Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Computer Science
25 | LIU-IDA/LITH-EX-A--25/020--SE

Implementation and Performance Evaluation of a Chameleon Signature Protocol

Implementation och prestandautvärdering av ett protokoll med kameleontsignaturer

John Engberg

Supervisor : Reyhane Falanji
Examiner : Mikael Asplund

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Chameleon Hash Functions (CHF) allow authorized entities to modify the input message without altering the hash value. Hence, by using a CHF in a digital signature scheme i.e. a chameleon signature scheme, authorized entities may alter signed messages without corrupting the signature. Communication protocols using chameleon signature schemes have not been extensively evaluated related to performance, especially not in real-world environments. In this thesis, a CHF and a communication protocol using said CHF are implemented. The protocol implementation is evaluated in a public cloud environment to investigate how the chameleon signature scheme affects the performance of the protocol. The protocol implementation is compared against a protocol using conventional digital signatures. The results show that the chameleon signature scheme reduces the network overhead while increasing the computational overhead of the protocol. Furthermore, the results show that the computational overhead of the chameleon signature scheme increases with the security parameter. However, by setting the security parameter such that an adequate level of security is achieved, the communication protocol using the CHF performs better than the protocol using the conventional digital signature scheme.

Acknowledgments

A big thank you to Reyhane Falanji and Mikael Asplund for their help throughout this thesis.

Contents

List of Figures

List of Tables

1 Introduction

Ever since its creation, the RSA digital signature scheme has enabled the signing and verification of messages [22]. By using such digital signatures communicating nodes can trust that received messages have not been tampered with and originated from the correct node [16]. However, networking concepts such as edge computing can benefit from modifiable signatures [7]. Chameleon signature schemes provide the possibility to alter signed messages while preserving the digital signatures, to authorized modifiers. This can potentially alleviate cloud and edge nodes of additional computations as well as reduce the amount of data being transmitted. However, performance evaluations of chameleon signature schemes are limited and unclear in the available literature.

1.1 Motivation

For example, consider a vehicle equipped with a camera, connected to other vehicles in a Vehicle Edge Computing (VEC) network. The camera has recorded a bicyclist who might cross the road dangerously. To reduce the risk of an accident, the collected data should be communicated to nearby vehicles. However, to preserve privacy, the identifying traits of the cyclist must be blurred before distribution. As such, the image processing task will be delegated to an edge server that will complete the task and distribute the data. Collisions with the cyclist are avoided while preserving privacy. However, no vehicle should act on the data without verifying its authenticity or integrity. Neglecting to do so provides an opportunity for a malicious entity to control the behavior of the vehicle potentially. In other words, the data are sent from the actual vehicle and have not been modified by another entity other than the edge server. Digital signatures can provide both authenticity and integrity [16].

Let us consider the scenario in the previous paragraph again, with the additional criteria that all data should have a digital signature of the vehicle that collected the data. Now the vehicle has to create two digital signatures every time it needs to distribute an image with privacy-preserving processing. Firstly, the unprocessed image has to be signed and sent to the edge server. Secondly, the blurred image has to be signed before distribution, since the image has been altered and its corresponding signature invalidated. To reduce the number of transmissions and the additional computation on the vehicle, the edge server could create a new signature after processing. However, this would require the edge server to access the signature keys of all vehicles, effectively creating a single point of failure. Additionally, it demands that all vehicles trust the edge server enough to share their secrets which is unrealistic.

An alternative approach is to use a chameleon signature scheme, a special case of the hash-and-sign approach of digital signatures where the hash function used is a chameleon hash function [14]. Chameleon hash functions (CHF) allow designated entities to find collisions. As such, an edge server can modify an image without modifying the image hash. If the image hash is not modified, then the corresponding signature is still valid. Thus, the vehicle does not have to re-sign the image. Using a chameleon signature scheme reduces the number of signatures created and the number of messages transmitted. However, it is not clear that using a chameleon signature scheme will lead to increased performance in communication networks, as, to the best of our knowledge, there are no studies evaluating the performance of chameleon signatures in communication protocols in a real-world environment.

1.2 Aim

This thesis aims to determine the performance impact of using chameleon signature schemes. Specifically, to determine the performance impact of chameleon signature schemes when a signed message requires modifications by an external entity before distribution, e.g. the scenario described in section 1.1. The goal is to implement a chameleon signature scheme and a protocol using it. A performance evaluation of nodes, deployed in a cloud environment, communicating through the implemented protocol will be conducted.

1.3 Research questions

The following research questions have been formulated to examine the performance of chameleon signature schemes:

RQ1: How does the chameleon signature scheme affect the efficiency of authenticated data distribution with modifications?

- RQ1.1: How does the chameleon signature scheme affect the computational overhead per node in the network?
- RQ1.2: How does the chameleon signature scheme affect the network overhead?

To enable answering the main question, two sub-questions have been created. Each sub-question focuses on a specific metric related to efficiency.

1.4 Approach

A CHF function is selected based on available literature and then implemented. Furthermore, a simplified version of the Modifiable Message Exchange Protocol (MoMEP) is implemented using the CHF implementation. In short, MoMEP uses chameleon signatures to enable messages to be modified while maintaining accountability. The adaption and implementation of MoMEP, which we will refer to as the CHF protocol (CHP), is the subject of evaluation. As a means of comparison, two baselines are created using conventional digital signatures. To evaluate the performance of CHP, the factorial experiment design is used [11], with the security parameter and the message size as factors. Furthermore, performance data is collected from protocol executions by multiple nodes in a cloud environment.

1.5 Contributions

- An implementation of a protocol relying on a chameleon signature scheme.

- Performance data of a chameleon signature scheme from a real-world environment.
- An extended understanding of the viability of chameleon signature schemes.

2 Background

This chapter covers the necessary theoretical concepts in the thesis. Firstly, the purpose of using hash functions together with digital signature schemes is explained. Secondly, the fundamentals of CHFs and a brief overview of the different levels of collision resistance for CHFs is described. Thirdly, the part of MoMEP that provides the foundation of CHP, which is covered in detail in chapter 4, is introduced. Finally, the Kruskal-Wallis test is explained since the output of the test is used frequently in chapter 6.

2.1 Digital Signature Performance

Digital signatures are based on public key cryptography and do not scale well with increased message size [16]. A common way to improve the efficiency of digital signatures is the hash-and-sign approach [16]. Instead of signing the entire message, a hash function is used with the message as input to produce a fixed-length output that can be signed. By applying the hash function to the message before signing, the cost of computing the digital signature is reduced. However, when the message is updated, the owner of the signing key needs to recompute the hash and sign it again.

2.2 Chameleon Hash Functions

This is an abstract description of a CHF based on the initial proposal by Krawczyk et al. [14]. A CHF supports three functions: $create\_hash$, $find\_collision$, and $create\_keys$. A user of a CHF needs two keys: one public hashing key and one secret key. These keys are acquired through the $create\_keys$ function and are needed as input to the other functions.

Given the security parameter λ, the $create\_keys$ eq. (2.1) returns one public key for creating hash values, and one secret key for finding hash collisions.

$$create\_keys(\lambda) \rightarrow (key, secret\_key) \tag{2.1}$$

Given the public key, a message, and a random value, the $create\_hash$ eq. (2.2) returns the hash value.

$$create\_hash(key, m_1, r_1) \rightarrow hash \tag{2.2}$$

Given the secret key, the new message, the original message, and the random value used to create the hash value of the original message, the $find\_collision$ eq. (2.3) returns the random value for which the new message and the original message produce the same output in eq. (2.2).

$$find\_collision(secret\_key, m_2, m_1, r_1) = r_2 \tag{2.3}$$

The CHF is designed in such a way that it is computationally unfeasible to find a collision for a message unless the user is in possession of the secret key. If the user is in possession of the secret key, then it should be computationally simple to find a collision. However, Ateniese and Medeiros [3] found that the definition proposed by Krawczyk and Rabin [14] suffers from the key-exposure problem, which means that an adversary only needs to observe one collision in order to retrieve the secret key.

2.3 Chameleon Hash Function Collision Resistance

Derler, Samelin, and Slamanig [6] formalize the notion of collision resistance for CHFs such that a CHF can have either: weak collision resistance (W-CollRes), extended collision resistance (E-CollRes), standard collision resistance (S-CollRes), or full collision resistance (F-CollRes). CHFs with W-CollRes allow an adversary to derive the secret key used after observing one collision. The original CHF proposed by Krawczyk and Rabin [14] is W-CollRes.

2.4 Modifiable Message Exchange Protocol

The Modifiable Message Exchange Protocol (MoMEP) enables authenticated data distribution with delegated modifications [8]. Through the use of a chameleon signature scheme, designated Modifier nodes can alter signed messages while keeping the signature intact. Figure 2.1 illustrates an exchange between three nodes using MoMEP.

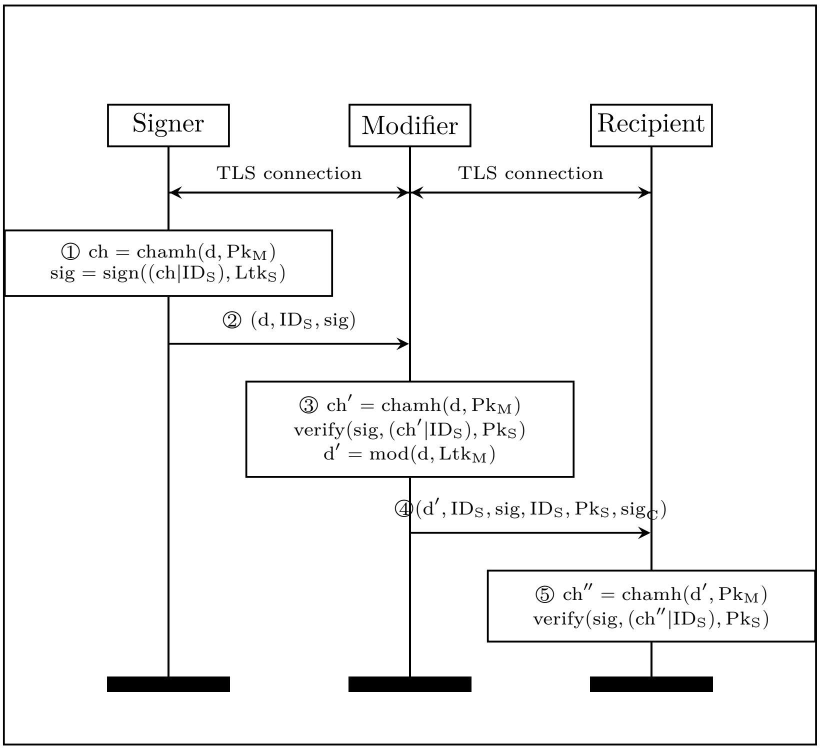


Figure 2.1: Sequence diagram of the message transmission phase of MoMEP, **A-alternative**, in which the modifier alters data. Permission to use obtained from Reyhane Falanji.

2.5 Kruskal-Wallis H Test

The Kruskal-Wallis H Test tests whether samples originate from the same distribution or not [15]. It is a non-parametric test, meaning that the test makes no assumptions about the underlying distribution. The null hypothesis of the test is that all samples originate from the same distribution. The output of the test is referred to as the test statistic or H value. Generally, large values for the test statistic lead to a rejection of the null hypothesis. However, depending on the number of samples and observations per sample, different approximations can be made for what the test statistic should be if all samples originate from the same distribution.

The SciPy implementation of the Kruskal-Wallis H test returns the test statistic and a p-value [26]. The p-value, signifies the probability that the test statistic is in line with the approximated distribution. The lower the p-value, the lower the probability of observing such different distributions without rejecting the null hypothesis.

3 Related Work

CHFs have existed for over two decades, and their performance has been evaluated in both studies proposing new CHFs and studies using CHFs. However, most performance evaluations are limited in scope, focusing on the time it takes to execute on a single device. Generally, the studies place more emphasis on security rather than performance. With that said, this chapter covers the most relevant studies to this thesis.

3.1 Performance Evaluation of CHF in the Blockchain

One application of CHFs is a redactable blockchain, first proposed by Ateniese et al. [1]. A redactable blockchain is a blockchain that can be modified by a selected group of entities. Allowing alterations of a blockchain is orthogonal to the concept itself, that data only can be appended. Nevertheless, the authors argue that some blockchain applications can benefit from allowing modifications. An example of such a modification is removing illegal content stored on the blockchain. Furthermore, the authors identify an issue with the CHF proposed by Krawczyk et al. [14]. The issue is that an adversary only needs to observe one CHF collision in order to determine the secret key used. To enable the use of CHF in this application, the authors devised a new CHF with extended collision resistance. With their CHF, collisions can be observed without revealing the secret key thus only allowing authorized entities to modify the blockchain. The authors implemented their redactable blockchain based on Bitcoin core. The CHF implementation was created using C/C++ and the Bignum library from OpenSSL. All parameters for the CHF were hard-coded in the implementation and were not included in the paper. A performance evaluation of the implementation was conducted using a network regression test feature of Bitcoin. The authors evaluated the time it took for their implementation to create, redact, and remove blocks. Blocks with varying sizes were used during the evaluation and time was measured using the time.time() function in Python. The Bitcoin code, that the implementation was based on, was used as a baseline. The performance evaluation indicated that the overhead imposed by the CHF was negligible.

Niya et al. [20, 21] aim to improve the redactable blockchain proposed by Ateniese et al. [1]. Instead of a single entity performing all modifications of the blockchain, each user has an individual secret key. When a block of data is added to the blockchain, it is hashed with the user's public key. As such, the user can find collisions for the hash since it is in possession of the corresponding secret key. The goal of their work is to increase the compliance of blockchain technology with the General Data Protection Regulation (GDPR). This means that the blockchain needs to comply with a user's "Right to be forgotten". Hence, if the user wants to remove their data from the blockchain, they can find a collision for an empty message and publish the result in an update request. The update request is then validated and processed by designated nodes. Due to the collision-resistant property of the CHF, no malicious third party can forge a valid update request. Their implementation allows multiple entities to modify data. However, the update requests contain traces of data that was deleted which reduces the overall GDPR compliance of the implementation. The only performance metric of their implementation is that it takes less than a minute to update an existing block in the blockchain.

An application of CHFs in VEC is proposed by Lu et al. [17]. The purpose of their work is to provide efficient and secure data access for edge nodes in VEC. The authors call their implementation Acce-chain and it is a modifiable blockchain using CHF. The authors utilize the modifiable blockchain to overwrite old data in the edge servers while preserving all historic data in the cloud layer, keeping only the relevant data at the edge to improve efficiency. They devise a threshold CHF scheme, named Dynamic Threshold Trapdoor Chameleon Hash (DTTCH), based on the work of Khalili et al. [13]. DTTCH allows an arbitrary group of edge servers to modify the blockchain, as long as the number of servers exceeds the defined threshold. The authors implement DTTCH and a prototype of the Acce-chain in Python. The performance of the DTTCH implementation is evaluated on a single computer effectively, multiple computers with the same specifications were used. The average time from 1000 executions with a message of 1KB was used in the paper. An entire network of 10 physical edge nodes located in four separate data centers is used to evaluate the performance of the Acce-chain implementation. The latency and throughput of the rewrite operation in the Acce-chain performed worse than the baseline. However, evaluation of the query times of different data sets showed that Acce-chain increased query time performance.

3.2 Performance Evaluation of CHF in Communication Protocols

Another use of CHFs for vehicular communication is proposed by Chen et al. [5]. They propose a new emergency reporting scheme called Secure and Onboard-unit-friendly Emergency Reporting Scheme (SOERS). The purpose of their scheme is to handle multiple vehicles reporting the same incident efficiently and to prevent malicious entities from reporting false accidents. To accomplish this, the authors combine a multi-signature scheme with a CHF. A multi-signature scheme enables n signers to create a signature for m such that the aggregated signature remains small and efficiently verifiable. Their scheme is implemented using elliptic curve cryptography in Java and evaluated on a single computer. The communication overhead, running time of the different algorithms in the scheme, and the total running time of the scheme based on the number of signers were evaluated. Each evaluation was run 50 times and the average value was used in the paper. Two other emergency reporting schemes were used as baselines for comparison. The performance evaluation of the SOERS implementation shows that it can reduce the resource usage of the vehicles and edge servers. One limitation of their scheme is that the scheme requires one vehicle to perform most of the computations without an incentive. The authors suggest combining their scheme with blockchain technology in the future, to provide such an incentive.

Xu et al. [27] propose an image modification scheme using a CHF. Their scheme uses the CHF proposed by Ateniese and de Medeiros [2] combined with the RSA signature scheme [22]. Using the CHF, sensitive areas of an image can be altered for privacy reasons while preserving the authenticity of the image. In their proposed scheme, editable areas of an image are called blocks. The owner of the image decides which blocks are editable. The editable blocks can then be altered by designated modifiers while preserving the initial signature created by the image owner. The authors create an implementation in C++ and state that it is efficient enough for practical use in applications with both security and privacy requirements. The performance of the implementation is evaluated on a single computer. For the evaluation, a 1024 bits RSA key is used. However, the security parameter of the CHF used for the evaluation is unknown. The performance evaluation is conducted on different images. With variations in the number of areas, number of immutable areas, and sizes. Furthermore, the authors compared the performance of their implementation with the results of an image modification scheme proposed by Chen et al. [4].

4 Designing the Chameleon Hash Protocol

This chapter describes the design decisions and choices concerning the implementation of CHP in software. In section 4.1, the design of CHP and how it differs from MoMEP is covered. The selection and implementation of a CHF is covered in section 4.2. The CHP implementation in software is then covered in detail in section 4.3. The choice of programming language is described in section 4.4.

4.1 Adaption From MoMEP

CHP deviates from MoMEP in the message sent between the Modifier and the Recipient in fig. 2.1. In CHP, the Modifier does not send the additional id of the Signer (IDs), the public key of the Signer (PKs), and the signature of PKs by the certificate authority (SIGc) to the Recipient. In the context of CHP, the assumption is that the Recipient has acquired PKs before the protocol is executed by the three participating nodes. The transmission and subsequent signature verification conducted by the Recipient has nothing to do with CHF since the signature by the certificate authority is on a conventional hash value of PKs. Therefore, the verification of SIGc will only obscure the computational overhead imposed by the chameleon signature scheme in CHP. Furthermore, there are multiple scenarios when the additional information included in MoMEP is superfluous. The Recipient could request PKs directly from the certificate authority, or the Modifier, during the setup phase. The Recipient could save PKs during the first transmission. Upon receiving the next message from the Modifier, verification of the message can be commenced directly after receiving the first half of the transmission while the latter half can be disregarded completely.

The data flow between nodes in the CHP is described in fig. 4.1. The figure contains pseudocode to signify the actions performed by each node. The *create_hash* function is the same as described in section 4.2.2. The *modify* function denotes that the message is first modified and then the *find_collision* function, described in section 4.2.2 is run. The *sign* and *verify* functions refer to the corresponding functions in the digital signature scheme. The || refers to the concatenation of two strings. Variables are used on the left-hand side of equal signs to denote that they get assigned the output of a function. The pre-defined variables used in this figure are defined as follows:

- msg: An arbitrary sequence of bytes of a particular length.

- CPKm: The public key of the CHF, the lowercase m signifies that the owner of the secret key is the Modifier.
- CSKm: The secret key of the CHF used to find hash collisions, the lowercase m signifies that the owner of the secret key is the Modifier.
- IDs: A string identifying the Signer.
- SKs: The secret key used to digitally sign messages using RSA, the lowercase s signifies that the owner of the secret key is the Signer.
- PKs: The public key used to verify digitally signed messages using RSA, the lowercase s signifies that the owner of the secret key is the Signer.

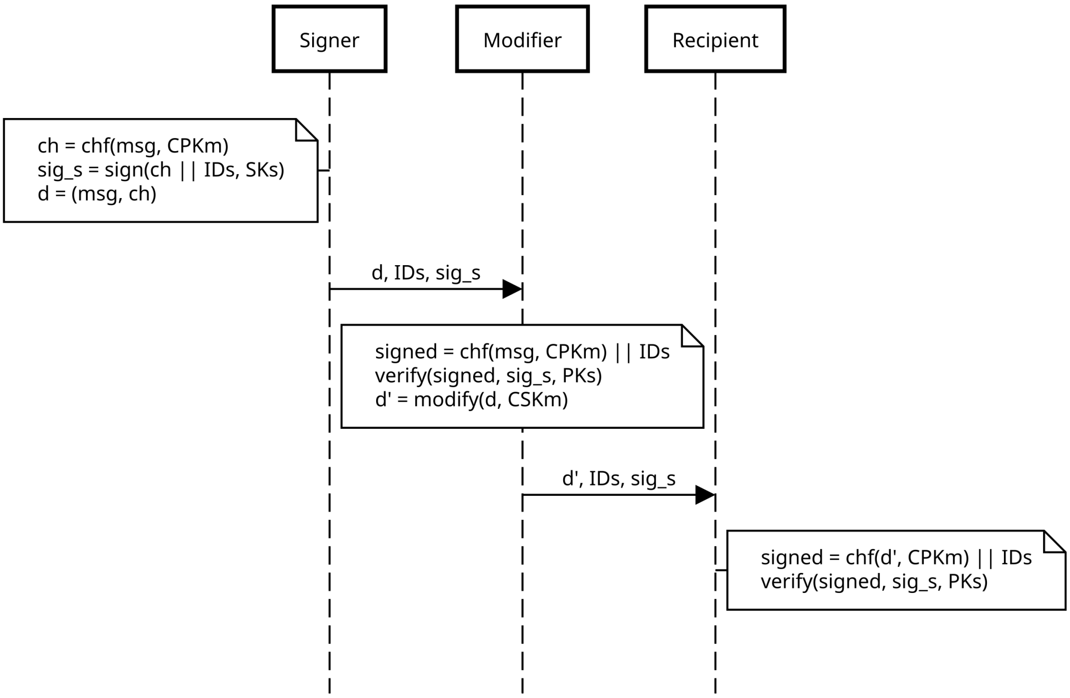


Figure 4.1: Data flow between nodes in the CHP implementation

4.2 Chameleon Hash Function

The core component of CHP is the CHF, as such a software implementation of one such function is required. Although there are existing implementations on GitHub [10], none of them appeared to be directly usable for the implementation of CHP. Either because of the lack of references to the corresponding literature in the repository, or due to the CHF implemented being more complex than required for use in CHP.

4.2.1 Selection

The first step of implementing a CHF is to decide which one to implement. By inspecting existing implementations on GitHub, a partial implementation, of one of the CHFs proposed by Ateniese et al. [1] created by Niya, Willems, and Stiller [21], was found. The construction by Ateniese et al. [1] is of a more manageable complexity compared to some of the more recent CHFs, at least for a thesis more focused on computer science than mathematics. As such, a CHF with lower complexity will aid in producing an implementation with fewer errors compared with a CHF with higher complexity. Furthermore, their construction addresses the key exposure problem, where an adversary can derive the secret key used to find a hash collision after observing a single collision [3]. Thus, the two main arguments for choosing to implement this particular CHF are that the complexity is low enough and that it does not suffer from the key exposure problem.

4.2.2 Implementation

The CHF is implemented as a library, with an Application Programming Interface (API) closely modeled after the mathematical construction in [1]. Additionally, to facilitate the use of the library in the evaluation program three more functions are exposed in the API: one of which stores all parameters to disk, one which stores all public parameters to disk, and one which loads stored data. Parameter generation is generally more costly than loading them from disk, and since parameter generation is not a part of CHP, the additional functions in the API reduces the evaluation time required.

The public API of the CHF implementation looks as follows:

- $create\_keys(\lambda) \rightarrow parameters$. The parameters consist of two prime numbers p and q, a generator g, the public hashing key y, and the secret key x, where p is λ bits long and q is another prime number such that $q = (p-1)/2$. Furthermore, g is a generator for $\mathbb{QR}_p$ which is a subgroup of $\mathbb{Z}_p^*$ consisting of quadratic residues. Therefore, to choose a value for g, a random value h is selected such that $h \in [0, p)$ and if $g = h^2 \mod p$ is relative prime to p that value for g is chosen. Otherwise, the process is repeated until a valid g is found. The secret key x is a random value such that $x \in \mathbb{Z}_q$ and the public hashing key y is $y = g^x \mod p$. All other functions use the parameters p, q, and g. When a function uses either x or y it is explicitly noted.
- $create\_hash(y, m) \rightarrow (h, m, r, s)$. This function creates a chameleon hash of the message m using the public hashing key y. Where r and s are random integers from $\mathbb{Z}_q$.
- $create\_hash(y, m, r, s) \rightarrow (h, m, r, s)$. Creates a chameleon hash from the supplied arguments, which is useful when re-computing a hash value for verification purposes.
- $find\_collision(x, (h, m, r, s), m') \rightarrow (r', s')$. Using the secret key x, this function finds new random values *r'* and *s'* such that $create\_hash(y, m, r, s) = create\_hash(y, m', r', s')$.
- $from\_string(s) \rightarrow parameters$. Loads all parameters from a string representation.
- $to\_private\_string() \rightarrow s$. Creates a string from all parameters.
- $to\_public\_string() \rightarrow s$. Creates a string from all parameters except for the secret key x.

4.3 CHP Implementation

The CHP implementation uses the CHF library implementation described in section 4.2.2. The digital signature scheme used is RSA, provided by the OpenSSL library, following the Digital Signature Standard [23]. The actual protocol implementation is staged by running three instances of the same binary. Each instance is configured using the Command Line Interface (CLI), which is described in detail in section 4.3.1. The instances communicate with each other using a simple binary protocol over Transmission Control Protocol (TCP) sockets, which is further described in section 4.3.2. Once one Signer, one Modifier, and one Recipient are connected the protocol is executed. The protocol execution requires valid certificates and cryptographic parameters to be run, in addition to the three role instances.

4.3.1 Command Line Interface

The CHP implementation is realized by a single binary, to maximize code reuse and development speed. The CLI allows it to either only generate parameters, or be run as a single role in CHP i.e. Signer, Modifier, or Recipient. Examples of the different applications of the binary are listed below.

Listing 4.1 shows an example of parameter generation. The *–security* flag determines the length of the RSA keys used for the digital signature as well as the length of the prime number p in the CHF implementation. The output is stored in a directory designated by the security parameter, which in this example is *data/parameters/1024*. The output of a parameter generation run consists of six files:

Listing 4.1: Parameter Generation Example

```
chp --security 1024 generate
```

- chfparameters_private - All CHF parameters with one parameter per line as a hexadecimal string.
- chfparameters_public - All CHF parameters except for x, with one parameter per line as a hexadecimal string.
- modifier_private.der - The private RSA key associated with the Modifier role in the DER format.
- modifier_public.der - The public RSA key associated with the Modifier role in the DER format.
- signer_private.der - The private RSA key associated with the Signer role in the DER format.
- signer_public.der - The public RSA key associated with the Signer role in the DER format.

Listing 4.2 shows an example of an instantiation of the Signer role. The *–security* and *–cert-file* flags denote which parameters to load. The certificate file is used to validate the certificate used by the Modifier to establish a Transmission Layer Security (TLS) secured TCP connection. The *–modifier-addr* and *–recipient-addr* flags set the socket addresses used to communicate with other nodes (the recipient socket is only used for time measurements which is further explained in chapter 5). The *–message-size* flag determines the size, in bytes, of the message that is transmitted using CHP. The actual message is then created by the Signer using the *OpenSSL* function *rand_bytes*. The *–no-extra-signature* is explained in chapter 5, since it is only used for the evaluation.

Listing 4.2: Signer Instance Run Example

```
chp --security 1024 run signer \
    --cert-file cert.pem \
    --modifier-addr 10.0.0.1:8888 \
    --recipient-addr 10.0.0.2:8889 \
    --message-size 1000 \
    --no-extra-signature
```

An example run of a Modifier instance is shown in listing 4.3. The *–security* flag works as for the Signer role. The certificate related flags, *–cert-key-file* and *–cert-chain-file*, are used to establish TLS connections to the Signer and Recipient. Both the certificate private key and the signed certificate chain file are needed since the Modifier role acts as the server for the TLS connections. The *–signer-addr* and *–recipient-addr* denote which ports the Modifier should listen for incoming connections. The *–forward-only* flag will inhibit the Modifier from actually modifying the incoming message from the Signer. Thus, if the *–forward-only* flag is omitted, the Modifier will alter the message using the *Rust* standard vector function *rotate_left*. The function is *O(n)* where n is the length of the message, and the modification does not alter the length of the message. The *–has-signer-key* is only used for evaluation and is explained in chapter 5.

Listing 4.3: Modifier Instance Run Example

```
chp --security 1024 run modifier \
    --cert-key-file key.pem \
    --cert-chain-file cert-chain.pem \
    --signer-addr 0.0.0.0:8888 \
    --recipient-addr 0.0.0.0:8889 \
    --forward-only \
    --has-signer-key
```

The recipient role contains a subset of the flags of the Signer role, as seen in the recipient instantiation shown in listing 4.4. The only difference is that for the Recipient role, the flag used for time measurements is called *–signer-addr*, which is further explained in chapter 5.

Listing 4.4: Recipient Instance Run Example

```
chp --security 1024 run recipient \
    --cert-file cert.pem \
    --modifier-addr 10.0.0.1:8888 \
    --signer-addr 10.0.0.2:8889
```

4.3.2 Communication

The communication between the different role instantiations is conducted over either TLS sockets or TCP sockets. The communication required by CHP is conducted over TLS sockets and the additional communication required for time measurements is conducted over TCP sockets. All data that is sent over the sockets is converted to a vector of bytes. The vector is transmitted by first sending the length of the vector, followed by the vector itself. This simple binary protocol is used instead of more conventional protocols for the transmission of application data, such as HTTP, since the overhead is minimal. The certificates used to establish the TLS sockets were created following the guide by Jamie Nguyen [19].

4.4 Programming Language

For the CHF implementation and the CHP implementation, the *Rust* programming language is used. The main argument for using *Rust* is the lack of runtime and garbage collector [18]. A managed runtime and garbage collector complicates the performance evaluation of an application since they are non-deterministic [9]. Considering that the programming language itself is not central to this thesis, it is simpler to select a language without such features that will inevitably make the evaluation methodology more complex [9]. Another advantage of using *Rust* compared to other languages is the memory safety guarantee. As long as the program is written only in *Rust* there will be no errors related to memory e.g. dangling pointers [18]. The guaranteed absence of such errors should reduce overall development time, which makes *Rust* preferable to e.g. *C/C++*. The cryptographic library used is *OpenSSL* through the rust crate *rust-openssl*[24].

5 Evaluation Method

This chapter describes the method used to evaluate the CHP implementation. More specifically, the chapter describes what is measured, the baseline of comparison for the CHP implementation, how the collected measurements are analyzed, as well as the resulting data.

5.1 Evaluation Principles

The evaluation method used is based on the work done on benchmarking parallel computing systems by Hoefler and Belli [11]. Even though the system that is evaluated in this thesis is not a parallel computing system, most of the rules presented by Hoefler and Belli [11] apply to performance evaluations of computing systems in general. In particular, the following rules will be adhered to:

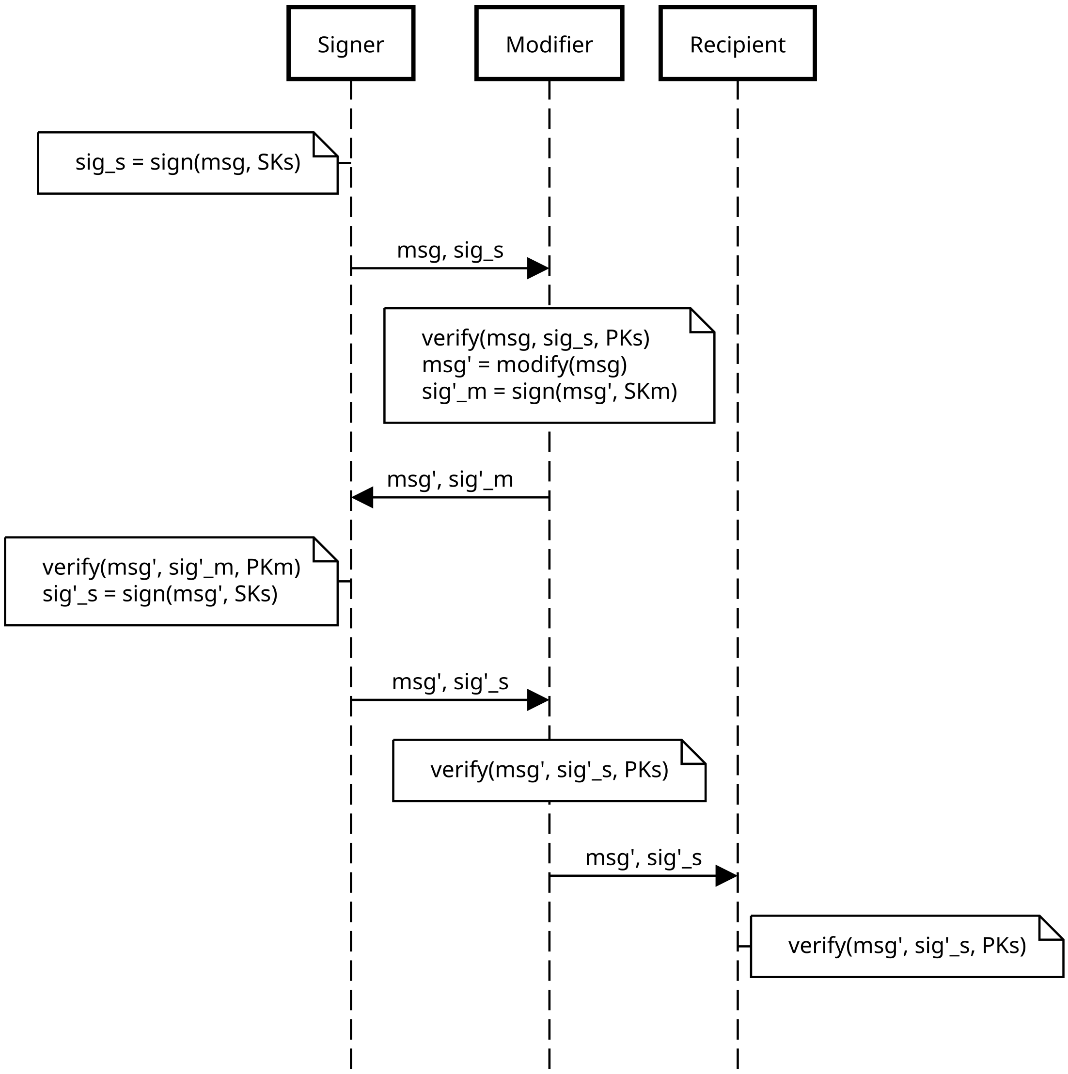
- Report if the measurement values are deterministic or not, and if not report confidence intervals (Rule 5): Confidence intervals are reported for the appropriate measurement values.
- Do not assume that the measurement values are normally distributed (Rule 6): The Shapiro-Wilk's test is used to determine normality of the data.
- Compare non deterministic data using a statistically sound method (Rule 7): The Kruskal-Wallis H test used is the same test as recommended by the authors but considering multiple varying factors instead of one [11].
- Document all varying factors of the setup (Rule 9): All of the source code for the application, shell scripts for evaluation, as well as hardware and software setup are included in this thesis. Either as a reference to a repository or as an appendix.

Due to the difficulty of finding a suitable baseline protocol for the CHP implementation, two baselines were created for this purpose. Both baselines aim to provide authenticated data distribution with delegated modifications using a conventional digital signature scheme, instead of a chameleon signature scheme as in CHP.

5.2 Naive Baseline

The naive baseline is constructed based on the premise that each role is only in possession of a private key if they are the owner. As such, the location of all keys is the same between the naive baseline and the CHP implementation. Due to the fact that the Modifier is not in possession of the key the Signer uses to sign messages, the modified message has to be sent to the Signer to get signed and then sent back to the Modifier. The messages are always sent together with the signature since the signature on its own has no meaning. Figure 5.1 is a message sequence diagram showing the naive baseline scenario.

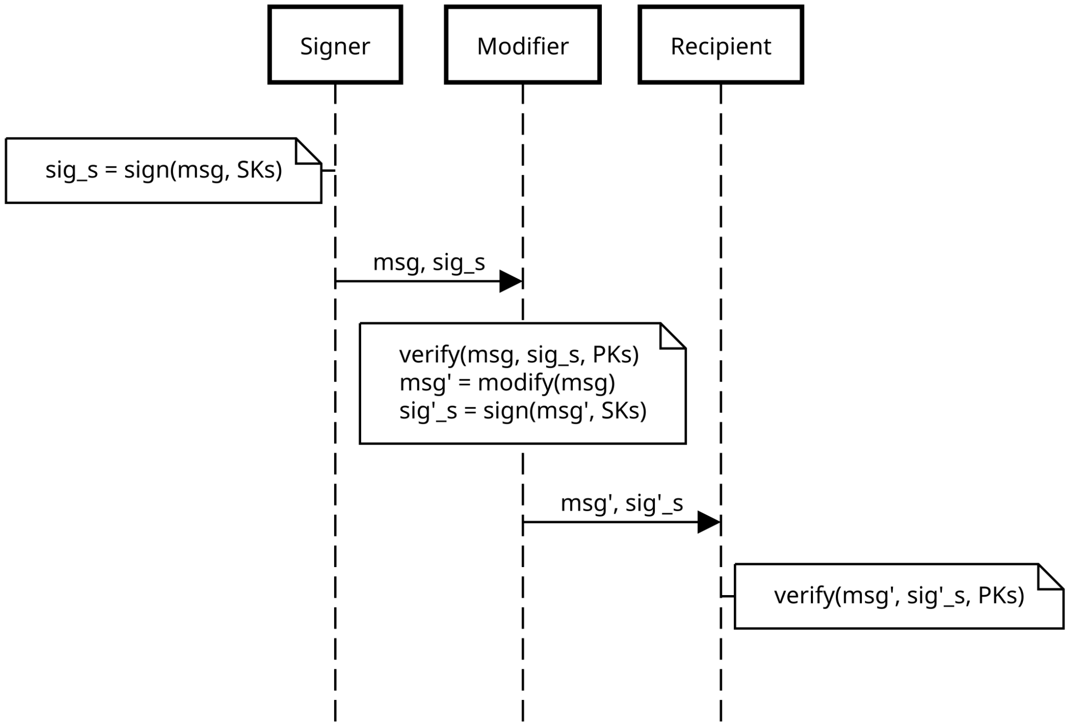
Figure 5.1: Data flow between nodes in the naive baseline implementation



5.3 Optimized Baseline

The optimized baseline is constructed based on the same data flow as in the CHP implementation. Thus, the modifier gets access to the Signer's private key, to remove the need to send data back and forth between the Signer and the Modifier. Figure 5.2 is a message sequence diagram showing the optimized baseline scenario.

Figure 5.2: Data flow between nodes in the optimized baseline implementation



The optimized baseline is not really feasible, since it relies on the Modifier node having access to the secret keys of all other nodes. The setup introduces a single point of failure, since the Modifier can fabricate message seemingly originating from any other node in the network. Thus from a security standpoint, the optimized baseline is not viable. However, it is interesting to use for comparison with CHP as an upper bound of authenticated data distribution with modifications using conventional digital signatures.

5.4 Metrics and Measurements

Since the goal is to evaluate the performance of the CHP implementation compared to the two baselines, measurements of all implementations are needed. The main measurements are different time intervals, both on a single node and on all nodes involved in the protocol [11]. The measurements on each node are written to a configured log file.

To measure time intervals, the *Instant* struct in the standard library which is a wrapper for an operating system monotonic clock. The time intervals measured are:

- before to after socket read: To measure how much time a node is waiting on and reading data from the network.
- before to after socket write: To measure how much time a node spends sending data to the network.
- before first socket read/write, to after the last socket read/write: To measure how much time a node spends actively partaking in the protocol.

In addition to measuring time intervals, the number of bytes sent and received from each node is measured. Furthermore, the measurements begin after all nodes have loaded their configuration and established connections to each other.

Each node then aggregates the measurements collected and produce the following metrics as output:

- Protocol Time (total_time_protocol_ns): The time in nanoseconds from the Signer creating the message to the Recipient completing the verification of the modified message. As such, this metric is only applicable for the Signer node. Before the first socket read/write, to after reading a byte from the Receiver node: To measure the total time taken to complete the protocol. Socket communication between the node initiating the protocol and the node ending the protocol is used to avoid clock synchronization issues. The round trip time of the end packet from the receiver is estimated afterward, such that it can be subtracted from the total time measured. The round trip time of the packet is estimated by sending five packets with a single byte payload and using the formula $EstimatedRTT = (1 - a) * EstimatedRTT + a * SampleRTT$, where $a = 0.125$.
- Execution Time (total_time_host_ns): The time in nanoseconds from the node being ready to partake in the protocol until the completion of the protocol based on the role of the node.
- Waiting Time (network_time_ns): The total time in nanoseconds the node spent waiting on a read or write call on a socket.
- Waiting Time Sending (network_time_sent_ns): The total time in nanoseconds the node spent waiting on a write call on a socket.
- Waiting Time Receiving (network_time_recv_ns): The total time in nanoseconds the node spent waiting on a read call on a socket.
- Bytes Transmitted (network_tx): The total number of bytes sent to other nodes.
- Bytes Received (network_rx): The total number of bytes received from other nodes.

The name in parentheses is the name of the corresponding metric in the output of the evaluation script.

5.5 Evaluation Configuration

Multiple executions of each protocol configuration are measured and collected. The first iteration of each configuration is treated as a warmup iteration and is discarded during analysis [11]. A protocol configuration consists of three variables: scenario, message size, and security parameter. The scenario determines the protocol used by the nodes and covers the two baselines, CHP, as well as when no modification is performed by the Modifier node. The message sizes range from 1 B to 100 MB. The upper limit is set based on the unlikeliness that message sizes will be larger than that in networks where the CHP protocol could be useful. The security parameters range from 1024 to 8192 bits. The security parameter determines the size of the numbers used for the calculations used when e.g. creating a hash value. The larger the number is, the longer the operation will take generally. Following the recommendations in [25], since safe primes are used for the modulus for the CHF, the modulus should be at least 2048 bits. The recommendations are about the Diffie-Hellman key exchange, which should still be applicable here since the security relies on the hardness of the discrete logarithm problem similar to the implemented CHF. All parameters used for the different security parameter values are stored on disk and loaded by each node before the measurements begin. Furthermore, all parameters can be found in the Gitlab repository containing all source code used in the thesis [12].

5.6 Evaluation Environment

The evaluation environment used consists of four virtual machines in Azure, with one virtual machine per node (Signer, Modifier, and Recipient) and one virtual machine to stage the protocol execution and collect the measurements. Executing the protocol in a cloud environment provides real-world interference such as network congestion and scheduling delay. The inclusion of such interference should provide more insight into how the different protocol implementations perform, in comparison to experiments using a simulated network, as Hoefler and Belli [11] did not consider papers that did not contain any performance measurements from real-world environments.

All virtual machines are identical and run on *ubuntu-24_04-lts* with Standard D2s v3 (2 vcpus, 8 GiB memory). The machines are connected to the same Azure virtual network, located in the Europe West region[1] and consisting of the addresses in 10.0.0.0/24. The full configuration of the Azure resources used for the evaluation can be found in appendix A.

Bash scripts were created to install the CHP implementation on each node, run the protocol configurations, and collect the measurements. The scripts are included in appendix B. The main evaluation script takes a single integer argument which determines how many iterations each protocol configuration should run. The evaluation script was run two times using a Lenovo laptop (6.13.8-arch1-1 x86_64 GNU/Linux, 13th Gen Intel(R) Core(TM) i7-1355U, 16GiB 1305MHz memory). In addition, the evaluation script was run five times using one of the virtual machines provisioned by Azure. The evaluation script only initializes the nodes for each protocol configuration and then collects the logs after completion. As such, the location and specs of the computer running the evaluation script are regarded as non-significant.

5.7 Data Analysis

The data subject to analysis consists of seven individual runs of the evaluation script. Two runs were done using the Lenovo laptop, both for 11 iterations. The remaining five runs were done using an Azure virtual machine for: 11, 21, 41, 51, and 28 iterations respectively. Since the first iteration of each run was considered a warmup iteration and subsequently ignored, the total number of iterations analyzed was 167. The number of iterations analyzed were limited in number by time constraints. More details of the data can be seen in appendix C.

Analysis of the measurements is conducted through a Python script relying on Pandas, Numpy, and Scipy for the data analysis functions. Each evaluation run produces a tarball containing the logs from all individual scenario runs. As such, the first part of the script consumes all tarballs in the working directory and creates a single csv file. The second part of the script consumes all data in the csv files stored and loads it into a Pandas data frame. The data frame contains all of the measurements and relevant configuration data for each node run, with one node run per row. Thus, to enable analysis of each protocol run, the data is grouped on scenario, message size, security parameter, iteration, and csv file name. By grouping the data on these columns, each group will contain only the Signer, Modifier, and Recipient node for a single protocol run. These three rows are combined by summing each column value. Since the Signer node is the only one reporting the total protocol time, this column will remain unchanged after the transformation. At this point, the data frame will contain multiple samples of all the different protocol run configurations.

By computing the sum for each column in the last step, extra calculations are needed to create a context for the total time spent executing on each node and total network time. Therefore one more column is created, to capture the sum of the total time spent executing and the total time spent waiting on network Input/Output (I/O). Using that column, two more columns are created capturing the relative time spent executing and waiting on network I/O).

[1]According to https://learn.microsoft.com/en-us/azure/virtual-machines/regions this means that the different virtual machines can be running in different data centers.

The next step is to aggregate all samples of each protocol configuration. Since the number of samples per configuration is relatively low, Shapiro-Wilk's test is suitable to determine if the samples are from a normal distribution [11]. Most of the time measurements are not from a normal distribution. As such the median with a 95% Confidence Interval (CI) is used as an aggregation method for all measurements since it does not rely on the data being drawn from a particular distribution [11].

To answer the research questions, the aggregated values have to be visualized and compared. Visualization is done using matplotlib and a Pandas function to generate latex tables from data frames. The Kruskal-Wallis test will be used to determine if there is a statistically significant difference between the median values [11]. More specifically, the Kruskal-Wallis test will determine if measured values of the CHP scenario and the baseline scenarios are drawn from the same distribution. In this thesis, a p-value of 0.05 or less is considered to indicate a statistically significant result. The analysis will be conducted for both a fixed security parameter and variable message length and vice versa.

6 Results

This chapter presents and describes data acquired from the analysis of the gathered measurements.

6.1 Protocol Time

Table 6.1 and table 6.2 show that there is a statistically significant difference in the total time to complete the protocol between CHP and the other scenarios when the security parameter is larger than 2048.

Table 6.1: Statistically significant Kruskal-Wallis test results of median 'Total protocol time' values between scenarios 'CHP' and 'Naive Baseline'

Test statistics	P-value	Security parameter (Bits)
8.877	0.003	6144
11.983	0.001	8192

Table 6.2: Statistically significant Kruskal-Wallis test results of median 'Total protocol time' values between scenarios 'CHP' and 'Opt Baseline'

Test statistics	P-value	Security parameter (Bits)
4.200	0.040	3072
8.272	0.004	4096
11.510	0.001	6144
15.015	0.000	8192

By viewing fig. 6.1 and fig. 6.2 it is evident that CHP requires more time to complete. However, the increase in completion time with message size is less than in the other scenarios. Table 6.3 and table 6.4 illustrate this in more detail, where CHP obtains faster times for the larger message sizes.

Table 6.3: Total protocol time with security parameter 6144

Message size (Bytes)	CHP median (95% CI)	Opt Baseline median (95% CI)	Naive Baseline median (95% CI)
1	0.342s (0.288s, 0.344s)	0.085s (0.081s, 0.086s)	0.152s (0.150s, 0.153s)
5	0.342s (0.288s, 0.344s)	0.086s (0.080s, 0.087s)	0.153s (0.151s, 0.174s)
10	0.343s (0.288s, 0.345s)	0.086s (0.080s, 0.087s)	0.152s (0.151s, 0.168s)
50	0.342s (0.288s, 0.344s)	0.086s (0.080s, 0.086s)	0.174s (0.153s, 0.176s)
100	0.342s (0.288s, 0.345s)	0.085s (0.080s, 0.086s)	0.152s (0.151s, 0.157s)
500	0.342s (0.287s, 0.346s)	0.086s (0.081s, 0.086s)	0.155s (0.152s, 0.175s)
1000	0.343s (0.287s, 0.345s)	0.045s (0.039s, 0.045s)	0.064s (0.056s, 0.066s)
5000	0.342s (0.288s, 0.344s)	0.045s (0.040s, 0.046s)	0.064s (0.057s, 0.066s)
10000	0.342s (0.288s, 0.345s)	0.045s (0.040s, 0.046s)	0.066s (0.056s, 0.067s)
50000	0.344s (0.289s, 0.346s)	0.047s (0.041s, 0.048s)	0.068s (0.059s, 0.069s)
100000	0.346s (0.290s, 0.349s)	0.049s (0.043s, 0.050s)	0.071s (0.061s, 0.072s)
500000	0.357s (0.300s, 0.359s)	0.065s (0.056s, 0.066s)	0.097s (0.096s, 0.098s)
1000000	0.372s (0.313s, 0.373s)	0.085s (0.077s, 0.086s)	0.127s (0.123s, 0.128s)
5000000	0.469s (0.393s, 0.472s)	0.227s (0.191s, 0.229s)	0.344s (0.288s, 0.348s)
10000000	0.644s (0.556s, 0.646s)	0.455s (0.400s, 0.458s)	0.723s (0.640s, 0.728s)
50000000	2.196s (1.975s, 2.202s)	2.402s (2.154s, 2.413s)	3.979s (3.607s, 3.988s)
100000000	4.118s (3.731s, 4.122s)	4.826s (4.345s, 4.834s)	8.023s (7.289s, 8.047s)

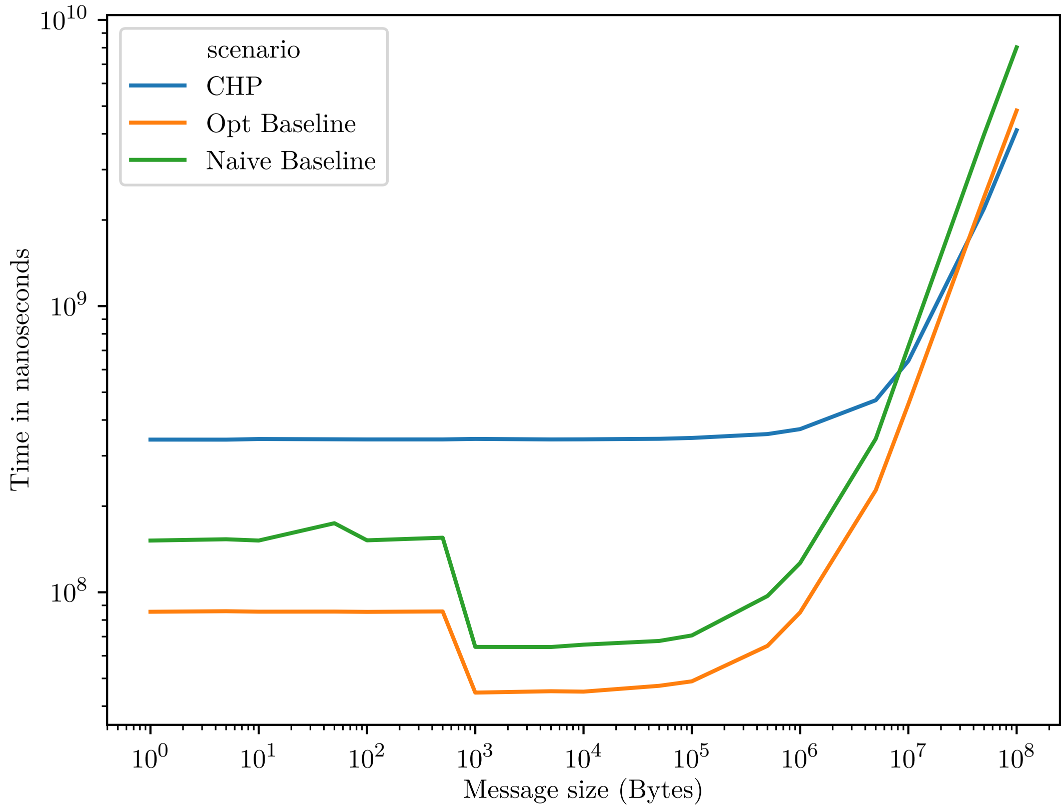


Figure 6.1: Total protocol time with security parameter 6144

Table 6.4: Total protocol time with security parameter 8192

Message size (Bytes)	CHP median (95% CI)	Opt Baseline median (95% CI)	Naive Baseline median (95% CI)
1	0.790s (0.667s, 0.796s)	0.137s (0.124s, 0.139s)	0.188s (0.186s, 0.189s)
5	0.793s (0.666s, 0.798s)	0.136s (0.124s, 0.138s)	0.186s (0.185s, 0.188s)
10	0.792s (0.666s, 0.798s)	0.137s (0.125s, 0.139s)	0.187s (0.187s, 0.189s)
50	0.792s (0.666s, 0.798s)	0.137s (0.124s, 0.138s)	0.186s (0.185s, 0.188s)
100	0.792s (0.667s, 0.798s)	0.137s (0.125s, 0.139s)	0.185s (0.182s, 0.187s)
500	0.791s (0.665s, 0.797s)	0.096s (0.084s, 0.097s)	0.140s (0.121s, 0.142s)
1000	0.791s (0.665s, 0.799s)	0.096s (0.084s, 0.098s)	0.140s (0.121s, 0.142s)
5000	0.792s (0.665s, 0.796s)	0.097s (0.084s, 0.098s)	0.140s (0.120s, 0.143s)
10000	0.793s (0.666s, 0.799s)	0.097s (0.084s, 0.097s)	0.141s (0.120s, 0.143s)
50000	0.792s (0.667s, 0.798s)	0.099s (0.085s, 0.100s)	0.143s (0.122s, 0.146s)
100000	0.794s (0.667s, 0.799s)	0.101s (0.087s, 0.102s)	0.146s (0.125s, 0.149s)
500000	0.808s (0.681s, 0.813s)	0.117s (0.102s, 0.118s)	0.173s (0.156s, 0.175s)
1000000	0.821s (0.721s, 0.828s)	0.136s (0.117s, 0.138s)	0.203s (0.170s, 0.204s)
5000000	0.918s (0.774s, 0.925s)	0.278s (0.235s, 0.280s)	0.419s (0.353s, 0.423s)
10000000	1.094s (0.931s, 1.099s)	0.505s (0.442s, 0.508s)	0.796s (0.699s, 0.801s)
50000000	2.645s (2.351s, 2.651s)	2.452s (2.192s, 2.458s)	4.051s (3.654s, 4.064s)
100000000	4.564s (4.104s, 4.574s)	4.875s (4.396s, 4.887s)	8.084s (7.372s, 8.111s)

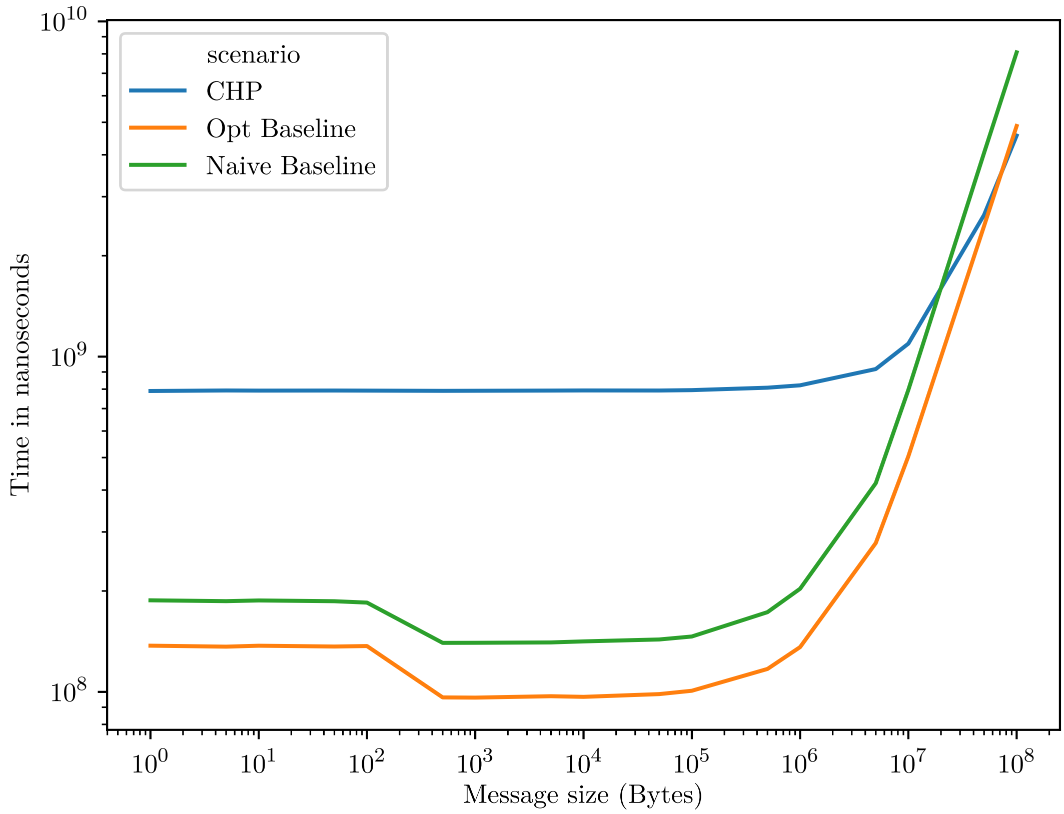


Figure 6.2: Total protocol time with security parameter 8192

Considering the measured times for smaller security parameter values, the difference between CHP and the baseline scenarios is not significant. However, in fig. 6.3 CHP performs better than both baselines from the message size of 5MB.

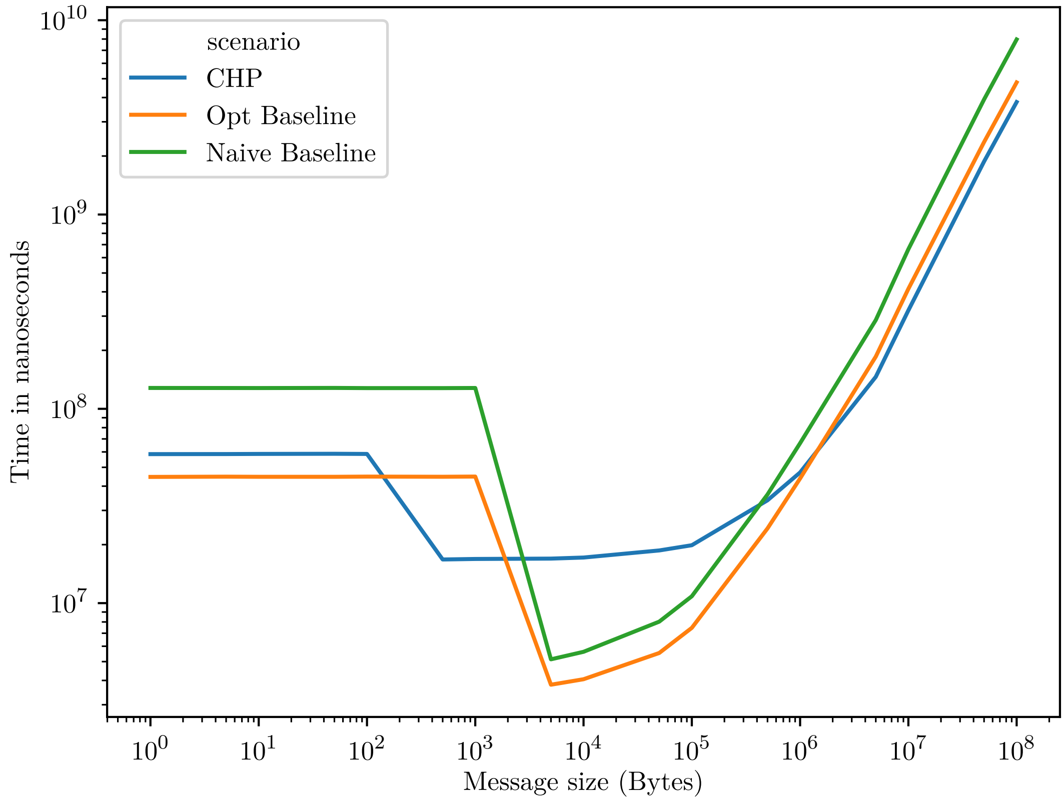


Figure 6.3: Total protocol time with security parameter 2048

6.2 Execution and Waiting Time

In this Section, the execution times of the individual nodes are examined and compared. Firstly, the execution time of CHP is compared to the Naive Baseline. Secondly, the execution time of CHP is compared to the Optimized Baseline. Finally, a discussion of the general differences between the three protocols is provided.

6.2.1 Comparison of CHP and the Naive Baseline

The comparison of CHP and the Naive and Optimized Baselines will be considered first. There is a statistically significant difference in execution time between CHP and the Naive Baseline for security parameters greater than or equal to 6144, as seen in table 6.5. By inspecting the execution time on a per-node basis, the execution times of both the Modifier and Recipient nodes also show a statistically significant difference when the security parameter is greater than or equal to 6144, as seen in table 6.7 and table 6.8. However, the Signer node in both protocols shows statistically significantly different execution times for the smaller security parameters and the largest i.e. 1024, 1536, 2048, and 8192, as seen in table 6.6. This indicates that the Signer node is what causes the difference in execution time between the two protocols.

Table 6.5: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Naive Baseline'

Test statistics	P-value	Security parameter (Bits)
8.075	0.004	6144
11.048	0.001	8192

Table 6.6: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Naive Baseline' (signer)

Test statistics	P-value	Security parameter (Bits)
11.048	0.001	1024
11.048	0.001	1536
9.292	0.002	2048
7.688	0.006	8192

Table 6.7: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Naive Baseline' (modifier)

Test statistics	P-value	Security parameter (Bits)
8.272	0.004	6144
11.510	0.001	8192

Table 6.8: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Naive Baseline' (recipient)

Test statistics	P-value	Security parameter (Bits)
8.673	0.003	6144
11.983	0.001	8192

By considering the largest security parameter for which there was a statistically significant difference in execution time between CHP and the Naive Baseline in fig. 6.4, and in more detail in table 6.9, the actual difference can be observed. The execution time for CHP is higher for messages with sizes less than or equal to 50MB, compared to the Naive Baseline. The longer execution time for CHP, with smaller message sizes, indicates that the overhead of the CHF is higher than the conventional SHA512 hash function used in the Naive Baseline. For the larger messages, sizes over 50MB, the execution time in CHP is about half that of the Naive Baseline. Since the Naive Baseline requires the message to be sent two more times than in CHP, the significant difference in execution time may be attributed to network I/O for the larger message sizes.

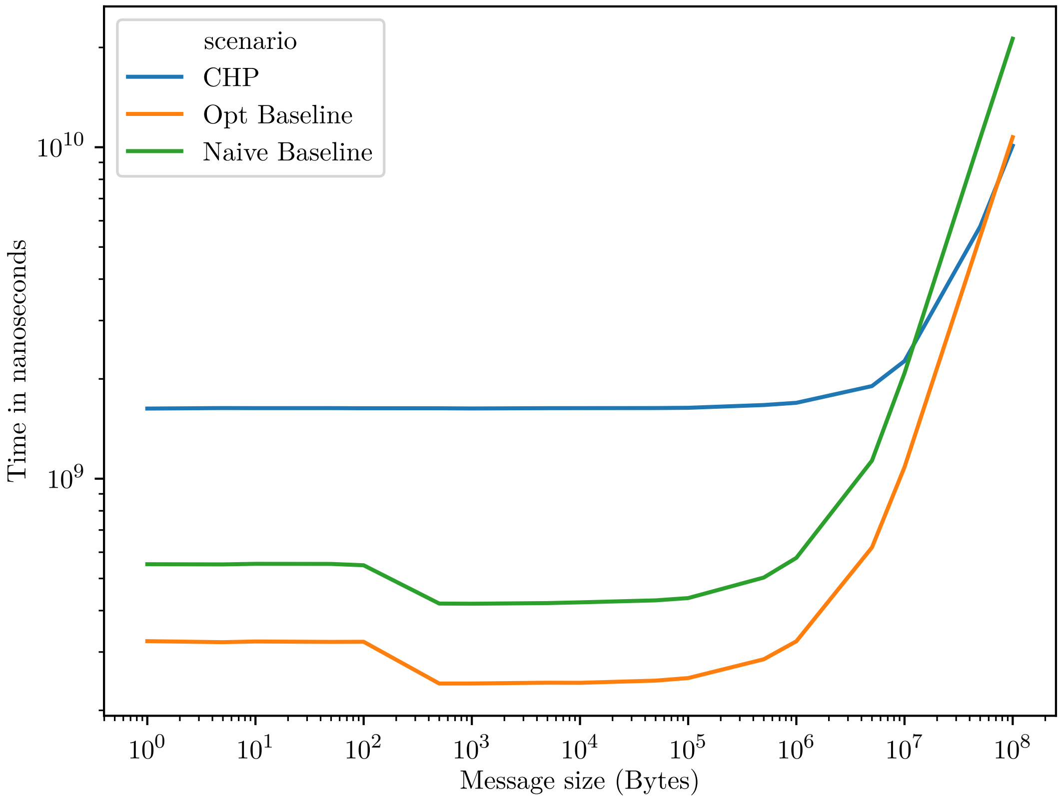


Figure 6.4: Total execution time with security parameter 8192 for all nodes

Table 6.9: Total execution time across all nodes with security parameter 8192

Message size (Bytes)	CHP median (95% CI)	Opt Baseline median (95% CI)	Naive Baseline median (95% CI)
1	1.628s (1.379s, 1.639s)	0.323s (0.291s, 0.328s)	0.552s (0.549s, 0.556s)
5	1.633s (1.375s, 1.646s)	0.321s (0.291s, 0.326s)	0.551s (0.549s, 0.556s)
10	1.632s (1.374s, 1.645s)	0.323s (0.291s, 0.328s)	0.553s (0.551s, 0.558s)
50	1.633s (1.373s, 1.644s)	0.322s (0.291s, 0.325s)	0.553s (0.549s, 0.557s)
100	1.631s (1.378s, 1.642s)	0.322s (0.293s, 0.327s)	0.548s (0.546s, 0.551s)
500	1.631s (1.371s, 1.643s)	0.241s (0.210s, 0.244s)	0.420s (0.363s, 0.427s)
1000	1.629s (1.371s, 1.646s)	0.241s (0.209s, 0.244s)	0.420s (0.363s, 0.425s)
5000	1.632s (1.368s, 1.642s)	0.242s (0.211s, 0.246s)	0.421s (0.362s, 0.430s)
10000	1.632s (1.377s, 1.649s)	0.242s (0.211s, 0.245s)	0.424s (0.359s, 0.428s)
50000	1.633s (1.375s, 1.645s)	0.246s (0.213s, 0.250s)	0.429s (0.366s, 0.436s)
100000	1.636s (1.375s, 1.648s)	0.250s (0.217s, 0.253s)	0.436s (0.374s, 0.443s)
500000	1.668s (1.408s, 1.677s)	0.285s (0.249s, 0.288s)	0.503s (0.455s, 0.510s)
1000000	1.694s (1.485s, 1.710s)	0.323s (0.279s, 0.329s)	0.577s (0.488s, 0.585s)
5000000	1.903s (1.604s, 1.920s)	0.620s (0.525s, 0.626s)	1.133s (0.955s, 1.144s)
10000000	2.264s (1.928s, 2.280s)	1.084s (0.952s, 1.093s)	2.082s (1.823s, 2.105s)
50000000	5.763s (5.153s, 5.776s)	5.370s (4.824s, 5.388s)	10.633s (9.600s, 10.676s)
100000000	10.109s (9.160s, 10.138s)	10.727s (9.769s, 10.754s)	21.257s (19.461s, 21.320s)

The measurements concerning waiting time aid in the further investigation of the impact of network I/O. Table 6.10 shows that there is a statistically significant difference in the total time spent waiting between CHP and the Naive Baseline for the same security parameters as for the total execution time. Furthermore, the Modifier and Recipient nodes only show statistically significant differences in the time waiting for the larger security parameters, as seen in table 6.12 and table 6.13. However, in table 6.11 there is a statistically significant difference in time waiting between the protocols for all evaluated security parameters.

Table 6.10: Statistically significant Kruskal-Wallis test results of median 'Waiting time' values between scenarios 'CHP' and 'Naive Baseline'

Test statistics	P-value	Security parameter (Bits)
5.567	0.018	6144
10.820	0.001	8192

Table 6.11: Statistically significant Kruskal-Wallis test results of median 'Waiting time' values between scenarios 'CHP' and 'Naive Baseline' (signer)

Test statistics	P-value	Security parameter (Bits)
11.983	0.001	1024
12.464	0.000	1536
12.955	0.000	2048
12.464	0.000	3072
13.710	0.000	4096
14.225	0.000	6144
15.553	0.000	8192

Table 6.12: Statistically significant Kruskal-Wallis test results of median 'Waiting time' values between scenarios 'CHP' and 'Naive Baseline' (modifier)

Test statistics	P-value	Security parameter (Bits)
7.880	0.005	8192

Table 6.13: Statistically significant Kruskal-Wallis test results of median 'Waiting time' values between scenarios 'CHP' and 'Naive Baseline' (recipient)

Test statistics	P-value	Security parameter (Bits)
8.272	0.004	6144
11.510	0.001	8192

In fig. 6.5 and table 6.14 the waiting time is less for the Naive Baseline for smaller messages with a size less than or equal to 5MB and more for larger messages with a size greater than 5MB and less than 100MB. For the largest evaluated message, the nodes in the Naive Baseline spend more than twice the amount of time waiting compared with CHP. The fact that the Naive Baseline spends relatively more time waiting than CHP suggests that the network I/O contributes to CHP having lower execution times for larger messages. For the smaller messages, the network overhead seems less substantial, suggesting that the overhead induced by the CHF used in CHP is greater than the network overhead.

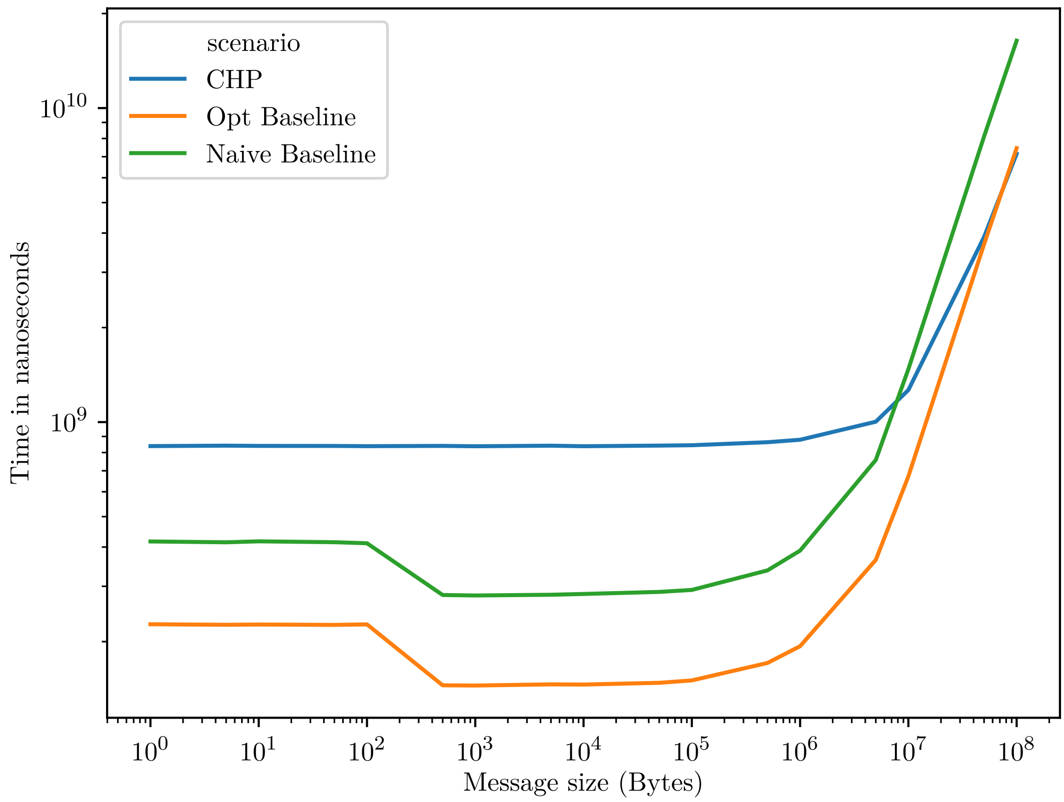


Figure 6.5: Total waiting time with security parameter 8192 for all nodes

Table 6.14: Total time waiting across all nodes with security parameter 8192

Message size (Bytes)	CHP median (95% CI)	Opt Baseline median (95% CI)	Naive Baseline median (95% CI)
1	0.839s (0.714s, 0.843s)	0.227s (0.208s, 0.230s)	0.417s (0.413s, 0.420s)
5	0.841s (0.710s, 0.848s)	0.226s (0.208s, 0.229s)	0.414s (0.411s, 0.418s)
10	0.840s (0.710s, 0.848s)	0.227s (0.208s, 0.230s)	0.417s (0.415s, 0.421s)
50	0.840s (0.709s, 0.845s)	0.226s (0.208s, 0.228s)	0.414s (0.412s, 0.419s)
100	0.838s (0.712s, 0.845s)	0.227s (0.209s, 0.229s)	0.411s (0.407s, 0.416s)
500	0.840s (0.709s, 0.847s)	0.145s (0.127s, 0.147s)	0.281s (0.243s, 0.286s)
1000	0.838s (0.708s, 0.848s)	0.145s (0.127s, 0.147s)	0.280s (0.243s, 0.284s)
5000	0.841s (0.706s, 0.844s)	0.146s (0.127s, 0.148s)	0.282s (0.243s, 0.288s)
10000	0.838s (0.712s, 0.849s)	0.146s (0.127s, 0.147s)	0.284s (0.241s, 0.287s)
50000	0.842s (0.710s, 0.848s)	0.148s (0.128s, 0.150s)	0.288s (0.245s, 0.292s)
100000	0.844s (0.710s, 0.850s)	0.150s (0.131s, 0.152s)	0.292s (0.251s, 0.297s)
500000	0.863s (0.729s, 0.869s)	0.171s (0.151s, 0.173s)	0.337s (0.304s, 0.342s)
1000000	0.879s (0.809s, 0.884s)	0.193s (0.167s, 0.197s)	0.390s (0.327s, 0.393s)
5000000	1.003s (0.849s, 1.014s)	0.364s (0.309s, 0.367s)	0.757s (0.637s, 0.765s)
10000000	1.265s (1.095s, 1.276s)	0.672s (0.605s, 0.679s)	1.474s (1.315s, 1.490s)
50000000	3.884s (3.572s, 3.893s)	3.684s (3.404s, 3.697s)	8.119s (7.490s, 8.142s)
100000000	7.153s (6.666s, 7.168s)	7.456s (6.982s, 7.469s)	16.392s (15.300s, 16.421s)

In fig. 6.6 and table 6.15 the execution time without waiting time can be seen more clearly. Similarly to the other time measurements per node, CHP has significantly larger times for the messages with size less than or equal to 10MB. However, for 50MB messages, CHP is faster than the Naive baseline and for 100MB messages, CHP is faster than both baselines in terms of active execution time. Based on the Kruskal-Wallis analysis of the active execution times between CHP and the Naive baseline, as seen in table 6.16, there is a statistically significant difference for security parameters greater than or equal to 3072. The difference in execution time on node level can be seen in table 6.17, table 6.18, and table 6.19. The Modifier and Recipient nodes show statistically significant differences between the active execution time for all security parameters evaluated except for the smallest (1024). Furthermore, the Signer node only shows a statistically significant difference in active execution time for security parameters greater than or equal to 4096. By comparing table 6.20 and table 6.21 it is evident that all nodes have more computational overhead in CHP for messages with sizes less than or equal to 1MB. Only for messages with a size 50MB or more does the Signer node have less active execution time in CHP compared with the Naive baseline. The Modifier and Recipient nodes have larger active execution times in CHP compared with the Naive baseline throughout.

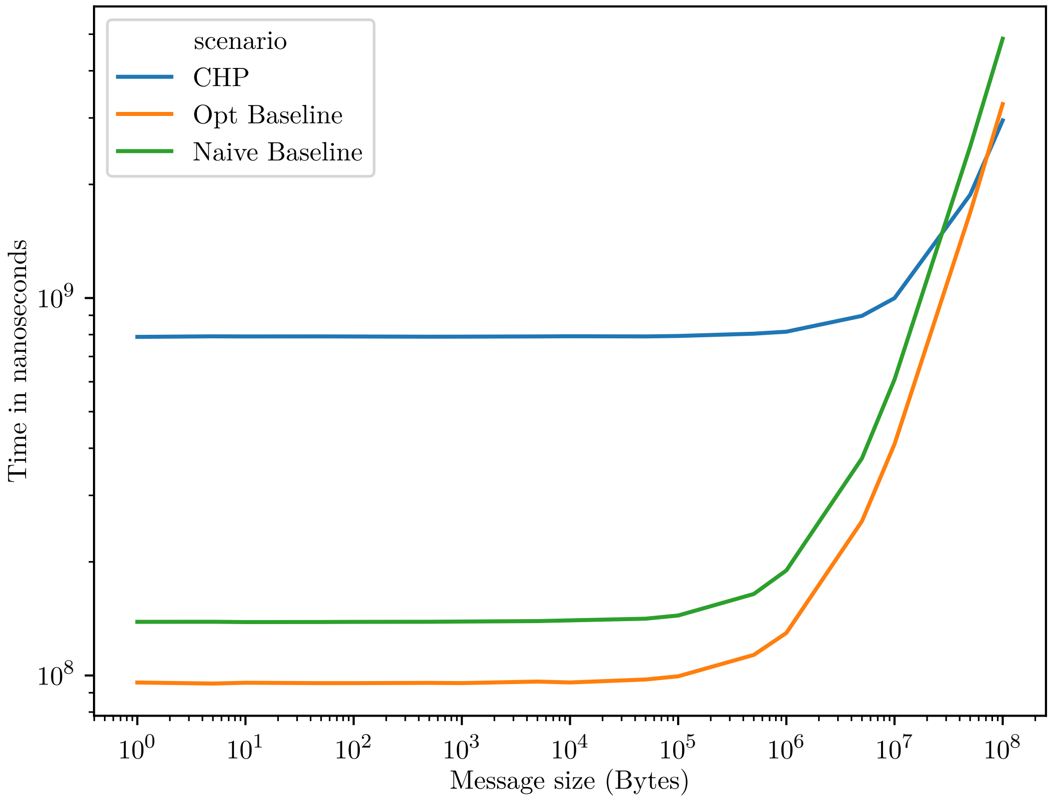


Figure 6.6: Total active execution time with security parameter 8192 for all nodes

Table 6.15: Total active execution time across all nodes with security parameter 8192

Message size (Bytes)	CHP median (95% CI)	Opt Baseline median (95% CI)	Naive Baseline median (95% CI)
1	0.789s (0.666s, 0.795s)	0.096s (0.083s, 0.097s)	0.139s (0.120s, 0.142s)
5	0.792s (0.665s, 0.797s)	0.095s (0.082s, 0.097s)	0.139s (0.120s, 0.141s)
10	0.791s (0.665s, 0.797s)	0.096s (0.083s, 0.098s)	0.139s (0.119s, 0.141s)
50	0.791s (0.665s, 0.797s)	0.095s (0.083s, 0.097s)	0.139s (0.120s, 0.141s)
100	0.791s (0.666s, 0.797s)	0.095s (0.084s, 0.097s)	0.139s (0.120s, 0.140s)
500	0.790s (0.664s, 0.796s)	0.096s (0.083s, 0.097s)	0.139s (0.120s, 0.141s)
1000	0.790s (0.664s, 0.798s)	0.095s (0.083s, 0.097s)	0.139s (0.120s, 0.141s)
5000	0.791s (0.664s, 0.795s)	0.096s (0.083s, 0.097s)	0.139s (0.119s, 0.142s)
10000	0.792s (0.665s, 0.798s)	0.096s (0.084s, 0.097s)	0.140s (0.118s, 0.142s)
50000	0.791s (0.665s, 0.797s)	0.098s (0.084s, 0.099s)	0.141s (0.121s, 0.144s)
100000	0.793s (0.665s, 0.797s)	0.100s (0.086s, 0.101s)	0.144s (0.123s, 0.147s)
500000	0.805s (0.676s, 0.809s)	0.113s (0.098s, 0.114s)	0.164s (0.140s, 0.168s)
1000000	0.814s (0.684s, 0.819s)	0.130s (0.111s, 0.131s)	0.190s (0.161s, 0.192s)
5000000	0.897s (0.756s, 0.905s)	0.256s (0.217s, 0.259s)	0.376s (0.316s, 0.379s)
10000000	0.999s (0.836s, 1.005s)	0.410s (0.348s, 0.414s)	0.608s (0.511s, 0.613s)
50000000	1.877s (1.583s, 1.884s)	1.683s (1.426s, 1.691s)	2.517s (2.118s, 2.528s)
100000000	2.955s (2.494s, 2.964s)	3.265s (2.786s, 3.277s)	4.865s (4.153s, 4.893s)

Table 6.16: Statistically significant Kruskal-Wallis test results of median 'Active execution time' values between scenarios 'CHP' and 'Naive Baseline'

Test statistics	P-value	Security parameter (Bits)
5.730	0.017	3072
7.880	0.005	4096
11.278	0.001	6144
14.749	0.000	8192

Table 6.17: Statistically significant Kruskal-Wallis test results of median 'Active execution time' values between scenarios 'CHP' and 'Naive Baseline' (signer)

Test statistics	P-value	Security parameter (Bits)
5.405	0.020	4096
7.688	0.006	6144
8.075	0.004	8192

Table 6.18: Statistically significant Kruskal-Wallis test results of median 'Active execution time' values between scenarios 'CHP' and 'Naive Baseline' (modifier)

Test statistics	P-value	Security parameter (Bits)
4.060	0.044	1536
4.200	0.040	2048
8.272	0.004	3072
8.471	0.004	4096
15.015	0.000	6144
15.283	0.000	8192

Table 6.19: Statistically significant Kruskal-Wallis test results of median 'Active execution time' values between scenarios 'CHP' and 'Naive Baseline' (recipient)

Test statistics	P-value	Security parameter (Bits)
3.922	0.048	1536
4.060	0.044	2048
8.075	0.004	3072
8.272	0.004	4096
11.510	0.001	6144
14.749	0.000	8192

Table 6.20: Total active execution time per node with security parameter 8192 for CHP

Role Message size (Bytes)	Modifier median (95% CI)	Recipient median (95% CI)	Signer median (95% CI)
1	0.442s (0.376s, 0.442s)	0.149s (0.127s, 0.150s)	0.195s (0.170s, 0.196s)
5	0.442s (0.375s, 0.443s)	0.149s (0.126s, 0.150s)	0.195s (0.165s, 0.197s)
10	0.443s (0.372s, 0.444s)	0.149s (0.127s, 0.150s)	0.195s (0.167s, 0.196s)
50	0.443s (0.375s, 0.444s)	0.149s (0.126s, 0.150s)	0.195s (0.168s, 0.197s)
100	0.443s (0.375s, 0.444s)	0.149s (0.127s, 0.150s)	0.194s (0.169s, 0.196s)
500	0.443s (0.371s, 0.443s)	0.149s (0.126s, 0.150s)	0.195s (0.168s, 0.198s)
1000	0.442s (0.375s, 0.443s)	0.149s (0.126s, 0.150s)	0.194s (0.167s, 0.197s)
5000	0.443s (0.374s, 0.443s)	0.149s (0.126s, 0.150s)	0.195s (0.165s, 0.197s)
10000	0.442s (0.374s, 0.443s)	0.150s (0.126s, 0.151s)	0.195s (0.169s, 0.198s)
50000	0.443s (0.376s, 0.444s)	0.149s (0.126s, 0.150s)	0.195s (0.166s, 0.197s)
100000	0.444s (0.372s, 0.444s)	0.150s (0.126s, 0.150s)	0.196s (0.167s, 0.199s)
500000	0.450s (0.382s, 0.451s)	0.151s (0.128s, 0.152s)	0.197s (0.168s, 0.200s)
1000000	0.457s (0.387s, 0.458s)	0.154s (0.130s, 0.154s)	0.200s (0.171s, 0.202s)
5000000	0.507s (0.428s, 0.507s)	0.170s (0.144s, 0.170s)	0.218s (0.189s, 0.219s)
10000000	0.567s (0.476s, 0.567s)	0.189s (0.159s, 0.190s)	0.240s (0.203s, 0.241s)
50000000	1.102s (0.930s, 1.102s)	0.351s (0.298s, 0.352s)	0.422s (0.359s, 0.423s)
100000000	1.754s (1.484s, 1.755s)	0.551s (0.466s, 0.552s)	0.645s (0.549s, 0.648s)

Table 6.21: Total active execution time per node with security parameter 8192 for Naive Baseline

Role Message size (Bytes)	Modifier median (95% CI)	Recipient median (95% CI)	Signer median (95% CI)
1	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.089s (0.078s, 0.092s)
5	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.089s (0.078s, 0.091s)
10	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.089s (0.078s, 0.091s)
50	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.089s (0.079s, 0.091s)
100	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.089s (0.078s, 0.091s)
500	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.090s (0.078s, 0.091s)
1000	0.048s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.089s (0.078s, 0.091s)
5000	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.090s (0.078s, 0.092s)
10000	0.048s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.090s (0.077s, 0.092s)
50000	0.048s (0.043s, 0.049s)	0.001s (0.001s, 0.001s)	0.091s (0.077s, 0.093s)
100000	0.049s (0.044s, 0.050s)	0.001s (0.001s, 0.002s)	0.092s (0.080s, 0.094s)
500000	0.056s (0.050s, 0.057s)	0.005s (0.004s, 0.005s)	0.101s (0.087s, 0.104s)
1000000	0.065s (0.057s, 0.066s)	0.009s (0.008s, 0.009s)	0.113s (0.096s, 0.115s)
5000000	0.129s (0.110s, 0.130s)	0.039s (0.033s, 0.039s)	0.204s (0.176s, 0.207s)
10000000	0.208s (0.179s, 0.208s)	0.076s (0.064s, 0.076s)	0.318s (0.269s, 0.322s)
50000000	0.855s (0.721s, 0.856s)	0.379s (0.324s, 0.379s)	1.267s (1.069s, 1.279s)
100000000	1.660s (1.410s, 1.662s)	0.756s (0.634s, 0.756s)	2.443s (2.063s, 2.453s)

The poor scaling with the message size of the Naive Baseline was expected by design, due to the additional messages sent. However, what was not expected, was that the messages had to be greater than or equal to 50MB in size for CHP to have less execution time than the Naive baseline. The computational overhead of the CHF appears to be greater than expected, considering the almost negligible overhead found by Ateniese et al. in [1] where the CHF used in the CHP implementation was defined. One possible reason why the overhead induced by the CHF in this thesis is greater may be due to the choice of CHF. Ateniese et al. introduced multiple CHFs in [1], which may induce different amounts of computational overhead. The CHF implemented in this thesis was chosen partially for the mathematical simplicity and was not the same CHF that Ateniese et al. evaluated in [1]. Since the two CHFs, and their application, are different, it is difficult to state anything general regarding the computational overhead of CHFs. Only the computational overhead induced by the CHF in this thesis is significant.

6.2.2 Comparison of CHP and the Optimized Baseline

Moving on to the comparison of CHP and the Optimized Baseline, the implications of the Kruskal-Wallis tests of CHP and the Optimized Baseline are quite similar to the ones between CHP and the Naive Baseline. As seen in table 6.22, there is a statistically significant difference in execution time for security parameters greater than or equal to 4096. Additionally, by considering the different nodes, as seen in tables 6.23 to 6.25, the Signer node is the main cause of the protocol level difference. Due to the difference in security parameter values for which there is a statistically significant difference between the two protocols (security parameter greater than or equal to 1536).

Table 6.22: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Opt Baseline'

Test statistics	P-value	Security parameter (Bits)
8.272	0.004	4096
11.510	0.001	6144
15.015	0.000	8192

Table 6.23: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Opt Baseline' (signer)

Test statistics	P-value	Security parameter (Bits)
4.060	0.044	1536
4.060	0.044	2048
6.065	0.014	3072
8.272	0.004	4096
11.278	0.001	6144
14.749	0.000	8192

Table 6.24: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Opt Baseline' (modifier)

Test statistics	P-value	Security parameter (Bits)
8.471	0.004	4096
11.510	0.001	6144
15.015	0.000	8192

Table 6.25: Statistically significant Kruskal-Wallis test results of median 'Execution time' values between scenarios 'CHP' and 'Opt Baseline' (recipient)

Test statistics	P-value	Security parameter (Bits)
8.272	0.004	4096
11.510	0.001	6144
15.015	0.000	8192

Looking at fig. 6.4 and table 6.9 it is clear that the Optimized Baseline outperforms CHP for all message sizes up until the largest message evaluated (100MB). The difference in waiting times for all nodes is statistically significant for the security parameters greater than or equal to 4096, as seen in table 6.26. By inspecting the actual waiting times in table 6.14 it appears that the main difference occurs when the messages have a size less than or equal to 10MB. This indicates that the main reason for the difference is due to the computational overhead of the CHF in CHP, similar to the comparison of CHP and the Naive baseline in section 6.2.1.

Table 6.26: Statistically significant Kruskal-Wallis test results of median 'Waiting time' values between scenarios 'CHP' and 'Opt Baseline'

Test statistics	P-value	Security parameter (Bits)
8.272	0.004	4096
11.278	0.001	6144
15.015	0.000	8192

Therefore, looking at the measurements for active execution time should provide more evidence that the computational overhead is the reason for CHP performing worse than the Optimized Baseline. Table 6.27 shows that there is a statistically significant difference in active execution time for CHP and the Optimized Baseline for all security parameters except 1024. Figure 6.6 shows that CHP has less active execution time across all nodes only for the largest message compared with the Optimized Baseline. Furthermore, by comparing table 6.20 and table 6.28 it is clear that all nodes in the Optimized Baseline have less active execution time than their CHP counterparts for all messages except the two largest. For the 50MB message, the Modifier node in CHP is the only node that requires more active execution than its Optimized Baseline counterpart. Based on the design of the Optimized Baseline, it was expected that it should perform better than CHP.

Table 6.27: Statistically significant Kruskal-Wallis test results of median 'Active execution time' values between scenarios 'CHP' and 'Opt Baseline'

Test statistics	P-value	Security parameter (Bits)
4.060	0.044	1536
4.200	0.040	2048
8.075	0.004	3072
8.272	0.004	4096
11.745	0.001	6144
15.015	0.000	8192

Table 6.28: Total active execution time per node with security parameter 8192 for Opt Baseline

Role Message size (Bytes)	Modifier median (95% CI)	Recipient median (95% CI)	Signer median (95% CI)
1	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.046s (0.041s, 0.047s)
5	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.047s (0.041s, 0.047s)
10	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.046s (0.041s, 0.048s)
50	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.046s (0.041s, 0.047s)
100	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.046s (0.042s, 0.048s)
500	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.046s (0.041s, 0.047s)
1000	0.048s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.046s (0.041s, 0.047s)
5000	0.047s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.047s (0.042s, 0.048s)
10000	0.048s (0.042s, 0.048s)	0.001s (0.001s, 0.001s)	0.046s (0.041s, 0.047s)
50000	0.048s (0.043s, 0.049s)	0.001s (0.001s, 0.001s)	0.047s (0.041s, 0.048s)
100000	0.049s (0.044s, 0.049s)	0.001s (0.001s, 0.002s)	0.047s (0.042s, 0.048s)
500000	0.056s (0.049s, 0.056s)	0.005s (0.004s, 0.005s)	0.051s (0.045s, 0.052s)
1000000	0.064s (0.056s, 0.065s)	0.009s (0.008s, 0.009s)	0.055s (0.048s, 0.056s)
5000000	0.129s (0.110s, 0.129s)	0.039s (0.034s, 0.039s)	0.087s (0.075s, 0.088s)
10000000	0.207s (0.179s, 0.208s)	0.076s (0.064s, 0.076s)	0.125s (0.107s, 0.126s)
50000000	0.854s (0.731s, 0.855s)	0.379s (0.320s, 0.379s)	0.447s (0.377s, 0.448s)
100000000	1.659s (1.430s, 1.661s)	0.756s (0.639s, 0.756s)	0.845s (0.744s, 0.847s)

The chameleon signature scheme in CHP definitively increases the computational overhead per node. However, there is only a significant difference between CHP and the Naive Baseline for security parameters greater than or equal to 6144, as discussed in section 6.2.1. Thus, CHP performs better than the Naive Baseline for security parameters in the range 2048 to 4096 while providing at least adequate levels of security based on the recommendations in [25]. The Optimized Baseline still performs better than CHP for most security parameters, only for security parameter 2048 is the difference not significant. Although the Optimized Baseline performs better, it is used here as an upper bound for the conventional digital signature scheme. Since it relies on the Modifier being in possession of all secret keys owned by the Signer nodes, it both creates a single point of failure as well as allows a malicious Modifier to forge signatures for all nodes in the network.

6.3 Communication Overhead

Based on table 6.29 and table 6.30 there is a statistically significant difference in the number of bytes transmitted in total between CHP and the two baselines for messages between 5KB and 100MB. The difference based on the security parameter is negligible, as such 8192 will be used for further analysis.

Table 6.29: Statistically significant Kruskal-Wallis test results of median 'Bytes transmitted' values between scenarios 'CHP' and 'Naive Baseline'

Test statistics	P-value	Message size (Bytes)
9.800	0.002	5000
9.800	0.002	10000
9.800	0.002	50000
9.800	0.002	100000
9.800	0.002	500000
9.800	0.002	1000000
9.800	0.002	5000000
9.800	0.002	10000000
9.800	0.002	50000000
9.800	0.002	100000000

Table 6.30: Statistically significant Kruskal-Wallis test results of median 'Bytes transmitted' values between scenarios 'CHP' and 'Opt Baseline'

Test statistics	P-value	Message size (Bytes)
5.588	0.018	1
5.588	0.018	5
5.588	0.018	10
5.588	0.018	50
5.588	0.018	100
5.588	0.018	500
5.588	0.018	1000
5.588	0.018	5000
5.588	0.018	10000
5.588	0.018	50000
5.588	0.018	100000
5.588	0.018	500000
5.588	0.018	1000000
5.588	0.018	5000000
5.588	0.018	10000000
5.588	0.018	50000000
5.588	0.018	100000000

In table 6.31 it is evident that the Optimized Baseline sends significantly fewer bytes over the network compared to CHP. In contrast, the Naive baseline sends significantly more bytes. These results are expected based on the design of the baseline scenarios as previously discussed in section 6.2. The fact that the Optimized Baseline sends fewer bytes compared with CHP indicates that the additional parameters sent with the chameleon hash induce a significant network overhead. However, as described in section 5.3, the Optimized Baseline symbolizes a realistically unattainable level of optimization for the authenticated data distribution with modification using conventional digital signatures. Taking that into account, combined with the fact that the difference in bytes transmitted between CHP and the Naive Baseline is higher than the difference between CHP and the Optimized Baseline, CHP does not have have major impact on the network overhead. On the contrary, considering the Naive Baseline, CHP reduces the network overhead.

Table 6.31: Bytes transmitted with security parameter 8192

Message size (Bytes)	CHP median (95% CI)	Opt Baseline median (95% CI)	Naive Baseline median (95% CI)
1	6244B	2082B	4164B
5	6252B	2090B	4180B
10	6262B	2100B	4200B
50	6342B	2180B	4360B
100	6442B	2280B	4560B
500	7242B	3080B	6160B
1000	8242B	4080B	8160B
5000	16242B	12080B	24160B
10000	26242B	22080B	44160B
50000	106242B	102080B	204160B
100000	206242B	202080B	404160B
500000	1006242B	1002080B	2004160B
1000000	2006242B	2002080B	4004160B
5000000	10006242B	10002080B	20004160B
10000000	20006242B	20002080B	40004160B
50000000	100006242B	100002080B	200004160B
100000000	200006242B	200002080B	400004160B

7 Discussion

This chapter contains a discussion pertaining to the method of obtaining the results, the societal and ethical implications of the thesis, and the occasional occurrence of a slight drop in measured time with increased message size. A summary of the findings from chapter 6 is also provided.

7.1 Results

CHP had a lower protocol time than the Naive Baseline at some point during all configurations evaluated, for some configurations CHP even had lower protocol time than the Optimized Baseline. Generally, when the message size increases CHP performs better and when the security parameter increases CHP performs worse compared to the other baselines. This seems to be due to the additional computational overhead from the CHF eventually being rendered negligible by the increasing network overhead in the Naive Baseline. Furthermore, the computational overhead imposed by the CHF is also less than the computational overhead of the conventional hash function for the largest messages evaluated. Overall, it appears that CHP scales worse with security parameter but better with message size compared with the baselines. This may be limited to this particular CHF and this particular implementation. However, if it is not, then CHP outperforms the conventional approaches to authenticated data distribution with modifications for the most reasonable security parameters (2048 - 4096).

The sudden drop in measured time when the message size increases, observed from multiple measurements (e.g. in fig. 6.1), is quite interesting. Especially since it is often only the baselines that experience the drop in measured time. By looking at the actual measured values it looks to consistent to be caused by the evaluation environment. One explanation of this behavior could be that the conventional hash function handles small input differently, and that the different behavior leads to different amounts of computational overhead. The digital signature algorithm always operates on a fixed size value, since the hash functions produce a fixed size output. Since the measurements of CHP also contain this drop in measured time occasionally, then the CHF should also behave differently on small input, which it does not as far as we know. A more reasonable explanation is that the drop observed is due to the nodes having fewer cache misses at the point of the drop i.e. the machines have warmed up sufficiently. To further support that explanation, the observed drop mostly occurs after only a few messages (for messages of sizes between 10B and 100B). The first iteration of each evaluation was discarded for this reason, however that may not have been sufficient if this drop is a result of initial cache misses.

7.2 Method

There are two potential issues with the design choices made, one for the implemented CHF and another one for the parameter selection for the CHF. Firstly, the decision to implement the less complex CHF proposed by Ateniese et al. in [1] made evaluating the computational overhead more difficult. For the current implementation, it is difficult to say why the computational overhead is as high as it is. Due to the lack of baseline CHF, the computational overhead may be attributed to implementation mistakes. Another potential issue, that also could result in additional computational overhead is the parameter selection for the CHF. Initially, the digital signature algorithm used was the Digital Signature Algorithm (DSA), but was later changed in favor of RSA due to the deprecation of DSA in [23]. The change of the digital signature scheme means that the underlying mathematical problem became different between CHF and the digital signature scheme, which could mean that the modulus size in bits does not necessarily have to be the same for the same level of security. More carefully tweaked parameters may have led to more reliable measurements concerning the execution times for the different protocols.

There is an issue concerning the measurements regarding the network wait time, in particular for the Recipient node. The network wait time depends on both execution and network time for the two other nodes since non-asynchronous sockets are used, and all scenarios end with the transmission from the Modifier node to the Recipient node. Thus, the timer should only start when the first packet arrives from the Modifier node as this improves the reliability of this measurement. However, by using this measurement to deduce the time a node is blocked it can provide valid insight into the node's performance. Since the total execution time is measured, the network I/O measurement can be deducted from it to provide a metric for active execution time. In doing so, the measurements for active execution time are a more reliable metric than network wait time. Furthermore, the number of bytes transmitted metric has higher validity than the network wait time metric when investigating the network overhead since it only counts the number of application-level bytes sent.

The replicability of this study is as high as a real-world environment evaluation can be. The source code for the implementations evaluated is available in a Gitlab repository [12], the evaluation environment is described in sufficient detail in section 5.6 and the data analysis performed is contained in the Python script. Furthermore, in appendix C the specific dates of when the evaluation was run are provided, such that someone attempting to reproduce the results in this thesis can orchestrate their evaluation runs at a similar time of day.

The inclusion of the work on redactable blockchains by Niya, Willems, and Stiller [21] might be questionable since it is published on *arXiv*. However, Niya, Willems, and Stiller did get a case study utilizing their work [21] published to *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*[20]. Although the acceptance of the case study does not mean that the underlying work underwent peer review, it does increase the credibility of the work published on arXiv.

7.3 The work in a wider context

The use of chameleon signature schemes in protocols offering both authenticity and delegated modifications is shown, in this thesis, to be a viable alternative to conventional signature schemes. On a societal level, this can potentially reduce the energy consumption of e.g. edge computing networks. Provided that the computational overhead per node requires less energy than the network would have required to transmit the additional bytes as in the Naive Baseline.

The energy savings would become even greater if edge nodes sent high-resolution photos requiring modifications by the edge server before distribution, since the difference between CHP and the Naive baseline in bytes transmitted scaled with message size. The additional computational overhead induced by the CHF could be managed by selecting the smallest security parameter that provides adequate security. The Optimized Baseline, introduced in this thesis, could also be used to achieve even greater energy savings in the network. However, giving all Modifier nodes access to the signing keys of all other nodes in the network poses several security and ethical problems. The fact that the signing key no longer only exists on the owner system will reduce the authenticity of any future signature made with that key. Since the Optimized Baseline requires a completely benign Modifier system, that can not be influenced externally or breached, it is a valid approach to reduce the energy consumption of a network.disc

8 Conclusion

This thesis aimed to implement a communication protocol using a chameleon signature scheme and to evaluate the performance of that implementation in a real-world environment. A CHF was implemented and used in the protocol implementation. The performance evaluation included the chameleon signature protocol CHP and two created baselines. The computational overhead in CHP was significantly higher than the baselines for the larger security parameters. However, for more reasonable security parameters e.g. 2048 the computational overhead of CHP was low enough such that the protocol completed faster than the Naive Baseline. The network overhead of CHP was significantly lower than the Naive Baseline and closer to the Optimized Baseline. Thus, to answer the main research question, the chameleon signature scheme makes authenticated data distribution with modifications more efficient by reducing the network overhead more than it increases the computational overhead per node (using an adequate level of security).

Since only a single CHF was implemented and evaluated in this thesis, future studies could use the implementation from this thesis as a baseline to investigate the performance impact of different CHFs. Additionally, the CHF implementation from this thesis can be used in other protocol implementations such as MoMEP.

Bibliography

[1] Giuseppe Ateniese, Bernardo Magri, Daniele Venturi, and Ewerton Andrade. "Redactable Blockchain – or – Rewriting History in Bitcoin and Friends". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pp. 111–126. DOI: 10.1109/EuroSP.2017.37.

[2] Giuseppe Ateniese and Breno de Medeiros. "Identity-Based Chameleon Hash and Applications". In: *Financial Cryptography*. Ed. by Ari Juels. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 164–180. ISBN: 978-3-540-27809-2. DOI: 10.1007/978-3-540-27809-2_19.

[3] Giuseppe Ateniese and Breno de Medeiros. "On the Key Exposure Problem in Chameleon Hashes". In: *Security in Communication Networks*. Ed. by Carlo Blundo and Stelvio Cimato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 165–179. ISBN: 978-3-540-30598-9. DOI: 10.1007/978-3-540-30598-9_12.

[4] Haixia Chen, Xinyi Huang, Wei Wu, and Yi Mu. "Privacy-Aware Image Authentication from Cryptographic Primitives". In: *The Computer Journal* 64.8 (Nov. 2020), pp. 1178–1192. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxaa127. eprint: https://academic.oup.com/comjnl/article-pdf/64/8/1178/39904402/bxaa127.pdf. URL: https://doi.org/10.1093/comjnl/bxaa127.

[5] Xiaohu Chen, Anjia Yang, Yao Tong, Jian Weng, Jiasi Weng, and Tao Li. "A Multisignature-Based Secure and OBU-Friendly Emergency Reporting Scheme in VANET". In: *IEEE Internet of Things Journal* 9.22 (2022), pp. 23130–23141. DOI: 10.1109/JIOT.2022.3184991.

[6] David Derler, Kai Samelin, and Daniel Slamanig. "Bringing Order to Chaos: The Case of Collision-Resistant Chameleon-Hashes". In: *Public-Key Cryptography – PKC 2020*. Ed. by Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas. Cham: Springer International Publishing, 2020, pp. 462–492. ISBN: 978-3-030-45374-9. DOI: 10.1007/978-3-030-45374-9_16.

[7] Daniel Díaz-Sánchez, Andrés Marín-Lopez, Florina Almenárez Mendoza, and Patricia Arias Cabarcos. "DNS/DANE Collision-Based Distributed and Dynamic Authentication for Microservices in IoT †". In: *Sensors* 19.15 (2019). ISSN: 1424-8220. DOI: 10.3390/s19153292. URL: https://www.mdpi.com/1424-8220/19/15/3292.

[8] Reyhane Falanji, Mikael Asplund, and Niklas Carlsson. *Provable Entity Accountability for Protocols with Modifiable Signed Messages*. `https://fcs-workshop.github.io/fcs2024/papers/FCS_Falanji_Asplund_Carlsson.pdf`, affiliated with CSF 2024, July 8, 2024, Enschede, The Netherlands.

[9] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous java performance evaluation”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 57–76. ISBN: 9781595937865. DOI: `10.1145/1297027.1297033`. URL: `https://doi.org/10.1145/1297027.1297033`.

[10] GitHub, inc. *GitHub*. Available: `https://github.com`. [Accessed: 2025-02-13].

[11] Torsten Hoefler and Roberto Belli. “Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: `10.1145/2807591.2807644`. URL: `https://doi-org.e.bibl.liu.se/10.1145/2807591.2807644`.

[12] John Åkerman. *thesis-impl*. Available: `https://gitlab.liu.se/johak760/thesis-impl`.

[13] Mojtaba Khalili, Mohammad Dakhilalian, and Willy Susilo. “Efficient chameleon hash functions in the enhanced collision resistant model”. In: *Information Sciences* 510 (2020), pp. 155–164. ISSN: 0020-0255. DOI: `https://doi.org/10.1016/j.ins.2019.09.001`. URL: `https://www.sciencedirect.com/science/article/pii/S002002551930831X`.

[14] Hugo Krawczyk and Tal Rabin. *Chameleon Hashing and Signatures*. Cryptology ePrint Archive, Paper 1998/010. 1998. URL: `https://eprint.iacr.org/1998/010`.

[15] William H. Kruskal and W. Allen Wallis and. “Use of Ranks in One-Criterion Variance Analysis”. In: *Journal of the American Statistical Association* 47.260 (1952), pp. 583–621. DOI: `10.1080/01621459.1952.10483441`. eprint: `https://www.tandfonline.com/doi/pdf/10.1080/01621459.1952.10483441`. URL: `https://www.tandfonline.com/doi/abs/10.1080/01621459.1952.10483441`.

[16] Yehuda Lindell and Jonathan Katz. *introduction to modern cryptography second edition*. Chapman and Hall/CRC, 2007. DOI: `10.1201/b17668`.

[17] Youshui Lu, Jingning Zhang, Yong Qi, Saiyu Qi, Yuanqing Zheng, Yuhao Liu, Hongyu Song, and Wei Wei. “Accelerating at the Edge: A Storage-Elastic Blockchain for Latency-Sensitive Vehicular Edge Computing”. In: *IEEE Transactions on Intelligent Transportation Systems* 23.8 (2022), pp. 11862–11876. DOI: `10.1109/TITS.2021.3108052`.

[18] Nicholas D. Matsakis and Felix S. Klock. “The rust language”. In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT ’14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 103–104. ISBN: 9781450332170. DOI: `10.1145/2663171.2663188`. URL: `https://doi.org/10.1145/2663171.2663188`.

[19] Jamie Nguyen. *OpenSSL Certificate Authority*. `https://jamielinux.com/docs/openssl-certificate-authority/index.html`. [Accessed: 2025-03-01].

[20] Sina Rafati Niya, Julius Willems, and Burkhard Stiller. “A Case Study of a Blockchain-GDPR Adaptation”. In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022, pp. 1–3. DOI: `10.1109/ICBC54727.2022.9805553`.

[21] Sina Rafati Niya, Julius Willems, and Burkhard Stiller. *On-Chain IoT Data Modification in Blockchains*. 2021. arXiv: 2103.10756 [cs.CR]. URL: https://arxiv.org/abs/2103.10756.

[22] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi-org.e.bibl.liu.se/10.1145/359340.359342.

[23] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. Tech. rep. Federal Information Processing Standards Publications (FIPS) 186-5. Washington, D.C.: U.S. Department of Commerce, 2023. DOI: 10.6028/NIST.FIPS.186-5.

[24] Alex Gaynor Steven Fackler. *rust-openssl*. https://crates.io/crates/openssl. [Accessed: 2025-05-06].

[25] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. “Measuring small subgroup attacks against Diffie-Hellman”. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. DOI: 10.14722/ndss.2017.23171. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/measuring-small-subgroup-attacks-against-diffie-hellman/.

[26] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[27] Junpeng Xu, Haixia Chen, Xu Yang, Wei Wu, and Yongcheng Song. “Verifiable image revision from chameleon hashes”. In: *Cybersecurity* 4 (2021), pp. 1–13. DOI: https://doi.org/10.1186/s42400-021-00097-3.

A Virtual Machines

Configuration


```
{
    "name": "chf-vnet",
    "id":
    "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourceGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/virtualNetworks/chf-vnet",
    "etag": "W/\"9578e06a-dadd-4658-9415-07498db2c803\"",
    "type": "Microsoft.Network/virtualNetworks",
    "location": "westeurope",
    "tags": {},
    "properties": {
        "provisioningState": "Succeeded",
        "resourceGuid": "4cef0958-16e9-4ea4-89ba-89702a1f1084",
        "addressSpace": {
            "addressPrefixes": [
                "10.0.0.0/24"
            ]
        },
        "encryption": {
            "enabled": false,
            "enforcement": "AllowUnencrypted"
        },
        "subnets": [
            {
                "name": "default",
                "id":
                "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourceGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/virtualNetworks/chf-vnet/subnets/default",
                "etag": "W/\"9578e06a-dadd-4658-9415-07498db2c803\"",
                "properties": {
```

```
                        "provisioningState": "Succeeded",
                        "addressPrefixes": [
                            "10.0.0.0/24"
                        ],
                        "ipConfigurations": [
                            {
                                "id":
                                "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc
/resourceGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/networkInterfaces/CH
F-MODIFIER24_Z1/ipConfigurations/IPCONFIG1"
                            },
                            {
                                "id":
                                "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc
/resourceGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/networkInterfaces/CH
F-RECIPIENT75_Z1/ipConfigurations/IPCONFIG1"
                            },
                            {
                                "id":
                                "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc
/resourceGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/networkInterfaces/CH
F-SIGNER526_Z1/ipConfigurations/IPCONFIG1"
                            }
                        ],
                        "delegations": [],
                        "privateEndpointNetworkPolicies": "Disabled",
                        "privateLinkServiceNetworkPolicies": "Enabled"
                    },
                    "type": "Microsoft.Network/virtualNetworks/subnets"
                }
            ],
            "virtualNetworkPeerings": [],
            "enableDdosProtection": false
        },
        "apiVersion": "2024-01-01"
}
```

67
Configuration

```
{
    "name": "chf-signer",
    "id":
    "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourceGroups/ASTECC-R
TSLAB-1/providers/Microsoft.Compute/virtualMachines/chf-signer",
    "type": "Microsoft.Compute/virtualMachines",
    "location": "westeurope",
    "properties": {
        "hardwareProfile": {
            "vmSize": "Standard_D2s_v3"
        },
        "provisioningState": "Succeeded",
        "vmId": "47d79ad5-2d8d-43a1-8bbe-96f6bca128ff",
        "additionalCapabilities": {
```

```
                "hibernationEnabled": false
            },
            "storageProfile": {
                "imageReference": {
                    "publisher": "canonical",
                    "offer": "ubuntu-24_04-lts",
                    "sku": "server",
                    "version": "latest",
                    "exactVersion": "24.04.202504080"
                },
                "osDisk": {
                    "osType": "Linux",
                    "name":
                    "chf-signer_OsDisk_1_fe311dff115e49fe864261df72862ff3",
                    "createOption": "FromImage",
                    "caching": "ReadWrite",
                    "managedDisk": {
                        "storageAccountType": "Premium_LRS",
                        "id":
                        "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourc
eGroups/ASTECC-RTSLAB-1/providers/Microsoft.Compute/disks/chf-signer_OsDisk_1_fe
311dff115e49fe864261df72862ff3"
                    },
                    "deleteOption": "Delete",
                    "diskSizeGB": 30
                },
                "dataDisks": []
            },
            "osProfile": {
                "computerName": "chf-signer",
                "adminUsername": "azureuser",
                "linuxConfiguration": {
                    "disablePasswordAuthentication": true,
                    "ssh": {
                        "publicKeys": [
                            {
                                "path":
                                "/home/azureuser/.ssh/authorized_keys",
                                "keyData": "ssh-rsa
                                AAAAB3NzaC1yc2EAAAADAQABAAABgQCy4smq96MjzSGLcdNZZsiq
nJEohcrL38Rc0QIO/VEuI3H1YJXYY4idjZHEXssBZjP36rzQb8FpI+UFBfyPq56Heboi+t3mJ68IckQa
G2/vDG68PUvhVV2rG3qqtF79BhLE2bD8hkWMsbQj2w18UpdRj9h7lZnkEZuiL98LGmuxN1PAOZ+EwGG0
jxmk1ypVN19OXzyDSitDEn+rAA9bgXtDBgwwR6ARdvF62zxTXHS2fdwfrdcoFCX27v02CCdkVx2rhZtO
6FpKb4SMEmTdZigKO6kxrFdPgsy3qEgV6akxDIqvV00GA/nsLsAnY8ite4ZNFZaC2+DyGXApK+3ORDwH
V7v68dPsaYmz/4YSstudqYdehDhghhKwFSzeG3XO4f0nHKezoSSjlHs5pNjXZRDq/J62ekq39hTwh1Zs
pvRcqyQ3nNzTDYcls+0MadBT6UalnAv6cT6SsCuXIwA28PNTn9sPXMlljjL0I2fi2gcNSr5wxznlIiCD
Tx6bm09ksBU=
                                generated-by-azure"
                            }
                        ]
                    },
                    "provisionVMAgent": true,
                    "patchSettings": {
```

```
                    "patchMode": "ImageDefault",
                    "assessmentMode": "ImageDefault"
                }
            },
            "secrets": [],
            "allowExtensionOperations": true,
            "requireGuestProvisionSignal": true
        },
        "securityProfile": {
            "uefiSettings": {
                "secureBootEnabled": true,
                "vTpmEnabled": true
            },
            "securityType": "TrustedLaunch"
        },
        "networkProfile": {
            "networkInterfaces": [
                {
                    "id":
                    "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourceGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/networkInterfaces/chf-signer795_z1",
                    "properties": {
                        "deleteOption": "Delete"
                    }
                }
            ]
        },
        "diagnosticsProfile": {
            "bootDiagnostics": {
                "enabled": true
            }
        }
    },
    "zones": [
        "1"
    ],
    "apiVersion": "2021-03-01"
}
```


Configuration


```
{
    "name": "chf-modifier",
    "id":
    "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourceGroups/ASTECC-RTSLAB-1/providers/Microsoft.Compute/virtualMachines/chf-modifier",
    "type": "Microsoft.Compute/virtualMachines",
    "location": "westeurope",
    "properties": {
        "hardwareProfile": {
            "vmSize": "Standard_D2s_v3"
        },
```

```
        "provisioningState": "Succeeded",
        "vmId": "e4d9efeb-f88a-4622-9c3a-88bee12e41de",
        "additionalCapabilities": {
            "hibernationEnabled": false
        },
        "storageProfile": {
            "imageReference": {
                "publisher": "canonical",
                "offer": "ubuntu-24_04-lts",
                "sku": "server",
                "version": "latest",
                "exactVersion": "24.04.202504080"
            },
            "osDisk": {
                "osType": "Linux",
                "name":
                "chf-modifier_OsDisk_1_0e59f75a27ab4baea19a66d1ac34b739",
                "createOption": "FromImage",
                "caching": "ReadWrite",
                "managedDisk": {
                    "storageAccountType": "Premium_LRS",
                    "id":
                    "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourc
eGroups/ASTECC-RTSLAB-1/providers/Microsoft.Compute/disks/chf-modifier_OsDisk_1_
0e59f75a27ab4baea19a66d1ac34b739"
                },
                "deleteOption": "Delete",
                "diskSizeGB": 30
            },
            "dataDisks": []
        },
        "osProfile": {
            "computerName": "chf-modifier",
            "adminUsername": "azureuser",
            "linuxConfiguration": {
                "disablePasswordAuthentication": true,
                "ssh": {
                    "publicKeys": [
                        {
                            "path":
                            "/home/azureuser/.ssh/authorized_keys",
                            "keyData": "ssh-rsa
                            AAAAB3NzaC1yc2EAAAADAQABAAABgQC2raE23eH4Pc3YWXH+l6x3
yMfFrS7pgiHvmgrQMAOi45CUnG4/9evrLUBBIu9B37eF+zut+nvBe78JLnKVlFIZlripcABoV6RIufxE
6J3YJdtuABuc2tWEwJtR4iKG5IMYho0BQzl1OGI0Njpq71XXj56lLvsOCGKbdomhWyQpH3cPLlXbjKDq
ecXvOL1lZywNZSPeHbIK9X16MomA07Ro0gupkDewN7olIVPd+rulXgX5QeGWLmyLYAVCEVLhGSbzMG6A
I5iw5j+qpw9H9KuYut1SgIEzvlqzQbNokFgt2BMSgKBUvnSD/yOWsZOPy+XCcsl9qVqRNB8Qr3oEOUQ6
YQFo1Jpl5H0+5CzD3KxkcSqFtZ3WhgReIO0SFL+0FP8Gc3uP8FfHlEUh6lrrBtWmBJuwCs6TPwygU8lz
UhpzjflVHGkFvylulYQMko+hgwZatsFXeW1cZEpUJqUtTtHVcEUwBvkHgzkV7W/nFGLLeUGrmXN/zFDV
gfwU9nO0Z0E=
                            generated-by-azure"
                        }
                    ]
```

```
                    },
                    "provisionVMAgent": true,
                    "patchSettings": {
                        "patchMode": "ImageDefault",
                        "assessmentMode": "ImageDefault"
                    }
                },
                "secrets": [],
                "allowExtensionOperations": true,
                "requireGuestProvisionSignal": true
            },
            "securityProfile": {
                "uefiSettings": {
                    "secureBootEnabled": true,
                    "vTpmEnabled": true
                },
                "securityType": "TrustedLaunch"
            },
            "networkProfile": {
                "networkInterfaces": [
                    {
                        "id":
                        "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourc
eGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/networkInterfaces/chf-modifi
er390_z1",
                        "properties": {
                            "deleteOption": "Delete"
                        }
                    }
                ]
            },
            "diagnosticsProfile": {
                "bootDiagnostics": {
                    "enabled": true
                }
            }
        },
        "zones": [
            "1"
        ],
        "apiVersion": "2021-03-01"
}
```


107
Configuration


```
{
    "name": "chf-recipient",
    "id":
    "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourceGroups/ASTECC-R
TSLAB-1/providers/Microsoft.Compute/virtualMachines/chf-recipient",
    "type": "Microsoft.Compute/virtualMachines",
    "location": "westeurope",
    "properties": {
```

```
            "hardwareProfile": {
                "vmSize": "Standard_D2s_v3"
            },
            "provisioningState": "Succeeded",
            "vmId": "41525cc7-4571-4a01-b356-ba83d25bde6a",
            "additionalCapabilities": {
                "hibernationEnabled": false
            },
            "storageProfile": {
                "imageReference": {
                    "publisher": "canonical",
                    "offer": "ubuntu-24_04-lts",
                    "sku": "server",
                    "version": "latest",
                    "exactVersion": "24.04.202504080"
                },
                "osDisk": {
                    "osType": "Linux",
                    "name":
                    "chf-recipient_OsDisk_1_846db42f2ee84856a5c5aba1fe00e51c",
                    "createOption": "FromImage",
                    "caching": "ReadWrite",
                    "managedDisk": {
                        "storageAccountType": "Premium_LRS",
                        "id":
                        "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourc
eGroups/ASTECC-RTSLAB-1/providers/Microsoft.Compute/disks/chf-recipient_OsDisk_1
_846db42f2ee84856a5c5aba1fe00e51c"
                    },
                    "deleteOption": "Delete",
                    "diskSizeGB": 30
                },
                "dataDisks": []
            },
            "osProfile": {
                "computerName": "chf-recipient",
                "adminUsername": "azureuser",
                "linuxConfiguration": {
                    "disablePasswordAuthentication": true,
                    "ssh": {
                        "publicKeys": [
                            {
                                "path":
                                "/home/azureuser/.ssh/authorized_keys",
                                "keyData": "ssh-rsa
                                AAAAB3NzaC1yc2EAAAADAQABAAABgQC4PsKe00O+5M35SqaEsiv3
KzxwvhhDC/gfvg8ZIvFty4eZGTXqR6YhOnMBQ5HNIL/MnPw0JR5HwL2jmVUrJdzgKgUfikZFGbphPdVZ
CC8Saxq+JM5CHWiJKuXom5MZOHvqF2rplZ02MT7m3gRfRY8D/1RypTAwuJn406dqcEeAb+QinlU6W4Vi
jYVzI6eA2aG5mlkB054nhBX7MLv3g0g1PE0FVtGdUxRSXAp29EV8U4bb0Pf+V01OnG9Iqw3Nwer765cP
pTKQfNdgsOQ7Gq+pDFRDqFijjlEG92m+GOdbLzzm40ciVidmBelOFa9NUeAq6XHHanXFNLFzhVp8elcu
6xTA79/W+F/jqdUAR7Hu6LFsy4V38ue4Yvkp1JxVyecki4/o92gyRUZgQFPftRO2WWGSX3vzjlvhOtCr
eGP6TCdkWxB7qf7LILoUbcJhtZ6G5Zn0c8aS18rtf2VL7oLZkz0PP+waB0cvc/xTlLxzikPEN8kpz8cb
qCdojbaOZX0=
```

```
                              generated-by-azure"
                          }
                      ]
                  },
                  "provisionVMAgent": true,
                  "patchSettings": {
                      "patchMode": "ImageDefault",
                      "assessmentMode": "ImageDefault"
                  }
              },
              "secrets": [],
              "allowExtensionOperations": true,
              "requireGuestProvisionSignal": true
          },
          "securityProfile": {
              "uefiSettings": {
                  "secureBootEnabled": true,
                  "vTpmEnabled": true
              },
              "securityType": "TrustedLaunch"
          },
          "networkProfile": {
              "networkInterfaces": [
                  {
                      "id":
                      "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourc
eGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/networkInterfaces/chf-recipi
ent75_z1",
                      "properties": {
                          "deleteOption": "Delete"
                      }
                  }
              ]
          },
          "diagnosticsProfile": {
              "bootDiagnostics": {
                  "enabled": true
              }
          }
      },
      "zones": [
          "1"
      ],
      "apiVersion": "2021-03-01"
}
```

107
Configuration

```
{
    "name": "chf-evaluator",
    "id":
    "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourceGroups/ASTECC-R
TSLAB-1/providers/Microsoft.Compute/virtualMachines/chf-evaluator",
```

```
        "type": "Microsoft.Compute/virtualMachines",
        "location": "westeurope",
        "properties": {
            "hardwareProfile": {
                "vmSize": "Standard_D2s_v3"
            },
            "provisioningState": "Succeeded",
            "vmId": "fe69719d-bbb2-4e92-966f-feb3c35f7eff",
            "additionalCapabilities": {
                "hibernationEnabled": false
            },
            "storageProfile": {
                "imageReference": {
                    "publisher": "canonical",
                    "offer": "ubuntu-24_04-lts",
                    "sku": "server",
                    "version": "latest",
                    "exactVersion": "24.04.202504080"
                },
                "osDisk": {
                    "osType": "Linux",
                    "name":
                    "chf-evaluator_disk1_66ba973ac299424ab043f334c9773e93",
                    "createOption": "FromImage",
                    "caching": "ReadWrite",
                    "managedDisk": {
                        "storageAccountType": "Premium_LRS",
                        "id":
                        "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourc
eGroups/ASTECC-RTSLAB-1/providers/Microsoft.Compute/disks/chf-evaluator_disk1_66
ba973ac299424ab043f334c9773e93"
                    },
                    "deleteOption": "Delete",
                    "diskSizeGB": 30
                },
                "dataDisks": []
            },
            "osProfile": {
                "computerName": "chf-evaluator",
                "adminUsername": "azureuser",
                "linuxConfiguration": {
                    "disablePasswordAuthentication": true,
                    "ssh": {
                        "publicKeys": [
                            {
                                "path":
                                "/home/azureuser/.ssh/authorized_keys",
                                "keyData": "ssh-rsa
                                AAAAB3NzaC1yc2EAAAADAQABAAABgQCcbb83js+9m04U/BdfQHZ9
I4HWW4vDbAGrqV0TodwwlAS2PLfw7gAC37QpRiDBteNTU+UiKcqHi36AvYN0g73inDFP66/IFh5UB85f
8RLFguJCszuTHQe7vDOA2+rQP3+bKLbGtFxXwZ2GTHCM6RGSeX9fRX+D0Leux8UK1Sb51bmLEvgcgZQZ
ilo6Oy8qA+hBSPeEL2v7GtHMUQLFMxoAYmZoYUp6NQ3pqQZhfamBqNwB98zjE5CTWHhfJ3UP0CNq1EtU
SWXdm/K6MFpSQ0BGlhsnds8HCGUqDFo5tWmHJkcdefXYaKuXgWx0129K7iSLSECRXL5mIR6fKpnhwqiX
```

```
q4buJl2Twt/f7ZhlOYrcZY/QMk1l7nc3tVuaZ91dXQBdBEPUQcJvq1E3sGtPeVsZiEgt0aRdUWhbOQUI
+KwKuoQLTiPpJ879Dfx7BkoPnq7Edtn7fmMj2bth1ZoPGbac8gnjxFQeJc2HHFTMM3s47kA8dIfSKWQr
sObK9U8qupk=
                              generated-by-azure"
                          }
                      ]
                  },
                  "provisionVMAgent": true,
                  "patchSettings": {
                      "patchMode": "ImageDefault",
                      "assessmentMode": "ImageDefault"
                  }
              },
              "secrets": [],
              "allowExtensionOperations": true,
              "requireGuestProvisionSignal": true
          },
          "securityProfile": {
              "uefiSettings": {
                  "secureBootEnabled": true,
                  "vTpmEnabled": true
              },
              "securityType": "TrustedLaunch"
          },
          "networkProfile": {
              "networkInterfaces": [
                  {
                      "id":
                      "/subscriptions/10f6ad59-3737-45b0-9c43-0b948155c6cc/resourc
eGroups/ASTECC-RTSLAB-1/providers/Microsoft.Network/networkInterfaces/chf-evalua
tor141_z1",
                      "properties": {
                          "deleteOption": "Delete"
                      }
                  }
              ]
          },
          "diagnosticsProfile": {
              "bootDiagnostics": {
                  "enabled": true
              }
          }
      },
      "zones": [
          "1"
      ],
      "apiVersion": "2021-03-01"
}
```

B Scripts Used for Evaluation

```bash
#!/bin/bash
base_dir=${HOME}/workspace/thesis/thesis-impl
# Go to work directory
cd ${base_dir}/chf-scenario

# Build release version of project
cargo build -r

# Go back
cd ${base_dir}

# Create dist
mkdir -p ${base_dir}/dist/tmp
cd ${base_dir}/dist/tmp
cp -r ${base_dir}/chf-scenario/data \
     ${base_dir}/chf-scenario/target/release/chf-scenario \
     ${HOME}/.ssh/chf-modifier_key.pem \
     ${HOME}/.ssh/chf-signer_key.pem \
     ${HOME}/.ssh/chf-recipient_key.pem \
     ${base_dir}/cloud_eval_chf_azure.sh \
     ${base_dir}/run_eval_chf_azure.sh .

tar czf chf.tar.gz *
cd ${base_dir}/dist
mv tmp/chf.tar.gz .
rm -rf tmp

update="sudo apt update && sudo apt upgrade -y"
echo "Running ${update}"
ssh -i ${HOME}/.ssh/chf-evaluator_key.pem azureuser@evaluator-vm ${update}

# Install
```

```
target_dir=/home/azureuser
scp -i ${HOME}/.ssh/chf-evaluator_key.pem ${base_dir}/dist/chf.tar.gz azureuser@
evaluator-vm:${target_dir}

install="cd ${target_dir}; tar xf chf.tar.gz; mv *.pem .ssh/; \
    rm chf.tar.gz; tar czf chf.tar.gz data chf-scenario"
echo "Running ${install}"
ssh -i ${HOME}/.ssh/chf-evaluator_key.pem azureuser@evaluator-vm ${install}
```

ssage Size and Security Parameters

```

#!/usr/bin/bash
signervm=10.0.0.4
modifiervm=10.0.0.5
recipientvm=10.0.0.6

setup () {
    # Set ip variables
    local get_ip="hostname -I;"
    echo "Running ${get_ip}"
    signer_ip=$(ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${g
et_ip} | awk '{$1=$1};1')
    modifier_ip=$(ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifierv
m} ${get_ip} | awk '{$1=$1};1')
    recipient_ip=$(ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipie
ntvm} ${get_ip} | awk '{$1=$1};1')

    signer_to_modifier="${modifier_ip}:8888"
    recipient_to_modifier="${modifier_ip}:8889"
    recipient_to_signer="${signer_ip}:8989"
}

run_scenario () {
    # RECIPIENT - background
    local recipient_cmd="chf-scenario --security ${sp} run ${scenario[1]}"
    echo "Running on ${recipientvm}: ${recipient_cmd}"
    ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm} ${recipie
nt_cmd} &

    # MODIFIER - background
    local modifier_cmd="chf-scenario --security ${sp} run ${scenario[2]}"
    echo "Running on ${modifiervm}: ${modifier_cmd}"
    ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm} ${modifier_
cmd} &

    # SIGNER - foreground (starts and ends protocol run)
    local signer_cmd="chf-scenario --security ${sp} run ${scenario[0]} --message
-size ${ms}"
    echo "Running on ${signervm}: ${signer_cmd}"
    ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${signer_cmd}

    # Copy logs
    scp -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm}:${cloud_workdir
```

```
}/chf-scenario.log ${out_dir}/${scenario_name_short}/${sp}/${ms}/signer.log
    scp -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm}:${cloud_wor
kdir}/chf-scenario.log ${out_dir}/${scenario_name_short}/${sp}/${ms}/modifier.lo
g
    scp -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm}:${cloud_w
orkdir}/chf-scenario.log ${out_dir}/${scenario_name_short}/${sp}/${ms}/recipient
.log

    # Clean up
    local clean="sudo rm ${cloud_workdir}/chf-scenario.log"
    ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${clean}
    ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm} ${clean}
    ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm} ${clean}
}

dry_run_scenario () {
    echo "Executing with security parameter = ${sp} and message size = ${ms}"
}

teardown () {
    echo ""
    echo "Aborting cloud_eval_chf.sh..."
    local terminate="pkill chf-scenario;"
    ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${terminate}
    ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm} ${terminate
}
    ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm} ${termina
te}
    exit
}

run_all () {
    # Setup directory structure for storing results
    # logs/security_parameter/message_size
    echo "Running scenario: ${scenario_name}"
    for sp in "${security_parameters[@]}"; do
        for ms in "${message_sizes[@]}"; do
            mkdir -p ${out_dir}/${scenario_name_short}/${sp}/${ms}
            dry_run_scenario
            run_scenario
        done
    done
}

trap teardown SIGINT
cloud_workdir=/home/azureuser
install_dir=/usr/local/bin

if [[ -z "$1" ]]; then
    echo "eval_chf.sh [naive, modsign, nomod, chf, nomodchf] <output directory>"
    exit -1
fi

```

```
if [[ -z "$2" ]]; then
    out_dir=logs/cloud
else
    out_dir=$2
fi

setup

case $1 in
    naive)
        scenario_name_short="naive"
        scenario_name="Naive RSA + sha512"
        scenario=( \
            "signer --modifier-addr ${signer_to_modifier}" \
            "recipient --modifier-addr ${recipient_to_modifier} --signer-addr ${recipient_to_signer}" \
            "modifier --signer-addr ${signer_to_modifier} --recipient-addr ${recipient_to_modifier}" \
        )
        ;;

    modsign)
        scenario_name_short="modsign"
        scenario_name="Modifier signs RSA + sha512"
        scenario=( \
            "signer --modifier-addr ${signer_to_modifier} --no-extra-signature" \
            "recipient --modifier-addr ${recipient_to_modifier} --signer-addr ${recipient_to_signer}" \
            "modifier --signer-addr ${signer_to_modifier} --recipient-addr ${recipient_to_modifier} --has-signer-key" \
        )
        ;;

    nomod)
        scenario_name_short="nomod"
        scenario_name="No modification RSA + sha512"
        scenario=( \
            "signer --modifier-addr ${signer_to_modifier} --no-extra-signature" \
            "recipient --modifier-addr ${recipient_to_modifier} --signer-addr ${recipient_to_signer}" \
            "modifier --signer-addr ${signer_to_modifier} --recipient-addr ${recipient_to_modifier} --forward-only" \
        )
        ;;

    chf)
        scenario_name_short="chf"
        scenario_name="RSA + CHF"
        scenario=( \
            "chf-signer --modifier-addr ${signer_to_modifier}" \
            "chf-recipient --modifier-addr ${recipient_to_modifier} --signer-add
```

```
r ${recipient_to_signer}" \
            "chf-modifier --signer-addr ${signer_to_modifier} --recipient-addr $
{recipient_to_modifier}" \
        )
        ;;

    nomodchf)
        scenario_name_short="nomodchf"
        scenario_name="No modification RSA + CHF"
        scenario=( \
            "chf-signer --modifier-addr ${signer_to_modifier}" \
            "chf-recipient --modifier-addr ${recipient_to_modifier} --signer-add
r ${recipient_to_signer}" \
            "chf-modifier --signer-addr ${signer_to_modifier} --recipient-addr $
{recipient_to_modifier} --forward-only" \
        )
        ;;
esac

# https://www.rfc-editor.org/rfc/rfc3526#section-8
# Two lower parameters + the ones listed in above rfc
# RSA and CHF security parameters are the same, even though RSA should be at lea
st 2048 while the discrete log requires slightly lower
security_parameters=(1024 2048 3072 4096 6144 8192)
# 1 byte -> 100 Megabyte
message_sizes=(\
    $((10**0)) $((5*10**0)) $((10**1)) $((5*10**1)) $((10**2)) $((5*10**2)) \
    $((10**3)) $((5*10**3)) $((10**4)) $((5*10**4)) $((10**5)) $((5*10**5)) \
    $((10**6)) $((5*10**6)) $((10**7)) $((5*10**7)) $((10**8)))

run_all
```

181
narios

```

#!/usr/bin/bash
signervm=10.0.0.4
modifiervm=10.0.0.5
recipientvm=10.0.0.6

setup () {
    # Update hosts
    local update="sudo apt update && sudo apt upgrade -y"
    echo "Running ${update}"
    ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${update}
    ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm} ${update}
    ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm} ${update}

    # Copy files
    scp -i ${HOME}/.ssh/chf-signer_key.pem chf.tar.gz azureuser@${signervm}:${cl
oud_workdir}
    scp -i ${HOME}/.ssh/chf-modifier_key.pem chf.tar.gz azureuser@${modifiervm}:
${cloud_workdir}
```

```
    scp -i ${HOME}/.ssh/chf-recipient_key.pem chf.tar.gz azureuser@${recipientvm
}:${cloud_workdir}

    # install program on hosts
    local install="tar xf chf.tar.gz; sudo mv chf-scenario ${install_dir}; rm ch
f.tar.gz;"
    echo "Running ${install}"
    ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${install}
    ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm} ${install}
    ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm} ${install
}
}

teardown () {
    echo ""
    echo "Exiting run_eval_chf.sh..."
    local terminate="pkill chf-scenario;"
    ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${terminate}
    ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm} ${terminate
}
    ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm} ${termina
te}

    local uninstall="sudo rm ${install_dir}/chf-scenario; rm -rf ${cloud_workdir
}/data/"
    ssh -i ${HOME}/.ssh/chf-signer_key.pem azureuser@${signervm} ${uninstall}
    ssh -i ${HOME}/.ssh/chf-modifier_key.pem azureuser@${modifiervm} ${uninstall
}
    ssh -i ${HOME}/.ssh/chf-recipient_key.pem azureuser@${recipientvm} ${uninsta
ll}
    exit
}

finalize_logs () {
    tar cvzf eval-$(date +'%s').tar.gz logs/ && rm -rf logs
}

run_timed_script () {
    start=$(date +'%F %T %z') &&
    ${script} ${scenario} ${path} &&
    end=$(date +'%F %T %z') &&
    echo ${start} > ${path}/${scenario}/start &&
    echo ${end} > ${path}/${scenario}/end
}

trap teardown SIGINT
cloud_workdir=/home/azureuser
install_dir=/usr/local/bin

if [[ -z "$1" ]]; then
    echo "run_eval_chf.sh <iterations>"
    exit -1
fi
```

```

script_name="cloud_eval_chf_azure.sh"
script_full_path=$(dirname "${script_name}")
script=${script_full_path}/${script_name}
scenarios=(naive modsign nomod chf nomodchf)

setup
for i in $(seq ${1}); do
    path=logs/cloud/iteration_${i}
    for scenario in "${scenarios[@]}"; do
        run_timed_script
    done
done
finalize_logs
teardown
```

C Metadata of data set

Table C.1: The start and end times for each iteration in all evaluation runs

filename	iteration	start	end
Azure (1)	1	2025-04-28 11:28:56 +0000	2025-04-28 12:50:02 +0000
Azure (1)	2	2025-04-28 12:50:02 +0000	2025-04-28 14:11:05 +0000
Azure (1)	3	2025-04-28 14:11:05 +0000	2025-04-28 15:32:07 +0000
Azure (1)	4	2025-04-28 15:32:07 +0000	2025-04-28 16:53:05 +0000
Azure (1)	5	2025-04-28 16:53:05 +0000	2025-04-28 18:14:00 +0000
Azure (1)	6	2025-04-28 18:14:00 +0000	2025-04-28 19:34:57 +0000
Azure (1)	7	2025-04-28 19:34:57 +0000	2025-04-28 20:55:55 +0000
Azure (1)	8	2025-04-28 20:55:55 +0000	2025-04-28 22:16:53 +0000
Azure (1)	9	2025-04-28 22:16:53 +0000	2025-04-28 23:37:51 +0000
Azure (1)	10	2025-04-28 23:37:51 +0000	2025-04-29 00:58:55 +0000
Azure (1)	11	2025-04-29 00:58:55 +0000	2025-04-29 02:19:51 +0000
Azure (2)	1	2025-04-29 08:45:10 +0000	2025-04-29 09:14:44 +0000
Azure (2)	2	2025-04-29 09:14:44 +0000	2025-04-29 09:44:11 +0000
Azure (2)	3	2025-04-29 09:44:11 +0000	2025-04-29 10:13:41 +0000
Azure (2)	4	2025-04-29 10:13:41 +0000	2025-04-29 10:43:14 +0000
Azure (2)	5	2025-04-29 10:43:14 +0000	2025-04-29 11:12:52 +0000
Azure (2)	6	2025-04-29 11:12:52 +0000	2025-04-29 11:42:26 +0000
Azure (2)	7	2025-04-29 11:42:26 +0000	2025-04-29 12:12:02 +0000
Azure (2)	8	2025-04-29 12:12:02 +0000	2025-04-29 12:41:30 +0000
Azure (2)	9	2025-04-29 12:41:30 +0000	2025-04-29 13:10:57 +0000
Azure (2)	10	2025-04-29 13:10:57 +0000	2025-04-29 13:40:29 +0000
Azure (2)	11	2025-04-29 13:40:29 +0000	2025-04-29 14:09:58 +0000
Azure (2)	12	2025-04-29 14:09:58 +0000	2025-04-29 14:39:28 +0000
Azure (2)	13	2025-04-29 14:39:28 +0000	2025-04-29 15:08:51 +0000
Azure (2)	14	2025-04-29 15:08:51 +0000	2025-04-29 15:38:15 +0000

Continued on next page

Table C.1: The start and end times for each iteration in all evaluation runs

filename	iteration	start	end
Azure (2)	15	2025-04-29 15:38:15 +0000	2025-04-29 16:07:46 +0000
Azure (2)	16	2025-04-29 16:07:46 +0000	2025-04-29 16:37:15 +0000
Azure (2)	17	2025-04-29 16:37:15 +0000	2025-04-29 17:06:41 +0000
Azure (2)	18	2025-04-29 17:06:41 +0000	2025-04-29 17:36:07 +0000
Azure (2)	19	2025-04-29 17:36:07 +0000	2025-04-29 18:05:30 +0000
Azure (2)	20	2025-04-29 18:05:30 +0000	2025-04-29 18:34:53 +0000
Azure (2)	21	2025-04-29 18:34:53 +0000	2025-04-29 19:04:16 +0000
Azure (3)	1	2025-05-01 07:57:24 +0000	2025-05-01 08:26:20 +0000
Azure (3)	2	2025-05-01 08:26:20 +0000	2025-05-01 08:55:13 +0000
Azure (3)	3	2025-05-01 08:55:13 +0000	2025-05-01 09:24:04 +0000
Azure (3)	4	2025-05-01 09:24:04 +0000	2025-05-01 09:52:56 +0000
Azure (3)	5	2025-05-01 09:52:56 +0000	2025-05-01 10:21:49 +0000
Azure (3)	6	2025-05-01 10:21:49 +0000	2025-05-01 10:50:43 +0000
Azure (3)	7	2025-05-01 10:50:43 +0000	2025-05-01 11:19:36 +0000
Azure (3)	8	2025-05-01 11:19:36 +0000	2025-05-01 11:48:22 +0000
Azure (3)	9	2025-05-01 11:48:22 +0000	2025-05-01 12:17:14 +0000
Azure (3)	10	2025-05-01 12:17:14 +0000	2025-05-01 12:46:07 +0000
Azure (3)	11	2025-05-01 12:46:07 +0000	2025-05-01 13:14:58 +0000
Azure (3)	12	2025-05-01 13:14:58 +0000	2025-05-01 13:43:54 +0000
Azure (3)	13	2025-05-01 13:43:54 +0000	2025-05-01 14:12:58 +0000
Azure (3)	14	2025-05-01 14:12:58 +0000	2025-05-01 14:41:58 +0000
Azure (3)	15	2025-05-01 14:41:58 +0000	2025-05-01 15:10:58 +0000
Azure (3)	16	2025-05-01 15:10:58 +0000	2025-05-01 15:40:06 +0000
Azure (3)	17	2025-05-01 15:40:06 +0000	2025-05-01 16:08:59 +0000
Azure (3)	18	2025-05-01 16:08:59 +0000	2025-05-01 16:37:53 +0000
Azure (3)	19	2025-05-01 16:37:53 +0000	2025-05-01 17:06:46 +0000
Azure (3)	20	2025-05-01 17:06:46 +0000	2025-05-01 17:35:45 +0000
Azure (3)	21	2025-05-01 17:35:45 +0000	2025-05-01 18:04:40 +0000
Azure (3)	22	2025-05-01 18:04:40 +0000	2025-05-01 18:33:49 +0000
Azure (3)	23	2025-05-01 18:33:49 +0000	2025-05-01 19:03:07 +0000
Azure (3)	24	2025-05-01 19:03:07 +0000	2025-05-01 19:32:27 +0000
Azure (3)	25	2025-05-01 19:32:27 +0000	2025-05-01 20:01:27 +0000
Azure (3)	26	2025-05-01 20:01:27 +0000	2025-05-01 20:30:29 +0000
Azure (3)	27	2025-05-01 20:30:29 +0000	2025-05-01 20:59:28 +0000
Azure (3)	28	2025-05-01 20:59:28 +0000	2025-05-01 21:28:34 +0000
Azure (3)	29	2025-05-01 21:28:34 +0000	2025-05-01 21:57:39 +0000
Azure (3)	30	2025-05-01 21:57:39 +0000	2025-05-01 22:26:37 +0000
Azure (3)	31	2025-05-01 22:26:37 +0000	2025-05-01 22:55:31 +0000
Azure (3)	32	2025-05-01 22:55:31 +0000	2025-05-01 23:24:31 +0000
Azure (3)	33	2025-05-01 23:24:31 +0000	2025-05-01 23:53:23 +0000
Azure (3)	34	2025-05-01 23:53:23 +0000	2025-05-02 00:22:16 +0000
Azure (3)	35	2025-05-02 00:22:16 +0000	2025-05-02 00:51:23 +0000
Azure (3)	36	2025-05-02 00:51:23 +0000	2025-05-02 01:20:18 +0000
Azure (3)	37	2025-05-02 01:20:18 +0000	2025-05-02 01:49:08 +0000
Azure (3)	38	2025-05-02 01:49:08 +0000	2025-05-02 02:17:58 +0000
Azure (3)	39	2025-05-02 02:17:58 +0000	2025-05-02 02:46:48 +0000
Azure (3)	40	2025-05-02 02:46:48 +0000	2025-05-02 03:15:36 +0000

Continued on next page

Table C.1: The start and end times for each iteration in all evaluation runs

filename	iteration	start	end
Azure (3)	41	2025-05-02 03:15:36 +0000	2025-05-02 03:44:25 +0000
Azure (4)	1	2025-05-05 07:27:00 +0000	2025-05-05 07:51:29 +0000
Azure (4)	2	2025-05-05 07:51:29 +0000	2025-05-05 08:15:55 +0000
Azure (4)	3	2025-05-05 08:15:55 +0000	2025-05-05 08:40:22 +0000
Azure (4)	4	2025-05-05 08:40:22 +0000	2025-05-05 09:04:46 +0000
Azure (4)	5	2025-05-05 09:04:46 +0000	2025-05-05 09:29:12 +0000
Azure (4)	6	2025-05-05 09:29:12 +0000	2025-05-05 09:53:38 +0000
Azure (4)	7	2025-05-05 09:53:38 +0000	2025-05-05 10:18:02 +0000
Azure (4)	8	2025-05-05 10:18:02 +0000	2025-05-05 10:42:28 +0000
Azure (4)	9	2025-05-05 10:42:28 +0000	2025-05-05 11:06:53 +0000
Azure (4)	10	2025-05-05 11:06:53 +0000	2025-05-05 11:31:17 +0000
Azure (4)	11	2025-05-05 11:31:17 +0000	2025-05-05 11:55:42 +0000
Azure (4)	12	2025-05-05 11:55:42 +0000	2025-05-05 12:20:05 +0000
Azure (4)	13	2025-05-05 12:20:05 +0000	2025-05-05 12:44:30 +0000
Azure (4)	14	2025-05-05 12:44:30 +0000	2025-05-05 13:08:55 +0000
Azure (4)	15	2025-05-05 13:08:55 +0000	2025-05-05 13:33:20 +0000
Azure (4)	16	2025-05-05 13:33:20 +0000	2025-05-05 13:57:45 +0000
Azure (4)	17	2025-05-05 13:57:45 +0000	2025-05-05 14:22:10 +0000
Azure (4)	18	2025-05-05 14:22:10 +0000	2025-05-05 14:46:35 +0000
Azure (4)	19	2025-05-05 14:46:35 +0000	2025-05-05 15:10:59 +0000
Azure (4)	20	2025-05-05 15:10:59 +0000	2025-05-05 15:35:24 +0000
Azure (4)	21	2025-05-05 15:35:24 +0000	2025-05-05 15:59:50 +0000
Azure (4)	22	2025-05-05 15:59:50 +0000	2025-05-05 16:24:15 +0000
Azure (4)	23	2025-05-05 16:24:15 +0000	2025-05-05 16:48:39 +0000
Azure (4)	24	2025-05-05 16:48:39 +0000	2025-05-05 17:13:04 +0000
Azure (4)	25	2025-05-05 17:13:04 +0000	2025-05-05 17:37:29 +0000
Azure (4)	26	2025-05-05 17:37:29 +0000	2025-05-05 18:01:55 +0000
Azure (4)	27	2025-05-05 18:01:55 +0000	2025-05-05 18:26:23 +0000
Azure (4)	28	2025-05-05 18:26:23 +0000	2025-05-05 18:50:49 +0000
Azure (4)	29	2025-05-05 18:50:49 +0000	2025-05-05 19:15:13 +0000
Azure (4)	30	2025-05-05 19:15:13 +0000	2025-05-05 19:39:38 +0000
Azure (4)	31	2025-05-05 19:39:38 +0000	2025-05-05 20:04:03 +0000
Azure (4)	32	2025-05-05 20:04:03 +0000	2025-05-05 20:28:34 +0000
Azure (4)	33	2025-05-05 20:28:34 +0000	2025-05-05 20:52:59 +0000
Azure (4)	34	2025-05-05 20:52:59 +0000	2025-05-05 21:17:26 +0000
Azure (4)	35	2025-05-05 21:17:26 +0000	2025-05-05 21:41:57 +0000
Azure (4)	36	2025-05-05 21:41:57 +0000	2025-05-05 22:06:22 +0000
Azure (4)	37	2025-05-05 22:06:22 +0000	2025-05-05 22:30:48 +0000
Azure (4)	38	2025-05-05 22:30:48 +0000	2025-05-05 22:55:14 +0000
Azure (4)	39	2025-05-05 22:55:14 +0000	2025-05-05 23:19:38 +0000
Azure (4)	40	2025-05-05 23:19:38 +0000	2025-05-05 23:44:03 +0000
Azure (4)	41	2025-05-05 23:44:03 +0000	2025-05-06 00:08:28 +0000
Azure (4)	42	2025-05-06 00:08:28 +0000	2025-05-06 00:32:54 +0000
Azure (4)	43	2025-05-06 00:32:54 +0000	2025-05-06 00:57:26 +0000
Azure (4)	44	2025-05-06 00:57:26 +0000	2025-05-06 01:21:57 +0000
Azure (4)	45	2025-05-06 01:21:57 +0000	2025-05-06 01:46:25 +0000
Azure (4)	46	2025-05-06 01:46:25 +0000	2025-05-06 02:10:54 +0000

Continued on next page

Table C.1: The start and end times for each iteration in all evaluation runs

filename	iteration	start	end
Azure (4)	47	2025-05-06 02:10:54 +0000	2025-05-06 02:35:22 +0000
Azure (4)	48	2025-05-06 02:35:22 +0000	2025-05-06 02:59:53 +0000
Azure (4)	49	2025-05-06 02:59:53 +0000	2025-05-06 03:24:22 +0000
Azure (4)	50	2025-05-06 03:24:22 +0000	2025-05-06 03:48:51 +0000
Azure (4)	51	2025-05-06 03:48:51 +0000	2025-05-06 04:13:18 +0000
Azure (5)	1	2025-05-06 20:18:47 +0000	2025-05-06 20:43:20 +0000
Azure (5)	2	2025-05-06 20:43:20 +0000	2025-05-06 21:07:51 +0000
Azure (5)	3	2025-05-06 21:07:51 +0000	2025-05-06 21:32:23 +0000
Azure (5)	4	2025-05-06 21:32:23 +0000	2025-05-06 21:56:53 +0000
Azure (5)	5	2025-05-06 21:56:53 +0000	2025-05-06 22:21:24 +0000
Azure (5)	6	2025-05-06 22:21:24 +0000	2025-05-06 22:45:54 +0000
Azure (5)	7	2025-05-06 22:45:54 +0000	2025-05-06 23:10:23 +0000
Azure (5)	8	2025-05-06 23:10:23 +0000	2025-05-06 23:34:52 +0000
Azure (5)	9	2025-05-06 23:34:52 +0000	2025-05-06 23:59:27 +0000
Azure (5)	10	2025-05-06 23:59:27 +0000	2025-05-07 00:23:55 +0000
Azure (5)	11	2025-05-07 00:23:55 +0000	2025-05-07 00:48:24 +0000
Azure (5)	12	2025-05-07 00:48:24 +0000	2025-05-07 01:12:54 +0000
Azure (5)	13	2025-05-07 01:12:54 +0000	2025-05-07 01:37:26 +0000
Azure (5)	14	2025-05-07 01:37:26 +0000	2025-05-07 02:01:57 +0000
Azure (5)	15	2025-05-07 02:01:57 +0000	2025-05-07 02:26:28 +0000
Azure (5)	16	2025-05-07 02:26:28 +0000	2025-05-07 02:51:01 +0000
Azure (5)	17	2025-05-07 02:51:01 +0000	2025-05-07 03:15:32 +0000
Azure (5)	18	2025-05-07 03:15:32 +0000	2025-05-07 03:40:02 +0000
Azure (5)	19	2025-05-07 03:40:02 +0000	2025-05-07 04:04:31 +0000
Azure (5)	20	2025-05-07 04:04:31 +0000	2025-05-07 04:29:02 +0000
Azure (5)	21	2025-05-07 04:29:02 +0000	2025-05-07 04:53:31 +0000
Azure (5)	22	2025-05-07 04:53:31 +0000	2025-05-07 05:18:01 +0000
Azure (5)	23	2025-05-07 05:18:01 +0000	2025-05-07 05:42:31 +0000
Azure (5)	24	2025-05-07 05:42:31 +0000	2025-05-07 06:07:02 +0000
Azure (5)	25	2025-05-07 06:07:02 +0000	2025-05-07 06:31:37 +0000
Azure (5)	26	2025-05-07 06:31:37 +0000	2025-05-07 06:56:09 +0000
Azure (5)	27	2025-05-07 06:56:09 +0000	2025-05-07 07:20:39 +0000
Azure (5)	28	2025-05-07 07:20:39 +0000	2025-05-07 07:45:10 +0000
Lenovo (1)	1	2025-04-26 12:33:20 +0200	2025-04-26 14:14:11 +0200
Lenovo (1)	2	2025-04-26 14:14:11 +0200	2025-04-26 15:54:49 +0200
Lenovo (1)	3	2025-04-26 15:54:49 +0200	2025-04-26 17:35:43 +0200
Lenovo (1)	4	2025-04-26 17:35:43 +0200	2025-04-26 19:17:34 +0200
Lenovo (1)	5	2025-04-26 19:17:34 +0200	2025-04-26 20:58:02 +0200
Lenovo (1)	6	2025-04-26 20:58:02 +0200	2025-04-26 22:38:34 +0200
Lenovo (1)	7	2025-04-26 22:38:34 +0200	2025-04-27 00:19:15 +0200
Lenovo (1)	8	2025-04-27 00:19:15 +0200	2025-04-27 02:00:26 +0200
Lenovo (1)	9	2025-04-27 02:00:26 +0200	2025-04-27 03:40:59 +0200
Lenovo (1)	10	2025-04-27 03:40:59 +0200	2025-04-27 05:21:41 +0200
Lenovo (1)	11	2025-04-27 05:21:41 +0200	2025-04-27 07:02:20 +0200
Lenovo (2)	1	2025-04-27 07:06:03 +0200	2025-04-27 08:47:20 +0200
Lenovo (2)	2	2025-04-27 08:47:20 +0200	2025-04-27 10:28:09 +0200
Lenovo (2)	3	2025-04-27 10:28:09 +0200	2025-04-27 12:09:21 +0200

Continued on next page

Table C.1: The start and end times for each iteration in all evaluation runs

filename	iteration	start	end
Lenovo (2)	4	2025-04-27 12:09:21 +0200	2025-04-27 13:50:30 +0200
Lenovo (2)	5	2025-04-27 13:50:30 +0200	2025-04-27 15:30:51 +0200
Lenovo (2)	6	2025-04-27 15:30:51 +0200	2025-04-27 17:11:31 +0200
Lenovo (2)	7	2025-04-27 17:11:31 +0200	2025-04-27 18:52:07 +0200
Lenovo (2)	8	2025-04-27 18:52:07 +0200	2025-04-27 20:32:30 +0200
Lenovo (2)	9	2025-04-27 20:32:30 +0200	2025-04-27 22:13:08 +0200
Lenovo (2)	10	2025-04-27 22:13:08 +0200	2025-04-27 23:54:04 +0200
Lenovo (2)	11	2025-04-27 23:54:04 +0200	2025-04-28 01:33:59 +0200