# Implementing Non-Malleable Zero Knowledge

*Ethan Lee*


4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2024

# Abstract

We implemented an efficient, constant round non-malleable zero knowledge scheme in Rust. The scheme only relies on one way function as cryptographic building blocks and is 20X faster for the prover than state of the art schemes.

# Table of Contents


**1 Introduction** **1**
- 1.1 Our Contribution . . . 2
- 1.2 Overview of Techniques . . . 2

**2 Preliminaries** **4**
- 2.1 Commitments . . . 4
- 2.2 Extractable Commitments . . . 5
- 2.3 Non-Malleable Commitments . . . 6
- 2.4 Arguments/Proofs . . . 7
- 2.5 Non-Malleable Interactive Arguments/Proofs . . . 8

**3 Protocols** **9**
- 3.1 Commitment Scheme $\Pi^{3R}_{BGRRV} = (\mathcal{C}^{3R}_{BGRRV} + \mathcal{R}^{3R}_{BGRRV})$ . . . 9
- 3.2 Extractable Commitment Scheme . . . 10
- 3.3 Weak Non-Malleable Extractable Commitment Scheme . . . 11
- 3.4 Black-Box Non-Malleable Zero Knowledge . . . 12

**4 Implementations** **14**
- 4.1 Implementation Requirements . . . 14
- 4.2 Implementing $\Pi_{com} = (Com, Dec)$ . . . 14
  - 4.2.1 Naor Commitment As A Concrete Implementation . . . 14
  - 4.2.2 Justesen Code . . . 16
  - 4.2.3 Choosing A Finite Field $\mathbb{F}_p$ For Faster FFT . . . 16
  - 4.2.4 Usage of Cryptographically Secure Algorithms . . . 17
- 4.3 Parameters for $\Pi^{3R}_{BGRRV}$ . . . 18
- 4.4 Concrete Implementation of $\Pi = (\mathcal{P}, \mathcal{V})$ . . . 19
  - 4.4.1 Schnorr's $\Sigma$-Protocol . . . 19
  - 4.4.2 ZKBoo . . . 19
- 4.5 Non-Malleable Zero Knowledge . . . 20
  - 4.5.1 Asynchronous Implementation of $\Pi_{NM}$ . . . 20
- 4.6 Miscellaneous . . . 21
  - 4.6.1 Building the Protocol with Rust . . . 21
  - 4.6.2 Modular Design of Codebase . . . 21
  - 4.6.3 Improving Code Readability . . . 21
  - 4.6.4 Error Handling Design . . . 23
  - 4.6.5 Testing & Profiling . . . 24

**5 Concluding Remarks** **25**
5.1 Division of Work . . . . . 25
5.2 Obstacles . . . . . 25
5.2.1 Interfacing Protocols with Traits . . . . . 25
5.3 Future Improvements . . . . . 25

**Bibliography** **26**

**A Reed Solomon Encoding** **30**

**B Trait Complexity When Implementing $\Pi_{3Ext}$** **31**

# Chapter 1

# Introduction

The concept of non-malleability within cryptographic protocols is pivotal for safeguarding against man-in-the-middle attacks [20]. In scenarios like internet communication, where malevolent actors may intercept network traffic [15], the vulnerability of malleable cryptographic protocols becomes evident. Zero-knowledge proofs represent a category of protocols enabling a prover to convince a verifier of the truth of a statement while revealing no additional information apart from statement validity. Since their inception, [31], substantial efforts have been made to enhance their practical efficiency [30, 43].

Recent strides in zero-knowledge proof systems have laid the groundwork for novel applications [8, 27], such as attested cameras, where verifiers can authenticate the origin of images taken by the cameras rather than being generated by software [37]. Moreover, they've been instrumental in validating the accuracy of machine learning models without divulging specific model details[40]. There have also been initiatives to establish general-purpose zero-knowledge virtual machines compatible with popular instruction set architectures like RISC-V [5, 52].

A crucial development in the realm of zero-knowledge argument protocols is the establishment that such protocols can be constructed using one-way functions with only four rounds of communication, as demonstrated in [18]. This result has been recognized as optimal, as proven by Bellare, Jakobsson, and Yung in [7].

Yet, the feasibility of zero-knowledge argument protocols based on black-box usage of one-way functions remains an open question. Notably, non-malleable commitments can be built with optimal asymptotic complexity using black-box usage of one-way functions, as demonstrated in [16] and [32].

In a recent development, Kim, Liang, and Pandey introduced a non-black-box non-malleable zero-knowledge argument system [38]. Their construction achieves practical efficiency while relying on cryptographic assumptions that are plausibly post-quantum secure. However, their prover efficiency and round complexity are influenced by their usage of Ligero protocol [3].

In this paper, we delve into the design and implementation of a non-malleable zero-

knowledge scheme that maintains a constant number of rounds and exclusively depends on the black-box utilization of cryptographic primitives and one-way functions. Importantly, our approach circumvents the need for general-purpose protocols like Ligero.

## 1.1 Our Contribution

We have successfully developed a fast and robust non-malleable zero-knowledge scheme using approximately 3479 lines of Rust code. Operating at 80 bits of security, our implementation demonstrates efficiency with a runtime of 151.59 ms for the prover, coupled with a communication footprint of less than 2.9MB. This represents a notable 23X improvement in runtime and a 8X improvement in communication when compared to previous schemes.

The codebase is designed with modularity in mind, allowing easy integration with other projects. Users can seamlessly incorporate specific sections of the code, such as the implemented sigma protocol, into their own projects, or employ the entire protocol across a network.

## 1.2 Overview of Techniques

### The Rust Programming Language

Given the cryptographic nature of our protocol implementation, we have opted to develop it in Rust due to its strong emphasis on safety and its extensive ecosystem of cryptography libraries. Rust is a modern programming language developed by Mozilla, designed for high-performance and concurrent systems development. At its core is a robust ownership system that ensures memory safety by enforcing rules around ownership and borrowing at compile time. This system eliminates common pitfalls related to memory, such as null pointer dereferencing and buffer overflows.

### The Rust Module System [39]

We leveraged the Rust crate module systems to make the implementation highly modular. The Rust crate and module system plays a role in organizing and structuring code within a Rust project. A crate is the fundamental compilation unit in Rust and serves as a container for one or more modules, providing a boundary for encapsulation and code reuse. Modules, defined with the mod keyword, allow developers to group related code together, creating a hierarchical structure that mirrors the project's organization.

### Speed Optimizations via Algebraic Techniques

Applying our knowledge in elliptic curves and finite field arithmetic, we've fine-tuned our implementation, leading to a substantial exponential acceleration in subprotocol executions. More precisely, we have harnessed Fast Fourier Transform (FFT) algorithms and methodologies to significantly boost the speed of our implementation [19]. In the context of Naor Commitment [42], a naive approach would incur a commitment cost of

$O(n^2)$, whereas our optimized implementation has achieved an impressive $O(n \log n)$ with the integration of FFTs.

## Parallelization

We incorporate parallelization into both loops and specific sections of our code. Rust provides powerful mechanisms for parallelization, allowing developers to efficiently leverage multi-core architectures while maintaining safety and avoiding common pitfalls associated with concurrent programming. One key feature is the ownership system, which enables safe sharing of data between threads through ownership and borrowing rules checked at compile time. The standard library includes the **std::sync** module with synchronization primitives like **Mutex** and **Arc** for shared data access. Additionally, the **std::thread** module facilitates the creation of threads and their synchronization [39]. The rayon crate is a popular choice for parallel programming in Rust, offering a data parallelism model with high-level abstractions like parallel iterators [46]. Rust's focus on zero-cost abstractions ensures that parallelism can be achieved without sacrificing performance.

## Testing, Benchmarking and Profiling

We employed a combination of unit testing, integration testing, and fuzzing to thoroughly test both our subprotocols and the entire protocol. Additionally, we utilized Criterion, a Rust benchmarking crate, to assess the performance of our protocol [10]. The benchmarking process was iterated 100 times to obtain both the average and standard deviation.

We leveraged flamegraphs to gain insights into the execution and resource utilization of our code. Flamegraphs is a visual profiling tool, offering a graphical representation of a program's call stack to pinpoint performance bottlenecks and comprehend resource utilization [34].

# Chapter 2

# Preliminaries

In this paper, we use $\mathbb{F}$ to denote a finite field. We will represent the security parameter as $\lambda$ and use $negl(\lambda)$ to indicate any function diminishing faster than $\lambda^{-c}$, for any constant $c$. The notation $[n]$ is used to signify the set $\{1,...,n\}$. The term **ppt** will abbreviate probabilistic polynomial time. Vectors are denoted in boldface, and $\langle \cdot, \cdot \rangle$ denotes the inner product of vectors.

Consider two interactive machines, $A$ and $B$, with $(A,B)$ denoting an interactive protocol between them. The interaction between $A$ and $B$ on a common input $x$, with private inputs $a$ and $b$ for $A$ and $B$, respectively, is denoted by $\langle A(a), B(b) \rangle (x)$. The transcript generated by this interaction is represented by $\tau$.

## 2.1 Commitments

A commitment scheme is a protocol between Alice and Bob that enables Alice to commit to a secret message, denoted as $m$. Subsequently, Alice has the option to open the commitment, disclosing to Bob the specific message $m$ she committed to. For a commitment scheme to be effective, it must have two essential properties: binding and hiding. Binding ensures that Alice cannot alter her commitment, restricting the opening of a commitment to only one message, $m$. On the other hand, hiding guarantees that Bob remains unaware of the actual message Alice committed to, preserving the confidentiality of the committed information.

**Definition 2.1.1** ***Commitment*** *A commitment scheme, denoted as $\Pi_{com} = (\mathcal{C}, \mathcal{R})$, constitutes a two-phase protocol involving two probabilistic polynomial time (ppt) interactive algorithms - a committer, $\mathcal{C}$, and a receiver, $\mathcal{R}$. During the initial commit phase, $\mathcal{C}$, provided with a message m and a randomness $r_c$, engages in an interaction with $\mathcal{R}$, who inputs $r_r$. The resulting commitment transcript, $\tau = \langle \mathcal{C}(m, r_c), \mathcal{R}(r_r) \rangle$, encapsulates the committer's input m, committer randomness $r_c$, and receiver randomness $r_r$. In the subsequent decommitment phase, $\mathcal{C}$ discloses $m'$, and $\mathcal{R}$ accepts the committed value as $m'$ only if $\mathcal{C}$ convincingly demonstrates that $\tau$ could be generated with $m'$ as input. The algorithm $Dec(\tau, m, r_c)$ determines the acceptance 1 or rejection 0 of the decommitment, considering the commitment transcript $\tau$ along with the committer's message m and*

*randomness $r_c$. A commitment scheme is classified as delayed input if the message for commitment is required solely in the final round of $C$ during the commit phase. [11]*

**Definition 2.1.2** ***Statical Biding*** *A commitment scheme $(\mathcal{C},\mathcal{R})$ is said to be statistically binding if for all adversaries $\mathcal{A}$, there exists a negligible function $negl(\lambda)$ such that*

$$Pr\left[\begin{array}{c} Dec(\tau,m_0,r_0)=Dec(\tau,m_1,r_1)=1 \\ \wedge\, m_0\neq m_1 \end{array}\middle|\begin{array}{c} \tau\leftarrow\langle\mathcal{A}(m,r_c),\mathcal{R}(r_r)\rangle \\ (m_0,r_0)\leftarrow\mathcal{A} \\ (m_1,r_1)\leftarrow\mathcal{A} \end{array}\right]\leq negl(\lambda)$$

**Definition 2.1.3** ***Computational Hiding*** *A commitment scheme $(\mathcal{C},\mathcal{R})$ is said to be computational hiding if for* **ppt** *adversaries $\mathcal{A}$, there exists a negligible function $negl(\lambda)$ such that*

$$\left|Pr\left[b=b'\middle|\begin{array}{c} m_0,m_1\leftarrow\mathcal{A} \\ b\leftarrow\{0,1\} \\ \tau\leftarrow\langle\mathcal{C}(m_b,r_c),\mathcal{R}(r_r)\rangle \\ b'\leftarrow\mathcal{A}(\tau)\end{array}\right]-\frac{1}{2}\right|\leq negl(\lambda)$$

This paper also uses non-interactive commitment schemes. In such instances, the commitment phase involves a singular message transmitted from the committer to the receiver, facilitated by an algorithm **Com** and incorporating a degree of randomness denoted as $r_c$. Additionally, we define a well-formed transcript as a commitment phase transcript, $\tau$, for which there exists a pair $(m,rc)$ such that the output of $Dec(\tau,m,rc)$ equals 1.

## 2.2 Extractable Commitments

A commitment scheme is considered extractable if there exists a **ppt** extractor capable of retrieving the committed value within a polynomial timeframe. To provide a more formal definition, consider the following.

**Definition 2.2.1** ***Extractable Commitment [11].*** *Consider any statistically binding, computationally hiding commitment scheme $\Pi_{comExt}=(\mathcal{C},\mathcal{R})$. Then $Pi_{comExt}$ is said the be extractable if there exists an expected* **ppt** *oracle algorithm* **Ext** *(the extractor), such that for any* **ppt** *committer $\mathcal{C}^\star$ the following holds. Let $\tau=\langle\mathcal{C}^\star,\mathcal{R}(r_r)\rangle$ denote a (potentially maliciously generated) transcript of the interaction between $\mathcal{C}^\star$ and $\mathcal{R}$. The extractor $Ext^{\mathcal{C}^\star}(\tau,r_r)$, with oracle output to $\mathcal{C}^\star$, output m such that, over the randomness of* **Ext** *and of sampling $\tau$,*

$$Pr\left[\exists\tilde{m}\neq m,\tilde{r}_c : Dec(\tau,\tilde{m},\tilde{r}_c)=1\right]\leq negl(\lambda)$$

*[11]*

**Definition 2.2.2** ***k-Extractable Commitments [11]*** *A commitment that has the conditions outlined in* ***Definition 2.2.1*** *is termed* ***k-extractable*** *when it has an algorithm, denoted as* **Ext***, capable of extracting the committed value from a set of k well-formed commitment phase transcripts* $\tau$*. Notably, each transcript within this set shares an identical first-round committer message. The extractor, under these conditions, effectively retrieves the committed value from the initial transcript, with the exception of negligible probability.*

## 2.3 Non-Malleable Commitments

A man-in-the-middle adversary intercepts and potentially alters the communication between two parties without their knowledge. Non-Malleability captures this types of attack.

Consider a scenario involving a man-in-the-middle (**MIM**) adversary, denoted as **MIM**, engaged in two interactions: the left interaction and the right interaction. In the right interaction, **MIM** plays the role of the receiver, interacting with an honest committer denoted as $\mathcal{C}$. Conversely, in the left interaction, **MIM** acts as the committer, engaging with an honest receiver labeled as $\mathcal{R}$. To distinguish entities in the right session from those in the left, a 'tilde'd' notation is employed, signifying mirrored counterparts. For instance, if $m$ represents the value committed by $\mathcal{C}$, $\tilde{m}$ denotes the value committed by **MIM** on the right. The committer is assumed to have an identity or 'tag,' id, chosen from the set $\{0,1\}^\lambda$. At the beginning of the commitment phase, $\mathcal{C}$ receives the local input $m$, while **MIM** is provided with an auxiliary input $aux$. Non-malleable commitments are defined following the simulation paradigm [11].

During the actual interaction, **MIM** concurrently participates in both left and right interactions. In the right session, the **MIM** adversary interacts with $\mathcal{C}$, obtaining a commitment to the message $m$ using the identity $id$. In the left session, **MIM** engages with $\mathcal{R}$, attempting to commit to a related value $\tilde{m}$ using a chosen identity $\tilde{id}$. If the right commitment is invalid or undefined, the committed value is set to $\perp$. Notably, if $id = \tilde{id}$, indicating that the adversary uses the same identity as the honest committer, the attack is deemed invalid.

Let $\mathbf{MIM}(\mathcal{C},\mathcal{R})((m),aux)$ be the random variable describing (view, $\tilde{m}$), encompassing the values committed by **MIM** and **MIM**'s view in the experiment. In the simulated execution, a simulator denoted as **SIM** directly interacts with $\mathcal{R}$. Here, $\mathbf{SIM}(\mathcal{C},\mathcal{R})(1^\lambda,aux)$ represents the random variable describing (view, $\tilde{m}$), comprising the values committed by **SIM** and its output. Similar to the real interaction, whenever **SIM** commits in the right interaction with an identity identical to that of the left interaction, the committed value is set to $\perp$. The discussion revolves around one-one secure non-malleable commitments, where **MIM** participates in one left and one right interaction. We use the definition from [33].

**Definition 2.3.1** ***Non-Malleable Commitments [11]*** *A commitment scheme is non-malleable with respect to commitment if, for every* **ppt** ***MiM*** *adversary* $\mathcal{A}$*, there exists a* **ppt** *simulator* ***Sim*** *such that for all* $m \in \{0,1\}^\lambda$ *the two ensembles* $\boldsymbol{MIM}(\mathcal{C},\mathcal{R})((m),aux)_{aux\in\{0,1\}^\star}$

*and* $\textbf{\textit{SIM}}(\mathcal{C},\mathcal{R})(1^{\lambda},aux)_{aux\in\{0,1\}^{\star}}$ *are computational indistinguishable.*

A non-malleable commitment is said to be synchronous if, during the interaction, the Man-in-the-Middle (**MIM**) immediately transmits the i-th round message on the right immediately after receiving the i-th round message in the left interaction. Likewise, the **MIM** sends the i-th round message on the left immediately after obtaining the i-th round message in the right interaction.

## 2.4 Arguments/Proofs

**Definition 2.4.1** ***Interactive Argument/Proof System [11]*** *A pair of* **ppt** *interactive algorithms* $\Pi=(\mathcal{P},\mathcal{V})$ *constitutes a proof (resp., argument) system for an* $\mathcal{NP}$*-language* $\mathcal{L}$ *that is associated with the relation* $\mathbf{Rel}_{\mathcal{L}}$*, if the following conditions hold:*

- *COMPLETENESS: For every* $x\in\mathcal{L}$ *and w such that* $\mathbf{Rel}_{\mathcal{L}}(x,w)=1$*, it holds that* $\mathcal{V}$ *accepts with probability 1.*
- *SOUNDNESS: For every algorithm* $\mathcal{P}^{\star}$ *(resp.* **ppt** *algorithm* $\mathcal{P}^{\star}$*) there exist a negligible function negl such that for every* $x\notin\mathcal{L}$ *and every auxiliary input aux:*

$$Pr\left[Out_{\mathcal{V}}\langle\mathcal{P}^{\star}(aux),\mathcal{V}(x)=1\rangle\right]\leq negl(|x|)$$

We characterize $\Pi=(\mathcal{P},\mathcal{V})$ as a public coin system if, at each round, the verifier $\mathcal{V}$ simply flips a predetermined number of coins, representing a random challenge, and communicates the outcome to the prover $\mathcal{P}$. The transcript generated by $\langle\mathcal{P}(w),\mathcal{V}\rangle$ is denoted as $\pi_1,\pi_2,...,\pi_{\ell}$, where $\ell$ is a constant. Additionally, we define the transcript $\tau$ of an execution $\langle\mathcal{P}(w),\mathcal{V}\rangle(x)$ as **accepting** if the output of the verifier, $Out_V(\langle\mathcal{P}(w),\mathcal{V}\rangle(x))$, equals 1. In the subsequent discussion, we focus on the specific scenario where the number of rounds in $\Pi$ is limited to 3, that is, $\ell=3$. Recall the following definition.

**Definition 2.4.2** ***Proof of Knowledge with Canonical Extractability [11]*** *A 3-round proof system* $\Pi=(\mathcal{P},\mathcal{V})$ *for an* $\mathcal{NP}$*-language* $\mathcal{L}$ *that is associated with the relation* $\mathbf{Rel}_{\mathcal{L}}$ *as defined above, is a proof of knowledge with canonical extractor if, for all* $k\in N$*, there exists an expected* **ppt** *extractor* ***Ext*** *such that, if* $\mathcal{P}^{\star}$ *interacting with* $\mathcal{V}$ *produces an accepting transcript for* $x\in\mathcal{L}$ *with non-negligible probability, then:*

- *On input x and accepting transcript* $(\pi^1,\pi^2,\pi^3)$*,* ***Ext*** *rewinding multiple times* $\mathcal{P}^{\star}$ *and playing each time a new random challenge obtains with overwhelming probability a constant number k of accepting transcripts all with the same* $\pi^1$ *and different challenges;*
- *On input k accepting transcripts* $(x,\pi^1,\pi^2,\pi^3)_{i\in[k]}$ *such that for each* $j,z\in[k]$*,* $j\neq z,\pi^2_j\neq\pi^2_z$*,* ***Ext*** *with probability 1 and a strict polynomial number of steps outputs a witness w such that* $Rel_{\mathcal{L}}(x,w)=1$

**Definition 2.4.3** ***SHVZK*** *A 3-round proof (resp., argument) system* $\Pi=(\mathcal{P},\mathcal{V})$ *as defined above, is special honest-verifier zero knowledge (SHVZK) if there exists a* **ppt** *algorithm* ***Sim*** *that for any* $x\in\mathcal{L}$*, where* $\mathcal{L}$ *is an* $\mathcal{NP}$*-language with the associated*

*relation $Rel_{\mathcal{L}}$, security parameter $\lambda$ and any challenge $\pi^2$ works as follow:* $(\pi^1, \pi^3) \leftarrow Sim(1^\lambda, x, \pi^2)$. *Furthermore, the distribution of the output of* ***Sim*** *is (computationally) indistinguishable from the distribution of a transcript obtained when $\mathcal{V}$ sends $\pi^2$ as challenge and $\mathcal{P}$ runs on common input x and any w such that* $Rel_{\mathcal{L}}(x,w) = 1$.

## 2.5 Non-Malleable Interactive Arguments/Proofs

Consider an interactive argument/proof system $\Pi = (\mathcal{P}, \mathcal{V})$ designed for an $\mathcal{NP}$ language $\mathcal{L}$, where the corresponding relation is denoted as $Rel_{\mathcal{L}}$. Let $x \in \mathcal{L}$, with $|x| = \lambda$, serving as the public input for the protocol. Additionally, let $w$ represent $\mathcal{P}$'s private input such that $Rel_{\mathcal{L}}(x,w) = 1$. Now, introduce **MIM** as a **ppt MIM** adversary simultaneously engaged in a left session and a right session. **MIM** receives auxiliary input $aux \in 0,1^\star$ [11].

In the left session, **MIM** assesses the validity of the statement $x$ by interacting with $\mathcal{P}$, employing an identity id of **MIM**'s choosing. In the right session, **MIM** substantiates the validity of the statement $\tilde{x}$ (dynamically chosen by **MIM**) to the honest verifier $\mathcal{V}$, utilizing an identity $\tilde{id}$ selected by **MIM**. The random variable $viewMIM(x, aux)$ is employed to describe the view of **MIM** in the aforementioned experimental setup.

**Definition 2.5.1** ***Non-Malleable Zero Knowledge [11]*** *An argument/proof system $\Pi = (\mathcal{P}, \mathcal{V})$ for an $\mathcal{NP}$ -language $\mathcal{L}$ with witness relation $Rel_{\mathcal{L}}$ is nonmalleable zero knowledge (NMZK) if for any man-in-the-middle adversary* ***MIM*** *that participates in one left session and one right session, there exists an expected* **ppt SIM**$(x, aux)$ *such that:*

1. *The two ensembles* $\{\mathbf{SIM}^1(x, aux)\}_{\lambda \in \mathbb{N}, z \in \{0,1\}^\star}$ *and* $\{view^{MIM}(x, aux)\}_{\lambda \in \mathbb{N}, z \in \{0,1\}^\star}$ *are computationally indistinguishable over $\lambda$, where $Sim^1(x, aux)$ denotes the first output of $Sim(x, aux)$.*
2. *Let $(view, \tilde{w})$ denote the output of $Sim(x, aux)$, for some $aux \in 0,1^\star$. Let $\tilde{x}$ be the right-session instance appearing in view and let id and $\tilde{id}$ be respectively the identity used in the left and right sessions appearing in view. If the right session is accepting and $id \neq \tilde{id}$, then $\tilde{w}$ is $s.t.(\tilde{x}, \tilde{w}) \in Rel_{\mathcal{L}}$.*

# Chapter 3

# Protocols

In this chapter we give specifications of the protocols we implemented, their properties and the intuition for their security. We refer the reader to the *Black-Box (and Fast) Non-Malleable Zero Knowledge* paper for detailed proofs of their properties and security [11]. The protocols are presented as in the paper.

## 3.1 Commitment Scheme $\Pi^{3R}_{BGRRV} = (\mathcal{C}^{3R}_{BGRRV} + \mathcal{R}^{3R}_{BGRRV})$

We revisit the non-malleable commitment scheme outlined in [13, 33]. This scheme involves a three-round public-coin commitment, succeeded by a zero-knowledge proof to verify the integrity of the committed phase. In Protocol 1, we have the initial stage of this protocol, specifically the commitment phase, as detailed in [38]. We represent this phase as $\Pi^{3R}_{BGRRV} = (\mathcal{C}^{3R}_{BGRRV} + \mathcal{R}^{3R}_{BGRRV})$. The commitment phase of $\Pi^{3R}_{BGRRV}$ utilizes a statistically-binding commitment scheme $\Pi = (Com, Dec)$ in a black-box manner.

---

**Protocol 1** Description of $\Pi^{3R}_{BGRRV} = (\mathcal{C}^{3R}_{BGRRV} + \mathcal{R}^{3R}_{BGRRV})$ [11]

---

PUBLIC PARAMETERS: An identity $id \in \{0,1\}^k$, a large prime $q$, an integer $\ell$, and vector spaces $V_1, ..., V_n \subset \mathbb{Z}_q^\ell$ derived from $id$. These parameters satisfy the following relation: $\ell = 2(k+1)$ and $n = k+1$.

PRIVATE INPUT: $\mathcal{C}^{3R}_{BGRRV}$ holds a private message $\mathbf{m} \in \mathbb{Z}_q^{\ell-1}$, where $\mathbf{m} = (m_1, ..., m_{\ell-1})$

*Commitment Phase:* It consists of the following steps.

**Round 1** ($\mathcal{C}^{3R}_{BGRRV} \rightarrow \mathcal{R}^{3R}_{BGRRV}$).

1. $\mathcal{C}^{3R}_{BGRRV}$ picks at random values $r_1, ..., r_n \in \mathbb{Z}_q$ and $s_1, ..., s_{\ell-1}, s'_1, ..., s'_n$ in $\{0,1\}^\lambda$. This defines vectors $z_1, ..., z_n \in \mathbb{Z}_q^\ell$ where $z_i = (r_i, m)$.
2. Let $cm_m = (Com(m_1; s_1), ..., Com(m_{\ell-1}; s_{\ell-1}))$ and $cm_r = (Com(r_1; s'_1), ..., Com(r_n; s'_n))$, $\mathcal{C}^{3R}_{BGRRV}$ sends $cm = (cm_m, cm_r)$ to $\mathcal{R}^{3R}_{BGRRV}$

**Round 2** ($\mathcal{R}^{3R}_{BGRRV} \rightarrow \mathcal{C}^{3R}_{BGRRV}$). Upon receiving $cm$ from $\mathcal{C}^{3R}_{BGRRV}$, $\mathcal{R}^{3R}_{BGRRV}$ picks at random challenge vector $\alpha = (\alpha_1, ..., \alpha_n)$, where $\alpha_i \in V_i \subset \mathbb{Z}_q^\ell$ and send $\alpha$ to $\mathcal{C}^{3R}_{BGRRV}$.

**Round 3** ($\mathcal{C}^{3R}_{BGRRV} \rightarrow \mathcal{R}^{3R}_{BGRRV}$). Upon receiving $\alpha$ from $\mathcal{R}^{3R}_{BGRRV}$, $\mathcal{C}^{3R}_{BGRRV}$ sends evaluations $w_i$, for $i \in [n]$, where each $w_i = \langle \alpha_i, z_i \rangle \in \mathbb{Z}_q$.

*Decommitment Phase:* ($\mathcal{C}^{3R}_{BGRRV} \rightarrow \mathcal{R}^{3R}_{BGRRV}$). $\mathcal{C}^{3R}_{BGRRV}$ sends $\mathbf{m}$ and values $r_1, ..., r_n$ and $s_1, ..., s_{\ell-1}, s'_1, ..., s'_n$ to $\mathcal{R}^{3R}_{BGRRV}$. $\mathcal{R}^{3R}_{BGRRV}$ checks that $cm_m = (Com(m_1; s_1), ..., Com(m_{\ell-1}; s_{\ell-1}))$ and $cm_r = (Com(r_1; s'_1), ..., Com(r_n; s'_n))$.

---

This protocol satisfy what *weak non-malleable commitment*, which is specified by the following theorem proven in [33].

**Theorem 1** *[33] Let **MIM** be a MiM* **ppt** *non-uniform adversary as defined in Definition 2.3.1 which interacts (in a synchronous way) with an honest sender in the left session and an honest receiver in the right session. Let* $\tau$ *be the transcript obtained at the end of this interaction and* $\tilde{m}$ *the message committed by **MIM** in the right session. Let* $\tilde{p}$ *be the probability with which **MIM** is successful in the indistinguishability game defined in Definition 2.3.1.*

*Then there exists a* **ppt** ***Ext*** *which has oracle access to **MIM** s.t.:*

$$\Pr_{\tau}\left[Ext^{MIM}(\tau) \neq \tilde{m} \middle| \tau \in ACC\right] \leq \tilde{p}/2$$

*where the probability is over the randomness of **Ext** and **ACC** denotes the set of well-formed transcripts of the commitment phase.*

Commentary on the application of $\Pi^{3R}_{BGRRV}$: In $\Pi^{3R}_{BGRRV}$, the commitment made in the initial round is calculated based on a message representing a tuple of elements in $\mathbb{Z}_q$, denoted as $m = (m_1, ..., m_{\ell-1}) \in \mathbb{Z}_q^{\ell-1}$. Looking ahead, our use of $\Pi^{3R}_{BGRRV}$ will require committing to a singular element of $\mathbb{Z}_q$, aligning with the approach in Protocol 2 of [38]. Let $m \in \mathbb{Z}_q$ be the message to be committed in $\Pi_{NM}$; we can express this as $m = (m, m_2, ..., m_{\ell-1})$, with $m_2, ..., m_{\ell-1}$ all set to 0.

## 3.2 Extractable Commitment Scheme

Within this section, we revisit the 3-round public-coin extractable commitment denoted as $\Pi_{3Ext} = (\mathcal{C}_{3Ext}, \mathcal{R}_{3Ext})$, which is elaborated upon in section 4 of [44]. The details of the extractable commitment $\Pi_{3Ext}$ are described in Protocol 2, with **Com** representing the commitment phase of a non-interactive statistically-binding commitment scheme.

**Protocol 2** Description of $\Pi_{3Ext} = (\mathcal{C}_{3Ext}, \mathcal{R}_{3Ext})$ [11]

NOTATION: Let **Com** be a statistically-binding commitment scheme.
PUBLIC PARAMETERS: $\lambda$.
PRIVATE INPUT: $\mathcal{C}_{3Ext}$ holds a private message $\mathbf{m} \in \{0,1\}^{\lambda}$.
*Commitment Phase:* It consists of the following steps.
**Round 1** ($\mathcal{C}_{3Ext} \rightarrow \mathcal{R}_{3Ext}$). $\mathcal{C}_{3Ext}$ commits using **Com** to $\lambda$ pairs of strings $\{v_i^0, v_i^1\}_{i \in [\lambda]}$, such that, for each $i \in [\lambda]$, $(v_i^0, v_i^1) = (v_i, m \oplus v_i)$ and $v_i$ are random strings in $\{0,1\}^{|m|}$. Let **c** be the list of the resulting $\lambda$ pairs of commitments, $\mathcal{C}_{3Ext}$ sends **c** $\mathcal{R}_{3Ext}$.
**Round 2** ($\mathcal{R}_{3Ext} \rightarrow \mathcal{C}_{3Ext}$). Upon receiving **c** from $\mathcal{C}_{3Ext}$, $\mathcal{R}_{3Ext}$ sends a random challenge $e = (e_1, ..., e_\lambda)$, with each $e_i \in \{0,1\}$ for $i \in [\lambda]$
**Round 3** ($\mathcal{C}_{3Ext} \rightarrow \mathcal{R}_{3Ext}$). Upon receiving **e** from $\mathcal{R}_{3Ext}$, $\mathcal{C}_{3Ext}$ opens the commitments $v_i^{e^i}$, $i \in [\lambda]$
*Decommitment Phase:* $\mathcal{C}_{3Ext} \rightarrow \mathcal{R}_{3Ext}$. $\mathcal{C}_{3Ext}$ sends **m** and the openning for all $\lambda$ pairs of strings. $\mathcal{R}_{3Ext}$ checks that

1. The openning is the same as the commitment phase with respect to **e**
2. $m = v_1^0 \oplus v_1^1 = ... = v_\lambda^0 \oplus v_\lambda^1$

Protocol 2 satisfies the following theorem.

**Theorem 2** *Let* ***Com*** *be a statistically-binding commitment scheme, then* $\Pi_{3Ext}$ *is a 3-round public-coin extractable statistically-binding commitment scheme that achieves 2-extractability.*

## 3.3 Weak Non-Malleable Extractable Commitment Scheme

We formulate a five round weak non-malleable extracted commitment scheme $\Pi_{5Ext} = (\mathcal{C}, \mathcal{R})$ [11]. This is a protocol with delayed-input, where the message to be committed can be committed in the last round. The protocol is built from the following:

1. The 3-round public-coin, statistically binding, weak-non-malleable commitment scheme $\Pi^{3R}_{BGRRV}$ that was specified in Section 3.1.
2. The 3-round 2-extractable, statistically binding, commitment scheme $\Pi_{3Ext}$ presented in Section 3.2.
3. A non-interactive statistically binging commitment $\Pi_{com} = (Com, Dec)$

At a broad level, our protocol $\Pi_{5Ext}$, illustrated in Protocol 3, aligns with the commitment phase of $\Pi^{3R}_{BGRRV}$ (outlined in Section 3.1). In the initial round, this protocol takes a vector of $\ell$ elements as input and commits to it component-wise using **Com**. The only modification we introduce is committing to the first component of this vector using $\Pi_{3Ext}$. Additionally, the delayed-input property is achieved by employing the technique from [17], where a two-out-of-two secret sharing of the committed message $m$ is computed. Specifically, $\mathcal{C}$ initially commits to a random share $k$, and in the final round, it sends the other share $c = m \oplus k$ in the clear.

**Protocol 3** Description of $\Pi_{5Ext} = (\mathcal{C}, \mathcal{R})$ [11]

NOTATION: We denote by $(ext_1, ext_2, ext_3)$ the 3 rounds of $\Pi_{3Ext}$. We denote by $(cm = (cm_m, cm_r), \alpha, a)$ the 3 messages of $\Pi^{3R}_{BGRRV}$.
PUBLIC PARAMETER: $\lambda$.
$\mathcal{C}$'s PRIVATE INPUT: $m \in \mathbb{Z}_q$, $m$ can be received at round 5.
*Commitment Phase:* It consists of the following steps.
**Round 1** ($\mathcal{C} \to \mathcal{R}$).
$\mathcal{C}$ samples a random share $k \in \mathbb{Z}_q$ and sets $m = (k, m_2, ..., m_{\ell-1})$, where $m_i = 0$ for each $i \in \{2, ..., \ell-1\}$.
$\mathcal{C}$ picks at random values $r_1, ..., r_n \in \mathbb{Z}_q$ and $s_1, ..., s_{\ell-1}, s'_1, ..., s'_n$ in $\{0,1\}^\lambda$. This defines vectors $z_1, ..., z_n \in \mathbb{Z}_q^{\{}\ell\}$ where $z_i = (r_i, m)$.
$\mathcal{C}$ computes the first round of $\Pi_{3Ext}$ w.r.t. message $k$ obtaining $ext_1$.
Let $cm_m = (ext_1, Com(m_2; s_2), ..., Com(m_{\ell-1}; s_{\ell-1}))$ and $cm_r = (Com(r_1; s'_1), ..., Com(r_n; s'_n))$, $\mathcal{C}$ sends $cm = (cm_m, cm_r)$ to $\mathcal{R}$.
**Round 2** ($\mathcal{R} \to \mathcal{C}$).
Upon receiving $cm$, $\mathcal{R}$ computes the 2nd round $\alpha$ of $\Pi^{3R}_{BGRRV}$, namely $\mathcal{R}$ picks at random challenge vector $\alpha = (\alpha_1, ..., \alpha_n)$, where $\alpha_i \in V_i \subset \mathbb{Z}_q^\ell$ and send $\alpha$ to $\mathcal{C}$.
**Round 3** ($\mathcal{C} \to \mathcal{R}$).
Upon receiving $\alpha$, $\mathcal{C}$ computes the 3rd round of $\Pi^{3R}_{BGRRV}$, namely $\mathcal{C}$ sends evaluations $w_i$, for $i \in [n]$, where each $w_i = \langle \alpha_i, z_i \rangle \in \mathbb{Z}_q$.
**Round 4** ($\mathcal{R} \to \mathcal{C}$).
$\mathcal{R}$ Upon receiving $\alpha$, computes the second round $ext_2$ of $\Pi_{3Ext}$. $\mathcal{R}$ sends $ext_2$ to $\mathcal{C}$.
**Round 5** ($\mathcal{C} \to \mathcal{R}$).
Upon receiving ext2, $\mathcal{C}$ computes the third round $ext_3$ of $\Pi_{3Ext}$ obtaining decommitment information $dec_{ext}$. $\mathcal{C}$ sends $ext_3$, $c$ to $\mathcal{R}$, where $m = c \oplus k$.
*Decommitment Phase:* ($\mathcal{C} \to \mathcal{R}$) $\mathcal{C}$ sends to $\mathcal{R}$ the decommitment value $dec = (m, (r_1, ..., r_n), (dec_{ext}, s_1, ..., s_{\ell-1}, s'_1, ..., s'_n))$. If $((cm, \alpha, a), m, (r_1, ..., r_n), (dec_{ext}, s_1, ..., s_{\ell-1}, s'_1, ..., s'_n))$ is a valid decommitment for $\Pi^{3R}_{BGRRV}$, then $\mathcal{R}$ obtains the opening computing $m = c \oplus k$

We have the following theorem about $\Pi_{5Ext}$.

**Theorem 3** *The protocol $\Pi_{5Ext}$ described in Protocol 3 is a statistically binding, computationally hiding 5-round delayed-input extractable weak-non-malleable commitment scheme, that achieves 2-extractability.*

## 3.4 Black-Box Non-Malleable Zero Knowledge

In this section we give the description of the black-box non-malleable zero-knowledge (NMZK) argument system $\Pi_{NM}$ [11].

$\Pi_{NM}$ is composed of

1. The 5-round public-coin 2-extractable weak-non-malleable delayed-input commitment scheme $\Pi_{5Ext} = (\mathcal{C}, \mathcal{R})$ described in Section 3.3
2. A 3-round public-coin SHVZK proof of knowledge $\Pi = (\mathcal{P}, \mathcal{V})$ for the language

$\mathcal{L}$ and relation $Rel_{\mathcal{L}}$

**Protocol 4** Description of $\Pi_{NM} = (\mathcal{P}_{NM}, \mathcal{V}_{NM})$[11]

NOTATION: Let $\Pi = (\mathcal{P}, \mathcal{V})$ be a 3-round public-coin SHVZK proof of knowledge for the language $\mathcal{L}$ and associated relation $Rel_{\mathcal{L}}$. Let $(\pi^1, \pi^2, \pi^3)$ denote the three round messages of $\Pi$. Let $\Pi_{5Ext}$ be the 5-round delayed input weak-non-malleable 2-extractable commitment scheme defined in Section 3.3. Let us denote with $(com_1, com_2, com_3, com_4, com_5)$ the transcript of the commit phase of $\Pi_{5Ext}$ and with $dec$ the corresponding opening information. For simplicity, we will implicitly assume that the same identity used in a session of $\Pi_{NM}$ is also used for the execution of $\Pi_{5Ext}$ inside the session of $\Pi_{NM}$.
PUBLIC PARAMETER: $\lambda$, $x \in \mathcal{L}$
$\mathcal{P}_{NM}$'s PRIVATE INPUT: $w$ such that $(x, w) \in Rel_{\mathcal{L}}$
**Round 1.** $\mathcal{P}_{NM}$ on input $(x, w)$, computes the 1st round $\pi^1$ of $\Pi$ and sends $\pi^1$ to $\mathcal{V}_{NM}$.
**Round 2.** $\mathcal{V}_{NM}$ computes the 1st round $com_1$of $\Pi_{5Ext}$ (being delayed input, there is no input to provide at this stage). $\mathcal{V}_{NM}$ sends $com_1$ to $\mathcal{P}_{NM}$.
**Round 3.** On input $com_1$, $\mathcal{P}_{NM}$ sends the 2nd round $com_2$ of $\Pi_{5Ext}$ and sends $com_2$ to $\mathcal{V}_{NM}$.
**Round 4.** On input $com_2$, $\mathcal{V}_{NM}$ computes the 3rd round $com_3$ of $\Pi_{5Ext}$ and sends $com_3$ to $\mathcal{P}_{NM}$.
**Round 5.** On input $com_3$, $\mathcal{P}_{NM}$ sends the 4nd round $com_4$ of $\Pi_{5Ext}$ and sends $com_4$ to $\mathcal{V}_{NM}$.
**Round 6.** $\mathcal{V}_{NM}$ samples $c_1 \leftarrow \{0,1\}^{\lambda}$. On input $com_4$, $\mathcal{V}_{NM}$ computes the 5th round $com_5$ and corresponding opening information $dec$ of $\Pi_{5Ext}$ w.r.t. message $c_1$ (i.e., the committed message is $c_1$). $\mathcal{V}_{NM}$ sends $com_5$ to $\mathcal{P}_{NM}$.
**Round 7.** $\mathcal{P}_{NM}$ samples $c_2 \leftarrow \{0,1\}^{\lambda}$ and sends $c_2$ to $\mathcal{V}_{NM}$.
**Round 8.** $\mathcal{V}_{NM}$ sends $(c_1, dec)$ to $\mathcal{P}_{NM}$.
**Round 9.** On input $(c_1, dec)$, $\mathcal{P}_{NM}$ acts as follows. If $dec$ is not a valid opening for $(com_1, com_2, com_3, com_4, com_5)$ w.r.t. committed message $c_1$ then $\mathcal{P}_{NM}$ aborts. Otherwise, $\mathcal{P}_{NM}$ sets $\pi_2 = c_1 \oplus c_2$ and computes the 3rd round $\pi_3$ of $\Pi$. $\mathcal{P}_{NM}$ sends $\pi_3$ to $\mathcal{V}_{NM}$.
**Verification Procedure** On input $\pi_3$, $\mathcal{V}_{NM}$ computes $\pi^2 = c_1 \oplus c_2$. If $(x, \pi^1, \pi^2, \pi^3)$ is an accepting transcript for $\Pi$, then $\mathcal{V}_{NM}$ accepts, otherwise it aborts.

The protocol satisfies the following theorem.

**Theorem 4** *Given a 3-round, public-coin SHVZK proof of knowledge $\Pi = (\mathcal{P}, \mathcal{V})$ with canonical extractor for an $\mathcal{NP}$ language $\mathcal{L}$, the 5-round public-coin 2-extractable delayed-input commitment scheme $\Pi_{5Ext}$ of Section 3.3, the protocol $\Pi_{NM}$ is a black-box non-malleable zero-knowledge argument system for $\mathcal{L}$ which makes black-box use of $\Pi_{5Ext}$ and $\Pi$.*

The protocol is also simulation extractable. We refer interested reader to the paper *Black-Box (and Fast) Non-Malleable Zero Knowledge* for detailed proofs of the theorems above and a description of the simulator.

The protocols are presented as implementation references.

# Chapter 4

# Implementations

## 4.1 Implementation Requirements

Our objective is to realize efficient and secure implementations of the protocols outlined in Section 3. Moreover, we aim for our codebase to be accessible and usable by other developers. In this chapter, we present the implementation details and design choices we made other than following the protocol specifications.

## 4.2 Implementing $\Pi_{com} = (Com, Dec)$

This section provides an overview of the Naor Commitment [42], a specific instantiation of the black box non-interactive statistically binding commitment scheme $\Pi_{com}$. It covers the error correcting code employed by Naor Commitment, the associated algorithms, and explores how the choice of the field $\mathbb{F}_p$ is influenced by efficiency considerations for the algorithm.

### 4.2.1 Naor Commitment As A Concrete Implementation

Recall that $\Pi^{3R}_{BGRRV}$, $\Pi_{3Ext}$, and $\Pi_{5Ext}$ use a non-interactive statistically binding commitment scheme denoted as $\Pi_{com} = (Com, Dec)$. In practice, we instantiate this commitment scheme using the Naor Commitment [42].

The work [42] introduces two protocols. In Section 3, the author describes the *Bit-Commitment Protocol*, and in Section 4, the *Commitment to Many Bits Protocol* is presented. Both protocols have been implemented in our codebase for completeness. However, in practical scenarios, we opt for the *Commitment to Many Bits Protocol* due to its efficiency comparing to the *Bit-Commitment Protocol*.

**Protocol 5** Description of Bit-Commitment Protocol [42]

PUBLIC PARAMETERS:

$n$: security parameter. chosen so that no adversary can break the pseudorandom generator for seeds of length $n$.

NOTATIONS:

$B_i(s)$: The ith bit of the pseudorandom sequence on seed s.

*Commit Stage*:

1. Bob selects a random vector $R = (r_1, r_2, ..., r_{3n})$ where $r_i \in \{0,1\}$ for $1 \leq i \leq 3n$ and send it to Alice.
2. Alice selects a seed $s \in \{0,1\}^n$ and sends to Bob the vector $D = (d_1, d_2, ..., d_{3n})$ where
   if $r_i = 0$
   $$d_i = B_i(s)$$
   if $r_i = 1$
   $$d_i = B_i(s) \oplus b$$

*Reveal Stage*: Alice sends $s$ and Bob verifies that, for all $1 \leq i \leq 3n$, if $r_i = 0$, then $d_i = B_i(s)$, and if $r_i = 1$, then $d_i = B_i(s) \oplus b$

It is evident that while this protocol is straightforward, its efficiency is compromised as bits are committed one at a time.

Consider the following protocol, which is significantly more efficient.

**Protocol 6** Description of Commitment to Many Bits Protocol [42]

PUBLIC PARAMETERS:

$n$: security parameter. chosen so that no adversary can break the pseudorandom generator for seeds of length $n$.

$m$: number of bits to be committed.

NOTATIONS:

$C \subset \{0,1\}^q$ is a code of $2^m$ words such that the hamming distance between any $c_1, c_2 \in C$ is at least $\varepsilon \cdot q$.

$E : \{0,1\}^m \to \{0,1\}^q$ is an efficient computable function for mapping words in $\{0,1\}^m$ to $C$.

$R = (r_1, r_2, ..., r_k)$ with $r_i \in \{0,1\}$ and with exactly $q$ indices $i$ such that $r_i = 1$.

Let $G_R(s)$ denote the vector $A = (a_1, a_2, ..., a_q)$ where $a_i = B_{j(i)}(s)$ and $j(i)$ is the index of the ith 1 in $R$.

*Commit Stage*:

1. Bob selects a random vector $R = (r_1, r_2, ..., r_{2q})$ where $r_i \in \{0,1\}$ for $1 \leq i \leq 2q$ and exactly $q$ of the $r_i$'s are 1 and send it to Alice.
2. Alice computes $c = E(b_1, b_2, ..., b_m)$. Alice selects a seed $s \in \{0,1\}^n$ and sends to Bob $e = c \oplus G_R(s)$, and for each $1 \leq i \leq 2q$ such that $r_i = 0$ she sends Bob $B_i(s)$.

*Reveal Stage*:

Alice sends $s$ and $b_1, b_2, ..., b_m$. Bob verifies that for all $1 \leq i \leq 2q$ such that $r_i = 0$ Alice had sent the correct $B_i(s)$, computes $c = E(b_1, b_2 ..... b_m)$ and $G_R(s)$, and verifies that $e = c \oplus G_R(s)$

In protocol 6, we have a non-interactive statistically binding commitment scheme that can commits to many bits efficiently. One obstacle remaining is that it uses a blackbox error correcting code $C$ that must satisfy $q \cdot \log(\frac{2}{2-\varepsilon}) \geq 3n$ and $\frac{q}{m}$ must be a constant [42]. We take the suggestion from the paper and implemented Justesen code which satisfies this property [36].

### 4.2.2 Justesen Code

The Justesen code is a concatenation code that combines an outer code, namely the **Reed-Solomon code** [47], with an inner code known as the **Wozencraft ensemble** [41].

**Definition 4.2.1** *[41] Let $\mathbb{F}_p$ be a field. the **Wozencraft ensemble** is defined as a function $f$ that takes an element $x \in \mathbb{F}_p$ and return the codeword $(x, \alpha x)$, where $\alpha$ is the primitive element of the field*

$$f(x) = (x, \alpha x)$$

In practice, the primitive element $\alpha$ is chosen to be the generator of the multiplicative subgroup of the field $\mathbb{F}_p$.

We use the definition of **Reed-Solomon code** from the book *Essential Coding Theory* [35].

**Definition 4.2.2** *[35] Let $\mathbb{F}_p$ be a finite field, and choose $n$ and $k$ satisfying $k \leq n \leq q$. Fix a sequence $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ of $n$ distinct elements (also called evaluation points) from $\mathbb{F}_q$. We define an encoding function for Reed-Solomon code $RS_q[\alpha, k] : \mathbb{F}_p^k \to \mathbb{F}_p^n$ as follows. Map a message $m = (m_0, m_1, ..., m_{k_1})$ with $m_i \in \mathbb{F}_q$ to the degree $k-1$ polynomial*

$$m \to f_m(X)$$

*where*

$$f_m(X) = \sum_{i=0}^{k-1} m_i X^i.$$

We choose the rate of the **Reed-Solomon code** to be $\frac{1}{2}$. This means that the code word will be twice as long as the original message.

The FFT algorithm is used to transform the polynomial from coefficient form to evaluation form and the inverse FFT algorithm is employed to perform the reverse [19]. The code can be found in Appendix A.

The inner code **Wozencraft ensemble** is implemented with a simple loop that adds one element $\alpha x$ to each element. Notice that this process is easily parallelizable.

### 4.2.3 Choosing A Finite Field $\mathbb{F}_p$ For Faster FFT

Since we are using the FFT algorithm over the field $\mathbb{F}_p$, it is more efficient if $\mathbb{F}_p$ have high two-dicity. This means that the field should have a multiplicative subgroup of order

$2^n$, where $n$ is an integer. High two-adicity means $n$ should be relatively big, usually double digits.

Various fields have been specifically designed to meet this criterion. For instance, the Goldilocks Field, introduced in [45], has $p = 2^{64} - 2^{32} + 1$. This $p$ is a proth prime [50], which has the form $N = k \cdot 2^n + 1$. This prime number can be written in the form $p = (2^{32} - 1) * 2^{32} + 1$, which means that the Goldilocks Field has a multiplicative subgroup of order $2^{32}$.

However, as we will later see in the paper, we also require the field $\mathbb{F}_p$ to be the scalar field of an elliptic curve. We choose the field to be the scalar field of the elliptic curve BLS12-377, which has

$$p = 12ab655e9a2ca55660b44d1e5c37b00159aa76fed00000010a11800000000001$$

in hexadecimal representation. This is a curve introduced in [12]. The two-adicity of this field is 46.

### 4.2.4 Usage of Cryptographically Secure Algorithms

Ensuring robust security is a fundamental necessity in our implementation, especially when it comes to cryptography protocol implementations. Aligned with the guidelines outlined in the National Institute of Standards and Technology (NIST) reports [6], our objective is to achieve a security level of 128 bits. This section will delve into the imperative need to not only attain a high level of security but also incorporate cryptographically secure shuffling algorithms and seeded random number generators into our implementation.

#### 4.2.4.1 Fisher-Yates Shuffling

Recall the specified stage within the commit phase of Protocol 6

1. Bob selects a random vector $R = (r_1, r_2, ..., r_{2q})$ where $r_i \in \{0, 1\}$ for $1 \leq i \leq 2q$ and exactly $q$ of the $r_i$'s are 1 and send it to Alice.

We need to randomly generate a vector $R$ of fixed length with respect to $q$, where precisely half of the elements are '0's and the remaining half are '1's. To achieve this, the initial step involves generating a vector $R$ containing the elements, followed by utilizing a cryptographically secure shuffling algorithm to shuffle them.

We picked the Fisher-Yates Shuffling Algorithm [22, 26]. The specifics of implementing this algorithm fall outside the scope of this paper, as we utilize the algorithm by importing a library.

#### 4.2.4.2 ChaCha8 Seeded Random Generator

Both Protocol 5 and 6 uses

- $B_i(s)$ : The ith bit of the pseudorandom sequence on seed $s$

To implement this, it is necessary to use a Seeded Random Generator. We utilize ChaCha8, which has a 256-bit seed and provides 256 bits of security [9, 48].

## 4.3 Parameters for $\Pi^{3R}_{BGRRV}$

Recall that we have the following public parameters for the $\Pi^{3R}_{BGRRV}$ protocol

1. $id \in \{0,1\}^k$
2. A large prime $q$
3. An integer $\ell$
4. Vector Spaces $V_1, ..., V_n \subset \mathbb{Z}_q^\ell$ derived from $id$

satisfying

- $\ell = 2(k+1)$
- $n = k+1$
- length of message $m = \ell - 1$

$id$ is a $k$ bit identity tag, with the parameters specified in [13], $k$ is set to be 16 or 32. We chose $k = 32$ in our implementation. The chosen value of $k$ holds notable performance implications for the protocol. The table below provides details on the selected $k$ value, the corresponding runtime of our implementation in the synchronous setting and the maximum length of message the tag can handle (denoted by $\beta$). $\lambda$ is the security parameter

Using Schnorr's $\Sigma$-Protocol to prove knowledge of discrete logarithm [49]:

| $(k, \lambda)$ | $\mathcal{P}$ time (ms) | $\mathcal{V}$ time (ms) | Comm. (MB) | $\beta$ ($\mathbb{F}$) |
|---|---|---|---|---|
| (16, 128) | 2.0145 | 1.9786 | 0.094698 | 33 |
| (32, 128) | 6.2036 | 5.9830 | 0.351798 | 65 |
| (64, 128) | 23.063 | 22.576 | 1.358058 | 129 |
| (128, 128) | 94.313 | 89.533 | 5.336298 | 257 |

Using ZKBoo to prove preimage of Sha256 hash function [29]:

| $(k, \lambda)$ | $\mathcal{P}$ time (ms) | $\mathcal{V}$ time (ms) | Comm. (MB) | $\beta$ ($\mathbb{F}$) |
|---|---|---|---|---|
| (16, 40) | 74.083 | 49.362 | 1.3509339 | 33 |
| (32, 40) | 79.028 | 53.355 | 1.6082139 | 65 |
| (16, 80) | 148.66 | 98.486 | 2.589146 | 33 |
| (32, 80) | 151.59 | 101.63 | 2.846426 | 65 |
| (16, 128) | 239.82 | 158.28 | 4.082284 | 33 |
| (32, 128) | 241.12 | 160.79 | 4.339564 | 65 |

The modulus of our field, denoted as $q$, is equivalent to the value of $p$ mentioned in Section 4.2.3. Given that $k = 32$, we can easily compute the remaining parameters through arithmetic calculations. Specifically, we have $\ell = 66$, $n = 33$, and the length of the message $m = 65$.

In alignment with the adjustment made in our final protocol $\Pi_{NM}$ – where the first element of the message is committed using $\Pi_{5Ext}$ instead of $\Pi_{BGRRV}$ – we decrease the

length of the message $m$ by 1, resulting in a new length of 64.

## 4.4 Concrete Implementation of $\Pi = (\mathcal{P}, \mathcal{V})$

In the final protocol, we make use of a black box $\Pi = (\mathcal{P}, \mathcal{V})$, which is a 3-round public-coin SHVZK proof of knowledge for the language $\mathcal{L}$ and associated relation $Rel_{\mathcal{L}}$. This is sometimes called Σ-Protocols.

In a Σ-Protocol, Both the Prover $\mathcal{P}$ and Verifier $\mathcal{V}$ knows public parameters and the Prover proves prove to the verifier a relation $(x, w) \in Rel_{\mathcal{L}}$, with the secret witness only $\mathcal{P}$ knows [14].

The protocol starts with the Prover $\mathcal{P}$ sending an initial message $i$, the verifier $\mathcal{V}$ respond with a challenge $c$, and the $\mathcal{P}$ replies with another message $s$. $(i, c, s)$ is sufficient to convince $\mathcal{V}$ the validity of $Rel_{\mathcal{L}}$.

### 4.4.1 Schnorr's Σ-Protocol

In Schnorr's Σ-Protocol [49], the Prover $\mathcal{P}$ wants to prove knowledge of a discrete logarithm relation, namely the Prover knows some $w$ such that $g^w = y$.

The protocol is specified as follows,

**Protocol 7** Schnorr's Σ-Protocol

PUBLIC PARAMETER: A (multiplicative) cyclic group $\mathbb{G}$ with prime order $q$ and generator $g$, and $y = g^w$. $w$ is only known to $\mathcal{P}$
**Round 1** The Prover $\mathcal{P}$ samples a random $r$ such that $0 \leq r \leq |\mathbb{G}| - 1$ and send $i = g^r$ to $\mathcal{V}$
**Round 2** $\mathcal{V}$ respond with a random number $c$ such that$0 \leq c \leq |\mathbb{G}| - 1$
**Round 3** $\mathcal{P}$ respond with $s = (wc + r)$
*Verification*: Verifier checks that $i \cdot y^c = g^s$

This protocol is has perfect completeness, special soundness, and honest verifier perfect zero-knowledge. We will omit the proofs of these property here and direct interested readers to [14].

### 4.4.2 ZKBoo

ZKBoo is a general purpose Σ-Protocol that can prove any statement of the form "$\mathcal{P}$ knows a $w$ such that $f(w) = y$" [29]. The protocol is presented as in the paper,

**Protocol 8** ZKBoo

The verifier and the prover have input $y \in L_\phi$. The prover knows $x$ such that $y = \phi(x)$. A $(2,3)$-decomposition of $\phi$ is given. Let $\Pi^\star_\phi$ be the protocol related to this decomposition.
**Round 1** The Prover $\mathcal{P}$ does the following:

1. Sample random tapes $k_1$, $k_2$, $k_3$;
2. Run $\Pi^\star_\phi(x)$ and obtain the views $w_1$, $w_2$, $w_3$ and the output shares $y_1$, $y_2$, $y_3$;
3. Commit to $c_i = Com(k_i, w_i)$ for all $i \in [3]$;
4. Send $a = (y_1, y_2, y_3, c_1, c_2, c_3)$

**Round 2** The verifier $\mathcal{V}$ choose an index $e \in [3]$ and sends it to the prover.
**Round 3** the prover $\mathcal{P}$ answers to the verifier's challenge sending opening $c_e, c_{e+1}$ thus revealing $z = (k_e, w_e, k_{e+1}, w_{e+1})$.
*Verification*: Verifier checks that

1. $Rec(y_1, y_2, y_3) = y$
2. $\forall i \in \{e, e+1\}$, $y_i = Output(w_i)$
3. $\forall j$, $w_e[j] = \phi_e^{(j)}(w_e, w_{e+1}, k_e, k_{e+1})$

The protocol satisfies perfect completeness, special soundness, and honest verifier perfect zero-knowledge same as Schnorr's Σ-Protocol.

We forked the ZKBoo implementation by Geometry Research [23, 28]. The original ZKBoo implementation by Geometry is made non-interactive via the Fiat-Shamir Heuristic [25]. We removed this heuristic and made the protocol interactive. The verification algorithm was changed as well after the heuristic removal for adaptation.

## 4.5 Non-Malleable Zero Knowledge

A complete, fast, secure and reusable non-malleable zero knowledge protocol as specified in Protocol 4 is implemented. We want to remark that we have implemented both an asynchronous version of the protocol and a synchronous version of the protocol. Developers that intend to use our codebase can easily integrate setup the protocol over a network. In general we found a 5 percent decrease in performance of the protocol in the asynchronous setting, which is largely due to serialization of the messages.

### 4.5.1 Asynchronous Implementation of $\Pi_{NM}$

The following steps differ the asynchronous protocol from the synchronous protocol

1. Messages are serialized to bytes for communication.
2. Serialized bytes are sent via a channel
3. Code follows the async/.await paradigm of Rust and executed via an executor.

We chose the channel to be a mpsc (Multi-producer, single-consumer) channel and the executor to be tokio [51].

## 4.6 Miscellaneous

In this section, we explain the implementation and design decisions we made for the entire codebase.

### 4.6.1 Building the Protocol with Rust

We chose the Rust programming language to implement the protocol.

Rust is a general purpose programming language that focuses on safety and performance. On the safety side, it focuses on type safety by have algebraic data types and enforces memory safety using lifetimes [39]. On the performance side, it has similar performance with C and C++ and is 10X to 100X or more faster than Python [1, 2]. In recent years, Rust became a choice for increasing numbers of cryptographic projects in industry, providing high quality, maintained and audited libraries [4, 24, 45, 53].

### 4.6.2 Modular Design of Codebase

The codebase is organized by leveraging the Rust Module System. Each protocol is implemented in a module. Developers can choose to use each protocol individually as a library. For instance, the black box $\Pi_{com}$ in $\Pi_{5Ext}$ and $\Pi_{BGRRV}$ can easily be swapped to another commitment scheme other than Naor Commitment. Schnorr's $\Sigma$ protocol can be used independently from $\Pi_{NM}$.

### 4.6.3 Improving Code Readability

This section outline the efforts to improve code readability within our codebase.

#### 4.6.3.1 Macros That Reduce Code Duplication

In the asynchronous protocol, message sending/receiving through channels was written with Rust declarative macros to avoid code duplication. Macros are a way of writing code that writes other code, which is known as metaprogramming [39].

Only variables, types and error messages are changed in the different messages that are sent and received through channels. We wrote the following macros that take in these fields and produce the corresponding code.

```
#[macro_export]
macro_rules! send_message {
  ($s:ident, $field: tt, $m:ident, $e:literal) => {
    let mut bytes = Vec::new();
    $m.serialize_uncompressed(&mut bytes)
    .map_err(|_|
      NMZKErrors::SerializationFailure($e.to_string()))?;
    $s.$field.send(bytes).map_err(|_|
      NMZKErrors::CannotSend($e.to_string()))?;
  };
```

```
}

#[macro_export]
macro_rules! receive_message {
  ($s:ident, $field: tt, $m:tt, $e:literal, $dety:ty) => {
    let r_bytes = $s.$field.recv().await;
    let $m = match r_bytes {
      Some(bytes) => <$dety>::deserialize_uncompressed(&*bytes)
      .map_err(|_|
        NMZKErrors::DeSerializationFailure($e.to_string()))?,
      None => return Err(NMZKErrors::NotReceived($e.to_string())),
    };
    $s.$m = $m.clone();
  };
}
```

After this was implemented, sending and receiving messages can be written in a single line of code without having to repeat 5-10 lines of code in distinct rounds of the protocol.

```
pub async fn round1(&mut self, w: G::ScalarField)
  -> Result<(), NMZKErrors> {
  ...

  send_message!(self, commiter_sender, pi1, "Pi_1");

  ...
}

...

pub async fn round5(&mut self) -> Result<(), NMZKErrors> {
  receive_message!(
    self,
    commiter_receiver,
    com3,
    "Com_3",
    InnerProductEvaluation::<G::ScalarField>
  );

  ...

  send_message!(self, commiter_sender, com4, "Com_4");

  ...
}
```

#### 4.6.3.2 Type Aliases That Avoid Type Complications

Messages types are transformed with type aliases to avoid writing long types in the codebase. In round 8 of the protocol, the receiver $\mathcal{R}$ sends committed message $c_1$ and decommit message of $\Pi_{5Ext}$ $dec$. The type of this message is aliased to be

```
pub(crate) type C1Dec<S> = (
    Vec<S>,
    (Vec<S>, bgrrv::commiter::DecommitMessage<S>,
    naor_commit::naor_ext_commit::DecommitMessage<S>),
);
```

Aliasing improves readability of our codebase by allowing us to write shorter types for function signatures and message serialization.

### 4.6.4 Error Handling Design

Thiserror crate is used for error handling [21]. The following is an example of an error enum implementation in our codebase.

```
#[derive(Error, Debug)]
pub enum NMZKErrors {
  #[error("failed to serialize {0}")]
  SerializationFailure(String),

  #[error("failed to deserialize {0}")]
  DeSerializationFailure(String),

  #[error("failed to send {0}")]
  CannotSend(String),

  #[error("{0} not received")]
  NotReceived(String),

  #[error(transparent)]
  SchnorrErrors(#[from] SchnorrErrors),

  #[error(transparent)]
  WEXTErrors(#[from] WEXTErrors),
}
```

This enum allows easy importation of error types for developers that indent to use our codebase, which allow flexibility of errors handling in their projects.

### 4.6.5 Testing & Profiling

We have employed both unit and integration tests of the entire protocol $\Pi_{NM}$ and its subcomponents. Here is an example snapshot of our asynchronous integration protocol test.

```
#[tokio::test]
async fn nmzk_protol_test() {
  use ark_bls12_377::{Fr, G1Affine};
  use ark_ff::UniformRand;
  let (mut c, mut r) = Protocol::<G1Affine>::init().unwrap();

  let mut rng = rand::thread_rng();
  let w = Fr::rand(&mut rng);
  c.round1(w).await.unwrap();
  r.round2().await.unwrap();
  c.round3().await.unwrap();
  r.round4().await.unwrap();
  c.round5().await.unwrap();
  r.round6().await.unwrap();
  c.round7().await.unwrap();
  r.round8().await.unwrap();
  c.round9().await.unwrap();
  r.receive_pi3().await.unwrap();
  r.verify().unwrap();
}
```

These tests revealed a number of bugs in our implementation. One of which is an xor bug of field elements $\mathbb{F} \otimes \mathbb{F}$. We were packing bits in field elements $\mathbb{F}$ and xoring them occasionally result in an element outside of the field. We have substituted xor with addition and subtraction to avoid “field overflow”.

# Chapter 5

# Concluding Remarks

## 5.1 Division of Work

My collaborator Xuhan Zhang implemented the initial version of $\Pi_{3Ext}$ and Schnorr's $\Sigma$ Protocol, while I implemented $\Pi^{3R}_{BGRRV}$, $\Pi_{com}$ (Naor Commitment Scheme), $\Pi_{5Ext}$ and $\Pi_{NM}$. I was in charge of merging our work and redesigned the $\Pi_{3Ext}$ and Schnorr's $\Sigma$-Protocol during the process. I was also responsible for testing and benchmarking the implementation. We collaborated on ZKBoo integration as an alternative $\Sigma$-Protocol.

## 5.2 Obstacles

### 5.2.1 Interfacing Protocols with Traits

One of our objectives is implement a codebase that is modular and extensible. However, we failed to do this for $\Pi_{3Ext}$ as we soon run into type complexity problems. See Appendix B for a snapshot of the trait "CommitmentScheme" and struct "ExtCommitmentCommiter" we implemented.

We did not use this version of $\Pi_{3Ext}$. Instead we implemented $\Pi_{5Ext}$ directly to avoid complex types.

## 5.3 Future Improvements

We anticipate numerous opportunities for enhancing the codebase. This includs implementing a more modular design, increasing test coverage, conducting code profiling and optimization, and incorporating more parallelization with Rayon [46].

# Bibliography


[1] Benchmarks for programming languages and compilers, Which programming language or compiler is faster. https://programming-language-benchmarks.vercel.app/.

[2] Which programming language is fastest? (Benchmarks Game). https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

[3] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[4] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[5] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217, 2023. `https://eprint.iacr.org/2023/1217`.

[6] Elaine B. Barker. Recommendation for key management:. Technical report, 5 2020.

[7] Mihir Bellare, Markus Jakobsson, and Moti Yung. Round-optimal zero-knowledge arguments based on any one-way function. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 280–305, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Annual International Cryptology Conference*, pages 701–732. Springer, 2019.

[9] Daniel Bernstein. Chacha, a variant of salsa20. 01 2008.

[10] Bheisler. Github - bheisler/criterion.rs: Statistics-driven benchmarking library for rust. https://github.com/bheisler/criterion.rs.

[11] Vincenzo Botta, Michele Ciampi, Orsini Emmanuela, Luisa Siniscalchi, and Ivan Visconti. Black-box (and fast) non-malleable zero knowledge.

[12] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964, 2020.

[13] Hai Brenner, Vipul Goyal, Silas Richelson, Alon Rosen, and Margarita Vald. Fast non-malleable commitments. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[14] David Butler, Andreas Lochbihler, David Aspinall, and Adrià Gascón. Formalising σ-protocols and commitment schemes using crypthol. *IACR Cryptol. ePrint Arch.*, 2019:1185, 2019.

[15] Franco Callegati, Walter Cerroni, and Marco Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security and Privacy*, 7(1):78–81, 2009.

[16] Michele Ciampi, Emmanuela Orsini, and Luisa Siniscalchi. *Four-Round Black-Box Non-malleable Schemes from One-Way Permutations*, pages 300–329. 01 2023.

[17] Michele Ciampi, Rafail Ostrovsky, Luisa Siniscalchi, and Ivan Visconti. Concurrent non-malleable commitments (and more) in 3 rounds. pages 270–299, 08 2016.

[18] Michele Ciampi, Rafail Ostrovsky, Luisa Siniscalchi, and Ivan Visconti. Four-round concurrent non-malleable commitments from one-way functions. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 127–157, Cham, 2017. Springer International Publishing.

[19] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297, April 1965.

[20] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

[21] Dtolnay. GitHub - dtolnay/thiserror: derive(Error) for struct and enum error types.

[22] Manuel Eberl. Fisher–yates shuffle. *Archive of Formal Proofs*, September 2016. `https://isa-afp.org/entries/Fisher_Yates.html`, Formal proof development.

[23] Ethan. GitHub - Ethan-000/zkboo: ZKBoo.

[24] facebook. GitHub - facebook/winterfell: A STARK prover and verifier for arbitrary computations. https://github.com/facebook/winterfell.

[25] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[26] Ronald Aylmer Fisher and F. Yates. Statistical tables for biological, agricultural and medical research. *Journal of the American Statistical Association*, 39(228):523, 12 1944.

[27] Ariel Gabizon, Zachary J. Williamson, and Oana-Madalina Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019.

[28] Geometryresearch. GitHub - geometryresearch/zkboo: ZKBoo.

[29] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *USENIX Security Symposium*, 2016.

[30] Shafi Goldwasser, Yael Kalai, and Guy Rothblum. Delegating computation: Interactive proofs for muggles. volume 62, pages 113–122, 5 2008.

[31] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Symposium on the Theory of Computing*, 1985.

[32] Vipul Goyal, Chen-Kuei Lee, Rafail Ostrovsky, and Ivan Visconti. Constructing non-malleable commitments: A black-box approach. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 51–60, 2012.

[33] Vipul Goyal, Silas Richelson, Alon Rosen, and Margarita Vald. An algebraic approach to non-malleability. *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 41–50, 2014.

[34] Brendan Gregg. Flame graphs. https://brendangregg.com/flamegraphs.html.

[35] Venkatesan Guruswami, Atri Rudra, and Madhu Sudan. Essential coding theory.

[36] Jørn Justesen. Class of constructive asymptotically good algebraic codes. *IEEE Trans. Inf. Theory*, 18:652–656, 1972.

[37] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Zk-img: Attested images via zero-knowledge proofs to fight disinformation, 2022.

[38] Allen Kim, Xiao Liang, and Omkant Pandey. A new approach to efficient non-malleable zero-knowledge. In *IACR Cryptology ePrint Archive*, 2022.

[39] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2017.

[40] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[41] James L. Massey. THRESHOLD DECODING. Technical report, 4 1963.

[42] Moni Naor. Bit commitment using pseudo-randomness. In *Annual International Cryptology Conference*, 1989.

[43] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[44] Rafael Pass and Hoeteck Wee. Black-box constructions of two-party protocols from one-way functions. In *Theory of Cryptography Conference*, 2009.

[45] PolygonZero. Plonky2: Fast recursive arguments with plonk and fri. https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf.

[46] Rayon-Rs. Github - rayon-rs/rayon: Rayon: A data parallelism library for rust. https://github.com/rayon-rs/rayon.

[47] Irving S. Reed, Gustave Solomon, and Kim Hamilton March. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[48] Matthew J. B. Robshaw and Olivier Billet. *New Stream Cipher Designs*. 1 2008.

[49] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

[50] Tsz-Wo Sze. Deterministic primality proving on proth numbers, 2011.

[51] Tokio-Rs. GitHub - tokio-rs/tokio: A runtime for writing reliable asynchronous applications with Rust. Provides I/O, networking, scheduling, timers, ... https://github.com/tokio-rs/tokio.

[52] Andrew Waterman and Krste Asanovć. The risc-v instruction set manual. 2019.

[53] Zcash. GitHub - zcash/halo2: The Halo2 zero-knowledge proving system. https://github.com/zcash/halo2.

# Appendix A

# Reed Solomon Encoding

```
pub(crate) fn reed_solomon_encode<F: PrimeField + FftField>
  (evals: &[F]) -> Vec<F> {
  let domain = GeneralEvaluationDomain::<F>::
    new(evals.len()).unwrap();
  let coeffs = domain.ifft(evals);
  let domain = GeneralEvaluationDomain::<F>::
    new(evals.len() * (1.0 / RSRATE) as usize).unwrap();
  domain.fft(&coeffs)
}
```

# Appendix B

# Trait Complexity When Implementing $\Pi_{3Ext}$

```
pub trait CommitmentScheme<
 F,
 E,
 InitMessage: IndexMut<usize>,
 Commitment: Clone,
 SecretState: std::cmp::PartialEq + Clone,
> where
 <InitMessage as Index<usize>>::
  Output: Sized + IndexMut<usize>,
 <<InitMessage as Index<usize>>::
  Output as Index<usize>>::Output: Sized + Clone,
{
  fn init(len: usize) -> Result<InitMessage, E>;
  fn commit(
  msg: &[F],
  init_m: <<InitMessage as Index<usize>>::
  Output as Index<usize>>::Output,
  ) -> (Commitment, SecretState);
  fn open(
msg: &[F],
  init_m: <<InitMessage as Index<usize>>::
  Output as Index<usize>>::Output,
  commit: Commitment,
  secret_state: SecretState,
  ) -> Result<(), E>;
}


type CommitmmentsAndSecreState<F, Commitment, SecretState> =
 Vec<((Vec<F>, Commitment, SecretState),
```

```
 (Vec<F>, Commitment, SecretState))>;

pub struct ExtCommitmentCommiter<
 C: CommitmentScheme<F, E, InitMessage, Commitment, SecretState>,
 F: PrimeField,
 E: Debug,
 InitMessage: IndexMut<usize>,
 Commitment: Clone,
 SecretState: std::cmp::PartialEq + Clone,
> where
 <InitMessage as Index<usize>>::
  Output: Sized + IndexMut<usize>,
 <<InitMessage as Index<usize>>::Output as Index<usize>>::
  Output: Sized + Clone,
{
  len: usize,
  message: Vec<F>,
  commitments_and_secrete_state:
  CommitmmentsAndSecreState<F, Commitment, SecretState>,
  c: PhantomData<C>,
  f: PhantomData<F>,
  e: PhantomData<E>,
  init_m: InitMessage,
}
```