Department of Computer Science,
Aarhus University

High Assurance Specification of the halo2 Protocol

Høj Garanti Specifikation af halo2 Protokollen

Master's Thesis (30 ECTS) in Computer Science

Lasse Bramer Schmidt 201805994
Rasmus Tomtava Bjerg 201809390

Advisor

Bas Spitters

June, 2023

Abstract

halo2 is a zk-SNARK building on the original halo. It is novel in that it has several interesting properties; in particular, it is the first system to have recursive proof composition without needing a trusted setup. It has been developed by the Zcash team and the Electric Coin Company for use in the Zcash blockchain, but it is general purpose and can be used in any zero-knowledge application.

This paper presents an executable specification of halo2's proving system, realized using `hacspec`. `hacspec` is a specification language for producing executable specifications of cryptographic primitives. As halo2 is one of the more extensive projects to be specified with `hacspec` , it has also been an exploration of `hacspec`'s abilities for projects of this size.

The tool rustdoc is used to organically present the specification together with its description, as `hacpsec` is a subset of rust.

The labor of this paper also led to contributions to the official halo2 protocol description, a `hacspec` specification of the Pasta curves, and a `hacspec` specification of a polynomial ring over the Vesta curve's base field.

Resumé

halo2 er en zk-SNARK som bygger oven på den originale Halo zk-Snark. halo2 er spændende da den har flere interessante egenskaber; specifikt er den det første system til at have rekursiv bevissammensætning uden af have brug for et trusted setup. Halo2 er udviklet af Zcash holdet og Electric Coin Company til at blive brugt på Zcash blockchainen, men den kan bruges til alle zero-knowledge applikationer.

I dette speciale vil vi præsentere en specifikation af halo2's bevissystem skrevet i `hacspec`. `hacspec` er et specifikationssprog til kryptografiske primitiver, som kan lave eksikverbare specifikationer. Da halo2 er en af de mere omfattende projekter der er blevet specificeret i `hacspec` har det også været et studie i `hacspec`s evner til at specificere mere omfattende projekter.

Rustdoc kan bruges til at præsentere `hacspec` specifikationer på en naturlig måde, da `hacspec` er et subset af rust.

Arbejdet med dette speciale har også ført til bidrag til den officielle Halo2 protokol beskrivelse, en `hacspec` specifikation af Pasta kurverne, og en `hacspec` specifikation af en polynomiering over Vesta kurvens base field.

Acknowledgments

First and foremost, we would like to thank our advisor Bas Spitters, for his advice on the thesis and guidance through the world of specifications. He also introduced us to the programming language `hacspec`, which has been a joy.

We thank the `hacspec` community, especially Franziskus Kiefer and Lucas Franceschino. It has been a pleasure to participate in the community, and we hope to be able to keep contributing in the future.

We would also like to thank the halo2 team for all their help and interest throughout the project. Working on a project is always more fun when someone else shows interest, and you did - a special thanks to Ying Tong Lai, Daira Emma Hopwood, and Jack Grigg.

Finally, a thank you to Diego F. Aranha for guidance on elliptic curves.

Contents

Chapter 1

Introduction

Zero-knowledge proofs can be useful in many situations, and one such situation is for blockchains. Traditionally, the complete history of a blockchain will be publicly available. Some cryptocurrencies are enhancing privacy by hiding receiver, sender, amounts, etc., of transactions. This can be achieved using zero-knowledge proof constructs, which is what the Zcash developers do with their zk-SNARK *halo2*[19]. Specifically, Zcash uses this for *shielded transactions*[20]. These allow for improved privacy on the blockchain, namely that you can hide most information in a transaction while still keeping it verifiable. One potential problem is that a verifier might need to go through all proofs to verify the entire blockchain. Luckily, halo2 provides an answer to this problem. halo2 is able to "accumulate" proofs into batches. This is done with an accumulation scheme, where you construct proofs that verify earlier instances upon verification. As such, the verifier can verify a batch's "final" proof and has then effectively verified the entire batch.

In practice, this zk-SNARK is realized by first having an interactive argument, which is then made non-interactive through the use of the *Fiat-Shamir transformation*[19, Sec. 4.2]. This paper focuses on formulating a specification of the protocol for the interactive argument. This is done in hacspec[15], a formal specification language for cryptography. `hacspec`, being a subset of Rust, aims to facilitate succinct, readable, and executable specifications. Conceptually, this should make it easier to port the specification to an efficient version, namely a rust implementation. Another benefit of `hacspec` being a subset of Rust is the ability to use `rustdoc`[1], which allows us the mix `hacspec` code with markdown to create a kind of unified document, linking the `hacspec` specification, the protocol description, and the official implementation together.

[1]https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html

We have an extensive test suite to ensure that it is a high-assurance specification. In fact, approximately ⅔ of the lines of code are tests. We hope our specification can be useful in future `hacspec` specifications that rely on a halo2 specification. We hope to get our specification upstreamed to the `hacspec` project, and if future modifications or additions to the specification should happen, these tests will hopefully aid that process.

As halo 2 depends on the *Pasta Curves*[12], we have also created a high-assurance specification for those, which we hope can be useful to others.

This paper will start by presenting some prerequisites for the following sections. Then follows an overview of the halo2 protocol, with the essential pieces laid out. Finally, we will present our contributions in the form of our specifications of the halo2 protocol and the Pasta-curves and our contributions to the halo2 book.

Sources for the full specifications and latex sources for this paper are available in the appendix.

Rasmus Bjerg and Lasse Schmidt are collectively responsible for every part of the paper.

Chapter 2

Prerequisites

2.1 Notation

a	A scalar (generally from a finite field)
G	A group element
X	An indeterminate
$[a]G$	Multiplication of a group element by a scalar
$\mathbb{F}^n, \mathbb{G}^n$	The sets of all vectors of length n
$\mathbf{a}$	A vector of scalars
$\mathbf{G}$	A vector of group elements
$\langle \mathbf{a}, \mathbf{b} \rangle$	Inner product
$\langle \mathbf{a}, \mathbf{G} \rangle$	Multiscalar multiplication
$\mathbf{G}_{lo}, \mathbf{a}_{lo}$	The first half of a vector
$\mathbf{G}_{hi}, \mathbf{a}_{hi}$	The last half of a vector
$p(X)$	A polynomial
$p(a)$	An evaluation
$\frac{p(X)}{q(X)}$	The quotient polynomial, omitting the remainder
ω	An nth primitive root of unity
$\mathcal{P}$	The *prover*
$\mathcal{V}$	The *verifier*

2.2 Finite Field Arithmetic

Fields are an abstraction over sets of objects(usually numbers) and the two operators *Multiplication*$(\cdot)$ and *Addition*$(+)$. For the combination of a set $(\mathbb{F})$

and the two operators$(+,\cdot)$ to be a field, they must satisfy various field axioms. The following definitions come from Guide to Elliptic Curve Cryptography[10]:

Definition 1 (Field)**.** A *field* is a set ($\mathbb{F}$) and two operators *addition* and *multiplication* which together satisfies the three arithmetic properties:

1. ($\mathbb{F}$,+) is an abelian group with (additive) identity denoted by 0.
2. ($\mathbb{F}$, $\cdot$) is an abelian group with (multiplicative) identity denoted by 1.
3. The distributive law holds: $(a + b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in \mathbb{F}$

An abelian group is defined as follows[10]:

Definition 2 (Abelian Group)**.** An *abelian* group ($\mathbb{G}$,$\star$) consists of a set $\mathbb{G}$ with a binary operation $\star$: $\mathbb{G}$ x $\mathbb{G} \rightarrow \mathbb{G}$ satisfying the following properties:

1. Associativity: $a \star (b \star c) = (a \star b) \star c$ for all $a, b, c \in \mathbb{G}$
2. Existence of an identity: There exists an element $e \in \mathbb{G}$ such that $a \star e = e \star a = a$ for all $a \in \mathbb{G}$.
3. Commutativity: $a \star b = b \star a$ for all $a, b \in \mathbb{G}$.

Fields are defined with two operations, *addition* and *multiplication.* From these, we define subtraction and division as follows:

Definition 3 (Field Subtraction)**.** Subtraction in fields is defined from addition as follows:

for $a, b \in \mathbb{F}$ $a - b = a + (-b)$ where $-b$ is unique element $\in \mathbb{F}$ such that $b + (-b) = 0$

Definition 4 (Field Division)**.** Division in fields is defined from multiplication as follows:

for $a, b \in \mathbb{F}$ with $b \neq 0, \frac{a}{b} = a \cdot b^{(-1)}$ where $b^{(-1)}$ is the multiplicative inverse of b such that $b \cdot b^{(-1)} = 1$

Cryptographic protocols often make use of *Prime Fields.* A Prime Field is a *Finite Field* where the size of the field is determined by a prime number:

Definition 5 (Finite Field)**.** If the set $\mathbb{F}$ is finite, the field is a *finite field.*

Definition 6 (Prime Field)**.** Let p be a prime number. A finite field with the elements $\{0, 1, 2, ..., p-1\}$ with addition and multiplication performed modulo p is a *Prime Field*

From a prime field $\mathbb{F}_p$, we can define the multiplicative group $\mathbb{F}_p^{\times}$, which is the group over $\mathbb{F}_p - \{0\}$ where the group operator is multiplication.

Definition 7 (Subgroup)**.** A *subgroup* of a group $\mathbb{G}$ with operation $\cdot$ is a subset of elements of $\mathbb{G}$ that also form a group under $\cdot$.

A multiplicative subgroup is then a subgroup where the operation is multiplication. From the Chinese remainder theorem, we know that a group of composite order has strict subgroups. This means the multiplicative group $\mathbb{F}_p^{\times}$ of order $(p-1)$ has strict subgroups. Furthermore, Lagrange's theorem says that the order of any subgroup of a finite group $\mathbb{G}$ must divide the order of $\mathbb{G}$. Meaning that we can create pairs of subgroups $(\mathbb{G}^a, \mathbb{G}^b)$ where $a * b = p - 1$. The generators for these groups α and β can then be used to generate all the elements from $\mathbb{F}_p^{\times}$ in the following manner:

$$\forall x \in \mathbb{F}_p^{\times} : x = \alpha^i \cdot \beta^j$$

for i mod a and j mod b.

2.3 Roots of Unity

In the context of fields, an nth root of unity is an element x in the field, which solves the equation $x^n = 1$ where n is some positive integer.

Definition 8 (nth Root of Unity)**.** The nth *Roots of unity* is the set of elements in $\mathbb{F}$ that solves $x^n = 1$

In finite fields, there are n nth roots of unity for each positive integer n. The primitive nth root of unity ω can generate the other nth roots of unity and is defined as:

Definition 9 (Primitive nth Root of Unity)**.** ω is a primitive nth root of unity if $\omega^n = 1$ and $\omega^k \neq 1$ for $0 < k < n$

We can now express all the nth roots of unity as powers of ω $\{1, \omega, \omega^2, \omega^3, ..., \omega^{n-1}\}$

2.4 Polynomials

2.4.1 Vanishing Polynomial

The vanishing polynomial $Z_H(X)$ over a domain D is the polynomial with roots at all points in D. The vanishing polynomial for a domain D with n elements can be found like:

$$Z_H(X) = (X - D_1)(X - D_2)(X - D_3)...(X - D_n)$$

If we consider the domain formed by the nth roots of unity, it holds that for all $i \in [0, n-1]$ we have that $(\omega^i)^n = (\omega^n)^i = (\omega^0)^i = 1$. This means that if we reduce the vanishing polynomial to:

$$Z_H(X) = (X - \omega^0)(X - \omega^1)(X - \omega^2)...(X - \omega^{n-1}) = X^n - 1$$

2.4.2 Polynomial Rings

Polynomial rings are algebraic structures that describe a set of polynomials defined over a ring. This is very helpful when working in fields, as fields are rings. *Polynomial Rings*[4] defines polynomials over a ring as:

Definition 10 (Polynomial over a Ring)**.** Let $[R; +, \cdot]$ be a ring. A polynomial, $f(x)$, over R is an expression of the form

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

where $n \geq 0$, and $a_0, a_1, a_2, ..., a_n \in R$.

The group of polynomials over a ring is called a polynomial ring and is expressed as $R[X]$.

2.4.3 Lagrange basis

The basis is a set of linearly independent polynomials that can be used to express any polynomials in a given polynomial Ring. The basis allows us to express any polynomial in the ring as a linear combination of the basis polynomials. When working with roots of unity, the basis of a ring can be expressed with the Lagrange basis. [19, Sec. 5.2] the Lagrange basis is defined as follows:

Definition 11 (Lagrange Basis)**.** Consider the order-n multiplicative subgroup $\mathcal{H}$ with primitive root of unity ω. The *Lagrange basis* corresponding to this subgroup is a set of functions $\{\mathcal{L}_i\}_{i=0}^{n-1}$, where

$$\mathcal{L}_i(\omega^j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

2.4.4 Lagrange Interpolation

Lagrange Interpolation is a way to form the lowest degree polynomial passing through a set of points. Given n points, the Lagrange polynomial will, at most, be of degree $n-1$. Given a set of points A, you run through them, find the Lagrange basis for the x-value, multiply that basis with the y-value of the point, and sum all the resulting polynomials together. Using the evaluation representation of the Lagrange polynomial, it can be defined as

Definition 12 (Lagrange Polynomial)**.** Given the evaluation representation of A:

$$A : \{(x_0, A(x_0)), (x_1, A(x_1)), ..., (x_{n-1}, A(x_{n-1}))\}$$

The Lagrange polynomial $A(X)$ is defined as:

$$A(X) = \sum_{i=0}^{n-1} A(x_i)\mathcal{L}_i(X)$$

2.4.5 Rotation of Polynomials

As we shall see, rotating polynomials are essential to halo2 and PLONKish circuits. The main idea is to essentially define new polynomials by multiplying the indeterminate of a polynomial $p(X)$ with a value ω

$$p_{rotated}(X) = p(\omega X)$$

Let us say for example that $p(X) = a + bX + cX^2$ and so we would have

$$\begin{aligned} p_{rotated}(X) &= p(\omega X) \\ &= a + b(\omega X) + c(\omega X)^2 \\ &= a + b\omega X + c\omega^2 X^2 \end{aligned}$$

This primarily makes sense in the context of the PLONKish matrix. Here the columns are defined by polynomials, where we can index rows using ω. When

ω is an n-th prime root of unity forming the domain $D = (\omega^0, \omega^1, ..., \omega^{n-1})$, we can "access" the i'th row of a column represented by some polynomial $a(X)$ with $a(\omega^i)$. The definition of ω also allows for indexing with *wrap-around*; that is to say if we want to access the "row beneath the current row", we can rotate the polynomial by ω, and if the current row was the final one, it wraps back to the first row, since $\omega^n = \omega^0 = 1$. We shall explore this more in 3.1 and 4.2.1.

2.5 Elliptic Curves

Elliptic curves can generate what the halo2 book [19, Sec. 5.3] calls *Cryptographic Groups*. These are groups where the discrete logarithm problem is hard. While this is an oversimplification, it serves as sufficient motivation for us. Elliptic curves come in many forms; we will focus on the curves with the following definition from *Guide to elliptic curve cryptography*:

Definition 13 (Elliptic Curve)**.** An *elliptic curve* E over a prime-field $\mathbb{F}_p$ is defined by the solutions to the equation

$$y^2 = x^3 + ax + b \tag{2.1}$$

where a and b is from $\mathbb{F}_p$

The points on the curve being pairs of (x, y) solving the equation, then forms an Additive Group. The group laws are defined as

Definition 14 (EC Additive Identity)**.** The point at infinity where $y = \infty$ is denoted as $\mathcal{O}$ and it is the additive identity as the following property holds:

$$P + \mathcal{O} = \mathcal{O} + P = P \tag{2.2}$$

Definition 15 (EC Point Negation)**.** Let $P = (x, y)$ be a point on the curve E. $-P$ is the negation of P and is defined by mirroring P in the x-axis:

$$-P = (x, -y)$$

also, $-\mathcal{O} = \mathcal{O}$

Definition 16 (EC Point Addition)**.** Let P, Q, and R be points on the curve, and ℓ is the line that intersects them. The group operation for addition is then defined as

$$P + Q = -R$$

where $P \neq \pm Q$

Definition 17 (EC Point Doubling)**.** Let P be a point on the curve where $P \neq -P$, ℓ be the tangent to P, and $-R$ the point on the curve where ℓ intersects. Then:

$$2P = R$$

From this, we can define subtraction and scalar multiplication:

Definition 18 (EC Point Subtraction)**.** Let P and Q be points on the curve:

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

Definition 19 (EC Scalar Multiplication)**.** Multiplying a point P with a scalar m is defined by adding P to itself m times, the first being done using Point Doubling. Scalar multiplication is denoted $m \cdot P$

The halo2 Book associates two fields with an elliptic curve, the *Base Field* and the *Scalar Field*[19, Sec. 5.4].

Definition 20 (Base Field and Scalar Field)**.** Let E be an elliptic curve over the prime field $\mathbb{F}_p$. This is denoted by $E/\mathbb{F}_p$. We denote the group created by this elliptic curve as $E(\mathbb{F}_p)$ with order q. The order of $E(\mathbb{F}_p)$ defines a new field $\mathbb{F}_q$. We call $\mathbb{F}_p$ the *base field* of $E/\mathbb{F}_p$ and $\mathbb{F}_q$ the scalar field of $E/\mathbb{F}_p$.

Furthermore the halo2 book introduces *Cycles of Curves*

Definition 21 (Cycles of Curves)**.** Let $E_p/\mathbb{F}_p$ be a curve over the field $\mathbb{F}_p$ and $E_q/\mathbb{F}_q$ be a curve over the field $\mathbb{F}_q$. If the scalar-field of $E_p/\mathbb{F}_p$ is $\mathbb{F}_q$ and the scalar-field of $E_q/\mathbb{F}_q$ is $\mathbb{F}_p$ they form a *Cycle of Curves*

2.6 Pedersen vector commitments

The Pedersen vector commitment scheme is an adapted version of Pedersen commitments for use with vectors, such as vectors with coefficients for a polynomial.

Definition 22 (Common Reference String)**.** A common reference string is needed for the Pedersen vector commitment scheme. It is defined as follows[19, Sec. 5.5]:

$$\sigma = (\mathbb{G}, \mathbf{G}, H, \mathbb{F}_p)$$

where

- $\mathbb{G}$ is a group of prime order p
- $\mathbf{G} \in \mathbb{G}^d$ is a vector of random group elements
- $H \in \mathbb{G}^d$ a random group element
- $\mathbb{F}_p$ is finite field of order p

A Pedersen vector commitment is then defined as follows:[19, Sec. 5.5]:

Definition 23.

$$\text{Commit}(\sigma, p(x); r) = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H$$

where each entry $\mathbf{a}_i \in \mathbb{F}_p$ of $\mathbf{a}$ is the coefficient in $p(x)$ for the term with degree i and $r \in \mathbb{F}_p$ is some blinding factor.

Pedersen Vector commitments are additively homomorphic[2]. This is defined as:

Definition 24 (Additively Homomorphic Scheme)**.** $\forall a, b, r, s \in \mathbb{F}_p$ and $p(X), q(X) \in \mathbb{F}_p[X]$, we have

$$[a]\text{Commit}(\sigma, p(X); r) + [b]\text{Commit}(\sigma, q(x); s) = \text{Commit}(\sigma, a \cdot p(X) + b \cdot q(X); ar + bs)$$

2.7 Zero-knowledge Proofs

From a high-level perspective, zero-knowledge proofs (and arguments) are cryptographic constructs that allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that some statement is true without revealing anything else. Three properties should hold for such systems[23, Sec. 1.1.1]:

- **Completeness:** If a statement is true and both $\mathcal{P}$ and $\mathcal{V}$ follows the protocol, then $\mathcal{V}$ will accept
- **Soundness:** If a statement is false and $\mathcal{V}$ follows the protocol, then $\mathcal{V}$ will reject
- **Zero-knowledge:** If a statements is true and $\mathcal{P}$ follows the protocol, then $\mathcal{V}$ will not learn anything but the statement's validity

For proofs these statements are in the form of proofs of membership. That is to say; for some common input x, you prove that it is in some language L, or $x \in L$. For suitably chosen languages, this generally requires the prover to be computationally unbounded[23, Sec. 1.4.3].

Another useful concept is proofs of knowledge, where the prover is given some auxiliary secret input (the knowledge). The goal is then to prove that this secret supports a statement without revealing the secret[23, Sec. 1.4]. More formally, we want to prove that for some relation R, we have $(x, w) \in R$, with w being the witness and x being the common input. This system has some special properties, but the notion is the same as above. Loosely speaking, these properties are[13]:

- **Knowledge Completeness:** If $\mathcal{P}$ knows the claimed information, they can almost always convince $\mathcal{V}$
- **Knowledge Soundness:** If $\mathcal{P}$ can convince $\mathcal{V}$ (using any strategy) with substantial probability, then $\mathcal{P}$ knows the claimed information

Indeed, this is what is important to us since halo2's protocol is a Zero knowledge Argument of Knowledge. Arguments must only preserve soundness against computationally bounded provers[23, Sec. 1.1.1]. Arguments and proofs are sometimes collectively referred to as proofs, even with this distinction.

2.8 zk-SNARKs

Zero-knowledge succinct non-interactive arguments of knowledge, abbreviated zk-SNARKs, are cryptographic constructs that are not unlike the concepts discussed above. In addition to what we saw there, they have two special properties succinctness and non-interactivity.

Succinctness, loosely speaking, mandates that a proof (a zk-SNARK) is "small" and computationally cheap, or fast, to verify. In the context of verifiable computation, the proof should be asymptotically smaller in size and less expensive to verify than the computation itself[2].

Non-interactivity refers to the fact that no interaction is required between a prover and a verifier. This has the benefit that a prover can publish a proof, which can then be independently verified. This is valuable in the context of blockchains since such proofs can then be embedded in the blockchain. Furthermore, by accumulating proofs and exploiting succinctness, proofs can attest to the validity of previous proofs[2]. This allows for recursive proof composition, which is an advantage for blockchains, where it might be undesirable to verify all blocks or messages/transactions.

2.9 hacspec

`hacspec` [15] is a domain-specific sub-language of the Rust programming language. Its main purpose is to be a specification language for cryptographic protocols and primitives. The four most notable features of `hacspec` are:

1. `hacspec` is syntactically very similar to Rust; this makes it accessible to most programmers and cryptographers, as Rust has gained traction in the field.
2. Specifications written in `hacspec` are executable, meaning they can be tested and act as prototype implementations.
3. Syntax is similar to pseudocode used in cryptographic standards; this allows the implementation to act as pseudocode in a standard.
4. It comes equipped with tools to translate specifications to languages such as Coq and F*.

These features allow a `hacspec` implementation to act as both the pseudocode in a standard, a formal specification, and a prototype implementation at the same time. This bridges three gaps we usually see. The informality of normal pseudocode would mean a separate specification would be needed for verification, and this specification is then written in a language different from the implementation and would not be usable as a prototype.

For this to be possible, `hacspec` has a fairly strict syntax where many of the enhancements of Rust are not available. While this inherently makes the language simpler and forces it to be more readable, it also makes it harder to express certain concepts. The syntax is a strict subset of Rust, with all the control flow operators, values, and functions. One big difference is the restriction on borrowing. In `hacspec`, only immutable borrowing in function arguments is allowed, as this simplifies the semantics. Another notable limitation is that `hacspec` only supports declarations of top-level functions. Furthermore, recursive function calls are forbidden, and the control flow is limited as `return` statements are forbidden.

While some of the syntactic enhancements of Rust are not available in `hacspec` , such as `while` loops, the `hacspec` library implements some wrapper types to bring some of the omitted functionality to `hacspec` . For example, the `Vec<T>` type is unavailable in `hacspec`. Instead, `Seq<T>` implements some of the same functionality but only allows for fixed length. Another significant contribution from the `hacspec` team is `Secret Integers`. Secret integers enforce `secret independence`, which prevents side-channel attacks.

Furthermore, external crates are not allowed in `hacspec` , as they would not pass the `hacspec` type check. Instead, the `hacspec` library and examples are equipped with an array of cryptographic primitives such as fields, hash functions, and elliptic curves. This allows for easier development, as these are ready to use and therefore do not need to be tested. The examples in this library can also be used for guidelines for future implementations. We have drawn great inspiration from the `bls12-381` curve implementation for our `pasta` curve implementation. It is also our goal to merge our contributions into this library. This includes a halo2 specification, a `pasta` curve specification, and possibly a suite for basic polynomial functionality.

`hacspec` allows certain parts of the code to be omitted from the compliance check. This is useful for writing tests as randomness is not permitted in `hacspec` . All our code is written to be `hacspec` compliant outside of testing.

While working on this paper, `hacspec2` is nearing feature parity with version 1. This is a very interesting development as it brings many features into `hacspec` , most notably generics, and traits, which would be very useful, specifically in our work with polynomials.

Chapter 3

The halo2 Protocol

halo2 is a zero-knowledge proving system developed in the open by the Zcash team[1]. It is the first system that does not require a trusted setup and allows for recursive proofs[22], which makes it suitable for use in the Zcash digital currency. Its application is, however, not limited to crypto-currencies; as it is a general-purpose ZKP, it is suitable for any ZKP application[22].

In the following, we will explore the different parts of halo2. As a general overview, this includes expressing computation as circuits, their arithmetization, and the actual protocol between a prover and a verifier. Essentially, these together make it possible to verify arbitrary computation using halo2.

In the following, *protocol* will be in reference to the halo2 book's protocol description[19, Sec. 4.2]. It can be found in appendix A and a version with our additions in appendix B.

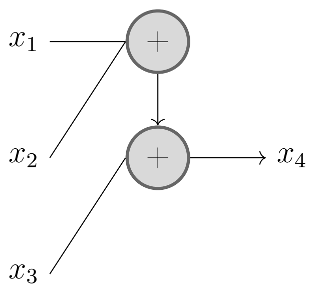
3.1 PLONK and circuits

halo2 is based on PLONK, specifically an extended version, which they call PLONKish. PLONK[7] describes a way to express constraints as polynomials. This is done by designing a circuit that expresses the desired computation. As a simple example, let us imagine we know x_1, x_2, x_3, x_4 such that

$$x_1 + x_2 + x_3 = x_4$$

[1]https://github.com/zcash/halo2

For this, we can design a small circuit as follows:



Now, for each gate with index i, we associate $\mathbf{x_a}_i$, $\mathbf{x_b}_i$, $\mathbf{x_c}_i$, which we can think of as the gate's left-hand input wire, right-hand input wire, and output wire. Furthermore, we have some selectors, $(\mathbf{q_L})_i$, $(\mathbf{q_R})_i$, $(\mathbf{q_O})_i$, $(\mathbf{q_M})_i$, and $(\mathbf{q_C})_i$. With these, we construct the following constraint:

$$(\mathbf{q_L})_i\mathbf{x_a}_i + (\mathbf{q_R})_i\mathbf{x_b}_i + (\mathbf{q_O})_i\mathbf{x_c}_i + (\mathbf{q_M})_i(\mathbf{x_a}_i\mathbf{x_b}_i) + (\mathbf{q_C})_i = 0$$

For example, to model addition, we set $(\mathbf{q_L})_i = 1$, $(\mathbf{q_R})_i = 1$, $(\mathbf{q_O})_i = -1$, $(\mathbf{q_M})_i = 0$ and $(\mathbf{q_C})_i = 0$. Similarly, for multiplication, we use $(\mathbf{q_L})_i = 0$, $(\mathbf{q_R})_i = 0$, $(\mathbf{q_O})_i = -1$, $(\mathbf{q_M})_i = 1$ and $(\mathbf{q_C})_i = 0$.

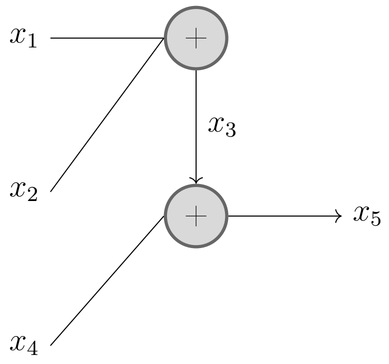
The $(\mathbf{q_C})_i$ selector can be used to include constants (which we do not use in our example).

Now we have the problem of having some intermediate result ($x_1 + x_2$, which then have to be added to x_3), which we cannot directly model with this construction. Therefore, we need to modify the original constraint a bit. We do this by adding some more variables, so we can capture intermediate values:

$$x_1 + x_2 = x_3$$
$$x_3 + x_4 = x_5$$

Note that x_3 and x_4 are not equivalent to their previous use and that we are now required to know the intermediate value x_3.

This new constraint system, in some sense, more closely resembles the corresponding circuit:



Let n denote the number of gates, then we require

$$(\mathbf{q_L})_i\mathbf{x_a}_i + (\mathbf{q_R})_i\mathbf{x_b}_i + (\mathbf{q_O})_i\mathbf{x_c}_i + (\mathbf{q_M})_i(\mathbf{x_a}_i\mathbf{x_b}_i) + (\mathbf{q_C})_i = 0 \ \forall i \in [0, n)$$

Finally, we can then encode these constraints as polynomials. First, we note that all the selectors and the "wires" define vectors, where the vector in question at index i has the corresponding value, e.g., $(\mathbf{q_L})_i$. If we look to our circuit, we could set

$$x_1 = 5, x_2 = 7, x_3 = 12, x_4 = 18, x_5 = 30$$

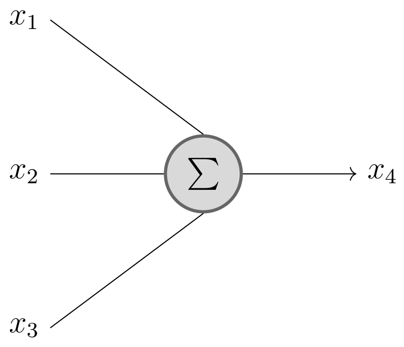
Here x_1 and x_3 would be in the $\mathbf{x_a}$ vector. Using Lagrange interpolation, we can represent $\mathbf{x_a} = [5, 7]$ as follows:

$$\mathbf{x_a}_i = \begin{cases} 5 & \text{if } i = 0 \\ 12 & \text{if } i = 1 \end{cases} \qquad \text{or} \qquad \mathbf{x_a}(X) = 5 + 7X$$

And similarly for the other vectors.

PLONKish works in roughly the same way as PLONK. PLONKish, however, abandons the strict form for constraints we saw in PLONK. Instead, we have much more flexibility. Where PLONK primarily facilitated addition and multiplication, we can define our own gates and logic with PLONKish. It also allows for looking up values relative to the current i. The halo2 book has a more formal and precise description of the differences[19, Sec. 1.2]. It might be argued that the term and idea of circuits might be a somewhat imprecise conceptualization for PLONKish arithmetization. For this example, we shall, however, continue the analogy.

If we again turn our attention to the original constraint $x_1 + x_2 + x_3 = x_4$, we can try to model this with PLONKish. Since we have complete control of the circuit, there are several ways to achieve this. One such way is to create an addition (*or sum*) gate with a fan-in of 3:



As before, we are interested in constraints that equal 0. We can rewrite the constraint to the equivalent form $x_1 + x_2 + x_3 - x_4 = 0$. Then we introduce the notion of defining PLONKish circuits in terms of a rectangular matrix of values. We use the conventional meaning of rows, columns, and cells. Let us assign some values again so that $x_1 = 5$, $x_2 = 7$, $x_3 = 18$, and $x_4 = 30$. Consider the following table:

i	a_0	a_1	a_2	q_{add}
0	5	7	18	1
1	30			0

Analogously to PLONK, each column defines a polynomial, but the distinction between in- and output wires is unclear. Instead of having a fixed form for the constraints (as we did in PLONK), we now have to construct it ourselves. For this construction of the table, we could express our constraint as the polynomial $g(X)$:

$$g(X) = q_{add}(X) \cdot (a_0(X) + a_1(X) + a_2(X) - a_0(\omega X))$$

If we added more types of gates, similar constructions would have to be added to $g(X)$. As is the case here, the construction should ensure that evaluating at each index returns 0 for valid inputs.

In many ways, this is similar to what we saw in PLONK. we still have a selector, q_{add}, which we can turn on and off. We have an n, but instead of describing the number of gates, it now describes the number of rows. As seen in the table, we have a term $a_0(\omega X)$. This ω should be a n-th prime root unity, allowing us to use rows with a relative offset to the current row. As explained in 9, we can form a domain with ω. This allows us to rotate polynomials and use values from the next row by multiplying the indeterminate in a polynomial with ω, as we did with $a_0(\omega X)$. Generally, we can multiply with ω^j to rotate the polynomial j times.

Now, the circuit is satisfied if:

$$g(\omega^i) = 0 \ \forall i \in [0, n)$$

With the flexibility of PLONKish, it is arguably easier to create matrices/circuits and reason about their behavior, especially if you consider the matrix the basis for expressing computation instead of explicit circuits. It is also worth noting that a PLONKish circuit (or matrix) defines different types of columns. The a_i columns are called *advice columns* and contain witness values. *Fixed columns* are fixed by the circuit (such as the selector q_{add}) and *instance columns* usually contain public inputs.

It is also possible to permute the PLONKish matrix to achieve a smaller size. These permutations can incur extra equality constraints, which must be part of the final polynomial g(X). Likewise, the use of lookup arguments[19, Sec. 4.1.1] will also have to be incorporated.

3.2 Proving System

Once done with arithmitazation, we arrive at the actual protocol for a zero-knowledge argument of knowledge. The protocol works with the following relation[19, Sec. 4.2]:

> Let $\omega \in \mathbb{F}$ be a $n = 2^k$ primitive root of unity forming the domain $D = (\omega^0, \omega^1, ..., \omega^{n-1})$ with $t(X) = X^n - 1$ the vanishing polynomial over this domain. Let n_g, n_e, n_a be positive integers with $n_a, n_e < n$ and $n_g > 4$.

$$\mathcal{R} = \left\{ \begin{array}{l} \left(\begin{array}{l} (g(X, C_0, ..., C_{n_a-1}, a_0(X), ..., a_{n_a-1}(X, C_0, ..., C_{n_a-1}, a_0(X), ..., a_{n_a-2}(X)))) \,; \\ (a_0(X), a_1(X, C_0, a_0(X)), ..., a_{n_a-1}(X, C_0, ..., C_{n_a-1}, a_0(X), ..., a_{n_a-2}(X))) \end{array} \right) : \\ \\ g(\omega^i, \cdots) = 0 \quad \forall i \in [0, 2^k) \end{array} \right\}$$

Here, g represents the whole PLONKish constraint system, and $a_i \ \forall i \in [0, n)$ represents the witness values as polynomials.

The proving system can then be broken into five phases[19, Sec. 4.1]:

1. Commit to polynomials encoding the circuit
2. Construct the vanishing argument to constrain all relations to 0
3. Evaluate the above polynomials at all the necessary points
4. Construct the multipoint opening argument to check that all evaluations are consistent with their respective commitments
5. Run the inner product argument to create a polynomial commitment opening proof for the multipoint opening argument polynomial

We will present a brief overview of each section; complete references can be found in the halo2 book[19].

3.2.1 Committing to the Circuit

As we saw in 3.1 when we have constructed our desired circuit, we arithmetize it and thus obtain a number of polynomials. As discussed, there are different types of columns (and so polynomials). While the fixed columns are provided by the verifier, the advice and instance columns are provided by the prover. The commitments for fixed columns can be precomputed based on the circuit with blinding factors of 1.

The commitments for advice and instance columns have to be constructed by the prover and sent over to the verifier. In practice, the commitments, for instance, and fixed columns, are computed by both the prover and the verifier so that only the advice column commitments are stored in the final proof[19, Sec. 4.1.3].

These circuit commitments translate to steps 1 and 2 of the protocol.

3.2.2 Vanishing Argument

When done committing to the circuit, we construct the vanishing argument to constrain all the relations for the circuit.

The simplest way to demonstrate this would be to divide each polynomial relation by the vanishing polynomial, which has roots at each value of the domain $(1, \omega, ...\omega^{n-1})$. In the case where the relation's polynomial is perfectly divisible by the vanishing polynomial (i.e., the remainder is 0), it proves that the relation's polynomial is 0 over the domain. Since this approach would incur the need for a polynomial commitment for each relation, halo2 utilizes another approach. In short, all the circuit's relations are committed to simultaneously[19, Sec. 4.1.4].

The verifier samples a random y and sends it to the prover. The prover then constructs a quotient polynomial

$$h(X) = \frac{gate_0(X) + y \cdot gate_1(X) + \cdots + y^i \cdot gate_i(X) + \cdots}{t(X)}$$

where $t(X)$ is the vanishing polynomial, and each $gate_i(X)$ is the polynomial corresponding to the given gate. As before, if it is perfectly divisible, then the relations are satisfied with high probability.

If we denote the maximum degree of any relation d, then the relation polynomials can have a degree of $d(n-1)$. This means that $h(x)$ has a degree of $d(n-1) - n$ (since the vanishing polynomial has degree n), which exceeds the degree which can be used with the protocol's commitment scheme. The prover therefore splits $h(x)$ into d polynomials $h_0(X), \ldots h_{d-1}(X)$ such that

$$\sum_{i=0}^{d-1} X^{ni} h_i(X) = h(X)$$

The verifier samples a random x and then wants to verify

$$\frac{gate_0(x) + y \cdot gate_1(x) + \cdots + y^i \cdot gate_i(x) + \cdots}{t(x)} = h_0(x) + \cdots + x^{n(d-1)} h_{d-1}(x)$$

In the protocol, this is combined into the multipoint opening argument. This translates approximately to step 4 through step 12 of the protocol but is carried on through the protocol.

3.2.3 Evaluating the Polynomials

Now that we have a number of polynomials for the gates and the vanishing argument, we want to evaluate them in the necessary points. Step 9 evaluates the a_i polynomials at the necessary rotations. From these evaluations, both the prover and the verifier can calculate the evaluations of the quotient polynomial from the vanishing argument. These evaluations will be used for the multipoint opening argument next.

3.2.4 Multipoint Opening Argument

Based on the example from the halo 2 book[19, Sec. 4.1.5], let us consider the commitments A, B, C, D for polynomials $a(X), b(X), c(X), d(X)$, where $a(X)$ and $b(X)$ have been queried at x while $c(X)$ and $d(X)$ have been queried at x as well as ωx. We want to be able to open these commitments with regard to the required evaluations (naturally, we should not reveal the polynomials). The simplest way to achieve this would be to have a polynomial for each point we queried at, but this would not be efficient; in our example, both $c(X)$ and $d(X)$ would appear in multiple polynomials. Instead, we collect polynomials queried in the same points into sets. In our example, we would have

$$\begin{aligned} \{x\} &: \{a(X), b(X)\} \\ \{x, \omega x\} &: \{c(X), d(X)\} \end{aligned}$$

For each of these sets, we can define polynomials as

$$\begin{aligned} q_1(X) &= a(X) + x_1 b(X) \\ q_2(X) &= c(X) + x_1 d(X) \end{aligned}$$

where x_1 is sampled randomly to keep a, b, c, d linearly independent. Likewise, we define polynomials

$$r_1(X) = \begin{cases} a(x) + x_1 b(x) & \text{if } X = x \end{cases}$$
$$r_2(X) = \begin{cases} c(x) + x_1 d(x) & \text{if } X = x \\ c(\omega x) + x_1 d(\omega x) & \text{if } X = \omega x \end{cases}$$

and so we can define some f polynomials, similar to what we saw in the vanishing argument:

$$f_1(X) = \frac{q_1(X) - r_1(X)}{X - x}$$
$$f_2(X) = \frac{q_2(X) - r_2(X)}{(X - x)(X - \omega x)}$$

Again, we can sample a random x_2 to combine these into

$$f(X) = f_1(X) + x_2 f_2(X)$$

which we can evaluate at a randomly sampled x_3:

$$\begin{aligned} f(x_3) &= f_1(x_3) + x_2 f_2(x_3) \\ &= \frac{q_1(x_3) - r_1(X)}{x_3 - x} + x_2 \frac{q_2(X) - r_2(X)}{(x_3 - x)(x_3 - \omega x)} \end{aligned}$$

where x_3 is the same point we see in the inner product argument in the following section. Finally, we sample x_4 and construct

$$p(X) = f(X) + x_4 q_1(X) + x_4^2 q_2(X)$$

corresponding to step 19.

This polynomial is what is used for the inner product argument.

3.2.5 Inner Product Argument

The inner product argument essentially allows the prover to show that for a commitment C to a polynomial $a(X)$, a challenge point x, and an evaluation v; we have $a(X) = v$.

This corresponds to a Pedersen vector commitment for $\mathbf{a} \in \mathbb{F}^n$, such that with a public $\mathbf{b} \in \mathbb{F}^n$ and $v \in \mathbb{F}$ it is the case that $\langle \mathbf{a}, \mathbf{b} \rangle = v$. Specifically, in our case, $\mathbf{a}$ is a polynomial $a(X)$ in its coefficient form and $\mathbf{b}$ is a vector on the form $(1, x_3, ..., x_3^{n-1})$. This means that $\langle \mathbf{a}, \mathbf{b} \rangle$ is actually equivalent to the evaluation $a(x_3)$. In the context of the protocol, we know that the evaluation v should be 0.

Let us consider the polynomial of degree $2^k - 1$:

$$p'(X) = p(X) - p(x_3) + \xi s(X)$$

, which is presented in step 23 of the protocol. We will not go into details about $p(X)$ and $\xi s(X)$, but we will note that if the parties have followed the protocol correctly, $s(X)$ should have a root at x_3 and so $p'(X)$ by construction will also have a root at x_3. For this example, we will say that the coefficient form for $p'(X)$ is $\mathbf{p}' = (p_1, p_2)$. Likewise, let us set $\mathbf{b} = (1, x_3)$, fix a basis $\mathbf{G}' = (G_1, G_2)$ with $G_1, G_2 \in \mathbb{G}$ and $U, W \in \mathbb{G}$. We then have a commitment for $p'(X)$ as $P' = \langle \mathbf{p}', \mathbf{G}' \rangle = ([p_1]G_1 + [p_2]G_2)$. Finally, the verifier should send a challenge $z \in \mathbb{F}$ after having received the commitment.

Only the prover should know $\mathbf{p}'$ while the other values are known to both the prover and the verifier. There will be $k = 1$ rounds of interaction, which means there will only be one round for this example. In the first and only round 0, the argument proceeds as follows:

1. The prover calculates and sends

$$L_0 = \langle \mathbf{p}'_{hi}, \mathbf{G}'_{lo} \rangle + [z \langle \mathbf{p}'_{hi}, \mathbf{b}_{lo} \rangle]U + [L_0^*]W$$

   and

$$R_0 = \langle \mathbf{p}'_{lo}, \mathbf{G}'_{hi} \rangle + [z \langle \mathbf{p}'_{lo}, \mathbf{b}_{hi} \rangle]U + [R_0^*]W$$

   (here, L_0^* and R_0^* is the blindness)

2. The verifier responds with a non-zero challenge u_0

3. Both the prover and the verifier set

$$\mathbf{G}'_0 = \mathbf{G}'_{lo} + u_0 \mathbf{G}'_{hi} = G_1 + [u_0]G_2$$

   and

$$\mathbf{b}_0 = \mathbf{b}_{lo} + u_0 \mathbf{b}_{hi} = 1 + u_0 x_3$$

4. The prover sets and sends

$$c = \mathbf{p}'_{lo} + u_0^{-1} \mathbf{p}'_{hi} = p_1 + u_0^{-1} p_2$$

   and

$$f = u_0^{-1} L_0^* + u_0 R_0^*$$

Now the verifier wants to check that $\langle \mathbf{p}', \mathbf{b} \rangle = v = 0$. This is done by the following check

$$[u_0^{-1}]L_0 + P' + [u_0]R_0 \overset{?}{=} [c]\mathbf{G'}_0 + [c\mathbf{b}_0 z]U + [f]W$$

Let us first examine the left-hand side

$$\begin{aligned}
\mathcal{L} =& [u_0^{-1}]L_0 + P' + [u_0]R_0 \\
=& [u_0^{-1}p_2]G_1 + [u_0^{-1}zp_2]U + [u_0^{-1}L_0^*]W + && ([u_0]L_0) \\
& [p_1]G_1 + [p_2]G_2 + && (P') \\
& [u_0p_1]G_2 + [u_0zp_1x_3]U + [u_0R_0^*]W && ([u_0]R_0)
\end{aligned}$$

and the right-hand side

$$\begin{aligned}
\mathcal{R} =& [c]\mathbf{G'}_0 + [c\mathbf{b}_0 z]U + [f]W \\
=& [p_1]G_1 + [u_0p_1]G_2 + [u_0^{-1}p_2]G_1 + [u_0^{-1}u_0p_2]G_2 + && ([c]\mathbf{G'}_0) \\
& [zp_1 + u_0zp_1x_3 + u_0^{-1}zp_2 + u_0^{-1}u_0zp_2x_3]U + && ([c\mathbf{b}_0 z]U) \\
& [u_0^{-1}L_0^* + u_0R_0^*]W && ([f]W) \\
=& [p_1]G_1 + [u_0p_1]G_2 + [u_0^{-1}p_2]G_1 + [p_2]G_2 + && ([c]\mathbf{G'}_0) \\
& [zp_1]U + [u_0zp_1x_3]U + [u_0^{-1}zp_2]U + [zp_2x_3]U + && ([c\mathbf{b}_0 z]U) \\
& [u_0^{-1}L_0^*]W + [u_0R_0^*]W && ([f]W)
\end{aligned}$$

So now we can subtract them and find

$$\begin{aligned}
\mathcal{R} - \mathcal{L} &= [zp_1]U + [zp_2x_3]U \\
&= [zp_1 + zp_2x_3]U \\
&= [z(p_1 + p_2x_3)]U
\end{aligned}$$

We notice that $p_1 + p_2x_3$ is the evaluation $p'(x_3)$, so the equation is satisfied exactly when $p'(X)$ has a root at x_3. If both parties followed the protocol, the verifier should be convinced that P' is a commitment to a polynomial with a root at x_3. Looking at the construction of P' in step 22, this ensures that the polynomial created in the multipoint opening argument $p(X)$ evaluates to v at x_3.

On top of proving this to the verifier, the inner product argument also allows for smaller proofs since we only need to send $2k$ group elements (the L_i's and R_i's) and the field elements c and f. The naive approach would be to send $\mathbf{p}'$, which has 2^k field elements. The gain for this example is not immense, but as it generalizes to much larger values for k, the reduction in size can be increased. Generally, we go from a size linear in the size of p' to a size logarithmic in the size of p'. Furthermore, we also eliminate the need to reveal $\mathbf{p}'$.

We note that for this example, we have omitted some of the accumulated blindness for f. At the time of writing, the Halo 2 book's section on the Inner product argument[19, Sec. 4.1.6] is not completed, so we have primarily used the protocol description[19, Sec. 4.2] and the original halo paper[2] as reference.

3.3 Proof Recursion

In the inner product argument, the verifier needs to compute $\mathbf{G'}_0 = \langle \mathbf{s}, \mathbf{G} \rangle$ and $\mathbf{b}_0 = \langle \mathbf{s}, \mathbf{b} \rangle$, which both are n-length multiexponentiations, and $\mathbf{s}$ is the challenges $u_0, ..., u_{k-1}$ represented in a binary counting structure:

$$s = \begin{pmatrix} 1 \\ u_0 \\ u_1 \\ u_0 u_1 \\ u_2 \\ \vdots \\ u_0 u_1 \cdots u_{k-1} \end{pmatrix}$$

This is an adapted version of what is presented in [2] since the original halo IPA is slightly different.

These calculations would undermine the optimization we have just achieved by using the inner product argument to reduce the communication cost to be logarithmic in n, as they are both linear in n. To reduce the cost of calculating $\mathbf{b}_0$ we realize that we can define the $n-1$ degree polynomial g:

$$g(X, u_0, ...u_k) = \prod_{i=1}^{k} (1 + u_i X^{2^{k-i}})$$

then we can calculate $\mathbf{b}_0$ in logarithmic time as:

$$\mathbf{b}_0 = g(x_3, u_0, ...u_k) = \langle \mathbf{s}, \mathbf{b} \rangle$$

Furthermore, this polynomial g allows us another possibility to express $\mathbf{G'}_0$:

$$\mathbf{G'}_0 = \text{Commit}(\sigma, (g(X, u_0, ...u_{k-1}))$$

where σ is the same common reference string used throughout the protocol. While committing to an n-degree polynomial is also linear in n, we have just shown that we can check an evaluation in logarithmic time. This suggests the following recursion strategy. Instead of the verifier calculating $\mathbf{G'}_0$ itself, some untrusted third party can send it (it could be the prover) and use it as if it was known to be correct (if the rest of the proof is correct) and then accumulates the actual check of it to the next proof in the following way.

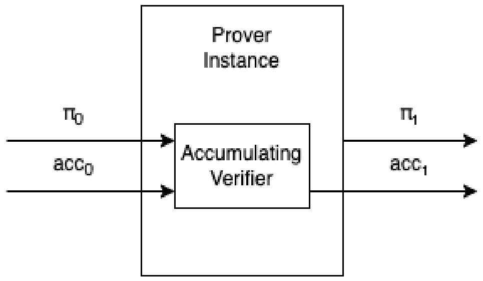


Figure 3.1: Single prove instance in an atomic accumulation scheme

π_0 is the proof from the previous round, π_1 is the proof for this round, acc_0 is the accumulation of the rounds before, and acc_1 is the accumulation for this round and the previous rounds. The accumulating verifier then only does the succinct check, the one including $\mathbf{b}_0$, and then passes the linear check on to the next round within an accumulation of the previous proof (π_0) and accumulation (acc_0). This accumulation works as follows. All π and acc have the same structure:

For $q_0 = \pi_0$ and $q_1 = acc_0$

$$q_i = (C_i, x_i, v_i, s_i, (L_i, R_i, \mathbf{G}_i^{(0)}, p_i^{(0)}))$$

where the last 4-tuple is an IPA proof and $\mathbf{G}^{(0)}$ is equivalent to what we earlier called $\mathbf{G'}_0$.

In each round, the accumulating verifier performs the following steps:

1. Perform succinct check on π_0 and acc_0
2. Generate random α
3. Set $C = \mathbf{G}_0^{(0)} + \alpha \mathbf{G}_1^{(0)}$
   Set $\mathbf{s} = \mathbf{s}_0 + \alpha \mathbf{s}_0$
4. Generate random x
5. Set $v = s(x)$

6. Run IPA with input $(C, \mathbf{s}, v)$ generating new IPA proof $(L, R, \mathbf{G}^{(0)}, p^{(0)})$
7. Set $\mathrm{acc}_1 = (C, x, v, \mathbf{(}s), (L, R, \mathbf{G}^{(0)}, p^{(0)}))$

This can be chained for as many proofs as we want. Each accumulation only costs $\log(n)$ time. When we have accumulated a desired number of proofs, we can run the expensive check that includes the linear $\langle \mathbf{s}, \mathbf{G} \rangle$ multiscalar multiplication for both the last acc and the last π. Due to the way the commitment (C) and challenges ($\mathbf{s}$) are accumulated by the accumulating verifier, the last check convinces the verifier that the prover could only have cheated in any earlier round with negligible probability.

Chapter 4

Our Contributions

4.1 Pasta Curves

For our implementation of the Pasta Curves, we used the blogpost[12] by the Electric Coin Company and the readme accompanying their implementation on github[11], both written by Daira Hopwood. We also leaned a lot on the `hacspec` implementation of the bls12-381 curve, copying and adapting most of their tests and logic.
The Pasta curves consist of two elliptic curves, the Pallas and the Vesta curves. Together these two curves form a curve cycle. This means that the order of each curve is exactly the order of the base field for the other. This is important for the efficiency of the recursive proof system in halo2. In essence, this allows the accumulation to happen over two proof systems on the different curves to efficiently accumulate proofs by sending anything not in their scalar field to the other proof system. Let us say we have proof system $\mathcal{P}_A$ to create a proof in its base field $\mathbb{F}_p$, which would create curve elements with coordinates in $\mathbb{F}_p$. This can be passed to proof system $\mathcal{P}_B$ with base field $\mathbb{F}_q$, and scalar field $\mathbb{F}_p$ and therefore $\mathbb{F}_p$-arithmetic circuit which can efficiently verify the proof from $\mathcal{P}_A$. This way, we can keep switching between $\mathcal{P}_A$ and $\mathcal{P}_B$.

This is great as out-of-field operations are generally very expensive.

This has not been a factor in our project, as we have not specified the accumulation scheme of halo2. However, since we needed an elliptic curve for the commitment scheme, we decided to implement the pasta curves. In our implementation of the halo2 protocol, we only used the Pallas curve.

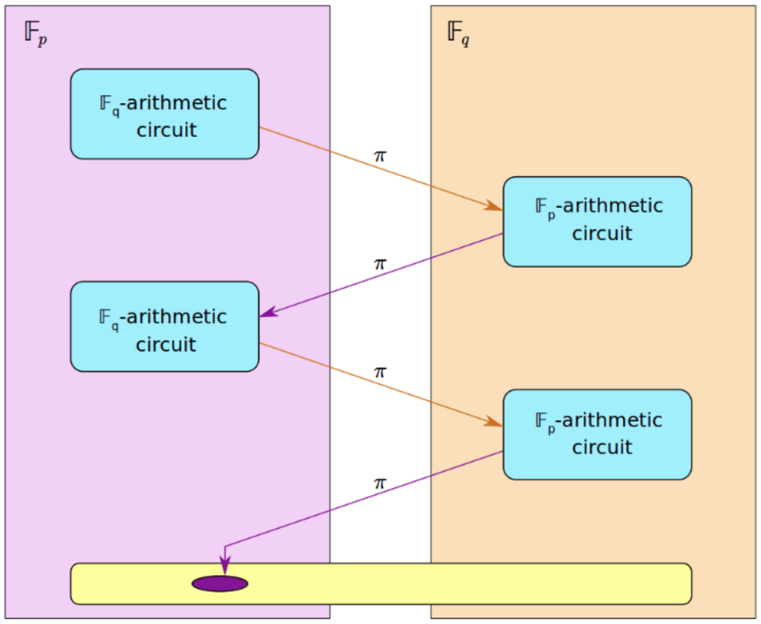

$\mathbb{F}_p$
$\mathbb{F}_q$
$\mathbb{F}_q$-arithmetic circuit
$\mathbb{F}_p$-arithmetic circuit
$\mathbb{F}_q$-arithmetic circuit
$\mathbb{F}_p$-arithmetic circuit
π
π
π
π

Figure 4.1: Simplified illustration from [19, Sec. 5.5] of two proof systems accumulating proves in different fields

The same curve equation defines both the Pallas and the Vesta curve

$$y^2 = x^3 + 5$$

but as mentioned before, over different fields.
Pallas over:

$$GF(40000000000000000000000000000000224698FC094CF91B992D30ED00000001)$$

Vesta over:

$$GF(40000000000000000000000000000000224698FC0994A8DD8C46EB2100000001)$$

where GF is short for Galois field, another name for a finite field.
We implemented this using the `public_nat_mod!()` macro from `hacspec`. This is a `hacspec` safe macro that can be used to define finite fields over a given modulo value.

```
public_nat_mod!(
    type_name: FpPallas,
    type_of_canvas: PallasCanvas,
    bit_size_of_field: 255,
    modulo_value:"
        40000000000000000000000000000000
        224698FC094CF91B992D30ED00000001"
);

public_nat_mod!(
    type_name: FpVesta,
    type_of_canvas: VestaCanvas,
    bit_size_of_field: 255,
    modulo_value:"
        40000000000000000000000000000000
        224698FC0994A8DD8C46EB2100000001"
);
```

The `public_nat_mod!()` comes with a fully implemented algebraic function set for the field elements, including addition, multiplication, division, and inverse. This greatly helped us as we did not need to implement and test these ourselves.

Another thing to consider is that normally elliptic curves would be associated with both a base field and a scalar field. Here the two curves are only defined by their base field. As mentioned earlier, each base field is the other curve's scalar field. If you wanted to do scalar multiplication with a curve point on the Pallas curve, the scalar should have type `FpVesta`.

As there is no specification available for the Pasta curves, we only implemented the defined elliptic curve operations from section 2.5, the additive identity, point-negation, -addition, -doubling, -subtraction, and scalar multiplication. All this was already implemented and tested in the bls12-381 curve implementation, so we copied most of it, only fixing some parameters depending on the size of the fields. This is one of the significant advantages of having a library of examples like `hacspec` does. It is possible to find examples of similar specifications of small primitives like point-negation for elliptic curves or even entire curve definitions. Our initial goal was to make a generic implementation of elliptic curves, over types implementing a Field trait, with basic elliptic curve functions. This could then be extended with curve-specific functions like pairings. While neither traits nor generics are allowed in `hacspec` v1, there was hope that `hacspec`-v2 might be ready in time for us to implement it there. This, however, turned out not to be the case. Instead, we created this specific implementation of the Pasta curves. After submitting a pull request[16] to the `hacspec` repository, we got it merged into the `hacspec` library alongside the existing example specifications.

Testing

As mentioned, all our testing of the arithmetic was copied from the bls12-381 curve implementation. The only thing left to test was the curves' cyclic property. This is tested by counting the order of one curve to see if it matches the order of the other curve's base field. This is done using some variation of Schoofs algorithm[18]. Ideally, we would have a verified implementation of this in `hacspec`. While this is feasible, it is outside the scope of this paper. Instead, we used the Pari/GP[3] tool to find the order of the curves. While this does not test our implementation of the curves exactly, it tests that the cyclic property holds true if the curves are implemented correctly. As expected, this property was upheld.

4.2 Polynomials

`hacspec` has a non `hacspec` compliant crate, `hacspec-lib`, which can still be used in `hacspec` . This library has a generic implementation of polynomials and most of the arithmetic concerning them. However, the `public_nat_mod!()` macro does not implement all the required traits to utilize them. As a result, we could not use the polynomials from `hacspec-lib` and had to implement our own. We once again had the idea of implementing a generic type for polynomials over rings in `hacspec` -v2, but we started by implementing a non-generic version for the FpVesta type. Since we only use commitments in the Pallas curve, we only needed to implement polynomials for Vesta's base field, which is the scalar field of the Pallas curve. We used the coefficient representation, where a polynomial is represented as a list of scalars. The index of the scalar is the power of the indeterminate. Let

$$A = [a_0, a_1, a_2, ..., a_{n-1}]$$

be a list of scalars of the polynomial $f(X)$, then

$$f(X) = \sum_{i=0}^{n-1} A_i X^i$$

This was implemented with a type alias over a sequence.

```
pub type Polyx = Seq<FpVesta>;
```

As mentioned in section 2.9, the `Seq<T>` is a byte array of fixed length used as a `hacspec` alternative to the `Vec<T>` from rust. The `FpVesta` type is a field point from the base field of the Vesta curve. For functionality, we implemented the basic set of arithmetic functions one would expect to be available for polynomials, a few wrapper functions, and some helper functions. Full documentation can be found in appendix C. The only function posing interesting choices was the `divide_polyx(n: Polyx, d: Polyx)` implementing polynomial long division. The following is the pseudocode for polynomial long division from [21]:

```
1: n / d **is**
2: require $d \neq 0$
3: $q \leftarrow 0$
4: $r \leftarrow n$
5: **while** $r \neq 0$ **and** degree(r) $\geq$ degree(d) **do**
6:     $t \leftarrow$ lead(r) / lead(d)
7:     $q \leftarrow q + t$
8:     $r \leftarrow r - t \times d$
9: **end while**
10: **return** (q, r)
```

Here we see the use of a while loop which is not permitted in `hacspec`, as they are not guaranteed to terminate. It is, however, possible to find an upper bound for this while loop, which we then use to replace the while loop with a for loop. The upper bound is found by realizing that we will remove the leading term from the `r` polynomial for each iteration. This is true as we define `t` to be the quotient of the two leading terms of `r` and `d`. In line 8, we subtract `t × d` from `r`. The polynomial `t × d` is guaranteed to include the leading term of `r`. This way, we get an upper bound of the while loop being the degree of the dividend (`n`). Actually, this could be set to the difference between the degree of the two polynomials, as the while loop stops once the degree of `r` is smaller than that of `d`. With this change, we could implement polynomial long division in the following way:

```
pub fn divide_polyx(n: Polyx, d: Polyx) -> (Polyx, Polyx) {
    let mut q: Polyx = Seq::<FpVesta>::create(n.len());
    let mut r: Polyx = n.clone();

    let mut loop_upper_bound = d.len();
    if q.len() > d.len() {
        loop_upper_bound = q.len();
    }
    for _ in 0..loop_upper_bound {
        if check_not_zero_polyx(r.clone())
        && degree_polyx(r.clone()) >= degree_polyx(d.clone()) {
            let t: Polyx =
            divide_leading_terms(r.clone(), d.clone());

            q = add_polyx(q, t.clone());
            let aux_prod: Polyx = mul_polyx(
                d.clone(), t.clone()
            );
            r = sub_polyx(r, aux_prod);
        }
    }

    (trim_polyx(q), trim_polyx(r))
}
```

We also implemented a function for rotating polynomials, a concept introduced in the halo2 book. The idea is that if you are only interested in the evaluations of a polynomial in a given domain, which has a generator ω, you can rotate the polynomial m times by multiplying with ω^m. Here the polynomial $f(X)$ (with coefficients $a_0, ..., a_{n-1}$) is rotated m times.

$$f(\omega^m X) = a_0(\omega^m X)^0 + a_1(\omega^m X)^1 + ... + a_{n-1}(\omega^m X)^{n-1}$$

In halo2, the domain we evaluate polynomials in is generated by the n-th primitive root of unity. These rotations are used to index into the circuit as described in section3.1. We implemented this just as you would expect, but we wanted to include it as it is non-standard.

```
fn rotate_polyx(p: Polyx, rotation: FpVesta) -> Polyx {
    let mut res = p;
    for i in 0..res.len() {
        let coef = res[i];
        let rot = rotation.pow((i as u128));
        res[i] = coef * rot;
    }

    res
}
```

4.2.1 Testing

For most of our tests, we have used `QuickCheck`[8] to do property-based testing. `QuickCheck` allows you to generate random values for types that implement the `Arbitrary` trait, which demands that you implement the `arbitrary` function. For the `Polyx` type our implementation of the `Arbitrary` trait looks like this:

```
#[derive(Clone, Debug, Default)]
struct PolyxContainer(Polyx);

impl Arbitrary for PolyxContainer {
    fn arbitrary(g: &mut quickcheck::Gen) -> PolyxContainer {
        let size = u8::arbitrary(g);
        let mut v: Vec<FpVesta> = vec![];
        for _ in 0..size {
            let new_val: FpVesta =
                FpVesta::from_literal(u128::arbitrary(g));
            v.push(new_val);
        }
        PolyxContainer(Seq::<FpVesta>::from_vec(v))
    }
}
```

Since `Polyx` is just a type alias, we need to wrap it in a container struct as Rust does not allow implementing traits (`Arbitrary`) from outside the crate, for types from outside the crate (`Seq<T>`).
We can then implement the `Arbitrary` trait and the `arbitrary` function for the container struct `PolyxContainer`. The `arbitrary` takes a `QuickCheck` generator `g`, which can be used to generate random elements of different types that imple-

ment the `Arbitrary` trait, like this: `T::arbitrary(g)`. In the arbitrary function for `PolyxContainer`, we use this to generate a random length for the polynomial and then generate random scalars for each entry.

For testing our polynomial implementation, we created two series of tests. One to test the algebraic functionality of our functions and one to test the ring property of our polynomial implementation.

Our functionality tests all followed the same idea: to check each operation against the same operation in the field, with some x.

Let:

$$p_1(X) = p_2(X) \times p_3(X)$$

Check:

$$p_1(x) \overset{?}{=} p_2(x) \times p_3(x)$$

for $\times \in \{+, \cdot, -\}$

This check should hold as is for addition, multiplication, and subtraction. Polynomial division is a little different as we have remainders which is not the case for division in fields. The idea is, however, the same. We divide the evaluated remainder with the evaluated divisor and add the evaluated quotient.

Let:

$$(p_q(X), p_r(X)) = \frac{p_1(X)}{p_2(X)}$$

Check:

$$p_q(x) + \frac{p_r(x)}{p_2(x)} \overset{?}{=} \frac{p_1(x)}{p_2(x)}$$

Our test for `divide_polyx(n: Polyx, d: Polyx) -> (Polyx, Polyx)` looks like this:

```
fn test_poly_div(
    p1: PolyxContainer,
    p2: PolyxContainer,
    x: u128)
{
    let p1 = p1.0;
    let p2 = p2.0;
    let x = FpVesta::from_literal(x);

    let (q, r) = divide_polyx(p1.clone(), p2.clone());
    let eval_q = eval_polyx(q, x);
    let eval_r = eval_polyx(r, x);
    let eval_r_div = eval_r / eval_polyx(p2.clone(), x);

    let expected = eval_polyx(p1, x) /
        eval_polyx(p2.clone(), x);
    let actual = eval_q + eval_r_div;

    assert_eq!(expected, actual);
}
```

For the second series of tests, we tested all the ring axioms [5]. Once again, we used `QuickCheck` to do property-based testing. All functions once again followed the same structure, where we used `QuickCheck` to generate random polynomials and check if the specific axiom held. We also tested if the ring was communicative by testing for the communicative property over multiplication:

$$f(x) \cdot g(x) \overset{?}{=} g(x) \cdot f(x)$$

which it was. These are straightforward and can be found in appendix C for further inspection.

Finally, we made a test for the `rotate_polyx` function. As this is not a common concept in the literature, we used a test from the `halo2` implementation and rewrote it for our implementation. This test generates a random polynomial of degree 7 and a random evaluation point x. It then rotates the polynomial by three different values, 1, ω and ω^{-1}, where ω is an n-th primitive root of unity from the `halo2` test. It then evaluates the rotated polynomials in x and compares them to the original polynomials evaluated in $1x$, ωx, and $\omega^{-1}x$. A transcript of this test can be found in appendix D.

4.3 The halo2 Protocol

The goal of our halo2 implementation is to specify the protocol described in the halo2 book[19, Sec. 4.2]. In this lies some very conscious decisions. The efficient implementation created by the halo2 team includes a lot of optimizations. These optimizations are described in the book and could have been part of our specification. We decided not to do this, as we wanted only to specify the math described in the protocol description. This decision was made because we were interested in the protocol's correctness, as opposed to the optimizations'. We could see new and better optimizations in the future, while the protocol should remain the same. Another approach could also have been to make `hacspec` specifications of the arguments presented in chapter 3. This would have been more in line with earlier specifications in `hacspec`, but as we wanted to explore `hacspec` capabilities, we decided to specify the protocol with all its parts as one complete implementation.

To stay as true as possible to the protocol, we decided to implement each step in the protocol as its own function. Furthermore, we have tried to keep the use of "helper functions" to a minimum, such that the functions can be understood in isolation. This should, in theory, help readability since you do not have to follow function calls around to understand what is going on. We have made an exception to this in step 24 in the first bullet where L_j and R_j are calculated:

- $\mathcal{P}$ sends $L_j = \langle \mathbf{p\prime}_{hi}, \mathbf{G\prime}_{lo} \rangle + [z\langle \mathbf{p\prime}_{hi}, \mathbf{b\prime}_{lo} \rangle]U + [\cdot]W$ and $R_j = \langle \mathbf{p\prime}_{lo}, \mathbf{G\prime}_{hi} \rangle + [z\langle \mathbf{p\prime}_{lo}, \mathbf{b\prime}_{hi} \rangle]U + [\cdot]W$

The calculations of L_j and R_j are very similar and quite long, so we decided that it would help readability to put it in a helper function. We called this function `calculate_L_or_R`:

```
fn calculate_L_or_R(
    p_part: Polyx,
    b_part: Polyx,
    g_part: Seq<G1_pallas>,
    z: FpVesta,
    U: G1_pallas,
    W: G1_pallas,
    blinding: FpVesta,
) -> G1_pallas {
    // <p'_part, G'_part>
    let p_g_msm: G1_pallas = msm(p_part.clone(), g_part);
    let p_b_ip: FpVesta = inner_product(p_part, b_part);
    let z_ip: FpVesta = z * p_b_ip;
    let z_ip_U: G1_pallas = g1mul_pallas(z_ip, U);

    let multed_W: G1_pallas = g1mul_pallas(blinding, W);

    let mut part_j: G1_pallas = g1add_pallas(p_g_msm, z_ip_U);
    part_j = g1add_pallas(part_j, multed_W);

    part_j
}
```

Other than that, we only allowed helper functions for well-defined ideas such as polynomial division, Lagrange interpolation, and vanishing polynomials.
Here is step 5 as presented in the protocol description in the halo2 book:

5. $\mathcal{P}$ computes at most $n-1$ degree polynomials $h_0(X), h_1(X), ..., h_{n_g-2}(X)$ such that $h(X) = \sum_{i=0}^{n_g-2} X^{ni} h_i(X)$.

And here is our implementation of step 5 in `hacspec`:

```
fn step_5(h: Polyx, n: u128, n_g: u128) -> Seq<Polyx> {
    let h = trim_polyx(h);
    let n_g = n_g as usize;
    let n = n as usize;

    let mut index_in_h = 0;
    let mut poly_parts = Seq::<Polyx>::create(n_g - 1);

    for i in 0..n_g - 1 {
        let mut current_poly_part = Seq::<FpVesta>::create(n);
        for j in 0..n {
            if index_in_h < h.len() {
                current_poly_part[j] = h[index_in_h];
                index_in_h = index_in_h + 1;
            }
        }
        poly_parts[i] = current_poly_part;
    }
    poly_parts
}
```

The implementation is not as readable as the short mathematical description from the book. This is, however, not the goal, as the goal is to have readability comparable to forms of pseudocode. The above implementation looks fairly similar to what you would expect from pseudocode, and using this as a reference for future implementation/optimization should be intuitive. Three things that stand out from a readability standpoint are

1. the curly brackets; and
2. the cumbersome `create` statement of `Seq<T>`
3. Function calls instead of mathematical operators

There is not much to do about the brackets. However, they are common in conventional programming languages and make the specification valid Rust, an essential idea of `hacspec`.

The need for the `create(l: usize)` statement is primarily a limitation of `hacspec`. Where you might usually just `pop` or `push` to some form of vector or list, `Seq<T>` needs the length at creation time.

Finally, in the above example, field elements are added together using the + operator. This is possible due to the `public_nat_mod!()` macro, which implements the necessary traits to allow the usage of such operators. This is not possible for custom types, such as group elements from the elliptic curves. Version 1 of `hacspec` does not allow implementing traits, which would have allowed this. Consider the following example for two elliptic curve group elements, G_1 and G_2

$$G_1 + G_2$$

is expressed as something akin to

```
g1add_pallas(g1, g2)
```

It suffers from some terseness, and we also involve the name of the specific curve used.

The halo2 protocol makes use of quite a bit of randomness, both in the form of challenges from the verifier and blindness for the Pedersen commitments. We do not use randomness in our specification for two reasons. Firstly `hacspec` does not allow randomness, as it, of course, is nondeterministic. Secondly, randomness might not be desirable in a reference implementation. What we do instead is to take all needed randomness as an argument in the different functions. This way, when using the `hacspec` specification as a reference implementation or testing up against your own code, you can fix the randomness and pass it to the `hacspec` function. That way, you can be sure the inputs are the same; therefore, the behavior should be the same. This also leads us to a rather unconventional implementation of steps 15, 16, and 17. All that happens is the verifier creates and sends challenges to the prover.

15. $\mathcal{V}$ responds with challenge x_3.

As we wanted to stay as true to the protocol as possible, we have implemented this as an identity function:

```
fn step_15(x_3: FpVesta) -> FpVesta {
    x_3
}
```

Finally, we decided to let the verifier and the prover act in the same function call. This is not optimal from a cryptographic standpoint where you would want the two to have separate views. As our focus is on specifying the protocol and its math and not actual application of the system, we consider this acceptable.

After discussing with the halo2 team, we decided not to include the first three steps of the protocol in our specification. This was done as these steps mainly relate to the arithmetization of PLONKish circuits, which is outside the scope of this paper.

In the halo2 book, the protocol makes use of a helper function $\sigma(i)$ defined as:

For all $i \in [0, n_a)$:

- Let $\mathbf{p}_i$ be the exhaustive set of integers j(modulo n) such that $a_i(\omega^j X, \cdots)$ appears as a term in $g(X, \cdots)$.
- let $\mathbf{q}$ be a list of distinct sets of integers containing $\mathbf{p}_i$ and set $\mathbf{q}_0 = \{0\}$
- let $\sigma(i) = \mathbf{q}_j$ when $\mathbf{q}_j = \mathbf{p}_i$

In other words, when passing an argument i to the sigma function, it should return a list of all the rotations of $a_i(X)$ present in $g(X, \cdots)$. We have implemented this as follows:

```
fn sigma(
    i: u128, sigma_list: Seq<u128>, q: Seq<Seq<u128>>
) -> Seq<u128> {
    let idx = sigma_list[i as usize];
    q[idx as usize].clone()
}
```

As we do not have a sense of state, we have to pass both the list `q` and `sigma_list` to the function. These have to be created manually beforehand as they depend on the arithmetization from PLONKish. `sigma_list` is a list of integers such that $\text{sigma\_list}_i = j$ when $\mathbf{q}_j = \mathbf{p}_i$.

4.3.1 Testing

The protocol itself proved rather challenging to test systematically. On the surface, it is composed of somewhat straightforward math without many obvious invariants. The underlying implementations of elliptic curves and polynomials have, of course, been tested thoroughly, as described in previous sections, but testing the logic described in the protocol quickly turned into retesting the underlying math. Because of this, we hoped to find some unit-tests in the efficient halo2 implementation, which we could rewrite to test our code. Unfortunately, most of their tests were testing either their PLONKish arithmetization or their optimization of the protocol.

Instead, we approached testing with the following strategies:

1. Small isolated tests
    (a) Unit tests, with fixed value
    (b) Automated tests
2. Cross-testing/Integration tests
3. Complete run of the protocol

This way, testing began with the small and isolated parts of the code and gradually turned to testing increasingly more integrated parts of the code.

Small Isolated Tests

The most basic approach was a unit test with predetermined inputs, where we calculated the result manually and compared it against our implementation of the given step. This served as a sanity check and confirmed our understanding. For some of the unit tests, we followed up with automated versions.

The automated tests were essentialy re-implementations of the function. These re-implementations were written with `QuickCheck` so it could be run with a range of random inputs. We ensured that whoever had written the implementation of the function did not write the re-implementation for the test. The idea behind this was that we might be able to catch some errors in our original implementation by rewriting with a fresh mind. Some of the automated tests were adapted version of tests with fixed inputs.

We explored using these different types of tests, so they have been used somewhat interchangably.

After these tests, we were somewhat confident in the functions' correctness from an isolated view. Consequently, we went on to start testing their integration and test the combination of larger parts of the protocol.

Cross-testing

The third approach was "cross-testing" where we tested properties across multiple steps of the protocol. Furthermore, we test across the view of the verifier and the prover. By doing this, we were able to find some properties we could use for property-based testing.

An example of this is steps 5 to 8. Looking at the protocol description[19, 4.2, Protocol], it can be seen that steps 5, 6, 7, and 8 are closely related. As such, testing that this relation holds for the specification is important. These relations we capture generally involve the prover knowing some polynomial $a(X)$, which the verifier only knows a commitment A for. We present a small example of how to test that A is a commitment for $a(X)$ with some specific randomness, or blindness, using the Pedersen vector commitment scheme.

First, we have to find a concrete way to associate the involved values.
Let us start by outlining the concerned values

- From step 5, we have $h_0(X), \dots, h_{n_g-2}(X)$ (and implicitly the vectors of coefficient $\mathbf{h}_0, \dots, \mathbf{h}_{n_g-2}$)
- From step 6, we have $H_i = \langle \mathbf{h_i}, \mathbf{G} \rangle + [\cdot]W$ for all $\mathbf{h}_i$
- From step 7, we have x and $H' = \sum_{i=0}^{n_g-2} [x^{ni}] H_i$
- From step 8, we have $h'(X) = \sum_0^{n_g-2} x^{ni} h_i(X)$

We start by exploring H'. Let us say we use randomness r_i for the commitment in the ith summand. We see that

$$\begin{aligned} H' = \sum_{i=0}^{n_g-2} [x^{ni}] H_i &= \sum_{i=0}^{n_g-2} [x^{ni}](\langle \mathbf{h}_i, \mathbf{G} \rangle + [r_i]W]) \\ &= \sum_{i=0}^{n_g-2} \big([x^{ni}] \langle \mathbf{h}_i, \mathbf{G} \rangle + [x^{ni} r_i] W \big) \\ &= \sum_{i=0}^{n_g-2} [x^{ni}] \langle \mathbf{h}_i, \mathbf{G} \rangle + \sum_{i=0}^{n_g-2} [x^{ni} r_i] W \\ &= \left\langle \sum_{i=0}^{n_g-2} x^{ni} \mathbf{h}_i, \mathbf{G} \right\rangle + \left[\sum_{i=0}^{n_g-2} x^{ni} r_i \right] W \end{aligned}$$

Now, let us look at a commitment to $h'(X)$. We recall Definition 23, which gives us a commitment to $h'(X)$ using some randomness r' and the same values for σ as above:

$$\text{Commit}(\sigma, h'(x); r') = \left\langle \sum_{i=0}^{n_g-2} x^{ni}\mathbf{h}_i, \mathbf{G} \right\rangle + [r']W$$

Let us set $r' = \sum_{i=0}^{n_g-2} x^{ni} r_i$. Comparing with what we found above, we now see that this is equal to H':

$$\text{Commit}\left(\sigma, h'(x); \sum_{i=0}^{n_g-2} x^{ni} r_i\right) = \left\langle \sum_{i=0}^{n_g-2} x^{ni}\mathbf{h}_i, \mathbf{G} \right\rangle + \left[\sum_{i=0}^{n_g-2} x^{ni} r_i\right] W = H'$$

This means we can test these steps' interdependence by running steps 5 to 8, then commit to $h'(X)$ with the "randomness" r' as above, and check that this commitment corresponds with H'. This corresponds to the accumulated blindness of the protocol. Looking at the protocol description with our annotations (appendinx B), it can be seen that these values matches.

We have similair tests, where we compare polynomials and their purported commitments, for

- Steps 11 and 12
- Steps 18 and 19
- Steps 22 and 23

These tests depend on previous values of the protocol, so they iteratively include more and more of the protocol. In the end we have essentialy included the whole protocol, which we present next.

Test Run

Finally, we made a test run of the complete protocol. This functions as a test for the entire protocol. The circuit we used was a simple circuit of just two addition gates:

i	a_0	a_1	a_2	q_{add}
0	2	3	4	1
1	10			0
2	5	8	13	1
3	26			

As we have not implemented the PLONKish arithmetization nor steps 1-3 in the protocol, we have to manually calculate a valid $g'(X)$. First, we created the $a_0(X), a_1(X), a_2(X)$ and $q_{add}(X)$ polynomials using Lagrange interpolation through the points specified in the circuit table. For example $a_0(X)$ was determined by Lagrange interpolation through $[(\omega^0, 2), (\omega^1, 10), (\omega^2, 5), (\omega^3, 26)]$. From these polynomials we set $g(X) = q_{add}(X) \cdot (a_0(X) + a_1(X) + a_2(X) - a_0(\omega X))$. We omit all the the challenges $c_0, ...c_{n_a-1}$ such that $a_i'(X) = a_i(X)$ and $g'(X) = g(X)$. The final thing we need to compute before the protocol is the $\mathbf{p}$ and $\mathbf{q}$ lists used by the sigma function. First we set $\mathbf{p} = [[0, 1], [0], [0]]$, where $\mathbf{p}_0$ is the list of rotations $a_0(X)$ is evaluated in. From $g(X)$ we see that $a_0(X)$ is the only one that is rotated, and therefore $\mathbf{p}_1$ and $\mathbf{p}_2$ are both $[0]$. Then we set $\mathbf{q} = [[0], [0, 1]]$, which is the list of unique lists from $\mathbf{p}$ with $\mathbf{q}_0 = [0]$. With this, we have all the inputs needed to start the protocol. Throughout the protocol, more inputs are needed in the form of blinding factors, challenges, and random polynomials. These are all selected according to the requirements stated in the protocol.

This test was critical as it did not pass at first. While it was hard to locate the errors causing this to fail (it depended on the entire implementation), we narrowed it down by iteratively letting some of the inputs be 0. We got it narrowed down to be an error with our calculation of f in step 25. The calculation of this variable was not included in the protocol, but after discussing with the halo2 team, we agreed that it should be included. Through a process we touch more on in section 4.5, we determined the calculation of f and sent a pull request with our changes to the halo2 protocol description. As this specification of the calculation for f was being reviewed by halo2 members, it turned out that we had missed part of the calculation. This shows the importance of proper testing. Not only had we made a critical omission in our implementation, but we had also tried to get an incomplete calculation of f merged into the halo2 book. After iteratively setting different blinding factors to 0, we were able to narrow the error down to the blinding factors used in steps 1, 3, and 6. After looking at how the accumulation of blinding factors (the calculation of f) is used to align the polynomials on the prover's side with the commitments on the verifier's side, we realized that we were missing some blindings and narrowed it down to step 12 of the protocol. Specifically, we did not update the q_0^* in bullet 2. Our calculation of f can be seen in the notes column of appendix B. After mitigating this omission, our test passed, and we sent the update to the halo2 team.

Based on this test run, we created some `QuickCheck` tests testing both some illegal and legal inputs. First, we made a positive test to test different inputs to the same circuit making sure the constraints were upheld in the following manner:

i	a_0	a_1	a_2	a_{add}
0	rnd_1	rnd_2	rnd_3	1
1	$rnd_1 + rnd_2 + rnd_3$			0
2	rnd_4	rnd_5	rnd_6	1
3	$rnd_4 + rnd_5 + rnd_6$			

We also created a positive test with random inputs for blinding and one with random inputs for challenges. Finally, we also made a negative test where all the inputs to the circuit were generated randomly. Specifically, the values at $(a_0, 1)$ and $(a_0, 3)$ were random and not calculated from the others. This means that the circuit would be satisfied with a very low probability.

4.4 halo2 Specification in Rustdoc

To make the specification more accessible, we have used the tool `rustdoc`, which can render a document with documentation for rust (or `hacspec`) code and arbitrary markdown. Rust constructs annotated with `rustdoc` comments are automatically included, so we have documented all functions so they can easily be viewed in the `rustdoc` document. As an example of this, we have our inner product function:

```
/// Inner product, between two equal-length vectors of field elements
///
/// # Arguments
///
/// * `u` - LHS vector
/// * `v` - RHS vector
fn inner_product(u: Polyx, v: Polyx) -> FpVesta {
    let mut res = FpVesta::ZERO();
    for i in 0..u.len() {
        res = res + u[i] * v[i];
    }

    res
}
```

The comments preceding the function signature are the `rustdoc` annotation. The first line is a high-level description of the function. Following is a markdown header "Arguments" and a list of arguments and their descriptions. Notice how this is just regular markdown.

As stated, it is also possible to include arbitrary markdown content by using the doc attribute

```
#![doc = include_str!("../table.md")]
```

Here we include a markdown file called `table.md`, which contains the complete halo2 protocol description. We have used a table where each row corresponds to a step in the protocol. For each of these steps, we have also included a reference to the corresponding `hacspec` specification and a link to the corresponding part of the efficient implementation. This way, we hope to make it easier to get an overview and compare our specification with the efficient implementation. Finally, there is also a "Notes" column. Here we have added some omissions from the protocol, specifically regarding how blindness is used and accumulated through the protocol. This has been instrumental in getting an overview of what we believe should be added to the protocol description and helped communicate and discuss these ideas with the halo2 developers. Our `rustdoc` document (appendix E) has been made publicly available with the use of GitHub Pages[1], which allows for free hosting of static content in a git repo.

Finally, the protocol description is written in markdown but uses a lot of latex math-mode. To be able to render this with `rustdoc`, we used "KaTeX"[2], which can generate math symbols, etc., to be viewed with a web browser. The precise procedure to make this possible was greatly inspired by the rust crate `rustdoc_katex_demo`[14]. Essentially, we have included a katex html header, where we also defined the latex macros that the protocol description uses. This header file is then placed in the root of the rust workspace. We can then generate the final document with the `cargo doc` command. We have to adjust the environment variable `RUSTDOCFLAGS` to utilize katex. One way to do this is by running the following commands:

```
# bash or similar
export RUSTDOCFLAGS="--html-in-header katex-header.html"
cargo doc --no-deps --document-private-items # or any desired flags
```

This will place the generated HTML in a directory `doc` under the rust `target` directory. This can be viewed locally in a browser, or hosted publicly, as we did.

4.5 Changes to halo2 book

While implementing the halo2 protocol[19, 4.2, Protocol], we communicated with the halo2 team in their open community Discord channel, where we had a dedicated thread. Through this discussion, we were able to clarify some uncertainties we had about the protocol, and in that process, we were also able to point out aspects of the protocol that might need to be changed. This could be due to actual errors, typos, or to improve exposition. As another result of this discussion, the halo2 team also detected some discrepancies between the protocol description and the efficient implementation in steps 18 and 19. These changes are part of the pull request [17]. The current version from the book (appendix A) can also be compared with our version with these changes and notes (appendix B).

Here the `rustdoc` specification again proved very useful as a communicative tool, as we were able to illustrate proposed changes to the protocol. One of these

[1]https://pages.github.com/ (accessed 2023-06-14)
[2]https://katex.org/ (accessed 2023-06-14)

proposals was to include the calculation of the variable f introduced in step 25. The variable was described as

> synthetic blinding factor f [is] computed from the elided blinding factors

in the protocol description. The decision not to include the calculation was made because it would allow eliding the blinding factors for exposition. While this elision helps the readability of the protocol, it also makes it impossible to describe the calculation of f, as the variables used in the calculation were not defined. After discussion with the halo2 team, we agreed that it would be suitable to include the calculation and the needed blinding factors explicitly named in the protocol. As a result, we sent a pull request with these changes [17]. However, as mentioned in section 4.3.1, there were some missing factors with the calculation of f. We once again presented these omissions to the halo2 team using our `rustdoc` as a medium for showing the corrections, and they were eventually merged into the earlier pull request.

4.6 Development of hacspec-v2

While we did not get to do any implementation specifically in `hacspec`-v2, we did get to try the early stages of the tools `into`, `linter`, and `json`. The `into` tool translates the Rust crate into Coq, fstar, or easycrypt. The `linter` tool has two options `hacspec`, which checks if your Rust code is in the Hacspec subset, that is, a simple enough subset of Rust suited for specifications, or `Rust`, which makes a fast check whether translation into a backend is possible, as the `into` tool is too slow for using in language server protocols. The `json` creates a typed AST of the Rust crate as a JSON file. Our halo2 implementation, including Pasta Curves and polynomial implementation, successfully ran the `lint` without any warnings. We were also able to extract the typed AST using the `JSON` tool. The engine to translate into a backend was, however, not fully implemented yet, and we ran into a problem with our for loop in our implementation of polynomial division. We reported the issue in the `hacspec`-v2 repository[6].

Chapter 5

Conclusion

This thesis presents our high-assurance (`hacspec`) specification for halo2's interactive argument of knowledge. `hacspec` , as a subset of Rust, is relatively similar to conventional programming languages, which is a great strength, but some features have been left out due to its rigorous nature. In particular, randomness is not allowed and has to be supplied from outside of `hacspec` . Similarly, only top-level functions are allowed, which restricts the possible model of interaction, which is at the core of an interactive argument. Since our focus has been on correctness and specification, this perceived inaccuracy has not been a problem for the paper.

`hacspec` has also made translating the protocol description to a specification relatively intuitive, even if the specification is not exactly as readable as the description. This partly comes from the fact that `hacspec` (version 1) does not allow implementing traits, which means that mathematical operators for custom types have been expressed as function calls. For instance, the addition of group elements $G_1 + G_2$ is expressed as `g1add_pallas(g1, g2)`. Version 2 of `hacspec` should improve on some of these aspects, which would be a natural development for future work on the specification.

Through the work with the protocol specification, we believe we have added some value to the protocol description. Initially, we believed the protocol description to be the authoritative source and expected that if any modifications were to happen, it would be in the efficient implementation. As we found out during the project, it turned out that the halo2 team adapted the protocol description to fit the efficient implementation instead. Through discussion with the developers, we have suggested additions to the description that we believe are key to the protocol, the specification, and anyone interested in alternative implementations of halo2. Practically, this has been done by bringing the protocol description closer to the efficient implementation, which in some sense has been the authoritative source.

As such, our specification should serve as a one-to-one mapping for the protocol description.

Finally, we have, through the work with this paper, created a high assurance specification of steps 4 through 26 of the halo2 protocol, a high assurance specification of a polynomial ring specifically over the scalar field of the Pallas curve, and a high assurance specification of the Pasta curves. The last of which has already been published in the `hacspec` library.

Chapter 6

Related Work in `hacspec`

In this chapter, we will briefly describe some other projects to be specified in `hacspec` and compare them to our specification of halo2. Looking through the examples included in the `hacspec` library[9], we have found two specifications that can be considered the largest (to the best of our knowledge). These two turned out to be the specifications of Gimli and bls12-381-hash to curve.

In an effort to compare the sizes of our specifications with these other specifications, we will present some measurements. Measuring the size of software is challenging, but we hope the measurements used can give some indication. These measurements are

- Number of non-blank lines for all source code of the given specification
- The size of all source code, after being ZIP compressed

Since some of the specifications include many constant declarations (using the `const` keyword), tests and constants for testing, we also show these measurements where they are excluded. In some way, this should represent the actual logic and code that went into the specification.

	gimli	bls12-381-hash	halo2	pasta
Zip size (KiB)	1.9	4.7 (9.7 with constants)	9.4	1.5
Non-blank lines	206	582 (1182 with constants)	1199	189

Size excluding tests and constant declerations

	gimli	bls12-381-hash	halo2	pasta
Zip size (KiB)	78.7	16.8	23.8	3.0
Non-blank lines	6989	2046	4889	468

Total size

This data can be interpreted in several ways, but it should underline that the halo2 specification is among the more extensive projects. If tests and constants are excluded, halo2 could be considered the largest. Pasta is also somewhat extensive. It is worth again pointing out that the Pasta specification includes both the Pallas and Vesta curves, which are essentially adapted versions of the bls12-381 curve specification.

Chapter 7

Future Work

In some sense, this has been a project of feasibility, exploring `hacspec`'s ability to specify more extensive protocols such as halo2. This project could have gone in many directions, and naturally, we could not pursue all of them. Therefore, we will present some of the projects that were outside our scope as proposals for future work. These proposals can be categorized as either halo2 or `hacspec` related.

As we have focused on specifying the proof protocol of the halo2 zk-SNARK, it would be interesting to look into specifying the entire zk-SNARK. This would include the PLONKish arithmetization before the protocol and the recursive proving system described in section 3.3. The specification of PLONKish would be a vast project, as it includes not only the arithmetization of a circuit but also the creation of an environment for writing gates and planning circuit layouts. Having this would also allow the specification of the entire protocol, including the first three steps, which we have omitted. Specifying the recursive proof system and the accumulation scheme would also be an interesting project, which might also be suitable for use in other specifications.

Another interesting project would be to look into generating random circuits for testing. This would allow us to give a higher assurance specification compared to the tests we use now, which only work on predefined circuits. This could possibly be done using something like Noir[1] from Aztec. Noir is an open-source programming language for writing and verifying zero-knowledge proofs, which uses the UltraPlonk backend, which resembles PLONKish a lot.

The final halo2-specific proposal would be to further integrate the specification with the efficient implementation. This would be done by making some integration tests where we would test our specification against the efficient implementation. This could be done by either testing entire protocol runs against each other or testing smaller parts. If any discrepancies were found, it would raise an interesting question as to which result is correct. As we have based our implementation solely on the protocol description from the halo2 book, such discrepancies should indicate differences between the protocol description and the efficient implementation. Going into the project, we expected the protocol description to be the authoritative source. Throughout the project, we have, however, seen the halo2 team make corrections to the protocol descriptions as a result of discrepancies between that and the efficient implementation.

As for future work with `hacspec`, the generic implementations of both the elliptic curves and the polynomial rings are natural suggestions. Most of the logic is already there; the only thing missing is the integration of generics and traits in `hacspec`. As of writing this, `hacspec`-v2 is very close to feature parity with `hacspec`. When that is done, generics for both elliptic curves and polynomial rings should be possible.

With the testing of the Pasta curves, we discovered the need for a verified implementation of Schoof's algorithm to determine the order of an elliptic curve group. This would also be an interesting project, as it would be a useful primitive to have in the `hacspec` library.

Bibliography

[1] Aztec. *Aztec - Noir*. [Online; accessed 2023-06-01]. URL: `https://aztec.network/noir/`.

[2] Sean Bowe, Jack Grigg, and Daira Hopwood. *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Paper 2019/1021. `https://eprint.iacr.org/2019/1021`. 2019. URL: `https://eprint.iacr.org/2019/1021`.

[3] PARI/GP Development. *PARI/GP*. [Online; accessed 2023-06-01]. URL: `https://pari.math.u-bordeaux.fr/`.

[4] Al Doerr and Ken Levasseur. *Polynomial Rings*. [Online; accessed 2023-05-28]. Aug. 2021.

[5] Al Doerr and Ken Levasseur. *Rings, Basic Definitions and Concepts*. [Online; accessed 2023-06-01]. Aug. 2021.

[6] Lucas Franceschino. *for loop not resugared #105*. [Online; accessed 2023-06-01]. 2023. URL: `https://github.com/hacspec/hacspec-v2/issues/105`.

[7] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. `https://eprint.iacr.org/2019/953`. 2019. URL: `https://eprint.iacr.org/2019/953`.

[8] Andrew Gallant. *GitHub repository BurntSushi/quickcheck: Automated property based testing for Rust (with shrinking)*. [Online; accessed 2023-05-28]. URL: `https://github.com/BurntSushi/quickcheck`.

[9] hacspec. *GitHub repository âhacspec/examples*. [Online; accessed 2023-03-11]. URL: `https://github.com/hacspec/hacspec/tree/master/examples`.

[10] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[11] Daira Hopwood. *GitHub repository zcash/pasta: Generator and supporting evidence for security of the Pallas/Vesta pair of elliptic curves suitable for Halo*. [Online; accessed 2023-05-28]. Nov. 2020. URL: `https://github.com/zcash/pasta`.

[12] Daira Hopwood. *The Pasta Curves for Halo 2 and Beyond.* [Online; accessed 2023-05-28]. Nov. 2020. URL: `https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond`.

[13] Jesper Buus Nielsen Ivan Damgård and Sophia Yakoubov. *Zero Knowledge Proofs and Arguments.* 2022.

[14] Paul Kernfeld. *Crate: rustdoc_katex_demo.* [Online; accessed 2023-06-01]. URL: `https://github.com/paulkernfeld/rustdoc-katex-demo`.

[15] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust.* Technical Report. Inria, Mar. 2021. URL: `https://inria.hal.science/hal-03176482`.

[16] Lasse Schmidt and Rasmus Bjerg. *Pasta curves specification #1.* [Online; accessed 2023-06-13]. 2023. URL: `https://github.com/hacspec/specs/pull/1`.

[17] Lasse Schmidt and Rasmus Bjerg. *Patch protocol p poly#779.* [Online; accessed 2023-06-01]. 2023. URL: `https://github.com/zcash/halo2/pull/779`.

[18] René Schoof. "Counting points on elliptic curves over finite fields." In: *Journal de théorie des nombres de Bordeaux* 7.1 (1995), pp. 219–254.

[19] the zcash team. *The halo2 book.* [Online; accessed: 17-March-2023]. URL: `https://zcash.github.io/halo2/`.

[20] the zcash team. *What are zk-SNARKs?* [Online; accessed: 17-March-2023]. URL: `https://z.cash/technology/zksnarks/`.

[21] Wikipedia contributors. *Polynomial long division — Wikipedia, The Free Encyclopedia.* [Online; accessed 31-May-2023]. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Polynomial_long_division&oldid=1154893948`.

[22] Zooko Wilcox and Ian Sagstetter. *Halo's contribution goes beyond efficiency.* [Online; accessed: 19-May-2023]. Nov. 2022. URL: `https://electriccoin.co/blog/halos-contribution-goes-beyond-efficiency/`.

[23] ZKProof. *ZKProof Community Reference. Version 0.3.* July 2022. Updated versions at https://docs.zkproof.org/reference.

Appendix

A halo2 Protocol Description

See next page. (from [19])

1. $\mathcal{P}$ and $\mathcal{V}$ proceed in the following n_a rounds of interaction, where in round j (starting at 0)

- $\mathcal{P}$ sets $a'_j(X) = a_j(X, c_0, c_1, ..., c_{j-1}, a_0(X, \cdots), ..., a_{j-1}(X, \cdots, c_{j-1}))$
- $\mathcal{P}$ sends a hiding commitment $A_j = \langle \mathbf{a'}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{a'}$ are the coefficients of the univariate polynomial $a'_j(X)$ and $\cdot$ is some random, independently sampled blinding factor elided for exposition. (This elision notation is used throughout this protocol description to simplify exposition.)
- $\mathcal{V}$ responds with a challenge c_j.

2. $\mathcal{P}$ sets $g'(X) = g(X, c_0, c_1, ..., c_{n_a - 1}, \cdots)$.
3. $\mathcal{P}$ sends a commitment $R = \langle \mathbf{r}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{r} \in \mathbb{F}^n$ are the coefficients of a randomly sampled univariate polynomial $r(X)$ of degree $n - 1$.
4. $\mathcal{P}$ computes univariate polynomial $h(X) = \frac{g'(X)}{t(X)}$ of degree $n_g(n-1) - n$.
5. $\mathcal{P}$ computes at most $n - 1$ degree polynomials $h_0(X), h_1(X), ..., h_{n_g - 2}(X)$ such that $h(X) = \sum\limits_{i=0}^{n_g - 2} X^{ni} h_i(X)$.
6. $\mathcal{P}$ sends commitments $H_i = \langle \mathbf{h_i}, \mathbf{G} \rangle + [\cdot]W$ for all i where $\mathbf{h_i}$ denotes the vector of coefficients for $h_i(X)$.
7. $\mathcal{V}$ responds with challenge x and computes $H' = \sum\limits_{i=0}^{n_g - 2} [x^{ni}] H_i$.
8. $\mathcal{P}$ sets $h'(X) = \sum\limits_{i=0}^{n_g - 2} x^{ni} h_i(X)$.
9. $\mathcal{P}$ sends $r = r(x)$ and for all $i \in [0, n_a)$ sends $\mathbf{a_i}$ such that $(\mathbf{a_i})_j = a'_i(\omega^{(\mathbf{p_i})_j} x)$ for all $j \in [0, n_e - 1]$.
10. For all $i \in [0, n_a)$ $\mathcal{P}$ and $\mathcal{V}$ set $s_i(X)$ to be the lowest degree univariate polynomial defined such that $s_i(\omega^{(\mathbf{p_i})_j} x) = (\mathbf{a_i})_j$ for all $j \in [0, n_e - 1)$.
11. $\mathcal{V}$ responds with challenges x_1, x_2 and initializes $Q_0, Q_1, ..., Q_{n_q - 1} = \mathcal{O}$.

- Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{V}$ sets $Q_{\sigma(i)} := [x_1] Q_{\sigma(i)} + A_i$.
- $\mathcal{V}$ finally sets $Q_0 := [x_1^2] Q_0 + [x_1] H' + R$.

12. $\mathcal{P}$ initializes $q_0(X), q_1(X), ..., q_{n_q - 1}(X) = 0$.

- Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{P}$ sets $q_{\sigma(i)} := x_1 q_{\sigma(i)} + a'(X)$.
- $\mathcal{P}$ finally sets $q_0(X) := x_1^2 q_0(X) + x_1 h'(X) + r(X)$.

13. $\mathcal{P}$ and $\mathcal{V}$ initialize $r_0(X), r_1(X), ..., r_{n_q - 1}(X) = 0$.

- Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{P}$ and $\mathcal{V}$ set $r_{\sigma(i)}(X) := x_1 r_{\sigma(i)}(X) + s_i(X)$.
- Finally $\mathcal{P}$ and $\mathcal{V}$ set $r_0 := x_1^2 r_0 + x_1 h + r$ and where h is computed by $\mathcal{V}$ as $\frac{g'(x)}{t(x)}$ using the values $r, \mathbf{a}$ provided by $\mathcal{P}$.

14. $\mathcal{P}$ sends $Q' = \langle \mathbf{q'}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{q'}$ defines the coefficients of the polynomial

$$q'(X) = \sum_{i=0}^{n_q-1} x_2^i \left(\frac{q_i(X) - r_i(X)}{\prod_{j=0}^{n_e-1} \left(X - \omega^{(\mathbf{q_i})_j} x \right)} \right)$$

15. $\mathcal{V}$ responds with challenge x_3.
16. $\mathcal{P}$ sends $\mathbf{u} \in \mathbb{F}^{n_q}$ such that $\mathbf{u}_i = q_i(x_3)$ for all $i \in [0, n_q)$.
17. $\mathcal{V}$ responds with challenge x_4.
18. $\mathcal{V}$ sets $P = Q' + x_4 \sum_{i=0}^{n_q-1} [x_4^i] Q_i$ and $v =$

$$\sum_{i=0}^{n_q-1} \left(x_2^i \left(\frac{\mathbf{u}_i - r_i(x_3)}{\prod_{j=0}^{n_e-1} \left(x_3 - \omega^{(\mathbf{q_i})_j} x \right)} \right) \right) + x_4 \sum_{i=0}^{n_q-1} x_4 \mathbf{u}_i$$

19. $\mathcal{P}$ sets $p(X) = q'(X) + [x_4] \sum_{i=0}^{n_q-1} x_4^i q_i(X)$.
20. $\mathcal{P}$ samples a random polynomial $s(X)$ of degree $n - 1$ with a root at x_3 and sends a commitment $S = \langle \mathbf{s}, \mathbf{G} \rangle + [\cdot] W$ where $\mathbf{s}$ defines the coefficients of $s(X)$.
21. $\mathcal{V}$ responds with challenges ξ, z.
22. $\mathcal{V}$ sets $P' = P - [v] \mathbf{G}_0 + [\xi] S$.
23. $\mathcal{P}$ sets $p'(X) = p(X) - p(x_3) + \xi s(X)$ (where $p(x_3)$ should correspond with the verifier's computed value v).
24. Initialize $\mathbf{p'}$ as the coefficients of $p'(X)$ and $\mathbf{G'} = \mathbf{G}$ and $\mathbf{b} = (x_3^0, x_3^1, ..., x_3^{n-1})$. $\mathcal{P}$ and $\mathcal{V}$ will interact in the following k rounds, where in the jth round starting in round $j = 0$ and ending in round $j = k - 1$:

- $\mathcal{P}$ sends $L_j = \langle \mathbf{p'}_{\text{hi}}, \mathbf{G'}_{\text{lo}} \rangle + [z \langle \mathbf{p'}_{\text{hi}}, \mathbf{b}_{\text{lo}} \rangle] U + [\cdot] W$ and $R_j = \langle \mathbf{p'}_{\text{lo}}, \mathbf{G'}_{\text{hi}} \rangle + [z \langle \mathbf{p'}_{\text{lo}}, \mathbf{b}_{\text{hi}} \rangle] U + [\cdot] W$.
- $\mathcal{V}$ responds with challenge u_j chosen such that $1 + u_{k-1-j} x_3^{2^j}$ is nonzero.
- $\mathcal{P}$ and $\mathcal{V}$ set $\mathbf{G'} := \mathbf{G'}_{\text{lo}} + u_j \mathbf{G'}_{\text{hi}}$ and $\mathbf{b} := \mathbf{b}_{\text{lo}} + u_j \mathbf{b}_{\text{hi}}$.
- $\mathcal{P}$ sets $\mathbf{p'} := \mathbf{p'}_{\text{lo}} + u_j^{-1} \mathbf{p'}_{\text{hi}}$.

25. $\mathcal{P}$ sends $c = \mathbf{p'}_0$ and synthetic blinding factor f computed from the elided blinding factors.
26. $\mathcal{V}$ accepts only if $\sum_{j=0}^{k-1} [u_j^{-1}] L_j + P' + \sum_{j=0}^{k-1} [u_j] R_j = [c] \mathbf{G'}_0 + [c \mathbf{b}_0 z] U + [f] W$.

B Rustdoc Protocol Description

See next page. (from C/E)

Crate hacspec_halo2

Step	Spec	Impl.	Protocol	Notes
1	NA	?	$\mathcal{P}$ and $\mathcal{V}$ proceed in the following n_a rounds of interaction, where in round j (starting at 0)	
			* $\mathcal{P}$ sets $a'_j(X) = a_j(X, c_0, c_1, \ldots, c_{j-1}, a_0(X, \cdots), \ldots, a_{j-1}(X, \cdots, c_{j-1}))$	
			* $\mathcal{P}$ sends a hiding commitment $A_j = \langle \mathbf{a'}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{a'}$ are the coefficients of the univariate polynomial $a'_j(X)$ and $\cdot$ is some random, independently sampled blinding factor elided for exposition. (This elision notation is used throughout this protocol description to simplify exposition.)	We denote the blinding $\cdot$ used in round j as a^*_j
			* $\mathcal{V}$ responds with a challenge c_j.	
2	NA	?	$\mathcal{P}$ sets $g'(X) = g(X, c_0, c_1, \ldots, c_{n_a-1}, \cdots)$.	
3	NA	link	$\mathcal{P}$ sends a commitment $R = \langle \mathbf{r}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{r} \in \mathbb{F}^n$ are the coefficients of a randomly sampled univariate polynomial $r(X)$ of degree $n-1$.	We denote the blinding $\cdot$ used as r^*
4	step_4	link	$\mathcal{P}$ computes univariate polynomial $h(X) = \frac{g'(X)}{t(X)}$ of degree $n_g(n-1) - n$.	
5	step_5	link	$\mathcal{P}$ computes at most $n-1$ degree polynomials $h_0(X), h_1(X), \ldots, h_{n_g-2}(X)$ such that $h(X) = \sum_{i=0}^{n_g-2} X^{ni} h_i(X)$.	
6	step_6	link	$\mathcal{P}$ sends commitments $H_i = \langle \mathbf{h_i}, \mathbf{G} \rangle + [\cdot]W$ for all i where $\mathbf{h_i}$ denotes the vector of coefficients for $h_i(X)$.	We denote the blinding $\cdot$ used in round j as h^*_j
7	step_7	link(?)	$\mathcal{V}$ responds with challenge x and computes $H' = \sum_{i=0}^{n_g-2} [x^{ni}] H_i$.	
8	step_8	link	$\mathcal{P}$ sets $h'(X) = \sum_{i=0}^{n_g-2} x^{ni} h_i(X)$.	$\mathcal{P}$ sets $h'^* = \sum_{i=0}^{n_g-2} x^{ni} h^*_i$.
9	step_9	link	$\mathcal{P}$ sends $r = r(x)$ and for all $i \in [0, n_a)$ sends $\mathbf{a_i}$ such that $(\mathbf{a_i})_j = a'_i(\omega^{(\mathbf{p_i})_j} x)$ for all $j \in [0, n_e - 1]$.	
10	step_10	link(?)	For all $i \in [0, n_a)$ $\mathcal{P}$ and $\mathcal{V}$ set $s_i(X)$ to be the lowest degree univariate polynomial defined such that $s_i(\omega^{(\mathbf{p_i})_j} x) = (\mathbf{a_i})_j$ for all $j \in [0, n_e - 1)$.	
11	step_11	link	$\mathcal{V}$ responds with challenges x_1, x_2 and initializes $Q_0, Q_1, \ldots, Q_{n_q-1} = \mathcal{O}$.	
			* Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{V}$ sets $Q_{\sigma(i)} := [x_1]Q_{\sigma(i)} + A_i$.	
			* $\mathcal{V}$ finally sets $Q_0 := [x_1^2]Q_0 + [x_1]H' + R$.	
12	step_12	link	$\mathcal{P}$ initializes $q_0(X), q_1(X), \ldots, q_{n_q-1}(X) = 0$.	$\mathcal{P}$ initializes $q^*_0, q^*_1, \ldots, q^*_{n_q-1} = 0$.
			* Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{P}$ sets $q_{\sigma(i)} := x_1 q_{\sigma(i)} + a'(X)$.	Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{P}$ sets $q^*_{\sigma(i)} := x_1 q^*_{\sigma(i)} + a^*_i$.
			* $\mathcal{P}$ finally sets $q_0(X) := x_1^2 q_0(X) + x_1 h'(X) + r(X)$.	$\mathcal{P}$ finally sets $q^*_0 := x_1^2 q^*_0 + x_1 h'^* + r^*$
13	step_13	?	$\mathcal{P}$ and $\mathcal{V}$ initialize $r_0(X), r_1(X), \ldots, r_{n_q-1}(X) = 0$.	

Step	Spec	Impl.	Protocol	Notes
			* Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{P}$ and $\mathcal{V}$ set $r_{\sigma(i)}(X) := x_1 r_{\sigma(i)}(X) + s_i(X)$.	
			* Finally $\mathcal{P}$ and $\mathcal{V}$ set $r_0 := x_1^2 r_0 + x_1 h + r$ and where h is computed by $\mathcal{V}$ as $\frac{g'(x)}{t(x)}$ using the values $r, \mathbf{a}$ provided by $\mathcal{P}$.	
14	step_14	link	$\mathcal{P}$ sends $Q' = \langle \mathbf{q'}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{q'}$ defines the coefficients of the polynomial $q'(X) = \sum_{i=0}^{n_q-1} x_2^{n_q-1-i} \left(\frac{q_i(X)-r_i(X)}{\prod_{j=0}^{n_e-1} \left(X-\omega^{(\mathbf{q_i})_j} x \right)} \right)$	The blinding $\cdot$ used here is denoted as q'^* .
15	step_15	link	$\mathcal{V}$ responds with challenge x_3.	
16	step_16	link	$\mathcal{P}$ sends $\mathbf{u} \in \mathbb{F}^{n_q}$ such that $\mathbf{u}_i = q_i(x_3)$ for all $i \in [0, n_q)$.	
17	step_17	link	$\mathcal{V}$ responds with challenge x_4.	
18	step_18	link	$\mathcal{V}$ sets $P = [x_4^{n_q}]Q' + \sum_{i=0}^{n_q-1} [x_4^{n_q-1-i}]Q_i$ and $v = x_4^{n_q} \cdot \sum_{i=0}^{n_q-1} \left(x_2^{n_q-1-i} \left(\frac{\mathbf{u}_i - r_i(x_3)}{\prod_{j=0}^{n_e-1} (x_3 - \omega^{(\mathbf{q}_i)_j} x)} \right) \right) + \sum_{i=0}^{n_q-1} x_4^{n_q-1-i} \mathbf{u}_i$	
19	step_19	link	$\mathcal{P}$ sets $p(X) = x_4^{n_q} \cdot q'(X) + \sum_{i=0}^{n_q-1} x_4^{n_q-1-i} \cdot q_i(X)$.	$\mathcal{P}$ sets $p^* = x_4^{n_q} \cdot q'^* + \sum_{i=0}^{n_q-1} x_4^{n_q-1-i} \cdot q_i^*$.
20	step_20	link	$\mathcal{P}$ samples a random polynomial $s(X)$ of degree $n-1$ with a root at x_3 and sends a commitment $S = \langle \mathbf{s}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{s}$ defines the coefficients of $s(X)$.	The blinding $\cdot$ used here is denoted as s^*.
21	step_21	link	$\mathcal{V}$ responds with challenges ξ, z.	
22	step_22	link	$\mathcal{V}$ sets $P' = P - [v]\mathbf{G}_0 + [\xi]S$.	
23	step_23	link	$\mathcal{P}$ sets $p'(X) = p(X) - p(x_3) + \xi s(X)$ (where $p(x_3)$ should correspond with the verifier's computed value v).	$\mathcal{P}$ sets $p'^* = s^* \cdot \xi + p^*$
24	step_24	link	Initialize $\mathbf{p'}$ as the coefficients of $p'(X)$ and $\mathbf{G'} = \mathbf{G}$ and $\mathbf{b} = (x_3^0, x_3^1, \ldots, x_3^{n-1})$. $\mathcal{P}$ and $\mathcal{V}$ will interact in the following k rounds, where in the jth round starting in round $j = 0$ and ending in round $j = k-1$:	
			* $\mathcal{P}$ sends $L_j = \langle \mathbf{p'}_{hi}, \mathbf{G'}_{lo} \rangle + [z \langle \mathbf{p'}_{hi}, \mathbf{b}_{lo} \rangle]U + [\cdot]W$ and $R_j = \langle \mathbf{p'}_{lo}, \mathbf{G'}_{hi} \rangle + [z \langle \mathbf{p'}_{lo}, \mathbf{b}_{hi} \rangle]U + [\cdot]W$.	The blinding $\cdot$ used for L_j is denoted L_j^* and the blinding used for R_j is denoted R_j^*
			* $\mathcal{V}$ responds with challenge u_j chosen such that $1 + u_{k-1-j} x_3^{2^j}$ is nonzero.	
			* $\mathcal{P}$ and $\mathcal{V}$ set $\mathbf{G'} := \mathbf{G'}_{lo} + u_j \mathbf{G'}_{hi}$ and $\mathbf{b} := \mathbf{b}_{lo} + u_j \mathbf{b}_{hi}$.	
			* $\mathcal{P}$ sets $\mathbf{p'} := \mathbf{p'}_{lo} + u_j^{-1} \mathbf{p'}_{hi}$.	
25	step_25	link	$\mathcal{P}$ sends $c = \mathbf{p'}_0$ and synthetic blinding factor f computed from the elided blinding factors.	$\mathcal{P}$ sets $f = p'^* + \sum_{j=0}^{k-1} L_j^* \cdot u_j^{-1} + r_j^* \cdot u_j$
26	step_26	link	$\mathcal{V}$ accepts only if $\sum_{j=0}^{k-1} [u_j^{-1}] L_j + P' + \sum_{j=0}^{k-1} [u_j] R_j = [c]\mathbf{G'}_0 + [c \mathbf{b}_0 z]U + [f]W$.	

C Git repository

https://github.com/Hvassaa/hacspec

D Test of rotate_polyx

```
fn test_rotate(
    x: u128, a0: u128, a1: u128, a2: u128,
    a3: u128, a4: u128, a5: u128, a6: u128)
{
    let omega =
        FpVesta::from_hex("
            00003a57bee9fb370430aa5f610ed09c17
            fe7e538bca7c94ad2b1ba3a33bc04980a4"
        ); //omega from halo2

    let x = FpVesta::from_literal(x);
    let a0 = FpVesta::from_literal(a0);
    let a1 = FpVesta::from_literal(a1);
    let a2 = FpVesta::from_literal(a2);
    let a3 = FpVesta::from_literal(a3);
    let a4 = FpVesta::from_literal(a4);
    let a5 = FpVesta::from_literal(a5);
    let a6 = FpVesta::from_literal(a6);
    let a7 = a0 * a1 * a2 + a5;
    // a7 is pseudorandom as quickheck only allows 8 arguments

    let poly = Seq::<FpVesta>::from_vec(vec!
        [a0, a1, a2, a3, a4, a5, a6, a7]
    );
    assert_eq!(poly.len(), 8);

    let poly_rotated_cur = rotate_polyx(
        poly.clone(), FpVesta::ONE()
    );
    let poly_rotated_next = rotate_polyx(
        poly.clone(), omega
    );
    let poly_rotated_prev = rotate_polyx(
        poly.clone(), omega.inv()
    );

    assert_eq!(
        eval_polyx(poly.clone(), x),
        eval_polyx(poly_rotated_cur.clone(), x),
    );
    assert_eq!(
        eval_polyx(poly.clone(), x * omega),
        eval_polyx(poly_rotated_next.clone(), x),
    );
    assert_eq!(
        eval_polyx(poly.clone(), x * omega.inv()),
        eval_polyx(poly_rotated_prev.clone(), x),
    );
}
```

E Rustdoc on Github Pages

https://hvassaa.github.io/hacspec_halo2/

F Latex Project

https://github.com/Hvassaa/Master-Thesis/