Disjunctive Zero Knowledge

Justin Tan
Department of Computer Science
University of Warwick

Supervised by Nicholas Spooner
Year of Study: 3rd

25 April 2023

WARWICK
THE UNIVERSITY OF WARWICK

Abstract

Zero-knowledge proofs are protocols that allow a prover to prove the validity of a statement to a verifier without revealing any information about the statement. There has been a long line of research into zero-knowledge proofs – in particular, zero-knowledge proofs for disjunctive statements have been a core target. Disjunctive statements are made up of a set of clauses joined by logical OR operators, and the goal of the prover is to prove that at least one of the clauses is true.

In this work we look at one approach for constructing disjunctive zero-knowledge proofs: general *compilers* for zero-knowledge proofs. This approach has been explored by in early work by Cramer and Damgård in 'Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols', and more recently by Goel *et al.* in 'Stacking Sigmas: A Framework to Compose Σ-Protocols for Disjunctions'. This project looks at implementing the compilers proposed in these two papers, benchmarking their performance, and comparing the results. There has been a lack of implementations of these compilers, and this work serves fill in this gap, as well as to provide a better understanding of their performances, and lay the groundwork for future work to benchmark against past work.

Our results show that Goel *et al.*'s compiler, Stacking Sigmas, outperforms Cramer and Damgård's compiler, CDS94, in terms of communication complexity, providing significant savings in the size of the proof as the number of clauses increase. We also show that CDS94 is faster than Stacking Sigmas when the number of clauses are small, but ultimately slows down as the number of clauses increase due to how we have implemented the compiler.

Contents

Chapter 1

Introduction

Zero-Knowledge proofs and arguments [GMR85] are protocols with two parties: a Prover, and a Verifier. The prover's goal is to convince the verifier of the validity of an NP statement, while revealing zero additional information except that the statement is valid. In other words, the verifier learns nothing new, except whether the statement being proven is true or false. Early research in this area showed that all languages in NP have zero-knowledge proof systems [GMW86], sparking a long line of research into developing more efficient zero-knowledge proofs for various use cases. Today, efficient zero-knowledge proofs are being used in practice, with several use cases such as e-voting and to secure decentralized systems [Sas+14; swi19].

In many cases, it is desirable to have a zero-knowledge proof for a *disjunctive statement*, which is an NP statement with a set of clauses that are connected with logical ORs:

$$clause_1 \vee clause_2 \vee \cdots \vee clause_n$$

Such zero-knowledge proofs fall under the category of "disjunctive zero-knowledge": the goal is to prove that at least 1 of the clauses are true while hiding the location of *active clauses* – the clauses that the Prover has knowledge of. Disjunctive statements have very useful properties that occur commonly in practice, and adding zero-knowledge can be very beneficial to systems that desire both privacy and verifiability. For example, a disjunctive zero-knowledge proof can be used to prove an individual's membership to a particular group while revealing nothing about the identity of the individual. They can also be used to reveal the existence of a bug in a verifier's code base as shown in [HK20]. Consequently, there has been wide research interest in determining how to construct disjunctive zero-knowledge proofs, and how to do so efficiently: saving on communication complexity (the size, in bytes, of the messages between prover and verifier) and computational complexity.

One approach is to modify the underlying protocols manually so that they support disjunctive statements. Exciting results from recent work has shown that it is possible to achieve protocols with a communication complexity sub-linear in the number of clauses using this approach [HK20; ACF21]. Unfortunately, this approach does not generalise well and only work for the individual protocols that they target. A solution to this is to develop generic compilers for disjunctive zero-knowledge [CDS94; Goe+21]. These compilers are able to target a large class of zero-knowledge proofs known as 3-round public coin proofs of knowledge (or more popularly known as Σ-protocols).

Research targeting *disjunctive zero-knowledge compilers* began with Cramer, Damgård, and Schoenmakers [CDS94]. They proposed a generic compiler to compose multiple instances of Σ-protocols into a disjunctive zero-knowledge proof. Their results showed that the communication complexity of the resulting proof is linear in the number of clauses. More recently, Goel *et al.* [Goe+21] improved on this further, providing a generic compiler that still targets a large class of Σ-protocols while achieving a communication complexity that is sub-linear in the number of clauses.

1.1 Our Contributions

While extensive research has been conducted, there is a lack of notable real-world implementations of these results [1]. In this work, we seek to fill in this gap and build upon past research and have implemented the CDS94 compiler [CDS94] and the Stacking Sigmas compiler from [Goe+21]. We also benchmark these protocols to measure and compare the difference in their performances. From our analysis of the collected data, we provide insights into the trade-offs between these protocols and suggest when they might be the most useful.

As a project extension, we are also working on an implementation of the compiler introduced in "Speed-Stacking: Fast Sublinear Zero-Knowledge Proofs for Disjunctions" [Goe+22]. This compiler builds on the work in [Goe+21] and explores how sublinear-sized zero-knowledge proofs can be compiled into a sublinear-sized disjunctive zero-knowledge proof, which has a *sublinear* running time. Specifically, we are working on applying this compiler to Compressed Σ-protocols [ACF21]. This is still a work-in-progress.

We hope that our work helps lay the foundation for future work to compare new compilers to existing ones and in turn lead to further improvements that may have a broad impact on existing and upcoming cryptographic systems that rely on such a use case.

1.2 Related work

As part of their work in [Goe+21], Hall-Andersen provides an implementation of the Stacking Sigmas (SS) compiler [Hal21]. In this implementation, they apply the Stacking Sigmas compiler to Schnorr over Ristretto25519 to obtain efficient ring signatures. In our work, we include a comparison of our implementation of the compilers (both [CDS94] and [Goe+21]) to Hall-Andersen's implementation. The key difference between our implementations is that we use a newer commitment scheme (present in the updated and latest version of [Goe+21]). Additionally, we also seek to improve the usability of the compiler by providing a more user-friendly API. More is discussed in Section **??**.

[1]It should be noted that Hall-Andersen [Hal21] has provided a benchmark of applying the compiler in [Goe+21] to Schnorr's discrete log protocol [Sch89].

Chapter 2

Background

Before discussing our implementation and results, we introduce the necessary background concepts for this project. We firstly formalise the notation we use in this report, and highlight broad concepts that are relevant to the entire project. These concepts include: zero-knowledge, disjunctive zero-knowledge, and Σ-protocols. After which, we split the remaining section into subsections: each for one of the compilers we implement [CDS94; Goe+21]

2.1 Notation

Table 2.1: Notation used in this report

Symbol	Details
λ	Computational security parameter. Refers to a cryptographic system's security against a computationally bounded adversary.
κ	Statistical security parameter. Refers to the security provided by negligible statistical probability.
$\overset{?}{=}$	Boolean assertion.
$\|\|$	Bit concatenation: $0000\|\|1111 = 00001111$
$\overset{\$}{\leftarrow}$	Sampling from a distribution: $x \overset{\$}{\leftarrow} \mathcal{D}$ is the sampling of "x" from the distribution "$\mathcal{D}$"
$[l]$	The range of integers from 1 to l

2.2 Disjunctive Zero-Knowledge

Definition 1 (NP Relations)**.** Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation. Then $w(x) = \{w \mid (x,w) \in R\}$ and $L_R = \{x \mid \exists w, (x,w) \in R\}$. If $(x,w) \in R$, we say that w is a witness for x. R is an NP-relation if it fulfils the following two properties:

1. **Polynomially bounded.** We say that R is *polynomially bounded* if there exists a polynomial p such that $|w| \leq p(|x|), \forall (x,w) \in R$.
2. **Polynomial-time verification.** There exists a polynomial-time algorithm for deciding membership in R. Consequently, $L_R \in NP$.

Throughout this document, we will use $\mathcal{R}$ to refer to a binary NP-relation.

Definition 2 (Zero-Knowledge [Tha22])**.** A proof or argument system (P, V) is zero-knowledge over $\mathcal{R}$ if there exists a *probabilistic polynomial time* (PPT) simulator $\mathcal{S}$, such that for all $(x, w) \in R$, the distribution of the output $\mathcal{S}(1^\lambda, x)$ of the simulator is indistinguishable from the distribution over the conversations generated by the interaction of P and V, from the perspective of V; we denote this with $\text{View}_V(P(x, w), V(x))$. Conversations between P and V are ordered triples of the form (a, c, z), and are known as *transcripts*.

Intuitively, this means that V should not learn anything from the transcripts with P that they cannot already learn on their own by running the simulator $\mathcal{S}$; they learn nothing new.

Definition 3 (Disjunctive Zero-Knowledge)**.** Given a sequence of statements $(x_1, x_2, \ldots, x_l)$, a *disjunctive zero-knowledge proof* is a protocol to prove in zero-knowledge that $x_1 \in \mathcal{L}_1 \vee x_2 \in \mathcal{L}_2 \vee \ldots \vee x_l \in \mathcal{L}_l$, for NP languages $\mathcal{L}_i$. We term clauses for which the prover has a witness for as *active* clauses.

Definition 4 (Honest-Verifier Zero-Knowledge – HVZK)**.** A proof system is HVZK if it only requires that $\mathcal{S}$ is an efficient simulator for honest (non-malicious) probabilistic polynomial time verifier strategies V. If V is malicious then the distribution of the output $\mathcal{S}(x)$ will no longer be indistinguishable from $\text{View}_V(P(x, w), V(x))$ for such proof systems.

2.3 Σ-protocols

Definition 5 (Σ-Protocol [Goe+21])**.** A Σ-protocol $\Pi = (A, Z, \phi)$ for $\mathcal{R}$ is a 3-round protocol between a prover algorithm P and a verifier algorithm V. The protocol consists of a tuple of probabilistic polynomial time algorithms (A, Z, ϕ) with the following interfaces:

- $a \leftarrow A(x, w; r^p)$: Given statement x, witness $w \in w(x)$, and prover randomness r^p as input; output the first message a that P sends to V in the first round.
- $c \overset{\$}{\leftarrow} \{0, 1\}^\kappa$: V samples a random challenge c and sends it to P in the second round.
- $z \leftarrow Z(x, w, c; r^p)$: Given x, w, c, and r^p as input; output the message z that P sends to V in the third round.
- $b \leftarrow \phi(x, a, c, z)$: Given x, and the messages in the transcript, output a bit $b \in \{0, 1\}$. This algorithm is executed by V, and V accepts if $b = 1$.

A Σ-protocol has the following properties:

1. **Completeness.** Π is complete if for any x, $w \in w(x)$, and any prover randomness $r^p \overset{\$}{\leftarrow} \{0, 1\}^\lambda$, the verifier accepts with probability 1. A proof is complete if for all valid transcripts (a, c, z), $\phi(x, a, c, z) = 1$.
$$Pr\left[\phi(x, a, c, z) = 1 \mid a \leftarrow A(x, w; r^p); c \overset{\$}{\leftarrow} \{0, 1\}^\kappa; z \rightarrow Z(x, w, c; r^p)\right] = 1$$
2. **Special Soundness.** Π is said to have special soundness if there exists a PPT extractor $\mathcal{E}$, such that given any two transcripts (a, c, z) and (a, c', z') for statement x, where $c \neq c'$ and $\phi(x, a, c, z) = \phi(x, a, c', z') = 1$, an element of $w(x)$ can be computed by $\mathcal{E}$. Soundness is concerned with ensuring that a prover cannot cheat – the verifier will always reject if the transcript is invalid.

3. **Special Honest-Verifier Zero-Knowledge (SHVZK).** Π is SHVZK if there exists a PPT simulator $\mathcal{S}$, such that for any x, w, $(x,w) \in \mathcal{R}$, the distribution over the output $\mathcal{S}(1^\lambda, x, c^*)$ is indistinguishable from the distribution over transcripts produced by the interaction between V and P when the challenge is c^*.

$$\{(a,z) \mid c^* \xleftarrow{\$} \{0,1\}^\kappa; (a,z) \leftarrow \mathcal{S}(1^\lambda, x, c^*)\} \approx_{c^*}$$
$$\{(a,z) \mid r^p \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow A(x,w;r^p); c^* \xleftarrow{\$} \{0,1\}^\kappa; z \leftarrow Z(x,w,c^*;r^p)\}$$

Definition 6 (Witness Indistinguishable – WI)**.** A Σ-protocol is witness indistinguishable over $\mathcal{R}$ if for any V', any large enough input x, any $w_1, w_2 \in w(x)$, and for any fixed challenge c^*, the distribution over transcripts in the form (a_1, c, z_1) and (a_2, c, z_2) are indistinguishable, where $a_i \leftarrow A(x, w_i; r^p)$ and $z_i \leftarrow Z(x, w_i, c^*; r^p)$ for $i \in \{1,2\}$. This means that the prover reveals no information about which are the active clauses.

Definition 7 (Informal definition of Witness Hiding – WH)**.** For any x that is generated with a certain probability distribution by a generator $\mathcal{G}$ which outputs pairs $(x,w) \in \mathcal{R}$, a Σ-protocol is witness hiding over $\mathcal{G}$, if it does not help even a cheating verifier to compute a witness for x with non-negligible probability. Refer to [FS90] for details. WH is a weaker property than general zero-knowledge, as it only asserts that the verifier cannot learn about the witness (not asserting anything about other information). That said, it can replace zero-knowledge in many protocol constructions, as it is in most Σ-protocols where the witness is the only private information to hide.

2.4 Schnorr's Identification Protocol

There are two parties in an *identification scheme*, the prover P and the verifier V, and the objective of the protocol is for the prover to convince the verifier that they are who they claim to be. In other words, V should be convinced that P knows the private key that corresponds to the public key of P. A familiar example is the standard protocol of password authentication.

Schnorr's protocol [Sch89] is an identification scheme where P proves knowledge of the discrete log w of a group element $H \in \mathbb{G}$, where $H = w \cdot G$ for some generator $G \in \mathbb{G}$. $(\mathbb{G}, +)$ is a finite abelian group with $+$ as the binary operator [1]. This protocol relies on the discrete log assumption [DH76] – the assumption states that finding w given only H and G is *currently* computationally infeasible, assuming that $\mathbb{G}$ is a sufficiently large group. Conversely, proving that $H = w \cdot G$ given w and G can be computed efficiently.

In our implementation, we plan to use the Ristretto group [Val+]: a construction of a prime order group from a family of elliptic curves known as Edwards curves [Edw07].

Definition 8 (Schnorr's Protocol [Sch89])**.** Let $\mathbb{G} = E(\mathbb{F}_q)$, where E is an elliptic curve over the finite field $\mathbb{F}_q$. Suppose that both the prover P and verifier V agree on E and $\mathbb{F}_q$, then let $H \in E(\mathbb{F}_q)$ be the public key that corresponds to the private key w such that $H = w \cdot G$. The prover convinces the verifier that they have knowledge of the private key by executing Protocol 1.

[1]We have chosen to define $\mathbb{G}$ with the $+$ operator because our implementation uses elliptic curves which are finite abelian groups over addition. The discrete log can be defined equivalently with multiplication like so: $h = g^w$

Protocol 1. Schnorr's protocol. Public information: $\mathbb{G} = E(\mathbb{F}_q),\ H \in \mathbb{G}$. Statement to prove: $H = w \cdot G$

$$\begin{array}{lcl}
\text{Prover}(w_p) & & \text{Verifier} \\
r \xleftarrow{\$} \mathbb{F}_q,\ \mathcal{A} = r \cdot G & \xrightarrow{\quad \mathcal{A} \quad} & \\
 & & c \xleftarrow{\$} \mathbb{F}_q \\
 & \xleftarrow{\quad c \quad} & \\
z \leftarrow cw_p + r & & \\
 & \xrightarrow{\quad z \quad} & z \cdot G \stackrel{?}{=} \mathcal{A} + c \cdot H
\end{array}$$

Communication between Prover and Verifier in Schnorr's protocol. Evaluating the final equation, we can easily see that V accepts if and only if $w = w_P$.

$$z \cdot G = \mathcal{A} + c \cdot H \iff r \cdot G + c \cdot w_P \cdot G = r \cdot G + c \cdot w \cdot G \iff w = w_P$$

A simulator for Schnorr's protocol. From our definition of a Zero Knowledge protocol in Definition 2, we shared that every zero knowledge proof or argument system has a PPT algorithm called a simulator. We can construct a simulator for Schnorr's protocol by running it "in reverse" shown in Figure 2.1.

```
Let our simulator be $S(H)$

$z \xleftarrow{\$} \mathbb{F}_q$
$c \xleftarrow{\$} \mathbb{F}_q$
$\mathcal{A} = z \cdot G - c \cdot H$

output $(\mathcal{A}, c, z)$
```

Figure 2.1: A Simulator for Schnorr

Since z and c are both chosen randomly, the resulting $\mathcal{A}$ is also random, and our output will have the same distribution as the distribution over transcripts in an actual interaction. Note that the simulator presented here achieves only honest-verifier zero-knowledge (Definition 4) as the only input to the simulator is the statement to prove H. We can easily modify this simulator to achieve special honest-verifier zero-knowledge (Definition 5) by adding a challenge c as an input to the simulator.

2.5 CDS94

In this section, we introduce the components necessary for the [CDS94] compiler. In addition to a Σ-protocol, which is relevant to every compiler in this project, the CDS94 compiler requires a *secret sharing scheme* and the compiler itself. In the following 2 subsections, we introduce these components.

2.5.1 Secret Sharing Scheme

A *secret sharing scheme* [Sha79; Bla79], is a method of distributing a secret s to n participants in a way that no one participant has intelligible information about the secret. This is achieved by splitting up s into *shares*, distributing one share to each participant in a way that *only* a subset of participants can reconstruct s. Subsets that can reconstruct the secret are called *qualified sets*. The set of all qualified sets is the secret sharing scheme's *access structure*.

In this work, we are concerned with *perfect* secret sharing schemes. Perfect secret sharing schemes are ones where the participants in *non-qualified* sets cannot obtain any information whatsoever about the secret [CDS94]. Additionally, for CDS94, we require our secret sharing scheme to have a few additional properties.

Definition 9 (Secret Sharing Schemes for [CDS94])**.** Let Π be a Σ-protocol for the relationship $\mathcal{R} = \{(x, w)\}$. We define a *secret sharing scheme* for CDS94 as $S(k)$, where k is the length of x in bits. $S(k)$ splits a secret s into n shares such that n is polynomial in k ($n = poly(k)$). Let $D(s)$ refer to the probability distribution of all the shares that are produced when the secret s is distributed. If we consider a subset of participants A, then $D_A(s)$ refers to the distribution of shares that only includes participants in A. Since the scheme is perfect, the probability distribution $D_A(s)$ is not affected by any other subset B for any non-qualified set A. Therefore, we can simply write D_A instead of $D_A(s)$ when A is non-qualified. We now define the properties we require:

1. The length of shares produced in $S(k)$ is related to k through a polynomial function.
2. The secret can be distributed and reconstructed in a time that is polynomial in k.
3. With a complete set of n shares and the secret s, it is possible to check in a time that is polynomial in k whether all qualified sets of shares determine s as the secret.
4. For any secret s, it is always possible to complete a set of shares for participants in a non-qualified set A, distributed according to D_A, to a full set of shares that are distributed according to $D(s)$ and consistent with the secret s. This completion process can be done in a time that is polynomial in k.
5. The probability distribution D_A for any non-qualified set A is such that shares for the participants in A are independently and uniformly chosen.

An $S(k)$ which fulfils properties 1-4, is known as *semi-smooth*. With a semi-smooth scheme, we can compile a SHVZK Σ-protocol using Theorem 9 of [CDS94]. A *smooth* scheme, is where $S(k)$ fulfils all 5 properties. With this we can compile a HVZK Σ-protocol using Theorem 8 of [CDS94].

2.5.2 Shamir's Secret Sharing

In our implementation of the CDS94 compiler, we choose Shamir's secret sharing scheme [Sha79]. First, we need to define Lagrange's interpolating polynomial as it is relevant in the definition of Shamir's secret sharing scheme.

Definition 10 (Lagrange's Interpolating Polynomial [BW])**.** Lagrange's interpolating polynomial is a polynomial that passes through a set of n data points $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$. It can be written in the form:

$$L(x) = \sum_{i=0}^{n-1} y_i \ell_i(x),$$

where $\ell_i(x)$ is the ith Lagrange basis polynomial, defined by:

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{x - x_j}{x_i - x_j}.$$

The Lagrange interpolating polynomial $L(x)$ is a polynomial of degree at most $n-1$ that satisfies $L(x_i) = y_i$ for $i = 0, 1, \ldots, n-1$. In Shamir's secret sharing scheme, we rely on polynomial interpolation to distribute and reconstruct the secret from a set of shares.

Definition 11 (Shamir's Secret Sharing (SSS) Scheme)**.** A smooth secret sharing scheme and a *threshold sharing scheme*, SSS produces qualified sets of size d. Any d out of n participants can reconstruct the secret; with $d-1$ shares and less, no information about the secret can be obtained.

To distribute a secret s:

1. We first choose a random polynomial of degree $d-1$ in which the constant term is s.
$$q(x) = s + a_1 x + \ldots + a_{d-1} x^{d-1}$$
2. Next, for participants $i \in \{1, \ldots, n\}$, we distribute the share $share(i) = (i, q(i))$ to the i-th participant. We can easily represent $share(i)$ in bits by simply concatenating the bits of i and $q(i)$.

To reconstruct a secret:

1. Any d or more participants can combine their shares. Using interpolation, a polynomial h of degree $d-1$ can be computed which passes through the d points that correspond to the shares. This polynomial h is equivalent to that which was used to distribute the secret, polynomial q. This means that the constant term can be derived by evaluating $h(0) = q(0) = s$.
2. If less than d participants combine their shares, they will not have enough information to reconstruct the secret. This is because if a random polynomial h is interpolated from d points with non-zero x-values, then the value of $h(0) = s$ is uniformly random. This means that with only $d-1$ points (or shares), the values that s can take are uniformly distributed in the domain of the y-axis.

Definition 12 (Qualified Set Completion for SSS)**.** Suppose we have a set $\mathcal{S}$ of shares, which have been previously distributed by Shamir's secret sharing scheme with the secret s using polynomial q with degree d. We define the algorithm $\mathsf{Complete}(s, U, A) \rightarrow Q$, where

- $A = \{i \mid (i, q(i)) \in \mathcal{S}\} \wedge |A| = |\mathcal{S}| - d$ is the set of indexes for a subset of shares in $\mathcal{S}$.
- $U = \{share(i) \mid i \in \mathcal{S} \setminus A\} \wedge |U| = d$ is a an unqualified set of shares.

$\mathsf{Complete}$ returns a qualified set of shares $Q = \{share(i) \mid i \in A\}$, which is the corresponding shares of the indexes in A. We briefly outline how this is done:

1. Using polynomial interpolation, construct a polynomial of degree d, with the shares in the set U and $share(0) = (0, s)$ (this gives us $d+1$ shares).
2. With this polynomial, interpolate at $x = i$ for $i \in A$ to determine the corresponding y_i for each $i \in A$.
3. Return $Q \leftarrow \{share(i) = (x_i, y_i) \mid i \in A\}$.

Notably, we do not use SSS within CDS94 to reconstruct a secret; we use it together with a known secret to complete a qualified set using the procedure outlined in Definition 12.

2.5.3 CDS94 Compiler

In this paper, Cramer *et al* [CDS94] presents 2 primary ways to compile Σ-protocols depending on the underlying choice of Σ-protocol and the secret sharing scheme. Our implementation makes use of Theorem 8 of the paper because we choose to use Schnorr's protocol (Section 2.4) and Shamir's secret sharing (Section 2.5.2). More details regarding our implementation will be discussed in a further section. Now, we recall theorem 8 of [CDS94] – note that we alter the notation slightly from the original paper to be more consistent with the rest of this report.

Theorem 8 [CDS94]. Given $\Pi = (A, Z, \phi)$, $\mathcal{R}_\Gamma$ and $\mathcal{S}(k)$ where

- Π is a 3-round public coin (Σ-protocol) HVZK proof of knowledge for relation $\mathcal{R}$.
- $\{\mathcal{S}(k)\}$ is a family of smooth secret sharing schemes.
- $\mathcal{R}_\Gamma$ is a relation where $((x_1, \ldots, x_m), (w_1, \ldots, w_m)) \in \mathcal{R}_\Gamma$ if and only if ($\Longleftrightarrow$)
  - all x_i's are of the same length k, and
  - the set of indices i for which $(x_i, w_i) \in \mathcal{R}$ corresponds to a qualified set for $S(k)$

Then, there exists a Σ-protocol, $\Pi' = (A', Z', \phi')$, that is witness indistinguishable (Definition 6) for the relation $\mathcal{R}_\Gamma$. The description of this protocol is outlined in Protocol 2. Interested readers can refer to the original paper for the proof of this theorem [CDS94].

Protocol 2. CDS94 Compiler. A compiler for composing n instances of a Σ-protocol Π into a single Σ-protocol Π' that proves the **disjunction** of these n instances.

Let A be the set of indices i of the *active clauses*.
Statement: $x = x_1, \ldots, x_n$
Witness: $w = \{w_i \mid i \in A \land (x_i, w_i) \in R\}$

- **First Round:** the Prover, P, computes $A'(x, w; r^P) \to a$ accordingly:
  - For each $i \in \bar{A}$, run the simulator for $\mathcal{P}$ for the statement x_i to produce the transcripts (m_1^i, c_i, m_2^i).
  - For each $i \in A$, compute $m_1^i \leftarrow A(x_i, w_i; r^P)$.
  - Send $a \leftarrow (m_1^1, \ldots, m_1^n)$ to V.
- **Second Round:** V sends $c_0 \leftarrow \{0,1\}^\lambda$ to P.
- **Third Round:** Prover computes $Z'(x, w, c; r^P) \to z$:
  - Compute $Q \leftarrow Complete(c_0, U, A)$, where $U = \{c_i \mid i \in \bar{A}\}$.
  - Set $c_i = share(i)$, for each $share(i) \in Q$
  - For $i \in A$, compute $m_2^i \leftarrow Z(x_i, w_i, c_i; r^P)$.
  - For $i \in \bar{A}$, m_2^i has already been computed in the first round using the simulator.
  - Send $z \leftarrow ((c_1, m_2^1), \ldots, (c_n, m_2^n))$ to V.
- **Verification:** Verifier computes $\phi'(x, a, c_0, z) \to b$ as follows:
  - Extract c_i and m_2^i from $z \leftarrow ((c_1, m_2^1), \ldots, (c_n, m_2^n))$
  - Check that all conversations (m_1^i, c_i, m_2^i) would lead to acceptance by the verifier in $\mathcal{P}$.

- Check that each $share(c_i)$ is consistent with the secret c_0 using the reconstruction algorithm in Definition 11.
- If any of the checks fail return $b \leftarrow 0$; else return $b \leftarrow 1$.

Using this compiler, we can construct a Σ-protocol that has a communication complexity linear in the number of clauses.

2.6 Stacking Sigmas

Moving on, we introduce the concept of a *partially-binding vector commitment scheme* and present the Stacking Sigmas compiler proposed by [Goe+21]. This compiler aims to improve on the communication complexity of the [CDS94] compiler, by reducing the communication size to $O(\log n)$, where n is the number of clauses in the disjunction.

2.6.1 Partially-Binding Vector Commitments

In Section 5 of [Goe+21], Goel *et al* introduce the concept of partially-binding vector commitment schemes. These commitment schemes allow a Prover to make a commitment to a vector of length l with t binding positions; the remaining positions (indexes) in the vector are equivocable. Here, we recall the definition of these commitment schemes and refer interested readers to Figure 2 of [Goe+21] for the construction of the general t-out-of-l scheme.

Definition 13 (t-out-of-l Binding Vector Commitment [Goe+21]). Given a message space $\mathcal{M}$, the security paramater λ, the length of the vector l, and the number of binding positions t: the tuple of PPT algorithms $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv}, \mathsf{BindCom})$ defines a t-out-of-l partially-binding vector commitment scheme. These algorithms are defined as follows:

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$: Given the security parameter λ, the Setup algorithm outputs public parameters pp for the commitment scheme.
- $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$: Given public parameters pp and a set B, where $B \subseteq [l] \wedge |B| = t$. The Gen algorithm returns a commitment key ck and equivocation key ek.
- $(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, v; r)$: Given public parameter pp, equivocation key ek, vector v with length l, and randomness r as input, the EquivCom algorithm returns a partially-binding commitment com and auxiliary equivocation information aux. This means that the vector v can be equivocated in equivocable locations and com will be the same.
- $r \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, v, v', \mathsf{aux})$: Given public parameters pp, equivocation key ek, original message vector v and updated message vector v' where $\forall i \in B : v_i = v'_i$, and auxiliary equivocation information aux. Equiv returns equivocation randomness r.
- $\mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, v; r)$: Given public parameters pp, commitment key ck, vector v, and randomness r as input, BindCom returns a commitment com. This algorithm plays a similar role to that of Open in a typical commitment scheme.

Partially-binding vector commitment schemes have the following properties

1. **(Perfect) Hiding.** Suppose there are two vectors $\mathbf{v}^{(1)}, \mathbf{v}^{(2)} \in \mathcal{M}^l$ and two corresponding sets of binding positions $B^{(1)}, B^{(2)} \in \binom{(l)}{t}$. A perfectly hiding commitment scheme is one where the commitment key *ck* and commitment *com* produced by any two sets of binding position and original vectors $(\mathbf{B}^{(1)}, \mathbf{v}^{(1)})$ and $(\mathbf{B}^{(2)}, \mathbf{v}^{(2)})$ are indistinguishable from each other when they are equivocated to the same vector $\mathbf{v}'$ (provided that $\mathbf{v}'$ is a valid equivocation for both vectors).
2. **(Computational) Partial Binding.** A malicious user (that generates ck itself) is not able to cheat the system by equivocating on more than $l - t$ positions.
3. **Partial Equivocation.** It is always possible to equivocate to any vector $\mathbf{v}'$ as long as $\forall i \in B : v_i = v_i'$.

There is an efficiency requirement which requires the size of the commitment to be independent from the size of the messages in the vector. This can be achieved by using a collision resistant hash function H. For more details on each property, we refer the reader to Definition 4 of [Goe+21].

2.6.2 Half-Bindings & Q-Bindings

In this work, we make use of the generic 1-out-of-2^q construction provided in Section 5.3 of [Goe+21] for our implementation of the Stacking Sigmas compiler. We use this particular construction as it allows us to yield commitments that are logarthmic in size (in bytes) to the original vector of messages (**Theorem 2 & Corollary 1** in [Goe+21]). A core ingredient in this construction is a 1-out-of-2 partially-binding vector commitment scheme, which is used recursively to form a *binary* "tree of commitments". For brevity, we refer to the generic construction as "q-bindings" and the 1-out-of-2 construction as "half-bindings". The leaves of this tree are the original messages in our vector; intermediate nodes are the resulting commitments to the nodes' children using half-bindings. Essentially, intermediate commitments are regarded as messages as well, and are commited to recursively until there is only 1 root node – the final commitment.

While the source material provides the general t-out-of-l construction for partially-binding vector commitments, it does not explicitly provide that for half-bindings. In Figure 2.2, we provide the construction for half-bindings derived from the general t-out-of-l construction. The proof of correctness for this construction is trivial as it can be easily seen to be a specific case of the general construction when $t = 1$ and $l = 2$.

Figure 2.2: Construction of a 1-out-of-2 partially-binding vector commitment scheme.

```
$pp \leftarrow \text{Setup}(1^\lambda)$:
    1. $\mathbb{G} \leftarrow \text{GenGroup}(1^\lambda)$; $g_0, h \xleftarrow{\$} \mathbb{G}$
    2. return $pp \leftarrow (\mathbb{G}, g_0, h)$

$(com, aux) \leftarrow \text{EquivCom}(pp, ek, m_1, m_2)$:
    1. Extract $ck$ from $ek$
    2. $r \xleftarrow{\$} \mathbb{Z}^2_{|\mathbb{G}|}$
    3. com $\leftarrow \text{BindCom}(pp, ck, m_1, m_2, r)$
    4. return (com, $r$)

$(ck, ek) \leftarrow \text{Gen}(pp, B)$:
    1. Let $E = \{1, 2\} \setminus B$ be the set of equivocal indexes. $|B| = 1$.
    2. Generate trapdoor $td$ for $i \in E : td \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}, g_i \leftarrow h \cdot td$
    3. Interpolate the first element: $g_1 = \begin{cases} g_2 - g_0 & \text{if } 2 \in E \\ g_1 & \text{if } 1 \in E \end{cases}$
    4. $ck \leftarrow g_1$
    5. $ek \leftarrow (B, td, ck)$
    6. return $(ck, ek)$
$com \leftarrow \mathrm{BindCom}(pp, ck, m_1, m_2, r)$:
    1. Interpolate $g_2$: $g_2 = g_1 + g_0$
    2. $(r_1, r_2) \leftarrow r$
    3. Commit individually for $j \in \{1,2\}$ : $com_j \leftarrow h \cdot r_j + g_j \cdot m_j \in \mathbb{G}$
    4. return $(com_1, com_2)$
$r \leftarrow \mathrm{Equiv}(pp, ek, m_1, m_2, m'_1, m'_2, aux)$:
    1. Extract $B$ and $td$ from $ek$; Let $E = \{1,2\} \setminus B$.
    2. Parse $aux = (r_1, r_2) \in \mathbb{Z}^2_{|\mathbb{G}|}$
    3. Interpolate $g_2 = g_1 + g_0$
    4. for $i \in B : r'_i \leftarrow r_i$
    5. for $i \in E : r'_i \leftarrow r_i - td \cdot (m'_i - m_i)$
    6. return $r' \leftarrow (r'_1, r'_2)$
```

Communication Complexity. Observing the construction of half-bindings, we can see that the size (in bytes) of the outputs of each method is directly related to the choice of the group $\mathbb{G}$ and the number of bytes required to represent a group element $g \in \mathbb{G}$ and scalars $r \in \mathbb{Z}_{|\mathbb{G}|}$.

2.6.3 Stacking Sigmas Compiler

In Section 6 of [Goe+21], Goel *et al* introduce the two main properties of stackable Σ-protocols: *extended honest verifier zero-knowledge*, and *recyclable third round messages*. Moreover, they go on to show that many Σ-protocols satisfy these properties, proving that all Σ-protocols can be made EHVZK (Observation 1), and many natural Σ-protocols have recyclable third round messages.

In this section, we state the definition of these two properties as shown in [Goe+21], but do not dive into the details of how a large class of Σ-protocols have these two properties. We encourage readers to refer to [Goe+21] for the proofs of these properties and the observations related to them.

Definition 14 (Extended Honest Verifier Zero-Knowledge – EHVZK)**.** The Σ-protocol $\Pi = (A, Z, \phi)$ is EHVZK if there exists a *deterministic* polynomial time "extended simulator" algorithm $\mathcal{S}^{\text{EHVZK}}(1^\lambda, x, c, z)$ such that an efficiently samplable distribution $\mathcal{D}^{(z)}_{x,c}$ exists, for any $(x, w) \in \mathcal{R}$ and $c \in \{0,1\}^\kappa$. $\mathcal{D}^{(}_{x,c}z)$ is a distribution of third round message z conditioned on x and c:

$$\left\{(a,c,z) \mid r^P \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow A(x,w;r^P); z \leftarrow Z(x,w,c;r^P)\right\}$$
$$\approx \left\{(a,c,z) \mid z \xleftarrow{\$} \mathcal{D}^{(z)}_{x,c}; a \leftarrow \mathcal{S}^{\text{EHVZK}}(1^\lambda,x,c,z)\right\}$$

Intuitively this means, that the such that the transcript (a,c,z) is indistinguishable from the output of the simulator $\mathcal{S}^{\text{EHVZK}}(1^\lambda,x,c,z)$.

Definition 15 (Recyclable 3rd Round Messages)**.** A Σ-protocol $\Pi = (A,Z,\phi)$ for $\mathcal{R}$ has recyclable third round messages if for every challenge $c \in \{0,1\}^\kappa$, there is an efficiently sampleable distribution $\mathcal{D}^{(z)}_c$:

$$\mathcal{D}^{(z)}_c \approx \left\{z \mid r^P \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow A(x,w;r^P); z \leftarrow Z(x,w,c;r^P)\right\} \quad \forall (x,w) \in \mathcal{R}$$

Essentially, this means that the distribution of z is not dependent on the statement x and hence does not "leak information" about the statement. Consequently, this means that z can be reused for any statement x, even non-active clauses, and still hide the active clauses.

Definition 16 (Stackable Σ-protocol – Definition 9 of [Goe+21])**.** A Σ-protocol $\Pi = (A,Z,\phi)$ is *stackable* if it is EHVZK and has recyclable third round messages.

Now, we present the Stacking Sigmas Compiler for stacking disjunctions of the same protocol (Section 7 of [Goe+21]). The compiler presented in Protocol 3 is identical to that in Figure 5 of [Goe+21], and is produced by Theorem 5 of [Goe+21].

Theorem 5 of [Goe+21]. Given a *stackable* Σ-protocol $\Pi = (A,Z,\phi)$ and a 1-out-of-l binding vector commitment scheme, we can produce a compiled Σ-protocol $\Pi' = (A',Z',\phi')$ that is also *stackable*. If Π is for the relation $\mathcal{R} : \mathcal{X} \times \mathcal{W} \to \{0,1\}$, then Π' is for the relation $\mathcal{R}' : \mathcal{X}^l \times ([l] \times \mathcal{W}) \to \{0,1\}$, where $\mathcal{R}'((x_1,\dots,x_n),(a,w)) := \mathcal{R}(x_a,w)$. The description of Π' is shown in Protocol 3, and the proof of this theorem can be found in [Goe+21].

Protocol 3. Stacking Sigmas Compiler. A compiler for composing n instances of a Σ-protocol Π into a single Σ-protocol Π' that proves the **disjunction** of these n instances.

Statement: $x = x_1,\dots,x_n$
Witness: $w = (\alpha, w_\alpha)$

- **First Round:** the Prover, P, computes $A'(x,w;r^P) \to a$ accordingly:
  - Parse $r^P = (r^P_\alpha \| r)$
  - Compute $a_\alpha \leftarrow A(x_\alpha, w_\alpha; r^P_\alpha)$
  - Set $\mathbf{v} = (v_1,\dots,v_l)$, where $v_\alpha = a_\alpha$ and $\forall i \in [l] \setminus \alpha,\ v_i = 0$.
  - Compute $(\mathsf{ck},\mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \{\alpha\})$.
  - Compute $(\mathsf{com},\mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp},\mathsf{ek},\mathbf{v};r)$.
  - Send $a = (\mathsf{ck},\ \mathsf{com})$ to the verifier.
- **Second Round:** V sends $c \leftarrow \{0,1\}^\lambda$ to P.
- **Third Round:** Prover computes $Z'(x,w,c;r^P) \to z$:
  - Parse $r^P = (r^P_\alpha \| r)$
  - Compute $z^* \leftarrow Z(x_\alpha, w_\alpha, c; r^P_\alpha)$
  - For $i \in [l] \setminus \alpha$, compute $a_i \leftarrow \mathcal{S}^{\text{EHVZK}}(x_i, c, z^*)$.

- Set $\mathbf{v}' = (a_1, \dots, a_l)$
- Compute $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}'; \mathsf{aux})$ (where aux can be regenerated with r).
- Send $z = (\mathsf{ck}, z^*, r')$ to the verifier.

- **Verification:** Verifier computes $\phi'(x, a, c_0, z) \to b$ as follows:
  - Parse $a = (\mathsf{ck}, \mathsf{com})$ and $z = (\mathsf{ck}, z^*, r')$.
  - Set $a_i \leftarrow \mathcal{S}^{\text{EHVZK}}(x_i, c, z^*)$
  - Set $\mathbf{v}' = (a_1, \dots, a_l)$
  - Compute and return:

$$b = (\mathsf{ck} \overset{?}{=} \mathsf{ck}') \wedge \left(\mathsf{com} \overset{?}{=} \mathsf{BindCom}(\mathsf{pp},\, \mathsf{ck}, \mathbf{v}'; r')\right) \wedge \left(\bigwedge_{i \in [l]} \phi(x_i, a_i, c, z^*)\right)$$

Communication Complexity. By using the Q-Binding construction (Figure 4 of [Goe+21]) together with Half-Bindings (Figure 2.2) we obtain a 1-out-of-2^q binding vector commitment scheme with communication complexity $q \cdot CC(\mathsf{HalfBinding})$ (**Theorem 2 & Corollary 1** in [Goe+21]). According to the proofs in Theorem 5 [Goe+21], the communication complexity of Π' is $CC(\Pi_l) = CC(\Pi) + |\mathsf{ck}| + |\mathsf{com}| + |\mathsf{r}'|$, where the last three components are directly related to the choice of the partially-binding vector commitment.

Since we are using a Q-Binding construction, we get the following communication complexity: $CC(\Pi_l) = CC(\Pi) + q(|\mathsf{ck}_{1/2}| + |\mathsf{com}_{1/2}| + |\mathsf{r}'_{1/2}|)$. From Figure 2.2, we can see that $|\mathsf{ck}_{1/2}| = |g|$, $|\mathsf{com}_{1/2} = 2|g|$, and $|r'_{1/2}| = 2|z|$, where $g \in \mathbb{G}$ and $z \in \mathbb{Z}_{|\mathbb{G}|}$. In this case, $|g|$ depends on the number of bytes required to represent the group elements in $\mathbb{G}$, and $|z|$ depends on the size of the group $|\mathbb{G}|$. Therefore, $CC(\Pi_l) = CC(\Pi) + q(3|g| + 2|z|) = CC(\Pi) + \log_2(l)(3|g| + 2|z|)$. Hence, the communication complexity grows logarithmically with the number of clauses l.

Chapter 3

Project Management

In this chapter, we discuss how we managed the project. We will cover the software development methodology we used, the various stages of the project and how it relates to our timeline.

3.1 Software Development Methodology

We use an agile software development methodology to manage the project. Specifically, we use a combination of Scrum and Kanban to manage our sprints and features. Moreover, we adopt Test Driven Development (TDD) to ensure that we have a thorough and robust verification workflow to ensure the correctness of our code.

We chose an agile software development methodology because there was a significant knowledge gap regarding the background material at the start of the project. As a result, it was difficult to accurately estimate the complexity of tasks and the time required to complete them. In the case that research and learning takes longer than expected, we would have to re-prioritise our tasks and change the scope of the project. Agile software development methodologies are well suited for this type of project because they allow for rapid iteration and adaptation to changing requirements. This actually proved useful in the later half of the project when issues with the code took longer to resolve than expected, causing delays. When this happened, we decided to prioritise the implementation of features and benchmarks and moved the implementation of security tests to the backlog. Even though this neglects and important aspect of cryptographic systems, it was necessary to ensure that we could complete the main objective of the project in time.

3.1.1 Test Driven Development

To prioritise functionality and validate that our features meet the requirements of the protocols, we use Test-Driven-Development (TDD) in our software development process. TDD is a programming style where tests are written based on requirements before features that aim to pass these tests are implemented. We show an illustration of the TDD software development lifecycle in Figure B.1 below.

TDD places an emphasis on first producing code that is minimally sufficient to pass the written tests. After which, code is refactored until it fulfils standard best practices such as readability, modularity, encapsulation etcetera. This appropriately prioritises functionality over readability, but still upholds written code to a high quality of design. This is especially so for this project as we prioritize correctness to ensure that we are providing meaningful results in our benchmarks.

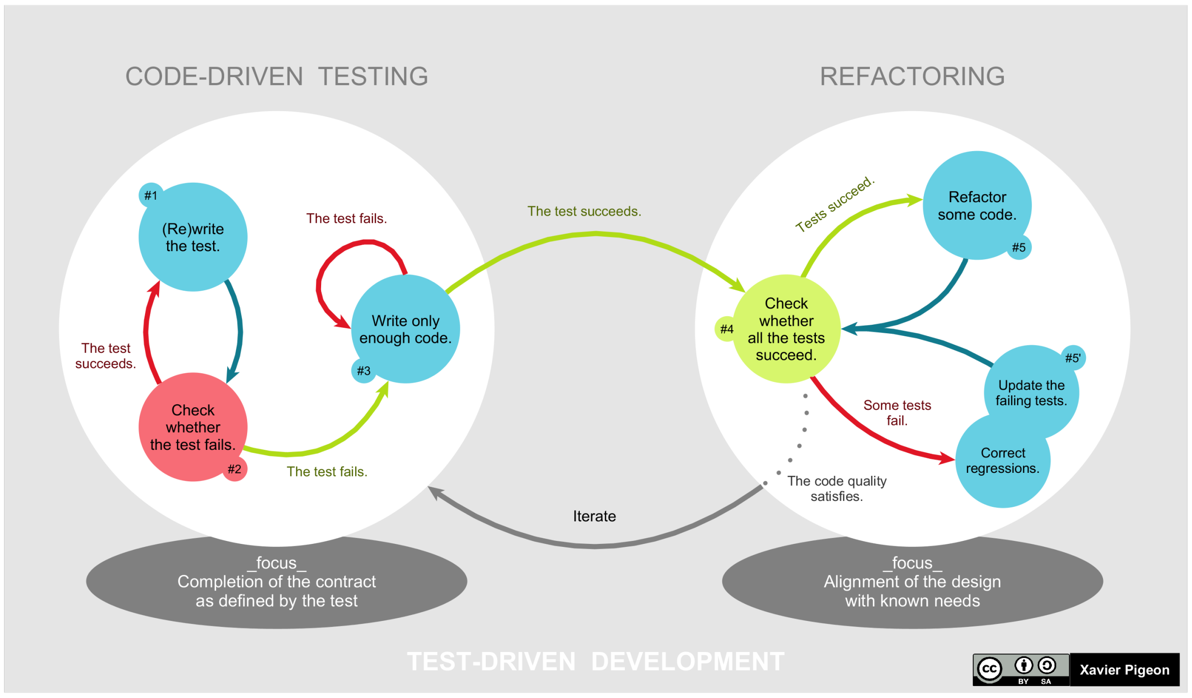


Figure 3.1: TDD Lifecycle [Pig15]

Writing tests before implementation also helps during regression testing, to identify bugs when changes are made. This will not only improve code quality, but also make development more efficient.

3.1.2 Scrum and Kanban

Scrum is an agile project management framework that emphasises on flexibility and continuous improvement of software [SS20]. It breaks down the development of a project into short iterations called sprints, which in our case are 2 weeks long. Meanwhile, Kanban is a another project management framework that share similar principles to Scrum and is often used in synergy with Scrum [SVY21]. Kanban focusses on visualising the workflow and limiting the number of features being worked on at any one time. This is done using a Kanban board, which is a visual representation of the tasks in the backlog, their current status, and the overall workflow. By controlling the number of features being worked on, it ensures that we complete features in a timely manner and do not run the risk of multitasking and causing delays.

In this project, we use Kanban to manage our features and Scrums to manage development of said features. We use the Kanban board provided by Github to track all the features that we have planned for the project.

1. Features are first added to the project backlog and then moved when the next sprint is planned. Features may be added to the backlog during development if need be, providing flexibility while keeping the current sprint manageable.
2. At the start of each sprint, we plan the group of features to work on for the next 2 weeks, and move them to the "To-do" column of the Kanban board. This column is also known as the "Sprint Backlog".

3. During the sprint, we move the features to the "Write Tests" column when we start writing tests for the feature, based on the requirements.
4. Once the tests are written, we start with the implementation and move the feature to the "Implement" column. Throughout the implementation phase, we regularly run tests to verify that the code is working as expected. We first focus on implementing a minimally functioning feature, and testing for correctness. After which, features are refactored to be more optimal if time allows for it. Tests may be refactored or added when necessary to be consistent with the updated code or requirements.
5. Once the feature is fully tested and verified to be working, we move it to the "Done" column, indicating that it is ready to be merged into our main branch.
6. At the end of the sprint, we review the features that were completed and ensure that the project is on track with the timeline. We also plan the next sprint, repeating the cycle again. Any additional features that were added to the backlog during implementation are usually added during this planning phase to the next sprint.

To summarise, we use Kanban as a visualisation tool and to control the features in progress, Scrum is used to manage the development of these features in sprints. Finally, TDD is used to ensure that the code is correct and that the features meet the requirements.

3.2 Timeline

We split the project into 3 main stages: research and learning, development of CDS94, and development of Stacking Sigmas. The two development stages also include benchmarking the respective compilers and anlaysing the data collected. We did not create a separate stage for the benchmarks because they were best done in parallel with the development of the compilers. In Table 3.1, we show the overall timeline of the project. In the "Goal" column, we indicate with a checkmark if the goal was achieved. Comments are provided in brackets where relevant.

Research and Learning. In the first stage of the project (6 weeks), time was dedicated to learning any background material and to study the literature. This was also the time when we familiarised ourselves with the programming language of choice: Rust. Tools and libraries that ended up being helpful to us in the project were also discovered during this period, through research and experimentation. It was imperative that we spent this time understanding the literature, as it would otherwise be difficult to implement the protocols accurately. To ensure that we were ready to start development, we organised meetings with the project supervisor dedicated to clarifying details on the research material and to validate our understanding of the background material as a whole.

Development. The second and third stages (18 weeks) are similar in that they are dedicated to the development of the project. Yet, we decided to separate them into 2 stages as we anticipated that the time taken for the first stage (research) may be longer than we expect. In this scenario, we would be able to quickly re-prioritise and change the scope of the project if necessary. Fortunately, this was not the case and we were able to complete our initial objectives in time.

This development phase is split into sprints: each of 2 weeks, except during the Christmas break where it was extended to 4 weeks. The CDS94 compiler was completed in the first 2 sprints, which surpassed our expectations. However, this did not include the implementation of the benchmarks and

Table 3.1: Overall Project Plan

Sprint	Main Task	Goal
T1 W1-2	Conduct necessary background reading and research.	Submit project specification. ✓
T1 W3-4	Understand the literature and extract requirements from the design of CDS94.	Must be capable of explaining CDS94 protocol to project supervisor. ✓
T1 W5-6	Familiarise with Rust and research libraries to use for implementation.	Finalise features to develop for CDS94 ✓
0 (T1 W7-8)	Begin implementation of CDS94.	Submit progress report. ✓
1 (T1 W9-10)	Implementing CDS94.	- (Completed goal in the next sprint early)
2 (Christmas Break)	Continue work on CDS94. Start implement benchmarks. Study and research [Goe+21].	~~Complete CDS94~~
3 (T2 W1-2)	Benchmark tests for CDS94 and collect data. Begin implementation of Stacking Sigmas.	Complete benchmarks for CDS94. ✓
4 (T2 W3-4)	Implementing Stacking Sigmas. Update benchmark tests if necessary.	Complete implementation of Stacking Sigmas. (Delayed: 1 sprint)
5 (T2 W5-6)	Implementing Stacking Sigmas. Benchmark Stacking Sigmas compiler and organise data collected for final presentation.	Able to explain concepts in [Goe+22] to project supervisor. (Delayed: 1 sprint)
6 (T2 W7-8)	Preparation for final presentation. Attempt basic implementation of Speed Stacking.	Present data to supervisor before presentation ✓
7 (T2 W9-10)	Continue with Speed Stacking.	Final Presentation. ✓
Easter Break	Write up of final report alongside exam revision.	Submit first draft of report for review (✓). Complete Speed Stacking (No longer possible).
T3 W1-2	Final proof reading and editing of report.	Submit final report.

benchmarking CDS94. The benchmarks were completed over the next 2 sprints before work on the Stacking Sigmas compiler began. Unforunately, this took longer than expected due to the complexity of implementing the partially-binding vector commitment scheme. Furthermore, issues with the code were discovered during testing, which took an entire sprint to solve. During this period, we decided to not prioritise security testing on our compilers, and focus on correctness and collecting data from the benchmarks instead. In the end, development and benchmarking of Stacking Sigmas required 3 sprints, which is longer than expected but still within the time frame of the project.

We also mentioned the possibility of implementing the Speed Stacking compiler [Goe+22] in the project specification, and started work on it, but ultimately are not able to complete it in time for this report. That said, we still deem the project a success as we were able to complete the CDS94 and Stacking Sigmas compilers and collect insightful data from the benchmarks.

3.3 Project Management Tools

We use Github to manage our entire project. Github is a web-based hosting service for version control using Git, and it also provides project management tools. Notably, it provides a Kanban board with "Github Projects" and sprint management with "Github Milestones". We manage features using the Kanban board, creating drafts when we are planning and converting these features into "Github Issues" when we are ready to start development. This automatically creates a Git branch for the feature, which we can work on and subsequently merge into our main branch when the feature is complete.

3.4 Risk Management

Below we list the risks that we identified at the start of the project and how we mitigated challenges that arose during development.

- **Lack of experience.** We were not familiar with the Rust programming language or the background material initially and planned the project with this in mind. This is why we dedicated the first 6 weeks to learning the language and the literature. Additionally, to prepare for the case that this phase would take longer than expected, we opted for an agile development methodology to remain adaptive to changes in the project scope if necessary.
- **Unexpected challenges.** To prepare for unexpected delays in development, we planned the project with a clear priority of the requirements: correctness firstly, then usability and generality, followed by performance and security. This allowed us to focus on the most important aspects of the project first, and to prioritise the remaining features accordingly. Splitting the development of the two compilers into separate stages also allowed for the possibility of changing the scope to focus on solely CDS94 if necessary.
- **Machine Failure.** We used proper version control practices with the aid of Git and Github to ensure that work can continue even if a machine fails. We also verified that our programming tools were compatible with the university machines. This came in useful when we had to switch to a different machine due to a hardware failure.
- **License Agreements.** The project relies on many existing libraries for cryptographic primitives and other utilities. We ensured that the licenses of these libraries were compatible with our project.

Chapter 4

Design & Implementation

In this chapter, we outline and discuss the design of the project, and our implementation of salient components. We first begin with a brief overview of our core requirements, and justify certain design choices. Next, we discuss of how we approach testing – a crucial process in the verification of our software components. This is followed by a section discussing the design choices we make for our benchmarks (Section 4.3). After which, we elaborate on the general approach we take in implementing the protocols described in the literature. Finally, we will elaborate on the design of important components and their implementation.

Here, we include the components associated with each compiler for easy reference:

1. CDS94 Compiler
   - Schnorr's Protocol
   - Shamir's Secret Sharing
   - CDS94 Compiler
2. Stacking Sigmas Compiler
   - Schnorr's Protocol
   - Partially-Binding Vector Commitment scheme: Q-Binding of half-bindings
   - Self-Stacking Compiler

4.1 Overview

In this section, we elaborate on the core requirements that influence the design choices of every component, discuss our choice of programming language, and justify our choice of libraries that we use throughout various components in the project.

4.1.1 Core Requirements

In the design of each component, we consider the following three core requirements: *correctness*, *generality*, and *usability*.

Correctness. Unsurprisingly, correctness has the highest priority, as we want to ensure that we are accurately comparing the performance of the compilers. In Section 4.2, we discuss how we test our compilers to ensure correctness.

Generality. This requirement is concerned with how easy it is for future developers to use our interfaces[1] and components for their own work. One of the motivations for this project is to lay the foundations for future researchers to conveniently compare their own implementations of existing or new compilers to ours. Additionally, this will also help developers to easily build on our work in the future. With this in mind, it is important that our code is easily extensible and has components (which may be shared) that are modular. Thus, we have organised our source code such that each component is an individual library and can be selectively chosen. This is coupled with interfaces designed to be as general as possible, allowing developers to choose which components they want to use, and which to implement themselves.

Usability. We strive to make our compilers easy to use, understand, and maintain using thorough documentation and convenient methods to easily use the components. In the following sections, we discuss how we attempt to achieve this in each component.

4.1.2 Programming Language

Our choice of programming language is Rust: a modern language that is designed with a focus on safety and reliability. It promises memory safety, by using an ownership and borrowing system unique to Rust, and it is a strongly typed language which helps to identify errors at compile-time. This helps to prevent errors such as integer overflows and null pointer references. Furthermore, Rust is known for its high performance and speed [SR22a], and has functional programming features such as pattern matching and closures. These features make Rust a good choice for our project, particularly because many of the components we are implementing are defined mathematically which lend themselves well to functional programming. Meanwhile, robustness and speed are highly ideal properties in any system, especially one that relies on cryptographic protocols.

Additionally, Hall-Andersen's [Hal21] Stacking Sigmas compiler is also implemented in the language. This makes Rust a natural choice for our project, it allows us to easily compare our implementation to Hall-Andersen's. Rust has other useful benefits that pertain to more specific components of our project, such as testing; we highlight these in their respective sections.

4.1.3 Libraries

Throughout this project, a number of notable libraries are repeatedly used across different components. The few that we would like to highlight are:

1. `curve25519-dalek` [LV22]: a Rust library that provides group operations on the Ristretto group [Val+]. In many of our protocols, a prime order group is required; we use this library as the underlying prime order group in the concrete implementations of our project. We choose the Ristretto group because it is a prime order group constructed from a non-prime-order Edwards curve [Edw07] (a family of elliptic curves), which is known for both security and speed. At the

[1]Rust has a different terminology for interfaces called "traits". While traits and classic interfaces are not exactly the same, they share many commonalities. For the sake of simplicity, we will substitute terms in Rust associated with traits with those associated with classic interfaces. Where this is not possible, we will explicitly use Rust terminology and explain the difference.

same time, the Ristretto group does not suffer from the same issues as other modern elliptic curve implementations that do not provide a prime-order group[2].

2. `rand_chacha` [HCC21]: a random number generator (RNG) that uses the "ChaCha20" stream cipher [Ber+]. We use this to create RNGs that are needed for our protocols. Notably, we chose this library specifically because it also provides seedable RNGs, which is useful for testing.

3. `group` [str+22]: a library that provides general interfaces for groups and fields. In the pursuit to make our components as general as possible, we make use of the interfaces in this library to define the kind of groups and fields that are accepted by our interfaces. When users develop their own concrete implementation of our interfaces, they can choose to use any group or field that implements the interfaces in this library and are not restricted to the concrete implementations that we provide.

4.2 Testing

As mentioned in Project Management (Chapter 3), we use Test-Driven Development to ensure correctness. By writing tests before features, we encourage focus on the requirements of our software and ensure that we design our components to target these requirements. Once these tests are written, we have an efficient workflow to consistently ensure our software is correct, even as we add new features. This improves development speed and reduces the time needed to identify and fix bugs.

In addition, we perform extensive static analysis to complement testing and improve debugging. To support static analysis, we intentionally design our components and interfaces to reflect their mathematical definitions in the literature. A good example of this is our implementation of q-bindings and half-bindings (Section 4.5.4). Doing this allows us to easily verify that our implementations adhere to their mathematical definitions as closely as possible, increasing the chances of identifying an issue if there is one.

Unit & Integration Testing. We use the built-in testing framework in Rust to test our components. This framework allows us to write unit tests within the same source code file as our implementation, and conveniently run them within our IDE (VSCode) which consequently speeds up development. We also write integration tests with the same framework, except that they are often located in a separate file to emphasise that they are testing multiple components as these components have to be explicitly imported into the scope of the this testing module.

Code Coverage. To complement testing, we use code coverage reports to ensure that we are testing as much of our code as possible. Code coverage is a way to track and determine which sections of code (e.g. functions, branches, lines etc.) are executed. This is useful for us as it highlights sections of code that are neglected by our existing tests and ensures that we cover important test cases. We use the `cargo-llvm-cov` [End23] tool to carry out code coverage analysis. This tool is a wrapper around the Rust compiler's in-built code coverage tool, providing a convenient way to generate code coverage reports as they integrate directly with our testing suite.

[2]Modern elliptic curve implementations usually provide a group of order $h \cdot q$, where h is a small cofactor (4 or 8), and q is a large prime number. When using these implementations for protocols that require a prime order group, the abstraction from non-prime order group to prime order group is often handled by developers up the stack (by users of the library). This means that they are not as familiar with the underyling implementation, and may introduce vulnerabilities due to subtle design complications.

4.2.1 Extracting Requirements & Writing Tests

Throughout this project, the requirements of our software components are mostly derived from the literature with the exception of our benchmarks. Therefore, our testing strategy is centered around testing these requirements against our implementation. Of course, additional unit tests are written to test the correctness of specific functions as well, but we will not discuss these in detail.

Extracting these requirements is a manual process that requires a thorough understanding of the literature. Our approach begins with reading the literature and writing down the requirements explicitly. This is a tedious process, but it is necessary to ensure that we have a complete understanding of the protocols and what is required of them. We then translate these requirements into tests.

1. Usually, the first step is to complete a function for setting up mock data that is likely to be used in the tests. We design this function to take in a set of parameters and returns a set of mock data. These can then be separately called in each test, with parameters, to generate data specific for that test.
2. Next, we write the content of the tests themselves. Because we are desining our components to model the mathematical definitions of the protocols, we can often easily determine the exact methods to call. Often, a challenge in TDD is not knowing what functions are available to test because they are not yet implemented. However, this is not a problem in this project as we have the definitions in the literature to refer to. There are two main kinds of tests that we write:
   - **Positive Tests:** These are tests that ensure the implementation works as expected.
   - **Negative Tests:** These are tests that ensure the implementation fails as expected.
3. Following this, we proceed to implement our components. This often starts with the interfaces (if any), and then to any classes that are required.
4. Finally, we run the tests to ensure that they pass. If they do not, we investigate and either fix the implementation or the tests.

In Appendix A.6, we provide an example of our tests with those for the CDS94 compiler. Often the first tests that are written target the main requirements that we glean from the literature. As development progresses, we continue to write tests or extend existing ones to ensure that we are testing comprehensively.

4.3 Benchmarks

In our benchmarks, we are concerned with measuring the growth of certain metrics when the number of clauses in the disjunction increases. Suppose we have a disjunctive zero-knowledge proof $\Pi = (A, Z, \phi)$, then the metrics we are concerned with are as follows:

1. **Communication Complexity:** size of communication between the prover and verifier (in bytes). This is the size of the messages $a \leftarrow A$, $z \leftarrow Z$, and $c \leftarrow \{0,1\}^{\kappa}$.
2. **Prover Computation Time:** time taken for the prover to run the algorithms A and Z
3. **Verifier Computation Time:** time taken for the verifier to run the algorithm ϕ.

Crucially, we are also interested in learning the **total computational time** of the compiled proof, which we can easily computed by summing the prover and verifier running times.

We target these three metrics as they are the main properties that are measurable and are used for evaluating the performance of the compiled proofs. Other properties such as security limitations (e.g. the requirement for a trusted setup) are not measurable, and thus are not considered in our benchmarks.

The range we use for our clauses is $(2, 2^2, \ldots, 2^{13})$; this means that at each step, we double the number of clauses until we reach $2^{13} = 8192$ clauses. Theoretically, the maximum number of clauses our compilers can accept is much larger than this[3], however the proofs take too long to run after a certain number of clauses. After consulting the project supervisor, we decided that this is a suitable range for our benchmarks, as it allows us to measure a large range while ensuring benchmarks keep to a reasonable running time.

4.3.1 Benchmarking Tools

Next, we discuss the choice of tools we use to construct and analyse our benchmarks.

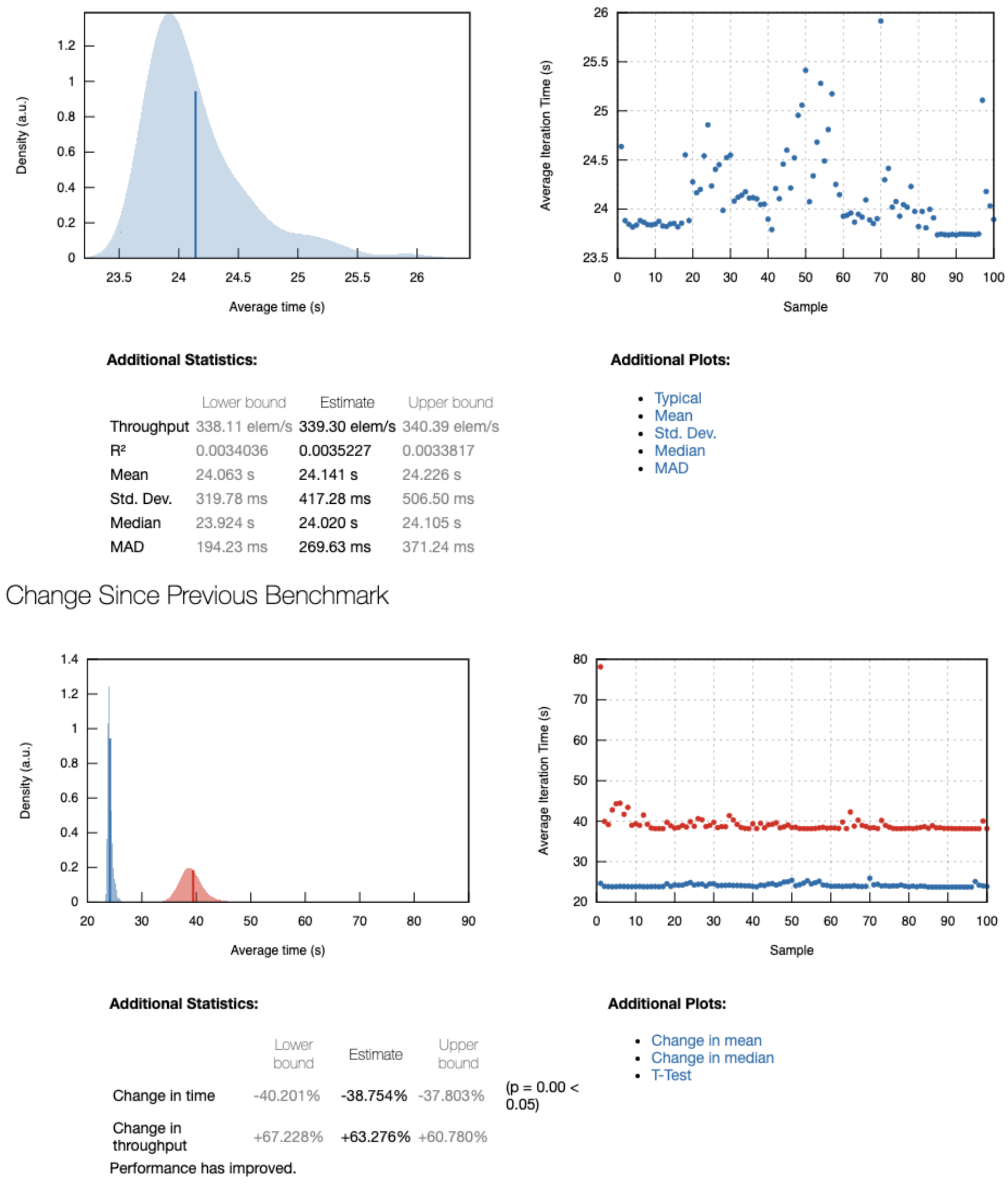
Criterion. We use the `criterion` library [HA22] to construct our benchmark tests. Compared to the standard benchmarking library provided by the Rust language [Org], `criterion` is a statistics driven benchmarking library that provides a simple API for writing benchmarks, automatically providing the mean, standard deviation, and the confidence interval for each benchmark. It also includes useful defaults such as the number samples to run for each benchmark, a standard warmup time before collecting, and a HTML report generator. We provide examples of the HTML report generated automatically by `criterion` in Figure 4.1 below. Naturally, after some initial research we made the decision to use `criterion` due to the abundance of useful features and applicability to our project.

Helper Interfaces. In our design, we also decided to include a `Message` interface that aids in the measurement of the communication size of each benchmark. This interface is inspired by one of the same name in Hall-Andersen's implementation [Hal21].

This interface requires that the type implement a few methods that are ultimately used to measure the size of the messages sent between the prover and verifier. By implementing this interface, future developers can easily measure the size of these messages within benchmarks with a single call to the `size()` method. Furthermore, we improved on the interface by requiring that it "inherit" another interface, `Default`, which asserts that a method is written for constructing a default instance of the type that implements `Message`. This provides further utility when writing tests and benchmarks, and helps to improve the overall usability of the compiler.

Data Analysis. To analyse our data, we collect the mean of each benchmark provided by `criterion` [HA22] and save it in a CSV (comma-separated values) file. We then employ the use of Python data analysis libraries `pandas` [tea23; McK10] and `plotly` [Inc15] to parse and plot the data respectively. We chose these two libraries as they are widely used in the data science, and have the necessary functions we need to parse data and produce high quality plots.

[3]The limit for CDS94 is 2^{252} (constrained by the size of the Ristretto group [Val+] that we use for Shamir's secret sharing), and for Stacking Sigmas the number is theoretically unbounded.



Additional Statistics:

	Lower bound	Estimate	Upper bound
Throughput	338.11 elem/s	339.30 elem/s	340.39 elem/s
R²	0.0034036	0.0035227	0.0033817
Mean	24.063 s	24.141 s	24.226 s
Std. Dev.	319.78 ms	417.28 ms	506.50 ms
Median	23.924 s	24.020 s	24.105 s
MAD	194.23 ms	269.63 ms	371.24 ms

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

Change Since Previous Benchmark

Additional Statistics:

	Lower bound	Estimate	Upper bound	
Change in time	-40.201%	-38.754%	-37.803%	(p = 0.00 < 0.05)
Change in throughput	+67.228%	+63.276%	+60.780%	

Performance has improved.

Additional Plots:

- Change in mean
- Change in median
- T-Test

Figure 4.1: Example of the HTML report generated by `criterion`. The first image shows an example of the additional statistics that are conveniently provided in the report. The second image shows an example of the performance comparison between two different benchmark runs. If the change in performance falls within a certain threshold, the change is marked as significant.

4.3.2 Implementation of Benchmarks

Our goal is to measure the metrics shared earlier and to visualise the data on a plot. The main challenge in this task is to ensure that the benchmarks are accurate and reliable – only necessary computation should be measured, and appropriate calculations should be used to measure communication size. Hence, tests are not suitable for evaluating the validity of our benchmarks, and will require a more manual process of accessing the data and ensuring that the benchmarks are appropriate. In the following subsections, we discuss how we implement our benchmarks to achieve these goals.

Computation Time

Recall the definition of a Σ-protocol $\Pi = (A, Z, \phi)$. To ensure that we are measuring an accurate estimate of the computation time of the prover and verifier, we design and implement the benchmarks such that they isolate the execution of the prover algorithms (A and Z), and the verifier algorithms (ϕ and $c \leftarrow \{0,1\}^{\kappa}$). The key idea is that we are implementing a benchmark, hence there is no need to run the Σ-protocol in order. We simply precompute the challenge c from the second round of the protocol without measuring the time taken to compute it. Next, we execute algorithms A and Z in the prover's benchmark, collecting data on the computation time. Within this benchmark, there may be additional executions that we are not directly interested in but are necessary. Here we provide an example from the benchmarks of CDS94:

```rust
// Precompute challenge
let challenge = Scalar::random(&mut verifier_rng);

// Intialize variables to store results from first and third round
let mut message_a: Vec<CompressedRistretto> =
    Vec::new();
let mut message_z: Vec<(usize, Scalar, Scalar)> =
    Vec::new();

// Benchmarking environment for Prover benchmark
group.bench_with_input(
    BenchmarkId::new("prover_bench", &proverparams),
    &mut proverparams,
    |b, s| {
        b.iter(|| {
            // Code within these braces are being benchmarked and measured
            // Should have negligible cost
            let prover_rng = &mut s
                .prover_rng
                .clone();
            // First round of protocol
            let (transcripts, commitments) =
                SelfCompiler94::first(
                    &s.statement,
                    &s.witness,
                    prover_rng,
                );
            // Third round of protocol
            let proof = SelfCompiler94::third(
                &s.statement,
31                 transcripts,
32                 &s.witness,
33                 &s.challenge,
34                 prover_rng,
35             );
36             // Should have negligible cost
37             message_a = commitments;
38             message_z = proof;
39         })
40     },
41 );
42 
43 /// ...
44 
45 let verifier_rng = &mut ChaCha20Rng::from_entropy();
46 
47 // Benchmarking environment for Verifier benchmark
48 group.bench_with_input(
49     BenchmarkId::new("verifier_bench", &v_params),
50     &v_params,
51     |b, s| {
52         b.iter(|| {
53             // Re-execution of second round
54             SelfCompiler94::<Schnorr>::second(
55                 verifier_rng,
56             );
57             // Verification algorithm
58             SelfCompiler94::verify(
59                 &s.statement,
60                 &s.message_a,
61                 &s.challenge,
62                 &s.message_z,
63             )
64         })
65     },
66 );
```

On line 17 and 36 of the code listing, we indicate (with comments) the blocks of code that are necessary and should have negligible cost. These are needed to ensure the protocol executes without error and that the output from the first and third round can be collected for the verifier's benchmark. Within the verifier's benchmark, we re-execute the second round of the protocol (line 53) to ensure that the verifier's computation time for this round is also measured even though the resulting challenge is not used in the verification algorithm.

Communication Size

As mentioned earlier, we use the `Message` interface to conveniently compute the communication size of each benchmark. Once the interface is implemented for each required type, measuring the size of the messages can be done with a single call to the `size()` method.

```
let mut communication_sizes: Vec<usize> =
      Vec::with_capacity(Q - 1);
```

```

/// ...

communication_sizes.push(
    message_a.size()
        + message_z.size()
        + challenge.size(),
);
```

To calculate the total communication size, we simply sum the sizes of the messages from each round of the protocol. We append this to a vector of communication sizes for each number of benchmarks, and use this to plot the growth of communication size against the number of clauses.

4.4 General Design Approach for Compiler Components

Our general approach to designing our compiler-related software components is inspired by the well known "SOLID" design principles [Mar00] of object-oriented programming. Crucially, we ensure that our general design approach contributes directly to our core requirements of generality and usability. We split our compiler-related software components into two broad categories: general interfaces and concrete implementations. The general interfaces define the structure of protocols, while the concrete implementations are the actual implementations of these interfaces (with possible extensions).

Firstly, we want to ensure that that we design our concrete implementations such that they *accept general types that implement these interfaces* instead of the concrete types directly. For example, our Stacking Sigmas compiler is designed to accept a generic type that requires an implementation of the `SigmaProtocol` interface and the `EHVzk` interface, instead of accepting the concrete type, which in our case is Schnorr's protocol. This allows developers to use the Stacking Sigmas compiler with any Σ-protocol implementation, as long as they implement the relevant interfaces. This also supports our choice to use a microservices architecture and organising our code into separate modules for each component. By doing this, modules which define our interfaces should not be importing concrete types; modules defining concrete types should only import the interfaces they need to implement and explicitly import concrete types that are absolutely necessary.

We also want to make sure that our interfaces have *a single responsibility* and do not encompass too many functionalities. For example, we intentionally segregate the two interfaces `SigmaProtocol` and `EHVzk`, even though we require both of them to be implemented for the Stacking Sigmas compiler. This is so that we do not assume how future users will use our interfaces. For the CDS94 compiler, there is the `HVzk` interface. We know that the HVZK property can be thought of as a superset of the EHVZK property because a Σ-protocol that is EHVZK is HVZK, but not necessarily the other way around. With this in mind, we should not need to implement the simulator as defined in the context of the `HVzk` interface if it is already defined in terms of `EHVzk`. Likewise, a future user may choose to develop a `SigmaProtocol` that has a different requirement for the zero-knowledge property. By segregating the interfaces into atomic functionalities, we provide users with more flexibility in how they use our interfaces.

Lastly, as mentioned in the testing section 4.2, to support static analysis and improve the readability of our code, we *design our interfaces to reflect the mathematical definitions* of the protocols as much as possible. This facilitates easy comparison between our implementations and the mathematical definitions, and helps to verify that our implementation is sound.

4.5 Design & Implementation of Key Components

In this section, we will highlight the design and implementation of important components of our project in their respective subsections.

4.5.1 Sigma Protocols

We model Σ-protocols with an interface. Firstly, we design a set of generic types associated with the interface which will fit into the definition of the methods in the interface. These types are:

- `Statement`: Public information about the protocol.
- `Witness`: Private information about the protocol (only known to Prover).
- `MessageA`: The first message of the protocol, sent from prover to verifier.
- `Challenge`: The challenge sent by the verifier to the prover.
- `MessageZ`: The third message of the protocol, sent from prover to verifier.
- `State`: For most Σ-protocols, there is a particular state associated to the execution of the first message. This state is often used by the prover again in the third message but must not be sent to the verifier.

In particular, the design of the generic `State` type is worth discussing in more detail. It is not explicitly mentioned in the formal definition of Σ-protocols, as it is assumed that the prover is able to track and store the private values that they obtain and require in the protocol. In the implementation, this has to be explicitly modelled and we do this using a functional programming approach, instead of an object-oriented programming (OOP) approach. Firstly, this is because the functional approach is simpler, as we do not need to implement separate interfaces for the prover and verifier and subsequently include them as fields within the `SigmaProtocol` interface. Secondly, the functional approach is more flexible, as it does not restrict the user to a particular way of implementing the prover or verifier. The functional approach simply provides two values in the first message (the state and the actual message), and the user of our interface can decide how to use these values.

Now, we present the methods that are associated with our Σ-protocol interface. These methods are:

- `first`: the first message of the protocol. This models algorithm $a \leftarrow A(x, w; r^P)$ in Definition 5.
- `second`: the second message of the protocol. $c \leftarrow \{0, 1\}^{\kappa}$.
- `third`: the third message of the protocol. $z \leftarrow Z(x, w, c; r^P)$.
- `verify`: verifies the transcript. $b \leftarrow \phi(x, a, c, z)$.

In Appendix A.1 we provide a code snippet of these methods within the interface, which outlines which generic types are given as input and which are returned as output. Referring to the code snippet, readers will observe that our methods almost model the algorithms in Definition 5 exactly in terms of input and output. The only difference is with the `State` type that we have already discussed.

4.5.2 Schnorr's Protocol

As a Σ-protocol, our first step in our implementation for Schnorr's protocol is to create concrete definitions for the methods and generic types of the `SigmaProtocol` interface. We use the following concrete types in the interface:

- `Statement`: `Schnorr` – a type that simply contains the "public key" (which is the group element $H = x \cdot G$) as a field. We use the `RistrettoPoint` type from `curve25519-dalek` to represent the public key, and it is essentially a point on an Edward's curve.
- `Witness`: `Scalar` – also from the `curve22519-dalek` library. It represents elements of the prime field in the Ristretto group.
- `MessageA`: `CompressedRistretto` – essentially the same as the `RistrettoPoint` type, but differs in its representation. `CompressedRistretto` is a compressed representation of the point, which is more efficient to store and transmit.
- `Challenge`: `Scalar`
- `MessageZ`: `Scalar`
- `State`: `Scalar`

The key takeaway is that we assign concrete types to the associated generic types provided in the interface for our implementation of Schnorr. The methods in the interface are then defined according to our definition of Schnorr in Definition 8.

4.5.3 Shamir's Secret Sharing

For our implementation of Shamir's Secret Sharing, we use Lagrange's interpolating polynomial to interpolate the x and y coordinates that correspond to shares. We chose to use Lagrange's interpolation formula because it is simple to implement, albeit not the most efficient. This trade off is acceptable as the main goal for the project is to measure the performance change with respect to the number of clauses, and not the pure performance of the compiler. A more efficient implementation is therefore not a priority, and can always be implemented in the future if needed.

Initially, we used an existing library `vsss-rs` [LH23] as our implementation of Shamir's secret sharing scheme. However, we encountered a few issues with this library. Firstly, the library does not provide the functionality to complete a qualified shares given the secret and an unqualified set of shares. This is not surprising as this is not the main use case for a secret sharing scheme like Shamir's. However, this is a necessary feature for our compiler as we use it to ensure that the prover is unable to arbitrarily select challenges to cheat the verifier. We initially tackled this by simply adding the missing functionality to the library, in the form of new functions. However, we later found out about the second issue with the library: its implementation of Lagrange's algorithm is inefficient.

In their implementation, they call the `invert` method in the inner-loop of the algorithm resulting in n^2 calls to the function, where n is the number of shares. The `invert` method is required to perform division on our group elements, but it is a costly operation. This can be avoided, however, by computing the numerator and denominator separately and performing the `invert` function only n times – outside of the inner-loop. The following snippet of code shows the comparison of the two implementations.

```
// Inefficient implementation
for i in 0..n { // outer-loop
  for j in 0..n { // inner-loop
    if i != j
      basis *= (x - x[j]) * (x[i] - x[j]).invert()
    result += basis * y[i]
  }
}

// More efficient implementation
for i in 0..n {
  for j in 0..n {
    if i != j {
      numerator *= x - x[j]
      denominator *= x[i] - x[j]
    }
  }
  result += numerator * denominator.invert() * y[i]
}
```

Due to this issue, we created our own implementation of Shamir's secret sharing scheme. We represent a Lagrange polynomial with two fields: x-coordinates and y-coordinates which are both vectors of the same length. For a polynomial with degree d, the vectors will have length d as well.

An alternative we considered is to represent the polynomial with a vector of coefficients, where the i-th element of the vector is the coefficient of the i-th degree term. This representation is useful in the method for distributing shares given the secret, as we can use Horner's method to evaluate the polynomial at a given point efficiently [Rem17]. However, CDS94 does not rely on the secret sharing scheme's functionality for distributing shares, and instead uses the Complete algorithm (Definition 12) to complete a qualified set. For this algorithm, we need to use polynomial interpolation as we are given the secret and an unqualified set of shares (and equivalently points on the polynomial). Therefore, we choose to represent the polynomial in this way because it is more natural for use in Lagrange's interpolation formula (Definition 10).

4.5.4 Half-Binding & Q-Binding

Before implementing half-binding and q-bindings, we implement the `PartialBindingCommScheme` interface to model the definition of partially-binding vector commitments provided in Definition 13. This interface has methods that correspond to each function in the definition: `setup`, `gen`, `bind` (equivalent to BindCom), `equiv`, and `equivcom`. The interface also has a set of generic types to complete the definition of these methods (similar to how they function in the `SigmaProtocol` interface). These generic types are:

- `PublicParams`: public parameters for the commitment scheme.
- `BindingIndex`: a type that represents the binding index(es) of the commitment.
- `CommitKey`: a type that represents the commitment key.
- `EquivKey`: a type that represents the equivocation key.
- `Commitment`: a type that represents the commitment.

- `Randomness`: a type that represents the auxiliary values that are needed for equivocation.
- `Msg`: a type that represents the vector of messages.

With this interface, we implement `HalfBinding` and `QBinding` according to their construction in Figure 2.2 of this report, and Figure 4 of [Goe+21] respectively. The definition of q-binding is recursive in nature, where the base case is the half-binding scheme. In an attempt to improve the usability of our q-binding implementation, we introduce some overhead in how we represent the tree of half-bindings. Intuitively, this could simply be a recursive data structure, in which the every node has two children which continue down until the leaves. However, due to Rust's strict type system and our limited experience with the language, we were unable to implement this without heavily sacrificing the usability of the interface.

Instead, we represent the tree as a set of vectors of data types, where each index in the vector represents a layer of the tree. We have multiple vectors for each concrete type for the half-binding implementation of the `PartialBindingCommScheme` interface. While this introduces some overhead as the vector must be traversed if we want to copy the data, it allows us to implement the interface in a way that has good usability and is still easy to understand. We hypothesize that this is the main reason our implementation of Stacking Sigmas is slower than Hall-Andersen's implementation [Hal21]. This will be discussed in Chapter 5 in more detail.

4.5.5 `HVzk` & `EHVzk` Interface

These two interface encompasses the honest-verifier and extended honest-verifier zero-knowledge property resepectively, which means that the only required method for these two interfaces is a simulator method (Definition 5) for the concrete Σ-protocol that implements this interface. As mentioned in the section on the general approach to design, we split this interface from the `SigmaProtocol` interface to adhere to the "single responsibility" principle in "SOLID" design [Mar00]. We provide code-snippets of these two interface in Appendix A.2.

4.5.6 CDS94 Compiler

We implement the CDS94 compiler for disjunctions over the same Σ-protocol. This means we are concerned with many instances of the same Σ-protocol, even though it is possible to use CDS94 to compile disjunctions over different Σ-protocols. We do this to simplify our implementation and to focus on the most basic implementation of the CDS94 compiler. The disjunctions that this compiler supports are d-out-of-n disjunctions, where d is the minimum number of active clauses, and n is the total number of clauses. At a high level, the design of the CDS94 compiler is not complex, and simply follows the construction provided in Protocol 2. That said, we make certain design decisions that are worth highlighting.

Firstly, we cannot only implement the `SigmaProtocol` interface for the underlying Σ-protocol that we want to use with the CDS94 compiler. This is because the CDS94 compiler requires further restrictions on the generic types that is used during a concrete implementation. Notably, Shamir's secret sharing scheme requires that the shares passed to it have x and y coordinates that are field elements. This means we need to provide a way to map the generic `Challenge` type (from `SigmaProtocol` interface) to a type that implements the `PrimeField` interface from the `group` library (Section 4.1.3). For Schnorr's protocol, this is not a theoretical issue because the challenges that are used are field elements and can be used directly with Shamir's secret sharing. However, in practice, the `Scalar` type from the `curve25519-dalek` library does not implement the required interface.

An alternative is to directly require that the `Challenge` type parameter in the `SigmaProtocol` interface to implement the `PrimeField` interface. However, this is arguably more restrictive than necessary. This is because, we may have existing implementations of a Σ-protocol that does not use a challenge that is a field element, but we still want to use it with the CDS94 compiler. In this case, we can simply define a mapping function that maps the challenge to a field element and use it with Shamir's secret sharing. Hence, to capture this relationship, we introduce the `Shareable` interface.

`Shareable`. A simple interface to ensure type-level compatability between the generic types used in the CDS94 compiler and the generic types used in Shamir's secret sharing scheme. This interface is defined as follows:

```rust
pub trait Shareable: Clone + Default {
    type F: PrimeField;
    // Map the type to a prime field element
    fn share(&self) -> Self::F;
    // Derive the instance of the type from a field element
    fn derive(elem: Self::F) -> Self;
    // Convert field element into usize
    fn to_usize(elem: Self::F) -> usize;
}
```

- The concrete type that implements this interface can be any type that also implements the `Clone` and `Default` traits.
- The `F` type parameter is a type that implements the `PrimeField` interface from the `group` library.
- The `share` method is a function that maps the generic type to the `F` type parameter.

This allows developers to implement the `Shareable` interface for any type that they want to use in the CDS94 compiler, as long as it is possible to define a mapping function from the chosen type to one that implements the `PrimeField` interface. For example, in the case of Schnorr's protocol, the `Challenge` type is a `Scalar` from the `curve25519-dalek` library. This type does not implement the `PrimeField` interface, but we can implement the `Shareable` interface for it as follows:

```rust
impl Shareable for Scalar {
    type F = WrappedScalar;
    // ...
}
```

where `WrappedScalar` is a wrapper type that implements the `PrimeField` interface for the `Scalar` type[4].

`Composable`. With the `Shareable` interface, we can now implement the `Composable` interface for the base Σ-protocol to be compiled by CDS94. This interface is defined as follows:

```rust
pub trait Composable: SigmaProtocol<Challenge: Shareable> + HVzk {}
```

This is a simple interface that enforces the requirement for the base protocol to implement the `SigmaProtocol` interface, with a `Challenge` type implementing the `Shareable` interface, and to also implement the `HVzk` interface.

[4]We omit the implementation of the methods as they are specific to the `WrappedScalar` type and are not relevant to the discussion here.

`CDS94`. With the `Shareable` and `Composable` interfaces, we can now implement the `CDS94` compiler. We require that the base Σ-protocol to be compiled implements the `Composable` interface. Let this generic type corresponding to the base Σ-protocol be denoted by `S`. With `S`, we can define the generic types associated with the `SigmaProtocol` interface for CDS94:

- `Statement`: `Statement94<S>` – a type that contains public information for the disjunctive proof: the number of clauses, the threshold, and the statement for each clause of type `S`.
- `Witness`: `Witness94<S>` – a type containing the private information for the disjunctive proof: the witnesses for each clause, and the indexes of the active clauses
- `MessageA`: `Vec<S::MessageA>` – a vector of messages corresponding to the first message in the Σ-protocol `S` for each clause.
- `Challenge`: `S::Challenge` – the challenge type for the base Σ-protocol `S`.
- `MessageZ`: `Vec<(usize, S::Challenge, S::MessageZ)>` – A vector of tuples containing the index of the clause, the challenge corresponding to the clause, and the third message `z` of that clause.
- `State`: `State94<S>` – the relevant information generated from the first round of the disjunctive protocol that should be private to the Prover. This contains the state of each instance of the base Σ-protocol (in each active clause), the simulated challenge and third messages c_i and z_i respectively for each inactive clause.

With these generic types, we implement the methods according to the construction in Protocol 2. Interested readers can compare how our implementation models the construction by referring to the source code in Appendix A.4.

4.5.7 Stacking Sigmas Compiler

Similarly to CDS94, we implement the self-stacking compiler for Stacking Sigmas where we are concerned with disjunctions of the same Σ-protocol. An important distinction between the two compilers is that the Stacking Sigmas compiler is a 1-out-of-n compiler, where n is the number of Σ-protocols in the disjunction. Goel *et al* does include a brief outline of how a d-out-of-n compiler may be constructed in Section 9 of [Goe+21], but we deem this out of scope for the project.

In our implementation, we compile Schnorr's protocol, using the compiler protocol together with the q-binding scheme (Definition 2.6.2). Overall, the methods associated with the `SigmaProtocol` library that we implement for the `SelfStacker` class follow the construction in Protocol 3 closely. More interestingly, the generic types and interfaces involved are worth discussing in detail.

`Message`. This interface is useful in the Stacking Sigmas compiler for hashing the messages provided to the partial-binding vector commitments. This is an important efficiency requirement outlined in Definition 13. With the `Message` interface, we require messages to the partial-binding vector commitments to be writable to a buffer, which can then be hashed. Additionally, this provides a convenient interface for our benchmarks to measure the size of our messages.

```rust
pub trait Message: Debug + Default + Clone {
    fn write<W: Write>(&self, writer: &mut W)
    where
        Self: Sized;

    fn size(&self) -> usize {
        let mut v: Vec<u8> = Vec::new();
        self.write(&mut v);
        v.len()
    }
}
```

Observing this code-snippet, we can see that once we are able to use the `write` method to write bits of the message to a buffer, we can trivially obtain the size of the message by writing it to a vector and then obtaining the length of the vector.

As mentioned earlier, this interface was implemented by Hall-Andersen in [Hal21], and we have extended it slightly. In particular, we further require types that implement the `Message` interface to also implement the `Default` interface. This is a useful addition because we can use this to create a default instance of the message type when we are using the q-binding scheme. Recall that this scheme requires the "0" message for the EquivCom, Equiv, and BindCom methods.

`Stackable`. This interface is similar to the `Composable` interface in the CDS94 compiler. This interface models a Σ-protocol that is stackable by the Stacking Sigmas compiler.

```
pub trait Stackable: SigmaProtocol<MessageA: Message, MessageZ: Message> + EHVzk {}
```

From its definition, we can see that the main requirements are the `SigmaProtocol` interface and the `EHVzk` interface, which corresponds correctly to the definition of a stackable Σ-protocol in Definition 16.

`StackingSigmas`. The concrete class of the Stacking Sigmas' "Self-Stacking" compiler is the `Self-Stacker<S>` class, where `S` is a class that implements the `Stackable` interface. The generic types that are associated with this class are:

- `Statement: StackedStatement<S>` – a type that contains public information for the disjunctive proof: the public parameters pp for the q-binding scheme, the height of the commitment tree, the number of clauses in total, and the vector of messages which correspond to the statement for each clause of type `S` .
- `Witness: StackedWitness<S::Witness>` – a type containing the private information for the disjunctive proof: the witnesses for the active clause (only 1), and the index of the active clause.
- `MessageA: StackedA` – the first message of the Stacking Sigmas protocol (Definition 3). This is a 2-tuple, containing the commitment key and the commitment.
- `Challenge: S::Challenge` – the challenge type for the base Σ-protocol `S`.
- `MessageZ: StackedZ<S>` – the third message of the Stacking Sigmas protocol containing: the commitment key, the third message (z) of the underlying Σ-protocol, and the auxiliary value needed for the q-binding scheme.
- `State: State94<S>` – the relevant information generated from the first round of the disjunctive protocol that should be private to the Prover. This contains the state of the underlying Σ-protocol, the first message of the underlying Σ-protocol, the initial messages given to the q-binding scheme in the first round of the Stacking Sigmas protocol, the commitment key, the equivocation key, and the auxiliary value for the q-binding scheme.

With these generic types, we implement the methods for the `SelfStacker` class according to Protocol 3. Refer to Appendix A.5 for the implementation.

Chapter 5

Evaluation

In this chapter, we evaluate our implementation with respect to testing and the results of our benchmarks.

5.1 Testing

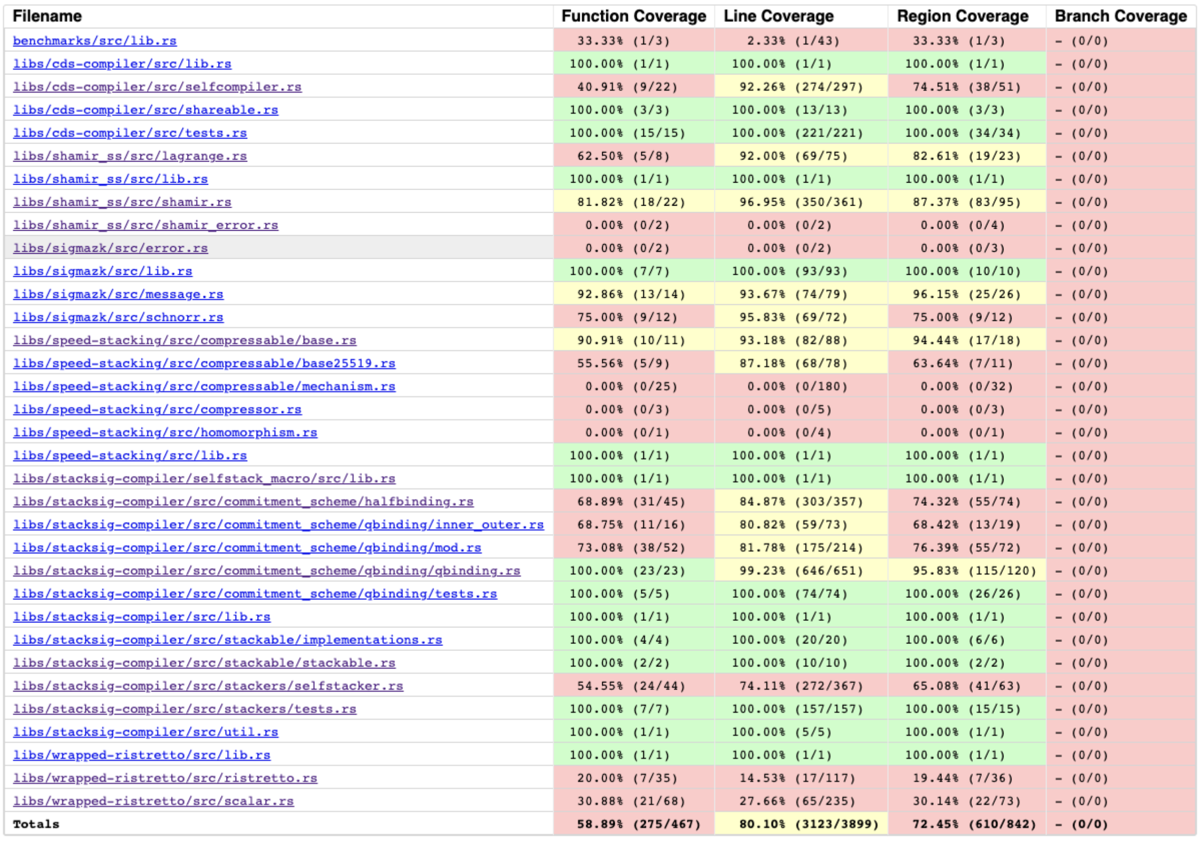
We implement tests for all major components of our system based on our approach described in Section 4.2. These tests have been scrutinised to ensure they test the expected behaviour of each component, especially those that are based on protocols from the literature. Readers who are interested can run these tests using the `cargo test` command, to verify that all tests pass. Using code coverage analysis, we also ensure that we do not miss any important test cases and that we are testing as much of our code and as many edge cases as possible. We use the following command to generate a code coverage report:

```
cargo llvm-cov --open --ignore-filename-regex libs/stacksig-compiler/src/rot256
```

The code coverage report produced by this command can be seen in Figure 5.1. From this report, we observe that we have only 58.89% function coverage (we only test 58.89% of the functions in our project), 80.10% line coverage, and 72.45% region coverage[1]. These numbers may appear to be low, but upon closer inspection, we observe the following:

1. **Functions**: the majority of the functions that are not tested are either
   - Getter and setter functions for classes (structs in Rust). These functions simply return or set a value within the class, and are not tested as they are very simple.
   - or derived traits [SR22b] in Rust. These are functions that are automatically generated according to "macros", which allow users to derive the implementation of specific interfaces for their classes automatically. These functions are not tested as they are automatically generated by the compiler.
2. **Lines**: A portion of these lines include those for the Speed Stacking compiler that we are still working on [Goe+22]. These could not be easily removed from the code coverage report, and affect the line coverage statistics. Additionally, many of these lines are automatically generated by the compiler from derived traits.

[1]Regions are blocks of code with respect to the compiler – these can be multiple lines of code with no control flow or a single line of code. These regions for Rust are determined by LLVM. [Tea23]



Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
benchmarks/src/lib.rs	33.33% (1/3)	2.33% (1/43)	33.33% (1/3)	- (0/0)
libs/cds-compiler/src/lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)	- (0/0)
libs/cds-compiler/src/selfcompiler.rs	40.91% (9/22)	92.26% (274/297)	74.51% (38/51)	- (0/0)
libs/cds-compiler/src/shareable.rs	100.00% (3/3)	100.00% (13/13)	100.00% (3/3)	- (0/0)
libs/cds-compiler/src/tests.rs	100.00% (15/15)	100.00% (221/221)	100.00% (34/34)	- (0/0)
libs/shamir_ss/src/lagrange.rs	62.50% (5/8)	92.00% (69/75)	82.61% (19/23)	- (0/0)
libs/shamir_ss/src/lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)	- (0/0)
libs/shamir_ss/src/shamir.rs	81.82% (18/22)	96.95% (350/361)	87.37% (83/95)	- (0/0)
libs/shamir_ss/src/shamir_error.rs	0.00% (0/2)	0.00% (0/2)	0.00% (0/4)	- (0/0)
libs/sigmazk/src/error.rs	0.00% (0/2)	0.00% (0/2)	0.00% (0/3)	- (0/0)
libs/sigmazk/src/lib.rs	100.00% (7/7)	100.00% (93/93)	100.00% (10/10)	- (0/0)
libs/sigmazk/src/message.rs	92.86% (13/14)	93.67% (74/79)	96.15% (25/26)	- (0/0)
libs/sigmazk/src/schnorr.rs	75.00% (9/12)	95.83% (69/72)	75.00% (9/12)	- (0/0)
libs/speed-stacking/src/compressable/base.rs	90.91% (10/11)	93.18% (82/88)	94.44% (17/18)	- (0/0)
libs/speed-stacking/src/compressable/base25519.rs	55.56% (5/9)	87.18% (68/78)	63.64% (7/11)	- (0/0)
libs/speed-stacking/src/compressable/mechanism.rs	0.00% (0/25)	0.00% (0/180)	0.00% (0/32)	- (0/0)
libs/speed-stacking/src/compressor.rs	0.00% (0/3)	0.00% (0/5)	0.00% (0/3)	- (0/0)
libs/speed-stacking/src/homomorphism.rs	0.00% (0/1)	0.00% (0/4)	0.00% (0/1)	- (0/0)
libs/speed-stacking/src/lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)	- (0/0)
libs/stacksig-compiler/selfstack_macro/src/lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)	- (0/0)
libs/stacksig-compiler/src/commitment_scheme/halfbinding.rs	68.89% (31/45)	84.87% (303/357)	74.32% (55/74)	- (0/0)
libs/stacksig-compiler/src/commitment_scheme/qbinding/inner_outer.rs	68.75% (11/16)	80.82% (59/73)	68.42% (13/19)	- (0/0)
libs/stacksig-compiler/src/commitment_scheme/qbinding/mod.rs	73.08% (38/52)	81.78% (175/214)	76.39% (55/72)	- (0/0)
libs/stacksig-compiler/src/commitment_scheme/qbinding/qbinding.rs	100.00% (23/23)	99.23% (646/651)	95.83% (115/120)	- (0/0)
libs/stacksig-compiler/src/commitment_scheme/qbinding/tests.rs	100.00% (5/5)	100.00% (74/74)	100.00% (26/26)	- (0/0)
libs/stacksig-compiler/src/lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)	- (0/0)
libs/stacksig-compiler/src/stackable/implementations.rs	100.00% (4/4)	100.00% (20/20)	100.00% (6/6)	- (0/0)
libs/stacksig-compiler/src/stackable/stackable.rs	100.00% (2/2)	100.00% (10/10)	100.00% (2/2)	- (0/0)
libs/stacksig-compiler/src/stackers/selfstacker.rs	54.55% (24/44)	74.11% (272/367)	65.08% (41/63)	- (0/0)
libs/stacksig-compiler/src/stackers/tests.rs	100.00% (7/7)	100.00% (157/157)	100.00% (15/15)	- (0/0)
libs/stacksig-compiler/src/util.rs	100.00% (1/1)	100.00% (5/5)	100.00% (1/1)	- (0/0)
libs/wrapped-ristretto/src/lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)	- (0/0)
libs/wrapped-ristretto/src/ristretto.rs	20.00% (7/35)	14.53% (17/117)	19.44% (7/36)	- (0/0)
libs/wrapped-ristretto/src/scalar.rs	30.88% (21/68)	27.66% (65/235)	30.14% (22/73)	- (0/0)
Totals	**58.89% (275/467)**	**80.10% (3123/3899)**	**72.45% (610/842)**	**- (0/0)**

Figure 5.1: Code coverage of our project

3. **Regions**: While most regions are affected by the same reasons as functions and lines, it should be noted that there are some regions that should be tested more thoroughly but are not. These regions are related to error producing branches of code, which should be tested to ensure that our implementation handles errors correctly. The bulk of these regions are located in our implementation of Shamir's secret sharing scheme. However, due to the low priority, and time constraints of this project, we have not been able to test these cases thoroughly.

Aside from these areas, all the main requirements of our compilers and respective software components are tested thoroughly. While it is ideal to achieve higher than 80% code coverage in all categories, we believe that code coverage useful in identifying potential areas of improvement for testing, but should not be used as the sole metric for evaluating the quality of our tests. In fact, the code coverage report was useful in highlighting that we previously did not implement a proper negative test for the CDS94 compiler as the block of code that returns false in the verification algorithm was not covered by any tests. This was subsequently fixed, improving the code coverage report marginally, but covering and important case for our compiler.

In summary, we assess that our quality and thoroughness of testing is sufficient for the scope of this project. However, we acknowledge that there is still room for improvement in our testing, especially in the areas of error handling.

5.2 Benchmark Results

In this section, we present the results of our benchmarks and discuss their implications. We have four benchmarks: two for the CDS94 compiler [CDS94], one for the Stacking Sigmas compiler [Goe+21], and one from Hall-Andersen's implementation [Hal21]. One of the two benchmarks for CDS94 compiler compares the growth of the key metrics against the number of active clauses (instead of the number of clauses). We will first present and discuss the results regarding the growth in communication size across all benchmarks. After which, we will discuss the results for the growth in the computation time. In table 5.1 and 5.2, we present the overall results of our benchmarks.

Table 5.1: Communication Size Results (in bytes)

Clauses	**CDS94**	**Stacking Sigmas**
2	224	-
4	416	256
8	800	320
16	1568	384
32	3104	448
64	6176	512
128	12320	576
256	24608	640
512	49184	704
1024	98336	768
2048	196640	832
4096	393248	896
8192	786368	960

Table 5.2: Computation Time Results (in milliseconds)

Clauses	**StackSig Prover**	**CDS Prover**	**StackSig Verifier**	**CDS Verifier**	**Rot256**
2	0.3	0.081124	0.3	0.10955	3.4760
4	0.64893	0.19991	0.64233	0.22346	7
8	1.8222	0.45870	1.3788	0.47586	10.9
16	4.2626	0.98115	2.8526	0.97444	15.021
32	9.3257	2.1472	5.9700	2.0985	20.069
64	19.616	5.0377	11.705	4.9214	37
128	40.584	13.105	24.474	12.696	37.131
256	82.243	37.381	47.087	36.691	54.213
512	165.70	120.33	93.798	118.92	85
1024	334.63	422.50	190.63	419.73	145.33
2048	671.82	1572.9	375.45	1583.5	257.25
4096	1359.1	6110.2	742.17	6081.9	482.62
8192	2685.4	24141	1477.9	23884	932.34

Running benchmarks. To run our benchmarks on your own machine, you can use the `cargo bench` command. To run the benchmarks for a specific compiler, you can use the `--bench` flag. Below we provide an example of how to run each benchmark available:

```
# run benchmark for CDS94 (as clauses increase)
cargo bench --bench cds_benchmark
# run benchmark for CDS94 (as active clauses increase)
cargo bench --bench cds_benchmark2
# run benchmark for Stacking Sigmas (as clauses increase)
cargo bench --bench stacksig_benchmark
# run benchmark for Hall-Andersen's implementation (as clauses increase)
cargo bench --bench rot256_benchmark
```

5.2.1 Communication Size

We present two figures, Figure 5.2 and Figure 5.3, for the CDS94 compiler and Stacking Sigmas compiler respectively. These figures show the growth in communication size for each compiler as the number of clauses increases. Note that the x-axis of these figures is logarithmic. Therefore, a linear and logarthmic growth in communication size is represented by a quadratic and linear growth in the logarthmic scale respectively.

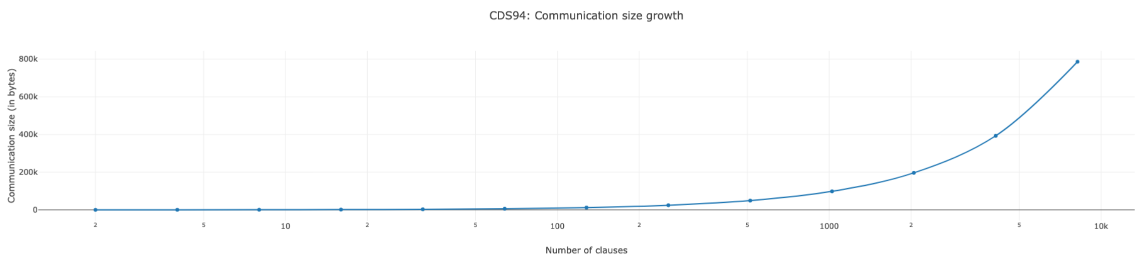


Figure 5.2: The growth in communication size for the CDS94 compiler.

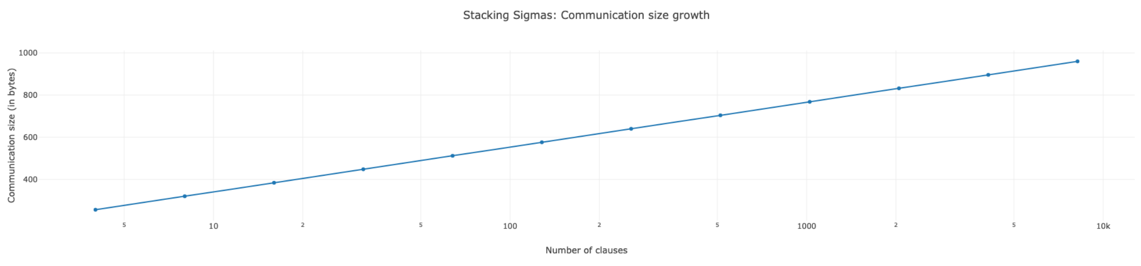


Figure 5.3: The growth in communication size for the Stacking Sigmas compiler.

In both cases, we point out that the communciation size growth is consistent with the theoretical proofs of the respective compilers. This further validates our implementation of the compilers. The expected growth in communication size for the CDS94 compiler is linear, which appears as a quadratic growth with a logarithmic scale. Meanwhile, the equivalent for the Stacking Sigmas compiler is a logarthmic growth, which appears as a linear growth in the logarthmic scale.

With an increasing number of active clauses, but a constant number of clauses, the communication size for the CDS94 compiler is not expected to change. This is because the number of active clauses does not affect the number of elements in the vector of messages sent by the prover. This is supported by the results in Figure 5.4. This figure shows that the communication size for 512 clauses is constant at 53.28KB, regardless of the number of active clauses.

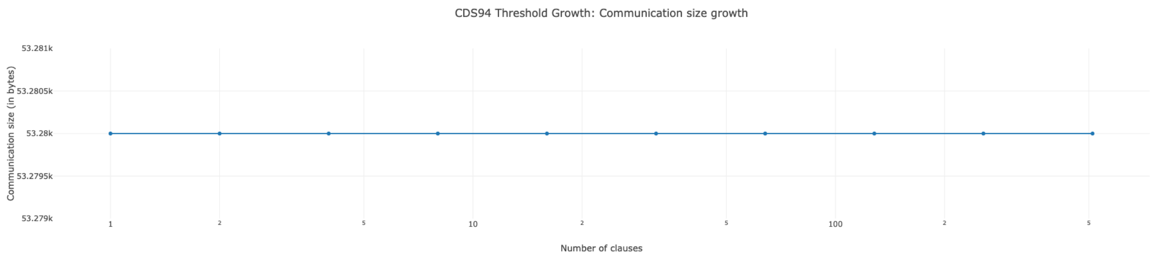


Figure 5.4: The growth in communication size for the CDS94 compiler as the number of active clauses increases (total clauses = 512).

We do not provide a similar figure for Hall-Andersen's implementation as the results are identical to Figure 5.3. In conclusion, the observed growth in communication size is consistent with the theoretical proofs of the respective compilers, further supporting the correctness of our implementations, and also validating these proofs.

5.2.2 Computation Time

Firstly, we present the comparison between prover and verifier running time for each compiler.

CDS94 Prover vs Verifier. In Figure 5.5, we compare the prover and verifier running time in our implementation of [CDS94]. From the figure, we observe that the both running times are quadratic as the number of clauses increase. This is mainly because polynomial interpolation using Lagrange is the bottleneck with a quadratic running time. This directly affects both the prover and verifier running times, as both the third round protocol and the verification algorithm uses this algorithm.

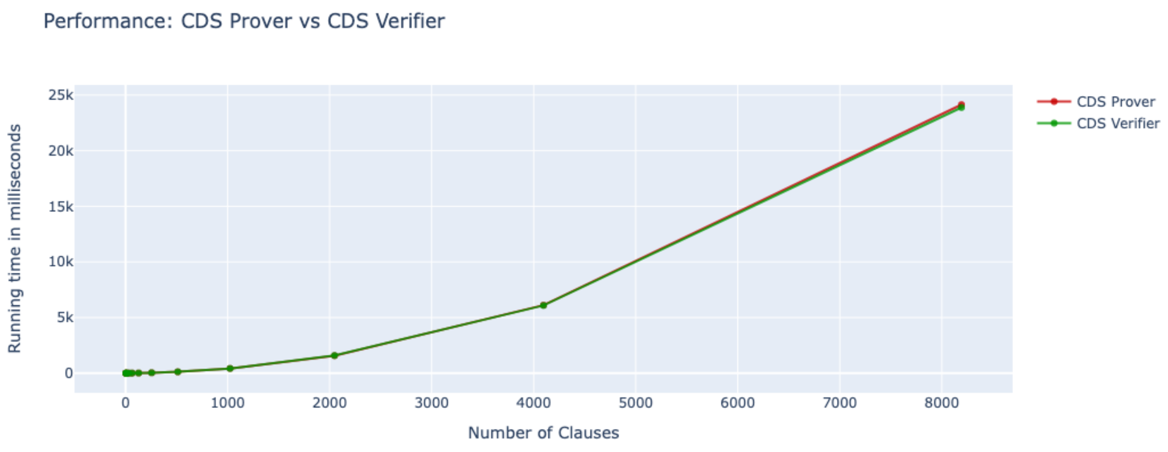


Figure 5.5: A comparison of the prover and verifier running time of CDS94.

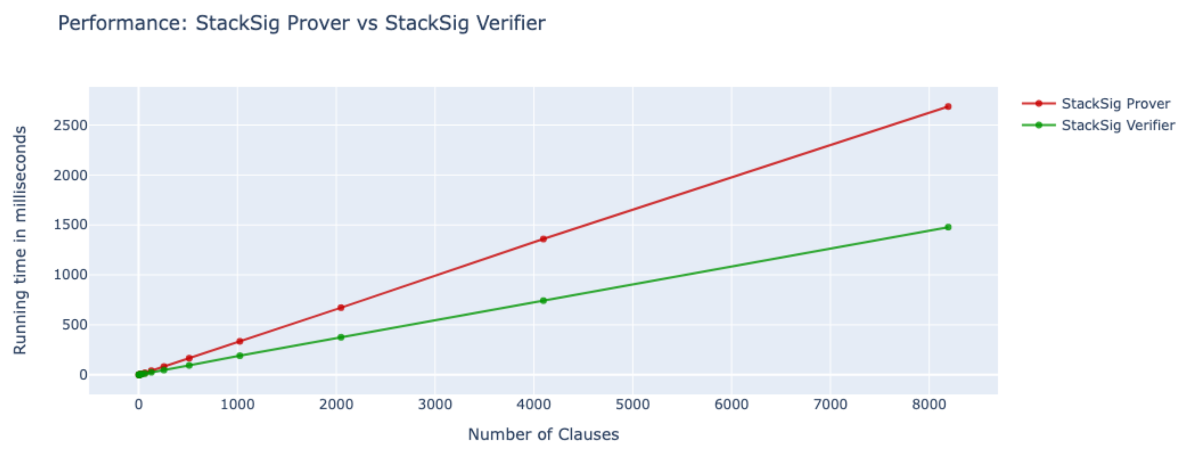


Figure 5.6: A comparison of the prover and verifier running time of Stacking Sigmas.

Stacking Sigmas Prover vs Verifier. In Figure 5.6, we compare the prover and verifier running time for the Stacking Sigmas compiler. This shows a linear growth in compuation time for both the prover and verifier, which is expected. The prover running time has a larger constant factor than the verifier running time. This is likely the case because the prover algorithms call the Equiv and EquivCom methods from the q-binding scheme, which are more computationally expensive than the Bind method used in the verifier algorithm.

CDS94 vs Stacking Sigmas. Next, we compare the prover algorithms of CDS94 and Stacking Sigmas. In Figure 5.7, we see that up until 512 clauses, the CDS94 prover is more performant. However, due to the quadratic running time of the CDS94 prover, the Stacking Sigmas prover starts to outperform CDS94 after 512 clauses. Similarly in Figure 5.8, where we compare the verifier algorithms across the two compilers, CDS94 is more performant when there are 256 or fewer clauses. After which, Stacking Sigmas overtakes and takes less time to run. Note that both the x-axis and the y-axis are on the logarithmic scale.

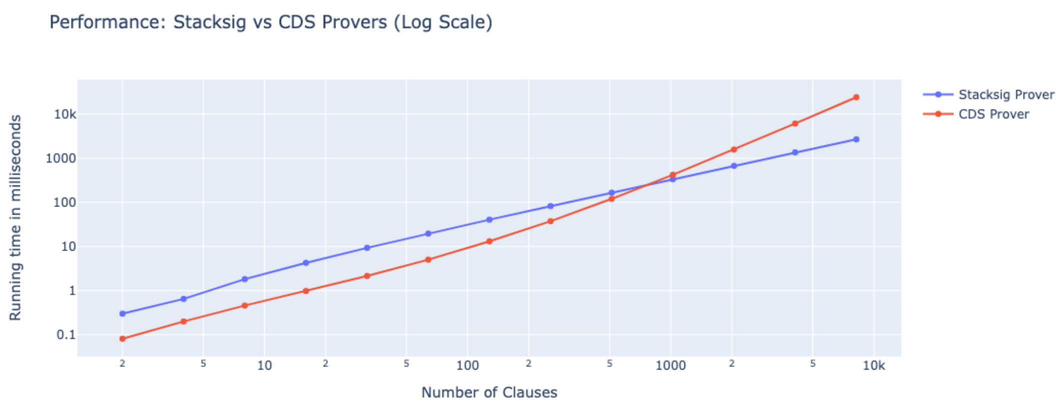


Figure 5.7: Comparison of CDS94 Prover and Stacking Sigmas Prover Algorithms.

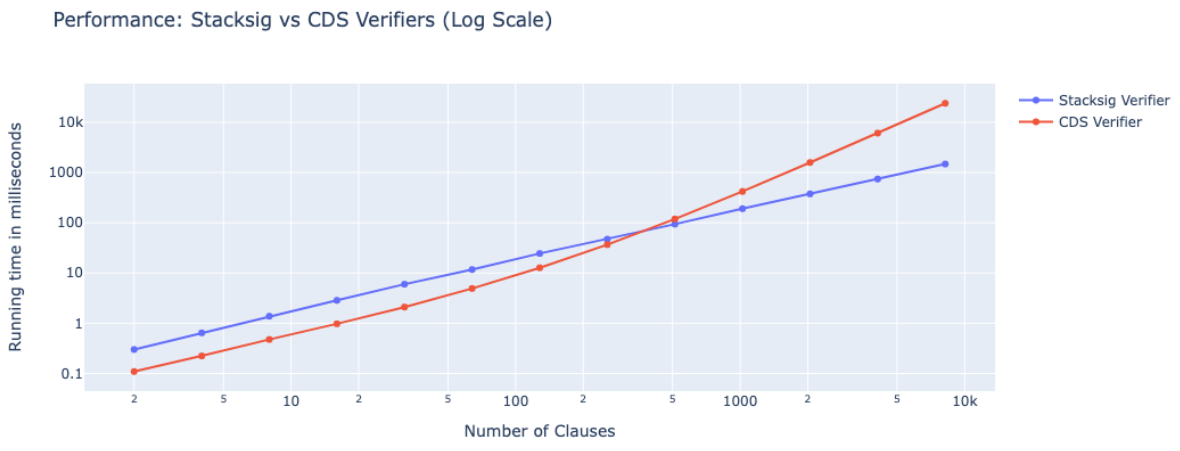


Figure 5.8: Comparison of the Verifier algorithms of CDS94 and Stacking Sigmas.

These results indicate that if computation speed is the priority, it may be more appropriate to use Stacking Sigmas when the number of clauses is large (i.e. 512 or more), compared to an implementation of CDS94 that uses Lagrange's interpolation [BW] with Shamir's secret sharing [Sha79]. That said, this only applies to 1-out-of-n disjunctive zero-knowledge, as k-out-of-n disjunctions are not supported by this implementation of Stacking Sigmas.

Total Running Time. Extending this comparison, we now compare the total running time (prover and verifier) of the three implementations. In Figure 5.9, we see that the total running time of the CDS94 compiler, the Stacking Sigmas compiler, and Mathias Hall-Andersen's implementation. We observe that the total running time of the CDS94 compiler and both impmlementations of Stacking Sigmas is similar to what we observer in Figures 5.7 and 5.8. Comparing the two Stacking Sigmas implementations, we notice that the running time of Hall-Andersen's implementation is better starting from 128 clauses. As mentioned in Section 4.5.4, this difference in performance is likely due to our decision to trade off performance for usability.

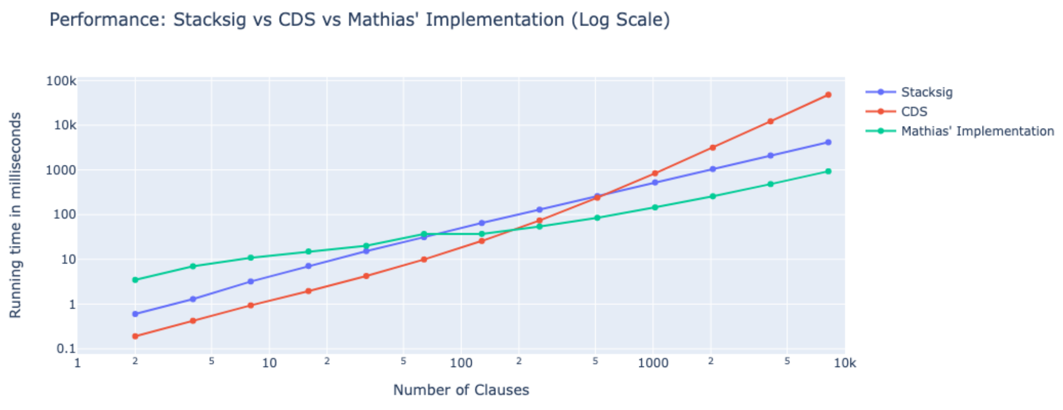


Figure 5.9: Comparison of the total running time of the 3 compilers: [CDS94], Stacking Sigmas [Goe+21], and Mathias Hall-Andersen's implementation [Hal21] of Stacking Sigmas.

Our current implementation of the q-binding scheme leads to many calls to the `clone` function which duplicates data. The reason for this is mainly due to our lack of experience with Rust, and can be improved in future work. Furthermore, in Hall-Andersen's implementation, a protocol for every power of two number of clauses requires the instantiation of a new type. This means that a disjunction of 2 clauses, 4 clauses, 8 clauses, and so on, must use a different type. This is not ideal as it requires the developer to write a lot of boilerplate code, and also results in code duplication. For this reason, we believe that the trade off is sound as it allows for a more usable interface for developers and can be further improved. Moreover, this performance difference does not affect the correctness of the implemention or the goal of our benchmarks.

CDS94 with increase in Active Clauses. Finally, we plot the running time of the CDS94 prover and verifier as the number of **active clauses** increase (every power of 2 from 1 to 512 active clauses). For this benchmark, we fix the total number of clauses to 512.

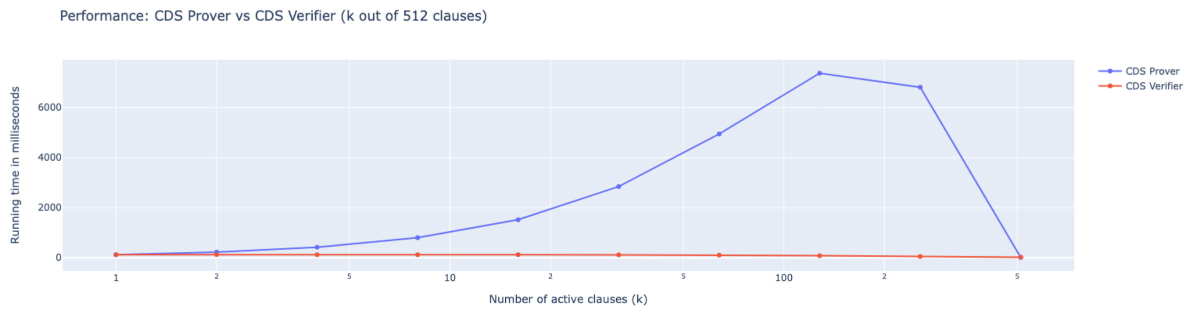


Figure 5.10: Comparison of prover and verifier algorithms as *active clauses* increase. Note that the x-axis is on the logarithmic scale.

In Figure 5.10, we see that the prover running time peaks at 128 clauses, and drops abruptly at 512 clauses. Meanwhile, the verifier running time is constant as expected because the number of active clauses do not affect the verifier algorithm – only the total number of clauses do. The behaviour of the prover's running time can be directly attributed to our Complete algorithm (Definition 12) which we show in Listing 5.1.

When completing the shares for the active clauses, we are required to interpolate at the point corresponding to the index of the active clause (this is the `x` variable in the code listing). This interpolation uses the secret and the maximally unqualified set of shares to obtain the polynomial. As mentioned earlier, Lagrange's polynomial interpolation is an $O(n^2)$ algorithm, where n is the number of points used to interpolate. Furthermore, the interpolation algorithm is called k times where k is the number of active clauses. This means that $n = \texttt{total clauses} - k + 1$.

Listing 5.1: Share Completion Algorithm

```
let remaining_shares = remaining_xs
    .iter()
    .map(|x| {
        let y = poly.interpolate(*x);
        Share { x: *x, y }
    })
    .collect();
```

Hence, the total running time of the Complete algorithm is $O(k \cdot n^2)$ (or $g(k) = k \cdot (513 - k)^2$ in our case). By plotting function g as a graph, we observe a direct correlation with what we observe in Figure 5.10. This is easy to see if we show this function as a table:

Table 5.3: Active clauses & $g(k)$

Active Clauses	**Threshold**	g
1	512	262144
2	511	522242
4	509	1036324
8	505	2040200
16	497	3952144
32	481	7403552
64	449	12902464
128	385	18972800
256	257	16908544
512	1	512

These results shed light on the effects of using Lagrange's polynomial interpolation algorithm with Shamir's secret sharing scheme. The CDS94 compiler [CDS94] with Shamir's Secret Sharing and Lagrange's interpolation is most effective when the number of active clauses matches the total number of clauses. However, this is arguably an uninteresting case as it is no different to a *conjunction* of clauses requiring the verifier to check every clause. In practice, we expect the number of active clauses to be in a smaller range. This indicates that Lagrange's interpolation algorithm is not the most ideal algorithm for Shamir's secret sharing scheme, as the running time increases cubically with the number of active clauses until it peaks at two powers of 2 below the total number of clauses.

5.2.3 Summary of Results

To summarise, we have firstly verified the proofs regarding the communication size of each compiler as the results here are consistent with the results in the literature [CDS94; Goe+21].

Next, we have revealed important insights into the performance of the CDS94 and Stacking Sigmas compiler. It is clear that our implementation of the CDS94 compiler is significantly faster than the Stacking Sigmas compiler when the number of clauses are small. That said, the Stacking Sigmas compiler has a communication size that grows logarithmically with the number of clauses, while the CDS94 compiler's communication size grows linearly. Hence, the trade off has to be made between the need for a smaller communication size and the need for a faster computation time.

In general, we believe that the Stacking Sigmas compiler is more suitable in any use case where a 1-out-of-n disjunctive proof is suitable. This is primarily because of the large savings in communication size (kilobytes of data) and the small increase in computation time (a few milliseconds) of using it when the number of clauses are small. When the number of clauses increase, CDS94 performs worse and the performance degradation is far more significant because of the large number of clauses. For use cases where a k-out-of-n threshold proof is required, the CDS94 compiler has to be used. Further improvements could be made to our implementation of CDS94 to further reduce computation time and improve performance as the number of active clauses or total number of claues increase. This will be interesting to explore in future work.

Chapter 6

Conclusions

Overall, we accomplished our goals of firstly implementing the CDS94 [CDS94] and Stacking Sigmas [Goe+21] compiler, and secondly benchmarking their performance. Through our work, we have shown that Stacking Sigmas outperforms CDS94 by providing significant savings in the communication size of the proof, especially as the number of clauses increase. Furthermore, while CDS94 is faster than Stacking Sigmas when the number of clauses are small, we show that the difference in speed is negligible because the number of clauses are small. When the number of clauses increase, the difference in speed is more significant and CDS94 performs worse than Stacking Sigmas in this case. Moreover, we also reveal that the bottleneck of the CDS94 compiler is the secret sharing scheme, and highlight how the choice of this scheme and its implementation has a significant impact on the performance of the compiler. This is all while we uphold our goal of keeping the implementation of the compiler easily usable and extensible, in order to lay a good foundation for future work in this area.

Additionally, there were many useful lessons that we learnt throughout the project. Firstly, we learnt the importance of proper project management and planning for a successful project. The risk management plan that we created at the start of the project helped us to identify potential risks and to mitigate them. This allowed us to focus on the project and to avoid any distractions that could have hindered our progress significantly. Secondly, we realised the importance and convenience of good testing practices. Because of our testing practices, we were able to identify and fix bugs early, accelerating our progress. Lastly, we were able to learn more about the inner workings of zero-knowledge proofs and other cryptographic concepts. Work in this field is often theoretical, and it is interesting to see how these concepts can be implemented in practice.

6.1 Future work

There are many directions that we can take this project in the future.

Speed Stacking. Firstly, our work on the Speed Stacking compiler [Goe+22] is still ongoing. We have implemented the base Σ-protocol of our choice (Compressed Σ-protocols or Folding Arguments) and are currently working on the compiler. Possible extensions to this, would be to use the same compiler for interactive oracle proofs [BCS16].

Stacking Sigmas. Next, our implementation of Stacking Sigmas is the Self-Stacking implementation, where we use the same Σ-protocol for each clause. We can also implement the Cross-Stacking compiler, which is compatible with different Σ-protocols for each clause. This would allow us to compare the performance of the compiler when using different Σ-protocols, and possibly test how the choice of Σ-protocols affects the performance of the compiler. Another extension is to explore Goel *et al.*'s idea for a k-out-of-l compiler (Section 9 of [Goe+21]). This idea relies on the parallel execution of k 1-out-of-l proofs together with a family of hash functions to ensure that the verifier can check that each execution is for a unique clause.

CDS94. Similar to Stacking Sigmas, we can also explore the idea of compiling different Σ-protocols with CDS94. However, a more closely related idea is to explore the idea of using a different secret sharing scheme or a more efficient implementation of Shamir's secret sharing scheme. This would allow us to compare the performance of the compiler when using different secret sharing schemes, and test how the choice of secret sharing scheme affects the performance of the compiler.

All these suggestions, rely on the benchmarking model that we have implemented. A useful extension to this would be to implement a benchmarking framework that provides a convenient testing interface for different compilers. This framework may have features to produce mock data automatically and simply take instances of compilers and base protocols as input, and subsequently run the benchmarking process. It would also be useful to automatically generate CSV files from the raw data collected from these benchmarks, as they are currently manually extracted. This will help to enhance the benchmarking process and make it far more convenient to use.

6.2 Acknowledgements

We would like to thank our supervisor, Nicholas Spooner, for his invaluable guidance and support throughout the project.

Bibliography

[ACF21] Thomas Attema, Ronald Cramer and Serge Fehr. 'Compressing Proofs of k-Out-Of-n Partial Knowledge'. In: *Advances in Cryptology – CRYPTO 2021*. Ed. by Tal Malkin and Chris Peikert. Cham: Springer International Publishing, 2021, pp. 65–91. ISBN: 978-3-030-84259-8. DOI: `10.1007/978-3-030-84259-8_3`.

[Ard+20] Luca Ardito et al. 'rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes'. In: *SoftwareX* 12 (2020), p. 100635. ISSN: 2352-7110. DOI: `https://doi.org/10.1016/j.softx.2020.100635`. URL: `https://www.sciencedirect.com/science/article/pii/S2352711020303484`.

[Aut22] Rust Fuzzing Authority. *afl.rs*. Online; accessed November 27, 2022. Feb. 2022. URL: `https://github.com/rust-fuzz/afl.rs`.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa and Nicholas Spooner. *Interactive Oracle Proofs*. Cryptology ePrint Archive, Paper 2016/116. `https://eprint.iacr.org/2016/116`. 2016. URL: `https://eprint.iacr.org/2016/116`.

[Ber+] Daniel J Bernstein et al. 'ChaCha, a variant of Salsa20'. In.

[Bla79] G. R. Blakley. 'Safeguarding cryptographic keys'. In: *1979 International Workshop on Managing Requirements Knowledge, MARK 1979, New York, NY, USA, June 4-7, 1979*. IEEE, 1979, pp. 313–318. DOI: `10.1109/MARK.1979.8817296`. URL: `https://doi.org/10.1109/MARK.1979.8817296`.

[BW] Branden and Eric W. Archer Weisstein. *Lagrange Interpolating Polynomial*. Online; accessed April 17, 2023. URL: `https://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html`.

[CDS94] Ronald Cramer, Ivan Damgård and Berry Schoenmakers. 'Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols'. In: *CRYPTO*. 1994.

[DH76] Whitfield Diffie and Martin E. Hellman. 'New Directions in Cryptography'. In: *Democratizing Cryptography* (1976).

[Edw07] Harold M. Edwards. 'A normal form for elliptic curves'. In: *Bulletin of the American Mathematical Society* 44 (3 July 2007), pp. 393–422. ISSN: 0273-0979. DOI: `10.1090/S0273-0979-07-01153-6`. URL: `https://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/`.

[End23] Taiki Endo. *taiki-e/cargo-llvm-cov*. Version 0.5.17. Apr. 2023. URL: `https://crates.io/crates/cargo-llvm-cov`.

[FS90] U. Feige and A. Shamir. 'Witness Indistinguishable and Witness Hiding Protocols'. In: *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: Association for Computing Machinery, 1990, pp. 416–426. ISBN: 0897913612. DOI: `10.1145/100216.100272`. URL: `https://doi.org/10.1145/100216.100272`.

[GMR85] S Goldwasser, S Micali and C Rackoff. 'The Knowledge Complexity of Interactive Proof-Systems'. In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC '85. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 291–304. ISBN: 0897911512. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178.

[GMW86] Oded Goldreich, Silvio Micali and Avi Wigderson. 'Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design (Extended Abstract)'. In: *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. IEEE Computer Society, 1986, pp. 174–187. DOI: 10.1109/SFCS.1986.47. URL: https://doi.org/10.1109/SFCS.1986.47.

[GNU07] GNU. *The GNU General Public License v3.0 - GNU Project - Free Software Foundation*. Online; accessed October 25, 2022. 2007. URL: https://www.gnu.org/licenses/gpl-3.0.html.

[Goe+21] Aarushi Goel et al. *Stacking Sigmas: A Framework to Compose Σ-Protocols for Disjunctions*. Cryptology ePrint Archive, Paper 2021/422. https://eprint.iacr.org/2021/422. 2021. URL: https://eprint.iacr.org/2021/422.

[Goe+22] Aarushi Goel et al. *Speed-Stacking: Fast Sublinear Zero-Knowledge Proofs for Disjunctions*. Cryptology ePrint Archive, Paper 2022/1419. https://eprint.iacr.org/2022/1419. 2022. URL: https://eprint.iacr.org/2022/1419.

[Goo+21] Mike Goodwin et al. *OWASP Threat Dragon*. Online; accessed November 27, 2022. 2021. URL: https://owasp.org/www-project-threat-dragon/.

[HA22] Brook Heisler and Jorge Aparicio. *bheisler/criterion.rs: Criterion*. Version 0.4.0. Sept. 2022. URL: https://crates.io/crates/criterion.

[Hal21] Mathias Hall-Andersen. *StackSig; Ring Signatures from Stacking*. Online; accessed October 25, 2022. 2021. URL: https://github.com/rot256/research-stacksig.

[HCC21] Digorry Hardy, Alex Crichton and Github Contributors. *rust-random/rand: rand$_c$hacha*. Version 0.3.1. June 2021. URL: https://crates.io/crates/rand_chacha.

[HK20] David Heath and Vladimir Kolesnikov. 'Stacked Garbling for Disjunctive Zero-Knowledge Proofs'. In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 136.

[Inc15] Plotly Technologies Inc. *Collaborative data science*. 2015. URL: https://plot.ly.

[LH23] Michael Lodder and Dave Huseby. *mikelodder7/vsss-rs*. Version 2.7.1. Mar. 2023. URL: https://crates.io/crates/vsss-rs.

[LV22] Isis Agora Lovecruft and Henry de Valence. *dalek-cryptography/curve25519-dalek*. Version 4.0.0-rc.2. Dec. 2022. URL: https://crates.io/crates/curve25519-dalek.

[Mar00] Robert C Martin. 'Design principles and design patterns'. In: *Object Mentor* 1.34 (2000), p. 597.

[McK10] Wes McKinney. 'Data Structures for Statistical Computing in Python'. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[Org] The Rust Programming Language Organisation. *cargo bench - The Cargo Book*. Online; accessed April 14, 2023. URL: https://doc.rust-lang.org/cargo/commands/cargo-bench.html.

[Pig15] Xavier Pigeon. *TDD Global Lifecycle*. Online; accessed November 27, 2022. 2015. URL: https://commons.wikimedia.org/wiki/File:TDD_Global_Lifecycle.png.

[Rem17] V.N. Remeslennikov. *Horner Scheme*. Online; accessed April 18, 2023. Oct. 2017. URL: http://encyclopediaofmath.org/index.php?title=Horner_scheme&oldid=41987.

[Sas+14] E. Ben Sasson et al. 'Zerocash: Decentralized Anonymous Payments from Bitcoin'. In: *2014 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2014, pp. 459–474. DOI: 10.1109/SP.2014.36. URL: https://doi.ieeecomputersociety.org/10.1109/SP.2014.36.

[Sch89] Claus-Peter Schnorr. 'Efficient Identification and Signatures for Smart Cards'. In: *CRYPTO*. 1989. URL: https://link.springer.com/chapter/10.1007/0-387-34805-0_22.

[Sha79] Adi Shamir. 'How to Share a Secret'. In: *Commun. ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176. URL: http://doi.acm.org/10.1145/359168.359176.

[SM20] Elie Saad and Rick Mitchell. *Web Security Testing Guide v4.2*. Online; accesed November 27, 2022. 2020. URL: https://owasp.org/www-project-web-security-testing-guide/v42/.

[SR22a] Carol Nichols Steve Klabnik and the Rust Community. *The Rust Programming Language*. Online; accessed April 15, 2023. Dec. 2022. URL: https://doc.rust-lang.org/stable/book.

[SR22b] Carol Nichols Steve Klabnik and the Rust Community. *The Rust Programming Language: Appendix C: Derivable Traits*. Online; accessed April 15, 2023. Dec. 2022. URL: https://doc.rust-lang.org/book/appendix-03-derivable-traits.html?highlight=derive#appendix-c-derivable-traits.

[SS20] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. Online; accessed April 17, 2023. Nov. 2020. URL: https://www.scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf.

[str+22] str4d et al. *zkcrypto/group*. Version 0.13.0. Dec. 2022. URL: https://crates.io/crates/group.

[SVY21] Scrum.org, Daniel Vacanti and Yuval Yeret. *The Kanban Guide for Scrum Teams*. Online; accessed April 17, 2023. Jan. 2021. URL: https://www.scrum.org/resources/kanban-guide-scrum-teams.

[swi19] swisspost-evoting. *E-Voting System 2019*. Online; accessed April 15, 2023. 2019. URL: https://gitlab.com/swisspost-evoting/e-voting-system-2019.

[Tea23] The Clang Team. *Source-based Code Coverage*. Online; accessed April 18, 2023. 2023. URL: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#interpreting-reports.

[tea23] The pandas development team. *pandas-dev/pandas: Pandas*. Version 1.5.3. Jan. 2023. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[Tha22] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf. June 2022. URL: https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf.

[Val+] Henry de Valence et al. *Ristretto - The Ristretto Group*. Online; accessed November 8, 2022. URL: https://ristretto.group/ristretto.html.

[Zal17] Michal Zalewski. *American fuzzy lop*. Online; accessed November 27, 2022. 2017. URL: https://lcamtuf.coredump.cx/afl/.

Appendix A

Code Snippets

Input types are presented within the brackets of each method and output types are presented after the -> symbol. For example, in section A.1, the `first` method has the input:

`statement: &Self::Statement, witness: &Self::Witness, prover_rng: &mut R`

and the output:

`(Self::State, Self::MessageA)`

A.1 `SigmaProtocol Interface`

```rust
fn first<R: CryptoRngCore>(
    statement: &Self::Statement,
    witness: &Self::Witness,
    prover_rng: &mut R,
) -> (Self::State, Self::MessageA)
where
    Self: Sized;

fn second<R: CryptoRngCore>(verifier_rng: &mut R) -> Self::Challenge
where
    Self: Sized;

fn third<R: CryptoRngCore>(
    statement: &Self::Statement,
    state: Self::State,
    witness: &Self::Witness,
    challenge: &Self::Challenge,
    prover_rng: &mut R,
) -> Self::MessageZ
where
    Self: Sized;

fn verify(
    statement: &Self::Statement,
    a: &Self::MessageA,
    c: &Self::Challenge,
    z: &Self::MessageZ,
) -> bool
where
    Self: Sized;
```

A.2 `HVzk` & `EHVzk` Interface

```rust
/// Honest-Verifier Zero Knowledge
pub trait HVzk: SigmaProtocol {
    fn simulate(
        statement: &Self::Statement,
    ) -> (Self::MessageA, Self::Challenge, Self::MessageZ);
}

/// Extended Honest-Verifier Zero Knowledge
pub trait EHVzk: SigmaProtocol {
    fn simulate(
        statement: &Self::Statement,
        challenge: &Self::Challenge,
        z: &Self::MessageZ,
    ) -> Self::MessageA;
}
```

A.3 Schnorr's Protocol

```rust
/// Sigma protocol implementation for Schnorr
impl SigmaProtocol for Schnorr {
    type Statement = Schnorr;
    type Witness = Scalar;

    type State = Scalar;
    type MessageA = CompressedRistretto;
    type Challenge = Scalar;
    type MessageZ = Scalar;

    /// First round of Schnorr's protocol
    fn first<R: CryptoRngCore>(
        _statement: &Schnorr,
        _witness: &Scalar,
        prover_rng: &mut R,
    ) -> (Self::State, Self::MessageA) {
        // Get trapdoor
        let state = Scalar::random(prover_rng);
        // Get group element (point on the curve)
        let message = &state * RISTRETTO_BASEPOINT_TABLE;

        (state, message.compress())
    }

    /// Second round of Schnorr's protocol. Random challenge.
    fn second<R: CryptoRngCore>(
        verifier_rng: &mut R,
    ) -> Self::Challenge {
        Scalar::random(verifier_rng)
    }

    /// Third round of Schnorr's protocol
    fn third<R: CryptoRngCore>(
        _statement: &Schnorr,
        state: Scalar,
        witness: &Scalar,
        challenge: &Scalar,
        _prover_rng: &mut R,
    ) -> Self::MessageZ {
        // z = r + cx
        challenge * witness + state
    }

    /// Verification of transcript algorithm
    fn verify(
        statement: &Schnorr,
        a: &CompressedRistretto,
        c: &Scalar,
        z: &Scalar,
    ) -> bool {
        // G * z  =?= a + c * H => G * z - c * H =?= a
        RISTRETTO_BASEPOINT_TABLE * z
            - c * statement.pub_key
            == a.decompress()
                .unwrap()
    }
}
```

A.4 CDS94 Compiler: `SelfCompiler`

```rust
/// Implementation of the Sigma Protocol trait for the CDS94 compiler protocol.
///
/// This implementation requires a SigmaProtocol (denoted by
/// S) that implements the Composable trait.
impl<S: Composable> SigmaProtocol for SelfCompiler94<S> {
    type Statement = Statement94<S>;
    type Witness = Witness94<S>;
    type State = State94<S>;
    type MessageA = Vec<S::MessageA>;
    type Challenge = S::Challenge;
    type MessageZ = Vec<(usize, S::Challenge, S::MessageZ)>;

    /// The algorithm for the first round of the protocol.
    fn first<R: CryptoRngCore + Clone>(
        statement: &Self::Statement,
        witness: &Self::Witness,
        prover_rng: &mut R,
    ) -> (Self::State, Self::MessageA)
    where
        Self: Sized,
    {
        // Clone the prover_rng as we need it to be the same value for third round
        let mut prover_rng = prover_rng.clone();
        // Deconstruct variables
        let (clauses, _cds_threshold, statements) =
            statement.pattern_match();
        let (witnesses, active_clauses) =
            witness.pattern_match();
        // Intialize vectors
        let mut inner_states: Vec<Option<S::State>> =
            Vec::with_capacity(*clauses);
        let mut challenges: Vec<Option<S::Challenge>> =
            Vec::with_capacity(*clauses);
        let mut zs: Vec<Option<S::MessageZ>> =
            Vec::with_capacity(*clauses);
        let mut message_as: Vec<S::MessageA> =
            Vec::with_capacity(*clauses);

        for i in 0..*clauses {
            // If the clause is active, run the first round of the underlying sigma protocol
            if active_clauses.contains(&i) {
                let (state, message_a) = S::first(
                    &statements[i],
                    &witnesses[i],
                    &mut prover_rng,
                );

                // Push relevant values to vectors
                message_as.push(message_a);
                inner_states.push(Some(state));
                challenges.push(None);
                zs.push(None);
            } else {
                // If the clause is not active, simulate the underyling sigma protocol
                let (message_a, c, z) =
                    S::simulate(&statements[i]);

                // Push relevant values to vectors
                message_as.push(message_a);
                inner_states.push(None);
                challenges.push(Some(c));
                zs.push(Some(z));
            }
        }

        (
            State94::new(inner_states, challenges, zs),
            message_as,
        )
    }

    /// Second round of the protocol. Simply generates a random challenge.
    fn second<R: CryptoRngCore>(
        verifier_rng: &mut R,
    ) -> Self::Challenge
    where
        Self: Sized,
    {
        let mut buffer = [0u8; 64];
        verifier_rng.fill_bytes(&mut buffer);
        Challenge::new(&buffer)
    }

    /// Third roud of the protocol
    fn third<R: CryptoRngCore + Clone>(
        statement: &Self::Statement,
        state: Self::State,
        witness: &Self::Witness,
        challenge: &Self::Challenge,
        prover_rng: &mut R,
    ) -> Self::MessageZ
    where
        Self: Sized,
    {
        // Deconstruct variables
        let (clauses, cds_threshold, statements) =
            statement.pattern_match();
        let (witnesses, active_clauses) =
            witness.pattern_match();

        // Create instance of Shamir Secret Sharing
        let shamirs_threshold = clauses - cds_threshold + 1;
        let active_count = active_clauses.len();

        let shamir = ShamirSecretSharing {
            threshold: shamirs_threshold,
            shares: *clauses,
        };

        // Initalize vectors
        let mut shares =
            Vec::with_capacity(shamirs_threshold);
        let mut remaining_xs =
            Vec::with_capacity(active_count);

        let challenges = state.challenges();
        for (i, ci) in challenges
            .iter()
            .enumerate()
        {
            // If clause is active, add the x-coordinate value to the remaining_xs vector
```

```rust
            if active_clauses.contains(&i) {
                remaining_xs.push(
                    <S::Challenge as Shareable>::F::from(
                        (i + 1) as u64,
                    ),
                );
            } else {
                // Otherwise, add the share to the shares vector
                let share = Share {
                    x: <S::Challenge as Shareable>::F::from(
                        (i + 1) as u64,
                    ),
                    y: ci
                        .clone()
                        .unwrap()
                        .share(),
                };

                shares.push(share);
            }
        }

        // Get the missing shares by completing the shares vector with the remaining_xs vector x_values
        let mut missing_shares = shamir
            .complete_shares(
                &challenge.share(),
                &shares,
                &remaining_xs,
            )
            .unwrap();

        // Append the missing shares to the shares vector
        shares.append(&mut missing_shares);

        // Get the message_zs and inner_states of underyling sigma protocols
        let message_zs = state.zs();
        let inner_states = state.inner_states();

        shares
            .iter()
            .map(|share| {
                // Derive the usize from the field element
                let i = S::Challenge::to_usize(share.x) - 1;

                match &challenges[i] {
                    // If this is simulated, return the simulated values
                    Some(ci) => (
                        i,
                        ci.clone(),
                        message_zs[i]
                            .clone()
                            .unwrap(),
                    ),
                    // If not simulated, run the third round of the underlying sigma protocol
                    None => {
                        let ci = Shareable::derive(share.y);
                        let zi = S::third(
                            &statements[i],
                            inner_states[i]
                                .clone()
                                .unwrap(),
                            &witnesses[i],
                            &ci,
                            prover_rng,
                        );

                        (i, ci, zi)
                    }
                }
            })
            .collect_vec()
    }

    /// Verification algorithm
    fn verify(
        statement: &Self::Statement,
        a: &Self::MessageA,
        secret: &Self::Challenge,
        z: &Self::MessageZ,
    ) -> bool
    where
        Self: Sized,
    {
        let valid = a.len() == z.len();

        let (clauses, cds_threshold, statements) =
            statement.pattern_match();

        let mut shares = Vec::with_capacity(*clauses);

        for (i, c, m2) in z {
            let m1 = &a[*i];

            // Firstly verify that the transcript for current index is valid for the instance
            if !S::verify(&statements[*i], m1, &c, &m2) {
                return false;
            }

            // Then add the share to the shares vector
            let share = Share {
                x: <S::Challenge as Shareable>::F::from(
                    (i + 1) as u64,
                ),
                y: c.share(),
            };

            shares.push(share);
        }

        let shamir = ShamirSecretSharing {
            threshold: clauses - cds_threshold + 1,
            shares: *clauses,
        };

        // Use shamir secret sharing to reconstruct secret
        let res = shamir.reconstruct_secret(&shares);

        let combined_secret = res.unwrap_or(
            <S::Challenge as Shareable>::F::default(),
        );

        combined_secret == secret.share() && valid
    }
}
```

A.5 Stacking Sigmas Compiler: `SelfStacker`

```
/// Sigma protocol implementation for self-stacking compiler
impl<S: Stackable> SigmaProtocol for SelfStacker<S> {
    type Statement = StackedStatement<S>;
    type Witness = StackedWitness<S::Witness>;
    type State = StackedState<S>;
    type MessageA = StackedA;
    type Challenge = S::Challenge;
    type MessageZ = StackedZ<S>;

    /// First round of the protocol
    fn first<R: CryptoRngCore + Clone>(
        statement: &StackedStatement<S>,
        witness: &StackedWitness<S::Witness>,
        prover_rng: &mut R,
    ) -> (Self::State, Self::MessageA) {
        // Deconstruct witness
        let StackedWitness {
            nested_witness,
            binding,
        } = witness;

        // First call the underlying protocol with the statement at the active clause
        let (nested_state, bound_message) = S::first(
            statement.bound_statement(binding),
            nested_witness,
            prover_rng,
        );
        // Instance of partial binding commitment scheme that we will use
        let q = statement.height();
```

```rust
        let qbinding = QBinding::new(q);
        // Instantiate pointer to the message from the active clause
        let bound_message = Rc::new(bound_message);

        // Determine our message vector
        let def = Rc::new(S::MessageA::default());
        let initial_messages: Vec<Rc<S::MessageA>> = (0
            ..statement.clauses())
            .map(|i| {
                // If active clause, we use the message from earlier
                if i == binding.index() {
                    bound_message.clone()
                } else {
                    // Otherwise we use the default message
                    def.clone()
                }
            })
            .collect();

        // Here we compute commitment key and equivocation key for the protocol. We appear to
            reuse the same
        // prover_rng but it is mutated and thus different. Still, the change is deterministic.
        let (ck, ek) = qbinding.gen(
            &statement.pp,
            *binding,
            prover_rng,
        );

        // Derive auxiliary value from prover's rng
        let aux = Randomness::random(prover_rng, q);
        // Compute commitment
        let (comm, aux) = qbinding.equivcom(
            &statement.pp,
            &ek,
            &initial_messages,
            Some(aux),
        );

        (
            StackedState {
                nested_state,
                bound_message,
                initial_messages,
                ck: ck.clone(),
                ek,
                aux,
            },
            StackedA(ck, comm),
        )
    }

    /// Second round of the protocol. Random challenge.
    fn second<R: CryptoRngCore>(
        verifier_rng: &mut R,
    ) -> Self::Challenge
    where
        Self: Sized,
    {
        let mut buffer = [0u8; 64];
        verifier_rng.fill_bytes(&mut buffer);
        Challenge::new(&buffer)
    }

    /// Third round of the protocol.
    fn third<R: CryptoRngCore + Clone>(
        statement: &Self::Statement,
        state: Self::State,
        witness: &Self::Witness,
        challenge: &Self::Challenge,
        prover_rng: &mut R,
    ) -> Self::MessageZ {
        // Deconstruct the state struct
        let StackedState {
            nested_state,
            bound_message,
            initial_messages,
            ck,
            ek,
            aux,
        } = state;

        // Deconstruct the witness struct
        let StackedWitness {
            nested_witness,
            binding,
        } = witness;
        let qbinding = QBinding::new(statement.height());

        // Call third round algorithm of underlying protocol
        let nested_z = S::third(
            statement.bound_statement(binding),
            nested_state,
            nested_witness,
            challenge,
            prover_rng,
        );

        let new_messages: Vec<Rc<S::MessageA>> = (0
            ..statement.clauses())
            .map(|i| {
                // Same as earlier, if active clause, we use the message from first round
                if i == binding.index() {
                    bound_message.clone()
                } else {
                    // Otherwise, we need to simulate the first message from statement,
                        challenge and z from earlier
                    Rc::new(S::simulate(
                        statement
                            .statement_at(i)
                            .unwrap(),
                        challenge,
                        &nested_z,
                    ))
                }
            })
            .collect();

        // Equivocate the initial vector of messages with new messages
        // obtaining new auxiliary value
        let aux_new = qbinding.equiv(
            &statement.pp,
            &ek,
            &initial_messages,
            &new_messages,
            &aux,
        );

        StackedZ {
            ck,
            message: nested_z,
            aux: aux_new,
        }
    }

    /// Verification algorithm for the protocol
    fn verify(
        statement: &Self::Statement,
        a: &Self::MessageA,
        c: &Self::Challenge,
        z: &Self::MessageZ,
    ) -> bool
    where
        Self: Sized,
    {
        // Deconstruct variables from structs
        // Here we get the commitment key, and commitment from first round of stacking protocol
        let StackedA(ck_a, comm) = a;
        // Here we get the commitment key, messages, and aux variable from the third round of
            stacker
        let StackedZ {
            ck: ck_z,
            message,
            aux,
        } = z;

        // Now we go through every statement and simulate with the recyclable third round
            message
        // and challenge from 2nd round
        let v: Vec<Rc<S::MessageA>> = statement
```

```
            .statements()
            .iter()
            .map(|s| Rc::new(S::simulate(s, c, &message)))
            .collect();

        // Using bindcom algorithm, we compute the commitment to this vector of messages
        let comm_check = QBinding::new(statement.height())
            .bind(&statement.pp, ck_a, &v, aux);

        // Now we want to verify that the messages are valid for every clause
        let nested_check = statement
            .statements()
            .iter()
            .zip(v.iter())
            .all(|(s, m)| S::verify(s, m, c, &message));

        ck_a == ck_z && *comm == comm_check && nested_check
    }
}
```

A.6 Test Example: CDS94 Tests

```
pub type CDS94Test = (
    SelfCompiler94<Schnorr>,
    Statement94<Schnorr>,
    Witness94<Schnorr>,
    Witness94<Schnorr>,
    ChaCha20Rng,
    ChaCha20Rng,
);

fn test_init<const N: usize, const D: usize>(
    is_positive: bool,
) -> CDS94Test {
    // INIT //
    assert!(D <= N);
    // closure to generate random witnesses
    let m = |_| {
        Scalar::random(&mut ChaCha20Rng::from_entropy())
    };
    // generate witnesses
    let actual_witnesses: Vec<Scalar> = (0..N)
        .map(m)
        .collect();
    // generate the prover's witnesses - for inactive clauses
    // the prover generates a random witness

    let provers_witnesses: Vec<Scalar> = if is_positive {
        actual_witnesses
            .to_owned()
            .iter()
            .enumerate()
            .map(|(i, s)| {
                if i < D {
                    s.clone()
                } else {
                    Scalar::random(
                        &mut ChaCha20Rng::from_entropy(),
                    )
                }
            })
            .collect()
    } else {
        actual_witnesses
            .to_owned()
            .iter()
            .enumerate()
            .map(|(_i, _s)| {
                Scalar::random(
                    &mut ChaCha20Rng::from_entropy(),
                )
            })
            .collect()
    };
    // Set of booleans indicating which clauses are active
    let active_clauses: HashSet<usize> = (0..D).collect();
    // generate the statement (aka protocol) for each clause
    let statements = actual_witnesses
        .to_owned()
        .iter()
        .map(|w| Schnorr::init(*w))
        .collect_vec();

    let protocol = SelfCompiler94::new(N, D);

    let statement = Statement94::new(N, D, statements);

    let actual_witnesses = Witness94::new(
        actual_witnesses,
        active_clauses.clone(),
    );

    let provers_witnesses = Witness94::new(
        provers_witnesses,
        active_clauses.clone(),
    );

    let provers_rng = ChaCha20Rng::from_seed([0u8; 32]);
    let verifiers_rng = ChaCha20Rng::from_seed([1u8; 32]);

    (
        protocol,
        statement,
        actual_witnesses,
```

```rust
        provers_witnesses,
        provers_rng,
        verifiers_rng,
    )
}

#[test]
fn first_message_works() {
    const N: usize = 2;
    const D: usize = 1;
    let (
        _protocol,
        statement,
        actual_witnesses,
        provers_witnesses,
        provers_rng,
        _verifiers_rng,
    ) = test_init::<N, D>(true);

    let (.., statements) = statement.pattern_match();

    let (_state, message_a) = SelfCompiler94::first(
        &statement,
        &provers_witnesses,
        &mut provers_rng.clone(),
    );
    assert!(message_a.len() == N);
    let (_, testc) = Schnorr::first(
        &statements[0],
        &actual_witnesses.witnesses()[0],
        &mut provers_rng.clone(),
    );
    assert!(testc == message_a[0]);
}

#[test]
fn third_message_works() {
    const N: usize = 2;
    const D: usize = 1;
    let (
        _protocol,
        statement,
        _actual_witnesses,
        provers_witnesses,
        mut provers_rng,
        verifiers_rng,
    ) = test_init::<N, D>(true);

    let (state, message_a) = SelfCompiler94::first(
        &statement,
        &provers_witnesses,
        &mut provers_rng.clone(),
    );
    let challenge = SelfCompiler94::<Schnorr>::second(
        &mut verifiers_rng.clone(),
    );
    // Third message
    let proof = SelfCompiler94::third(
        &statement,
        state,
        &provers_witnesses,
        &challenge,
        &mut provers_rng,
    );

    assert!(SelfCompiler94::verify(
        &statement, &message_a, &challenge, &proof
    ));
}

#[test]
fn cds_works() {
    // INIT //
    // number of clauses
    const N: usize = 128;
    const D: usize = 1;
    let (
        _protocol,
        statement,
        _actual_witnesses,
        provers_witnesses,
        mut provers_rng,
        verifiers_rng,
    ) = test_init::<N, D>(true);

    let (state, message_a) = SelfCompiler94::first(
        &statement,
        &provers_witnesses,
        &mut provers_rng,
    );
    let challenge = SelfCompiler94::<Schnorr>::second(
        &mut verifiers_rng.clone(),
    );
    // Third message
    let proof = SelfCompiler94::third(
        &statement,
        state,
        &provers_witnesses,
        &challenge,
        &mut provers_rng,
    );

    assert!(SelfCompiler94::verify(
        &statement, &message_a, &challenge, &proof
    ));
}
```

```

#[test]
fn cds_fails() {
    // INIT //
    // number of clauses
    const N: usize = 128;
    const D: usize = 1;
    let (
        _protocol,
        statement,
        _actual_witnesses,
        provers_witnesses,
        mut provers_rng,
        verifiers_rng,
    ) = test_init::<N, D>(false);

    let (state, message_a) = SelfCompiler94::first(
        &statement,
        &provers_witnesses,
        &mut provers_rng,
    );
    let challenge = SelfCompiler94::<Schnorr>::second(
        &mut verifiers_rng.clone(),
    );
    // Third message
    let proof = SelfCompiler94::third(
        &statement,
        state,
        &provers_witnesses,
        &challenge,
        &mut provers_rng,
    );

    assert!(!SelfCompiler94::verify(
        &statement, &message_a, &challenge, &proof
    ));
}
```

Appendix B

Progress Report

B.1 Introduction

Zero-Knowledge proofs [GMR85] are protocols that allow a prover to convince a verifier that an NP statement is true, while revealing no additional information except the validity of their assertion. Early research proved that all languages in NP have zero-knowledge proof systems [GMW86], and recent results have provided more efficient zero-knowledge proofs that are being used in practice.

In many cases, it is desirable to have a zero-knowledge proof for a disjunctive statement, which is an NP statement with a set of clauses that are connected with logical ORs. Disjunctive statements have very useful properties that occur commonly in practice, such as proving ones membership to a particular group, or showing the existence of a bug in a verifier's code base [HK20]. Zero-knowledge becomes an important property in cases where revealing the exact clause (or clauses) that is true may reveal private information about the prover, such as their identity. A long line of research has focussed on how n zero-knowledge proofs, each for one statement, can be composed into a new zero-knowledge proof of the disjunction of these statements.

In their 1994 paper, Cramer, Damgård, and Schoenmakers [CDS94] provide a generic compiler to compose 3-round public coin proofs of knowledge, or more succinctly (and more popularly) known as Σ-protocols. More recently, Goel *et al.* [Goe+21] improved on this further, providing a generic compiler for a large class of Σ-protocols and also reducing the size of the resulting proof.

While extensive research has been conducted, there is a lack of notable real-world implementations of these results [1]. This project seeks to build upon their work by implementing the compilers described in [CDS94] and [Goe+21]. Once implemented, we aim to provide a benchmark for both protocols to explore and measure how they differ. This will hopefully provide some valuable insights as to how these designs perform in practice, which may in turn lead to further improvements in the future that may have a broad impact on existing and upcoming cryptographic systems that rely on such a use case.

B.1.1 Related work

As part of their work in [Goe+21], Hall-Andersen provides an implementation of the Stacking Sigmas (SS) compiler [Hal21]. In this implementation, they apply the SS compiler to Schnorr over Ristretto25519 to obtain efficient ring signatures from discrete log and random oracles.

[1]It should be noted that Hall-Andersen [Hal21] has provided a benchmark of applying the compiler in [Goe+21] to Schnorr's discrete log protocol [Sch89].

B.2 Background

B.2.1 Notation

Throughout this paper, we use λ to denote the computational security parameter and κ to denote the statistical security parameter. Computational security refers to a cryptographic system's security against a computationally bounded adversary, while statistical security refers to security that is not dependent on the computational power of the adversary but instead on the security that negligible statistical probability provides.

Additionally, we denote by $x \xleftarrow{\$} \mathcal{D}$ the sampling of "x" from the distribution "$\mathcal{D}$".

B.2.2 Disjunctive Zero-Knowledge

Definition 17 (NP Relations). Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation. Then $w(x) = \{w \mid (x,w) \in R\}$ and $L_R = \{x \mid \exists w, (x,w) \in R\}$. If $(x,w) \in R$, we say that w is a witness for x. R is an NP-relation if it fulfils the following two properties:

1. **Polynomially bounded.** We say that R is *polynomially bounded* if there exists a polynomial p such that $|w| \leq p(|x|), \forall (x,w) \in R$.
2. **Polynomial-time verification.** There exists a polynomial-time algorithm for deciding membership in R. Consequently, $L_R \in NP$.

Throughout this document, we will use $\mathcal{R}$ to refer to a binary NP-relation.

Definition 18 (Zero-Knowledge). A proof or argument system (P,V) is zero-knowledge over $\mathcal{R}$ if there exists a *probabilistic polynomial time* (PPT) simulator $\mathcal{S}$, such that for all $(x,w) \in R$, the distribution of the output $\mathcal{S}(1^\lambda, x)$ of the simulator is indistinguishable from the distribution over the conversations generated by the interaction of P and V, from the perspective of V; we denote this with $\text{View}_V(P(x,w), V(x))$. Conversations between P and V are ordered triples of the form (a,c,z), and are known as *transcripts*.

Intuitively, this means that V should not learn anything from the transcripts with P that they cannot already learn on their own by running the simulator $\mathcal{S}$; they learn nothing new.

Definition 19 (Disjunctive Zero-Knowledge). Given a sequence of statements $(x_1, x_2, \ldots, x_l)$, a *disjunctive zero-knowledge proof* is a protocol to prove in zero-knowledge that $x_1 \in \mathcal{L}_1 \vee x_2 \in \mathcal{L}_2 \vee \ldots \vee x_l \in \mathcal{L}_l$, for NP languages $\mathcal{L}_i$. We term clauses for which the prover has a witness for as *active* clauses.

Definition 20 (Honest-Verifier Zero-Knowledge). A proof system is *honest-verifier zero-knowledge* if it only requires that $\mathcal{S}$ is an efficient simulator for honest (non-malicious) probabilistic polynomial time verifier strategies V. If V is malicious then the distribution of the output $\mathcal{S}(x)$ will no longer be indistinguishable from $\text{View}_V(P(x,w), V(x))$ for such proof systems.

B.2.3 Σ-protocols

Definition 21 (Σ-Protocol). Let $\mathcal{R}$ be an NP relation. A Σ-protocol $\Pi = (A, Z, \phi)$ for $\mathcal{R}$ is a 3-round protocol between a prover algorithm P and a verifier algorithm V. The protocol consists of a tuple of probabilistic polynomial time algorithms (A, Z, ϕ) with the following interfaces:

- $a \leftarrow A(x, w; r^p)$: Given statement x, witness $w \in w(x)$, and prover randomness r^p as input; output the first message a that P sends to V in the first round.
- $c \xleftarrow{\$} \{0,1\}^\kappa$: V samples a random challenge c and sends it to P in the second round.
- $z \leftarrow Z(x, w, c; r^p)$: Given x, w, c, and r^p as input; output the message z that P sends to V in the third round.
- $b \leftarrow \phi(x, a, c, z)$: Given x, and the messages in the transcript, output a bit $b \in \{0,1\}$. This algorithm is executed by V, and V accepts if $b = 1$.

A Σ-protocol has the following properties:

1. **Completeness.** Π is complete if for any $x, w \in w(x)$, and any prover randomness $r^p \xleftarrow{\$} \{0,1\}^\lambda$, the verifier accepts with probability 1.
$$Pr\left[\phi(x, a, c, z) = 1 \mid a \leftarrow A(x, w; r^p); c \xleftarrow{\$} \{0,1\}^\kappa; z \rightarrow Z(x, w, c; r^p)\right] = 1$$
2. **Special Soundness.** Π is said to have special soundness if there exists a PPT extractor $\mathcal{E}$, such that given any two transcripts (a, c, z) and (a, c', z') for statement x, where $c \neq c'$ and $\phi(x, a, c, z) = \phi(x, a, c', z') = 1$, an element of $w(x)$ can be computed by $\mathcal{E}$.
3. **Special Honest-Verifier Zero-Knowledge.** Π is SHVZK if there exists a PPT simulator $\mathcal{S}$, such that for any x, w, $(x, w) \in \mathcal{R}$, the distribution over the output $\mathcal{S}(1^\lambda, x, c^*)$ is indistinguishable from the distribution over transcripts produced by the interaction between V and P when the challenge is c^*.
$$\{(a, z) \mid c^* \xleftarrow{\$} \{0,1\}^\kappa; (a, z) \leftarrow \mathcal{S}(1^\lambda, x, c^*)\} \approx_{c^*}$$
$$\{(a, z) \mid r^p \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow A(x, w; r^p); c^* \xleftarrow{\$} \{0,1\}^\kappa; z \leftarrow Z(x, w, c^*; r^p)\}$$

Definition 22 (Witness Indistinguishable (WI))**.** A Σ-protocol is witness indistinguishable over $\mathcal{R}$ if for any V', any large enough input x, any $w_1, w_2 \in w(x)$, and for any fixed challenge c^*, the distribution over transcripts in the form (a_1, c, z_1) and (a_2, c, z_2) are indistinguishable, where $a_i \leftarrow A(x, w_i; r^p)$ and $z_i \leftarrow Z(x, w_i, c^*; r^p)$ for $i \in \{1, 2\}$. This means that the prover reveals no information about which are the active clauses.

Definition 23 (Informal definition of Witness Hiding (WH))**.** For any x that is generated with a certain probability distribution by a generator $\mathcal{G}$ which outputs pairs $(x, w) \in \mathcal{R}$, a Σ-protocol is witness hiding over $\mathcal{G}$, if it does not help even a cheating verifier to compute a witness for x with non-negligible probability. Refer to [FS90] for details. WH is a weaker property than general zero-knowledge, as it only asserts that the verifier cannot learn about the witness (not asserting anything about other information). That said, it can replace zero-knowledge in many protocol constructions, as it is in most Σ-protocols.

B.2.4 Schnorr's Identification Protocol

There are two parties in an *identification scheme*, the prover P and the verifier V, and the objective of the protocol is for the prover to convince the verifier that they are who they claim to be. More precisely, V is convinced that P knows the private key that corresponds to the public key of P. A familiar example is the standard protocol of password authentication.

Schnorr's protocol [Sch89] is an identification scheme where P proves knowledge of the discrete log x of a group element $H \in \mathbb{G}$, where $H = x \cdot G$ for some generator $G \in \mathbb{G}$. $(\mathbb{G}, +)$ is a finite abelian group with $+$ as the binary operator[2]. The protocol relies on the assumption that finding x given only H and G is computationally difficult – this is not always the case. The hardness of finding the discrete log depends on the choice of group $\mathbb{G}$ (cite some resource or elaborate). Conversely, proving that $H = x \cdot G$ given x and G can be computed efficiently.

In our implementation, we plan to use the Ristretto group [Val+]: a construction of a prime order group from a family of elliptic curves known as Edwards curves [Edw07].

Definition 24 (Schnorr's Protocol [Sch89])**.** Let $\mathbb{G} = E(\mathbb{F}_q)$, where E is an elliptic curve over the finite field $\mathbb{F}_q$. Suppose that both the prover P and verifier V agree on E and $\mathbb{F}_q$, then let $H \in E(\mathbb{F}_q)$ be the public key that corresponds to the private key x such that $H = x \cdot G$. The prover convinces the verifier that they have knowledge of the private key by doing the following:

1. P generates random $r \in \mathbb{F}_q$, and computes the point $\mathcal{A} = r \cdot G$. P sends the point $\mathcal{A}$ to V.
2. V computes a random $c \in \mathbb{F}_q$, and sends c to the P.
3. P computes $z = (r + c \cdot x_P) \mod n$, and sends z to the V.
4. V checks that $z \cdot G = \mathcal{A} + c \cdot H$.

Evaluating the final equation, we can easily see that V accepts if and only if $x = x_P$.

$$z \cdot G = \mathcal{A} + c \cdot H \iff r \cdot G + c \cdot x_P \cdot G = r \cdot G + c \cdot x \cdot G \iff x = x_P$$

A simulator for Schnorr's protocol. We construct a simulator for Schnorr's protocol by running it "in reverse":

```
Let our simulator be $S(H)$

$z \xleftarrow{\$} \mathbb{F}_q$
$c \xleftarrow{\$} \mathbb{F}_q$
$\mathcal{A} = z \cdot G - c \cdot H$
---------------------------------
output $(\mathcal{A}, c, z)$
```

Since z and c are both chosen randomly, the resulting $\mathcal{A}$ is also random, and our output will have the same distribution as the distribution over transcripts in an actual interaction.

B.2.5 Shamir's Secret Sharing Scheme

A *secret sharing scheme* is a method of distributing a secret s to n participants in a way that no one participant has intelligible information about the secret. This is achieved by splitting up s into *shares*, distributing one share to each participant in a way that a subset of participants can reconstruct s. Subsets that can reconstruct the secret are called *qualified sets*. For *perfect* secret sharing schemes, like

[2]We have chosen to define $\mathbb{G}$ with the $+$ operator because our implementation uses elliptic curves which are finite abelian groups over addition. The discrete log can be defined equivalently with multiplication like so: $h = g^x$

Shamir's, participants in the complement *non-qualified* sets cannot obtain any information about the secret.

Shamir's secret sharing scheme [Sha79] is also what is known as a *threshold sharing scheme*. These are schemes that produce qualified sets of size d. Any d out of n participants can reconstruct the secret; with $d-1$ shares and less, no information about the secret can be obtained.

B.2.6 CDS94 Compiler

In our implementation, we will use Schnorr's discrete log protocol over Ristretto25519 and Shamir's secret sharing scheme to demonstrate the compilation of Σ-protocol into a Σ-protocol for the disjunction of n statements.

We will attempt to make the implementation as general as possible to open up for the future possibility to take any Σ-protocol that suits our requirements and transform it disjunctive zero-knowledge Σ-protocol.

The Witness Indistinguishable (WI) compilation In their paper, Cramer *et al* [CDS94] presents 2 primary ways to construct a WI protocol from a Σ-protocol $\mathcal{P}$ (Theorem 8 and 9).

- Theorem 8 requires a smooth secret sharing scheme, and a HVZK Σ-protocol, while
- Theorem 9 requires special honest-verifier ZK (SHVZK) with at least a semi-smooth secret sharing scheme.

Since, SSS is a smooth threshold secret sharing scheme (required for 8), and Schnorr's protocol is SHVZK (required for 9), we can choose either construction. *We will use **Theorem 8** in this project.*

Theorem 8 [CDS94]. Given $\mathcal{P}$, R_Γ, Γ, and $\mathcal{S}(k)$ where

- $\mathcal{P}$ is a 3-round public coin (Σ-protocol) HVZK proof of knowledge for relation R, which satisfies the special soundness property.
- $\Gamma = \{\Gamma(k)\}$ is a family of monotone access structures
- $\{\mathcal{S}(k)\}$ is a family of smooth secret sharing schemes such that the access structure of $\mathcal{S}(k)$ is $\Gamma(k)^*$
- R_Γ is a relation where $((x_1, \ldots, x_m), (w_1, \ldots, w_m)) \in R_\Gamma$ if and only if ($\iff$)
  - all x_i's are of the same length k, and
  - the set of indices i for which $(x_i, w_i) \in R$ corresponds to a qualified set in $\Gamma(k)$

Then, there exists a Σ-protocol that is witness indistinguishable for the relation R_Γ. (The proof of Theorem 8 and 9 is given in their paper.)

Protocol Description Let $A \in \Gamma$ denote the set of indices i for which our prover P knows a witness for x_i

1. For each $i \in \bar{A}$, P runs a simulator on input x_i to produce transcripts of conversations in the form (m_1^i, c_i, m_2^i).

- For each $i \in A$, P inputs the witness w_i for x_i to $\mathcal{P}$ and takes what the prover P^* in $\mathcal{P}$ sends as m_1 as m_1^i.
- Essentially we take the return value of the first round of $\mathcal{P}$ as our message m_1^i.
- Finally, P sends all m_1^i to V, where $i = 1, \ldots, n$

2. V chooses a t-bit string s at random and sends it to P
3. P forms challenges c_i for $i \in A$, such that $share(c_i) \cup \{share(c_j) | j \in \bar{A}\}$ is a qualified set in Γ consistent with s.
    - For $i \in A$, P uses it's knowledge of $w(x_i)$ to compute a valid m_2^i for (m_1^i, c_i) by running the prover's algorithm in $\mathcal{P}$.
    - P then sends c_i, m_2^i for $i = 1, \ldots, n$ to V.
4. V checks that all conversations (m_1^i, c_i, m_2^i) would lead to acceptance by the verifier in $\mathcal{P}$
    - During this process, V checks that $share(c_i)$ is consistent with secret s.
    - Accept if all true, otherwise reject.

B.3 Progress

Currently, a basic implementation of Schnorr's protocol and CDS94 has been completed. More work is required to ensure that the compiler is truly witness-indistinguishable, as the implementation thus far does not adhere exactly to the protocol defined. This is primarily because existing libraries of Shamir's secret sharing scheme do not provide the ability to reconstruct shares to a qualified set given the secret and an unqualified set. It is not surprising that such a feature is unimplemented as it is not relevant to using the scheme to distribute a secret. However, it is a crucial feature for the CDS94 compiler as the secret sharing scheme is used to ensure that the prover will have to generate a random c_i for each statement x_i, and cannot cheat. To overcome this, we will implement Shamir's secret sharing scheme with this feature, instead of relying on an external crate.

Aside from this, we have dedicated time to review our choice of tools and software development methodology for the project. In later sections, we justify why we are committing to use Rust and Github, and why we are changing our decision on Notion for project management and using a plan-based software development methodology. Additionally, a significant portion of time was spent studying the results in [CDS94] and extracting the requirements from its design. We have also conducted preliminary reading of [Goe+21] to understand if there are any components that can be shared by both compilers and therefore must be prioritised.

Referring to Table **??** in Section **??** of the appendix, our current progress is inline with our initial project plan. In fact, we are likely to complete our implementation of CDS94 early. This in part due to our initial underestimate of how much progress can be made, and the lack of context of the problem before work began. Hence, we have provided an updated project plan in Section B.4.2. In the following subsections, we present the core requirements of our system, our benchmark plan, and discuss changes that we have made from the project specification.

B.3.1 Requirements and Design Decisions

In this section, we outline the core requirements of our system and present solutions to achieve them.

Interoperability. A core requirement of our project is to have a good interface for developers to use or extend our compiler. This will require that we develop an API that strictly defines its required inputs and expected outputs, which should not only be clear within the source code but must be thoroughly documented as well.

Within Rust, we can enforce the properties of inputs to our API with Rust traits[3] and generics. This tells *rustc* (the Rust compiler) to conduct type checks at compile-time to ensure that developers are appropriately interfacing with our API. For example, the Σ-protocols composed by the CDS94 compiler must have a way to simulate transcripts and we can enforce this through a `SigmaProtocol` trait that requires the implementation of a `simulate()` method.

Comprehensive documentation can be achieved by using `rustdoc`: a tool to automatically generate documentation that comes with the standard Rust distribution. It parses comments in a Rust project and produces HTML, CSS, and JavaScript, that can then be viewed from a browser. This will help developers understand how to use our system.

Maintainability. As more work is done, it will become increasingly important that our code is maintainable and easily extensible. Therefore, each component of our system that has a distinct responsibility should be separated into its own Rust *crate*[4] and must provide an interface for other crates to use it. This aligns with the microservices architecture that many software applications are built upon.

Considering this, we have organised our source code into a mono-repository using *cargo workspaces*. This enables us to separate each software component (for CDS94, schnorr's protocol, and shamir's scheme) into distinct crates, where its dependencies are defined and managed in isolation. At the same time, cargo optimises the compilation and build process of all the crates in the workspace, fetching shared dependencies once instead of several times if each crate was in a different workspace. If required, we can easily move each crate into its own repository in the future. Having a mono-repository also has the added benefit of improving the development process, as all source code is housed in a central repository.

Benchmarking. The main of goal of this project is to provide benchmarks to compare different compilers for composing Σ-protocols into a disjunctive zero-knowledge proof. These benchmarks should measure the performance of each compiler on different proof sizes, and measure the run-times of the verifier and prover respectively. These metrics are discussed further in Section B.3.4.

Benchmarking is not a simple endeavour without the right tools, as algorithm run-times depend on the state of the computer during benchmark tests. Therefore, we will use the `criterion` crate [HA22] which is a statistics-driven micro-benchmarking tool. It collects and stores statistical information from each run and can provide statistical confidence of run-times, allowing us to provide some statistical guarantee for our results. Furthermore, the crate also provides plots and graphs to visualise the performance of each benchmark.

Testing. We are implementing a cryptographic system and will require rigorous and comprehensive testing to ensure not only meets its requirements but is also robust against potential attack vectors. We split testing into two categories: *application* and *security* testing. We discuss our approach to application testing in Section B.3.2, and our considerations and methods for security testing in Section B.3.3.

[3]Traits in Rust are similar to interfaces in Java, and most similar to typeclasses in Haskell.

[4]Crates are Rust's equivalent of libraries or modules in other programming languages.

B.3.2 Application Testing

Unit and integration tests must be written for every feature in each crate and these must include both positive and negative tests. Test coverage should be used to reveal parts of the code that are not covered by our tests. This will help identify branches of code that are either redundant or are not yet tested.

To prioritise functionality and validate that our features meet their requirements, we plan to use Test-Driven-Development (TDD) in our software development process. TDD is a programming style where tests are written based on requirements before features that aim to pass these tests are implemented. We show an illustration of the TDD software development lifecycle in Figure B.1 below.

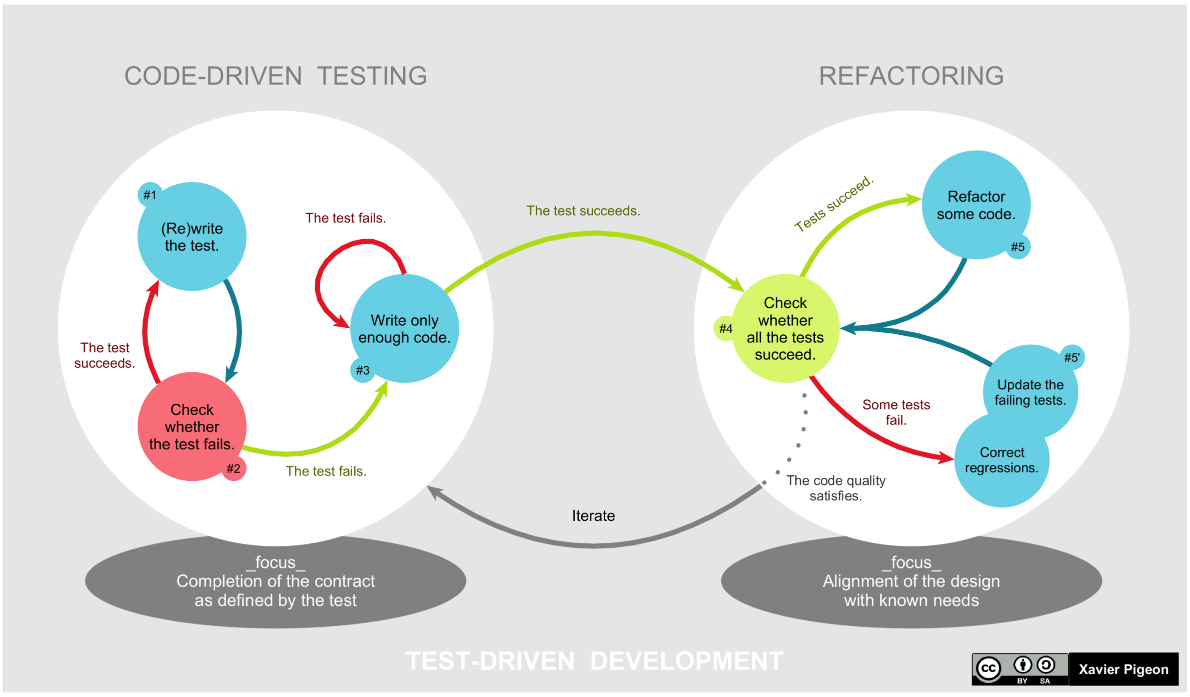

CODE-DRIVEN TESTING
REFACTORING
#1 (Re)write the test.
The test fails.
Write only enough code. #3
The test succeeds.
Check whether the test fails. #2
The test fails.
focus Completion of the contract as defined by the test
Tests succeed.
Refactor some code. #5
#4 Check whether all the tests succeed.
#5' Update the failing tests.
Some tests fail.
Correct regressions.
The code quality satisfies.
Iterate
focus Alignment of the design with known needs
TEST-DRIVEN DEVELOPMENT
Xavier Pigeon

Figure B.1: TDD Lifecycle [Pig15]

TDD places an emphasis on first producing code that is minimally sufficient to pass the written tests. After which, code is refactored until it fulfils standard best practices such as readability, modularity, encapsulation etcetera. This appropriately prioritises functionality over readability, but still upholds written code to a high quality of design. Writing tests before implementation also helps during regression testing, to identify bugs when changes are made. This will not only improve code quality, but also make development more efficient.

We use Rust's native testing framework which provides a convenient interface to write both unit and integration tests. Additionally, the compiler *rustc* can be configured to generate code coverage reports, which we can use to monitor development progress. Testing can be automated using Travis and Github Actions to incorporate tests into the software development lifecycle, although this is not a priority.

B.3.3 Security Testing

In this section, we present our security testing strategy. We follow practices suggested by the OWASP Web Security Testing Guide [SM20], where they cover four main testing techniques: (1) manual inspection and reviews, (2) threat modeling, (3) code review, (4) penetration testing. Some practices covered in the guide are not specifically relevant to our system, considering the small-scale of our project. Nevertheless, the core idea remains and informs our approach.

Monthly Review and Reporting. We propose conducting monthly (every 5 weeks) code reviews to identify potential security problems that are difficult to identify through black-box testing. Given that there is only one developer working on this project, this process is still subject to confirmation with the project supervisor. Ideally, the main developer for a feature presents their code to another party, as it is often more natural for a third-party to identify potential problems. Otherwise, it is still possible for the main developer to conduct a personal source code review. This can be done after a certain amount of time passes since implementing the feature, after initial assumptions that were made during development have been forgotten. Our proposed duration is 5 weeks. That said, we acknowledge the potential problems with this proposal, bringing us to our second initiative.

Threat Modeling. A popular technique to help system designers think about potential security threats that their system might face. We employ this technique in our software development process to ensure that we incorporate security into the design of our features, before development even begins. These design requirements should be captured within our application tests if possible. In this way, threat modeling and TDD complement each other. Refer to Section D for our current threat model (as of November 27 2022).

This section details the software architecture of our system from a security perspective and how various components interact. We identify trust boundaries and channels which may have potential vulnerabilities, and outline the potential attack vectors and our mitigation strategy. Currently, this section is still premature and more emphasis will be placed on it for subsequent sprint cycles. The threat model should be updated in every sprint cycle, whenever new features are added, or when a code review identifies an issue. We currently use OWASP Threat Dragon to create our threat model [Goo+21].

Fuzzing. We have limited resources for this project, and carrying out penetration tests is infeasible. To compensate for this, we intend to use fuzz testing to carry out black-box testing. Fuzzing is a technique to automatically inject random, invalid, and unexpected data into our API. This may help to identify bugs and edge cases that can cause problems. To do this, we make use of `afl.rs` [Aut22] – Rust's variant of a popular fuzzing tool for C and C++ called AFL (American fuzzy lop) [Zal17].

Static Analysis. Static code analysis can be used to provide metrics that may help identify coding issues that could lead to deeper security bugs. We use Mozilla's Rust Code Analysis library to carry out static code analysis [Ard+20].

B.3.4 Benchmark Plan

To compare the relative performance of each compiler to the other, we have defined a few metrics as our benchmarks:

1. Proof size (in bytes)
2. Prover runtime
3. Verifier runtime

Crucially, we will be measuring how each metric changes with the *number of clauses* in our disjunctive proof. Additionally, we will also make use of the same set of Σ-protocols to remove any minor differences in performance that may arise due to the implementation and type of Σ-protocol used.

Instructions on how to run our benchmarks will be provided to allow anyone to reproduce results on a machine of their choice. We will only be producing benchmarks on a single machine (Intel chip). We have determined that producing benchmarks on different machines to compare hardware performance is out-of-scope for this project. We list our hardware specification in Appendix C.

B.4 Project management

In this section, we first cover two notable project management changes since the project specification. We then present our updated plan for term 2 and risk management plan.

B.4.1 Software Development Methodology

We have opted to change from an incremental plan-based methodology, to Scrum, an agile software development methodology. After the first few weeks of development, we found that many necessary requirements of a real-world implementation of the compiler are not covered in research papers. This is not unexpected as the protocol's description within literature focuses on the essence of the design rather than how it can be practically built. Moreover, many requirements become obvious only after the initial implementation, implying that a more agile approach is necessary. In light of this, having the flexibility of an agile methodology like Scrum will allow us to iteratively design our compiler, starting from minimally functioning code and continuously improving on its design. This is a more natural prioritisation of functioning code and testing instead of spending time designing a comprehensive plan that has a high probability of changing.

Scrum Sprints. Our sprints will be split into four phases: (1) Feature creation, (2) Scrum planning, (3) Implementation and Testing, (4) Sprint Review. During feature creation, we begin by translating user requirements into application features. Each feature will have a one-to-one mapping to a Github Issue, which will be created to keep track of the progress toward implementing said feature. Next, scrum planning is about starting the new sprint which we represent with Github Milestones. During this process, we assign issues that are on the backlog to the current sprint, signifying that these features should be completed by the end of the two weeks and, thus, quantifying our milestone. Lastly, the sprint review process will involve documenting the progress made in the last sprint, introducing new issues if applicable, and updating the roadmap of the project. These will be documented using Github Wiki.

Project Management Tools We decided to make a switch from Notion to Github for project management. This is primarily due to the integration of Github Projects and our Github repositories. While Notion provides more note taking capabilities, it is limited in its integration with Github. For example, when a task is created on Notion, there is no integration with Github to automatically create an issue on the repository. We found that having issues recorded on Github is a useful way to document the features of our system while keeping track of current task, all while helping to manage source code history.

Hence, we decide on using Github's native project management tools like Github Projects, Issues, and Milestones. We will be using Github Milestones to manage our two-week long sprints. Github Projects will be used to manage issues and tasks across repositories. Github Issues will be created and linked to a Milestone (or Sprint) and define tasks that should be completed by the end of the sprint. Lastly, Github Wiki shall be used as a replacement for Notion's note taking capabilities, serving as a convenient place to document work on the project.

B.4.2 Updated Timetable

In the following updated schedule, we present a more ambitious project plan which aims to complete an implementation of [Goe+22] before the start of Easter. This timetable will be adjusted accordingly based on our sprint reviews, and any adjustments will be made known to the project supervisor.

Table B.1: Updated Project Plan

Sprint	**Main Task**	**Goal**
0 (T1 W9-10)	Complete basic implementation of CDS94 (WI & WH)	-
~~1 (Christmas)~~	~~Set up security tests and benchmark tests. Study [Goe+21].~~	~~-~~
~~2 (Christmas)~~	~~Continue from previous sprint. Attempt basic implementation of SS.~~	~~Complete benchmarks for CDS94 and collect data.~~
3 (T2 W1-2)	~~Implementation of SS compiler. Update benchmark tests if necessary.~~ Tidy up CDS tests and documentation. Benchmark tests.	~~-~~ Complete benchmarks for CDS94 and collect data
4 (T2 W3-4)	~~Implement final features of SS compiler. Study [Goe+22].~~ Begin implementation of SS compiler. Update benchmark tests if necessary.	Complete implementation of SS.
5 (T2 W5-6)	Benchmark SS compiler and organise data collected for final presentation. Attempt basic implementation of [Goe+22].	Able to explain concepts in [Goe+22] to project supervisor.
6 (T2 W7-8)	Preparation for final presentation. Continue with [Goe+22].	Final Presentation.
7 (T2 W9-10)	Complete implementation of [Goe+22].	Benchmark [Goe+22] and collect data.
Easter Break	Write up of final report alongside exam revision.	Proof read and check report.
T3 W1-2	-	Submit final report.

B.4.3 Risk Management Plan

In this section we highlight the dependencies of the project and present potential risks and our plan to mitigate such risks:

- This project relies on existing implementations of cryptographic protocols; before using any of them, a thorough check should be made to ensure that we are not violating any license agreements.
- The benchmark tests should not be resource intensive and working on the department's machines should be sufficient in the worst case. Rust has also been verified to work on the department's machines. In case a different version of Rust is required, a request to the department will be made to install it.
- Understanding the literature may require more time than expected. In our case, covering [CDS94] before [Goe+21] was helpful in familiarising ourselves with cryptographic concepts surrounding zero-knowledge. This is also why not all research for the project has been done yet.
  - Currently, we are confident that we can complete the SS compiler. However, if this is not complete before the final presentation, then we will only present the performance of CDS94 and [Hal21].
  - In the case that our implementation of SS only completes by Week 6, we will not implement [Goe+22] and instead prioritise our benchmarks for SS and collecting relevant data.
- Proper version control practices will be crucial in ensuring that we can continue work on a different machine if our primary computer were to malfunction.

B.4.4 Legal, social, ethical and professional issues

For the next 6 months, work on the project will remain private, and made available only to direct stakeholders. That said, there is a chance that our work may be made public and published in the future, and users could accidentally or willfully misuse our program. As such, we currently intend to host the program publicly on Github under the GNU General Public License v3 (GPLv3), which is a free, copyleft license for software [GNU07]. More research will be done to ascertain if this is the most appropriate license for the project. Moreover, the program may not be ready for production the moment it is published; in such a case, a disclaimer will be made warning potential users.

Appendix C

Hardware Specification

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              2
Core(s) per socket:              2
Socket(s):                       1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i7-8500Y CPU @ 1.50GHz
Stepping:                        9
CPU MHz:                         1608.000
BogoMIPS:                        3216.00
Virtualization:                  VT-x
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                        512 KiB
L3 cache:                        4 MiB
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional cache flushes, SMT
    vulnerable
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted, no microcode; SMT
    Host state unknown
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and
    seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer
    sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, IBPB conditional, IBRS_FW,
     STIBP conditional, RSB
                                  filling
Vulnerability Srbds:             Unknown: Dependent on hypervisor status
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
    pat pse36 clflush mmx f
                                 xsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc
                                     rep_good nopl xtopology
                                 cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2
                                     movbe popcnt aes xsave avx
                                 f16c rdrand hypervisor lahf_lm abm 3dnowprefetch
                                     invpcid_single pti ssbd ibrs ibpb st
                                 ibp tpr_shadow vnmi ept vpid ept_ad fsgsbase bmi1 avx2 smep
                                     bmi2 erms invpcid rdseed
                                 adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush_l1d
                                     arch_capabilities
```

Appendix D

Threat Model

D.1 Overview

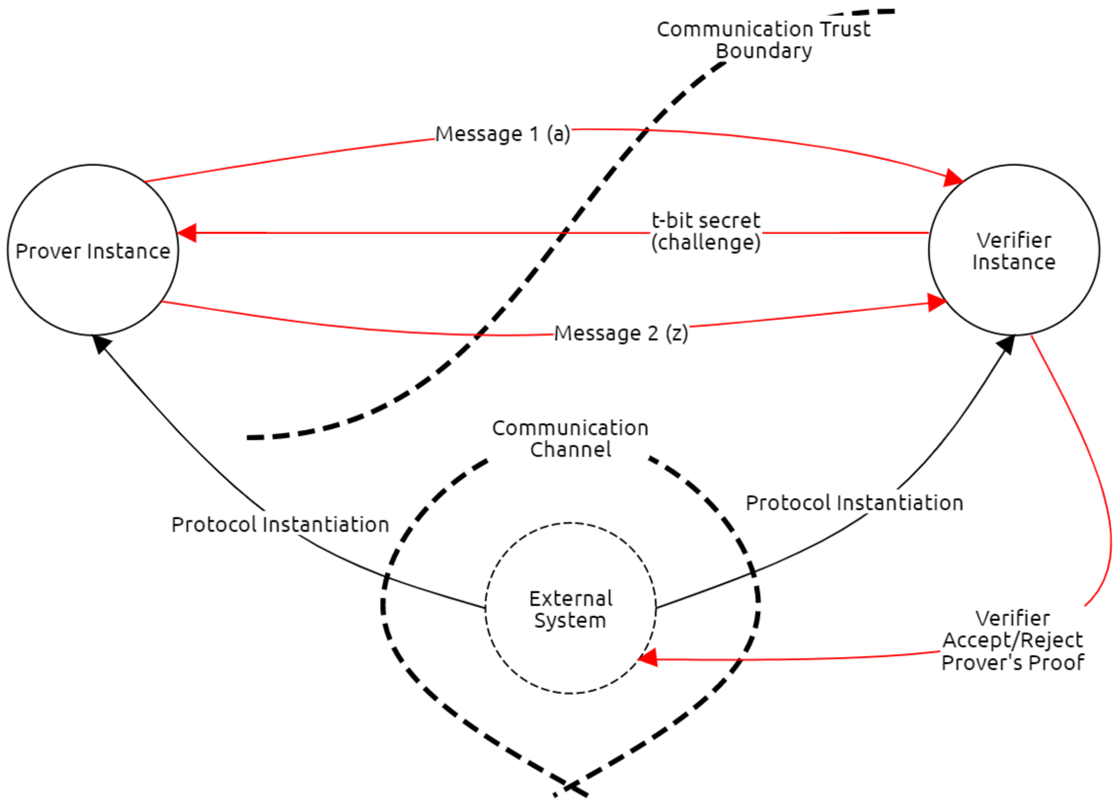


Figure D.1: Threat Model Diagram

There should be no known way for adversaries to use the protocol in a way that it is not designed to. For example, under normal circumstances, the verifier should not have rewind access to a prover, this means that we will need to model state within the protocol and ensure that it is used properly (state-machine model). This should be a target of penetration tests.