incrementing by 10 kilobytes each message).

For custom flags, this test program defines a *handshake* and a *rw* flag which specify whether the respective tests should be run, allowing for separate testing of effect of handshakes and encryption on the CPU. Further, an *iterations* flag is defined to specify the amount of iterations of each test that will be run. This was done instead of simply using the native *count* flag [64] to minimize overhead from starting and cleaning up the test with each iteration. The *keytype* and *keylen* flags are also defined as in Time- and Space-Performance. The test program can be found in this thesis' git repository.

5.3 System Specification

All the tests discussed in this work were executed on a desktop computer running the Linux distribution Debian, the one exception being the CPU tests described in section CPU Performance, which were run on the same computer running Windows 10. This was done as there (to the knowledge of the author) are no comparable tools as HWiNFO [36] for Linux which monitor CPU power consumption as accurately. The used system's CPU is a stock Ryzen 7 3700X. It has 16 gigabytes of ram running at 3200 MHz.

6 Analysis

This chapter presents the results of the various analyses conducted. First, time-performance measurements are presented for each protocol alongside the corresponding space-performance. Also given are crucial points such as the most impactful functions for both categories. Afterwards, both CPU load and CPU power consumption generated by each protocol are given. The last section supplements the preceding data with various additional information. A synopsis of this chapter's findings, as well as a quick evaluation, can be gathered from chapter Summary.

As this work serves as analysis specifically of the libp2p implementations of each protocol, the read and write functions' performances are analysed instead of the encrypt and decrypt functions to include all of the libp2p overhead generated by the respective security protocol each time data is sent / received. Primarily, the read / write functions consist of packet-size handling measures, the actual receiving / sending of the data and, of course, the decryption / encryption process. "Handshake" similarly includes various mandatory libp2p overhead, such as creation of an instance of *SecureSession* (see libp2p Signal Protocol Implementation). Supplementing these data points, the "Insecure" protocol is included in the diagrams when it does not affect readability. It emulates a security protocol, performs the mentioned primary functions but does not encrypt / decrypt the data. For the handshake, it only performs aspects mandatory for libp2, namely exchanging peer IDs and public keys.

All analysis has been conducted using SPSS created by IBM [37] and the raw data gathered using the methods described in chapter Methodology. SPSS is a statistical analysis tool which takes in data in the form of a spreadsheet. It performs various analytical functions on the input and allows for creation of diagrams using the data created in the analysis. All diagrams in this chapter, including the tables, have been created using SPSS.

6.1 Time- and Space-Performance

Each protocol's overall performance is presented, divided into handshake and encryption/decryption performance. While libp2p supports four types of key, only results for the RSA key type (using a key length of 2048 bits, the smallest RSA key length supported by libp2p) and Ed25519 key type are reflected. The decision to forego presenting values for some key types was made to improve readability of the following diagrams. RSA and Ed25519 are chosen because performance varied most prominently between these two key types, with RSA being the most demanding out of the four available ones and Ed25519 being the least demanding. The other two key types' results were usually much closer to the performance measured when using the Ed25519 key type. All values presented in this section have been gathered using the test program described in Benchmarking the Security Protocols with flag *benchtime* set to 100. Therefore, each median or mean value given here is derived of a set of 100 data points. The paper "Performance Analysis of Encryption Algorithms for Security" by Madhumita Panda [79] uses averages of ten iterations for each of its analyses [79]. A higher amount of iterations was chosen here due to relatively high variance in time performance, in some cases.

Figure 9 shows the time-performance of each protocol's handshake for both considered key types. As can be seen, Signal's handshake is the slowest in both cases, the median performance (denoted by the line in each of the boxes) taking roughly one millisecond longer to conclude than the second slowest. In the case of RSA, Noise is the second slowest with Secio being slightly faster than it and finally TLS being the fastest, also having by far the least variance with barely any outliers compared to the other protocols. When using Ed25519 keys, as expected due to its smaller key length and signature, each of the protocols' time-performance strongly improves, the measurements also being more consistent. The execution time of Noise, Secio and Signal also stay relative to each other, with Signal still being about a millisecond slower than Noise and Noise in turn being roughly half a millisecond slower than Secio. TLS' performance however stays nearly unchanged, the only real difference being more outliers toward the bottom of the graph. The most time performant combination of protocol and key is Secio using a key of type Ed25519. Considering Secio's deprecated status, the best performing combination still valid in the current version of libp2p is Noise, also using an Ed25519 key.

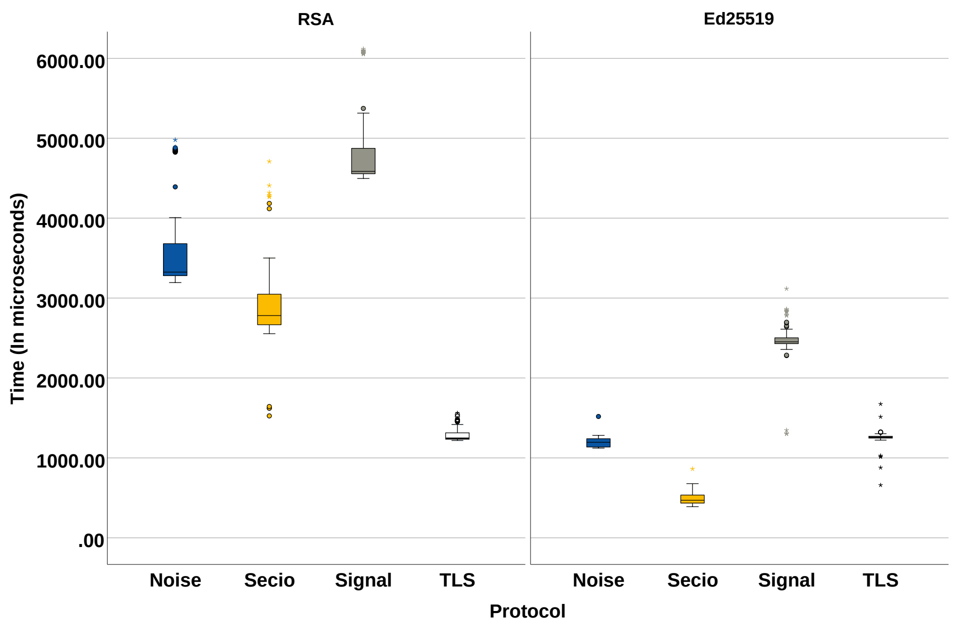

Figure 9: *Time-performance of each protocol's handshake, per key type*

Each protocols' read and write performance are presented in figure 10. To improve readability, only median time-performance of each protocol's read and write operations are displayed. For the same reason, the scale used is to the power of 0.5, instead of being linear. Performance when using Ed25519 keys is also omitted as the difference in relative performance is minor. It is instead reflected in table 7.

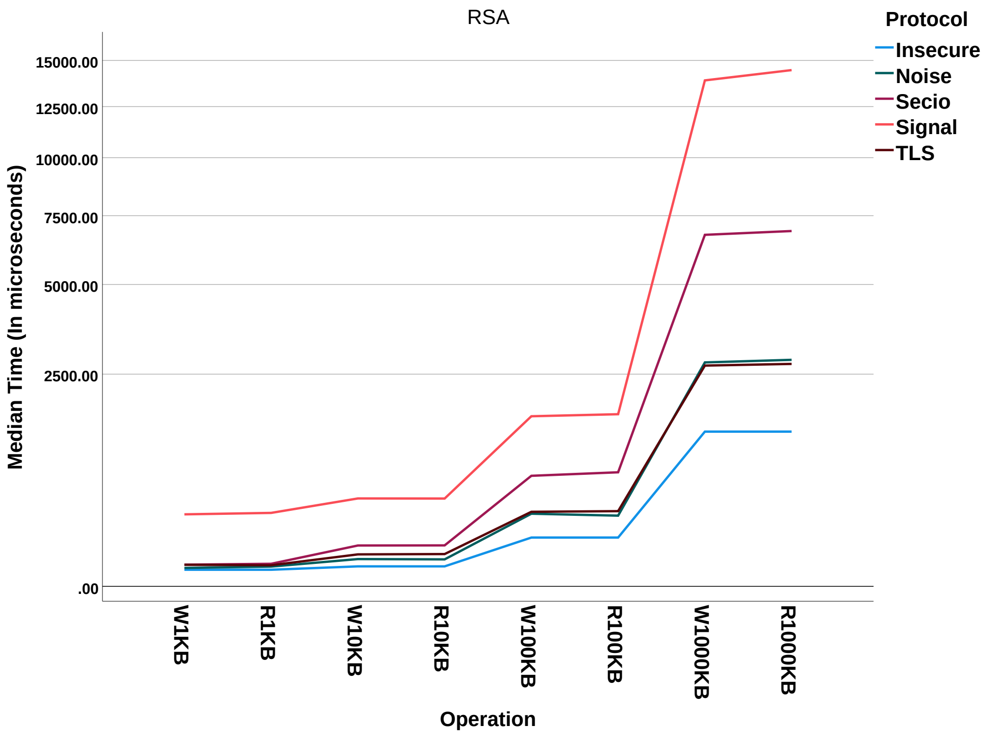

Figure 10: *Median time-performance of each protocol read and write operations using RSA*

As is evident from figure 10, not only are the TLS and Noise implementations by far the fastest of the actual protocols, their time-performance is also nearly identical. Growing at a similar rate, their execution time doubles from the 1KB operations to 10KB, increases about six-fold for the 100KB operations and finally increases eightfold for the last jump. Noise performs very slightly faster than TLS however, with TLS outperforming Noise, just as slightly, for the 1000KB packets. Secio's execution time increases at a similar rate, although moderately increased at each step. Signal displays the worst performance by a great amount not due to an increased growth rate, which is at some steps as good as Noise's and at others better. Instead, Signal's time-performance "starts" almost a full magnitude slower than Noise. The most time performant combination of key and protocol for read / write operations is either Noise or TLS using an Ed25519 key, although, as mentioned, the performance using a RSA key is near-identical.

Signal is not included in the space-performance analysis with the other protocols. For a quick discussion of this, as well as stand-alone space-performance analysis of Signal, see section Signal's Performance. As can be seen in figure 11, although TLS overall has the best time performance out of the protocols, it also has the worst space performance out of all of them by far (not considering Signal). Using the RSA key type, allocates more than twice the amount of bytes that either Noise or Secio do. All individual functions, both read / write and handshake, also allocate more than the corresponding function of the Noise and Secio implementation. Switching to an Ed25519 key only worsens TLS' performance in contrast to the other protocols; while both Noise and Secio only levy about a third as much memory space as they do when using an RSA key, TLS still allocates 86.2% of the data it allocated before. Noise and Secio perform very similar in the case of Ed25519, within 13%, and almost identical in the case of RSA, within 2%. Still, Secio slightly edges out Noise in space-performance. As clearly evident from figure 11, Secio performs fastest when using an Ed25519 key. Compared to time-performance, variation of memory allocated is minimal per function and therefore is not reflected here. The amount of allocations performed, while also not reflected here, is presented in section 6.3.

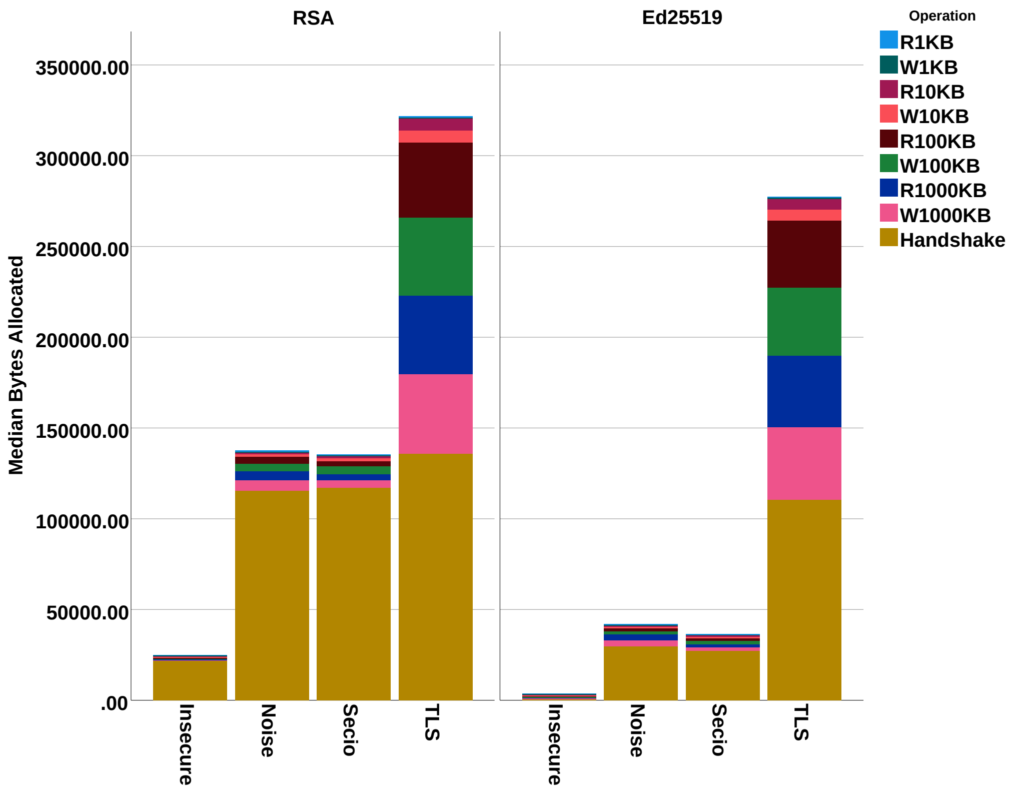

Figure 11: *Memory allocated by each function, per protocol and keytype*

Signal's Performance

As some of the time-performance measurements have already shown, the Signal implementation performs poorly compared to the officially implemented libp2p protocols. This is all the more the case for its space performance; the amount of data allocated, which is displayed in figure 12, exceeds even TLS allocation so much that it had to be omitted from figure 11 in order to preserve readability.

The Signal implementation created for this thesis is based on an inofficial library, as explained in section libp2p Signal Protocol Implementation, the reasons for which are also given in said section. As has been ascertained using the native Go tool "pprof" [84], the causes of these massive amounts of allocated memory stem primarily from the serializer functions. Before sending data, the message, originally in the "SignalMessage" format, has to serialized into a byte array. Once received the array is deserialized back into its original format. Over the 100 executions of each function (handshake, encrypt/decrypt 1/10/100/1000KB), these serializer functions allocate over

1.4 gigabytes of memory, about 40% of all memory allocated during execution. As the amount of data that needs to be allocated is dependent on the size of the message that is to be (de)serialized, these functions impact is also strongly reflected in figure 12. The memory allocations scale roughly with the size of the data. Various other functions of the library also levy inordinate amounts of memory, such as the encrypt and decrypt functions, which are used by the libp2p's implementation of Signal within the read/write methods, that allocate a quarter of a gigabyte each. For comparison, TLS, the second-most memory demanding implementation, allocates around 300 megabytes overall. Due to the limited timeframe of this thesis, implementing a fix for this was not possible. This is also mentioned in chapter Future Work.

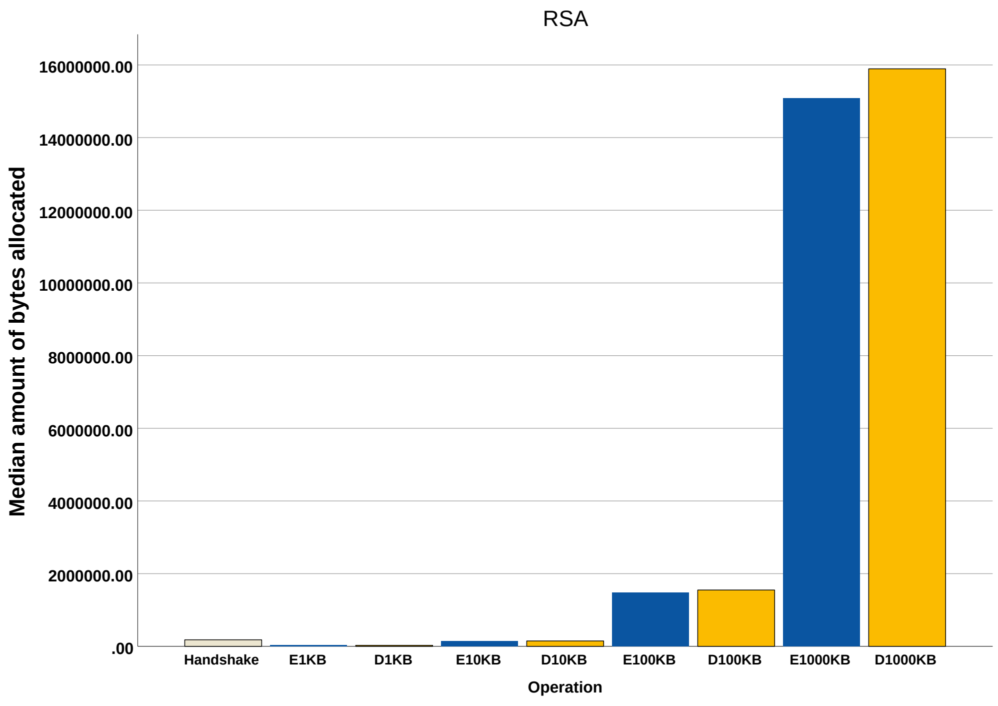

Figure 12: *Memory allocated by each function of the Signal implementation, using RSA*

6.2 CPU-Performance

Each protocol's impact on the CPU is discussed, both in terms of usage and power consumption. All of the values presented here are purely comparative and are not meant to suggest that these are the usage percentages / power consumption generated by the protocols alone. Again, for the exact methodology used, refer to chapter Methodology.

The test program described in chapter Methodology is executed 1000 times for each protocol using the go test command [64] and the relevant CPU values logged once a second during execution by HWiNFO. An even higher amount iterations as compared to the time and space performance analysis was chosen here to better control for natural variance in power consumption and load. According to its output, power deviation accuracy hovers around 104% during all tests, implying a slight but also uniform overestimation of power drawn from the CPU. While more accurate measurement tools were not available for this thesis, these tests have been executed multiple times and the results were very similar each time.

Below, table 13 shows the average CPU usage generated by each protocol and operation. As can be seen, the loads generated during the handshake operation are nearly identical for all protocols, the differences that are there being too small to draw any conclusions as they may well be just standard fluctuations in usage. The values generated during the rw operation however differentiate much more between protocols. While Noise and Secio still perform similarly, Secio outperforming Noise by around four percent (relative), TLS and especially Signal generate much higher CPU usage. TLS creates about a percent higher CPU load than Noise, whereas Signal generates even more at 9.16% total CPU usage, or about 2.3% more than Noise. When compared to the space-performance measurements of the previous sections, it is noteworthy that the protocols relative performance is analogous; Noise is slightly outperformed by Secio, TLS is more clearly outperformed by Noise and finally Signal is the outlier with the worst performance between all four protocols. This suggests that the bulk of the CPU load differences may stem from the varying amounts of memory allocated.

		Protocol			
		Noise	Secio	Signal	TLS
		Load	Load	Load	Load
		Mean	Mean	Mean	Mean
Operation	Handshake	6.55	6.65	6.62	6.57
	RW	6.86	6.58	9.16	7.84

Figure 13: *Average CPU load during the rw function in percent, per protocol*

Figure 14 presents the amount of watt-seconds (equivalent to joule) the CPU generated during execution of each protocol. These values are derived by multiplying execution time of the test program with the arithmetic mean of the corresponding power consumption logs. All of these source values can also be gathered from the same figure. As can be seen, the performance for the handshake operation is, similar to the CPU usage results, near-identical for all of the four protocols. For the rw operation, TLS' and Noise's performance is about equal, TLS being a little faster but also drawing more power. Secio performs worse than during the handshake, having a lower average power draw but also a considerably longer execution time, overall generating about 20% more watt-seconds than Noise and TLS do. Signal's performance is again the worst out of the four, producing almost twice the watt-seconds Secio does.

6.3 Other aspects

Table 7 reflects some minor aspects which serve to supplement the previous analysis. As can be seen in it, the file size, when compiled, is relatively similar for each of the protocols; Signal takes up the most space with 12.1 megabytes and Noise the least at 9.56 megabytes. TLS and Secio fall roughly in the middle. These values have been determined by compiling the "chat" test program from the official libp2p repository [54] four times, using each of the protocols. Especially on mobile

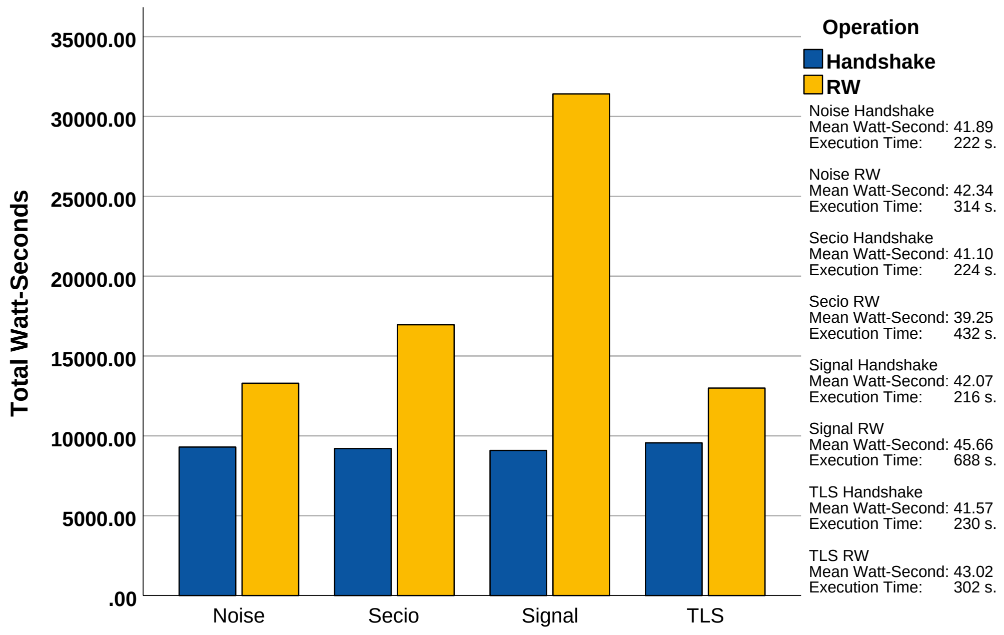

Figure 14: *Total wattseconds, per operation and protocol*

devices, where storage space is often more limited, these minor differences can still be considered somewhat relevant.

Also displayed are the amount of memory allocations performed by each operation excluding, due to space constraints, the write operations. As can be gathered from the table, the amount of allocations is identical when using either key type, with the exception of the handshake. This shows that the growing amount of memory allocated, which grows with the amount of data that is being read, is not due to more allocations, but instead due to larger individual allocations.

Additionally, the table includes the mean time performance of the read operations when using an Ed25519 key, which had been omitted in section Time- and Space-Performance.

Within this thesis' git repository, SPSS tables of the raw data presented in this chapter are contained. In addition to this, it also contains memory and CPU profiles generated using the Go tool pprof [84] and the methodology described in the preceding chapter.

Table 7: *Various other aspects; Compiled file size, amount of allocations and time performance using Ed25519 keys*

Compiled Size	Noise 9.56MB	Secio 11.2MB	Signal 12.1MB	TLS 10.5MB
Amount of allocations (RSA)				
Handshake	758	821	1612	1711
R1KB	10	8	219	12
R10KB	10	8	260	17
R100KB	21	8	1366	32
R10000KB	137	9	12851	123
Amount of allocations (Ed25519)				
Handshake	290	358	1144	1509
R1KB	10	8	219	12
R10KB	10	8	260	17
R100KB	21	8	1366	32
R10000KB	137	8	12851	123
Mean Time (Ed25519)				
R1KB	28.69	36.02	278.98	33.49
R10KB	50.12	107.73	449.12	68.88
R100KB	282.96	766.66	1742.18	330.79
R10000KB	2870.17	6893.23	15187.80	2773.09

7 Summary

This bachelor thesis conducts a comparative analysis of four libp2p security protocols in regards to their performance. More specifically, the time and space performance of each protocol's handshakes and read/write operations are considered along with its CPU performance. The set of analysed protocols includes all three of the security protocols implemented in the officially libp2p repositories, namely TLS, Noise and Secio. In addition to these, the Signal protocol is also considered. Implemented for libp2p specifically for this thesis, it is based on an inofficial Signal library developed by RadicalApp, modified in parts. Moreover, the different key types supported by libp2p for the peers' key pair are also taken into account.

In order to gather the data required for the analysis, two test programs are developed. The first of these programs creates two peers and connects them using the respective protocol's handshake. It then sends messages of various sizes between them, utilizing the protocol's read and write functions. Using Go's native benchmarking tool, both execution time and memory space allocated by each of these operations is recorded. The second program operates in a similar manner. Instead of making use of said native benchmarking tool though, it simply executes these functions. The tool HWiNFO is used concurrently to record both CPU power draw and workload generated during the handshake and read and write functions. Both of these test programs are executed multiple times, the data generated during these tests serving as input to the statistical analysis tool SPSS by IBM. SPSS is used to conduct all of the analysis presented in this thesis, of which a quick summary is given in the next paragraphs.

The analysis shows that, on average, TLS has the best time performance of the four considered protocols. While execution time is nearly identical to Noise for the read/write operations, it considerably outperforms its handshake when using RSA keys, although in turn being slightly outperformed by Noise when using Ed25519 keys. For the handshake, Secio performs slightly better than Noise, with Signal performing the worst, although not by a huge margin. With the read/write operations, the difference between protocols is much more severe, with Secio, in the most extreme cases being around three times as slow as Noise and TLS, and Signal even being yet twice slower.

The amount of memory allocated by Noise and Secio is, in accumulation of all considered functions, nearly identical, with little deviation between memory levied by each function. TLS however has worse space performance, allocating a little over twice as much memory when using RSA keys, and around six times as much when using Ed25519 keys. Signal's space performance in comparison, at least for the read/write operations, is much, much worse. In the most extreme case, reading one megabyte of data, it allocates around 320 times as much memory as TLS does for the same operation.

This trend continues into CPU performance. For the workload generated, performance during handshake is very similar across the board. During the read/write operations, Noise and Secio perform similarly, although Secio slightly outperforms Noise. TLS generates about 19% more CPU load than Secio, with Signal once again performing the worst, generating around 39% more CPU load than Secio. Signal also consumes the most power of the protocols during the read and write operations, generating around 30000 joules during the 1000 iterations the test program is running for. In comparison, Secio performs the second-worst, generating about 17000 joules. Noise and TLS perform similarly here, consuming around 14000 joules. Once again, performance for the protocols' handshakes is the same for all protocols.

7.1 Future Work

Due to the limited scope and timeframe of this thesis, there are various aspects not implemented in this paper which would have improved the quality of the conducted analysis.

As is obvious from the analysis, Signal vastly underperforms compared to the other protocols, at least for the read and write operations. Its underperformance in all categories stems primarily from the huge amounts of data allocated, as discussed in section Signal's Performance. The major performance problems are therefore a product of the utilized library. A fix for these largely unnecessary allocations was not possible to implement within the timeframe of the thesis anymore. Ideally, as there is no official Go implementation of the Signal library, the Signal C library [96] would have been used. As it is possible to call from Go into C code using Go's, this would have been feasible to implement. Although there would be additional overhead generated from calling into another language, which would again taint analysis results, additional analysis of the overhead would have allowed for adjusting the results. Alternatively, it would also have been possible to use the rust implementation of libp2p in conjunction with the official rust Signal library [95]. This was decided against as there are fewer officially implemented security protocols for this implementation of libp2p [60].

More generally, the presented analysis is primarily descriptive. Only in the case of Signal's space performance, due to its status as extreme outlier, are the results reasoned in further detail. While primarily due to the actual topic of the thesis, this is also in part the case because of the limited timeframe. It could be of interest to give a more detailed and thorough analysis of the reasons, especially in making more clear a distinction between libp2p overhead and the actual cryptographic operations. Following that, suggestions and implementations of improvements for the various protocols may be added.

As is mentioned in chapter Considered Protocols, only the standard primitives / cipher suites of each protocol are tested, due to the time limitation of this work. For a more thorough analysis of each protocol, they could be slightly modified to use different supported primitives / cipher suites. This would be very relevant to real-world applications of these implementations as, when the remote peer only supports specific primitives, the local peer will use these as well instead of the standard ones, assuming the local peer supports them.

As the analysis includes only RSA and Ed25519 of the four key types supported by libp2p, and in cases where performance difference was small even just one of them, including all four types for each of the analysis would also add to a more thorough analysis.

Further, for the CPU performance analysis, more accurate measurement methods could be employed.

References

[1] Anaya, Edardo et al.: *Performance Study on Cryptographic Algorithms for IoT Devices*. Tech. rep. California State University, 2020.

[2] Andrea, Harris: *Different Types of Servers in Computer Networks*. 2022. url: https://www.networkstraining.com/different-types-of-servers/ (visited on 04/05/2022).

[3] Argyroudis, Patroklos G. et al.: *Performance Analysis of Cryptographic Protocols on Handheld Devices*. Tech. rep. 2004.

[4] AWS: *Quotas and constraints for Amazon RDS*. 2022. url: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Limits.html (visited on 06/15/2022).

[5] BitTorrent: *About BitTorrent | Creator of the World's Leading P2P Protocol*. 2021. url: https://www.bittorrent.com/company/about-us/ (visited on 12/11/2021).

[6] Briar: *Secure messaging, anywhere - Briar*. url: https://briarproject.org/ (visited on 12/11/2021).

[7] Brodie, Andy: *Overview of TLS v1.3*. 2018. url: https://owasp.org/www-chapter-london/assets/slides/OWASPLondon20180125_TLSv1.3_Andy_Brodie.pdf (visited on 06/14/2022).

[8] Carle, Prof. Dr.-Ing. Georg: *Network Security - Chapter 7 Cryptographic Protocols*. 2003. url: https://web.archive.org/web/20170829004310/http://www.ccs-labs.org/~dressler/teaching/netzsicherheit-ws0304/07_CryptoProtocols_2on1.pdf (visited on 12/06/2021).

[9] Case, Amber: *Why The Internet Needs IPFS Before It's Too Late*. 2015. url: https://techcrunch.com/2015/10/04/why-the-internet-needs-ipfs-before-its-too-late/ (visited on 06/14/2022).

[10] Cloudflare: *What is a network protocol?* 2020. url: https://www.cloudflare.com/learning/network-layer/what-is-a-protocol/ (visited on 12/06/2021).

[11] Cloudflare: *What is IPsec?* url: https://www.cloudflare.com/learning/network-layer/what-is-ipsec/ (visited on 06/18/2022).

[12] Cloudflare: *What is Transport Layer Security (TLS)?* url: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/ (visited on 06/14/2022).

[13] Coarfa, Cristian ; Druschel, Peter ; Wallach, Dan S.: *Performance Analysis of TLS Web Servers*. Tech. rep. 2006.

[14] Cohn-Gordon, Katriel et al.: *A Formal Security Analysis of the Signal Messaging Protocol*. Tech. rep. 2020.

[15] Consulting, Encryption: *What is the difference between Symmetric and Asymmetric Encryption? Which is better for data security?* 2018. url: https://www.encryptionconsulting.com/education-center/symmetric-vs-asymmetric-encryption/ (visited on 03/05/2022).

[16] crossle: *libsignal-protocol-go Fork*. 2021. url: https://github.com/crossle/libsignal-protocol-go (visited on 06/16/2022).

[17] Dixon, S.: *WhatsApp - Statistics & Facts*. 2022. url: https://www.statista.com/topics/2018/whatsapp/ (visited on 06/15/2022).

[18] Documentation, LibP2P Official: *What is libP2P? :: libp2p Documentation*. url: https://docs.libp2p.io/introduction/what-is-libp2p/ (visited on 06/14/2022).

[19] Dowling, Benjamin ; Rösler, Paul ; Schwenk, Jörg: *Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework*. Tech. rep. 2020.

[20] Dunstan, Jackson: *Hash Algorithm Performance*. 2015. url: https://www.jacksondunstan.com/articles/3206 (visited on 04/05/2022).

[21] Elminaam, Diaa Salama Abdul et al.: *Performance Evaluation of Symmetric Encryption Algorithms*. Tech. rep. Higher Technological Institute, 10th of Ramadan City, 2008.

[22] Encyclopedia, Network: *Peer-to-Peer Network (P2P)*. 2021. url: https://networkencyclopedia.com/peer-to-peer-network-p2p/ (visited on 12/11/2021).

[23] Ermoshina, Ksenia ; Musiani, Francesca ; Halpin, Harry: "End-to-End Encrypted Messaging Protocols: An Overview". In: *Internet Science*. Springer International Publishing, 2016.

[24] Ethereum: *Ethereum 2.0 Consensus Specs - Phase 0 – Networking*. 2022. url: https://github.com/ethereum/consensus-specs/blob/0e6a7cd39a44ba64897c359e1d62c8b475e5d926/specs/phase0/p2p-interface.md (visited on 06/14/2022).

[25] Filecoin: *Filecoin Spec - Libp2p*. 2022. url: https://spec.filecoin.io/libraries/libp2p/ (visited on 06/14/2022).

[26] G., Fox: "Peer-to-peer networks". In: *Computing in Science Engineering* (2001).

[27] GlobalStats: *Desktop vs Mobile vs Tablet Market Share Worldwide*. 2022. url: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet (visited on 03/05/2022).

[28] Grieger, Tobias: *The Cost and Complexity of Cgo*. 2015. url: https://www.cockroachlabs.com/blog/the-cost-and-complexity-of-cgo/ (visited on 06/15/2022).

[29] Group, Network Working: *HTTP Over TLS*. 2000. url: https://datatracker.ietf.org/doc/html/rfc2818 (visited on 06/14/2022).

[30] Group, Network Working: *What is IPsec?* url: https://datatracker.ietf.org/doc/html/rfc4347 (visited on 06/18/2022).

[31] Guidi, Barbara ; Michienzi, Andrea ; Ricci, Laura: *A libP2P Implementation of the Bitcoin Block Exchange Protocol*. 2021. (Visited on 06/17/2022).

[32] Hagenlocher, Philipp: *Performance of Message Authentication Codes for Secure Ethernet*. Tech. rep. Technical Universtiy of Munich, 2018.

[33] Heise: *FBI über Messenger: An welche Daten von WhatsApp und Co. US-Strafverfolger kommen*. 2021. url: https://www.heise.de/news/FBI-ueber-Messenger-An-welche-Daten-von-WhatsApp-Co-US-Strafverfolger-kommen-6282456.html (visited on 12/05/2021).

[34] Heun, Jacob: *We're removing support for the SECIO security transport*. 2020. url: https://blog.ipfs.io/2020-08-07-deprecating-secio/ (visited on 06/14/2022).

[35] Ho, Son et al.: *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*. Tech. rep. IEEE, 2022.

[36] HWiNFO: *HWiNFO - Free System Information, Monitoring and Diagnostics*. 2022. url: `https://www.hwinfo.com/` (visited on 03/05/2022).

[37] IBM: *SPSS Statistics | IBM*. 2022. url: `https://www.ibm.com/products/spss-statistics` (visited on 06/20/2022).

[38] IBM: *The TLS 1.3 Handshake*. 2022. url: `https://www.ibm.com/docs/en/sdk-java-technology/8?topic=works-tls-13-handshake` (visited on 06/15/2022).

[39] ico.: *What types of encryption are there?* url: `https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/encryption/what-types-of-encryption-are-there/` (visited on 12/11/2021).

[40] IETF: *Deprecating Secure Sockets Layer Version 3.0*. 2015. url: `https://datatracker.ietf.org/doc/html/rfc7568` (visited on 06/14/2022).

[41] IETF: *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. url: `https://datatracker.ietf.org/doc/html/rfc5751` (visited on 06/18/2022).

[42] Ivh: *Crypto101*. 2013. url: `crypto101.io` (visited on 03/05/2022).

[43] Jaganathan et al.: "A quick synopsis of Blockchain Technology". In: *International Journal of Blockchains and Cryptocurrencies* 1 (2019).

[44] Jeeva, AL. ; Palanisamy, Dr. V. ; Kanagaram, K.: *COMPARATIVE ANALYSIS OF PERFORMANCE EFFICIENCY AND SECURITY MEASURES OF SOME ENCRYPTION ALGORITHMS*. Tech. rep. Alagappa University, Dept of Computer Sci & Engg, 2004.

[45] jlund: *Signal partners with Microsoft to bring end-to-end encryption to Skype*. 2018. url: `https://signal.org/blog/skype-partnership/` (visited on 06/15/2022).

[46] Kario, Hubert: *RSA AND ECDSA PERFORMANCE*. 2017. url: `https://securitypitfalls.wordpress.com/2014/10/06/rsa-and-ecdsa-performance/` (visited on 03/05/2022).

[47] Katz, Eric Dean ; Butler, Michelle ; McGrath, Robert: "A scalable HTTP server: The NCSA prototype". In: *Computer Networks and ISDN Systems* 27.2 (1994).

[48] Kessler, Gary C.: *An Overview of Cryptography*. 2022. url: `https://www.garykessler.net/library/crypto.html` (visited on 01/16/2022).

[49] Krawczyk, H. ; Eronen, P.: *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. 2010.

[50] Krawczyk, Hugo ; Paterson, Kenneth G. ; Wee, Hoeteck: "On the Security of the TLS Protocol: A Systematic Analysis". In: *Advances in Cryptology – CRYPTO 2013*. Springer Berlin Heidelberg, 2013.

[51] LibP2P: *Bundles*. url: `https://libp2p.io/bundles` (visited on 06/14/2022).

[52] LibP2P: *libp2p implementation in Go*. 2022. url: `https://github.com/libp2p/go-libp2p` (visited on 06/14/2022).

[53] LibP2P: *Noise / TLS official libp2p implementation*. 2022. url: `https://github.com/libp2p/go-libp2p/tree/master/p2p/security` (visited on 06/14/2022).

[54] LibP2P: *p2p chat app with libp2p*. 2022. URL: https://github.com/libp2p/go-libp2p/tree/master/examples/chat (visited on 06/14/2022).

[55] LibP2P: *The JavaScript Implementation of libp2p networking stack*. 2022. URL: https://github.com/libp2p/js-libp2p (visited on 06/14/2022).

[56] libp2p: *Design considerations for the libp2p TLS Handshake*. 2019. URL: https://github.com/libp2p/specs/blob/master/tls/design%20considerations.md (visited on 06/14/2022).

[57] libp2p: *libp2p Git Repository - key.go*. 2021. URL: https://github.com/libp2p/go-libp2p-core/blob/master/crypto/key.go (visited on 06/15/2022).

[58] libp2p: *libp2p TLS Git Repository*. 2022. URL: https://github.com/libp2p/go-libp2p/tree/master/p2p/security/tls (visited on 06/15/2022).

[59] libp2p: *noise-libp2p - Secure Channel Handshake*. 2021. URL: https://github.com/libp2p/specs/blob/master/noise/README.md (visited on 04/05/2022).

[60] libp2p: *rust-libp2p Git Repository*. 2022. URL: https://github.com/libp2p/rust-libp2p (visited on 06/15/2022).

[61] libp2p: *sec Documentation*. 2022. URL: https://pkg.go.dev/github.com/libp2p/go-libp2p-core/sec?utm_source=godoc (visited on 06/16/2022).

[62] library, Go Standard: *flag package*. 2022. URL: https://pkg.go.dev/flag (visited on 06/19/2022).

[63] library, Go Standard: *Golang TLS package*. 2022. URL: https://pkg.go.dev/crypto/tls (visited on 06/15/2022).

[64] library, Go Standard: *testing package*. 2022. URL: https://pkg.go.dev/testing (visited on 04/21/2022).

[65] LLC, RadicalApp: *RadicalApp LLC*. 2020. URL: https://github.com/RadicalApp (visited on 06/15/2022).

[66] LLC, RadicalApp: *RadicalApp LLC*. 2022. URL: https://play.google.com/store/apps/developer?id=Radical+App+LLC&hl=en&gl=US (visited on 06/15/2022).

[67] Mallik, Avijit: *Man-In-The-Middle-Attack: Understanding in simple words*. 2018.

[68] Maly, Robin Jan: *Comparison of Centralized (Client-Server) and Decentralized (Peer-to-Peer) Networking*. Tech. rep. 2003.

[69] moxieo: *Facebook Messenger deploys Signal Protocol for end-to-end encryption*. 2016. URL: https://signal.org/blog/facebook-messenger/ (visited on 06/15/2022).

[70] moxieo: *Signal on the outside, Signal on the inside*. 2016. URL: https://signal.org/blog/signal-inside-and-out/ (visited on 06/15/2022).

[71] moxieo: *WhatsApp's Signal Protocol integration is now complete*. 2016. URL: https://signal.org/blog/whatsapp-complete/ (visited on 06/15/2022).

[72] Nadeem, A. ; Javed, M.Y.: *A Performance Comparison of Data Encryption Algorithms*. Tech. rep. 2005.

[73] Nakajim, Junko ; Matsui, Mitsuru: *Performance Analysis and Parallel Implementation of Dedicated Hash Functions*. Tech. rep. Mitsubishi Electric Corporation, 2002.

[74] Nakov, Svetlin: *ECC Algorithms*. 2021. url: https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc (visited on 06/15/2022).

[75] Nakov, Svetlin: *ECDH Key Exchange*. 2021. url: https://cryptobook.nakov.com/asymmetric-key-ciphers/ecdh-key-exchange (visited on 06/15/2022).

[76] Narayanan, Arvind et al.: *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[77] NIST: *Message Authentication Codes MAC*. 2021. url: https://www.heise.de/news/FBI-ueber-Messenger-An-welche-Daten-von-WhatsApp-Co-US-Strafverfolger-kommen-6282456.html (visited on 03/05/2022).

[78] OmniSci: *Client-Server*. url: https://www.omnisci.com/technical-glossary/client-server (visited on 12/11/2021).

[79] Panda, Madhumita: *Performance Analysis of Encryption Algorithms for Security*. 2016.

[80] Paulevé, Loïc ; Jégou, Hervé ; Amsaleg, Laurent: *Locality sensitive hashing: A comparison of hash function types and querying mechanisms*. Tech. rep. IEEE, 2004.

[81] Perrin, Trevor: *Noise Protocol Framework*. 2022. url: https://noiseprotocol.org/noise.html (visited on 06/15/2022).

[82] Pohlmann, Norbert: *One-Way-Hashfunktionen*. url: https://norbert-pohlmann.com/glossar-cyber-sicherheit/one-way-hashfunktionen/ (visited on 03/05/2022).

[83] Polkadot: *Polkadot Documentation - FAQ*. 2022. url: https://wiki.polkadot.network/docs/faq#does-polkadot-use-libp2p (visited on 06/14/2022).

[84] pprof: *pprof Git Repository*. 2022. url: hhttps://github.com/google/pprof (visited on 06/14/2022).

[85] Qualys: *SSL Pulse*. 2022. url: https://www.ssllabs.com/ssl-pulse/ (visited on 06/14/2022).

[86] RadicalApp: *libsignal-protocol-go Git Repository*. 2017. url: https://github.com/RadicalApp/libsignal-protocol-go (visited on 06/15/2022).

[87] Rountree, Derrick: *Security for Microsoft Windows System Administrators*. Syngress, 2011.

[88] Rubertis, Antonio De et al.: *Performance evaluation of end-to-end security protocols in an Internet of Things*. Tech. rep. IEEE, 2013.

[89] Schollmeiter, Rüdiger et al.: *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications*. Tech. rep. Institute of Communication Networks, Technische Universität München, 2021.

[90] Science, Teach Computer: *Symmetric and Asymmetric Encryption*. 2020. url: https://csrc.nist.gov/Projects/Message-Authentication-Codes (visited on 03/22/2022).

[91] Shaik, Farooq et al.: “Certificate Based Authentication Mechanism for PMU Communication Networks Based on IEC 61850-90-5”. In: 7 (2018).

[92] Sicherheit in der Informationstechnik, Bundesamt für: *Hashfunktion*. 2021. url: https://www.bsi.bund.de/DE/Service-Navi/Cyber-Glossar/Functions/glossar.html?nn=520190&cms_lv2=132804 (visited on 03/05/2022).

[93] Signal: *Signal » Documentation*. 2022. url: https://signal.org/docs/ (visited on 06/15/2022).

[94] Signal: *The X3DH Key Agreement Protocol*. 2016. url: https://signal.org/docs/specifications/x3dh/ (visited on 06/15/2022).

[95] signalapp: *libsignal Git Repository*. 2022. url: https://github.com/signalapp/libsignal (visited on 06/15/2022).

[96] signalapp: *libsignal-protocol-c Git Repository*. 2020. url: https://github.com/signalapp/libsignal-protocol-c (visited on 06/15/2022).

[97] Specs, LibP2P: *libp2p specification framework – lifecycle: maturity level and status*. 2019. url: https://github.com/libp2p/specs/blob/master/00-framework-01-spec-lifecycle.md (visited on 06/15/2022).

[98] Specs, LibP2P: *libp2p TLS Handshake*. 2019. url: https://github.com/libp2p/specs/blob/master/00-framework-01-spec-lifecycle.md (visited on 06/15/2022).

[99] Specs, LibP2P: *noise-libp2p - Secure Channel Handshake*. 2021. url: https://github.com/libp2p/specs/tree/master/noise (visited on 06/15/2022).

[100] Specs, LibP2P: *Secio 1.0.0*. 2021. url: https://github.com/libp2p/specs/tree/master/secio (visited on 06/15/2022).

[101] Steinmetz, Ralf ; Wehrle, Klaus: *Peer-to-Peer Systems and Applications*. Springer, 2005.

[102] Substrate: *Substrate Documentation - Architecture*. 2022. url: https://docs.substrate.io/v3/getting-started/architecture/ (visited on 06/14/2022).

[103] Trautwein, Dennis ; Schubotz, Moritz ; Gipp, Bela: *Introducing Peer Copy - A Fully Decentralized Peer-to-Peer File Transfer Tool*. 2021. (Visited on 06/10/2022).

[104] vasa: *Understanding IPFS in Depth: What is Libp2p?* 2019. url: https://medium.com/coinmonks/understanding-ipfs-in-depth-5-6-what-is-libp2p-f8bf7724d452 (visited on 06/14/2022).

[105] Vyzovitis, Dimitris ; Psaras, Yiannis: *GossipSub: A Secure PubSub Protocol for Unstructured, Decentralised P2P Overlays*. 2019. (Visited on 06/10/2022).

[106] Wagner, David et al.: *Cryptographic Hashes*. 2021. url: https://textbook.cs161.org/crypto/hashes.html (visited on 03/05/2022).

[107] Wagner, David et al.: *Message Authentication Codes (MACs)*. 2021. url: https://textbook.cs161.org/crypto/macs.html (visited on 03/05/2022).

[108] WhatsApp: *WhatsApp Encryption Overview - Technical white paper*. 2021. url: https://scontent-dus1-1.xx.fbcdn.net/v/t39.8562-6/271639644_1080641699441889_2201546141855802968_n.pdf?_nc_cat=108&ccb=1-7&_nc_sid=ad8a9d&_nc_ohc=cyr6x-PanboAX-Lcm2D&_nc_ht=scontent-dus1-1.xx&oh=00_AT_o2nCWkMcSaBFf2TSI3QYuM_cYCf0tN3uhLy1r8mZWxA&oe=62D4B23D (visited on 06/15/2022).

List of Figures

List of Tables

Statement of Authorship

Bonn, July 15, 2022

Luca Weist