University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Homomorphic Encryption: Implementing the Comparison Operation

Gabriel Jonathan Stegmaier
Zurich, Switzerland
Student ID: 22-711-832

Supervisor: Daria Schumm, Thomas Grübl, Prof. Dr. Burkhard Stiller
Date of Submission: July 01, 2025

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 30.06.2025 G. Stegmaier

Signature of student

Abstract

Dezentralisierte Identitätsanwendungen versuchen, die Nachteile heutiger zentralisierter Identitätsmanagementsystem abzumildern, die Sicherheit zu erhöhen und Nutzenden mehr Kontrolle über ihre digitale Identität zu geben. Homomorphe Verschlüsselung ermöglicht arithmetische Operationen mit verschlüsselten Daten. Die Kombination dieser beiden Konzepte ergibt interessante Anwendungen in der Verifikation digitaler Identifikationsnachweise. Beispielsweise kann deren Zusammenspiel verwendet werden, um das Ausstelldatum eines Identitätsnachweises durch den Vergleich mit einem Schwellwert zu verifizieren. Dies ist der Anwendungsfall, welcher in dieser Arbeit angenommen wird. Um eine solche Überprüfung zu ermöglichen, ergibt sich schnell der Bedarf nach einer verschlüsselten Vergleichsoperation (Schwellwert $\leq$ Ausstelldatum). Vor dieser Arbeit gab es keine Implementierung einer solchen Vergleichsoperation in JavaScript. Mit diesem Hintergrund werden verschiedene Wege untersucht, eine Implementierung in JavaScript vorzunehmen und sie in den existierenden Prototypen einer dezentralisierten Identitätsanwendung zu integrieren. Die implementierte Version ist vollständig funktionsfähig und wurde erfolgreich in den existierenden Prototypen integriert. Allerdings weist die neue Implementierung gewisse Leistungsnachteile gegenüber alternativen Ansätzen auf.

Decentralized identity applications attempt to mitigate the shortcomings of today's centralized identity management systems, improving security and granting users more control over their digital identity. Homomorphic encryption allows for arithmetic operations on encrypted data. The combination of these two concepts results in interesting applications for the verification of digital credentials. For example, they can be combined to verify the issuance date of a credential against a threshold. This is the use case assumed in this work. To render such a verification possible, the need for an encrypted comparison operation (threshold date $\leq$ issuance date) soon emerges. However, prior to this thesis, no such homomorphic comparison operation has been implemented in JavaScript. Considering this, this work explores ways of implementing the homomorphic comparison operation in JavaScript, as well as integrating it with an existing prototype of a decentralized identity application. The implemented version of the comparison operation is fully functional and was successfully integrated into the existing decentralized identity application. However, it shows some performance disadvantage when compared to alternative approaches.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Daria Schumm. Her support, constructive feedback and ideas throughout the writing of this Bachelor's thesis have been essential for its completion. Also, I would like to extend my sincere thanks to Prof. Dr. Burkhard Stiller for the opportunity of writing my Bachelor's thesis at the Communication Systems Group of the University of Zurich. Finally, I appreciate the encouragement from my friends and family, whose belief in my work kept me motivated.

Contents

Chapter 1

Introduction

1.1 Motivation

Homomorphic encryption (HE) allows arithmetic operations to be performed directly on encrypted data, returning the correct result upon decryption. This encryption method is usually proposed for applications like genomics, finances and the protection of data privacy in machine learning [10]. However, another interesting application of it are decentralized identity applications [43]. Specifically, homomorphic encryption can be used for the verification of credentials without disclosing the information to be verified. In such an application, the need for a homomorphic comparison operation (e.g. enc(a) $\leq$ enc(b)) emerges. For example, if a prover (e.g. a student) wants to prove to a verifier (e.g. a company) that a credential was issued after a certain threshold date, they need to homomorphically compute the expression

$$\text{enc(threshold date)} \leq \text{enc(issuance date)}.$$

This is the use case this work builds upon. The problem arising is that the encrypted comparison operation is not trivial to compute and often not provided by HE libraries [24]. Especially in JavaScript, which is used for the existing prototype of a decentralized identity application [40], the homomorphic comparison operation was not supported prior to this work. The original prototype uses node-seal, a homomorphic encryption library based on Microsoft SEAL[29], for homomorphic calculations. Since no comparison operation is available in node-seal, the prototype relies on a workaround for the verification process. This motivates the need for a cleaner version using the comparison operation.

1.2 Thesis Goals

The main goal of this thesis is the implementation of a homomorphically encrypted comparison operation in JavaScript. The comparison operation has to support 64-bit integers (Unix timestamps) and should be integrated into the original prototype using an appropriate protocol.

A more fine-grained outline of the thesis goals, as described in the task description, consists of the following points:

1. Establishing a background on decentralized identity systems and encryption
2. Conducting a comprehensive literature review on homomorphic encryption and the homomorphically encrypted comparison operation
3. Formulation of a problem statement motivating the need for an encrypted comparison operation
4. Design of a prototype which effectively uses the comparison operation in a decentralized identity system
5. Implementation of the comparison operation and extension of provided code base
6. Evaluation of the newly implemented prototype against the existing Zero-Knowledge proof (ZKP) approach.

1.3 Methodology

The first part of this thesis is of theoretical character, focusing on research existing on decentralized identity applications, homomorphic encryption and the comparison operation in homomorphic encryption. To get an overview about existing homomorphic encryption schemes and corresponding implementations, a general survey on homomorphic encryption schemes [2] is taken as a starting point. From the sources cited in this general survey, forward snowballing is used to find state-of-the-art papers on homomorphic encryption. After the identification of the most important homomorphic encryption schemes, methods for comparing two numbers are researched for each scheme.
The design part of this thesis is more practical, focusing on the feasibility and most suitable approach of implementing the comparison operation in JavaScript and integrating it with the existing decentralized application prototype. The choice of the encryption scheme and ways of translating existing implementations to JavaScript are explored in this part. Thereafter, two ways of computing the homomorphically encrypted comparison operation are implemented, and one of these implementations is integrated into the decentralized identity application. In the evaluation, the method implemented in this work is compared with the Zero-Knowledge Proof (ZKP) and the original HE prototype provided in [43].

1.4 Thesis Outline

This thesis will first focus on the background knowledge needed for understanding from a mathematical as well as an implementation perspective. Related works that discuss

the comparison operation in homomorphic encryption as well as decentralized identity applications are presented in chapter 2. In chapter 3, the design choices will be discussed, providing diagrams and explanations needed to understand the implementation. The implementation of the comparison operation in JavaScript and evaluation thereof are discussed in chapters 4 and 5.

Chapter 2

Fundamentals

In this chapter, the first three goals of this thesis:

- Establishing a background on decentralized identity systems and encryption
- Conducting a comprehensive literature review on homomorphic encryption and the homomorphically encrypted comparison operation
- Formulation of a problem statement

are addressed. A background on decentralized identity systems and related encryption concepts (with a special focus on homomorphic encryption) is established in section 2.1. In section 2.2, comparison algorithms for each encryption scheme as well as libraries compatible with these algorithms are discussed. The problem statement is formulated in section 2.3.

2.1 Background

2.1.1 Decentralized identity applications

A decentralized identity application consists of three main contributors: an issuer, a holder/prover and a verifier [42]. At the core, these applications have been designed to give the holder as much control over their digital identity as possible. The protocol between involved parties is as follows: The issuer issues a verifiable credential (e.g. a University degree) to a holder (e.g. a student), who stores the credential in their digital wallet associated with a decentralized identifier (DID). Thereafter, the holder (now in the role of a prover) can present the data contained in the credential to a verifier (e.g. a company), who is able to verify its validity via cryptographic proofs. A concept especially useful in terms of data privacy is selective disclosure, where the prover can give the verifier only exactly the information needed and e.g. only reveal first and last name without disclosing the address. In decentralized identity applications, it is often useful

and even more privacy-preserving if there is a way of proving e.g. age over 18 without disclosing the birth date, or showing that a document was issued after a threshold date without disclosing the issuance date. The latter is the scenario assumed in this work. To avoid information misuse, it must additionally be possible to revoke credentials in decentralized identity applications on time [41]. Cryptographic accumulators are a useful tool for implementing such credential revocation.

2.1.2 Private key and public key cryptography

An important and fundamental concept in cryptography related to decentralized identity applications is the distinction between private and public key cryptography [22]. Traditionally in cryptography, a single key is used to encrypt and decrypt data and ensure secure communication in a method called private key cryptography. However, this causes the problem of needing a secure channel for exchanging this single key between sender and receiver 2.1, which can be cumbersome.

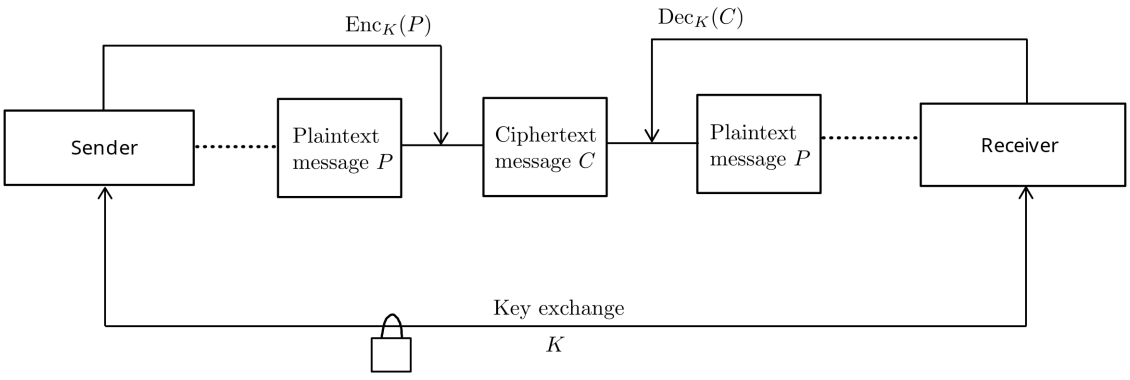

Figure 2.1: Private key/symmetric cryptography

As a solution, an asymmetric (resp. public key) cryptographic scheme can be used, which solves the problem of needing a secure channel for key exchange. In this method, the receiver shares a public key with the sender, who can encrypt his message with this key. Since decryption requires a second, private key, there is no need to encrypt the exchange of the public key, see figure 2.2.

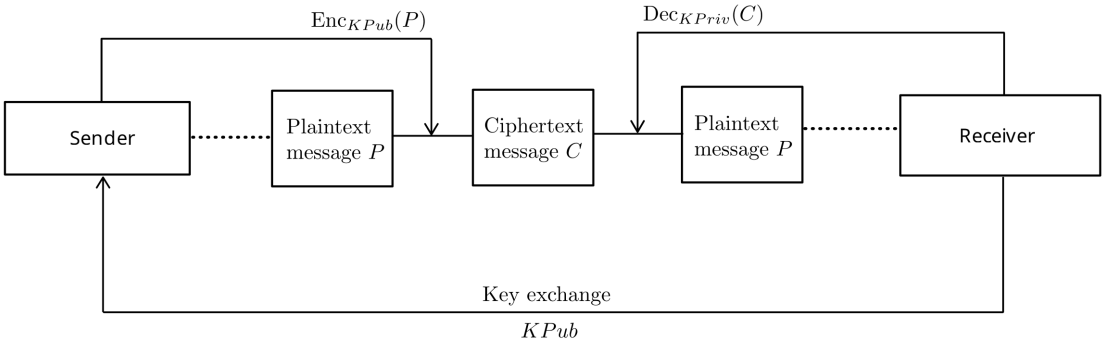

Figure 2.2: Public key/asymmetric cryptography

In decentralized identity applications, public key cryptography is used in the form of digital signatures [42]. Digital signatures enable the signing of digital documents with a private key such that this signature can be verified with the corresponding public key [22]. Hence, in decentralized identity applications, the issuer can digitally sign a credential and the verifier can verify it with the signature.

2.1.3 Zero-knowledge proofs

Zero-knowledge proofs (ZKPs) are a cryptographic method introduced in [18] with the goal of proving the possession of knowledge without revealing the knowledge itself. In the context of decentralized identity applications, ZKPs are very well suited for e.g. the age verification problem. Generally, applications range from electronic voting [30] to nuclear arms control agreements [36]. There are two main categories of Zero-Knowledge Proofs: Interactive ZKPs and non-interactive ZKPs [18], where the former requires multiple iterations of communication and the latter does not.

2.1.4 Homomorphic encryption

Similarly to zero-knowledge proofs, homomorphic encryption can be used to verify an issuance date against a threshold without revealing the issuance date to the verifier. The mechanisms enabling homomorphic encryption are described in the following.

Homomorphisms

To understand the idea of homomorphic encryption, it is useful to know the algebraic concept of a homomorphism. A mapping $f : G \to G'$ is called a homomorphism if it holds for all $a, b \in G$:

$$f(a).f(b) = f(a.b)$$

where G, G' are groups and . is the group operation [26]. The concept can be extended to other algebraic structures such as rings and fields. A homomorphism makes sure that

each operation can be executed in the domain and the range of a set, yielding the same result in both cases.

Homomorphic encryption schemes

An encryption scheme is called homomorphic if it is possible to perform arithmetic operations on encrypted data, producing the correct result in the decrypted domain as well:

$$E(m_1) \star E(m_2) = E(m_1 \star m_2), \forall m_1, m_2 \in M$$

where E is the encryption algorithm, M is the set of all possible messages and $\star$ is the arithmetic operation [2].

Typical (non-homomorphic) encryption schemes require three algorithms [50]:

- Key generation: Generate public and private keys.
- Encrypt: Encryption of a plaintext into a ciphertext using the public key.
- Decrypt: Decrypt ciphertext into plaintext using the private key.

In order to make an encryption scheme homomorphic, a fourth algorithm is typically required [50]:

- Evaluate: Evaluation of an operation in the encrypted domain, typically addition and/or multiplication.

Various schemes for homomorphic encryption can be found in literature. All of them are usually divided into three main categories: Partially Homomorphic Encryption (PHE), Somewhat Homomorphic Encryption (SWHE) and Fully Homomorphic Encryption (FHE) [2]. PHE only allows for homomorphic addition or multiplication, but not both. In SWHE, the number of operations performed in the encrypted domain is limited, e.g. due to increase in noise with each calculation [17]. As the name implies, fully homomorphic encryption allows for an unlimited number of evaluations of arbitrary functions [2].

Another way of categorizing encryption schemes is the division into bit-wise and word-wise encoding [52]. The first category includes the FHEW (fastest homomorphic encryption in the west) and the TFHE (fully homomorphic encryption over the torus) scheme, while examples of the second category are the BGV (Brakerski-Gentry-Vaikuntanathan), BFV (Brakerski-Fan-Vercauteren) and CKKS (Cheon-Kim-Kim-Song) schemes [52]. In the word-wise encryption schemes, BGV and BFV use integer values, while the CKKS is an approximative scheme handling floating-point numbers.

Learning with errors (LWE)

Since many fully homomorphic encryption schemes are based on a mathematical problem called *learning with errors* [37], this shall be described briefly in the following section. The problem consists of a set of equations modulo a large prime number p:

$$\begin{aligned}\langle \mathbf{s}, \mathbf{a}_1 \rangle &\approx_\chi b_1 \pmod p \\ \langle \mathbf{s}, \mathbf{a}_2 \rangle &\approx_\chi b_2 \pmod p \\ &\vdots\end{aligned}$$

where $\mathbf{s} \in \mathbb{Z}_p^n$ (a vector of dimension n over the finite field $\mathbb{Z}_p$), $\mathbf{a}_i$ are chosen at random from $\mathbb{Z}_p^n$, and $b_i \in \mathbb{Z}_p$. The errors in every equation are taken from a probability distribution χ. Learning with errors now describes the problem of recovering s from such equations.

Since this notation is mathematically abstract, a short numerical example is provided in the following [38]:

$$\begin{aligned}14s_1 + 15s_2 + 5s_3 + 2s_4 &\approx 8 \pmod{17} \\ 13s_1 + 14s_2 + 14s_3 + 6s_4 &\approx 16 \pmod{17} \\ 6s_1 + 10s_2 + 13s_3 + 1s_4 &\approx 3 \pmod{17} \\ 10s_1 + 4s_2 + 12s_3 + 16s_4 &\approx 12 \pmod{17} \\ 9s_1 + 5s_2 + 9s_3 + 6s_4 &\approx 9 \pmod{17} \\ 3s_1 + 6s_2 + 4s_3 + 5s_4 &\approx 16 \pmod{17} \\ &\vdots \\ 6s_1 + 7s_2 + 16s_3 + 2s_4 &\approx 3 \pmod{17}\end{aligned}$$

The goal is to recover the secret $\mathbf{s} \in \mathbb{Z}_p^n$ (here, $\mathbf{s} = (0, 13, 9, 11)$). Without the errors, this could easily be achieved via Gaussian elimination. Due to the small errors added on the right side however, it is computationally hard to retrieve the secret.

Based on the hardness of the LWE problem, Oded Regev also proposed the following public key cryptosystem [37]:

- **Parameters**: integer m, prime number p and probability distribution χ on $\mathbb{Z}_p$.
- **Private key:** $\mathbf{s} \in \mathbb{Z}_p^n$ chosen uniformly at random.
- **Public key:** Choose m vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m \in \mathbb{Z}_p^n$ independently from the uniform distribution and $e_1, \ldots, e_m \in \mathbb{Z}_p$ from the distribution χ. The public key now is $(\mathbf{a}_i, b_i)_{i=1}^m$, where $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i$
- **Encryption:** For encrypting a bit we choose a subset S of $[m]$. The encryption is $(\sum_{i \in S} \mathbf{a}_i, \sum_{i \in S} b_i)$ if the bit is 0 and $(\sum_{i \in S} \mathbf{a}_i, \lfloor \frac{p}{2} \rfloor + \sum_{i \in S} b_i)$ if the bit is 1.

- **Decryption:** The decryption of a pair $(\mathbf{a}, b)$ is 0 if $b - \langle \mathbf{a}, \mathbf{s} \rangle$ is closer to 0 than to $\lfloor \frac{p}{2} \rfloor$ modulo p. Otherwise the decryption is 1.

The encryption and decryption of 0 and 1 can easily be imagined with the help of a clock dial [23]:

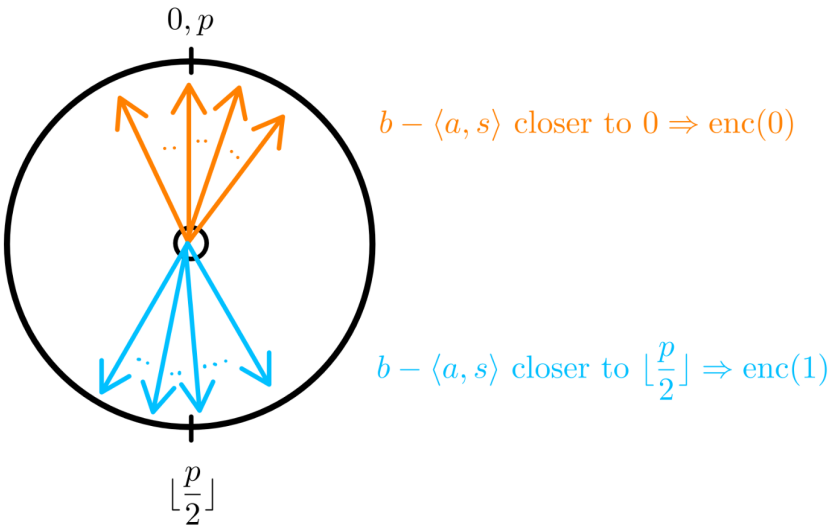

Figure 2.3: Visual representation of the decryption of 0 and 1 via LWE

Ring learning with errors (RLWE)

Ring learning with errors (RLWE) is a problem closely related to LWE, but with more efficient computations involved [35]. It operates in polynomial rings, i.e. all values are polynomials instead of field elements. This is the mathematical problem behind many of today's homomorphic encryption schemes.

Bootstrapping

The fundamental concept introduced by [17] to allow for the first *fully* homomorphic encryption scheme is called bootstrapping, which is a creative way of reducing the noise in a ciphertext after it has increased due to a homomorphic operation.
Before Gentry's breakthrough, the problem in the construction of a fully homomorphic encryption scheme was that noise increased with every operation performed. In figure 2.3, this would correspond to a shift of an arrow around the dial from the original position towards the opposite side. When the noise becomes too large, an encrypted bit can not be correctly decrypted anymore.
Bootstrapping solves this problem by "decrypting within an encrypted environment", as is very well illustrated in [12]:

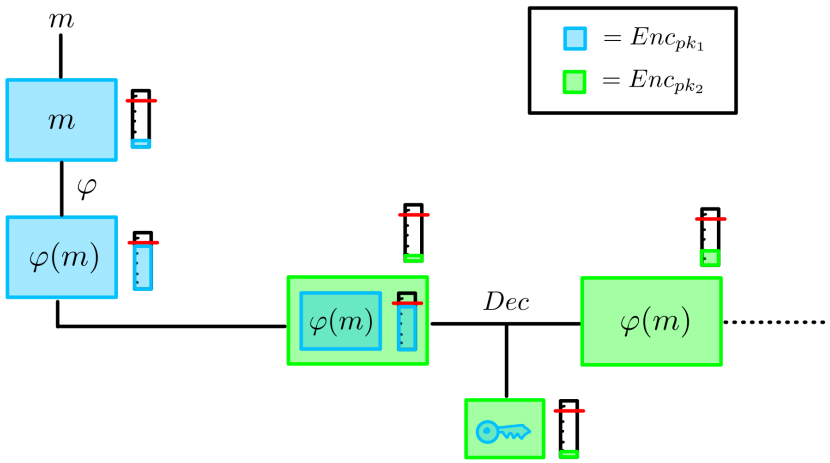

Figure 2.4: A simplified representation of Gentry's bootstrapping procedure [12]

At first, the plaintext message m is encrypted with public key pk_1, giving a ciphertext with small noise, represented by the first blue box and the noise indicator next to it. After performing the homomorphic operation φ, the noise has increased to an extent that does not allow for further operations. Hence, the whole ciphertext is encrypted again with a new public key pk_2. The private key of the first encryption is also encrypted with the second public key.
Within this new encrypted environment, the noisy ciphertext can be decrypted, resulting in an encryption of $\varphi(m)$ with reduced noise. Like that, arbitrarily many homomorphic operations are possible, transforming a former SWHE scheme into an FHE scheme.

Leveled homomorphic encryption

In leveled homomorphic encryption, arbitrary logic gates can be evaluated, but the total depth of the evaluation circuit is predetermined respectively limited [8]. Often, leveled encryption schemes are less computationally costly than other encryption methods, making them suitable for many practical applications.

2.2 Related Work

An encryption scheme is said to be fully homomorphic if it allows an unlimited amount of additions and multiplications on encrypted data. In theory, all higher-level operations can be built upon these two operations, but often these further operations are not implemented in real-world encryption schemes. The comparison operation is one of these examples that rarely exist in the bare encryption schemes. All comparison operations can be built from the less-than and the equality function.

For a totally ordered set S with binary relation $<$, the less-than and equality functions can be defined as follows for $x, y \in S$ [24]:

$$LT_S(x, y) = \begin{cases} 1 & \text{if } x < y; \\ 0 & \text{if } x \geq y; \end{cases}$$

$$EQ_S(x, y) = \begin{cases} 1 & \text{if } x = y; \\ 0 & \text{if } x \neq y; \end{cases}$$

To get a less-than-or-equal function from this, the logical OR operation between these two function is computed. To get the corresponding greater-than respectively greather-than-or-equal functions, the positions of x and y are switched.

2.2.1 Methods for computing the comparison operation

Boolean circuits

An elegant solution for FHE comparison is the comparison via boolean circuits, such as outlined in [9]. This solution works with bitwise encryption schemes such as TFHE and FHEW, and only for integers represented using the two's complement. In this approach, existing FHE submodules are used to construct homomorphic comparison: FHE subtraction, FHE bit inversion and FHE equality check. All of these can be constructed from basic FHE arithmetic operations. For example, FHE subtraction is possible via the addition of the minuend with the two's complement of the subtrahend, figure 2.5.

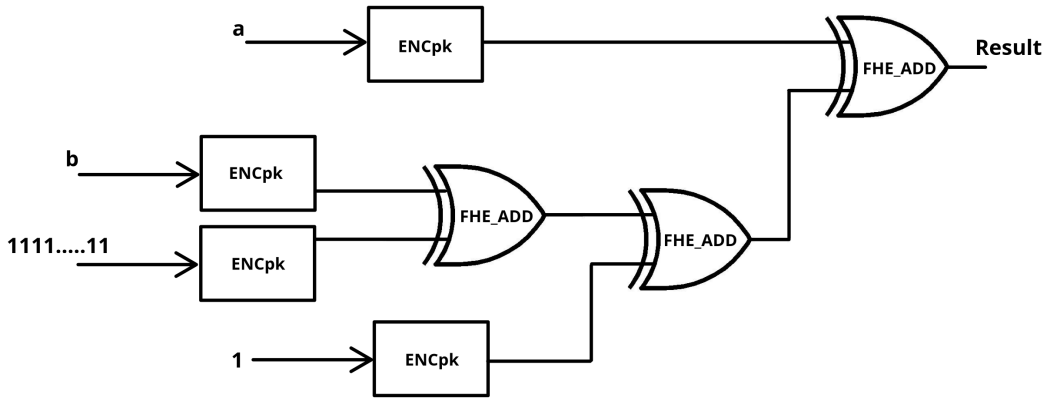

Figure 2.5: Boolean circuit for homomorphic subtraction [9]

This subtraction circuit can then be used to construct an FHE_isgreater circuit thanks to the fundamental properties of the two's complement: In this representation, all negative numbers start with a 1 in the most significant bit (MSB). Hence, it is sufficient to look at the most significant bit of the homomorphic subtraction between two numbers in order to

be able to compare them: If the first number was larger than the second, the difference between them will be positive, otherwise negative [9]. Since this approach only works for integers represented in the two's complement and needs a way of extracting the most significant bit from a number, other methods exist for different use cases.

Divide and Conquer

Computing the comparison operation on the bit-level as well, a divide-and-conquer approach was presented in [16, 50]. Since access to single bits is needed, this approach will work in the bitwise schemes (TFHE, FHEW). The following expressions were used to compare two single bits x and y:

$$x > y \Leftrightarrow xy + x = 1$$
$$x = y \Leftrightarrow x + y + 1 = 1$$

This idea for single bits was extended to n-bit integers with the following auxiliary functions:

- $t_{i,j}$ corresponds to the evaluation of the expression $\overline{x_{i+j-1} \cdots x_i} > \overline{y_{i+j-1} \cdots y_i}$
- $z_{i,j}$ corresponds to the evaluation of the expression $\overline{x_{i+j-1} \cdots x_i} = \overline{y_{i+j-1} \cdots y_i}$

where the two functions were defined as follows (choosing $l = \lceil j/2 \rceil$ in each iteration):

$$t_{i,j} = \begin{cases} x_i y_i + x_i & j = 1 \\ t_{i+l,j-l} + z_{i+l,j-l} & j > 1 \end{cases}$$

$$z_{i,j} = \begin{cases} x_i + y_i + 1 & j = 1 \\ z_{i+l,j-l} z_{i,l} & j > 1 \end{cases}$$

This corresponds to the following divide-and-conquer scheme: The whole bit sequence is repeatedly divided into two parts. Then, we check if either the MSB part (left part) of one number is larger than the MSB part of the other, or the MSB parts are equal and the least significant bit part (right part) of one number is larger than the LSB part of the other. In formulas, this looks as follows:

$$\overbrace{x_{n-1} \cdots x_l}^{msb(X)} \overbrace{x_{l-1} \cdots x_0}^{lsb(X)} > \overbrace{y_{n-1} \cdots y_l}^{msb(Y)} \overbrace{y_{l-1} \cdots y_0}^{lsb(Y)} \Leftrightarrow$$

$$(msb(X) > msb(Y)) \vee ((msb(X) = msb(Y)) \wedge (lsb(X) > lsb(Y)))$$

Deterministic finite automaton

Another interesting approach for bitwise schemes is the use of a deterministic finite automaton (DFA), a concept commonly known from theoretical computer science, to compute the comparison operation [13].

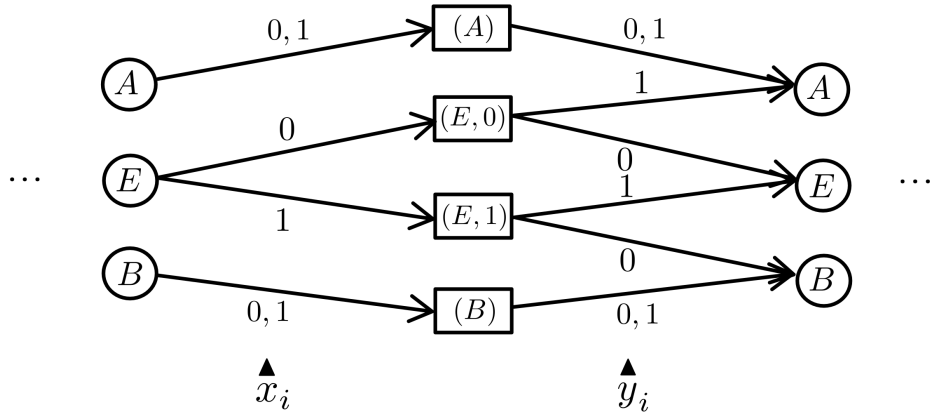

Figure 2.6: Deterministic finite automaton for comparing two numbers[13]

In this DFA, the bit sequences of the two numbers x and y are read from the most significant bit to the least significant bit and start in state E, which means the two numbers are equivalent. As long as we read the same bit from x and y, we end up in state E again. As soon as we read a 0 from x and a 1 from y, we end up in state A, which means that y is larger than x. We remain in this state when we have reached it since only bits of lower significance are read thereafter. When reading a 1 from x and a 0 from y, we end up in state B, which is opposite to A, and represents $x > y$.

Word-wise comparison

In word-wise encryption schemes, it is not possible to use techniques such as looking at the most significant bit (MSB), since operations are not carried out on every individual bit [24]. In these cases, mathematical theorems over finite fields can be used, for example Euler's theorem in the case of the equality operation.
For a positive integer m, Euler's Theorem (ET) states the following:

$$x^{\varphi(m)} \equiv 1 \bmod m$$

for any integer x coprime to m [14]. Two integers are coprime if 1 is their only common divisor. Euler's totient function $\varphi(m)$ represents the number of integers coprime to m. In the case where m is a prime number, Euler's theorem can be reformulated as [14]

$$x^{m-1} \equiv 1 \bmod m$$

Using this theorem, [24] formulated the following equality function:

$$\mathrm{EQ}_S(x, y) = 1 - (x - y)^{p-1}$$

This function will be 1 if $x = y$, and 0 otherwise since $(x - y)^{p-1}$ behaves according to Euler's theorem.

A similar but mathematically more elaborate approach can be used to interpolate the less-than function. Generally, the comparison operation on word-wise encryption performs worse than the bit-wise techniques [32].

Numerical approximations

[11] used an iterative numerical approximation to calculate the comparison operation. The advantage of this approach is the very large speed on word-wise encryption schemes, where the comparison operation was quite expensive to compute before. A disadvantage is that a small error arises due to the approximate character of the method.

For example, the sign function can be regarded as a simple step function that is equal to zero for negative numbers and equal to one for positive numbers:

$$\chi_{(0,\infty)}(x) := \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

This function is difficult to implement in the encrypted domain due to its discontinuity. However, it can easily be approximated by sigmoid functions as shown in figure 2.7.

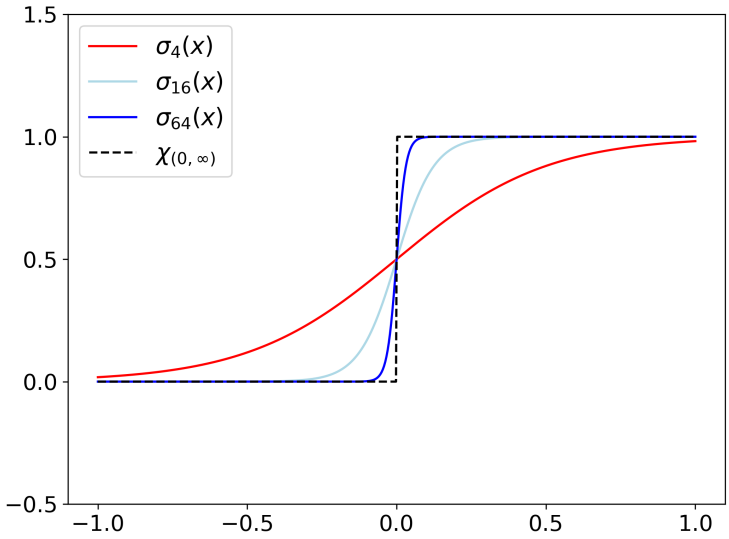

Figure 2.7: Approximation of the step function $\chi_{0,\infty}$ by sigmoid functions [11]

In order for this function to be evaluated homomorphically and used for the comparison operation, the numbers to be compared must be subtracted and scaled. Then the

comparison operation can be approximated by the following function:

$$\text{comp}(a,b) \approx \sigma_k(\log a - \log b) = \frac{e^{k \log a}}{e^{k \log a} + e^{k \log b}} = \frac{a^k}{a^k + b^k}$$

In later work, this concept of numerical approximation was further refined [10]. By using composite polynomials instead of the approximate equality above, the computational complexity of the evaluation can be further reduced.

2.2.2 Encryption libraries compatible with comparison computation

TFHE-rs

TFHE-rs [55] is an implementation of the TFHE scheme (Fully Homomorphic Encryption over the Torus), which operates on the bit-level, encrypting each bit separately. As the name suggests, the library is written in the Rust programming language. This implementation allows for comparison operations out-of-the-box with 64-bit integers [56].
This out-of-the-box implementation is based on an approach similar to the boolean circuit approach presented in [9], but with additional overflow handling [5]. To check for a potential overflow in subtraction, the input and output borrow to the last bit are compared - if they differ, an overflow has occured. By computing the XOR operation between the overflow flag and the sign, the final `less-than` comparison operation can be evaluated. This shall be illustrated by the following four-bit number subtraction scenario, figure 2.8. Important to note is the fact that four-bit signed integers range from the values -8 to 7 in the two's complement representation.

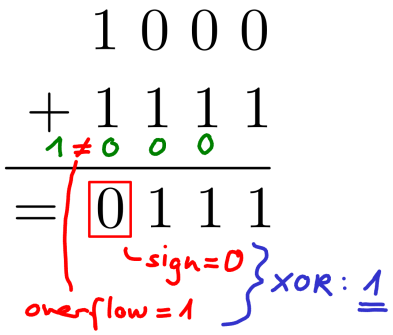

Figure 2.8: Comparison operation with overflow detection used in TFHE-rs implementation

In this example, the numbers -8 and 1 are compared. To be exact, we want to evaluate the operation

$$-8 < 1 \Rightarrow true$$

To evaluate this expression, we subtract the two values, respectively add the negation of 1 (-1 = 1111 in two's complement) to -8 (1000 in two's complement). Since the input

carry and the output carry of this addition to the last bit are not equal, an overflow has occured and the overflow flag is equal to 1. This also makes sense intuitively because -9 is out of range for a four-bit number. The sign bit obtained in the calculation is 0. By computing the XOR operation between these two, the final result of 1 (*true*) is obtained, confirming that -8 < 1.

A speciality of the TFHE scheme is the so-called Programmable Bootstrap (PBS) operation, which is extensively used in the algorithms of TFHE-rs [5]. This operation, like bootstrapping, reduces the noise in ciphertexts to a fixed level. In addition to that however, the PBS operation also allows to evaluate a lookup table on the ciphertext, which is equivalent to applying a univariate or bivariate function to it.

The developers of TFHE-rs also used parallel programming to make the encryption library more efficient. The bits of the ciphertexts are grouped together, and operations evaluated on all the groups in parallel. After that, only operations that affect multiple groups have to be computed (e.g. group carries).

HElib

[20] described in detail the design and implementation of the open source (Apache License v2.0) library `HElib` [21], which provides the BGV, BFV and CKKS schemes as an API. HElib is written in C++ and provides a large number of low-level routines as well as automatic noise management. There are also a number of examples that can easily be run for testing purposes and to get a better understanding of the library.

Together with the paper, [24] provided a GitHub repository which implements comparison circuits. The library allows for automatic tests demonstrating the functionalities and can also compute sorting functions and the minimum of an array. Once HElib and the library are installed, the circuit can be started via a terminal by specifying some cryptographic parameters as well as the number of experiments carried out:

```
./comparison_circuit circuit_type p d m q l runs print_debug_info
```

OpenFHE

OpenFHE [3] is an open-source fully homomorphic encryption library that has a large functional range and includes design ideas from various other (prior) FHE projects. For example, it includes ideas of Microsoft SEAL and HElib as well as several new concepts. Since it supports boolean circuits, it is possibly suitable for the implementation of a comparison operation.

FHE Transpilers and Compilers

Various examples of FHE transpilers and compilers exist. Concrete [54] is an open source software framework that targets easy use of FHE without deeper knowledge of the mechanisms behind it. Concrete is based on the TFHE scheme, like TFHE-rs. The software is written in python and can be installed via PyPI or using the Docker image provided by ZAMA. The framework can transform usual python functions to the encrypted domain with a compiler dedicated to do exactly this. A simple decorator can be added to a function to compile it. Of course, the library has some limitations to it, e.g. the size of integers is bounded.

Furthermore, Google presented a general purpose transpiler project [19] in 2021. It allows to convert high-level code that works on unencrypted data into high-level code that works on encrypted data. This potentially allows for the implementation of functions like the comparison operation without in-depth knowledge on cryptography.

2.3 Problem statement

A comparison operation is needed to make homomorphic encryption a suitable option for the use in a decentralized identity application. Specifically, such an operation is necessary to securely verify the issuance date of a credential against a threshold. Even though implementations of the comparison operation in homomorphic encryption exist, there is no such implementation available in JavaScript. Prior to this work, there was hence no possibility of extending the existing decentralized identity application prototype [40] with a comparison operation. This is the gap this work attempts to fill by implementing an encrypted comparison operation and integrating it into the prototype using an appropriate protocol.

Chapter 3

Design

In this chapter, the fourth thesis goal is addressed:

- Design of a prototype that effectively uses the comparison operation in a decentralized identity system

Section 3.1 discusses different protocols thinkable for integrating the homomorphic comparison operation into the existing prototype. In section 3.2, the choice of the encryption scheme TFHE/FHEW is motivated. Different ways of porting this encryption scheme to JavaScript are discussed in section 3.3 and the choice of TFHE-rs along with NAPI-RS justified. The concept of serialization, which plays an important role in the creation of an addon via NAPI-RS, is discussed in section 3.4. Since TFHE-rs provides the homomorphic comparison operation out-of-the-box, no further design choices are necessary. However, an experimental self-implemented version of the comparison operation will be developed in chapter 4, to gain a deeper understanding of the challenges that come with the details of such an implementation.

3.1 Protocol for Decentralized Identity Application

3.1.1 Existing protocol

This thesis can be considered follow-up work based on Schumm et al. [43]. In their work, two prototypes for a decentralized identity application were built, focusing on preserving the privacy of the issuance date of a credential. While one prototype uses Zero-Knowledge Proof, the other is based on homomorphic encryption. Both prototypes have two parties involved: a prover (e.g. a student) who holds a unix timestamp of the issuance date, and a verifier who holds a unix timestamp of a threshold date. The goal is to compare the two timestamps without revealing the value held by each party to the other party.

In the HE prototype, the current implementation uses a trick due to the missing comparison operation, see figure 3.1: The two timestamps are subtracted homomorphically by the prover and then multiplied by a random floating point number to hide the absolute value of the difference. The result is sent back to the verifier, who decrypts the data and compares it with zero.

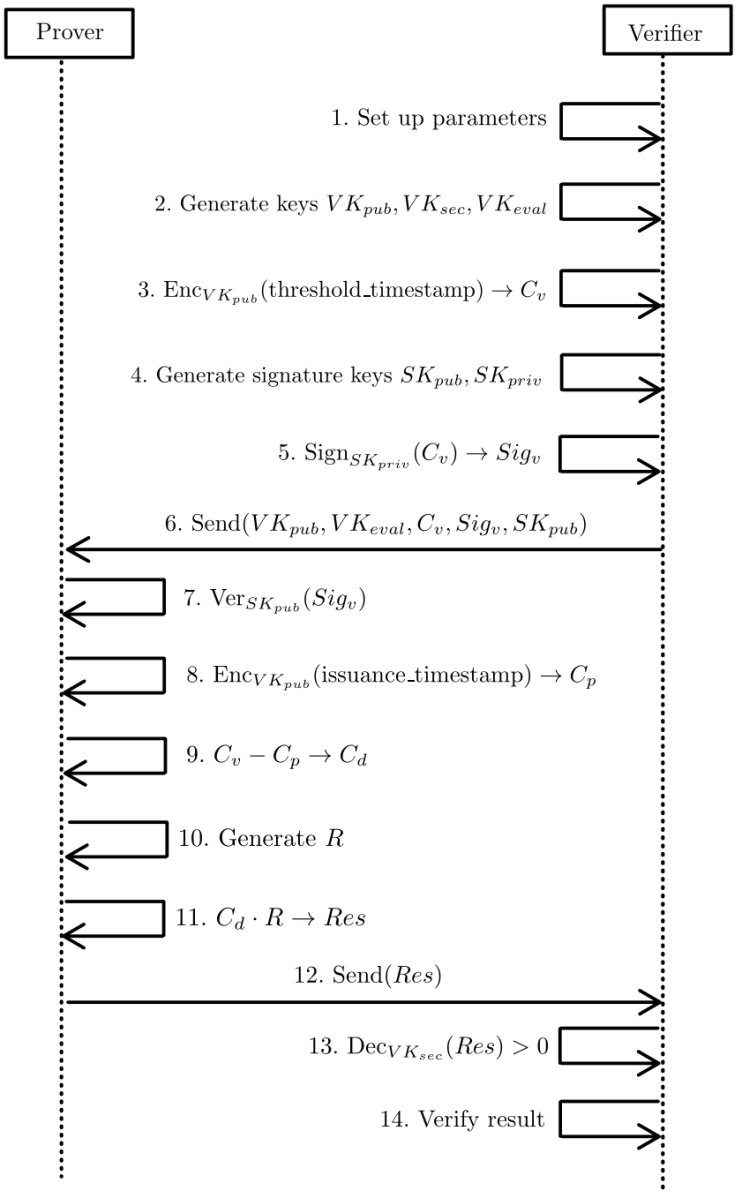

Figure 3.1: Sequence diagram of the original HE prototype [43]

This is a valid solution in principle, but it has a significant flaw. The generated random float, even though it is a "secure" random float, has a known statistical property: its expected value. In the generator, a random float between zero and one is multiplied by a randomly chosen magnitude in {1, 0.1, 0.01, 0.001}. The expected value $\mathbb{E}(R)$ of the generated number is hence

$$\frac{1}{4} \cdot (0.5 + 0.05 + 0.005 + 0.0005) \approx 0.138875$$

The result sent to the verifier is calculated by (threshold_timestamp - issuance_timestamp)$\cdot R$, hence issuance_timestamp = threshold_timestamp $- \frac{Res}{R}$. The exact value of R in each iteration is unknown, but by averaging the results of multiple queries, the verifier can hence derive the approximate value of the issuance date as follows:

$$\text{issuance\_timestamp} \approx \text{threshold\_timestamp} - \frac{mean(Res)}{\mathbb{E}(R)}$$

This was tested with a python script containing a list of 800 results. With an original issuance timestamp of 14/07/2017, this method found a timestamp on 01/06/2017, which is already quite close to the original value and shows the security impact.

3.1.2 Protocol using the comparison operation

From the previous section we can deduce that a protocol using the homomorphic comparison operation not only involves less steps, shown in figure 3.2, but also is more secure from the prover's perspective.

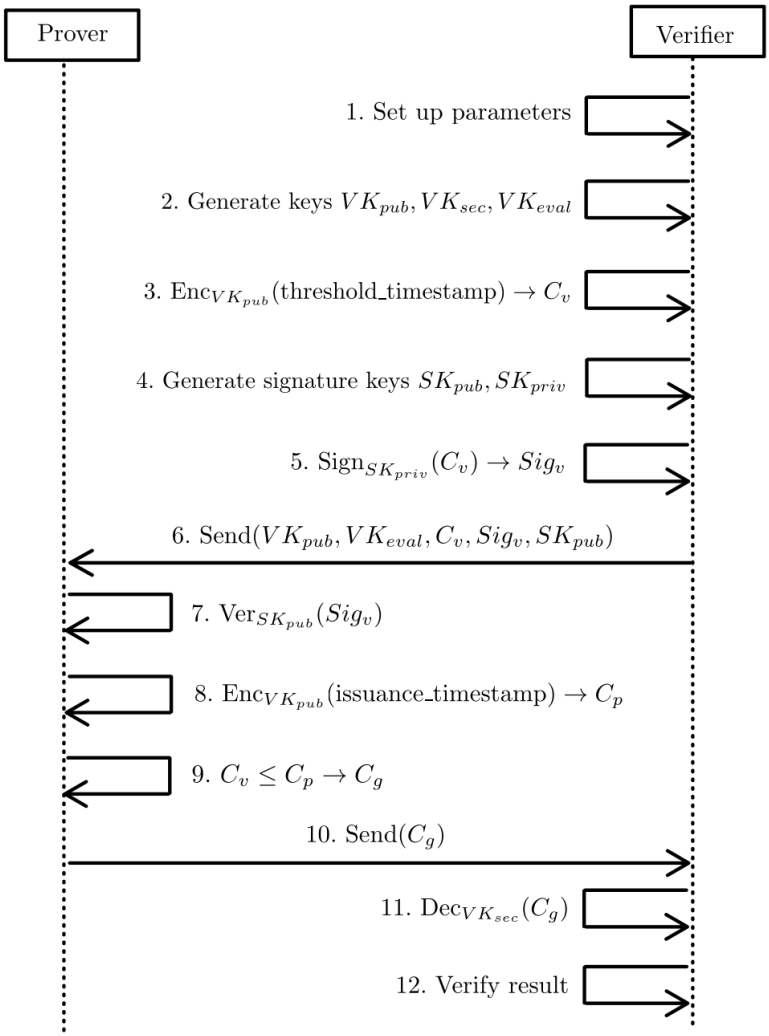

Figure 3.2: Adapted sequence diagram with comparison operation

In this adapted approach, the prover simply compares the two encrypted timestamps homomorphically using the less-than-or-equal operation and stores the result in the ciphertext C_g. This encrypted result is sent back to the verifier. The verifier can just decrypt the received ciphertext using the private key and gets the result of the less-than-or-equal operation (1 if $C_v \leq C_p$ and 0 otherwise). The less-than-or-equal operation was chosen due to the following reasoning: In the existing implementation, the inequality

$$\text{threshold\_timestamp - issuance\_timestamp} \leq 0$$

is computed after decryption. If the inequality is true, the issuance timestamp is valid, otherwise the issuance timestamp is invalid. By rearranging the terms, the correct computation using the homomorphic comparison operation can be found:

$$\text{threshold\_timestamp} \leq \text{issuance\_timestamp}$$

Intuitively, the issuance timestamp should hence be a date equal to or later than the threshold timestamp.

3.1.3 Security considerations

An important aspect to consider about fully homomorphic encryption schemes is the one of underlying security assumptions, [51]. In the classical scenario with a client and a server where the client requests some encrypted computation from the server, the assumption of a honest-but-curious server is often made: The server would be interested in the results obtained, but computes the correct homomorphic operation demanded by the client. This is an assumption sufficient for regulatory compliance in many cases, but there are scenarios where an actively malicious or compromised server has to be assumed. For this case, various techniques exist to ensure integrity of calculations, e.g. Message Authentication Codes, Zero-Knowledge Proofs and TEE attestation[51].

In the scenario of a decentralized identity application, more questions arise from a security perspective. The following possible manipulations were identified in both schemes shown above:

1. The prover encrypts a random positive number (resp. +1 in case of the comparison operation scheme) and sends this to the verifier without doing a computation
2. The prover encrypts a date surely complying with the verifiers demands instead of the actual issuance date
3. The verifier sends multiple requests with different threshold dates to the prover to find the issuance date with e.g. a binary search algorithm

The first vulnerability can possibly be mitigated by the use of verifiable homomorphic encryption as described for the server/client scenario. With verifiable homomorphic encryption, the verifier knows that the prover actually performed the computation and the result was not just chosen by the prover.

The second vulnerability is more tricky: Since the date is encrypted, it is not trivial for the verifier to know whether the date was issued by the issuer or just created by the prover themselves.

3.1.4 Role-switched protocol

To render such verification possible, an alternative role-switched protocol is thinkable, figure 3.3:

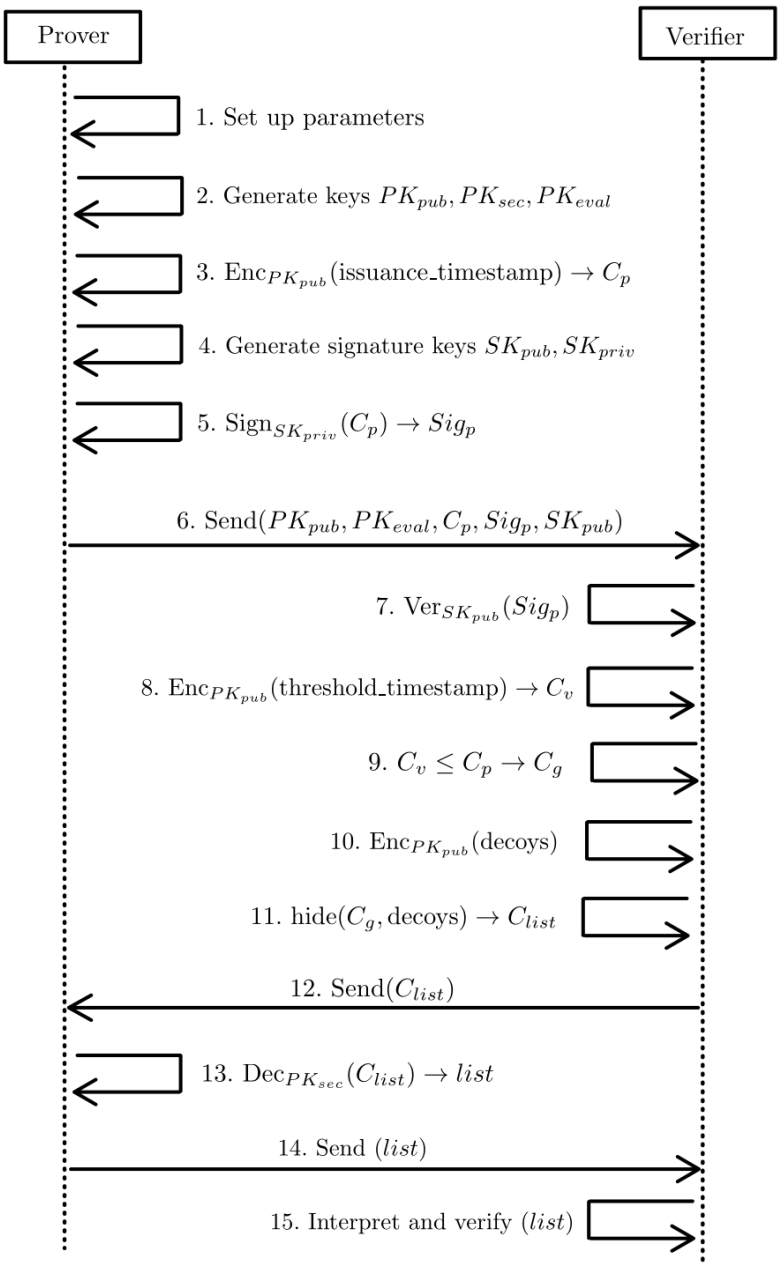

Figure 3.3: Alternative Sequence Diagram for the HE Prototype

In this alternative protocol, the encryption of the issuance timestamp could also be executed by the issuer, who can digitally sign the encrypted issuance timestamp and hence create a proof of its validity verifiable by the verifier. Since the prover decrypts the data in this protocol and has some interest in influencing the result, the verifier "hides" the actual computation result at a random position among decoy results known to the verifier. When the prover sends the decrypted result back to the verifier, the verifier can check with the help of the known decoys whether the decryption was likely executed in the correct way or not.

Hiding the actual computation among $n-1$ decoys still gives the prover a high chance $(\frac{1}{n})$ of guessing the correct position and manipulating that bit to their advantage. Hence, a more rigid hiding strategy was developed:

- Exactly one half of the list elements are decoys
- The other half of the list elements are computation results, whereof ...
- ... on average $\frac{1}{2}$ of the bits are flipped

This principle is illustrated by the following toy example, figure 3.4:

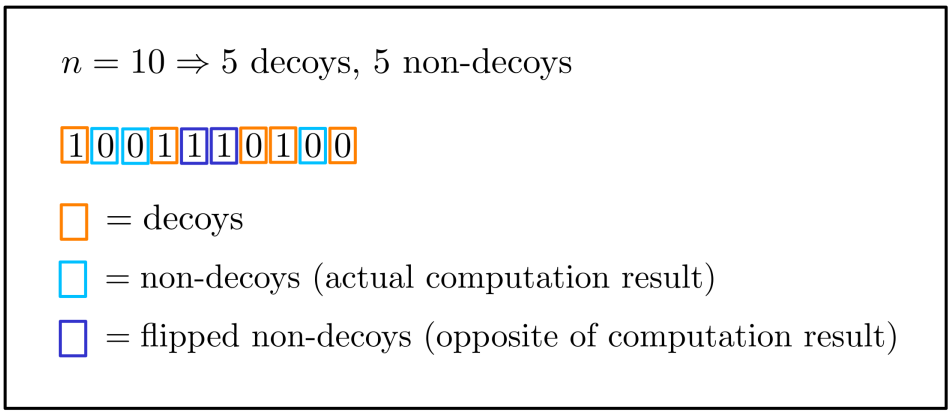

Figure 3.4: Example of decoy/non-decoy selection for $n = 10$

In the decoy/non-decoy selection, *exactly* one half of the elements are assigned to each group instead of taking a random proportion. This decision is based on the need for at least one computation result in the list, which cannot be guaranteed in conventional random distributions. A more elaborate design might randomly choose the proportion with a guarantee for at least one version of the computation result in the list. With the described version, there are $\binom{n}{n/2}$ possibilities of assigning the n positions to either group. Hence, the probability of finding the used decoy/non-decoy configuration with random guessing is $\frac{1}{\binom{n}{n/2}}$, which is already a large improvement compared to $\frac{1}{n}$. With the additional random flipping of computation result bits, the final probability of finding the used configuration becomes

$$\frac{1}{\binom{n}{n/2}} \cdot \left(\frac{1}{2}\right)^{n/2} \tag{3.1}$$

However, the prover can also randomly choose a bit sequence in the hope of finding the one in their favor, with a probability of success of $(\frac{1}{2})^n$. Since this probability is larger than the one in expression 3.1, n should be chosen large enough to make $(\frac{1}{2})^n$ negligibly small.

In TFHE-rs versions before v1.0, sharing the results of decrypted computations was not secure since this could leak the secret encryption key, as mentioned in the security section of the documentation [55]. With version 1.0 the security model was however updated, making the risk of such leakage negligible. In technical terms, the newer versions are secure in the IND-CPA$^{\mathrm{D}}$ (indistinguishability under chosen plaintext attacks with decryption oracle) model [27]. Therefore, the prover can safely share the decrypted decoys and computation result with the verifier, assuming some ordinary end-to-end encryption for this data transfer.

In the third possible manipulation, the verifier tries to find out the issuance date by sending multiple requests to the prover. This could potentially be solved by restricting the number of requests the verifier can send. Apart from this solution, the problem is difficult to solve due to the variety of threshold dates in our scenario. In the seemingly analogous scenario of age verification, the threshold typically is (current date - 18 years). With only this one threshold date, it is possible to restrict the dates for which the verifier can send a query. In the scenario of document issuance dates, thresholds can vary widely between different verifiers, making this approach impractical.

In the protocol shown in figure 3.3, another manipulation related to the third point is thinkable. Instead of encrypting decoys, the verifier could compute further comparison operations with the prover's encrypted issuance date and let the prover decrypt all of these comparisons. Then, the verifier can also draw conclusions about the secret value of the issuance date.

The decision between the two protocol types (prover vs. verifier doing homomorphic computations) is dependent on the context. In a typical scenario, the verifier doing computations however has some advantages. It allows to make the encrypted issuance date verifiable, which usually is a core requirement in decentralized identity applications. The verifier gets a statistical guarantee that the prover actually has a valid issuance date. If the verifier is honest-but-curious, they will not use the malicious method of computing results for different thresholds to infer information about the issuance date. However, the role-switched protocol developed creates a lot of overhead due to additional computations and decoys needed. The version where the prover does comparison computations is more problematic from a security perspective. By giving the prover the possibility to influence the result to their liking, the verifier cannot gain a lot of value from the given information.

3.1.5 Hypothetical protocol using multiparty homomorphic encryption

A possible solution to the problems of both the original and the role-switched protocol is multiparty homomorphic encryption resp. Homomorphic Proxy Reencryption [28]. In this method, two users can encrypt data with their respective private key, and a so-called

Proxy in between can transform ciphertexts between the two encryption domains and perform homomorphically encrypted computations. With an encryption scheme supporting Homomorphic Proxy Reencryption, the prover and the verifier could send the issuance and the threshold date to some Proxy, who evaluates the comparison operation. Then, the Proxy could send the result back to the verifier, encrypted with the verifier's private key.

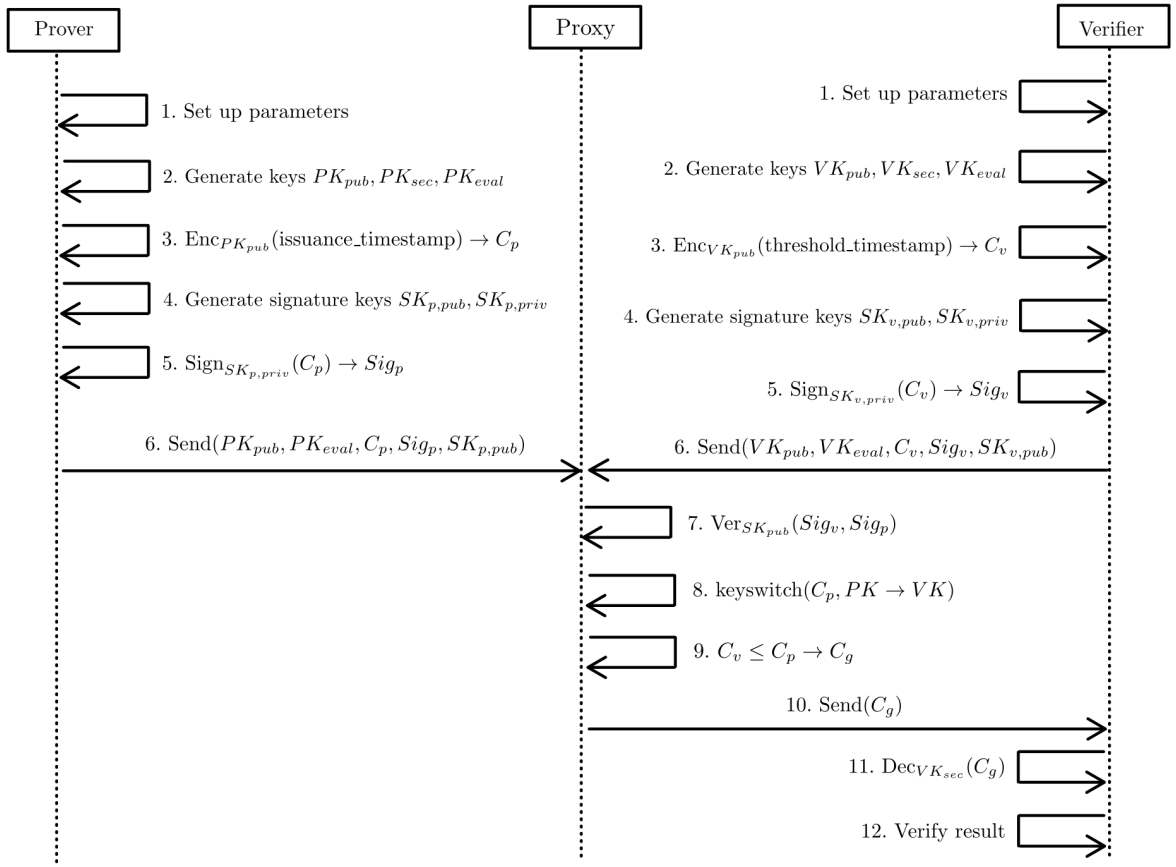

Figure 3.5: Hypothetical protocol using Homomorphic Proxy Reencryption

To ensure authenticity of the issuance date, the prover could also first send the encrypted issuance date to the verifier, who may check its authenticity with a digital signature before sending both dates to the Proxy.

3.2 Encryption scheme

For the purpose of this work, the FHEW/TFHE encryption scheme will be used. This is easily justifiable: Since two integers have to be compared with each other, the CKKS scheme with focus on floating-point numbers and approximate arithmetic is not best suited for this problem. Furthermore, research on the comparison operation in the BGV/BFV scheme is still quite experimental with only very few, small implementations [24]. Also,

while additions and multiplications are very fast in BGV/BFV, the comparison operation is less efficient [32] since this is a word-wise encryption approach. FHEW/TFHE is the most suitable homomorphic encryption scheme: Thanks to bitwise encryption, boolean circuit techniques can be used for the comparison operation, which is also very performant.

The current prototype of the decentralized identity application is implemented using node-seal [4] and the CKKS scheme (due to the multiplication with the random number, which is a floating-point number). At first, it might seem most reasonable to keep using node-seal and transform the prototype using the approximative approach outlined in [11]. However, changing the encryption library is straightforward due to similar interfaces, and using the TFHE scheme is more reasonable since integers are compared.

3.3 Porting the TFHE/FHEW scheme to JavaScript

Theoretically, the following options exist to work with a TFHE/FHEW implementation in JavaScript:

- Using an existing implementation of the scheme in JavaScript
- Implementing the scheme from scratch
- Building an addon from an implementation existing in a different programming language
- Running an implementation existing in a different language in a child process

The four options shall be discussed in the following sections 3.3.1 to 3.3.4 before choosing the most suitable approach and its limitations in sections 3.3.5 and 3.3.6.

3.3.1 Using an existing implementation of the scheme in JavaScript

There are existing implementations of bitwise encryption schemes in JavaScript. For example, the OpenFHE project has a a WebAssembly port available on its GitHub page [25], which theoretically allows for homomorphic calculations. However, the code base of this implementation has not been updated for almost two years (except for a minor change in the README file) and does not work with the current version of OpenFHE. It is also marked as work in progress and seems to be highly experimental. The project has a small number of three contributors. Overall, it seems unsuitable for the purpose of this work.

Another implementation of a bitwise scheme is available in JavaScript: ZAMA provides an API for WebAssembly of the TFHE-rs library. Even though this library allows for key generation, encryption and decryption, it does not support FHE computations. The functionality of computations might be added in the future, and the project remains an

interesting possible option for future research. Still, the current lack of FHE computation functionality makes it unsuitable for the purpose of this work, since comparison operations have to be performed here.

3.3.2 Implementation from scratch of the scheme

Even though the basic concepts of homomorphic encryption schemes can be understood quite easily, implementing a whole scheme from scratch takes a lot of time and resources. Well-known implementations such as HElib[21] and OpenFHE[3] have thousands of commits on their repositories and dozens of contributors, which would go beyond the scope of this work. To make sure an implementation complies with security standards, extensive review of the code is needed. Also, an implementation of the whole scheme in JavaScript would most probably suffer from performance deficits compared to implementations in lower-level languages. Performance is a key requirement for cryptographic libraries, which is also the reason why most such libraries are written in languages like C, C++ or Rust.

3.3.3 Building an addon from a different programming language

Building an addon using Node-API

Node-API [34] is an API that allows to build native addons for Node.js. This means that e.g. C++ code can be compiled in a way that the functions can be called and data structures can be used in Node.js. The addons are stored in the form of a binary file with `.node` file extension. Especially noteworthy is the Application Binary Interface (ABI) stability of this API. This means it is much more stable over version changes than previous similar interfaces. Node-API can be used with CMake.js, a build tool based on the CMake build system. There are various homomorphic encryption libraries written in C++ that are built using CMake (Microsoft SEAL, HElib, TFHE, OpenFHE). This makes the combination of Node-API and CMake.js especially suitable for porting certain functionalities of such libraries to Node.js. Node-API provides an extensive documentation.

Building an addon using NAPI-RS

NAPI-RS [31] is a framework very similar to Node-API, but designed for building addons from the Rust programming language. Since Rust is a highly popular language that also provides memory safety (preventing undefined behaviour on memory access), NAPI-RS would also be a suitable tool for building an addon. The documentation of NAPI-RS provides many details and examples on how to use the framework. In this thesis specifically, it could be used to build an addon from TFHE-rs [55], which is a pure Rust implementation of TFHE.

Building an addon using a Foreign Function Interface

In addition to Node-API and NAPI-RS, various other Foreign Function Interfaces (FFIs) exist for Node.js, among them the packages node-ffi, ffi, ffi-napi and ffi-rs [39]. The packages, similarly to Node-API and NAPI-RS, provide a way of making code written in another programming language available to JavaScript. Many of the mentioned packages have however not been updated for multiple years and are mostly maintained by a small group of developers. Also, they often do not provide in-depth documentation. The interfaces are sometimes based on Node-API, adding overhead to the already sufficient functionality of this library.

3.3.4 Running an implementation from a different language in a child process

Another way of using an existing library from a language like C++ or Rust is the use of the Node.js child processes module [33], which among other functionalities allows us to execute shell commands from a JavaScript application. Like that, data could be encrypted and decrypted and computations performed on it by repeatedly calling the corresponding functions of the encryption scheme through the terminal. However, the code needed might differ between platforms and executing shell commands could possibly be blocked by malware protection software. Stability between different versions and operating systems can not be guaranteed. Hence, this is not the most elegant solution for the purpose of this work.

3.3.5 Most suitable approach

The most suitable approach for the purpose of implementing a homomorphically encrypted integer comparison operation in JavaScript was evaluated to be the usage of TFHE-rs together with NAPI-RS. As mentioned before, TFHE-rs already provides an integer comparison operation built-in. It is among the most actively developed homomorphic encryption libraries at the moment, showing almost daily commits on GitHub. The library is free to use under the BSD 3-Clause Clear license for research purposes, making it suitable for a Bachelor's thesis. For commercial use, a patent license has to be purchased by Zama - however, there are currently no plans of using this project for an application inside a company.

The use of NAPI-RS can also be justified easily. NAPI-RS is used in large projects such as the Bitwarden password manager. It is actively developed and provides an extensive documentation with examples. In contrast to child processes, it works independently of the platform used and supports all important platforms. The (highly simplified) matrix of comparison in figure 3.6 summarizes the decision-making further.

	Performance	Cross-platform compatibility	Actively developed	Well-documented	Allows for HE computations
TFHE-rs WASM port	✓	✓	✓	✓	✗
OpenFHE WASM	✓	✓	✗	✗	✓
Implementation from scratch	✗	✓	✓	✓	✓
Addon with Node-API	✓	✓	✓	✓	✓
Addon with NAPI-RS	✓	✓	✓	✓	✓
Foreign function interface	✗	✓	✗	✗	✓
Child process	✗	✗	✓	✓	✓

Figure 3.6: Comparison matrix between different implementation approaches

3.3.6 Limitations

Even though the approach chosen is highly suitable for the given problem, it has some limitations that are especially noteworthy if we want to extend the usecases. The most obvious limitation is that we are restricted to integer and boolean arithmetic. Due to this, the model will soon suffer from restrictions if we want to extend its use case beyond the comparison of Unix timestamps. Furthermore, even though the TFHE-rs library is highly optimized, it still suffers from speed limitations. This will be discussed in-depth in the evaluation part, but it can already be estimated that the execution time will lie in the order of magnitude of seconds instead of milliseconds. Also, the implementation is dependent on the further development of TFHE-rs, which currently lies in the hands of Zama. Zama is a relatively young cryptography startup founded five years ago. Given the growing interest in homomorphic encryption and the presence of reputable cryptography experts like Pascal Paillier at Zama, the development of TFHE-rs will however probably not cease in the foreseeable future.

3.4 Serialization

An important concept needed for using the TFHE-rs library with NAPI-RS is serialization [55]. Serialization is the process of converting programming objects into a persistent file, for example into a binary or JSON file. Usually, objects are serialized in order to be sent over a network. Since ciphertexts and keys of a cryptosystem typically are sent over a network, most encryption schemes provide a serialization module built-in, from which TFHE-rs is no exception. Serialization is not only useful for sending data over the network, it also helps to handle data from Rust in JavaScript. Even though NAPI-RS provides advanced features such as the conversion from Rust structs to Javascript classes, this rapidly becomes highly complex. For example, converting the whole FheInt64 struct (representing a 64-bit integer ciphertext) spread over multiple Rust crates would require a

lot of handcraft. By using the serialization feature of TFHE-rs, such a complex structure is converted into a simple Buffer and can be exchanged between Rust and JavaScript out-of-the-box.
This is also how the implementation is designed: All data exchanged between the NAPI-RS addon and the JavaScript program is converted to a simple data structure supported by NAPI-RS right away.

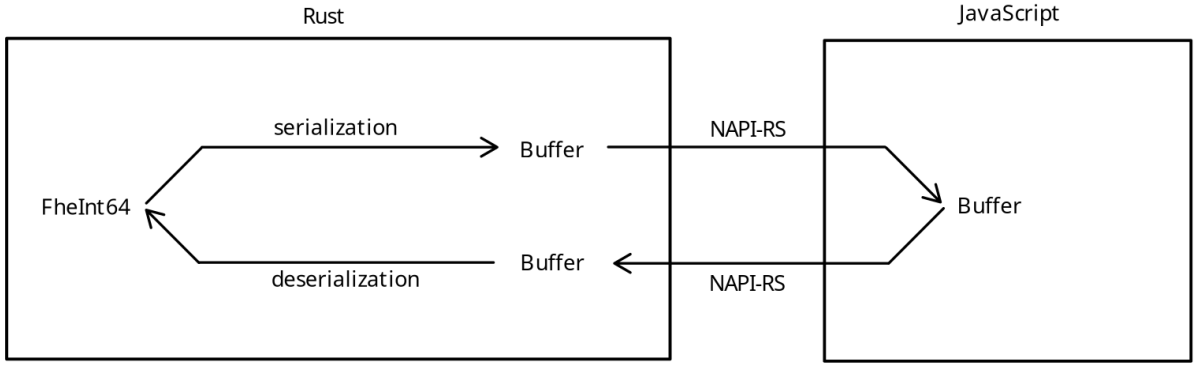

Figure 3.7: Flow of serialization and deserialization between Rust and JavaScript by the example of an encrypted 64 bit integer

Chapter 4

Implementation

To implement the comparison operation with its standalone functionalities, the repository `HE_comparison_implementation` [48] was created on GitHub. For its integration into the existing `credchain` repository [40], the `credchain` repository was forked and a new branch created on the fork [46]. This partitioning allowed for a more structured development process and makes it possible to integrate the comparison operation into different projects without having to extract it from the credchain repository first.

The implementation process was hence split into four parts. At first, the relevant parts of the TFHE-rs library [55] were made available to Node.js by the use of NAPI-RS [31], section 4.1. Secondly, all parts of the existing protocol using node-seal were moved to this TFHE-rs addon and the subtraction/multiplication with random float replaced by the comparison operation 4.2. In a third stage, the existing protocol was supplemented with the second, role-switched protocol, see section 4.3. Lastly, the comparison operation was implemented from scratch using OpenFHE [3] to gain a deeper understanding of it, section 4.4.

4.1 Making TFHE-rs functionality available to Node.js

4.1.1 Usage of NAPI-RS

The NAPI-RS library [31] allows to run Rust code from a Node.js environment. It provides an intuitive integration with Rust. The first thing needed for it to work is the package manager of Node.js: npm. Once npm is installed, NAPI-RS can be installed via the command

```
1 $ npm install -g @napi-rs/cli
```

Listing 4.1: Installation of NAPI-RS (`command line`)

It can then be added to Rust as a Cargo crate, adapting the `Cargo.toml` configuration file, where the following specifications are needed as a minimum [31]:

```
[package]
name = "tfhe_comparison"

[lib]
crate-type = ["cdylib"]

[dependencies]
napi = "2"
napi-derive = "2"

[build-dependencies]
napi-build = "1"
```

Listing 4.2: Configuration of NAPI-RS (`Cargo.toml`)

Quite self-explanatory, the `[package]` section defines the name and further information about the Rust package to be built. Importantly, the `[lib]` section needs to specify the `crate-type` as `cdylib` to ensure a dynamic system library will be produced. This `crate-type` is used when a Rust library is to be loaded from another language [49], which is the main purpose of NAPI-RS. The `[dependencies]` and `[build-dependencies]` are the sections where the NAPI-RS crate is actually specified.

In the project we want to make available in Node.js, a `build.rs` file has to be created with the following content:

```
// build.rs
extern crate napi_build;

fn main() {
  napi_build::setup();
}
```

Listing 4.3: Build script for NAPI-RS (`build.rs`)

When the crates have been imported, a function to be exposed to JavaScript can be preceded by the attribute macro `#[napi]`:

```
#[napi]
fn example() -> i64 { ... }
```

Listing 4.4: Using attribute macro for exposing function to JavaScript (`Rust`)

On the JavaScript side, the package.json file also needs to be adapted. The package name and build scripts have to be specified [31]:

```
{
  ...
  "devDependencies": {
    "@napi-rs/cli": "^1.0.0"
  },
  "napi": {
    "name": "tfhe_comparison"
  },
  "scripts": {
    "build": "napi build --release",
```

```
11      "build:debug": "napi build"
12    }
13  }
```

Listing 4.5: Node.js configuration for NAPI-RS (`package.json`)

Then, the package can be used in Node.js by simply requiring it:

```
1  const addon = require("./tfhe_comparison.node");
```

Listing 4.6: Usage of NAPI-RS addon in Javascript (e.g. `HomomorphicEncryption/verifier.js`)

Also noteworthy is the way NAPI-RS handles naming conventions. Since Rust is typically styled in `snake_case` and JavaScript in `camelCase`, NAPI-RS automatically converts between the two. For example, a function defined as `get_keys()` in Rust hence has to be called as `getKeys()` in JavaScript.

4.1.2 Usage of TFHE-rs

TFHE-rs has a very simple interface and allows to use Rust's built-in operators for most FHE computations on ciphertexts. The only exception to this are the comparison operations since these are required to return booleans in Rust, which cannot be done in FHE [55]. Hence, comparisons were implemented as follows:

```
1  let eq = a.eq(&b);
2  let ne = a.ne(&b);
3  let gt = a.gt(&b);
4  let lt = a.lt(&b);
```

Listing 4.7: Comparison operations in TFHE-rs (`Rust`)

4.1.3 Exposing TFHE-rs functions to Node.js

In order to use the functionalities of TFHE-rs in Node.js, the functions from TFHE-rs were wrapped into further functions that provide a JavaScript compatible input-output flow. This principle shall be illustrated by the following key-generation function:

```
1  #[napi]
2  fn get_keys() -> Vec<Vec<u8>> {
3      let config = ConfigBuilder::default().use_custom_parameters(...).
           build();
4      let client_key = ClientKey::generate(config);
5      let compressed_server_key = CompressedServerKey::new(&client_key)
           ;
6      let compressed_public_key = CompressedCompactPublicKey::new(&
           client_key);
7
8      let mut client_key_ser = vec![];
9      safe_serialize(&client_key, &mut client_key_ser, 1 << 30).unwrap
           ();
```

```
10      let mut compressed_server_key_ser = vec![];
11      safe_serialize(&compressed_server_key, &mut
            compressed_server_key_ser, 1 << 30).unwrap();
12      let mut compressed_public_key_ser = vec![];
13      safe_serialize(&compressed_public_key, &mut
            compressed_public_key_ser, 1 << 30).unwrap();
14
15      return vec![client_key_ser, compressed_server_key_ser,
            compressed_public_key_ser];
16  }
```

Listing 4.8: Key generation and serialization in TFHE-rs (`src/lib.rs`)

In this function, the client key, server key and public key are generated as usual after setting up the configuration. Then, the three keys are serialized using the `safe_serialize` function of TFHE-rs, which uses the Serde framework [45] with some additional checks [55]. After serialization, the three serialized keys are converted to Buffer objects (resp. `Vec<u8>` which will be discussed later) and then combined into a vector with three elements. The vector of `Vec<u8>` objects is compatible with JavaScript and hence returned from the `get_keys()` function.
In JavaScript, the function can then be called and keys separated with the following short commands:

```
1       const keys = tfhe_rs.getKeys();
2
3       const instances = {
4           clientKey: keys[0],
5           evaluator: keys[1],
6           publicKey: keys[2],
7       }
```

Listing 4.9: Key generation and their assignment to variables in JavaScript using TFHE-rs addon built with NAPI-RS (`HomomorphicEncryption/verifier.js`)

4.2 Transition of prototype to TFHE-rs addon

4.2.1 Translation between node-seal and TFHE-rs

Since the encryption scheme used is changed from node-seal to TFHE-rs, the parallels and differences between the two libraries were compared at first. In the following table 4.1, all steps of the encryption, evaluation and decryption processes are shown with the corresponding commands.

Table 4.1: Comparison between node-seal and TFHE-rs

Step	Command
Server/Evaluation key generation	node-seal: `const evaluator = seal.Evaluator(context);` TFHE-rs: `let (_, server_key) = generate_keys(config);`
Client/Secret key generation	node-seal: `const secretKey = keyGenerator.secretKey();` TFHE-rs: `let (client_key, _) = generate_keys(config);`
Public key generation	node-seal: `const publicKey = keyGenerator.createPublicKey();` TFHE-rs: `let public_key = PublicKey::new(&client_key);`
Encryption	node-seal: `const cipher = encryptor.encrypt(plain);` TFHE-rs: `let cipher = FheInt64::encrypt(plain, &client_key);`
Decryption	node-seal: `const pResult = decryptor.decrypt(value);` TFHE-rs: `let plain: bool = cipher.decrypt(&client_key);`

As can be seen, the basic syntax is very similar between the two libraries. However, the basic configuration setup is much more straightforward in TFHE-rs. With the single line

```
1 let config = ConfigBuilder::default().build();
```

Listing 4.10: Configuration creation in TFHE-rs (`Rust`)

a default configuration can be set up (however, note that the key generation and configuration setup were later refined to allow for compression and the usage of the compact public key). In node-seal, the scheme type and all its configuration parameters (security level, polymodulus degree, coefficient modulus, bit sizes) have to be set manually. Another important difference between the two is that node-seal requires the extra steps of encoding the data to the correct format before encryption and decoding it after decryption.

4.2.2 Integration with existing prototype

To integrate the TFHE-rs addon with the existing prototype [43], all Rust files were added to the project. The `package.json` was adapted as described in the NAPI-RS documentation. The files `prover.js` and `verifier.js` (and analogously `student.js` and `company.js`) were identified to be the relevant parts of the project to be adapted to the new design as a first step. In these files, all functions to be modified can be summarized in the following tables 4.2, 4.3 and 4.4:

Table 4.2: Functions in file `prover.js` and corresponding adaptions

Function	Adaptions
`generateSecureRandomFloat`	Delete function, not needed due to comparison operation.
`signatureVerify`	No change required.
`computeResult`	Remove encoder from parameters in function definition, use the addon's `greaterThan` function instead of multiplication by random float.
`proverEncodeEncrypt`	Change to `proverEncrypt`, remove encoder from parameters, use the addon's `encryptPublicKey` function instead of node-seal `encrypt`.
`proverCalculate`	Remove encoder in all places, leave unchanged otherwise.

Table 4.3: Functions in file `verifier.js` and corresponding adaptions

Function	Adaptions
`generateEncryptionKeys`	Remove encoder, replace encryptor, evaluator and decryptor of node-seal with public key, server key and private key of addon.
`generateSignatureKeys`	No change required.
`verifierSign`	No change required.
`verifierEncodeEncrypt`	Remove encoder, replace node-seal `encrypt` by addon's `encrypt` function.
`verifierDecryptDecode`	Remove decoder, replace node-seal `decrypt` by addon's `decrypt` function.
`verifierSetUp`	Remove encoder everywhere in the function.
`verifierProve`	Remove encoder everywhere in the function.

Table 4.4: Additional functions in files `student.js` and `company.js` with corresponding adaptions

Function	Adaptions
`studentMain`	Replace node-seal functions by addon and subtraction/-multiplication by random number by `greaterThan` function.
`companySetup`	Remove encoder, replace node-seal `encrypt` by addon's `encrypt` function.
`companyMain`	Remove setup procedure by simpler procedure in addon, replace node-seal `decrypt` by addon's `decrypt` function.

4.2.3 Debugging memory consumption

After implementing these changes, the existing tests in the file `test/textHE.js` were executed using the hardhat test environment. The execution of the tests was however not successful and resulted in an error message:

```
1 FATAL ERROR: invalid table size Allocation failed - JavaScript heap
    out of memory
```

Listing 4.11: Error message from first tests with adapted credchain repository

According to the documentation, the maximum memory available to Node.js can be set using the `--max-old-space-size=SIZE` option. However, even with a drastic increase to 20GB of available memory (which triggers swapping on the machine used), the error did not disappear. Hence, via a process of elimination, the key generation procedure was identified to be the most memory-intensive function of the addon. Since the keys are stored as vectors of unsigned 8-bit integers, the length of these vectors correspond to their length in bytes. The following sizes were identified for each of the three keys using the `len()` function of Rust vectors:

- client key: 24'126 bytes (ca. 24 kB)
- server key: 131'072'487 bytes (ca. 131 MB)
- public key: 2'151'679'908 bytes (ca. 2.15 GB)

Clearly, the public key has the largest impact on memory usage and has to be reduced in size. As shown in the documentation of TFHE-rs, keys as well as ciphertexts can be compressed to reduce the required storage. By using a compressed public key, its size could be reduced to 1'050'562 bytes resp. approximately 1.05 MB. With this adaption, the error was eliminated and all tests succeeded. At a later stage, the compressed public key was replaced by the compressed *compact* public key for performance reasons based on a conversation with the TFHE-rs technical support [53].

An effect related to the encountered error was observed when running the procedure of key generation, encryption, comparison and decryption multiple times without compression in a for loop. This led to an increase in memory consumption with every iteration, figure 4.1:

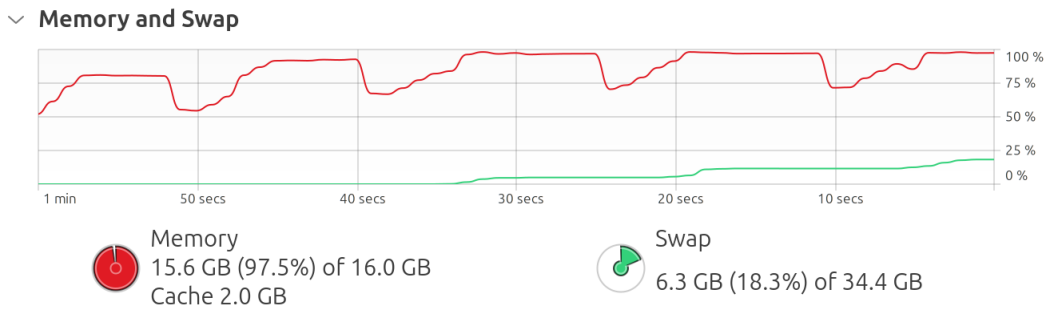

Figure 4.1: Increase in memory consumption with every iteration of the for loop

This phenomenon only occurred when using the addon in Node.js, but not when running a similar for loop in Rust itself. It was therefore concluded that an implementation detail of NAPI-RS caused this issue, possibly together with Javascript's garbage collection.

Most likely, the cryptographic keys are kept in memory even when they are not needed anymore.
To reduce the impact of this, the server key was also compressed in the implementation, reducing its size to 30'158'665 bytes (ca. 30 MB). With these adaptions, the memory consumption increase per iteration was reduced to 43MB on average over 30 iterations, which is still significant.

To fully eliminate this problem and not only reduce its impact, the data structure used to transfer data between Rust and JavaScript was investigated. Even though a Buffer takes the least space in absolute terms, the garbage collection seems not to eliminate buffer objects from memory. Hence the data structure was changed to `Vec<u8>`. In a second test run with this setup, there is a small downward dip in memory consumption every few iterations of the loop, most probably showing the effect of garbage collection, figure 4.2.

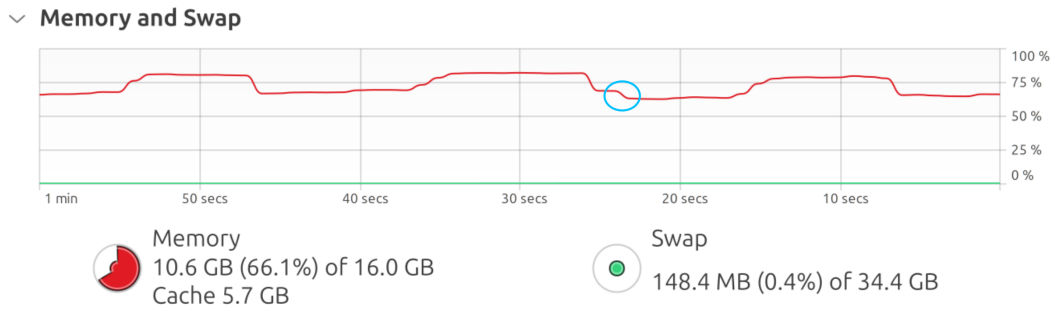

Figure 4.2: Garbage collection taking effect and keeping memory consumption constant over time

4.2.4 Redundancies in implementation and refactoring

In the existing project, some redundancies were identified. Especially, the renaming of the entities from student/company to prover/verifier that took place before this work left some inconsistencies in the code. For example, the unit testing folder contained one test file for the functionality of `student.js` and a second file for the functionality of `prover.js`, even though the two files serve the same purpose at their core. Hence, all the files were harmonized to the prover/verifier naming convention and the redundant files deleted. In the final implementation, the two files `prover.js` and `verifier.js` provide the core functionality.

4.3 Implementation of role-switched protocol

In order to implement the second protocol with switched roles between the prover and the verifier (described in section 3.1.4, especially the verifier's code needed some adaptions. In order to clearly separate the role-switched from the original protocol, a new folder `HEroleSwitched` was created in the credchain repository. There, the files `prover.js` and `verifier.js` were created to hold the adapted code. Also, the `HEperformance` file and a

test file were created to evaluate and test the role-switched protocol. In a first step, the verifier needs to be able to sample a secure random float in the range [0,1], which was implemented with the following code:

```
let random = ()=> crypto.getRandomValues(new Uint32Array(1))
    [0]/2**32;
```

Listing 4.12: Creation of a secure random float (`HEroleSwitched/verifier.js`)

Based on this random number generator, the function `verifierChoose(n,r)` was designed, which allows to choose r indices from a total of n. With this, the verifier can create decoy values. To achieve this, three arrays are initialized:

```
const decoyValueOrFlipped = [];
const cipher = [];
const computationIdxs = await verifierChoose(n, n/2);
```

Listing 4.13: Initialization of arrays needed in role-switched protocol (`HEroleSwitched/verifier.js`)

The first array `decoyValueOrFlipped` holds the following information:

- if the item in the ciphertext array at position [i] is a decoy: `decoyValueOrFlipped[i]` holds the bit value of the decoy (0 or 1)
- if the item in the ciphertext at position [i] is not a decoy: `decoyValueOrFlipped[i]` holds the information whether the computation result bit was flipped (1 if the bit was flipped, 0 otherwise)

The array `cipher` holds the actual ciphertext sent to the prover for decryption, while `computationIdxs` contains the indices where the actual computation results (non-decoys) are stored in the ciphertext array.

With this preparatory code in place, the decoy creation can be started with a for loop from 0 to $n-1$. In this for loop, two cases are considered:

- The position i contains a non-decoy $\Rightarrow$ the threshold ciphertext is encrypted and the comparison operation evaluated. The resulting encrypted bit is flipped in 50% of the cases and then pushed to the ciphertext array
- The position i contains a decoy $\Rightarrow$ a random bit is encrypted and pushed to the ciphertext array.

```
for (let i = 0; i < n; i++) {
    if (computationIdxs.includes(i)) {
        const thresholdCiphertext = await verifierEncryptPublicKey(
            thresholdPlaintext, encryptor);
        let compResult = await verifierComputeResult(evaluator,
            thresholdCiphertext, issuanceCiphertext);
        const flipBit = Boolean(Math.floor(random() * 2));
        if (flipBit) {
```

```
7                 compResult = tfhe_rs.flipBit(compResult, evaluator);
8             }
9             decoyValueOrFlipped.push(flipBit);
10            cipher.push(compResult);
11        } else {
12            const decoyPlain = Boolean(Math.floor(random() * 2));
13            const decoyCipher = await verifierEncryptPublicKey(
                  decoyPlain, encryptor);
14            decoyValueOrFlipped.push(decoyPlain);
15            cipher.push(decoyCipher);
16        }
17    }
```

Listing 4.14: Setup of ciphertext array in the role-switched protocol (`HEroleSwitched/verifier.js`)

A key point about this algorithm is the encryption of the threshold ciphertext in every iteration, which makes sure all ciphertexts are different. If the threshold was encrypted only once and reused in every iteration, all non-flipped non-decoy ciphertexts would be identical, giving an unwanted hint to a malicious prover.

Once the ciphertext array is decrypted by the prover and sent back to the verifier, verification is done analogously to the the decoy creation.

```
1  async function verifierVerify(plain, decoyValueOrFlipped,
       computationIdxs) {
2      let n = plain.length;
3      let validCount = 0, invalidCount = 0;
4      for (let i = 0; i < n; i++) {
5          if (computationIdxs.includes(i) && !decoyValueOrFlipped[i])
               {
6              plain[i] ? validCount++ : invalidCount++;
7          } else if (computationIdxs.includes(i) &&
               decoyValueOrFlipped[i]) {
8              !plain[i] ? validCount++ : invalidCount++;
9          } else {
10             assert(plain[i] === decoyValueOrFlipped[i], "tampered
                   result");
11         }
12     }
13     assert(validCount === n/2 || invalidCount === n/2, "tampered
           result");
14     console.log(validCount === n/2 ? "\tVALID Issuance Date" : "\
           tINVALID Issuance Date");
15     return validCount === n/2;
16 }
```

Listing 4.15: Verification of ciphertext array (`HEroleSwitched/verifier.js`)

In this code, all actual comparison results are used to check the issuance date for validity, while the decoys are used to make sure the prover has not tampered with the decrypted data.

Overall, the role-switched implementation works, but comes with a lot of overhead. Not

only one, but $n/2$ comparison operations and approximately n encryptions and decryptions are performed. To achieve a security level of 128 bit [6], an array of length $n = 128$ is needed, which is a highly significant overhead compared to only a single encrypted value transferred in the original protocol.

4.4 OpenFHE implementation

The TFHE-rs library provides a very convenient way of using the homomorphically encrypted comparison operation and can efficiently be exposed to Node.js via NAPI-RS. Nevertheless, the main focus of this Bachelor's thesis is the homomorphic comparison operation itself, which is already ready-to-use in TFHE-rs. To gain a deeper understanding of the challenges in implementing this comparison operation and to demonstrate the practicability of theoretical algorithms described in section 2.2, the divide-and-conquer algorithm for bitwise encryption schemes [16, 50] was translated to code using OpenFHE [3].

A few peculiarities had to be considered, the first of which are the indices used. In the papers [16, 50], the index i starts from zero, while the index j is one-based. To make this compatible with `C++` and keep the implementation comprehensible from the paper, j was kept one-based conceptually and all relevant `C++` array positions simply decreased by one. Furthermore, the binary calculations had to be translated to binary gates implemented in OpenFHE. For example, the equality

$$t_{i,j} = x_i y_i + x_i$$

for $j = 1$ corresponds to $x_i \wedge \neg y_i$. Similarly, the other expressions were translated to binary gates, according to the following table, where $\oplus$ corresponds to the exclusive OR gate:

Table 4.5: Conversions of definitions to binary gates

Definition in [16, 50]		Corresponding binary gates
$t_{i,j} = x_i y_i + x_i,$	$j = 1$	$t_{i,j} = x_i \wedge \neg y_i$
$t_{i,j} = t_{i+l,j-l} + z_{i+l,j-l} t_{i,l}$	$j > 1$	$t_{i,j} = (t_{i+l,j-l}) \oplus (z_{i+l,j-l} \wedge t_{i,l})$
$z_{i,j} = x_i + y_i + 1$	$j = 1$	$z_{i,j} = \neg(x_i \oplus y_i)$
$z_{i,j} = z_{i+l,j-l} z_{i,l}$	$j > 1$	$z_{i,j} = z_{i+l,j-l} \wedge z_{i,l}$

In the OpenFHE implementation, the first expression for example looks as follows:

```
1 t[i][j-1] = cc.EvalBinGate(AND, cipherXBits[i], cc.EvalNOT(
      cipherYBits[i]));
```

Listing 4.16: Implementation of $t_{i,j} = x_i \wedge \neg y_i$ in OpenFHE (`node-api_OpenFHE/greater_than_pke.cc`)

As can be seen from the example above, the integers x and y were implemented as arrays of encrypted bits (`cipherXBits, cipherYBits`). $t_{i,j}$ and $z_{i,j}$ are also arrays, but two-dimensional ones. To extract the final comparison result, the value `t[0][INTSIZE-1]` has

to be decrypted, where `INTSIZE` represents the size in bits of the integers to be compared. For a 64-bit integer, this is the comparison over all bits from bit 0 to 63.

To make the program more user-friendly and allow for a simple input of the decimal (base 10) number instead of an array of bits, the `std::bitset` class template was used, which allows to convert a number to an array of bits. Then, each of these bits is encrypted and pushed to the back of a vector of `LWECiphertexts`:

```
1 std::bitset<INTSIZE> plainXBits(plainX);
2 std::vector<LWECiphertext> cipherXBits;
3 for (int i=0; i<INTSIZE; i++) {
4   cipherXBits.push_back(cc1.Encrypt(cc1.GetPublicKey(), plainXBits[i
      ]));
5 }
```

Listing 4.17: Conversion of integer to array of bits and subsequent OpenFHE encryption (`node-api_OpenFHE/greater_than_pke.cc`)

To test the compatibility with Node.js, the program was also implemented as an addon via Node-API [34]. A few more steps are required in Node-API compared to NAPI-RS when exposing a function to JavaScript.

```
1  napi_value GreaterThan(napi_env env, napi_callback_info info) {
2      size_t argc = 2;
3      napi_value args[2];
4      int64_t plainX;
5      int64_t plainY;
6      int64_t greaterThanResult;
7      napi_value output;
8
9      napi_get_cb_info(env, info, &argc, args, NULL, NULL);
10
11     napi_get_value_int64(env, args[0], &plainX);
12     napi_get_value_int64(env, args[1], &plainY);
13
14     greaterThanResult = greaterThan(plainX, plainY);
15
16     napi_create_int64(env, greaterThanResult, &output);
17
18     return output;
19 }
20
21 napi_value init(napi_env env, napi_value exports) {
22     napi_value greaterThan;
23     napi_create_function(env, nullptr, 0, GreaterThan, nullptr, &
         greaterThan);
24     return greaterThan;
25 }
```

Listing 4.18: Usage of Node-API to expose C++ function to JavaScript (`node-api_OpenFHE/greater_than_pke.cc`)

Here, the number of arguments of the addon's function (`argc`), and an array storing the actual arguments (`args[2]`) from JavaScript are defined first. The three `int64_t` values are the arguments in C++ (`plainX, plainY`) respectively the variable for storing the

result of the comparison operation in C++ (`greaterThanResult`). The `output` variable is the `napi_value` returned to Node.js. After retrieving details about the environment and arguments with `napi_get_cb_info`, the arguments are read from `args` and stored in `plainX, plainY` with `napi_get_value_int64`. Then, the comparison operation is computed with the corresponding function in C++ and the result stored in the `output` variable via `napi_create_int64`.

This implementation was carried out primarily for the purpose of getting a deeper understanding of the inner workings of the homomorphic comparison operation. Since the main implementation was created with TFHE-rs and NAPI-RS, further details like Serialization of encryption keys, ciphertexts etc. were omitted in this part.

Chapter 5

Evaluation

The evaluation of this work is conducted with two main focus areas:

- Comparison between the original (node-seal) protocol and its newly implemented counterpart (TFHE-rs, without switched roles)
- Comparison between the ZKP prototype and the newly implemented protocol (TFHE-rs, without switched roles)

Furthermore, an estimate of the overhead between native TFHE-rs and the corresponding NAPI-RS addon is made. The role-switched protocol as well as the OpenFHE implementation is also be covered briefly, but in less detail than the main focus areas.

5.1 Measurement methods

5.1.1 Hardware and OS

For evaluating the performance of the new implementation, a commercially available notebook with the following specifications was used:

- CPU: Intel® Core™ i7-1360P (4×2.2 GHz and 8×1.6 GHz)
- RAM: 16GB (LPDDR5-5200)
- OS: Ubuntu 24.10 (64-bit)

The performance was evaluated in terms of CPU and memory usage as well as execution time. The `credchain` repository already provides some mechanisms for evaluation, however these were partially adapted for more accurate results.

5.1.2 CPU usage

The CPU usage measurement was left unchanged from the original `credchain` measurement method. The method uses the `pidusage` package to track the CPU usage of the current `Node.js` process. Even though there is some overhead due to the measurement environment, this gives a fairly accurate measure of the CPU usage. Important to note is the compatibility scheme of the `pidusage` package, figure 5.12.

Property	Linux	FreeBSD	NetBSD	SunOS	macOS	Win	AIX	Alpine
`cpu`	✅	❓	❓	❓	✅	ℹ️	❓	✅
`memory`	✅	❓	❓	❓	✅	✅	❓	✅
`pid`	✅	❓	❓	❓	✅	✅	❓	✅
`ctime`	✅	❓	❓	❓	✅	✅	❓	✅
`elapsed`	✅	❓	❓	❓	✅	✅	❓	✅
`timestamp`	✅	❓	❓	❓	✅	✅	❓	✅

✅ = Working ℹ️ = Not Accurate ❓ = Should Work ❌ = Not Working

Figure 5.1: Compatibility scheme of the `pidusage` package [7]

For reproducing results, a Linux or macOS system should hence be used, since CPU usage measurement is inaccurate on Microsoft Windows. Furthermore, as already mentioned in [43], CPU usage is measured in % and percentages over 100% mean that multiple cores were used in the computation.

5.1.3 Memory usage

When measuring the memory usage, a problem similar to the one appearing during implementation seemed to occur: With every iteration of the evaluation, memory consumption increased. Since the initial problem has however been resolved by changing the data structure returned by Rust functions from `Buffer` to `Vec<u8>`, it was concluded that some element of the evaluation measurement was causing the problem this time. Soon, the problem could be identified in the return statement of the `measureFunctionExecution` function:

```
1 return { cpu: Number(cpu.toFixed(2)), memory: Number((memory / 1024
    / 1024).toFixed(2)), duration: Number(duration.toFixed(2)),
    result };
```

Listing 5.1: Return statement of `measureFunctionExecution`

The last element of the returned object contains all the values returned by the function for which the metrics are measured. Later, this function is used and the results are stored

every iteration. With usual values, this will not create a lot of overhead, however the cryptographic keys of the TFHE-rs library have a significant size. Hence, the measurement environment itself has a non-negligible impact on the measured memory usage.
To mitigate this issue, the `result` value was removed from the objects stored in every iteration, since they are not needed for calculating the statistics about memory usage. After this, there is still a slight increase in memory usage in every iteration due to storing the statistics. However, broken down this does not take any significant memory space: According to the Mozilla Developer Network web docs [1], the `Number` type consists of a 64-bit value. For statistics, 9 such values are stored, which gives a total of 72 bytes per iteration. Hence, even after hundreds of iterations, these will only occupy a few kB in memory.

Storing the result during measurement will again be considered in the scalability section as in [43], since the accumulated memory consumption can serve as a measure of scalability.

5.1.4 Execution time

The execution time was measured with the `perf_hooks` package from npm, an API for performance measurements in Node.js. The measurement functionality was already present in the credchain repository. Before the execution of a function, a start mark is set, then the function is executed, and after execution an end mark is set. In code, this looks as follows [40]:

```
1 performance.mark(`${label}-start`);
2 const result = await func(...args);
3 performance.mark(`${label}-end`);
```

Listing 5.2: Measurement of execution time using `perf_hooks`

In this context, the label is a freely choosable name given by the user to the function to be evaluated. After the measurement, the duration between start and end mark is extracted as follows, using exactly this label:

```
1 const duration = performance.getEntriesByName(label)[0].duration;
```

Listing 5.3: Extraction of duration

5.1.5 Costs

The costs of the implemented decentralized identity application prototypes were evaluated by estimating the costs of storing the relevant data on the Ethereum blockchain. This is a useful estimate for real-world application, since many decentralized identity applications use blockchain technology as a distributed ledger on which public keys for signatures are stored [42].

5.1.6 Scalability

Estimating the scalability of software is a non-trivial task. In the real-world scenario, it could be said that mainly the scalability of the verifier's code has to be considered. The prover probably rarely has to prove more than a few claims in parallel. However, if e.g. a company (in the role of the verifier) gets thousands of applications to a job in the seconds after opening an online portal, the resource consumption might be more relevant. Nevertheless, both the verifier and the prover were considered in the analysis.

5.2 Comparison with original protocol

To summarize, the new prototype using TFHE-rs has similar CPU usage, higher memory consumption and a longer execution time in comparison with the original prototype.

5.2.1 CPU usage

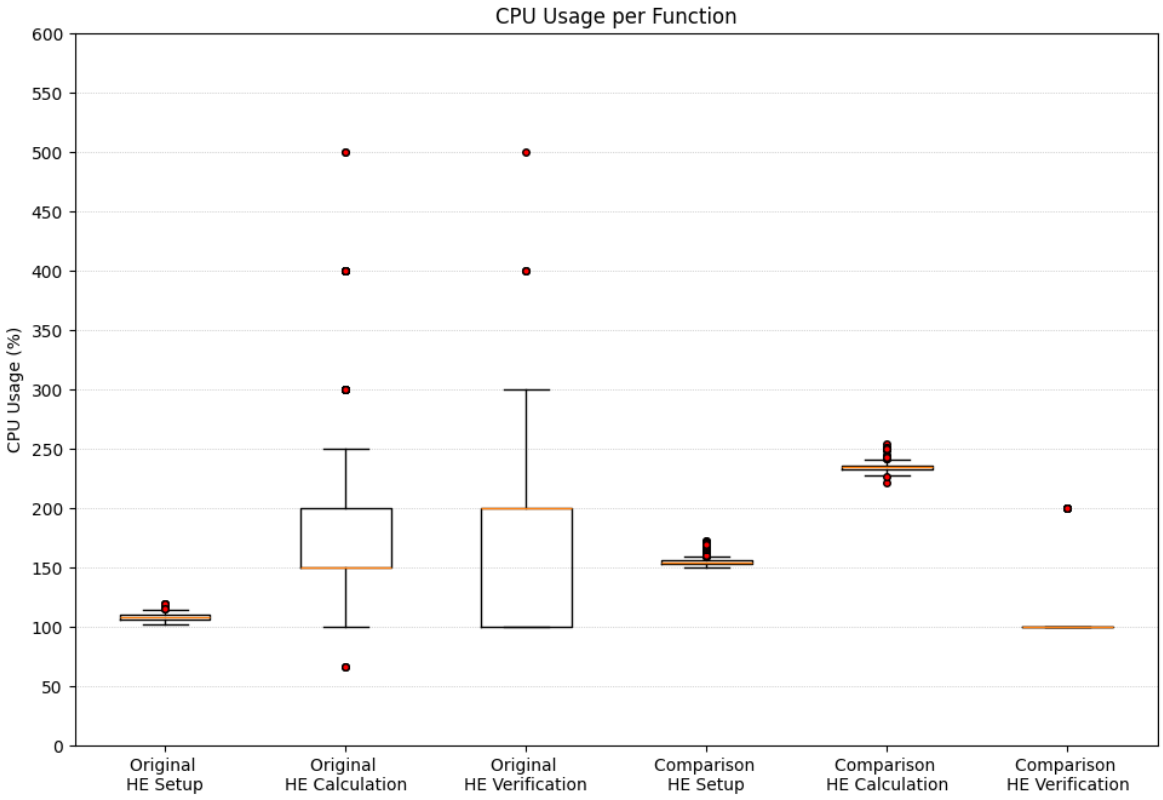

Figure 5.2: Comparison of CPU usage between original implementation and new TFHE-rs implementation

Table 5.1: CPU usage for the original and new HE functions (%)

	Orig Gen	Orig Cal	Orig Ver	Gen	Cal	Ver
Maximum	120.00	500.00	500.00	172.29	254.05	200.00
Average	108.44	183.39	177.24	156.21	234.70	117.39
Median	108.16	150.00	200.00	154.38	234.52	100.00
Minimum	102.04	66.67	100.00	150.13	221.58	100.00

From the data, similar average CPU usage is visible between the original and the TFHE-rs implementation. However, there is a wider spread of the measurements in the original implementation. This spread for the calculation and verification of the original HE prototype can possibly be explained by the short execution time, see section 5.2.3, not allowing the CPU usage to stabilize in the same way as for the setup of the original prototype as well as all three phases of the new prototype.

5.2.2 Memory usage

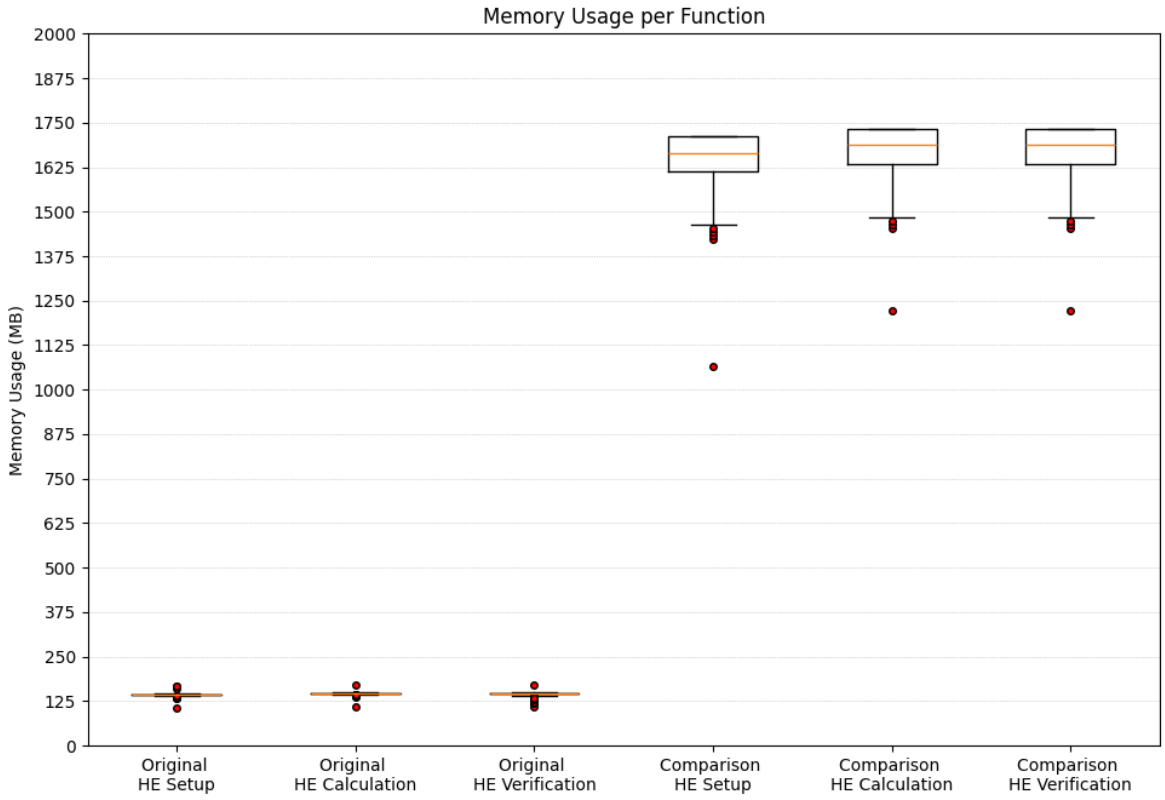

Figure 5.3: Comparison of memory usage between original implementation and new TFHE-rs implementation

Table 5.2: Memory usage for the original and new HE functions (MB)

	Orig Gen	Orig Cal	Orig Ver	Gen	Cal	Ver
Maximum	166.77	171.64	171.64	1712.76	1732.51	1732.51
Average	143.10	146.58	145.51	1655.25	1676.15	1676.15
Median	143.30	146.82	146.59	1664.49	1688.94	1688.94
Minimum	104.79	108.92	109.04	1065.26	1221.11	1221.11

The memory usage is significantly larger in the implementation using the TFHE-rs addon than in the original implementation. Notably, no evaluation/server key is needed in node-seal, whereas the server key is the largest key in TFHE-rs, which explains part of the difference in memory consumption. However, the evaluation key size of approximately 100 MB does not explain the overhead of ∼1500 MB in the TFHE-rs addon. As discussed later, the usage of TFHE-rs as a NAPI-RS addon instead of using it in pure Rust causes an overhead of more than 1000 MB.

5.2.3 Execution time

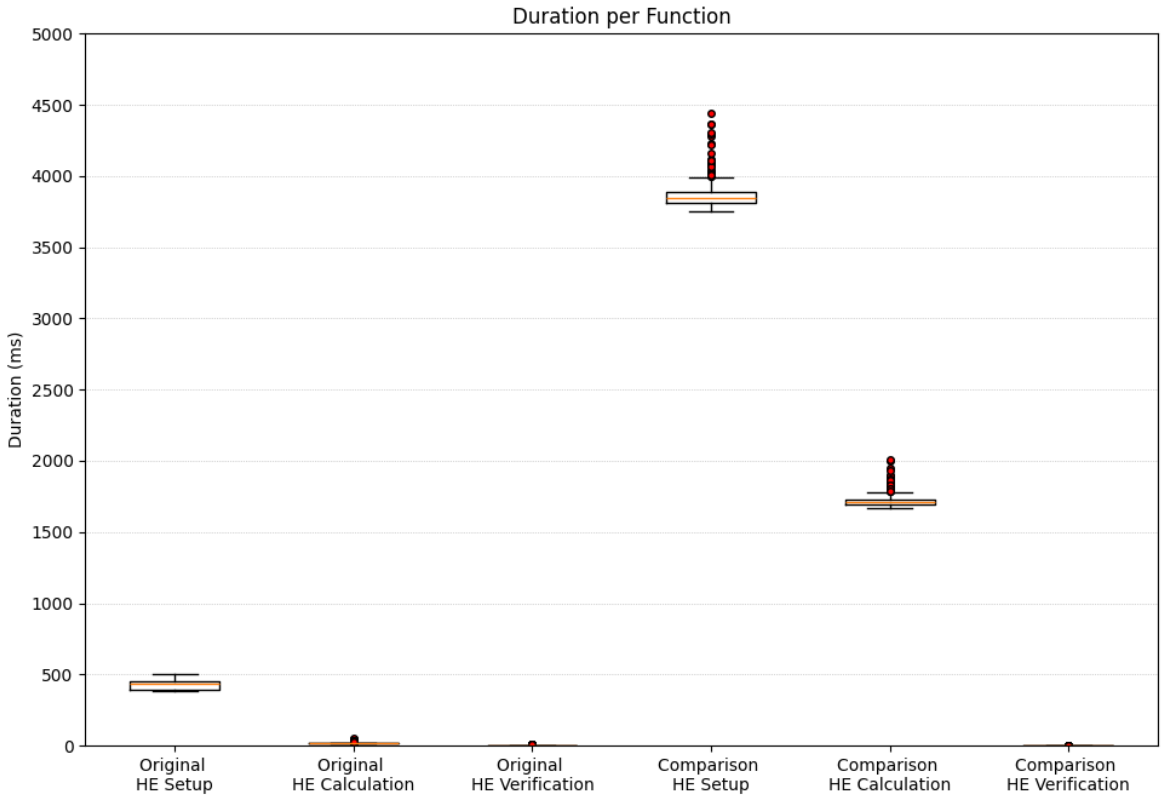

Figure 5.4: Comparison of execution time between original implementation and new TFHE-rs implementation

Table 5.3: Execution time for the original and new HE functions (ms)

	Orig Gen	Orig Cal	Orig Ver	Gen	Cal	Ver
Maximum	503.24	49.03	8.73	4442.80	2012.36	3.86
Average	428.41	18.12	2.96	3867.24	1718.28	2.14
Median	431.62	16.28	2.52	3847.35	1710.94	2.00
Minimum	387.35	13.83	2.07	3751.96	1663.86	1.91

The execution takes significantly longer in the implementation using the comparison operation. As will be discussed when contrasting TFHE-rs in pure Rust compared to its NAPI-RS counterpart in JavaScript, a significant part of the time loss (approximately 3000 ms) can be explained by the usage of TFHE-rs as a JavaScript addon.

Since the user experience is an important factor for real-world implementation, the total average waiting time for the user of 449.49ms resp. 5587.66ms can be analyzed from a human-computer interaction perspective. According to [15], three categories of waiting times can be distinguished:

- 0.1s: The user perceives the system as reacting instantaneously
- 1.0s: The user's flow of thought remains uninterrupted, but the delay is noticed
- 10s: Limit for the user's attention to remain focused on the dialogue

In this scheme, the original prototype falls into a more optimal category than the new prototype using the comparison operation. While the original prototype does not interrupt the flow of thought of the user, even though a slight delay might be noticed, the newer prototype will interrupt the user's flow of thought. However, according to this scheme, both prototypes would keep the user's attention focused on the dialogue of the decentralized identity application. Also, as pointed out by [44], such thresholds are highly dependent on the context. In a decentralized identity application, where users are often reliant on the success of verification, the willingness to wait might exceed the limits mentioned.

5.2.4 Costs

To estimate the costs of storing relevant data on the Ethereum blockchain, an Ether (ETH) price of 2600 USD was assumed, as well as a gas price of 4 gwei.

Table 5.4: Costs for the original and new HE functions

	Orig. HE Key	Orig. HE Cipher	HE Key	HE Cipher
Data (KB)	623.78	452.55	99323.48	1834.90
Gas	441'139'356	320'075'556	71'807'498'216	1'326'564'640
ETH	1.764557	1.280302	287.229993	5.306259
USD	4587.85	3328.79	746'797.98	13'796.27

As seen in table 5.4, the costs for storing keys and ciphertexts on the blockchain sum up to multiple thousand U.S. dollars. Therefore, it is not feasible in practice to store values on a blockchain for either prototype.

5.2.5 Scalability

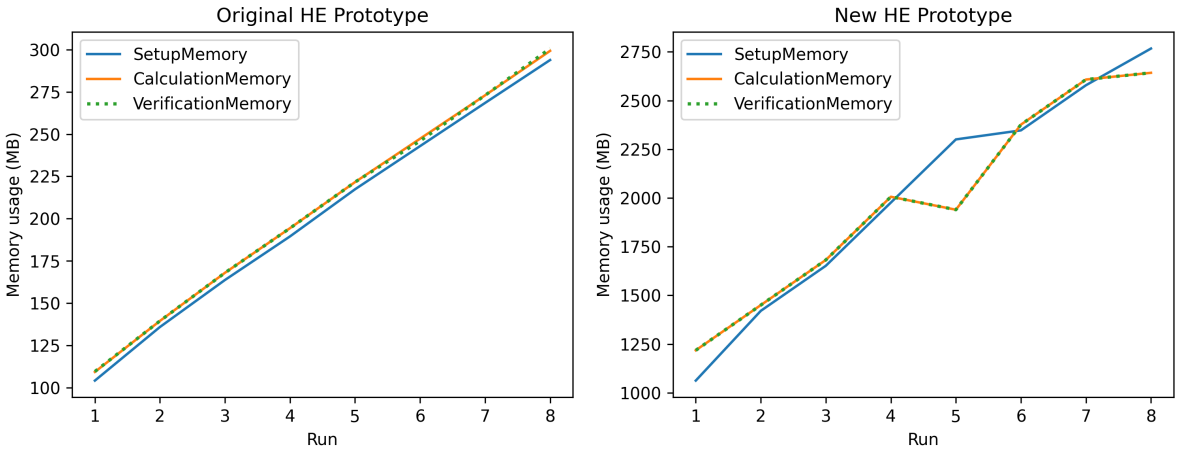

Figure 5.5: Memory increase with storage of results after each iteration

To measure the scalability, all results from the examined functions were stored in memory over multiple runs (e.g. signature public key, signature, threshold ciphertext, HE public key, HE server key and HE client key for the setup). With the new HE prototype, a maximum of 8 runs could be executed with this method before the JavaScript heap went out of memory. In the original HE prototype, this limit was not reached within 400 iterations. For comparability, the memory usage of both prototypes is shown for 8 consecutive iterations in figure 5.5.

During these 8 runs, memory increased from 1062.41 MB to 2767.02 MB for the verifier setup of the TFHE-rs implementation, which corresponds to an average increase of 213.07 MB per iteration. In the original prototype, memory usage in the same protocol phase went from 104.25 MB to 293.83 MB (increase of 23.70 MB per iteration). Clearly, it is not practical to store the necessary keys and ciphertexts of thousands of verification processes in memory with the new prototype. The original prototype is significantly more scalable in terms of memory, but the memory consumption should nevertheless be kept in mind in this prototype as well. To improve scalability for both prototypes, persisting the data in the main storage is necessary.

Important to mention are the identical values of memory consumption in the new HE prototype's calculation and verification steps. This can be explained by the very small resource needs of the verification step together with its short execution time described in section 5.2.3, which does not allow garbage collection to take its effect in between.

A further argument speaking against the scalability of both prototypes are the costs

of storing all necessary data on a blockchain, which was estimated not to be practical, see section 5.2.4. Additionally, there is currently no way of running multiple verification processes simultaneously, which would be necessary to keep the execution time constant while increasing the number of users.

5.3 Comparison with ZKP

5.3.1 CPU usage

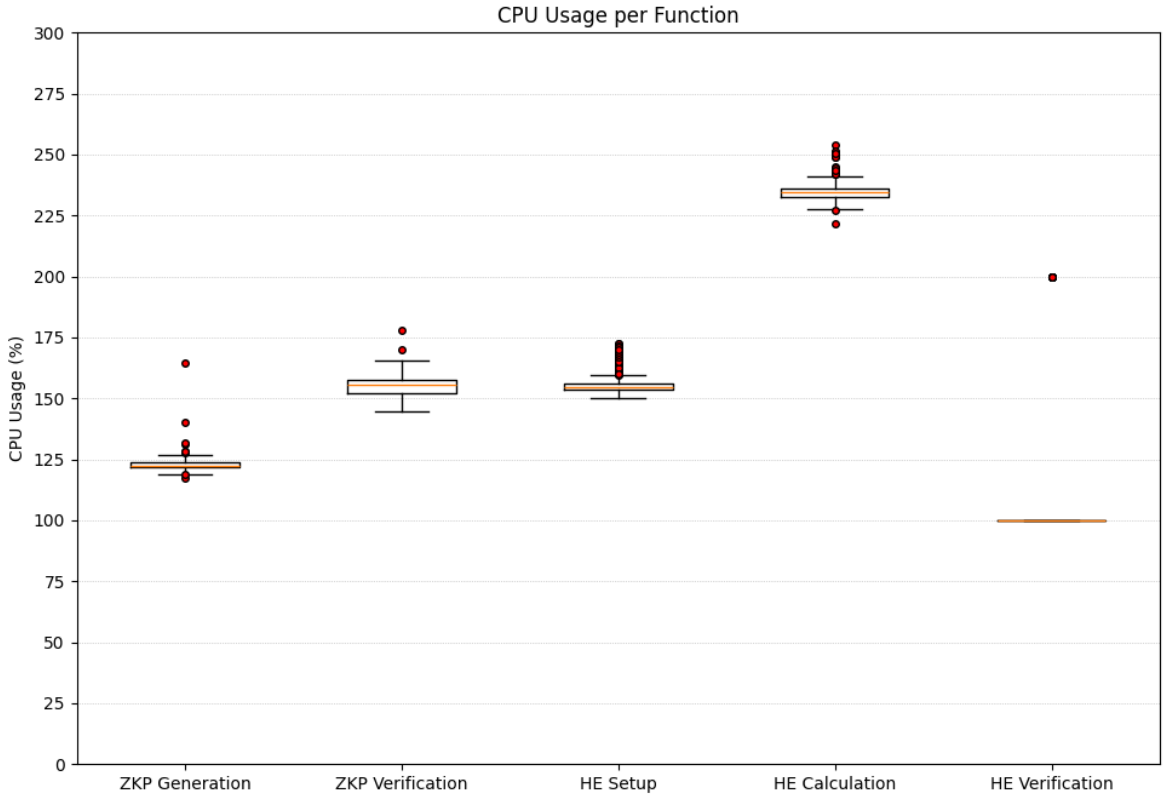

Figure 5.6: Comparison of CPU usage between ZKP implementation and TFHE-rs implementation

Table 5.5: CPU usage for the ZKP and new HE functions (%)

	ZKP Gen	ZKP Ver	Gen	Cal	Ver
Maximum	164.38	177.78	172.29	254.05	200.00
Average	122.90	155.29	156.21	234.70	117.39
Median	122.58	155.56	154.38	234.52	100.00
Minimum	117.46	144.44	150.13	221.58	100.00

The HE protocol using TFHE-rs has slightly larger CPU usage for the setup and calculation compared to the ZKP prototype, but lower CPU usage for the HE verification process. As discussed in section 5.3.4, ZKP has a significantly smaller key size, naturally leading to less intense computation and therefore lower CPU usage.

5.3.2 Memory usage

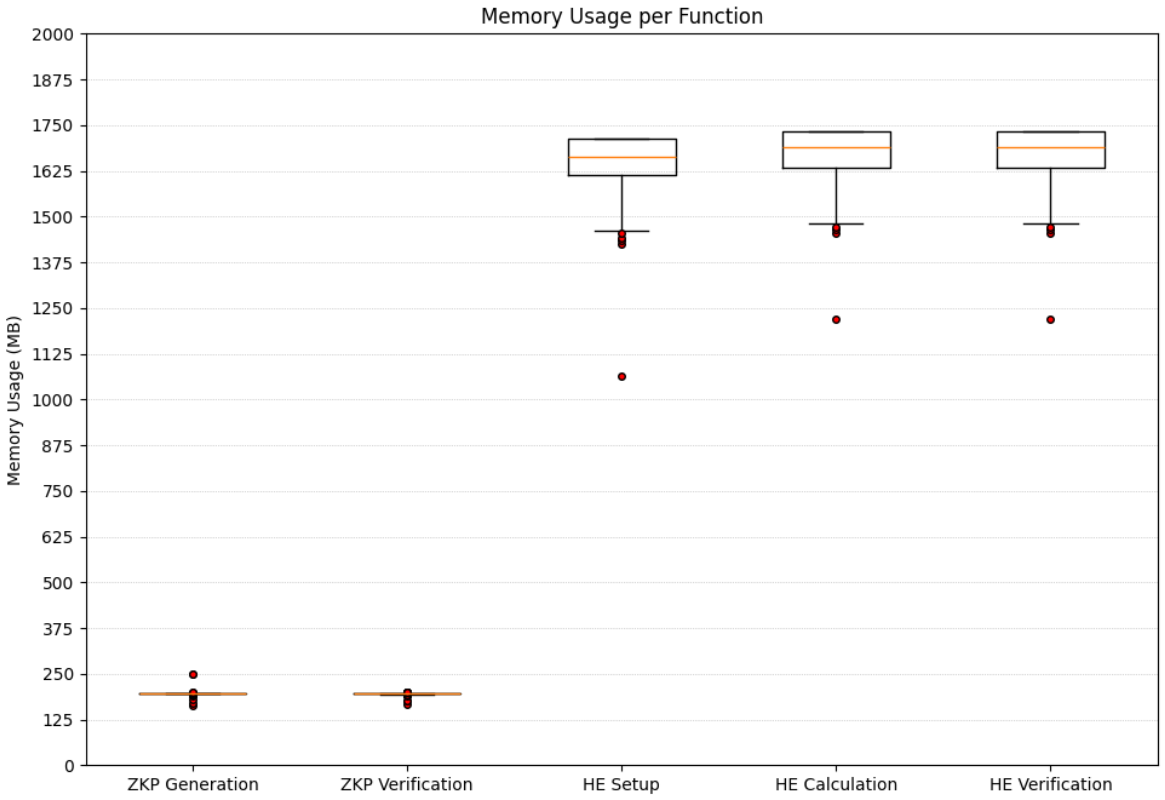

Figure 5.7: Comparison of memory usage between ZKP implementation and TFHE-rs implementation

Table 5.6: Memory usage for the ZKP and new HE functions (MB)

	ZKP Gen	ZKP Ver	Gen	Cal	Ver
Maximum	250.90	198.65	1712.76	1732.51	1732.51
Average	196.96	194.85	1655.25	1676.15	1676.15
Median	197.04	195.13	1664.49	1688.94	1688.94
Minimum	163.60	167.83	1065.26	1221.11	1221.11

The ZKP prototype has a slightly higher memory usage than the original HE prototype, but its memory usage is still significantly smaller than the memory usage of the new HE

prototype using TFHE-rs. As discussed in section 5.2.2, the main part of this overhead comes from the usage of TFHE-rs as a Rust addon, not from TFHE-rs itself.

5.3.3 Execution time

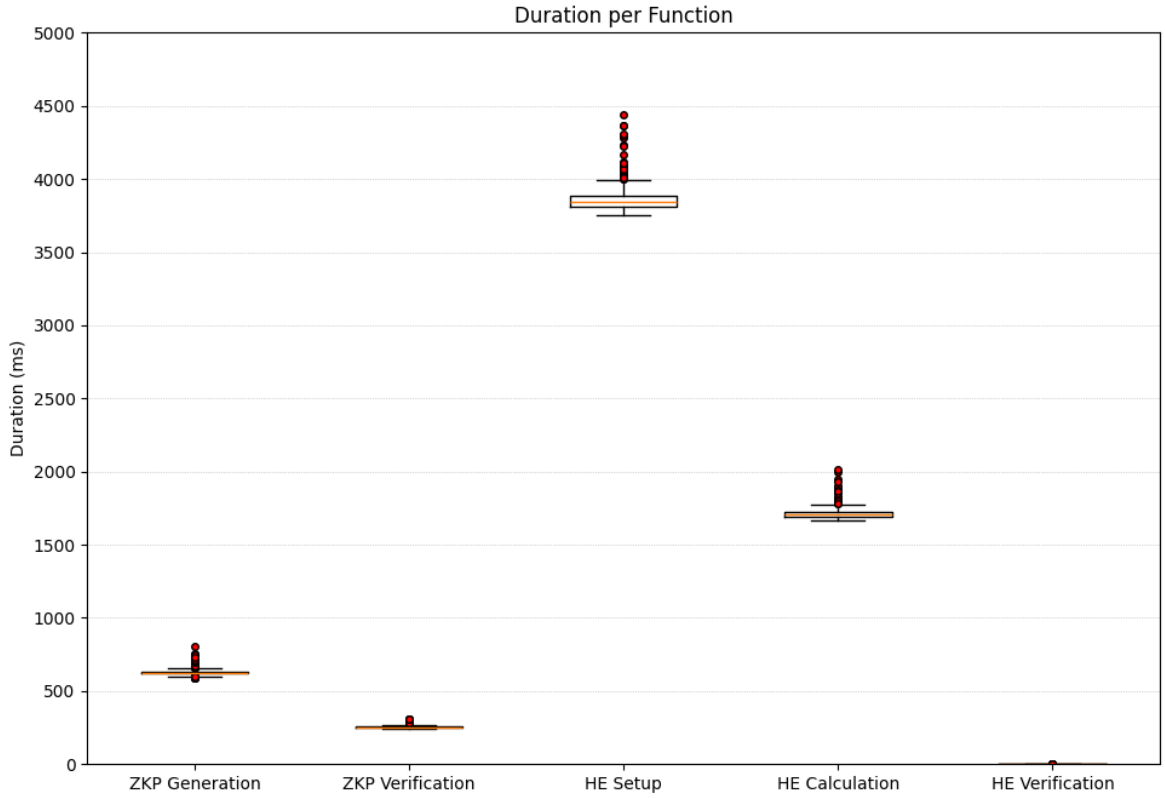

Figure 5.8: Comparison of execution time between ZKP implementation and TFHE-rs implementation

Table 5.7: Execution time for the ZKP and new HE functions (ms)

	ZKP Gen	ZKP Ver	Gen	Cal	Ver
Maximum	802.22	306.23	4442.80	2012.36	3.86
Average	632.99	257.68	3867.24	1718.28	2.14
Median	625.42	253.54	3847.35	1710.94	2.00
Minimum	591.51	242.27	3751.96	1663.86	1.91

As for the original HE prototype, the ZKP prototype is significantly more performant in terms of execution time. The ZKP falls into the same category as the original HE prototype on the waiting time scheme presented in [15], not interrupting the flow of

thought of the user. The ZKP prototype can therefore be considered more user friendly than the new HE prototype.

5.3.4 Costs

Table 5.8: Costs for the ZKP and new HE functions

	ZKP Key	ZKP Proof	HE Key	HE Cipher
Data (KB)	1.45	0.75	99323.48	1834.90
Gas	1'082'408	561'492	71'807'498'216	1'326'564'640
ETH	0.004330	0.002246	287.229993	5.306259
USD	11.26	5.84	746'797.98	13'796.27

The sizes of the ZKP key and ZKP proof are significantly smaller than the size of HE key and HE ciphertext, bringing the ZKP prototype closer to the possibility of a real-world implementation. Nevertheless, the price of 11.26 USD resp. 5.84 USD is still too high to sensefully scale the prototype.

5.3.5 Scalability

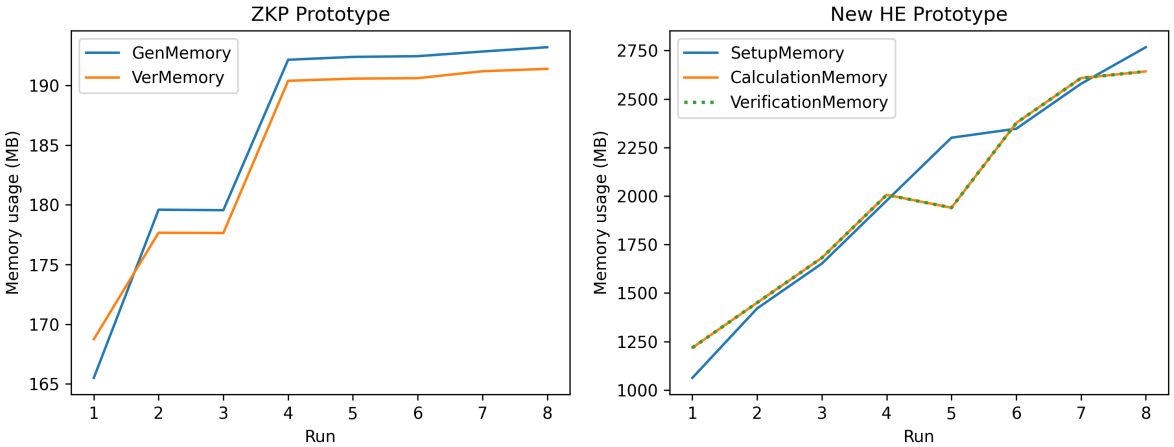

Figure 5.9: Memory increase with storage of all results after each iteration

During eight consecutive iterations, the ZKP prototype's memory usage increased from 165.51 MB to 193.19 MB for the generation step, corresponding to a rise of 3.46 MB per iteration. Hence, the increase in memory consumption due to the storage of results is minimal in the ZKP version, especially compared to the new TFHE-rs prototype. As mentioned, the ZKP prototype also has restricted scalability in terms of costs.

5.4 Overhead in JavaScript compared to Rust

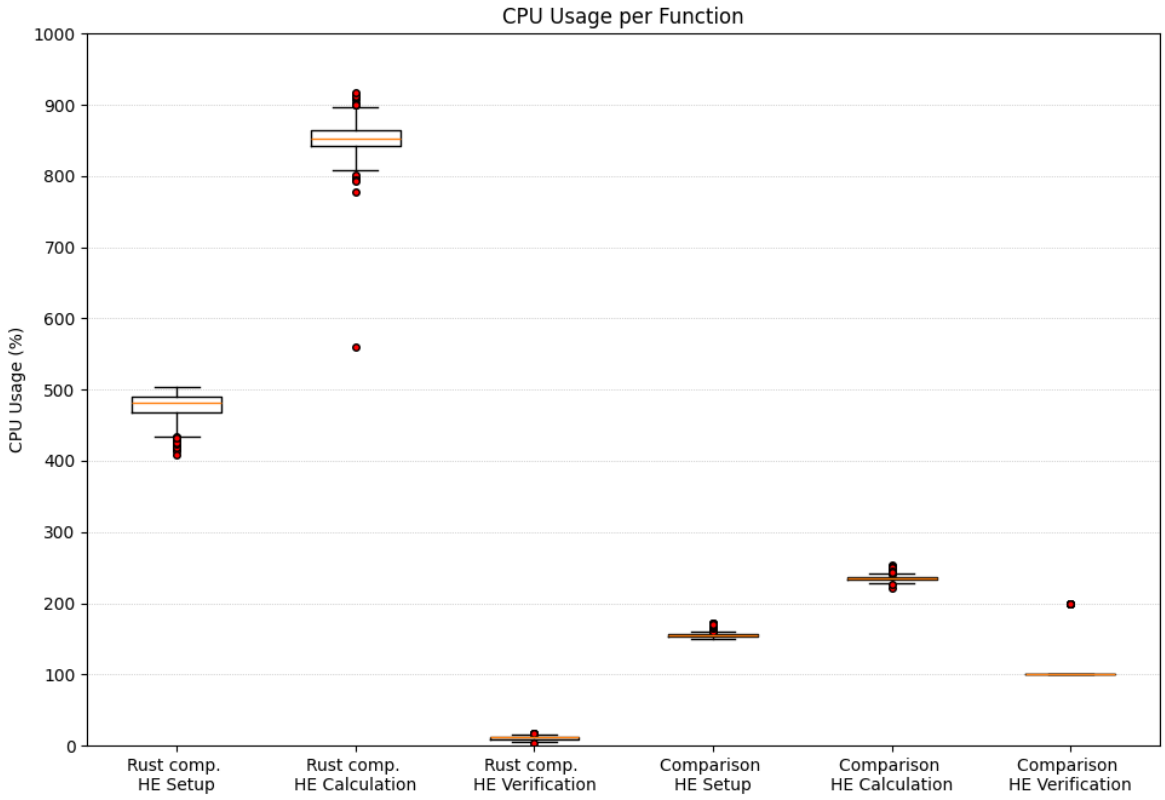

Figure 5.10: Comparison of CPU usage between pure Rust and addon

Table 5.9: CPU usage between pure Rust and JavaScript addon (%)

	Rust Gen	Rust Cal	Rust Ver	Gen	Cal	Ver
Maximum	504.17	917.29	17.98	172.29	254.05	200.00
Average	475.33	853.03	10.76	156.21	234.70	117.39
Median	471.37	852.45	11.50	154.38	234.52	100.00
Minimum	408.81	559.90	3.99	150.13	221.58	100.00

Compared to the JavaScript version, the setup and calculation steps have higher CPU usage in the pure Rust version [47]. Verification seems to have lower CPU usage in pure Rust, but this might be caused by the measurement environment. Since zeroes are filtered out, many measurements in the JavaScript version are not counted. The Rust measurement environment seems to be more precise and also catch values close to zero accurately, making less zero values appear in total. Hence, the Rust version seems to have higher CPU usage overall. This can be explained by the various optimizations in terms of

parallelization used in TFHE-rs [5]. These optimizations are apparently only applicable partially when used as an addon, leading to less parallelization and hence lower CPU usage.

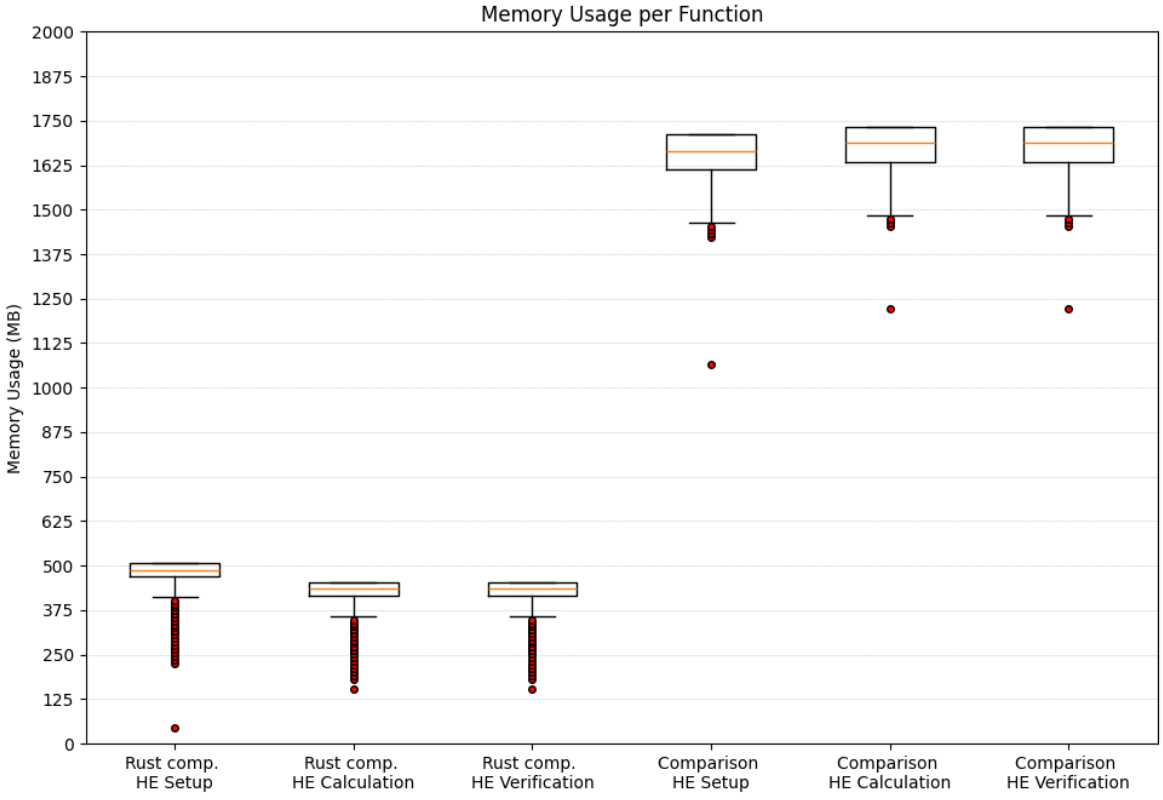

Figure 5.11: Comparison of memory usage between pure Rust and addon

Table 5.10: Memory usage between pure Rust and JavaScript addon (MB)

	Rust Gen	Rust Cal	Rust Ver	Gen	Cal	Ver
Maximum	508.33	453.81	453.81	1712.76	1732.51	1732.51
Average	474.15	420.80	420.80	1655.25	1676.15	1676.15
Median	488.53	434.18	434.18	1664.49	1688.94	1688.94
Minimum	42.91	152.33	152.33	1065.26	1221.11	1221.11

In contrast to CPU usage, the memory usage seems to be lower in the Rust implementation when compared to JavaScript. This overhead most likely comes from the type conversions between Rust and Node.js, which probably cause the existence of multiple copies of equivalent data structures in memory.

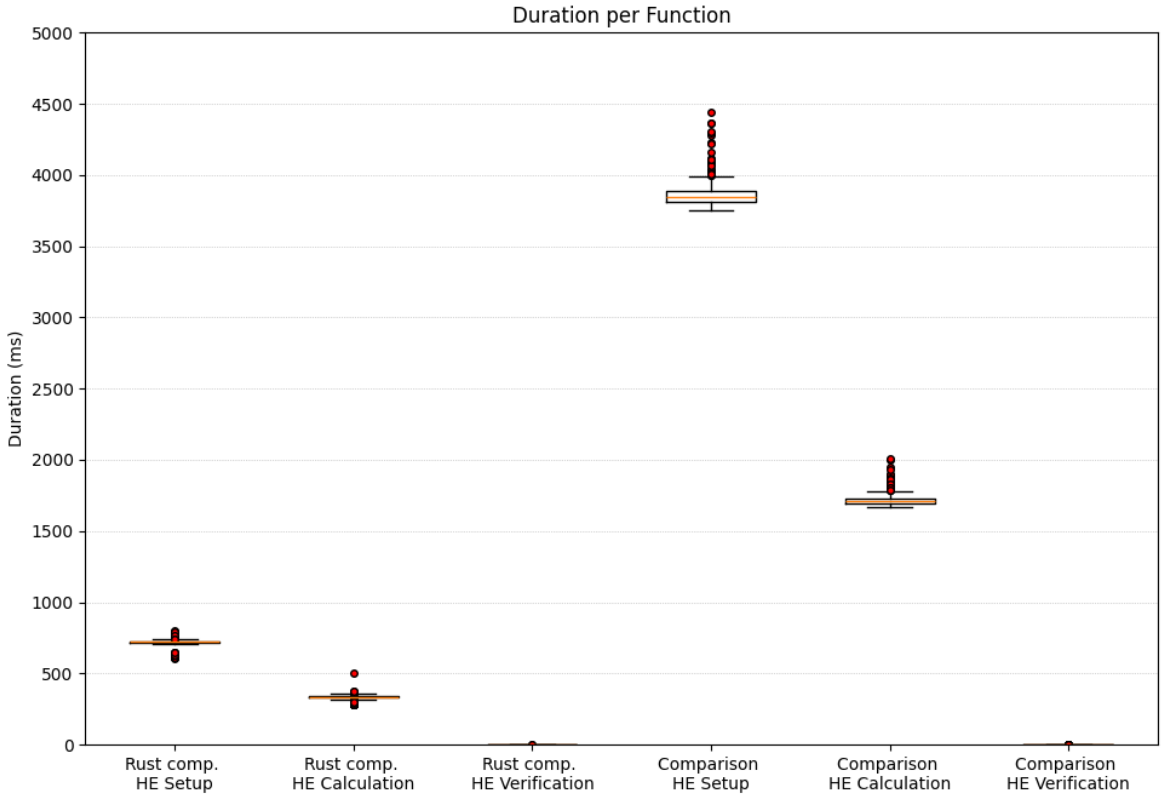

Figure 5.12: Comparison of duration between pure Rust and addon

Table 5.11: Execution time between pure Rust and JavaScript addon (ms)

	Rust Gen	Rust Cal	Rust Ver	Gen	Cal	Ver
Maximum	799.31	499.82	0.04	4442.80	2012.36	3.86
Average	711.02	333.06	0.01	3867.24	1718.28	2.14
Median	721.11	336.40	0.01	3847.35	1710.94	2.00
Minimum	608.32	282.45	0.01	3751.96	1663.86	1.91

The execution time is significantly lower in the Rust version compared to JavaScript. Possible explanations are the additional serialization and deserialization processes needed as well as type conversions between Rust and JavaScript. Since less CPU cores are used in JavaScript, this has an additional negative impact on the execution time.

5.5 Role-switched protocol

For the role-switched protocol, CPU usage was measured using the package `cpu-percentage` instead of `pidusage`, since the latter reported an unrealistic CPU usage of 0% for all runs

of the calculation phase.

Table 5.12: CPU usage for the role-switched protocol (%)

	Gen	Cal	Dec	Ver
Maximum	201.00	219.20	117.82	569.70
Average	187.28	211.28	106.65	225.61
Median	187.33	211.28	104.65	185.40
Minimum	175.06	202.21	100.72	113.60

The role-switched protocol shows a CPU usage comparable to the protocol without switched roles, which is expected since the same fundamental building blocks were used.

Table 5.13: Memory usage for the role-switched protocol (MB)

	Gen	Cal	Dec	Ver
Maximum	1350.48	1145.71	1142.22	1127.74
Average	1300.73	1099.46	1092.66	1087.27
Median	1310.94	1112.15	1105.10	1100.75
Minimum	515.59	887.62	887.62	887.62

Surprisingly, the role-switched protocol shows lower overall memory usage than the version without switched roles. Most likely, this again has to do with the measurement method rather than actual significant differences. Since the memory usage measurement is only executed at a single instance after the execution of the function, different garbage collection timings can explain this result.

Table 5.14: Execution time for the role-switched protocol (ms)

	Gen	Cal	Dec	Ver
Maximum	3288.77	185'764.50	326.26	2.42
Average	2618.04	159'687.19	250.04	1.00
Median	2596.35	159'139.84	247.28	0.99
Minimum	2497.38	140'759.91	233.27	0.13

Execution time is significantly larger for the calculation phase in the role-switched protocol, which is expected since 128 ciphertexts have to be encrypted compared to a single one in the protocol without switched roles.

5.6 OpenFHE implementation

Since every iteration of the comparison operation implemented in OpenFHE takes around 370 seconds, the number of iterations was reduced to 10 for the following measurement.

Furthermore, CPU usage was measured using the package `cpu-percentage` instead of `pidusage`, since the latter reported a CPU usage of 0% for all runs.

Table 5.15: Performance metrics for the OpenFHE comparison operation

	CPU usage (%)	Memory usage (MB)	Execution time (ms)
Maximum	129.40	2440.26	382'239.95
Average	128.75	2372.26	373'377.87
Median	128.97	2439.19	376'488.98
Minimum	127.55	1771.70	363'165.57

The implementation in OpenFHE shows lower CPU usage, higher memory usage and a larger execution time compared to the TFHE-rs addon. This is expected due to less in-depth optimizations such as parallelization.

Chapter 6

Final Considerations

6.1 Summary

During the course of this work, the comparison operation in homomorphic encryption and its integration into a decentralized identity application was investigated. In the first phase, the foundations of homomorphic encryption and decentralized identity applications were discussed. Then, two protocols were designed: In the first one, the prover executes the comparison operation, whereas the verifier computes this homomorphic operation in the second protocol. Since the comparison operation needs to compare two 64-bit integers in this use case, the bitwise encryption scheme TFHE was chosen as the most suitable variant of HE. A Rust implementation of TFHE called TFHE-rs already provides the homomorphic comparison operation, which made it a suitable library for porting to JavaScript. The port to JavaScript was achieved by means of NAPI-RS, which allows developers to write native addons executable in JavaScript. Once functional, the addon's encryption, decryption and comparison operation were integrated into the existing repository of a decentralized identity application prototype. This new HE prototype was then evaluated against the original HE prototype as well as a prototype using Zero-Knowledge Proofs.

6.2 Conclusions

6.2.1 Achievement of objectives

The first objective of this Bachelor's thesis was to establish a background on decentralized identity systems and encryption. Since the focus was put on the homomorphic comparison operation at first, this goal was addressed only in the second half of the time schedule. Nevertheless, a thorough understanding of such systems was then established, integrating the gained knowledge e.g. into considerations on the protocol designs.

A second key objective of this thesis was to conduct a comprehensive literature review

on homomorphic encryption and different HE schemes as well as corresponding comparison operations. This objective was fully met during this thesis. In-depth mathematical mechanisms, available schemes and libraries as well as various approaches to the homomorphically encrypted comparison operation were explored.

Thirdly, the problem posed by homomorphic encryption and the lack of the comparison operation in various HE libraries was required to be outlined. This objective was fulfilled.

Concerning the prototype design, which was the fourth goal of this Bachelor's thesis, all considerations and choices were critically discussed. In consultation with my supervisor Daria Schumm, the design choice fell on a method not proposed in the task description (integration of TFHE-rs as a plugin to Node.js). In addition to the discussion of possible implementations of the comparison operation itself, different decentralized identity application protocols were proposed, addressing some drawbacks of the given original protocol.

Since credential issuance and revocation was already provided by the code base, the main part of the fifth goal (implementation of the comparison operation by extending the provided code base) consisted of integrating the comparison operation into the code. This was achieved, and the second proposed (role-switched) protocol was also integrated into the code base. In addition, an experimental version of the comparison operation implemented from scratch was programmed to gain a deeper understanding of the mechanisms involved.

In the evaluation part, the newly implemented HE approach was compared with the existing Zero-Knowledge proof (ZKP) approach in terms of performance, costs, and scalability. Additionally, the new HE approach was also compared to the original HE approach. Also, the role-switched protocol as well as the experimental OpenFHE implementation were briefly discussed.

6.2.2 Key takeaways

A key finding from this work is that many different possibilities exist to implement the homomorphically encrypted comparison operation in JavaScript, but that execution time performance is the most significant limiting factor. The usage of the comparison operation was found to have benefits in terms of security compared to the previous version using multiplication with a random number for obscuring the issuance date. Lastly, the protocol used between prover and verifier was found to be a key factor for the usability and security of the decentralized identity application. The decision which party executes the comparison operation has a significant impact on the implied design needs.

6.2.3 Difficulties encountered

During the course of this work, quite a few difficulties were encountered. Finding a way to reduce memory usage and prevent extensive accumulation of data in memory was challenging, since the specific problem was not documented and only some general information

about it could be gathered. This required experimentation to find a working solution.

Another challenge encountered was the poor performance of the ”classical public key” in TFHE-rs. Even though the documentation mentioned a smaller sized ”compact public key”, it was not mentioned that this would also improve performance in terms of execution time. This challenge was overcome by contacting the TFHE-rs technical support [53] and asking about reasons for the poor performance. The team then emphasized the much shorter execution time of the ”compact public key”.

Furthermore, getting a grasp of the mathematical concepts behind homomorphic encryption was demanding. Many concepts were remotely familiar, but put together in an unknown level of complexity. For this part, it helped that I had visited linear algebra courses with mathematics students beforehand.

Additionally, it was sometimes difficult to keep track of both the highly specific view on the comparison operation as well as the broader context of a decentralized identity application.

Finding a meaningful ”information hiding strategy” for the role-switched protocol posed another challenge. It took a lot of sketches and brainstormings to get to the current solution. Nevertheless, the process was also a creative and interesting one, and helped to better understand the usage of homomorphic encryption in decentralized identity applications.

The self-implemented comparison operation in OpenFHE was mainly challenging in the translation of the algorithm from the theoretical paper to actual code. To find out the correct implementation of indices, handwritten toy examples were played through, which was a time consuming process.

6.2.4 Modifications to original scope

No major modifications to the originally arranged scope of this Bachelor’s thesis were undertaken. The main deviation are the additional explorations of the role-switched protocol and the experimental implementation in OpenFHE, which however do not change the scope but only extend it.

6.2.5 Differences between proposed and actual schedule

The schedule outlined at the beginning of this Bachelor’s thesis was much more linear than the actual schedule followed during writing. At many points, going back to previous points or anticipating points planned at a later stage was necessary. Still, the rough schedule was adhered to.

6.3 Future Work

Further research building on top of this Bachelor's thesis is possible in many different directions. Firstly, more optimizations for the TFHE-rs plugin built with NAPI-RS could be explored, since this is a major source of performance loss in comparison with node-seal. Also, the TFHE-rs JS on WASM API might allow for homomorphic computations at some point, providing an interesting subject for performance comparison. Another possible topic for further research is the hypothetical protocol presented in section 3.1.5, where approaches to implementing such a protocol could be investigated and its benefits and drawbacks could be discussed in detail. Additionally, a general exploratory study on possible uses of homomorphic encryption in decentralized identity applications could be useful.

Since the prover's algorithm will likely be executed on a mobile device, trying to run parts of the prototype on devices with restricted hardware could bring this research closer to a real-world implementation. Lastly, for performance reasons, it would also be beneficial to discuss the possibility of moving parts of the prototype (e.g. the verifier's code) entirely to a more performant programming language such as Rust or C++.

Bibliography

[1] Mozilla Developer Network. *JavaScript Data Types and Data Structures*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Data_structures.

[2] Abbas Acar et al. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *ACM Computing Surveys* 51.4 (July 2019), pp. 1–35.

[3] Ahmad Al Badawi et al. "OpenFHE: Open-Source Fully Homomorphic Encryption Library". In: *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. Nov. 2022, pp. 53–63.

[4] Nick Angelou. *node-seal*. URL: https://www.npmjs.com/package/node-seal.

[5] Mathieu Ballandras et al. *TFHE-rs: A (Practical) Handbook*. Feb. 2025.

[6] Elaine Barker. *Recommendation for Key Management: Part 1 - General*. Tech. rep. NIST SP 800-57pt1r5. National Institute of Standards and Technology, May 2020, NIST SP 800–57pt1r5.

[7] Antoine Bluchet. *pidusage*. URL: https://www.npmjs.com/package/pidusage.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *ACM Transactions on Computation Theory* 6.3 (July 2014), pp. 1–36.

[9] Ayantika Chatterjee and Indranil Sengupta. "Sorting of Fully Homomorphic Encrypted Cloud Data: Can Partitioning Be Effective?" In: *IEEE Transactions on Services Computing* 13.3 (May 2020), pp. 545–558.

[10] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. "Efficient Homomorphic Comparison Methods with Optimal Complexity". In: *Advances in Cryptology – ASIACRYPT 2020*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12492. 2020, pp. 221–256.

[11] Jung Hee Cheon et al. "Numerical Method for Comparison on Homomorphically Encrypted Numbers". In: *Advances in Cryptology – ASIACRYPT 2019*. Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11922. 2019, pp. 415–445.

[12] Ilaria Chillotti. *Introduction to Practical FHE and the TFHE Scheme*. 2020.

[13] Ilaria Chillotti et al. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *Journal of Cryptology* 33.1 (Jan. 2020), pp. 34–91.

[14] Fernanda D. De Melo Hernández, César A. Hernández Melo, and Horacio Tapia-Recillas. "Fermat's Little Theorem and Euler's Theorem in a Class of Rings". In: *Communications in Algebra* 50.7 (July 2022), pp. 3064–3078.

[15] S. Egger et al. "Waiting Times in Quality of Experience for Web Based Services". In: *2012 Fourth International Workshop on Quality of Multimedia Experience*. July 2012, pp. 86–96.

[16] Juan Garay, Berry Schoenmakers, and José Villegas. "Practical and Secure Solutions for Integer Comparison". In: *Public Key Cryptography – PKC 2007*. Ed. by Tatsuaki Okamoto and Xiaoyun Wang. Vol. 4450. 2007, pp. 330–342.

[17] Craig Gentry. "A Fully Homomorphic Encryption Scheme". PhD thesis. Stanford University.

[18] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof Systems". In: *SIAM Journal on Computing* 18.1 (Feb. 1989), pp. 186–208.

[19] Shruthi Gorantala et al. *A General Purpose Transpiler for Fully Homomorphic Encryption*. 2021.

[20] Shai Halevi and Victor Shoup. "Design and Implementation of HElib: A Homomorphic Encryption Library". In: *Cryptology ePrint Archive* (2020).

[21] Shai Halevi and Victor Shoup. *HElib*. URL: `https://github.com/homenc/HElib`.

[22] M. Hellman. "An Overview of Public Key Cryptography". In: *IEEE Communications Society Magazine* 16.6 (Nov. 1978), pp. 24–32.

[23] Kelsey Houston-Edwards. *Learning with Errors: Encrypting with Unsolvable Equations*. Jan. 2023. URL: `https://www.youtube.com/watch?v=K026C5YaB3A`.

[24] Ilia Iliashenko and Vincent Zucca. "Faster Homomorphic Comparison Operations for BGV and BFV". In: *Proceedings on Privacy Enhancing Technologies* 2021.3 (July 2021), pp. 246–264.

[25] Muhammad Ismail and Ted Brownlow. *OpenFHE-WASM*. URL: `https://github.com/openfheorg/openfhe-wasm`.

[26] Andrew Kresch. *Skript: Lineare Algebra I*. 2023.

[27] Baiyu Li and Daniele Micciancio. "On the Security of Homomorphic Encryption on Approximate Numbers". In: *Advances in Cryptology – EUROCRYPT 2021*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. 2021, pp. 648–677.

[28] Chiara Marcolla et al. "Survey on Fully Homomorphic Encryption, Theory, and Applications". In: *Proceedings of the IEEE* 110.10 (Oct. 2022), pp. 1572–1609.

[29] Microsoft. *SEAL*. URL: `https://github.com/microsoft/SEAL`.

[30] Eduardo Morais et al. "A Survey on Zero Knowledge Range Proofs and Applications". In: *SN Applied Sciences* 1.8 (Aug. 2019), p. 946.

[31] *NAPI-RS*. URL: `https://napi.rs/docs/introduction/getting-started`.

[32] Harika Narumanchi et al. "Performance Analysis of Sorting of FHE Data: Integer-Wise Comparison vs Bit-Wise Comparison". In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. Mar. 2017, pp. 902–908.

[33] Node.js. *Child Processes*. URL: `https://nodejs.org/api/child_process.html`.

[34] Node.js. *Node-API*. URL: `https://nodejs.org/api/n-api.html`.

[35] Chris Peikert. *Algebraic Lattices and Ring Learning with Errors*. Jan. 2020. URL: `https://simons.berkeley.edu/talks/algebraic-lattices-ring-learning-errors`.

[36] Sébastien Philippe et al. "A Physical Zero-Knowledge Object-Comparison System for Nuclear Warhead Verification". In: *Nature Communications* 7.1 (Sept. 2016), p. 12890.

[37] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. May 2005, pp. 84–93.

[38] Oded Regev. "The Learning with Errors Problem (Invited Survey)". In: *2010 IEEE 25th Annual Conference on Computational Complexity*. June 2010, pp. 191–204.
[39] Isaac Z. Schlueter and npm Inc. *Npm*. URL: https://www.npmjs.com/.
[40] Daria Schumm. *Credchain (Original)*. URL: https://github.com/schummd/credchain/tree/privacy.
[41] Daria Schumm, Rahma Mukta, and Hye-young Paik. "Efficient Credential Revocation Using Cryptographic Accumulators". In: *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. May 2023, pp. 1–3.
[42] Daria Schumm, Katharina O. E. Müller, and Burkhard Stiller. *Are We There Yet? A Study of Decentralized Identity Applications*. 2025.
[43] Daria Schumm et al. "Metadata Privacy in Decentralized Identity Applications". Under Review. 2025.
[44] Steven C. Seow. *Designing and Engineering Time: The Psychology of Time Perception in Software*. 2008.
[45] *Serde*. URL: https://docs.rs/serde/latest/serde/.
[46] Gabriel Stegmaier. *Credchain (Fork)*. URL: https://github.com/gstegm/credchain/tree/HE_comparison.
[47] Gabriel Stegmaier. *Credchain (pure Rust version)*. URL: https://github.com/gstegm/credchain_rust.
[48] Gabriel Stegmaier. *HE_comparison_implementation*. URL: https://github.com/gstegm/HE_comparison_implementation.
[49] *The Rust Reference*. Rust. URL: https://doc.rust-lang.org/stable/reference/.
[50] Mihai Togan and Cezar Plesca. "Comparison-Based Computations over Fully Homomorphic Encrypted Data". In: *2014 10th International Conference on Communications (COMM)*. May 2014, pp. 1–6.
[51] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. *Verifiable Fully Homomorphic Encryption*. 2023. (Visited on Mar. 11, 2025).
[52] Hao Yang et al. "Phantom: A CUDA-Accelerated Word-Wise Homomorphic Encryption Library". In: *IEEE Transactions on Dependable and Secure Computing* 21.5 (Sept. 2024), pp. 4895–4906.
[53] ZAMA. *Community: Technical support*. URL: https://discord.com/invite/zama.
[54] ZAMA. *Concrete*. URL: https://docs.zama.ai/concrete.
[55] ZAMA. *TFHE-rs*. URL: https://docs.zama.ai/tfhe-rs.
[56] ZAMA. *TFHE-rs: Benchmark*. URL: https://docs.zama.ai/tfhe-rs/get-started/benchmarks.

Abbreviations

HE	Homomorphic Encryption
PHE	Partially Homomorphic Encryption
SWHE	Somewhat Homomorphic Encryption
FHE	Fully Homomorphic Encryption
MSB	Most Significant Bit
LWE	Learning with errors
RLWE	Ring learning with errors
TFHE	Fully Homomorphic Encryption over the Torus
FHEW	Fastest Homomorphic Encryption in the West
CKKS	Cheon-Kim-Kim-Song
BGV	Brakerski-Gentry-Vaikuntanathan
BFV	Brakerski-Fan-Vercauteren
DFA	Deterministic Finite Automaton

List of Figures

List of Tables

Listings

Appendix A

Contents of the Repository

The full code and implementation details are available at:

- `https://github.com/gstegm/credchain/tree/HE_comparison` (adapted credchain repository)
- `https://github.com/gstegm/HE_comparison_implementation` (standalone comparison operations)
- `https://github.com/gstegm/credchain_rust` (pure Rust implementation of credchain)