Preon: zk-SNARK based Signature Scheme

Ming-Shing Chen[1], Yu-Shian Chen[2], Chen-Mou Cheng[3], Shiuan Fu[3], Wei-Chih Hong[3], Jen-Hsuan Hsiang[3], Sheng-Te Hu[3], Po-Chun Kuo[3], Wei-Bin Lee[2], Feng-Hao Liu[4], and Justin Thaler[5]

[1]Academia Sinica, Taipei, Taiwan
[2]Hon Hai Research Institute, Taipei, Taiwan
[3]BTQ Technologies Corp.
[4]Florida Atlantic University, FL 33431, USA
[5]Department of Computer Science, Georgetown University, Washington, DC 20057, USA

May 2023

Contents

In this proposal, we construct a signature scheme based on the general-purpose zero-knowledge proving system Aurora [BCR+19].

1 Introduction

General-purpose proving systems have been undergoing rapid development in recent years. A general-purpose proving system is a Fiat-Shamir transformed interactive protocol in which a prover can convince a verifier that the prover knows a secret witness for the truthfulness of a somewhat general statement. When this statement is about knowledge of a secret, we can construct signature schemes, e.g., following the MPC-in-the-head paradigm [CDG+17, KKW18, dSGMOS19, BdK+21, DKR+22], as well as based on zk-STARK [STA].

More specifically, one such way is for the prover to prove the statement "I know a secret key sk corresponding to the public key pk through the relation $\mathsf{pk} = \mathrm{OWF}(\mathsf{sk})$," where OWF is an appropriate function that is conjectured to be one-way, and then use the Fiat-Shamir transform to turn an interactive protocol into a non-interactive one. In this paradigm, signing a message msg with the secret key sk amounts to generating a proof in the non-interactive protocol with msg used to generate the public random coins needed in producing the proof. Likewise, signature verification is simply proof verification, which any verifier can carry out.

One may (rightly) expect that a major drawback of such an approach is the overhead in terms of space and time one needs to pay in constructing a signature scheme from a general-purpose proving system, as we do not have access to any of the optimization opportunities brought about by specialization. However, we argue for this approach because it can bring a long-term advantage as follows. Once we have a (secure) signature scheme constructed from a general-purpose proving system like this, the flexibility of the latter would easily allow us to enhance the functionalities of the former and build at a minimum cost advanced schemes like group signatures [CvH91], attribute-based signatures [MPR11], functional signatures [BGI14], ..., to name a few, by proving a suitable (and potentially more complicated) statement in the proving system. Thus, the tremendous amount of investment that goes into a unified process of security analysis, standardization, implementation, deployment, as well as post-deployment continual improvement and optimization can pay lucrative dividends across a broader and fast-growing landscape of applications, compared with the alternative approach of independently standardizing all these different signature schemes individually, necessarily having to start from scratch and repeating much of the work every time.

In this document, we will detail a simple proposal following the above philosophy. Specifically, we will give the following information.

- A complete specification will be give in sections 3 through 7.
- A detailed performance analysis will be given in section 8.
- A description of the expected security strength will be given in sections 4.5, 6.2, and 6.3.
- An analysis of the algorithm with respect to known attacks will be given in section 4.6.

The rest of this document is organized as follows. In section 2, we give an over of the notations used in this document. In section 3, we give our signature construction and a high-level argument why it is secure. In section 4, we introduce some basics of the Aurora proving system. In section 5, we describe our choice of AES-based one-way function, as well as how we encode it into Aurora. In section 6, we give our suggestions of security parameters for different security levels. In section 7, we give pseudocode of the signature scheme to help implementers better understand the components in our construction and reference implementation. In section 8, we demonstrate the performance of the resulting signature schemes.

2 Preliminary

2.1 Notations

We use f, g, h, p, q, r with subscripts to denote polynomials, bold font lowercase letters like $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{v}, \mathbf{w}$ to denote vectors, as well as bold font uppercase letters like $\mathbf{A}, \mathbf{B}, \mathbf{C}$ to denote matrices. Other usual lowercase letters like a, b, t, m, n are used as constants or parameters, some of the lower case letters are used for both polynomials and constants, in such cases we will make sure the contents are clear. The lowercase serif letters like b, r and Greek letters like $\alpha, \delta, \rho, \epsilon, \lambda$ are also used to denote numbers or parameters. We use usual uppercase letters with subscripts like L, S, H_1, H_2 to denote sets of numbers, and use $\mathbb{F}$ to denote a field. The Reed-Solomon codeword obtained when evaluating a polynomial f on a set L is denoted as $\hat{f}$, which we will go into more detail in section 2.4. We use $\mathtt{i}, \mathtt{w}$ to denote instances and witnesses. We use $\mathcal{R}$ to denote a relation and use $\mathcal{C}$ to denote a circuit.

A notable exception is that we use the uppercase letter Z with subscripts to denote vanishing polynomials of the form $Z(x) = \Pi_{s \in S}(x - s)$ for some set S.

2.2 R1CS

A Rank-1 Constraint System (R1CS) can be formalized as a four-tuple $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{z})$, with three matrices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^{a \times b}$ and a vector $\mathbf{z} \in \mathbb{F}^b$, such that the i-th entry of the vector $\mathbf{Az}$ times the i-th entry of $\mathbf{Bz}$ equals the i-th entry of $\mathbf{Cz}$ for all $i \in \{1, 2, ..., a\}$, which is denoted as $\mathbf{Az} \circ \mathbf{Bz} = \mathbf{Cz}$, where $\circ$ denotes the operator of entry-wise multiplication of vectors.

It is known that any satisfiable arithmetic circuit $\mathcal{C}$ over a finite field $\mathbb{F}$ can be efficiently transformed into R1CS $(\mathbf{A}_\mathcal{C}, \mathbf{B}_\mathcal{C}, \mathbf{C}_\mathcal{C}, \mathbf{z})$, where $\mathbf{A}_\mathcal{C}, \mathbf{B}_\mathcal{C}, \mathbf{C}_\mathcal{C}$ correspond to the fixed circuit $\mathcal{C}$ itself, while $\mathbf{z} = (1, \mathbf{v}, \mathbf{w})$ gives an instance of computation trace satisfying the circuit $\mathcal{C}$ with public instance values $\mathbf{v}$ and private witness values $\mathbf{w}$.

2.3 Interactive Oracle Proof

An interactive oracle proof (IOP) is a multi-round two-party interactive protocol [BCS16]. In the i-th round, one party (the verifier) sends a message m_i to the other party (the prover), and then the prover replies to the verifier with the commitment π_i of a message. After all communication rounds end, the verifier will then challenge the prover to partially open these commitments in a specific way and, based on the answers to these queries, decide whether to accept or reject the claim that nthe prover knows some private witness $\mathtt{w}$ that is in a prescribed relation $\mathcal{R}$ with some public instance $\mathtt{i}$.

Interactive oracle proof combines the feature of interactive proof (IP) and probabilistically checkable proof (PCP). IOP has multiple interaction rounds as IP and at the same time is similar to PCP in that the verifier does not need to read the full messages π_i for all i. Instead, it only queries some positions of the messages in order to accept or reject the prover's claim. In other words, the prover's messages $\{\pi_i\}_i$ can be thought of as "oracles" for the verifier to query at certain positions.

We say that an IOP is "public-coin" if all the verifier messages $\{m_i\}_i$ are uniformly chosen from some domains.

Here we state some properties that will be considered when it comes to IOPs, adapted from the definitions in [BCR+19].

Completeness. An IOP protocol for a relation $\mathcal{R}$ is said to be complete if for any valid instance-witness pair $(\mathtt{i}, \mathtt{w}) \in \mathcal{R}$, the probability that the verifier outputs 1 (accept) is 1.

Soundness. An IOP protocol is said to have soundness error ϵ if for any invalid instance $\mathbb{i}$ for which there is no witness $\mathbb{w}$ such that $(\mathbb{i}, \mathbb{w}) \in \mathcal{R}$), the probability for a computationally unbounded prover to convince the verifier otherwise is at most ϵ, which is allowed to depend on a security parameter. If ϵ is a negligible function, then we say the IOP is sound.

Proof of Knowledge. An IOP protocol is said to give a proof of knowledge with knowledge soundness error $\epsilon(n)$ if there exists a probabilistic polynomial-time algorithm E, called an extractor algorithm, such that for every instance $\mathbb{i}$ and every computationally unbounded prover $\tilde{P}$ that convinces the verifier to accept $\mathbb{i}$ with probability μ, $\mathsf{E}^{\tilde{P}}(\mathbb{i}, 1^{w_{\mathcal{R}(n)}})$ outputs $\mathbb{w}$ such that $(\mathbb{i}, \mathbb{w}) \in \mathcal{R}$ with probability at least $\mu - \epsilon(n)$, where $w_{\mathcal{R}(n)} := \max\{|\mathbb{w}| : (\mathbb{i}, \mathbb{w}) \in \mathcal{R}, |\mathbb{i}| = n\}$. If ϵ is a negligible function, then the IOP is said to be knowledge sound, or simply a proof of knowledge.

Intuitively, if an IOP is knowledge sound, and there exists a prover that can convince the verifier to accept instance $\mathbb{i}$ with high probability, then we can conclude that this prover must actually know a valid witness $\mathbb{w}$ for $\mathbb{i}$.

Remark 2.1 *When constructing a signature scheme, we will focus on the knowledge soundness (or proof of knowledge) property of the proving system, since we need to guarantee that the signer does have the knowledge of a secret key rather than the mere existence of a secret key.*

Zero Knowledge. We refer to Definition 4.2, 4.3, and 4.4 of [BCR+19] for a formal definition of the zero-knowledge properties of an IOP. Intuitively, we say that an IOP for a relation $\mathcal{R}$ has zero-knowledge property if the verifier cannot get any information about the prover's private witness after the interaction, whether it is convinced or not.

Remark 2.2 *When constructing a signature scheme, we will need the zero-knowledge property of the proving system since the signer doesn't want any part of the private key been revealed to the verifier.*

2.4 Reed-Solomon Code

For the sake of completeness, we restate the definition of Reed-Solomon code in [BCR+19]. Given a subset L of a field $\mathbb{F}$ and $\rho \in (0, 1]$, we denote by $RS[L, \rho] \subset \mathbb{F}^{|L|}$ all evaluations over L of univariate polynomials of degree less than $\rho \cdot |L|$. That is, a word $\mathbf{c} \in \mathbb{F}^{|L|}$ is a codeword of a Reed-Solomon code $RS[L, \rho] \subset \mathbb{F}^{|L|}$ if there exists a polynomial p of degree less than $\rho \cdot |L|$ such that $\mathbf{c}_i = p(i)$ for every $i \in L$.

Distance Measure. For two vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{F}^k$, we define $\Delta(\mathbf{v}_1, \mathbf{v}_2)$ to be the relative Hamming distance between $\mathbf{v}_1$ and $\mathbf{v}_2$, that is, $\Delta(\mathbf{v}_1, \mathbf{v}_2) := \frac{1}{k} \cdot \#\{i \in \{1, 2, ..., k\} | \mathbf{v}_{1,i} \neq \mathbf{v}_{2,i}\}$. For a set $S \subset \mathbb{F}^k$, we define $\Delta(\mathbf{v}_1, S) := \min_{\mathbf{v}_2 \in S} \Delta(\mathbf{v}_1, \mathbf{v}_2)$. When we discuss the distance measure for Reed-Solomon codes, we will use these two measures.

3 Our Proposed Signature Scheme

In this section, we present our proposed signature scheme, which will be organized as follows: (1) we first recall the syntax and security notion for a signature scheme; (2) we present a general paradigm of constructions and the required building blocks in an abstract way; (3) we present how to instantiate the building blocks, aiming at different security levels as required by the NIST. Our concrete parameters are given in the later sections as we will point out as references.

3.1 Syntax of a Signature Scheme

A digital signature scheme consists of three algorithms, namely $\{\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}\}$, which work as follows.

- *Key generation*: The key generation algorithm $\mathsf{KeyGen}(\cdot)$ takes a security parameter 1^κ as input, and then outputs a pair of secret and public keys $(\mathsf{sk}, \mathsf{pk})$.
- *Signing*: The signing algorithm $\mathsf{Sign}(\cdot)$ takes a message msg and a secret key sk as input, and then outputs a signature sig.
- *Verification*: The verification algorithm $\mathsf{Verify}(\cdot)$ takes a message msg, a public key pk and a signature sig as input, and then outputs 0 or 1 to indicate rejecting/accepting the signature with respect to the message.

Next, we restate an important security notion—Existential Unforgeability under Chosen Message Attack, also known as EUF-CMA.

Definition 3.1 *Let* $\mathsf{SIG} = \{\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}\}$ *be a signature scheme. We consider the following experiment, which is parameterized by an adversary* $\mathcal{A}$ *and security parameter* κ*.*

- $\mathsf{KeyGen}(1^\kappa)$ *is run to obtain keys* $(\mathsf{pk}, \mathsf{sk})$*.*
- *Adversary* $\mathcal{A}$ *is given* pk *and access to a signing oracle* $\mathcal{O}_{\mathsf{sk}}(\cdot)$*, which returns a signature with respect to each input query made by the adversary* $\mathcal{A}$*. Let* $\mathcal{Q}$ *be the set of queries made by* $\mathcal{A}$*.*
- *At the end,* $\mathcal{A}$ *outputs a pair* (msg^*, sig^*) *as the forgery.* $\mathcal{A}$ *wins the security game if and only if (1)* $\mathsf{Verify}(\mathsf{pk}, msg^*, sig^*) = 1$*, and (2)* $msg^* \notin \mathcal{Q}$*.*

A signature scheme is EUF-CMA secure (or simply secure when it is clear from the context) if, for any probabilistic polynomial-time $\mathcal{A}$*, the winning probability is upper bounded by* $\mathsf{negl}(\kappa)$ *for some negligible function* $\mathsf{negl}(\cdot)$*.*

3.2 General Construction Paradigm

Now we present a general paradigm to construct a digital signature scheme, using as building blocks (1) a (true)-simulation extractable non-interactive zero-knowledge proof (NIZK) that supports labels, and (2) a hard NP relation. We elaborate on these notions next.

Building Blocks. We use $\Pi.\{\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}\}$ to denote a NIZK for general NP relations, and $\mathcal{R} = \{(x, w)\}$ a hard NP relation. They should satisfy the following security notions.

Definition 3.2 ((True) Simulation Extractable NIZK) *Let* $\mathcal{R}$ *be an NP relation on pairs* (x, w) *with corresponding language* $L_{\mathcal{R}} = \{x : \exists\ w \text{ such that } (x, w) \in \mathcal{R}\}$*. A true-simulation extractable non-interactive zero-knowledge (NIZK) argument supporting labels and for relation* $\mathcal{R}$ *consists of three algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$*:*

- $(\mathsf{Pub}, \mathsf{TK}, \mathsf{EK}) \leftarrow \mathsf{Setup}(1^\kappa)$*: creates a public string* Pub*, a trapdoor* TK*, and an extraction key* EK*.*
- $\pi \leftarrow \mathsf{Prove}^L(\mathsf{Pub}, x, w)$*: creates an argument* π *that* $\mathcal{R}(x, w) = 1$*, where* L *is a string to indicate label.*
- $0/1 \leftarrow \mathsf{Verify}^L(\mathsf{Pub}, x, \pi)$*: verifies whether or not the argument* π *is correct, where* L *is a string to indicate label.*

For clarity of presentation, we omit Pub *in the* Prove *and* Verify. *We require that three basic properties hold:*

- ***Completeness.*** *For any* $(x, w) \in \mathcal{R}$ *and any label* L, *if* $(\mathsf{Pub}, \mathsf{TK}, \mathsf{EK}) \leftarrow \mathsf{Setup}(1^\kappa)$, $\pi \leftarrow \mathsf{Prove}^L(x, w)$, *then* $\mathsf{Verify}^L(x, \pi) = 1$.
- ***Soundness.*** *For any probabilistic polynomial-time adversary* $\mathcal{A}$ *and any label* L, *the probability for the following event is negligible:* $(\mathsf{Pub}, \mathsf{TK}, \mathsf{EK}) \leftarrow \mathsf{Setup}(1^\kappa)$, $(x^*, \pi^*) \leftarrow \mathcal{A}(\mathsf{Pub})$ *such that* $x^* \notin L_{\mathcal{R}}$ *but* $\mathsf{Verify}^L(\mathsf{Pub}, x^*, \pi^*) = 1$.
- ***Composable Zero-knowledge.*** *There exists a probabilistic polynomial-time simulator* $\mathcal{S}$ *such that for any label* L *and any probabilistic polynomial-time* $\mathcal{A}$, *the advantage (the probability* $\mathcal{A}$ *wins minus* 0.5*) is negligible in the following game.*
  - *A challenger samples* $(\mathsf{Pub}, \mathsf{TK}, \mathsf{EK}) \leftarrow \mathsf{Setup}(1^\kappa)$ *and sends* $(\mathsf{Pub}, \mathsf{TK})$ *to* $\mathcal{A}$
  - $\mathcal{A}$ *chooses* $(x, w) \in \mathcal{R}$ *and sends to the challenger.*
  - *The challenger generates* $\pi_0 \leftarrow \mathsf{Prove}^L(x, w)$, $\pi_1 \leftarrow \mathcal{S}(x, L, \mathsf{TK})$, *and then samples a random bit* $b \leftarrow \{0, 1\}$. *Then it sends* π_b *to* $\mathcal{A}$.
  - $\mathcal{A}$ *outputs a guess bit* b' *and wins if* $b' = b$.
- ***Extractibility.*** *Additionally, true simulation extractability requires that there exists a probabilistic polynomial-time extractor* Ext *such that for any probabilistic polynomial-time adversary* $\mathcal{A}$, *the probability* $\mathcal{A}$ *wins is negligible in the following game:*
  - *A challenger samples* $(\mathsf{Pub}, \mathsf{TK}, \mathsf{EK}) \leftarrow \mathsf{Setup}(1^\kappa)$ *and sends* Pub *to* $\mathcal{A}$.
  - $\mathcal{A}$ *is allowed to make oracle queries to the simulation algorithm* $\mathcal{S}'((x, w), L, \mathsf{TK})$ *adaptively with input* (x, w) *and* L, *where* $\mathcal{S}'$ *first checks if* $(x, w) \in \mathcal{R}$ *and returns* $\mathcal{S}(x, L, \mathsf{TK})$ *if that is the case.*
  - $\mathcal{A}$ *outputs a tuple* x^*, L^*, π^*.
  - *The challenger runs the extractor* $w^* \leftarrow \mathsf{Ext}(L^*, (x^*, \pi^*), \mathsf{EK})$.
  - $\mathcal{A}$ *wins if (1) the pair* (x^*, L^*) *was not part of the simulator query, (2) the proof* π^* *verifies, i.e.,* $\mathsf{Verify}^{L^*}(x^*, \pi^*) = 1$ *and (3)* $\mathcal{R}(x^*, w^*) = 0$.

The simulation extractability captures the concept of non-malleability, meaning that the adversary cannot generate a valid proof after seeing several simulated proofs (as well real proofs), without knowing a witness. This property is crucial for the signature design based on NIZK. Next, we define our next building block—a hard NP relation.

Definition 3.3 *Let* $\mathcal{R}$ *be an NP relation associated with the probabilistic polynomial-time sampling algorithm* $\mathsf{Gen}(\cdot)$. *The relation* $\mathcal{R}$ *is hard if and only if the following holds:*

- *For any* $(x, w) \in \mathsf{Gen}(1^\kappa)$, *we have* $(x, w) \in \mathcal{R}$.
- *There is a polynomial-time algorithm that determines whether* $(x, w) \in \mathcal{R}$ *for any input* (x, w). *The decision algorithm is denoted as* $\mathcal{R}(x, w) \in \{0, 1\}$.
- *For any probabilistic polynomial-time adversary* $\mathcal{A}$, *the following probability is negligible:*

$$\Pr\left[\mathcal{R}(x, w^*) = 1 \mid w^* \leftarrow \mathcal{A}(x), (x, w) \leftarrow \mathsf{Gen}(1^\kappa)\right] \leq \mathsf{negl}(\kappa).$$

General Construction. Given $\Pi.\{\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}\}$ as a true-simulation extractable NIZK supporting labels for general NP relations (as Definition 3.2), and $\mathcal{R} = \{(x, w)\}$ as a hard NP relation (as Definition 3.3), below is a general paradigm to construct signature schemes.

Construction 3.4 (General Paradigm) *Consider the following scheme* $\mathsf{SIG}.\{\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}\}$*:*

- *There is a one-time system setup algorithm that runs the NIZK* $\mathsf{Pub} \leftarrow \Pi.\mathsf{Setup}(1^\kappa)$*. Then it publishes* Pub*. In all the following three algorithms, we assume that* Pub *is implicitly taken as the public input.*
- $\mathsf{KeyGen}(1^\kappa)$*: on input the security parameter* 1^κ*, the algorithm runs the relation sampling algorithm of* $\mathcal{R}$*, i.e.,* $(x, w) \leftarrow \mathsf{Gen}(1^\kappa)$*, and sets* $\mathsf{pk} := x$*, and* $\mathsf{sk} := w$*.*
- $\mathsf{Sign}(\mathsf{sk}, msg)$*: to sign a message* msg *with secret key* $\mathsf{sk} = w$*, the algorithm runs* $\pi \leftarrow \Pi.\mathsf{Prove}^{msg}(x, w)$ *with respect to the relation* $\mathcal{R}$*, using* msg *as the label. It outputs the signature as* $sig := \pi$*.*
- $\mathsf{Verify}(\mathsf{pk}, (msg, sig))$*: to verify a message-signature pair* (msg, sig)*, the algorithm parses* sig *as a NIZK proof* π*, and outputs* $\Pi.\mathsf{Verify}^{msg}(\mathsf{pk}, \pi)$ *where* msg *is the label.*

To analyze the security of the design, we can use the result of the work [DHLW10] as summarized in the following theorem.

Theorem 3.5 ([DHLW10]) *Assuming that* $\mathcal{R}$ *is a hard relation and* Π *is a NIZK that is true-simulation extractable, then Construction 3.4 is EUF-CMA secure.*

3.3 Instantiations

In this section, we present our concrete parameters to instantiate the building blocks—(1) the hard relation, and (2) the NIZK proof system.

3.3.1 Hard Relation

We propose to use AES to build the hard relation, following the work of BBQ [dSGMOS19] and Banquet[BdK+21]. The general idea is to set the public string as (r, y) and the secret witness as sk such that $y = \mathsf{AES}_{\mathsf{sk}}(r)$. Below we present the formal forms for different security levels.

- (Level 1) Let $\mathcal{R}_1 = \left\{(x = (r, y), w = \mathsf{sk}) : r, y \in \{0,1\}^{128}, y = \mathsf{AES}_{\mathsf{sk}}(r)\right\}$. Here the AES uses the L1-level parameters as suggested by NIST. The Gen algorithm is: (1) choose uniformly at random $r \in \{0,1\}^{128}$ and $\mathsf{sk} \in \{0,1\}^{128}$; (2) compute $y = \mathsf{AES}_{\mathsf{sk}}(r)$; (3) output $x = (r, y)$ and $w = \mathsf{sk}$.
- (Level 3) Let $\mathcal{R}_3 = \Big\{(x = (r, y = (y_1, y_2)), w = \mathsf{sk}) : r \in \{0,1\}^{192}, y_1, y_2 \in \{0,1\}^{128}, y_1 = \mathsf{AES}_{\mathsf{sk}}(r_1), y_2 = \mathsf{AES}_{\mathsf{sk}}(r_2)\Big\}$, where r_1 is the first 128 bits of r and r_2 is the last 64 bits of r padded with 64 zeros. Here AES uses the L3-level parameters as suggested by NIST. The Gen algorithm is: (1) choose uniformly at random $r \in \{0,1\}^{192}$ and $\mathsf{sk} \in \{0,1\}^{192}$; (2) set r_1 and r_2 as above, and then compute $y_1 = \mathsf{AES}_{\mathsf{sk}}(r_1)$, $y_2 = \mathsf{AES}_{\mathsf{sk}}(r_2)$; (3) output $x = (r, y = (y_1, y_2))$ and $w = \mathsf{sk}$.
- (Level 5) Let $\mathcal{R}_5 = \Big\{(x = (r_1, r_2, y_1, y_2), w = \mathsf{sk}) : r_1, r_2 \in \{0,1\}^{128}, y_1, y_2 \in \{0,1\}^{128}, y_1 = \mathsf{AES}_{\mathsf{sk}}(r_1), y_2 = \mathsf{AES}_{\mathsf{sk}}(r_2)\Big\}$. Here AES uses the L5-level parameters as suggested by NIST. The Gen algorithm is: (1) choose uniformly at random $r_1, r_2 \in \{0,1\}^{128}$ and $\mathsf{sk} \in \{0,1\}^{256}$; (2) then compute $y_1 = \mathsf{AES}_{\mathsf{sk}}(r_1)$, $y_2 = \mathsf{AES}_{\mathsf{sk}}(r_2)$; (3) output $x = (r_1, r_2, y_1, y_2)$ and $w = \mathsf{sk}$.

3.3.2 NIZK

In this section, we highlight how we instantiate the required NIZK. First, we notice that the building block as presented above is a proof system that supports labels. This is easy to achieve for an IOP (interactive oracle proof) system with the BCS transform (using Fiat-Shamir techniques). Particularly, we can embed the label to some hash function, i.e., $H(L, \cdot)$, and then use it as the random oracle. In this way, the label can be associated with the proof as we need.

To instantiate the basic NIZK, we choose the scheme Aurora, whose details are presented in Section 4. We notice that a true-simulation extractable NIZK can be achieved by using a CCA2 encryption scheme and a regular NIZK via the simple construction of [DHLW10]. Thus Aurora can be used to realize this notion together with a CCA2 encryption via their framework.

This work further conjectures that the Aurora itself (without the construction of [DHLW10]) is already true-simulation extractable. The work [FKMV12] showed that Fiat-Shamir transform can already achieve some form of non-malleability. This provide some confidence for our conjecture theoretically.

4 Aurora

In this section, we describe the Aurora proving system which is proposed in [BCR$^+$19].

Aurora is a public-coin IOP protocol for R1CS relations, where the prover's goal is to convince the verifier of the statement "I know a witness vector $\mathbf{w}$ such that $\mathbf{Az} \circ \mathbf{Bz} = \mathbf{Cz}$ for $\mathbf{z} = (1, \mathbf{v}, \mathbf{w}) \in \mathbb{F}^{n+1}$." Here the arithmetic take place over a finite field $\mathbb{F}$; both the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^{m \times (n+1)}$ and the vector $\mathbf{v} \in \mathbb{F}^k$ are publicly known.

The truthfulness of the statement is equivalent to that of the following two statements: (1) "I know the witness vector $\mathbf{w}$, along with three vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ such that $\mathbf{a} = \mathbf{Az}, \mathbf{b} = \mathbf{Bz}$ and $\mathbf{c} = \mathbf{Cz}$"; (2) "$\mathbf{a} \circ \mathbf{b} = \mathbf{c}$."

The high-level idea of Aurora is thus to use the "Lincheck protocol" to prove the first statement, and then use the "Rowcheck protocol" to prove the second statement. Further details of "Lincheck" and "Rowcheck," as well as other subprotocols are presented in Appendix A.

In section 4.3, we introduce a transform that can turn the interactive Aurora protocol into a non-interactive one. In section 4.4, we describe how to add a zero-knowledge property to the Aurora protocol. We call the protocol after applying the techniques in section 4.3 and section 4.4 to the interactive Aurora IOP the "non-interactive zero-knowledge Aurora protocol."

4.1 Full Aurora IOP Flow

In this section, we explain the way how Aurora combines all the components in order to form an IOP for R1CS relations.

4.1.1 Preparation

The prover and the verifier agree upon the following:

- A predetermined R1CS relation that they want to prove;
- A finite field $\mathbb{F}$;
- Subspaces $H_1, H_2 \subset \mathbb{F}$ with proper sizes, where $H_1 = \{h_1, ..., h_m\}$ and $H_2 = \{h_1, ..., h_{n+1}\}$;
- Three positive numbers $\lambda_{\mathrm{i}}, \lambda'_{\mathrm{i}}, \ell \in \mathbb{N}$ as repetition parameters;

- A configuration of the FRI protocol, which contains additive cosets $\{L_i\}_i$ with proper sizes and polynomials $\{q_i\}_i$.

The prover and verifier also precompute the polynomial $Z_{H_1}(x) := \Pi_{j\in\{1,...,m\}}(x - h_j)$, the polynomial $Z_{H_2}^{\leq(k+1)}(x) := \Pi_{j\in\{1,...,k+1\}}(x - h_j)$, and the polynomial $Z_{H_1\cup H_2}(x) := \Pi_{h\in H_1\cup H_2}(x - h)$ for later use.

4.1.2 Inputs to Prover and Verifier

The prover takes as inputs the private witness vector $\mathbf{w} \in \mathbb{F}^{(n-k)}$, the public instance vector $\mathbf{v} \in \mathbb{F}^k$, and three public matrices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^{m\cdot(n+1)}$, where m denotes the number of rows of the matrices, and $n + 1$ denotes the number of columns (as well as the length of the vector $\mathbf{z} := (1, \mathbf{v}, \mathbf{w})$). The verifier takes inputs the public instance vector $\mathbf{v} \in \mathbb{F}^k$ and the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^{m\cdot(n+1)}$.

4.1.3 Main Interactive Protocol

1. **Polynomial Interpolation.** In this part, the prover uses its inputs to compute $f_{\mathbf{w}}, f_{\mathbf{Az}}, f_{\mathbf{Bz}}, f_{\mathbf{Cz}}$ via Lagrange interpolation, such that

$$f_{(1,\mathbf{v})}(x) = \begin{cases} 1 & \text{for} \quad x = h_i \text{ where } i = 1 \\ \mathbf{v}_{i-1} & \text{for} \quad x = h_i \text{ where } i \in \{2, ..., k+1\} \end{cases} \tag{1}$$

$$f_{\mathbf{w}}(x) = \begin{cases} \frac{\mathbf{w}_{i-k-1} - f_{(\mathbf{1},\mathbf{v})}(x)}{Z_{H_2}^{\leq(k+1)}(x)} & \text{for} \quad x = h_i \text{ where } i \in \{k+2, ..., n+1\} \end{cases} \tag{2}$$

$$f_{\mathbf{Az}}(x) = \{ \ (\mathbf{Az})_i \quad \text{for} \quad x = h_i \text{ where } i \in \{1, ..., m\} \tag{3}$$

$$f_{\mathbf{Bz}}(x) = \{ \ (\mathbf{Bz})_i \quad \text{for} \quad x = h_i \text{ where } i \in \{1, ..., m\} \tag{4}$$

$$f_{\mathbf{Cz}}(x) = \{ \ (\mathbf{Cz})_i \quad \text{for} \quad x = h_i \text{ where } i \in \{1, ..., m\} \tag{5}$$

Note that if we define the polynomial $f_{\mathbf{z}}$ via Lagrange interpolation such that

$$f_{\mathbf{z}}(x) = \{ \ \mathbf{z}_i \quad \text{for} \quad x = h_i \text{ where } i \in \{1, ..., n+1\} \tag{6}$$

, then by the uniqueness of Lagrange interpolation and comparing (1), (2), (6), we have that

$$f_{\mathbf{z}}(x) = f_{(1,\mathbf{v})}(x) + f_{\mathbf{w}}(x) \cdot Z_{H_2}^{\leq(k+1)}(x) \tag{7}$$

The prover then evaluates these polynomials over a domain L, i.e., computes the Reed-Solomon codewords $\hat{f}_{\mathbf{w}} := f_{\mathbf{w}}|_L \in RS[L, \frac{n-k}{|L|}], \hat{f}_{\mathbf{Az}} := f_{\mathbf{Az}}|_L \in RS[L, \frac{m}{|L|}], \hat{f}_{\mathbf{Bz}} := f_{\mathbf{Bz}}|_L \in RS[L, \frac{m}{|L|}], \hat{f}_{\mathbf{Cz}} := f_{\mathbf{Cz}}|_L \in RS[L, \frac{m}{|L|}]$, and sends these codewords as message oracles to the verifier.

Note that after receiving these oracles, if the verifier wants to know $f_{\mathbf{z}}(x_0)$ for some point $x_0 \in L$, the verifier only needs to query the prover to open $f_{\mathbf{w}}(x_0)$, since the verifier can compute $f_{(1,\mathbf{v})}(x_0), Z_{H_2}^{\leq(k+1)}(x_0)$ by itself and then compute $f_{\mathbf{z}}(x_0)$ via formula (7). In Aurora's original paper, the authors called polynomials like $f_{\mathbf{z}}$ "virtual oracles," as their value can be computed by the verifier from querying some previously received message oracles.

2. **Lincheck.** The prover and the verifier will repeat this part λ_i times, during which they engage in an amortized Lincheck to prove that if $f_{\mathbf{z}}$ is indeed the Lagrange interpolation of $\mathbf{z}$, then $f_{\mathbf{Az}}$, $f_{\mathbf{Bz}}$, and $f_{\mathbf{Cz}}$ are those of $\mathbf{Az}$, $\mathbf{Bz}$, and $\mathbf{Cz}$, respectively, for the public matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ that are known to both of them. To do this, in the i-th iteration for $i \in \{1, ..., \lambda_i\}$, the verifier first samples four uniformly random elements $\alpha_i, s_{i,1}, s_{i,2}, s_{i,3} \in \mathbb{F}$ and sends them to the prover. The prover then interpolates a polynomial $p^1_{\alpha_i}$ such that

$$p^1_{\alpha}(x) = \begin{cases} \alpha^{i-1} & \text{for} \quad x = h_i \text{ where } i \in \{1, 2, ..., m\} \\ 0 & \text{for} \quad x \in \{H_1 \cup H_2\} \setminus H_1, \end{cases} \tag{8}$$

as well as polynomials $p^{2,\mathbf{A}}_{\alpha_i}, p^{2,\mathbf{B}}_{\alpha_i}, p^{2,\mathbf{C}}_{\alpha_i}$ such that

$$p^{2,\mathbf{A}}_{\alpha}(x) = \begin{cases} \sum_{i=1}^{m}(\mathbf{A}_{i,j} \cdot \alpha^{i-1}) & \text{for} \quad x = h_j \text{ where } j \in \{1, 2, ..., n+1\} \\ 0 & \text{for} \quad x \in \{H_1 \cup H_2\} \setminus H_2, \end{cases} \tag{9}$$

replacing $\mathbf{A}$ by $\mathbf{B}$ and $\mathbf{C}$ the remaining two polynomials $p^{2,\mathbf{B}}_{\alpha_i}$ and $p^{2,\mathbf{C}}_{\alpha_i}$. Then the prover computes low-degree polynomials g_i, h_i such that

$$\begin{aligned} s_{i,1}(f_{\mathbf{Az}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{A}}_{\alpha_i}) + s_{i,2}(f_{\mathbf{Bz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{B}}_{\alpha_i}) + s_{i,3}(f_{\mathbf{Cz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{C}}_{\alpha_i}) \\ = g_i + h_i \cdot Z_{H_1 \cup H_2} \end{aligned} \tag{10}$$

The prover then sends the Reed-Solomon codeword $\hat{h}_i := h_i|_L \in RS[L, \frac{|H_1 \cup H_2|}{|L|}]$ to the verifier. Note that after receiving $\hat{h}_i$, the verifier can compute queries to the virtual oracle $\hat{g}_i := g_i|_L = [s_{i,1}(f_{\mathbf{Az}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{A}}_{\alpha_i}) + s_{i,2}(f_{\mathbf{Bz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{B}}_{\alpha_i}) + s_{i,3}(f_{\mathbf{Cz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{C}}_{\alpha_i}) - h_i \cdot Z_{H_1 \cup H_2}]|_L$ via queries to $\hat{f}_{\mathbf{Az}}, \hat{f}_{\mathbf{Bz}}, \hat{f}_{\mathbf{Cz}}, \hat{f}_{\mathbf{z}}$ and $\hat{h}_i$, since it can compute $p^1_{\alpha_i}, p^{2,\mathbf{B}}_{\alpha_i}, Z_{H_1 \cup H_2}$ by itself.

3. **Reduction to FRI.** Finally, the prover and the verifier will repeat this part λ'_i times. In the j-th repetition for $j \in \{1, ..., \lambda'_i\}$, the verifier needs to use FRI low-degree test protocol (Section 4.2) to show that all the committed Reed-Solomon codewords correspond to low-degree polynomials of varying bounds. The prover needs to show that $\hat{f}_{\mathbf{w}}$ is indeed in the codeword set $RS[L, \frac{n-k}{|L|}]$, which is equivalent to showing that the degree of $f_{\mathbf{w}}$ is lower than $n-k$. The prover also needs to show that $\hat{f}_{\mathbf{Az}}, \hat{f}_{\mathbf{Bz}}, \hat{f}_{\mathbf{Cz}} \in RS[L, \frac{m}{|L|}]$, $\hat{h}_i \in RS[L, \frac{\max\{m,n+1\}}{|L|}]$, the virtual oracle $\hat{g}_i \in RS[L, \frac{\max\{m-1,n\}}{|L|}]$, and the virtual oracle $\frac{\hat{f}_{\mathbf{Az}} \cdot \hat{f}_{\mathbf{Bz}} - \hat{f}_{\mathbf{Cz}}}{Z_{H_1}|_L} \in RS[L, \frac{n}{|L|}]$. This last check is the Rowcheck procedure; recall formula (46). We also note that the degree bound for g is extremely important, since in Sumcheck we need to check g has a degree exactly lower than $\max\{m, n+1\} - 1$.

   To simultaneously run all above low-degree tests on all polynomials, Aurora combines all polynomials into one. The verifier first samples random coefficient vector $\mathbf{y}_j = (\mathbf{y}_{j,1}, \mathbf{y}_{j,2}, ..., \mathbf{y}_{j,5+3\lambda_i}) \in \mathbb{F}^{5+3\lambda_i}$ and sends $\mathbf{y}_j$ to the prover. The prover and the verifier then engage in one run of the FRI protocol where the first prover message is the virtual oracle corresponding to the Reed-Solomon codeword of the following polynomial with a degree less than $2\max\{m, n+1\}$:

$$\begin{aligned} f_{j,0} := \mathbf{y}_{j,1} \cdot f_{\mathbf{w}} + \mathbf{y}_{j,2} \cdot f_{\mathbf{Az}} + \mathbf{y}_{j,3} \cdot f_{\mathbf{Bz}} + \mathbf{y}_{j,4} \cdot f_{\mathbf{Cz}} + \mathbf{y}_{j,5} \cdot \frac{f_{\mathbf{Az}} \cdot f_{\mathbf{Bz}} - f_{\mathbf{Cz}}}{Z_{H_1}} \\ + \sum_{i=1}^{\lambda_i}(\mathbf{y}_{j,5+i} \cdot h_i) + \sum_{i=1}^{\lambda_i}(\mathbf{y}_{j,5+\lambda_i+i} \cdot g_i) \\ + \sum_{i=1}^{\lambda_i}(\mathbf{y}_{j,5+2\lambda_i+i} \cdot X^{(2\max\{m,n+1\})-(\max\{m,n+1\}-1)} \cdot g_i) \end{aligned} \tag{11}$$

Intuitively, Aurora takes a random linear combination of all the committed polynomials and performs only one low-degree test on this random linear combination. However, since all polynomials have different claimed degrees, Aurora raises the degree of g_i to the maximum degree bound for the committed polynomials by multiplying the term $X^{(2\max\{m,n+1\})-(\max\{m,n+1\}-1)}$. This way, the polynomials $\{X^{(2\max\{m,n+1\})-(\max\{m,n+1\}-1)} \cdot g_i\}_i$ have degree bound $2\max\{m, n+1\}$, which becomes the maximum degree among all the tested polynomials.

To formally prove that the idea of shifting important polynomials and using random linear combination really works, we refer to the original paper [BCR+19], the soundness part of Protocol 8.2 for a detailed analysis.

We summarize the flow of the Aurora IOP in Figure 1.

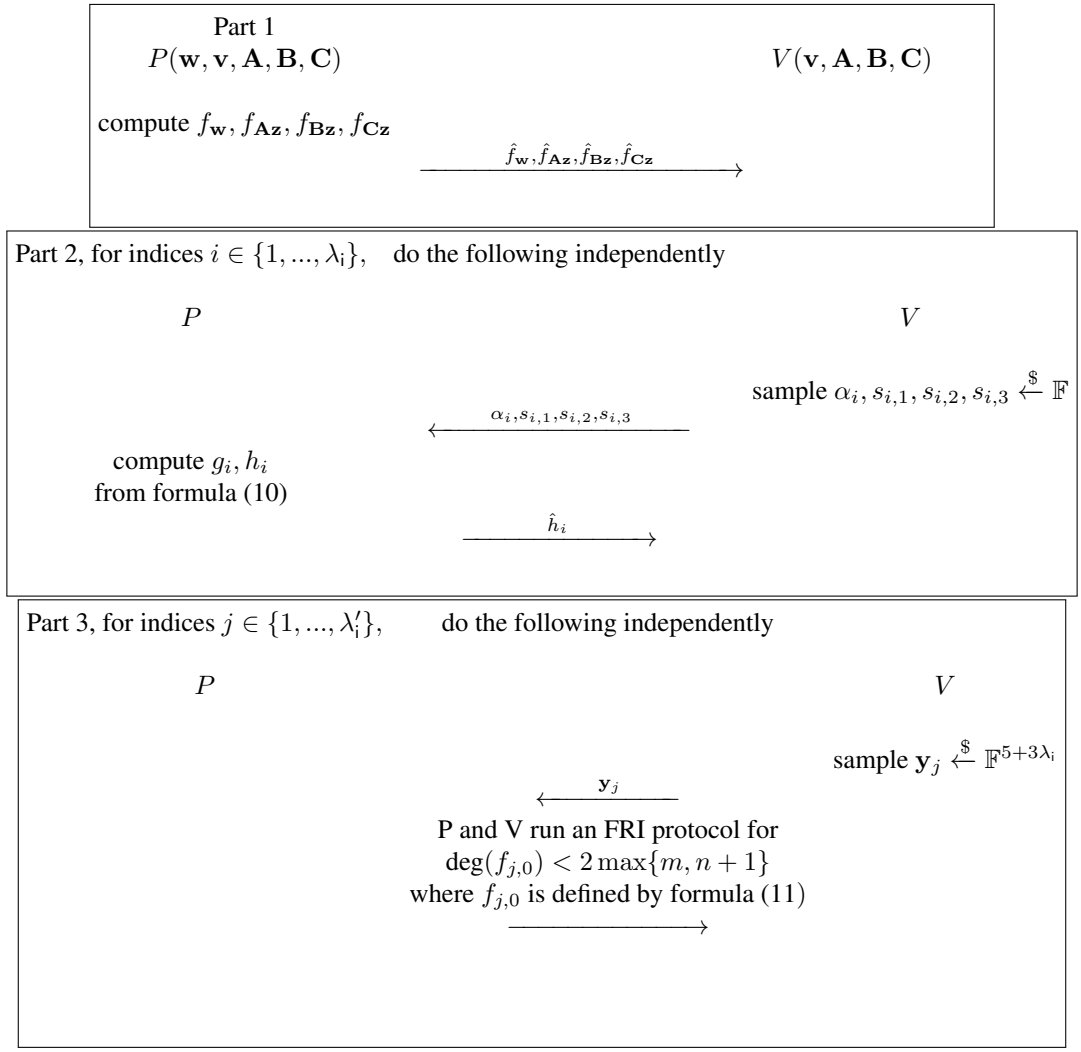


Figure 1: The Aurora IOP

4.2 FRI Low-degree Test

Aurora uses the Fast Reed-Solomon Interactive oracle proof of proximity (FRI) low-degree test protocol as one of its core components. Here we describe how the protocol works at a high level. For a more detailed protocol description and analysis, refer to [BBHR17].

In the FRI protocol, the prover wants to prove the statement "I know a polynomial f whose degree is lower than d." The FRI protocol contains two phases: the commit phase and the query phase.

4.2.1 Commit Phase

The commit phase contains about $\log d$ rounds. In each round, the prover starts with a polynomial f_i, where f_0 is defined as the claimed low-degree polynomial f. The prover first computes the Reed-Solomon codeword of f_i over an additive coset L_i, where L_0 is the predetermined space L. In order to encode the full information of f_0, the set L should be chosen to be sufficiently large. Then the prover sends the corresponding Reed-Solomon codeword $\hat{f}_i$ as a message oracle to the verifier as the commitment. The verifier then replies with a uniformly sampled random element $x_i \in \mathbb{F}$ to the prover. After receiving the reply, the prover first rewrites f_i with a predetermined public round polynomial $\{q_i(x)\}_i$ as

$$f_i(x) = f_{i,0}(q_i(x)) + x \cdot f_{i,1}(q_i(x)) + x^2 \cdot f_{i,2}(q_i(x)) + ... + x^{deg(q_i)-1} \cdot f_{i,deg(q_i)-1}(q_i(x)) \tag{12}$$

and then computes

$$f_{i+1}(x) := f_{i,0}(x) + x_i \cdot f_{i,1}(x) + x_i^2 \cdot f_{i,2}(x) + ... + x_i^{deg(q_i)-1} \cdot f_{i,deg(q_i)-1}(x) \tag{13}$$

by evaluating $f_{i,0}$ and $f_{i,1}$ at the receiving point x_i. They then continue to the next round with the folded polynomial f_{i+1}, as well as the domain

$$L_{i+1} := \{q_i(x) | x \in L_i\}, \tag{14}$$

over which the Reed-Solomon codewords will be constructed.

The commit phase ends when the polynomial f_i has a degree at most one. In this case, the prover sends all the coefficients of f_i (one or two field elements) directly to the verifier. Form (12) and (13), we have $deg(f_{i+1}) \leq \frac{deg(f_i)}{deg(q_i)}$. So, for example, if all the polynomials $\{q_i\}_i$ have degree $2^\eta := 2$, then there will be $\lfloor \log d \rfloor$ rounds. After these rounds, $f_{\lfloor \log d \rfloor}$ will have a degree at most one.

The concrete setting of the predetermined round polynomials $q_i(x)$ for our use will be presented in section 7.

4.2.2 Query Phase

After the commit phase, the prover and the verifier enter the query phase. In this phase, the verifier asks the prover to open some entries of committed Reed-Solomon codewords and then checks the consistency between each f_i and f_{i+1} using these opened values. The consistency check fails if the prover did not honestly compute f_{i+1} from f_i and x_i via the rules (12) and (13).

Specifically, suppose there are r rounds in the commit phase. At the start of the query phase, the verifier first samples a challenge element y_0 uniformly from L_0 and sends y_0 to the prover. Then the prover opens values $f_i(y_i), f_i(y_i^{(1)}), f_i(y_i^{(2)}), ..., f_i(y_i^{(deg(q_i)-1)})$ for all $i \in \{0, 1, ..., \mathsf{r}-2\}$ to the verifier, where $y_i, y_i^{(1)}, ..., y_i^{(deg(q_i)-1)}$ are $deg(q_i)$ distinct elements in L_i that share the same image under q_i, i.e., $q_i(y_i) = q_i(y_i^{(1)}) = ... = q_i(y_i^{(deg(q_i)-1)})$. The prover also computes $y_{i+1} := q_i(y_i)$ and opens $f_{i+1}(y_{i+1})$

to the verifier. To check the consistency of f_i and f_{i+1}, the verifier interpolates a polynomial $F_i(x)$ with degree less than $deg(q_i)$ such that

$$F_i(x) = \begin{cases} f_i(y_i) & \text{for} \quad x = y_i \\ f_i(y_i^{(j)}) & \text{for} \quad x = y_i^{(j)} \end{cases} \text{where } j = 1, 2, ..., deg(q_i) - 1 \tag{15}$$

and then checks if $F_i(x_i)$ equals to $f_{i+1}(y_{i+1})$. If $F_i(x_i)$ did not equal to $f_{i+1}(y_{i+1})$ for any $i \in \{0, 1, ..., \mathsf{r} - 1\}$, the verifier rejects. The verifier only accepts if all checks pass. Note when $i = \mathsf{r} - 1$, since the verifier already received all the coefficients of f_{r}, the verifier can compute $f_{\mathsf{r}}(y_{\mathsf{r}})$ alone. The prover only opens $f_{\mathsf{r}-1}(y_{\mathsf{r}-1}), f_{\mathsf{r}-1}(y_{\mathsf{r}-1}^{(1)}), ..., f_{\mathsf{r}-1}(y_{\mathsf{r}-1}^{(deg(q_{\mathsf{r}-1})-1)})$.

To boost the soundness of the FRI protocol, one will parallel execute the query phase for ℓ times independently. The verifier then outputs accept if and only if it accepts all the query phases.

The interaction flow of the FRI protocol is shown in Figure 2.

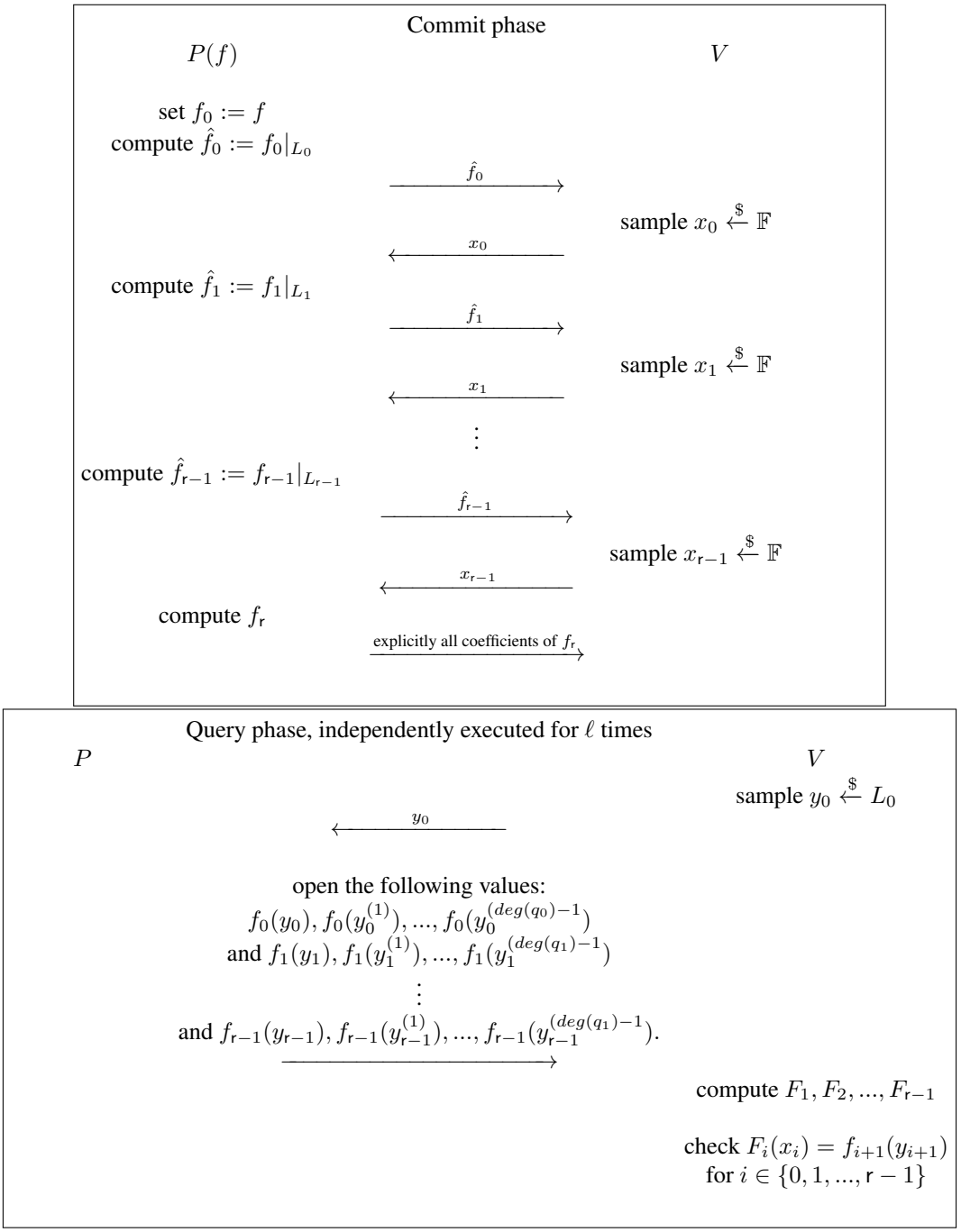


Figure 2: FRI protocol

4.3 The BCS Transform

In [BCS16], the authors introduced a transformation, called the BCS transform in this document, which transforms a public-coin IOP into a non-interactive random oracle proof.

The BCS transform contains two parts. The first part uses Merkle commitment to instantiate two procedures of IOP: "commit to message oracle" and "open at given queried positions." Using a random function, the prover builds a Merkle tree with the messages to commit as leaves and sends the Merkle root as the commitment to the verifier. To answer a query from the verifier, the prover sends the message leaf along with the authentication path from that leaf to the Merkle root.

The other part makes the protocol non-interactive using the Fiat-Shamir transform. All random coins in the protocol are generated by applying a random function, again modeled as a random oracle, on previously generated public messages.

In [BCS16], the BCS transform is proven to preserve soundness, proof-of-knowledge, as well as zero-knowledge properties in the random oracle model. In [CMS19], the authors prove that the BCS transform (with a slight modification) preserves these properties in the quantum random oracle model.

4.4 Achieving Zero Knowledge

The Aurora protocol described so far doesn't have the zero-knowledge property yet. It can be made to be zero-knowledge by using the following modifications, which are designed to prevent the verifier from obtaining private information through queries to the message oracles.

Choosing L to be Disjoint from $H_1 \cup H_2$. In zero-knowledge mode, Aurora should be configured such that the subspace $H_1 \cup H_2$ does not intersect the additive coset L, so that the verifier cannot query the private witness values directly.

Adding Random Polynomials to Mask Original Polynomials. We will add a new parameter $\mathsf{b} \in \mathbb{N}$ to represent an estimated upper bound for the total number of queries made by the verifier to all of the message oracles.

In zero-knowledge mode, the polynomial $f_{\mathbf{w}}$ is chosen uniformly at random from all polynomials with degrees lower than $n - k + \mathsf{b}$ that satisfy the system of equations (2). Similarly, each of the polynomials $f_{\mathbf{Az}}, f_{\mathbf{Bz}}, f_{\mathbf{Cz}}$ are chosen uniformly at random from polynomials with degree lower than $m + \mathsf{b}$ such that $f_{\mathbf{Az}}$ satisfies (3), $f_{\mathbf{Bz}}$ satisfies (4), and $f_{\mathbf{Cz}}$ satisfies (5), respectively.

In addition, at the beginning of Aurora's Lincheck protocol, for each index $i \in \{1, ..., \lambda_{\mathsf{i}}\}$, the prover first samples a random polynomial r_i with degree less than $2|H_1 \cup H_2| + \mathsf{b} - 1 = 2\max\{m, n+1\} + \mathsf{b} - 1$, computes $\mu_i := \sum_{x \in H_1 \cup H_2} r_i(x) \in \mathbb{F}$, and then sends μ_i and the Reed-Solomon codeword message oracle $\hat{r}_i := r_i|_L \in RS[L, \frac{2\max\{m,n+1\}+\mathsf{b}-1}{|L|}]$ to the verifier. The verifier replies with random elements $\alpha_i, s_{i,1}, s_{i,2}, s_{i,3}$. The prover then proves the statement "$s_{i,1}(f_{\mathbf{Az}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{A}}_{\alpha_i}) + s_{i,2}(f_{\mathbf{Bz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{B}}_{\alpha_i}) + s_{i,3}(f_{\mathbf{Cz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{C}}_{\alpha_i}) + r_i$ sums to μ_i on the set $H_1 \cup H_2$." To do this, we need to adjust the amortized Sumcheck protocol. The prover computes $\xi := \sum_{x \in H_1 \cup H_2} x^{(\max\{m,n+1\}-1)}$ and then computes

$$\begin{aligned} s_{i,1}(f_{\mathbf{Az}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{A}}_{\alpha_i}) + s_{i,2}(f_{\mathbf{Bz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{B}}_{\alpha_i}) + s_{i,3}(f_{\mathbf{Cz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{C}}_{\alpha_i}) + r_i \\ = g_i + \frac{\mu_i}{\xi} \cdot x^{|H_1 \cup H_2|-1} + h_i \cdot Z_{H_1 \cup H_2} \end{aligned} \tag{16}$$

to get the polynomial g_i with degree less than $|H_1 \cup H_2| - 1$ and the polynomial h_i with degree less than $|H_1 \cup H_2| + \mathsf{b}$. The prover then sends the Reed-Solomon codeword message oracle $\hat{h}_i := h_i|_L \in$

$RS[L, \frac{\max\{m,n+1\}+\mathsf{b}}{|L|}]$ to the verifier. From this point, the queries to the virtual oracle $\hat{g}_i := g_i|_L \in RS[L, \frac{\max\{m,n+1\}-1}{|L|}]$ where

$$g_i = s_{i,1}(f_{\mathbf{Az}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{A}}_{\alpha_i}) + s_{i,2}(f_{\mathbf{Bz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{B}}_{\alpha_i}) + s_{i,3}(f_{\mathbf{Cz}} \cdot p^1_{\alpha_i} - f_{\mathbf{z}} p^{2,\mathbf{C}}_{\alpha_i}) + r_i \\ - \frac{\mu_i}{\xi} \cdot x^{|H_1 \cup H_2|-1} - h_i \cdot Z_{H_1 \cup H_2} \tag{17}$$

can be computed by the verifier via queries to $\hat{f}_{\mathbf{z}}, \hat{f}_{\mathbf{Az}}, \hat{f}_{\mathbf{Bz}}, \hat{f}_{\mathbf{Cz}}$ and $\hat{h}_i$.

Finally, in Aurora's reduction to FRI, for each index $j \in \{1, ..., \lambda'_{\mathsf{i}}\}$, the prover samples a random masking polynomial $r^{(j)}_{LDT}$ with degree less than $2\max\{m, n+1\} + 2\mathsf{b}$ and sends message oracle $\hat{r}^{(j)}_{LDT} := r^{(j)}_{LDT}|_L \in RS[L, \frac{2\max\{m,n+1\}+2\mathsf{b}}{|L|}]$ to the verifier. The verifier then sample random coefficient vector $\mathbf{y}_j = (\mathbf{y}_{j,1}, \mathbf{y}_{j,2}, ..., \mathbf{y}_{j,5+4\lambda_{\mathsf{i}}}) \in \mathbb{F}^{5+4\lambda_{\mathsf{i}}}$ and sends $\mathbf{y}_j$ to the prover. Then the prover and the verifier engage in one run of FRI protocol where the first prover message is the virtual oracle corresponding to the Reed-Solomon codeword of the following polynomial with degree less than $2\max\{m, n+1\} + 2\mathsf{b}$:

$$\begin{aligned} f_{j,0} := \mathbf{y}_{j,1} \cdot f_{\mathbf{w}} + \mathbf{y}_{j,2} \cdot f_{\mathbf{Az}} + \mathbf{y}_{j,3} \cdot f_{\mathbf{Bz}} + \mathbf{y}_{j,4} \cdot f_{\mathbf{Cz}} + \mathbf{y}_{j,5} \cdot \frac{f_{\mathbf{Az}} \cdot f_{\mathbf{Bz}} - f_{\mathbf{Cz}}}{Z_{H_1}} \\ + \sum_{i=1}^{\lambda_{\mathsf{i}}} (\mathbf{y}_{j,5+i} \cdot r_i) + \sum_{i=1}^{\lambda_{\mathsf{i}}} (\mathbf{y}_{j,5+\lambda_{\mathsf{i}}+i} \cdot h_i) + \sum_{i=1}^{\lambda_{\mathsf{i}}} (\mathbf{y}_{j,5+2\lambda_{\mathsf{i}}+i} \cdot g_i) \\ + \sum_{i=1}^{\lambda_{\mathsf{i}}} (\mathbf{y}_{j,5+3\lambda_{\mathsf{i}}+i} \cdot X^{(2\max\{m,n+1\}+2\mathsf{b})-(\max\{m,n+1\}-1)} \cdot g_i) \\ + r^{(j)}_{LDT} \end{aligned} \tag{18}$$

Remark 4.1 *An optimization for choosing the random polynomials r_i is that one can choose r_i uniformly from the set of polynomials that with degree less than $2\max\{m, n+1\} + \mathsf{b} - 1$ and sums to zero on the set $|H_1 \cup H_2|$. In this way, the protocol is still zero-knowledge since the simulator can still simulate the distribution of the masked polynomial perfectly, and one can set the μ_i in formula (16), (17) to be zero which saves some computation effort. We note that this sampling procedure of r_i can be done efficiently (see our implementation for more details).*

Adding Randomness to Merkle Tree Construction. In [BCS16], the authors show that the BCS transform preserves zero-knowledge property if the prover pads the messages at the leaves of the Merkle tree with random bit strings, which will be included as part of the authentication path when opening a message leaf.

4.5 Security Analyses

4.5.1 Knowledge Soundness

Interactive Aurora Knowledge Soundness. We have the following theorem from [BCR+19].

Theorem 4.2 *([BCR+19], Theorem 9.2) Given an R1CS with n variables and m constraints, let $\rho \in (0,1)$ be a constant and L be any subspace of $\mathbb{F}$ such that $\rho|L| > 2\max\{m, n+1\}+2\mathsf{b}$ and let $\delta := \min\{\frac{1-2\rho}{2}, \frac{1-\rho}{3}, 1-\rho\}$, then the interactive Aurora protocol is a zero-knowledge proof of knowledge IOP against b queries with knowledge soundness error upper bounded by:*

$$\epsilon \leq (\frac{m+1}{|\mathbb{F}|})^{\lambda_{\mathsf{i}}} + (\frac{|L|}{|\mathbb{F}|})^{\lambda'_{\mathsf{i}}} + \epsilon_{FRI}(\delta), \tag{19}$$

where $\epsilon_{FRI}(\delta)$ is the soundness error of the FRI protocol with proximity parameter δ, and $\lambda_{\mathrm{i}}, \lambda'_{\mathrm{i}}$ are repetition parameters described in section 4.1.

It is conjectured that the requirement of δ is not necessary for the theorem:

Conjecture 4.3 *Theorem 4.2 holds for $\delta := 1 - \rho$ instead of $\delta := \min\{\frac{1-2\rho}{2}, \frac{1-\rho}{3}, 1-\rho\}$.*

Interactive FRI Soundness. [BBHR17] gives the soundness error of the FRI protocol as follows.

Theorem 4.4 *([BBHR17], Theorem 3.3) Let L_0 be an additive coset of $\mathbb{F}$ with order $|L_0|$ a power of 2. When the FRI protocol is invoked on the oracle $f_0 : L_0 \to \mathbb{F}$ with localization parameter η, rate parameter ρ such that $\rho|L_0| > 16$ and query phase repetition parameter ℓ, let $\delta := \Delta(f_0, RS[L_0, \rho])$ be the proximity parameter. Then the soundness error of the FRI protocol is upper bounded by:*

$$\epsilon_{FRI}(\delta) \leq \frac{3|L_0|}{|\mathbb{F}|} + (1 - \min\{\delta, \frac{1 - 3\rho - \frac{2^\eta}{\sqrt{|L_0|}}}{4}\})^\ell. \tag{20}$$

We mention that the FRI protocol is an interactive oracle proof "of proximity," which has a different definition of the soundness property. We say that FRI has soundness error $\epsilon_{FRI}(\delta)$ with proximity parameter δ if for every $f_0 \in \mathbb{F}^{|L_0|}$ that has been sent by the prover as the first message oracle to the verifier with $\Delta(f_0, RS[L_0, \rho]) \geq \delta$, the probability that the verifier accepts after executing the FRI protocol is at most $\epsilon_{FRI}(\delta)$.

After [BBHR17], a series of works tried to improve the bound [BGKS19, BCI$^+$20]. To the best of our knowledge, the best upper bound for the soundness error of the FRI protocol is given by:

Theorem 4.5 *([BCI$^+$20], Theorem 8.3, adapted to our notation) When the FRI protocol is invoked on the oracle $f_0 : L_0 \to \mathbb{F}$ with localization parameter η, rate parameter ρ, query phase repetition parameter ℓ, let u be an integer with $u \geq 3$. Let $\delta := \Delta(f_0, RS[L_0, \rho])$ be the proximity parameter. Then under the condition $1 - \sqrt{\rho}(1 + \frac{1}{2u}) < \delta$, the soundness error of the FRI protocol is upper bounded by:*

$$\epsilon_{FRI}(\delta) \leq \frac{(u + \frac{1}{2})^7 \cdot |L_0|^2}{2\rho^{3/2}|\mathbb{F}|} + \frac{(2u+1)(|L_0|+1)}{\sqrt{\rho}} \cdot \frac{(log_{2^\eta}|L_0|)(2^\eta - 1)}{|\mathbb{F}|} + (\sqrt{\rho}(1 + \frac{1}{2u}))^\ell. \tag{21}$$

In our case, unfortunately, the Aurora protocol uses the proximity parameter $\delta := \min\{\frac{1-2\rho}{2}, \frac{1-\rho}{3}, 1-\rho\}$, which may result in $u < 3$ for our concrete parameter set, so we have to use some heuristics in order to use the result from Theorem 4.5.

BCS Transform Preserves Knowledge Soundness. According to [BCS16], the BCS transform can preserve soundness in the random oracle model:

Theorem 4.6 *([BCS16], Theorem 7.1) For every relation $\mathcal{R}$, let (P, V) be an IOP protocol for relation $\mathcal{R}$, and let $(\mathbb{P}, \mathbb{V})$ be the non-interactive oracle proof obtained by applying the BCS transform to (P, V). Then we have the following:*

$$\epsilon'(q) \leq \bar{\epsilon}_{sr}(q) + \frac{3(q^2+1)}{2^\lambda}, \tag{22}$$

where $\epsilon'(q)$ is the knowledge soundness error of $(\mathbb{P}, \mathbb{V})$ against a malicious adversary which can perform q queries to the random oracle, $\bar{\epsilon}_{sr}(q)$ is the restricted state-restoration attack knowledge soundness error of (P, V) against q times of state restoration, and λ is the random oracle output length measured in bits.

Informally speaking, the state-restoration attack knowledge soundness error is analogous to the knowledge soundness error. The difference is that here we are considering a malicious "state-restoring prover" to convince the verifier to accept at the end of the attack. A state-restoration prover is a generalization of the usual malicious prover where the prover has the added ability to "restore" the verifier to any previously seen verifier state, continues the protocol from that point for one more round of interaction, and then records the verifier's state and performs the next state restoration. For the formal definition of state-restoration attack, refer to [BCS16], section 5.1. A state-restoration prover can restore the verifier's state many times when trying to convince the verifier to accept at the end of the attack. Note that in order to use this theorem, we have to use the following conjectured Aurora property:

Conjecture 4.7 *The restricted state-restoration knowledge soundness error of the interactive Aurora protocol against q times of state-restoration is upper bounded by the formula*

$$\bar{\epsilon}_{sr}(q) \leq q \cdot \epsilon, \tag{23}$$

where ϵ is the knowledge soundness error of the interactive Aurora protocol, which is in turn upper bounded by formula (19).

Remark 4.8 *The restricted state-restoration knowledge soundness is a weaker notion of state-restoration knowledge soundness in the sense that in the restricted case, the state-restoration prover cannot restore the verifier's state to the null state (except for the initial trial). This means that we have $\bar{\epsilon}_{sr}(q) \leq \epsilon_{sr}(q)$ where $\epsilon_{sr}(q)$ is the state-restoration attack soundness error against q times of state restoration. In the case of Aurora, restricted state-restoration attack and state-restoration attack are equivalent, since the prover is the one who makes the first move in the interactive protocol (here we use the same argument as in the analysis of [MRV+21] about state-restoration soundness).*

We recall that in [CMS19], the authors are able to show that the BCS transform (with some modification) can preserve knowledge soundness property in the quantum random oracle model, provided that the original IOP protocol satisfies a knowledge soundness notion called "round-by-round knowledge soundness":

Theorem 4.9 *([CMS19], Theorem 8.6, item 1) Let (P, V) be an IOP for a relation $\mathcal{R}$ with proof length l and query complexity q. Then the BCS transform, when based on (P, V), is a non-interactive argument for relation $\mathcal{R}$ such that if (P, V) have round-by-round knowledge soundness error ϵ, then the argument has knowledge soundness error $O(t^2\epsilon + t^3/2^\lambda)$ against quantum attackers that makes at most $t - O(q \log l)$ quantum queries to the random oracle.*

Informally speaking, the round-by-round soundness error is a quantity that upper bounds the probability that a malicious prover can cheat the verifier in each single round of the IOP protocol, and the round-by-round knowledge soundness error is the "proof of knowledge" version of this notion. For the formal definition of round-by-round knowledge soundness, refer to [CMS19], section 8.3. In [Hol19], the author shows a relation between the state-restoration soundness error and the round-by-round soundness error:

Theorem 4.10 *([Hol19], Theorem 3.2, adapted to our notation) If (P, V) is an r-round IOP with state-restoration soundness error $\epsilon_{sr}(q)$, then (P, V) is an IOP with round-by-round soundness error $\epsilon_{rbr} \leq \frac{r}{q} \ln \frac{2r}{1-\epsilon_{sr}(q)}$.*

Here we conjecture that this formula also holds in the knowledge soundness case:

Conjecture 4.11 *If (P, V) is an r-round IOP with state-restoration knowledge soundness error $\epsilon_{sr}(q)$, then (P, V) is an IOP with round-by-round knowledge soundness error $\epsilon_{rbr} \leq \frac{r}{q} \ln \frac{2r}{1-\epsilon_{sr}(q)}$.*

For example, if we assume Conjecture 4.11 holds, and the interactive Aurora protocol has restricted state-restoration knowledge soundness error $\bar{\epsilon}_{sr}(q) \leq \frac{q}{2^{128}}$ (as the state-restoration knowledge soundness error by Remark 4.8), then we can see that the round-by-round knowledge soundness error of the interactive Aurora protocol is at most $\frac{1}{2^{115}}$ by using $r = 32$ as an upper bound of rounds for $q = 2^{128} - 1$.

When trying to apply Theorem 4.9, we are also going to consider the following conjecture for Aurora's knowledge soundness property:

Conjecture 4.12 *The round-by-round knowledge soundness error of the interactive Aurora protocol is upper bounded by the formula*

$$\epsilon_{rbr} \leq \epsilon, \tag{24}$$

where ϵ is the knowledge soundness error of the interactive Aurora protocol, which is in turn upper bounded by formula (19)

4.5.2 Zero Knowledge

Interactive Aurora Protocol. The interactive Aurora protocol is an IOP protocol with perfect zero-knowledge against b queries to the message oracle [BCR+19]. This means that there exists a probabilistic polynomial-time simulator algorithm S that satisfies the following. S takes only the public instance as input. When given straight-line access to a verifier that makes at most b queries to the message oracle, S can output a view of the verifier having the same distribution as that of a verifier that interacts with a prover truly possessing the knowledge of a private witness of the claimed relation. In other words, the simulator is able to simulate the view of the verifier perfectly without knowing the private witness, which means that the verifier cannot get any information about the witness from the prover during the interaction.

BCS Transform Preserves Zero Knowledge. The BCS transform can preserve zero-knowledge in the explicit programmable random oracle model. More precisely, in [BCS16] the author shows that if the original IOP protocol is z-statistical zero-knowledge, then after applying BCS transform, the protocol becomes a non-interactive random oracle proof that is z'-statistical zero-knowledge in the explicit programmable random oracle model for

$$z' = z + \frac{p}{2^{\lambda/4+2}}, \tag{25}$$

where p is the proof length and λ, the output length of the random oracle.

Note that the bound used in the paper (Fact 1 in Lemma 3.4 [BCS16]) is very loose, so it is possible to adjust the Chernoff bound to get a loss smaller than the term $\frac{p}{2^{\lambda/4+2}}$. We provide a more detailed explanation for how to tweak formula (25) in Appendix B.

4.6 Possible Attacks to the Non-interactive Zero-knowledge Aurora Protocol

4.6.1 Grinding Attack

Grinding attack is a well-known attack on non-interactive protocols obtained from transforms analogous to Fiat-Shamir. The idea is that since the verifier's random coins come from the hash value of previous committed information, a malicious prover can control each round's random coins by trying different prover messages.

We prevent this type of attack by setting the public coin space (the hash output length and the size of the public coin domain) sufficiently large so that it's impractical for the prover to grind the public coins.

4.6.2 Attacks to the FRI Protocol

We describe a known attack to the FRI protocol where a malicious prover can convince the verifier that it knows a polynomial f_0 with corresponding Reed-Solomon codeword $\hat{f}_0 := f_0 \in RS[L, \rho]$ with probability ρ^ℓ no matter what initial polynomial f_0 is. In the attack, the prover will commit the message $f_0|_L$ to the verifier, and then the verifier will sample x_0 to the prover. The malicious prover then first picks $\rho|L|$ distinct positions $l_1, l_2, ..., l_{\rho|L|} \in L_0$ in a way that $l_1, l_2, ..., l_{\rho|L|}$ can be partitioned into $\frac{\rho|L|}{deg(q_0)}$ sets such that each of the sets corresponds to the same image value when applying the polynomial $q_0(x)$ to its elements, and then compute $\bar{f}_0$ to be the unique polynomial with degree less than $\rho|L|$ such that $\bar{f}_0(l_i) = f_0(l_i)$ for all $i \in \{1, 2, ..., \rho|L|\}$. The malicious prover uses $\bar{f}_0$ and x_0 to compute f_1 honestly as FRI does and then sends the commitment of the message $f_1|_{L_1}$ to the verifier. It then continues the FRI protocol honestly for the remaining rounds. Note that the verifier will sample a point $y_0 \in L_0$ to challenge the prover to open the corresponding preimage set of f_0 with image value $q_0(y_0)$ and $f_1(q_0(y_0))$. Observe that if $y_0 \in \{l_1, ..., l_{\rho|L|}\}$ then there is no way for the verifier to distinguish f_0 and $\bar{f}_0$ by the opening value given by the prover, so in this case, the verifier will accept the proof. The probability that $y_0 \in \{l_1, ..., l_{\rho|L|}\}$ is exactly $\frac{\rho|L|}{|L|} = \rho$. Since there are total ℓ independent repetitions of the query phase in a single execution of FRI, we can see that this malicious prover has probability ρ^ℓ to succeed.

We mention that the malicious prover's success probability can be improved to $(1 - \delta_0)^\ell$ for $\delta_0 := \Delta(\hat{f}_0, RS[L, \rho])$ by first using list decodeing algorithm such as Guruswami–Sudan algorithm to find a nearest codeword $\hat{f}'_0 \in RS[L, \rho]$ such that $\Delta(\hat{f}_0, \hat{f}'_0) = \delta_0$ and then use $\hat{f}'_0$ as $\bar{f}_0$ in the attack described in the last paragraph. But this attack won't happen in the scenario of the Aurora's use of FRI, since in the Aurora case the probability that $(1-\delta_0)|L|$ is smaller than the tested degree (over the public coins of the verfier which are used to compute f_0) is negligible due to the guarantee of the Lincheck phase and the Reduce to FRI phase of the Aurora protocol.

5 AES Constraints

In this section, we describe a method to turn each component of the AES cipher [DBN+01] into R1CS format. We use $GF_8 := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$ to denote the native field of AES. Let $GF_{64} := \mathbb{F}_2[x]/(x^{64} + x^4 + x^3 + x + 1)$, and our native field for the Aurora proving system will be extension fields of GF_{64} (extended to a proper size for security).

Since there are a large number of variables that would be present in this section, the notations in this section are independent from the other sections of this document. We use $\bar{x}$ to indicate the variable x is a Boolean variable. Lower-case variables in a bold font like $\mathbf{v}$ are vectors, while upper-case variables in a bold font like $\mathbf{A}$ are matrices.

5.1 Constraints for the Four Steps of AES

Let b_j be the j-th input byte of AES to a step function for $j \in \{0, 1, \cdots, 15\}$. We use b'_j to represent the j-th output byte from the step function for $j \in \{0, 1, \cdots, 15\}$. Let $k_{i,j}$ be the round keys of AES for $j \in \{0, 1, \cdots, 15\}$ in i-th round.

AddRoundKey In each round, $b' = b \oplus k$, where k is the corresponding round key of b.

SubBytes In this step, the Sbox function is applied to each input byte b. The first operation of Sbox is to invert b in a suitable field. For $b \neq 0$, the set of inversion constraints that ensures $y = b^{-1} \mod (x^8 + x^4 + x^3 + x + 1)$ consists of $b * y = 1 + h \times (x^8 + x^4 + x^3 + x + 1)$, as well as the range proof for h, which

consists of $h = \bar{h}_0 + \bar{h}_1 x + \cdots + \bar{h}_6 x^6$ and $\bar{h}_k \times (\bar{h}_k - 1) = 0$ for $k \in \{0, \cdots, 6\}$. For the bit-rotation operation in SubBytes step, the constraints of bit-wise representation for b^{-1} are: $y = \bar{y}_0 + \bar{y}_1 x + \cdots + \bar{y}_7 x^7$, $\bar{y}_k \times (\bar{y}_k - 1) = 0$ for $k \in \{0, \cdots, 7\}$. The output of this step is thus:

$$b' = \begin{pmatrix} \bar{b}'_0 \\ \bar{b}'_1 \\ \bar{b}'_2 \\ \bar{b}'_3 \\ \bar{b}'_4 \\ \bar{b}'_5 \\ \bar{b}'_6 \\ \bar{b}'_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \bar{y}_0 \\ \bar{y}_1 \\ \bar{y}_2 \\ \bar{y}_3 \\ \bar{y}_4 \\ \bar{y}_5 \\ \bar{y}_6 \\ \bar{y}_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{26}$$

The field element 0 is non-invertible, so the "inverse" of $b = 0$ is set to be 0 in the Sbox of AES. However, we can combine these two conditions of b into one logic expression: $(b \neq 0 \wedge b * y = 1 + h(x^8 + x^4 + x^3 + x + 1) \wedge deg(y) \leq 7 \wedge deg(h) \leq 6) \vee (b = 0 \wedge y = 0)$. The corresponding constraints are:

- $b * y = 1 + h \times (x^8 + x^4 + x^3 + x + 1) + \bar{h}_7$;
- the range proof for h:
  $h = \bar{h}_0 + \bar{h}_1 x + \cdots + \bar{h}_6 x^6$, $\bar{h}_k \times (\bar{h}_k - 1) = 0$ for $k \in \{0, \cdots, 7\}$;
- $y = \bar{y}_0 + \bar{y}_1 x + \cdots + \bar{y}_7 x^7$, $\bar{y}_k \times (\bar{y}_k - 1) = 0$ for $k \in \{0, \cdots, 7\}$;
- $(\bar{h}_7)(\bar{h}_0 + \bar{h}_1 x + \cdots + \bar{h}_6 x^6 + (\bar{h}_7 + 1)x^7 + \bar{y}_0 x^8 + \cdots \bar{y}_7 x^{15}) = 0$.

As a result, this accounts for 19 constraints per byte so far.

Next, we will show an algebraic technique to merge two constraints of the SubBytes phase with the ShiftRows and MixColumns steps into a single constraint.

ShiftRows The constraints of this phase can be combined into MixColumns, since the operation of ShiftRows is equivalent to "re-indexing" the variables. Therefore, we can re-index the variables when loading them in the MixColumns step.

MixColumns We first re-index both for the ShiftRows and this step. Let

$$\begin{pmatrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \\ b_5 & b_6 & b_7 & b_4 \\ b_{10} & b_{11} & b_8 & b_9 \\ b_{15} & b_{12} & b_{13} & b_{14} \end{pmatrix}. \tag{27}$$

This re-indexing is just for readability. We don't need to write any constraint for the variable substitution. The MixColumns constraints are as follows:

$$\begin{pmatrix} c'_0 & c'_1 & c'_2 & c'_3 \\ c'_4 & c'_5 & c'_6 & c'_7 \\ c'_8 & c'_9 & c'_{10} & c'_{11} \\ c'_{12} & c'_{13} & c'_{14} & c'_{15} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{pmatrix}. \tag{28}$$

Normally we would need a modulo constraint to ensure the degree of the MixColumns output is less than 8. Here we introduce an efficient technique to avoid this additional constraint. The multiplication by 2 and

modulo operations on the outputs of SubBytes can be done by multiplying a matrix to the left of (26), which is

$$
\begin{aligned}
2b' &= \begin{pmatrix} 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&1\\ 0&0&0&1&0&0&0&1\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0 \end{pmatrix} \left(\begin{pmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} \bar{y}_0\\ \bar{y}_1\\ \bar{y}_2\\ \bar{y}_3\\ \bar{y}_4\\ \bar{y}_5\\ \bar{y}_6\\ \bar{y}_7 \end{pmatrix} + \begin{pmatrix} 0\\1\\1\\0\\0\\0\\1\\1 \end{pmatrix} \right) \\
&= \begin{pmatrix} 0&0&0&1&1&1&1&1\\ 1&0&0&1&0&0&0&0\\ 1&1&0&0&0&1&1&1\\ 1&1&1&1&1&1&0&0\\ 1&1&1&0&1&1&1&0\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0 \end{pmatrix} \begin{pmatrix} \bar{y}_0\\ \bar{y}_1\\ \bar{y}_2\\ \bar{y}_3\\ \bar{y}_4\\ \bar{y}_5\\ \bar{y}_6\\ \bar{y}_7 \end{pmatrix} + \begin{pmatrix} 0\\1\\1\\0\\0\\0\\1\\1 \end{pmatrix}. \qquad (29)
\end{aligned}
$$

Furthermore, $3b'$ can be obtained by adding equations (26) and (29). Note that the MixColumns operations are basically linear combinations of the input variables and thus can easily be merged with other constraints.

AES **Single Round.** Based on the description above, we are able to merge the linear operations of all three other steps into the constraints of the SubBytes step. First, the constraint of AddRoundKey is simply an addition, so it can be combined with the SubBytes step as the No. 1 constraint in Table 1. Secondly, the constraint of MixColumns is a linear combination of the input, which is a combination of the inverse of the SubBytes input and additional constant terms. Therefore, these two linear combinations can be combined, and the output of MixColumns is a linear combination of y, which costs only one constraint for each output byte. For the last round of AES, the whitening AddRoundKey before outputting the ciphertext can be combined into the last constraint in SubBytes as well. Furthermore, the last constraint of each round can further be combined with the first constraint of the next round.

As a result, the numbers of constraints describing the round functions are $16 \times 18 \times 10 = 2880$ for AES-128, 3456 for AES-192, and 4032 for AES-256, respectively.

No.	Constraints
1	$(b+k) * y = 1 + h(x^8 + x^4 + x^3 + x + 1) + \bar{h}_7$
2-9	$\bar{y}_i(\bar{y}_i + 1) = 0$ for $i \in \{0, \ldots, 7\}$
10-17	$\bar{h}_i(\bar{h}_i + 1) = 0$ for $i \in \{0, \ldots, 7\}$
18	$\bar{h}_7(\bar{y}_0 + \bar{y}_1 x + \ldots + \bar{y}_7 x^7 + \bar{h}_0 x^8 + \ldots + \bar{h}_6 x^{14} + (\bar{h}_7 + 1)x^{15}) = 0$

Table 1: Constraints for AES single round

Key Schedule. Let N be the length of the key in 32-bit words; that is, $N = 4, 6, 8$ for AES-128,192,256, respectively. The round keys of AES are expanded using the following formula.

$$W_i = \begin{cases} K_i & \text{for } i < N, \\ W_{i-N} \oplus \mathsf{SubWord}(\mathsf{RotWord}(W_{i-1})) \oplus rcon_{i/N} & \text{for } i \leq N \text{ and } i = 0 \bmod N, \\ W_{i-N} \oplus \mathsf{SubWord}(W_{i-1}) & \text{for } i \leq N, N > 6 \text{ and } i = 4 \bmod N, \\ W_{i-N} \oplus W_{i-1} & \text{otherwise,} \end{cases} \tag{30}$$

where W_i is a 4-byte word, RotWord ($[b_0\ b_1\ b_2\ b_3]$) = $[b_1\ b_2\ b_3\ b_0]$, and SubWord ($[b_0\ b_1\ b_2\ b_3]$) = $[(b_0)$ Sbox(b_1) Sbox(b_2) Sbox$(b_3)]$. The round constant $rcon_i = [rc_i\ 00_{16}\ 00_{16}\ 00_{16}]$ for round i of the key expansion is the 32-bit word, where rc_i can be computed in advance as shown in Table 2. The operations in the key expansion are basically Sbox and selective addition with the round constant rc_i to the first element in each round.

	1	2	3	4	5	6	7	8	9	10
rc_i	01	02	04	08	10	20	40	80	1B	36

Table 2: Round constant in AES key schedule with elements in hexadecimal representation

Note that for round i, we only need to store the variables $R_{i,j}$, which are the output of the SubWord function added by the $rcon$ of round i for $0 \leq j \leq 3$. Then the round keys can be expressed as combinations of the key basis $K = \{K_0, ..., K_{15}, R_{1,0}, ..., R_{l',3}\}$. For example, we can obtain the round keys of AES-128 using

$$k_{i,j} = \begin{cases} K_j & \text{for } i = 0, \\ R_{i,j} \oplus K_{i-1,j} & \text{for } j < 4, \\ K_{i,j-4} \oplus K_{i-1,j} & \text{otherwise.} \end{cases} \tag{31}$$

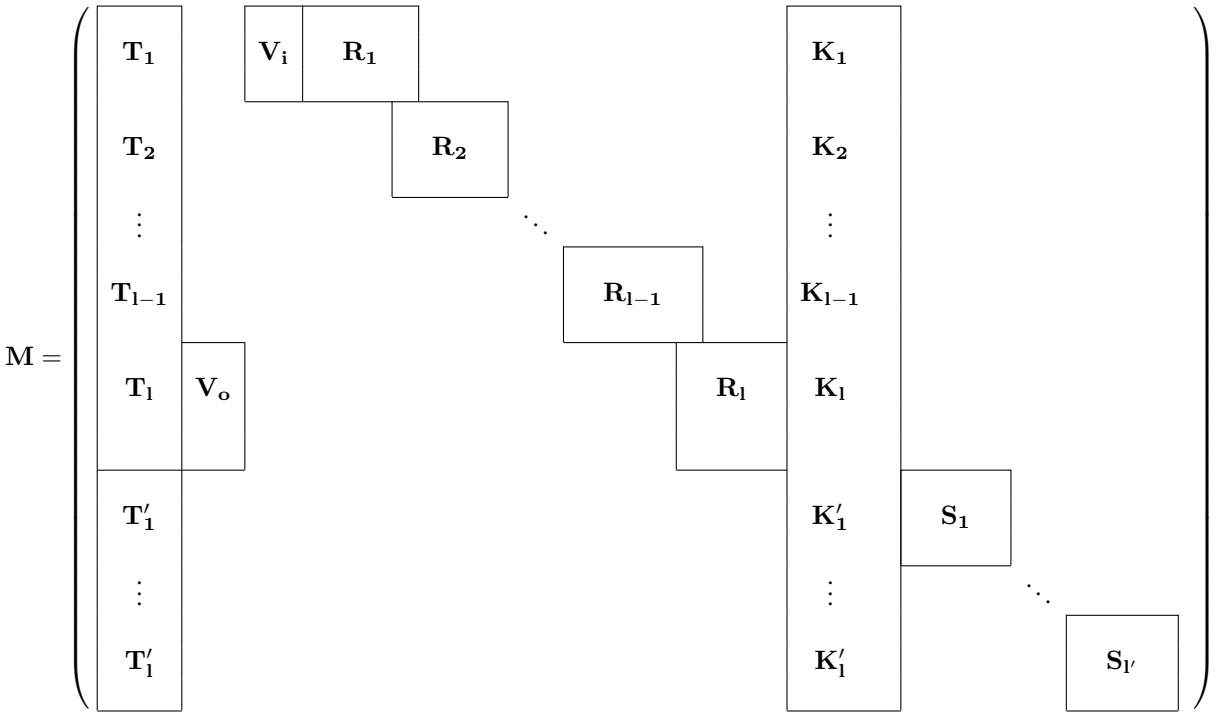
The formulas for AES-192 and AES-256 are similar to (31).

Next, similar techniques to what we used for merging the four steps in a single round can be used here again to combine the RotWord and addition of $rcon$ in key expansion, reducing the number of constraints to 18 for each variable. A slight difference is that we use one constraint for each $R_{i,j}$ to simplify the representation of $k_{i,j}$, unlike the merging of all the linear combinations in the AES rounds, where l' is the number of "AES key schedule rounds," and $l' = 10, 8, 13$ for AES-128, 192, 256, respectively. Therefore, the key expansion requires $19 \times 4 \times 10 = 760$ constraints for AES-128, and 380 for AES-192. For AES-256, the number of constraints is $19 \times 4 \times 13 = 988$.

To describe the entire AES encryption, we require the constraints for all round functions and key schedule, together with additional 16 constraints for the output (ciphertext). As a result, there are a total number of 3656, 4080, and 5036 constraints for AES-128, 192, 256, respectively.

5.2 R1CS Circuit for AES

To express the AES computation in an R1CS circuit $\mathbf{Az} \circ \mathbf{Bz} = \mathbf{Cz}$, it suffices to construct the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, all of which follow a general pattern as described by the following matrix $\mathbf{M}$.
Note that all elements outside the submatrix blocks in $\mathbf{M}$ are zero.



Here, the matrices $\mathbf{T_1}, \ldots, \mathbf{T_l}$ correspond to some constants in AES constraints, whereas the matrices $\mathbf{V}_{out}$, $\mathbf{V}_{in}$ correspond to the public outputs and inputs of AES. The matrices $\mathbf{R}_i$ are the round constraints, where $\mathbf{R}_2 = \ldots = \mathbf{R}_{l-1}$. $\mathbf{R}_1$ is the constraint for the first round, which takes input from the public variables instead. $\mathbf{R}_l$ is the constraint for the last round, which does not have MixColumns step but has an additional AddRoundKey step.

Specifically, $[\mathbf{T_1}, \ldots, \mathbf{T_l}]^T \in \mathbb{F}^{(288l+16)\times 1}$ and $[\mathbf{T'_1}, \ldots, \mathbf{T'_{l'}}]^T \in \mathbb{F}^{76l\times 1}$ are the matrices corresponding to the constants for AES round and key schedule round, respectively. $\mathbf{V}_o \in \mathbb{F}^{304\times 16}$ is the matrix corresponding to the output, that is, the ciphertext of the AES. $\mathbf{V}_i \in \mathbb{F}^{288\times 16}$ is the matrix corresponding to the input, that is, the plaintext of the AES.

The dimensions and arrangement of the variables in $\mathbf{z}$ and its subvectors are as follows.

$$\mathbf{z} = (1, \mathbf{v}, \mathbf{w}) \tag{32}$$

$$\mathbf{v} = (\underbrace{\mathbf{v}_{out}}_{16}, \underbrace{\mathbf{v}_{in}}_{16}) \tag{33}$$

$$\mathbf{w} = (\underbrace{\mathbf{w}_{R_1}, ..., \mathbf{w}_{R_l}}_{(16\times 8\times 2)\times l}, \underbrace{\mathbf{k}}_{56,56, \text{ or } 84}, \underbrace{\mathbf{w}_{S_1}, ..., \mathbf{w}_{S_{l'}}}_{(4\times 8\times 2)\times l'}) \tag{34}$$

First, $\mathbf{w}_{R_i} = (\bar{h}_{i,\alpha,0}, ..., \bar{h}_{i,\alpha,7})_{\alpha=0}^{15}, (\bar{y}_{i,\alpha,0}, ..., \bar{y}_{i,\alpha,7})_{\alpha=0}^{15}$, so the length of $\mathbf{w}_{R_i}$ is $(8+8)\times 16 = 256$, where l is the number of AES round. $\mathbf{w}_{S_i} = (\bar{h}_{i,\alpha,0}, ..., \bar{h}_{i,\alpha,7})_{\alpha=0}^{3}, (\bar{y}_{i,\alpha,0}, ..., \bar{y}_{i,\alpha,7})_{\alpha=0}^{3}$, so the length of $\mathbf{w}_{S_i}$ is $(8+8)\times 4 = 64$.

5.2.1 Circuit for AES Round

We need to following data to describe the submatrices for the AES circuit.

$$\begin{aligned}
\mathbf{e_7} &= (0,0,0,0,0,0,0,1)\\
\mathbf{v_1} &= (1, x, x^2, x^3, x^4, x^5, x^6, x^7)\\
\mathbf{v_2} &= \mathbf{v_1}x^8 = (x^8, x^9, x^{10}, x^{11}, x^{12}, x^{13}, x^{14}, x^{15})\\
f_8 &= 1 + x + x^3 + x^4 + x^8\\
\mathbf{v_f} &= (f_8, xf_8, x^2f_8, x^3f_8, x^4f_8, x^5f_8, x^6f_8, 1)
\end{aligned}$$

Let $\mathbf{w} \in \mathbb{F}^m$, we use $\mathbf{I}_\mathbf{w}^{(n)} = \mathbf{w} \otimes \mathbf{I_n}$ denote the following matrix.

$$\mathbf{I}_\mathbf{w}^{(n)} = \mathbf{w} \otimes \mathbf{I_n} = \begin{pmatrix} \mathbf{w} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \mathbf{w} \end{pmatrix} \in \mathbb{F}^{n \times mn}$$

We use the notation $\mathbf{A_{M_1||M_2||M_3}}$ to denote the matrix obtained by arranging the submatrices $\mathbf{M_1}, \mathbf{M_2}$, and $\mathbf{M_3}$ of $\mathbf{A}$ side by side; similarly, we build matrices like $\mathbf{B_{M_1||M_2||M_3}}$ and $\mathbf{C_{M_1||M_2||M_3}}$ from some submatrices $\mathbf{M_1}, \mathbf{M_2}$, and $\mathbf{M_3}$ of $\mathbf{B}$ and $\mathbf{C}$, respectively.

First, $\mathbf{A_{V_i||R_1||K_1}}$ describes the first round of AES: it takes the input from the public input in $\mathbf{V_i}$ and adds the round key k_i from $\mathbf{k}$ submatrix, both of which are identity matrix $\mathbf{I_{16}}$. $\mathbf{B_{T_i||R_i}}$ matrix takes the inverse of input, and $\mathbf{C_{T_i||R_i}}$ describes the right-hand side of the equation. For the second and third parts, the constraints on the variables are either 0 or 1 for all 128 variables. Fourth, the $\mathbf{A}$ matrix takes "$\bar{h}_7$" for each variable and $\mathbf{B}$ matrix takes $\bar{h}_0 + \bar{h}_1 x + \cdots + \bar{(h_7 + 1)}x^7 + \bar{y}_0 x^8 + \cdots + \bar{y}_6 x^{14} + \bar{y}_7 x^{15}$. The counterpart in $\mathbf{C}$ is 0. Note that $\mathbf{A_{T_1}}, \mathbf{A_{V_o}}, \mathbf{B_{V_o}}, \mathbf{B_{V_i}}, \mathbf{C_{V_i}}, \mathbf{B_{K_i}}$, and $\mathbf{C_{K_i}}$ are all zero for $1 \le i \le l$.

$$\mathbf{A_{V_i||R_1||K_1}} = \left(\begin{array}{c|c|c} \mathbf{I_{16}} & & \mathbf{I_{16}} \\ & \mathbf{I_{128}} & \\ & \mathbf{I_{128}} & \\ & \mathbf{I}_{\mathbf{e_7}}^{(16)} & \end{array}\right) \in F^{288 \times (16+256+56)}$$

$$\mathbf{B}_{\mathbf{T_i}||\mathbf{R_i}} = \left(\begin{array}{c|c|c} \begin{matrix} \\ \\ 1 \\ \vdots \\ 1 \\ 1 \\ \vdots \\ 1 \\ x^7 \\ \vdots \\ x^7 \end{matrix} & \mathbf{O}_{288\times 128} & \begin{matrix} \mathbf{I}^{(16)}_{\mathbf{v_1}} \\ \mathbf{I_{128}} \\ \mathbf{I_{128}} \\ \mathbf{I}^{(16)}_{\mathbf{v_1}} \end{matrix} \quad \begin{matrix} \\ \\ \\ \mathbf{I}^{(16)}_{\mathbf{v_2}} \end{matrix} \end{array}\right), \text{ for } 1 \le i < l$$

$$\mathbf{C}_{\mathbf{T_i}||\mathbf{R_i}} = \left(\begin{array}{c|c|c|c} \begin{matrix} 1 \\ \vdots \\ 1 \end{matrix} & \mathbf{O}_{16\times 128} & \mathbf{I}^{(16)}_{\mathbf{v_f}} & \mathbf{O}_{16\times 128} \end{array}\right), \text{ for } 1 \le i \le l$$

The matrices for AES round 2 to $l-1$ are similar to that of the first round, except for the inputs and round keys of each round. Specifically, the input is a combination of the "$\bar{y}$" variables from the previous round, while the round keys are combinations of the key basis. The $\mathbf{0}$ in $\mathbf{MC}$ is actually a zero vector of dimension 8. We use the hexadecimal number $63 = 1 + x + x^5 + x^6$ to simplify the representation of $\mathbf{A}_{\mathbf{T_i}||\mathbf{R_i}||\mathbf{K_i}}$.

$$\mathbf{A}_{\mathbf{T_i}||\mathbf{R_i}||\mathbf{K_i}} = \left(\begin{array}{c|c|c|c} \begin{matrix} 63 \\ \vdots \\ 63 \end{matrix} & \mathbf{MC} & \begin{matrix} \\ \mathbf{I_{128}} \\ \mathbf{I_{128}} \\ \mathbf{I}^{(16)}_{\mathbf{e_7}} \end{matrix} & \mathbf{K_i} \end{array}\right), \text{ for } 1 < i < l$$

$$\mathbf{MC} = \begin{pmatrix}
\mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} \\
\mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} \\
\mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} \\
\mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_3} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w_2} & \mathbf{w_3} & \mathbf{0} & \mathbf{0} & \mathbf{0}
\end{pmatrix} \in F^{16\times 128}$$

The three vectors of dimension 8 can be derived from SubBytes, ShiftRows, and MixColumns steps, which are listed as follows.

$$\begin{aligned}
\mathbf{w_1} &= (\ \texttt{0x1f},\ \texttt{0x3e},\ \texttt{0x7c},\ \texttt{0xf8},\ \texttt{0xf1},\ \texttt{0xe3},\ \texttt{0xc7},\ \texttt{0x8f}) \\
\mathbf{w_2} &= (\ \texttt{0x3e},\ \texttt{0x7c},\ \texttt{0xf8},\ \texttt{0xeb},\ \texttt{0xf9},\ \texttt{0xdd},\ \texttt{0x95},\ \texttt{0x05}) \\
\mathbf{w_3} &= (\ \texttt{0x21},\ \texttt{0x42},\ \texttt{0x84},\ \texttt{0x13},\ \texttt{0x08},\ \texttt{0x3e},\ \texttt{0x52},\ \texttt{0x8a})
\end{aligned}$$

The differences between the last round and the previous rounds are the lack of the MixColumns step and an additional AddRoundKey in the end. We construct an $\mathbf{I}^{(16)}_{\mathbf{w_1}}$ matrix to describe the relations of the outputs (ciphertext) in $\mathbf{V_o}$ with the corresponding keys $\{k_{l,j}\}_{j=0}^{15}$ in the matrix $\mathbf{K_l}$.

$$\mathbf{A}_{\mathbf{T_l}||\mathbf{R_l}||\mathbf{K_l}} = \left(\begin{array}{c|c|c}
63 & \multirow{3}{*}{$\mathbf{MC}$} & \multirow{3}{*}{$\mathbf{I_{16}}$} \\
63 & & \\
63 & & \\
\vdots & \multirow{3}{*}{$\mathbf{I_{128}}$} & \\
 & & \\
 & & \\
\vdots & \multirow{3}{*}{$\mathbf{I_{128}}$} & \\
 & & \\
 & & \\
\vdots & \multirow{3}{*}{$\mathbf{I^{(16)}_{e_7}}$} & \\
 & & \\
 & & \\
63 & \multirow{3}{*}{$\mathbf{I^{(16)}_{w_1}}$} & \multirow{3}{*}{$\mathbf{K_l}$} \\
\vdots & & \\
63 & &
\end{array}\right)$$

$\mathbf{B}_{\mathbf{T_l}||\mathbf{R_l}}$ is similar to $\mathbf{B}_{\mathbf{T_l}||\mathbf{R_l}}$ but with 16 vectors of (1,0,...,0) appended.

$$\mathbf{C}_{\mathbf{T_l}||\mathbf{V_o}||\mathbf{R_l}} = \left(\begin{array}{c|c|ccc} \begin{matrix}1\\ \vdots \\ 1\end{matrix} & & \mathbf{O}_{16\times 128} & \boxed{\mathbf{I}^{(16)}_{\mathbf{v_f}}} & \mathbf{O}_{16\times 128} \\ & \boxed{\mathbf{I_{16}}} & & & \end{array}\right)$$

5.2.2 Circuit for AES Key Schedule

This section presents the constraints for AES key schedule. First, let $i' = i \bmod 2$:

$$\begin{aligned} R_{i,j} &= (\mathsf{SubWord}(\mathsf{RotWord}(W_{i-1})) \oplus rcon_{r/N})_j \\ &= \begin{cases} \mathsf{Sbox}(k_{i-1,12+(j+1 \bmod 4)}) \oplus rcon_{i,j} & \text{for AES-128;} \\ \mathsf{Sbox}(k_{\frac{3i-2+i'}{2},12-8i'+(j+1 \bmod 4)}) \oplus rcon_{i,j} & \text{for AES-192;} \\ \mathsf{Sbox}(k_{i,12+(j+i' \bmod 4)}) \oplus (rcon_{i,j} \wedge i') & \text{for AES-256.} \end{cases} \end{aligned}$$

The $\mathbf{S_i}$ matrix is similar to $\mathbf{R_i}$ matrix except for the RotWord and adding $rcon$ functions. The RotWord function is described in $\mathbf{K'_i}$, and the $rcon$ addition is described in the last part of $\mathbf{A}_{\mathbf{T'_i}||\mathbf{K'_i}||\mathbf{S_i}}$ matrix. Note that the constraints consist of both adding the constant in Sbox as well as the round constant. For AES-256, there is no need to add round constants in the even rounds, so there is a note $rc_i/0$ in $\mathbf{A}_{\mathbf{T'_i}||\mathbf{K'_i}||\mathbf{S_i}}$ matrix. A slight difference is that we need to use one more constraint for each variable $R_{i,j}$ to reduce the non-zero entries in matrix $\mathbf{A}$ (since it can be represented by "$\bar{y}_i$" as in AES round), and this leads to 40, 32, and 52 more constraints for AES-128, 192, and 256, respectively. Note that $\mathbf{B}_{\mathbf{K'_i}}$ is zero matrix for $1 \leq i \leq l$.

$$\mathbf{A}_{\mathbf{T}'_\mathbf{i}||\mathbf{K}'_\mathbf{i}||\mathbf{S_i}} = \left(\begin{array}{c|c|cc} & \mathbf{K}'_\mathbf{i} & & \\ & & & \mathbf{I_{32}} \\ & & \mathbf{I_{32}} & \\ & & \mathbf{I}^{(4)}_{\mathbf{e_7}} & \\ rc_i/0+63 & & & \mathbf{I}^{(4)}_{\mathbf{w_1}} \\ 63 & & & \\ 63 & & & \\ 63 & & & \end{array}\right), \text{ for } 1 \leq i \leq l'$$

$$\mathbf{B}_{\mathbf{T}'_\mathbf{i}||\mathbf{S_i}} = \left(\begin{array}{c|cc} & & \mathbf{I}^{(4)}_{\mathbf{v_1}} \\ 1 & & \\ \vdots & & \mathbf{I_{32}} \\ 1 & & \\ 1 & & \\ \vdots & \mathbf{I_{32}} & \\ 1 & & \\ x^7 & & \\ \vdots & \mathbf{I}^{(4)}_{\mathbf{v_1}} & \mathbf{I}^{(4)}_{\mathbf{v_2}} \\ x^7 & & \\ 1 & & \\ \vdots & & \\ 1 & & \end{array}\right), \text{ for } 1 \leq i \leq l'$$

$$\mathbf{C}_{\mathbf{T}'_\mathbf{i}||\mathbf{K}'_\mathbf{i}||\mathbf{S}_\mathbf{i}} = \left(\begin{array}{c|c|cc} 1 & & \multirow{3}{*}{$\mathbf{I}^{(4)}_{\mathbf{v_f}}$} & \multirow{3}{*}{$\mathbf{O_{4\times 32}}$} \\ \vdots & & & \\ 1 & & & \\ & \multirow{3}{*}{$\mathbf{I_4}$} & & \\ & & & \\ & & & \end{array}\right), \text{ for } 1 \le i \le l'$$

Last, for repeating the AES in the security level 3 or 5, we construct the matrix as following. The output matrix $\mathbf{V_o}, \mathbf{V'_o}$ and input matrix $\mathbf{V_i}, \mathbf{V'_i}$ take the corresponding variables in $\mathbf{z}$ as AES outputs and inputs in two AES and the constraints for key schedule keep the same. The corresponding matrix $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ can be derived by the method we used for the single AES.

$$\mathbf{M} = \left(\begin{array}{c|c|c|c|c|c} \mathbf{T_1} & & \mathbf{V_i} & \mathbf{R_1} & & \mathbf{K_1} \\ \vdots & & & \ddots & & \vdots \\ \mathbf{T_l} & \mathbf{V_o} & & \mathbf{R_l} & & \mathbf{K_l} \\ \hline \mathbf{T_1} & & \mathbf{V'_i} & & \mathbf{R_1} & \mathbf{K_1} \\ \vdots & & & & \ddots & \vdots \\ \mathbf{T_l} & \mathbf{V'_o} & & & \mathbf{R_l} & \mathbf{K_l} \\ \hline \mathbf{T'_1} & & & & & \mathbf{K'_1} \quad \mathbf{S_1} \\ \vdots & & & & & \vdots \quad \ddots \\ \mathbf{T'_l} & & & & & \mathbf{K'_l} \quad \mathbf{S_{l'}} \end{array}\right)$$

6 Recommended Parameter Sets

In this section, we recommend parameters for Aurora at different security levels. Together with the hard relation $\mathcal{R}$ in section 3.3.1, this suffices to derive a secure signature scheme.

Particularly, this section recommends the sizes of $|\mathbb{F}|$ and $|L|$ (and the other parameters) in Aurora. As the security analysis only depends on the sizes but not particular choices of the field and set, this section would

only determine their sizes. In section 7, we recommend how to choose the particular finite field $\mathbb{F}$ and the set L in the implementation.

6.1 Parameter Sets for Different Security Levels for Aurora

In this section, we provide a list of suggested parameter sets for the L1, L3, and L5 security levels required by NIST. Table 3 summarizes security- and implementation-related parameters.

$\lvert\mathbb{F}\rvert$	the size of the native field $\mathbb{F}$
m	the number of constraints in the R1CS of the hard relation (see section 3.3.1 and section 5)
n	the number of public and private variables in the R1CS of the hard relation
t	an upper bound for $\max\{m, n+1\}$
$\lvert L\rvert$	the size of the set L in the Aurora protocol
ρ	the rate of the Reed-Solomon code
b	upper bound for the number of opening values provided from the prover to the verifier
λ_{i}	number of parallel repetitions of the Lincheck phase of the Aurora protocol
λ'_{i}	number of parallel repetitions of the Reduce to FRI phase of the Aurora protocol
η	logarithm of the degree of the polynomials $\{q_i\}_i$ in FRI (which means the degree of each q_i is 2^η)
ℓ	number of parallel repetitions of query phases correspond to single commit phase in the FRI
λ	number of the output bits of the hash function

Table 3: List of parameters related to both security and implementation

We provide three parameter sets for each of the L1, L3, and L5 security levels: aggressive (A), balanced (B), and conservative (C). In all the parameter sets, we suggest $\rho := \frac{1}{32}$, $\lambda_{\mathsf{i}} := 1$, $\lambda'_{\mathsf{i}} := 1$, and $\eta := 1$. Table 4 gives our suggested parameter sets for different security levels.

ρ	λ_{i}	λ'_{i}	η
$\frac{1}{32}$	1	1	1

security level	Parameter Set	$\lvert\mathbb{F}\rvert$	t	$\lvert L\rvert$	b	ℓ	λ
L1	Preon128A	2^{192}	2^{12}	2^{19}	1040	26	256
	Preon128B	2^{192}	2^{12}	2^{19}	2320	58	384
	Preon128C	2^{192}	2^{12}	2^{21}	16764	381	384
L3	Preon192A	2^{256}	2^{13}	2^{20}	1638	39	384
	Preon192B	2^{256}	2^{13}	2^{20}	3654	87	512
	Preon192C	2^{256}	2^{13}	2^{21}	24464	556	512
L5	Preon256A	2^{320}	2^{14}	2^{20}	2184	52	512
	Preon256B	2^{320}	2^{14}	2^{20}	4956	118	512
	Preon256C	2^{320}	2^{14}	2^{22}	33534	729	512

Table 4: Summary of different parameter sets, in all of which we use $\rho = \frac{1}{32}$, $\lambda_{\mathsf{i}} := 1$, $\lambda'_{\mathsf{i}} := 1$, and $\eta := 1$

Although we suggest three parameter sets for each of the L1, L3, and L5 security levels, ***we stress that we only recommend the aggressive and balanced parameter sets in answer to NIST's call for general-purpose signature schemes.*** The (overly) conservative parameter set leads to too large a signature size for general-

purpose applications; we believe it is only justifiable to use in those rare and extreme applications for which only paranoiacally low risk-taking is allowed.

6.2 Justification for the Parameter Sets

6.2.1 Knowledge Soundness of the Non-interactive Zero-knowledge Aurora Protocol

L1 Security, Conservative (Preon128C). In this parameter set, we assume Conjecture 4.7 is true, and the hidden constant in Theorem 4.9 is close to 1. We configure the first and second terms of formula (19) to be much smaller than $\frac{1}{2^{141}}$, and we use formula (20) as the third term of formula (19). Overall, we configure the formula (19) to be less than $\frac{1}{2^{141}}$.

Assuming Conjecture 4.7 is true and combining this parameter set with Theorem 4.6, we can see that the non-interactive zero-knowledge Aurora protocol has a knowledge soundness error smaller than $\frac{q}{2^{141}}$ in the random oracle model against q classical queries to the random oracle.

To further argue that our configuration is secure against quantum attackers that makes quantum queries to the random oracle in the quantum random oracle model, we use this configuration with Theorem 4.9 and Theorem 4.10 (with a possible 2^{13} factor of security lost here, which is why we configure the state restoration knowledge error to be smaller than $\frac{1}{2^{141}}$), assuming that the hidden constant in Theorem 4.9 is close to 1.

For the zero-knowledge aspect, combining our choice of hash output length along with the result in Appendix B shows that the security lost when applying BCS transform is less than $\frac{1}{2^{128}}$ in the classical setting.

L1 Security, Balanced (Preon128B). In this parameter set, we assume Conjecture 4.3 and Conjecture 4.12 are true, and the hidden constant in Theorem 4.9 is close to 1.

Assuming Conjecture 4.3 is true, we can use Theorem 4.5. Applying this parameter set, using $u = 4$ to Theorem 4.5, and combining with formula (19), one gets that the knowledge soundness error of the interactive Aurora protocol is smaller than $\frac{1}{2^{128}}$.

To argue for quantum security, unfortunately when we use Theorem 4.5, we can not afford a security lost factor like 2^{13} as in the conservative setting, so we have to make a stronger assumption that Conjecture 4.12 is true, with this assumption and the assumption that the hidden constant in Theorem 4.9 is close to 1, the quantum security comes directly from Theorem 4.9.

For the zero-knowledge aspect, the security argument is the same as the conservative setting.

L1 Security, Aggressive (Preon128A). In this parameter set, we assume that Conjecture 4.3 and Conjecture 4.12 are true, the hidden constant in Theorem 4.9 is close to 1, and moreover, we assume the following Conjecture 6.1 is true:

Conjecture 6.1 *When invoked on the oracle $f_0 : L_0 \to \mathbb{F}$ with localization parameter η, rate parameter ρ, query phase repetition parameter ℓ, the soundness error of the FRI protocol is upper bounded by:*

$$\epsilon_{FRI} \leq \frac{|L_0|}{|\mathbb{F}|} + (1-\delta)^{\ell} \tag{35}$$

In other words, we conjecture that the attack presented in section 4.6.2 is the best attack on the FRI protocol in Aurora.

The security of this parameter set comes directly from our assumption and the argument in the medium setting, where we replace ϵ_{FRI} by formula (35) with $\delta := 1 - \rho$.

Note that in the aggressive setting, since we set λ to be exactly twice the security parameter, the security guarantee from Theorem 4.6 and Appendix B will be slightly less than 128 bits but larger than 125 bits.

L3 and L5 Security. For the justification of L3 and L5 security parameter sets, we use the same argument as in the L1 security parameter sets. Let q be the number of queries to the (classic) random oracle. In L3 parameter sets, the knowledge soundness error of the non-interactive zero-knowledge Aurora protocol is configured to be less than $\frac{q}{2^{192}}$ in the conservative (Preon192C) and balanced (Preon192B) setting (using the same assumption as the L1 cases respectively, and using $u = 4$ to Theorem 4.5 in the balanced parameter set), and is slightly larger than $\frac{q}{2^{192}}$ but smaller than $\frac{q}{2^{189}}$ in the aggressive (Preon192A) setting (using the same assumption as the L1 case), where q is the number of classical queries to the random oracle. In L5 parameter sets, the knowledge soundness error of the non-interactive zero-knowledge Aurora protocol is configured to be less than $\frac{q}{2^{256}}$ in the conservative (Preon256C) and balanced (Preon256B) setting (using the same assumption as the L1 cases respectively, and using $u = 3$ to Theorem 4.5 in the balanced parameter set), and is slightly larger than $\frac{q}{2^{256}}$ but smaller than $\frac{q}{2^{253}}$ in the aggressive (Preon256A) setting (using the same assumption as the L1 case). For the knowledge soundness of non-interactive zero-knowledge Aurora protocol in the quantum random oracle model against q' queries to the quantum random oracle, we replace q by $\frac{q'}{2}$ in the above knowledge soundness formulas. Last but not least, we note that, unlike L1 and L3, for L5 we set the output length of the random oracle to $\lambda := 512$ in balanced and conservative settings, same as in the aggressive setting and leaving no further margin. We feel that this is justifiable because the 256-bit security requirement of SHA3-512 already provides a substantial margin such that a relatively small number of bits of security loss should not result in any calamities, rendering our proposal insecure in any practical sense.

6.3 UF-CMA Security of the Signature Scheme

In section 6.2.1, we have justified that the non-interactive zero-knowledge Aurora protocol is a NIZK proof of knowledge, so when we instantiate the signature scheme using the non-interactive zero-knowledge Aurora prover algorithm and verifier algorithm as Π.Prove and Π.Verify for hard relation $\mathcal{R}$ in section 3.3.1, the UF-CMA security of the signature scheme directly follows from Theorem 3.5 and the following conjecture, which we have mentioned in section 3.3.2:

Conjecture 6.2 *The non-interactive zero-knowledge Aurora protocol is true-simulation extractable.*

7 Pseudocode

7.1 Building Blocks

7.1.1 GetWitness and GetInstance Functions

The GetWitness and GetInstance functions are deterministic algorithms used to generate the private witness vector $\mathbf{w}$ for the Aurora prover and the public instance vector $\mathbf{v}$ for the Aurora verifier in a specific format. The format follows from the specific form of the R1CS system of the AES described in the section 5, and the R1CS system for the hard relation described in section 3.3.1. We refer to our C reference implementation for more details on the format.

Algorithm 1 GetWitness(κ, sk)

Input: κ: security parameter, sk: secret key
Output: $\mathbf{w}$: witness vector
1: $\mathbf{w} \leftarrow sk$
2: **return** $\mathbf{w}$

Algorithm 2 $\mathsf{GetInstance}(\kappa, pk)$

Input: κ: security parameter, pk: public key
Output: $\mathbf{v}$: instance vector
1: $\mathbf{v} \leftarrow pk$
2: **return** $\mathbf{v}$

7.1.2 Instantiation of the Random Oracle

Each security level is associated with a parameter λ indicating the number of output bits of the random oracle. We will use two independent random oracles G_λ and H_λ in our signature scheme: G_λ is used to construct Merkle trees, while H_λ is used to generate non-interactive verifier's random coins. We model the random oracles G_λ and H_λ with an arbitrary-length input and a λ-bit output:

$$\mathsf{G}_\lambda : \{0,1\}^* \rightarrow \{0,1\}^\lambda$$
$$\mathsf{H}_\lambda : \{0,1\}^* \rightarrow \{0,1\}^\lambda$$

In practice, G_λ can be instantiated with any cryptographic hash function with proper security level. For H_λ, we use the duplex construction [BDPVA12] to instantiate the random oracle, which acts like a reseedable pseudo-random bit sequence generator. We refer to our C reference implementation for more details on the instantiation of G_λ and H_λ.

7.1.3 Merkle Tree

We use the Merkle tree construction described in [BCS16] to commit and open the prover's message oracle (message vector).

$\mathsf{Merkle.GetRoot}(\mathsf{G}_\lambda, \mathbf{m})$: this is a randomized algorithm that takes a random oracle G_λ and a vector $\mathbf{m} = (\mathbf{m}_1, ..., \mathbf{m}_{|\mathbf{m}|})$ as input, uses $\mathbf{m}$ to construct a Merkle tree using random oracle G_λ, and then outputs the Merkle root $rt \in \{0,1\}^\lambda$ of $\mathbf{m}$ as shown in figure 3, along with the random strings used to generate the tree (denoted as $rand$).

Algorithm 3 $\mathsf{Merkle.GetRoot}(\mathsf{G}_\lambda, \mathbf{m})$

Input: G_λ: a random oracle, $\mathbf{m}$: a vector with length $|\mathbf{m}|$
Output: rt: a λ bits string
1: **for** $i = 1$ to $|\mathbf{m}|$ **do**
2: $\quad r_i \xleftarrow{\$} \{0,1\}^{2\lambda}$
3: **end for**
4: $rand := (r_1, ..., r_{|\mathsf{m}|})$
5: $rt \leftarrow$ Merkle root of of the tree in figure 3
6: **return** $(rt, rand)$

Remark 7.1 *As [BCS16] stated, in order to construct a hiding commitment scheme, we first accompany each element $\mathbf{m}_i$ with a random value $r_i \in \{0,1\}^{2\lambda}$ and then perform the two-to-one hash process to construct the Merkle tree, as shown in figure 3.*

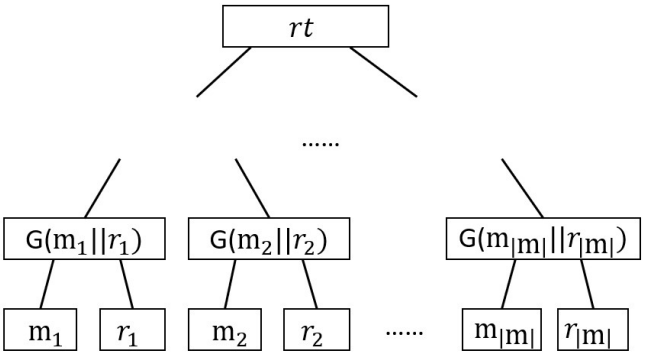


Figure 3: Structure of the Merkle tree for vector $(\mathbf{m}_1, ..., \mathbf{m}_{|\mathsf{m}|})$ using random oracle G

$\mathsf{Merkle.BatchOpen}(\mathsf{G}_\lambda, \mathbf{m}, \mathsf{p}_1, ..., \mathsf{p}_t)$: this is a deterministic algorithm which takes a random oracle G_λ, a vector $\mathbf{m}$, a vector $rand = (r_1, ..., r_{|\mathbf{m}|})$ of randomness strings, and t position indices $\mathsf{p}_1, \mathsf{p}_2, ..., \mathsf{p}_t$ as input, and then outputs the authentication set $Auth$, which contains the number t, the opening values $\mathbf{m}_{\mathsf{p}_1}, ..., \mathbf{m}_{\mathsf{p}_t}$, randomness strings $r_{\mathsf{p}_1}, ..., r_{\mathsf{p}_t}$, and several λ-bit strings that contain the necessary information to authenticate the opening values. We refer to our C reference implementation for the specific format of $Auth$.

Algorithm 4 $\mathsf{Merkle.BatchOpen}(\mathsf{G}_\lambda, \mathbf{m}, rand, \mathsf{p}_1, ..., \mathsf{p}_t)$

Input: G_λ: a random oracle, $\mathbf{m}$: a vector with length $|\mathbf{m}|$, $rand$: strings for the randomness of the Merkle tree, $\mathsf{p}_1, ..., \mathsf{p}_t$: the positions indices which are wanted to be opened
Output: $Auth$: an authentication set for the opening values $\mathbf{m}_{\mathsf{p}_1}, ..., \mathbf{m}_{\mathsf{p}_t}$
1: $Auth \leftarrow$ authentication set of $\mathbf{m}_{\mathsf{p}_1}, ..., \mathbf{m}_{\mathsf{p}_t}$ from the Merkle tree in figure 3
2: **return** $Auth$

Merkle.BatchVerify: this is a deterministic algorithm that takes a random oracle G_λ, a Merkle tree root $rt \in \{0,1\}^\lambda$ corresponding to some vector $\mathbf{m}$, and an authentication set $Auth$ as input, and then outputs 0 or 1 to indicate whether the $Auth$ truly gives a valid batch opening for values $\mathbf{m}_{\mathsf{p}_1}, ..., \mathbf{m}_{\mathsf{p}_t}$. The verification process basically keeps performing two-to-one hashes for the values and strings in $Auth$, and then checks if the final outcome equals rt. We refer to our C reference implementation for the more detailed hashing procedure.

Algorithm 5 $\mathsf{Merkle.BatchVerify}(\mathsf{G}_\lambda, rt, Auth)$

Input: G_λ: a random oracle, rt: a λ bits string, $Auth$: an authentication set
Output: 0 or 1
1: $rt' \leftarrow$ final output from the two-to-one hashing process of $Auth$ using G_λ
2: **if** $rt' \neq rt$ **then**
3: **return** 0
4: **else**
5: **return** 1
6: **end if**

7.2 Pseudocode for the Signature Scheme

In this section, we provide the pseudocode for our signature scheme using building blocks outlined in section 7.1. We follow Construction 3.4 for the hard relation in section 3.3.1 to construct the signature scheme.

7.2.1 KeyGen Algorithm

Algorithm 6 $\mathsf{KeyGen}(1^\kappa)$

Input: κ: security parameter
Output: $(\mathsf{sk}, \mathsf{pk})$: (secret key, public key) pair

if $\kappa = 128$ **then** ▷ L1 security level
  $r \xleftarrow{\$} \{0,1\}^{128}, \mathsf{sk} \xleftarrow{\$} \{0,1\}^{128}$
  $y \leftarrow \mathsf{AES}_{\mathsf{sk}}(r)$
  $\mathsf{pk} := r \| y$
else if $\kappa = 192$ **then** ▷ L3 security level
  $r \xleftarrow{\$} \{0,1\}^{192}, \mathsf{sk} \xleftarrow{\$} \{0,1\}^{192}$
  Parse $r = \underbrace{r_1}_{\text{128 bits}} \| \underbrace{r'}_{\text{64 bits}}$
  $r_2 := r' \| 0^{64}$
  $y_1 \leftarrow \mathsf{AES}_{\mathsf{sk}}(r_1)$
  $y_2 \leftarrow \mathsf{AES}_{\mathsf{sk}}(r_2)$
  $\mathsf{pk} := r \| y_1 \| y_2$
else if $\kappa = 256$ **then** ▷ L5 security level
  $r_1 \xleftarrow{\$} \{0,1\}^{128}, r_2 \xleftarrow{\$} \{0,1\}^{128}, \mathsf{sk} \xleftarrow{\$} \{0,1\}^{256}$
  $y_1 \leftarrow \mathsf{AES}_{\mathsf{sk}}(r_1)$
  $y_2 \leftarrow \mathsf{AES}_{\mathsf{sk}}(r_2)$
  $\mathsf{pk} := r_1 \| r_2 \| y_1 \| y_2$
end if
return $(\mathsf{sk}, \mathsf{pk})$

7.2.2 Sign and Verify Algorithms

Public Parameters. We first list the public parameters that can be predetermined once the security parameter is determined. These parameters will be treated as global variables in the pseudocode of Sign and Verify, which correspond to Pub in Construction 3.4.

- $\mathbb{F}$: this is the native field of the proving system. In order to embed AES constraint into R1CS over the field $\mathbb{F}$, as suggested in section 5, we will consider the field $GF_{64} := \mathbb{F}_2[x]/(x^{64} + x^4 + x^3 + x + 1)$. In L1 setting, we set $\mathbb{F}$ to be a degree 3 extension of GF_{64}; in L3 setting, we set $\mathbb{F}$ to be a degree 4 extension of GF_{64}; in L5 setting, we set $\mathbb{F}$ to be a degree 5 extension of GF_{64}. We view the elements in GF_{64} as polynomials over $\mathbb{F}_2$ and denote them using the coefficients as 64-bit strings in the descending power order. We view the elements in $\mathbb{F}$ as polynomials over GF_{64} and denote the elements in $\mathbb{F}$ by concatenation of elements in GF_{64} again in the descending power order. We will also write the strings as integers in decimal for convenience (for example, we write $000...011$ as $(3)_2$). We refer to the C reference implementation for more details on the arithmetic of $\mathbb{F}$.
- $\mathbf{A}, \mathbf{B}, \mathbf{C}$: these three matrices in $\mathbb{F}^{m \cdot (n+1)}$ corresponds to the R1CS circuit of the hard relation we are using in different security level; c.f. section 3.3.1. The R1CS circuit for AES is constructed via the

method in section 5. We refer to the C reference implementation for the specific format and sizes we are using for these matrices.

- m, n, k: The integers m denotes the number of rows in $\mathbf{A}, \mathbf{B}, \mathbf{C}$, whereas the integer $n+1$ denotes the number of columns in $\mathbf{A}, \mathbf{B}, \mathbf{C}$. m is also the number of R1CS constraints, and n, the number of R1CS variables. The integer k indicates the number of public R1CS variables.
- b: this integer is the query upper bound and also represents the degree that needs to be added to the committed polynomials in order to achieve zero-knowledge property; c.f. section 4.4: Adding random polynomials to mask original polynomials.
- H_1: H_1 is defined to be the additive subspace $\{(0)_2, (1)_2, ..., (2^m - 1)_2\} \subset \mathbb{F}$, and we also denote the elements of H_1 in this order by $H_1 = \{h_1, h_2, ..., h_m\}$.
- H_2: H_2 is defined to be the additive subspace $\{(0)_2, (1)_2, ..., (2^{n+1} - 1)_2\} \subset \mathbb{F}$, and we also denote the elements of H_2 in this order by $H_2 = \{h_1, h_2, ..., h_{n+1}\}$.
- $Z_{H_1}(x)$, $Z_{H_2}^{\leq(k+1)}(x)$ and $Z_{H_1 \cup H_2}(x)$: these three polynomials are defined by $Z_{H_1}(x) := \Pi_{j \in \{1,...,m\}}(x - h_j)$, $Z_{H_2}^{\leq(k+1)}(x) := \Pi_{j \in \{1,...,k+1\}}(x - h_j)$, and $Z_{H_1 \cup H_2}(x) := \Pi_{y \in H_1 \cup H_2}(x - y)$.
- L: L is defined to be the additive affine subspace $\{(0)_2, (1)_2, ..., (|L| - 1)_2\} + (|L|)_2$ of $\mathbb{F}$, i.e., L is the additive subspace $\{(0)_2, (1)_2, ..., (|L| - 1)_2\}$ shifted by $(|L|)_2$. The suggested size of L is defined in Table 4 for different security parameter sets. We denote the elements of L with the aforementioned order by $L = \{l_1, l_2, ..., l_{|L|}\}$ in the pseudocode.
- Configuration for the FRI protocol (L_i, $q_i(x)$): We define the element $s_0 := (|L|)_2 \in \mathbb{F}$, the set $L_0 := L \subset \mathbb{F}$, the polynomial $q_0(x) := (x - (0)_2)(x - (1)_2) + (s_0 - (0)_2)(s_0 - (1)_2) \in \mathbb{F}[x]$, and then we set $s_1 := (s_0 - (0)_2)(s_0 - (1)_2)$, $L_1 := \{(0)_2, ..., (\frac{|L_0|}{2} - 1)_2\} + s_1$, $q_1(x) := (x - (0)_2)(x - (1)_2) + (s_1 - (0)_2)(s_1 - (1)_2)$. We continue this process recursively, defining $s_i := (s_{i-1} - (0)_2)(s_{i-1} - (1)_2)$, $L_i := \{(0)_2, ..., (\frac{|L_{i-1}|}{2} - 1)_2\} + s_i$, $q_i(x) := (x - (0)_2)(x - (1)_2) + (s_i - (0)_2)(s_i - (1)_2)$. We denote the elements of L_i with the aforementioned order by $L_i = \{l_{i,1}, l_{i,2}, ..., l_{i,|L_i|}\}$ in the pseudocode.
- ℓ: the integer ℓ is the number of repetitions of query phases corresponding to a single commit phase in the FRI. The suggested value of ℓ is provided in Table 4 for different security parameter sets.
- λ: the integer λ is the output length of the random oracle and is defined in Table 4 for different security parameter sets.

Remark 7.2 *We implicitly use the repetition parameter $\lambda_{\mathsf{i}} = 1$ for Lincheck phase and $\lambda'_{\mathsf{i}} = 1$ for Reduce to FRI phase of the Aurora protocol in the pseudocode, since we suggest these values in all of the parameter sets. In the configuration of FRI, we implicitly consider $\eta = 1$ so that all the polynomials $\{q_i\}_i$ have a degree equal to $2^\eta = 2$.*

Special Symbols Used in Sign **and** Verify**.**

- We use the symbol $f \leftarrow$ eq.(b) for a polynomial f, an integer b to indicate that the polynomial f is computed from formula eq.(b).
- We use the symbol $f \underset{a}{\leftarrow}$ eq.(b) for a polynomial f, integers a and b to indicate that the polynomial f is derived by using Lagrange interpolation on the constraint system eq.(b) in this document to obtain the unique polynomial with a degree less than a. In this case, the number of constraints of the system eq.(b) will be equal to a. If the constraint system is simple, we will write the system directly on the right side of the left arrow.

- We use the symbol $f \xleftarrow[a]{\$}$ eq.(b) for a polynomial f, integers a and b to indicate that the polynomial f is obtained by uniformly choosing a polynomial that satisfies the constraint system eq.(b) in this document and with degree less than a. In this case, the number of constraints of the system eq.(b) will be smaller than a.
- We use the symbol $f \xleftarrow[a]{\$} Poly$ for a polynomial f, integers a and b to indicate that the polynomial f is obtained by uniformly choosing a polynomial with a degree less than a.
- We use the symbol $f \xleftarrow[a]{\$} Poly_{\sum_S=0}$ for a polynomial f, integers a and b and a set S to indicate that the polynomial f is obtained by uniformly choosing a polynomial that satisfies $\sum_{x\in S} f(x) = 0$ and with degree less than a.
- We use the symbol $st_1 \xleftarrow[a]{trunc} st_2$ for a bit string st_1, an integer a, and a bit string st_2 to indicate that st_1 is obtained by truncating st_2 to a bit string with a bits. In this case, we assume that the length of st_2 is longer than a. We refer to the C reference implementation for the specific truncation method.

Pseudocode for Sign **and** Verify**.**

Algorithm 7 $\mathsf{Sign}(\kappa, \mathsf{sk}, msg)$

Input: κ: security parameter, sk: secret key, msg: message
Output: sig: a signature that corresponds to $(\mathsf{sk}, \mathsf{pk})$ and msg

$\sigma_0 \leftarrow \mathsf{H}_\lambda(\textbf{Public parameters} \| msg)$
$\mathbf{w} \leftarrow \mathsf{GetWitness}(\kappa, \mathsf{sk})$
$\mathbf{v} \leftarrow \mathsf{GetInstance}(\kappa, \mathsf{pk})$
$f_{(1,\mathbf{v})} \xleftarrow[k+1]{}$ eq.(1) ▷ Start polynomial interpolation
$f_{\mathbf{w}} \xleftarrow[n-k+\mathsf{b}]{\$}$ eq.(2) ▷ Sample random polynomials
$f_{\mathbf{Az}} \xleftarrow[m+\mathsf{b}]{\$}$ eq.(3), $f_{\mathbf{Bz}} \xleftarrow[m+\mathsf{b}]{\$}$ eq.(4), $f_{\mathbf{Cz}} \xleftarrow[m+\mathsf{b}]{\$}$ eq.(5)
$r \xleftarrow[2\max\{m,n+1\}+\mathsf{b}-1]{\$} Poly_{\sum_{H_1\cup H_2}=0}$
$r_{LDT} \xleftarrow[2\max\{m,n+1\}+2\mathsf{b}]{\$} Poly$
$\hat{f}_{\mathbf{w}} \leftarrow f_{\mathbf{w}}|_L := (f_{\mathbf{w}}(l_1), f_{\mathbf{w}}(l_2), ..., f_{\mathbf{w}}(l_{|L|}))$
$\hat{f}_{\mathbf{Az}} \leftarrow f_{\mathbf{Az}}|_L$, $\hat{f}_{\mathbf{Bz}} \leftarrow f_{\mathbf{Bz}}|_L$, $\hat{f}_{\mathbf{Bz}} \leftarrow f_{\mathbf{Bz}}|_L$
$\hat{r} \leftarrow r|_L$
$\hat{r}_{LDT} \leftarrow r_{LDT}|_L$
$\tilde{f} \leftarrow (\tilde{f}(l_1), ..., \tilde{f}(l_{\frac{|L|}{2}}))$
where $\tilde{f}(l_i) := f_{\mathbf{w}}(l_{2i}) \parallel f_{\mathbf{w}}(l_{2i+1}) \parallel f_{\mathbf{Az}}(l_{2i}) \parallel f_{\mathbf{Az}}(l_{2i+1}) \parallel f_{\mathbf{Bz}}(l_{2i}) \parallel f_{\mathbf{Bz}}(l_{2i+1}) \parallel f_{\mathbf{Cz}}(l_{2i}) \parallel f_{\mathbf{Cz}}(l_{2i+1}) \parallel f_r(l_{2i}) \parallel f_r(l_{2i+1}) \parallel f_{r_{LDT}}(l_{2i}) \parallel f_{r_{LDT}}(l_{2i+1})$
$(rt_1, rand_1) \leftarrow \mathsf{Merkle.GetRoot}(\mathsf{G}_\lambda, \tilde{f})$
$\sigma_1 \leftarrow \mathsf{H}_\lambda(\sigma_0 \| rt_1)$ ▷ Start Lincheck
$\alpha \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 1)$
$s_1 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 2)$

$s_2 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 3)$

$s_3 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 4)$

$p_\alpha^1 \leftarrow$ eq.(8)

$p_\alpha^{2,\mathbf{A}} \leftarrow$eq.(9), $p_\alpha^{2,\mathbf{B}} \leftarrow$eq.(9), $p_\alpha^{2,\mathbf{C}} \leftarrow$eq.(9) ▷ Replace $\mathbf{A}$ with $\mathbf{B}$, $\mathbf{C}$ in eq.(9) respectively

$f_{\mathbf{z}} \leftarrow$eq.(7)

$g, h \leftarrow$eq.(16) ▷ $\mu = 0$ and ignore i in eq.(16)

$\tilde{h} \leftarrow (\tilde{h}(l_1), ..., \tilde{h}(l_{\frac{|L|}{2}}))$ where $\tilde{h}(l_i) := h(l_{2i}) \parallel h(l_{2i+1})$

$(rt_2, rand_2) \leftarrow \mathsf{Merkle.GetRoot}(\mathsf{G}_\lambda, \tilde{h})$

$\sigma_2 \leftarrow \mathsf{H}_\lambda(\alpha \| s_1 \| s_2 \| s_3 \| rt_2)$ ▷ Start Reduce to FRI

for $i = 1$ to 9 **do**

  $\mathbf{y}_i \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_2 \| 2 \| i)$

end for

$f_0 \leftarrow$ eq.(18) ▷ $\lambda_\mathsf{i} = 1$ and ignore j in eq.(18)

$\mathsf{r} := \lfloor \log(2 \max\{m, n+1\} + 2\mathsf{b} - 1) \rfloor$ ▷ Commit phase of FRI

$x_0 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_2 \| 3 \| 1)$

for $i = 1$ to $\mathsf{r} - 1$ **do**

  $\tilde{f}_i \leftarrow (\tilde{f}_i(l_{i,1}), ..., \tilde{f}_i(l_{i, \frac{|L_i|}{2}}))$

  where $\tilde{f}_i(l_{i,j}) := f_i(l_{i,2j}) \parallel f_i(l_{i,2j+1})$

  $(rt_{i+2}, rand_{i+2}) \leftarrow \mathsf{Merkle.GetRoot}\ (\mathsf{G}_\lambda, \tilde{f}_i)$

  $\sigma_{i+2} \leftarrow \mathsf{H}_\lambda(x_{i-1} \| rt_{i+2})$

  $x_i \leftarrow \mathsf{H}_\lambda(\sigma_{i+2} \| i + 3 \| 1)$

  $f_{i+1} \leftarrow$ eq.(13)

end for

$c_1, c_2 \leftarrow$ linear term and constant term of f_r ▷ f_r should be a linear polynomial

$\sigma_{\mathsf{r}+2} \leftarrow \mathsf{H}_\lambda(x_{\mathsf{r}-1} \| c_1 \| c_2)$

for $i = 1$ to ℓ **do** ▷ ℓ independent FRI query phase

  $y_{i,0} \xleftarrow[\log|L_0|-1]{trunc} \mathsf{H}_\lambda(\sigma_{\mathsf{r}+2} \| \mathsf{r} + 3 \| i)$ ▷ $y_{i,0} \in \{1, 2, ..., \frac{|L_0|}{2}\}$ is the query position index

  for $j = 1$ to $\mathsf{r} - 1$ **do**

    $y_{i,j} := \lceil \frac{y_{i,j-1}}{2} \rceil$

  end for

end for

$Auth_{rt_1} \leftarrow \mathsf{Merkle.BatchOpen}(\mathsf{G}_\lambda, \tilde{f}, rand_1, y_{1,0}, y_{2,0}, ..., y_{\ell,0})$

$Auth_{rt_2} \leftarrow \mathsf{Merkle.BatchOpen}(\mathsf{G}_\lambda, \tilde{h}, rand_2, y_{1,0}, y_{2,0}, ..., y_{\ell,0})$

for $i = 1$ to $\mathsf{r} - 1$ **do**

  $Auth_{rt_{i+2}} \leftarrow \mathsf{Merkle.BatchOpen}(\mathsf{G}_\lambda, \tilde{f}_i, rand_{i+2}, y_{1,i}, y_{2,i}, ..., y_{\ell,i})$

end for

$sig \leftarrow (rt_1, rt_2, ..., rt_{\mathsf{r}+1}, c_1, c_2, Auth_{rt_1}, ..., Auth_{rt_{\mathsf{r}+1}})$

Algorithm 8 $\mathsf{Verify}(\kappa, \mathsf{pk}, msg, sig)$

Input: κ: security parameter, pk: public key, msg: message, sig: signature

Output: 0 or 1

Parse $sig = (rt_1, rt_2, ..., rt_{\mathsf{r}+1}, c_1, c_2, Auth_{rt_1}, ..., Auth_{rt_{\mathsf{r}+1}})$

$\mathbf{v} \leftarrow \mathsf{GetInstance}(\kappa, \mathsf{pk})$

3: $f_{(1,\mathbf{v})} \xleftarrow[k+1]{}$ eq.(1)
4: $\sigma_0 \leftarrow \mathsf{H}_\lambda(\textbf{Public parameters} \| msg)$ ▷ Start Lincheck
5: $\sigma_1 \leftarrow \mathsf{H}_\lambda(\sigma_0 \| rt_1)$
6: $\alpha \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 1)$ ▷ Use the same truncation method as Sign
7: $s_1 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 2)$
8: $s_2 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 3)$
9: $s_3 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_1 \| 1 \| 4)$
10: $p_\alpha^1 \leftarrow$ eq.(8)
11: $p_\alpha^{2,\mathbf{A}} \leftarrow$eq.(9), $p_\alpha^{2,\mathbf{B}} \leftarrow$eq.(9), $p_\alpha^{2,\mathbf{C}} \leftarrow$eq.(9) ▷ Replace $\mathbf{A}$ with $\mathbf{B}$, $\mathbf{C}$ in eq.(9) respectively
12: $\sigma_2 \leftarrow \mathsf{H}_\lambda(\alpha \| s_1 \| s_2 \| s_3 \| rt_2)$
13: **for** $i = 1$ to 9 **do**
14: $\quad \mathbf{y}_i \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_2 \| 2 \| i)$
15: **end for**
16: $\mathsf{r} := \lfloor \log(2\max\{m, n+1\} + 2\mathsf{b} - 1) \rfloor$
17: $x_0 \xleftarrow[\log|\mathbb{F}|]{trunc} \mathsf{H}_\lambda(\sigma_2 \| 3 \| 1)$
18: **for** $i = 1$ to $\mathsf{r} - 1$ **do**
19: $\quad \sigma_{i+2} \leftarrow \mathsf{H}_\lambda(x_{i-1} \| rt_{i+2})$
20: $\quad x_i \leftarrow \mathsf{H}_\lambda(\sigma_{i+2} \| i+3 \| 1)$
21: **end for**
22: $\sigma_{\mathsf{r}+2} \leftarrow \mathsf{H}_\lambda(x_{\mathsf{r}-1} \| c_1 \| c_2)$
23: **for** $i = 1$ to ℓ **do** ▷ ℓ independent FRI query phase verification
24: $\quad y_{i,0} \xleftarrow[\log|L_0|-1]{trunc} \mathsf{H}_\lambda(\sigma_{\mathsf{r}+2} \| \mathsf{r}+3 \| i)$ ▷ $y_{i,0} \in \{1, 2, ..., \frac{|L_0|}{2}\}$ is the query position index
25: $\quad$ Parse $\tilde{f}(l_{y_{i,0}})$ from $Auth_{rt_1}$
26: $\quad$ Parse $\tilde{f}(l_{y_{i,0}}) = f_{\mathbf{w}}(l_{2y_{i,0}}) \parallel f_{\mathbf{w}}(l_{2y_{i,0}+1}) \parallel f_{\mathbf{Az}}(l_{2y_{i,0}}) \parallel f_{\mathbf{Az}}(l_{2y_{i,0}+1}) \parallel f_{\mathbf{Bz}}(l_{2y_{i,0}}) \parallel f_{\mathbf{Bz}}(l_{2y_{i,0}+1}) \parallel f_{\mathbf{Cz}}(l_{2y_{i,0}}) \parallel f_{\mathbf{Cz}}(l_{2y_{i,0}+1}) \parallel f_r(l_{2y_{i,0}}) \parallel f_r(l_{2y_{i,0}+1}) \parallel f_{r_{LDT}}(l_{2y_{i,0}}) \parallel f_{r_{LDT}}(l_{2y_{i,0}+1})$
27: $\quad$ Parse $\tilde{h}(l_{y_{i,0}})$ from $Auth_{rt_2}$
28: $\quad$ Parse $\tilde{h}(l_{y_{i,0}}) = h(l_{2y_{i,0}}) \parallel h(l_{2y_{i,0}+1})$
29: $\quad g(l_{2y_{i,0}}), g(l_{2y_{i,0}+1}) \leftarrow$ eq.(17) ▷ $\mu = 0$ and ignore i in eq.(17)
30: $\quad f_0(l_{0,2y_{i,0}}), f_0(l_{0,2y_{i,0}+1}) \leftarrow$ eq.(18) ▷ $\lambda_{\mathsf{i}} = 1$ and ignore j in eq.(18)
31: $\quad$ **for** $j = 1$ to $\mathsf{r} - 1$ **do**
32: $\quad\quad y_{i,j} := \lceil \frac{y_{i,j-1}}{2} \rceil$
33: $\quad\quad$ Parse $\tilde{f}_j(l_{j,y_{i,j}})$ from $Auth_{rt_{j+2}}$
34: $\quad\quad$ Parse $\tilde{f}_j(l_{j,y_{i,j}}) = f_j(l_{j,2y_{i,j}}) \parallel f_j(l_{j,2y_{i,j}+1})$
35: $\quad\quad F_{j-1} \xleftarrow[2]{} \begin{cases} F_{j-1}(l_{j-1,2y_{i,j-1}}) & = f_{j-1}(l_{j-1,2y_{i,j-1}}) \\ F_{j-1}(l_{j-1,2y_{i,j-1}+1}) & = f_{j-1}(l_{j-1,2y_{i,j-1}+1}) \end{cases}$
36: $\quad\quad$ **if** $F_{j-1}(x_{j-1}) \neq f_j(l_{j,2y_{i,j}})$ **then** ▷ Check the consistency of f_{j-1} and f_j
37: $\quad\quad\quad$ **return** 0
38: $\quad\quad$ **end if**
39: $\quad$ **end for**

40: $F_{\mathsf{r}-1} \xleftarrow[2]{} \begin{cases} F_{\mathsf{r}-1}(l_{\mathsf{r}-1,2y_{i,\mathsf{r}-1}}) & = f_{\mathsf{r}-1}(l_{\mathsf{r}-1,2y_{i,\mathsf{r}-1}}) \\ F_{\mathsf{r}-1}(l_{\mathsf{r}-1,2y_{i,\mathsf{r}-1}+1}) & = f_{\mathsf{r}-1}(l_{\mathsf{r}-1,2y_{i,\mathsf{r}-1}+1}) \end{cases}$

41: $y_{i,\mathsf{r}} := \lceil \frac{y_{i,\mathsf{r}-1}}{2} \rceil$

42: **if** $F_{\mathsf{r}-1}(x_{\mathsf{r}-1}) \neq c_1 \cdot l_{\mathsf{r},2y_{i,\mathsf{r}}} + c_2$ **then** ▷ Check the consistency of $f_{\mathsf{r}-1}$ and f_{r}

43: **return** 0

44: **end if**

45: **end for**

46: **for** $i = 1$ to $\mathsf{r}+1$ **do** ▷ Verify all openings of the i-th Merkle tree

47: **if** $\mathsf{Merkle.BatchVerify}(\mathsf{G}_\lambda, rt_i, Auth_{rt_i}) \neq 1$ **then**

48: **return** 0

49: **end if**

50: **end for**

51: **return** 1

7.2.3 Making the Algorithms Deterministic

The pseudocodes of KeyGen, Sign, and Merkle.GetRoot functions shown in this section are randomized algorithms. To make KeyGen deterministic, one can take a seed as its input and generate random numbers with a proper PRNG. For deterministic signing, the seed of the PRNG can be derived from the secret key sk and the message msg, and the generated random numbers can be used by the Merkle.GetRoot function as well.

8 Performance

Asymptotic Analysis. As shown in figures 1 and 2 of [BCR+19], for an arithmetic circuit of N gates, the asymptotic prover time is $O(N \log N)$ field operations (assuming the number of non-zero elements in the matrices of the circuit's corresponding R1CS is linear in N), the asymptotic verifier time is $O(N)$ field operations, and the final argument size is $O_\kappa(\log^2 N)$ for security parameter κ. The argument size scales poly-logarithmically in N since the FRI protocol has $O(\log N)$ rounds, and each Merkle tree has height $O(\log N)$ so the authentication set for the openings in each round also scales as $O(\log N)$.

Implementation Performance. The Preon reference implementation on a single core of AMD EPYC 73F3 3.5GHz 16-core achieves the following performance. Though we only have very few data points, the results seem to fit the above asymptotics. Last but not least, the numbers here are not at all suggestive of Preon's performance in practice, as the reference implementation is meant for expository purposes only. We are sure that there is plenty of room for further optimization, which we expect to see in the few months to come.

Security Level	Parameter Set	Size [bytes] sk	pk	signature	timing [ms] KeyGen	Sign	Verify
L1	Preon128A	16	32	139k	0.002	76,067	397
	Preon128B	16	32	372k	0.002	80,870	561
	Preon128C	16	32	1,725k	0.002	76,919	2,303
L3	Preon192A	24	56	312k	0.002	132,411	2,182
	Preon192B	24	56	778k	0.002	137,306	2,688
	Preon192C	24	56	3,541k	0.002	137,252	7,460
L5	Preon256A	32	64	598k	0.002	414,713	11,552
	Preon256B	32	64	1,157k	0.002	415,207	12,487
	Preon256C	32	64	5,248k	0.002	417,464	26,835

Table 5: Performance of different parameter sets

References

[BBHR17] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. *Electron. Colloquium Comput. Complex.*, TR17-134, 2017.

[BCI+20] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for reed-solomon codes. In *61st Annual Symposium on Foundations of Computer Science*, pages 900–909, Durham, NC, USA, November 16–19, 2020. IEEE Computer Society Press.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 103–128, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 31–60, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany.

[BdK+21] Carsten Baum, Cyprien de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 12710 of *Lecture Notes in Computer Science*, pages 266–297, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

[BDPVA12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, pages 320–337, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014: 17th International Conference on Theory*

and Practice of Public Key Cryptography, volume 8383 of *Lecture Notes in Computer Science*, pages 501–519, Buenos Aires, Argentina, March 26–28, 2014. Springer, Heidelberg, Germany.

[BGKS19] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: sampling outside the box improves soundness. *CoRR*, abs/1903.12243, 2019.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1825–1842, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[CMS19] Alessandro Chiesa, Peter Manohar, and Nicholas Spooner. Succinct arguments in the quantum random oracle model. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part II*, volume 11892 of *Lecture Notes in Computer Science*, pages 1–29, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany.

[CvH91] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 257–265, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[DBN+01] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

[DHLW10] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana López-Alt, and Daniel Wichs. Efficient public-key cryptography in the presence of key leakage. In Masayuki Abe, editor, *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 613–631, Singapore, December 5–9, 2010. Springer, Heidelberg, Germany.

[DKR+22] Christoph Dobraunig, Daniel Kales, Christian Rechberger, Markus Schofnegger, and Greg Zaverucha. Shorter signatures based on tailor-made minimalist symmetric-key crypto. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 843–857, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[dSGMOS19] Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P. Smart. BBQ: using AES in picnic signatures. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 669–692. Springer, 2019.

[FKMV12] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the non-malleability of the Fiat-Shamir transform. In Steven D. Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012: 13th International Conference in Cryptology in India*, volume 7668 of *Lecture Notes in Computer Science*, pages 60–79, Kolkata, India, December 9–12, 2012. Springer, Heidelberg, Germany.

[Hol19] Justin Holmgren. On round-by-round soundness and state restoration attacks. Cryptology ePrint Archive, Report 2019/1261, 2019. `https://eprint.iacr.org/2019/1261`.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[MPR11] Hemanta K. Maji, Manoj Prabhakaran, and Mike Rosulek. Attribute-based signatures. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 376–392, San Francisco, CA, USA, February 14–18, 2011. Springer, Heidelberg, Germany.

[MRV+21] Silvio Micali, Leonid Reyzin, Georgios Vlachos, Riad S. Wahby, and Nickolai Zeldovich. Compact certificates of collective knowledge. In *2021 IEEE Symposium on Security and Privacy*, pages 626–641, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[STA] STARKWARE. Ziggy: Post-quantum-secure signature scheme based on a ZK-STARK. `https://github.com/starkware-libs/ethSTARK/tree/ziggy#11-ziggy`.

A Components of Aurora

Here we describe each of the components of Aurora and explain what relation each component can proof, for more general and complete description and analysis of these components, please refer to the original paper [BCR+19].

We mention that a core protocol which is used in Aurora is the fast Reed-Solomon interactive oracle proof of proximity (FRI) low degree test protocol. Which uses properties of the Reed-Solomon code, thus for most of its component, the input vectors are encoded as a polynomial and then be transformed into Reed-Solomon codes over some domain. Through this way Aurora can leverage the benefit of asymptotically $O(\log^2 N)$ argument size from FRI, where N is the number of gates of the arithmetic circuit. We will see how the vectors are encoded in the following sections.

In the following, we assume that all the arithmetic operations are over a native binary field $\mathbb{F}$, which contains two subspaces H_1, H_2 where $|H_1| = a, |H_2| = b$ and an additive coset L with suitable size in order to encode the vector polynomials into Reed-Solomon codes, if $a \geq b$ we will set $H_1 \supset H_2$ and if $a < b$ we will set $H_1 \subset H_2$.

A.1 Lincheck

In the Lincheck protocol, the prover's input is a matrix $\mathbf{A}$, the vector $\mathbf{z}$ and the vector $\mathbf{a}$, and the verifier's input is the matrix $\mathbf{A}$.

The goal of Lincheck is to let the prover shows the statement "I know secret vectors $\mathbf{a}, \mathbf{z}$ such that $\mathbf{a} = \mathbf{A}\mathbf{z}$", the idea is to let the verifier sample a field element α, and check if the inner product $< \mathbf{a} -$

$\mathbf{Az}, (1, \alpha, ..., \alpha^{a-1}) >$ equals to 0. The inner product can be rewrite as

$$(\sum_{i=1}^{a} \mathbf{a}_i \cdot \alpha^{i-1}) - [\sum_{i=1}^{a} (\sum_{j=1}^{b} \mathbf{A}_{i,j} \cdot \mathbf{z}_j) \cdot \alpha^{i-1}] \tag{36}$$

$$= (\sum_{i=1}^{a} \mathbf{a}_i \cdot \alpha^{i-1}) - [\sum_{j=1}^{b} \mathbf{z}_j \cdot (\sum_{i=1}^{a} \mathbf{A}_{i,j} \cdot \alpha^{i-1})] \tag{37}$$

With the above observation in mind, we first order all the elements in H_1 and H_2. Write $H_1 = \{h_1, ..., h_a\}$, and $H_2 = \{h_1, ..., h_b\}$ (remember that either $H_1 \supset H_2$ or $H_1 \subset H_2$). The prover uses Lagrange interpolation to compute two polynomials $f_{\mathbf{a}}, f_{\mathbf{z}}$, where $deg(f_{\mathbf{a}}) < |H_1|, deg(f_{\mathbf{z}}) < |H_2|$, such that

$$f_{\mathbf{a}}(x) = \{ \quad \mathbf{a}_i \quad \text{for} \quad x = h_i \text{ where } i \in \{1, 2, ..., a\} \tag{38}$$

$$f_{\mathbf{z}}(x) = \{ \quad \mathbf{z}_j \quad \text{for} \quad x = h_j \text{ where } j \in \{1, 2, ..., b\} \tag{39}$$

, and then after receives verifiers random element α, the prover uses Lagrange interpolation to compute two polynomials $p_\alpha^1, p_\alpha^{2,\mathbf{A}}$ such that

$$p_\alpha^1(x) = \begin{cases} \alpha^{i-1} & \text{for} \quad x = h_i \text{ where } i \in \{1, 2, ..., a\} \\ 0 & \text{for} \quad x \in \{H_1 \cup H_2\} \setminus H_1 \end{cases} \tag{40}$$

$$p_\alpha^{2,A}(x) = \begin{cases} \sum_{i=1}^{a} (\mathbf{A}_{i,j} \cdot \alpha^{i-1}) & \text{for} \quad x = h_j \text{ where } j \in \{1, 2, ..., b\} \\ 0 & \text{for} \quad x \in \{H_1 \cup H_2\} \setminus H_2 \end{cases} \tag{41}$$

After computing this four vector polynomials, the inner product formula

$$(\sum_{i=1}^{a} \mathbf{a}_i \cdot \alpha^{i-1}) - [\sum_{j=1}^{b} \mathbf{z}_j \cdot (\sum_{i=1}^{a} \mathbf{A}_{i,j} \cdot \alpha^{i-1})] = 0 \tag{42}$$

becomes to

$$\sum_{x \in H_1 \cup H_2} [f_{\mathbf{a}}(x) p_\alpha^1(x) - f_{\mathbf{z}}(x) p_\alpha^{2,\mathbf{A}}(x)] = 0 \tag{43}$$

, which means that the prover need to convince the verifier that the polynomial $f_{\mathbf{a}}(x) p_\alpha^1(x) - f_{\mathbf{z}}(x) p_\alpha^{2,A}(x)$ sums to 0 over the domain $H_1 \cup H_2$, this is done by engaging a Sumcheck protocol with the verifier, Sumcheck will be discussed in section A.2.

Naively, the prover needs to use three Linchecks for proving $\mathbf{a} = \mathbf{Az}, \mathbf{b} = \mathbf{Bz}$ and $\mathbf{c} = \mathbf{Cz}$. An optimization using random linear combination can reduce the number of Linchecks to one, the modified Lincheck is called the "amortized Lincheck protocol", in which the verifier first samples three uniformly random elements $s_1, s_2, s_3 \in \mathbb{F}$ and send to the prover, and then asks the prover to show that the private vector $s_1 \cdot \mathbf{a} + s_2 \cdot \mathbf{b} + s_3 \cdot \mathbf{c}$ is equal to $s_1 \cdot \mathbf{Az} + s_2 \cdot \mathbf{Bz} + s_3 \cdot \mathbf{Cz}$ using only one call to Lincheck protocol.

Figure 4 shows the interactive flow of Lincheck protocol under IOP model, here one thing to keep in mind is that when the prover sends a polynomial to the verifier in the figure, it actually means to send a commitment of the polynomial at that time, and then after all the interactions, the verifier will ask the prover to open some values of the committed polynomial at some points.

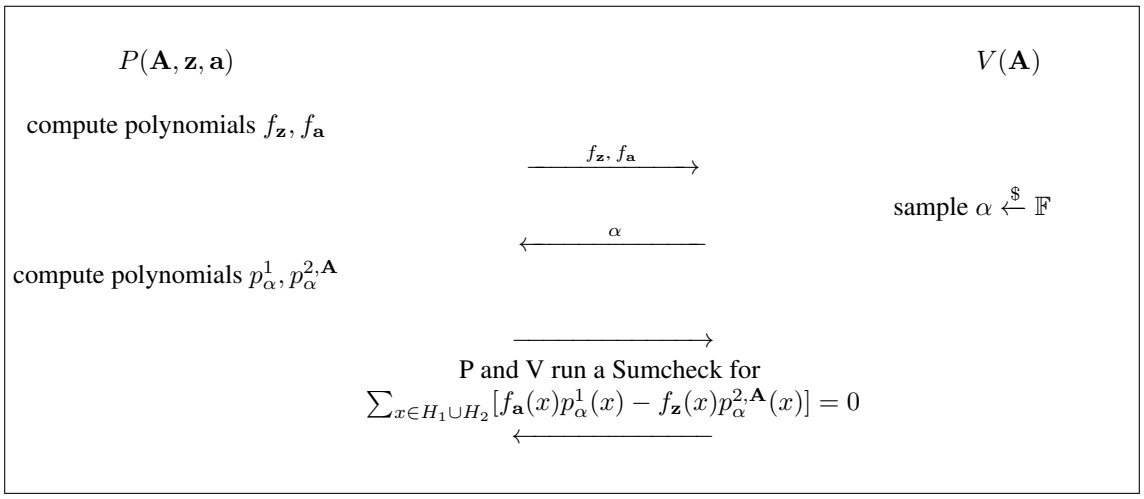


Figure 4: Lincheck protocol

A.2 Sumcheck

The Sumcheck protocol is an interactive protocol which the prover can show the statement "I know a polynomial f such that $\sum_{x \in \mathbf{S}} f(x) = 0$" to the verifier, where $\mathbf{S}$ is a public know set predetermined by the prover and the verifier. In the use of Lincheck, prover's claimed polynomial is $f_{\mathbf{a}}(x)p_\alpha^1(x) - f_{\mathbf{z}}(x)p_\alpha^{2,A}(x)$ and the set is $H_1 \cup H_2$. The prover first computes a polynomial division of $f(x)$ divided by $Z_{H_1 \cup H_2}(x) := \Pi_{y \in H_1 \cup H_2}(x - y)$, getting polynomials g and h such that $f(x) = g(x) + h(x) \cdot Z_{H_1 \cup H_2}$. We have that $\sum_{x \in H_1 \cup H_2} f(x) = \sum_{x \in H_1 \cup H_2} g(x) + h(x) \cdot Z_{H_1 \cup H_2} = \sum_{x \in H_1 \cup H_2} g(x)$. Now, we use a mathematical fact that $\sum_{x \in \mathbf{S}} x^t = 0$ for any $t \in \{0, 1, 2, ..., |S| - 2\}$ and for any non-trivial subspace $\mathbf{S}$ of $\mathbb{F}$. Using this fact, the term $\sum_{x \in H_1 \cup H_2} g(x)$ actually equals to $\sum_{x \in H_1 \cup H_2} \beta \cdot x^{|H_1 \cup H_2| - 1}$ where $\beta \in \mathbb{F}$ is the coefficient of $x^{|H_1 \cup H_2| - 1}$ in g. Thus $\sum_{x \in H_1 \cup H_2} f = 0$ if and only if g is a polynomial with degree strictly less than $|H_1 \cup H_2| - 1$. The prover sends polynomials f and h to the verifier and then engages a low degree test with the verifier to check that $f - h \cdot Z_{H_1 \cup H_2} (= g)$ has degree less than $|H_1 \cup H_2| - 1$. Figure 5 shows the interactive flow of Lincheck protocol under IOP model.

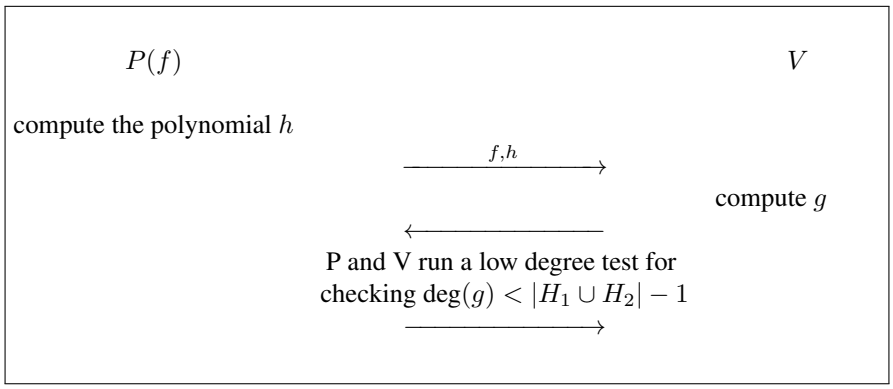


Figure 5: Sumcheck protocol

A.3 Rowcheck

After using three Linchecks (or one amortized Lincheck) to show that prover knows $\mathbf{a} = \mathbf{Az}, \mathbf{b} = \mathbf{Bz}, \mathbf{c} = \mathbf{Cz}$, the prover still needs to prove that $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$, this is equavalent to show that the corresponding encoded polynomials $f_{\mathbf{a}}, f_{\mathbf{b}}, f_{\mathbf{c}}$ in the Lincheck protocol satisfies the following:

$$f_{\mathbf{a}}(x) \cdot f_{\mathbf{b}}(x) = f_{\mathbf{c}}(x) \text{ for all } x \in H_1 \tag{44}$$

, which is equivalent to

$$f_{\mathbf{a}}(x) \cdot f_{\mathbf{b}}(x) - f_{\mathbf{c}}(x) = p(x) \cdot Z_{H_1}(x) \text{ for all } x \in \mathbb{F} \tag{45}$$

for some polynomial p with degree less than $\max\{deg(f_{\mathbf{a}}) + deg(f_{\mathbf{b}}) - deg(Z_{H_1}), deg(f_{\mathbf{c}}) - deg(Z_{H_1})\} + 1 = |H_1| - 1$, and that $Z_{H_1}(x) := \Pi_{y \in H_1}(x - y)$. Therefore if we choose L to have size larger than $\max\{deg(f_{\mathbf{a}}) + deg(f_{\mathbf{b}}), deg(f_{\mathbf{c}})\} = 2|H_1|$, then (45) holds if and only if the polynomial p' defined by the corresponding Reed-Solomon code

$$p'(x) := \frac{f_{\mathbf{a}}(x) \cdot f_{\mathbf{b}}(x) - f_{\mathbf{c}}(x)}{Z_{H_1}(x)} \text{ for all } x \in L \tag{46}$$

over space L, is a polynomial with degree less than $|H_1| - 1$.

In summary, for the prover to prove the statement "I know secret vectors $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{F}^a$ such that $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$", the prover and the verifier only need to engage an FRI protocol to check the polynomial f with corresponding Reed-Solomon codeword (46) has degree less than $|H_1| - 1$, this process is called Rowcheck.

B An Improved Chernoff Bound

In this section, we explain how to use Chernoff bound more carefully in order to get a tighter bound of formula (25). We will use the notation in [BCS16], Lemma 3.4, Fact 1. Recall that if we let ρ be a random function chosen uniformly from the functions with domain $\{0,1\}^{2\lambda}$ and range $\{0,1\}^{\lambda}$, and let $z \in \{0,1\}^{\lambda}$ be a fixed value, then the authors define the random variable (over the choice of ρ) $\delta_\rho(z) := |\Pr_r[\rho(r) = z] - \Pr[X = z]|$ where r is taken over $\{0,1\}^{2\lambda}$ and X has the uniform distribution over $\{0,1\}^{\lambda}$. Note that the definition of $\delta_\rho(z)$ is equivalent to $\delta_\rho(z) := |\frac{\sum_r \chi_{\rho(r)=z}}{2^{2\lambda}} - \frac{1}{2^{\lambda}}|$ where r is taken over $\{0,1\}^{2\lambda}$ and $\chi_{\rho(r)=z}$ is the indicator function to indicate whether $\rho(r) = z$.

Now we define a random variable Y_ρ (the randomness is taken over random choice of ρ) for a fixed value $z \in \{0,1\}^{\lambda}$ to be $Y_\rho(z) := \sum_r \chi_{\rho(r)=z}$, note that the expectation for $Y_\rho(z)$ is $\mu = 2^{\lambda}$ for all z.

We have

$$\Pr_\rho[\delta_\rho(z) \geq \delta \cdot 2^{-\lambda}] = \Pr_\rho[|\frac{\sum_r \chi_{\rho(r)=z}}{2^{2\lambda}} - 2^{-\lambda}| \geq \delta \cdot 2^{-\lambda}] \tag{47}$$

$$= \Pr_\rho[|(\sum_r \chi_{\rho(r)=z}) - 2^{\lambda}| \geq \delta \cdot 2^{\lambda}] \tag{48}$$

$$= \Pr[|Y_\rho(z) - \mu| \geq \delta \cdot \mu] \tag{49}$$

$$\leq 2e^{-\frac{\delta^2 \mu}{3}} \tag{50}$$

where the first equality uses the equivalent definition of $\delta_\rho(z)$, the second equality comes from multiplying both side in the probability by $2^{2\lambda}$, the third equation uses the definition of $Y_\rho(z)$ and the fact that $\mu = 2^{\lambda}$, and the final inequality comes from the Chernoff bound.

In the original paper, the authors set δ to be $2^{-\frac{\lambda}{4}}$ and use the inequality $2e^{-\frac{\delta^2\mu}{3}} \leq 2^{-2\lambda}$, note that here the inequality is very loose if λ is large enough. The value δ will eventually shows up as a major part of the denominator of the second term of formula (25). Here we also recall that λ represents the output length of the random oracle (measured in bits).

In our L1 security parameter sets, λ is always larger than or equal to 256, so we can actually set δ to be $2^{-0.48\lambda}$ without violating the inequality $2e^{-\frac{\delta^2\mu}{3}} \leq 2^{-2\lambda}$, and the resulting formula for the zero-knowledge lost becomes

$$z' = z + \frac{p}{2^{0.48\lambda+2}} \tag{51}$$

In our L3 security parameter sets, λ is always larger than or equal to 384, so we can set $\delta := 2^{-0.486\lambda}$ without violating the inequality $2e^{-\frac{\delta^2\mu}{3}} \leq 2^{-2\lambda}$, and the resulting formula for the zero-knowledge lost becomes

$$z' = z + \frac{p}{2^{0.486\lambda+2}} \tag{52}$$

In our L5 security parameter sets, λ is always larger than or equal to 512, so we can set $\delta := 2^{-0.489\lambda}$ without violating the inequality $2e^{-\frac{\delta^2\mu}{3}} \leq 2^{-2\lambda}$, and the resulting formula for the zero-knowledge lost becomes

$$z' = z + \frac{p}{2^{0.489\lambda+2}} \tag{53}$$