High Assurance Cryptography: Implementing Bulletproofs in Hacspec

Rasmus Kirk Jakobsen – 201907084
Anders Wibrand Larsen – 201906147
Advisor: Bas Spitters

2022-08-15

Bachelor report (15 ECTS) in Computer Science
Department of Computer Science, Aarhus University

Abstract:

Bulletproofs is a zero-knowledge range proof protocol. In this thesis we shall describe the implementation of this protocol in Hacspec, a cryptographic specification language, defined to be a subset of Rust. Hacspec allows the programs written in it to be transpiled to so-called proof assistants such as Coq, which can be used to prove various properties about the specification. We will use rigorous property based testing which will ensure that our implementation functions identically to a much more efficient Rust implementation. Thus, if a property is proven for our Hacspec specification, it is plausible that it also holds for the Rust implementation.

In section 3 we will go through the prerequisites required to understand the Bulletproofs protocol and give a short introduction to Hacspec. In section 4 we will describe the Bulletproofs protocol in detail. In section 5 we describe how the protocol was implemented in Hacspec and what testing was done. In section 6 and 7 we outline the implications of our work, including how it can be used to prove properties about the equivalent Rust implementations and how our work can aid others specifiying cryptographic protocols in Hacspec.

Acknowledgements:

We would like to express our gratitude and thanks towards all who have helped us with making this project possible.

Firstly, we would like to thank Bas Spitters, our supervisor, for his guidance and expertise on the subject, as well as giving us the idea for the project. Without him this project would not have been possible. Secondly, we thank Franziskus Kiefer for his helpfulness in answering questions about various Hacspec-related quirks, issues, as well as being swift in these responses despite his large workload. Thirdly, we want to thank Las Safin, who helped proofreading and thus make this report more bulletproof. Finally, we thank everyone who helped answer questions over the various stages of the project. Your assistance is greatly appreciated.

Contents

1 Introduction:

Privacy-enhanced cryptocurrencies use cryptographic protocols to hide sender, receiver and amount sent. Zero-knowledge proofs are a popular tool to solve some of these problems and are increasingly used today. However, these protocols can also carry vulnerabilities, potentially leading to particularly severe inflation bugs, which allows malicious actors to double-spend currency. Two of the currently most popular privacy-enhanced cryptocurrencies, Monero and Zcash, have already had such vulnerabilities [15] [14]. Both of these cryptocurrencies have marketcaps over a billion dollars, therefore it is crucial to minimize the attack surface. This gives rise to the concept of high assurance cryptography, where we want to minimize assumptions and acquire guarantees about our software. One way of doing this is by using proof assistants, such as Coq, on the implementations of these protocols, proving that necessary properties hold.

However, efficient implementations are often infeasible to implement and run in proof assistant languages. A relatively new tool in this space is Hacspec [2], a subset of Rust that functions as a specification language. It implements cryptographic primitives allowing for convenient implementation of cryptographic algorithms and supports transpiling into proof assistant languages, namely Coq and F*, such that properties can be proven about said implementations. This leads to greater security which can be transferred over to a Rust implementation using property based testing to ensure that the Hacspec and Rust implementations function identically.

The overall goal of this paper is to implement the zero-knowledge range proof protocol, Bulletproofs, created by Bünz et al. [3], in Hacspec. The protocol creates a compact zero-knowledge proof to show that one or more values lie within a range of $[0 : 2^n)$ for a provided n, without revealing any other information about the values themselves. A variant of this protocol is currently utilized in the cryptocurrency Monero. Additionally, we also wish to write our implementation in a modular fashion that allows future cryptographic protocols specified in Hacspec to utilize parts of our specification.

We will be testing our implementation against the current Rust Bulletproofs crate [10] created by the Dalek team, using property-based testing to assert, with high probability, that our implementation of the Bulletproofs protocol is equivalent to the Rust Bulletproofs crate. This will mean that using a proof assistant on our implementation will be approximately equivalent to using it on the Rust implementation.

In this paper we will briefly go over the prerequisites for understanding the bulletproofs protocol, describe the protocol itself, as well as relevant material for understanding the implementation and finally everything we implemented including discussions about the implementation hurdles encountered in the process. We end this paper by discussing possible future work that can be done on the back of our work, concluding the overall project.

All the project's Hacspec code as well as the latex to compile this report can be found on our GitHub repository [17].

2 Notation:

A table of notation used throughout this report:

a	A scalar
$\boldsymbol{a}$	A vector
A	A curve point
$\boldsymbol{A}$	A vector of curve points
a	A challenge scalar
$\mathbf{a}$	A vector of challenge scalars
$\mathbf{A}$	A vector of random curve points
$\boldsymbol{A}_{[b:c]}$	The slice $[A_b, A_{b+1}, \cdots A_{c-1}]$
$\boldsymbol{a}_{\mathbf{lo}}$	The first half of vector $\boldsymbol{a}$, ($\boldsymbol{a}_{\mathbf{lo}} = [a_1, \cdots, a_{n/2}]$)
$\boldsymbol{a}_{\mathbf{hi}}$	The second half of vector $\boldsymbol{a}$, ($\boldsymbol{a}_{\mathbf{hi}} = [a_{n/2+1}, \cdots, a_n]$)
$\boldsymbol{a^n}$	A vector of scalars which are made up of powers of a i.e. $[1, a, a^2, \cdots, a^{n-1}]$
$\boldsymbol{a} \mathbin{+\!\!+} \boldsymbol{b}$	A vector, $\boldsymbol{a}$, concatenated with another vector, $\boldsymbol{b}$
$\mathbb{A}$	A set
$\mathbb{A}^n$	A vector space of dimension n
$\mathbb{A}_n$	A set whose elements are mod n
$a(x)$	A function mapping integers mod p to integers mod p: $\mathbb{Z}_p \to \mathbb{Z}_p$
$\boldsymbol{a}(x)$	A function mapping integers mod p to vectors containing integers mod p: $\mathbb{Z}_p \to \mathbb{Z}_p^n$
$\langle \boldsymbol{a}, \boldsymbol{b} \rangle$	Dot product of $\boldsymbol{a}$ and $\boldsymbol{b}$
$\langle \boldsymbol{a}, \boldsymbol{A} \rangle$	The sum of scalar-point products of $\boldsymbol{a}$ and $\boldsymbol{A}$ ($\langle \boldsymbol{a}, \boldsymbol{A} \rangle = a_1A_1 + a_2A_2 + \cdots + a_nA_n$)

3 Prerequisites:

In this section we will be covering the necessary background knowledge to understand the Bulletproofs protocol. This includes finite field arithmetic (section 3.1), elliptic curves (section 3.2), Pedersen commitments (section 3.3) and finally the concept of zero-knowledge proofs (section 3.4).

3.1 Finite Field Arithmetic:

We start with *fields* and the operations defined within them, as the operations over elliptic curve points are built on fields. The definitions presented here are largely from [4].

Definition 3.1 (Field)**.** *A field is a set $\mathbb{F}$, along with the addition and multiplication operations. These two operations must uphold the so-called field axioms:*

- *Associativity of addition and multiplication:* $\forall a, b, c \in \mathbb{F} : a + (b + c) = (a + b) + c \wedge a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- *Commutativity of addition and multiplication:* $\forall a, b \in \mathbb{F} : a + b = b + a \wedge a \cdot b = b \cdot a$
- *Additive and multiplicative identity:* $\exists 0, 1 \in \mathbb{F} : a + 0 = a \wedge a \cdot 1 = a$
- *Additive inverses:* $\forall a \in \mathbb{F},\ \exists {-a} \in \mathbb{F} : a + (-a) = 0$
- *Multiplicative inverses:* $\forall a \neq 0 \in \mathbb{F},\ \exists a^{-1} \in \mathbb{F} : a \cdot a^{-1} = 1$
- *Distributivity over addition:* $\forall a, b, c \in \mathbb{F} : a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

Note that this also means that subtraction and division is defined, due to the existence of the additive and multiplicative inverses respectively:

$$a - b = a + (-b)$$
$$\frac{a}{b} = a \cdot b^{-1}$$

This leads us to *finite fields*:

Definition 3.2 (Finite Field)**.** *A finite field, is a field that contains a finite number of elements.*

One of the most commonly used types of finite fields, and those used throughout this report are *Prime Fields:*

Definition 3.3 (Prime Field)**.** *A prime field $\mathbb{F}_p$ is a finite field with elements $\{0, 1, \cdots, p-1\}$ where each operation is performed over integers modulo p with p being a prime number.*

Something important to note about this definition is that inverses in this kind of field are also positive whole numbers. This will also be brought up in section 5.2.

3.2 Elliptic Curves:

To start our explanation of elliptic curves we add a few definitions. Again, these definitions are mostly from [4]:

Definition 3.4 (Elliptic Curves)**.** *An elliptic curve, E, is an algebraic curve defined over a prime field, $\mathbb{F}_p$, defined by the formula:*

$$y^2 = x^3 + ax + b$$

Definition 3.5 (EC Additive Identity)**.** *Let $\mathcal{O}$ be the point where $y = \infty$. For every point, P, on E the following property holds:*

$$P + \mathcal{O} = \mathcal{O} + P = P$$

Definition 3.6 (EC Negation)**.** *Let $P = (x, y)$ be a point on E. The negation of P, called $-P$ is defined as the mirror of P over the x-axis, or $-P = (x, -y)$.*

Definition 3.7 (EC Addition)**.** *Let ℓ be a line intersecting E at three points, P, Q and $-R$. The group operation addition, for two points, is defined as:*

$$P + Q = -R$$

Definition 3.8 (EC Doubling)**.** *Let ℓ be the tangent to the point P on E, which intersects points P and $-R$. Doubling P can be defined as:*

$$2P = P + P = R$$

An important thing to note about point addition. In the case where you add points P and Q where ℓ is a tangent to P then $P + Q = -P$. This is due to the fact that the *only* case in which this can happen is when $Q = -(P + P)$, and thus you get $P + Q = P + (-(P + P)) = -P$. Another similar case is using point doubling on any point $P = (x, 0)$ on E. The tangent of P does not intersect a second point. However, note that the negation of P here is $-P = (x, -0) = (x, 0) = P$. Thus point doubling on these points is equivalent to the equation $P + (-P) = \mathcal{O}$.

From these definitions we extrapolate two more definitions:

Definition 3.9 (EC Subtraction)**.** *Let P and Q be elliptic curve points. Subtraction is defined by:*

$$P - Q = P + (-Q)$$

Definition 3.10 (EC Scalar Multiplication)**.** *Let m be a scalar and let P be a point on E. Scalar multiplication is defined by adding P to itself m times and denoted a $m \cdot P$.*

Due to the absence of scalar division it is impossible to multiply by anything other than integers as we cannot have something akin to $2.5 \cdot P = \frac{5 \cdot P}{2}$. Therefore, scalars are always integers when doing computations for elliptic curves. Multiplying by 0 will naturally yield the identity element $\mathcal{O}$ by definition. Additionally multiplying by a negative integer, $-m$, is defined as $-m \cdot P = m \cdot (-P)$.

These definitions give rise to the concept of an elliptic curve's *generator points*:

Definition 3.11 (EC Generator)**.** *A Generator for any given Elliptic curve is a point, for which all possible points can be reached using addition or scalar multiplication of this point. The set of all possible generators for a curve is denoted* $\mathbb{G}$.

3.3 Pedersen Commitments:

A Pedersen commitment is a commitment scheme originally defined by Torben Pryds Pedersen [6]. The essence of Pedersen commitments is the so-called *Additive Homomorphism* (def. 3.13). For our purposes we redefine the scheme to work with commitments to elliptic curve points rather than a value. Firstly, two generator points B and $\widetilde{B}$ on an elliptic curve are chosen. B will be the standard generator, called the base point, with $\widetilde{B}$ being the so-called *blinding point* to B. These two points are agreed upon by both the sender of the commitments and the receiver.

Definition 3.12 (Pedersen Commitment)**.** *A Pedersen commitment, to some message, a, with a randomly chosen integer, $\widetilde{a}$, is defined as:*

$$C(a, \widetilde{a}) = aB + \widetilde{a}\widetilde{B}$$

where B is a canonical generator for the curve and $\widetilde{B}$ is a curve point where no one knows q such that $\widetilde{B} = qB$.

The construction of $\widetilde{a}\widetilde{B}$ ensures *perfect hiding* as there is many possible combinations of two points that can add to any given point. However, it does not provide *perfect binding*, because multiple a's can map to multiple P's. It is important to note that the person constructing the Pedersen commitment cannot *predict* the other valid a's that maps to P, without using brute force. And by the same logic the receiver of the commitment cannot infer neither a nor $\widetilde{a}$. A crucial property of Pedersen commitments as mentioned previously is the property of *Additive Homomorphism*:

Definition 3.13 (Additive Homomorphism)**.** *For any two Pedersen commitments $C(a_1, \widetilde{a}_1), C(a_2, \widetilde{a}_2)$ we have:*

$$C(a_1, \widetilde{a}_1) + C(a_2, \widetilde{a}_2) = C(a_1 + a_2, \widetilde{a}_1 + \widetilde{a}_2)$$

This is simple to show by applying the definition of Pedersen commitments:

$$\begin{aligned} C(a_1, \widetilde{a}_1) + C(a_2, \widetilde{a}_2) &= \widetilde{a}_1\widetilde{B} + a_1B + \widetilde{a}_2\widetilde{B} + a_2B \\ &= (\widetilde{a}_1 + \widetilde{a}_2)\widetilde{B} + (a_1 + a_2)B \\ &= C(a_1 + a_2, \widetilde{a}_1 + \widetilde{a}_2) \end{aligned}$$

Which ensures a homomorphism between the sum of commitments and the commitment to the sum.

3.4 Zero-Knowledge Proofs:

A zero-knowledge proof is a cryptographic concept in which a prover, $\mathcal{P}$, will convince a verifier, $\mathcal{V}$, that a certain property holds, and $\mathcal{P}$ will do so without $\mathcal{V}$ gaining any additional information. To exemplify this we will show a simple example of a zero-knowledge proof, using the so-called Schnorr Identity Protocol [19].

The crux of the Schnorr Identity Protocol is for $\mathcal{P}$ to show that they know the secret value x. This secret value has the following relation to the public elliptic curve points X and G:

$$X = xG$$

$\mathcal{V}$ knows both X and G but not x. $\mathcal{P}$ wishes to show $\mathcal{V}$ that they know x without revealing it directly.

The protocol is rather simple. First, $\mathcal{P}$ chooses a secret random value k and commits to this value by sending K, such that $K = kG$, to $\mathcal{V}$. $\mathcal{V}$ then chooses a random challenge value e and sends it to $\mathcal{P}$. Finally, $\mathcal{P}$ sends s such that $s = k + ex$. Written out as a transcript this would be:

$$\begin{aligned} \mathcal{P} &\rightarrow \mathcal{V} : K \\ \mathcal{V} &\rightarrow \mathcal{P} : \mathrm{e} \\ \mathcal{P} &\rightarrow \mathcal{V} : s \end{aligned}$$

From here $\mathcal{V}$ can do a simple check to see if $\mathcal{P}$ truly knows x:

$$sG \overset{?}{=} K + \mathrm{e}X$$

Completeness:

For a proof to be 'complete' it means that an honest $\mathcal{P}$ will convince an honest $\mathcal{V}$ that the statement they wish to prove holds. The Schnorr Identity protocol is complete because of the following:

$$\begin{aligned} sG &= K + \mathrm{e}X \\ &= kG + \mathrm{e}(xG) \\ &= (k + \mathrm{e}x)G \\ &= sG \end{aligned}$$

Soundness:

For a proof to be sound it means that a *dishonest* $\mathcal{P}$ *cannot* convince an honest $\mathcal{V}$ that their statement is true. For the Schnorr Identity protocol it means that $\mathcal{P}$ does not know x but wishes to convince $\mathcal{V}$ that it does, and for the protocol to be sound this should not be possible. In order for $\mathcal{P}$ to convince $\mathcal{V}$ that it knows x without knowing x it must create a K and s such that $sG = K + eX$ without knowing x, which is used in the creation of s. This *could* be done if $\mathcal{P}$ had knowledge of e by constructing K as follows:

$$K = s(G - \mathrm{e}X)$$

and then choosing a random value s. If the verifier then did the check they would find that it holds. However, this is *not* a possibility because $\mathcal{P}$ must commit to a K before they are given e. There is a small probability that $\mathcal{P}$ could randomly guess e correctly, but this is miniscule, and does not nullify soundness for a zero-knowledge proof.

Zero-knowledge:

For the property of zero-knowledgeness to hold $\mathcal{V}$ must not have gained any additional information. To see this we look at the transcript and see that K does not include x in its calculations, but s does. However, s is blinded by k meaning that $\mathcal{V}$ cannot infer x without knowing k. And $\mathcal{V}$ cannot infer k from K any easier than it can infer x from X, thus this proof is also zero-knowledge.

3.5 Hacspec:

Hacspec is a subset of the programming language of Rust. Its purpose is to be a specification language for cryptographic algorithms, with the special properties that it is executable (since it is valid rust code), but also that it is defined such that it can easily be transpiled into theorem solvers, namely Coq and F*. The cost for having this property is that certain language features available in Rust are not defined in Hacspec. This makes the language simpler, but it also makes it harder to express certain concepts. However, the language is currently still in development, so some features are still being balanced or added. Most notably generics are currently on the roadmap, and these would be useful for our purposes.

External crates are not allowed in Hacspec as they cannot be automatically translated to any of the Hacspec target languages. However, Hacspec implements various cryptographic primitives, such as field elements, which allow for easier cryptographic specification. Some types and functions from the Rust standard library are not available, however some helpful wrapper types are defined in the Hacpsec standard library. `Seq<T>`, for example, implements most of the same functionality as `Vec<T>` and serves as a Hacspec counterpart to it. When defining custom types, the Hacspec standard library differentiates between so-called *secret integers* and *public integers*. Secret integers are protected from side-channel attacks, but do not implement certain operations, namely equality. We have used secret integers wherever possible, but while they are more secure, they are not always feasible.

The Hacspec repository have an array of code examples. These examples can serve as a guideline for future implementations, but can also be used as libraries. In our project we utilize the Sha-3 Hacspec example in our Merlin specification. It is our goal to eventually merge the entirety of our Hacspec code into the main repository. So far, the entire the linear algebra library (section 5.1) and approximately half of the ristretto specification (section 5.2) have been merged into Hacspec.

All code, except tests, written for this project was written to be Hacspec compliant. Our Bulletproofs specification is meant to be simply understood compared to the more obfuscated, but highly optimized Dalek implementation [10]. We will use property based testing with QuickCheck [1] where possible and convenient to test our specifications against the equivalent Rust implementations. We have also created a minimal linear algebra library, in Hacspec, with the intention of it aiding in our implementation while also creating a library to be used with other cryptographic algorithms.

4 The Bulletproofs Protocol

The ultimate goal of this project is to implement Bulletproofs in Hacspec. Bulletproofs is a zero-knowledge range proof protocol that supports aggregation of the range proofs using multiparty computation (MPC). The following subsections are based on the original Bulletproofs paper [3] as well as the implementation notes from the Dalek Bulletproofs project [11]. We will describe this protocol in detail on a theoretic level, including the inner product proof (section 4.1), the range proof (section 4.2) and the aggregation of the aforementioned range proofs (section 4.3)

4.1 Inner Product Proof:

A prover, $\mathcal{P}$, wants to prove to a verifier, $\mathcal{V}$, that he has knowledge of two vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and know their inner product $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = c$. The prover could simply send the verifier $\boldsymbol{a}, \boldsymbol{b}, c$, but this would take up $O(n)$ bandwidth and time to verify. Instead, we will use a compression technique included in the Bulletproofs paper which allows for $O(\log_2(n))$ bandwidth and time to verify.

4.1.1 Prover's Algorithm:

$\mathcal{P}$ is given values with the following definitions:

$$\begin{aligned} \boldsymbol{a}, \boldsymbol{b} &\in \mathbb{Z}_p^n \\ \boldsymbol{G}, \boldsymbol{H} &\in \mathbb{G}^n \end{aligned} \tag{1}$$

$$\begin{aligned} P &= \langle \boldsymbol{a}, \boldsymbol{G} \rangle + \langle \boldsymbol{b}, \boldsymbol{H} \rangle \\ c &= \langle \boldsymbol{a}, \boldsymbol{b} \rangle \end{aligned} \tag{2}$$

Here P is similar to a Pedersen Commitment with the crucial difference that *there is no blinding.* The blinding will be introduced later on in the range proofs section (section 4.2). Note that $\boldsymbol{G}$ and $\boldsymbol{H}$ are publically agreed upon points, available to both prover and verifier. We will introduce a combined form of P and c in the form of $P^{(k)}$, The parenthesized superscript, denotes that it's a new variable, it does *not* refer to exponentiation:

$$P^{(k)} = \langle \boldsymbol{a}, \boldsymbol{G} \rangle + \langle \boldsymbol{b}, \boldsymbol{H} \rangle + \langle \boldsymbol{a}, \boldsymbol{b} \rangle Q$$

The Q introduced here is another public point, like $\boldsymbol{G}$ and $\boldsymbol{H}$. This redefinition will be useful, as it will allow us to redefine $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{G}, \boldsymbol{H}$ ($\boldsymbol{a}^{(k)}, \boldsymbol{b}^{(k)}, \boldsymbol{G}^{(k)}, \boldsymbol{H}^{(k)}$) into a shorter format, while keeping the properties in equation 2 as invariants.

$$\begin{aligned} \boldsymbol{a}^{(k-1)} &= \boldsymbol{a}_{\mathbf{lo}}^{(\mathrm{k})} \cdot \mathrm{u}_k + \mathrm{u}_k^{-1} \cdot \boldsymbol{a}_{\mathbf{hi}}^{(\mathrm{k})} \\ \boldsymbol{b}^{(k-1)} &= \boldsymbol{b}_{\mathbf{lo}}^{(\mathrm{k})} \cdot \mathrm{u}_k^{-1} + \mathrm{u}_k \cdot \boldsymbol{b}_{\mathbf{hi}}^{(\mathrm{k})} \\ \boldsymbol{G}^{(k-1)} &= \boldsymbol{G}_{\mathbf{lo}}^{(\mathrm{k})} \cdot \mathrm{u}_k^{-1} + \mathrm{u}_k \cdot \boldsymbol{G}_{\mathbf{hi}}^{(\mathrm{k})} \\ \boldsymbol{H}^{(k-1)} &= \boldsymbol{H}_{\mathbf{lo}}^{(\mathrm{k})} \cdot \mathrm{u}_k + \mathrm{u}_k^{-1} \cdot \boldsymbol{H}_{\mathbf{hi}}^{(\mathrm{k})} \end{aligned}$$

Notice that these redefined vectors will have a length half that of the original vectors, satisfying our requirement for more compact vectors. The random variable, u_k, will be provided by $\mathcal{V}$ to ensure that it is indeed random. From these new vectors we will define a new compressed $P^{(k-1)}$ and isolate the variable u:

$$P^{(k-1)} = \langle \boldsymbol{a}^{(k-1)}, \boldsymbol{G}^{(k-1)} \rangle + \langle \boldsymbol{b}^{(k-1)}, \boldsymbol{H}^{(k-1)} \rangle + \langle \boldsymbol{a}^{(k-1)}, \boldsymbol{b}^{(k-1)} \rangle \cdot Q$$

Notice that $P^{(k-1)}$ is still defined the same way that we defined $P^{(k)}$, and that the property $c^{(k-1)} = \langle \boldsymbol{a}^{(k-1)}, \boldsymbol{b}^{(k-1)} \rangle$ still holds. If we substitute our defined variables:

$$\begin{aligned} P^{(k-1)} = \; & \langle \boldsymbol{a}_{\mathbf{lo}} \cdot u_k + u_k^{-1} \cdot \boldsymbol{a}_{\mathbf{hi}}, \quad \boldsymbol{G}_{\mathbf{lo}} \cdot u_k + u_k^{-1} \cdot \boldsymbol{G}_{\mathbf{hi}} \rangle + \\ & \langle \boldsymbol{b}_{\mathbf{lo}} \cdot u_k^{-1} + u_k \cdot \boldsymbol{b}_{\mathbf{hi}}, \quad \boldsymbol{H}_{\mathbf{lo}} \cdot u_k^{-1} + u_k \cdot \boldsymbol{H}_{\mathbf{hi}} \rangle + \\ & \langle \boldsymbol{a}_{\mathbf{lo}} \cdot u_k + u_k^{-1} \cdot \boldsymbol{a}_{\mathbf{hi}}, \quad \boldsymbol{b}_{\mathbf{lo}} \cdot u_k^{-1} + u_k \cdot \boldsymbol{b}_{\mathbf{hi}} \rangle \cdot Q \end{aligned}$$

And group the terms:

$$\begin{aligned} P^{(k-1)} = \; & \langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{G}_{\mathbf{lo}} \rangle + \langle \boldsymbol{a}_{\mathbf{hi}}, \boldsymbol{G}_{\mathbf{hi}} \rangle \quad + u_k^2 \langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{G}_{\mathbf{hi}} \rangle + u_k^{-2} \langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{G}_{\mathbf{hi}} \rangle + \\ & \langle \boldsymbol{b}_{\mathbf{lo}}, \boldsymbol{H}_{\mathbf{lo}} \rangle + \langle \boldsymbol{b}_{\mathbf{hi}}, \boldsymbol{H}_{\mathbf{hi}} \rangle \quad + u_k^2 \langle \boldsymbol{b}_{\mathbf{lo}}, \boldsymbol{H}_{\mathbf{hi}} \rangle + u_k^{-2} \langle \boldsymbol{b}_{\mathbf{lo}}, \boldsymbol{H}_{\mathbf{hi}} \rangle + \\ & (\langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{b}_{\mathbf{lo}} \rangle + \langle \boldsymbol{a}_{\mathbf{hi}}, \boldsymbol{b}_{\mathbf{hi}} \rangle) \cdot Q + (u_k^2 \langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{b}_{\mathbf{lo}} \rangle + u_k^{-2} \langle \boldsymbol{a}_{\mathbf{hi}}, \boldsymbol{b}_{\mathbf{hi}} \rangle) \cdot Q \end{aligned}$$

We can see that this compressed $P^{(k-1)}$ contains $c = (\langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{b}_{\mathbf{lo}} \rangle + \langle \boldsymbol{a}_{\mathbf{hi}}, \boldsymbol{b}_{\mathbf{hi}} \rangle)$.

We can simplify further by defining variables L_k and R_k:

$$\begin{aligned}
P^{(k-1)} &= P_k + u_k^2 \cdot L_k + u_k^{-2} \cdot R_k \\
\textbf{where:} & \\
L_k &= \langle \boldsymbol{a}_{\mathbf{lo}}^{(\mathrm{k})}, \boldsymbol{G}_{\mathbf{hi}}^{(\mathrm{k})} \rangle + \langle \boldsymbol{b}_{\mathbf{hi}}^{(\mathrm{k})}, \boldsymbol{H}_{\mathbf{lo}}^{(\mathrm{k})} \rangle + \langle \boldsymbol{a}_{\mathbf{lo}}^{(\mathrm{k})}, \boldsymbol{b}_{\mathbf{hi}}^{(\mathrm{k})} \rangle \cdot Q \\
R_k &= \langle \boldsymbol{a}_{\mathbf{hi}}^{(\mathrm{k})}, \boldsymbol{G}_{\mathbf{lo}}^{(\mathrm{k})} \rangle + \langle \boldsymbol{b}_{\mathbf{lo}}^{(\mathrm{k})}, \boldsymbol{H}_{\mathbf{hi}}^{(\mathrm{k})} \rangle + \langle \boldsymbol{a}_{\mathbf{hi}}^{(\mathrm{k})}, \boldsymbol{b}_{\mathbf{lo}}^{(\mathrm{k})} \rangle \cdot Q
\end{aligned}$$

So, $\mathcal{P}$ calculates L_k, R_k and sends it to $\mathcal{V}$. Afterwards, $\mathcal{V}$ responds with u_k. This is repeated $k = \log_2(n)$ times, until we get the final P, $P^{(0)}$:

$$P^{(0)} = a_0^{(0)} G_0^{(0)} + b_0^{(0)} H_0^{(0)} + a_0^{(0)} b_0^{(0)} Q$$
$$P^{(0)} = P^{(k)} + \sum_{i=1}^{k} (L_i u_i^2 + u_i^{-2} R_i)$$

Finally, $\mathcal{P}$ sends $(a^{(0)}, b^{(0)})$ to $\mathcal{V}$.

The entirety of the dialogue between $\mathcal{P}$ and $\mathcal{V}$, the so-called *transcript*, can be summarized as the following:

$$\begin{aligned}
&\mathcal{P} \to \mathcal{V} : L_k, R_k \\
&\mathcal{V} \to \mathcal{P} : u_k \\
&\mathcal{P} \to \mathcal{V} : L_{k-1}, R_{k-1} \\
&\mathcal{V} \to \mathcal{P} : u_{k-1} \\
&\qquad \vdots \\
&\mathcal{P} \to \mathcal{V} : L_1, R_1 \\
&\mathcal{V} \to \mathcal{P} : u_1 \\
&\mathcal{P} \to \mathcal{V} : a^{(0)}, b^{(0)}
\end{aligned}$$

To make the inner product proof *non-interactive*, the Fiat-Shamir heuristic (4.4) can be used.

4.1.2 Verifier's Algorithm

$\mathcal{V}$ has knowledge of the following values:

$$\begin{aligned}
a^{(0)}, b^{(0)} &\in \mathbb{Z}_p \\
\boldsymbol{L}, \boldsymbol{R} &\in \mathbb{G}_p^k \\
\boldsymbol{G}, \boldsymbol{H} &\in \mathbb{G}^n \\
P^{(k)}, Q &\in \mathbb{G} \\
\mathbf{u} &\in \mathbb{Z}^k
\end{aligned}$$

We need a way to get the final $G^{(0)}$ and $H^{(0)}$, from $\boldsymbol{G}$ and $\boldsymbol{H}$, let's start with the former. We recall our reduced version of $\boldsymbol{G}$:

$$\boldsymbol{G}^{(j-1)} = \boldsymbol{G}_{\mathbf{lo}}^{(j)} \mathrm{u}_j^{-1} + \boldsymbol{G}_{\mathbf{hi}}^{(j)} \mathrm{u}_j^{1} \tag{3}$$

We want to define a vector $\boldsymbol{s}$ such that $\langle \boldsymbol{s}, \boldsymbol{G} \rangle = G^{(0)}$. From equation 3, we conclude that each s_i is defined as:

$$\begin{aligned}
&s_i = \mathrm{u}_k^{b(i,k)} \cdots \mathrm{u}_1^{b(i,1)} \\
&\textbf{where:} \\
&b(i,j) = \begin{cases} \text{-1} & g(i,j) = \top \\ 1 & g(i,j) = \bot \end{cases} \\
&g(i,j) = \begin{cases} \top & \text{if } (i \bmod 2^j) < 2^{j-1} \\ \bot & \text{if } (i \bmod 2^j) \geq 2^{j-1} \end{cases}
\end{aligned}$$

We define $b(i,j)$[1] to be -1 if G_i appears in the left half of $\boldsymbol{G}^{(j)}$, and 1 if G_i appears in the right half of $\boldsymbol{G}^{(j)}$. Therefore, G_i appears in the right side of $\boldsymbol{G}^{(j)}$ if $(i \bmod 2^j) < 2^{j-1}$.

We make a similar argument for $H^{(0)}$. We want to define a vector $\boldsymbol{s'}$ such that $\langle \boldsymbol{s'}, \boldsymbol{H} \rangle = H^{(0)}$:

$$\boldsymbol{H}^{(k-1)} = \boldsymbol{H}_{\textbf{lo}}^{(k)} \mathrm{u}_j^1 + \boldsymbol{H}_{\textbf{hi}}^{(j)} \mathrm{u}_j^{-1} \tag{4}$$

To construct $\boldsymbol{s'}$:

$$\begin{aligned}
&s'_i = \mathrm{u}_k^{b'(i,k)} \cdots \mathrm{u}_1^{b'(i,1)} \\
&\textbf{where:} \\
&b'(i,j) = \begin{cases} \text{-1} & \neg g(i,j) = \top \\ 1 & \neg g(i,j) = \bot \end{cases}
\end{aligned}$$

But this is indeed:

$$\boldsymbol{s'} = \frac{1}{s_1}, \frac{1}{s_2}, \cdots \frac{1}{s_n}$$

Leading us to our desired $G^{(0)}, H^{(0)}$:

$$\begin{aligned}
G^{(0)} &= \langle \boldsymbol{s}, \boldsymbol{G} \rangle \\
H^{(0)} &= \langle \boldsymbol{s'}, \boldsymbol{H} \rangle
\end{aligned}$$

Now, $\mathcal{V}$ can check if $P^{(k)} \stackrel{?}{=} P^{\mathcal{V}}$, if the former statement is true, then the $\mathcal{V}$ concludes that $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = c$:

$$\begin{aligned}
P^{(k)} &\stackrel{?}{=} P^{\mathcal{V}} \\
&\stackrel{?}{=} a^{(0)} G^{(0)} + b^{(0)} H^{(0)} + a^{(0)} b^{(0)} Q - \sum_{i=1}^{k} (L_i \mathrm{u}_i^2 + \mathrm{u}_i^{-2} R_i) \\
&\stackrel{?}{=} \langle a\boldsymbol{s}, \boldsymbol{G} \rangle + \langle b\boldsymbol{s'}, \boldsymbol{H} \rangle + abQ - \sum_{i=1}^{k} (L_i \mathrm{u}_i^2 + \mathrm{u}_i^{-2} R_i)
\end{aligned}$$

[1]Note that in the code we zero-index so $j_{\text{code}} = j + 1$.

4.2 Range Proofs:

A range proof is a zero-knowledge proof which seeks to prove that for some value v, v lies within the range of $[0, 2^n)$ without revealing v or any additional information about v, thus zero-knowledge. We wish to express this property of v using a single inner product to be able to use an inner product proof as described in section 4.1.

4.2.1 Prover's Algorithm

To start, the prover, $\mathcal{P}$, is given the following values:

$$\begin{aligned} v &\in \mathbb{Z}_p \\ n &\in \mathbb{Z} \\ B, \widetilde{B} &\in \mathbb{G} \\ \boldsymbol{G}, \boldsymbol{H} &\in \mathbb{G}^n \end{aligned} \tag{5}$$

$\mathcal{P}$ wishes to convince the verifier, $\mathcal{V}$, that the value, v lies within the range $[0, 2^n)$.

The first step towards this is for $\mathcal{P}$ to commit to v using a Pedersen commitment V, and sending this to the $\mathcal{V}$, as well committing to the desired range n, which is not blinded. With the Pedersen-commitment having base-point B and blinding point $\widetilde{B}$:

$$V = vB + \widetilde{v}\widetilde{B}$$

If we let $\boldsymbol{a}$ be v expressed in bits then we know the following:

$$\langle \boldsymbol{a}, \boldsymbol{2^n} \rangle = v$$

Additionally, as we also need a guarantee that $\boldsymbol{a}$ consists entirely of bits i.e. $\boldsymbol{a} \in \{0,1\}^n$. This can be proven by the following property:

$$\boldsymbol{a} \circ (\boldsymbol{a} - \boldsymbol{1}) = \boldsymbol{0}$$

This property will only hold if $\boldsymbol{a}$ has entries that are either 0 or 1. Due to needing to commit to both $\boldsymbol{a}$ and $\boldsymbol{a} - \boldsymbol{1}$, they will henceforth be referred to as $\boldsymbol{a}_\mathbf{L}$ and $\boldsymbol{a}_\mathbf{R}$ respectively. Additionally, we add another property to show the relationship between these two which gives us the following properties:

$$\begin{aligned} \langle \boldsymbol{a}_\mathbf{L}, \boldsymbol{2^n} \rangle &= v \\ \boldsymbol{a}_\mathbf{L} \circ \boldsymbol{a}_\mathbf{R} &= \boldsymbol{0} \\ (\boldsymbol{a}_\mathbf{L} - \boldsymbol{1}) - \boldsymbol{a}_\mathbf{R} &= \boldsymbol{0} \end{aligned}$$

We wish to combine these properties into a single inner product, which $\mathcal{P}$ will then prove to $\mathcal{V}$. To do this, we observe that $\boldsymbol{a} = \boldsymbol{0} \iff \forall y \in \mathbb{Z} : \langle \boldsymbol{a}, \boldsymbol{y^n} \rangle = \boldsymbol{0}$. So $\mathcal{P}$ lets $\mathcal{V}$ chose a random challenge scalar y and using y we get a new set of properties:

$$\begin{aligned} \langle \boldsymbol{a}_\mathbf{L}, \boldsymbol{2^n} \rangle &= v \\ \langle \boldsymbol{a}_\mathbf{L} - \boldsymbol{1} - \boldsymbol{a}_\mathbf{R}, \mathbf{y^n} \rangle &= 0 \\ \langle \boldsymbol{a}_\mathbf{L}, \boldsymbol{a}_\mathbf{R} \circ \mathbf{y^n} \rangle &= 0 \end{aligned}$$

Next, $\mathcal{P}$ lets $\mathcal{V}$ chose another random challenge scalar z, and using this we can combine the three properties above to the following single statement:

$$\mathrm{z}^2 v = \mathrm{z}^2\langle \boldsymbol{a}_{\mathbf{L}}, \mathbf{2}^{\boldsymbol{n}}\rangle + \mathrm{z}\langle \boldsymbol{a}_{\mathbf{L}} - \mathbf{1} - \boldsymbol{a}_{\mathbf{R}}, \mathbf{y}^{\boldsymbol{n}}\rangle + \langle \boldsymbol{a}_{\mathbf{L}}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle$$

With this, we have condensed the needed properties into a single equation. We wish to rewrite this as a single inner product where $\boldsymbol{a}_{\mathbf{L}}$ appears only in the first argument of the inner product and $\boldsymbol{a}_{\mathbf{R}}$ appears only in the second argument and all non-secret terms are factored out.

First, we can rewrite the middle term:

$$\begin{aligned}
\mathrm{z}^2 v &= \mathrm{z}^2\langle \boldsymbol{a}_{\mathbf{L}}, \mathbf{2}^{\boldsymbol{n}}\rangle + \mathrm{z}\langle \boldsymbol{a}_{\mathbf{L}}, \mathbf{y}^{\boldsymbol{n}}\rangle - \mathrm{z}\langle \mathbf{1}, \mathbf{y}^{\boldsymbol{n}}\rangle - \mathrm{z}\langle \boldsymbol{a}_{\mathbf{R}}, \mathbf{y}^{\boldsymbol{n}}\rangle + \langle \boldsymbol{a}_{\mathbf{L}}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle \\
\mathrm{z}^2 v + \mathrm{z}\langle \mathbf{1}, \mathbf{y}^{\boldsymbol{n}}\rangle &= \mathrm{z}^2\langle \boldsymbol{a}_{\mathbf{L}}, \mathbf{2}^{\boldsymbol{n}}\rangle + \mathrm{z}\langle \boldsymbol{a}_{\mathbf{L}}, \mathbf{y}^{\boldsymbol{n}}\rangle - \mathrm{z}\langle \mathbf{1}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle + \langle \boldsymbol{a}_{\mathbf{L}}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle \\
\mathrm{z}^2 v + \mathrm{z}\langle \mathbf{1}, \mathbf{y}^{\boldsymbol{n}}\rangle &= \langle \boldsymbol{a}_{\mathbf{L}}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}}\rangle + \langle \boldsymbol{a}_{\mathbf{L}}, \mathrm{z}\mathbf{y}^{\boldsymbol{n}}\rangle + \langle -\mathrm{z}\mathbf{1}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle + \langle \boldsymbol{a}_{\mathbf{L}}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle \\
\mathrm{z}^2 v + \mathrm{z}\langle \mathbf{1}, \mathbf{y}^{\boldsymbol{n}}\rangle &= \langle \boldsymbol{a}_{\mathbf{L}}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}} + \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle + \langle -\mathrm{z}\mathbf{1}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle
\end{aligned}$$

For the final step towards combining all the conditions into a single inner product we add $\langle -\mathrm{z}\mathbf{1}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}}\rangle$ to both sides and simplify again:

$$\begin{aligned}
\mathrm{z}^2 v + \mathrm{z}\langle \mathbf{1}, \mathbf{y}^{\boldsymbol{n}}\rangle + \langle -\mathrm{z}\mathbf{1}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}}\rangle &= \langle \boldsymbol{a}_{\mathbf{L}}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}} + \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle + \langle -\mathrm{z}\mathbf{1}, \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle + \langle -\mathrm{z}\mathbf{1}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}}\rangle \\
\mathrm{z}^2 v + (\mathrm{z} - \mathrm{z}^2)\langle \mathbf{1}, \mathbf{y}^{\boldsymbol{n}}\rangle - \mathrm{z}^3\langle \mathbf{1}, \mathbf{2}^{\boldsymbol{n}}\rangle &= \langle \boldsymbol{a}_{\mathbf{L}}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}} + \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle + \langle -\mathrm{z}\mathbf{1}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}} + \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle \\
\mathrm{z}^2 v + (\mathrm{z} - \mathrm{z}^2)\langle \mathbf{1}, \mathbf{y}^{\boldsymbol{n}}\rangle - \mathrm{z}^3\langle \mathbf{1}, \mathbf{2}^{\boldsymbol{n}}\rangle &= \langle \boldsymbol{a}_{\mathbf{L}} - \mathrm{z}\mathbf{1}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + \mathrm{z}\mathbf{y}^{\boldsymbol{n}} + \boldsymbol{a}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}\rangle
\end{aligned}$$

We define a function $\delta(y, z) = (z - z^2)\langle \mathbf{1}, \boldsymbol{y}^{\boldsymbol{n}}\rangle - z^3\langle \mathbf{1}, \mathbf{2}^{\boldsymbol{n}}\rangle$ and finish simplifying:

$$\mathrm{z}^2 v + \delta(\mathrm{y}, \mathrm{z}) = \langle \boldsymbol{a}_{\mathbf{L}} - \mathrm{z}\mathbf{1}, \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + (\boldsymbol{a}_{\mathbf{R}} + \mathrm{z}\mathbf{1}) \circ \mathbf{y}^{\boldsymbol{n}}\rangle$$

From here on we will refer to the first argument of the inner product as $\boldsymbol{l}_{\mathbf{0}}$ and the second argument as $\boldsymbol{r}_{\mathbf{0}}$:

$$\mathrm{z}^2 v + \delta(\mathrm{y}, \mathrm{z}) = \langle \boldsymbol{l}_{\mathbf{0}}, \boldsymbol{r}_{\mathbf{0}}\rangle$$

If the goal was simply to construct a single inner product which would prove that v lies in $[0, 2^n)$, then $\mathcal{P}$ could send these values to $\mathcal{V}$ and be done. However, we wish for the proof to be zero-knowledge and therefore this inner product needs to be blinded. To do this, $\mathcal{P}$ constructs a polynomial $t(x)$. This polynomial will be constructed using $\boldsymbol{l}_{\mathbf{0}}$ and $\boldsymbol{r}_{\mathbf{0}}$ in a way that blinds them, while still allowing $\mathcal{P}$ to prove to $\mathcal{V}$ that v lies in $[0, 2^n)$.

First, $\mathcal{P}$ chooses random blinding factors $\widetilde{\boldsymbol{s}}_{\mathbf{L}}$ and $\widetilde{\boldsymbol{s}}_{\mathbf{R}}$ such that $\widetilde{\boldsymbol{s}}_{\mathbf{L}}, \widetilde{\boldsymbol{s}}_{\mathbf{R}} \in \mathbb{Z}_p^n$. Using these $\mathcal{P}$ can construct $\boldsymbol{l}_{\mathbf{1}} = \widetilde{\boldsymbol{s}}_{\mathbf{L}}$ and $\boldsymbol{r}_{\mathbf{1}} = \widetilde{\boldsymbol{s}}_{\mathbf{R}} \circ \mathbf{y}^{\boldsymbol{n}}$, and with these, $\mathcal{P}$ constructs two vector-function $\boldsymbol{l}(x)$ and $\boldsymbol{r}(x)$ in the following way:

$$\begin{aligned}
\boldsymbol{l}(x) &= \boldsymbol{l}_{\mathbf{0}} + \boldsymbol{l}_{\mathbf{1}} x \\
&= (\boldsymbol{a}_{\mathbf{L}} - \mathrm{z}\mathbf{1}) + \widetilde{\boldsymbol{s}}_{\mathbf{L}} x \\
&= (\boldsymbol{a}_{\mathbf{L}} + \widetilde{\boldsymbol{s}}_{\mathbf{L}} x) - \mathrm{z}\mathbf{1}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{r}(x) &= \boldsymbol{r}_{\mathbf{0}} + \boldsymbol{r}_{\mathbf{1}} x \\
&= (\mathrm{z}^2\mathbf{2}^{\boldsymbol{n}} + (\boldsymbol{a}_{\mathbf{R}} + \mathrm{z}\mathbf{1}) \circ \mathbf{y}^{\boldsymbol{n}}) + \widetilde{\boldsymbol{s}}_{\mathbf{R}} x \circ \mathbf{y}^{\boldsymbol{n}} \\
&= ((\boldsymbol{a}_{\mathbf{R}} + \widetilde{\boldsymbol{s}}_{\mathbf{R}} x) + \mathrm{z}\mathbf{1}) \circ \mathbf{y}^{\boldsymbol{n}} + \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}}
\end{aligned}$$

Now, $\mathcal{P}$ can construct $t(x)$ as follows:

$$t(x) = \langle \boldsymbol{l}(x), \boldsymbol{r}(x) \rangle$$

We group x to get:

$$\begin{aligned}
&t(x) = t_0 + t_1 x + t_2 x^2 \\
&\textbf{where:} \\
&t_0 = \langle \boldsymbol{l_0}, \boldsymbol{r_0} \rangle \\
&t_2 = \langle \boldsymbol{l_1}, \boldsymbol{r_1} \rangle \\
&t_1 = \langle \boldsymbol{l_0} + \boldsymbol{l_1}, \boldsymbol{r_0} + \boldsymbol{r_1} \rangle - t_0 - t_2
\end{aligned}$$

Now, $\mathcal{P}$ wishes to prove to $\mathcal{V}$ that t_0 and $t(x)$ are correctly constructed. So, $t_0 = \mathrm{z}^2 v + \delta(\mathrm{y}, \mathrm{z})$, and $t(x) = \langle \boldsymbol{l}(x), \boldsymbol{r}(x) \rangle$ such that, $\boldsymbol{l}(x) = \boldsymbol{l_0} + \boldsymbol{l_1} x$, and, $\boldsymbol{r}(x) = \boldsymbol{r_0} + \boldsymbol{r_1} x$.

First, $\mathcal{P}$ wants to show that t_0 is correctly constructed. $\mathcal{P}$ starts by making commitments to each of the coefficients of $t(x)$. Here we make note of the fact that $\mathcal{P}$ has already committed to t_0. Specifically, commitment V, made at the very start, is also a commitment to t_0 as per the construction of t_0. $\mathcal{P}$ makes commitments to $T_1 = C(t_1, \widetilde{t}_1)$ and $T_2 = C(t_2, \widetilde{t}_2)$. After committing these values to $\mathcal{V}$, $\mathcal{V}$ sends back a challenge scalar x. We know from the definitions of V, T_1 and T_2 that these commitments relate in the following manner:

$$\begin{aligned}
t(\mathrm{x})B &= \mathrm{z}^2 vB + \delta(\mathrm{y},\mathrm{z})B + \mathrm{x}t_1 B + \mathrm{x}^2 t_2 B \\
\widetilde{t}(\mathrm{x})\widetilde{B} &= \mathrm{z}^2 \widetilde{v}\widetilde{B} + 0\widetilde{B} \qquad\quad + \mathrm{x}\widetilde{t}_1 \widetilde{B} + \mathrm{x}^2 \widetilde{t}_2 \widetilde{B} \\
t(\mathrm{x})B + \widetilde{t}(\mathrm{x})\widetilde{B} &= \mathrm{z}^2 V \quad + \delta(\mathrm{y},\mathrm{z})B + \mathrm{x}T_1 \quad + \mathrm{x}^2 T_2
\end{aligned}$$

The sum of the two top-most coefficients in each 'column' is also equal to the bottom coefficient of that column. To convince $\mathcal{V}$, $\mathcal{P}$ sends $t(\mathrm{x})B$ and $\widetilde{t}(\mathrm{x})\widetilde{B}$, evaluated at x to $\mathcal{V}$. This is enough to convince $\mathcal{V}$ that $t_0 = \mathrm{z}^2 v + \delta(\mathrm{y}, \mathrm{z})$, as seen in section 4.2.2.

Next, $\mathcal{P}$ wishes to convince $\mathcal{V}$ that $\boldsymbol{l}(x)$ and $\boldsymbol{r}(x)$ are both constructed correctly and that $t(x) = \langle \boldsymbol{l}(x), \boldsymbol{r}(x) \rangle$. To achieve this $\mathcal{P}$ will use a similar construction as seen above. We want $\mathcal{P}$ to make commitments that allows $\mathcal{V}$ to relate $\boldsymbol{l}(x)$ and $\boldsymbol{r}(x)$ to $\boldsymbol{a_L}, \widetilde{\boldsymbol{s}}_{\mathbf{L}}, \boldsymbol{a_R}$ and $\widetilde{\boldsymbol{s}}_{\mathbf{R}}$. Recall the definition of $\boldsymbol{r}(x)$:

$$\boldsymbol{r}(x) = \mathrm{z}^2 \mathbf{2}^{\boldsymbol{n}} + ((\boldsymbol{a_R} + \widetilde{\boldsymbol{s}}_{\mathbf{R}} x) + \mathrm{z}\mathbf{1}) \circ \mathbf{y}^{\boldsymbol{n}}$$

We want $\mathcal{P}$ to make commitments to $\mathbf{y}^{\boldsymbol{n}} \circ \boldsymbol{a_R}$ and $\mathbf{y}^{\boldsymbol{n}} \circ \widetilde{\boldsymbol{s}}_{\mathbf{R}}$. This is a problem as these commitments need to be made and sent to $\mathcal{V}$ *before* $\mathcal{P}$ gets the challenge y from $\mathcal{V}$. To fix this issue $\mathcal{P}$ will make a special commitment to $\boldsymbol{a_L}$ and $\boldsymbol{a_R}$ as follows:

$$C(\boldsymbol{a_L}, \boldsymbol{a_R}, \widetilde{a}) = \langle \boldsymbol{a_L}, \boldsymbol{G} \rangle + \langle \boldsymbol{a_R}, \boldsymbol{H} \rangle + \widetilde{a}\widetilde{B}$$

This same construction is used for the commitment to $C(\widetilde{\boldsymbol{s}}_{\mathbf{L}}, \widetilde{\boldsymbol{s}}_{\mathbf{R}}, \widetilde{s})$. Here we notice that:

$$\begin{aligned}
C(\boldsymbol{a_L}, \boldsymbol{a_R}, \widetilde{a}) &= \langle \boldsymbol{a_L}, \boldsymbol{G} \rangle + \langle \boldsymbol{a_R}, \boldsymbol{H} \rangle + \widetilde{a}\widetilde{B} \\
&= \langle \boldsymbol{a_L}, \boldsymbol{G} \rangle + \langle \mathbf{y}^{\boldsymbol{n}} \circ \boldsymbol{a_R}, \mathbf{y}^{-\mathbf{n}} \circ \boldsymbol{H} \rangle + \widetilde{a}\widetilde{B}
\end{aligned}$$

We define $\boldsymbol{H'} = \mathbf{y}^{-\mathbf{n}} \circ \boldsymbol{H}$, $A = C(\boldsymbol{a}_{\mathbf{L}}, \boldsymbol{a}_{\mathbf{R}}, \widetilde{a})$ and $S = C(\widetilde{\boldsymbol{s}}_{\mathbf{L}}, \widetilde{\boldsymbol{s}}_{\mathbf{R}}, \widetilde{s})$ and can now construct our argument similarly to what was done in the previous segment:

$$\begin{aligned}
\langle \boldsymbol{l}(\mathrm{x}), \boldsymbol{G}\rangle &= \langle \boldsymbol{a}_{\mathbf{L}}, \boldsymbol{G}\rangle &&+ \mathrm{x}\langle \widetilde{\boldsymbol{s}}_{\mathbf{L}}, \boldsymbol{G}\rangle &&+ \langle -\mathrm{z}\mathbf{1}, \boldsymbol{G}\rangle \\
\langle \boldsymbol{r}(\mathrm{x}), \boldsymbol{H'}\rangle &= \langle \boldsymbol{a}_{\mathbf{R}}, \boldsymbol{H}\rangle &&+ \mathrm{x}\langle \widetilde{\boldsymbol{s}}_{\mathbf{R}}, \boldsymbol{H}\rangle &&+ \langle \mathrm{z}\mathbf{y}^{\boldsymbol{n}} + \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}}, \boldsymbol{H'}\rangle \\
\widetilde{e}\widetilde{B} &= \widetilde{a}\widetilde{B} &&+ \mathrm{x}\widetilde{s}\widetilde{B} &&+ 0\widetilde{B} \\
\langle \boldsymbol{l}(\mathrm{x}), \boldsymbol{G}\rangle + \langle \boldsymbol{r}(\mathrm{x}), \boldsymbol{H'}\rangle + \widetilde{e}\widetilde{B} &= A &&+ \mathrm{x}S &&+ (\langle \mathrm{z}\mathbf{y}^{\boldsymbol{n}} + \mathrm{z}^2\mathbf{2}^{\boldsymbol{n}}, \boldsymbol{H'}\rangle - \mathrm{z}\langle \mathbf{1}, \boldsymbol{G}\rangle)
\end{aligned}$$

From here, using the same argument as above, $\mathcal{P}$ commits $\widetilde{e}$ to $\mathcal{V}$. This will be enough to convince $\mathcal{V}$ that $\boldsymbol{l}(x)$ and $\boldsymbol{r}(x)$ are correctly constructed and that $t(x) = \langle \boldsymbol{l}(x), \boldsymbol{r}(x)\rangle$ and together with the previous commitments to $\mathcal{V}$, $\mathcal{P}$ will be able to successfully convince $\mathcal{V}$ that v lies in the range $[0, 2^n)$.

In summary the transcript between $\mathcal{P}$ and $\mathcal{V}$, will end up being the following:

$$\begin{aligned}
&\mathcal{P} \to \mathcal{V} : V, n, A, S \\
&\mathcal{V} \to \mathcal{P} : \mathrm{y}, \mathrm{z} \\
&\mathcal{P} \to \mathcal{V} : T_1, T_2 \\
&\mathcal{V} \to \mathcal{P} : \mathrm{x} \\
&\mathcal{P} \to \mathcal{V} : t(\mathrm{x})B, \widetilde{t}(\mathrm{x})\widetilde{B}, \widetilde{e} \qquad (t(x) \text{ evaluated with challenge x})
\end{aligned}$$

Just as with the inner product the Fiat-Shamir heuristic (section 4.4) can be used to make this process non-interactive.

4.2.2 Verifier's Algorithm:

$\mathcal{V}$ has access to the following values:

$$\begin{aligned}
\mathrm{y}, \mathrm{z}, \mathrm{x}, \widetilde{e} &\in \mathbb{Z}_p \\
\boldsymbol{L}, \boldsymbol{R} &\in \mathbb{G}_p^k \\
\boldsymbol{G}, \boldsymbol{H}, \boldsymbol{H'} &\in \mathbb{G}^n \\
V, A, B, T_0, T_1, B, \widetilde{B} &\in \mathbb{G} \\
t(\mathrm{x})B, \widetilde{t}(\mathrm{x})\widetilde{B} &\in \mathbb{G} \qquad \text{(evaluated at x)}
\end{aligned} \tag{6}$$

$\mathcal{V}$ needs to make a check for each of the two properties that $\mathcal{P}$ wants to prove.

To check the first property $\mathcal{V}$ checks if the following equation holds:

$$t(\mathrm{x})B + \widetilde{t}(\mathrm{x})\widetilde{B} \overset{?}{=} \mathrm{z}^2 + \delta(\mathrm{y}, \mathrm{z})B + \mathrm{x}T_1 + \mathrm{x}^2 T_2$$

And if it does then that will convince $\mathcal{V}$ that $t(x) = \mathrm{z}^2 v + \delta(\mathrm{y}, \mathrm{z}) + t_1 x + t_2 \mathrm{x}^2$ due to the column-sum argument made in section 4.2.1.

Similarly to become convinced of the second property $\mathcal{V}$ checks if this equation holds:

$$\begin{aligned} P &= -\widetilde{e}\widetilde{B} + A + \mathrm{x}S + \langle \mathrm{z}\mathbf{y^n} + \mathrm{z}^2\boldsymbol{2^n}, \boldsymbol{H'}\rangle \qquad - \mathrm{z}\langle \mathbf{1}, \boldsymbol{G}\rangle \\ &= -\widetilde{e}\widetilde{B} + A + \mathrm{x}S + \langle \mathrm{z}\mathbf{y^n} + \mathrm{z}^2\boldsymbol{2^n} \circ \mathbf{y^{-n}}, \boldsymbol{H}\rangle - \mathrm{z}\langle \mathbf{1}, \boldsymbol{G}\rangle \end{aligned}$$

From the column-sum argument we can deduce that $P = \langle \boldsymbol{l}(x), \boldsymbol{G}\rangle + \langle \boldsymbol{r}(x), \boldsymbol{H'}\rangle$ assuming an honest $\mathcal{P}$. Thus, $\mathcal{V}$ can use $t(x)$ and P as inputs into the inner product protocol to prove that $t(x) = \langle \boldsymbol{l}(x), \boldsymbol{r}(x)\rangle$. And if this proof is verified then $\mathcal{V}$ is now convinced that $v \in [0, 2^n)$, thus the range proof is complete.

Additionally, by looking at the transcript one can deduce that the only way for $\mathcal{P}$ to construct A and V such that they can convince $\mathcal{V}$ that v lies in the range, even if it does not, would be if they can correctly guess *all* three challenge scalars y, z and x. This happens with a probability so small it is entirely negligible, and thus the proof is also sound.

But the proof being complete and sound does not necessarily mean it is zero-knowledge. If we look at all the values available to $\mathcal{V}$ we can see that it has only two of these values have any direct relation to v, these being V and $t(\mathrm{x})B$. It is trivial to see that $\mathcal{V}$ cannot learn anything about v from V without knowledge of $\widetilde{v}$ which $\mathcal{V}$ does not know. If we look at the definition of $t(\mathrm{x})B$ we can see that it DOES use v in the calculation directly. However, the definition of $t(x)$ uses t_2, which is blinded using $\widetilde{\boldsymbol{s}}_{\mathbf{L}}$ and $\widetilde{\boldsymbol{s}}_{\mathbf{R}}$ neither of which $\mathcal{V}$ has any way to access. Additionally, t_1 also uses the definition of t_2 and thus is *also* cannot be inferred. Thus, there is no amount of calculations that $\mathcal{V}$ can do to figure out anything about v and thus the proof is also zero-knowledge.

4.3 Range Proofs Aggregation:

An aggregated range-proof is a range proof which proves that a series of values, $\boldsymbol{v} \in \mathbb{Z}^m$, lie within a certain range:

$$[0, 2^n) \tag{7}$$

for some integer n. This is done by aggregating m range-proofs for each individual value v_j into a single proof which, if verified, proves that each v_j given lies within the range $[0, 2^n)$, while giving away no more information about the values. This is done by using a multi-party computation (MPC) protocol where each party $\mathcal{P}_{(j)}$, takes on the role of an independent prover $\mathcal{P}$, meaning it has a value v_j it needs to prove lies within the range (7). Each $\mathcal{P}_{(j)}$ will converse with the so-called dealer $\mathcal{D}$, who takes on the role of $\mathcal{V}$, thus being responsible for collecting the various commitments from each $\mathcal{P}_{(j)}$ and generating challenges which will be used by all $\mathcal{P}_{(j)}$'s. At the very end, $\mathcal{D}$ will then collect all the individual range proofs for each value and aggregate them. It is important to note that all parties are in agreement over a specific n.

The steps used to do this are the same steps used in the construction of a range proof with a few additional steps which aggregate the proofs into a single range proof.

Each $P_{(j)}$ is also assigned a 'position' j. All parties will individually commit the needed values to the dealer in order to convince them that their value lie within the range, with one key difference. Each party will also use an offset to ensure the order matters to the dealer, preventing the parties from working together to 'cheat'. This offset is defined as such for the jth party:

$$\mathrm{z}_{(j)} = \mathrm{z}^j \cdot \mathbf{y}^{\boldsymbol{n}}_{(j)}$$

where:

$$\mathbf{y}^{\boldsymbol{n}}_{(j)} = \mathbf{y}^{\boldsymbol{n \cdot m}}_{[j \cdot n : (j+1) \cdot n]}$$

The evaluation point x is the same for all parties. The choice of y, z and x is done where it would ordinarily be done, but only once $\mathcal{D}$ has received the needed values from *all* $\mathcal{P}_{(j)}$. Once again, this can also be done using the Fiat-Shamir heuristic (4.4).

After each party has created $t(x)_{(j)} = \langle \boldsymbol{l}(x)_{(j)}, \boldsymbol{r}(x)_{(j)} \rangle$ it would be sufficient to perform the proving steps from 4.2 for each party. However, this will require $\mathcal{D}$ to make m checks to see if each v_j lies within the range (7). However, $\mathcal{D}$ can do a few more computations to create a singular proof which requires a single check to see that *all* v_j lie within the range (7).

For $\mathcal{P}_{(j)}$ to prove that its polynomial $t(x)_{(j)}$ is correct means proving that $t_{0,(j)}$ is correct as well as proving that $\boldsymbol{l}(x)_{(j)}$ and $\boldsymbol{r}(x)_{(j)}$ are created correctly and that $t(x)_{(j)} = \langle \boldsymbol{l}(x)_{(j)}, \boldsymbol{r}(x)_{(j)} \rangle$.

Each $\mathcal{P}_{(j)}$ sends their $T_{1,(j)}$ and $T_{2,(j)}$, to the dealer, which can then compute:

$$T_1 = \sum_{j=0}^{m-1} T_{1,(j)}$$
$$T_2 = \sum_{j=0}^{m-1} T_{2,(j)}$$
$$\delta(y,z) = \sum_{j=0}^{m-1} \delta_{(j)}(y,z)$$

From here $\mathcal{D}$ sends back the challenge x and $\mathcal{P}_{(j)}$ will return $t(\mathrm{x})_{(j)}B$ and $\widetilde{t}(\mathrm{x})_{(j)}\widetilde{B}$. $\mathcal{D}$ can then sum these together as well:

$$t(x)B = \sum_{j=0}^{m-1} t_j(x)B$$
$$\widetilde{t}(x)\widetilde{B} = \sum_{j=0}^{m-1} \widetilde{t}_j(x)\widetilde{B}$$

And finally convince themselves that for all $t_{0,(j)}$ are correct by performing the following check:

$$t(\mathrm{x})B + \widetilde{t}(\mathrm{x})\widetilde{B} \stackrel{?}{=} \mathrm{z}^2 \sum_{j=0}^{m-1} \mathrm{z}_{(\mathrm{j})} \cdot V_{(j)} + \delta(\mathrm{y},\mathrm{z})B + \mathrm{x}T_1 + \mathrm{x}^2 T_2$$

we know that $z_{(j)} = z^j$ we can make a small rewrite:

$$t(\mathrm{x})B + \widetilde{t}(\mathrm{x})\widetilde{B} \stackrel{?}{=} \sum_{j=0}^{m-1} \mathrm{z}^{j+2} \cdot V_{(j)} + \delta(\mathrm{y},\mathrm{z})B + \mathrm{x}T_1 + \mathrm{x}^2 T_2 \tag{8}$$

And this check will then convince $\mathcal{D}$ that all $t_{0,(j)}$ are correct.

Just as with a singular range proof each $\mathcal{P}_{(j)}$ also wishes to prove that each $\boldsymbol{l}(x)_{(j)}$ and $\boldsymbol{r}(x)_{(j)}$ are constructed correctly. This is done similarly to proving that $t_{0,(j)}$ are all correct. Proving this property for the jth party would be proving the following:

$$\langle \boldsymbol{l}(\mathrm{x})_{(j)}, \boldsymbol{G}_{(j)} \rangle + \langle \boldsymbol{r}(\mathrm{x})_{(j)}, \boldsymbol{H'}_{(j)} \rangle \stackrel{?}{=} -\widetilde{e}_{(j)}\widetilde{B} + A_{(j)} + \mathrm{x}S_{(j)} + (\langle \mathrm{z}\mathbf{y}^{\boldsymbol{n}}_{(j)} + \mathrm{z}^2\mathrm{z}_{(j)}\mathbf{2}^{\boldsymbol{n}}, H' \rangle_{(j)} - \mathrm{z}\langle \mathbf{1}, \boldsymbol{G}_{(j)} \rangle)$$

$\mathcal{P}_{(j)}$ will send their $\langle \boldsymbol{l}(\mathrm{x})_{(j)}, \boldsymbol{G}_{(j)} \rangle$, $\langle \boldsymbol{r}(\mathrm{x})_{(j)}, \boldsymbol{H'}_{(j)} \rangle$ and $\widetilde{e}_{(j)}$ which $\mathcal{D}$ can then combine in the following manner:

$$\begin{aligned}
\boldsymbol{l}(x) &= \boldsymbol{l}(x)_{(0)} \mathbin{+\!\!+} \boldsymbol{l}(x)_{(1)} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \boldsymbol{l}(x)_{(m-1)} \\
\boldsymbol{r}(x) &= \boldsymbol{r}(x)_{(0)} \mathbin{+\!\!+} \boldsymbol{r}(x)_{(1)} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \boldsymbol{r}(x)_{(m-1)} \\
\boldsymbol{G} &= \boldsymbol{G}_{(0)} \mathbin{+\!\!+} \boldsymbol{G}_{(1)} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \boldsymbol{G}_{(m-1)} \\
\boldsymbol{H'} &= \boldsymbol{H'}_{(0)} \mathbin{+\!\!+} \boldsymbol{H'}_{(1)} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \boldsymbol{H'}_{(m-1)}
\end{aligned}$$

Additionally, $\mathcal{D}$ can construct the following:

$$\begin{aligned}
\mathbf{y}^{\boldsymbol{n}}_{(j)} &= \mathbf{y}^{\boldsymbol{n \cdot m}}_{[j \cdot n:(j+1) \cdot n]} \\
\mathrm{z}_{(j)} &= \mathrm{z}^{j} \\
\widetilde{e} &= \sum_{j=0}^{m-1} \widetilde{e}_{(j)} \\
A &= \sum_{j=0}^{m-1} A_{(j)} \\
S &= \sum_{j=0}^{m-1} S_{(j)}
\end{aligned}$$

Note that $\mathcal{D}$ has already received $A_{(j)}$ and $S_{(j)}$ from the first step of the transcript. $\mathcal{D}$ can then use the following check to convince itself that all $\boldsymbol{l}(x)_{(j)}$ and $\boldsymbol{r}(x)_{(j)}$ are correctly constructed:

$$\boldsymbol{l}(\mathrm{x})_{(j)} + \boldsymbol{r}(\mathrm{x})_{(j)} \stackrel{?}{=} -\widetilde{e}\widetilde{B} + A + \mathrm{x}S - \mathrm{z}\langle \mathbf{1}, \boldsymbol{G} \rangle + \mathrm{z}\langle \mathbf{y}^{\boldsymbol{n \cdot m}}, \boldsymbol{H'} \rangle + \sum_{j=0}^{m-1} \langle \mathrm{z}^{\mathrm{j}+2} \cdot \mathbf{2}^{\boldsymbol{n}}, \boldsymbol{H'}_{[j \cdot n:(j+1) \cdot n]} \rangle \tag{9}$$

And if this holds then $\mathcal{D}$ is convinced that $\boldsymbol{l}(x)_{(j)}$ and $\boldsymbol{r}(x)_{(j)}$ are correctly constructed and that $t(x)_{(j)} = \langle \boldsymbol{l}(x)_{(j)}, \boldsymbol{r}(x)_{(j)} \rangle$. After checking that equation 8 and 9 hold, $\mathcal{D}$ is now convinced that all parties' $v_{(j)}$ are within the range (equation 7) and thus the proof is complete. Additionally, using the same arguments as in section 4.2.2 for each $\mathcal{P}_{(j)}$, this is also sound and zero-knowledge.

This protocol in transcript form would be the following:

$$\begin{aligned}
&\mathcal{P}_{(j)} \rightarrow \mathcal{D} &&: V_{(j)}, n, A_{(j)}, S_{(j)} \\
&\mathcal{D} \quad \rightarrow \mathcal{P}_{(j)} &&: \mathrm{y}, \mathrm{z}, j \\
&\mathcal{P}_{(j)} \rightarrow \mathcal{V} &&: T_{1,(j)}, T_{2,(j)} \\
&\mathcal{D} \quad \rightarrow \mathcal{P}_{(j)} &&: \mathrm{x} \\
&\mathcal{P}_{(j)} \rightarrow \mathcal{V} &&: t(\mathrm{x})_{(j)}B, \widetilde{t}(\mathrm{x})_{(j)}\widetilde{B}, \widetilde{e}
\end{aligned}$$

With $t(x)_{(j)}$ and $\widetilde{t}(x)_{(j)}$ both evaluated with challenge x.

4.4 Fiat-Shamir Heuristic

The Fiat-Shamir heuristic [18] is a cryptographic concept, which allows $\mathcal{P}$ to compute challenges without the need to consult $\mathcal{V}$ first. This is done by defining a hash function $\mathbf{H}()$ which can hash any number of values into a single relatively unpredictable value. When $\mathcal{V}$ would be needed to provide a challenge, instead, all public values that $\mathcal{V}$ would have access to are hashed using this hash function and the resulting value is used as the challenge. This way $\mathcal{P}$ can send all values it must generate at once, and $\mathcal{V}$ can reconstruct the hashes which an honest $\mathcal{P}$ would have calculated and assert correctness. To show this we will once again look at the Schnorr Identity Protocol [19], using it to show how the Fiat-Shamir heuristic can be applied. We will use the same notation we used in section 3.4.

$\mathcal{P}$ wishes to show $\mathcal{V}$ that they know the secret value x such that $X = xG$ for the public curve-points X and G. Ordinarily $\mathcal{P}$ would send a commitment K, to a randomly chosen k, to $\mathcal{V}$, who would then send back a challenge e. However, with the Fiat-Shamir heuristic, $\mathcal{P}$ will instead compute:

$$e = \mathbf{H}(X, P, K)$$

From here $\mathcal{P}$ can compute $s = k + ex$ and send K and s to $\mathcal{V}$.

Then $\mathcal{V}$ can compute $e = \mathbf{H}(X, P, K)$ and at this point the rest of the protocol functions the same as it does in section 3.4. The transcript for this proof would thus go from:

$$\begin{gathered}\mathcal{P} \to \mathcal{V} : K \\ \mathcal{V} \to \mathcal{P} : e \\ \mathcal{P} \to \mathcal{V} : s\end{gathered}$$

To the much shorter:

$$\mathcal{P} \to \mathcal{V} : K, s$$

Crucially allowing $\mathcal{P}$ to construct the entire proof without interacting with $\mathcal{V}$. The proof is still sound as $\mathcal{P}$ cannot get e without committing to a K first.

Our Fiat-Shamir heuristic for the Bulletproofs algorithm requires an *overseer* for the parties who can gather all their commitments to create the challenges. In essence, when a challenge is needed all parties will send their commitments to this local overseer who can then generate the challenges using the Fiat-Shamir heuristic. Specifically $\mathrm{y} = \mathbf{H}(\boldsymbol{V}, \boldsymbol{A}, \boldsymbol{S})$, $\mathrm{z} = \mathbf{H}(\boldsymbol{V}, \boldsymbol{A}, \boldsymbol{S}, \mathrm{y})$ and $\mathrm{x} = \mathbf{H}(\boldsymbol{V}, \boldsymbol{A}, \boldsymbol{S}, \mathrm{y}, \mathrm{z}, \boldsymbol{T_1}, \boldsymbol{T_2})$. This will result in the following final transcript:

$$\mathcal{P}_{(j)} \to \mathcal{D} : V_{(j)}, n, A_{(j)}, S_{(j)}, j, T_{1,(j)}, T_{2,(j)}, t_{(j)}(\mathrm{x})B, \widetilde{t}_{(j)}(\mathrm{x})\widetilde{B}, \widetilde{e}$$

Worth noting is that a recently disclosed vulnerability, the Frozen Heart vulnerability [13], describes how in the Fiat-Shamir heuristic suggested in the original Bulletproofs paper [3], a malicious $\mathcal{P}$ could successfully convince $\mathcal{V}$ that v is in range when it is *not*. This is because the Fiat-Shamir heuristic suggested did not include V in their hash function which allowed a malicious $\mathcal{P}$ to cheat.

This could be done by committing an A which was constructed from a v which lies in the range, but the Fiat-Shamir heuristic did not require the use of V in its hashing, therefore this V could then be forged from a v outside of the desired range while still successfully convincing $\mathcal{V}$ that this v lies inside of the range. This is not a problem for the Dalek implementation as they include V in the transcript, therefore our implementation also does not have this vulnerability. The paper has since been updated, but any implementation that has followed the old paper's Fiat-Shamir specification could still be vulnerable.

5 Our Contributions:

5.1 Linear Algebra Library:

It was decided that the specification should, of course, consist of what we need, but should also include some general linear algebra functions that could be used by other specifications. The implemented operations included matrix instantiation, slicing, transposition, scalar multiplication, addition, subtraction, Hadamard product and matrix multiplication. These operations were tested against the Rust Nalgebra crate [5] to ensure that they had equivalent functionality.

Since we were not able to find any general linear algebra specification, we simply implemented the desired functionality, consulting [8] when necessary. All functions were only defined over matrices, because any vector operation can also be performed as a matrix operation, if the vector is modelled as a matrix with one dimension set to one. This simplified our library somewhat. These matrices consist of a pair of numbers, representing the dimension, and a list of scalars, representing the matrix elements:

```rust
pub type DimType = usize;
pub type Scalar = i128;
pub type Dims = (DimType, DimType);
pub type Matrix = (Dims, Seq<Scalar>);
pub type MatRes = Result<Matrix, u8>;
pub type ScalRes = Result<Scalar, u8>;
```

We model matrices as a `Seq<Scalar>` instead of a `Seq<Seq<Scalar>>` since we had problems with the Hacspec type checker when double indexing (`list[i][j]`), and because it is more efficient. Since this part was intended to be a library we wanted to generalize over the `Scalar` type. We wanted the implementation to be generalizable over any type that can be defined using Hacspec macros at the very least. For example the following Scalar:

```rust
public_nat_mod!(
  type_name: Scalar,
  type_of_canvas: ScalarCanvas,
  bit_size_of_field: 256,
  modulo_value: "1000000000000000000000000000000014def9dea2f79cd65812631a5cf5d3ed"
);
```

This turned out to be a problem though, as Hacspec does not support generics. Luckily, generics is on the Hacspec developement roadmap and after discussion with the Hacspec team, it was decided that we would generalize our library with generics, so the Hacspec developers can use it as an example of how generics might be needed. Take the following matrix initialization function:

```rust
pub fn new(rows: DimType, cols: DimType, seq: Seq<Scalar>) -> MatRes {
...
```

It could be rewritten as:

```rust
pub fn new<T>(rows: DimType, cols: DimType, seq: Seq<T>) -> MatRes<T>
where
  T: hacspec_lib::Integer, {
...
```

With the following type aliases used instead:

```rust
type DimType = usize;
type Dims = (DimType, DimType);
type MatRes<T> = Result<Matrix<T>, u8>;
type ScalRes<T> = Result<T, u8>;
```

Which would allow anyone using the library to use matrices with their own defined Hacspec types. Notice the `hacspec_lib::Integer`, this trait is implemented by all integers, including Hacspec customly defined integers.

Testing:

We utilized QuickCheck to do property based testing, in order to ensure that our specification had the same functionality as Nalgebra. All tests followed the same standard pattern, where we create a helper function that does the following: Construct two random matrices of random dimensions, make sure no dimension is set to 0, run equivalent operations on them and finally assert that the operations had the same result. This was done for every function implemented. A simple example is the `zeros()` function:

```
#[test]
fn test_nalg_zeros() {
  fn helper(n: u8, m: u8) -> TestResult {
    let n = n as usize; let m = m as usize;

    if n * m == 0 {
        return TestResult::discard();
    }

    let hac = zeros(n, m); let ext = DMatrix::zeros(n, m);

    TestResult::from_bool(assert_matrices(hac, ext))
  }
  quickcheck(helper as fn(u8, u8) -> TestResult);
}
```

Here, the only operation that is performed is the construction of the vectors. The `assert_matrices` function checks each element in the two vectors and asserts that they are all equal. We ran 100000 equivalence tests, taking 4.63 seconds in total. We also have simple unit tests that check our implementation against manually derived results.

5.2 Implementing Ristretto:

For the Ristretto Curve25519 implementation we used the Ristretto25519 IETF-standard [7] and the "Twisted Edwards Curves Revisited" paper [12] which was used as a guide for our implementation. This specification was very helpful, as it clearly defined the types and functionalities needed to implement Ristretto securely.

Ristretto is, in essence, a way to form a prime-order subgroup of a non-prime-order Edwards curve which has a co-factor of 4 or 8. The subgroup formed under Ristretto is constructed in such a way that it eliminates this co-factor, which is useful to avoid additional checks and also avoids the risk of leaving in exploitable vulnerabilities which utilize the co-factor. One such vulnerability was found that affects all CryptoNote-based cryptocurrencies, most notably Monero [15]. Ristretto eliminates all of these checks and potential vulnerabilities by automatically mapping all points on the curve directly into the subset of the curve in the operations themselves. Additionally, Ristretto defines equality such that equivalent representations of the same point are considered equal. In the same vein, the encoding function for points encodes equivalent points to the same encoding and the decoding function decodes those points to the same encoded point.

For our implementation of Bulletproofs, seeing as we needed to test it against an existing implementation, we made use of the same elliptic curve. The elliptic curve in question is Curve25519, which is widely used in encryption as is the case here. Curve25519, is defined by the following formula:

$$y^2 = x^3 + 48662x^2 + x$$

This curve is defined over the prime field $p = 2^{255} - 19$ and base point defined at $x = 9$ with the positive y value. This results in a subgroup of order $2^{252} + 27742317777372353535851$, which has a co-factor of 8, thus meeting the criteria for using Ristretto to eliminate this co-factor.

Each internal representation of a Ristretto point is an Edwards point composed of four field elements $(X : Y : Z : T)$. Field-elements, as defined by the standard are values modulo p, with p being the prime field for Curve25519, $p = 2^{255} - 19$. This was achieved using the `public_nat_mod!()` macro which defines finite fields over a given modulo. Conveniently it accepts hex values as modulo values, which allowed us to easily make the modulo value $2^{255} - 19$:

```
public_nat_mod!(
  type_name: FieldElement,
  type_of_canvas: FieldCanvas,
  bit_size_of_field: 256,
  modulo_value: "7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffed"
);
```

Additionally, the standard specified a series of constants. These constants were too large to be converted from fix-sized integers and Hacspec did not allow for the use `from_hex()` method. As such, after converting each of these constants into their corresponding hex-values manually, we built them as byte sequences for which we used the `from_byte_seq_be()` function, which created the correct constants. This worked as intended but inevitably reduced readability of the code. An example of such a constant definition can be seen below:

```
fn P() -> FieldElement {
  FieldElement::from_byte_seq_be(&byte_seq!(
    0x7fu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8,
    0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8,
    0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8,
    0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xffu8, 0xedu8
  ))
}
```

While it is impossible for a legally encoded point to have its decoding fail, the decoding function has several checks that ensure that the input given is legal. The reason for this is that decoding is a public function and as such there is no guarantee that the inputted byte-strings should decode to legal points. To this end the standard defines a list of properties that the inputted byte-string must satisfy in order for the decoding to be canonical and thus give a proper point as a result.

Divisions over fields did not function correctly in Hacspec, since regular integer division was performed instead of the desired $a/b = a \cdot b^{-1}$. This was luckily not a problem for our implementation since the result of both types of division are the same in our case.

While the IETF-standard was used for the implementation of nearly every operation implemented, the scalar multiplication was not specified, therefore we used an algorithm described in [4]. This proved to work as intended, passing the tests we set up. Due to the need for comparing points, field elements had to be implemented using public integers rather than the more secure secret integers. We also ran into an issue with how we had to use the field elements when we had to perform a check in the `decode()` function:

```
pub fn decode(u: RistrettoPointEncoded) -> DecodeResult {
  let mut ret = DecodeResult::Err(DECODING_ERROR);

  let s = FieldElement::from_byte_seq_le(u);

  if !geq_p(u.to_le_bytes()) && !is_negative(s) {
    [...]
  }
}

[...]

fn geq_p(x: Seq<U8>) -> bool {
  let p_seq = byte_seq!([....]);
  let mut res = true;

  for index in 0..p_seq.len() {
    let x_index = x[index].declassify();
    let p_index = p_seq[index].declassify();
    if x_index != p_index {
      res = x_index > p_index;
    }
  }
  res
}
```

This check involves checking to see if the input byte sequence, if interpreted as a field element, is larger than p. This is naturally not possible to do directly, since all field elements in our implementation are integers mod p. As such, it is done in this manner to circumvent this issue.

Another issue we encountered was in the implementation of the one-way map, where we must mod the 255 least significant bits of both halves of the split 512 bits. However, the two halves of our ByteStrings are each 256 bits, so the most significant bit must be set to 0, therefore we mod the most significant byte by 128:

```rust
pub fn one_way_map(b: ByteString) -> RistrettoPoint {
  let mut r0_bytes = b.slice(0, 32).declassify();
  let mut r1_bytes = b.slice(32, 32).declassify();

  // The specification states:
  // Set r0 to the low 255 bits of b[ 0..32], taken mod p
  // Set r1 to the low 255 bits of b[32..64], taken mod p
  // Note the low 255 bits. NOT 256 bits! This is why we mod the most significant byte
  r0_bytes[31] = r0_bytes[31] % 128u8;
  r1_bytes[31] = r1_bytes[31] % 128u8;

  let r0 = FieldElement::from_public_byte_seq_le(r0_bytes);
  let r1 = FieldElement::from_public_byte_seq_le(r1_bytes);

[...]
```

This interaction was tested thoroughly to ensure it functions as we intended it to.

Testing:

When testing our implementation we made two series of tests. One was unit-tests, defined by the IETF-standard, which were designed to catch edge-cases in the various functions. One such test can be seen below, testing if various one-way-maps all map to the same point:

```rust
#[test]
fn unit_test_same_point_map() {
  let input_hexs = Seq::<&str>::from_vec(vec![
    "edffff [...] 000000",
    "edffff [...] ffffff",
    "000000 [...] ffff7f",
    "000000 [...] 000080",
  ]);
  let result_hex = "30428 [...] cb8e5ef0517f";
  let result_point = decode(RistrettoPointEncoded::from_hex(result_hex)).unwrap();

  input_hexs
    .iter()
    .for_each(|x| assert!(equals(result_point, one_way_map(ByteString::from_hex(x)))))
}
```

There were four of these tests in total, testing the one-way-map, encoding, and both a positive and a negative test for decoding. In the negative test, a series of points must fail to decode to ensure our implementation works as intended.

The second series of tests check if our implementation functions equivalently to the Dalek implementation of Ristretto using randomized inputs. This was done by making a test for every public function that went through the following phases: Generate random, but equivalent, inputs for both implementations, run the same operations on those inputs and finally compare the results. Below is an example of one of these tests, testing that both point negation functions work identically:

```rust
#[test]
fn test_dalek_point_negation() {
  fn helper(v: Vec<u8>) -> TestResult {
    if v.len() < 64 {
      return TestResult::discard();
    }

    let (hac_pnt, dal_pnt) = create_points(v);

    let hac_neg = neg(hac_pnt);
    let dal_neg = dal_pnt.neg();

    TestResult::from_bool(cmp_points(hac_neg, dal_neg))
  }
  quickcheck(100, helper as fn(Vec<u8>) -> TestResult)
}
```

The `create_points()` helper function uses the one-way-map to generate random points, this one-way-map is also tested on its independently. We ran each function 100,000 times each with random input, except for the `mul()` function which was only run 10,000 times due to it being considerably slower. In total, these tests took 23,236.58 seconds to complete. The relatively few amount of tests for point multiplication, means it can be considered a weak-point, as there is greater risk of undiscovered edge cases. However, we test edge cases using unit tests, so this should not be a major concern.

5.3 Implementing Merlin:

Merlin [9] is, for the most part, a wrapper over the Strobe protocol [16]. It performs the Fiat-Shamir heuristic by keeping a running hash of the transcript. Importantly, it does this in such a way that algorithms implementing it will appear to interact with a verifier, while remaining non-interactive. It achieves this by only allowing the algorithm to commit or receive challenge scalars from the transcript. This allows for simpler code that is less error-prone than if the Fiat-Shamir heuristic were done manually with a hash function.

The merlin library implements four functions: `new()` used for transcript instantiation, `append_message()` used for appending a byte sequence to the transcript, `append_U64()` for appending a 64-bit secret integer to the transcript and finally `challenge_bytes()` for getting challenge bytes that can be converted to whatever challenge type is needed.

As these functions simply call a few functions from the Strobe module, most of the work consisted of implementing the Strobe subset used in Merlin. There were no great difficulties in translating most functions, except that the `Strobe` type could not be passed as a mutable borrow, as in the Dalek implentation. Therefore, in our implementation, we return a strobe object for each function instead. Since all operations needed were supported by the secret integer Hacspec type, we naturally used this type over the less secure public integers.

Merlin uses Keccakf1600 for its hashing. Fortunately, this cryptographic function was already defined in the SHA-3 Hacspec library. We ran into issues however, since Keccakf1600 is defined for 64-bit integers and strobe operates on bytes. Therefore, we had to transmute the `State` object, from a list of bytes into a shorter list of 64-bit integers. In the Dalek Merlin implementation this transformation is simple:

```
fn transmute_state(st: &mut AlignedKeccakState) -> &mut [u64; 25] {
  unsafe { &mut *(st as *mut AlignedKeccakState as *mut [u64; 25]) }
}
```

But in our implementation we instead do the following:

```
// Turns a stateU8 into a StateU64
fn transmute_state_to_u64(state: StateU8) -> StateU64 {
  let mut new_state = StateU64::new();

  for i in 0..new_state.len() {
    let mut word = U64Word::new();
    for j in 0..word.len() {
      word[j] = state[i * 8 + j];
    }
    new_state[i] = U64_from_le_bytes(word);
  }

  new_state
}
```

```rust
// Turns a stateU64 into a StateU8
fn transmute_state_to_u8(state: StateU64) -> StateU8 {
  let mut new_state = StateU8::new();

  for i in 0..state.len() {
    let bytes = state[i].to_le_bytes();
    for j in 0..bytes.len() {
      new_state[i * 8 + j] = bytes[j]
    }
  }

  new_state
}
```

Although this method is significantly slower, it's not a big problem as the overall runtime of this library is quite fast compared to other libraries, namely the Ristretto library. It is therefore not a detrimental bottleneck.

Testing:

For testing the library, we created a single test that runs the `append_message()`, `append_u64()` and `challenge_bytes()` public functions for both implementations, with random inputs, and asserts that the challenge bytes returned are equivalent between implementations. These functions are sufficient to test equivalence as they utilize all other functions implemented. The entire test can be seen below:

```rust
// Tests all merlin functions against the external merlin counterparts.
#[test]
fn merl_test_all_funcs() {
  fn helper(msg: Vec<u8>, n: u64, buf_size: u8) -> TestResult {
    let mut ret = true;
    let mut t1 = new(b("str"));
    let mut t2 = merlin::Transcript::new(b"str");
    let mut buf = Vec::<u8>::new();
    buf.resize(buf_size as usize, 0);

    t1 = append_message(t1, b("msg"), to_seq(msg.clone()));
    t1 = append_U64(t1, b("n"), U64::classify(n));

    t2.append_message(b"msg", &msg);
    t2.append_u64(b"n", n);

    let (_, data) = challenge_bytes(t1, b("challenge_label"), to_seq(buf.clone()));
    t2.challenge_bytes(b"challenge_label", &mut buf);

    if data.len() != buf.len() {
      ret = false
    }
    for i in 0..data.len() {
      if data[i].declassify() != buf[i] {
        ret = false;
      }
    }

    TestResult::from_bool(ret)
  }
  quickcheck(helper as fn(Vec<u8>, u64, u8) -> TestResult);
}
```

We ran 100,000 of these tests with randomized `msg`'s, `n`'s and `buf_size`'s, which took 119.81 seconds.

5.4 Implementing Inner Product Proof:

We'll start by presenting the prover and verifier's algorithms as implemented in pseudocode. Note that we index at 1:

Algorithm 1: IPP: The Prover's Algorithm

```
   Input: $\boldsymbol{a} \in \mathbb{Z}_p^n, \boldsymbol{b} \in \mathbb{Z}_p^n, Q \in \mathbb{G}, \boldsymbol{G} \in \mathbb{G}^n, \boldsymbol{H} \in \mathbb{G}^n$
 1 $n :=$ $\boldsymbol{a}$.len()
 2 assert($n$.is_power_of_two())
 3 assert($n =$ $\boldsymbol{a}$.len() $=$ $\boldsymbol{b}$.len() $=$ $\boldsymbol{G}$.len() $=$ $\boldsymbol{H}$.len())
 4 transcript.append($n$)
 5 for $i \in [0 : \log_2(n)]$ do
 6     $L = \langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{G}_{\mathbf{hi}} \rangle + \langle \boldsymbol{b}_{\mathbf{hi}}, \boldsymbol{H}_{\mathbf{lo}} \rangle + \langle \boldsymbol{a}_{\mathbf{lo}}, \boldsymbol{b}_{\mathbf{hi}} \rangle$
 7     $R = \langle \boldsymbol{a}_{\mathbf{hi}}, \boldsymbol{G}_{\mathbf{lo}} \rangle + \langle \boldsymbol{b}_{\mathbf{lo}}, \boldsymbol{H}_{\mathbf{hi}} \rangle + \langle \boldsymbol{a}_{\mathbf{hi}}, \boldsymbol{b}_{\mathbf{lo}} \rangle$
 8
 9     transcript.append($L$)
10     transcript.append($R$)
11     $u :=$ transcript.challenge_scalar()
12
13     $\boldsymbol{a} := \boldsymbol{a}_{\mathbf{lo}} \cdot u + u^{-1} \cdot \boldsymbol{a}_{\mathbf{hi}}$
14     $\boldsymbol{b} := \boldsymbol{b}_{\mathbf{lo}} \cdot u^{-1} + u \cdot \boldsymbol{b}_{\mathbf{hi}}$
15     $\boldsymbol{G} := \boldsymbol{G}_{\mathbf{lo}} \cdot u + u^{-1} \cdot \boldsymbol{G}_{\mathbf{hi}}$
16     $\boldsymbol{H} := \boldsymbol{H}_{\mathbf{lo}} \cdot u^{-1} + u \cdot \boldsymbol{H}_{\mathbf{hi}}$
17 end for
18 return $(a_1, b_1, \boldsymbol{L}, \boldsymbol{R})$
```

Algorithm 2: IPP: The Verifier's Algorithm

```
   Input: $a \in \mathbb{Z}_p, b \in \mathbb{Z}_p, n \in \mathbb{Z}, P' \in \mathbb{G}, Q \in \mathbb{G}, \boldsymbol{G} \in \mathbb{G}^n, \boldsymbol{H} \in \mathbb{G}^n$
 1 $k := \log_2(n)$
 2 assert($k < 32 \land n = 2^k$)
 3 transcript.append($n$)
 4 for $i \in [1 : k+1]$ do
 5     transcript.append($L_i$)
 6     transcript.append($R_i$)
 7     $u_i :=$ transcript.challenge_scalar()
 8 end for
 9 for $i \in [1 : n+1]$ do
10     $s_i := u_k^{b(i,k)} \cdots u_1^{b(i,1)}$
11     $s'_i := s_i^{-1}$
12 end for
13 Result $:= P' \overset{?}{=} \langle a\boldsymbol{s}, \boldsymbol{G} \rangle + \langle b\boldsymbol{s'}, \boldsymbol{H} \rangle + abQ - \sum_{i=1}^{k}(L_i u_i^2 + u_i^{-2} R_i)$
14 return Result
```

We tried to find a way to incorporate the linear algebra library into our implementation, but this turned out to be cumbersome. Lines 13-16 are run in a for-loop. Although less efficient, it would be more clear to have $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{G}, \boldsymbol{H}$, calculated using built-in linear algebra functions. If we take the actual implementation:

```
for i in 0..n {
  a_L[i] = a_L[i] * u + u_inv * a_R[i];
  b_L[i] = b_L[i] * u_inv + u * b_R[i];
  G_L[i] = add(mul(u_inv, G_L[i]), mul(u, G_R[i]));
  H_L[i] = add(mul(u, H_L[i]), mul(u_inv, H_R[i]));
}
```

We could rewrite this with vector operations, but this would not necessarily make the code more readable.

Take for example:

```
a_L = v_add(v_scale(u, a_L), v_scale(u_inv, a_R));
b_L = v_add(v_scale(u_inv, b_L), v_scale(u, b_R));
G_L = v_p_add(v_p_mul(u_inv, G_L), v_p_mul(u, G_R));
H_L = v_p_add(v_p_mul(u, H_L), v_p_mul(u_inv, H_R));
```

Where `v_` indicates a vector operation and `p_` indicates a point operation. Having this many nested functions negatively impacts readability, but supposing that we could define the multiplicative and additive operators for types in Hacspec, as we would be able to in Rust this would simplify to:

```
a_L = a_L * u     + u_inv * a_R;
b_L = b_L * u_inv + u     * b_R;
G_L = G_L * u_inv + u     * G_R;
H_L = H_L * u     + u_inv * H_R;
```

This is indeed more intuitive and closer to the math, but it is not possible in Hacspec. While Hacspec supports limited structs, it does *not* support the `impl` keyword required for the example shown above, and its implementation is not currently on the development roadmap for Hacspec, and may not ever be incorporated. Even if this were solved by introducing `impl` in some capacity, we would still have to deal with the fact that the element-wise multiplicative operation in linear algebra is defined over two vectors of the *same* type. This could be solved by extending the hypothetical linear algebra matrix struct, with a method that can multiply a vector of points with a vector of scalars.

In our implementation, we also have the inputs `G_factors` and `H_factors`, which are merely factors applied to $\boldsymbol{G}$ and $\boldsymbol{H}$ respectively. We have these as inputs because the Dalek implementation also has them. The reason they do this is that they have optimizations which make the code faster than if they multiply the factors on before running the function. We chose for our implementation to take them as an input as well, so that the two functions take the same arguments, however we apply the factors before running the loop as it simplifies and shortens the code. The performance increase resulting from this, while noticeable, was not significant compared to other bottlenecks, namely Ristretto.

Another aspect of this library worth noting is the file structure:

```
inner-product-proof
├─ src
│  ├─ transcript
│  │  └─ errors.rs
│  ├─ errors.rs
│  ├─ ipp.rs
│  └─ transcript.rs
└─ tests
   ├─ test_prop_ipp.rs
   └─ Cargo.toml
```

We created a `transcript.rs` file which implements wrapper functions over the Fiat-Shamir heuristic using the Merlin library (see section 5.3). This file needs access to the same defined errors as the `ipp.rs` file. Normally in Rust the solution would be:

```
use crate::errors::*;
```

But as this is not valid Hacspec code, as the only file allowed to access modules in this library's src folder is `ipp.rs`.

To solve this we resorted to following:

```rust
mod errors;
use errors::*;
```

We created the **transcript** folder containing a symbolic link *`errors.rs`* $\rightarrow$ `errors.rs`. This lets both `ipp.rs` and `transcript.rs` access the `errors.rs` file without code duplication. This approach is not ideal, but we have been unable to find any functional issues with it. If there should come objections during the merge process it could be solved using code duplication or by reducing modularization by having all the methods in a single file.

Testing:

To test the inner product one big test was created. Only this one test is needed as this test both asserts that our implementation constructs valid proofs by passing the verification check on valid inputs and asserts that the output of each public function matches that of the Dalek implementation. This test creates an inner product for both implementations using random values for $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{G}, \boldsymbol{H}, Q, n$ where n is randomly selected as a power of two. QuickCheck was not used for this randomness, as there is a significant amount of random values to be generated, some with specific requirements.

Since the $n \in \{2, 4, 8, 16, 32, 64\}$ is chosen randomly some tests take longer since there are exponentially longer loops. While n could be almost arbitrarily large, having *very* large inputs does not help much more in proving the equivalence property, hence the limited range for n in our tests. Various smaller helper functions were also created to convert between types, so we could easily convert the random data to the types of each respective implementation.

As the inner product part of the Dalek implementation is not publically exposed, we had to fork the library and change the relevant functions to be globally public instead of only being public within the crate. No other changes were made to the Dalek inner product implementation, to ensure that we did not change its functionality.

Testing using our full implementation is quite slow, and as such does not allow us to run as many tests, reducing the guarantee that the two implementations function equivalently. To remedy this, we created a wrapper library which implements the same functions as our Hacspec Ristretto specification, but for each operation it instead calls equivalent functions from the Dalek Ristretto crate. This still gives us an equivalence guarantee, as we have already tested our Hacspec Ristretto specification against the Dalek Ristretto crate. This lead to much more rigorous testing, as it caused a *significant* performance increase. The table below compares runtimes using our Hacspec Ristretto library and the wrapper over the Dalek Ristretto crate:

	Using Hacspec Ristretto	Using Dalek Ristretto	Times faster
$n = 2$	27.02 s	0.07 s	386.00
$n = 4$	61.78 s	0.12 s	514.83
$n = 8$	144.57 s	0.24 s	602.37
$n = 16$	260.30 s	0.33 s	788.78
$n = 32$	517.62 s	0.61 s	848.55
$n = 64$	1,004.01 s	1.08 s	929.63

With these performance increases we ran 100,000 tests using randomized inputs, which passed successfully, taking 17,392.57 seconds.

5.5 Implementing Bulletproofs:

The final part of the project was the implementation of the Bulletproofs protocol. The code is partly based on the the Dalek implementation and partly based on the mathematical formulas explained in section 4.2 and 4.3. Our specification models the same types and functions as the Dalek implementation, therefore the steps performed in constructing the range proofs will be identical. A flow graph over the steps involved can be seen on the next page:

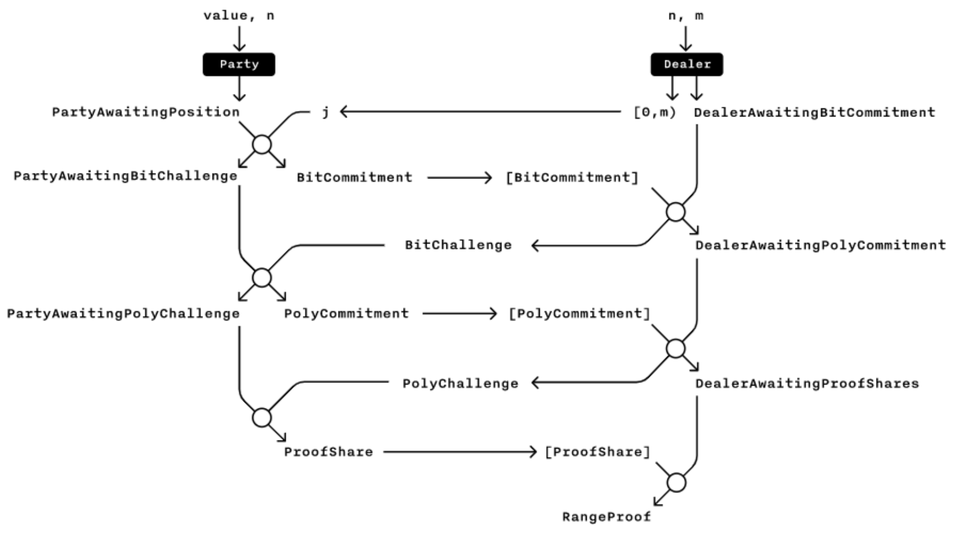

Figure 1: Graph over type and function interactions. Graph from [11].

This flow graph is an outline for the full Bulletproofs protocol, with each node being a function called in the code. Below we outline a simplified pseudocode representation of our implementation:

Algorithm 3: Bulletproof algorithm

```
Input: $n \in \mathbb{Z}, \boldsymbol{v} \in \mathbb{Z}_p^m, \widetilde{\boldsymbol{v}} \in \mathbb{Z}_p^m, B \in \mathbb{G}, \widetilde{B} \in \mathbb{G}, \boldsymbol{G} \in \mathbb{G}^n, \boldsymbol{H} \in \mathbb{G}^n$
1  perform_checks()
2  $m :=$ $\boldsymbol{v}$.len()
3  // DealerAwaitingBitCommitment
4  $dealer\_step1 :=$ create_dealer($B, \widetilde{B}, \boldsymbol{G}, \boldsymbol{H}, n, m$)
5  for $j \in [1 : m+1]$ do
6      // PartyAwaitingPosition
7      $party_j\_step1, V_j :=$ create_Party($B, \widetilde{B}, \boldsymbol{G}, \boldsymbol{H}, v, \widetilde{v}, n$)
8      // PartyAwaitingBitChallenge
9      $party_j\_step2, A_j, S_j :=$ create_bit_commitment($party\_step1, j$)
10 end for
11 // DealerAwaitingPolyCommitment
12 $(dealer\_step2, \mathrm{y}, \mathrm{z}) :=$ receive_bit_commitments($dealer\_step1, \boldsymbol{V}, \boldsymbol{A}, \boldsymbol{S}$)
13 for $j \in [1 : m+1]$ do
14     // PartyAwaitingPolyChallenge
15     $(party_j\_step3, T_{1,j}, T_{2,j}) :=$ create_poly_commitment($party_j\_step2, \mathrm{y}, \mathrm{z}$)
16 end for
17 // DealerAwaitingProofShare
18 $(dealer\_step3, \mathrm{x}) :=$ receive_poly_commitments($dealer\_step2, \boldsymbol{T_1}, \boldsymbol{T_2}$)
19 for $j \in [1 : m+1]$ do
20     // ProofShare
21     $t(\mathrm{x})_j B, (\mathrm{x})_{\mathrm{j}} \widetilde{\mathrm{B}} :=$ create_proofshare($party_j\_step3, \mathrm{x}$)
22 end for
23 // RangeProof
24 $aggregated\_rangeproof :=$ receive_proofshares($dealer\_step3, \boldsymbol{t}(\mathbf{x})\boldsymbol{B}, (\mathbf{x})\widetilde{\mathrm{B}}$)
25 return $(\boldsymbol{V}, aggregated\_rangeproof)$
```

The functions called on lines 4, 12, 18, and 24 are from the `dealer` module while the functions called on lines 7, 9, 15, and 21 are from the `party` module. Both sets of functions correspond to the nodes in figure 1 with the left nodes being the `party` functions and the right nodes being the `dealer` functions. Each of these functions, in the simplest terms, compute the values needed for the current step of the protocol, and generate the next party or dealer instance in the sequence. The names of these have been shortened due to formatting, but they correspond to the types from figure 1, with the graph names written as comments above the line where they are created in the pseudocode.

To keep our implementation as simple as possible, we only implemented what would be equivalent to the `prove_multiple_with_rng()` function from the crate we test against. We felt this was sufficient as the `prove_multiple()`, `prove_single_with_rng()` and `prove_single()` functions would all call `prove_multiple_with_rng()` in some way.

The Dalek implementation uses structs to represent the values for each of the different types of party and dealer, created by the functions described above. Each dealer struct implements the next function required to progress to the next `dealer` struct, and the same goes for each `party` struct. This way of implementing the functions ensures that a potential user of the crate cannot accidentally perform the steps out of order, due to the next step only being possible given the successful execution of the previous step.

When we began our implementation we made use of tuples to represent our types. However, not using structs eventually caused severe code-bloating since tuples do not implement the `Copy` or `Clone` traits, requiring some cumbersome and sizable rewrites. Not using structs also breaks the property of only being able to perform the steps in the right order. Therefore, we switched to using structs which can derive the `clone()` method.

There is, however, one place where our implementation differs from this convention. Namely with the `ProofShare` type. This is due to both the `dealer` and `party` modules needing direct access to the struct's definition, this was not possible as `ProofShare` instances created in the `party` module will have type `party::types::ProofShare` where the needed type is `types::ProofShare`. As such, `ProofShare` is a simple type-aliased tuple. This *does* break the guarantee that the steps will always be performed in the right order, but this is not a major issue as our implementation is not meant to be used as a library, it is only intended for proving properties using proof assistants.

A big roadblock we hit while implementing this protocol was the immense length it would have. This meant that writing any sort of test for the code would require it to be finished in nearly its entirety. This resulted in a lot of debugging through printing out values and manually comparing them with the Dalek implementation. Another issue was the fact that their library has built-in randomness. This meant that our code had no way of utilizing the exact same random values, which is vital when comparing the outputs of the implementations. Therefore, we forked their library and refactored all their random values to be function inputs. This in no way changes the functionality of the Dalek implementation, while allowing us to control the random values it uses. This allows us to feed both implementations with the same random inputs, thus allowing us to run our equivalence tests.

Additionally, due to the lack of optimizations our implementation is immensely slow, to the point where testing our full implementation takes 22 hours to complete, while the equivalent checks for the Dalek implementation takes a mere 2 minutes. To remedy this we instead utilized a wrapper module to use the Dalek Ristretto implementation instead of our own, as was done when testing the inner product proof (section 5.4). This significantly increased the speed of our implementation and thus allowed much more rigorous testing.

Testing:

To test the code we created a main helper function `test_bulletproofs()`:

```rust
fn test_bulletproofs(m: usize, n: usize) {
  let (bp_gens_hac, bp_gens_rust) = create_bp_gens(number_of_values, n);
  let (pc_gens_hac, pc_gens_rust) = create_pc_gens();
  let (transcript_hac, mut transcript_rust) = create_transcript();

  let (values_hac, values_rust) = generate_random_values(number_of_values, n);
  let (blindings_hac, blindings_rust) = generate_random_scalars(number_of_values);
  let (a_blindings_hac, a_blindings_rust) = generate_random_scalars(number_of_values);
  let (s_blindings_hac, s_blindings_rust) = generate_random_scalars(number_of_values);
  let (s_L_hac, s_L_rust) = generate_many_random_scalars(n, number_of_values);
  let (s_R_hac, s_R_rust) = generate_many_random_scalars(n, number_of_values);
  let (t1_blindings_hac, t1_blindings_rust) = generate_random_scalars(number_of_values);
  let (t2_blindings_hac, t2_blindings_rust) = generate_random_scalars(number_of_values);

  [...]
```

`BulletproofGens` and `PedersenGens` are converted from the Dalek implementation and the transcripts are initialized with the same label. The rest of the values, except `n` and `m`, are generated randomly using helper functions. The values `n` and `m` are taken as function parameters. This lets us run many short test functions testing for different `m`'s and `n`'s:

```rust
#[test]
fn m2n8() {
    test_bulletproofs(2,8);
}

[...]
```

After running a single test for the most reasonable combinations of n and m, we replaced the Hacspec Ristretto library with our wrapped Ristretto library. This greatly increased speed and therefore the amount of tests we were able to run. Below is a table of what tests were done, how long they took, and the speed increase:

	Using Hacspec Ristretto	Using Dalek Ristretto	Times faster
$m = 2, n = 8$	420.02 s	0.57 s	736.84
$m = 2, n = 16$	809.14 s	0.97 s	834.16
$m = 2, n = 32$	1,563.87 s	1.83 s	854.57
$m = 2, n = 64$	2,073.97 s	3.56 s	582.57
$m = 4, n = 8$	823.68 s	0.97 s	849.15
$m = 4, n = 16$	1,585.96 s	1.79 s	886.01
$m = 4, n = 32$	3,075.16 s	3.32 s	926.25
$m = 4, n = 64$	6,107.19 s	6.55 s	932.39
$m = 8, n = 8$	1,624.98 s	1.85 s	878.36
$m = 8, n = 16$	3,127.60 s	3.49 s	896.16
$m = 8, n = 32$	6,114.07 s	6.63 s	922.18
$m = 8, n = 64$	10,418.94 s	12.69 s	821.03
$m = 16, n = 8$	3,191.85 s	3.63 s	879.29
$m = 16, n = 16$	6,209.07 s	7.20 s	862.37
$m = 16, n = 32$	12,246.25 s	13.56 s	903.11
$m = 16, n = 64$	24,303.13 s	26.06 s	932.58
$m = 32, n = 64$	39,100.84 s	41.70 s	937.67
$m = 64, n = 64$	67,508.52 s	64.59 s	1,045.18

With these performance increases, we ran 5,000 tests on randomized n and m values, which all successfully passed, taking 43,017.28 seconds. The reason we did not run more tests is because, despite these speed increases, it still runs slow compared to our other libraries. This means that the Bulletproofs library is not as thoroughly tested as we would have liked, which makes the property of equivalence between the Bulletproofs implementations weaker compared to the other specifications. This is mediated by how comprehensive the protocol is, meaning a single randomized test passing is good evidence to support that the code functions properly for that combination of n and m. With 5,000 tests, we have 200 tests for each combination of inputs on average.

6 Future work:

Currently, our implementation functions as intended, however parts has yet to be properly merged into Hacspec. This is still being worked on in collaboration with the Hacspec team, which can take time and will involve refactoring parts of the code to improve overall code quality.

After this is completed the intended work is finished, but the obvious next step is to use a proof assistant on our implementations to prove properties about the equivalent, but much faster, Rust implementations, which would improve their security.

More thorough testing could be done to ensure that the Hacspec and Rust implementations function identically, which could involve writing more property-based and unit tests as well as running more of the randomized tests.

7 Conclusion:

Our intended goal of a complete implementation of Bulletproofs was achieved. Necessary libraries, namely Ristretto and Merlin, were also implemented. These implementations are constructed in a way that makes them function identically to their Rust counterparts, with high probability, due to rigorous property-based testing. Thus, by proving properties about our implementations, those same properties should hold for the respective Rust implementations.

Additionally, we have also highlighted how a somewhat abstract cryptographic protocol could be implemented in Hacspec. Thus, it can act as a reference for how such protocols might be implemented and as an example of how to utilize our Ristretto and Merlin libraries.

Although we failed to create a generalizable linear algebra library which could be useful for specifications that need to make use of linear algebra, we have paved the way for how such a library could look by creating a reference implementation. The Hacspec developers can use this as a reference to identify what features Hacspec would need in order to support such a library.

In essence, we have laid nearly all the groundwork for the future work of proving properties about the Rust implementations, and introduced a few powerful building blocks allowing future developers to more easily implement other cryptographic algorithms, by allowing them to use our Ristretto and Merlin libraries for implementing protocols that utilize elliptic-curve cryptography and zero-knowledge proofs respectively.

8 References

References

[1] Andrew Gallant. *QuickCheck*. https://github.com/BurntSushi/quickcheck. Accessed: 2022-08-14.

[2] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. "hacspec: Towards verifiable crypto standards". In: *International Conference on Research in Security Standardisation*. Springer. 2018, pp. 1–20.

[3] Benedikt Bünz et al. "Bulletproofs: Short proofs for confidential transactions and more". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 315–334.

[4] S. Vanstone D. Hankerson A. Memezes. *Guide to Elliptic Curve Cryptography*. Springer Science & Business Media, 2010. ISBN: 9781441929297.

[5] Dimforge. *Nalgebra*. https://nalgebra.org/. Accessed: 2022-08-14.

[6] Joan Feigenbaum. *Advances in Cryptology—CRYPTO'91: Proceedings*. Vol. 576. Springer, 2003.

[7] H. de Valence, J. Grigg, G. Tankersley, F. Valsorda, I. Lovecruft. *The ristretto255 Group*. https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-00.html. Accessed: 2022-08-14.

[8] Jim Hefferon. *Linear Algebra*. Fourth edition. 2020.

[9] Henry de Valence. *Merlin Transcripts*. https://merlin.cool/. Accessed: 2022-08-14.

[10] Henry de Valence, Cathie Yun, Oleg Andreev. *Bulletproofs*. https://github.com/BurntSushi/quickcheck. Accessed: 2022-08-14.

[11] Henry de Valence, Cathie Yun, Oleg Andreev. *Bulletproofs Notes*. https://doc-internal.dalek.rs/bulletproofs/notes/index.html. Accessed: 2022-08-14.

[12] Huseyin Hisil et al. *Twisted Edwards Curves Revisited*. Cryptology ePrint Archive, Paper 2008/522. https://eprint.iacr.org/2008/522. 2008. URL: https://eprint.iacr.org/2008/522.

[13] Jim Miller. *Bulletproofs Notes*. https://blog.trailofbits.com/2022/04/15/the-frozen-heart-vulnerability-in-bulletproofs/. Accessed: 2022-08-14.

[14] Josh Swihart, Benjamin Winston and Sean Bowe. *Zcash Counterfeiting Vulnerability Successfully Remediated*. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/. Accessed: 2022-08-14.

[15] luigi1111, Riccardo "fluffypony" Spagni. *Disclosure of a Major Bug in CryptoNote Based Currencies*. https://www.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html. Accessed: 2022-08-14.

[16] Mike Hamburg. *Strobe Protocol Framework*. https://strobe.sourceforge.io/. Accessed: 2022-08-14.

[17] Rasmus Kirk Jakobsen and Anders Wibrand Larsen. *Bulletproofs implemented in Hacspec*. https://github.com/HacspecBulletproofs/HacspecBulletproofs. Accessed: 2022-08-14.

[18] Trail of Bits. *Fiat-Shamir transformation*. https://www.zkdocs.com/docs/zkdocs/protocol-primitives/fiat-shamir/. Accessed: 2022-08-14.

[19] Trail of Bits. *Schnorr's identification protocol*. https://www.zkdocs.com/docs/zkdocs/zero-knowledge-protocols/schnorr/. Accessed: 2022-08-14.