CONCORDIUM

Cryptographic Bluepaper

October 15, 2025

Version: 2.3.1-pre

CONCORDIUM

Contents

Part VI   Byttes   278

23 Preliminaries   279

24 Network Serialization   304

Part VII   Initial Konsensus   327

A Konsensus Preliminaries   328

B Skov Data-Structure   330

Chapter 1

Overview

1.1 Outline

This document outlines the cryptographic specifications of the Concordium blockchain. Each part of the document focuses on a specific aspect of the system, allowing readers to quickly locate relevant technical details: Part I provides an inventory of cryptographic primitives, such as signature schemes and hash functions. Part II presents the BFT-style consensus protocol. Part III describes the identity layer and how it is integrated into the account system. Part IV discusses the update mechanism of the blockchain. Part V explains how to bootstrap the blockchain and securely generate genesis data. Part VI provides information on the serialization of various objects and types. In the appendix, Part VII describes the consensus protocol initially used by the Concordium blockchain.

1.2 How To Read This Document

This reference document is not intended to be read sequentially. Use the table of contents and the index to locate[1] relevant information.

1.2.1 Information Boxes

Colored boxes are used throughout this document to highlight information. Each box type draws attention to information of varying importance or context, helping readers prioritize and interpret content efficiently.

> ⚠ A *warning box* contains information of high importance. Such information may have implications for the security of protocols. For example, a warning box may highlight the range of parameters for which a protocol remains secure.
> *Look for these boxes to avoid critical mistakes or misunderstandings.*

> ❶ A *notice box* contains information of medium importance. For example, a notice box may explain how a protocol can be implemented more efficiently, if needed.

---

[1]In the electronic version, hyperlinks and the search function of your document viewer may also be helpful.

An *info box* contains information of low importance. For example, an info box may elaborate on non-critical design choices within a protocol.

A *post-quantum info box* contains information regarding the post-quantum security of a component. For example, it may discuss replacement options for components that could be broken by a quantum computer.

An *idea box* such as this provides suggestions or designs for possible future improvements.

An *implementation information box* such as this one contains information about the actual implementation of the protocol.
Consult these for practical implementation details.

Chapter 2

Preliminaries

This chapter provides an overview of the mathematical notation and concepts used to define the cryptographic primitives and protocols discussed in later sections.

2.1 Notation

We use $X \coloneqq Y$ to denote that the expression X is defined as Y. For example, $\mathbb{N} \coloneqq \{0, 1, 2, 3, \ldots\}$.

2.1.1 Types and Variables

We use the `variable` font for variables and the TYPE font for types. For example, we write `block` $\in$ BLOCKS to denote that the variable `block` (a block) is of type BLOCKS (the set of all blocks). We denote by TYPE* the type of finite sequences of a given type TYPE. For example, `blocklist` $\in$ BLOCKS* denotes a finite sequence of blocks.

2.1.2 Basic Types

This section defines basic types, that is, specific sets used throughout the bluepaper.

Numbers.

$\mathbb{N}$ Denotes the set of *natural numbers*. This normally includes 0, i.e., $\mathbb{N} = \{0, 1, 2, \ldots\}$. In cases where the inclusion or exclusion of 0 matters, we use $\mathbb{N}_0$ or $\mathbb{N}^+$.

$\mathbb{N}_0$ Denotes the set of *non-negative integers*, i.e., $\mathbb{N}_0 \coloneqq \{0, 1, 2, \ldots\}$.

$\mathbb{N}^+$ Denotes the set of *strictly positive integers*, i.e., $\mathbb{N}^+ \coloneqq \{1, 2, 3, \ldots\}$.

$\mathbb{Q}$ Denotes the set of *rational numbers*.

$\mathbb{R}$ Denotes the set of *real numbers*.

$\mathbb{R}^+$ Denotes the set of *non-negative real numbers*, i.e., $\mathbb{R}^+ \coloneqq \{x \in \mathbb{R} \mid x \geq 0\}$.

$\mathbb{Z}$ Denotes the set of *integers*, i.e., $\mathbb{Z} \coloneqq \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

$\mathbb{Z}_m$ Denotes the *integers modulo m*, i.e., $\mathbb{Z}_m \coloneqq \{0, \ldots, m-1\}$. Depending on context, this also denotes the *additive group of integers modulo m* or the *ring of integers modulo m*. In particular, for m prime, $\mathbb{Z}_m$ denotes the *finite field* of integers modulo m (see $\mathbb{F}$).

$\mathbb{Z}_m^*$ Denotes the *multiplicative group of invertible integers modulo m*. For m prime, $\mathbb{Z}_m^* = \mathbb{Z}_m \setminus \{0\}$.

$[a, b]$	Denotes the *closed interval* or *range*, i.e., $[a, b] \coloneqq \{x \in R \mid a \leq x \leq b\}$.
$[a, b)$	Denotes the *right-open interval*, i.e., $[a, b) \coloneqq \{x \in R \mid a \leq x < b\}$.

Boolean and bit strings.

BOOL	Denotes the *Boolean* type, i.e., BOOL $= \{\texttt{false}, \texttt{true}\} \coloneqq \{0, 1\}$.
$\{0, 1\}^k$	Denotes the set of *bit strings of length k*, e.g., BOOL $= \{0, 1\}^1$.
$\{0, 1\}^*$	Denotes the set of finite *bit strings*, i.e., $\{0, 1\}^* \coloneqq \bigcup_{k=1} \{0, 1\}^k$.

Sets.

$\varnothing$	Denotes the *empty set*.

2.1.3 Basic Operations

This section defines basic operators and notation used throughout the bluepaper.

Sets and functions.

$x \in S$	Denotes that x is an element of the set S.
$x \notin S$	Denotes that x is *not* an element of the set S.
$A \subseteq S$	Denotes that A is a *subset* of the set S.
$A \subsetneq S$	Denotes that A is a *strict subset* of the set S.
$A \cup B$	Denotes the *union* of sets A and B, i.e., $A \cup B \coloneqq \{c \mid c \in A \vee c \in B\}$.
$A \cap B$	Denotes the *intersection* of sets A and B, i.e., $A \cap B \coloneqq \{c \mid c \in A \wedge c \in B\}$.
$A \setminus B$	Denotes the *set difference* of sets A and B, i.e., $A \setminus B \coloneqq \{c \mid c \in A \wedge c \notin B\}$.
$\overline{A}$	Denotes the *complement* of set A relative to some universe, i.e., $\overline{A} \coloneqq U \setminus A$. The universe is often implicit and given by context.
$A \times B$	Denotes the *Cartesian product* of sets A and B, i.e., $A \times B \coloneqq \{(a, b) \mid a \in A \wedge b \in B\}$.
A^n	Denotes the *n-ary Cartesian product* of set A, i.e., $A^n = A \times \cdots \times A$ (n times).
$\|S\|$	Denotes the *cardinality* of set S.
$\mathcal{P}(S)$	Denotes the *power set* of set S, i.e., $\mathcal{P}(S) \coloneqq \{A \mid A \subseteq S\}$.
$f \times g$	Denotes the *Cartesian product of functions* f and g, i.e., $(f \times g)(x, y) \coloneqq (f(x), g(y))$.

Vectors.

S^*	Denotes the set of *finite (possibly empty) lists* with elements in a given set S.
$\vec{a}$	Denotes a *vector* of elements from a set S, i.e., $\vec{a} = (a_1, \ldots, a_n) \in S^*$.
$\mathbf{a}$	Alternative notation for *vectors*, used if the arrow notation $\vec{a}$ makes formulas too cluttered.
$\langle \vec{a}, \vec{b} \rangle$	Denotes the *inner product* of vectors $\vec{a}$ and $\vec{b}$ in a real vector space, i.e., $\langle \vec{a}, \vec{b} \rangle = \sum_{i=1}^n a_i \cdot b_i$. An example of a real vector space is $\mathbb{Z}_q^n$.

$\vec{a} \circ \vec{b}$ Denotes the *Hadamard product* of vectors $\vec{a}$ and $\vec{b}$ in a real vector space, i.e., $\vec{a} \circ \vec{b} = (a_1 \cdot b_1, \ldots, a_n \cdot b_n)$.

Boolean and logic.

$x || y$ Denotes the *concatenation* of bit strings x and y. This also applies to other values; for example, $1||$"hello world" denotes the concatenation of 1 (the integer) and the string "hello world" as bit strings, where the encoding of both is defined by the context.

$x \wedge y$ Denotes the *Boolean conjunction*, also known as the *AND operator*. For bit strings x and y, it denotes the bitwise AND. For two logical statements S_1 and S_2, $S_1 \wedge S_2$ is true if and only if both S_1 and S_2 are true.

$x \vee y$ Denotes the *Boolean disjunction*, also known as the *OR operator*. For bit strings x and y, it denotes the bitwise OR. For two logical statements S_1 and S_2, $S_1 \vee S_2$ is true if at least one of them is true.

$x \oplus y$ Denotes the *exclusive OR*, also known as the *XOR operator*. For bit strings x and y, it denotes the bitwise XOR. For two logical statements S_1 and S_2, $S_1 \oplus S_2$ is true if exactly one of them is true.

Arithmetic.

$a + b$ Denotes *addition* of a and b in a ring, e.g., in $\mathbb{Z}_q$. It can also denote the *group operation* in a group $\mathbb{G}$ with *additive notation*. Groups of this type are mainly elliptic curve groups. It should be clear from context whether addition or group operation is meant.

$\sum_{i=1}^{n} a_i$ Denotes the *sum* $a_1 + \cdots + a_n$ of the n elements $a_1, \ldots, a_n$. We write $\sum_{i \in \mathcal{I}} a_i$ for the sum of the elements a_i where $i \in \mathcal{I}$ for index set $\mathcal{I}$, or $\sum_{a \in A} a$ for the sum of all elements in A.

$a \cdot b$ or ab Denotes *multiplication* of a and b in a ring, e.g., in $\mathbb{Z}_q$. It can also denote the *group operation* in a group $\mathbb{G}$ with *multiplicative notation*. It should be clear from context whether multiplication or group operation is meant.

$\prod_{i=1}^{n} a_i$ Denotes the *product* $a_1 \cdot \cdots \cdot a_n$ of the n elements $a_1, \ldots, a_n$. We write $\prod_{i \in \mathcal{I}} a_i$ for the product of the elements a_i where $i \in \mathcal{I}$ for index set $\mathcal{I}$, or $\prod_{a \in A} a$ for the product of all elements in A.

$a \pmod{b}$ Denotes a *modulo* b for integers a and b. That is, $a \pmod{b}$ denotes the unique element c in $\mathbb{Z}_b$ such that there exists a $k \in \mathbb{Z}$ with $a = c + k \cdot b$.

$\lfloor a \rfloor$ Denotes the *floor* of a, i.e., the greatest integer less than or equal to a. This is also known as the *integral part* of a.

$\lceil a \rceil$ Denotes the *ceiling* of a, i.e., the least integer greater than or equal to a.

$\log a$ Denotes the *binary logarithm* of a, i.e., the logarithm using base 2.

Probability theory.

$y \leftarrow S$ Denotes that y is *sampled uniformly at random* from the set S. If explicitly stated in the context, the sampling distribution may be different.

$y \leftarrow \mathsf{R}(x)$ Denotes that y is the *output* of algorithm R with input x. This applies to both deterministic and probabilistic algorithms.

$\Pr[X]$ Denotes the *probability* of *event X*.

Cryptography.

κ Denotes the *security parameter*.

$\mathcal{O}(g)$ Denotes the class of functions with growth rate bounded by g. If $f \in \mathcal{O}(g)$ there exists $c > 0$ and x_0 such that for all $x \geq x_0$ $|f(x)| \leq c|g(x)|$. $\mathcal{O}$ is known as *Big O notation* or *Landau notation*.

$\mathsf{poly}(m)$ Denotes the class of *polynomial bounded functions* in m. If $f \in \mathsf{poly}(m)$ there exists a c such that $f \in \mathcal{O}(m^c)$.

$\mathsf{negl}(m)$ Denotes the class of *negligible functions* in m. If $f \in \mathsf{negl}(m)$ then for all $p \in \mathsf{poly}(m)$ $f \in \mathcal{O}\left(\frac{1}{p(m)}\right)$.

2.1.4 Access Structures

An access structure defines which sets of participants are authorized to access a secret or resource (e.g., in a secret sharing scheme). More formally, an access structure is defined as follows.

Definition 1. Let S be a set. An *access structure* $\mathfrak{A}$ over S is a set of subsets of S: $\mathfrak{A} \subseteq \mathcal{P}(S)$. Its elements $A \in \mathfrak{A}$ are called *authorized sets*.

Threshold access structures. A simple example of an access structure over a set S is obtained by defining the authorized sets to be all subsets of S of cardinality at least some threshold T:

$$\mathfrak{A} := \{A \subseteq S \mid |A| \geq T\}.$$

Multi-signature schemes. An important use case of threshold access structures is in multi-signature schemes. In such a scheme, there are n signing keys in total and at least $k \leq n$ are required to sign a message for the multi-signature to be valid. We additionally use the following rules:

- At least k valid signatures are required, but more than k are allowed. This means the corresponding access structure is monotone.

- If an invalid signature is part of the multi-signature, it is considered invalid, even if there are k valid signatures in total.

- If the multi-signature contains a (valid) signature from a key that is not part of the n authorized keys, the multi-signature is considered invalid, even if there are k valid signatures from the correct keys in total.

> ⓘ In the last two cases, when there are k valid signatures, one can, of course, obtain a valid signature by removing the invalid or unauthenticated signature.

2.2 Groups, Fields, Pairings

This section assumes some familiarity with basic concepts from number theory used in cryptography. For an introduction to the topic, in particular the definitions of groups and fields, see, e.g., [KL20] (Section 9 and Appendix A). In the following, let q be a prime.

> ⚠ The choice of q is individual for each use (of e.g., a cryptographic primitive) and depends on the security requirements.

2.2.1 Group

The cryptographic schemes and protocols in the bluepaper use groups of prime order. These groups are cyclic, i.e., they can be generated by a single element.

Definition 2. We denote by $\mathbb{G}$ a *group* of order q with neutral element $1_{\mathbb{G}}$ and generator g.

2.2.2 Finite Field

A finite field is a set of elements in which addition, subtraction, multiplication, and division are defined and satisfy the usual rules of arithmetic, and which contains a finite number of elements. An example is $\mathbb{Z}_p$ for a prime p.

Definition 3. We denote by $\mathbb{F}$ a *finite field*.

For a field $\mathbb{F}$, we denote by $\mathbb{F}^{\times}$ the group of units, i.e., $\mathbb{F}^{\times} = \mathbb{F} \setminus \{0\}$.

2.2.3 Pairing

A pairing is a bilinear map that takes elements from two groups and outputs an element in a target group. This type of mapping allows for efficient cryptographic schemes, such as signature schemes.

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be groups of prime order q. Let g_1 (respectively, g_2) be a generator of $\mathbb{G}_1$ (respectively, $\mathbb{G}_2$).

Definition 4. A *pairing* on $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ such that

Bilinearity: For all $u_1, u_2 \in \mathbb{G}_1$, $v_1, v_2 \in \mathbb{G}_2$,

$$e(u_1 + u_2, v_1) = e(u_1, v_1)e(u_2, v_1) \text{ and } e(u_1, v_1 + v_2) = e(u_1, v_1)e(u_1, v_2),$$

Non-degeneracy: $e(g_1, g_2) \neq 1_{\mathbb{G}_T}$,

Computability: e can be efficiently computed.

> ⚠ The fact that e is non-degenerate implies that $e(g_1, g_2)$ generates $\mathbb{G}_T$.

We now define three types of pairings.

Definition 5. A pairing e on $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ is

- of *Type 1* if $\mathbb{G}_1 = \mathbb{G}_2$,

- of *Type 2* if $\mathbb{G}_1 \neq \mathbb{G}_2$ and there exists an efficiently computable isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$,

- of *Type 3* if $\mathbb{G}_1 \neq \mathbb{G}_2$ and there exists no efficiently computable isomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$ in either direction.

A pairing of Type 1 is called *symmetric*, and pairings of Types 2 and 3 are called *asymmetric*.

> ⚠ Different schemes will require different types of pairing. Using the wrong type can be a security issue.

2.2.4 Instances

The following groups and fields are used in the Concordium blockchain system. First, we define a simple elliptic curve group.

Definition 6. Consider the twisted Edwards curve $-x^2 + y^2 = 1 - \frac{121665}{121666} x^2 y^2$ over the finite field of order $2^{255} - 19$. We denote by $\mathbb{G}^{\mathtt{Ed}}$ the subgroup of large prime order q over that curve with generator $g_{\mathtt{Ed}}$ as defined in [JL17].

If it is clear from context, we will omit the subscript from the generator $g_{\mathtt{Ed}}$ and simply write g.

> **i** The order q of $\mathbb{G}^{\mathtt{Ed}}$ is the prime $2^{252} + 27742317777372353535851937790883648493$.

> **i** See also [Ber06] for information on the closely related Curve25519.

Next, we define a pairing with corresponding groups.

Definition 7. [Bow17] We denote by $(\mathbb{G}_1^{\mathtt{BLS}}, \mathbb{G}_2^{\mathtt{BLS}}, \mathbb{G}_T^{\mathtt{BLS}})$ the groups over elliptic curve BLS12-381 with pairing e as defined in [Bow17]. Denote by $g_{\mathtt{BLS},1}$, $g_{\mathtt{BLS},2}$ generators for $\mathbb{G}_1^{\mathtt{BLS}}$ and $\mathbb{G}_2^{\mathtt{BLS}}$, respectively. We denote by $\mathbb{F}_{\mathtt{BLS}}$ a finite field of the same order as these groups.

The above pairing is of Type 3. We will drop the BLS subscript for generators if the context permits.

> **i** The high-level reason for also having an elliptic curve group without pairing is efficiency.
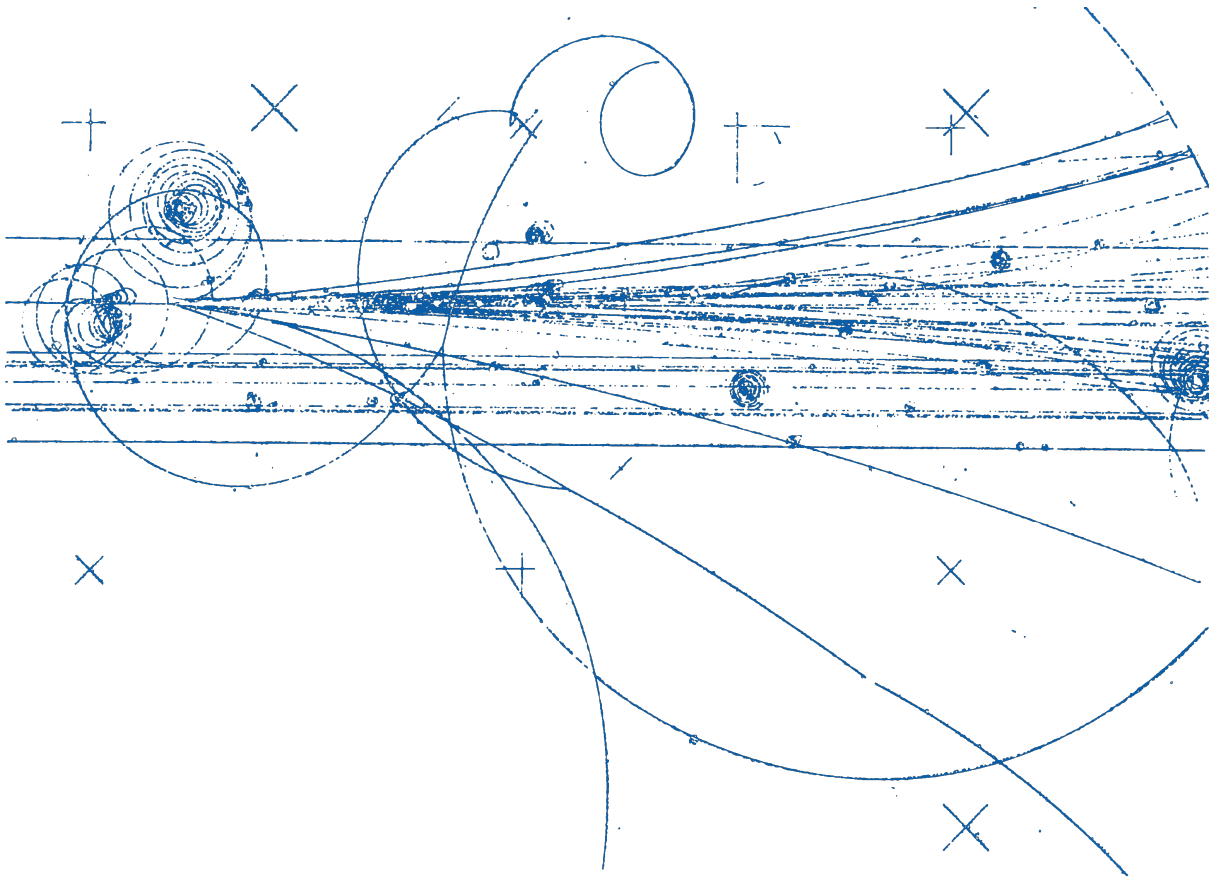
Part I

Cryptographic Primitives

Chapter 3

Hash Functions

A *hash* function is an algorithm that transforms data of arbitrary size into a fixed-size bit string, known as a *hash value* or digest. These functions are essential in computer science for various applications, including data storage, retrieval, and security. Cryptographic hash functions—a specialized type—are designed with specific security properties:

Pre-image resistance It should be computationally infeasible to find an input that produces a specific hash output.

Second pre-image resistance Given an input, it should be computationally infeasible to find a different input that produces the same hash output.

Collision resistance It should be computationally infeasible to find two different inputs that produce the same hash output.

A more detailed introduction to hash functions can be found in Chapter 5 of [KL20].

3.1 Secure Hash Algorithms

We use *Secure Hash Algorithms* (SHA) in various components.

> ⚛ It is conjectured that hash functions such as SHA-256 and SHA-3 remain secure in a post-quantum setting. Due to brute-force attacks using Grover's algorithm, a hash function with k-bit classical security will have $\frac{k}{2}$ bits of quantum security. Therefore, in the long term, it is advisable to switch to a SHA-3 variant with higher security, e.g., SHA-3-512.

3.1.1 SHA-256

The $\mathsf{H}_{\mathsf{SHA256}}$ *hash function* is a cryptographic hash function defined in a NIST standard [NIS15].

Function $\mathsf{H}_{\mathsf{SHA256}}$

Inputs
  $\mathtt{str} \in \{0,1\}^*$
        The bit string whose hash we want to compute.

Outputs

$$\texttt{hash} \in \text{HASH}_{\texttt{SHA256}} := \{0,1\}^{256}$$
  The hash of string `str`.

> ❗ The function $\mathsf{H}_{\texttt{SHA256}}$ is assumed to be *collision resistant*.

3.1.2 SHA-3

The $\mathsf{H}_{\texttt{SHA3}}$ *hash function* is the successor to SHA-256 and is also defined in a NIST standard [Div15]. We use the variant with 256-bit output.

Function $\mathsf{H}_{\texttt{SHA3}}$

Inputs
  $\texttt{str} \in \{0,1\}^*$
    The bit string whose hash is to be computed.

Outputs
  $\texttt{hash} \in \text{HASH}_{\texttt{SHA3}} := \{0,1\}^{256}$
    The hash of string `str`.

> ❗ We will use $\mathsf{H}_{\texttt{SHA3}}$ whenever a protocol requires a *random oracle*. While hash functions can never perfectly replace a random oracle, using $\mathsf{H}_{\texttt{SHA3}}$ is sufficient for most applications.

3.1.3 Use Cases

Block hash function. Konsensus uses $\mathsf{H}_{\texttt{SHA256}}$ as the *block hash* function H to link blocks.

Definition 8. The hash function H is $\mathsf{H}_{\texttt{SHA256}}$ and $\text{HASH} := \text{HASH}_{\texttt{SHA256}}$.

> ❗ The block hash function H **must be** collision-resistant.

> ℹ️ When we write $\mathsf{H}(\texttt{block})$ in Konsensus, we usually refer to a specific hash tree of the block `block`, where the hash function used to build the tree is H.

Random oracle replacement. Various components use $\mathsf{H}_{\texttt{SHA3}}$ as the *random oracle* $\mathsf{H}_{\texttt{RO}}$.

Definition 9. The hash function $\mathsf{H}_{\texttt{RO}}$ is $\mathsf{H}_{\texttt{SHA3}}$ and $\text{HASH}_{\texttt{RO}} := \text{HASH}_{\texttt{SHA3}}$.

> ❗ The random oracle replacement $\mathsf{H}_{\texttt{RO}}$ **must be** (computationally) indistinguishable from a random oracle.

3.2 Group Hash Functions

This section describes hash functions that map directly to groups. These functions are, for example, used to generate commitment keys (cf. Section 22.1.1). This application rules out constructions where the output is computed as some generator raised to an exponent that is computed by hashing the input.

> ⚙ The *group hash functions* in this section should still be safe (they use SHA-256 internally). However, they will become obsolete as our `BLS12-381`-based signature schemes become unsafe to use.

3.2.1 BLS12-381-G1 Hash Function

The following hash function is used as a random oracle that maps bit strings to elements in group $\mathbb{G}_1^{\mathtt{BLS}}$ of the `BLS12-381` curve (see Definition 7 in Section 2.2.4).

Implementation. For the implementation, the hash-to-curve suite

$$\mathtt{BLS12381G1\_XMD:SHA-256\_SSWU\_RO\_}$$

in [Faz+20] is used. This includes the use of domain separation tags as defined in Section 3.1 of [Faz+20]. Suite `BLS12381G1_XMD:SHA-256_SSWU_RO_` corresponds to Construction #2 in [WB19].

Interface.

> **Function** $\mathsf{H}_{\mathbb{G}_1^{\mathtt{BLS}}}^{\mathtt{BLS12\text{-}381}}$
>
> **Inputs**
> $\mathtt{str} \in \{0,1\}^*$
>   The bit string whose hash is to be computed.
>
> **Outputs**
> $\mathtt{hash} \in \mathbb{G}_1^{\mathtt{BLS}}$
>   The hash of string $\mathtt{str}$.

> ❗ The hash function $\mathsf{H}_{\mathbb{G}_1^{\mathtt{BLS}}}^{\mathtt{BLS12\text{-}381}}$ is assumed to be a *random oracle*.

Use cases. The hash function $\mathsf{H}_{\mathbb{G}_1^{\mathtt{BLS}}}^{\mathtt{BLS12\text{-}381}}$ is used as part of the BLS aggregate signature scheme (cf. Section 5.2). It is also used to securely generate the global Pedersen commitment keys (cf. Section 22.1.1) and to generate the dummy privacy guardian keys for the initial accounts in the genesis block (cf. Section 22.4.1).

> ⚠ These use cases require that $\mathsf{H}_{\mathbb{G}_1^{\mathtt{BLS}}}^{\mathtt{BLS12\text{-}381}}$ acts as a random oracle.

> ⚠ $\mathsf{H}^{\mathtt{BLS12\text{-}381}}_{\mathbb{G}^{\mathtt{BLS}}_1}$ *must not* allow one to learn the discrete logarithm of the hash output relative to a nontrivial[a] base.
> _____
> [a]That is, taking the output itself as a base.

3.2.2 BLS12-381-G2 Hash Function

The following hash function is used as a random oracle that maps bit strings to elements in group $\mathbb{G}^{\mathtt{BLS}}_2$ of the BLS12-381 curve (see Definition 7 in Section 2.2.4).

Implementation. For the implementation, the hash-to-curve suite

$$\mathtt{BLS12381G2\_XMD:SHA\text{-}256\_SSWU\_RO\_}$$

in [Faz+20] is used. This includes the use of domain separation tags as defined in Section 3.1 of [Faz+20].

Interface.

> **Function $\mathsf{H}^{\mathtt{BLS12\text{-}381}}_{\mathbb{G}^{\mathtt{BLS}}_2}$**
>
> **Inputs**
> $\mathtt{str} \in \{0,1\}^*$
>   The bit string whose hash is to be computed.
>
> **Outputs**
> $\mathtt{hash} \in \mathbb{G}^{\mathtt{BLS}}_2$
>   The hash of string $\mathtt{str}$.

> ❶ The hash function $\mathsf{H}^{\mathtt{BLS12\text{-}381}}_{\mathbb{G}^{\mathtt{BLS}}_2}$ is assumed to be a *random oracle*.

Use cases. The hash function $\mathsf{H}^{\mathtt{BLS12\text{-}381}}_{\mathbb{G}^{\mathtt{BLS}}_2}$ is used to generate dummy IDP keys required for the initial accounts in the genesis block (cf. Section 22.4.2).

> ⚠ These use cases require that $\mathsf{H}^{\mathtt{BLS12\text{-}381}}_{\mathbb{G}^{\mathtt{BLS}}_2}$ acts as a random oracle.

> ⚠ $\mathsf{H}^{\mathtt{BLS12\text{-}381}}_{\mathbb{G}^{\mathtt{BLS}}_1}$ *must not* allow one to learn the discrete logarithm of the hash output relative to a nontrivial base.

Chapter 4

Secret Sharing

A secret sharing scheme is a method of distributing a secret among a set of parties, where each party receives a share of the secret. No individual share, or any subset of shares below a certain threshold, reveals any information about the secret. Only when a sufficient number of shares are combined can the secret be reconstructed.

4.1 Shamir's secret sharing

The Personligt and Konto layers need a *secret sharing* scheme with threshold d which allows to split a secret s into n shares of s, $sh(s)_1, \ldots, sh(s)_n$. Any $d+1$ shares allow to reconstruct the secret s, whereas fewer shares contain no information about s.

> ⚛ Shamir's secret sharing is unconditionally secure. Quantum computer cannot help to speed up attacks.

4.1.1 Implementation

We use *Shamir's secret sharing* [Sha79] over a field $\mathbb{F}$. For the reconstruction we use the Lagrange interpolation.

How it works. To share a secret s, a random polynomial of degree d is constructed with s as the constant term. Each share is the evaluation of this polynomial at a distinct, nonzero point. To reconstruct the secret, any $d+1$ shares are sufficient to interpolate the polynomial and recover s as its value at zero.

Definition 10. Given a set of $d+1$ distinct *interpolation points* $\alpha_{i_1}, \ldots, \alpha_{i_{d+1}} \in \mathbb{F}$, the *Lagrange basis polynomial* ℓ_j for $j \in \{i_1, \ldots, i_{d+1}\}$ is defined as

$$\ell_j(x) = \prod_{k \in \{i_1, \ldots, i_{d+1}\} \setminus \{j\}} \frac{x - \alpha_k}{\alpha_j - \alpha_k}.$$

4.1.2 Interface

The following pseudocode describes the algorithms for sharing and reconstructing a secret.

Function Share

Inputs

$n \in \mathbb{N}^+$
    The number of shares.

$d \in \mathbb{N}^+$
    The *sharing threshold* where $d < n$.

$\alpha_1, \ldots, \alpha_n \in \mathbb{F} \setminus \{0\}$
    The interpolation points. They must be *distinct*.

$s \in \mathbb{F}$
    The value to be shared.

Outputs

$(\mathtt{sh}(s)_1, \ldots, \mathtt{sh}(s)_n) \in \mathbb{F}^n$
    A tuple of n shares.

Implementation

1: Choose random $a_1, \ldots, a_d \in \mathbb{F}$.
2: Set $f(x) = a_d x^d + \cdots + a_1 x + s$.
3: Set $\mathtt{sh}(s)_i = f(\alpha_i)$ for $i \in 1, \ldots, n$.
4: **return** $\mathtt{sh}(s)_1, \ldots, \mathtt{sh}(s)_n$.

Function Reconstruct

Inputs

$n \in \mathbb{N}^+$
    The number of shares.

$d \in \mathbb{N}^+$
    The sharing threshold where $d < n$.

$\alpha_1, \ldots, \alpha_n \in \mathbb{F} \setminus \{0\}$
    The interpolation points used in the sharing.

$(\mathtt{sh}(s)_{i_1}, \ldots, \mathtt{sh}(s)_{i_{d+1}}) \in \mathbb{F}^{d+1}$
    A tuple of $d + 1$ shares.

Outputs

$s \in \mathbb{F}$
    The reconstructed value.

Implementation

1: Use Lagrange interpolation to reconstruct $s = f(0)$, i.e. output

$$s = \sum_{j \in \{i_1, \ldots, i_{d+1}\}} \mathtt{sh}(s)_j \ell_j(0).$$

4.1.3 Use Cases

We use the above secret sharing scheme for shared encryption (see Section 7.1.5).

> ⚠️ We assume that any d shares contain no information about the secret, but that any d + 1 shares are sufficient to reconstruct s efficiently.

Chapter 5

Signature Schemes

This chapter describes the digital signature schemes used in Concordium, their security properties, and their roles in the protocol. Each scheme is selected for its suitability to a particular use case, such as block validation, account management, or credential issuance.

Digital *signature schemes* provide a way to verify the authenticity and integrity of digital messages, such as transactions on a blockchain. This involves a pair of keys: a signing key, also known as a private key, kept secret by the signer, and a verification key, also known as a public key, which is widely distributed (e.g., published on a blockchain). The signer uses their signing key to create a digital signature. Anyone with the corresponding verification key can then use it to verify the signature. A signature scheme should be *existentially unforgeable* under an *adaptive chosen-message attack*. Roughly, this means that an adversary cannot forge signatures for a given key and message, even if they have seen signatures on other messages. A more detailed introduction to digital signatures can be found in Chapter 12 of [KL20].

5.1 Edwards-Curve Digital Signature Algorithm

The *Edwards-curve Digital Signature Algorithm* (EdDSA) is used by Konsensus for block signatures and finalization.

> ⚛ The EdDSA signature scheme will be unsafe in a post-quantum setting, as Shor's algorithm allows computing discrete logarithms efficiently. NIST's post-quantum cryptography standardization [NIS17] will provide candidate schemes that could replace EdDSA in the future. In the short term, one could consider adding Lamport signatures [Lam79] (a type of one-time signature) as a backup to ease the transition to a post-quantum signature scheme.

5.1.1 Implementation

For the implementation, the `Ed25519` instance in [JL17] is used; see Definition 6 in Section 2.2.4.

5.1.2 Interface

The *key generation* function $\mathsf{KeyGen}_{\texttt{EdDSA}}$ allows creating key pairs.

Function $\mathsf{KeyGen_{EdDSA}}$

Outputs

$\mathtt{sk^{EdDSA}} \in \mathrm{SIGNKEYS_{EdDSA}} := \{0,1\}^{256}$
>   The *signing key*.

$\mathtt{vk^{EdDSA}} \in \mathrm{VERIFICATIONKEYS_{EdDSA}} := \mathbb{G}^{\mathrm{Ed}}$
>   The *verification key*.

The *sign* function $\mathsf{Sign_{EdDSA}}$ allows one to sign a message.

Function $\mathsf{Sign_{EdDSA}}$

Inputs

$\mathtt{sk^{EdDSA}} \in \mathrm{SIGNKEYS_{EdDSA}}$
>   The signing key.

$\mathtt{str} \in \{0,1\}^*$
>   The bit string we want to sign.

Outputs

$\sigma^{\mathrm{EdDSA}} \in \mathrm{SIGNATURES_{EdDSA}} := \mathbb{G}^{\mathrm{Ed}} \times \{1,\ldots,|\mathbb{G}^{\mathrm{Ed}}|-1\}$
>   The *signature* of $\mathtt{str}$ under $\mathtt{sk^{EdDSA}}$.

The *verification* function $\mathsf{Verify_{EdDSA}}$ allows one to verify a signature.

Function $\mathsf{Verify_{EdDSA}}$

Inputs

$\mathtt{vk^{EdDSA}} \in \mathrm{VERIFICATIONKEYS_{EdDSA}}$
>   The verification key.

$\mathtt{str} \in \{0,1\}^*$
>   The signed bit string.

$\sigma^{\mathrm{EdDSA}} \in \mathrm{SIGNATURES_{EdDSA}}$
>   The signature.

Outputs

$\mathtt{b} \in \mathrm{BOOL}$
>   Indicates if σ^{EdDSA} is a valid signature.

5.1.3 Use Cases

Block signature scheme. Konsensus requires a signature scheme $\mathsf{Signature_{BC}}$ which is used by validators to sign blocks. The EdDSA signature scheme is used as *block signature scheme* $\mathsf{Signature_{BC}}$ in Konsensus.

Definition 11. The $\mathsf{Signature_{BC}}$ is the EdDSA scheme. In particular:

- $\mathrm{SIGNKEYS_{BC}} := \mathrm{SIGNKEYS_{EdDSA}}$,

- $\mathrm{VERIFICATIONKEYS_{BC}} := \mathrm{VERIFICATIONKEYS_{EdDSA}}$,

- $\mathrm{SIGNATURES} \coloneqq \mathrm{SIGNATURES_{EdDSA}}$

- $\mathsf{KeyGen_{BC}} \coloneqq \mathsf{KeyGen_{EdDSA}}$, $\mathsf{Sign_{BC}} \coloneqq \mathsf{Sign_{EdDSA}}$, $\mathsf{Verify_{BC}} \coloneqq \mathsf{Verify_{EdDSA}}$

> ❶ Block signatures **must be** existentially unforgeable under an adaptive chosen message attack.

Account signature scheme. The signature scheme is used in Konto as the *account signature scheme* $\mathsf{Signature_{ACC}}$.

Definition 12. The $\mathsf{Signature_{ACC}}$ is the EdDSA scheme. In particular:

- $\mathrm{SIGNKEYS_{ACC}} \coloneqq \mathrm{SIGNKEYS_{EdDSA}}$,

- $\mathrm{VERIFICATIONKEYS_{ACC}} \coloneqq \mathrm{VERIFICATIONKEYS_{EdDSA}}$,

- $\mathrm{SIGNATURES_{ACC}} \coloneqq \mathrm{SIGNATURES_{EdDSA}}$

- $\mathsf{KeyGen_{ACC}} \coloneqq \mathsf{KeyGen_{EdDSA}}$, $\mathsf{Sign_{ACC}} \coloneqq \mathsf{Sign_{EdDSA}}$, $\mathsf{Verify_{ACC}} \coloneqq \mathsf{Verify_{EdDSA}}$

> ❶ This scheme must be existentially unforgeable under an adaptive chosen message attack.

Update signature scheme. The signature scheme is used as the signature scheme $\mathsf{Signature_{UPD}}$ to sign updates (cf. Renovatio).

Definition 13. The $\mathsf{Signature_{UPD}}$ is the EdDSA scheme. In particular:

- $\mathrm{SIGNKEYS_{UPD}} \coloneqq \mathrm{SIGNKEYS_{EdDSA}}$,

- $\mathrm{VERIFICATIONKEYS_{UPD}} \coloneqq \mathrm{VERIFICATIONKEYS_{EdDSA}}$,

- $\mathrm{SIGNATURES_{UPD}} \coloneqq \mathrm{SIGNATURES_{EdDSA}}$

- $\mathsf{KeyGen_{UPD}} \coloneqq \mathsf{KeyGen_{EdDSA}}$, $\mathsf{Sign_{UPD}} \coloneqq \mathsf{Sign_{EdDSA}}$, $\mathsf{Verify_{UPD}} \coloneqq \mathsf{Verify_{EdDSA}}$

> ❶ This scheme must be existentially unforgeable under an adaptive chosen message attack.

Fast signature scheme for finalization. The finalization layer Afgjort of the initial protocol requires a signature scheme $\mathsf{Signature_{Fin}}$ which is used by finalizers to sign finalization entries. The EdDSA signature scheme is used as the $\mathsf{Signature_{Fin}}$ signature scheme by finalizers during finalization (cf. Part VII).

Definition 14. The $\mathsf{Signature_{Fin}}$ is the EdDSA scheme. In particular:

- $\mathrm{SIGNKEYS_{Fin}} \coloneqq \mathrm{SIGNKEYS_{EdDSA}}$,

- $\mathrm{VERIFICATIONKEYS_{Fin}} \coloneqq \mathrm{VERIFICATIONKEYS_{EdDSA}}$,

- $\mathrm{SIGNATURES_{Fin}} \coloneqq \mathrm{SIGNATURES_{EdDSA}}$

- $\mathsf{KeyGen_{Fin}} \coloneqq \mathsf{KeyGen_{EdDSA}}$, $\mathsf{Sign_{Fin}} \coloneqq \mathsf{Sign_{EdDSA}}$, $\mathsf{Verify_{Fin}} \coloneqq \mathsf{Verify_{EdDSA}}$

> ❶ Finalization signatures **must be** existentially unforgeable under an adaptive chosen message attack.

5.2 BLS Aggregate Signature Scheme

The *BLS aggregate signature scheme* $\mathsf{Signature}_{\mathsf{AggFin}}$ is used to create compact finalization entries.

⚙ The BLS signature scheme will be unsafe in a post-quantum setting, as Shor's algorithm allows computing discrete logarithms efficiently. The current finalists of NIST's post-quantum cryptography standardization [NIS17] do not provide signature aggregation out of the box. Aggregation could be achieved using (post-quantum) SNARKs; see, e.g., [Zha+24] for a discussion of this approach. In the short term, one could consider adding Lamport signatures [Lam79] (a type of one-time signature) as a backup to ease the transition to a post-quantum signature scheme.

5.2.1 Implementation

For the implementation, the cipher suite

$$\texttt{BLS\_SIG\_BLS12381G1\_XMD:SHA-256\_SSWU\_RO\_POP\_}$$

in [Bon+20] is used. This cipher suite uses the pairing groups $(\mathbb{G}_1^{\mathsf{BLS}}, \mathbb{G}_2^{\mathsf{BLS}}, \mathbb{G}_T^{\mathsf{BLS}})$ on the BLS12-381 curve (see Definition 7 in Section 2.2.4). The cipher suite uses the hash function $\mathsf{H}_{\mathbb{G}_1^{\mathsf{BLS}}}^{\mathsf{BLS12\text{-}381}}$ from Section 3.2.1 that maps bit strings to elements in $\mathbb{G}_1^{\mathsf{BLS}}$.

The cipher suite corresponds to the signature scheme presented in [Bon+03], with additional proof-of-possession functions to allow for aggregate signatures on the same message.

❗ The security proof presented in [Bon+03] requires that the signature scheme use a **Type 2** pairing. However, according to [Cha+09], the scheme is secure even if a **Type 3** pairing (e.g., BLS12-381) is used. See [CM09] for a more general discussion of this topic.

5.2.2 Interface

The following functions are provided by the aggregated signature scheme. All functions mentioned below in the implementation information are the variants for the above cipher suite.

The *key-generation* function $\mathsf{KeyGen}_{\mathsf{BLS}}$ allows creating key pairs.

Function $\mathsf{KeyGen}_{\mathsf{BLS}}$

Outputs
  $\mathtt{sk}^{\mathsf{BLS}} \in \mathrm{SIGNKEYS}_{\mathsf{BLS}} := \mathbb{F}_{\mathsf{BLS}}$
      The *signing key*.
  $\mathtt{vk}^{\mathsf{BLS}} \in \mathrm{VERIFICATIONKEYS}_{\mathsf{BLS}} := \mathbb{G}_2^{\mathsf{BLS}}$
      The *verification key*.

Implementation A combination of KeyGen and SkToPk in [Bon+20].

The *sign* function $\mathsf{Sign}_{\mathsf{BLS}}$ allows to sign a message.

Function $\mathsf{Sign}_{\mathrm{BLS}}$

Inputs

$\mathtt{sk}^{\mathrm{BLS}} \in \mathrm{SIGNKEYS}_{\mathrm{BLS}}$
> The signing key.

$\mathtt{str} \in \{0,1\}^*$
> The bit string we want to sign.

Outputs

$\sigma^{\mathrm{BLS}} \in \mathrm{SIGNATURES}_{\mathrm{BLS}} := \mathbb{G}_1^{\mathrm{BLS}}$
> The *signature* of $\mathtt{str}$ under $\mathtt{sk}^{\mathrm{BLS}}$.

Implementation Sign in [Bon+20].

The *verification* function $\mathsf{Verify}_{\mathrm{BLS}}$ allows to verify a signature.

Function $\mathsf{Verify}_{\mathrm{BLS}}$

Inputs

$\mathtt{vk}^{\mathrm{BLS}} \in \mathrm{VERIFICATIONKEYS}_{\mathrm{BLS}}$
> The verification key.

$\mathtt{str} \in \{0,1\}^*$
> The signed bit string.

$\sigma^{\mathrm{BLS}} \in \mathrm{SIGNATURES}_{\mathrm{BLS}}$
> The signature.

Outputs

$\mathtt{b} \in \mathrm{BOOL}$
> Indicates if σ^{BLS} is a valid signature of $\mathtt{str}$ under $\mathtt{sk}^{\mathrm{BLS}}$.

Implementation Verify in [Bon+20].

The *aggregation* function $\mathsf{Aggregation}_{\mathrm{BLS}}$ allows aggregating signatures.

Function $\mathsf{Aggregation}_{\mathrm{BLS}}$

Inputs

$\sigma_1^{\mathrm{BLS}}, \ldots, \sigma_k^{\mathrm{BLS}} \in \mathrm{SIGNATURES}_{\mathrm{BLS}}{}^k$
> A set of signatures of strings $\mathtt{str}_1, \ldots, \mathtt{str}_k \in \{0,1\}^*$.

Outputs

$\sigma^{\mathrm{BLS}} \in \mathrm{SIGNATURES}_{\mathrm{BLS}}$
> The *aggregated signature*.

Implementation Aggregate in [Bon+20].

The *aggregation verification* function $\mathsf{AggregationVerification}_{\mathrm{BLS}}$ allows to verify an aggregated signature.

Function AggregationVerification$_{\text{BLS}}$

Inputs

$\text{vk}_1^{\text{BLS}}, \ldots, \text{vk}_k^{\text{BLS}} \in \text{VERIFICATIONKEYS}_{\text{BLS}}{}^k$
    The set of verification keys.

$\text{str}_1, \ldots, \text{str}_k \in \{0,1\}^*$
    The set of strings.

$\sigma^{\text{BLS}} \in \text{SIGNATURES}_{\text{BLS}}$
    The aggregated signature.

Outputs

$\text{b} \in \text{BOOL}$
    Indicates if σ^{BLS} is a valid signature of $\text{str}_1, \ldots, \text{str}_k$ under $\text{vk}_1^{\text{BLS}}, \ldots, \text{vk}_k^{\text{BLS}}$.

Implementation FastAggregateVerify in [Bon+20].

⚠ If AggregationVerification$_{\text{BLS}}$ is used to check aggregated signatures where some strings $\text{str}_1, \ldots, \text{str}_k$ are equal, the corresponding verification keys **must** come with valid proofs-of-possession.

To allow for aggregate signatures on the same message, the parties need to be able to prove that they know the signing key of their verification key. This can be done using the following two functions.

The *proof of possession* function PopProve$_{\text{BLS}}$ allows to generate a proof that one knows the signing key corresponding to a verification key.

Function PopProve$_{\text{BLS}}$

Inputs

$\text{sk}^{\text{BLS}} \in \text{SIGNKEYS}_{\text{BLS}}$
    The signing key.

Outputs

$\pi \in \{0,1\}^*$
    The proof of possession.

Implementation PopProve in [Bon+20].

The *proof of possession verification* function PopVerify$_{\text{BLS}}$ allows to verify a proof of possession for a given verification key.

Function PopVerify$_{\text{BLS}}$

Inputs

$\text{vk}^{\text{BLS}} \in \text{VERIFICATIONKEYS}_{\text{BLS}}$
    The verification key.

$\pi \in \{0,1\}^*$
    The proof of possession.

Outputs

$b \in \text{BOOL}$
    Indicates if π is a valid proof of possession for vk^{BLS}.

Implementation PopVerify in [Bon+20].

ⓘ Alternatively, one could also use (the non-interactive variant of) the sigma protocol dlog from Section 9.2.1. This might be useful in cases where the proof of possession needs to be context dependent.

5.2.3 Use Cases

Finalization entry. The BLS signature scheme is used as $\text{Signature}_{\text{AggFin}}$ signature scheme to create finalization entries at the end of a finalization run (cf. Section D.5).

Definition 15. The $\text{Signature}_{\text{AggFin}}$ is the BLS signature scheme. In particular:

- $\text{SIGNKEYS}_{\text{AggFin}} \coloneqq \text{SIGNKEYS}_{\text{BLS}}$,

- $\text{VERIFICATIONKEYS}_{\text{AggFin}} \coloneqq \text{VERIFICATIONKEYS}_{\text{BLS}}$,

- $\text{SIGNATURES}_{\text{AggFin}} \coloneqq \text{SIGNATURES}_{\text{BLS}}$,

- $\text{KeyGen}_{\text{AggFin}} \coloneqq \text{KeyGen}_{\text{BLS}}$, $\text{Sign}_{\text{AggFin}} \coloneqq \text{Sign}_{\text{BLS}}$, $\text{Verify}_{\text{AggFin}} \coloneqq \text{Verify}_{\text{BLS}}$,

- $\text{Aggregation}_{\text{AggFin}} \coloneqq \text{Aggregation}_{\text{BLS}}$,

- $\text{AggregationVerification}_{\text{AggFin}} \coloneqq \text{AggregationVerification}_{\text{BLS}}$,

- $\text{PopProve}_{\text{AggFin}} \coloneqq \text{PopProve}_{\text{BLS}}$,

- $\text{PopVerify}_{\text{AggFin}} \coloneqq \text{PopVerify}_{\text{BLS}}$

❗ Finalization signatures **must be** existentially unforgeable under an adaptive chosen message attack.

5.3 Pointcheval-Sanders Signature Scheme

The *Pointcheval-Sanders signature scheme* (PSS) [PS15] signature scheme is used by identity providers to sign credentials (cf. Section 15.2.1 in Personligt).

⚙ The Pointcheval-Sanders signature scheme will be unsafe in a post-quantum setting, as Shor's algorithm allows computing discrete logarithms efficiently. The current finalists of NIST's post-quantum cryptography standardization [NIS17] do not provide the blind signature property we need. There exist some post-quantum constructions [Rüc08; Ove09; PSM17] based on different assumptions. However, no de facto standard construction has emerged yet. In the short term, one could consider adding Lamport signatures [Lam79] (a type of one-time signature) as a backup to ease the transition to a post-quantum signature scheme.

5.3.1 Algebraic Parameter

This scheme requires a Type 3 pairing and uses the pairing e with groups $(\mathbb{G}_1^{\mathsf{BLS}}, \mathbb{G}_2^{\mathsf{BLS}}, \mathbb{G}_T^{\mathsf{BLS}})$ on the BLS12-381 curve (see Definition 7 in Section 2.2.4). We denote by $\mathbb{F}_{\mathsf{BLS}}$ a field of the same prime-order q as $\mathbb{G}_1^{\mathsf{BLS}}$.

5.3.2 Basic Signature Scheme

The basic Pointcheval-Sanders scheme for messages consisting of ℓ field elements is defined as follows.

The *key-generation* function $\mathsf{BasicKeyGen}_{\mathsf{PS}}$ allows creating key pairs.

Function $\mathsf{BasicKeyGen}_{\mathsf{PS}}$

Outputs
  $\mathsf{sk}^{\mathsf{PS}} \in \mathbb{F}_{\mathsf{BLS}}{}^{\ell+1}$
      The *signing key*.
  $\mathsf{vk}^{\mathsf{PS}} \in (\mathbb{G}_2^{\mathsf{BLS}})^{\ell+1}$
      The *verification key*.

Implementation
  1: Choose random $x, \{y_i\}_{i=1,\dots,\ell} \leftarrow \mathbb{F}_{\mathsf{BLS}}$.
  2: Set $\widetilde{X} = (g_{\mathsf{BLS},2})^x$ and $\widetilde{Y}_i = (g_{\mathsf{BLS},2})^{y_i}$ for $i = 1, \dots, \ell$.
  3: **return** $\mathsf{sk}^{\mathsf{PS}} = (x, \{y_i\}_{i=1,\dots,\ell})$ and $\mathsf{vk}^{\mathsf{PS}} = (\widetilde{X}, \{\widetilde{Y}_i\}_{i=1,\dots,\ell})$.

The *sign* function $\mathsf{BasicSign}_{\mathsf{PS}}$ allows to sign a message.

Function $\mathsf{BasicSign}_{\mathsf{PS}}$

Inputs
  $\mathsf{sk}^{\mathsf{PS}} \in \mathbb{F}_{\mathsf{BLS}}{}^{\ell+1}$
      The signing key.
  $\vec{m} \in \mathbb{F}_{\mathsf{BLS}}{}^{\ell}$
      The message vector to sign.

Outputs
  $\sigma^{\mathsf{PS}} \in (\mathbb{G}_1^{\mathsf{BLS}})^2$
      The *signature* of $\vec{m}$ under $\mathsf{sk}^{\mathsf{PS}}$.

Implementation

1: Let $\mathtt{sk}^{\mathtt{PS}} = (x, \{y_i\}_{i=1,\ldots,\ell})$ and $\vec{m} = (m_1, \ldots, m_\ell)$.
2: Choose random $\alpha \leftarrow \mathbb{F}_{\mathrm{BLS}}{}^\times$.
3: Set $h = (g_{\mathrm{BLS},1})^\alpha$
4: Compute $b = h^{x + \sum_{i=1}^{\ell} y_i m_i}$
5: **return** $\sigma^{\mathtt{PS}} := (h, b)$.

The *verification* function $\mathsf{BasicVerify}_{\mathsf{PS}}$ allows to verify a signature.

Function $\mathsf{BasicVerify}_{\mathsf{PS}}$

Inputs

$\mathtt{vk}^{\mathtt{PS}} \in (\mathbb{G}_2^{\mathrm{BLS}})^{\ell+1}$
    The verification key.

$\vec{m} \in \mathbb{F}_{\mathrm{BLS}}{}^\ell$
    The message vector to sign.

$\sigma^{\mathtt{PS}} \in (\mathbb{G}_1^{\mathrm{BLS}})^2$
    The signature.

Outputs

$\mathtt{b} \in \mathrm{BOOL}$
    Indicates if $\sigma^{\mathtt{PS}}$ is a valid signature

Implementation

1: Let $\mathtt{vk}^{\mathtt{PS}} = (\widetilde{X}, \{\widetilde{Y_i}\}_{i=1,\ldots,\ell})$, $\vec{m} = (m_1, \ldots, m_\ell)$, and $\sigma^{\mathtt{PS}} = (a, b)$.
2: Check that $a \neq 1_{\mathbb{G}_1^{\mathrm{BLS}}}$.
3: Check that $e\big(a, \widetilde{X} \cdot \prod_{i=1}^{\ell} \widetilde{Y_i}^{m_i}\big) = e(b, g_{\mathrm{BLS},2})$
4: **return** **true** if all checks passed and **false** otherwise.

5.3.3 Signature Scheme For Partially Unknown Messages

The Pointcheval-Sanders signature scheme can be extended to allow the signing of messages that are unknown to the signer. The user that is to receive the signature sends a commitment to the message, and proves he knows how to open it. The signer can then sign using only the commitment.

In our implementation, we use a hybrid-version where the signer knows the first part of the message vector. We describe the scheme in the following.

Key generation. The basic public-keys are extended and contain ℓ additional $\mathbb{G}_1^{\mathrm{BLS}}$ elements. The *key-generation* protocol is defined as follows.

Function $\mathsf{KeyGen}_{\mathsf{PS}}$

Outputs

$\mathtt{sk}^{\mathtt{PS}} \in \mathrm{SIGNKEYS}_{\mathsf{PS}} := \mathbb{F}_{\mathrm{BLS}}{}^{\ell+1}$
    The *signing key*.

$\mathsf{vk}^{\mathsf{PS}} \in \mathrm{VERIFICATIONKEYS}_{\mathsf{PS}} := (\mathbb{G}_2^{\mathsf{BLS}})^{\ell+1} \times (\mathbb{G}_1^{\mathsf{BLS}})^{\ell}$

    The *verification key.*

Implementation

1: Choose random $x, \{y_i\}_{i=1,\ldots,\ell} \leftarrow \mathbb{F}_{\mathsf{BLS}}$.
2: Set $\widetilde{X} = (g_{\mathsf{BLS},2})^x$, $Y_i = (g_{\mathsf{BLS},1})^{y_i}$, and $\widetilde{Y}_i = (g_{\mathsf{BLS},2})^{y_i}$ for $i = 1,\ldots,\ell$.
3: **return** $\mathsf{sk}^{\mathsf{PS}} = (x, \{y_i\}_{i=1,\ldots,\ell})$ and $\mathsf{vk}^{\mathsf{PS}} = (\widetilde{X}, \{\widetilde{Y}_i, Y_i\}_{i=1,\ldots,\ell})$.

Signing. The user wants signatures on the message vector $(m_1,\ldots,m_\ell)$, where the first part $(m_1,\ldots,m_{k_1})$ is unknown to the signer and should in particular not be revealed to the signer. Roughly, the *signing protocol* works as follows. The user creates a commitment to the first part of the message vector and generates a non-interactive zero-knowledge proof showing that they can open the commitment. This prevents the user from getting signatures on messages they don't know. The user then sends the commitment, the NIZK proof, and (if needed) the second part of the message vector to the signer. The signer verifies the proof and if it is valid, creates the signature which is then sent back to the user.

As in the basic variant a PS signature consists of two elements.

Definition 16. A *Pointcheval-Sanders signature* is a tuple $(a,b) \in \mathrm{SIGNATURES}_{\mathsf{PS}} := (\mathbb{G}_1^{\mathsf{BLS}})^2$.

The signer uses the following function for signature requests. It takes as input a commitment to the unknown message vector, a NIZK proof, and the known message vector. It produces a "blinded" signature, in the sense that the real signature can be computed from it using the commitment randomness.

Function $\mathsf{Sign}_{\mathsf{PS}}$

Inputs

$\mathsf{sk}^{\mathsf{PS}} \in \mathrm{SIGNKEYS}_{\mathsf{PS}}$

    The signing key, this implicitly defines the verification key $\mathsf{vk}^{\mathsf{PS}}$.

$(m_{k_1+1},\ldots,m_\ell) \in \mathbb{F}_{\mathsf{BLS}}{}^{\ell-k_1}$

    The known part of the message vector $\vec{m}$.

$M' \in \mathbb{G}_1^{\mathsf{BLS}}$

    The commitment to the unknown part of the message vector $\vec{m}$ (see Section 8.1.4).

$\pi \in \{0,1\}^*$

    The non-interactive $\mathsf{agg\text{-}dlog}$ proof for M' (see Section 9.2.2).

Outputs

$\sigma' \in \mathrm{SIGNATURES}_{\mathsf{PS}}$

    The *blinded signature* of $\vec{m}$ under $\mathsf{sk}^{\mathsf{PS}}$.

Implementation

1: Let $\mathsf{sk}^{\mathsf{PS}} = (x, \{y_i\}_{i=1,\ldots,\ell})$ and $\mathsf{vk}^{\mathsf{PS}} = (\widetilde{X}, \{\widetilde{Y}_i, Y_i\}_{i=1,\ldots,\ell})$.
2: **return** $\perp$ if π is *not* a valid NIZK proof for

$$\mathrm{PK}\left\{(m_0,\ldots,m_{k_1}) : M' = (g_{\mathsf{BLS},1})^{m_0} \prod_{i=1}^{k_1} Y_i^{m_i}\right\}.$$

3: Compute $M = M' \prod_{i=k_1+1}^{\ell} Y_i^{m_i}$.
4: Choose random $\alpha \leftarrow \mathbb{F}_{\texttt{BLS}}$ and set $h := (g_{\texttt{BLS},1})^\alpha$.
5: **return** $\sigma' = (a,b) := (h, h^x \cdot M^\alpha)$.

The two-party protocol $\mathsf{SignPartUnknownMessage}_{\mathsf{PS}}$ between a signer and a user allows the user to receive a signature on message vector $(m_1, \ldots, m_\ell) \in \mathbb{F}_{\texttt{BLS}}^\ell$ where $m_1, \ldots m_{k_1}$ are unknown to the signer.

Protocol $\mathsf{SignPartUnknownMessage}_{\mathsf{PS}}$

Let $\mathtt{vk}^{\mathsf{PS}} = (\widetilde{X}, \{\widetilde{Y}_i, Y_i\}_{i=1,\ldots,\ell})$.

User$(\mathtt{m}_0, (m_1, \ldots, m_\ell), \mathtt{vk}^{\mathsf{PS}})$	**Signer**$(\mathtt{sk}^{\mathsf{PS}})$

Compute commitment
$M' = (g_{\texttt{BLS},1})^{\mathtt{m}_0} \prod_{i=1}^{k_1} Y_i^{m_i}$
and agg-dlog NIZK π for
M'.

$$\xrightarrow{\quad M', \pi, \vec{m}' = (m_{k_1+1}, \ldots, m_\ell) \quad}$$

Compute $\sigma' \leftarrow$
$\mathsf{Sign}_{\mathsf{PS}}(\mathtt{sk}^{\mathsf{PS}}, \vec{m}', M', \pi)$.

$$\xleftarrow{\qquad\qquad \sigma' \qquad\qquad}$$

If $\sigma' = (a,b) \neq \perp$ return
$\sigma^{\mathsf{PS}} = (a, ba^{-\mathtt{m}_0})$.

⚠ The privacy of the signing process requires that $\mathtt{m}_0$ is chosen uniformly at random from $\mathbb{F}_{\texttt{BLS}}$ by the user and kept secret. We provide it as an explicit input to allow the user to generate it from a random seed phrase, cf. Section 18.

❗ The NIZK π ensures that the user knows the full message vector. This is useful, e.g., it prevents the user to act as a (blind) intermediary when registering at the IDP (see Section 15.2.1). Note that the commitment opening proof could also be done in an interactive manner if required. Depending on the application the proof π might be outsourced to the overarching protocol or might prove additional statements that are relevant in the specific use case.

ℹ The choice of k_1 depends on the use case. If the protocol is used with a fixed k_1, the verification key only needs to contain the Y_i up to k_1. Note that for $k_1 = 0$, the $\mathsf{Sign}_{\mathsf{PS}}$ corresponds essentially to $\mathsf{BasicSign}_{\mathsf{PS}}$. On the other hand for $k_1 = \ell$, one obtains a protocol for signing a completely secret message vector.

Verification. The *verification* is the same as in the basic scheme. We note however that the interface of the verification function is different, since it uses the extended public-key.

Function Verify$_{\text{PS}}$

Inputs

$\text{vk}^{\text{PS}} \in \text{VERIFICATIONKEYS}_{\text{PS}}$
    The verification key.

$\vec{m} \in \mathbb{F}_{\text{BLS}}{}^{\ell}$
    The message vector to sign.

$\sigma^{\text{PS}} \in \text{SIGNATURES}_{\text{PS}}$
    The signature.

Outputs

$\text{b} \in \text{BOOL}$
    Indicates if σ^{PS} is a valid signature

Implementation

1: Let $\text{vk}^{\text{PS}} = (\widetilde{X}, \left\{\widetilde{Y}_i, Y_i\right\}_{i=1,\dots,\ell})$, $\vec{m} = (m_1, \dots, m_\ell)$, and $\sigma^{\text{PS}} = (a, b)$.
2: Check that $a \neq 1_{\mathbb{G}_1^{\text{BLS}}}$.
3: Check that $e\big(a, \widetilde{X} \cdot \prod_{i=1}^{\ell} \widetilde{Y}_i{}^{m_i}\big) = e(b, g_{\text{BLS},2})$
4: **return true** if all checks passed and **false** otherwise.

That is $\text{Verify}_{\text{PS}} \coloneqq \text{BasicVerify}_{\text{PS}}$. In our use case, the signature is often verified by means of a ZK proof (see Section 5.3.5).

5.3.4 Shared Signing Key

In our application, Pointcheval-Sanders signatures are used by identity providers to create identity certificates. This makes the signing key (of an identity provider) very sensitive. It is therefore advisable to split it into shares which can be stored in k secure enclaves (e.g., in HSMs). In the following, we show how to adapt the functions from Section 5.3.3 to this shared signing key setting. Changes are indicated by purple background .

The key-generation $\text{SharedKeyGen}_{\text{PS}}$ generates key-shares instead of a single signing key. In particular, the key-shares are generated within the enclaves.

Function SharedKeyGen$_{\text{PS}}$

Outputs

$\text{sk}_1^{\text{PS}}, \dots, \text{sk}_k^{\text{PS}} \in \mathbb{F}_{\text{BLS}}{}^{\ell+1}$
    The *signing key shares* stored in the enclaves.

$\text{vk}^{\text{PS}} \in \text{VERIFICATIONKEYS}_{\text{PS}}$
    The *verification key*.

Implementation

1: **for all** $j \in 1, \ldots, k$ **do**
2:      Choose random $x_j, \{y_{i,j}\}_{i=1,\ldots,\ell} \leftarrow \mathbb{F}_{\text{BLS}}$.
3:      **Store** key share $\mathsf{sk}_j^{\text{PS}} = (x_j, \{y_{i,j}\}_{i=1,\ldots,\ell})$ in enclave j.
4:      **Output** $\widetilde{X}_j = (g_{\text{BLS},2})^{x_j}$, $Y_{i,j} = (g_{\text{BLS},1})^{y_{i,j}}$, and $\widetilde{Y}_{i,j} = (g_{\text{BLS},2})^{y_{i,j}}$.
5: Set $\widetilde{X} = \prod_{j=1}^{k} \widetilde{X}_j$, $Y_i = \prod_{j=1}^{k} Y_{i,j}$, and $\widetilde{Y}_i = \prod_{j=1}^{k} \widetilde{Y}_{i,j}$ for $i = 1, \ldots, \ell$.
6: **Return** verification key $\mathsf{vk}^{\text{PS}} = (\widetilde{X}, \left\{ Y_i, \widetilde{Y}_i \right\}_{i=1,\ldots,\ell})$.

ℹ️ The overall signing key is sum-shared between the enclaves, that is $x = \sum_{j=1}^{k} x_j$, and $y_i = \sum_{j=1}^{k} y_{i,j}$ for $i = 1, \ldots, \ell$. Note that each key-share has the same form as a normal signing key.

The shared signing function allows to create a signature on a partly unknown message where the signing key is shared.

Function SharedSign$_{\text{PS}}$

Inputs

$\mathsf{sk}_1^{\text{PS}}, \ldots, \mathsf{sk}_k^{\text{PS}} \in \mathbb{F}_{\text{BLS}}^{\ell+1}$
     The signing key, this implicitly defines the verification key vk^{PS}.

$(m_{k_1+1}, \ldots, m_\ell) \in \mathbb{F}_{\text{BLS}}^{\ell-k_1}$
     The known part of the message vector $\vec{m}$.

$M' \in \mathbb{G}_1^{\text{BLS}}$
     The commitment to the unknown part of the message vector $\vec{m}$ (see Section 8.1.4).

$\pi \in \{0,1\}^*$
     The non-interactive `agg-dlog` proof for M' (see Section 9.2.2).

Outputs

$\sigma^{\text{PS}} \in \text{SIGNATURES}_{\text{PS}}$
     The *blinded signature* of $\vec{m}$ under sk^{PS}.

Implementation

1: Let $\mathtt{sk}_j^{\mathtt{PS}} = (x_j, \{y_{i,j}\}_{i=1,\ldots,\ell})$ and $\mathtt{vk}^{\mathtt{PS}} = (\widetilde{X}, \{\widetilde{Y}_i, Y_i\}_{i=1,\ldots,\ell})$.

2: **return** $\perp$ if π is *not* a valid NIZK proof for

$$\mathrm{PK}\left\{(m_0,\ldots,m_{k_1}) : M' = (g_{\text{BLS},1})^{m_0} \prod_{i=1}^{k_1} Y_i^{m_i}\right\}.$$

3: **for all** $j \in 1,\ldots,k$ **do**

4:      Choose random $\alpha_j \leftarrow \mathbb{F}_{\text{BLS}}{}^{\times}$ in enclave j.

5:      **Output** $h_j = (g_{\text{BLS},1})^{\alpha_j}$.

6: Compute $h = \prod_{j=1}^{k} h_j$

7: **for all** $j \in 1,\ldots,k$ **do**

8:      **Input** M and h to enclave j.

9:      **Output** $B_j = h^{x_j} M^{\alpha_j}$.

10: **return** $\sigma' = (a,b) := (h, \prod_{j=1}^{k} B_j)$.

ⓘ In the protocol from Section 5.3.3 the signer outputs $\sigma' = (h, h^x M^\alpha)$. This is also the case here as $\alpha = \sum_{j=1}^{k} \alpha_j$ and

$$\prod_{j=1}^{k} B_j = \left(\prod_{j=1}^{k} h^{x_j}\right)\left(\prod_{j=1}^{k} M^{\alpha_j}\right) = h^x M^\alpha$$

5.3.5 Proof of Knowledge of a Signature with Public Values

Let $m = (m_1,\ldots,m_\ell)$ be a message and let $J \subseteq \{1,\ldots,\ell\}$ be such that (j, m_j) for $j \in J$ are public. In this section, we describe how a party can prove that they have a signature $\sigma^{\mathtt{PS}} = (a,b)$ on m with respect to the verification key $\mathtt{vk}^{\mathtt{PS}} = \left(\widetilde{X}, \{Y_i, \widetilde{Y}_i\}_{i=1,\ldots,\ell}\right)$. More specifically, they want to prove knowledge of the signature $\sigma^{\mathtt{PS}}$ and the non-public messages $(m_j)_{j \in \overline{J}}$ without revealing either of the two.

This proof is used in the account creation process in the special case where $J = \varnothing$. Revealing either the signature or the message would allow gaining information on the account holder (e.g., by linking accounts). The proof can be used as part of a larger ZK proof.

Preparation. The prover first computes a blinded version of $\sigma^{\mathtt{PS}}$ by choosing random $r, r' \leftarrow \mathbb{F}_{\text{BLS}}$ and setting $(\hat{a}, \hat{b}) = (a^r, (b \cdot a^{r'})^r)$. The blinded signature $(\hat{a}, \hat{b})$ is sent to the verifier. Both the prover and verifier locally compute the public inputs from this, which are defined lower down.

❗ The values r and r' need to be nonzero for this to work. However, by choosing them uniformly at random, it is very likely that this requirement is met.

Inputs. The proof has the following inputs:

Implicit Public Input: Pairing e on $(\mathbb{G}_1^{\text{BLS}}, \mathbb{G}_2^{\text{BLS}}, \mathbb{G}_T^{\text{BLS}})$ and generators $g_{\text{BLS},1}, g_{\text{BLS},2}$.

Public Input: $v_1 = e(\hat{a}, g_{\text{BLS},2})$, $v_2 = e(\hat{b}, g_{\text{BLS},2})$, $v_3 = e(\hat{a}, \widetilde{X} \cdot \prod_{j \in J} \widetilde{Y_j}^{m_j})$, $\forall i \in \overline{J}$ $u_i = e(\hat{a}, \widetilde{Y_i})$.

The public inputs must be computed from $(\hat{a}, \hat{b})$ and $(m_j)_{j \in J}$ for both proof generation and verification.

Witness: $(i, m_i)_{i \in \overline{J}}$, r'

Statement. The formal proof statement is

$$\mathrm{PK}\left\{\left(r', (m_i)_{i \in \overline{J}}\right) : v_2 = v_3 \cdot v_1^{r'} \prod_{i \in \overline{J}} u_i^{m_i}\right\}.$$

Construction. This statement can be proven using the Σ-protocol agg-dlog from Section 9.2.2. In the following we sketch the correctness, soundness, zero-knowledge and witness extractability properties of this construction.

Correctness means that for any valid $\sigma^{\text{PS}} = (a, b)$ on m, the above equation holds. Observe that $(\hat{a}, \hat{b})$ is a valid signature on the message vector for the secret key $x + r', y_1, \ldots, y_\ell$. From the signature verification equation for $(\hat{a}, \hat{b})$, see Section 5.3.2, we can derive the following equation, which proves correctness.

Lemma 17.

$$v_2 = v_3 \cdot v_1^{r'} \prod_{i \in \overline{J}} u_i^{m_i}. \tag{5.1}$$

We prove Lemma 17 further down.

Soundness means that it is only possible to provide such a proof if $\sigma^{\text{PS}} = (a, b)$ is a valid signature on m. This follows from witness extractability, so it is sufficient to prove the latter.

Witness extractability is what is needed to show that a proof is a proof of knowledge: there exists an efficient *extractor*, which can extract a witness from the prover (e.g., by rewinding it, and running it a second time with the same randomness but a different challenge). To construct an extractor for a valid signature and the complete message, we first run the extractor of the underlying Σ-protocol agg-dlog. This produces $(i, m_i)_{i \in \overline{J}}$ and r'. We then use r' to compute the signature $(a^r, b^r) = (\hat{a}, \hat{b} \cdot \hat{a}^{-r'})$, which is a valid Pointcheval-Sanders signature on m.

Finally, the zero-knowledge property follows directly from the zero-knowledge of the underlying Σ-protocol agg-dlog: the simulator can pick random $(\hat{a}, \hat{b})$, since this corresponds to random r and r', and run the simulator of the Σ-protocol agg-dlog.

We now provide the proof of Lemma 17.

Proof of Lemma 17. Recall that $e(x^i, y^j) = e(x, y)^{ij}$ and let us use the shorthand $e_{\hat{a},g} = e(\hat{a}, g_{\text{BLS},2}) = e(a, g_{\text{BLS},2})^r = e(g_{\text{BLS},1}, g_{\text{BLS},2})^{\alpha r}$. Looking at the left-hand side,

$$v_2 = e(\hat{b}, g_{\text{BLS},2}) = e((b \cdot a^{r'})^r, g_{\text{BLS},2}) = e((a^{x + \sum_{i=1}^{ell} y_i m_i} \cdot a^{r'})^r, g_{\text{BLS},2})$$

$$= e(a^r, g_{\text{BLS},2})^{x + r' + \sum_{i=1}^{\ell} y_i m_i} = e_{\hat{a},g}^{x + r' + \sum_{i=1}^{\ell} y_i m_i}.$$

Next, the different parts of the right-hand side,

$$v_3 = e(\hat{a}, \widetilde{X} \cdot \prod_{j \in J} \widetilde{Y_j}^{m_j}) = e(\hat{a}, g_{\text{BLS},2}^x \cdot \prod_{j \in J} g_{\text{BLS},2}^{y_j m_j}) = e(\hat{a}, g_{\text{BLS},2}^{x + \sum_{j \in J} y_j m_j}) = e_{\hat{a},g}^{x + \sum_{j \in J} y_j m_j},$$

$$v_1^{r'} = e(\hat{a}, g_{\text{BLS},2})^{r'} = e_{\hat{a},g}^{r'},$$

$$\prod_{i \in \overline{J}} u_i^{m_i} = \prod_{i \in \overline{J}} e(\hat{a}, \widetilde{Y_i})^{m_i} = \prod_{i \in \overline{J}} e(\hat{a}, g_{\text{BLS},2})^{y_i m_i} = \prod_{i \in \overline{J}} e_{\hat{a},g}^{y_i m_i} = e_{\hat{a},g}^{\sum_{i \in \overline{J}} y_i m_i} = e_{\hat{a},g}^{\sum_{i \in \overline{J}} y_i m_i}.$$

Since the sum of the exponents on the right is the same as on the left, this concludes the proof. $\square$

5.3.6 Use Cases

Identity-provider signature scheme. Personligt requires a signature scheme $\mathsf{Signature}_{\text{ID}}$ for identity providers.

Definition 18. The $\mathsf{Signature}_{\text{ID}}$ is the Pointcheval-Sanders signature scheme. In particular:

- $\text{SIGNKEYS}_{\text{ID}} \coloneqq \text{SIGNKEYS}_{\text{PS}}$,
- $\text{VERIFICATIONKEYS}_{\text{ID}} \coloneqq \text{VERIFICATIONKEYS}_{\text{PS}}$,
- $\text{SIGNATURES}_{\text{ID}} \coloneqq \text{SIGNATURES}_{\text{PS}}$,
- $\mathsf{KeyGen}_{\text{ID}} \coloneqq \mathsf{KeyGen}_{\text{PS}}$, $\mathsf{Sign}_{\text{ID}} \coloneqq \mathsf{Sign}_{\text{PS}}$, $\mathsf{Verify}_{\text{ID}} \coloneqq \mathsf{Verify}_{\text{PS}}$,
- $\mathsf{SignPartUnknownMessage}_{\text{ID}} \coloneqq \mathsf{SignPartUnknownMessage}_{\text{PS}}$

If the identity provider wants to store their signature key split up in HSMs, the schemes from Section 5.3.4 can be used.

> ⚠ This scheme must be existentially unforgeable under an adaptive chosen message attack.

Chapter 6

Pseudo Random Functions

A *pseudorandom function* (PRF) is a deterministic function that mimics a truly random function. It takes two inputs: a *secret key* and an input string, and produces an output that appears random to anyone who does not know the key. Crucially, given the same key and input, the PRF will always produce the same output, ensuring determinism. However, without the key, the output is computationally indistinguishable from a truly random value. A more detailed introduction to PRFs can be found in Chapter 3.5.1 of [KL20].

6.1 Algebraic Pseudorandom Function

We use the Dodis et al. [DY05] pseudorandom functions in Personligt and Konto to generate account identifiers. The algebraic nature of the scheme allows for Sigma protocol based proofs of evaluation.

> ⚙ The Dodis et al. PRF will be unsafe in a post-quantum setting, as Shor's algorithm allows for efficient computation of discrete logarithms. A post-quantum secure option for a PRF would be a hash-based message authentication code (HMAC) based on SHA-3. Note that the choice of replacement will depend heavily on the zero-knowledge proof scheme used, as an accompanying proof of evaluation is required.

6.1.1 Implementation

For the implementation, any group $\mathbb{G}$ of prime order q in which computing discrete logarithms is infeasible can be used. Let g be a publicly known generator of $\mathbb{G}$, and let $\mathbb{F}$ be the field of prime order q, i.e., $\mathbb{F} = \mathbb{Z}_q$. The PRF is shown to be pseudorandom for small domains under the *Decisional Diffie-Hellman* inversion assumption.

6.1.2 Interface

Function PRF

Inputs
    $\text{key}_{\text{PRF}} \in \text{PRFKEYS} := \mathbb{F} = \mathbb{Z}_q$
        A *PRF key*
    $x \in \mathbb{F} = \mathbb{Z}_q$
        A bit string (mapped to a field element)

Outputs

$y \in \mathbb{G}$

A group element in $\mathbb{G}$

Implementation

$$\mathsf{PRF}(\mathtt{key}_{\mathsf{PRF}}, x) = g^{\frac{1}{\mathtt{key}_{\mathsf{PRF}} + x}}$$

❗ This PRF is secure only for small domains; inputs that are slightly super-logarithmic in the security parameter. We expect this to be sufficient for our application, as currently we only need to apply it to numbers less than the maximal number of accounts a user can open. However, this is something to keep in mind. We can examine the group size in our instantiations and the constants in the PRF security proof to estimate the number of accounts this will allow. Another alternative is to move to AES plus SNARK in the next version.

</> In the current implementation, the maximal number of accounts per identity credential is limited to 255; that is, the number must fit into one byte. This should be well within the security limits of the PRF.

6.1.3 Use Cases

The PRF is used in the Personligt and Konto layers with $\mathbb{G}_1^{\mathsf{BLS}}$ as the underlying group (see also Section 2.2.4). The choice of group is dictated by other cryptographic parts of the application.

❗ For encrypted transfers (cf. shielded transfers in Section 16.3), the PRF is used to generate public keys such that the PRF key and the input define the secret key. If this feature is used, the generator and the group used here *must* match the one used for encrypted transfers.

6.2 Verifiable Random Function

Verifiable Random Functions (VRFs) are cryptographic tools that produce random outputs along with proofs of their validity. Essentially, a VRF allows someone with a *private key* to generate a random value, and also create a *proof* that this value was indeed generated correctly. Anyone with the corresponding *public key* can then verify that the output is legitimate, meaning it was produced by the holder of the secret key.

Konsensus requires a *verifiable random function* (VRF) for leader election.

⚙ The elliptic curve based VRF will be unsafe in post-quantum setting as Shor's algorithm allows computing discrete logarithms efficiently. Finding a good replacement is an active area of academic research.

6.2.1 Implementation

We follow the elliptic curve VRF (EC-VRF) scheme presented in RFC9381 [Gol+23], originating from [Pap+17]. More precisely, we use the cipher suite `ECVRF-EDWARDS25519-SHA512-TAI` from RFC9381. The scheme uses e with groups $(\mathbb{G}_1^{\mathtt{BLS}}, \mathbb{G}_2^{\mathtt{BLS}}, \mathbb{G}_T^{\mathtt{BLS}})$ on the `BLS12-381` curve (see Definition 7 in Section 2.2.4).

> ⚠ We are in a setting with untrusted keys. This means that all keys must be verified using VRFVerifyKey before they are first used. It must further be ensured that the curve and the generator of the subgroup are obtained from a trusted source. See Section 7.1 in [Gol+23] and Appendix B.0.3 in [Pap+17].

6.2.2 Interface

The VRF scheme consists of the following algorithms.

The *key-generation* function VRFKeyGen allows to generate a key pair.

Function VRFKeyGen

Outputs
 $\mathtt{sk}^{\mathtt{VRF}} \in \mathrm{PRIVATEKEY}_{\mathtt{VRF}} \coloneqq \{0,1\}^{256}$
  The *VRF private key.*
 $\mathtt{pk}^{\mathtt{VRF}} \in \mathrm{PUBLICKEY}_{\mathtt{VRF}} \coloneqq \mathbb{G}^{\mathtt{Ed}}$
  The *VRF public key.*

The *VRF-hash* function VRFHash generates a VRF hash.

Function VRFHash

Inputs
 $\mathtt{sk}^{\mathtt{VRF}} \in \mathrm{PRIVATEKEY}_{\mathtt{VRF}}$
  The VRF private key.
 $\mathtt{str} \in \{0,1\}^*$
  The VRF input bit string.

Outputs
 $h \in \mathrm{HASH}_{\mathtt{VRF}}$
  The *VRF hash* for `str`.

We denote by ℓ_{VRF} the length of a VRF hash in bits.

The *VRF-prove* function VRFProve generates a VRF proof.

Function VRFProve

Inputs
 $\mathtt{sk}^{\mathtt{VRF}} \in \mathrm{PRIVATEKEY}_{\mathtt{VRF}}$
  The VRF secret key.

> str ∈ $\{0, 1\}^*$
>> The VRF input bit string.

Outputs

> p ∈ PROOF$_{\mathsf{VRF}}$
>> The *VRF proof* for str.

The *VRF-proofToHash* function VRFproof2hash generates a VRF hash from a VRF proof.

Function VRFproof2hash

Inputs

> p ∈ PROOF$_{\mathsf{VRF}}$
>> A *VRF proof*.

Outputs

> h ∈ HASH$_{\mathsf{VRF}}$
>> The VRF hash for p.

Promise

> It holds that $\mathsf{VRFHash}(\mathrm{sk}^{\mathsf{VRF}}, \mathrm{str}) = \mathsf{VRFproof2hash}(\mathsf{VRFProve}(\mathrm{sk}^{\mathsf{VRF}}, \mathrm{str}))$.

The *VRF-verification* function VRFVerify allows to check a VRF proof.

Function VRFVerify

Inputs

> $\mathrm{pk}^{\mathsf{VRF}}$ ∈ PUBLICKEY$_{\mathsf{VRF}}$
>> The VRF public key.
> str ∈ $\{0, 1\}^*$
>> The input bit string.
> p ∈ PROOF$_{\mathsf{VRF}}$
>> The VRF proof.

Outputs

> b ∈ BOOL
>> Indicates if p is a valid VRF proof.

> ⚠ We here slightly deviate from the draft [Gol+20]: In our description, VRFVerify only returns whether the proof is valid or not, the actual hash value can then be extracted using VRFproof2hash afterwards. In the draft, VRFVerify directly outputs the extracted hash for valid proofs. We prefer to keep this separate for the description, but it is also fine to implement it as in the draft.

The *VRF-keyVerification* function VRFVerifyKey allows checking if a given bit string represent a valid public key.

Function VRFVerifyKey

Inputs

    $\mathtt{str} \in \{0, 1\}^*$

        The input bit string.

Outputs

    $\mathtt{b} \in \mathrm{BOOL}$

        Indicates if $\mathtt{str}$ represents a valid VRF public key.

Promise

        This function returns $\mathtt{true}$ if and only if $\mathtt{str}$ represents a VRF public key $\mathtt{pk}^{\mathtt{VRF}} \in \mathrm{PUBLICKEY}_{\mathtt{VRF}}$.

6.2.3 Security Requirements

Apart from standard security properties, the following security properties, as discussed in [Gol+23], are required for our application in Konsensus.

Full uniqueness: For a fixed key pair $(\mathtt{pk}^{\mathtt{VRF}}, \mathtt{sk}^{\mathtt{VRF}})$ and any message m, a (computationally bounded) adversary cannot produce two valid VRF hashes for m. This property must hold even if the key pair was generated maliciously. This property prevents grinding attacks in the leader election process.

Unpredictability under Malicious Key Generation: For a fixed key pair $(\mathtt{pk}^{\mathtt{VRF}}, \mathtt{sk}^{\mathtt{VRF}})$ and for an "unpredictable" message m (i.e. with high entropy), the output of VRFHash should be indistinguishable from uniform. This must hold even if the key pair was generated maliciously. This property ensures that the leader election is random.

Chapter 7

Public-Key Encryption

Public key encryption, also known as asymmetric encryption, employs a pair of keys: a public *encryption key*, also known as public key, and a private *decryption key*, also known as private key. The encryption key can be shared with anyone, while the decryption key is kept secret. When someone encrypts data using a recipient's encryption key, only that recipient's corresponding decryption key can decrypt it.

7.1 ElGamal Encryption Scheme

The Konto and Personligt layers require a public-key encryption scheme for identity disclosure and for shielded transfers. We use the *ElGamal encryption scheme* [Elg85].

> ⚛ The ElGamal encryption will be unsafe in post-quantum setting as Shor's algorithm allows computing discrete logarithms efficiently. Note that ElGamal is vulnerable to harvest attacks, i.e. an attacker can collect ciphertexts now and decrypt them later when quantum computers are available. At the moment there seems to be no post-quantum scheme that has all the nice properties of ElGamal. Finding a good replacement will heavily depend on the exact use case.

7.1.1 Implementation

The ElGamal encryption scheme is presented in [Elg85]. The scheme is defined over the prime-order q group $\mathbb{G}$. Denote by g a publicly known generator of $\mathbb{G}$ and let $\mathbb{F}$ be the field of prime-order q, i.e., $\mathbb{F} = \mathbb{Z}_q$.

> ⚠ The *Decisional Diffie-Hellman* (DDH) problem *must* be hard within group $\mathbb{G}$.

The choice of $\mathbb{G}$ and generator g depends on the application. Both $\mathbb{G}_1^{\mathtt{BLS}}$ and $\mathbb{G}^{\mathtt{Ed}}$ satisfy the requirements. See Section 7.1.6 for more details.

7.1.2 Interface

The *key-generation* function $\mathsf{KeyGen}_{\mathtt{EG}}$ allows creating key pairs.

Function $\mathsf{KeyGen}_{\mathrm{EG}}$

Outputs

  $\mathrm{dk}_{\mathrm{EG}} \in \mathrm{PRIVATEKEYS}_{\mathrm{EG}} := \mathbb{F} = \mathbb{Z}_q$
      A *decryption key*.

  $\mathrm{pk}_{\mathrm{EG}} \in \mathrm{PUBLICKEYS}_{\mathrm{EG}} := \mathbb{G}$
      An *encryption key*.

Implementation

1: Select random $\mathrm{dk}_{\mathrm{EG}} \leftarrow \mathbb{F} = \mathbb{Z}_q$.
2: Set $\mathrm{pk}_{\mathrm{EG}} = g^{\mathrm{dk}_{\mathrm{EG}}}$.
3: **return** $(\mathrm{dk}_{\mathrm{EG}}, \mathrm{pk}_{\mathrm{EG}})$.

❗ Some implementations restrict the key private space to $\mathbb{Z}_q^* = \{1, \ldots, q-1\}$. They consider 0 or even 1 *weak keys* as the encryption under zero does not "hide" the message. However, the same can be said about any fixed key (e.g. 42 or 123456789), thus it is fine to use the full space $\mathbb{Z}_q$ (cf. [Ber+12]). The probability of actually generating a 0 or 1 key is negligible due to the group size.

In practice, one should make sure that the implementation can handle all keys, e.g. one should check that the 0 key does not crash the implementation (in an implementation using an elliptic curve, 0 as private key means the public key is the point at infinity).

The *encryption* function $\mathsf{Enc}_{\mathrm{EG}}$ allows encrypting messages.

Function $\mathsf{Enc}_{\mathrm{EG}}$

Inputs

  $\mathrm{pk}_{\mathrm{EG}} \in \mathrm{PUBLICKEYS}_{\mathrm{EG}} := \mathbb{G}$
      An encryption key.

  $m \in \mathbb{G}$
      The message, an element of group $\mathbb{G}$.

Outputs

  $c \in \mathrm{CIPHERTEXTS}_{\mathrm{EG}} := \mathbb{G}^2$
      The *ciphertext*.

Implementation

1: Select random $r \leftarrow \mathbb{F} = \mathbb{Z}_q$.
2: **return** $c = (g^r, m \cdot \mathrm{pk}_{\mathrm{EG}}^r)$.

ℹ If the message is a k-bit string instead of group element, an additional injective, efficiently computable function mapping k-bit strings to group elements is needed. For decryption the inverse function needs to be efficiently computable as well. See also, the encryption in the exponent in Section 7.1.4.

The *decryption* function $\mathsf{Dec}_{\mathrm{EG}}$ allows decrypting messages.

Function Dec_{EG}

Inputs

$\quad \text{dk}_{\text{EG}} \in \text{PRIVATEKEYS}_{\text{EG}} := \mathbb{F} = \mathbb{Z}_q$
$\qquad$ A decryption key.

$\quad c \in \text{CIPHERTEXTS}_{\text{EG}} := \mathbb{G}^2$
$\qquad$ The ciphertext.

Outputs

$\quad m \in \mathbb{G}$
$\qquad$ The message.

Implementation

1: Let $c = (a, b)$.
2: **return** $m = b \cdot a^{-\text{dk}_{\text{EG}}}$.

7.1.3 Homomorphic Operations

ElGamal ciphertexts can be multiplied to obtain a ciphertext for the product of the corresponding messages.

Function $\text{HomMultEnc}_{\text{EG}}$

Inputs

$\quad (c_1, \ldots, c_n) \in \text{CIPHERTEXTS}_{\text{EG}}^n$
$\qquad$ A list of ciphertexts.

Outputs

$\quad c \in \text{CIPHERTEXTS}_{\text{EG}}$
$\qquad$ A ciphertext for the product of the messages encrypted in the c_i.

Promise

$\qquad$ All c_i are encryptions under the same public key.

Implementation

1: let $(c_{i,1}, c_{i,2}) := c_i$ for $i = 1, \ldots, n$.
2: **return** $c := \left(\prod_{i=1}^n c_{i,1}, \prod_{i=1}^n c_{i,2} \right)$.

> **ⓘ** Note that if $c_i = (g^{r_i}, m_i \cdot \text{pk}_{\text{EG}}^{r_i})$, we have
>
> $$c = \left(\prod_{i=1}^n c_{i,1}, \prod_{i=1}^n c_{i,2} \right) = \left(g^{\sum_{i=1}^n r_i}, \prod_{i=1}^n m_i \cdot \text{pk}_{\text{EG}}^{\sum_{i=1}^n r_i} \right).$$
>
> That is, c is an encryption of $\prod_{i=1}^n m_i$ under pk_{EG} with randomness $\sum_{i=1}^n r_i$.

By repeated applications of homomorphic multiplication, one can further raise ciphertexts to a given power.

Function HomPowEnc$_\text{EG}$

Inputs

$c \in \text{CIPHERTEXTS}_\text{EG}$
  A ciphertext.

$n \in \mathbb{N}^+$
  The power to which the message in c should be raised.

Outputs

$c' \in \text{CIPHERTEXTS}_\text{EG}$
  A ciphertext for m^n, where m is the message encrypted in c.

Implementation

1: $c' := c$
2: **for** $i = 1, \ldots, n - 1$ **do**
3: $\quad\lfloor \quad c' := \text{HomMultEnc}_\text{EG}(c', c)$
4: **return** c'.

</> This method can be implemented more efficiently using square-and-multiply.

7.1.4 Encryption of Data in the Exponent

ElGamal encryption can be adapted to encrypt messages $x \in \mathbb{F}$. This adaptation involves encoding the message in the exponent, effectively encryption h^x for some generator h. This approach is particularly interesting for use in ZK proofs.

However, a challenge arises from encoding the message in the exponent. Standard ElGamal decryption Dec_EG leads to h^x and not the original message x. Recovering x requires a brute-force search. This is only feasible for small x; otherwise one could easily decrypt ciphertexts by computing r from g^r.

The solution for arbitrary x is to split x into small pieces such that for each piece one can brute-force the discrete logarithm. For this we first split x (as a bit string) in $\mathtt{t}$ parts $x = x_1 || x_2 || \ldots || x_\mathtt{t}$ of size $\mathtt{s}$ (in bits), i.e., $x = \sum_{i=1}^{\mathtt{t}} 2^{\mathtt{s} \cdot (i-1)} x_i$. We define *encryption* with *data in the exponent* as

$$\text{Enc}_\text{EG}^{\mathtt{t},\mathtt{s}}(\text{pk}_\text{EG}, x) := \left\{ \text{Enc}_\text{EG}(\text{pk}_\text{EG}, h^{x_i}) \right\}_{i=1,\ldots,\mathtt{t}}.$$

For the decryption $\text{Dec}_\text{EG}^{\mathtt{t},\mathtt{s}}(\text{dk}_\text{EG}, (c_1, \ldots, c_\mathtt{t}))$, each of the $\mathtt{t}$ parts is decrypted using regular decryption Dec_EG as defined above. This leads to the set $\{h^{x_i}\}_{i=1,\ldots,\mathtt{t}}$. Now brute-force search is used to get each of the x_i's. This is feasible assuming each part is small enough. Finally, x is computed from the parts as $x = \sum_{i=1}^{\mathtt{t}} 2^{\mathtt{s} \cdot (i-1)} x_i$.

ⓘ If each part is of size $\mathtt{s} = 64$ bits, then each part x_i can be determined in time about $2^{64/2} = 2^{32}$ operations, using the so-called *baby-step giant-step algorithm*. The size of parts depends on the application and must be specified. The size of parts is directly correlated to the number of parts $\mathtt{t}$.

> ℹ️ Normally, one chooses $h = g$. However, for some ZK proofs it can be useful to use a h such that no one knows its discrete logarithm with respect to g and vice-versa. This implies that the same should hold for h and $\mathtt{pk}_{\mathrm{EG}} = g^{\mathtt{dk}_{\mathrm{EG}}}$. This makes $(h, \mathtt{pk}_{\mathrm{EG}})$ a Pedersen commitment key and makes the second part of the ciphertext a Pedersen commitment. See Section 8.1 for more details on Pedersen commitments. Thus, the generator h depends on the use case and must be specified.

Joining data in exponent. Since the x_i are encrypted in the exponent, $\mathsf{HomPowEnc}_{\mathrm{EG}}$ can be used to multiply the x_i by constants, and $\mathsf{HomMultEnc}_{\mathrm{EG}}$ can be used for addition. Since $x = \sum_{i=1}^{\mathtt{t}} 2^{\mathtt{s} \cdot (i-1)} x_i$, we can thus join the data parts to obtain an encryption of the original x in the exponent in a single ciphertext.

Function $\mathsf{JoinEncExp}_{\mathrm{EG}}^{\mathtt{t},\mathtt{s}}$

Inputs

$c = (c_1, \ldots, c_{\mathtt{t}}) \in \textsc{ciphertexts}_{\mathrm{EG}}{}^{\mathtt{t}}$
    A list of ciphertexts obtained by $\mathsf{Enc}_{\mathrm{EG}}^{\mathtt{t},\mathtt{s}}(\mathtt{pk}_{\mathrm{EG}}, x)$

Outputs

$c' \in \textsc{ciphertexts}_{\mathrm{EG}}$
    A single ElGamal ciphertext encrypting h^x under $\mathtt{pk}_{\mathrm{EG}}$.

Implementation

1: $c' := c_1$
2: **for** $i = 2, \ldots, \mathtt{t}$ **do**
3:     $c' := \mathsf{HomMultEnc}_{\mathrm{EG}}\big(c', \mathsf{HomPowEnc}_{\mathrm{EG}}\big(c_i, 2^{\mathtt{s} \cdot (i-1)}\big)\big)$
4: **return** c'.

7.1.5 Shared Encryption

ElGamal encryption can be combined with Shamir's secret sharing from Section 4. The idea is to first share the message and then encrypt each share. We present two variants. In the first the message is a group element, and in the second the message is a field element (similar to encryption with data in the exponent).

> ⚠️ Shared encryption **requires** that $\mathbb{G}$ is of prime-order to ensure that the exponents form a field that can be used for Shamir's secret sharing.

Shared encryption of group elements. The shared encryption of a group element $m = g^x$ is parametrized by the number of shares $\mathtt{n}$ and the sharing threshold $\mathtt{d}$. For each choice of $\mathtt{n}$, we assume $\mathtt{n}$ interpolation points $\alpha_1, \ldots, \alpha_{\mathtt{n}}$. The interpolation points are distinct, nonzero, and publicly-known, and we assume them to be a deterministic function of $\mathbb{F}$ and $\mathtt{n}$.

Definition 19. The *shared encryption* of $m = g^x$ under public-keys $\mathrm{PK}_{\mathrm{PG}} = \left\{ \mathtt{pk}_{\mathrm{PG}_1}, \ldots, \mathtt{pk}_{\mathrm{PG}_{\mathtt{n}}} \right\}$ is defined as

$$\mathsf{Enc}_{\mathrm{PG}}^{\mathtt{n},\mathtt{d}}(\mathrm{PK}_{\mathrm{EG}}, g^x) := \left\{ \mathsf{Enc}_{\mathrm{EG}}(\mathtt{pk}_{\mathrm{EG}_i}, g^{\mathtt{sh}(x)_i}) \right\}_{i=1,\ldots,\mathtt{n}},$$

where shares $\mathtt{sh}(x)_1, \ldots, \mathtt{sh}(x)_{\mathtt{n}}$ are the output of $\mathsf{Share}(\mathtt{n}, \mathtt{d}, (\alpha_1, \ldots, \alpha_{\mathtt{n}}), x)$.

> ⚠ For encryption the exponent x (i.e., the discrete log) of the message $m = g^x$ *must* be known. This is not the case for decryption.

Decryption requires at least $d + 1$ of the n involved keys.

Protocol $\mathsf{Dec}_{\mathrm{EG}}^{\mathtt{n},\mathtt{d}}(\vec{c})$

Let $J \subseteq \{1, \ldots, \mathtt{n}\}$ be an index set of size $d + 1$. We assume that $\vec{c}$ contains c_j for $j \in J$.
1. For all $j \in J$ send c_j to a party knowing $\mathsf{dk}_{\mathrm{EG}_j}$.
2. For all $j \in J$ decrypt c_j using key $\mathsf{dk}_{\mathrm{EG}_j}$, i.e., $m_j = \mathsf{Dec}_{\mathrm{EG}}(\mathsf{dk}_{\mathrm{EG}_j}, c_j)$.
3. Compute $m = \prod_{j \in J} (m_j)^{\ell_j(0)}$ where the ℓ_j's are the Lagrange basis polynomials defined by the interpolation points $\{\alpha_j\}_{j \in J}$ (cf. Definition 10).
4. **Return** m.

> ℹ The shared encryption for group elements can use a generator $\hat{g}$ different from g as base. The encryption becomes $\left\{ \mathsf{Enc}_{\mathrm{EG}}(\mathsf{pk}_{\mathrm{EG}_i}, \hat{g}^{\mathsf{sh}(x)_i}) \right\}_{i=1,\ldots,\mathtt{n}}$ while the decryption stays the same.

Shared encryption of field elements. The *shared encryption* with *data in the exponent* of a field element $x \in \mathbb{F}$ is parametrized by the number of shares $\mathtt{n}$, the sharing threshold $\mathtt{d}$, the number of parts $\mathtt{t}$, and the part size $\mathtt{s}$. The shared encryption of x under public-keys $\mathrm{PK}_{\mathrm{PG}} = \left\{ \mathsf{pk}_{\mathrm{PG}_1}, \ldots, \mathsf{pk}_{\mathrm{PG}_{\mathtt{n}}} \right\}$ is defined as

$$\mathsf{Enc}_{\mathrm{EG}}^{\mathtt{n},\mathtt{d},\mathtt{t},\mathtt{s}}(\mathrm{PK}_{\mathrm{PG}}, x) := \left\{ \mathsf{Enc}_{\mathrm{EG}}^{\mathtt{t},\mathtt{s}}(\mathsf{pk}_{\mathrm{PG}_i}, \mathsf{sh}(x)_i) \right\}_{i=1,\ldots,\mathtt{n}},$$

where shares $\mathsf{sh}(x)_1, \ldots, \mathsf{sh}(x)_{\mathtt{n}}$ are the output of $\mathsf{Share}(\mathtt{n}, \mathtt{d}, (\alpha_1, \ldots, \alpha_{\mathtt{n}}), x)$.

> ❗ Note that $\mathtt{t} \cdot \mathtt{s}$ defines the number of bits that can be used to encode $x \in \mathbb{F}$, and $\mathbb{F}$ is also the field used for Shamir's secret sharing. These values thus need to be chosen accordingly.

Decryption is a protocol that requires participation of parties that hold at least $d + 1$ of the $\mathtt{n}$ involved keys.

Protocol $\mathsf{Dec}_{\mathrm{EG}}^{\mathtt{n},\mathtt{d},\mathtt{t},\mathtt{s}}(\vec{c})$

Let $J \subseteq \{1, \ldots, \mathtt{n}\}$ be an index set of size $d + 1$. We assume that $\vec{c}$ contains $c_j \in \mathrm{CIPHERTEXTS}_{\mathrm{EG}}^{\mathtt{t}}$ for $j \in J$.
1. For all $j \in J$ send c_j to a party knowing $\mathsf{dk}_{\mathrm{EG}_j}$.
2. For all $j \in J$ decrypt c_j using key $\mathsf{dk}_{\mathrm{EG}_j}$, i.e., $\mathsf{sh}(x)_j = \mathsf{Dec}_{\mathrm{EG}}^{\mathtt{t},\mathtt{s}}(\mathsf{dk}_{\mathrm{EG}_j}, c_j)$.
3. Compute $x = \mathsf{Reconstruct}(\mathtt{n}, \mathtt{d}, (\alpha_j)_{j \in J}, (\mathsf{sh}(x)_j)_{j \in J})$
4. **Return** x.

7.1.6 Use Cases

Public-key encryption for identity disclosure. The Personligt layer requires a public-key encryption scheme for identity disclosure. For this we use ElGamal encryption with the above extensions.

Definition 20. The *identity public-key encryption scheme* is the ElGamal encryption scheme with data in the exponent. In particular,

- $\text{SECRETKEYS}_{\text{PG}} := \text{PRIVATEKEYS}_{\text{EG}}$,

- $\text{PUBLICKEYS}_{\text{PG}} := \text{PUBLICKEYS}_{\text{EG}}$,

- $\text{CIPHERTEXTS}_{\text{PG}} := \text{CIPHERTEXTS}_{\text{EG}}$,

- $\text{KeyGen}_{\text{PG}} := \text{KeyGen}_{\text{EG}}$, $\text{Enc}_{\text{PG}} := \text{Enc}_{\text{EG}}$, $\text{Dec}_{\text{PG}} := \text{Dec}_{\text{EG}}$,

- $\text{Enc}_{\text{PG}}^{\text{t,s}} := \text{Enc}_{\text{EG}}^{\text{t,s}}$, $\text{Dec}_{\text{PG}}^{\text{t,s}} := \text{Dec}_{\text{EG}}^{\text{t,s}}$,

- $\text{Enc}_{\text{PG}}^{\text{n,d}} := \text{Enc}_{\text{EG}}^{\text{n,d}}$, $\text{Dec}_{\text{PG}}^{\text{n,d}} := \text{Dec}_{\text{EG}}^{\text{n,d}}$, $\text{Enc}_{\text{PG}}^{\text{n,d,t,s}} := \text{Enc}_{\text{EG}}^{\text{n,d,t,s}}$, and $\text{Dec}_{\text{PG}}^{\text{n,d,t,s}} := \text{Dec}_{\text{EG}}^{\text{n,d,t,s}}$.

The scheme operates over group $\mathbb{G}_1^{\text{BLS}}$ using generators $g := \bar{g}_{\text{com}}$, $h := \bar{h}_{\text{com}}$ from the commitment scheme (cf. Section 8.1.5).

> The part size used in the implementation is $\text{s} := 32$ bits.

Public-key encryption for accounts. The Konto layer requires a public-key encryption scheme for encrypted amounts in accounts. For this we use the encryption with data in the exponents scheme from Section 7.1.4.

Definition 21. The *account public-key encryption scheme* is the ElGamal encryption scheme with data in the exponent. In particular,

- $\text{SECRETKEYS}_{\text{ACC}} := \text{PRIVATEKEYS}_{\text{EG}}$,

- $\text{PUBLICKEYS}_{\text{ACC}} := \text{PUBLICKEYS}_{\text{EG}}$,

- $\text{CIPHERTEXTS}_{\text{ACC}} := \text{CIPHERTEXTS}_{\text{EG}}{}^{\text{t}}$,

- $\text{KeyGen}_{\text{ACC}} := \text{KeyGen}_{\text{EG}}$, $\text{Enc}_{\text{ACC}}^{\text{t,s}} := \text{Enc}_{\text{EG}}^{\text{t,s}}$, and $\text{Dec}_{\text{ACC}}^{\text{t,s}} := \text{Dec}_{\text{EG}}^{\text{t,s}}$.

The scheme operates over group $\mathbb{G}_1^{\text{BLS}}$ using generators $g := \bar{g}_{\text{com}}$, $h := \bar{h}_{\text{com}}$ where $\text{ck} = (\bar{g}_{\text{com}}, \bar{h}_{\text{com}})$ is the global commitment key used for the commitment scheme from Section 8.1.2. The part size is $\text{s} := 32$ bits.

3d 18 4b ba 8e bb e3 fe 40 6d 2d
0a f9 ae f3 df a8 d0 2b 43 30 82
00 fa fb ec 94 c5 f4 73 86 0c f9
31 32 83 8b ca d3 8d 6e 33 3e 43
89 b4 3a ac

Chapter 8

Commitment Schemes

A *commitment scheme* allows a person, the committer, to *commit* to a message m using *randomness* r by computing $c = \mathsf{commit_P}(\mathsf{ck}, m, r)$. The committer can later *open* the commitment by revealing m and r so that any verifier can check the opening.

The commitment scheme has two properties:

Binding The committer cannot open the commitment to two different values.

Hiding A commitment provides no information about the committed value.

8.1 Pedersen Commitment Scheme

The Konto and Personligt layers require a commitment scheme that is *unconditionally hiding* and *computationally binding*. We use the *Pedersen commitment scheme* [Ped92].

> ⚛ The Pedersen commitment scheme is not secure in a post-quantum setting, since Shor's algorithm can be used to compute discrete logarithms efficiently. Because Pedersen commitments are unconditionally (perfectly) hiding, they are not vulnerable to so-called "harvest" attacks that rely on collecting many ciphertexts or commitments: the hiding property still holds. A quantum attacker, however, can break the binding property (i.e., soundness) by computing discrete logarithms. If post-quantum security is required, Pedersen commitments can be replaced by a hash-based or lattice-based commitment scheme. Any replacement must be compatible with the zero-knowledge proof systems we use (including post-quantum variants where needed).

8.1.1 Implementation

We use Pedersen commitments [Ped92] over a cyclic group $\mathbb{G}$ of prime order q. Denote by $\mathbb{F}$ the finite field of order q, i.e., $\mathbb{F} = \mathbb{Z}_q$.

> ⚠ The *commitment key* $\mathsf{ck} = (\bar{g}_{\mathsf{com}}, \bar{h}_{\mathsf{com}}) \in \mathbb{G}^2$ must be generated so that no one (in particular the committer) knows the discrete logarithm of $\bar{h}_{\mathsf{com}}$ with respect to $\bar{g}_{\mathsf{com}}$ or vice versa. See Section 22.1.1 for a secure method to generate a global commitment key.

> ℹ The choice of group $\mathbb{G}$ depends on the use case and must be specified.

8.1.2 Interface

The commitment scheme consists of two functions.

Function $\mathsf{commit_P}$

Inputs

   $\mathsf{ck} \in \textsc{comkeys} := \mathbb{G}^2$
      The *commitment key* which is publicly known.

   $m \in \mathbb{F} = \mathbb{Z}_q$
      The message to commit to.

   $r \in \mathbb{F} = \mathbb{Z}_q$
      The commitment randomness.

Outputs

   $\mathsf{C} \in \mathbb{G}$
      The *commitment*, an element of group $\mathbb{G}$.

Implementation Let $\mathsf{ck} = (\bar{g}_{\mathsf{com}}, \bar{h}_{\mathsf{com}})$. Then

$$\mathsf{C} = \bar{g}_{\mathsf{com}}^m \cdot \bar{h}_{\mathsf{com}}^r.$$

For some cryptographic protocols, e.g., a zero-knowledge proof or a multi-party computation, it is convenient to be able to commit to public values in a deterministic fashion.

Definition 22. The *default commitment* to a public message $m \in \mathbb{F}$ is $\mathsf{C}_m := \mathsf{commit_P}(\mathsf{ck}, m, 0)$, i.e.,

$$\mathsf{C}_m = \bar{g}_{\mathsf{com}}^m \cdot \bar{h}_{\mathsf{com}}^0 = \bar{g}_{\mathsf{com}}^m.$$

Opening a commitment allows to verify that it contains a given value.

Function $\mathsf{checkCom_P}$

Inputs

   $\mathsf{ck} \in \textsc{comkeys}$
      The commitment key which is publicly known.

   $\mathsf{C} \in \mathbb{G}$
      The commitment

   $m \in \mathbb{F} = \mathbb{Z}_q$
      The message.

   $r \in \mathbb{F} = \mathbb{Z}_q$
      The commit randomness.

Outputs

   $b \in \textsc{bool}$
      Returns true iff $\mathsf{C} = \mathsf{commit_P}(\mathsf{ck}, m, r)$

Implementation Let $\mathsf{ck} = (\bar{g}_{\mathsf{com}}, \bar{h}_{\mathsf{com}})$. Then check that

$$\mathsf{C} \overset{!}{=} \bar{g}_{\mathsf{com}}^m \cdot \bar{h}_{\mathsf{com}}^r.$$

8.1.3 Homomorphic Operations

Pedersen commitments are *commitment!homomorphic*homomorphic: the product of two commitments is a commitment to the sum of the underlying values. Concretely, if $C_1 = \mathsf{commit_P}(\mathsf{ck}, v_1, r_1)$ and $C_2 = \mathsf{commit_P}(\mathsf{ck}, v_2, r_2)$, then

$$C_1 C_2 = \mathsf{commit_P}(\mathsf{ck}, v_1 + v_2, r_1 + r_2).$$

8.1.4 Commitment to Message Vector

The scheme generalizes to vectors when the commitment key is of the form $\mathsf{ck} = (\bar{g}_{\mathsf{com_1}}, \ldots, \bar{g}_{\mathsf{com_k}}, \bar{h}_{\mathsf{com}})$.

> ⚠ The commitment key $\mathsf{ck} = (\bar{g}_{\mathsf{com_1}}, \ldots, \bar{g}_{\mathsf{com_k}}, \bar{h}_{\mathsf{com}})$ must be generated so that no one (in particular the committer) knows a nonzero vector $(x, x_1, \ldots, x_k) \in \mathbb{F}^{k+1}$ for which
>
> $$\bar{h}_{\mathsf{com}}^x \cdot \prod_{i=1}^{k} \bar{g}_{\mathsf{com_i}}^{x_i} = 1,$$
>
> where 1 is the identity in $\mathbb{G}$. See Section 22.1.1 for a secure key-generation method.

Function $\mathsf{VecCommit_P}$

Inputs

$\mathsf{ck} \in \mathrm{COMKEYS} := \mathbb{G}^{k+1}$
    The commitment key which is publicly known.

$m_1, \ldots, m_k \in \mathbb{F} = \mathbb{Z}_q$
    The messages to commit to.

$r \in \mathbb{F} = \mathbb{Z}_q$
    The commit randomness.

Outputs

$\mathsf{C} \in \mathbb{G}$
    The commitment, an element of group $\mathbb{G}$.

Implementation Let $\mathsf{ck} = (\bar{g}_{\mathsf{com_1}}, \ldots, \bar{g}_{\mathsf{com_k}}, \bar{h}_{\mathsf{com}})$. Then

$$\mathsf{C} = \bar{h}_{\mathsf{com}}^r \cdot \prod_{i=1}^{k} \bar{g}_{\mathsf{com_i}}^{m_i}.$$

Function $\mathsf{VecCheckCom_P}$

Inputs

$\mathsf{ck} \in \mathrm{COMKEYS}$
    The commitment key which is publicly known.

$\mathsf{C} \in \mathbb{G}$
    The commitment, an element of group $\mathbb{G}$.

$m_1, \ldots, m_k \in \mathbb{F} = \mathbb{Z}_q$
    The messages.

$r \in \mathbb{F} = \mathbb{Z}_q$
    The commit randomness.

Outputs

$b \in \text{BOOL}$
    Returns true if and only if $\mathtt{C} = \mathsf{commit}_\mathsf{P}(\mathtt{ck}, m_1, \ldots, m_k, r)$

Implementation Let $\mathtt{ck} = (\bar{g}_{\mathtt{com}_1}, \ldots, \bar{g}_{\mathtt{com}_k}, \bar{h}_{\mathtt{com}})$. Then check that

$$\mathtt{C} \overset{!}{=} \bar{h}_{\mathtt{com}}^r \cdot \prod_{i=1}^{k} \bar{g}_{\mathtt{com}_i}^{m_i}.$$

8.1.5 Use Cases

We use the commitment scheme in the Konto and Personligt layers as an auxiliary tool for zero-knowledge proofs. Unless specified otherwise, we use $\mathbb{G} := \mathbb{G}_1^{\mathsf{BLS}}$ with $\mathtt{ck} = (\bar{g}_{\mathtt{com}}, \bar{h}_{\mathtt{com}})$, where $\mathtt{ck}$ is a securely generated, publicly known commitment key (see Section 22.1.1).

> **!** We assume that the commitment scheme is unconditionally (perfectly) hiding and computationally binding.

Chapter 9

Zero-Knowledge Proofs

Cryptographic protocols known as *zero-knowledge* (ZK) *proofs* are extensively used in Concordium blockchain. Such protocols allow one party, the *prover*, to prove to another party, the *verifier*, that a certain statement is true without revealing any additional information beyond the truth of the *statement*. The concept of ZK proofs was introduced by Goldwasser, Micali, and Rackoff in 1985 [GMR85] and since then, various types of zero-knowledge proofs have been developed, each with its own specific characteristics and use cases. Concordium uses classic Σ-*protocols* (see e.g. [Dam10]) and Bulletproofs [Bün+18] because they are transparent, and also known for their simplicity, clarity, and compatibility with algebraic statements. However, in the future, especially when quantum computers are a threat, it will become necessary to replace them with a post-quantum succinct proof system. Σ-protocols are used for simple algebraic statements such as proper encryption of user's data in ID creation, and Bulletproofs for range proofs and set (non-)membership of user's attributes in a public set. We provide detailed descriptions of each in this chapter.

> ⚙ Both Σ-protocols and Bulletproofs will be unsafe in post-quantum setting as Shor's algorithm allows computing discrete logarithms efficiently. Proofs should not be vulnerable to harvest attacks.
> Potential replacements are zero-knowledge protocols based on lattice assumptions or based on hash functions.

9.1 Abstract Treatment of Sigma Protocols

In this section, we describe an abstract protocol capturing the type of Σ-*protocols* we will be using. Section 9.2 shows how to instantiate this abstract protocol in different ways to prove concrete statements. Finally, in Section 9.3, we show how to make these protocols non-interactive via the Fiat-Shamir paradigm.

9.1.1 Abstract Protocol

All our Σ-protocols follow Maurer's abstraction [Mau15], which we briefly describe in this section.

Let $(G, +)$ and $(H, \cdot)$ be *group*s, where "+" and "·" denote the respective group operations. Further, let $\varphi\colon G \to H$ be a *group homomorphism*, i.e., for all $g_1, g_2 \in G$,

$$\varphi(g_1 + g_2) = \varphi(g_1) \cdot \varphi(g_2).$$

The following abstract protocol allows a *prover* to prove knowledge of a *preimage* of φ. That is, the prover convinces the *verifier* about the knowledge of a value $x \in G$, called the *witness*,

such that $\varphi(x) = y$, where $y \in H$ is known to the verifier. For that protocol, the *challenge space* $\mathcal{C} \subseteq \mathbb{N}$ is an arbitrary (sufficiently large, see Theorem 23) subset of $\mathbb{N}$. Mostly, the challenge space is a subset of $\mathbb{Z}_{|G|}$. The practical choice of the challenge spaces for our implementations are discussed in Section 9.4.3.

Protocol PK $\{(x) : y = \varphi(x)\}$

Implicitly known public values: G, H, φ, $\mathcal{C}$.

Prover(x) **Verifier**(y)

Choose uniformly random
$\alpha \leftarrow G$ and compute
$a := \varphi(\alpha)$.

$\xrightarrow{\quad\quad a \quad\quad}$

Choose uniformly random
$c \leftarrow \mathcal{C}$.

$\xleftarrow{\quad\quad c \quad\quad}$

Compute[a] $z := \alpha - cx$. $\xrightarrow{\quad\quad z \quad\quad}$ Accept iff $a = \varphi(z) \cdot y^c$.

---

[a]Note that we use additive notation for the group G, so cx refers to $x + x + \ldots + x$, c times.

Remark. We adjust Maurer's notation in [Mau15] slightly to map it to the more common Σ-protocol literature where the *commit secret* is α, *commit message* is a, *challenge* is c and *response* is z.

The following theorem allows us to easily verify the security of an instance of this abstract protocol.

Theorem 23 (cf. Theorem 3 in [Mau15]). *The abstract protocol above has special soundness and special honest verifier zero-knowledge if values $k \in \mathbb{Z}$ and $u \in G$ are known such that*

- $\forall c_1, c_2 \in \mathcal{C} \; c_1 \neq c_2 \;\Rightarrow\; \gcd(c_1 - c_2, k) = 1$,
- $\varphi(u) = y^k$, *and*
- $1/|\mathcal{C}|$ *is negligible.*

> ℹ The way challenges are used in the protocol implies that they are implicitly reduced modulo the order of the group G. It therefore typically does not make sense to include numbers exceeding $|G|$ in $\mathcal{C}$. Note that if the premise of Theorem 23 is satisfied for some $\mathcal{C}$ containing c_1, c_2 with $c_1 \equiv c_2 \pmod{|G|}$, then a preimage of y is generally known and thus the proof of knowledge is of no interest: The first condition implies in this case that k and $|G|$ are co-prime, i.e., k can be inverted modulo $|G|$. Using this, one can efficiently compute a preimage of y using the second condition.

9.1.2   AND-Composition of Proofs

Given group homomorphisms $\varphi_1\colon G_1 \to H_1, \ldots, \varphi_n\colon G_n \to H_n$, the abstract model from Section 9.1.1 allows proving knowledge of preimages under all of them simultaneously as follows: Consider the *Cartesian product* $(\varphi_1 \times \ldots \times \varphi_n)(x_1, \ldots, x_n) := (\varphi_1(x_1), \ldots, \varphi_n(x_n))$ of the homomorphisms and use the abstract protocol for $\varphi_1 \times \ldots \times \varphi_n$. Knowledge of a preimage of $(y_1, \ldots, y_n)$ under $\varphi_1 \times \ldots \times \varphi_n$ is then equivalent to knowing a preimage of y_1 under φ_1 and a

preimage of y_2 under φ_2 etc. As challenge space, one can use the intersection $\mathcal{C} := \bigcap_{i=1}^{n} \mathcal{C}_i$ of challenge spaces used in the individual proofs.

More concretely, this yields the following protocol for *proving conjunctions*.

Protocol PK $\{(x_1, \dots, x_n) : \bigwedge_{i=1}^{n} y_i = \varphi_i(x_i)\}$

Implicitly known public values: $G_1, H_1, \varphi_1, \dots, G_n, H_n, \varphi_n, \mathcal{C}$.

Prover$(x_1, \dots, x_n)$ **Verifier**$(y_1, \dots, y_n)$

Choose uniformly random
$\alpha_i \leftarrow G_i$ and compute
$a_i := \varphi_i(\alpha_i)$ for
$i = 1, \dots, n.$ $\xrightarrow{\quad a_1, \dots, a_n \quad}$

                                                  Choose uniformly random
                         $\xleftarrow{\quad\quad c \quad\quad}$ $c \leftarrow \mathcal{C}.$

Compute $z_i := \alpha_i - cx_i$ for   Accept iff $a_i = \varphi_i(z_i) \cdot y_i^c$
$i = 1, \dots, n.$ $\xrightarrow{\quad z_1, \dots, z_n \quad}$ for $i = 1, \dots, n.$
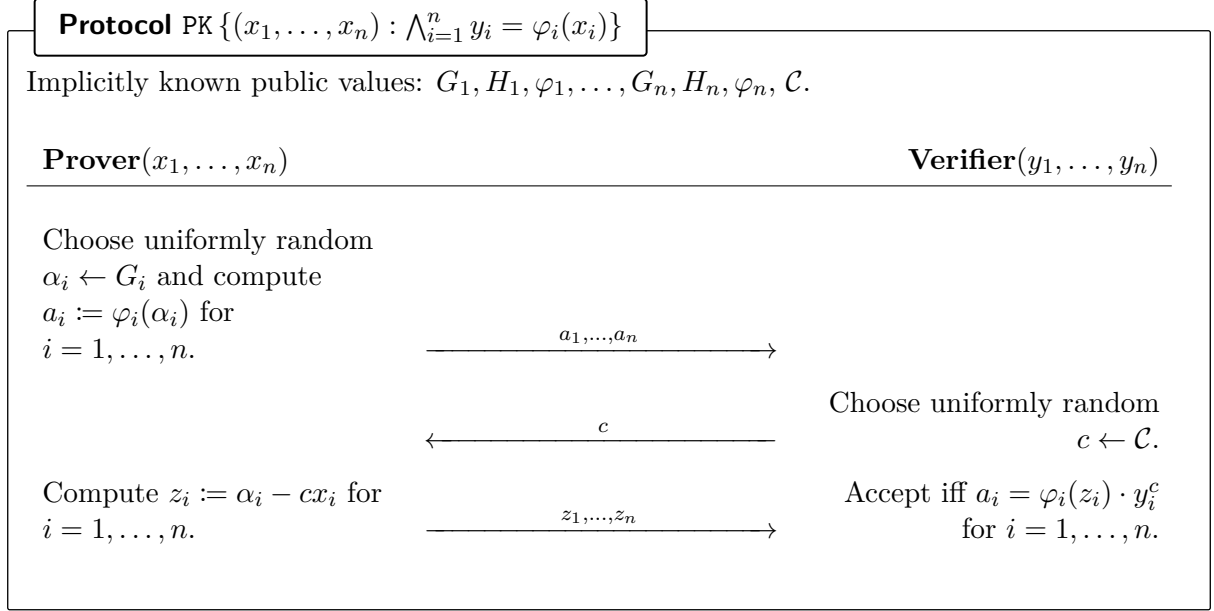
We next show that this protocol is secure if the individual homomorphisms φ_i satisfy the premise of Theorem 23, and if $\mathcal{C} := \bigcap_{i=1}^{n} \mathcal{C}_i$ is sufficiently large.

Lemma 24. *The protocol above has special soundness and special honest verifier zero-knowledge if values $k_i \in \mathbb{Z}$ and $u_i \in G_i$ are known such that for all $i \in \{1, \dots, n\}$,*

- $\forall c_1, c_2 \in \mathcal{C}_i \ c_1 \neq c_2 \ \Rightarrow \ \gcd(c_1 - c_2, k_i) = 1$,

- $\varphi_i(u_i) = y_i^{k_i}$, *and*

- $1/|\mathcal{C}|$ *is negligible.*

Proof. Our goal is to apply Theorem 23 to $\varphi_1 \times \dots \times \varphi_n$. We first show that $\varphi_1 \times \dots \times \varphi_n$ is a group homomorphism if all φ_i are:

$$
\begin{aligned}
&(\varphi_1 \times \dots \times \varphi_n)((x_1, \dots, x_n) + (y_1, \dots, y_n)) \\
&= (\varphi_1(x_1 + y_1), \dots, \varphi_n(x_n + y_n)) \\
&= (\varphi_1(x_1) \cdot \varphi_1(y_1), \dots, \varphi_n(x_n) \cdot \varphi_n(y_n)) \\
&= (\varphi_1 \times \dots \times \varphi_n)(x_1, \dots, x_n) \cdot (\varphi_1 \times \dots \times \varphi_n)(y_1, \dots, y_n).
\end{aligned}
$$

Now let $k' := k_1 \cdot \dots \cdot k_n$ and let for $i \in \{1, \dots, n\}$, $u_i' := \frac{k'}{k_i} \cdot u_i \in G_i$.[1] We then have for all distinct $c_1, c_2 \in \mathcal{C} = \bigcap_{i=1}^{n} \mathcal{C}_i$ that $\gcd(c_1 - c_2, k') = 1$ (since otherwise, some prime p would divide both k' and $c_1 - c_2$ and that prime would then also divide some k_i, contradicting $\gcd(c_1 - c_2, k_i) = 1$). Furthermore, we have for all $i \in \{1, \dots, n\}$,

$$
\varphi_i(u_i') = \varphi_i\left(\frac{k'}{k_i} \cdot u_i\right) = \varphi_i(u_i)^{\frac{k'}{k_i}} = (y_i^{k_i})^{\frac{k'}{k_i}} = y_i^{k'}.
$$

---

[1] Note that $\frac{k'}{k_i} \cdot u_i = u_i + \dots + u_i$, $\frac{k'}{k_i} = \prod_{j \in \{1, \dots, n\} \setminus \{i\}} k_j$ times.

Therefore,

$$(\varphi_1 \times \ldots \times \varphi_n)(u'_1, \ldots, u'_n) = (\varphi_1(u'_1), \ldots, \varphi_n(u'_n)) = (y_1^{k'}, \ldots, y_n^{k'}) = (y_1, \ldots, y_n)^{k'}.$$

Hence, the result follows by applying Theorem 23 to $\varphi_1 \times \ldots \times \varphi_n$ and the values k' and $u'_1, \ldots, u'_n$. $\qquad\square$

9.1.3 Proving Equality of Known Values

Consider a group homomorphism $\varphi \colon G_1 \times \ldots \times G_5 \to H$ (possibly obtained via AND-composition as described in Section 9.1.2) for which we can prove knowledge of preimages. We now want to additionally prove that certain values in the preimage are equal. For this to make sense, the corresponding groups need to be equal. For example, assume we want to prove that we know a preimage $x_1, \ldots, x_5$ of $y \in H$ such that $x_1 = x_3 = x_4$, and assume we have $G_1 = G_3 = G_4$. This can be done by applying the abstract protocol to the homomorphism ψ we define as follows: Let $G' := G_1 (= G_3 = G_4)$ and let

$$\psi \colon G' \times G_2 \times G_5 \to H, (g', g_2, g_5) \mapsto \varphi(g', g_2, g', g', g_5).$$

We now define this operation more generally.

Definition 25. Let $G_1, \ldots, G_n, H$ be groups and let $\varphi \colon G_1 \times \ldots \times G_n \to H$ be a group homomorphism. Further, let $L \subseteq \{1, \ldots, n\}$, $L \neq \varnothing$, such that $G_i = G_j$ for all $i, j \in L$, and let $G' := G_i$ for $i \in L$. We then define for $J := \{1, \ldots, n\} \setminus L$,

$$\mathrm{EqH}(\varphi, L) \colon G' \times \bigtimes_{j \in J} G_i \to H, \left(x', (x_j)_{j \in J}\right) \mapsto \varphi(\tilde{x}), \quad \text{with } \tilde{x}_i := \begin{cases} x_i, & i \in J, \\ x', & i \in L. \end{cases}$$

Using this definition, we define the following *protocol for equality of know values*, which simply applies the abstract protocol from Section 9.1.1 to $\mathrm{EqH}(\varphi, L)$.

Protocol $\mathrm{PK}\left\{(x_1, \ldots, x_n) : y = \varphi(x_1, \ldots, x_n) \wedge \forall i, j \in L : x_i = x_j\right\}$

Implicitly known public values: $G_1, \ldots, G_n, H, \varphi, L, \mathcal{C}$.

Prover$(x_1, \ldots, x_n)$	**Verifier**(y)

Let $J := \{1, \ldots, n\} \setminus L$ and $G' := G_i$ for $i \in L$. Choose uniformly random $\alpha' \leftarrow G'$, $\alpha_j \leftarrow G_j$ for $j \in J$, and compute $a := \mathrm{EqH}(\varphi, L)(\alpha', (\alpha_j)_{j \in J})$.

$\xrightarrow{\quad a \quad}$

Choose uniformly random $c \leftarrow \mathcal{C}$.

$\xleftarrow{\quad c \quad}$

Compute $z_j := \alpha_j - c x_j$ for $j \in J$, and $z' := \alpha' - c x_l$ for some $l \in L$.

$\xrightarrow{\quad z',(z_j)_{j \in J} \quad}$

Accept iff $a = \mathrm{EqH}(\varphi, L)(z', (z_j)_{j \in J}) \cdot y^c$.

We next prove the security of the above protocol. We need for this that the premise of Theorem 23 is satisfied for φ and additionally, all u_i from Theorem 23 are equal for all $i \in L$.

Lemma 26. *Let $G_1, \ldots, G_n, H$ be groups, let $\varphi \colon G_1 \times \ldots \times G_n \to H$ be a group homomorphism, and let $L \subseteq \{1, \ldots, n\}$, $L \neq \varnothing$, such that $G_i = G_j$ for all $i, j \in L$. Furthermore, assume values $k \in \mathbb{Z}$ and $u_1 \in G_1, \ldots, u_n \in G_n$ are known such that*

- $\forall c_1, c_2 \in \mathcal{C}\; c_1 \neq c_2 \;\Rightarrow\; \gcd(c_1 - c_2, k) = 1$,

- $\varphi(u_1, \ldots, u_n) = y^k$

- $\forall i, j \in L : u_i = u_j$, *and*

- $1/|\mathcal{C}|$ *is negligible,*

then the protocol above has special soundness and special honest verifier zero-knowledge.

Proof. We first show that $\mathrm{EqH}(\varphi, L)$ is a group homomorphism:

$$
\begin{aligned}
\mathrm{EqH}(\varphi, L)(x' + y', (x_j + y_j)_{j \in J}) &= \varphi(\tilde{x} + \tilde{y}) \\
&= \varphi(\tilde{x}) \cdot \varphi(\tilde{y}) \\
&= \mathrm{EqH}(\varphi, L)(x', (x_j)_{j \in J}) \cdot \mathrm{EqH}(\varphi, L)(y', (y_j)_{j \in J}).
\end{aligned}
$$

Let $u' = u_l$ for some $l \in L$. Note that by assumption,

$$
\mathrm{EqH}(\varphi, L)(u', (u_j)_{j \in J}) = \varphi(\tilde{u}) = \varphi(u_1, \ldots, u_n) = y^k.
$$

We can thus apply Theorem 23 to conclude the proof. $\qquad\square$

> ℹ️ If φ is the result of an AND-composition as described in Section 9.1.2, one can use $\prod_i k_i$ as the k required in Lemma 26, see Lemma 24.

9.1.4 Proving Linear Relation of Known Values

Consider a group homomorphism $\varphi \colon G_1 \times \ldots \times G_n \to H$ (possibly obtained via AND-composition as described in Section 9.1.2) for which we can prove knowledge of preimages. We now want to additionally prove that one of the preimages can be expressed as some *linear relation*. That is, we want to prove that $x_{i_0} = \sum_{i=1}^{m} v_i t_i$ for some i_0, and (publicly known) $v_1, \ldots, v_m \in \mathbb{Z}$, where $t_1, \ldots, t_m$ are part of the new witness.

> ⚠️ The t_i can be seen as temporary variables that can be proven to be equal to some other values using the method described in Section 9.1.3.

The proof can be constructed by defining a new group homomorphism that replaces x_{i_0} with $\sum_{i=1}^{m} v_i t_i$. We next define a generic transformation for obtaining this homomorphism.

Definition 27. Let $G_1, \ldots, G_n, H$ be groups and let $\varphi \colon G_1 \times \ldots \times G_n \to H$ be a group homomorphism. Further, let $i_0 \in \{1, \ldots, n\}$ and $v_1, \ldots, v_m \in \mathbb{Z}$. We then define

$$
\mathrm{LinH}(\varphi, i_0, (v_1, \ldots, v_m)) \colon G_1 \times \ldots \times G_{i_0-1} \times G_{i_0+1} \times \ldots \times G_n \times G_{i_0}^m \to H,
$$

$$
(x_1, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_n, t_1, \ldots, t_m) \mapsto \varphi\left(x_1, \ldots, x_{i_0-1}, \sum_{i=1}^{m} v_i t_i, x_{i_0+1}, \ldots, x_n\right).
$$

Using this definition, we define the following protocol, which simply applies the abstract protocol from Section 9.1.1 to $\mathrm{LinH}(\varphi, i_0, (v_1, \ldots, v_m))$.

Protocol PK $\{(x_1, \ldots, x_n, t_1, \ldots, t_m) : y = \varphi(x_1, \ldots, x_n) \wedge x_{i_0} = \sum_{i=1}^m v_i t_i\}$

Implicitly known public values: $G_1, \ldots, G_n$, H, φ, L, i_0, $v_1, \ldots, v_m$, $\mathcal{C}$.

Prover$(x_1, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_n)$	**Verifier**(y)

Choose uniformly random
$\alpha_i \leftarrow G_i$ for
$i \in \{1, \ldots, n\} \setminus \{i_0\}$ and
$\beta_1, \ldots, \beta_m \leftarrow G_{i_0}$, and
compute $a :=$
$\text{LinH}(\varphi, i_0, (v_1, \ldots, v_m))(\alpha_1,$
$\ldots, \alpha_{i_0-1}, \alpha_{i_0+1}, \ldots, \alpha_n,$
$\beta_1, \ldots, \beta_m)$.

$$\xrightarrow{\quad a \quad}$$

Choose uniformly random
$c \leftarrow \mathcal{C}$.

$$\xleftarrow{\quad c \quad}$$

Compute $z_i := \alpha_i - c x_i$ for
$i \in \{1, \ldots, n\} \setminus \{i_0\}$ and
$z_i' := \beta_i - c t_i$ for
$i \in \{1, \ldots, m\}$.

$$\xrightarrow{(z_1, \ldots, z_{i_0-1}, z_{i_0+1}, \ldots, z_n, z_1', \ldots, z_m')}$$

Accept iff $a =$
$\text{LinH}(\varphi, i_0, (v_1, \ldots, v_m))(z_1,$
$\ldots, z_{i_0-1}, z_{i_0+1}, \ldots, z_n,$
$z_1', \ldots, z_m') \cdot y^c$.

We next prove the security of that protocol. We need for this that the premise of Theorem 23 is satisfied for φ and additionally, there are $u_1', \ldots, u_m'$ known such that the u_{i_0} from Theorem 23 equals $u_{i_0} = \sum_{i=1}^m v_i u_i'$.

Lemma 28. *Let $G_1, \ldots, G_n, H$ be groups, let $\varphi \colon G_1 \times \ldots \times G_n \to H$ be a group homomorphism, and $i_0 \in \{1, \ldots, n\}$. Further, let $v_1, \ldots, v_m \in \mathbb{Z}$, and assume values $k \in \mathbb{Z}$ and $u_1 \in G_1, \ldots, u_n \in G_n$ and $u_1', \ldots, u_m' \in G_{i_0}$ are known such that*

- *$\forall c_1, c_2 \in \mathcal{C} \; c_1 \neq c_2 \;\Rightarrow\; \gcd(c_1 - c_2, k) = 1$,*

- *$\varphi(u_1, \ldots, u_n) = y^k$*

- *$u_{i_0} = \sum_{i=1}^m v_i u_i'$, and*

- *$1/|\mathcal{C}|$ is negligible,*

then the protocol above has special soundness and special honest verifier zero-knowledge.

Proof. We first show that $\text{LinH}(\varphi, i_0, (v_1, \ldots, v_m))$ is a group homomorphism:

$$\text{LinH}(\varphi, i_0, (v_1, \ldots, v_m))\big((x_1, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_n, t_1, \ldots, t_m)$$
$$+ (y_1, \ldots, y_{i_0-1}, y_{i_0+1}, \ldots, y_n, s_1, \ldots, s_m)\big)$$
$$= \varphi\left(x_1 + y_1, \ldots, x_{i_0-1} + y_{i_0-1}, \sum_{i=1}^m v_i(t_i + s_i), x_{i_0+1} + y_{i_0+1}, \ldots, x_n + y_n\right)$$
$$= \varphi\left(x_1 + y_1, \ldots, x_{i_0-1} + y_{i_0-1}, \sum_{i=1}^m v_i t_i + \sum_{i=1}^m v_i s_i, x_{i_0+1} + y_{i_0+1}, \ldots, x_n + y_n\right)$$
$$= \text{LinH}(\varphi, i_0, (v_1, \ldots, v_m))(x_1, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_n, t_1, \ldots, t_m)$$
$$\cdot \text{LinH}(\varphi, i_0, (v_1, \ldots, v_m))(y_1, \ldots, y_{i_0-1}, y_{i_0+1}, \ldots, y_n, s_1, \ldots, s_m).$$

Note that by assumption,

$$\mathrm{LinH}(\varphi, i_0, (v_1, \ldots, v_m))(u_1, \ldots, u_{i_0-1}, u_{i_0+1}, \ldots, u_n, u'_1, \ldots, u'_m)$$

$$= \varphi\left(u_1, \ldots, u_{i_0-1}, \sum_{i=1}^{m} v_i u'_i, u_{i_0+1}, \ldots, u_n\right)$$

$$= \varphi(u_1, \ldots, u_{i_0-1}, u_{i_0}, \ldots, u_n)$$

$$= y^k.$$

We can thus apply Theorem 23 to conclude the proof. □

9.2 Instantiations of Sigma Protocols

9.2.1 Proof of Knowledge of Discrete Logarithm

Let $(\mathbb{G}, \cdot)$ be a group of prime order q with generator g. The prover wants to show knowledge of the *discrete logarithm* x (with respect to g) of a group element $y = g^x$.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)$ and $H := (\mathbb{G}, \cdot)$. The homomorphism φ is defined as

$$\varphi(x) := g^x.$$

This is a homomorphism since $\varphi(x + y) = g^{x+y} = g^x \cdot g^y = \varphi(x) \cdot \varphi(y)$. The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = 0$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$. Furthermore, $\varphi(0) = 1 = y^q$, since all elements raised to the group order q yield the neutral element $1 \in \mathbb{G}$.

Full protocol

$$\boxed{\begin{array}{l}
\textbf{Protocol } \mathsf{dlog}\colon \mathrm{PK}\,\{(x) : y = g^x\} \\[4pt]
\text{Implicit public values: } \mathbb{G}, q, g \\[8pt]
\begin{array}{lr}
\mathbf{Prover}(x) & \mathbf{Verifier}(y) \\
\hline \\
\text{Choose random } \alpha \leftarrow \mathbb{Z}_q & \\
\text{and compute } a = g^\alpha. \quad \xrightarrow{\quad a \quad} & \\
\quad \xleftarrow{\quad c \quad} & \text{Choose random } c \leftarrow \mathcal{C}. \\
\text{Compute } z = \alpha - cx & \\
(\mathrm{mod}\ q). \quad \xrightarrow{\quad z \quad} & \text{Accept iff } a = g^z \cdot y^c.
\end{array}
\end{array}}$$
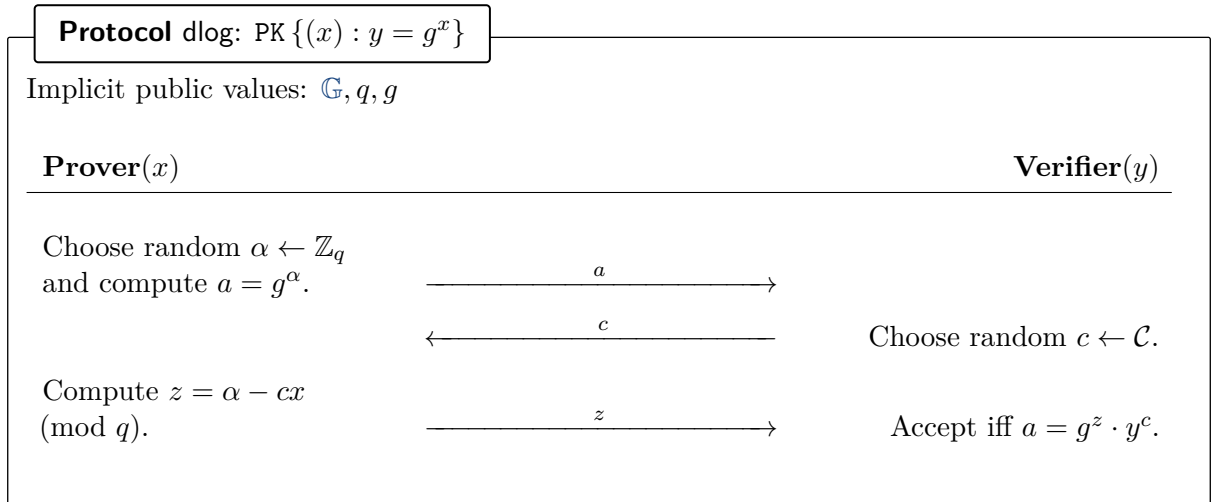
9.2.2 Proof of Knowledge for Aggregate Discrete Logarithms

Let $(\mathbb{G}, \cdot)$ be a group of prime order q, and let $g_1, \ldots, g_n$ be generators of $\mathbb{G}$. For $y = \prod_{i=1}^{n} g_i^{x_i}$, the prover wants to show *knowledge of discrete logarithms* $x_1, \ldots, x_n$.

! It may be desirable for applications that the prover does not know the discrete logarithm of g_i with respect to g_j for any $i \neq j$. For the security of this protocol, however, this is not needed.

i Note that the protocol in Section 9.2.1 is a special case of this with $n = 1$.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^n$ (with component-wise group operation) and $H := (\mathbb{G}, \cdot)$. The homomorphism φ is defined as

$$\varphi(x_1, \ldots, x_n) := \prod_{i=1}^{n} g_i^{x_i}.$$

This is a homomorphism since

$$\varphi((x_1, \ldots, x_n) + (y_1, \ldots, y_n)) = \prod_{i=1}^{n} g_i^{x_i + y_i} = \prod_{i=1}^{n} g_i^{x_i} \cdot \prod_{i=1}^{n} g_i^{y_i} = \varphi(x_1, \ldots, x_n) \cdot \varphi(y_1, \ldots, y_n).$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, \ldots, 0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0, \ldots, 0) = 1 = y^q$.

Full protocol

Protocol agg-dlog: $\mathrm{PK}\left\{(x_1, \ldots, x_n) : y = \prod_{i=1}^{n} g_i^{x_i}\right\}$

Implicit public values: $\mathbb{G}, q, g_1, \ldots, g_n$

Prover$(x_1, \ldots, x_n)$ **Verifier**(y)

Choose random $\alpha_i \leftarrow \mathbb{Z}_q$
and compute $a = \prod_{i=1}^{n} g_i^{\alpha_i}$. $\xrightarrow{\quad a \quad}$

       Choose random
       $c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$\xleftarrow{\quad c \quad}$

Compute $z_i = \alpha_i - c x_i$
$\pmod{q}$. $\xrightarrow{\quad (z_1, \ldots, z_n) \quad}$ Accept iff $a = \prod_{i=1}^{n} g_i^{z_i} \cdot y^c$.

9.2.3 Proof of Knowledge of Opening of Commitment

Let $(\mathbb{G}, \cdot)$ be a group of prime order q, and let g and h be generators of $\overline{\mathbb{G}}$. The prover has produced a commitment $C = g^x h^r$ to the value x with randomness r (cf. Section 8.1), and wants to show *knowledge of these values* x and r.

! Security of the commitment requires that the discrete logarithm of h with respect to g is not known. For the security of the Σ-protocol, this is not required, though.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^2$ (with component-wise group operations) and $H := (\overline{\mathbb{G}}, \cdot)$. The homomorphism φ is defined as

$$\varphi(x, r) := g^x h^r.$$

This is a homomorphism since

$$\varphi((x, r) + (y, s)) = g^{x+y} h^{r+s} = g^x h^r g^y h^s = \varphi(x, r) \cdot \varphi(y, s).$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, 0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0, 0) = 1 = C^q$.

Full protocol

Protocol com: $\mathrm{PK}\{(x, r) : C = g^x h^r\}$

Implicit public values: $\overline{\mathbb{G}}, q, g, h$

Prover(x, r)		**Verifier**(C)

Choose random $\alpha, \beta \leftarrow \mathbb{Z}_q$ and compute $a := g^\alpha h^\beta$.

$\xrightarrow{\quad\quad a \quad\quad}$

Choose random $c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$\xleftarrow{\quad\quad c \quad\quad}$

Compute $s := \alpha - cx$ $(\bmod\ q)$, $t := \beta - cr$ $(\bmod\ q)$.

$\xrightarrow{\quad\quad (s,t) \quad\quad}$

Accept iff $a = g^s h^t \cdot C^c$.
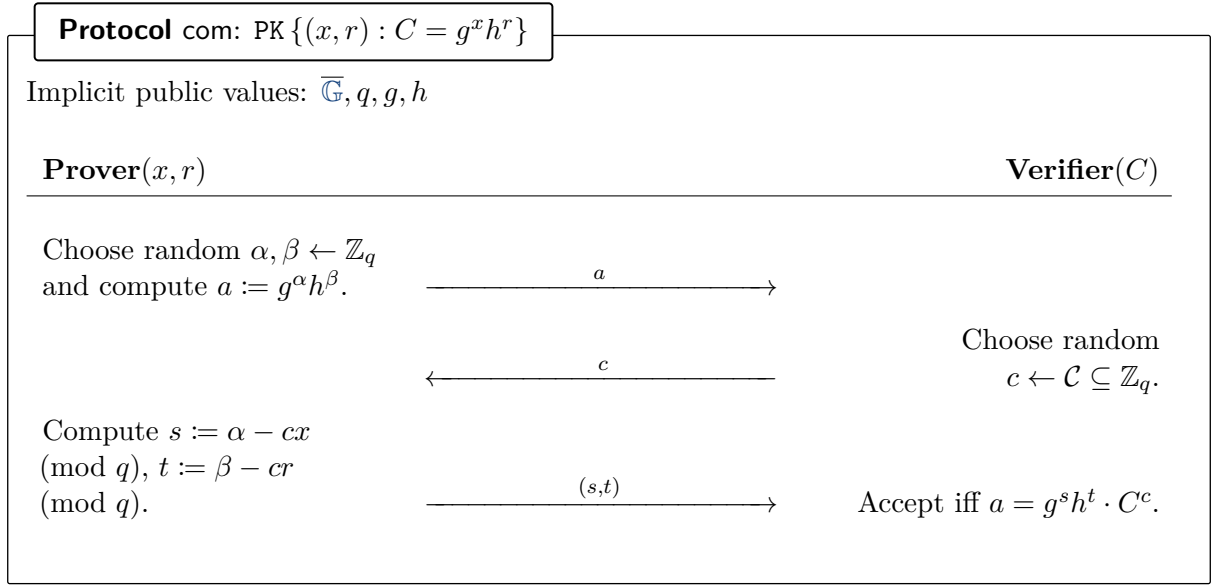
9.2.4 Proof of Knowledge of Opening of Commitment with Public Value

Let $(\overline{\mathbb{G}}, \cdot)$ be a group of prime order q, and let g and h be generators of $\overline{\mathbb{G}}$. The prover has produced a commitment $C = g^x h^r$ to the public value x with (secret) randomness r (cf. Section 8.1). The prover wants to show that the *commitment C is a commitment to x*, i.e.[2], that they know the randomness r which opens C to x.

Since x is public, one can reduce the problem to show knowledge of the discrete logarithm r of $C' := C \cdot g^{-x}$ (with respect to h). This can be done by calling protocol 9.2.1 with witness r and public input C'.

For completeness, we provide the full protocol here.

---

[2]Pedersen commitments are perfectly hiding, thus C can be seen as a commitment to any value v. However, computing the opening information for v given C requires to know the discrete log relation between g and h.

Implicit public values: $\overline{\mathbb{G}}, q, g, h$

Prover(r)		**Verifier**(C, x)

Choose random $\alpha \leftarrow \mathbb{Z}_q$
and compute $a := h^\alpha$.

$\xrightarrow{\quad\quad a \quad\quad}$

Choose random
$c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$\xleftarrow{\quad\quad c \quad\quad}$

Compute $z := \alpha - cr$
$\pmod q$.

$\xrightarrow{\quad\quad z \quad\quad}$

Accept iff
$a = h^z \cdot (C \cdot g^{-x})^c$.

9.2.5 Proof of Equality for Aggregated Discrete Logarithms and Commitments

Let $(\mathbb{G}, \cdot)$, and $(\overline{\mathbb{G}}, \cdot)$ be groups of prime order q. Let $g_1, \ldots, g_n$ be generators of $\mathbb{G}$ and let g and h be generators of $\overline{\mathbb{G}}$.

The prover has produced $y = \prod_{i=1}^n g_i^{x_i}$ and commitments $\{C_i = g^{x_i} h^{r_i}\}_{i=1,\ldots,n}$ (cf. Section 8.1). The prover wants to show that the same $x_1, \ldots, x_n$ appear in y and in the commitments. We do this by showing that the prover actually knows such values. This is a *generalization* of the protocol from Section 9.2.2 where equality with the committed values is shown in addition.

> **!** Security of the commitment requires that the discrete logarithm of h with respect to g is not known. For the security of the Σ protocol, this is not required, though.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^{2n}$ and $H := (\mathbb{G}, \cdot) \times (\overline{\mathbb{G}}, \cdot)^n$ (with component-wise group operations). The homomorphism φ is defined as

$$\varphi(x_1, \ldots, x_n, r_1, \ldots, r_n) := \left(\prod_{i=1}^n g_i^{x_i}, g^{x_1} h^{r_1}, \ldots, g^{x_n} h^{r_n} \right).$$

This is a homomorphism since

$$\varphi((x_1, \ldots, x_n, r_1, \ldots, r_n) + (y_1, \ldots, y_n, s_1, \ldots, s_n))$$
$$= \left(\prod_{i=1}^n g_i^{x_i + y_i}, g^{x_1 + y_1} h^{r_1 + s_1}, \ldots, g^{x_n + y_n} h^{r_n + s_n} \right)$$
$$= \left(\prod_{i=1}^n g_i^{x_i}, g^{x_1} h^{r_1}, \ldots, g^{x_n} h^{r_n} \right) \cdot \left(\prod_{i=1}^n g_i^{y_i}, g^{y_1} h^{s_1}, \ldots, g^{y_n} h^{s_n} \right)$$
$$= \varphi(x_1, \ldots, x_n, r_1, \ldots, r_n) \cdot \varphi(y_1, \ldots, y_n, s_1, \ldots, s_n).$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, \ldots, 0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0, \ldots, 0) = (1, \ldots, 1) = (y, C_1, \ldots, C_n)^q$.

Full protocol

Protocol com-dlog-eq: PK $\left\{(x_1, \ldots, x_n, r_1, \ldots, r_n) : y = \prod_{i=1}^{n} g_i^{x_i} \wedge C_i = g^{x_i} h^{r_i} (i = 1, \ldots, n)\right\}$

Implicit public values: $\mathbb{G}, \overline{\mathbb{G}}, q, g_1, \ldots, g_n, g, h$

Prover$(x_1, \ldots, x_n, r_1, \ldots, r_n)$ **Verifier**$(y, C_1, \ldots, C_n)$

Choose random $\alpha_i, \tilde{r}_i \leftarrow \mathbb{Z}_q$
and compute $a = \prod_{i=1}^{n} g_i^{\alpha_i}$
and $a_i = g^{\alpha_i} h^{\tilde{r}_i}$.

$\xrightarrow{\quad (a, a_1, \ldots, a_n) \quad}$

Choose random
$c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$\xleftarrow{\quad c \quad}$

Compute $s_i = \alpha_i - cx_i$
$\pmod{q}$, $t_i = \tilde{r}_i - cr_i$
$\pmod{q}$.

$\xrightarrow{\quad (s_1, \ldots, s_n, t_1, \ldots, t_n) \quad}$

Accept iff $a = \prod_{i=1}^{n} g_i^{s_i} \cdot y^c$
and $a_i = g^{s_i} h^{t_i} \cdot C_i^c$.
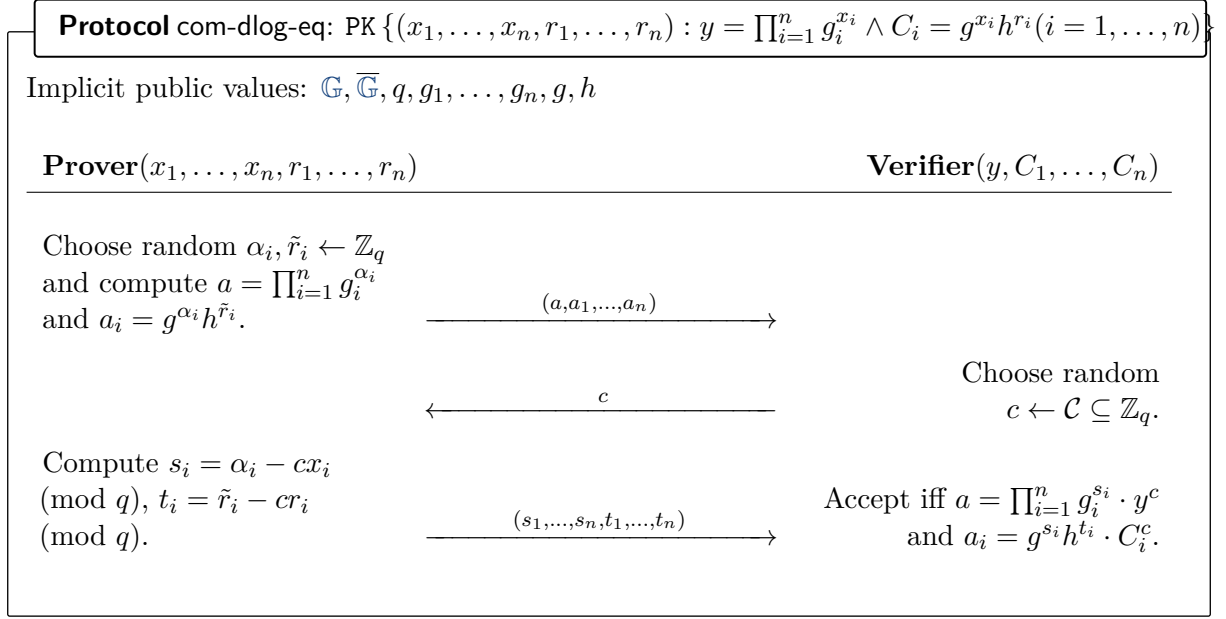
9.2.6 Proof of Equality for Aggregated Discrete Logarithms and Separate Discrete Logarithms

Let $(\mathbb{G}, \cdot)$, and $(\overline{\mathbb{G}}, \cdot)$ be groups of prime order q. Let $g_1, \ldots, g_n$ be generators of $\mathbb{G}$ and let g be a generator of $\overline{\mathbb{G}}$.

The prover has produced $y = \prod_{i=1}^{n} g_i^{x_i}$ and for subset $J \subseteq \{1, \ldots, n\}$ $y_j = g^{x_j}$ where $j \in J$. The prover wants to show that the same $x_1, \ldots, x_n$ appear in y and (the ones for subset J) in the elements $\{y_j\}$. We do this by showing that the prover actually knows such values. This is a generalization of the protocol from Section 9.2.2 where equality with the separate discrete logs is shown in addition.

Instantiation of abstract protocol

Let $|J| = \ell$. We set $G := (\mathbb{Z}_q, +)^n$ and $H := (\mathbb{G}, \cdot) \times (\overline{\mathbb{G}}, \cdot)^{\ell}$ (with component-wise group operations). The homomorphism φ is defined as

$$\varphi(x_1, \ldots, x_n) := \left(\prod_{i=1}^{n} g_i^{x_i}, g^{x_{j_1}}, \ldots, g^{x_{j_\ell}} \right).$$

This is a homomorphism that can be constructed from the homomorphism in Sections 9.2.1 & 9.2.2 using the techniques in Sections 9.1.2 & 9.1.3.

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

Analogous to the protocol in Section 9.2.5, the conditions of Theorem 23 are satisfied.

Full protocol

Protocol dlog-dlog-eq: PK $\left\{(x_1, \ldots, x_n) : y = \prod_{i=1}^{n} g_i^{x_i} \wedge y_j = g^{x_i} (j \in J)\right\}$

Implicit public values: $\mathbb{G}, \overline{\mathbb{G}}, q, g_1, \ldots, g_n, g, J$

Prover$(x_1, \ldots, x_n)$	**Verifier**$(y, \{y_j\}_{j \in J})$

Choose random $\alpha_i \leftarrow \mathbb{Z}_q$
and compute $a = \prod_{i=1}^{n} g_i^{\alpha_i}$
and $a_j = g^{\alpha_j}$ for $j \in J$.

$$\xrightarrow{\quad (a, \{a_j\}_{j \in J}) \quad}$$

Choose random
$c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$$\xleftarrow{\quad c \quad}$$

Compute $s_i = \alpha_i - cx_i$
$\pmod{q}$.

$$\xrightarrow{\quad (s_1, \ldots, s_n) \quad}$$

Accept iff $a = \prod_{i=1}^{n} g_i^{s_i} \cdot y^c$
and $a_j = g^{s_j} \cdot y_j^c$ for $j \in J$.
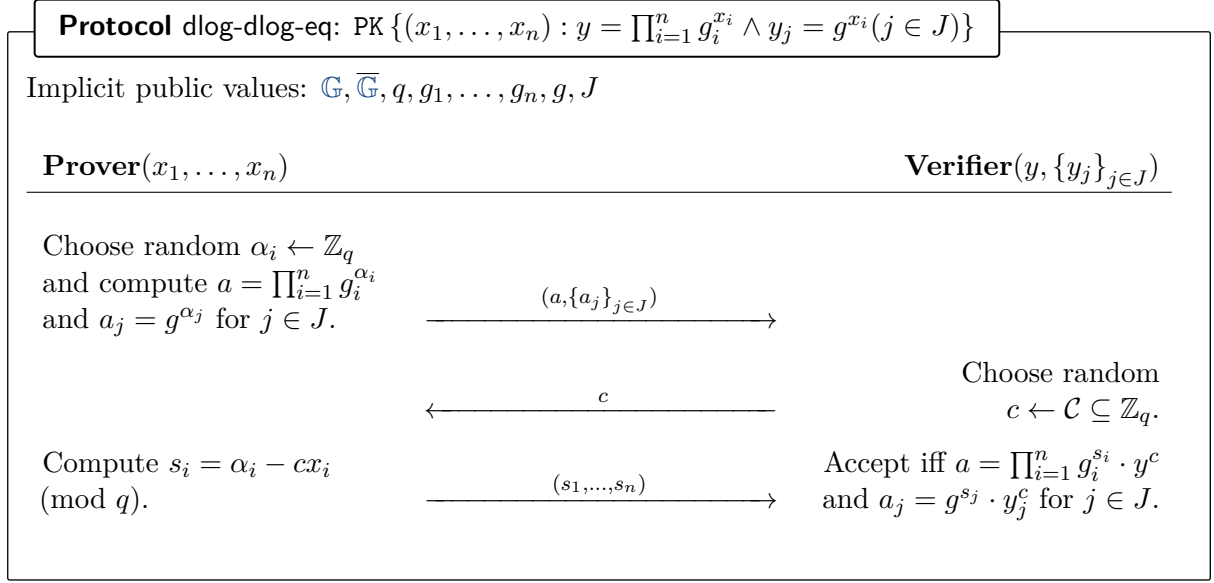
9.2.7 Proof of Equality for Committed Value and ElGamal Encrypted Value

Let $\mathbb{G}$ be a group of prime order q with generators g_1, g, h. The prover has produced an (additively homomorphic) ElGamal encryption $(e_1, e_2) = (g_1^r, g_1^x \cdot \texttt{pk}^r)$ of x in the exponent under public key $\texttt{pk}$ (see Section 7.1.4). The prover is also committed to x with commitment $C = g^x h^{\tilde{r}}$. The prover wants to show that the same x *appears in the encryption and in the commitment*.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^3$ and $H := (\mathbb{G}, \cdot)^3$ (with component-wise group operations). The homomorphism φ is defined as

$$\varphi(x, r, \tilde{r}) := (g_1^r, g_1^x \cdot \texttt{pk}^r, g^x \cdot h^{\tilde{r}}).$$

This is a homomorphism since

$$\varphi((x, r, \tilde{r}) + (y, s, \tilde{s})) = \left(g_1^{r+s}, g_1^{x+y} \cdot \texttt{pk}^{r+s}, g^{x+y} \cdot h^{\tilde{r}+\tilde{s}}\right)$$
$$= \left(g_1^r, g_1^x \cdot \texttt{pk}^r, g^x \cdot h^{\tilde{r}}\right) \cdot \left(g_1^s, g_1^y \cdot \texttt{pk}^s, g^y \cdot h^{\tilde{s}}\right)$$
$$= \varphi(x, r, \tilde{r}) \cdot \varphi(y, s, \tilde{s}).$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, 0, 0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0, 0, 0) = (1, 1, 1) = (e_1, e_2, C)^q$.

Full protocol

Protocol com-enc-eq: PK $\left\{(x, r, \tilde{r}) : e_1 = g_1^r \wedge e_2 = g_1^x \texttt{pk}^r \wedge C = g^x h^{\tilde{r}}\right\}$

Implicit public values: $\mathbb{G}, q, \texttt{pk}, g_1, g, h$
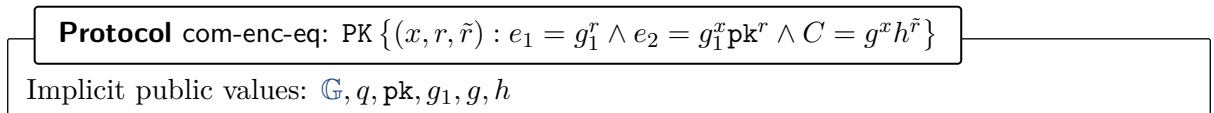
Prover$(x, r, \tilde{r})$		**Verifier**(e_1, e_2, C)

Choose random
$\alpha, \beta, \gamma \leftarrow \mathbb{Z}_q$ and compute
$a_1 = g_1^\beta$, $a_2 = g_1^\alpha \mathtt{pk}^\beta$,
$a_3 = g^\alpha h^\gamma$.

$$\xrightarrow{\quad (a_1, a_2, a_3) \quad}$$

Choose random
$c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$$\xleftarrow{\quad c \quad}$$

Compute $z_1 = \alpha - cx$
$(\mathrm{mod}\ q)$, $z_2 = \beta - cr$
$(\mathrm{mod}\ q)$, $z_3 = \gamma - c\tilde{r}$
$(\mathrm{mod}\ q)$.

Accept iff $a_1 = g_1^{z_2} \cdot e_1^c$,
$a_2 = g_1^{z_1} \mathtt{pk}^{z_2} \cdot e_2^c$,
$a_3 = g^{z_1} h^{z_3} \cdot C^c$.

$$\xrightarrow{\quad (z_1, z_2, z_3) \quad}$$

ℹ️ Note that the proof can trivially be extended to the case where the ElGamal encryption and the commitment are over different groups.

9.2.8 Proof of ElGamal Decryption (Without Revealing the Message)

Let $\mathbb{G}$ be a group of prime order q with generators g, h, and let $(e_1, e_2) = (g^r, h^x \cdot \mathtt{pk}_{\mathtt{EG}}^r)$ be an ElGamal encryption of x in the exponent under public key $\mathtt{pk}_{\mathtt{EG}} = g^{\mathtt{dk}_{\mathtt{EG}}}$ (see Section 7.1.4). The prover wants to show knowledge of $\mathtt{dk}_{\mathtt{EG}}$ and x such that (e_1, e_2) decrypts to x under $\mathtt{dk}_{\mathtt{EG}}$.

In contrast to the protocol in Section 9.2.7, the ciphertext (e_1, e_2) here has not necessarily been created by the prover, so the prover may not know the randomness used to encrypt. We therefore have to use a different proof strategy. The basic idea is to show that $e_2 = h^x \cdot e_1^{\mathtt{dk}_{\mathtt{EG}}}$.

⚠️ Note that this proof alone is not very meaningful because different decryption keys may decrypt the same ciphertext to different values. Hence, this proof should be combined with a proof that $\mathtt{dk}_{\mathtt{EG}}$ is the "correct" private key, matching a known public key of the prover.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^2$ (with component-wise group operation) and $H := (\mathbb{G}, \cdot)$. The homomorphism φ is defined as
$$\varphi_{e_1}(\mathtt{dk}_{\mathtt{EG}}, x) := h^x \cdot e_1^{\mathtt{dk}_{\mathtt{EG}}}.$$
Assuming $\mathtt{pk}_{\mathtt{EG}} = g^{\mathtt{dk}_{\mathtt{EG}}}$ and $e_1 = g^r$, knowing a preimage $(\mathtt{dk}_{\mathtt{EG}}, x)$ of e_2 under φ_{e_1} shows that
$$e_2 = \varphi_{e_1}(\mathtt{dk}_{\mathtt{EG}}, x) = h^x \cdot e_1^{\mathtt{dk}_{\mathtt{EG}}} = h^x \cdot (g^r)^{\mathtt{dk}_{\mathtt{EG}}} = h^x \cdot (g^{\mathtt{dk}_{\mathtt{EG}}})^r = h^x \cdot \mathtt{pk}_{\mathtt{EG}}^r.$$

The function φ_{e_1} is a homomorphism since
$$\varphi_{e_1}\big((\mathtt{dk}_{\mathtt{EG}}, x) + (\mathtt{dk}_{\mathtt{EG}}', x')\big) = \left(h^{x+x'} \cdot e_1^{\mathtt{dk}_{\mathtt{EG}} + \mathtt{dk}_{\mathtt{EG}}'} \right)$$
$$= \left(h^x \cdot e_1^{\mathtt{dk}_{\mathtt{EG}}} \right) \cdot \left(h^{x'} \cdot e_1^{\mathtt{dk}_{\mathtt{EG}}'} \right)$$
$$= \varphi_{e_1}(\mathtt{dk}_{\mathtt{EG}}, x) \cdot \varphi_{e_1}(\mathtt{dk}_{\mathtt{EG}}', x').$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0,0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0,0) = 1 = e_2^q$.

Full protocol

Protocol elg-dec: PK $\left\{ (\mathsf{dk_{EG}}, x) : e_2 = h^x \cdot e_1^{\mathsf{dk_{EG}}} \right\}$

Implicit public values: $\mathbb{G}, q, h, e_1$

Prover$(\mathsf{dk_{EG}}, x)$ **Verifier**(e_2)

Choose random $\alpha, \beta \leftarrow \mathbb{Z}_q$
and compute $a := h^\beta \cdot e_1^\alpha$. $\xrightarrow{\quad\quad a \quad\quad}$

                                                   Choose random
                         $\xleftarrow{\quad\quad c \quad\quad}$ $c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

Compute $z_1 := \alpha - c \cdot \mathsf{dk_{EG}}$
$(\bmod\ q)$, $z_2 := \beta - c \cdot x$
$(\bmod\ q)$. $\xrightarrow{\quad (z_1, z_2) \quad}$ Accept iff $a = h^{z_2} e_1^{z_1} \cdot e_2^c$.
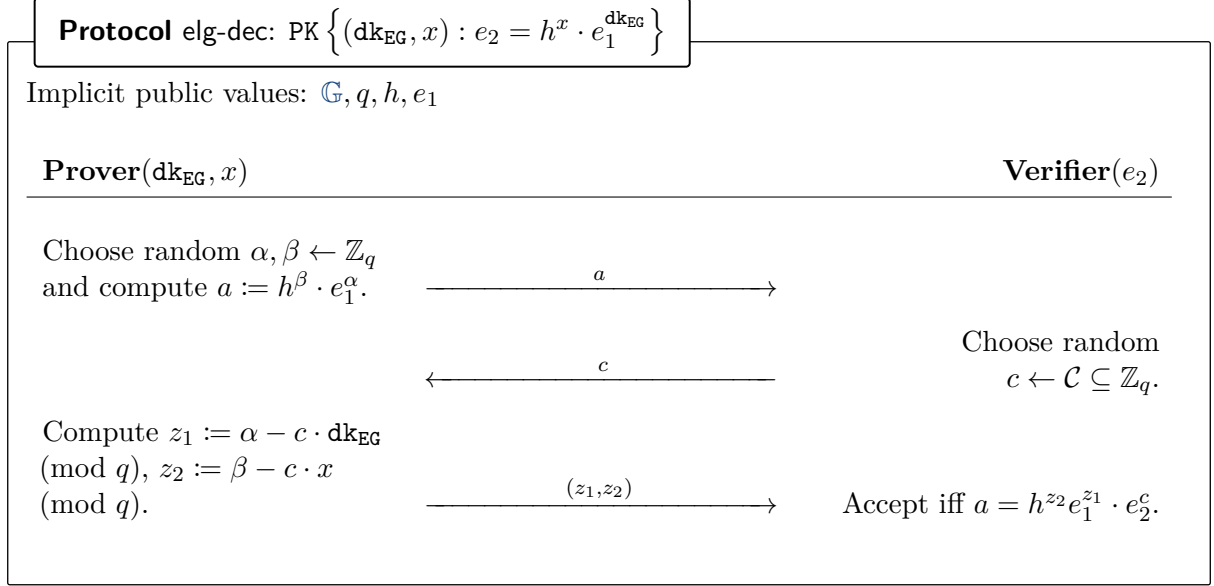
9.2.9 Proof of Equality for Commitments in Different Groups

Let $\mathbb{G}$ and $\overline{\mathbb{G}}$ be groups of prime order q. Let g_1, g_2 be generators of $\mathbb{G}$, and let g, h be generators of $\overline{\mathbb{G}}$. The prover has produced $y = g_1^{x_1} g_2^{x_2}$ which can be seen as a commitment to x_1 with randomness x_2. The prover has also produced a commitment $C = g^{x_1} h^r$. The prover then wants to show that y and C are *commitments* to the same value.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^3$ and $H := (\mathbb{G}, \cdot)^2$ (with component-wise group operations). The homomorphism φ is defined as

$$\varphi(x_1, x_2, r) := \left(g_1^{x_1} g_2^{x_2}, g^{x_1} h^r \right).$$

This is a homomorphism since

$$
\begin{aligned}
\varphi((x_1, x_2, r) + (y_1, y_2, s)) &= \left(g_1^{x_1+y_1} g_2^{x_2+y_2}, g^{x_1+y_1} h^{r+s} \right) \\
&= \left(g_1^{x_1} g_2^{x_2}, g^{x_1} h^r \right) \cdot \left(g_1^{y_1} g_2^{y_2}, g^{y_1} h^s \right) \\
&= \varphi(x_1, x_2, r) \cdot \varphi(y_1, y_2, s).
\end{aligned}
$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0,0,0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0,0,0) = (1,1) = (y, C)^q$.

Full protocol

> **Protocol** com-com-eq: PK $\{(x_1, x_2, r) : y = g_1^{x_1} g^{x_2} \wedge C = g^{x_1} h^r\}$

Implicit public values: $\mathbb{G}, \overline{\mathbb{G}}, q, g_1, g_2, g, h$

Prover(x_1, x_2, r) **Verifier**(y, C)

Choose random
$\alpha_1, \alpha_2, \beta \leftarrow \mathbb{Z}_q$ and
compute $a_1 = g_1^{\alpha_1} g_2^{\alpha_2}$ and
$a_2 = g^{\alpha_1} h^\beta$.

$\xrightarrow{\quad (a_1, a_2) \quad}$

                                                Choose random
                                               $c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$\xleftarrow{\quad\quad c \quad\quad}$

Compute $s_i = \alpha_i - c x_i$
$\pmod{q}$, $t = \beta - cr$ Accept iff $a_1 = g_1^{s_1} g_2^{s_2} \cdot y^c$
$\pmod{q}$. $\xrightarrow{\quad (s_1, s_2, t) \quad}$ and $a_2 = g^{s_1} h^t \cdot C^c$.
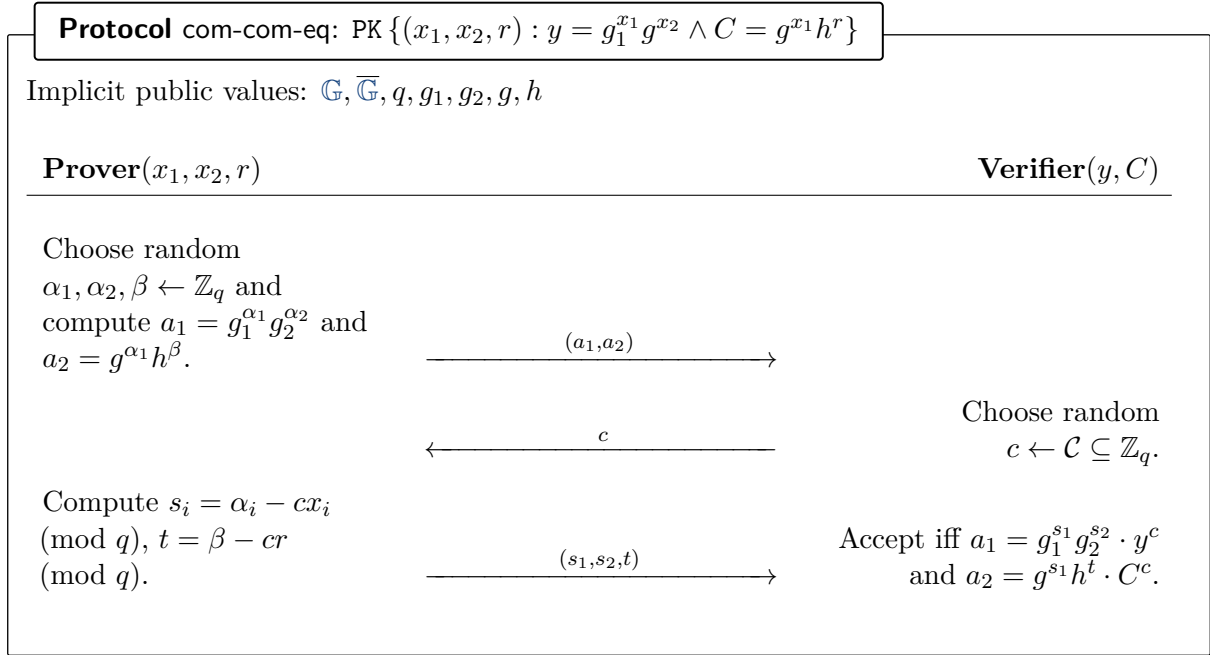
9.2.10 Proof of Multiplicative Relation on Commitments

Let $\mathbb{G}$ be a group of prime order q and let g and h be generators of $\mathbb{G}$. The prover has produced commitments $\{C_i = g^{x_i} h^{r_i}\}_{i=1,2,3}$ for values x_1, x_2, x_3. The prover wants to show that the committed values *satisfy* $x_1 x_2 = x_3 \pmod{q}$.

The protocol idea is as follows. To show the relation on the committed values we show that the first two commitments have the form $C_i = g^{x_i} h^{r_i}$ and that C_3 can be opened to $x_1 x_2$. The latter is done by showing that $C_3 = C_1^{x_2} h^r$ for $r = r_3 - r_1 x_2 \pmod{q}$. This of course assumes that the commitment scheme is binding.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^5$ and $H := (\mathbb{G}, \cdot)^3$ (with component-wise group operations). We then define the homomorphism φ_{C_1} relative to the publicly known commitment C_1 as

$$\varphi_{C_1}(x_1, x_2, r_1, r_2, r) := \left(g^{x_1} h^{r_1}, g^{x_2} h^{r_2}, C_1^{x_2} h^r \right).$$

Knowing a preimage (x_1, x_2, r_1, r_2, r) of (C_1, C_2, C_3) under φ_{C_1} shows that $C_1 = g^{x_1} h^{r_1}$, $C_2 = g^{x_2} h^{r_2}$, and $C_3 = C_1^{x_2} h^r = g^{x_1 x_2} h^{r_1 x_2 + r}$. Note that $r_3 := r_1 x_2 + r$ is uniformly distributed in G if that is the case for r, and vice versa.

The function φ_{C_1} is a homomorphism since

$$
\begin{aligned}
\varphi_{C_1}((x_1, x_2, r_1, r_2, r) + (y_1, y_2, s_1, s_2, s)) &= \left(g^{x_1 + y_1} h^{r_1 + s_1}, g^{x_2 + y_2} h^{r_2 + s_2}, C_1^{x_2 + y_2} h^{r+s} \right) \\
&= \left(g^{x_1} h^{r_1}, g^{x_2} h^{r_2}, C_1^{x_2} h^r \right) \cdot \left(g^{y_1} h^{s_1}, g^{y_2} h^{s_2}, C_1^{y_2} h^s \right) \\
&= \varphi_{C_1}(x_1, x_2, r_1, r_2, r) \cdot \varphi_{C_1}(y_1, y_2, s_1, s_2, s).
\end{aligned}
$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, 0, 0, 0, 0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi_{C_1}(0, 0, 0, 0, 0) = (1, 1, 1) = (C_1, C_2, C_3)^q$.

Full protocol

Protocol com-mult: PK $\{(x_1, x_2, r_1, r_2, r_3) : C_i = g^{x_i} h^{r_i} (i = 1, 2) \wedge C_3 = g^{x_1 x_2} h^{r_3}\}$

Implicit public values: $\mathbb{G}, q, g, h$

Prover$(C_1, x_1, x_2, r_1, r_2, r = r_3 - r_1 x_2 \pmod{q})$		**Verifier**(C_1, C_2, C_3)

Choose random
$\alpha_i, \tilde{r}_i, \tilde{r} \leftarrow \mathbb{Z}_q$ and compute
$a_i = g^{\alpha_i} h^{\tilde{r}_i}$ and $a = C_1^{\alpha_2} h^{\tilde{r}}$
for $i = 1, 2$.

$$\xrightarrow{\quad (a_1, a_2, a) \quad}$$

Choose random
$c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$$\xleftarrow{\quad c \quad}$$

Compute $s_i = \alpha_i - cx_i$
$\pmod{q}$, $t_i = \tilde{r}_i - cr_i$
$\pmod{q}$ for $i = 1, 2$,
$t = \tilde{r} - cr \pmod{q}$.

Accept iff $a_i = g^{s_i} h^{t_i} \cdot C_i^c$
for $i = 1, 2$, and
$a = C_1^{s_2} h^t \cdot C_3^c$.

$$\xrightarrow{\quad (s_1, s_2, t_1, t_2, t) \quad}$$
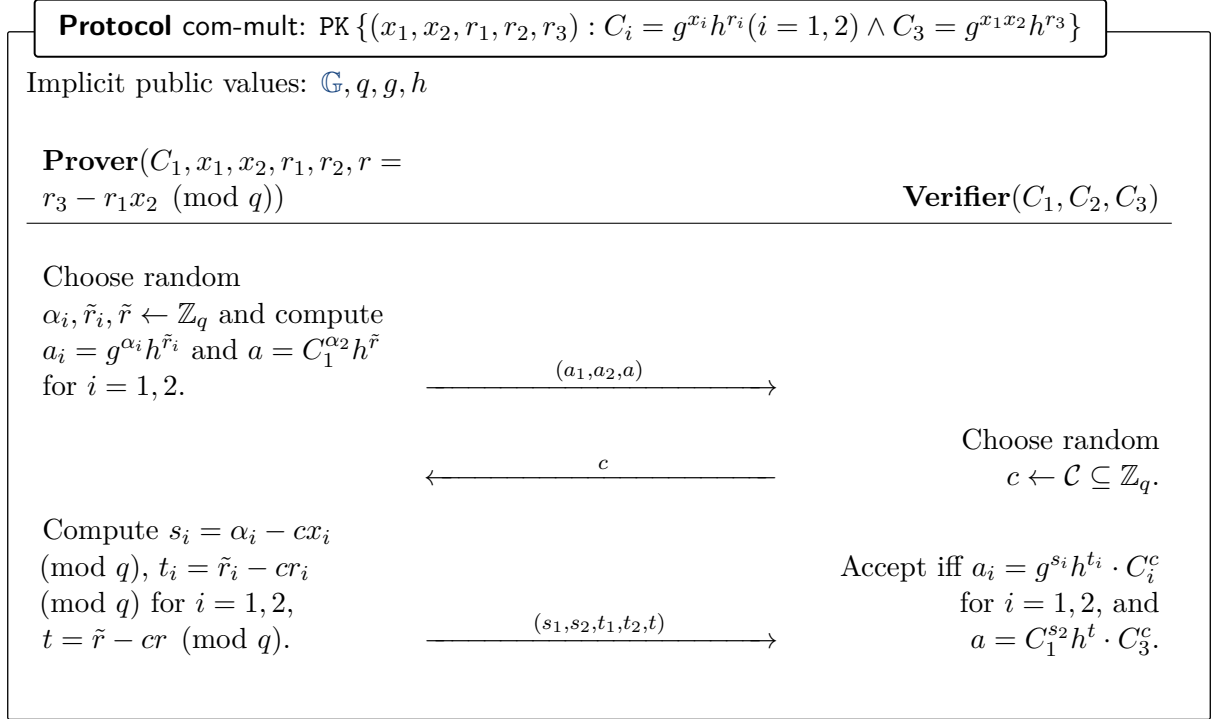
9.2.11 Linear Relationship of Committed Values

Let $\mathbb{G}$ be a group of prime order q and let g and h be generators of $\mathbb{G}$. The prover has produced commitments $\{C_i = g^{x_i} h^{r_i}\}_{i=1,\ldots,n}$ and $C = g^x h^r$ for values $x_1, \ldots, x_n$ and x, and wants to show that the committed values satisfy $\sum_{i=1}^{n} v_i x_i = x \pmod{q}$, for publicly known $v_i \in \mathbb{Z}_q$.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^{2n+1}$ and $H := (\mathbb{G}, \cdot)^{n+1}$ (with component-wise group operations). We then define the homomorphism φ as

$$\varphi(x_1, \ldots, x_n, r_1, \ldots, r_n, r) := \left(g^{x_1} h^{r_1}, \ldots, g^{x_n} h^{r_n}, g^{v_1 x_1 + \ldots + v_n x_n} h^r\right).$$

Knowing a preimage $(x_1, \ldots, x_n, r_1, \ldots, r_n, r)$ of $(C_1, \ldots, C_n, C)$ under φ shows that $C_1 = g^{x_1} h^{r_1}$, $\ldots$, $C_n = g^{x_n} h^{r_n}$, and $C = g^{v_1 x_1 + \ldots + v_n x_n} h^r$.

The function φ is a homomorphism since

$$\varphi((x_1, \ldots, x_n, r_1, \ldots, r_n, r) + (y_1, \ldots, y_n, s_1, \ldots, s_n, s))$$
$$= \left(g^{x_1 + y_1} h^{r_1 + s_1}, \ldots, g^{x_n + y_n} h^{r_n + s_n}, g^{v_1(x_1 + y_1) + \ldots + v_n(x_n + y_n)} h^{r+s}\right)$$
$$= \left(g^{x_1} h^{r_1}, \ldots, g^{x_n} h^{r_n}, g^{v_1 x_1 + \ldots + v_n x_n} h^r\right) \cdot \left(g^{y_1} h^{s_1}, \ldots, g^{y_n} h^{s_n}, g^{v_1 y_1 + \ldots + v_n y_n} h^s\right)$$
$$= \varphi(x_1, \ldots, x_n, r_1, \ldots, r_n, r) \cdot \varphi(y_1, \ldots, y_n, s_1, \ldots, s_n, s).$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, \ldots, 0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0, \ldots, 0) = (1, \ldots, 1) = (C_1, \ldots, C_n, C)^q$.

Full protocol

Protocol com-lin: PK $\left\{(x_1, \ldots, x_n, r_1, \ldots, r_n, r) : C_i = g^{x_i} h^{r_i} (i = 1, \ldots, n) \wedge C = g^{v x_1 + \ldots + v_n x_n} h^r\right\}$

Public values: $\mathbb{G}, q, g, h, v_1, \ldots, v_n$

Prover$(x_1, \ldots, x_n, r_1, \ldots, r_n, r)$	**Verifier**$(C_1, \ldots, C_n, C)$

Choose random
$\alpha_i, \tilde{r}_i, \tilde{r} \leftarrow \mathbb{Z}_q$ and compute
$a_i = g^{\alpha_i} h^{\tilde{r}_i}$ for $i = 1, \ldots, n$,
and $a = g^{v_1 \alpha_1 + \ldots + v_n \alpha_n} h^{\tilde{r}}$.

$$\xrightarrow{\quad (a_1, \ldots, a_n, a) \quad}$$

Choose random
$c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$.

$$\xleftarrow{\quad c \quad}$$

Compute $z_i = \alpha_i - c x_i$
$(\bmod\ q)$, $s_i = \tilde{r}_i - c r_i$
$(\bmod\ q)$ $(i = 1, \ldots, n)$,
$s = \tilde{r} - c r \pmod{q}$.

$$\xrightarrow{\quad (z_1, \ldots, z_n, s_1, \ldots, s_n, s) \quad}$$

Accept iff $a_i = g^{z_i} h^{s_i} \cdot C_i^c$
$(i = 1, \ldots, n)$ and
$a = g^{v_1 z_1 + \ldots + v_n z_n} h^s \cdot C^c$.

9.2.12 Proof of Nonzero for Committed Value

Let $(\overline{\mathbb{G}}, \cdot)$ be a group of prime order q, and let g and h be generators of $\overline{\mathbb{G}}$. The prover has produced a commitment $C = g^x h^r$ to the value x with randomness r (cf. Section 8.1), and wants to show that the committed value x is different from 0.

The *nonzero proof* can be reduced to the com-mult protocol (cf. Section 9.2.10) using the following Lemma.

Lemma 29.
$$\forall x \in \mathbb{Z}_q \quad x \neq 0 \iff \exists y \in \mathbb{Z}_q \ x \cdot y = 1$$

The prover first computes the inverse y of x, e.g. using the extended Euclidean algorithm, and computes the auxiliary commitments $C_2 := g^y h^s$. Denote by $C_3 := g^1 h^0$ the default commitment to 1. To prove nonzero x, the prover shows that the C_3 contains the product of the values in C_1 and C_2. That corresponds to protocol com-mult with the statement

$$\text{PK}\left\{(x, y, r, s, 0) : C = g^x h^r \wedge C_2 = g^y h^s \wedge C_3 = g^{xy} h^0\right\}$$

9.2.13 Proof of Inequality for Committed Value and Public Value

Let $(\overline{\mathbb{G}}, \cdot)$ be a group of prime order q, and let g and h be generators of $\overline{\mathbb{G}}$. The prover has produced a commitment $C = g^x h^r$ to the value x with randomness r (cf. Section 8.1), and wants to show that the *committed value x is different from a public value y.*

This can be reduced to the protocol for proving nonzero committed value from Section 9.2.12. Denote by $C' := g^y h^0$ the default commitment to y, then $x \neq y$ if $\widehat{C} := C - C' = g^{x-y} h^r$ contains a nonzero value.

9.2.14 Proof of Inequality for Committed Values

Let $(\mathbb{G}, \cdot)$ be a group of prime order q, and let g and h be generators of $\overline{\mathbb{G}}$. The prover has produced commitment $C = g^x h^r$ to the value x with randomness r (cf. Section 8.1), commitment $C' = g^y h^{r'}$ to the value y with randomness r', and wants to show that the *committed value x is different from committed value y*.

This can be reduced to the protocol for proving nonzero committed value from Section 9.2.12. We have $x \neq y$ if commitment $\widehat{C} := C - C' = g^{x-y} h^{r-r'}$ contains a nonzero value.

9.2.15 Encrypted Shares

Let $\mathbb{G}$ be a group of prime order q with generators $g, \bar{h}$. The prover has shared a value $x \in \mathbb{Z}_q$ using the Shamir sharing scheme in Section 4.1 and then encrypted each share using the ElGamal encryption scheme to encrypt $\bar{h}^{\mathsf{sh}(x)_i}$ (see Section 7.1.4 ignoring the splitting in parts) resulting in ciphertexts $c_1, \ldots, c_{\mathtt{n}}$. The prover wants to show that the *ciphertexts contain valid shares of a Shamir sharing*.

More precisely, given a set of public keys $\mathtt{pk}_{\mathrm{EG}_1}, \ldots, \mathtt{pk}_{\mathrm{EG}_{\mathtt{n}}}$ and supporting points $\alpha_1, \ldots, \alpha_{\mathtt{n}}$ the prover wants to show knowledge of $x, a_1, \ldots, a_{\mathtt{d}} \in \mathbb{Z}_q$ and $r_1, \ldots, r_{\mathtt{n}}$ such that

$$\forall i \in \{1, \ldots, \mathtt{n}\} \ c_i = (g^{r_i}, \bar{h}^{x + \sum_{j=1}^{\mathtt{d}} a_j \alpha_i^j} \mathtt{pk}_{\mathrm{EG}_i}^{r_i}).$$

Instantiation of abstract protocol

We set $G := (\mathbb{F}_q, +)^{\mathtt{d}+\mathtt{n}+1}$ and $H := (\mathbb{G}, \cdot)^{2(\mathtt{n})}$ (with component-wise group operations). We then define the homomorphism φ relative to public values $\alpha_1, \ldots, \alpha_{\mathtt{n}}$ and $\mathtt{pk}_{\mathrm{EG}_1}, \ldots, \mathtt{pk}_{\mathrm{EG}_{\mathtt{n}}}$ as

$$\varphi(x, a_1, \ldots, a_{\mathtt{d}}, r_1, \ldots, r_{\mathtt{n}}) := \left((g^{r_i}, \bar{h}^{x + \sum_{j=1}^{\mathtt{d}} a_j \alpha_i^j} \mathtt{pk}_{\mathrm{EG}_i}^{r_i}) \right)_{1 \leq i \leq \mathtt{n}}$$

Knowing a pre-image under φ for $c_1, \ldots, c_{\mathtt{n}}$ shows that the ciphertexts are of the form $c_i = (g^{r_i}, \bar{h}^{x + \sum_{j=1}^{\mathtt{d}} a_j \alpha_i^j} \mathtt{pk}_{\mathrm{EG}_i}^{r_i})$.

The function φ is a homomorphism since

$$\varphi((x, a_1, \ldots, a_{\mathtt{d}}, r_1, \ldots, r_{\mathtt{n}}) + (y, b_1, \ldots, b_{\mathtt{d}}, s_1, \ldots, s_{\mathtt{n}}))$$
$$= \left((g^{r_i + s_i}, \bar{h}^{x + y + \sum_{j=1}^{\mathtt{d}} (a_j + b_j) \alpha_i^j} \mathtt{pk}_{\mathrm{EG}_i}^{r_i + s_i}) \right)_{1 \leq i \leq \mathtt{n}}$$
$$= \left((g^{r_i}, \bar{h}^{x + \sum_{j=1}^{\mathtt{d}} a_j \alpha_i^j} \mathtt{pk}_{\mathrm{EG}_i}^{r_i}) \right)_{1 \leq i \leq \mathtt{n}} \cdot \left((g^{s_i}, \bar{h}^{y + \sum_{j=1}^{\mathtt{d}} b_j \alpha_i^j} \mathtt{pk}_{\mathrm{EG}_i}^{s_i}) \right)_{1 \leq i \leq \mathtt{n}}$$
$$= \varphi(x, a_1, \ldots, a_{\mathtt{d}}, r_1, \ldots, r_{\mathtt{n}}) \cdot \varphi(y, b_1, \ldots, b_{\mathtt{d}}, s_1, \ldots, s_{\mathtt{n}}).$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$. The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, \ldots, 0)$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0, \ldots, 0) = ((1, 1), \ldots, (1, 1)) = (c_1, \ldots, c_{\mathtt{n}})^q$.

Applying the abstract protocol from Section 9.1.1 to φ results in Σ-protocol shared-enc.

9.3 Non-interactive Proofs using the Fiat-Shamir Transformation

In this section we describe the *Fiat-Shamir Transformation* which allows us to get non-interactive versions of the above Σ-protocols. Since this transformation will later be used also for Bulletproofs protocols, the following description is not tied to Σ-protocols and is given for a more general class of *public-coin multi-round* protocols as defined below.

Public-Coin Multi-Round Proof Systems. Let $\Pi = (\mathcal{P}, \mathcal{V})$ be a k-round *interactive* proof system between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ that exchange massages as depicted in Figure 9.1. The prover speaks first and last; so the number of exchanged messages in a proof transcript $(a_0, c_1, a_1, \ldots, c_k, a_k)$ is the odd number $2k + 1$. The message c_i of $\mathcal{V}$ in the $2i$-th move is called the i-th challenge. We say that Π is *public-coin* if all the challenges c_i-s are chosen uniformly at random and independently of the prover's messages.

$$\mathcal{P} \qquad\qquad \mathcal{V}$$

$$\xrightarrow{\quad a_0 \quad}$$
$$\xleftarrow{\quad c_1 \quad}$$
$$\xrightarrow{\quad a_1 \quad}$$
$$\vdots$$
$$\xleftarrow{\quad c_k \quad}$$
$$\xrightarrow{\quad a_k \quad}$$
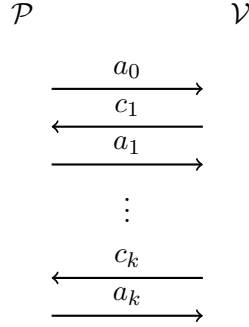
Figure 9.1 k-round interactive proof systems

Fiat-Shamir Transformation. The Fiat-Shamir transformation [FS86] turns a public-coin interactive proof system into a *non-interactive* proof system. The core idea of the transformation is to replace the verifier's random choices (i.e., *challenges*) with the output of a random oracle. More precisely, let $\mathsf{H_{RO}}$ be a hash function which will be used as random oracle (cf. Section 3.1.3). For a k-round public-coin interactive proof, the idea is to compute the i-th challenge c_i as the hash of the statement and all previous prover messages.

Definition 30. Let $\Pi = (\mathcal{P}, \mathcal{V})$ be a k-round public-coin interactive proof system. The Fiat-Shamir transformation of Π is a non-interactive proof system where the prover runs $\mathcal{P}$ but instead of asking $\mathcal{V}$ for the challenge c_i on message a_{i-1}, it computes the challenge as

$$c_i = \mathsf{H_{RO}}(y||a_0||\ldots||a_{i-1})$$

where y is the input statement. The output of the prover is a proof $(a_0, \ldots, a_k)$. On input a statement y and a proof $(a_0, \ldots, a_k)$, the verifier accepts the proof if, for c_i computed as above, $\mathcal{V}$ accepts the transcript $(a_0, c_1, a_1, \ldots, c_k, a_k)$ on input y.

> ⚠ It is important that the public statement y is part of the hash input when computing the challenge c_i. Using just $c_i = \mathsf{H_{RO}}(a_0||\ldots||a_{i-1})$ results in an *insecure* Fiat-Shamir transformation.

> ❗ For simplicity, we assume here that the hash function $\mathsf{H_{RO}}$ maps uniformly to the challenge space $\mathcal{C}$. See Section 9.4.3 for a discussion of the more general case.

> ℹ A Σ-protocol can be seen as a 1-round public-coin interactive proof system, hence the challenge in the non-interactive version is computed as $c = \mathsf{H_{RO}}(y||a)$.

> ℹ Compared to the above generic verification, there exists a more space efficient verification approach for a subset of Σ-protocols that is explained in Section 9.4.3.

Making proofs context-sensitive. In some applications it can be desirable to tie a non-interactive proof to a context, e.g., to prevent the reuse of a proof. We assume that the *context* can be described as a bit string (e.g., the hash of the genesis block). To make a context-sensitive non-interactive proof, the context string `ctx` is used as an additional input to the challenge hash, i.e., the challenges are computed as $c_i = \mathsf{H}_{\mathsf{RO}}(\mathtt{ctx}||y||a_0||\ldots||a_{i-1})$.

For example, the context for a set membership proof (cf. Section 9.5.4) could be "SetMember-ShipProof". If the proof needs to be tied to a specific blockchain, the hash of the genesis block could be part of the context string.

> </> In the implementation, Fiat-Shamir hashing is done in a transcript like manner. When adding values to the transcript a label must be provided that gives context on the added value. This prevents malleability issues that can occur in a naïve implementation.

9.4 Sigma Protocol Interface

In this section we describe the Σ-protocol *interface*. It allows generating and combine Σ-protocols, compile them into context-sensitive non-interactive proofs, and verify them.

9.4.1 Sigma Protocol Proof Information

We first define the concept of proof information which contains the raw information required to generate and verify proofs.

Definition 31. A Σ-*protocol proof information* $\varsigma \in \Sigma\text{-}\textsc{Informations}$ consists of

- a group homomorphism $\varphi\colon G \to H$,
- a challenge space $\mathcal{C}$,
- and a public value $y \in H$.

Together with a *witness* $x \in G$, a ς can be compiled into a non-interactive zero-knowledge proof, cf. Section 9.4.3.

The homomorphism and the challenge space are determined by the statement one wants to prove together with the shared public values (e.g., the used generator of the group). The basic statements have been described in Section 9.2.

Definition 32. A Σ *statement* is a description of the statement to be proven from which one can extract the corresponding group homomorphism φ and the challenge space $\mathcal{C}$. This homomorphism implicitly also defines the groups G and H, determining the types $\Sigma\text{-}\textsc{Witnesses}$ for the *witness*, type $\Sigma\text{-}\textsc{publicInputs}$ for the *public input*, and $\Sigma\text{-}\textsc{sharedPublicValues}$ for the *shared public values*.

For our purposes, we have (see Section 9.2)

$$\Sigma\text{-}\textsc{Statements} = \{\mathsf{id}, \mathsf{dlog}, \mathsf{agg\text{-}dlog}, \mathsf{com\text{-}dlog\text{-}eq}, \mathsf{dlog\text{-}dlog\text{-}eq}, \mathsf{com\text{-}enc\text{-}eq}, \mathsf{elg\text{-}dec},$$
$$\mathsf{com\text{-}com\text{-}eq}, \mathsf{com\text{-}mult}, \mathsf{com\text{-}lin}, \mathsf{shared\text{-}enc}\}.$$

Protocol genSigProofInfo

Inputs

   statement $\in \Sigma$-STATEMENTS

       A description of the statement to be proven.

   spub $\in \Sigma$-SHAREDPUBLICVALUES

       The shared public values.

   $y \in \Sigma$-PUBLICINPUTS

       The public input.

Outputs

   $\varsigma \in \Sigma$-INFORMATIONS

       A Σ-protocol proof information.

Implementation

1: Let φ and $\mathcal{C}$ be the group homomorphism and challenge space corresponding to statement and spub.

2: **return** $\varsigma := (\varphi, \mathcal{C}, y)$.

9.4.2 Combining Sigma Protocols

Using the techniques from Sections 9.1.2, 9.1.3, and 9.1.4, we can combine multiple Σ-protocol proof information to obtain proofs for new statements.

AND-composition. Given several Σ-protocols for different statements, one can combine them into a new Σ-protocol that shows all of these statements simultaneously, as described below.

Protocol genAndComp

Inputs

   $(\varsigma_1, \ldots, \varsigma_n) \in \Sigma$-INFORMATIONS*

       List of Σ-protocol proof information.

Outputs

   $\varsigma \in \Sigma$-INFORMATIONS

       A Σ-protocol proof information for the AND-composition of $\varsigma_1, \ldots, \varsigma_n$.

Promise

     $1/|\mathcal{C}_1 \cap \ldots \cap \mathcal{C}_n|$ is negligible.

Implementation

1: Let $(\varphi_i, \mathcal{C}_i, y_i) = \varsigma_i$ for $i = 1, \ldots, n$.

2: $\varphi := \varphi_1 \times \ldots \times \varphi_n$

3: $\mathcal{C} := \mathcal{C}_1 \cap \ldots \cap \mathcal{C}_n$

4: $y := (y_1, \ldots, y_n)$

5: **return** $\varsigma := (\varphi, \mathcal{C}, y)$

Proving equality of values. The following allows to prove that certain preimages of the homomorphism are equal. This will be particularly useful in combination with the AND-composition.

Protocol genEqComp

Inputs

$\varsigma \in \Sigma\text{-Informations}$
    A Σ-protocol proof information with witness vector of length n.

$L \in \{1, \ldots, n\}^*$
    A subset of $\{1, \ldots, n\}$.

Outputs

$\varsigma' \in \Sigma\text{-Informations}$
    A Σ-protocol proof information for the same statement as ς, additionally showing that the prover knows witness $x_1, \ldots, x_n$ such that all x_l for $l \in L$ are equal.

Promise

    The domain of the homomorphism in ς is $G_1 \times \ldots \times G_n$ with $G_i = G_j$ for all $i, j \in L$.

Implementation

1: Let $(\varphi, \mathcal{C}, y) = \varsigma$.
2: $\varphi' := \mathrm{EqH}(\varphi, L)$ // See Definition 25.
3: **return** $\varsigma' := (\varphi', \mathcal{C}, y)$

Proving linear relation of values. It will be useful to be able to prove that certain values satisfy a linear relation. We first introduce a generic variant that allows to show that some x_{i_0} satisfies $x_{i_0} = \sum_{i=1}^{m} v_i t_i$ for given v_i and newly introduced witness variables t_i.

Protocol genLinRelComp

Inputs

$\varsigma \in \Sigma\text{-Informations}$
    A Σ-protocol proof information with witness vector of length n.

$i_0 \in \{1, \ldots, n\}$
    An element of $\{1, \ldots, n\}$.

$(v_1, \ldots, v_m) \in \mathbb{Z}^m$
    Coefficients of the linear relation.

Outputs

$\varsigma' \in \Sigma$-INFORMATIONS

A Σ-protocol proof information for the same statement as ς, additionally showing that the prover knows witness vector $(x_1, \ldots, x_n)$ and (newly introduced) $t_1, \ldots, t_m$ such that all $x_{i_0} = \sum_{i=1}^m v_i t_i$.

Promise

The domain of the homomorphism in ς is $G_1 \times \ldots \times G_n$.

Implementation

1: Let $(\varphi, \mathcal{C}, y) = \varsigma$.
2: $\varphi' := \text{LinH}(\varphi, i_0, (v_1, \ldots, v_m))$ *// See Definition 27.*
3: **return** $\varsigma' := (\varphi', \mathcal{C}, y)$

ℹ️ This transformation adds new $t_1, \ldots, t_m$ to the witness. Since nothing else is required from these t_i, the proof on its own is not very meaningful. It can, however, be combined with genEqComp to show that these t_i are equal to some other values.

A common use case for proving linear relations is showing that the x_i satisfy a certain linear relation. This can be obtained from genLinRelComp by additionally proving that the t_j are equal to those x_i using genEqComp. We provide a protocol that combines these two steps below.

Protocol genLinRelCompEx

Inputs

$\varsigma \in \Sigma$-INFORMATIONS

A Σ-protocol proof information with witness vector of length n.

$L \in \{1, \ldots, n\}^*$

A subset of $\{1, \ldots, n\}$ determining which values are involved in the linear relation.

$i_0 \in \{1, \ldots, n\}$

An element of $\{1, \ldots, n\}$.

$(v_l)_{l \in L} \in \mathbb{Z}^{|L|}$

Coefficients of the linear relation.

Outputs

$\varsigma' \in \Sigma$-INFORMATIONS

A Σ-protocol proof information for the same statement as ς, additionally showing that the prover knows witness $(x_1, \ldots, x_n)$ such that all $x_{i_0} = \sum_{l \in L} v_l x_l$.

Promise

The domain of the homomorphism in ς is $G_1 \times \ldots \times G_n$ with $G_l = G_{i_0}$ for all $l \in L$.

Implementation

1: $\varsigma' := \text{genLinRelComp}(\varsigma, i_0, (v_l)_{l \in L})$
2: let $(l_1, \ldots, l_{|L|}) := L$
3: **for** $j = 1, \ldots, |L|$ **do**
4:     $\varsigma' := \text{genEqComp}(\varsigma', \{l_j, n + j\})$ *// show equality of l_j and new variable t_j*

> ℹ This transformation also does not change the witness. Similarly to genEqComp, one can as optimization skip x_{i_0} since it is determined by the other values.

9.4.3 Generating Non-Interactive Proofs

Using the Fiat-Shamir transform described in Section 9.3, one can compile a Σ-protocol proof information together with the corresponding witness into a non-interactive zero-knowledge proof.

Challenge mapping. In Section 9.3, we assumed that the hash function $\mathsf{H_{RO}}$ maps to group elements. In reality, $\mathsf{H_{RO}}$ maps to bit strings. Thus, an additional *challenge map* $\mathsf{getChallenge}_{\mathcal{C}}$ is needed to map the challenge bit string of the non-interactive proof to an element of the actual challenge space.

> ⚠ Assuming that $\mathsf{H_{RO}}$ maps uniformly to k-bit strings, the challenge map $\mathsf{getChallenge}_{\mathcal{C}}$ shall map k-bit strings uniformly on to the challenge space $\mathcal{C}$.

In all of our Σ-protocols, the challenge space $\mathcal{C}$ is a subset of $\mathbb{Z}_q$. A simple choice of challenge space $\mathcal{C}$ and map $\mathsf{getChallenge}_{\mathcal{C}}$ is as follows. The mapping $\mathsf{getChallenge}_{\mathcal{C}}$ takes the first $\lfloor \log_2 q \rfloor$ bits of the challenge hash and interprets it as an element in $\mathbb{Z}_q$. That is, the image of $\mathsf{getChallenge}_{\mathcal{C}}$ is $\mathcal{C} = \mathbb{Z}_{2^k}$ for the largest k such that $2^k < q$.

Non-interactive proof. For a given Σ-protocol proof information $\varsigma = (\varphi, \mathcal{C}, y)$, a non-interactive sigma proof is defined as follows.

Definition 33. A *non-interactive sigma proof* is a tuple $(h, z) \in \Sigma\text{-}\mathrm{PROOFS} := (\{0,1\}^*, H)$ where H is the co-domain of φ.

Proof generation. The following protocol allows to generate a non-interactive proof.

Protocol genSigmaNIZKProof

Inputs

  $\mathtt{ctx} \in \{0,1\}^*$
    A context-string.
  $\varsigma = (\varphi, \mathcal{C}, y) \in \Sigma\text{-}\mathrm{INFORMATIONS}$
    A Σ-protocol proof information.
  $x \in \Sigma\text{-}\mathrm{WITNESSES}$
    The corresponding witness.

Outputs

  $\pi = (h, z) \in \Sigma\text{-}\mathrm{PROOFS}$
    A non-interactive zero-knowledge proof.

Implementation

  1: Let G be the domain of φ
  2: Choose uniformly random $\alpha \leftarrow G$
  3: $a := \varphi(\alpha)$
  4: $h := \mathsf{H_{RO}}(\mathtt{ctx}||y||a)$
  5: Set the challenge to $c := \mathsf{getChallenge}_{\mathcal{C}}(h)$

6: Compute the response $z := \alpha - cx$
7: **return** non-interactive zero-knowledge proof $\pi := (h, z)$

Recall that in our abstract Σ-protocol the check done by the verifier is of the form

$$a \overset{?}{=} \varphi(z)y^c.$$

This allows to compute the "right" a given (c, z) and makes the protocol what we call *A-extractable*. So it is fine to just use (c, z) as proof instead of the more common (a, z).

Proof verification. The following protocol allows to verify a non-interactive proof.

Protocol verifySigmaNIZKProof

Inputs

    ctx $\in \{0, 1\}^*$
        A context-string.

    $\varsigma = (\varphi, \mathcal{C}, y) \in \Sigma\text{-}\textsc{Informations}$
        A Σ-protocol proof information.

    $\pi = (h, z) \in \Sigma\text{-}\textsc{Proofs}$
        A non-interactive zero-knowledge proof.

Outputs

    $b \in \textsc{bool}$
        Returns true iff the proof is valid in this context.

Implementation

1: $c := \mathsf{getChallenge}_{\mathcal{C}}(h)$
2: $a := \varphi(z) \cdot y^c$
3: **return true** iff $h = \mathsf{H}_{\mathsf{RO}}(\mathtt{ctx}||y||a)$.

> ⓘ   As mentioned above, we use (c, z) as proof instead of the more common (a, z) or even the full tuple (a, c, z) described in Section 9.3. This is fine as long as the Σ-protocol is A-extractable. The advantage of using (c, z) is space efficiency.
> For non-A-extractable Σ-protocols we would have to use (a, z). In the verification, one would instead compute $c = \mathsf{H}_{\mathsf{RO}}(y||a)$ and then check if $a = \varphi(z)y^c$.

9.4.4 OR composition

In this section we describe the OR-composition technique developed in [CDS94] that allows a prover to prove that given Σ-protocol statements $y_1, \ldots, y_n$, it knows a witness x_s for y_s without revealing s. In other words, x_s is a valid witness for y_1, or for y_2, ..., or for y_n. While the technique allows OR-composition of any statements admissible by Maurer's abstraction [Mau15], the resulting protocol itself is not of this type as OR statements cannot be represented as knowledge of a pre-image of a group homomorphism.

Given group homomorphisms $\varphi_1 \colon G_1 \to H_1, \ldots, \varphi_n \colon G_n \to H_n$, and $(y_1, y_2, \ldots, y_n) \in H_1 \times H_2 \times \ldots \times H_n$, the protocol allows proving knowledge of a preimage x_s of y_s under φ_s, for

some (secret) index $s \in \{1, \ldots, n\}$. The idea at a high level is to let the prover who holds x_s complete n instances of Σ-protocols for given statements $(y_1, \ldots, y_n)$, so that the proof for "knowledge of $x_s = \varphi_s^{-1}(y_s)$" part is real, whereas the proof for "knowledge of $\varphi_i^{-1}(y_i)$" (for all $i \neq s$) part is fake, generated by the Σ-protocols simulators. Since the real part and fake parts are indistinguishable, no one can learn for which statement the prover holds a witness.

In more detail, the protocol works as follows. Let $\Sigma_1, \ldots, \Sigma_n$ be n Σ-protocols, respectively for NP relations that are defined by group homomorphisms $\varphi_1, \ldots, \varphi_n$. The protocols are assumed to be special honest verifier zero-knowledge and hence are equipped with simulators $\mathcal{S}_1, \ldots, \mathcal{S}_n$. The prover $\mathcal{P}$ who has a witness x_s for y_s invokes the simulator $\mathcal{S}_i$ of Σ_i on y_i for $i \in \{1, \ldots, n\} \setminus \{s\}$, and obtains the simulated transcripts $\{(a_i, c_i, z_i)\}_{i \in \{1, \ldots, n\} \setminus \{s\}}$. $\mathcal{P}$ also computes the first message a_s of the real transcript and sends $(a_1, \ldots, a_n)$ to the verifier. The verifier sends a challenge c to the prover who first decomposes it as $c = c_s \bigoplus_{i \neq s} c_i$ and then computes the third message z_s of the real transcript according to (a_s, c_s, x_s). $\mathcal{P}$ finally sends $(z_1, \ldots, z_n)$ along with $(c_1, \ldots, c_{n-1})$ to the verifier. The verifier computes $c_n = c \bigoplus_{i=1}^{n-1} c_i$ and checks the validity of all transcripts $\{(a_i, c_i, z_i)\}_{i \in \{1, \ldots, n\}}$.

We provide the full protocol for proving disjunctions below. Note that as challenge space, one can use the intersection $\mathcal{C} := \bigcap_{i=1}^{n} \mathcal{C}_i$ of challenge spaces used in the individual proofs.

Protocol PK $\{(x_s) : \bigvee_{i=1}^{n} y_i = \varphi_i(x_s)\}$

Implicitly known public values: $G_1, H_1, \varphi_1, \ldots, G_n, H_n, \varphi_n, \mathcal{C}$.

Prover(x_s) **Verifier**$(y_1, \ldots, y_n)$

Invoke the simulators $\mathcal{S}_i$
for $i \in \{1, \ldots, n\} \setminus \{s\}$:

$\{(a_i, c_i, z_i) \leftarrow \mathcal{S}_i(y_i)\}_{i \neq s}$

Choose uniformly random
$\alpha_s \leftarrow G_s$ and compute
$a_s := \varphi_s(\alpha_s)$.

$$\xrightarrow{\quad a_1, \ldots, a_n \quad}$$

Choose uniformly random
$c \leftarrow \mathcal{C}$.

$$\xleftarrow{\quad c \quad}$$

Compute $c_s := c \bigoplus_{i \neq s} c_i$.

Compute $z_s := \alpha_s - c_s x_s$.

$$\xrightarrow{\quad z_1, \ldots, z_n, c_1, \ldots, c_{n-1} \quad}$$

Compute $c_n = c \bigoplus_{i=1}^{n-1} c_i$.

Accept iff $a_i = \varphi_i(z_i) \cdot y_i^{c_i}$
for $i = 1, \ldots, n$.
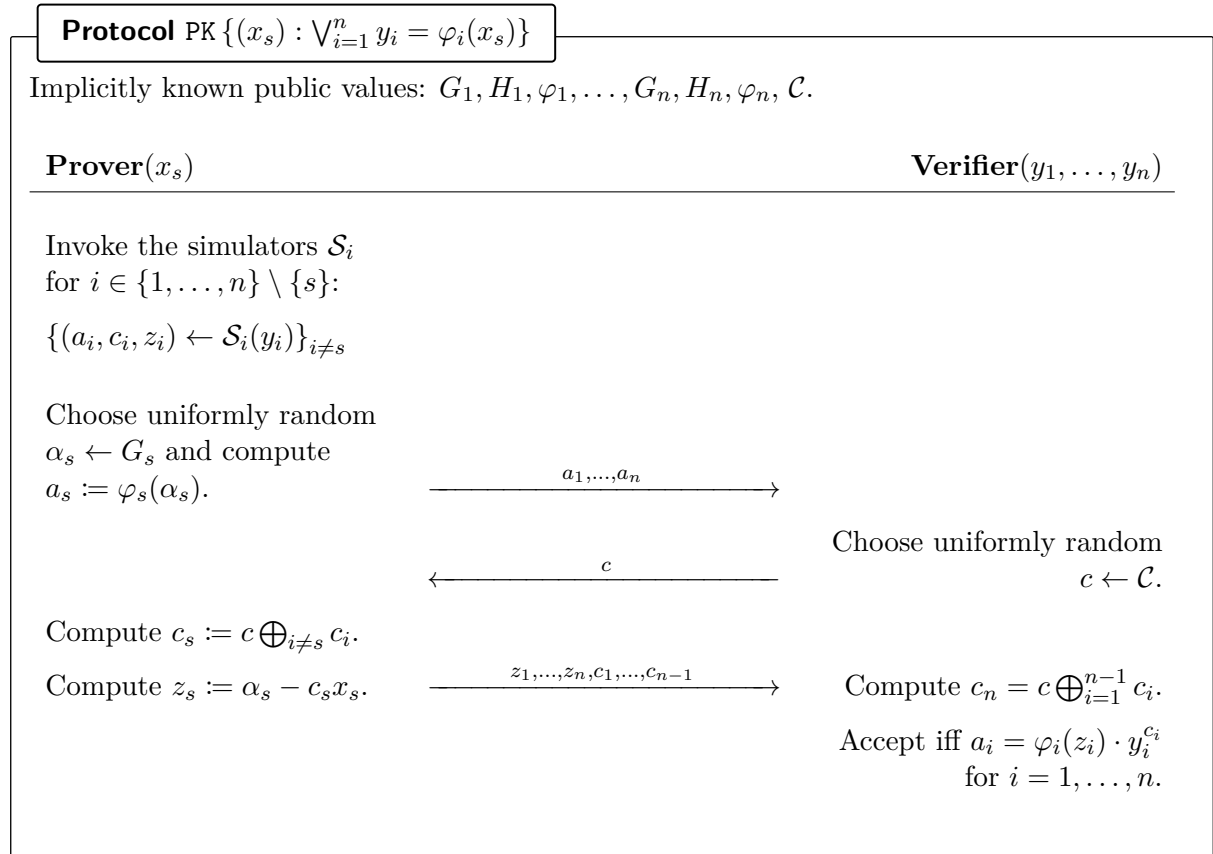
> ⓘ   The reason that the prover does not send the last challenge c_n is to improve efficiency. A less efficient approach is for the prover to send all challenges $c_1, \ldots, c_n$ and then require the verifier to additionally check $c = \bigoplus_{i=1}^{n} c_i$.

We next show that the above protocol is a Σ-protocol for the OR relation.

Lemma 34. *The protocol above has* special soundness *and* special honest verifier zero-knowledge.

Proof. To show special soundness, the extractor takes two distinct accepting proof transcripts with the same first messages, and different challenges $c \neq c'$, i.e.,

$$(a_1, c_1, z_1, \ldots, a_n, c_{n-1}, z_n, c)$$
$$(a_1, c_1', z_1', \ldots, a_n, c_{n-1}', z_n', c').$$

Because $c \neq c'$, we have $c_k \neq c_k'$ for some $k \in \{1, \ldots, n\}$. Therefore, by special soundness of the underlying Σ-protocol, the extractor can compute a valid witness x from (a_k, c_k, z_k) and (a_k, c_k', z_k') such that $y_k = \varphi_k(x)$.

To show special honest verifier zero-knowledge, the simulator, given statement $(y_1, \ldots, y_n)$ and a challenge c, chooses random c_i's such that $c = \oplus_{i=1}^n c_i$. It then runs $\mathcal{S}_i$ on (y_i, c_i) for all $i \in \{1, \ldots, n\}$.

$\square$

> **ⓘ** A Σ-protocol can be seen as a 1-round public-coin interactive proof system. A more generic OR-composition protocol for multi-round protocols is given in Section 9.6.1.

9.5 Bulletproofs

Bulletproofs [Bün+18] are short zero-knowledge proofs that can be used to prove that a value is in a range or in a given *set*. At the core is a proof for the inner product relation that has logarithmic size in the vector length.

9.5.1 Notation

In this section let $\mathbb{G}$ be a group of prime order q.

Vectors. In Bulletproofs vectors play an important role, we therefore repeat and extend the vector notation defined in Section 2.1.3. We use bold font to denote vectors, i.e., $\mathbf{a} \in \mathbb{Z}_q^n$ is a vector with elements $a_0, \ldots, a_{n-1}$ in $\mathbb{Z}_q$. Throughout the section we assume that the dimension n is a power of 2.[3] For a scalar $c \in \mathbb{Z}_q$ and a vector $\mathbf{a} \in \mathbb{Z}_q^n$, we represent by $\mathbf{b} = c \cdot \mathbf{a} \in \mathbb{Z}_q^n$ the vector where $b_i = c \cdot a_i$. For two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$, their inner product is denoted by $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=0}^{n-1} a_i \cdot b_i$, and their Hadamard product is denoted by $\mathbf{a} \circ \mathbf{b} = (a_0 \cdot b_0, \ldots, a_{n-1} \cdot b_{n-1}) \in \mathbb{Z}_q^n$. The concatenation of two vectors $\mathbf{a} \in \mathbb{Z}_q^n$ and $\mathbf{b} \in \mathbb{Z}_q^m$ is denoted by $\mathbf{a} || \mathbf{b} \in \mathbb{Z}_q^{n+m}$.

For $n' = \frac{n}{2}$, we define $\mathbf{a}_{\text{lo}} \in \mathbb{Z}_q^{n'}$ and $\mathbf{a}_{\text{hi}} \in \mathbb{Z}_q^{n'}$ to be respectively the low and high halves of $\mathbf{a} \in \mathbb{Z}_q^n$.

$$\mathbf{a}_{\text{lo}} = (a_0, \ldots, a_{n'-1}) \in \mathbb{Z}_q^{n'}, \qquad \mathbf{a}_{\text{hi}} = (a_{n'}, \ldots, a_{n-1}) \in \mathbb{Z}_q^{n'}.$$

Moreover, we denote a slice $\mathbf{a}_{[i:j]}$ of $\mathbf{a}$ as $(a_i, a_{i+1} \ldots, a_j) \in \mathbb{Z}_q^{j-i+1}$.

For $k \in \mathbb{Z}_q$, we use $\mathbf{k}^n$ to denote the vector containing the first n powers of k, i.e., $\mathbf{k}^n = (1, k, k^2, \ldots, k^{n-1}) \in \mathbb{Z}_q^n$. Equivalently for $k \neq 0$, $\mathbf{k}^{-n} = (\mathbf{k}^{-1})^n$. Denote by $\mathbf{e}_j \in \mathbb{Z}_q^n$ the unit vector in direction j, i.e. the vector that is all zero except at component j where it is 1.

Similarly, we define for vectors $\mathbf{g} = (g_0, \ldots, g_{n-1}), \mathbf{h} = (h_0, \ldots, h_{n-1}) \in \mathbb{G}^n$, the Hadamard product $\mathbf{g} \circ \mathbf{h} = (g_0 h_0, \ldots, g_{n-1} h_{n-1})$ and for $n \in \mathbb{Z}$ let $\mathbf{g}^n = (g_0^n, \ldots, g_{n-1}^n)$. Finally, we define *multi-exponentiation* of vector $\mathbf{g}$ with scalars $\mathbf{a} \in \mathbb{Z}_q$ by $\mathbf{g} \uparrow \mathbf{a} := \prod_j g_j^{a_j}$.

---

[3]One can easily pad the inputs to ensure that this holds.

Commitments. Let $g, h, \mathbf{g} = (g_0, \ldots, g_{n-1}), \mathbf{h} = (h_0, \ldots, h_{n-1})$ be generators of the $\mathbb{G}$. Group elements that represent commitments are capitalized, and blinding factors are denoted with a tilde, i.e., $\mathsf{C} = g^a h^{\tilde{r}} \in \mathbb{G}$ is a Pedersen commitment to a with blinding factor $\tilde{r}$. We denote a Pedersen commitment to a (with blinding $\tilde{r}$) by $\mathsf{Com}(a; \tilde{r})$.

We also define the vector commitment $\mathsf{VCom}(\mathbf{a}, \mathbf{b}; \tilde{r}) := (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) h^{\tilde{r}}$. If the generators need to be mentioned explicitly we use $\mathsf{VCom}(\mathbf{a}, \mathbf{b}; \tilde{r}; \mathbf{g}, \mathbf{h}, h) := (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) h^{\tilde{r}}$.

> ⚠ The generators $g, h, g_0, \ldots, g_{n-1}, h_0, \ldots, h_{n-1}$ must be generated such that no one knows the pairwise discrete logs.

9.5.2 Inner Product Proof

The core of Bulletproofs is an Inner Product Argument (IPA) that allows a prover to convince a verifier that an inner product on Pedersen commitments is correct. The IPA is based on a "divide-and-conquer" strategy, where the problem turns into smaller ones (i.e., half-size at each iteration) of the same type until it becomes of a small constant size that the prover can simply output everything and prove that the statement is correct.

The inputs to the IPA are independent generators $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, h \in \mathbb{G}$, a group element $\mathsf{P} \in \mathbb{G}$, and a scalar $c \in \mathbb{Z}_q$. The goal is for the prover to convince the verifier that she knows two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$ such that

$$\mathsf{P} = (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) \qquad \text{and} \qquad c = \langle \mathbf{a}, \mathbf{b} \rangle \tag{9.1}$$

As shown in the Bulletproof paper [Bün+18], the two constraints above can be proven using a subprotocol in which the prover shows that she knows two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$ such that

$$\mathsf{P} = (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) h^{\langle \mathbf{a}, \mathbf{b} \rangle} \tag{9.2}$$

for some generator h. We thus first describe the Bulletproof inner-product proof system for Relation (9.2), and then show how to obtain a proof system for Relation (9.1) from it.

Inner Product Subprotocol

Let $n' := \frac{n}{2}$ and fix generators $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, h \in \mathbb{G}$. Let $H : \mathbb{Z}_q^{2n+1} \to \mathbb{G}$ be a function that takes as input $\mathbf{a}, \mathbf{a}', \mathbf{b}, \mathbf{b}' \in \mathbb{Z}_q^{n'}$ and scalar $c \in \mathbb{Z}_q$, and outputs

$$H(\mathbf{a}, \mathbf{a}', \mathbf{b}, \mathbf{b}', c) := (\mathbf{g}_{\mathrm{lo}} \uparrow \mathbf{a})(\mathbf{g}_{\mathrm{hi}} \uparrow \mathbf{a}')(\mathbf{h}_{\mathrm{lo}} \uparrow \mathbf{b})(\mathbf{h}_{\mathrm{hi}} \uparrow \mathbf{b}') h^c \in \mathbb{G}$$

Note that P can be written as $\mathsf{P} = H(\mathbf{a}_{\mathrm{lo}}, \mathbf{a}_{\mathrm{hi}}, \mathbf{b}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}}, \langle \mathbf{a}, \mathbf{b} \rangle)$:

$$\mathsf{P} = (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) h^{\langle \mathbf{a}, \mathbf{b} \rangle}$$
$$= (\mathbf{g}_{\mathrm{lo}} \uparrow \mathbf{a}_{\mathrm{lo}})(\mathbf{g}_{\mathrm{hi}} \uparrow \mathbf{a}_{\mathrm{hi}})(\mathbf{h}_{\mathrm{lo}} \uparrow \mathbf{b}_{\mathrm{lo}})(\mathbf{h}_{\mathrm{hi}} \uparrow \mathbf{b}_{\mathrm{hi}}) h^{\langle \mathbf{a}, \mathbf{b} \rangle}$$
$$= H(\mathbf{a}_{\mathrm{lo}}, \mathbf{a}_{\mathrm{hi}}, \mathbf{b}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}}, \langle \mathbf{a}, \mathbf{b} \rangle)$$

Now, consider the following protocol for Relation (9.2), where P is given as input to both the prover and verifier:

- The prover computes L and R

$$\mathsf{L} := H(\mathbf{0}^{n'}, \mathbf{a}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}}, \mathbf{0}^{n'}, \langle \mathbf{a}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}} \rangle)$$
$$\mathsf{R} := H(\mathbf{a}_{\mathrm{hi}}, \mathbf{0}^{n'}, \mathbf{0}^{n'}, \mathbf{b}_{\mathrm{lo}}, \langle \mathbf{a}_{\mathrm{hi}}, \mathbf{b}_{\mathrm{lo}} \rangle)$$

and sends them to the verifier.

- The verifier selects a random $u \leftarrow \mathbb{Z}_q$ and sends it to the prover.

- The prover computes $\mathbf{a}', \mathbf{b}'$

$$\mathbf{a}' := u\mathbf{a}_{\mathrm{lo}} + u^{-1}\mathbf{a}_{\mathrm{hi}} \in \mathbb{Z}_q^{n'}$$
$$\mathbf{b}' := u^{-1}\mathbf{b}_{\mathrm{lo}} + u\mathbf{b}_{\mathrm{hi}} \in \mathbb{Z}_q^{n'}$$

  and sends them to the verifier.

- Given the proof $(\mathsf{L}, \mathsf{R}, \mathbf{a}', \mathbf{b}')$, the verifier computes $\mathsf{P}' := \mathsf{L}^{(u^2)} \cdot \mathsf{P} \cdot \mathsf{R}^{(u^{-2})}$ and accepts the proof if
$$\mathsf{P}' = H(u^{-1}\mathbf{a}', u\mathbf{a}', u\mathbf{b}', u^{-1}\mathbf{b}', \langle \mathbf{a}', \mathbf{b}' \rangle).$$

The correctness of the protocol can be shown by the following equalities:

$$\begin{aligned}
\mathsf{P}' &= \mathsf{L}^{(u^2)} \cdot \mathsf{P} \cdot \mathsf{R}^{(u^{-2})} \\
&= H(\mathbf{0}^{n'}, u^2\mathbf{a}_{\mathrm{lo}}, u^2\mathbf{b}_{\mathrm{hi}}, \mathbf{0}^{n'}, u^2\langle \mathbf{a}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}} \rangle) \cdot \\
&\quad H(\mathbf{a}_{\mathrm{lo}}, \mathbf{a}_{\mathrm{hi}}, \mathbf{b}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}}, \langle \mathbf{a}, \mathbf{b} \rangle) \cdot H(u^{-2}\mathbf{a}_{\mathrm{hi}}, \mathbf{0}^{n'}, \mathbf{0}^{n'}, u^{-2}\mathbf{b}_{\mathrm{lo}}, u^{-2}\langle \mathbf{a}_{\mathrm{hi}}, \mathbf{b}_{\mathrm{lo}} \rangle) \\
&= H(\mathbf{a}_{\mathrm{lo}} + u^{-2}\mathbf{a}_{\mathrm{hi}}, u^2\mathbf{a}_{\mathrm{lo}} + \mathbf{a}_{\mathrm{hi}}, u^2\mathbf{b}_{\mathrm{hi}} + \mathbf{b}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}} + u^{-2}\mathbf{b}_{\mathrm{lo}}, \langle \mathbf{a}', \mathbf{b}' \rangle) \\
&= H(u^{-1}\mathbf{a}', u\mathbf{a}', u\mathbf{b}', u^{-1}\mathbf{b}', \langle \mathbf{a}', \mathbf{b}' \rangle)
\end{aligned}$$

The proof in this protocol has size $n + 2$ which is about half of the trivial proof where the prover sends $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$ to the verifier. To shrink the proof further, we observe that P' can also be written as $\mathsf{P}' = \mathsf{VCom}(\mathbf{a}', \mathbf{b}'; \langle \mathbf{a}', \mathbf{b}' \rangle; \mathbf{g}', \mathbf{h}', h)$, where $\mathbf{g}' := \mathbf{g}_{\mathrm{lo}}^{u^{-1}} \circ \mathbf{g}_{\mathrm{hi}}^u$, and $\mathbf{h}' := \mathbf{h}_{\mathrm{lo}}^u \circ \mathbf{h}_{\mathrm{hi}}^{u^{-1}}$:

$$\begin{aligned}
\mathsf{P}' &= H(u^{-1}\mathbf{a}', u\mathbf{a}', u\mathbf{b}', u^{-1}\mathbf{b}', \langle \mathbf{a}', \mathbf{b}' \rangle) \\
&= \left(\mathbf{g}_{\mathrm{lo}} \uparrow u^{-1}\mathbf{a}' \right) \left(\mathbf{g}_{\mathrm{hi}} \uparrow u\mathbf{a}' \right) \left(\mathbf{h}_{\mathrm{lo}} \uparrow u\mathbf{b}' \right) \left(\mathbf{h}_{\mathrm{hi}} \uparrow u^{-1}\mathbf{b}' \right) h^{\langle \mathbf{a}', \mathbf{b}' \rangle} \\
&= \left(\mathbf{g}_{\mathrm{lo}}^{u^{-1}} \circ \mathbf{g}_{\mathrm{hi}}^u \uparrow \mathbf{a}' \right) \left(\mathbf{h}_{\mathrm{lo}}^u \circ \mathbf{h}_{\mathrm{hi}}^{u^{-1}} \uparrow \mathbf{b}' \right) h^{\langle \mathbf{a}', \mathbf{b}' \rangle} \\
&= \mathsf{VCom}(\mathbf{a}', \mathbf{b}'; \langle \mathbf{a}', \mathbf{b}' \rangle; \mathbf{g}', \mathbf{h}', h)
\end{aligned}$$

This can be seen as another inner product argument w.r.t. generators $(\mathbf{g}', \mathbf{h}', h)$ through which the prover can prove knowledge of $\mathbf{a}', \mathbf{b}'$ (instead of directly sending $\mathbf{a}'$ and $\mathbf{b}'$) to the verifier. After repeating this process $\log(n)$ times (i.e., $\log(n)$ round of interaction between the prover and verifier), we will have two vectors $\hat{a}, \hat{b}$ of dimension 1 at the end of the recursion and hence a proof of size $2\log(n) + 2$ in total, consisting of $(\mathsf{L}_0, \mathsf{R}_0), \ldots, (\mathsf{L}_{\log(n)-1}, \mathsf{R}_{\log(n)-1}), \hat{a}, \hat{b}$.

To verify the proof, the verifier first needs to compute the set of vectors $(\mathbf{g}, \mathbf{h}, \mathsf{P})$, $(\mathbf{g}', \mathbf{h}', \mathsf{P}')$, $\ldots$, $(\hat{g}, \hat{h}, \hat{\mathsf{P}})$ and then at the final round check if $\hat{\mathsf{P}} = \hat{g}^{\hat{a}}\hat{h}^{\hat{b}}h^{\hat{c}}$, where $\hat{c} = \hat{a} \cdot \hat{b}$. This requires $\frac{n}{2^{j-2}}$ exponentiations in round j and hence $4n = \sum_{j=1}^{\log(n)} \frac{n}{2^{j-2}}$ exponentiations in total. By delaying all the exponentiations until the last round, the $4n$ number of exponentiations can be reduced to a single multi-exponentiation of size $2n$. The idea is that by unrolling the recursion, we can express $\hat{g}$ and $\hat{h}$ as

$$\hat{g} = \mathbf{g} \uparrow \mathbf{s} = \prod_{i=0}^{n-1} g_i^{s_i} \in \mathbb{G}, \qquad \hat{h} = \mathbf{h} \uparrow \mathbf{s}^{-1} = \prod_{i=0}^{n-1} h_i^{\frac{1}{s_i}} \in \mathbb{G}$$

where $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ depends on the verifier's challenges $u_0, \ldots, u_{\log(n)-1}$ in the $\log(n)$ rounds of the protocol and is computed as follows:

$$s_i = \prod_{j=0}^{\log(n)-1} u_j^{b(i,j)} \qquad \text{where} \qquad b(i,j) = \begin{cases} 1 & (j+1)\text{-th bit of } i \text{ is } 1 \\ -1 & \text{otherwise} \end{cases}$$

Given this, the verification reduces to the following single multi-exponentiation of size $2n + 2\log(n) + 1$:

$$(\mathbf{g} \uparrow \mathbf{s})^{\widehat{a}} \cdot \left(\mathbf{h} \uparrow \mathbf{s}^{-1}\right)^{\widehat{b}} \cdot h^{\widehat{a} \cdot \widehat{b}} \stackrel{?}{=} \mathsf{P} \cdot \prod_{j=0}^{\log(n)-1} \mathsf{L}_j^{(u_j^2)} \cdot \mathsf{R}_j^{(u_j^{-2})}$$

The resulting recursive protocol is described below, where the group $\mathbb{G}$ and generators $\mathbf{g} = (g_0, \ldots, g_{n-1}), \mathbf{h} = (h_0, \ldots, h_{n-1})$ and h are implicit public values.

Protocol Prover for PK $\{(\mathbf{a}, \mathbf{b}) : \mathsf{P} = \mathsf{VCom}(\mathbf{a}, \mathbf{b}; \langle \mathbf{a}, \mathbf{b} \rangle; \mathbf{g}, \mathbf{h}, h)\}$

For $j := 0, 1 \ldots, \log(n) - 1$:

  Compute partial inner products.

  Compute $c_{L_j} := \langle \mathbf{a}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}} \rangle$ and $c_{R_j} := \langle \mathbf{a}_{\mathrm{hi}}, \mathbf{b}_{\mathrm{lo}} \rangle$.

  Commit to vectors.

  Commit $\mathsf{L}_j := \mathsf{VCom}(\mathbf{a}_{\mathrm{lo}}, \mathbf{b}_{\mathrm{hi}}; c_{L_j}; \mathbf{g}_{\mathrm{hi}}, \mathbf{h}_{\mathrm{lo}}, h)$.

  Commit $\mathsf{R}_j := \mathsf{VCom}(\mathbf{a}_{\mathrm{hi}}, \mathbf{b}_{\mathrm{lo}}; c_{R_j}; \mathbf{g}_{\mathrm{lo}}, \mathbf{h}_{\mathrm{hi}}, h)$. $\xrightarrow{\quad \mathsf{L}_j, \mathsf{R}_j \quad}$

                                                            $\xleftarrow{\quad u_j \quad}$

  Update the generators vectors.

  Compute $\mathbf{g} := \mathbf{g}_{\mathrm{lo}}^{u_j^{-1}} \circ \mathbf{g}_{\mathrm{hi}}^{u_j}$ and $\mathbf{h} := \mathbf{h}_{\mathrm{lo}}^{u_j} \circ \mathbf{h}_{\mathrm{hi}}^{u_j^{-1}}$.

  Update the statement.

  Compute $\mathsf{P} := \mathsf{L}_j^{(u_j^2)} \cdot \mathsf{P} \cdot \mathsf{R}_j^{(u_j^{-2})}$.

  Update the witness.

  Compute $\mathbf{a} := u_j \mathbf{a}_{\mathrm{lo}} + u_j^{-1} \mathbf{a}_{\mathrm{hi}}$ and $\mathbf{b} := u_j^{-1} \mathbf{b}_{\mathrm{lo}} + u_j \mathbf{b}_{\mathrm{hi}}$.

Last step $//\ j := \log(n) - 1; n := 1$

                                                     $\xrightarrow{\quad a, b \quad}$

Protocol Verifier for PK $\{(\mathbf{a}, \mathbf{b}) : \mathsf{P} = \mathsf{VCom}(\mathbf{a}, \mathbf{b}; \langle \mathbf{a}, \mathbf{b} \rangle; \mathbf{g}, \mathbf{h}, h)\}$

For $j := 0, 1 \ldots, \log(n) - 1$

  $\xrightarrow{\quad \mathsf{L}_j, \mathsf{R}_j \quad}$

  $\xleftarrow{\quad u_j \quad}$ Choose uniformly random $u_j \leftarrow \mathbb{Z}_q$.

Last step $//\ j := \log(n) - 1; n := 1$

$$\xrightarrow{\quad a,b \quad}$$ Compute the verification scalars $\mathbf{s} = (s_1, \ldots, s_n)$.

Compute $s_i = \prod_{j=0}^{\log(n)-1} u_j^{b(i,j)}$, where $b(i,j) = (-1)^{1+(j+1)\text{-th bit of } i}$.

Verification check

Verify $(\mathbf{g} \uparrow \mathbf{s})^a \cdot \left(\mathbf{h} \uparrow \mathbf{s}^{-1}\right)^b \cdot h^{a \cdot b} \overset{?}{=} \mathbf{P} \cdot \prod_{j=0}^{\log(n)-1} \mathsf{L}_j^{(u_j^2)} \cdot \mathsf{R}_j^{(u_j^{-2})}$.
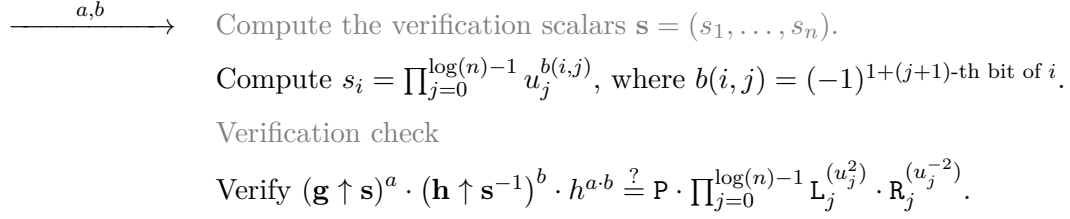
Inner Product Proof Protocol

We now present the protocol from the Bulletproof paper [Bün+18] that turns a proof system for $\mathsf{PK}\{(\mathbf{a}, \mathbf{b}) : \mathbf{P} = \mathsf{VCom}(\mathbf{a}, \mathbf{b}; \langle \mathbf{a}, \mathbf{b} \rangle; \mathbf{g}, \mathbf{h}, h)\}$ into one for $\mathsf{PK}\{(\mathbf{a}, \mathbf{b}) : \mathbf{P} = (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) \wedge c = \langle \mathbf{a}, \mathbf{b} \rangle\}$.

Protocol $\mathsf{PK}\{(\mathbf{a}, \mathbf{b}) : \mathbf{P} = (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) \wedge c = \langle \mathbf{a}, \mathbf{b} \rangle\}$

Implicitly known public values: $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, \tilde{g} \in \mathbb{G}$.

Prover$(\mathbf{P}, c, \mathbf{a}, \mathbf{b})$ **Verifier**$(\mathbf{P}, c)$

Choose uniformly random

$$\xleftarrow{\quad w \quad} \qquad w \leftarrow \mathbb{Z}_q.$$

Compute $\mathbf{P}' := \mathbf{P} \cdot \tilde{g}^{wc}$.

Execute subprotocol for

$$\mathsf{PK}\{(\mathbf{a}, \mathbf{b}) : \mathbf{P}' = \mathsf{VCom}(\mathbf{a}, \mathbf{b}; \langle \mathbf{a}, \mathbf{b} \rangle; \mathbf{g}, \mathbf{h}, \tilde{g}^w)\}.$$

$$\xleftrightarrow[a,b,\{\mathsf{L}_j, \mathsf{R}_j\}_j]{u_0, \ldots, u_{k-1}}$$
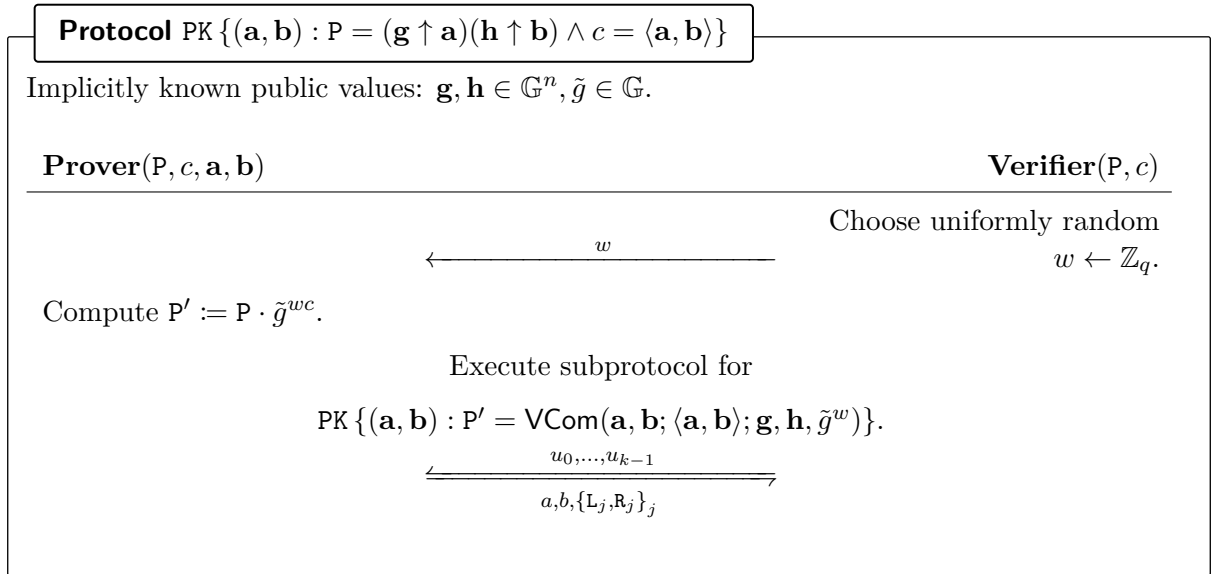
The following theorem proving completeness and knowledge soundness was shown in [Bün+18].

Theorem 35. *Assuming no nontrivial discrete logarithm relation between the generators in $\mathbf{g}, \mathbf{h}, \tilde{g}$ can be found, the protocol above has perfect completeness and statistical knowledge soundness.*

9.5.3 Range Proof

This section describes a proof that allows to convince the verifier that a committed value v is in range $[0, 2^n)$ for given n.

Range proof from inner product. The goal of a *range proof* is for a prover to convince a verifier that a secret value v is in a range $[0, 2^n)$ for a given public value n.

That is, the prover knows the randomness $\tilde{v}$ of a public commitment $\mathtt{V} = g^v h^{\tilde{v}}$ to value v and wants to convince the verifier that $v \in [0, 2^n)$ without revealing v.

In the following, the range proof is reduced to a single inner product proof.

Let $\mathbf{a}_L$ be the vector containing the bits of v so that $\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v$ and let $\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}$. The idea is for the prover to commit to $\mathbf{a}_L, \mathbf{a}_R$ and then prove to the verifier the knowledge of the

commitment opening (i.e., $\mathbf{a}_L, \mathbf{a}_R$) and $v, \tilde{v}$ so that $\mathsf{V} = g^v h^{\tilde{v}}$ and

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v \quad \text{// range proof check}$$
$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0} \quad \text{// for each coordinate one of } \mathbf{a}_L \text{ and } \mathbf{a}_R \text{ is zero} \qquad (9.3)$$
$$(\mathbf{a}_L - \mathbf{1}) - \mathbf{a}_R = \mathbf{0} \quad \text{// } \mathbf{a}_R \text{ is } \mathbf{a}_L - 1$$

This will prove that $\mathbf{a}_L = (a_1, \ldots, a_n)$ is in $\{0, 1\}^n$, and that it represents bits of v. Note that the last two constraints prove that $\mathbf{a}_L$ is a bit-vector, as in a finite field there are no zero divisors thus $a \cdot b = 0 \Rightarrow a = 0 \vee b = 0$.

Since $\mathbf{b} = \mathbf{0}$ if and only if $\langle \mathbf{b}, \mathbf{y}^n \rangle = 0$ for every y, we can convert all these $2n + 1$ constraints into three inner product constraints

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v$$
$$\langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle = 0$$
$$\langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle = 0$$

where $\mathbf{y}^n = (1, y, \ldots, y^{n-1})$, and y is chosen at random by the verifier.

Using a similar trick, these constraints are reduced into a single inner product constraint

$$z^2 v = z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle$$

for the verifier's choice of z.

Finally, the terms are combined into a single inner product where $\mathbf{a}_L$ only appears on the left, and $\mathbf{a}_R$ only on the right. Public terms are factored out into term $\delta(y, z)$

$$
\begin{aligned}
z^2 v &= z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^2 v &= z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L, \mathbf{y}^n \rangle - z \langle \mathbf{1}, \mathbf{y}^n \rangle - z \langle \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^2 v + z \langle \mathbf{1}, \mathbf{y}^n \rangle &= z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L, \mathbf{y}^n \rangle - z \langle \mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^2 v + z \langle \mathbf{1}, \mathbf{y}^n \rangle &= \langle \mathbf{a}_L, z^2 \mathbf{2}^n \rangle + \langle \mathbf{a}_L, z \mathbf{y}^n \rangle + \langle -z \mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^2 v + \langle z \mathbf{1}, \mathbf{y}^n \rangle &= \langle \mathbf{a}_L, z^2 \mathbf{2}^n + z \mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle -z \mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle
\end{aligned}
$$

Adding $\langle -z \mathbf{1}, z^2 \mathbf{2}^n + z \mathbf{y}^n \rangle$ to both sides

$$
\begin{aligned}
\Leftrightarrow \quad z^2 v + \langle z \mathbf{1}, \mathbf{y}^n \rangle + \langle -z \mathbf{1}, z^2 \mathbf{2}^n + z \mathbf{y}^n \rangle &= \langle \mathbf{a}_L - z \mathbf{1}, z^2 \mathbf{2}^n + z \mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^2 v + \langle z \mathbf{1}, \mathbf{y}^n - z \mathbf{y}^n - z^2 \mathbf{2}^n \rangle &= \langle \mathbf{a}_L - z \mathbf{1}, z^2 \mathbf{2}^n + z \mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^2 v + (z - z^2) \langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{2}^n \rangle &= \langle \mathbf{a}_L - z \mathbf{1}, z^2 \mathbf{2}^n + z \mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle
\end{aligned}
$$

Define $\delta(y, z) \coloneqq (z - z^2) \langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{2}^n \rangle$

$$\Leftrightarrow \quad z^2 v + \delta(y, z) = \langle \mathbf{a}_L - z \mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R + z \mathbf{1}) + z^2 \mathbf{2}^n \rangle$$

This is a single inner product constraint with $\mathbf{a}_L$ on the left, $\mathbf{a}_R$ on the right, and non-secret terms factored out. For the proof to be zero-knowledge, the prover needs to blind the left and right vectors by choosing random blinding factors $\mathbf{s}_L, \mathbf{s}_R \in \mathbb{Z}_q^n$. Using polynomial evaluation at challenge point x we get

$$\mathbf{l}(x) = \mathbf{l}_0 + x \mathbf{l}_1 = (\mathbf{a}_L + x \mathbf{s}_L) - z \mathbf{1}$$
$$\mathbf{r}(x) = \mathbf{r}_0 + x \mathbf{r}_1 = \mathbf{y}^n \circ (\mathbf{a}_R + x \mathbf{s}_R + z \mathbf{1}) + z^2 \mathbf{2}^n$$
$$t(x) = t_0 + x t_1 + x^2 t_2 = \langle \mathbf{l}(x), \mathbf{r}(x) \rangle$$

where
$$t_0 = \langle \mathbf{l}_0, \mathbf{r}_0 \rangle, \quad t_2 = \langle \mathbf{l}_1, \mathbf{r}_1 \rangle, \quad t_1 = \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$$

Finally, observing that
$$t_0 = \langle \mathbf{l}_0, \mathbf{r}_0 \rangle = \left\langle \mathbf{a}_L - z\mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}) + z^2 \mathbf{2}^n \right\rangle = z^2 v + \delta(y, z),$$

the range proof boils down to show that at point x it holds $\langle \mathbf{l}(x), \mathbf{r}(x) \rangle = t(x)$ and that $t_0 = z^2 v + \delta(y, z)$.

Protocol. We can now describe the range proof protocol with the following implicit public values: group $\mathbb{G}$, and generators $g, h, \mathbf{g} = (g_0, \ldots, g_{n-1}), \mathbf{h} = (h_0, \ldots, h_{n-1})$.

Protocol Prover for PK $\left\{ (v, \tilde{v}) : \mathtt{V} = g^v h^{\tilde{v}} \ \wedge \ v \in [0, 2^n) \right\}$

Let $\mathbf{a}_L$ be the vector containing the bits of v and $\mathbf{a}_R := \mathbf{a}_L - \mathbf{1}$

Select blinding factors.
Choose uniform random vectors $\mathbf{s}_L, \mathbf{s}_R \leftarrow \mathbb{Z}_q^n$.

Commit to vectors $\mathbf{a}_L, \mathbf{a}_R$ and blinding vectors.
Commit $\mathtt{A} := \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a})$ for uniform random $\tilde{a} \leftarrow \mathbb{Z}_q$.
Commit $\mathtt{S} := \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})$ for uniform random $\tilde{s} \leftarrow \mathbb{Z}_q$.

$\xrightarrow{\quad \mathtt{A}, \mathtt{S} \quad}$

$\xleftarrow{\quad y, z \quad}$

Compute LHS polynomial coefficients.
Compute $\mathbf{l}_0 := \mathbf{a}_L - z\mathbf{1}$ and $\mathbf{l}_1 := \mathbf{s}_L$.

Compute RHS polynomial coefficients.
Compute $\mathbf{r}_0 := \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}) + z^2 \mathbf{2}^n$ and $\mathbf{r}_1 := \mathbf{y}^n \circ \mathbf{s}_R$.

Compute inner product polynomial coefficients.
Compute $t_0 := \langle \mathbf{l}_0, \mathbf{r}_0 \rangle$, $t_2 := \langle \mathbf{l}_1, \mathbf{r}_1 \rangle$, and
$t_1 := \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$.

Commit to inner product polynomial coefficients t_1 and t_2.
Commit $\mathtt{T}_1 := \mathsf{Com}(t_1; \tilde{t}_1)$ for uniform random $\tilde{t}_1 \leftarrow \mathbb{Z}_q$.
Commit $\mathtt{T}_2 := \mathsf{Com}(t_2; \tilde{t}_2)$ for uniform random $\tilde{t}_2 \leftarrow \mathbb{Z}_q$.

$\xrightarrow{\quad \mathtt{T}_1, \mathtt{T}_2 \quad}$

$\xleftarrow{\quad x \quad}$

Evaluate polynomials at x.
Compute $\mathbf{l}_x := \mathbf{l}_0 + x\mathbf{l}_1$, and $\mathbf{r}_x := \mathbf{r}_0 + x\mathbf{r}_1$, and $t_x := t_0 + xt_1 + x^2 t_2$.

Compute opening commitment opening information.
Compute $\tilde{t}_x := z^2 \tilde{v} + x\tilde{t}_1 + x^2 \tilde{t}_2$, and $\tilde{e} := \tilde{a} + x\tilde{s}$.

$\xrightarrow{\quad t_x, \tilde{t}_x, \tilde{e} \quad}$

Compute new generators.
Compute $\hat{g} := g^w$ and set $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$.

Compute inner product commitment.
Compute $\mathsf{P}' := \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})$.

Execute inner product proof for
$\mathsf{PK}\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})\}$

$\xleftarrow{\quad w \quad}$

$\xleftarrow{u_0,\ldots,u_{k-1}}$
$\xrightarrow{a,b,\{\mathsf{L}_j,\mathsf{R}_j\}_j}$

Protocol Verifier for $\mathsf{PK}\{(v, \tilde{v}) : \mathsf{V} = g^v h^{\tilde{v}} \ \wedge \ v \in [0, 2^n)\}$

$\xrightarrow{\quad \mathsf{A},\mathsf{S} \quad}$ Choose uniformly random $y, z \leftarrow \mathbb{Z}_q$.

$\xleftarrow{\quad y,z \quad}$

$\xrightarrow{\quad \mathsf{T}_1,\mathsf{T}_2 \quad}$ Choose uniformly random $x \leftarrow \mathbb{Z}_q$.

$\xleftarrow{\quad x \quad}$

$\xrightarrow{\quad t_x,\tilde{t}_x,\tilde{e} \quad}$ Choose uniformly random $w \leftarrow \mathbb{Z}_q$.

$\xleftarrow{\quad w \quad}$

Verify consistency of t_0.
Compute $\delta(y, z) := (z - z^2)\langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{2}^n \rangle$.
Verify $1_{\mathbb{G}} \stackrel{?}{=} \mathsf{V}^{z^2} g^{\delta(y,z)} \mathsf{T}_1^x \mathsf{T}_2^{x^2} g^{-t_x} h^{-\tilde{t}_x}$.

Compute new generators.
Compute $\hat{g} := g^w$ and $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$.

Compute commitment for inner product.
Compute $\mathsf{P}' := (\mathbf{g} \uparrow -z\mathbf{1})(\mathbf{h} \uparrow z\mathbf{1} + z^2 \mathbf{y}^{-n} \circ \mathbf{2}^n) \hat{g}^{t_x} h^{-\tilde{e}} \mathsf{A}\mathsf{S}^x$.

Verify inner product proof for
$\mathsf{PK}\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})\}$.

$\xleftarrow{u_0,\ldots,u_{k-1}}$
$\xrightarrow{a,b,\{\mathsf{L}_j,\mathsf{R}_j\}_j}$

Aggregated range proof. In most applications, the prover needs to perform multiple (say, m) range proofs simultaneously. In this case, the naive approach would be to create m independent range proofs by simply repeating the range proof protocol described in Section 9.5.3. This section describes a more efficient proof system where the proof size grows only by an additive term $2\log(m)$ as opposed to a multiplicative factor of m when creating m individual range proofs.

The goal is for a prover to convince a verifier that it knows the vector of randomnesses $\tilde{\mathbf{v}} \in \mathbb{Z}_q^m$ and the vector of values $\mathbf{v} \in \mathbb{Z}_q^m$ of a public vector of commitments $\mathbf{V} \in \mathbb{G}^m$ such that $\mathsf{V}_j = g^{v_j} h^{\tilde{v}_j}$,

and wants to convince the verifier that each $v_j \in [0, 2^n)$ without revealing v_j:

$$\text{PK}\left\{(\mathbf{v}, \tilde{\mathbf{v}}) \in (\mathbb{Z}_q^m)^2 : V_j = g^{v_j} h^{\tilde{v}_j} \ \wedge \ v_j \in [0, 2^n) \quad \forall j \in [1, m]\right\}$$

The prover is very similar to the prover in Section 9.5.3 with the following modifications:

- $\mathbf{a}_L \in \mathbb{Z}_q^{nm}$ is now the concatenation of all the bits of all values $v_j \in \mathbf{v}$, i.e., $\langle \mathbf{2}^n, \mathbf{a}_L[(j-1)n : jn-1]\rangle = v_j$ for all j in $[1, m]$. This implies that $\mathbf{l}(x)$ and $\mathbf{r}(x)$ should be adjusted accordingly so that

$$\mathbf{l}(x) = \mathbf{l}_0 + x\mathbf{l}_1 = (\mathbf{a}_L + x\mathbf{s}_L) - z\mathbf{1}^{nm} \in \mathbb{Z}_q^{nm}$$

$$\mathbf{r}(x) = \mathbf{r}_0 + x\mathbf{r}_1 = \mathbf{y}^{nm} \circ (\mathbf{a}_R + x\mathbf{s}_R + z\mathbf{1}^{nm}) + \sum_{j=0}^{m-1} z^{2+j}\left(\mathbf{0}^{(j-1)n} \parallel \mathbf{2}^n \parallel \mathbf{0}^{(m-j)n}\right) \in \mathbb{Z}_q^{nm}$$

- In the computation of $\tilde{t}_x$, the randomness of each commitment V_j should be adjusted so that

$$\tilde{t}_x := \sum_{j=0}^{m-1} z^{2+j} \tilde{v}_j + x\tilde{t}_1 + x^2 \tilde{t}_2.$$

- The public quantity $\delta(y, z)$ should be updated to

$$\delta(y, z) := (z - z^2)\langle \mathbf{1}, \mathbf{y}^{nm}\rangle - \sum_{j=0}^{m-1} z^{3+j}\langle \mathbf{1}, \mathbf{2}^{nm}\rangle.$$

In the verifier side, we need to apply the following modifications.

- The verification check for the consistency of t_0 should be updated to include all the V_j commitments, i.e.,

$$1 \stackrel{?}{=} \left(\mathbf{V} \uparrow \mathbf{z}^{m+2}\right) g^{\delta(y,z)} T_1^x T_2^{x^2} g^{-t_x} h^{-\tilde{t}_x}$$

where $\mathbf{z}^{m+2} = (z^2, z^3, \ldots, z^{m+1})$.

- The computation of P' should be updated such that it is a commitment to the new $\mathbf{r}$, i.e.,

$$P' := \hat{g}^{t_x} h^{-\tilde{e}} AS^x (\mathbf{h} \uparrow z\mathbf{1} + z^2 \mathbf{y}^{-nm} \circ \mathbf{2}^{nm})(\mathbf{g} \uparrow -z\mathbf{1})$$

The security of this is proven in the Bulletproof paper [Bün+18].

> 💡 We could probably have a generic aggregation that can combine range proofs and set-(non-)membership proofs.

Protocol. We can now describe the *aggregated range proof* protocol. The protocol is similar to the protocol in Section 9.5.3 with the changes highlighted in blue. Different from Section 9.5.3, the verification here is described in a more explicit way, so that it combines the verification of the inner product proof with the rest of the verification.

Protocol Prover for PK $\{(\mathbf{v}, \tilde{\mathbf{v}}) : \mathtt{V}_j = g^{v_j} h^{\tilde{v}_j} \ \wedge \ v_j \in [0, 2^n) \quad \forall j \in [1, m]\}$

Let $\mathbf{a}_L$ be the vector containing the bits of all values $v_j \in \mathbf{v}$ and $\mathbf{a}_R := \mathbf{a}_L - \mathbf{1}$.

Select blinding factors.
Choose uniform random vectors $\mathbf{s}_L, \mathbf{s}_R \leftarrow \mathbb{Z}_q^{nm}$.

Commit to vectors $\mathbf{a}_L, \mathbf{a}_R$ and blinding vectors.
Commit $\mathtt{A} := \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a})$ for uniform random $\tilde{a} \leftarrow \mathbb{Z}_q$.
Commit $\mathtt{S} := \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})$ for uniform random $\tilde{s} \leftarrow \mathbb{Z}_q$.

$\xrightarrow{\quad \mathtt{A},\mathtt{S} \quad}$

$\xleftarrow{\quad y,z \quad}$

Compute LHS polynomial coefficients.
Compute $\mathbf{l}_0 := \mathbf{a}_L - z\mathbf{1}^{nm}$ and $\mathbf{l}_1 := \mathbf{s}_L$.

Compute RHS polynomial coefficients.
Compute $\mathbf{r}_0 := \mathbf{y}^{nm} \circ (\mathbf{a}_R + z\mathbf{1}^{nm}) + \sum_{j=0}^{m-1} z^{2+j} \left(\mathbf{0}^{(j-1)n} \parallel \mathbf{2}^n \parallel \mathbf{0}^{(m-j)n} \right)$.
Compute $\mathbf{r}_1 := \mathbf{y}^{nm} \circ \mathbf{s}_R$.

Compute inner product polynomial coefficients.
Compute $t_0 := \langle \mathbf{l}_0, \mathbf{r}_0 \rangle$, $t_2 := \langle \mathbf{l}_1, \mathbf{r}_1 \rangle$, and
$t_1 := \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$.

Commit to inner product polynomial coefficients t_1 and t_2.
Commit $\mathtt{T}_1 := \mathsf{Com}(t_1; \tilde{t}_1)$ for uniform random $\tilde{t}_1 \leftarrow \mathbb{Z}_q$.
Commit $\mathtt{T}_2 := \mathsf{Com}(t_2; \tilde{t}_2)$ for uniform random $\tilde{t}_2 \leftarrow \mathbb{Z}_q$.

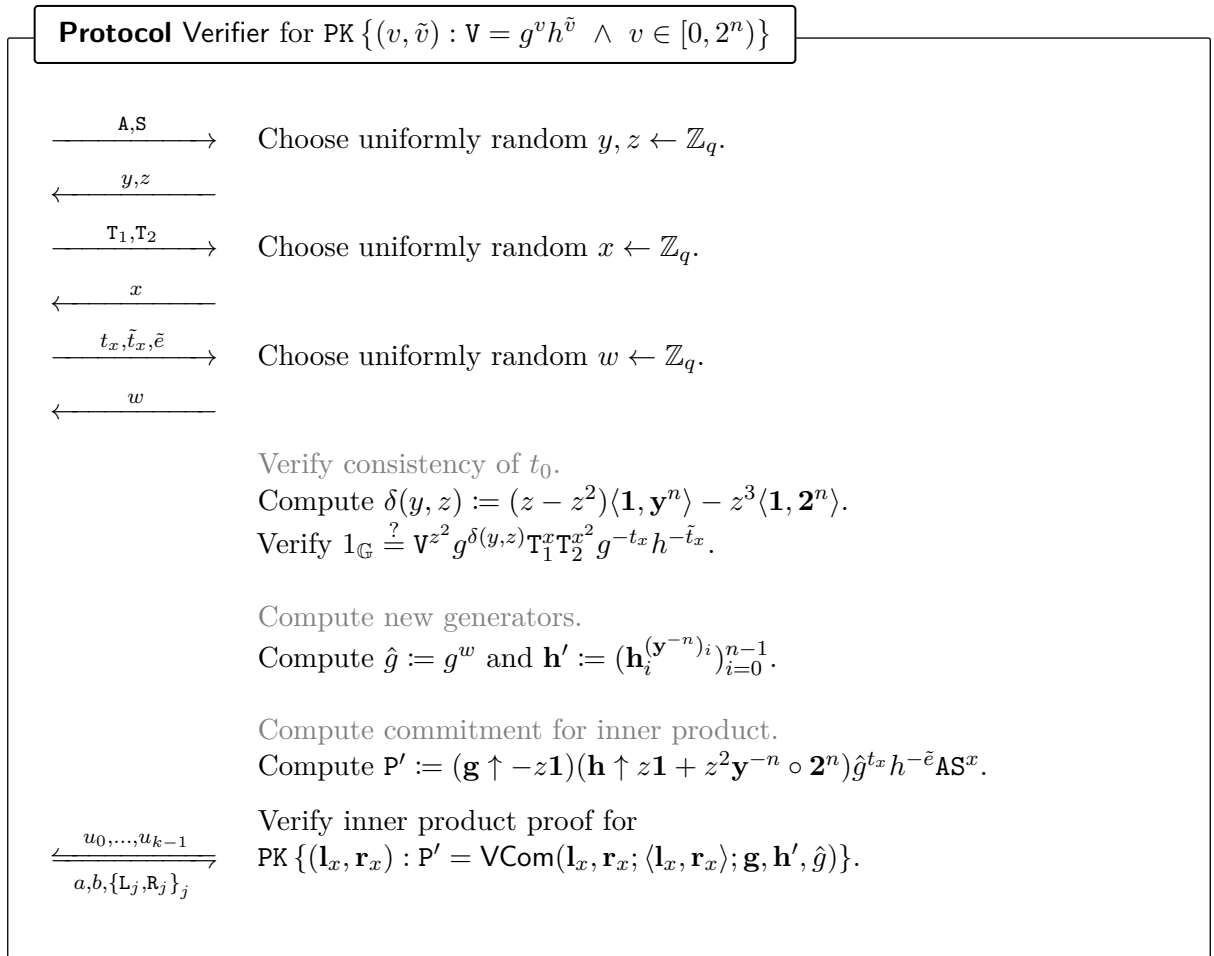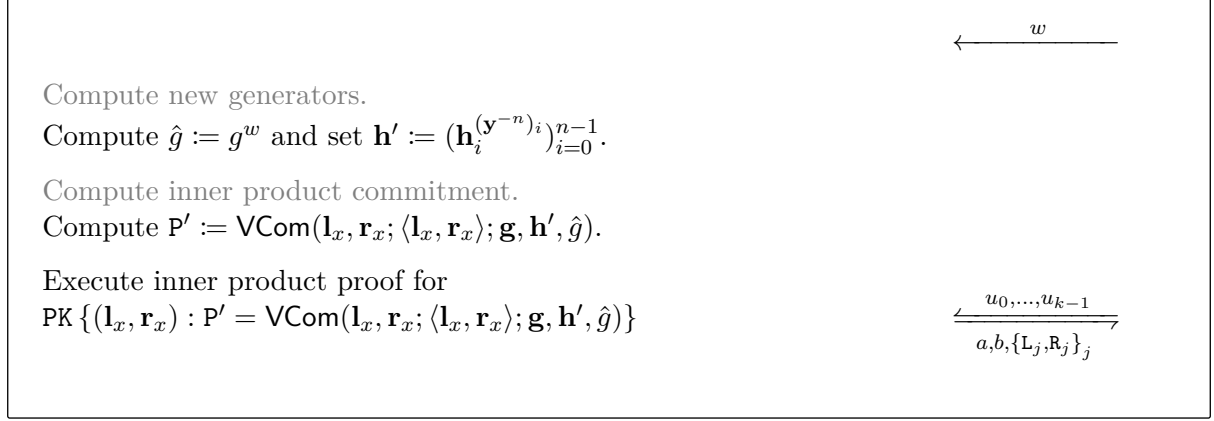$\xrightarrow{\quad \mathtt{T}_1,\mathtt{T}_2 \quad}$

$\xleftarrow{\quad x \quad}$

Evaluate polynomials at x.
Compute $\mathbf{l}_x := \mathbf{l}_0 + x\mathbf{l}_1$, and $\mathbf{r}_x := \mathbf{r}_0 + x\mathbf{r}_1$, and $t_x := t_0 + xt_1 + x^2 t_2$.

Compute opening commitment opening information.
Compute $\tilde{t}_x := \sum_{j=0}^{m-1} z^{2+j} \tilde{v}_j + x\tilde{t}_1 + x^2 \tilde{t}_2$, and $\tilde{e} := \tilde{a} + x\tilde{s}$.

$\xrightarrow{\quad t_x, \tilde{t}_x, \tilde{e} \quad}$

$\xleftarrow{\quad w \quad}$

Compute new generators.
Compute $\hat{g} := g^w$ and set $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-nm})_i})_{i=0}^{nm-1}$.

Compute inner product commitment.
Compute $\mathtt{P}' := \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})$.

Execute inner product proof for
PK $\{(\mathbf{l}_x, \mathbf{r}_x) : \mathtt{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})\}$

$\xleftarrow{\quad u_0, \ldots, u_{k-1} \quad}$

$\xrightarrow{\quad a, b, \{\mathtt{L}_j, \mathtt{R}_j\}_j \quad}$

Protocol Verifier for PK $\left\{ (\mathbf{v}, \tilde{\mathbf{v}}) : \mathtt{V}_j = g^{v_j} h^{\tilde{v}_j} \ \wedge \ v_j \in [0, 2^n) \quad \forall j \in [1, m] \right\}$

$\xrightarrow{\quad \mathtt{A,S} \quad}$ Choose uniformly random $y, z \leftarrow \mathbb{Z}_q$.

$\xleftarrow{\quad y, z \quad}$

$\xrightarrow{\quad \mathtt{T_1, T_2} \quad}$ Choose uniformly random $x \leftarrow \mathbb{Z}_q$.

$\xleftarrow{\quad x \quad}$

$\xrightarrow{\quad t_x, \tilde{t}_x, \tilde{e} \quad}$ Choose uniformly random $w \leftarrow \mathbb{Z}_q$.

$\xleftarrow{\quad w \quad}$

Verify consistency of t_0.
Compute $\delta(y,z) := (z - z^2)\langle \mathbf{1}, \mathbf{y}^{nm} \rangle - \sum_{j=0}^{m-1} z^{3+j} \langle \mathbf{1}, \mathbf{2}^{nm} \rangle$.
Verify $1_{\mathbb{G}} \stackrel{?}{=} (\mathbf{V} \uparrow \mathbf{z}^{m+2}) \, g^{\delta(y,z)} \mathtt{T}_1^x \mathtt{T}_2^{x^2} g^{-t_x} h^{-\tilde{t}_x}$.

Compute new generators.
Compute $\hat{g} := g^w$ and $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-nm})_i})_{i=0}^{nm-1}$.

Compute commitment for inner product.
Compute $\mathtt{P}' := \hat{g}^{t_x} h^{-\tilde{e}} \mathtt{A} \mathtt{S}^x (\mathbf{h} \uparrow z\mathbf{1} + z^2 \mathbf{y}^{-nm} \circ \mathbf{2}^{nm})(\mathbf{g} \uparrow -z\mathbf{1})$.

Verify inner product proof for
$\xleftarrow{\quad u_0, \dots, u_{k-1} \quad}$
$\xrightarrow{\quad a, b, \{\mathtt{L}_j, \mathtt{R}_j\}_j \quad}$ PK $\left\{ (\mathbf{l}_x, \mathbf{r}_x) : \mathtt{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g}) \right\}$.

Range proofs for arbitrary interval

The range proof protocol described above can be used to show that a committed value v is in range $[0, u)$, where $u = 2^n$ for some given public value n. It is not hard to extend the protocol to the case where the prover wants to show that $v \leq u$ for some arbitrary public value u. Let us assume that $v, u \in [0, 2^n)$ and that the underlying commitment scheme can commit to messages of size at least 2^{n+1}. To prove $v \leq u$, the prover commits to $u - v$ and shows that $u - v \in [0, 2^n)$. This convinces the verifier that $u - v$ is non-negative and hence $v \leq u$. Note that a commitment to $u - v$ can be computed by doing homomorphic operations on the given commitment $\mathtt{V}$ to v and a publicly computable commitment to u; e.g., $\mathtt{U} = g^u$.

9.5.4 Set Membership Proof

This section describes a set membership proof that allows the prover to convince the verifier that a committed value v is in a given set S.

Set membership proof from inner product. The goal of a *set membership proof* is for a prover to convince a verifier that a value v lies in a given set S without revealing any additional information about v. That is, the prover knows the randomness $\tilde{v}$ of a public commitment $\mathtt{V} = g^v h^{\tilde{v}}$ to value v and wants to convince the verifier that $v \in S$ without revealing v.

In the following, similar to the range proof in Section 9.5.3, the set membership proof is reduced to a single inner product proof.

Let $S = \{s_0, \ldots, s_{n-1}\}$ and denote by $\mathbf{s} := (s_0, \ldots, s_{n-1})$ the vector[4] of set elements. If $v \in S$, then there is a $j \in \mathbb{Z}_q$ such that $v = s_j$. This can be represented by the bit vector $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in \{0,1\}^n$ which is all zeroes except for $a_j = 1$. Then we have $\langle \mathbf{a}, \mathbf{s} \rangle = v$.

Similar to the range proof in the above, we can represent the constraint $v \in S$ using vectors $\mathbf{a}_L = \mathbf{a}$ and $\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}$ as follows

$$
\begin{aligned}
\langle \mathbf{a}_L, \mathbf{s} \rangle &= v \quad && \text{// set membership check} \\
\langle \mathbf{a}_L, \mathbf{1} \rangle &= 1 \quad && \text{// } \mathbf{a}_L \text{ contains a single 1} \\
\mathbf{a}_L \circ \mathbf{a}_R &= \mathbf{0} \quad && \text{// for each coordinate one of } \mathbf{a}_L \text{ and } \mathbf{a}_R \text{ is zero} \\
(\mathbf{a}_L - \mathbf{1}) - \mathbf{a}_R &= \mathbf{0} \quad && \text{// } \mathbf{a}_R \text{ is } \mathbf{a}_L - \mathbf{1}
\end{aligned}
\tag{9.4}
$$

This will prove that $\mathbf{a}_L = (a_1, \ldots, a_n)$ is a binary vector in $\{0,1\}^n$ (last two constraints) with only one nonzero entry (second constraint) that corresponds to the position of v in S (first constraint).

Next, we use this observation that proving a committed vector $\mathbf{b}$ satisfies $\mathbf{b} = \mathbf{0}$ can be reduced to proving that $\langle \mathbf{b}, \mathbf{y}^n \rangle = 0$, where $\mathbf{y}^n = (1, y, \ldots, y^{n-1})$ and $y \in \mathbb{Z}_q$ is chosen at random by the verifier. Note that if $\mathbf{b} \neq \mathbf{0}$, the equality holds with probability at most n/q which is supposed to be negligible. Thus, $\langle \mathbf{b}, \mathbf{y}^n \rangle = 0$ convinces the verifier that $\mathbf{b} = \mathbf{0}$.

Using this observation, we convert all the above constraints into inner product constraints

$$
\begin{aligned}
\langle \mathbf{a}_L, \mathbf{s} \rangle &= v, \\
\langle \mathbf{a}_L, \mathbf{1} \rangle &= 1, \\
\langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle &= 0, \\
\langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle &= 0.
\end{aligned}
$$

Furthermore, to improve efficiency, the four inner product constraints are combined into a single constraint using polynomial evaluation at challenge point z resulting in:

$$
\begin{aligned}
0 &= \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle + z \cdot \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle + z^2 \cdot (\langle \mathbf{a}_L, \mathbf{s} \rangle - v) + z^3 \cdot (\langle \mathbf{a}_L, \mathbf{1} \rangle - 1) \\
\Leftrightarrow \quad z^3 + z^2 v &= z^3 \langle \mathbf{a}_L, \mathbf{1} \rangle + z^2 \langle \mathbf{a}_L, \mathbf{s} \rangle + z \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle
\end{aligned}
$$

Finally, the terms are combined into a single inner product where $\mathbf{a}_L$ only appears on the left, and $\mathbf{a}_R$ only on the right. Public terms are factored out into the term $\delta(y, z)$:

$$
\begin{aligned}
z^3 + z^2 v &= z^3 \langle \mathbf{a}_L, \mathbf{1} \rangle + z^2 \langle \mathbf{a}_L, \mathbf{s} \rangle + z \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow z^3 + z^2 v &= z^3 \langle \mathbf{a}_L, \mathbf{1} \rangle + z^2 \langle \mathbf{a}_L, \mathbf{s} \rangle + z \langle \mathbf{a}_L, \mathbf{y}^n \rangle - z \langle \mathbf{1}, \mathbf{y}^n \rangle - z \langle \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow z^3 + z^2 v + z \langle \mathbf{1}, \mathbf{y}^n \rangle &= z^3 \langle \mathbf{a}_L, \mathbf{1} \rangle + z^2 \langle \mathbf{a}_L, \mathbf{s} \rangle + z \langle \mathbf{a}_L, \mathbf{y}^n \rangle - z \langle \mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow z^3 + z^2 v + z \langle \mathbf{1}, \mathbf{y}^n \rangle &= \langle \mathbf{a}_L, z^3 \mathbf{1} \rangle + \langle \mathbf{a}_L, z^2 \mathbf{s} \rangle + \langle \mathbf{a}_L, z \mathbf{y}^n \rangle + \langle -z\mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow z^3 + z^2 v + \langle z\mathbf{1}, \mathbf{y}^n \rangle &= \langle \mathbf{a}_L, z^3 \mathbf{1} + z^2 \mathbf{s} + z \mathbf{y}^n \rangle + \langle \mathbf{a}_L - z\mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle
\end{aligned}
$$

Adding $\langle -z\mathbf{1}, z^3 \mathbf{1} + z^2 \mathbf{s} + z \mathbf{y}^n \rangle$ to both sides yields:

$$
\begin{aligned}
\Leftrightarrow \quad z^3 + z^2 v + \langle z\mathbf{1}, \mathbf{y}^n \rangle + \langle -z\mathbf{1}, z^3 \mathbf{1} + z^2 \mathbf{s} + z\mathbf{y}^n \rangle &= \langle \mathbf{a}_L - z\mathbf{1}, z^3 \mathbf{1} + z^2 \mathbf{s} + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^3 + z^2 v + \langle z\mathbf{1}, -z^3 \mathbf{1} + \mathbf{y}^n - z\mathbf{y}^n - z^2 \mathbf{s} \rangle &= \langle \mathbf{a}_L - z\mathbf{1}, z^3 \mathbf{1} + z^2 \mathbf{s} + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle \\
\Leftrightarrow \quad z^3 + z^2 v - z^4 n + (z - z^2) \langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{s} \rangle &= \langle \mathbf{a}_L - z\mathbf{1}, z^3 \mathbf{1} + z^2 \mathbf{s} + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle
\end{aligned}
$$

---

[4] The order of elements in $\mathbf{s}$ does not matter.

For $\delta(y,z) := z^3(1 - zn - \langle \mathbf{1}, \mathbf{s} \rangle) + (z - z^2)\langle \mathbf{1}, \mathbf{y}^n \rangle$, this is equivalent to:

$$\Leftrightarrow \quad z^2 v + \delta(y,z) = \left\langle \mathbf{a}_L - z\mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}) + z^3\mathbf{1} + z^2\mathbf{s} \right\rangle \tag{9.5}$$

We thus reduced the problem of proving the constraints (9.4) to proving the following inner-product constraint: given a commitment $\mathbf{V}$, the prover knows an opening $v, \tilde{v}$ of $\mathbf{V}$ (i.e., $\mathbf{V} = g^v h^{\tilde{v}}$) so that

$$z^2 v + \delta(y,z) = \left\langle \mathbf{a}_L - z\mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}) + z^3\mathbf{1} + z^2\mathbf{s} \right\rangle \tag{9.6}$$

The naive approach to prove the above constraint could be for the prover to send the two vectors $\mathbf{a}_L, \mathbf{a}_R$ and then the verifier checks 9.6 using $\mathbf{a}_L, \mathbf{a}_R$ along with the commitment $\mathbf{V}$ and publicly computable value $\delta(y,z)$. This direct approach would reveal some information about the prover's witness.

To pass zero information on, the prover—as for the range proof—needs to blind the vectors $\mathbf{a}_L, \mathbf{a}_R$ of the inner product by choosing random blinding factors $\mathbf{s}_L, \mathbf{s}_R \in \mathbb{Z}_q^n$. Using polynomial evaluation at challenge point x, we get

$$\mathbf{l}(x) = \mathbf{l}_0 + x\mathbf{l}_1 = (\mathbf{a}_L + x\mathbf{s}_L) - z\mathbf{1}$$
$$\mathbf{r}(x) = \mathbf{r}_0 + x\mathbf{r}_1 = \mathbf{y}^n \circ (\mathbf{a}_R + x\mathbf{s}_R + z\mathbf{1}) + z^3\mathbf{1} + z^2\mathbf{s}$$
$$t(x) = t_0 + xt_1 + x^2 t_2 = \langle \mathbf{l}(x), \mathbf{r}(x) \rangle$$

where

$$t_0 = \langle \mathbf{l}_0, \mathbf{r}_0 \rangle, \quad t_2 = \langle \mathbf{l}_1, \mathbf{r}_1 \rangle, \quad t_1 = \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$$

Finally, the set membership proof boils down to show that at point x we have $\langle \mathbf{l}(x), \mathbf{r}(x) \rangle = t(x)$ and $t_0 = z^2 v + \delta(y,z)$.

Protocol. We can now describe the set membership proof protocol with the following implicit public values: group $\mathbb{G}$, generators $g, h, \mathbf{g} = (g_0, \ldots, g_{n-1}), \mathbf{h} = (h_0, \ldots, h_{n-1})$, and set $S = \{s_0, \ldots, s_{n-1}\}$ (and set size n).

Protocol Prover for PK $\left\{ (v, \tilde{v}) : \mathbf{V} = g^v h^{\tilde{v}} \ \wedge \ v \in S \right\}$

Let $v = s_j \in S$, set $\mathbf{a}_L := \mathbf{e}_j$ and $\mathbf{a}_R := \mathbf{a}_L - \mathbf{1}$.

Select blinding factors.
Choose uniform random vectors $\mathbf{s}_L, \mathbf{s}_R \leftarrow \mathbb{Z}_q^n$.

Commit to vectors $\mathbf{a}_L, \mathbf{a}_R$ and blinding vectors.
Commit $\mathbf{A} := \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a})$ for uniform random $\tilde{a} \leftarrow \mathbb{Z}_q$.
Commit $\mathbf{S} := \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})$ for uniform random $\tilde{s} \leftarrow \mathbb{Z}_q$.

$\xrightarrow{\quad \mathbf{A}, \mathbf{S} \quad}$

$$\xleftarrow{\quad y,z \quad}$$

Compute LHS polynomial coefficients.
Compute $\mathbf{l}_0 := \mathbf{a}_L - z\mathbf{1}$ and $\mathbf{l}_1 := \mathbf{s}_L$.

Compute RHS polynomial coefficients.
Compute $\mathbf{r}_0 := \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}) + z^3\mathbf{1} + z^2\mathbf{s}$ and $\mathbf{r}_1 := \mathbf{y}^n \circ \mathbf{s}_R$.

Compute inner product polynomial coefficients.
Compute $t_0 := \langle \mathbf{l}_0, \mathbf{r}_0 \rangle$, $t_2 := \langle \mathbf{l}_1, \mathbf{r}_1 \rangle$, and
$t_1 := \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$.

Commit to inner product polynomial coefficients t_1 and t_2.
Commit $\mathsf{T}_1 := \mathsf{Com}(t_1; \tilde{t}_1)$ for uniform random $\tilde{t}_1 \leftarrow \mathbb{Z}_q$.
Commit $\mathsf{T}_2 := \mathsf{Com}(t_2; \tilde{t}_2)$ for uniform random $\tilde{t}_2 \leftarrow \mathbb{Z}_q$.

$$\xrightarrow{\quad \mathsf{T}_1,\mathsf{T}_2 \quad}$$

$$\xleftarrow{\quad x \quad}$$

Evaluate polynomials at x.
Compute $\mathbf{l}_x := \mathbf{l}_0 + x\mathbf{l}_1$, and $\mathbf{r}_x := \mathbf{r}_0 + x\mathbf{r}_1$, and $t_x := t_0 + xt_1 + x^2t_2$.

Compute commitment opening information.
Compute $\tilde{t}_x := z^2\tilde{v} + x\tilde{t}_1 + x^2\tilde{t}_2$. Compute $\tilde{e} := \tilde{a} + x\tilde{s}$.

$$\xrightarrow{\quad t_x,\tilde{t}_x,\tilde{e} \quad}$$

$$\xleftarrow{\quad w \quad}$$

Compute new generators.
Compute $\hat{g} := g^w$ and set $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$

Compute inner product commitment.
Compute $\mathsf{P}' := \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})$.

Execute inner product proof for
$\mathsf{PK}\left\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})\right\}$.

$$\xleftarrow{\quad u_0,\ldots,u_{k-1} \quad}$$
$$\xrightarrow{\quad a,b,\{\mathsf{L}_j,\mathsf{R}_j\}_j \quad}$$

Protocol Verifier for $\mathsf{PK}\left\{(v, \tilde{v}) : \mathsf{V} = g^v h^{\tilde{v}} \ \wedge \ v \in S\right\}$

$$\xrightarrow{\quad \mathsf{A,S} \quad}$$ Choose uniformly random $y, z \leftarrow \mathbb{Z}_q$.

$$\xleftarrow{\quad y,z \quad}$$

$$\xrightarrow{\quad \mathsf{T}_1,\mathsf{T}_2 \quad}$$ Choose uniformly random $x \leftarrow \mathbb{Z}_q$.

$$\xleftarrow{\quad x \quad}$$

$$\xrightarrow{\quad t_x,\tilde{t}_x,\tilde{e} \quad}$$ Choose uniformly random $w \leftarrow \mathbb{Z}_q$.

$$\xleftarrow{\quad w \quad}$$

Compute $\delta(y,z) := z^3(1 - zn - \langle \mathbf{1}, \mathbf{s} \rangle) + (z - z^2)\langle \mathbf{1}, \mathbf{y}^n \rangle$.

Verify consistency of t_0.

Verify $1_{\mathbb{G}} \overset{?}{=} \mathsf{V}^{z^2} g^{\delta(y,z)} \mathsf{T}_1^x \mathsf{T}_2^{x^2} g^{-t_x} h^{-\tilde{t}_x}$.

Compute new generators.

Compute $\hat{g} := g^w$ and $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$.

Compute commitment for inner product.

Compute $\mathsf{P}' := (\mathbf{g} \uparrow -z\mathbf{1})(\mathbf{h} \uparrow z\mathbf{1} + z^2 \mathbf{y}^{-n} \circ \mathbf{s} + z^3 \mathbf{y}^{-n}) \hat{g}^{t_x} h^{-\tilde{e}} \mathsf{A} \mathsf{S}^x$.

Verify inner product proof for

$$\mathsf{PK}\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})\}.$$

$$\xleftarrow{\quad u_0, \ldots, u_{k-1} \quad}$$
$$\xrightarrow{\quad a, b, \{\mathsf{L}_j, \mathsf{R}_j\}_j \quad}$$

</> The verification right before the inner product proof can be merged with the verification of the inner product proof to increase the efficiency.

9.5.5 Set Non-membership Proof

This section describes a set non-membership proof that allows the prover to convince the verifier that a committed value v is *not* in a given set S.

Set non-membership proof from inner product. The goal of a *set non-membership proof* is for a prover to convince a verifier that a value v is not a member of S without revealing any additional information about v. That is, the prover knows the randomness $\tilde{v}$ of a public commitment $\mathsf{V} = g^v h^{\tilde{v}}$ to value v and wants to convince the verifier that $v \notin S$ without revealing v.

In the following, similar to the range proof in Section 9.5.3, the set non-membership proof is reduced to a single inner product proof. Let $S = \{s_0, \ldots, s_{n-1}\}$ and denote by $\mathbf{s} := (s_0, \ldots, s_{n-1})$ the vector[5] of set elements. If $v \notin S$, then $v \neq s_j$ for all $j \in \mathbb{Z}_q$, or stated differently $v - s_j \neq 0$. Using Lemma 29 from Section 9.2.12, this is equivalent to the existence of a_j such that $a_j(v - s_j) = 1$. That is, for $\mathbf{a}_L = (a_0, \ldots, a_{n-1})$ and $\mathbf{a}_R = v\mathbf{1}$,

$$\begin{aligned} \mathbf{a}_L \circ (\mathbf{a}_R - \mathbf{s}) &= \mathbf{1}, \\ \mathbf{a}_R &= v\mathbf{1}. \end{aligned} \tag{9.7}$$

Using polynomial evaluation at a challenge point y we can represent $v \notin S$ as inner product constraints:

$$\begin{aligned} \langle \mathbf{1}, \mathbf{y}^n \rangle &= \langle \mathbf{a}_L, (\mathbf{a}_R - \mathbf{s}) \circ \mathbf{y}^n \rangle \\ v\langle \mathbf{1}, \mathbf{y}^n \rangle &= \langle \mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle, \end{aligned}$$

where the second constraint ensures that $\mathbf{a}_R = v\mathbf{1}$.

The two inner product constraints are combined into a single constraint using polynomial evaluation at challenge point z resulting in:

$$zv\langle \mathbf{1}, \mathbf{y}^n \rangle + \langle \mathbf{1}, \mathbf{y}^n \rangle = \langle \mathbf{a}_L, (\mathbf{a}_R - \mathbf{s}) \circ \mathbf{y}^n \rangle + z\langle \mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle$$

---

[5]The order of elements in $\mathbf{s}$ does not matter.

Adding $\langle z\mathbf{1}, -\mathbf{s} \circ \mathbf{y}^n \rangle$ to both sides results in

$$\Leftrightarrow \quad zv\langle \mathbf{1}, \mathbf{y}^n \rangle + \langle \mathbf{1}, \mathbf{y}^n \rangle - z\langle \mathbf{1}, \mathbf{s} \circ \mathbf{y}^n \rangle = \langle \mathbf{a}_L + z\mathbf{1}, (\mathbf{a}_R - \mathbf{s}) \circ \mathbf{y}^n \rangle$$
$$\Leftrightarrow \quad zv\langle \mathbf{1}, \mathbf{y}^n \rangle + \langle \mathbf{1}, \mathbf{y}^n \rangle - z\langle \mathbf{s}, \mathbf{y}^n \rangle \quad = \langle \mathbf{a}_L + z\mathbf{1}, (\mathbf{a}_R - \mathbf{s}) \circ \mathbf{y}^n \rangle$$

For $\delta(y, z) := \langle \mathbf{1}, \mathbf{y}^n \rangle - z\langle \mathbf{s}, \mathbf{y}^n \rangle$, this is equivalent to:

$$\Leftrightarrow \quad zv\langle \mathbf{1}, \mathbf{y}^n \rangle + \delta(y, z) = \langle \mathbf{a}_L + z\mathbf{1}, (\mathbf{a}_R - \mathbf{s}) \circ \mathbf{y}^n \rangle$$

As for the range proof, the prover needs to blind the vectors of the inner product by choosing random blinding factors $\mathbf{s}_L, \mathbf{s}_R \in \mathbb{Z}_q^n$. Using polynomial evaluation at challenge point x we get

$$\mathbf{l}(x) = \mathbf{l}_0 + x\mathbf{l}_1 = (\mathbf{a}_L + x\mathbf{s}_L) + z\mathbf{1}$$
$$\mathbf{r}(x) = \mathbf{r}_0 + x\mathbf{r}_1 = \mathbf{y}^n \circ (\mathbf{a}_R + x\mathbf{s}_R - \mathbf{s})$$
$$t(x) = t_0 + xt_1 + x^2t_2 = \langle \mathbf{l}(x), \mathbf{r}(x) \rangle$$

where
$$t_0 = \langle \mathbf{l}_0, \mathbf{r}_0 \rangle, \quad t_2 = \langle \mathbf{l}_1, \mathbf{r}_1 \rangle, \quad t_1 = \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$$

Finally, the set non-membership proof boils down to show that at point x, we have $\langle \mathbf{l}(x), \mathbf{r}(x) \rangle = t(x)$ and $t_0 = zv\langle \mathbf{1}, \mathbf{y}^n \rangle + \delta(y, z)$.

Protocol.   We can now describe the set non-membership proof protocol with the following implicit public values: group $\mathbb{G}$, generators $g, h, \mathbf{g} = (g_0, \ldots, g_{n-1}), \mathbf{h} = (h_0, \ldots, h_{n-1})$, and set $S = \{s_0, \ldots, s_{n-1}\}$ (and set size n).

Protocol Prover for PK $\{(v, \tilde{v}) : \mathbf{V} = g^v h^{\tilde{v}} \wedge v \notin S\}$

Compute $\mathbf{a}_L := (v\mathbf{1} - \mathbf{s})^{-1}$ (the component-wise inverse) and set $\mathbf{a}_R := v\mathbf{1}$.

Select blinding factors.
Choose uniform random vectors $\mathbf{s}_L, \mathbf{s}_R \leftarrow \mathbb{Z}_q^n$.

Commit to vectors $\mathbf{a}_L, \mathbf{a}_R$ and blinding vectors.
Commit $\mathbf{A} := \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a})$ for uniform random $\tilde{a} \leftarrow \mathbb{Z}_q$.
Commit $\mathbf{S} := \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})$ for uniform random $\tilde{s} \leftarrow \mathbb{Z}_q$.

$\xrightarrow{\quad \mathsf{A,S} \quad}$

$$\xleftarrow{\quad y,z \quad}$$

Compute LHS polynomial coefficients.
Compute $\mathbf{l}_0 := \mathbf{a}_L + z\mathbf{1}$ and $\mathbf{l}_1 := \mathbf{s}_L$.

Compute RHS polynomial coefficients.
Compute $\mathbf{r}_0 := \mathbf{y}^n \circ (\mathbf{a}_R - \mathbf{s})$ and $\mathbf{r}_1 := \mathbf{y}^n \circ \mathbf{s}_R$.

Compute inner product polynomial coefficients.
Compute $t_0 := \langle \mathbf{l}_0, \mathbf{r}_0 \rangle$, $t_2 := \langle \mathbf{l}_1, \mathbf{r}_1 \rangle$, and
$t_1 := \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$.

Commit to inner product polynomial coefficients t_1 and t_2.
Commit $\mathsf{T}_1 := \mathsf{Com}(t_1; \tilde{t}_1)$ for uniform random $\tilde{t}_1 \leftarrow \mathbb{Z}_q$.
Commit $\mathsf{T}_2 := \mathsf{Com}(t_2; \tilde{t}_2)$ for uniform random $\tilde{t}_2 \leftarrow \mathbb{Z}_q$.

$$\xrightarrow{\quad \mathsf{T}_1,\mathsf{T}_2 \quad}$$

$$\xleftarrow{\quad x \quad}$$

Evaluate polynomials at x.
Compute $\mathbf{l}_x := \mathbf{l}_0 + x\mathbf{l}_1$, and $\mathbf{r}_x := \mathbf{r}_0 + x\mathbf{r}_1$, and $t_x := t_0 + xt_1 + x^2 t_2$.

Compute commitment opening information.
Compute $\tilde{t}_x := z\langle \mathbf{1}, \mathbf{y}^n \rangle \tilde{v} + x\tilde{t}_1 + x^2\tilde{t}_2$. Compute $\tilde{e} := \tilde{a} + x\tilde{s}$.

$$\xrightarrow{\quad t_x,\tilde{t}_x,\tilde{e} \quad}$$

$$\xleftarrow{\quad w \quad}$$

Compute new generators.
Compute $\hat{g} := g^w$ and set $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$.

Compute inner product commitment.
Compute $\mathsf{P}' := \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; t_x; \mathbf{g}, \mathbf{h}', \hat{g})$.

Execute inner product proof for
$\mathsf{PK}\left\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})\right\}.$

$$\xleftarrow{\quad u_0,\ldots,u_{k-1} \quad}$$
$$\xrightarrow{\quad a,b,\{\mathsf{L}_j,\mathsf{R}_j\}_j \quad}$$

Protocol Verifier for $\mathsf{PK}\left\{(v, \tilde{v}) : \mathsf{V} = g^v h^{\tilde{v}} \ \wedge \ v \notin S\right\}$

$$\xrightarrow{\quad \mathsf{A},\mathsf{S} \quad}$$ Choose uniformly random $y, z \leftarrow \mathbb{Z}_q$.
$$\xleftarrow{\quad y,z \quad}$$

$$\xrightarrow{\quad \mathsf{T}_1,\mathsf{T}_2 \quad}$$ Choose uniformly random $x \leftarrow \mathbb{Z}_q$.
$$\xleftarrow{\quad x \quad}$$

$$\xrightarrow{\quad t_x,\tilde{t}_x,\tilde{e} \quad}$$ Choose uniformly random $w \leftarrow \mathbb{Z}_q$.
$$\xleftarrow{\quad w \quad}$$

Compute $\delta(y,z) := \langle \mathbf{1}, \mathbf{y}^n \rangle - z\langle \mathbf{s}, \mathbf{y}^n \rangle$.

Verify consistency of t_0.

Verify $1_{\mathbb{G}} \stackrel{?}{=} \mathsf{V}^{z\langle \mathbf{1}, \mathbf{y}^n \rangle} g^{\delta(y,z)} \mathsf{T}_1^x \mathsf{T}_2^{x^2} g^{-t_x} h^{-\tilde{t}_x}$.

Compute new generators.

Compute $\hat{g} := g^w$ and $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$.

Compute commitment for inner product.

Compute $\mathsf{P}' := (\mathbf{g} \uparrow z\mathbf{1})(\mathbf{h} \uparrow -\mathbf{s})\hat{g}^{t_x} h^{-\tilde{e}} \mathsf{A} \mathsf{S}^x$.

Verify inner product proof for
$$\mathsf{PK}\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P}' = \mathsf{VCom}(\mathbf{l}_x, \mathbf{r}_x; \langle \mathbf{l}_x, \mathbf{r}_x \rangle; \mathbf{g}, \mathbf{h}', \hat{g})\}.$$

$$\xleftarrow{\quad u_0, \dots, u_{k-1} \quad}$$
$$\xrightarrow{\quad a, b, \{\mathsf{L}_j, \mathsf{R}_j\}_j \quad}$$

9.5.6 Proof of Security

Apart from completeness property, we require the following security properties for our Bulletproof-style protocols, namely range proof, set membership proof, and set non-membership proof protocols. We will in the following denote by $\mathcal{R}$ a NP relation. For pairs $(y,x) \in \mathcal{R}$ we call y the statement and x the witness. Further, pp denotes public parameters implicitly given as input to the prover and to the verifier.

Knowledge soundness: A (computationally bounded) adversary cannot produce an acceptable proof unless they "know" the witness. This is formalized by the existence of an efficient algorithm called "extractor" that can extract a witness from any polynomial-time adversary that can convince the verifier.

More formally, we say that a protocol $(\mathcal{P}, \mathcal{V})$ has computational knowledge soundness for NP relation $\mathcal{R}$ if for any deterministic polynomial time algorithm $\mathcal{P}^*$, there exists an efficient extractor $\mathcal{E}$ such that for all probabilistic polynomial time adversaries $\mathcal{A}$, the following probability is negligible in the security parameter.

$$\Pr[tr \text{ is accepting } \wedge (y,x) \notin \mathcal{R} | (y,s) \leftarrow \mathcal{A}(pp); tr \leftarrow \langle \mathcal{P}^*(s), \mathcal{V}(y) \rangle; x \leftarrow \mathcal{E}^{\langle \mathcal{P}^*(s), \mathcal{V}(y) \rangle}(pp, y)]$$

Here s can be seen as the state of the (deterministic) prover defined by $\mathcal{A}$. The definition can be explained as saying that if $\mathcal{P}^*$ in state s makes a convincing proof, then we can extract a valid witness.

Zero-knowledge: A proof should not leak any information about the witness beyond the fact that the statement is true. This is formalized by the existence of an efficient algorithm called "simulator" that can simulate the entire proof without knowing the witness. As in the Bulletproof paper [Bün+18], we show that the above protocols are *special honest verifier zero-knowledge* (SHVZK), meaning if the verifier's challenges are generated honestly, the simulator can simulate the proof without knowing the witness. Note that SHVZK is sufficient as the Fiat-Shamir transformation can then take a public-coin interactive protocol with SHVZK property and turn it into a non-interactive *zero-knowledge* proof in the random oracle model.

More formally, we say that a public-coin protocol $(\mathcal{P}, \mathcal{V})$ is (perfect) special honest verifier zero-knowledge for a relation $\mathcal{R}$ if there exists an efficient simulator $\mathcal{S}$ such that for any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$\Pr[(y,x) \in \mathcal{R} \wedge \mathcal{A}_2(tr) = 1 \mid (y,x,\rho) \leftarrow \mathcal{A}_1(pp), tr \leftarrow \langle \mathcal{P}(y,x), \mathcal{V}(y;\rho) \rangle]$$
$$= \Pr[(y,x) \in \mathcal{R} \wedge \mathcal{A}_2(tr) = 1 \mid (y,x,\rho) \leftarrow \mathcal{A}_1(pp), tr \leftarrow \mathcal{S}(y,\rho)]$$

where ρ is the public-coin randomness used by the verifier.

In order to show that our protocols satisfy these properties, we recall that the proven statement in all Bulletproof-style protocols can be reduced to a single inner product instance of the form $\langle \mathbf{w}_L, \mathbf{w}_R \rangle = w_O$, where $\mathbf{w}_L, \mathbf{w}_R, w_O$ are some functions in the witness. This implies that the prover cannot send these elements to the verifier in clear. To solve this problem, the prover instead runs a subprotocol that transforms the above inner product into a "zero-knowledge" inner product instance. It then invokes a logarithmic-sized proof system for proving correctness of this augmented instance.

Below we abstract this subprotocol and prove a lemma that shows it satisfies the desired security properties. The security of range proof and set (non-)membership proof protocols can then be shown by invoking this lemma and the inner product argument of [Bün+18] in a black-box way.

Abstract Zero-Knowledge Protocol

Let $g, h, \mathbf{g} = (g_0, \ldots, g_{n-1}), \mathbf{h} = (h_0, \ldots, h_{n-1})$ be some group generators such that no nontrivial discrete logarithm relations are known among them. Let $\mathtt{V} = g^v h^{\tilde{v}}$ be a commitment to a value v using randomness $\tilde{v}$. The goal of the abstract protocol is to prove knowledge of v and $\tilde{v}$ and vectors $\mathbf{a}_L, \mathbf{a}_R$ such that $\mathtt{V} = g^v h^{\tilde{v}}$ and

$$\forall y, z \in \mathbb{Z}_q^* \ f_3(v, y, z) = \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle,$$

where

$$
\begin{aligned}
f_1(\mathbf{a}_L, z)_i &= (\mathbf{a}_L)_i + p_{L,i}(z), \\
f_2(\mathbf{a}_R, y, z)_i &= (\mathbf{a}_R)_i + y^{-i} \cdot p_{R,i}(y, z), \\
f_3(v, y, z) &= \gamma(z) \cdot v + \delta(y, z),
\end{aligned}
$$

for some polynomials $p_{L,i}$, $p_{R,i}$, $\gamma \neq 0$, and δ. We denote by $\mathbf{p}_L(z)$ and $\mathbf{p}_R(y, z)$ the vectors $(p_{L,i}(z))_i$ and $(p_{R,i}(y, z))_i$, respectively. Different instantiations of the functions f_1, f_2, f_3 yield concrete protocols for different statements, including range-proofs, set-membership proofs, and set-non-membership proofs.

At a high-level, the protocol works as follows. The prover first computes a commitment $\mathtt{A} = \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a})$ to two witness-dependent vectors $\mathbf{a}_L, \mathbf{a}_R \in \mathbb{Z}_q^n$, that are involved in the left and right inputs of the inner product constraint, respectively. The prover sends $\mathtt{A}$ to the verifier along with another commitment $\mathtt{S} = \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})$ to two randomly chosen vectors $\mathbf{s}_L, \mathbf{s}_R \leftarrow \mathbb{Z}_q^n$ that will later be used to hide $\mathbf{a}_L, \mathbf{a}_R$.

Using challenges y, z from the verifier, the prover computes

$$
\begin{aligned}
\mathbf{w}_L &\coloneqq f_1(\mathbf{a}_L, z) = \mathbf{a}_L + \mathbf{p}_L(z), \\
\mathbf{w}_R &\coloneqq f_2(\mathbf{a}_R, y, z) = \mathbf{a}_R + \mathbf{y}^{-n} \circ \mathbf{p}_R(y, z), \\
w_O &\coloneqq f_3(v, y, z) = \gamma(z) \cdot v + \delta(y, z), \\
\widetilde{w_O} &\coloneqq \gamma(z) \cdot \tilde{v}.
\end{aligned}
$$

The prover then defines two linear vector polynomials $l(X), r(X) \in \mathbb{Z}_q^n[X]$, and a quadratic polynomial $t(X) \in \mathbb{Z}_q[X]$ as follows:

$$
\begin{aligned}
l(X) &= \mathbf{w}_L + \mathbf{s}_L \cdot X \\
r(X) &= (\mathbf{y}^n \circ \mathbf{w}_R) + (\mathbf{y}^n \circ \mathbf{s}_R) \cdot X \\
t(X) &= \langle l(X), r(X) \rangle = t_0 + t_1 \cdot X + t_2 \cdot X^2
\end{aligned}
$$

The prover computes commitments $\mathtt{T}_i = g^{t_i} h^{\tilde{t}_i}$ to the coefficient t_i of $t(X)$ for $i = 1, 2$, and sends them to the verifier. Next, after receiving a challenge point x, the prover evaluates $\mathbf{l}_x = l(x)$ and $\mathbf{r}_x = r(x)$ and sends to the verifier $\mathbf{l}_x, \mathbf{r}_x, t_x = \langle \mathbf{l}_x, \mathbf{r}_x \rangle$ along with some opening information $\tilde{t}_x = \widetilde{w_O} + x\tilde{t}_1 + x^2\tilde{t}_2$, and $\tilde{e} := \tilde{a} + x\tilde{s}$.

Given the commitment $\mathtt{A}$ to $\mathbf{a}_L, \mathbf{a}_R$, the verifier can compute a commitment $\mathtt{W}$ to $\mathbf{w}_L, \mathbf{w}_R$ using

$$\mathtt{A} = \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a}) \iff \mathtt{W} := \mathtt{A} \cdot \big(\mathbf{g} \uparrow \mathbf{p}_L(z)\big)\big(\mathbf{h} \uparrow \mathbf{p}_R(y, z)\big) = \mathsf{VCom}(\mathbf{w}_L, \mathbf{w}_R; \tilde{a}). \qquad (9.8)$$

Furthermore, the verifier can turn the commitment $\mathtt{V}$ into a commitment $\mathtt{W}_O$ to w_O using (for $\gamma(z) \neq 0$)

$$\mathtt{V} = g^v h^{\tilde{v}} \iff \mathtt{W}_O := \mathtt{V}^{\gamma(z)} g^{\delta(y,z)} = g^{w_O} h^{\widetilde{w_O}}.$$

Using this, the verifier can check that $\mathbf{l}_x$ and $\mathbf{r}_x$ are in fact $l(x)$ and $r(x)$, and that $t(x) = \langle \mathbf{l}_x, \mathbf{r}_x \rangle$.

We now describe the protocol in detail. Let witness $w := (v, \tilde{v}, \mathbf{a}_L, \mathbf{a}_R)$ and statement $st := \mathtt{V} = g^v h^{\tilde{v}} \wedge \forall y, z \in \mathbb{Z}_q^* \; f_3(v, y, z) = \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle$.

Protocol Prover for PK $\{w : st\}$

Implicitly known public values: $f_1, f_2, f_3, \mathbf{g}, \mathbf{h}, g, h$
Inputs to prover: $\mathtt{V}, v, \tilde{v}, \mathbf{a}_L, \mathbf{a}_R$

Select blinding factors.
Choose uniform random vectors $\mathbf{s}_L, \mathbf{s}_R \leftarrow \mathbb{Z}_q^n$.

Commit to vectors $\mathbf{a}_L, \mathbf{a}_R$, and blinding vectors.
Commit $\mathtt{A} := \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a})$ for uniform random $\tilde{a} \leftarrow \mathbb{Z}_q$.
Commit $\mathtt{S} := \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})$ for uniform random $\tilde{s} \leftarrow \mathbb{Z}_q$. $\xrightarrow{\mathtt{A},\mathtt{S}}$

$\xleftarrow{y,z}$

Compute vectors $\mathbf{w}_L, \mathbf{w}_R$, and values $w_O, \widetilde{w_O}$.
Compute $\mathbf{w}_L := f_1(\mathbf{a}_L, z)$ and $\mathbf{w}_R := f_2(\mathbf{a}_R, y, z)$.
Compute $w_O := f_3(v, y, z)$ and $\widetilde{w_O} := \gamma(z) \cdot \tilde{v}$.

Compute coefficients for LHS polynomial $l(X) = \mathbf{w}_L + \mathbf{s}_L \cdot X$.
Compute $\mathbf{l}_0 := \mathbf{w}_L$ and $\mathbf{l}_1 := \mathbf{s}_L$.

Compute coefficients for RHS polynomial
$r(X) = (\mathbf{y}^n \circ \mathbf{w}_R) + (\mathbf{y}^n \circ \mathbf{s}_R) \cdot X$.
Compute $\mathbf{r}_0 := \mathbf{y}^n \circ \mathbf{w}_R$ and $\mathbf{r}_1 := \mathbf{y}^n \circ \mathbf{s}_R$.

Compute inner product polynomial coefficients.
Compute $t_0 := \langle \mathbf{l}_0, \mathbf{r}_0 \rangle$, $t_2 := \langle \mathbf{l}_1, \mathbf{r}_1 \rangle$, and
$t_1 := \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2$.

Commit to inner product polynomial coefficients t_1 and t_2.
Commit $\mathtt{T}_1 := \mathsf{Com}(t_1; \tilde{t}_1)$ for uniform random $\tilde{t}_1 \leftarrow \mathbb{Z}_q$.
Commit $\mathtt{T}_2 := \mathsf{Com}(t_2; \tilde{t}_2)$ for uniform random $\tilde{t}_2 \leftarrow \mathbb{Z}_q$. $\xrightarrow{\mathtt{T}_1,\mathtt{T}_2}$

$$\xleftarrow{\quad x \quad}$$

Evaluate polynomials at x.
Compute $\mathbf{l}_x := \mathbf{l}_0 + x\mathbf{l}_1$, and $\mathbf{r}_x := \mathbf{r}_0 + x\mathbf{r}_1$, and $t_x := w_O + xt_1 + x^2 t_2$.

Compute commitment opening information.
Compute $\tilde{t}_x = \widetilde{w_O} + x\tilde{t}_1 + x^2 \tilde{t}_2$, and $\tilde{e} := \tilde{a} + x\tilde{s}$.

$$\xrightarrow{\quad t_x, \tilde{t}_x, \tilde{e} \quad}$$

Compute new generators $\mathbf{h}'$ and value P for inner product proof.
Compute $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$ and $\mathsf{P} := (\mathbf{g} \uparrow \mathbf{l}_x)(\mathbf{h}' \uparrow \mathbf{r}_x)$.

Execute inner product proof for
$\mathrm{PK}\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P} = (\mathbf{g} \uparrow \mathbf{l}_x)(\mathbf{h}' \uparrow \mathbf{r}_x) \wedge t_x = \langle \mathbf{l}_x, \mathbf{r}_x \rangle\}.$

$$\xleftarrow{\quad w, u_0, \ldots, u_{k-1} \quad}$$
$$\xrightarrow{\quad a, b, \{\mathsf{L}_j, \mathsf{R}_j\}_j \quad}$$

Note that $t_0 = \langle \mathbf{l}_0, \mathbf{r}_0 \rangle = \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle = f_3(v, y, z) = w_O$ for honestly generated values. Hence, the prover does not need to compute both values t_0 and w_O separately but can instead re-use the one that can be computed more efficiently.

Protocol Verifier for PK $\{w : st\}$

Implicitly known public values: $f_1, f_2, f_3, \mathbf{g}, \mathbf{h}, g, h$
Inputs to verifier: V

$$\xrightarrow{\quad \mathsf{A}, \mathsf{S} \quad}$$
$$\xleftarrow{\quad y, z \quad}$$

Choose uniformly random $y, z \leftarrow \mathbb{Z}_q$.

$$\xrightarrow{\quad \mathsf{T}_1, \mathsf{T}_2 \quad}$$
$$\xleftarrow{\quad x \quad}$$

Choose uniformly random $x \leftarrow \mathbb{Z}_q$.

Compute (homomorphically) commitments W and W_O
Compute $\mathsf{W} := \mathsf{A} \cdot (\mathbf{g} \uparrow \mathbf{p}_L(z))(\mathbf{h} \uparrow \mathbf{p}_R(y, z))$,
Compute $\mathsf{W}_O := \mathsf{V}^{\gamma(z)} g^{\delta(y,z)}$.

Check $t_x = t(x)$.

$$\xrightarrow{\quad t_x, \tilde{t}_x, \tilde{e} \quad}$$

Check $g^{t_x} h^{\tilde{t}_x} \overset{?}{=} \mathsf{W}_O \cdot \mathsf{T}_1^x \cdot \mathsf{T}_2^{x^2}$.

Compute new generators $\mathbf{h}'$ and value P for inner product proof.
Compute $\mathbf{h}' := (\mathbf{h}_i^{(\mathbf{y}^{-n})_i})_{i=0}^{n-1}$ and $\mathsf{P} := \mathsf{W}\mathsf{S}^x h^{-\tilde{e}}$.

Verify inner product proof for
$\mathrm{PK}\{(\mathbf{l}_x, \mathbf{r}_x) : \mathsf{P} = (\mathbf{g} \uparrow \mathbf{l}_x)(\mathbf{h}' \uparrow \mathbf{r}_x) \wedge t_x = \langle \mathbf{l}_x, \mathbf{r}_x \rangle\}.$

$$\xleftarrow{\quad w, u_0, \ldots, u_{k-1} \quad}$$
$$\xrightarrow{\quad a, b, \{\mathsf{L}_j, \mathsf{R}_j\}_j \quad}$$

The concrete protocols in the previous sections as well as our implementation do not use the abstraction $\mathsf{PK}\{(\mathbf{a}, \mathbf{b}) : \mathsf{P} = (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) \wedge c = \langle \mathbf{a}, \mathbf{b} \rangle\}$ but instead inline that protocol and directly use the protocol for $\mathsf{PK}\{(\mathbf{a}, \mathbf{b}) : \mathsf{P} = \mathsf{VCom}(\mathbf{a}, \mathbf{b}; \langle \mathbf{a}, \mathbf{b} \rangle; \mathbf{g}, \mathbf{h}, h)\}$. Note that this does not change the resulting proofs.

Furthermore, these protocols and our implementation use the same generator $g = \hat{g}$ for the commitment V and the internal generator in the protocol $\mathsf{PK}\{(\mathbf{a}, \mathbf{b}) : \mathsf{P} = (\mathbf{g} \uparrow \mathbf{a})(\mathbf{h} \uparrow \mathbf{b}) \wedge c = \langle \mathbf{a}, \mathbf{b} \rangle\}$. This requires one generator less in total. Note that this is not a security issue because Theorem 35 only requires an unknown discrete logarithm relation between the generators in $\mathbf{g}, \mathbf{h}, \tilde{g}$ and does not depend on g. Further, note that none of the protocols reveal any such relation about any of the generators since neither prover nor verifier know such a relation.

We show the above protocol satisfies completeness and the desired security properties. At a high-level, special honest verifier zero-knowledge follows in a straightforward way from the fact that $l(X)$ and $r(X)$ have at least one coefficient independent of the witness and thus can be efficiently simulated. To show knowledge soundness, we build an efficient extractor that takes $\mathbf{l}_x$ and $\mathbf{r}_x$ as input and either extracts a valid witness, or finds a nontrivial discrete logarithm relation between the generators $g, h, \mathbf{g}, \mathbf{h}$.

Lemma 36. *The protocol described above has perfect completeness.*

Proof. Let $(\mathsf{A}, \mathsf{S}, \mathsf{T}_1, \mathsf{T}_2, t_x, \tilde{t}_x, \tilde{e}, \pi_{\mathrm{IP}})$ be an honestly generated proof. In the first verification equation, the verifier checks whether

$$g^{t_x} h^{\tilde{t}_x} \overset{?}{=} \mathsf{W}_O \cdot \mathsf{T}_1^x \cdot \mathsf{T}_2^{x^2},$$

for $\mathsf{W}_O := \mathsf{V}^{\gamma(z)} g^{\delta(y,z)}$. For $\mathsf{V} = g^v h^{\tilde{v}}$ and honestly generated values $w_O := f_3(v, y, z) = \gamma(z) \cdot v + \delta(y, z)$, $\widetilde{w_O} := \gamma(z) \cdot \tilde{v}$, $t_x := w_O + xt_1 + x^2 t_2$, $\tilde{t}_x = \widetilde{w_O} + x\tilde{t}_1 + x^2 \tilde{t}_2$, $\mathsf{T}_1 := \mathsf{Com}(t_1; \tilde{t}_1)$, and $\mathsf{T}_2 := \mathsf{Com}(t_2; \tilde{t}_2)$, we have

$$
\begin{aligned}
\mathsf{W}_O \cdot \mathsf{T}_1^x \cdot \mathsf{T}_2^{x^2} &= \mathsf{V}^{\gamma(z)} g^{\delta(y,z)} \cdot g^{xt_1} h^{x\tilde{t}_1} \cdot g^{x^2 t_2} h^{x^2 \tilde{t}_2} \\
&= g^{\gamma(z) \cdot v + \delta(y,z) + xt_1 + x^2 t_2} h^{\gamma(z) \cdot \tilde{v} + x\tilde{t}_1 + x^2 \tilde{t}_2} \\
&= g^{w_O + xt_1 + x^2 t_2} h^{\widetilde{w_O} + x\tilde{t}_1 + x^2 \tilde{t}_2} \\
&= g^{t_x} h^{\tilde{t}_x}.
\end{aligned}
$$

Hence, this check always succeeds.

Next, the prover computes P as

$$
\begin{aligned}
\mathsf{P} &= (\mathbf{g} \uparrow \mathbf{l}_x)(\mathbf{h}' \uparrow \mathbf{r}_x) \\
&= (\mathbf{g} \uparrow \mathbf{l}_0 + x\mathbf{l}_1)(\mathbf{h} \uparrow \mathbf{y}^{-n} \circ (\mathbf{r}_0 + x\mathbf{r}_1)) \\
&= (\mathbf{g} \uparrow f_1(\mathbf{a}_L, z) + x \cdot \mathbf{s}_L)(\mathbf{h} \uparrow f_2(\mathbf{a}_R, y, z) + x \cdot \mathbf{s}_R) \\
&= (\mathbf{g} \uparrow \mathbf{a}_L + \mathbf{p}_L(z) + x \cdot \mathbf{s}_L)(\mathbf{h} \uparrow \mathbf{a}_R + \mathbf{p}_R(y, z) + x \cdot \mathbf{s}_R).
\end{aligned}
$$

The verifier computes P as $\mathsf{W}\mathsf{S}^x h^{-\tilde{e}}$ with $\mathsf{W} = \mathsf{A} \cdot (\mathbf{g} \uparrow \mathbf{p}_L(z))(\mathbf{h} \uparrow \mathbf{p}_R(y, z))$, $\mathsf{A} = \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{a})$, $\mathsf{S} := \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})$, and $\tilde{e} := \tilde{a} + x\tilde{s}$. Hence,

$$
\begin{aligned}
\mathsf{W}\mathsf{S}^x h^{-\tilde{e}} &= \mathsf{A} \cdot (\mathbf{g} \uparrow \mathbf{p}_L(z))(\mathbf{h} \uparrow \mathbf{p}_R(y, z)) \cdot \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s})^x \cdot h^{-\tilde{e}} \\
&= (\mathbf{g} \uparrow \mathbf{a}_L)(\mathbf{h} \uparrow \mathbf{a}_R) h^{\tilde{a}} \cdot (\mathbf{g} \uparrow \mathbf{p}_L(z))(\mathbf{h} \uparrow \mathbf{p}_R(y, z)) \cdot (\mathbf{g} \uparrow x \cdot \mathbf{s}_L)(\mathbf{h} \uparrow x \cdot \mathbf{s}_R) h^{x \cdot \tilde{s}} \cdot h^{-\tilde{e}} \\
&= (\mathbf{g} \uparrow \mathbf{a}_L + \mathbf{p}_L(z) + x \cdot \mathbf{s}_L)(\mathbf{h} \uparrow \mathbf{a}_R + \mathbf{p}_R(y, z) + x \cdot \mathbf{s}_R).
\end{aligned}
$$

Therefore, prover and verifier compute the same value $\mathtt{P}$.

Finally, prover and verifier engage in the inner product argument for

$$\mathtt{PK}\left\{(\mathbf{l}_x, \mathbf{r}_x) : \mathtt{P} = (\mathbf{g} \uparrow \mathbf{l}_x)(\mathbf{h}' \uparrow \mathbf{r}_x) \wedge t_x = \langle \mathbf{l}_x, \mathbf{r}_x \rangle\right\}.$$

Note that we have for the values computed by the prover

$$
\begin{aligned}
\langle \mathbf{l}_x, \mathbf{r}_x \rangle &= \langle \mathbf{l}_0 + x\mathbf{l}_1, \mathbf{r}_0 + x\mathbf{r}_1 \rangle \\
&= \langle \mathbf{l}_0, \mathbf{r}_0 \rangle + x(\langle \mathbf{l}_0, \mathbf{r}_1 \rangle + \langle \mathbf{l}_1, \mathbf{r}_0 \rangle) + x^2 \langle \mathbf{l}_1, \mathbf{r}_1 \rangle \\
&= \langle \mathbf{w}_L, \mathbf{y}^n \circ \mathbf{w}_R \rangle + xt_1 + x^2 t_2 \\
&= \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle + xt_1 + x^2 t_2 \\
&= f_3(v, y, z) + xt_1 + x^2 t_2 \\
&= w_O + xt_1 + x^2 t_2 \\
&= t_x.
\end{aligned}
$$

Thus, the perfect completeness property of the inner-product argument (see Theorem 35) implies that the verifier always accepts the inner-product argument. This completes the proof of perfect completeness. $\qquad\square$

Lemma 37. *The protocol described above has statistical special honest verifier zero-knowledge.*

Proof. To show special honest verifier zero-knowledge, an efficient simulator can be constructed by selecting all the proof elements and challenges at random or computing them directly according to the protocol description. More precisely, the simulator proceeds as follows:

- Choose uniformly random $\mathtt{A}, \mathtt{T}_2 \leftarrow \mathbb{G}$

- Choose uniformly random $x, y, z \leftarrow \mathbb{Z}_q$

- Choose uniformly random $\mathbf{l}_x, \mathbf{r}_x \leftarrow \mathbb{Z}_q^n$

- Compute $t_x = \langle \mathbf{l}_x, \mathbf{r}_x \rangle$

- Choose uniformly random $\tilde{t}_x, \tilde{e} \leftarrow \mathbb{Z}_q$

Next, the simulator computes the following values according to the verification equations (and aborts if $x = 0$):

$$
\begin{aligned}
\mathtt{W} &:= \mathtt{A} \cdot \left(\mathbf{g} \uparrow \mathbf{p}_L(z)\right)\left(\mathbf{h} \uparrow \mathbf{p}_R(y, z)\right), \\
\mathtt{W}_O &:= \mathtt{V}^{\gamma(z)} g^{\delta(y,z)}, \\
\mathtt{P} &:= (\mathbf{g} \uparrow \mathbf{l}_x)(\mathbf{h}' \uparrow \mathbf{r}_x), \\
\mathtt{S} &:= \left(\mathtt{W}^{-1}\mathtt{P}h^{\tilde{e}}\right)^{x^{-1}}, \\
\mathtt{T}_1 &:= \left(g^{t_x} h^{\tilde{t}_x} \mathtt{W}_O^{-1} \mathtt{T}_2^{-x^2}\right)^{x^{-1}}.
\end{aligned}
\tag{9.9}
$$

Finally, the simulator runs the inner-product argument with the simulated witness $(\mathbf{l}_x, \mathbf{r}_x)$ and the verifier's randomness. Note that the inner product argument remains zero knowledge as we can successfully simulate the witness. Thus, revealing the witness or leaking information about it does not change the zero-knowledge property of the overall protocol. The simulator returns the transcript $(\mathtt{A}, \mathtt{S}; y, z; \mathtt{T}_1, \mathtt{T}_2; x; t_x, \tilde{t}_x, \tilde{e}; \pi_{\mathrm{IP}})$, where π_{IP} is the simulated inner-product proof.

We show that the distribution of the real transcript and the distribution of the simulated transcript for the non-challenge part $(\mathtt{A}, \mathtt{S}, \mathtt{T}_1, \mathtt{T}_2, t_x, \tilde{t}_x, \tilde{e}, \pi_{\mathrm{IP}})$ are identical. Let $\xi_{\mathbf{g}} \in \mathbb{Z}_q^n, \xi_{\mathbf{h}} \in \mathbb{Z}_q^n, \xi_g \in \mathbb{Z}_q$

be respectively the discrete logarithms of $\mathbf{g}, \mathbf{h}, g$ with base h, i.e.,

$$\xi_{\mathbf{g}} := \mathrm{dlog}_h(\mathbf{g}), \qquad \xi_{\mathbf{h}} := \mathrm{dlog}_h(\mathbf{h}), \qquad \xi_g := \mathrm{dlog}_h(g).$$

We observe that for a real transcript $(\mathtt{A}, \mathtt{S}, \mathtt{T}_1, \mathtt{T}_2, t_x, \tilde{t}_x, \tilde{e}, \pi_{\mathrm{IP}})$ created by an honest prover and verifier, given $x, y \neq 0$, there is a one-to-one correspondence between $(\mathrm{dlog}_h(\mathtt{A}), \mathrm{dlog}_h(\mathtt{T}_2), \tilde{t}_x, \tilde{e}, \mathbf{l}_x, \mathbf{r}_x)$ and the (uniformly random) field elements $(\tilde{a}, \tilde{t}_1, \tilde{t}_2, \tilde{s}, \mathbf{s}_L, \mathbf{s}_R) \in \mathbb{Z}_q^{4+2n}$.[6]

$$
\begin{bmatrix} \mathrm{dlog}_h(\mathtt{A}) \\ \mathrm{dlog}_h(\mathtt{T}_2) \\ \tilde{t}_x \\ \tilde{e} \\ \mathbf{l}_x \\ \mathbf{r}_x \end{bmatrix}
=
\begin{bmatrix} \langle \xi_{\mathbf{g}}, \mathbf{a}_L \rangle + \langle \xi_{\mathbf{h}}, \mathbf{a}_R \rangle \\ 0 \\ \widetilde{w_O} \\ 0 \\ \mathbf{w}_L \\ \mathbf{y}^n \circ \mathbf{w}_R \end{bmatrix}
+
\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & \xi_g \\ 0 & x & x^2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x\mathbf{y}^n & 0 \end{bmatrix}
\begin{bmatrix} \tilde{a} \\ \tilde{t}_1 \\ \tilde{t}_2 \\ \tilde{s} \\ \mathbf{s}_L \\ \mathbf{s}_R \\ \langle \mathbf{s}_L, \mathbf{y}^n \circ \mathbf{s}_R \rangle \end{bmatrix}
$$

The one-to-one correspondence can be shown by the fact that if two different tuples $(\tilde{a}^{(1)}, \tilde{t}_1^{(1)}, \tilde{t}_2^{(1)}, \tilde{s}^{(1)}, \mathbf{s}_L^{(1)}, \mathbf{s}_R^{(1)})$ and $(\tilde{a}^{(2)}, \tilde{t}_1^{(2)}, \tilde{t}_2^{(2)}, \tilde{s}^{(2)}, \mathbf{s}_L^{(2)}, \mathbf{s}_R^{(2)})$ map to the same transcript elements $(\mathrm{dlog}_h(\mathtt{A}), \mathrm{dlog}_h(\mathtt{T}_2), \tilde{t}_x, \tilde{e}, \mathbf{l}_x, \mathbf{r}_x)$, then we have

$$
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
=
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
+
\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & \xi_g \\ 0 & x & x^2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x\mathbf{y}^n & 0 \end{bmatrix}
\begin{bmatrix} \tilde{a}^{(1)} - \tilde{a}^{(2)} \\ \tilde{t}_1^{(1)} - \tilde{t}_1^{(2)} \\ \tilde{t}_2^{(1)} - \tilde{t}_2^{(2)} \\ \tilde{s}^{(1)} - \tilde{s}^{(2)} \\ \mathbf{s}_L^{(1)} - \mathbf{s}_L^{(2)} \\ \mathbf{s}_R^{(1)} - \mathbf{s}_R^{(2)} \\ \langle \mathbf{s}_L^{(1)}, \mathbf{y}^n \circ \mathbf{s}_R^{(1)} \rangle - \langle \mathbf{s}_L^{(2)}, \mathbf{y}^n \circ \mathbf{s}_R^{(2)} \rangle \end{bmatrix}
$$

which implies that

$$\tilde{a}^{(1)} = \tilde{a}^{(2)} \;\textit{// implied by the first row}$$
$$\mathbf{s}_L^{(1)} = \mathbf{s}_L^{(2)} \;\textit{// implied by the fifth row}$$
$$\mathbf{s}_R^{(1)} = \mathbf{s}_R^{(2)} \;\textit{// implied by the sixth row}$$
$$\tilde{s}^{(1)} = \tilde{s}^{(2)} \;\textit{// implied by the fourth row and that } \tilde{a}^{(1)} = \tilde{a}^{(2)}$$
$$\tilde{t}_2^{(1)} = \tilde{t}_2^{(2)} \;\textit{// implied by the second row and that } \mathbf{s}_L^{(1)} = \mathbf{s}_L^{(2)} \textit{ and } \mathbf{s}_R^{(1)} = \mathbf{s}_R^{(2)}$$
$$\tilde{t}_1^{(1)} = \tilde{t}_1^{(2)} \;\textit{// implied by the third row and that } \tilde{t}_2^{(1)} = \tilde{t}_2^{(2)}$$

Since the prover chooses the tuple $(\tilde{a}, \tilde{t}_1, \tilde{t}_2, \tilde{s}, \mathbf{s}_L, \mathbf{s}_R)$ uniformly at random, the one-to-one correspondence implies that (for $x, y \neq 0$) $(\mathtt{A}, \mathtt{T}_2, \tilde{t}_x, \tilde{e}, \mathbf{l}_x, \mathbf{r}_x)$ are also distributed uniformly. Since the simulator chooses the latter values uniformly, those are distributed equally in an honest execution and the simulation. Furthermore, the verifier accepts an honest proof with probability 1 (see Lemma 36). The checks the verifier performs imply that $t_x = \langle \mathbf{l}_x, \mathbf{r}_x \rangle$ and the values $(\mathtt{S}, \mathtt{T}_1)$ must be equal to how they are computed in Equation (9.9). Hence, for $x, y \neq 0$, the remaining transcript values $(t_x, \mathtt{S}, \mathtt{T}_1)$ are uniquely determined. We can thus conclude that conditioned on $x, y \neq 0$, a simulated transcript is equally distributed as a real transcript. Therefore, the statistical distance between such transcripts is negligible. $\qquad\square$

---

[6]Note that taking the discrete logarithm of group elements here is sufficient to analyze their distribution in the transcript.

To prove soundness, the following simple lemma will be useful.

Lemma 38. *Let $n \in \mathbb{N}$, let $\mathbb{G}$ be a group of prime order q, let $A_0, \ldots, A_n \in \mathbb{G}$, and let $x_0, \ldots, x_n \in \mathbb{Z}_q$ be $n+1$ distinct numbers such that*

$$\forall j \in: \prod_{i=0}^{n} A_i^{x_j^i} = 1.$$

Then, $A_0 = \ldots = A_n = 1$.

Proof. Since q is prime, there exist a generator g of $\mathbb{G}$ and $a_0, \ldots, a_n \in \mathbb{Z}_q$ such that $A_i = g^{a_i}$ for $i \in$. Hence,

$$\forall j \in: 1 = \prod_{i=0}^{n} A_i^{x_j^i} = \prod_{i=0}^{n} g^{a_i x_j^i} = g^{\sum_{i=0}^{n} a_i x_j^i}.$$

Since g is a generator, this implies that $\sum_{i=0}^{n} a_i x_j^i = 0 \in \mathbb{Z}_q$ for $j \in$. Because q is prime, $\mathbb{Z}_q$ is a field, so a nonzero polynomial of degree n can have at most n distinct roots. Therefore, we have $a_i = 0$ for all i, which implies $A_0 = \ldots = A_n = 1$. $\qquad\square$

Lemma 39. *Assuming no nontrivial discrete logarithm relation between $\mathbf{g}, \mathbf{h}, g, h$ can be found, the protocol described above has computational knowledge soundness for the statement*

$$\mathrm{PK}\left\{(v, \tilde{v}, \mathbf{a}_L, \mathbf{a}_R) : \mathtt{V} = g^v h^{\tilde{v}} \wedge \forall y, z \in \mathbb{Z}_q^* \ f_3(v, y, z) = \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z)\rangle\right\}.$$

Proof. In order to show knowledge soundness, we construct an efficient extractor $\mathcal{E}$ that extracts a valid witness by rewinding the prover and invoking the inner-product argument extractor. The extractor $\mathcal{E}$ proceeds as follows. It rewinds the prover using two distinct x challenges x_1 and x_2 (and the same $y, z, \mathtt{A}, \mathtt{S}$) to obtain $(t_{x_1}, \tilde{t}_{x_1}, \tilde{e}_1), (t_{x_2}, \tilde{t}_{x_2}, \tilde{e}_2)$ from the transcripts. It then invokes the extractor for the inner-product argument (guaranteed to exist by Theorem 35) to extract witnesses $(\mathbf{l}_{x_i}, \mathbf{r}_{x_i})$ (using $O(n^2)$ transcripts) such that $\mathtt{P} = (\mathbf{g} \uparrow \mathbf{l}_{x_i})(\mathbf{h}' \uparrow \mathbf{r}_{x_i})$ and $t_{x_i} = \langle \mathbf{l}_{x_i}, \mathbf{r}_{x_i}\rangle$ for $i \in$. The two values $\mathbf{l}_{x_1}, \mathbf{l}_{x_2}$ for x_1 and x_2 uniquely determine a linear function $l(X) = \mathbf{l}_0 + X\mathbf{l}_1$ such that $l(x_i) = \mathbf{l}_{x_1}$ for $i \in$. Similarly, $\mathbf{r}_{x_1}, \mathbf{r}_{x_2}$ and $\tilde{e}_1, \tilde{e}_2$ define linear functions r and $\tilde{e}$. We can thus find values $\mathbf{l}_0, \mathbf{l}_1, \mathbf{r}_0, \mathbf{r}_1, \tilde{w}, \tilde{s}$ such that

$$\begin{aligned}
\mathbf{l}_{x_1} &= l(x_1) = \mathbf{l}_0 + x_1\mathbf{l}_1, & \mathbf{r}_{x_1} &= r(x_1) = \mathbf{r}_0 + x_1\mathbf{r}_1, & \tilde{e}_1 &= \tilde{e}(x_1) = \tilde{w} + x_1\tilde{s}, \\
\mathbf{l}_{x_2} &= l(x_2) = \mathbf{l}_0 + x_2\mathbf{l}_1, & \mathbf{r}_{x_2} &= r(x_2) = \mathbf{r}_0 + x_2\mathbf{r}_1, & \tilde{e}_2 &= \tilde{e}(x_2) = \tilde{w} + x_2\tilde{s}.
\end{aligned}$$

Now define $\mathbf{w}_L := \mathbf{l}_0$, $\mathbf{s}_L := \mathbf{l}_1$, $\mathbf{w}_R := \mathbf{y}^{-n} \circ \mathbf{r}_0$, and $\mathbf{s}_R := \mathbf{y}^{-n} \circ \mathbf{r}_1$. This implies

$$\langle \mathbf{w}_L, \mathbf{y}^n \circ \mathbf{w}_R\rangle = \langle \mathbf{l}_0, \mathbf{r}_0\rangle.$$

For $i \in$, the verifier sets

$$\mathtt{P}_i := \mathtt{WS}^{x_i} h^{-\tilde{e}_i}.$$

Hence,

$$\begin{aligned}
\mathtt{WS}^{x_i} &= (\mathbf{g} \uparrow \mathbf{l}_{x_i})(\mathbf{h}' \uparrow \mathbf{r}_{x_i})h^{\tilde{e}_i} \\
&= (\mathbf{g} \uparrow \mathbf{l}_0 + x_i\mathbf{l}_1)(\mathbf{h}' \uparrow \mathbf{r}_0 + x_i\mathbf{r}_1)h^{\tilde{w}+x_i\tilde{s}} \\
&= (\mathbf{g} \uparrow \mathbf{w}_L + x_i\mathbf{s}_L)(\mathbf{h}' \uparrow \mathbf{y}^n \circ (\mathbf{w}_R + x_i\mathbf{s}_R))h^{\tilde{w}+x_i\tilde{s}} \\
&= (\mathbf{g} \uparrow \mathbf{w}_L + x_i\mathbf{s}_L)(\mathbf{h} \uparrow \mathbf{w}_R + x_i\mathbf{s}_R)h^{\tilde{w}+x_i\tilde{s}} \\
&= (\mathbf{g} \uparrow \mathbf{w}_L)(\mathbf{h} \uparrow \mathbf{w}_R)h^{\tilde{w}} \cdot ((\mathbf{g} \uparrow \mathbf{s}_L)(\mathbf{h} \uparrow \mathbf{s}_R)h^{\tilde{s}})^{x_i}.
\end{aligned}$$

Note that all values except for x_i in the above equality are the same for $i = 1$ and $i = 2$ and $x_1 \neq x_2$. We can therefore use Lemma 38 to conclude that

$$\mathsf{W} = (\mathbf{g} \uparrow \mathbf{w}_L)(\mathbf{h} \uparrow \mathbf{w}_R)h^{\tilde{w}} = \mathsf{VCom}(\mathbf{w}_L, \mathbf{w}_R; \tilde{w}) \quad \text{and}$$
$$\mathsf{S} = (\mathbf{g} \uparrow \mathbf{s}_L)(\mathbf{h} \uparrow \mathbf{s}_R)h^{\tilde{s}} = \mathsf{VCom}(\mathbf{s}_L, \mathbf{s}_R; \tilde{s}).$$

Note that the same values $\mathbf{w}_L, \mathbf{w}_R, \mathbf{s}_L, \mathbf{s}_R, \tilde{w}, \tilde{s}$ (and consequently $\mathbf{l}_0$, $\mathbf{l}_1$, $\mathbf{r}_0$, and $\mathbf{r}_1$) will be computed for all transcriptions with additional x_i because otherwise, a nontrivial discrete logarithm relation between $\mathbf{g}, \mathbf{h}, h$ could be found from the equations above. In particular, for all challenges X resulting in a value t_X, we have

$$t_X = \langle l(X), r(X) \rangle.$$

The extractor then computes

$$\mathbf{a}_L := \mathbf{w}_L - \mathbf{p}_L(z),$$
$$\mathbf{a}_R := \mathbf{w}_R - \mathbf{y}^{-n} \circ \mathbf{p}_R(y, z).$$

Since the verifier computes $\mathsf{W} = \mathsf{A} \cdot (\mathbf{g} \uparrow \mathbf{p}_L(z))(\mathbf{h} \uparrow \mathbf{p}_R(y, z))$ and we have $\mathbf{w}_L = f_1(\mathbf{a}_L, z)$ and $\mathbf{w}_R = f_2(\mathbf{a}_R, y, z)$, we can use Equation (9.8) to conclude that

$$\mathsf{A} = \mathsf{VCom}(\mathbf{a}_L, \mathbf{a}_R; \tilde{w}).$$

This implies that the same values $\mathbf{a}_L$, $\mathbf{a}_R$, and $\tilde{w}$ will be computed by the extractor for different x, y, z challenges (with the same A and S) since otherwise, a nontrivial discrete logarithm relation between $\mathbf{g}, \mathbf{h}$, and h would be found.

Next, the extractor $\mathcal{E}$ considers transcripts for three distinct x challenges x_1, x_2, x_3 (and the same $y, z, \mathsf{T}_1, \mathsf{T}_2$, possibly, but not necessarily with the same x_1, x_2 as above) to obtain $(t_{x_i}, \tilde{t}_{x_i})$, for $i \in$ from the transcripts. Similarly, as above, the three different t_{x_i} and $\tilde{t}_{x_i}$ correspond to unique polynomials $t, \tilde{t}$ of degree 2 with coefficients $w_O := t_0, t_1, t_2$ and $\widetilde{w_O} := \tilde{t}_0, \tilde{t}_1, \tilde{t}_2$, respectively. That is,

$$t_{x_i} = t(x_i) = w_O + x_i t_1 + x_i^2 t_2 \quad \text{and} \quad \tilde{t}_{x_i} = \tilde{t}(x_i) = \widetilde{w_O} + x_i \tilde{t}_1 + x_i^2 \tilde{t}_2$$

for $i \in$. The extractor can thus efficiently compute these coefficients solving the system of linear equations. For all i, the verifier checks that

$$\mathsf{W}_O \cdot \mathsf{T}_1^{x_i} \cdot \mathsf{T}_2^{x_i^2} = g^{t_{x_i}} h^{\tilde{t}_{x_i}}$$
$$= g^{w_O + x_i t_1 + x_i^2 t_2} h^{\widetilde{w_O} + x_i \tilde{t}_1 + x_i^2 \tilde{t}_2}$$
$$= (g^{w_O} h^{\widetilde{w_O}}) \cdot (g^{t_1} h^{\tilde{t}_1})^{x_i} \cdot (g^{t_2} h^{\tilde{t}_2})^{x_i^2}.$$

Hence, we can again use Lemma 38 to infer that

$$\mathsf{W}_O = g^{w_O} h^{\widetilde{w_O}}, \quad \mathsf{T}_1 = g^{t_1} h^{\tilde{t}_1}, \quad \mathsf{T}_2 = g^{t_2} h^{\tilde{t}_2}.$$

Since the verifier sets $\mathsf{W}_O = \mathsf{V}^{\gamma(z)} g^{\delta(y,z)}$, this implies

$$\mathsf{V}^{\gamma(z)} g^{\delta(y,z)} = g^{w_O} h^{\widetilde{w_O}}.$$

If $\gamma(z) = 0$, the extractor repeats for another z. Otherwise, it computes

$$v := (w_O - \delta(y, z)) \cdot \gamma(z)^{-1},$$
$$\tilde{v} := \widetilde{w_O} \cdot \gamma(z)^{-1}.$$

We then have

$$\mathtt{V} = g^v h^{\tilde v}.$$

This implies that the same v and $\tilde v$ will be computed for repeated extractions with different challenges x, y, z. Furthermore, we know from above that for $i \in$,

$$t(x_i) = \langle l(x_i), r(x_i) \rangle.$$

Since the polynomials on both sides of the equality are of degree at most 2, and they agree on 3 different values, the two polynomials must be equal. Hence,

$$w_O = t(0) = \langle l(0), r(0) \rangle = \langle \mathbf{w}_L, \mathbf{y}^n \circ \mathbf{w}_R \rangle.$$

We then have

$$f_3(v, y, z) = \gamma(z) \cdot v + \delta(y, z) = w_O = \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle.$$

Note that $f_3(v, y, z) - \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle$ is a polynomial in y and z and all coefficients are the same for repeated extractions with different y and z as argued above. Hence, the extractor can repeat this process for different y and z exceeding the maximal y and z-degrees of the polynomial. Since it always evaluates to 0, we can conclude that it must be the zero-polynomial. In particular, the values on the left- and right-hand side of the equation above are equal for all $y, z \in \mathbb{Z}_q^*$. The forking lemma proven in [Bün+18] implies that the nested repetitions of the extractor yield a sound overall extractor. $\square$

Lemmata 36, 37, and 39 together imply the following theorem.

Theorem 40. *Assume no nontrivial discrete logarithm relation between $\mathbf{g}, \mathbf{h}, g, h$ can be found. Then the protocol above has perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness for the statement*

$$\mathtt{PK}\left\{ (v, \tilde v, \mathbf{a}_L, \mathbf{a}_R) : \mathtt{V} = g^v h^{\tilde v} \wedge \forall y, z \in \mathbb{Z}_q^* \ f_3(v, y, z) = \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle \right\}.$$

Instantiation for range proof.

Corollary 41. *Assume no nontrivial discrete logarithm relation between $\mathbf{g}, \mathbf{h}, g, h$ can be found, and instantiate the above protocol with the following functions:*

$$f_1(\mathbf{a}_L, z) := \mathbf{a}_L - z\mathbf{1},$$
$$f_2(\mathbf{a}_R, y, z) := \mathbf{a}_R + z\mathbf{1} + z^2 \mathbf{y}^{-n} \circ \mathbf{2}^n,$$
$$f_3(v, y, z) := z^2 \cdot v + \delta(y, z),$$
$$\delta(y, z) := (z - z^2)\langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{2}^n \rangle.$$

Then the protocol above has perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness for the statement

$$\mathtt{PK}\left\{ (v, \tilde v) : \mathtt{V} = g^v h^{\tilde v} \wedge v \in [0, 2^n) \right\}.$$

Proof. Theorem 40 implies perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness for

$$z^2 \cdot v + (z - z^2)\langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{2}^n \rangle$$
$$= \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle$$
$$= \langle \mathbf{a}_L - z\mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1} + z^2 \mathbf{y}^{-n} \circ \mathbf{2}^n) \rangle$$
$$= \langle \mathbf{a}_L, \mathbf{y}^n \circ \mathbf{a}_R \rangle + z\langle \mathbf{a}_L, \mathbf{y}^n \rangle + z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle - z\langle \mathbf{y}^n, \mathbf{a}_R \rangle - z^2 \langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{2}^n \rangle.$$

for all $y, z \in \mathbb{Z}_q^*$. This is equivalent to

$$0 = \langle \mathbf{a}_L, \mathbf{y}^n \circ \mathbf{a}_R \rangle + z \langle \mathbf{a}_L - \mathbf{a}_R - \mathbf{1}, \mathbf{y}^n \rangle + z^2 (\langle \mathbf{a}_L, \mathbf{2}^n \rangle - v).$$

Since this holds for all $y, z \in \mathbb{Z}_q^*$, this is equivalent to

$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0} \qquad\qquad\qquad \text{// from } z^0 \text{ coefficient}$$
$$\mathbf{a}_L - \mathbf{1} = \mathbf{a}_R \qquad\qquad\qquad \text{// from } z^1 \text{ coefficient}$$
$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v \qquad\qquad\qquad \text{// from } z^2 \text{ coefficient}$$

These are exactly the linear constraints we have in (9.3) in Section 9.5.3. Hence, the existence of such $\mathbf{a}_L, \mathbf{a}_R$ is equivalent to $v \in [0, 2^n)$ as argued there. $\qquad\square$

Instantiation for set-membership.

Corollary 42. *Let $\mathbf{s} \in \mathbb{Z}_q^n$, assume no nontrivial discrete logarithm relation between $\mathbf{g}, \mathbf{h}, g, h$ can be found, and instantiate the above protocol with the following functions:*

$$f_1(\mathbf{a}_L, z) := \mathbf{a}_L - z\mathbf{1},$$
$$f_2(\mathbf{a}_R, y, z) := \mathbf{a}_R + z\mathbf{1} + \mathbf{y}^{-n} \circ (z^3 \mathbf{1} + z^2 \mathbf{s}),$$
$$f_3(v, y, z) := z^2 \cdot v + \delta(y, z),$$
$$\delta(y, z) := z^3 (1 - zn - \langle \mathbf{1}, \mathbf{s} \rangle) + (z - z^2)\langle \mathbf{1}, \mathbf{y}^n \rangle.$$

Then the protocol above has perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness for the statement

$$\mathtt{PK}\left\{ (v, \tilde{v}) : \mathtt{V} = g^v h^{\tilde{v}} \wedge v \in \mathbf{s} \right\}.$$

Proof. Theorem 40 implies perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness for

$$z^2 \cdot v + \delta(y, z) = \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle$$
$$= \langle \mathbf{a}_L - z\mathbf{1}, \mathbf{y}^n \circ \mathbf{a}_R + \mathbf{y}^n \circ z\mathbf{1} + z^3 \mathbf{1} + z^2 \mathbf{s} \rangle$$
$$= \langle \mathbf{a}_L, \mathbf{y}^n \circ \mathbf{a}_R \rangle + z \langle \mathbf{a}_L, \mathbf{y}^n \rangle + z^3 \langle \mathbf{a}_L, \mathbf{1} \rangle + z^2 \langle \mathbf{a}_L, \mathbf{s} \rangle - z \langle \mathbf{1}, \mathbf{y}^n \circ \mathbf{a}_R \rangle - z^2 \langle \mathbf{1}, \mathbf{y}^n \rangle$$
$$\quad - z^4 \langle \mathbf{1}, \mathbf{1} \rangle - z^3 \langle \mathbf{1}, \mathbf{s} \rangle$$
$$= \langle \mathbf{a}_L, \mathbf{y}^n \circ \mathbf{a}_R \rangle + z (\langle \mathbf{a}_L, \mathbf{y}^n \rangle - \langle \mathbf{1}, \mathbf{y}^n \circ \mathbf{a}_R \rangle) + z^2 (\langle \mathbf{a}_L, \mathbf{s} \rangle - \langle \mathbf{1}, \mathbf{y}^n \rangle)$$
$$\quad + z^3 (\langle \mathbf{a}_L, \mathbf{1} \rangle - \langle \mathbf{1}, \mathbf{s} \rangle) - z^4 n,$$

for all $y, z \in \mathbb{Z}_q^*$. This is equivalent to

$$0 = \langle \mathbf{a}_L, \mathbf{y}^n \circ \mathbf{a}_R \rangle + z (\langle \mathbf{a}_L - \mathbf{1}, \mathbf{y}^n \rangle - \langle \mathbf{1}, \mathbf{y}^n \circ \mathbf{a}_R \rangle) + z^2 (\langle \mathbf{a}_L, \mathbf{s} \rangle - v) + z^3 (\langle \mathbf{a}_L, \mathbf{1} \rangle - 1).$$

Since this holds for all $y, z \in \mathbb{Z}_q^*$, this is equivalent to

$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0} \qquad\qquad\qquad \text{// from } z^0 \text{ coefficient}$$
$$(\mathbf{a}_L - \mathbf{1}) - \mathbf{a}_R = \mathbf{0} \qquad\qquad\qquad \text{// from } z^1 \text{ coefficient}$$
$$\langle \mathbf{a}_L, \mathbf{s} \rangle = v \qquad\qquad\qquad \text{// from } z^2 \text{ coefficient}$$
$$\langle \mathbf{a}_L, \mathbf{1} \rangle = 1 \qquad\qquad\qquad \text{// from } z^3 \text{ coefficient}$$

These are exactly the linear constraints we have in (9.4) in Section 9.5.4. Hence, the existence of such $\mathbf{a}_L, \mathbf{a}_R$ is equivalent to $v \in S$ as argued there. $\qquad\square$

Instantiation for set-non-membership.

Corollary 43. *Let* $\mathbf{s} \in \mathbb{Z}_q^n$, *assume no nontrivial discrete logarithm relation between* $\mathbf{g}, \mathbf{h}, g, h$ *can be found, and instantiate the above protocol with the following functions:*

$$f_1(\mathbf{a}_L, z) := \mathbf{a}_L + z\mathbf{1},$$
$$f_2(\mathbf{a}_R, y, z) := \mathbf{a}_R - \mathbf{s},$$
$$f_3(v, y, z) := z\langle \mathbf{1}, \mathbf{y}^n \rangle \cdot v + \delta(y, z),$$
$$\delta(y, z) := \langle \mathbf{1}, \mathbf{y}^n \rangle - z\langle \mathbf{s}, \mathbf{y}^n \rangle.$$

Then the protocol above has perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness for the statement

$$\mathtt{PK}\left\{ (v, \tilde{v}) : \mathtt{V} = g^v h^{\tilde{v}} \wedge v \notin \mathbf{s} \right\}.$$

Proof. Theorem 40 implies perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness for

$$
\begin{aligned}
z\langle \mathbf{1}, \mathbf{y}^n \rangle \cdot v + \langle \mathbf{1}, \mathbf{y}^n \rangle - z\langle \mathbf{s}, \mathbf{y}^n \rangle &= \langle f_1(\mathbf{a}_L, z), \mathbf{y}^n \circ f_2(\mathbf{a}_R, y, z) \rangle \\
&= \langle \mathbf{a}_L + z\mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R - \mathbf{s}) \rangle \\
&= \langle \mathbf{a}_L \circ (\mathbf{a}_R - \mathbf{s}), \mathbf{y}^n \rangle + z\langle \mathbf{a}_R - \mathbf{s}, \mathbf{y}^n \rangle,
\end{aligned}
$$

for all $y, z \in \mathbb{Z}_q^*$. This is equivalent to

$$0 = \langle \mathbf{a}_L \circ (\mathbf{a}_R - \mathbf{s}) - \mathbf{1}, \mathbf{y}^n \rangle + z\langle \mathbf{a}_R - v\mathbf{1}, \mathbf{y}^n \rangle.$$

Since this holds for all $y, z \in \mathbb{Z}_q^*$, this is equivalent to

$$\mathbf{a}_L \circ (\mathbf{a}_R - \mathbf{s}) = \mathbf{1} \qquad \textit{// from } z^0 \textit{ coefficient}$$
$$\mathbf{a}_R = v\mathbf{1} \qquad \textit{// from } z^1 \textit{ coefficient}$$

These are exactly the linear constraints we have in (9.7) in Section 9.5.5. Hence, the existence of such $\mathbf{a}_L, \mathbf{a}_R$ is equivalent to $v \notin S$ as argued there. $\qquad \square$

9.5.7 Bulletproof Information

Similarly to Σ-protocol proof information, we introduce such information for Bulletproofs. The proof information together with the witness allows generating a Bulletproof.

Definition 44. A *Bulletproof information* $\mathtt{BPInfo} \in$ BP-INFORMATIONS consists of

- an integer $m \in \mathbb{Z}_q$,
- generators g, h of a group $\mathbb{G}$ of prime order q,
- and the public commitments $\mathtt{C}_1, \ldots, \mathtt{C}_n \in \mathbb{G}$.

Protocol genBPInfo

Inputs

$m \in \mathbb{Z}_q$
    An integer.
$(g, h) \in \mathbb{G}^2$
    Generators of the group $\mathbb{G}$.

$(C_1, \ldots, C_n) \in \mathbb{G}^n$
> Commitments in the group $\mathbb{G}$.

Outputs

$\text{BPInfo} \in \text{BP-INFORMATIONS}$
> Bulletproof information.

Protocol aggregateBPInfo

Inputs

$(\text{BPInfo}_1, \ldots, \text{BPInfo}_l) \in \text{BP-INFORMATIONS}^l$
> A list of Bulletproof information.

Outputs

$\text{BPInfo} \in \text{BP-INFORMATIONS}$
> Bulletproof information for the AND composition of $\text{BPInfo}_1, \ldots, \text{BPInfo}_l$.

Promise

> $\text{BPInfo}_1, \ldots, \text{BPInfo}_l$ all contain the same m and the same generators g and h.

Implementation

1: let m and g, h in BPInfo be the same as in $\text{BPInfo}_1, \ldots, \text{BPInfo}_l$
2: let $C = (C_{1,1}, \ldots, C_{1,n_1}, C_{2,1}, \ldots, C_{l,n_l})$ be the concatenation of all commitments in $\text{BPInfo}_1, \ldots, \text{BPInfo}_l$
3: **return** $\text{BPInfo} = (m, g, h, C)$

ⓘ The witness, i.e., values and commit randomness, for aggregated proofs is also a concatenation of the ones for the single proofs.

9.5.8 Generating Non-Interactive Proofs

All the protocols in Section 9.5 are described as multi-round interactive protocols. Since the verifier is public-coin, we can convert the protocols into non-interactive ones using the Fiat-Shamir transform as explained in Section 9.3. In this case, all the verifier's random challenges are replaced by hashes of the transcript up to that point (including the public statement). As shown in [AFK21], the Fiat-Shamir transformation of a non-constant round protocol results in a sound non-interactive proof system as long as the underlying interactive protocol is sound.

ⓘ More information on the security of non-interactive Bulletproofs can be found in [Gan+24]. In particular, the article shows that Fiat-Shamir Bulletproofs are non-malleable. This roughly means that observing proofs does not make it easier to forge one.

Protocol genBulletproof

Inputs

ctx $\in \{0,1\}^*$
        A context-string.

BPInfo $\in$ BP-INFORMATIONS
        A Bulletproof information.

$(x_1, \ldots, x_n, r_1, \ldots, r_n) \in \mathbb{Z}_q^{2n}$
        The corresponding witness, i.e., values and randomness values.

Outputs

bp $\in$ BULLETPROOFS
        A (non-interactive) Bulletproof.

Promise

        BPInfo contains n commitments.

Protocol verifyBulletproof

Inputs

ctx $\in \{0,1\}^*$
        A context-string.

BPInfo $\in$ BP-INFORMATIONS
        A Bulletproof information.

bp $\in$ BULLETPROOFS
        A (non-interactive) Bulletproof.

Outputs

$b \in$ BOOL
        Returns true iff the proof is valid in this context.

9.5.9 Combining Bulletproofs and Σ-Protocols

We will typically not just use a standalone Bulletproof, but combine it with a Σ-protocol. The following protocols can be used to generate and verify such combined proofs.

9.5.10 Generating Non-Interactive Proofs

Protocol genSigmaANDBulletproof

Inputs

ctx $\in \{0,1\}^*$
        A context-string.

$\varsigma \in \Sigma$-INFORMATIONS
        A Σ-protocol proof information.

$(\text{BPInfo}_1, \ldots, \text{BPInfo}_l) \in$ BP-INFORMATIONS^l
        A list of Bulletproof information.

$x_\Sigma \in \Sigma\text{-Witnesses}$
> The witness for the Σ-protocol.

$(x_1, \ldots, x_n, r_1, \ldots, r_n) \in \mathbb{Z}_q^{2n}$
> The witness, i.e., secret inputs and randomness values for the Bulletproofs.

Outputs

$\Sigma\text{-bp} \in \Sigma\text{-Bulletproofs}$
> A non-interactive zero-knowledge proof for the Σ-protocol and the AND combination of all Bulletproofs.

Promise
> $\mathtt{BPInfo}_1, \ldots, \mathtt{BPInfo}_l$ together contain n commitments in total.

ⓘ The protocol takes l Bulletproofs but only a single Σ-protocol. Several Σ-protocols can be combined into one using genAndComp. Bulletproofs can only be aggregated using aggregateBPInfo if they are proofs for the same generators (and should be if possible). We therefore allow the protocol to take several Bulletproofs for different generators and combine them at this stage.

Protocol verifySigmaANDBulletproof

Inputs

$\mathtt{ctx} \in \{0,1\}^*$
> A context-string.

$\varsigma \in \Sigma\text{-Informations}$
> A Σ-protocol proof information.

$(\mathtt{BPInfo}_1, \ldots, \mathtt{BPInfo}_l) \in \text{BP-Informations}^l$
> A list of Bulletproof information.

$\Sigma\text{-bp} \in \Sigma\text{-Bulletproofs}$
> A non-interactive zero-knowledge proof.

Outputs

$b \in \text{bool}$
> Returns true iff the proof is valid in this context.

9.6 Combined Protocols

In this section, we describe zero-knowledge proofs that are obtained by combining proofs described in the above sections.

9.6.1 Generic OR

We describe the *OR-composition* technique that allows a prover to prove that given two input statements y_0, y_1, it knows a witness x such that "either $(y_0, x) \in \mathcal{R}_0$ or $(y_1, x) \in \mathcal{R}_1$".

Let Π_0 and Π_1 be two multi-round interactive proof systems for NP relations $\mathcal{R}_0$ and $\mathcal{R}_1$, respectively. We assume the prover has a witness x and wants to prove that $(y_b, x) \in \mathcal{R}_b$ without

revealing $b \in \{0, 1\}$. In the OR-composition technique, the prover completes two instances of Π_0 and Π_1 with respect to y_0 and y_1, so that the proof for "$(y_b, x) \in \mathcal{R}_b$" part is real, whereas the proof for "$y_{\bar{b}} \in \mathcal{R}_{\bar{b}}$" part is fake generated by using the simulator of $\Pi_{\bar{b}}$. More accurately, the prover in the OR-composition technique proceeds as follows:

- The prover runs the simulator of $\Pi_{\bar{b}}$ on input $y_{\bar{b}}$ and obtains a simulated proof.

- The prover calls the prover of Π_b to compute the first round message of the real proof using x.

- The prover returns the first round messages of both proofs.

- After receiving the first challenge c_1 from the verifier, the prover "decomposes" it as $c_1 = c_{b,1} \oplus c_{\bar{b},1}$, where $c_{\bar{b},1}$ is the first challenge defined in the simulated proof.

- The prover uses $c_{b,1}$ to compute the next round message of Π_b.

- The prover continues this "challenge decomposition" for every round until it finishes the two parallel executions.

While this approach works well for three-round protocols, such as Σ-protocols (cf. Section 9.1), it runs into the following issue in the general case of k-round protocols for $k > 3$: remember that the prover (who knows a witness for y_b) is simulating the proof $\pi_{\bar{b}}$ for the $y_{\bar{b}}$ part of the statement, and $\pi_{\bar{b}}$ contains the (simulated) verifier's challenges, say $c_{\bar{b},1}, c_{\bar{b},2}, c_{\bar{b},3}, \ldots$. The problem now is that the prover has the opportunity to decompose the verifier's challenges $c_1, c_2, c_3, \ldots$ arbitrarily. In fact, there is no mechanism that enforces a malicious prover to commit to the simulated challenges, and avoid it to replace them with some maliciously chosen ones. Having such freedom for the prover in decomposing challenges yields an unsound insecure protocol.

The issue can be resolved by forcing the prover to commit to its decomposition in advance. That is, the prover should generate a commitment C to the simulated challenges $c_{\bar{b},1}, c_{\bar{b},2}, c_{\bar{b},3}, \ldots$ at the beginning of the protocol, and then at the end of the execution generate a witness-indistinguishable proof for the statement "C is a commitment to either $\{c_{b,1}, c_{b,2}, c_{b,3}, \ldots\}$ or $\left\{ c_{\bar{b},1}, c_{\bar{b},2}, c_{\bar{b},3}, \ldots \right\}$".

Protocol. We now describe the generic OR adapter. For $\ell = 0, 1$, let $\Pi_\ell = (\mathcal{P}_\ell, \mathcal{V}_\ell, \mathcal{S}_\ell)$ be a k_ℓ-round interactive proof system for a relation $\mathcal{R}_\ell$. Without loss of generality, we assume $k_0 \geq k_1$. A group $\mathbb{G}$ of order q, and generators $(g_1, \ldots, g_{k_0}, h) \in \mathbb{G}^{k_0+1}$ are implicit public values. We denote by $a_{b,j} \leftarrow \mathcal{P}_b(y_b, x, a_{b,0}, c_{b,1}, a_{b,1}, \ldots, a_{b,j-1}, c_{b,j})$ the output of $\mathcal{P}_b$ in round $j \geq 1$ on input $(y_b, x, a_{b,0}, c_{b,1}, a_{b,1}, \ldots, a_{b,j-1}, c_{b,j})$[7]. The prover has a witness x for a statement y_b and wants to prove that $(y_0, x) \in \mathcal{R}_0$, or $(y_1, x) \in \mathcal{R}_1$, for some predefined statement $y_{\bar{b}}$.

Protocol ORAdapter: Prover for PK $\{x : (y_0, x) \in \mathcal{R}_0 \vee (y_1, x) \in \mathcal{R}_1\}$

Initial step.

---

[7]The first message of the prover is denoted by $a_{b,0} \leftarrow \mathcal{P}_b(y_b, x)$.

Run $\mathcal{S}_{\bar{b}}$ to compute $(a_{\bar{b},0}, c_{\bar{b},1}, a_{\bar{b},1}, c_{\bar{b},2}, \ldots, c_{\bar{b},k_{\bar{b}}}, a_{\bar{b},k_{\bar{b}}}) \leftarrow \mathcal{S}_{\bar{b}}(y_{\bar{b}})$.

Add dummy challenges if necessary.
If $b = 0$, sample dummy challenges $(c_{1,k_1+1}, \ldots, c_{1,k_0}) \leftarrow \mathbb{Z}_q^{k_0-k_1}$.

Commit to the simulated challenges.
Compute $\mathtt{C} := (\prod_{j=1}^{k_0} g_j^{c_{\bar{b},j}}) h^{\tilde{r}}$ for uniformly random $\tilde{r} \leftarrow \mathbb{Z}_q$.

Define dummy responses.
For $j \in [k_1 + 1, k_0]$, set $a_{1,j} := \perp$.

Compute the first prover's message of the real proof.
Compute $a_{b,0} \leftarrow \mathcal{P}_b(y_b, x)$.

$\xrightarrow{\mathtt{C}, a_{0,0}, a_{1,0}}$

Challenge-Response step. For $j := 1 \ldots, k_0$:

$\xleftarrow{\quad c_j \quad}$

Split the challenge.
Compute $c_{b,j} = c_j \oplus c_{\bar{b},j}$.

Compute $(j+1)$-th prover's message.
Run the prover of Π_b on input $(y_b, x, a_{b,0}, c_{b,1}, a_{b,1}, \ldots, a_{b,j-1}, c_{b,j})$
and obtain $a_{b,j} \leftarrow \mathcal{P}_b(y_b, x, a_{b,0}, c_{b,1}, a_{b,1}, \ldots, a_{b,j-1}, c_{b,j})$.

$\xrightarrow{c_{0,j}, a_{0,j}, a_{1,j}}$

Last step: proving well-formedness of the commitment by executing a Σ-protocol for
$\mathrm{PK}\left\{\tilde{r} : \mathtt{C} = (\prod_{j=1}^{k_0} g_j^{c_{b,j}}) h^{\tilde{r}} \vee \mathtt{C} = (\prod_{j=1}^{k_0} g_j^{c_{\bar{b},j}}) h^{\tilde{r}}\right\}$.

Simulate the proof for the "$\mathtt{C} = (\prod_{j=1}^{k_0} g_j^{c_{b,j}}) h^{\tilde{r}}$" part.
Select uniformly random $c_{b,\Sigma}, z_{b,\Sigma} \leftarrow \mathbb{Z}_q$.
Compute $a_{b,\Sigma} = h^{z_{b,\Sigma}} \mathtt{C}^{-c_{b,\Sigma}} (\prod_{j=1}^{k_0} g_j^{c_{b,j}})^{c_{b,\Sigma}}$.

Compute the first message for the "$\mathtt{C} = (\prod_{j=1}^{k_0} g_j^{c_{\bar{b},j}}) h^{\tilde{r}}$" part.
Compute $a_{\bar{b},\Sigma} = h^{\alpha}$ for uniformly random $\alpha \leftarrow \mathbb{Z}_q$.

$\xrightarrow{a_{0,\Sigma}, a_{1,\Sigma}}$

$\xleftarrow{\quad c_\Sigma \quad}$

Split the challenge.
Compute $c_{\bar{b},\Sigma} = c_\Sigma \oplus c_{b,\Sigma}$.

Compute the response.
Compute $z_{\bar{b},\Sigma} = \alpha + c_{\bar{b},\Sigma} \tilde{r}$.

$\xrightarrow{c_{0,\Sigma}, z_{0,\Sigma}, z_{1,\Sigma}}$

Protocol ORAdapter: Verifier for $\mathrm{PK}\{x : (y_0, x) \in \mathcal{R}_0 \vee (y_1, x) \in \mathcal{R}_1\}$

Initial step.

$\xrightarrow{\mathtt{C}, a_{0,0}, a_{1,0}}$

Challenge-Response step.
For $j := 1 \ldots, k_0$:

$$\xleftarrow{\quad c_j \quad} \qquad \text{Choose uniformly random } c_j \leftarrow \mathbb{Z}_q$$

$$\xrightarrow{\quad c_{0,j}, a_{0,j}, a_{1,j} \quad}$$

Last step: proving well-formedness of the commitment by executing a Σ-protocol for $\mathrm{PK}\left\{ \tilde{r} : \mathtt{C} = (\prod_{j=1}^{k_0} g_j^{c_{b,j}})h^{\tilde{r}} \vee \mathtt{C} = (\prod_{j=1}^{k_0} g_j^{c_{\bar{b},j}})h^{\tilde{r}} \right\}$.

$$\xrightarrow{\quad a_{0,\Sigma}, a_{1,\Sigma} \quad}$$

$$\xleftarrow{\quad c_\Sigma \quad} \qquad \text{Choose uniformly random } c_\Sigma \leftarrow \mathbb{Z}_q.$$

$$\xrightarrow{\quad c_{0,\Sigma}, z_{0,\Sigma}, z_{1,\Sigma} \quad}$$

Compute the challenges of Π_1.
Compute $c_{1,j} = c_{0,j} \oplus c_j$ for $1 \leq j \leq k_0$, and $c_{1,\Sigma} = c_{0,\Sigma} \oplus c_\Sigma$.

Verify the two real/fake proofs.
Execute the verifier $\mathcal{V}_0$ on transcript $(a_{0,0}, c_{0,1}, a_{0,1}, \ldots, c_{0,k_0}, a_{0,k_0})$.
Execute the verifier $\mathcal{V}_1$ on transcript $(a_{1,0}, c_{1,1}, a_{1,1}, \ldots, c_{1,k_1}, a_{1,k_1})$.

Verify the two Sigma protocol proofs $(a_{0,\Sigma}, c_{0,\Sigma}, z_{0,\Sigma})$ and $(a_{1,\Sigma}, c_{1,\Sigma}, z_{1,\Sigma})$.
Verify $h^{z_{0,\Sigma}} \stackrel{?}{=} a_{0,\Sigma}(\frac{\mathtt{C}}{\prod_{j=1}^{k_0} g_j^{c_{0,j}}})^{c_{0,\Sigma}}$, and $h^{z_{1,\Sigma}} \stackrel{?}{=} a_{1,\Sigma}(\frac{\mathtt{C}}{\prod_{j=1}^{k_0} g_j^{c_{1,j}}})^{c_{1,\Sigma}}$.

⚠ The generators $g_1, \ldots, g_{k_0}, h$ must be generated such that no one knows the pairwise discrete logs.

❗ We assume that both involved proof systems Π_0 and Π_1 are special honest verifier zero-knowledge. This is needed as it guarantees the existence of the efficient simulator in the protocol.

ℹ The protocol described above is for the OR composition of only two statements. In the case of having more statements, we can produce a proof for 1-out-of-many statements by using a generalized variant of the above technique in a straightforward manner.

9.6.2 Generic AND

We describe the *AND-composition* technique that allows a prover to prove to a verifier that given two input statements y_0, y_1, it knows witnesses[8] x_0, x_1 such that $(y_\ell, x_\ell) \in \mathcal{R}_\ell$ for $\ell \in \{0, 1\}$. Let Π_0 and Π_1 be two multi-round interactive proof systems for NP relations $\mathcal{R}_0$ and $\mathcal{R}_1$, respectively. In the AND-composition technique, the prover runs Π_0 for $\mathcal{R}_0$, and Π_1 for $\mathcal{R}_1$ in parallel, using a *common* challenge.

---

[8]Note that the two witnesses here are not necessarily distinct.

> **!** We assume that both involved proof systems Π_0 and Π_1 use the same challenge space. Otherwise, an injective, deterministic mapping from e.g. bitstring challenges to the different challenge spaces must exist.

Protocol. We now describe the generic AND adapter. For $\ell = 0, 1$, let $\Pi_\ell = (\mathcal{P}_\ell, \mathcal{V}_\ell, \mathcal{S}_\ell)$ be a k_ℓ-round interactive proof system for a relation $\mathcal{R}_\ell$. Without loss of generality, we assume $k_0 \geq k_1$. We denote by $a_{\ell,j} \leftarrow \mathcal{P}_\ell(y_\ell, x_\ell, a_{\ell,0}, c_{\ell,1}, a_{\ell,1}, \ldots, a_{\ell,j-1}, c_{\ell,j})$ the output of $\mathcal{P}_\ell$ in round $j \geq 1$ on input $(y_\ell, x_\ell, a_{\ell,0}, c_{\ell,1}, a_{\ell,1}, \ldots, a_{\ell,j-1}, c_{\ell,j})$[9]. The prover has witnesses x_0, x_1 respectively for statements y_0, y_1 and wants to prove that $(y_\ell, x_\ell) \in \mathcal{R}_\ell$, for $\ell \in \{0, 1\}$.

Protocol ANDAdapter: Prover for PK $\{(x_0, x_1) : (y_0, x_0) \in \mathcal{R}_0 \wedge (y_1, x_1) \in \mathcal{R}_1\}$

Initial step.

Compute the first prover's messages for both proofs.
Compute $a_{\ell,0} \leftarrow \mathcal{P}_\ell(y_\ell, x_\ell)$, for $\ell \in \{0, 1\}$.

$\xrightarrow{\quad a_{0,0}, a_{1,0} \quad}$

Challenge-Response step
For $j := 1 \ldots, k_0$:

$\xleftarrow{\quad c_j \quad}$

Compute $(j + 1)$-th prover's messages for both proofs.
Obtain $a_{0,j} \leftarrow \mathcal{P}_0(y_0, x_0, a_{0,0}, c_1, a_{0,1}, \ldots, a_{0,j-1}, c_j)$.
If $j \leq k_1$, obtain $a_{1,j} \leftarrow \mathcal{P}_1(y_1, x_1, a_{1,0}, c_1, a_{1,1}, \ldots, a_{1,j-1}, c_j)$; Else, $a_{1,j} := \bot$.

$\xrightarrow{\quad a_{0,j}, a_{1,j} \quad}$

Protocol ANDAdapter: Verifier for PK $\{(x_0, x_1) : (y_0, x_0) \in \mathcal{R}_0 \wedge (y_1, x_1) \in \mathcal{R}_1\}$

Initial step.

$\xrightarrow{\quad a_{0,0}, a_{1,0} \quad}$

Challenge-Response step.
For $j := 1 \ldots, k_0$:

$\xleftarrow{\quad c_j \quad}$ Choose uniformly random $c_j \leftarrow \mathbb{Z}_q$

Verify the two proofs.
Execute the verifier $\mathcal{V}_0$ on transcript $(a_{0,0}, c_1, a_{0,1}, \ldots, c_{k_0}, a_{0,k_0})$.
$\xrightarrow{\quad a_{0,j}, a_{1,j} \quad}$ Execute the verifier $\mathcal{V}_1$ on transcript $(a_{1,0}, c_1, a_{1,1}, \ldots, c_{k_1}, a_{1,k_1})$.

---

[9]The first message of the prover is denoted by $a_{\ell,0} \leftarrow \mathcal{P}_\ell(y_\ell, x_\ell)$.

> ⓘ The generic AND-composition described above is called parallel AND-composition as the two proof systems are composed in parallel, i.e., round by round. This is shown to be (knowledge) sound and special honest verifier ZK as long as the underlying proof systems have these properties. Note that the soundness error will be the product of the individual soundness errors [AF21].

9.6.3 Proof of Correct ElGamal Encryption

We show how to prove that a value was encrypted properly in the exponent (without splitting) using ElGamal encryption.

Protocol genEncExpNoSplitInfo

Inputs

$(g, \bar{h}) \in \mathbb{G}^2$
  A Pedersen commitment key.

$\mathrm{pk}_{\mathrm{EG}} \in \textsc{PublicKeys}_{\mathrm{EG}} := \mathbb{G}$
  A public key, $\mathrm{pk}_{\mathrm{EG}} = g^{\mathrm{dk}_{\mathrm{EG}}}$.

$q \in \mathbb{N}$
  The order of the group $\mathbb{G}$.

$c = (c_1, c_2) \in \mathbb{G}^2$
  A ciphertext generated with $\mathsf{Enc}_{\mathrm{EG}}(\mathrm{pk}_{\mathrm{EG}}, \cdot)$.

Outputs

$\varsigma \in \Sigma\text{-}\textsc{Informations}$
  A Σ-protocol proof information for

$$\mathrm{PK}\left\{(r, x) : c_1 = g^r \wedge c_2 = \bar{h}^x \cdot \mathrm{pk}_{\mathrm{EG}}^r\right\}.$$

Implementation

1: $\varsigma_1 := \mathsf{genSigProofInfo}(\mathrm{dlog}, (g, q), c_1)$ // $\mathrm{PK}\left\{(r) : c_1 = g^r\right\}$
2: $\varsigma_2 := \mathsf{genSigProofInfo}(\mathrm{com}, (\bar{h}, \mathrm{pk}_{\mathrm{EG}}, q), c_2)$ // $\mathrm{PK}\left\{(x, r') : c_2 = \bar{h}^x \cdot \mathrm{pk}_{\mathrm{EG}}^{r'}\right\}$
3: $\varsigma' := \mathsf{genAndComp}(\varsigma_1, \varsigma_2)$ // $\mathrm{PK}\left\{(r, x, r') : c_1 = g^r \wedge c_2 = \bar{h}^x \cdot \mathrm{pk}_{\mathrm{EG}}^{r'}\right\}$
4: $\varsigma := \mathsf{genEqComp}(\varsigma', \{1, 3\})$ // *as above, plus* $r = r'$
5: **return** ς

9.6.4 Proof of Correct Encryption in Exponent

In Section 7.1.4, we have described how to encode a number in the exponent of a group element and encrypt it using ElGamal encryption. We here show how to prove that a value was encrypted properly using that method. This boils down to proving correct encryption of each part and showing that all encrypted parts are sufficiently small.

Protocol genEncExpInfo

Inputs

$(g, \bar{h}) \in \mathbb{G}^2$
    A Pedersen commitment key.

$\mathrm{pk}_{\mathrm{EG}} \in \textsc{PublicKeys}_{\mathrm{EG}} \coloneqq \mathbb{G}$
    A public key, $\mathrm{pk}_{\mathrm{EG}} = g^{\mathrm{dk}_{\mathrm{EG}}}$.

$q \in \mathbb{N}$
    The order of the group $\mathbb{G}$.

$\mathbf{s} \in \mathbb{N}$
    The size (in bits) of each encrypted part.

$c = ((c_{1,1}, c_{1,2}), \ldots, (c_{\mathsf{t},1}, c_{\mathsf{t},2})) \in \mathbb{G}^{2\mathsf{t}}$
    A ciphertext generated with $\mathsf{Enc}_{\mathrm{EG}}^{\mathsf{t},\mathsf{s}}(\mathrm{pk}_{\mathrm{EG}}, \cdot)$.

Outputs

$\varsigma \in \Sigma\text{-}\textsc{Informations}$
    A Σ-protocol proof information.

$\mathtt{BPInfo} \in \mathrm{BP\text{-}Informations}$
    A bulletproof information.

Implementation

1:   /* Prove $\mathrm{PK}\left\{(r_i, x_i) : c_{i,1} = g^{r_i} \wedge c_{i,2} = \bar{h}^{x_i} \cdot \mathrm{pk}_{\mathrm{EG}}^{r_i}\right\}$ for all i */
2: **for** $i = 1, \ldots, \mathsf{t}$ **do**
3:     $\varsigma_i \coloneqq \mathsf{genEncExpNoSplitInfo}((g, \bar{h}), \mathrm{pk}_{\mathrm{EG}}, q, (c_{i,1}, c_{i,2}))$
4:   $\varsigma \coloneqq \mathsf{genAndComp}(\varsigma_1, \ldots, \varsigma_{\mathsf{t}})$
5:   /* $\mathrm{PK}\left\{x_1', \ldots, x_{\mathsf{t}}', r_1'', \ldots, r_{\mathsf{t}}'' : \forall i \in \{1, \ldots, \mathsf{t}\} \;\; (c_{i,2} = \bar{h}^{x_i'} \cdot \mathrm{pk}_{\mathrm{EG}}^{r_i''} \;\wedge\; x_i' \in [0, 2^{\mathsf{s}} - 1])\right\}$ */
6:   $\mathtt{BPInfo} \coloneqq \mathsf{genBPInfo}(\mathsf{s}, (\bar{h}, \mathrm{pk}_{\mathrm{EG}}), (c_{1,2}, \ldots, c_{\mathsf{t},2}))$
7:   **return** $(\varsigma, \mathtt{BPInfo})$

9.6.5 Proof of ElGamal Decryption With Public Message

Let $\mathbb{G}$ be a group of prime order q with generator g, and let $(c_1, c_2) = (g^r, m \cdot \mathrm{pk}_{\mathrm{EG}}^r)$ be an ElGamal encryption of m under public key $\mathrm{pk}_{\mathrm{EG}} = g^{\mathrm{dk}_{\mathrm{EG}}}$ (see Section 7.1). The prover wants to show knowledge of $\mathrm{dk}_{\mathrm{EG}}$ such that (c_1, c_2) decrypts to m under $\mathrm{dk}_{\mathrm{EG}}$. For this proof, the public key $\mathrm{pk}_{\mathrm{EG}}$ and the message m are considered to be publicly known. For a protocol that does not reveal the encrypted value, see Section 9.2.8.

The prover shows knowledge of the secret key matching the public key (which corresponds to knowledge of a discrete logarithm) and that this secret key is the discrete logarithm of $m \cdot c_2^{-1}$ to the base c_1^{-1}. That is, $c_1^{-\mathrm{dk}_{\mathrm{EG}}} = m \cdot c_2^{-1}$. Hence, this shows that decrypting (c_1, c_2) with the secret key matching the prover's public key yields m.

> ℹ The same method can be used if the message is split into smaller parts and encrypted in the exponent, by just proving decryption of each part separately. Since the resulting message is public, all parts can be revealed as well, and one can publicly compute the linear combination to obtain the combined value.

Protocol genPubDecInfo

Inputs

$g \in \mathbb{G}$
> The generator of the group used for key generation.

$\mathrm{pk_{EG}} \in \textsc{publickeys}_{EG} := \mathbb{G}$
> A public key, $\mathrm{pk_{EG}} = g^{\mathrm{dk_{EG}}}$.

$q \in \mathbb{N}$
> The order of the group $\mathbb{G}$.

$m \in \mathbb{G}$
> The encrypted message.

$(c_1, c_2) \in \mathbb{G}^2$
> An encryption of m.

Outputs

$\varsigma \in \Sigma\text{-}\textsc{Informations}$
> A Σ-protocol proof information for
> $$\mathrm{PK}\left\{\mathrm{dk_{EG}} : \mathrm{pk_{EG}} = g^{\mathrm{dk_{EG}}} \ \wedge \ m = c_2 \cdot c_1^{-\mathrm{dk_{EG}}}\right\}.$$

Implementation

1: $\varsigma_1 := \mathsf{genSigProofInfo}(\mathrm{dlog}, (g, q), \mathrm{pk_{EG}})$ // $\mathrm{PK}\left\{\mathrm{dk_{EG}} : \mathrm{pk_{EG}} = g^{\mathrm{dk_{EG}}}\right\}$
2: $\varsigma_2 := \mathsf{genSigProofInfo}(\mathrm{dlog}, (c_1^{-1}, q), m \cdot c_2^{-1})$ // $\mathrm{PK}\left\{\mathrm{dk'_{EG}} : m \cdot c_2^{-1} = c_1^{-\mathrm{dk'_{EG}}}\right\}$
3: $\varsigma' := \mathsf{genAndComp}(\varsigma_1, \varsigma_2)$ // $\mathrm{PK}\left\{(\mathrm{dk_{EG}}, \mathrm{dk'_{EG}}) : \mathrm{pk_{EG}} = g^{\mathrm{dk_{EG}}} \ \wedge \ m \cdot c_2^{-1} = c_1^{-\mathrm{dk'_{EG}}}\right\}$
4: $\varsigma := \mathsf{genEqComp}(\varsigma', \{1, 2\})$ // *as in ς', plus* $\mathrm{dk_{EG}} = \mathrm{dk'_{EG}}$
5: **return** ς

9.7 Adapting Proofs to Vector Pedersen Commitments

In Section 9.5, we described protocols for proving (in)equality, set-(non-)membership, and ranges. In this section, we show how to adapt these protocols to use vector Pedersen commitments (cf. Section 8.1.4).

9.7.1 Proof of Equality for Vector Pedersen Commitment and Pedersen Commitments

Let $(\mathbb{G}, \cdot)$, and $(\overline{\mathbb{G}}, \cdot)$ be groups of prime order q. Let $g_1, \ldots, g_n, h$ be generators of $\mathbb{G}$ and let $\bar{g}$ and $\bar{h}$ be generators of $\overline{\mathbb{G}}$.

Let $I \subseteq \{1, \ldots, n\}$ be an index set of size n_I. The prover has produced a vector Pedersen commitment $\mathtt{C} = h^r \prod_{i=1}^{n} g_i^{x_i}$ and commitments $\left\{\mathtt{C}_i = \bar{g}^{x_i} \bar{h}^{r_i}\right\}_{i \in I}$. The prover wants to show that the same $\{x_i\}_{i \in I}$ appear in $\mathtt{C}$ and in the commitments. We do this by showing that the prover actually knows such values.

Instantiation of abstract protocol

We set $G := (\mathbb{Z}_q, +)^{n+n_I+1}$ and $H := (\mathbb{G}, \cdot) \times (\overline{\mathbb{G}}, \cdot)^{n_I}$ (with component-wise group operations). The homomorphism φ is defined as

$$\varphi(x_1, \ldots, x_n, r, \{r_i\}_{i \in I}) := \left(h^r \prod_{i=1}^n g_i^{x_i}, \left\{ \bar{g}^{x_i} \bar{h}^{r_i} \right\}_{i \in I} \right).$$

This is a homomorphism since

$$\varphi((x_1, \ldots, x_n, r, \{r_i\}_{i \in I}) + (y_1, \ldots, y_n, s, \{s_i\}_{i \in I}))$$

$$= \left(h^{r+s} \prod_{i=1}^n g_i^{x_i+y_i}, \left\{ \bar{g}^{x_i+y_i} \bar{h}^{r_i+s_i} \right\}_{i \in I} \right)$$

$$= \left(h^r \prod_{i=1}^n g_i^{x_i}, \left\{ \bar{g}^{x_i} \bar{h}^{r_i} \right\}_{i \in I} \right) \cdot \left(h^s \prod_{i=1}^n g_i^{y_i}, \left\{ \bar{g}^{y_i} \bar{h}^{s_i} \right\}_{i \in I} \right)$$

$$= \varphi(x_1, \ldots, x_n, r, \{r_i\}_{i \in I}) \cdot \varphi(y_1, \ldots, y_n, s, \{s_i\}_{i \in I}).$$

The challenge space $\mathcal{C} \subseteq \mathbb{Z}_q$ is any large subset of $\mathbb{Z}_q$.

The conditions for Theorem 23 are satisfied for $k = q$ and $u = (0, \ldots, 0) \in G$: Since q is prime, $\gcd(c_1 - c_2, q) = 1$ for all $c_1 \neq c_2$, and $\varphi(0, \ldots, 0) = (1, \ldots, 1) = (\mathsf{C}, \{\mathsf{C}_i\}_{i \in I})^q$.

Full protocol

Protocol Vcom-com-eq: PK $\left\{ (x_1, \ldots, x_n, r, \{r_i\}_{i \in I}) : \mathsf{C} = h^r \prod_{i=1}^n g_i^{x_i} \wedge \left\{ \mathsf{C}_i = \bar{g}^{x_i} \bar{h}^{r_i} \right\}_{i \in I} \right\}$

Implicit public values: $\mathbb{G}, \overline{\mathbb{G}}, q, g_1, \ldots, g_n, h, \bar{g}, \bar{h}, I$

Prover$(x_1, \ldots, x_n, r, \{r_i\}_{i \in I})$	**Verifier**$(\mathsf{C}, \{\mathsf{C}_i\}_{i \in I})$

Choose random
$\alpha_1 \ldots, \alpha_n, \tilde{r}, \tilde{r}_i \leftarrow \mathbb{Z}_q$ for
$i \in I$, and compute
$a = h^{\tilde{r}} \prod_{i=1}^n g_i^{\alpha_i}$ and
$a_i = \bar{g}^{\alpha_i} \bar{h}^{\tilde{r}_i}$ for $i \in I$

$$\xrightarrow{\quad (a, \{a_i\}_{i \in I}) \quad}$$

Choose random
$c \leftarrow \mathcal{C} \subseteq \mathbb{Z}_q$

$$\xleftarrow{\quad c \quad}$$

Compute $s_i = \alpha_i - cx_i$
$(\bmod\ q)$ for $i \in \{1, \ldots, n\}$,
$t = \tilde{r} - cr$, $t_i = \tilde{r}_i - cr_i$
$(\bmod\ q)$ for $i \in I$

$$\xrightarrow{\quad (s_1, \ldots, s_n, t, \{t_i\}_{i \in I}) \quad}$$

Accept iff
$a = h^t \prod_{i=1}^n g_i^{s_i} \cdot \mathsf{C}^c$ and
$a_i = \bar{g}^{s_i} \bar{h}^{t_i} \cdot \mathsf{C}_i^c$ for $i \in I$.

9.7.2 Proof of Equality for Vector Pedersen Commitment and Pedersen Commitments with Partial Reveal of Values

Let $(\mathbb{G}, \cdot)$, and $(\overline{\mathbb{G}}, \cdot)$ be groups of prime order q. Let $g_1, \ldots, g_n, h$ be generators of $\mathbb{G}$ and let $\bar{g}$ and $\bar{h}$ be generators of $\overline{\mathbb{G}}$.

Let $I, J \subseteq \{1, \ldots, n\}$ be two disjoint index sets. Let $\bar{I} := \{1, \ldots, n\} \setminus I$ be the complement of I. The prover has produced a vector Pedersen commitment $\mathtt{C} = h^r \prod_{i=1}^n g_i^{x_i}$ and commitments $\left\{ \mathtt{C}_j = \bar{g}^{x_j} \bar{h}^{r_j} \right\}_{j \in J}$. The prover wants to show that the values $\{x_i\}_{i \in I}$ are equal to some public values (i.e., revealing the values), and that the same $\{x_j\}_{j \in J}$ appear in $\mathtt{C}$ and in the commitments. To do this, the prover and the verifier first compute a new vector Pedersen commitment $\hat{\mathtt{C}} = \mathtt{C} \cdot \prod_{i \in I} g_i^{-x_i}$ and then call Vcom-com-eq in Section 9.7.1 with statement

$$\mathsf{PK}\left\{ \left(\{x_i\}_{i \in \bar{I}}, \{r_j\}_{j \in J}, r\right) : \hat{\mathtt{C}} = h^r \prod_{i \in \bar{I}} g_i^{x_i} \wedge \mathtt{C}_j = \bar{g}^{x_j} \bar{h}^{r_j} (j \in J) \right\}.$$

9.7.3   Overall Protocol

We describe the overall protocol for proving (in)equality, set-(non-)membership, or ranges given a vector Pedersen commitment. The prover first generates single Pedersen commitments for all values used in inequality, set-(non-)membership, or range proofs. The prover then uses the protocol from Section 9.7.2 to link the new commitments with the vector Pedersen commitment and to reveal values. Finally, the prover uses the protocols for inequality (cf. Section 9.2.13), set-(non-)membership (cf. Section 9.5.4, 9.5.5), or range proofs (cf. Section 9.5.3).

> 💡 The proof size in the above Σ-protocols is linear in the number of secret committed values. Using compressed Σ-protocol techniques introduced in [AC20], one can improve the proof size to logarithmic.

9.7.4   Overall Protocol with Non-Transferability

In some applications, we may need to ensure that the interaction between the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ is *non-transferable*, i.e., only $\mathcal{V}$ who had interaction with $\mathcal{P}$ can be convinced about the truth of prover's statements, and it cannot *additionally* prove to someone else that $\mathcal{P}$ sent a proof.

A simple approach to provide non-transferability is to use the following two-round proof: $\mathcal{V}$ first sends a commitment $\mathtt{pk}$ to a secret $\mathtt{sk}$ together with a ZK proof of knowledge (PoK) of $\mathtt{sk}$. In the second round, $\mathcal{P}$ who aims to show that y belongs to some language $\mathcal{L}$ checks the verifier's proof and if valid, proves an OR statement "$y \in \mathcal{L}$ OR I know the secret of $\mathtt{pk}$".

The above OR statement consists of two parts, namely a Σ-protocol-friendly PoK of secret, and the actual statement y. In the case that y is also a Σ-protocol-friendly statement, we can use the OR-composition technique for Σ-protocols described in 9.4.4, whereas in other cases where y is proven by a different type of multi-round interactive protocol (e.g., Bulletproofs), the generic OR adapter described in 9.6.1 is required. Finally, we note that one can make the proofs of statements in Section 9.7.3 non-transferable without using a generic OR adapter by modifying the protocol in Section 9.7.3 as follows: after generating Pedersen commitments for all witness values, instead of directly using the Σ-protocol in Section 9.7.2 for linking commitments, the prover OR-composes it with the verifier's statement. The rest of the protocol where the prover uses the protocols for inequality (cf. Section 9.2.13), set-(non-)membership (cf. Section 9.5.4, 9.5.5), or range proofs (cf. Section 9.5.3) is unchanged.
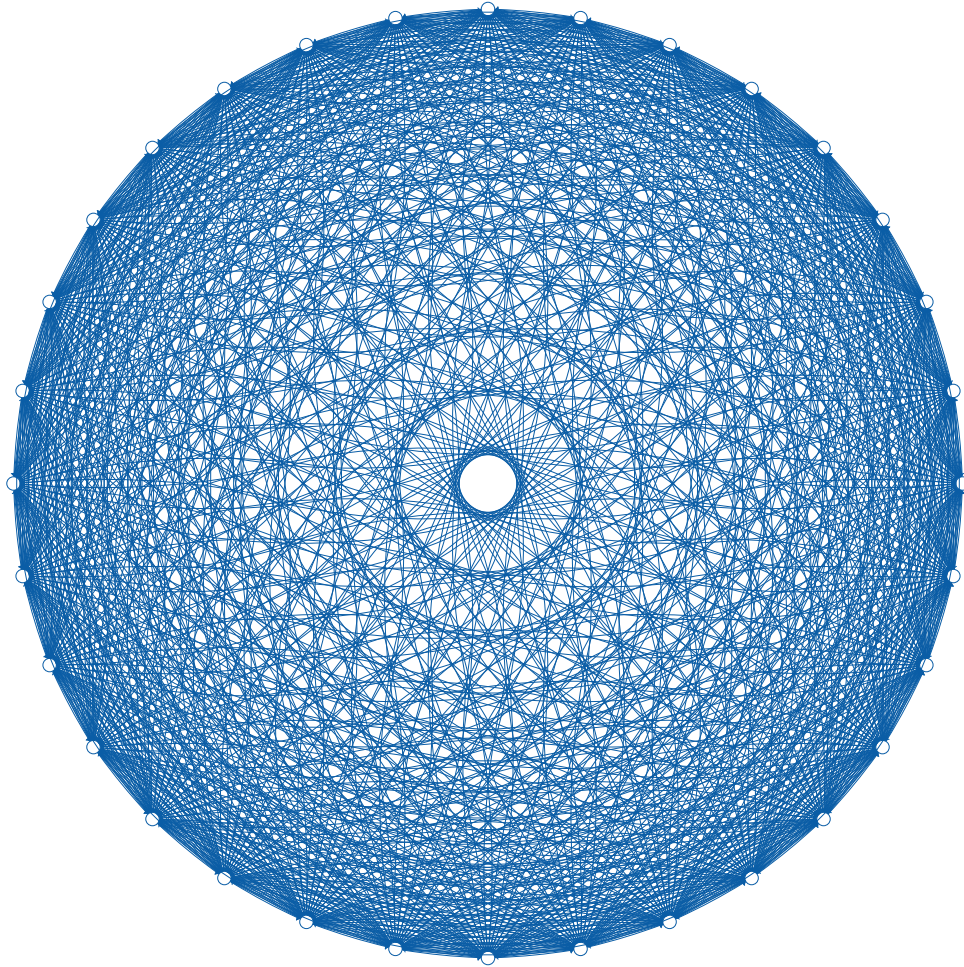
Part II

Konsensus

Illustration: Complete graph of 32 nodes.

Chapter 10

Konsensus Preliminaries

10.1 Introduction

We use a variant of HotStuff [Yin+19] based on Jolteon [Gel+21] for our consensus. Jolteon is a variant of the HotStuff consensus protocol that finalizes a block after two rounds of signing instead of three. It achieves this by including the best quorum certificate[1] seen so far in a timeout and having the next leader prove that they extend the best certificate provided in the timeouts.

This document mostly follows Jolteon, but makes the following changes.

1. The protocol is adapted to take the stake of parties into account. In particular, the signature of each party is weighted by their stake and aggregated signature schemes are used instead of threshold signatures.

2. We do not use the bound of 1/3 corruption, but instead have different thresholds of safety s and liveness ℓ such that $s + 2\ell < 1$.

3. The signature of a block is not sent to the next leader only, but to everyone.[2]

4. The timeout certificate is not sent around by parties who compute it. The next leader must compute it for themselves.

5. The timeout is extended to include the timeout certificate of the previous round, if it exists. This allows parties lagging behind to jump to the round where others are timing out.

6. We redefined the timeout certificate so that it does not consist of all timeouts received (which has space complexity $\mathcal{O}(n^2)$), but only the minimal information needed to verify that the best certificate is selected (which has space complexity $\mathcal{O}(n)$).

7. Epoch and epoch transitions are added to the protocol. Stake can be updated at best every epoch from the transaction of 2 epochs ago (though we update it only once every 24h). And proofs of epoch transition are included in the first block of an epoch and some timeouts.

8. A leader election scheme is given explicitly.

---

[1]A quorum certificate is an aggregated signature on a block that has sufficient weight—see Definition 50. A timeout certificate is the same but for timeouts.

[2]At first glance this might look like $\mathcal{O}(n^2)$ communication complexity instead of $\mathcal{O}(n)$. But firstly, the Concordium network does not have direct communication between parties; flooding the network with the message is the only option, so this change does not increase the communication complexity. Secondly, if everyone has the certificate, then this block can be saved in case the next leader times out. And thirdly, this allows for a fully unpredictable leader election—though the leader election we consider is not fully unpredictable, as it computes the set of leaders for the next hour every hour.

10.2 Participants

We work in the permissionless setting where the number of *parties* is arbitrary. Any parties can access the data stored in Konsensus. The consensus on the stored data is maintained by special parties, called validators. Validators also ensure that new data is properly added. More information on validators is found below in Section 10.2.1.

Definition 45. A party is called *live* if it is connected to the network, up-to-date on the Konsensus data (in the form of a block tree), and awake (that is participates in the protocol). A party is called *honest* if it follows the protocol instructions. A party is called *parat* if it is live and honest.

10.2.1 Validators

The consensus protocol is run by parties called *validators*. A validator is identified by a unique id and has a public-lottery power assigned.

Definition 46. A *validator*, `validator` $\in$ VALIDATORS, is an entity and has the following values assigned.

The following values are publicly known and stored on the blockchain.

Public Values

    `vid` $\in$ VALIDATORIDS
        The unique *validator identifier*.

    `lotteryPower` $\in$ LOTTERYPOWER
        The *lottery power*.

    $\text{vk}^{\text{LE}} \in$ VERIFICATIONKEY$_{\text{LE}}$ = PUBLICKEY$_{\text{VRF}}$
        The *leader election verification key*, a VRF verification key, cf. Section 6.2.

    $\text{vk}^{\text{BC}} \in$ VERIFICATIONKEYS$_{\text{BC}}$
        The *block signature verification key*, cf. Section 5.1.3.

The following values are only known to the validator itself.

Private Values

    $\text{sk}^{\text{LE}} \in$ PRIVATEKEY$_{\text{LE}}$ = PRIVATEKEY$_{\text{VRF}}$
        The *leader election private key*, a VRF secret key, cf. Section 6.2.

    $\text{sk}^{\text{BC}} \in$ SIGNKEYS$_{\text{BC}}$
        The *block signature signing key*, cf. Section 5.1.3.

> ℹ️ The Konsensus layer only requires that validator identifiers are unique. Otherwise, the IDs can be arbitrary bit strings. In an implementation, a validator-ID could be a public-key which is assigned with an identity or an account.

> </> In the implementation, validators are identified by the index of their account.

Validators can have two roles. As *leader* they will produce a block that is then sent to the *finalizer*s who sign off on correct blocks.

> **</>** Finalizers are additionally identified by their index in the finalization committee.

10.3 Time and Synchrony

Except for assumptions on the communication network (cf. Section 10.4), the consensus protocol does not require any synchrony assumptions.

However, in a practical implementation it is useful to assume that honest parties have somewhat synchronized clocks. Thus, we assume that there is a notion of global[3] physical time t. Each party P_i has a local clock $\texttt{Clock}_i$. We assume that there is a bound $\texttt{MaxDrift}$ on clock drift. Specifically we assume that $|t - \texttt{Clock}_i| < \frac{\texttt{MaxDrift}}{2}$. This implies that the clocks of any two parat parties P_i and P_j satisfy $|\texttt{Clock}_i - \texttt{Clock}_j| < \texttt{MaxDrift}$.

> **!** The bound $\texttt{MaxDrift}$ is not known to honest parties and is *not* used in the protocol. However, the security guarantees of the implementation may depend on this bound. For example, parties may reject blocks that come from too far in the future.

10.4 Communication Network

We assume that parties have access to a diffusion network which allows them to multi-cast messages. Parties can join or leave the network at any point in time. If an honest party sends a message, any other honest party that stays connected long enough is guaranteed to receive the message.

More formally, we assume a partially synchronous network with periods of synchrony.

Definition 47 ([CPS18]). A *network* is *partially synchronous* with *periods of synchrony* if there exists a bound $\Delta_{\texttt{net}}$ such that

- for any period of synchrony $[\tau_0, \tau_1]$ and any message sent by an honest party at (local) time $t < \tau_1$ will arrive at time $\max(t_0, t + \Delta_{\texttt{net}})$ at any other honest party,

- the *network delay* bound $\Delta_{\texttt{net}}$ is known to honest parties, but honest parties do not know when periods of synchrony occur.

The above is a somewhat pessimistic model of a peer-to-peer network over the internet. Under normal conditions, i.e., in a period of synchrony, messages will arrive within a short delay $\Delta_{\texttt{net}}$. But outside those periods, the network delay can be arbitrarily long, e.g., when a submarine communication cable gets damaged by a fishing trawler. As we will see in the section below, the protocol is secure independently of the network condition and guaranteed to be live in a period of synchrony that is sufficiently long.

The assumption that all honest messages arrive eventually is somewhat ideal and will not be achieved by most implementations. We therefore need to provide the following warning.

> **⚠** For simplicity, we here assume that "relevant" messages (e.g., ones that allow parties to form a quorum certificate and advance their round) from honest parties are not ignored by other honest parties. See the liveness proofs in Section 13.3 for more details.

---

[3] We do not consider parties moving at relativistic speeds here.

More realistically is the assumption that dishonest parties may send messages to a select set of honest parties.

> ❗ Our network model does not provide any guarantees (for honest parties) on messages sent by dishonest parties. A dishonest party may send a message to a select group of honest parties.

> </> Our implementation network is a *gossip protocol* where joining nodes get random peers assigned by a bootstrapper. Peers communicate over secure channels which are established using the Noise Protocol Framework.

Chapter 11

ConcordiumBFT

This chapter describes the consensus protocol. It is organized as follows. Section 11.1 provides the data structures needed. Section 11.2 introduces the notion of epochs. Section 11.3 gives details on the leader election. Section 11.4 describes the actual consensus protocol.

11.1 Data-Structures

The Skov[1] layer maintains the data structure for the Konsensus layer, see Section 11.1.6. In particular, it contains the block tree, Skov.tree.

11.1.1 Quorum Signatures and Certificates

Here, we define the *quorum certificates* and related data structures starting with quorum signatures.

When a finalizer receives a valid block at the right place in the block tree, they will sign it and multicast the resulting quorum signature.

Definition 48. A *quorum signature*, quorumSignature $\in$ QUORUMSIGNATURETYPE, is a tuple consisting of

vid $\in$ VALIDATORIDS
  identifier of the sender.

ptr $\in$ HASH
  The *hash* of the block that is validated.

round $\in \mathbb{N}$
  The *round number* of the block that is validated.

epoch $\in \mathbb{N}$
  The *epoch number* of the block that is validated.

$\sigma^{\texttt{AggFin}} \in \text{SIGNATURES}_{\texttt{AggFin}}$
  Signature from the sender on $(\mathsf{H}(\texttt{genesis}), \texttt{ptr}, \texttt{round}, \texttt{epoch})$ using an aggregate signature scheme.

---

[1]Skov means forest in Danish.

> :warning: The genesis hash $H(\texttt{genesis})$ is as a domain separation string to differentiate between networks or protocol versions within a network.

> </> `round` and `epoch` are redundant in the sense that they can be queried via `ptr.round` and `ptr.epoch`. Including them directly in the quorum signatures simplifies some witnesses for faulty behavior (e.g., a witness for double signing in the same round is shorter because the signature is on the round as well) and simplifies some tests (e.g., we can dismiss signatures on past rounds without having to find the corresponding block), but comes at the cost of having these extra fields in the quorum signature and having to check consistency (e.g., `round = ptr.round`).

The validity of a quorum signature is defined as follows.

Definition 49. A quorum signature

$$\texttt{quorumSignature} = (\texttt{vid}, \texttt{ptr}, \texttt{round}, \texttt{epoch}, \sigma^{\texttt{AggFin}})$$

is *valid* with respect to $\mathsf{Skov.tree}$ if the following holds:

- `ptr` points to a block in a valid chain.
- Let $\texttt{finCom} \coloneqq \mathsf{Kontrol.getFinalizationCommittee}(\texttt{epoch}, \mathsf{Skov.epochFinEntries})$.
- The sender is a finalizer, namely $\texttt{vid} \in \texttt{finCom}$.
- The round and epoch numbers are consistent, i.e., $\texttt{round} = \texttt{ptr.round}$ and $\texttt{epoch} = \texttt{ptr.epoch}$.
- $\sigma^{\texttt{AggFin}}$ is a valid aggregate signature by $\texttt{vid}$ on $(H(\texttt{genesis}), \texttt{ptr}, \texttt{round}, \texttt{epoch})$.

Once enough signatures have been received, a quorum certificate (QC) can be formed, which is included in the next block. The QC structure is defined as follows.

Definition 50. A *quorum certificate*, $\texttt{qc} \in \textsc{quorumcertificates}$, is a tuple consisting of

$\texttt{ptr} \in \textsc{hash}$
  The hash of the block to which this certificate refers.

$\texttt{round} \in \mathbb{N}$
  The round number of the block to which this certificate refers.

$\texttt{epoch} \in \mathbb{N}$
  The epoch number of the block to which this certificate refers.

$\sigma^{\texttt{AggFin}} \in \textsc{signatures}_{\texttt{AggFin}}$
  An aggregate signature on $(H(\texttt{genesis}), \texttt{ptr}, \texttt{round}, \texttt{epoch})$, cf. Section 5.1.3.

$\texttt{signers} \in \textsc{validatorids}^*$
  List of finalizers whose signature is in $\sigma^{\texttt{AggFin}}$.

> round and epoch are redundant in the sense that they can be queried via `ptr.epoch` and `ptr.round`. Including them directly in the QC simplifies some witnesses for invalid messages (a signature on a block signs the QC round and epoch numbers which are the parent round and epoch numbers) and finalization proofs (the signature weights can be checked without needing access to the block), but comes at the cost of having these extra fields in the QC and having to check consistency (e.g., round = `ptr.round`).

A quorum certificate is considered valid with respect to Skov.tree if it was properly signed by the finalization committee. More formally, validity is defined as follows.

Definition 51. A quorum certificate $qc = (ptr, round, epoch, \sigma^{AggFin}, signers)$ is *valid* with respect to Skov.tree if the following holds:

- `ptr` points to a block in a valid chain.

- The round and epoch numbers are consistent, namely round = `ptr.round` and epoch = `ptr.epoch`.

- Let finCom := Kontrol.getFinalizationCommittee(epoch, Skov.epochFinEntries).

- Let sigThreshold := Kontrol.getSigThreshold(ptr).

- The set `signers` contains parties in `finCom` accumulating relative weight at least equal to `sigThreshold`.

- σ^{AggFin} is a valid aggregate signature from all parties in `signers` on the string $(H(genesis), ptr, round, epoch)$.

> **!** As stated above, the genesis hash $H(\texttt{genesis})$ is used to separate different networks or protocol versions within a network.

Definition 52 (QC order). We say that qc_1 is *higher than* qc_2 if $qc_1.round > qc_2.round$.

We call a block certified if there exists a valid quorum certificate for that block.

Definition 53. A block `block` is *certified* if there exists a valid quorum certificate qc with $qc.ptr = H(\texttt{block})$.

The highest QC seen by a party is a certificate for the newest block for which that party has seen enough signatures.

11.1.2 Timeout Messages and Certificates

If not enough signatures for a new quorum certificate are received within a certain time, a finalizer will call for a *timeout* by generating a timeout message. A timeout message contains the highest QC seen by the finalizer (`timeout.qc`) and an aggregate signature (`timeout.`σ^{AggFin}) on the timeout round, and the round and epoch of the seen QC. This can be seen as a vote for the block from which the chain should continue to grow once we proceed to the next round. Enough of these votes form a timeout certificate.

A timeout message is sent for a specific round. However, rounds are not assigned a specific epoch, i.e., two honest parties may be in different epochs for the same round. Thus, there is no well-defined set of finalizers and stake associated with the round. Instead, we take the epoch (and thus the finalizer committee and the corresponding keys) of the block pointed at by the timeout, i.e., the block for which a QC is included in the timeout message.

Definition 54. A *timeout message*, $\texttt{timeout} \in \textsc{TimeoutMessages}$, is a tuple consisting of

$\texttt{vid} \in \textsc{ValidatorIds}$
> identifier of the sender.

$\texttt{round} \in \mathbb{N}$
> The round number to which this timeout refers.

$\texttt{epoch} \in \mathbb{N}$
> The epoch number of the party generating this timeout.

$\texttt{qc} \in \textsc{QuorumCertificates}$
> The highest quorum certificate known to the sender at the time of the timeout.

$\sigma^{\texttt{AggFin}} \in \textsc{Signatures}_{\texttt{AggFin}}$
> Signature from the sender on $(\mathsf{H}(\texttt{genesis}), \texttt{round}, \texttt{qc.round}, \texttt{qc.epoch})$ using an aggregate signature scheme.

$\sigma^{\texttt{BC}} \in \textsc{Signatures}$
> Signature from the sender on all the entries above and $\mathsf{H}(\texttt{genesis})$.

The validity of a timeout is defined as follows.

Definition 55. A timeout message

$$\texttt{timeout} = (\texttt{vid}, \texttt{round}, \texttt{epoch}, \texttt{qc}, \sigma^{\texttt{AggFin}}, \sigma^{\texttt{BC}})$$

is *partially valid* with respect to $\mathsf{Skov.tree}$ if the following holds:

- Let $\texttt{finCom} \coloneqq \mathsf{Kontrol.getFinalizationCommittee}(\texttt{qc.epoch}, \mathsf{Skov.epochFinEntries})$.
- The sender is a finalizer, namely $\texttt{vid} \in \texttt{finCom}$.
- $\sigma^{\texttt{BC}}$ is a valid signature by $\texttt{vid}$ on the pair $(\mathsf{H}(\texttt{genesis}), \mathsf{H}(\texttt{timeout}))$, where $\mathsf{H}(\texttt{timeout})$ is the hash of everything in the timeout message except $\sigma^{\texttt{BC}}$.
- The $\texttt{qc}$ round and epoch are coherent, i.e., $\texttt{qc.round} < \texttt{round}$ and $\texttt{qc.epoch} \leq \texttt{epoch}$.
- $\sigma^{\texttt{AggFin}}$ is a valid aggregate signature by $\texttt{fid}$ on $(\mathsf{H}(\texttt{genesis}), \texttt{round}, \texttt{qc.round}, \texttt{qc.epoch})$.

It is *completely valid* with respect to $\mathsf{Skov.tree}$ if in addition

- $\texttt{qc.ptr}$ points to a block in a valid chain.
- $\texttt{qc}$ is a valid QC.

> ⚠ Checking that all timeout messages are completely valid would have computational complexity $\mathcal{O}(n^2)$. By only verifying that certain key timeout messages are completely valid (e.g., those that improve on the best known QC), the computational complexity can be kept down to $\mathcal{O}(n \log n)$ if all participants are honest.

Performing too many complete validity checks of timeout messages can increase the computational complexity of the protocol. But if we perform too few, we cannot guarantee that for any two timeouts with round-epoch pairs (r_1, e_1), (r_2, e_2) that pass the validity check we have $r_2 \geq r_1 \implies e_2 \geq e_1$. This property can be useful to more efficiently store, retrieve and manipulate timeouts that are being kept to build a timeout certificate (TC). It also simplifies some other checks performed.

In addition to the validity checks above, one can also verify that the QC in the timeout does not point past the last known finalized block. This is not needed for security of the protocol, but allows us to purge timeouts from dishonest parties that could require keys from pay days far in the past to verify.

A timeout certificate (TC) is formed for a round `round` when enough timeout messages are received for that round. As said above, each timeout message contains the highest QC seen by that party. So the TC will list all the corresponding round (and epoch) numbers in each timeout (`tc.qcrounds`) along with the `vid` of the senders of the timeouts (`tc.signers`). The TC will also contain the aggregate signature of these rounds (`tc.`$\sigma^{\texttt{AggFin}}$). This list of QC round numbers is used to prove that the QC in the next block is at least as high as all those signed in the TC.

Definition 56. A *timeout certificate*, `tc` $\in$ TIMEOUTCERTIFICATES, is a tuple consisting of

`round` $\in \mathbb{N}$
>   The round number to which this certificate refers.

`qcrounds` $\in \mathbb{N}^*$
>   A list of round and epoch numbers. In the protocol, this will correspond to the round numbers of the best blocks of the parties contributing to the timeout certificate.

$\sigma^{\texttt{AggFin}} \in$ SIGNATURES$_{\texttt{AggFin}}$
>   An aggregate signature on $(\mathsf{H}(\texttt{genesis}), \texttt{round}, \texttt{round}_i, \texttt{epoch}_i)_{(\texttt{round}_i, \texttt{epoch}_i) \in \texttt{qcrounds}}$.

`signers` $\in$ VALIDATORIDS*
>   List of finalizers whose signature is in $\sigma^{\texttt{AggFin}}$. There exists a one-to-one mapping between the $\texttt{vid}_i \in \texttt{signers}$ and $(\texttt{round}_i, \texttt{epoch}_i) \in \texttt{qcrounds}$, which we denote by the index i.

The most efficient algorithm for verifying a BLS aggregate signature has two loops. An outer loop that goes over all *different* messages that have been signed, and an inner loop that goes over all finalizers that have signed the same message. The BLS pairing function is called once in every iteration of the outer loop. The data structure of `tc.qcrounds` and `tc.signers` should be chosen such that these loops can be efficiently implemented.

Validity of a timeout certificate (TC) is defined with respect to a QC with which the TC is paired. In the protocol, this corresponds to the QC included in the block along with the TC, i.e., the QC of the parent block.

⚠ Note that the epoch with respect to which the TC is valid can be different from the epoch with respect to which (some of) the timeouts included in the TC are valid. We thus need to fetch the corresponding keys from different epochs for checking all signatures. Furthermore, since the epoch used to check the validity of the TC is not included in the TC, it is important to always be able to recover this information when storing or sharing the TC, e.g., to include the paired QC along with the TC.

Definition 57. A timeout certificate $\mathtt{tc} = (\mathtt{round}, \mathtt{qcrounds}, \sigma^{\mathtt{AggFin}}, \mathtt{signers})$ is *valid* with respect to $\mathsf{Skov.tree}$ and some QC $\mathtt{qc}$ if the following holds:

- $\mathtt{qc.ptr}$ points to a block in a valid chain.

- Let $\mathtt{finCom} \coloneqq \mathsf{Kontrol.getFinalizationCommittee}(\mathtt{qc.epoch}, \mathsf{Skov.epochFinEntries})$.

- Let $\mathtt{sigThreshold} \coloneqq \mathsf{Kontrol.getSigThreshold}(\mathtt{qc.ptr})$.

- The set $\mathtt{signers}$ contains parties in $\mathtt{finCom}$ accumulating relative weight (with respect to the stake from epoch $\mathtt{qc.epoch}$) at least equal to $\mathtt{sigThreshold}$.

- For every $\mathtt{fid}_i \in \mathtt{signers}$, $\sigma^{\mathtt{AggFin}}$ contains a valid signature on the string $(\mathsf{H}(\mathtt{genesis}), \mathtt{round}, \mathtt{round}_i, \mathtt{epoch}_i)$, where the common index i denotes the one-to-one mapping between elements in $\mathtt{signers}$ and $\mathtt{qcrounds}$.

> </> The above conditions must be satisfied for the protocol to be secure. Additionally, to simplify the manipulation of TCs, one can require that
> - The epochs in $\mathtt{qcrounds}$ do not cover more than two consecutive epochs.
> - The $(\mathtt{round}, \mathtt{epoch})$ pairs stored in $\mathtt{qcrounds}$ satisfy that for any $(r_1, e_1), (r_2, e_2) \in \mathtt{qcrounds}$, $r_2 \geq r_1 \implies e_2 \geq e_1$.
>
> Note that these two rules are automatically satisfied if all parties are honest, and can be enforced if desired. The rule that the epoch numbers do not differ by more than 1 means that we do not need to store and retrieve very old keys to verify the signature $\sigma^{\mathtt{AggFin}}$. The rule $r_2 \geq r_1 \implies e_2 \geq e_1$ can make it easier to store and handle timeouts in order to build a TC.

> ❶ The genesis hash $\mathsf{H}(\mathtt{genesis})$ is used to separate different networks or protocol versions within a network.

11.1.3 Block Finalization Entries

An important part of the consensus protocol is the finalization of blocks as the chain of finalized blocks defines the totally ordered list of broadcast messages.

Finalization entries act as proof for the finalization of a block. If a block and its direct descendant have consecutive round numbers, are in the same epoch and are both certified, then the first of the two—and any previous blocks—are final. Thus, quorum certificates for the two blocks serve as a finalization proof. We formally define finalization entries as follows.

Definition 58. A *finalization entry*, $\mathtt{finEntry} \in \textsc{FinalEntries}$, is a tuple consisting of

$\mathtt{qc}_1 \in \textsc{QuorumCertificates}$
    Quorum certificate for the finalized block.

$\mathtt{qc}_2 \in \textsc{QuorumCertificates}$
    Quorum certificate for the block in the round after the finalized block.

$\mathtt{str} \in \textsc{Hash}$
    A hash of everything in the block $\mathtt{qc}_2.\mathtt{ptr}$ that is needed to verify that it is the direct descendant of $\mathtt{qc}_1.\mathtt{ptr}$.

> ⓘ The hash of the finalized block is contained in $\mathtt{qc}_1$ as $\mathtt{qc}_1.\mathtt{ptr}$.

> ⚠ The finalized block (namely the block with hash $qc_1.ptr$) will be available on-chain, but $qc_2.ptr$ might not be there. str is provided so that the validity of a `finEntry` can be verified without access to the block with hash $qc_2.ptr$. In the current implementation, the hash of a block is obtained by hashing together the hash of a header (`BlockHeaderHash`) and the hash of the body (`BlockQuasiHash`). The header is a tuple of the block `round`, `epoch` and parent hash `ptr`. Thus, the string in Definition 58 is $str = $ `BlockQuasiHash`. And to verify that the block certified by qc_2 is a direct descendant of the block certified by qc_1, it is sufficient to check that qc_2 is valid and that $H(H(qc_2.\text{round}, qc_2.\text{epoch}, qc_1.\text{ptr}), str) = qc_2.ptr$.

A finalization entry is considered valid if both quorum certificates are valid and they certify consecutive blocks with adjacent round numbers and the same epoch number.

Definition 59. A finalization entry $\texttt{finEntry} = (qc_1, qc_2, str)$ is *valid* with respect to $\mathsf{Skov.tree}$ if the following holds:

- qc_1 and qc_2 are valid quorum certificates, cf. Definition 51.

- The second certified block extends the first one, i.e.,
  $qc_2.\text{ptr} = H(H(qc_2.\text{round}, qc_2.\text{epoch}, qc_1.\text{ptr}), str)$.

- The blocks have consecutive round numbers, i.e., $qc_2.\text{round} = qc_1.\text{round} + 1$.

- The two blocks are in the same epoch, i.e., $qc_1.\text{epoch} = qc_2.\text{epoch}$.

> ⓘ The exact formula in Definition 59 (and in the notice box just before the definition) for verifying that $qc_2.ptr$ is a child of $qc_1.ptr$ depends on how blocks are hashed before signing. If a change is made to the corresponding hash tree, this formula would need updating.

> ⓘ The security proof only requires the two blocks to be in the same pay day, not in the same epoch, because this is sufficient to guarantee that the stake is the same for both QCs. If the two QCs are in different pay days, then a proof that the first is finalized only holds up to $1/3 - \alpha$ corruption, where α is the total variation distance between the distribution of stake of the two pay days. See Section 11.2 for more information on epochs and pay days.

11.1.4 Epoch Finalization Entries

Every epoch contains a block called *trigger block*, namely the first block with timestamp after the nominal end of the epoch (see also Section 11.2). The next epoch may only be started once this trigger block is finalized.[2] We call an epoch *finalized* if such a finalization proof exists.

Definition 60. We call an epoch *finalized* if there exists a finalization proof for the trigger block.

An epoch finalization proof thus consists of a finalization proof for any block that is a descendant of the trigger block.

Definition 61. An *epoch finalization entry*, $\texttt{epochFinEntry} \in \text{EPOCHFINALENTRIES}$, for an epoch $\texttt{epoch} \in \mathbb{N}$ consists in a finalization entry

---

[2]In the case of a fork before the nominal end of an epoch, there may be multiple trigger blocks on different branches. But only one of these will get finalized.

`finEntry` ∈ FINALENTRIES
> A finalization entry for a block after the trigger block of the corresponding epoch.

An epoch finalization entry is considered valid if `finEntry` is indeed a finalization proof for the trigger block of the corresponding epoch. We formally define validity as follows.

Definition 62. An epoch finalization entry $\texttt{epochFinEntry} = (\texttt{finEntry})$ is *valid* with respect to `Skov.tree` and an epoch `epoch` if the following holds:

- Let $(\texttt{qc}_1, \texttt{qc}_2, \texttt{str}) := \texttt{finEntry}$.

- `finEntry` is a valid finalization entry.

- $\texttt{qc}_1.\texttt{epoch} = \texttt{epoch}$.

- $\texttt{qc}_1.\texttt{ptr.timestamp} \geq$ nominal end of epoch `epoch`.

Trigger blocks are also used to define the moment a protocol update happens. So in case of a protocol update, the state defined by a finalized trigger block will define the new genesis state. For more information see Section 11.6.

11.1.5 Blockchain and Block Tree

The goal of the consensus layer is to maintain a sequence of data blocks. These data blocks are encoded in the form of a blockchain which is a linked list of blocks. Each block contains a pointer to the previous block, information on its creation, and the block data.

Definition 63. A *block*, $\texttt{block} \in$ BLOCKS, is a tuple consisting of

`round` ∈ ℕ
> The *round number* in which this block is created.

`epoch` ∈ ℕ
> The *epoch* to which this block belongs.

`timestamp` ∈ ℕ
> Time at which the block was created as Unix timestamp in milliseconds.

`vid` ∈ VALIDATORIDS
> ID of the *validator* who created the block.

`qc` ∈ QUORUMCERTIFICATES
> A *quorum certificate* of the previous block in the blockchain this block is extending.

`tc` ∈ TIMEOUTCERTIFICATES ∪ {⊥}
> A *timeout certificate* of the previous round if it timed out, and ⊥ otherwise.

`epochFinEntry` ∈ EPOCHFINALENTRIES ∪ {⊥}
> For the first block of an epoch, an *epoch finalization entry* for the previous epoch.

`nonce` ∈ BLOCKNONCES
> The *block nonce* is a proof `prf` that a random string `val` has been correctly generated, cf. Section 11.3. The block nonce is used to generated randomness for the leader election.

`data` ∈ DATA*
> `data` is the transactions stored in this block.

transactionsOutcomesHash $\in$ HASH

> **transactionsOutcomesHash** is the hash of the transactions outcomes, e.g., cost, sender, success/failure, list of effects on the chain..

stateHash $\in$ HASH

> A hash of the state after executing the transactions in this block.

blockHash $\in$ HASH

> The root of the hash tree of all the entries above in this block and some extra information. For a given block **block**, we define it as **blockHash** = H(**block**).

$\sigma^{\text{BC}} \in$ SIGNATURES

> The signature of the validator **validator** on the pair $(\text{H}(\texttt{genesis}), \texttt{blockHash})$, i.e., on the genesis block (to uniquely determine the network and protocol version) and all the values above.

> ⓘ The block data structure does not explicitly contain a pointer to the parent block because the quorum certificate in the block already contains such a pointer. Hence, the parent of **block** is by definition **block.qc.ptr**.

> </> In the implementation the block size is limited (see Kontrol.maxBlockEnergy). It is part of engineering to find the right size of blocks. Intuitively, smaller blocks propagate faster over the network while bigger blocks allow adding more data per time unit. This is linked to the Kontrol.minBlockTime parameter, which controls how fast blocks can be created. We can have small blocks created fast, or larger blocks with a bigger **minBlockTime**. The former should favor a smaller latency, the latter a higher TPS.

Definition 64. A block

$$\texttt{block} = (\texttt{round}, \texttt{epoch}, \texttt{timestamp}, \texttt{vid}, \texttt{qc}, \texttt{tc}, \texttt{epochFinEntry}, \texttt{nonce}, \texttt{data},$$
$$\texttt{transactionsOutcomesHash}, \texttt{stateHash}, \texttt{blockHash}, \sigma^{\text{BC}})$$

is *valid* with respect to Skov.tree if the following holds:

- **qc.ptr** points to a block in a valid chain,

- **vid** = getLeader(**round**, **epoch**, Skov.epochFinEntries), i.e., the validator was elected this round,

- σ^{BC} is a valid signature by **vid** on $(\text{H}(\texttt{genesis}), \texttt{blockHash})$,

- **timestamp** $\geq$ **qc.ptr.timestamp** + Kontrol.getMinBlockTime(**timestamp**, **qc.ptr**),

- **qc** is a valid QC,

- either (**qc.round** = **round** $- 1$ and **tc** $= \perp$) or (**tc.round** = **round** $- 1$ and **qc.round** $\geq$ max{**tc.qcrounds**} and **tc** is a valid TC with respect to **qc.ptr**),

- an epoch finalization entry is provided if and only if this is the first block in a new epoch, i.e., (**epoch** = **qc.epoch** and **epochFinEntry** $= \perp$) or (**epoch** = **qc.epoch** $+ 1$ and **epochFinEntry** is a valid epoch finalization entry for **epoch** $- 1$),

- **nonce** is a valid block nonce, i.e., verifyBlockNonce(**round**, **nonce**$^{\text{LE}}$, **vid.vk**$^{\text{LE}}$, **nonce**) = true,

- **data** is valid,

- `transactionsOutcomesHash` is correct,

- `stateHash` is correct,

- `blockHash = H(block)`.

> **!** As mentioned in Definition 63, $H(\texttt{block})$ does not just compute a hash of `block`, but generates a hash tree of all the entries (except for `blockHash` and $\sigma^{\texttt{BC}}$).

The blockchain starts with the genesis block `genesis`. This unique block contains the initial system parameters of the Konsensus layer. A blockchain is a linked list of blocks where the first block must be the genesis block `genesis`.

Definition 65. A *blockchain*, $\texttt{chain} \in \textsc{chains} \coloneqq \textsc{blocks}^*$, is a sequence of blocks starting with the genesis block, i.e., $\texttt{chain} = \texttt{block}_0, \ldots, \texttt{block}_n$, where the first block is the genesis block $\texttt{block}_0 = \texttt{genesis}$.

For a chain $\texttt{chain} = \texttt{block}_0, \ldots, \texttt{block}_n$, we use the following notation and terminology:

- The *chain data* is the sequence of the block data, i.e., $\texttt{chain.chainData} = (\texttt{block}_0.\texttt{data}, \ldots, \texttt{block}_n.\texttt{data})$

- The *head* of the chain is the last block in the chain, i.e., $\texttt{chain.head} = \texttt{block}_n$.

- The *depth* of a block is its index in the blockchain, i.e., $\texttt{depth}(\texttt{block}_i) = i$. Note that the genesis block has depth 0.

> **i** The chain is implicit in the definition of the depth of a block and assumed to be clear from context.

A block tree is a collection of blockchains which all start with the same genesis block.

Definition 66. A *block tree* $\texttt{tree} \in \textsc{trees}$ is a tree of blocks with root `genesis` where every path from the root to a leaf forms a blockchain.

11.1.6 Skov Data Layer

The Skov data layer is used by a party, e.g., a validator, to store all data related to the consensus layer.

Definition 67. The Skov data structures are

$\textsf{Skov.tree} \in \textsc{trees}$
> The *block tree* maintained by Skov. It initially contains the genesis block `genesis`.

$\textsf{Skov.blockPool} \in \textsc{blocks}^*$
> The *block pool* is a list of pending (valid) blocks which have not yet been added to Skov.tree. Initially Skov.blockPool is empty.

$\textsf{Skov.inputPool} \in \textsc{inputdata}^*$
> The *input pool* is a list of pending input data which has not yet been added to blocks in Skov.tree.

$\textsf{Skov.finEntries} \in \textsc{finalentries}^*$
> The list of *finalization entries* maintained by Skov. In practice one only needs to store the last finalized entry. All ancestors are automatically final.

Skov.epochFinEntries ∈ EPOCHFINALENTRIES*
> List of *epoch finalization proofs* for all previous epochs.

ℹ️ We explicitly distinguish between input data and (the processed) data which is in a block. For instance, the input data could be a transaction and the corresponding data (in a block) is the resulting update of the involved account (balances).

The interfaces of Skov allow to read the current data sequence or to provide new input in the form of blocks or input data. It is guaranteed that Skov incorporates new blocks periodically.

Data Output

The function Skov.getData returns the finalized data sequence currently represented by Skov.tree.

Function Skov.getData

Inputs
Skov.tree ∈ TREES
> The block tree maintained by Skov.

Outputs
data ∈ DATA*
> The data is the *full* data sequence of the longest finalized chain in Skov.tree.

⚠️ If the security assumptions of the consensus protocol hold, the longest finalized chain will be unique and any other finalized chain will be a prefix of the longest one.

Add Block

The function Skov.addBlock adds a block to Skov.blockPool, the pool of pending blocks. Then it adds all possible blocks from Skov.blockPool to Skov.tree.

Function Skov.addBlock

Inputs
block ∈ BLOCKS
> The block to be added to the block pool.
Skov.blockPool ∈ BLOCKS*
> The block pool maintained by Skov.
Skov.tree ∈ TREES
> The block tree maintained by Skov.

Outputs
Skov.blockPool ∈ BLOCKS*
> The updated block pool.
Skov.tree ∈ TREES
> The updated tree.

> **Promise**
>
> The input `block` is either in the updated Skov.blockPool or the updated Skov.tree. The updated Skov.blockPool does not contain any block which can be added to the updated Skov.tree. All blocks removed from Skov.blockPool have been added to Skov.tree.
>
> This function must not add blocks with future timestamps to Skov.tree.

> ℹ The promise of `addBlock` guarantees that a block stays in Skov.blockPool until it can be added to Skov.tree. However, the function could in principle also remove invalid blocks (including the input block) from Skov.blockPool, i.e., blocks which can never be added to Skov.tree. Note that the removal of invalid blocks is *not* required for security, but will increase the efficiency of the protocol and does *not* harm the security.

Add Input Data

The function Skov.addInputs adds input data to Skov.`inputPool`, the pool of input data.

> **Function** Skov.addInputs
>
> **Inputs**
>
> `inputData` ∈ INPUTDATA
>   The input data to be added to the input pool.
>
> Skov.`inputPool` ∈ INPUTDATA*
>   The input pool maintained by Skov.
>
> **Outputs**
>
> Skov.`inputPool` ∈ INPUTDATA*
>   The input pool with newly added `inputData`.

> ℹ The function `addInputs` could filter out entries which will under no circumstances be added to Skov.`inputPool` (e.g., as they are invalid). Note that the filtering is *not* required for security, but will increase the efficiency of the protocol.

Get Chain

The function Skov.getChain returns the chain from `genesis` up to the input block.

> **Function** Skov.getChain
>
> **Inputs**
>
> Skov.tree ∈ TREES
>   The block tree maintained by Skov.
>
> block ∈ BLOCKS
>   A block in Skov.tree
>
> **Outputs**

> `chain` ∈ CHAINS
>   The chain from `genesis` up to `block`.

11.1.7 Catchup

Catch-up is the synchronization process that allows a party that knows at least the genesis block `genesis` to become parat. This works as follows.

To synchronize with a peer a party sends a catch-up requests. This catch-up request includes a summary of the node's consensus status, including the last finalized block hash and round, the hashes of any subsequent blocks, the current round number, and a summary of what validators have signed in the current round. Based on the request, the peer can determine any blocks and signatures it has that the node is missing. The peer then sends these to the node in the appropriate order, followed by a catch-up response message with similar information to the catch-up request, but from the perspective of the peer. If the node is very far behind, the peer will limit the number of blocks it sends in response to a request. The catch-up response message allows the node to determine if it is now fully up-to-date with the peer, or if it needs to make further catch-up requests. The node attempts to catch-up with each of its peers one at a time in round-robin fashion until it is up-to-date with each of them. This distributes the burden across the network, and ensures the node does not receive redundant information from multiple peers.

> ⚠ For liveness—or more precisely to have the guarantee that an honest party becomes eventually parat—it is crucial that it can catch-up with at least one honest peer. This implies that honest parties must be well-connected within the communication network. Note that there is no requirement to trust peers during catch-up, i.e. dishonest peers will not be able to convince the honest party of wrong information (assuming the honest party has the right genesis block).

> </> The implementation also supports an out-of-band catch-up mechanism that is useful when a party is far behind the rest of the network—for instance, if it is joining for the first time. In this mechanism, the party downloads blocks from a web server before it ever connects to peers. These blocks are processed and verified just as if they were received on the peer-to-peer network, so there is no requirement to trust the server.

Faster Catch-Up

Catching-up block by block is not very efficient. Instead, parties could catch-up by jumping from one finalization committee update block to the next:

- The party knows some old finalization committee, e.g., the one from the genesis block.

- It then requests from its peers the block that defines the next committee (including finalization proofs) and verifies the block using the current finalization committee information.

- The process is repeated until the party know the latest finalization committee.

- Then the party will request the latest block (including state) of the blockchain and verifies them using the committee information.

11.1.8 Leader Election

The leader election module provides a function for determining the designated validator for a given epoch and round with getLeader. It also provides two functions for computing block- and leader-election-nonces, respectively, namely computeBlockNonce and getLENonce. These are defined in Section 11.3.

Any choice of leader election method would have to provide a function getLeader, but computeBlockNonce and getLENonce are specific to the leader election method chosen for our implementation.

11.1.9 Input Data Processing

We distinguish between input data (of type INPUTDATA) and block data (of type DATA). Validators collect input data in Skov.inputPool. This input data is then processed to get the data for a new block. The exact selection of data for a new block depends on the application. We therefore assume that the consensus protocol has access to the function processInputs.

Function processInputs

Inputs

chain ∈ CHAINS
> The chain which should be extended.

round ∈ $\mathbb{N}$
> The round number of the to-be-created block.

maxBlockEnergy ∈ $\mathbb{N}$
> The maximum size of the block to be created.

Skov.inputPool ∈ INPUTDATA*
> The block input pool maintained by Skov.

Outputs

data ∈ DATA*
> The block data for the to-be-created block.

Skov.inputPool ∈ INPUTDATA*
> The updated input pool.

Promise

> The returned data allows to create a block for slot which can be validly added to chain. All inputs removed from the updated Skov.inputPool have been used to create

---

```
        data.
```

> **!** This function abstracts the data selection for new blocks. The implementation of this algorithm highly depends on the actual data layer of the project. The validators should be incentivized to employ a function `processInputs` which adds as much data as possible (within the limits of the maximum block size). Bitcoin uses transaction fees to achieve this incentive.

11.2 Epochs and Pay Days

Every block is assigned an *epoch* number $e \in \mathbb{N}$ in increasing order, which is added to the block in the field `block.epoch`. The leaders for every round in an epoch—the validator that produces the block in that round—are chosen deterministically from a $\texttt{nonce}^{\texttt{LE}}$, and in every new epoch the $\texttt{nonce}^{\texttt{LE}}$ is recomputed unpredictably from hashes of nonces present in each block from the previous epoch. Thus, the leaders for an epoch can be determined at the beginning of an epoch for the whole epoch.

The initial value of `epochDuration` is set to be 1 hour. The *nominal* begin and end of an epoch are determined by a clock, e.g., 8:00, 9:00, etc. But the actual way in which blocks are assigned to epochs is not determined by block time only. The first block produced after the nominal end of an epoch is called the *trigger block*, as it starts the epoch transition phase. The validators continue producing blocks in the old epoch, until the trigger block has been finalized. Once such a finalization proof is available, the validators compute the new $\texttt{nonce}^{\texttt{LE}}$ and the new leader for the next round. When this validator generates the next block, they increase the epoch by 1 and add the epoch finalization proof to the block. If the finalization proof for the trigger block is not seen by enough validators, it is possible that a few more blocks are produced in the old epoch before switching to the new one.

The stake of validators can only be updated at the end of an epoch. Only transactions in blocks up to and including epoch $e-2$ with transaction time before the nominal end of epoch $e-1$ are taken into account when determining the stake of validators for epoch e. Though we choose to not update the stake every epoch, but only every `rewardPeriodLength` epochs, which is initially set to 24. We call this a *pay day*, and update stake and distribute baking rewards at the beginning of every new pay day. Thus, validators and finalizers can only join and leave at the beginning of a new pay day and the stake is stable for an entire pay day.

11.3 Leader Election

ConcordiumBFT proceeds in rounds, where in every round of an epoch, a unique *leader* is elected to propose a block. The probability for a given validator to be elected in a given round corresponds to the relative stake of that validator. As described in Section 11.2, a new leader-election nonce is determined for every epoch. The leader of a round is computed deterministically from the current leader-election nonce and the stake distribution. The leader-election nonce is computed from block nonces in blocks from the previous epoch and the stake distribution.

> **!** Once the leader-election nonce and stake distribution for an epoch are determined, all leaders for that epoch are publicly predictable.

11.3.1 Lottery Power

The lottery power of a validator `validator` $\in$ VALIDATORS corresponds to the fraction of stake in that validator's pool.[3]

Definition 68. The *lottery power*, `lotteryPower` $\in$ LOTTERYPOWER, is a real number between 0 and 1, i.e., LOTTERYPOWER $= [0, 1]$. A lottery power of 1 means that the validator controls all lottery tickets.

> ⓘ In an implementation the lottery power of a validator should roughly correspond to their current stake in the system. The actual translation of stake to lottery power is done at the application layer. ConcordiumBFT just assumes that the list of validators with their lottery power can be somehow extracted from the blockchain (see Section 12.2.3).

11.3.2 Block Nonce and Leader-Election Nonce

The block nonce is required to generate the randomness which is used to update the leader-election nonce.

Definition 69. A *block nonce* `nonce` $\in$ BLOCKNONCES consists of

`prf` $\in$ PROOF$_{\text{VRF}}$
    A proof for the correct generation of a hash `val` is a VRF proof, cf. Section 6.2.

The leader-election nonce is used to make the leader election random and unpredictable.

Definition 70. A *leader election nonce*, `nonce`$^{\text{LE}}$ $\in$ NONCE$_{\text{LE}}$, is a bit string of length k, i.e., NONCE$_{\text{LE}} := \{0, 1\}^k$.

11.3.3 Block-Nonce Functions

The *block nonce generation* function computeBlockNonce allows the block nonce for a specific round, a specific epoch and a specific validator to be computed.

Function computeBlockNonce

Inputs

`round` $\in \mathbb{N}$
    The slot for which the nonce should be computed.
`nonce`$^{\text{LE}}$ $\in$ NONCE$_{\text{LE}}$
    The leader-election nonce for the desired epoch.
`sk`$^{\text{LE}}$ $\in$ PRIVATEKEY$_{\text{LE}}$
    The leader election secret key of the validator.

Outputs

`nonce` $\in$ BLOCKNONCES
    The block nonce.

Implementation   The function is implemented using a VRF scheme, cf. Section 6.2.

---

[3]Since the Concordium chain allows delegation, the stake controlled by a validator is equal to their own stake plus the stake delegated to their pool minus some penalties if their pool is too large. The actual formula is not relevant to the consensus protocol and out of the scope of this Blue Paper. More information on this may be found in the White Paper [Con24].

1: Compute $\texttt{nonce} = \mathsf{VRFProve}(\texttt{sk}^{\mathrm{LE}}, \text{"NONCE"}||\texttt{nonce}^{\mathrm{LE}}||\texttt{round})$.
2: **return** $\texttt{nonce}$.

The *block-nonce verification* function verifyBlockNonce allows verifying the block nonce of a block.

Function verifyBlockNonce

Inputs

$\texttt{round} \in \mathbb{N}$
    The round for which the nonce should be verified.

$\texttt{nonce}^{\mathrm{LE}} \in \mathrm{NONCE}_{\mathrm{LE}}$
    The leader-election nonce for the corresponding epoch.

$\texttt{vk}^{\mathrm{LE}} \in \mathrm{VERIFICATIONKEY}_{\mathrm{LE}}$
    The leader election verification key of the validator.

$\texttt{nonce} \in \mathrm{BLOCKNONCES}$
    The block nonce.

Outputs

$b \in \mathrm{BOOL}$
    Indicates if the $\texttt{nonce}$ is valid

Implementation The function is implemented using a VRF scheme, cf. Section 6.2.

1: **if** $\mathsf{VRFVerifyKey}(\texttt{vk}^{\mathrm{LE}}) = \texttt{false}$ **then** return $\texttt{false}$.
2: **if** $\mathsf{VRFVerify}(\texttt{vk}^{\mathrm{LE}}, \text{"NONCE"}||\texttt{nonce}^{\mathrm{LE}}||\texttt{round}, \texttt{nonce}) = \texttt{false}$ **then** return $\texttt{false}$.
3: Return $\texttt{true}$.

> ❗ Block nonces are used to compute future leader-election nonces. To prevent grinding-attacks, it must hold that for fixed $\texttt{vk}^{\mathrm{LE}}$ (or the corresponding private key $\texttt{sk}^{\mathrm{LE}}$) and $\texttt{nonce}^{\mathrm{LE}}$, the block nonce for round $\texttt{round}$ is unique. To prevent parties from learning future leader-election nonces too early, it must hold that the block nonce for round $\texttt{round}$ and private key $\texttt{sk}^{\mathrm{LE}}$ from an honest party looks random if one does not know the private key $\texttt{sk}^{\mathrm{LE}}$. The leader-election nonce $\texttt{nonce}^{\mathrm{LE}}$ is included in the block nonce computation to make block nonces for a given validator not predetermined forever.

11.3.4 Leader Election Functions

The function getLENonce returns the leader-election nonce for a given epoch.

Function getLENonce

Inputs

$\texttt{epoch} \in \mathbb{N}$
    The epoch for which to compute the leader-election nonce.

$\texttt{epochFinEntries} \in \mathrm{EPOCHFINALENTRIES}^*$
    List of epoch finalization proofs for all previous epochs.

$\text{nonce}^{\text{LE}} \in \text{NONCE}_{\text{LE}}$

The leader-election nonce for epoch `epoch`.

Promise

The nonce^{LE} is uniquely determined if `epoch` $= 1$ or `epochFinEntries[epoch − 1]` is not empty.

Implementation

1: **if** `epoch` $= 1$ **then**
2:     **return** `genesisData.nonce`$^{\text{LE}}$
3: $\text{nonce}^{\text{LE}} := \text{getLENonce}(\text{epoch} - 1, \text{epochFinEntries})$
4: Let $\mathcal{B}$ be the set all blocks from the first in epoch `epoch` $- 1$ to the trigger block of `epoch` $- 1$ (included).
5: **for** `block` $\in \mathcal{B}$ **do**
6:     $\text{nonce}^{\text{LE}} := \text{H}(\text{nonce}^{\text{LE}} \| \text{VRFproof2hash}(\text{block.nonce}))$
7: $\text{validators} := \text{Kontrol.getValidatorSet}(\text{epoch}, \text{epochFinEntries})$
8: */* The following ensures that the leader-election nonce depends on the validator set, including the stake distribution. */*
9: $\text{nonce}^{\text{LE}} := \text{H}(\text{nonce}^{\text{LE}} \| \text{epoch} \| \text{validators})$
10: **return** nonce^{LE}

Finally, we define how the leader for a given round is selected based on the leader-election nonce and the validator set, which includes the stake distribution.

Function getLeader

Inputs

`round` $\in \mathbb{N}$

The round for which the leader is selected.

$\text{nonce}^{\text{LE}} \in \text{NONCE}_{\text{LE}}$

The leader-election nonce of the desired epoch.

`validators` $\in \text{VALIDATORS}^*$

List of validators with their lottery power.

Outputs

`validator` $\in \text{VALIDATORS}$

The leader for round `round`.

Promise

For fixed nonce^{LE} and `validators`, the function is deterministic. For uniformly random nonce^{LE}, the returned `validator` is randomly chosen from `validators`, weighted by lottery power. That is, $\Pr[\text{validator} = B] = \text{validators}[B].\text{lotteryPower}$. For different rounds, the leaders are independent.

Implementation

1: */* This is an example implementation. */*
2: Assign each validator $B \in$ `validators` a sub-interval of $[a, a + B.\texttt{lotteryPower}) \subseteq [0, 1)$ of length $B.\texttt{lotteryPower}$

3: $h := \mathsf{H}(\mathsf{nonce}^{\mathrm{LE}} \| \mathsf{round})$, interpreted as real number in $[0, 1)$
4: let `validator` be the validator assigned the interval containing h
5: **return validator**

</> Finding the validator assigned the interval containing h takes time logarithmic in the number of validators if implemented using a binary search. For many validators, more efficient algorithms for weighted sampling exist, see [Wal77; Vos91].

We also provide a wrapper for this function to compute the leader-election nonce given a round number, epoch number, pointer to a chain, and epoch finalization proofs.

Function getLeader

Inputs

`round` $\in \mathbb{N}$
  The round for which the leader is selected.

`epoch` $\in \mathbb{N}$
  The epoch for which to compute the leader-election nonce.

`epochFinEntries` $\in$ EPOCHFINALENTRIES*
  List of epoch finalization proofs for all previous epochs.

Outputs

`validator` $\in$ VALIDATORS
  The leader for round `round`.

Promise

  $\mathsf{epoch} = 1$ or $\mathsf{epochFinEntries}[\mathsf{epoch} - 1]$ is not empty.

Implementation

1: $\mathsf{nonce}^{\mathrm{LE}} := \mathsf{getLENonce}(\mathsf{epoch}, \mathsf{epochFinEntries})$
2: $\mathsf{validators} := \mathsf{Kontrol.getValidatorSet}(\mathsf{epoch}, \mathsf{epochFinEntries})$
3: $\mathsf{validator} := \mathsf{getLeader}(\mathsf{round}, \mathsf{nonce}^{\mathrm{LE}}, \mathsf{validators})$
4: **return validator**

ⓘ The condition "$\mathsf{epochFinEntries}[\mathsf{epoch} - 1]$ is not empty" above is required for the leader to be uniquely determined. However, if we already have a trigger block for $\mathsf{epoch} - 1$, one can still compute what the leaders would be for the next epoch if this trigger block gets finalized.

11.4 Consensus Protocol

ConcordiumBFT proceeds in rounds. In every round a leader is elected, and they produce a new block. To progress to the next round, either enough parties sign and receive signatures for this block or enough parties throw and receive a timeout.

In our implementation, parties that are validators can get chosen to be the leader for a round and those that are additionally finalizers will sign blocks and throw timeouts. By requiring

finalizers to have a minimum amount of stake that is higher than for other validators, we prevent the network from being clogged by messages with too little weight.

> ⓘ If an error is detected, e.g., an invalid block or double signing, we call a `flag` function to flag the behavior. This is described in more detail in Section 11.8.

Local variables. Most functions defined in this section query the same local variables, e.g., a variable `highestQC` which stores the highest QC seen so far. We define these variables here.

`crntRound` $\in \mathbb{N}$
> The current round number.

`crntEpoch` $\in \mathbb{N}$
> The current epoch number.

`currentLENonce` $\in \text{NONCE}_{\text{LE}}$
> Leader-election nonce of the current epoch.

`nextSignableRound` $\in \mathbb{N}$
> Blocks in this round and higher can be signed.

`currentTimeout` $\in \mathbb{N}$
> The current timeout time. Initially set as `currentTimeout := genesisData.timeoutBase`.

`highestQC` $\in \text{QUORUMCERTIFICATES}$
> The highest QC seen so far.

`roundType` $\in \{\text{QC}, \text{TC}\}$
> Whether a QC or TC made us progress to the current round.

`roundTC` $\in \text{TIMEOUTCERTIFICATES} \cup \{\bot\}$
> The TC from the previous round if it exists, and $\bot$ otherwise.

`roundTCsQC` $\in \text{QUORUMCERTIFICATES} \cup \{\bot\}$
> The QC paired with `roundTC`, namely the highest QC when the `roundTC` was built, or $\bot$ if `roundTC` = $\bot$.

`lastEpochFinEntry` $\in \text{EPOCHFINALENTRIES}$
> The `epochFinEntry` for the previous epoch.

`me` $\in \text{VALIDATORS}$
> A constant. The ID of the party running the code.

11.4.1 Making Blocks

This procedure is only called by the elected leader for the corresponding round and epoch. Once the block is made, we also sign it as valid if we are a finalizer (see Section 11.4.2 for the signing procedure).

Protocol makeBlock

Promise
> `me` is the winner of the leader election for `crntRound` and `crntEpoch`. `roundType`,

---

roundTC, roundTCsQC and `lastEpochFinEntry` have been assigned correct values before calling makeBlock. In particular if `roundType` == "TC" then `highestQC.round` < `crntRound` − 1.

Implementation

1: **if** `roundType` == "TC" AND `highestQC.epoch` > `roundTCsQC.epoch` **then**
2:     /* *It is unclear if in practice we could have* `highestQC` ≠ `roundTCsQC`, *because as soon as we build the TC, we advance rounds and make the block, so we would need a process working in parallel to update* `highestQC` *for this to happen. But if it does happen and the epoch changes, then we might have a TC+QC spanning 3 epochs or a change of payday (which could change the weights), which could invalidaty the TC we built.* */
3:     blockQC ← roundTCsQC
4: **else**
5:     blockQC ← highestQC
6: **if** `crntEpoch` > `blockQC.epoch` **then**
7:     epochFinEntry ← lastEpochFinEntry
8:     **if** `crntEpoch` is the last epoch of the payday **then**
9:        take the snapshot of the validators for the next payday
10:     **if** `crntEpoch` is the first epoch of the payday **then**
11:        add rewards to the block
12: **else**
13:     epochFinEntry ← ⊥
14: nonce ← computeBlockNonce(crntRound, currentLENonce, me.sk$^{\text{LE}}$).
15: (data, inputPool) ← processInputs(blockQC.ptr, crntRound, maxBlockEnergy, inputPool)
16: execute `data`
17: compute `transactionsOutcomesHash`
18: compute `stateHash`
19: **while** currentTime < blockQC.timestamp + minBlockTime **do**
20:     wait
21: timestamp ← currentTime
22: blockBody ← (crntRound, crntEpoch, timestamp, me, blockQC, roundTC, epochFinEntry, nonce, data, transactionsOutcomesHash, stateHash)
23: compute `blockHash` as the root of the hash tree from `blockBody`
24: σ^{BC} = Sign$_{\text{BC}}$(H(genesis)‖blockHash, me.sk$^{\text{BC}}$)
25: block ← (blockBody, blockHash, σ^{BC})
26: (Skov.blockPool, Skov.tree) = Skov.addBlock(block, Skov.blockPool, Skov.tree).
27: Multicast `block` over the network.
28: isFinalizer ← me ∈ Kontrol.getFinalizationCommittee(block.epoch, epochFinEntries)
29: **if** `isFinalizer` **then**
30:     validateBlock(block)

11.4.2 Validating Blocks

Once a block is received, all validators check it to see if it is valid, but only finalizers sign it if in addition to being valid it is having the correct round number.

The following procedure has three parts:

1. verifying that the block is valid,

2. learning new QCs, TCs, and epochFinEntries from the block that we may be missing, and advancing rounds when possible,

3. signing the block.

If we receive a block after having seen a QC for it, e.g., via a catch-up mechanism:
1. If we can first very its QC in the descendant block (which is always possible if the payday of the block is the same as the payday of `crntEpoch`), we do not need to verify the block for validity anymore, because we wouldn't get a valid QC if it weren't valid.
2. Learning the new QCs, etc, is only needed for the last valid block. So if we have a chain of blocks to catch-up and can verify the QCs on all of them, then it is sufficient to run this step on the one before the last (we could also first verify the last, and if it is correct, we run this step on the last, not the one before the last).
3. Signing the block is optional, since it already has enough signatures for a QC (assuming we verified that the QC is correct).

Protocol uponReceivingBlock

Inputs

   `block` $\in$ BLOCKS

       The block we just received.

Implementation

1: /* PART 1: verifying the validity of the block   */
2: /* We start by ignoring blocks from old epochs. We don't want to have to find old
     keys to check signatures.   */
3: **if** we have already seen a finalized block in an epoch $e > $ `block.epoch` **then**
4:     /* Either the block is invalid or it is in a branch that cannot be finalized   */
5:     ignore this block, terminate this procedure
6: /* Then we check if the signature is valid.   */
7: **if** we have the validator list for the payday of epoch `block.epoch` **then**
8:     **if** `block`.σ^{BC} is not valid **then**
9:        ignore this block, terminate this procedure
10: **else**
11:     /* We must be more than an epoch behind `block` to end up in this branch.   */
12:     start catch-up to get the missing info
13: /* Next we check if we have the parent block. It simplifies things to make this case
     distinction at the start. In particular, we might not be able to verify if the leader is
     correct with missing blocks. But this order isn't necessary. If we have the required
     information, we could do all checks needed up to retransmitting the block without
     verifying that we have the parent.   */
14: **if** we do not have `block.qc.ptr` **then**
15:     /* Alternatively, if the payday of `block.qc.ptr` is the same as the payday
     of `crntEpoch`, we first verify `block.qc`. If this succeeds, we mark that
     `block.qc.ptr` is valid and simplify the procedure once we get it from catch-up.
     See remark just before this procedure.   */
16:     mark `block` as pending, initiate catch-up
17:     terminate this procedure

18: /* Now we check if the new block is a fork in the past that can be ignored. This test
    is not needed for security, it only prevents processing a block that can never be
    finalized. So it can be ignored.   */
19: **if** the last finalized block is not in the chain to `block` **then**
20:     /* Remark: this could make us ignore perfectly valid blocks (but that can never be
    finalized unless there is an old key attack). So we could still receive signatures
    validating such a block. Maybe we could remember a block hash if it is ignored
    for this reason so that we don't try to find it again if we receive a signature. */
21:     ignore this block, terminate this procedure
22: /* Next we check if this is the correct leader. If `block.epoch > crntEpoch`, we first
    need to compute the new LE-nonce, which involves some sanity checks, which
    should all raise flags if they do not pass. If we are up-to-date with `block.qc.ptr`,
    at the minimum we check (i) that `block.epoch == block.qc.ptr.epoch + 1`, (ii)
    whether `block.epochFinEntry` finalizes a trigger block that is in this branch.
    Checking (iii) the signatures in `epochFinEntry` is possible, but is much more time
    consuming, and we want to keep the checks to the minimum before retransmit-
    ting a block. If we have not processed `block.qc.ptr`, then we might be missing
    information needed to compute the new LE-nonce.   */
23: **if** `block.vid` is not the correct leader for epoch `block.epoch` and round `block.round`
    then
24:     flag(block.vid, offense = "Not leader", severity = minor, witness = block)
25:     ignore this block, terminate this procedure
26: /* The final check before meeting the minimal requirements to retransmit a block is
    that the leader isn't sending multiple blocks.   */
27: **if** `block.vid` has already produced a different (valid) block this round **then**
28:     /* We refuse to retransmit or sign the new block, but if it is valid we store the
    block in case it gets enough signatures. We may consider storing the block
    without doing the remaining validity checks. If it gets enough signatures, then
    we do not need to check validity anymore, as others have done it.   */
29:     flag(block.vid, offense = "duplicate block", severity = security, witness =
    two blocks)
30:     duplicateBlock ← true   // To remember there are multiple blocks
31: **else**
32:     duplicateBlock ← false
33:     /* The minimal conditions for retransmitting a block are a valid signature, valid
    leader, first block from that leader and not an old key attack. Checking other
    header or body information is not necessary.   */
34:     /* Note that we might be retransmitting a block for round r while `crntRound` $< r$. */
35:     retransmit block
36: /* Next we do the easy checks.   */
37: /* Check that the QC is consistent   */
38: **if** `block.qc.round` $\neq$ `block.qc.ptr.round` **then**
39:     flag(block.vid, offense = "QC round inconsistent (block)", severity =
    network, witness = (block, block.qc.ptr))
40:     ignore this block, terminate this procedure
41: **if** `block.qc.epoch` $\neq$ `block.qc.ptr.epoch` **then**
42:     flag(block.vid, offense = "QC epoch inconsistent (block)", severity =
    network, witness = (block, block.qc.ptr))
43:     ignore this block, terminate this procedure
44: /* Check that the block round and epoch numbers are meaningful.   */

45: **if** block.qc.round $\geq$ block.round **then**
46:     flag(block.vid, offense = "Block round inconsistent", severity = network, witness = block,)
47:     ignore this block, terminate this procedure
48: **if** block.qc.epoch $>$ block.epoch or block.qc.epoch $<$ block.epoch $-$ 1 **then**
49:     flag(block.vid, offense = "Block epoch inconsistent", severity = network, witness = block,)
50:     ignore this block, terminate this procedure
51: */* Check that the timestamp isn't before the parent. Note that we aren't checking if the timestamp is in the future at this point, we just refuse to sign in this case.   */
52: **if**   block.timestamp $<$ block.qc.ptr.timestamp $+$ Kontrol.getMinBlockTime(block.timestamp, block.qc.ptr) **then**
53:     flag(block.vid, offense = "Too fast", severity = network, witness = (block, block.qc.ptr))
54:     ignore this block, terminate this procedure
55: */* Check that the block nonce is correctly formed.   */
56: **if** not verifyBlockNonce(block.round, nonce$^{\text{LE}}$, block.vid.vk$^{\text{LE}}$, block.nonce) **then**
57:     flag(block.vid, offense = "Bad nonce", severity = network, witness = block)
58:     ignore this block, terminate this procedure
59: */* Check that we have a TC if needed. Assign a bool to remember that this is a block with TC.   */
60: **if** block.round $>$ block.qc.round $+$ 1 **then**
61:     timeoutRound $\leftarrow$ true
62:     **if** block.tc $==$ $\bot$ **then**
63:         flag(block.vid, offense = "TC missing", severity = network, witness = block)
64:         ignore this block, terminate this procedure
65: **else**
66:     timeoutRound $\leftarrow$ false
67:     **if** block.tc $\neq$ $\bot$ **then**
68:         flag(block.vid, offense = "TC not empty", severity = network, witness = block)
69:         ignore this block, terminate this procedure
70: */* Check that the TC round is meaningful   */
71: **if** timeoutRound and block.tc.round $\neq$ block.round $-$ 1 **then**
72:     flag(block.vid, offense = "TC round inconsistent", severity = network, witness = block)
73:     ignore this block, terminate this procedure
74: */* Check that we have an epoch finalization if needed. Assign a bool to remember that this starts a new epoch.   */
75: **if** block.epoch $==$ block.qc.epoch $+$ 1 **then**
76:     newEpoch $\leftarrow$ true
77:     **if** block.epochFinEntry $==$ $\bot$ **then**
78:         flag(block.vid, offense = "Epoch finalization missing", severity = network, witness = block)
79:         ignore this block, terminate this procedure
80:     **if** block.epoch is the last epoch of the payday **then**
81:         take the validator snapshot for the next payday // *Needed to verify* stateHash
82: **else**
83:     newEpoch $\leftarrow$ false

84:     **if** block.epochFinEntry $\neq \perp$ **then**

85:        flag(block.vid, offense = "Epoch finalization not empty", severity = network, witness = block)

86:        ignore this block, terminate this procedure

87: /* Now we check the validity of the QC signatures   */

88: **if** block.qc.$\sigma^{\texttt{AggFin}}$ is not valid or it doesn't have the sigThreshold weight required **then**

89:     flag(block.vid, offense = "Invalid QC (block)", severity = major, witness = block)

90:     ignore this block, terminate this procedure

91: /* And the validity of the TC signatures   */

92: **if** timeoutRound and (block.tc.$\sigma^{\texttt{AggFin}}$ is not valid or doesn't have the sigThreshold weight required according to the stake of the payday of block.qc.epoch) **then**

93:     flag(block.vid, offense = "Invalid TC (block)", severity = major, witness = block)

94:     ignore this block, terminate this procedure

95: **if** timeoutRound **then**

96:     /* Check that the QC round is correct with respect to the TC, i.e., it is at least as good as the best round in the TC and the TC does not cover more than 2 epochs.   */

97:     let $r_{\max}$ be the largest round in block.tc.qcrounds

98:     let $e_{\max}$ be the largest epoch in block.tc.qcrounds

99:     let $e_{\min}$ be the smallest epoch in block.tc.qcrounds

100:     **if** block.qc.round $< r_{\max}$ or block.qc.epoch $< e_{\max}$ **then**

101:        flag(block.vid, offense = "QC inconsistent with TC", severity = major, witness = block)

102:        ignore this block, terminate this procedure

103:     **if** $e_{\max} - e_{\min} > 2$ **then**

104:        flag(block.vid, offense = "TC spans too many epochs", severity = major, witness = block)

105:        ignore this block, terminate this procedure

106: /* And finally the validity of the epoch finalization signatures   */

107: **if** newEpoch and block.epochFinEntry is not valid **then**

108:     /* See Definition 62 in Section 11.1.4 for the validity of epochFinEntry   */

109:     flag(block.vid, offense = "Epoch finalization invalid (block)", severity = major, witness = block)

110:     ignore this block, terminate this procedure

111: /* Now the checks that involve the data in the block.   */

112: **if** something wrong with block.data **then**

113:     flag(block.vid, offense = "Bad data", severity = major, witness = block)

114:     ignore this block, terminate this procedure

115: /* We have checked everything that can be done without executing the transactions. */

116: execute the transactions

117: /* Now we check that outcomes, state and block hashes are correct   */

118: **if** transactionsOutcomesHash is not what we expect **then**

119:     flag(block.vid, offense = "Bad transactions outcomes", severity = major, witness = block and parent block)

120:     ignore this block, terminate this procedure

121: **if** stateHash is not what we expect **then**

122:     flag(block.vid, offense = "Bad state hash", severity = major, witness =
        block and parent block)
123:     ignore this block, terminate this procedure
124: **if** blockHash is not what we expect **then**
125:     flag(block.vid, offense = "Bad block hash", severity = major, witness =
        block)
126:     ignore this block, terminate this procedure
127: */* The block is good, we can add it to the tree. Though as said above, we might not
        want to add duplicate blocks to the tree unless they get (enough) signatures.   */
128: (Skov.blockPool, Skov.tree) ← Skov.addBlock(block, Skov.blockPool, Skov.tree)
129: /* ********** */
130: /* PART 2: learning from any QCs, TCs or finaliztion proofs that we may not have
        yet, and advancing rounds and epoch when possible.   */
131: /* We start with the data from block.epochFinEntry   */
132: **if** newEpoch **then**
133:     (qc_1, qc_2, str) ← block.epochFinEntry
134:     /* Updating the list of final blocks   */
135:     **if** qc_1.ptr isn't already in Skov.finEntries **then**
136:         Add qc_1.ptr to Skov.finEntries.
137:         currentTimeout ← max{timeoutDecrease ∗ currentTimeout, timeoutBase}
138:     /* Updating the highest QC   */
139:     **if** highestQC.round < qc_2.round **then**
140:         highestQC ← qc_2
141:     /* Updating the epoch   */
142:     **if** crntEpoch < block.epoch **then**
143:         advanceEpoch(block.epochFinEntry, block.epoch)
144: /* Now the data from block.qc   */
145: /* Updating the list of final blocks   */
146: checkFinality(block.qc)
147: /* Updating the highest QC   */
148: **if** highestQC.round < block.qc.round **then**
149:     highestQC ← block.qc
150: /* Finally, we advance rounds if needed   */
151: **if** crntRound < block.round **then**
152:     advanceRound(block.round, timeoutRound, block.qc, block.tc)
153: /* ********** */
154: /* PART 3: we sign the block if we are a finalizer, it isn't a duplicate, and it isn't in
        the future   */
155: isFinalizer ← me ∈ Kontrol.getFinalizationCommittee(block.epoch, epochFinEntries)
156: **if** isFinalizer AND not duplicateBlock **then**
157:     **while** block.timestamp > currentTime **do**
158:         wait
159:     validateBlock(block)

Protocol validateBlock

Inputs

block ∈ BLOCKS
        The block to be signed.

Promise

The block has been verified and I am a finalizer.

Implementation

1: **if** block.round == crntRound AND nextSignableRound ≤ block.round AND block.epoch == crntEpoch **then**
2:     $\sigma^{\texttt{AggFin}} \leftarrow \mathsf{Sign}_{\texttt{AggFin}}(\mathsf{H}(\text{genesis})\|\mathsf{H}(\text{block})\|\text{block.round}\|\text{block.epoch}, \text{me.sk}^{\texttt{AggFin}})$
3:     quorumSignature ← (me, H(block), block.round, block.epoch, $\sigma^{\texttt{AggFin}}$)
4:     nextSignableRound ← block.round + 1
5:     Multicast quorumSignature over the network.
6:     processQuorumSignature(quorumSignature)

ℹ️ The condition block.round == crntRound in validateBlock is not needed for security: there is no harm in signing blocks from previous rounds as long as nextSignableRound has been correctly updated. But we do not want to clog the network by adding more signatures to past rounds.

11.4.3 Processing Validated Blocks

Signatures are stored and if they have enough weight, a QC is formed, as described below. The first procedure, uponReceivingQuorumSignature, checks that the quorum signature is valid.

Protocol uponReceivingQuorumSignature

Inputs

quorumSignature ∈ QUORUMSIGNATURETYPE
    The signature validating a block.

Implementation

1: **if** quorumSignature.round < crntRound and quorumSignature.epoch ≤ crntEpoch **then**
2:     */* We aren't interested in signatures from past rounds, unless they are from a future epoch, in which case we have missed a crucial block and need to catch-up */*
3:     ignore this signature, terminate this procedure
4: */* Checking validity of the signature, the signer and the block   */*
5: **if** we have the finalizer list for the payday of epoch quorumSignature.epoch **then**
6:     **if** $\sigma^{\texttt{AggFin}}$ is not valid **then**
7:         ignore this signature, terminate this procedure
8: **else**
9:     initiate catch-up to get the missing information
10:     restart this procedure once we have it
11: **if** Not fid ∈ Kontrol.getFinalizationCommittee(quorumSignature.epoch, epochFinEntries) **then**
12:     flag(fid, offense = "Not a finalizer (sig)", severity = minor, witness = fid and ptr)
13:     ignore this signature, terminate this procedure
14: **if** we do not have the block corresponding to quorumSignature.ptr **then**
15:     initiate catch-up to get quorumSignature.ptr

16: └─ restart this procedure from the start once we have the block
17: **if** ptr is not a valid block **then**
18: │ */* If* ptr *is not a valid block, we probably aren't even executing this code. This*
│   *is more of a reminder that if we can, we should flag people who sign invalid*
│   *blocks.*   */
19: │ flag(fid, offense = "Signed invalid block"), severity = major, witness =
│   block and $\sigma^{\texttt{AggFin}}$)
20: └─ ignore this signature, terminate this procedure
21: */* Security check: no double signing*   */
22: **if** fid already signed a different block in quorumSignature.round **then**
23: │ flag(fid, offense = "Signed two blocks", severity = security, witness =
│   the two quorumSignatures)
24: │ */* We continue processing the signature, because maybe this block will end up in*
│   *the chain, but we will not retransmit it.*   */
25: **else**
26: └─ retransmit quorumSignature
27: */* We have done the minimal checks before retransmitting the* quorumSignature,
│   *now the last checks to be sure that it is valid.*   */
28: **if** quorumSignature.round ≠ quorumSignature.ptr.round **then**
29: │ flag(fid, offense = "Round inconsistent (sig)", severity = network, witness =
│   quorumSignature and quorumSignature.ptr)
30: └─ ignore this signature, terminate this procedure
31: **if** quorumSignature.epoch ≠ quorumSignature.ptr.epoch **then**
32: │ flag(fid, offense = "Epoch inconsistent (sig)", severity = network, witness =
│   quorumSignature and quorumSignature.ptr)
33: └─ ignore this signature, terminate this procedure
34: */* We have checked that everything is valid, and continue processing the signature.* */
35: processQuorumSignature(quorumSignature)

The process uponReceivingQuorumSignature should ideally have runtime $\mathcal{O}(1)$—possibly $O(\log n)$ if it takes that amount of steps to find whether a validator is double signing—because this function may be called n times per round when there are n validators.

The second procedure, processQuorumSignature, stores the signature and checks if we can build a QC.

Protocol processQuorumSignature

Inputs

quorumSignature ∈ QUORUMSIGNATURETYPE
    The signature validating a block.

Promise

    Only called if the signature and the block are valid, and quorumSignature.round == crntRound.

Implementation

1: */* We store all relevant signatures*   */

```
2: store quorumSignature
3: /* If we have enough signatures for a QC, we build this and advance rounds        */
4: if the weight of signatures for quorumSignature.ptr is ≥ sigThreshold then
5:     highestQC ← makeQC(quorumSignature.ptr)
6:     /* Finalize block and advance epoch if possible                                */
7:     checkFinality(highestQC)
8:     /* We have a QC for crntRound, so we advance rounds                            */
9:     advanceRound(quorumSignature.round + 1, false, ⊥, ⊥)
```

</> Same as above, processQuorumSignature should ideally have runtime $\mathcal{O}(1)$—possibly $O(\log n)$ if it takes that amount of steps to add the new quorumSignature to the data structure—because this function may be called n times per round when there are n validators.

The third procedure, makeQC, builds the QC.

Function makeQC

Inputs

ptr ∈ HASH

      The hash of the block for which we want the QC.

Implementation

```
1: σ^AggFin ← Aggregation_AggFin(all the signatures we have gathered for ptr)
2: signers ← the list of fid that signed ptr
3: return (ptr, ptr.round, ptr.epoch, σ^AggFin, signers)
```

11.4.4 Throwing Timeouts

If a timeout occurs, finalizers will generate and send a corresponding message, as specified below.

Protocol uponTimeoutEvent

Promise

      This function is triggered when we have not progressed to the next round after currentTimeout. Only finalizers start the timer and react to this.

Implementation

```
1: /* After throwing a timeout, we refuse to sign for this round. This is a safety issue! */
2: nextSignableRound ← crntRound + 1
3: currentTimeout ← timeoutIncrease * currentTimeout   // Increase timeout time.
4: /* The aggregate signature on our best QC round and epoch.                          */
5: σ^AggFin ← Sign_AggFin((H(genesis), crntRound, highestQC.round, highestQC.epoch), me.sk^AggFin)
6: /* Signing the entire timeout                                                       */
7: timoutBody ← (me, crntRound, crntEpoch, highestQC, σ^AggFin)
8: σ^BC ← Sign_BC(H(genesis)‖timoutBody, me.sk^BC)
9: /* To shorten witnesses we could hash some of the above before signing             */
```

```
10:  /* We send then process our own timeout                                              */
11:  timeout ← (timoutBody, σ^BC)
12:  Multicast timeout
13:  processTimeout(timeout)
```

11.4.5 Processing Timeouts

Once we receive a `timeout`, we need to first check if it is valid, then add it to our list of valid timeouts while waiting to have enough to form a TC. Definition 55 distinguishes between a `timeout` being partially and completely valid, i.e., where we check everything except the validity of `timeout.qc` and where the QC is verified as well. The reasons for not systematically verifying the QC, is that this takes $\mathcal{O}(n)$ steps (where n is the number of finalizers) and is performed $\mathcal{O}(n)$ times, making the complexity per round $O(n^2)$. Note that the complexity is always $O(n^2)$ when dishonest parties are present, because they can force us to verify all their QCs. But if only honest people run the protocol, it can be brought down to $O(n \log n)$.

> One can consider 4 different levels of checks on the QCs in the timeouts.
> 1. We systematically check all QCs. This has complexity $O(n^2)$ even when everyone is honest.
> 2. We check the QC if we haven't already accepted a timeout with a QC that has the same round number (`timeout.qc.round`). This has worst case complexity $O(n^2)$ even when everyone is honest, because in theory it could be that honest parties have $\mathcal{O}(n)$ different highest QCs. But in practice this won't happen, all honest parties should have the same highest QC, and the complexity will be $\mathcal{O}(n)$ (or $O(n \log n)$ if it takes $O(\log n)$ to store and retrieve timeouts from the chosen data structure).
> 3. We store globally whenever we check a QC for some round, and when verifying timeouts, we only check QCs that haven't been checked at any point by any process before. This has complexity per round of $\mathcal{O}(n)$ (or $O(n \log n)$ if it takes $O(\log n)$ to store and retrieve timeouts from the chosen data structure).
> 4. We only check QCs if `timeout.qc.round > highestQC.round`. These are the minimal checks that are needed for the protocol to be secure. But it is too few checks to guarantee that for any two timeouts with round-epoch pairs (r_1, e_1), (r_2, e_2) that pass the partial validity check we have $r_2 \geq r_1 \implies e_2 \geq e_1$. If this property does not hold, then additional verification are needed in a few place and sorting stored timeouts by round does not sort them by epoch, making finding minimums and maximums a little more complex.
>
> The pseudo-code below describes the version 2.

The following algorithms explain how to process the received timeouts and generate a TC if we have enough timeouts. We have divided **uponReceivingTimeout** in four parts. (i) checking the signature and catching up. (ii) minor validity checks after which the timeout is partially valid. (iii) checking the QC, in particular, this part contains the condition under which we check the QC. (iv) processing the timeout that has past verification.

Protocol uponReceivingTimeout

Inputs

---

timeout $\in$ TIMEOUTMESSAGES
    The timeout message received.

Implementation

1: */* PART 1: signature check and catch-ups   */*
2: */* First we check if we want and can process this timeout   */*
3: */* We aren't interested in timeouts on past rounds   */*
4: **if** timeout.round < crntRound **then**
5:   | ignore this timeout, terminate this procedure
6: */* We aren't interested in timeouts that point too far back. Timeouts pointing too
    far back must have been generated by a dishonest party, so can be ignored without
    affecting liveness even if they are valid. We exclude them already here, because
    we might not even have the keys in memory to check their validity.   */*
7: */* The test $r <$ timeout.round $- 1$ below is to exclude the case where the block in
    round timeout.round $- 1$ got finalized by a QC in round timeout.round without
    the sender of the timeout even seeing the QC in round timeout.round $- 1$. By
    "directly finalized block", we mean a block that is finalized because itself and its
    consecutive direct descendant both have a QC. Normally, if we see a QC in round
    timeout.round we would already move on to the next round and not process a
    timeout message in this round anymore. But we include this test to cover the case
    of processes running in parallel, where things might happen in different orders
    than expected.   */*
8: let (r, e) be the round and epoch of the last directly finalized block with $r <$
    timeout.round $- 1$
9: **if** timeout.qc.round < r or timeout.qc.epoch < e **then**
10:   | */* The timeout points too far in the past, but we aren't flagging it, because we
      | would need more work to generate a proof that is wrong (we haven't even
      | checked the signature yet).   */*
11:   | ignore this timeout, terminate this procedure
12: */* Next we verify the signature on the timeout (if we can)   */*
13: **if** we have the finalizer list of the payday of timeout.qc.epoch **then**
14:   | **if** timeout.σ^{BC} is not valid **then**
15:   | | ignore the timeout, terminate this procedure
16: **else**
17:   | initiate catch-up to get the missing information
18:   | restart this procedure once we have it
19: */* If we aren't in round timeout.round, but timeout.qc.round $==$ timeout.round$-$
    1 then we can catch-up from that QC and do not need to call catch-up explicitly.
    But if we do not have a QC for round timeout.round $- 1$—e.g., that round ended
    with a TC—then we do call catch-up.   */*
20: **if** timeout.round > crntRound and timeout.qc.round < timeout.round $- 1$ **then**
21:   | initiate catch-up to get the missing information
22:   | resume this procedure from the start once we have it
23: */* Another scenario where catch-up is needed is if we will verify timeout.qc in detail
    but do not have timeout.qc.ptr. This condition will change if we change our
    rule for verifying QCs. Note that if we do not intend to verify the QC, we do not
    need the block it points to.   */*
24: **if** we do not already have a timeout entry for timeout.qc.round and we do not know
    timeout.qc.ptr **then**
25:   | initiate catch-up to get the missing information

26:     resume this procedure from the start once we have it

27: */* The final case where catch-up is needed is if the sender is an epoch ahead of us and* `timeout.qc` *won't allow us to advance epochs. We break this up into multiple if statements so that each one can be explained.* */

28: **if** `timeout.epoch` $>$ `crntEpoch` **then**

29:     **if** we already have a timeout entry for `timeout.qc.round` **then**

30:         */* In this branch we know that we can't learn the* `epochFinEntry` *from* `timeout.qc`, *because we already have this QC.* */

31:         initiate catch-up to get the missing epoch finalization proof

32:         resume this procedure from the start once we have it

33:     **else**

34:         */* Due to the a test a little higher we know that we have* `timeout.qc.ptr`, *so we can check to see if this would allow us to finalize the trigger block of* `crntEpoch` */

35:         let $qc_1 \leftarrow$ `timeout.qc.ptr.qc`

36:         let $qc_2 \leftarrow$ `timeout.qc`

37:         **if** qc_1`.ptr` is not after the nominal end of `crntEpoch` or qc_2`.round` $\neq$ qc_1`.round`$+1$ **then**

38:             */* Note that we did not test that* qc_1 *and* qc_2 *are in the same epoch. If* qc_1`.ptr` *is after the nominal end of* `crntEpoch` *and* qc_2 *is in a different epoch, then* qc_2`.epoch` $>$ `crntEpoch` *and we would have incremented* `crntEpoch` *from knowing* qc_2`.ptr`. */

39:             initiate catch-up to get the missing epoch finalization proof

40:             resume this procedure from the start once we have it

41: */* *********** */

42: */* PART 2: minor checks* */

43: */* We have all information we need to proceed and have checked the signature, now we perform the minor checks.* */

44: */* Check that the sender is a finalizer* */

45: **if** not `fid` $\in$ `Kontrol.getFinalizationCommittee(timeout.qc.epoch, Skov.epochFinEntries)` **then**

46:     `flag(fid, offense = `"Not a finalizer (timeout)"`, severity = minor, witness = timeout)`

47:     ignore this signature, terminate this procedure

48: */* Check that there are not many timeouts from the same person* */

49: **if** the same person already sent a different timeout this round **then**

50:     `flag(fid, offense = `"Multiple timeouts"`, severity = minor, witness = ` 2 timeouts`)`

51:     ignore this timeout, terminate this procedure

52: */* We have done the minimal checks needed before retransmitting. So we do this, then continue with other basic validity checks.* */

53: */* Note that we might be retransmitting a timeout for round* r *while* `crntRound` $< r$. */

54: retransmit the timeout

55: */* Verify* `timeout.`$\sigma^{\texttt{AggFin}}$ */

56: **if** `timeout.`$\sigma^{\texttt{AggFin}}$ is a not valid aggregate signature by `timeout.fid` on $\big($`H(genesis), timeout.round, timeout.qc.round, timeout.qc.epoch`$\big)$ **then**

57:     `flag(fid, offense = `"Invalid timeout sig"`, severity = network, witness = timeout)`

58:     ignore this signature, terminate this procedure

59: */* Check that the QC round is meaningful.* */

```
60: if timeout.qc.round ≥ timeout.round then
61:     flag(timeout.fid, offense = "QC round incoherent", severity =
        network, witness = timeout, )
62:     ignore this block, terminate this procedure
63: /* The current version assumes that we check every QC on a new round num-
        ber.  If we do not do this, we need to additionally check that we do
        not have timeout.qc.round ≤ highestQC.round AND timeout.qc.epoch >
        highestQC.epoch.  If that were the case, we could throw the same error as
        above, namely "QC round incohrent".                                    */
64: /* **********                                                              */
65: /* PART 3: checking the QC.                                                */
66: /* In the following we do not systematically check if the QC is valid, we only do
        so if the QC is on a round that we haven't checked before.  There's a discussion
        at the beginning of this subsection about alternative ways of deciding which QCs
        should be checked.                                                     */
67: if timeout.qc.round > highestQC.round then
68:     if timeout.qc is not valid then
69:         flag(timeout.fid, offense = "Invalid QC (timeout)", severity =
            major, witness = timeout)
70:         ignore this message, terminate this procedure
71:     /* Check if this finalizes a block (advance epoch is called by checkFinality if needed) */
72:     checkFinality(timeout.qc)
73:     /* Update highestQC whenever we find a better QC.                      */
74:     highestQC ← timeout.qc
75:     /* And advance round if possible                                       */
76:     if crntRound ≤ timeout.qc.round then
77:         /* It must be that timeout.qc.round == timeout.round − 1, otherwise we
            wouldn't be in this branch.                                        */
78:         advanceRound(timeout.qc.round + 1, false, ⊥, ⊥)
79: else
80:     if we have already checked a QC qc' this round with qc'.round == timeout.qc.round
        then
81:         if qc'.epoch ≠ timeout.qc.epoch then
82:             flag(timeout.fid, offense = "Invalid QC (timeout)", severity =
                major, witness = timeout)
83:             ignore this message, terminate this procedure
84:     else
85:         if timeout.qc is not valid then
86:             flag(timeout.fid, offense = "Invalid QC (timeout)", severity =
                major, witness = timeout)
87:             ignore this message, terminate this procedure
88:         /* Check if this finalizes a block                                 */
89:         checkFinality(timeout.qc)
90: /* **********                                                              */
91: /* PART 4: the timeout has passed all the checks. We now add it to our collection
        of partially valid timeouts for this round.                           */
92: processTimeout(timeout)
```

The process processTimeout stores the timeout that has passed the validity checks, potentially purges timeouts that turn out to be from dishonest parties, and verifies if the conditions are met

to build a TC.

Protocol processTimeout

Inputs

  timeout $\in$ TIMEOUTMESSAGES
      The timeout message received.

Implementation

1: store timeout locally
2: **if** the epochs of all stored timeouts span strictly more than 2 epochs **then**
3: ⎿ delete all the timeouts with smallest epoch, they are from dishonest people.
4: **if** the weight of timeouts for round crntRound is $>$ sigThreshold with weights from the payday of highestQC.epoch **then**
5: ⎿ advanceRound(crntRound $+ 1$, true, highestQC, makeTC(crntRound))

Function makeTC

Inputs

  round $\in \mathbb{N}$
      The round for which we make the TC.

Implementation

1: qcrounds $\leftarrow$ make the list of all (timeout.qc.round, timeout.qc.epoch) for every timeout stored for round
2: signers $\leftarrow$ make the list of all timeout.fid for every timeout stored for round
3: */* There must be a way of matching the messages in qcrounds to the IDs in signers. */*
4: $\sigma^{\texttt{AggFin}} \leftarrow \mathsf{Aggregation}_{\texttt{AggFin}}$ (all timeout.$\sigma^{\texttt{AggFin}}$ for every timeout stored for round)
5: **return** (round, qcrounds, $\sigma^{\texttt{AggFin}}$, signers)

11.4.6 Advancing Rounds, Finalizing blocks and Transitioning Epochs

There are two conditions which allow us to advance to the next round:

  1. getting enough signatures on the block of the current round to form a QC,

  2. getting enough timeouts in the current round to form a TC.

When either of these conditions is met, the advanceRound procedure is called, which will query getLeader and make the next block if we are elected.

Protocol advanceRound

Inputs

  round $\in \mathbb{N}$
      The new round number to which we jump.

  timeoutRound $\in$ BOOL
      Whether we advance with a TC or a QC from round round $- 1$

$\texttt{qc} \in \text{QUORUMCERTIFICATES} \cup \{\bot\}$
  A QC for which $\texttt{tc}$ is valid, only needed if $\texttt{timeoutRound} = \texttt{true}$.
$\texttt{tc} \in \text{TIMEOUTCERTIFICATES} \cup \{\bot\}$
  A TC, only needed if $\texttt{timeoutRound} = \texttt{true}$

Promise

 highestQC, crntEpoch and lastEpochFinEntry have been correctly updated each time we see a better QC or see an epoch finalization proof. If $\texttt{timeoutRound} = \texttt{false}$, then $\texttt{highestQC.round} = \texttt{round} - 1$. If $\texttt{timeoutRound} = \texttt{true}$, then $\texttt{tc} \neq \bot$, $\texttt{qc} \neq \bot$, $\texttt{tc.round} = \texttt{round} - 1$, and $\texttt{tc}$ is valid with respect to $\texttt{qc}$ (i.e., the weights are sufficient with the stake of $\texttt{qc.epoch}$ and $\texttt{qc.round}$ is at least as good as the highest round number in $\texttt{tc}$).

Implementation

1: $\texttt{crntRound} \leftarrow \texttt{round}$
2: **if** $\texttt{me} \in \textsf{Kontrol.getFinalizationCommittee}(\texttt{crntEpoch}, \texttt{epochFinEntries})$ **then**
3:     /* *Only finalizers send timeouts*   */
4:     $\textsf{resetTimer}(\texttt{currentTimeout})$ // *Restart the countdown from* $\texttt{currentTimeout}$
5: /* *The information here below is assigned to variables as it is required by both* $\textsf{makeBlock}$ *and* $\textsf{uponTimeoutEvent}$.   */
6: **if** $\texttt{timeoutRound}$ **then**
7:     $\texttt{roundType} \leftarrow$ "TC"
8:     $\texttt{roundTC} \leftarrow \texttt{tc}$
9:     $\texttt{roundTCsQC} \leftarrow \texttt{qc}$
10: **else**
11:     $\texttt{roundType} \leftarrow$ "QC"
12:     $\texttt{roundTC} \leftarrow \bot$
13:     $\texttt{roundTCsQC} \leftarrow \bot$
14: **if** $\textsf{getLeader}(\texttt{crntRound}, \texttt{crntEpoch}, \texttt{epochFinEntries}) == \texttt{me}$ **then**
15:     $\textsf{makeBlock}$

When we get a new QC, we call $\textsf{checkFinality}$ to check if it finalizes a block—namely, if parent and child have consecutive round numbers and same epoch numbers—or advances an epoch—if it finalizes a trigger block.

Protocol checkFinality

Inputs

$\texttt{qc} \in \text{QUORUMCERTIFICATES}$
  A new QC.

Promise

 $\texttt{qc}$ is valid and we have access to $\texttt{qc.ptr}$ and its parent.

Implementation

1: **if** $\texttt{qc.round} == \texttt{qc.ptr.qc.round} + 1$ and $\texttt{qc.epoch} == \texttt{qc.ptr.qc.epoch}$ **then**
2:     /* *Two consecutive QCs in the same epoch.*   */
3:     **if** $\textsf{Skov.finEntries}$ does not already contain an entry for $\texttt{qc.ptr.qc.ptr}$ **then**

```
4:        finEntry ← (qc.ptr.qc, qc, hash of what is needed from the block qc.ptr to
          satisfy Definition 59)
5:        add finEntry to Skov.finEntries
6:        currentTimeout ← max{timeoutDecrease * currentTimeout, timeoutBase}
7:        if crntEpoch ≤ qc.ptr.qc.epoch and qc.ptr.qc.ptr.timestamp is after the nom-
          inal end of the epoch qc.ptr.qc.epoch then
8:           advanceEpoch(finEntry, qc.ptr.qc.epoch + 1)
```

Advancing epoch consists in storing the epoch finalization proof, increasing the epoch counter and computing the new nonce^{LE}.

Protocol advanceEpoch

Inputs

epochFinEntry ∈ EPOCHFINALENTRIES
    The finalization proof for the previous epoch.

epoch ∈ ℕ
    The new epoch.

Implementation

1: lastEpochFinEntry ← epochFinEntry
2: Add epochFinEntry to epochFinEntries
3: crntEpoch ← epoch
4: currentLENonce = getLENonce(epoch, epochFinEntries)

11.5   Epoch transitions, joining and leaving the finalization committee

Keeping old keys.   When we see a finalized block in some epoch e, we should normally not need the keys from epochs $e' < e$ anymore. But a dishonest person could send a timeout requiring a key from epoch $e - 1$, and the person building the TC does not notice—they might not have seen the finalized block in epoch e and still think that timeouts pointing to $e - 1$ are fine. If this is the case, an honest person could create a TC requiring keys from epoch $e - 1$. We should thus always keep the keys and finalizer list from epoch $e - 1$ until we see a finalized block in epoch $e + 1$. At this point a dishonest timeout using keys from epoch $e - 1$ would be caught with complexity $\mathcal{O}(1)$ by the check that a TC doesn't cover more than 2 epochs.

11.6   Protocol Updates

❗ A protocol update (PU) also happens after a trigger block that has to be finalized. But in this case a finalization proof for the PU-trigger block would additionally have to show that $qc_1.\text{ptr}$ is a descendant of the trigger block. To do this, a list of partial hashes of the blocks between the PU-trigger block and $qc_1.\text{ptr}$ is needed. See Section 11.6 for more information on protocol updates.

11.7   Optimizations

In this section we discuss various optimizations that could be done. Those in Section 11.7.1 deal with optimistically executing certain parts of the consensus protocol while waiting for confirmation that one may proceed. We believe that these will significantly increase TPS and decrease latency. In Section 11.7.2 we discuss some other ideas which may or may not be relevant.

11.7.1   Parallelization

- leader: start generating next block before receiving signatures for previous block (i.e., before advancing the round as defined in the pseudo/code).

- leader: send block transactions before completing block

- finalizer: start executing these transactions before being able to verify the entire blocks

11.7.2   Various

- Each timeout contains $\mathcal{O}(n)$ signatures and each party receives $\mathcal{O}(n)$ timeouts, where n is the number of parties. A non-optimized implementation would give complexity $\mathcal{O}(n^2)$ for each party by verifying all signatures. But if we only verify signatures when they are relevant, we should get $\mathcal{O}(n)$ complexity.

- Do we need to send transactions with blocks, or can we count on them being distributed in parallel?

- If $w > \max\{2\texttt{sigThreshold} - 1, 1 - \texttt{sigThreshold}\}$ have already signed a block, we do not need to verify that the block is valid, we just need to check the other conditions before signing.

11.8   Flagging errors

At this point we only flag dishonest behavior. When an error is detected, the function flag is called. It takes four inputs.

validator ∈ VALIDATORS
> The ID of the offending party.

offense ∈ $\{0,1\}^*$
> A description of the offense.

severity ∈ $\{0,1\}^*$
> We currently have three types of offenses.

> > minor   An error which is detected before retransmitting and is caught with an $\mathcal{O}(1)$ check.

> > network   An error which is detected after retransmitting and is caught with an $\mathcal{O}(1)$ check.

> > major   An error detected after retransmitting and is caught with $\mathcal{O}(n)$ checks—all checks with complexity $\mathcal{O}(n)$ are done after retransmitting to not slow block propagation.

> > security   An error which could break the security of the protocol if the adversary controls more than 1/3 stake and can schedule messages arbitrarily.

witness ∈ $\{0,1\}^*$
> A witness of the offense, e.g., a block.

The errors are summarized in the following tables.

Table 11.1 Errors in blocks that are detected before retransmitting the block.

offense	description	severity	witness
Not leader	A block is generated by someone who is not the leader for the corresponding round and epoch	minor: checking the signature and the leader catches it before retransmitting	block.vid, block.σ^{BC}, block.round, block.epoch, hash of the rest of the block.
Duplicate block	The same leader produced two blocks in the same round.	security: if the adversary controls $> 1/3$ stake and can time messages at will, they can fork the chain	block$_1$.vid == block$_2$.vid, block$_1$.round == block$_2$.round, block$_1$.σ^{BC}, block$_2$.σ^{BC}, hash of the rest of both blocks.

Table 11.2 Errors in blocks that are detected in time $\mathcal{O}(1)$, but only after retransmitting the block.

offense	description	severity	witness
QC round inconsistent (block)	The declared QC round doesn't match the actual QC round.	network: it takes $\mathcal{O}(1)$ to verify, but we already retransmitted the block	block.vid, block.σ^{BC}, block.qc.round, block.qc.ptr, hash of the rest of block, block.qc.ptr.vid, block.qc.ptr.σ^{BC}, block.qc.ptr.round, hash of the rest of block.qc.ptr.
QC epoch inconsistent (block)	The declared QC epoch doesn't match the actual QC epoch.	network: it takes $\mathcal{O}(1)$ to verify, but we already retransmitted the block	block.vid, block.σ^{BC}, block.qc.epoch, block.qc.ptr, hash of the rest of block, block.qc.ptr.vid, block.qc.ptr.σ^{BC}, block.qc.ptr.epoch, hash of the rest of block.qc.ptr.
Block round inconsistent	The block round is before the parent.	network: it takes $\mathcal{O}(1)$ to verify, but we already retransmitted the block	block.vid, block.σ^{BC}, block.round, block.qc.round, hash of the rest of the block.

Table 11.2 Continued. $\mathcal{O}(1)$-detectable block errors.

offense	description	severity	witness
Block epoch inconsistent	The block epoch is not equal or one more than the parent.	`network`: it takes $\mathcal{O}(1)$ to verify, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.epoch`, `block.qc.epoch`, hash of the rest of the block.
Too fast	Timestamp difference between this block and parent block is too small	`network`: it takes $\mathcal{O}(1)$ to verify the timestamp, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.timestamp`, `block.qc.ptr`, hash of the rest of `block`, `block.qc.ptr.vid`, `block.qc.ptr.`$\sigma^{\mathtt{BC}}$, `block.qc.ptr.timestamp`, hash of the rest of `block.qc.ptr`.
Bad nonce	The block nonce is badly formed	`network`: it takes $\mathcal{O}(1)$ to verify, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.round`, `block.nonce`, hash of the rest of the block.
TC missing	The block round is not consecutive, but there is no TC	`network`: it takes $\mathcal{O}(1)$ to verify the presence of a TC, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.round`, `block.qc.round`, `block.tc`, hash of the rest of the block.
TC not empty	The block round is consecutive, but there is a TC	`network`: it takes $\mathcal{O}(1)$ to verify the presence of a TC, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.round`, `block.qc.round`, `block.tc`, hash of the rest of the block.
TC round inconsistent	The TC is not from the previous round.	`network`: it takes $\mathcal{O}(1)$ to verify, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.round`, `block.tc.round`, hash of the rest of the block.
Epoch finalization missing	The block is the first in a new epoch, but does not have the finalization proof	`network`: it takes $\mathcal{O}(1)$ to verify the presence of a proof, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.epoch`, `block.qc.epoch`, `block.epochFinEntry`, hash of the rest of the block.
Epoch finalization not empty	The block is not the first in a new epoch, but it has a finalization proof	`network`: it takes $\mathcal{O}(1)$ to verify the presence of a proof, but we already retransmitted the block	`block.vid`, `block.`$\sigma^{\mathtt{BC}}$, `block.epoch`, `block.qc.epoch`, `block.epochFinEntry`, hash of the rest of the block.

Table 11.3 Errors in blocks that take time $\mathcal{O}(n)$ to detect. These are always checked after retransmitting the block to not slow block propagation.

offense	description	severity	witness
Invalid QC (block)	The QC in a block is invalid	major: it takes $\mathcal{O}(n)$ to verify a QC, and this is done after retransmitting the block.	`block.vid`, `block.`σ^{BC}, `block.qc`, hash of the rest of the block.
Invalid TC (block)	The TC in a block is invalid	major: it takes $\mathcal{O}(n)$ to verify a TC, and this is done after retransmitting the block.	`block.vid`, `block.`σ^{BC}, `block.tc`, hash of the rest of the block.
QC inconsistent with TC	The QC has to be at least as good as the best timeout vote	major: it takes $\mathcal{O}(n)$ to find the smallest or biggest timeout vote and we already retransmitted the block	`block.vid`, `block.`σ^{BC}, `block.round`, `block.qc.round`, `block.tc.qcrounds`, hash of the rest of `block`.
TC spans too many epochs	The TC can cover at most 2 epochs	major: checking the smallest or biggest timeout vote is efficient, but we already performed $\mathcal{O}(n)$ checks before getting here and we already retransmitted the block	`block.vid`, `block.`σ^{BC}, `block.tc.qcrounds`, hash of the rest of `block`.
Epoch finalization invalid (block)	The block is the first in a new epoch, but has an incorrect finalization proof	major: it takes $\mathcal{O}(n)$ to verify the correctness of a proof, and this is done after retransmitting the block	`block.vid`, `block.`σ^{BC}, `block.epoch`, `block.qc.epoch`, `block.epochFinEntry`, hash of the rest of the block.
Bad transactions outcomes	The transactions outcomes is wrong	major: checking the hash does not depend on the number of finalizers, but only comes after executing the block and doing other complex checks	`block.vid`, `block.`σ^{BC}, `block.data`, hash of the rest of `block`, `block.qc.ptr.stateHash`, `block.qc.ptr.vid`, `block.qc.ptr.`σ^{BC}, hash of the rest of `block.qc.ptr`

Table 11.3 Continued. $\mathcal{O}(n)$-detectable block errors.

offense	description	severity	witness
Bad state hash	The state hash is wrong	**major:** checking the hash does not depend on the number of finalizers, but only comes after executing the block and doing other complex checks	`block.vid`, `block.`σ^{BC}, `block.data`, hash of the rest of `block`, `block.qc.ptr.stateHash`, `block.qc.ptr.vid`, `block.qc.ptr.`σ^{BC}, hash of the rest of `block.qc.ptr`
Bad block hash	The block hash is wrong	**major:** checking the hash does not depend on the number of finalizers, but only comes after executing the block and doing other complex checks	the entire block.

Table 11.4 Errors when signing a block.

offense	description	severity	witness
Not a finalizer (sig)	A party validated a block without having the stake needed to be a finalizer.	**minor:** caught as soon as we check before retransmitting.	`fid`, σ^{AggFin}, `ptr`, `ptr.epoch`, `ptr.vid`, `ptr.`σ^{BC}, hash of the rest of the block `ptr`.
Signed invalid block	A finalizer signed a block that is not valid.	**major:** caught before retransmitting, but it could take $\mathcal{O}(n)$ to notice that the block is invalid.	all of `quorumSignature`, the witness proving that the block `quorumSignature.ptr` is not valid.
Signed two blocks	The same finalizer signed two different blocks in the same round.	**security:** this could result in a fork if the adversary controls the scheduling and $\geq \frac{1}{3}$ of the stake.	the two `quorumSignature`s.
Round inconsistent (sig)	The declared round doesn't match the actual block round.	**network:** this is a simple equality check, but we already retransmitted the `quorumSignature`	all of `quorumSignature`

offense	description	severity	witness
Epoch inconsistent (sig)	The declared epoch doesn't match the actual block epoch.	`network`: this is a simple equality check, but we already retransmitted the `quorumSignature`	all of `quorumSignature`

Table 11.5 Errors in timeouts.

offense	description	severity	witness
Not a finalizer (timeout)	A party sent a timeout but is not a finalizer.	`minor`: caught as soon as we check before retransmitting.	all of `timeout`.
Mutliple timeouts	A party sent multiple different timeouts for the same round.	`minor`: caught as soon as we check before retransmitting.	both `timeouts`.
Invalid timeout sig	The $\sigma^{\texttt{AggFin}}$ is not a valid signature of the expected message.	`network`: simple sig check performed after retransmitting	all of `timeout`.

Chapter 12

Kontrol Layer

The control layer Kontrol makes tweaks and parameters available to the other layers. The core idea is that (almost) all data can be computed from Skov.tree (this includes Skov.finEntries). Kontrol provides the necessary function to do these computations. The actual management of tweakable parameters is done in Renovatio (see Section 12.1.1).

12.1 Required Interfaces

12.1.1 Renovatio Interface

The Kontrol layer requires the following functions from Renovatio, cf. Section 19.6.3 for more details.

Current Parameter Function The function Renovatio.getCurrentParameter returns the current value of a parameter in a given block `block`.

This provides access to the parameter update system of Renovatio. The ranges and initial values for all updatable parameters are formally defined in the genesis block (cf. Section 21). For completeness, we provide them here as well.

> ❗ We assume that the initial values of parameters are part of the genesis data. For more details on the genesis data see Section 21.

12.2 Stake-Related Functions

Parameters which influence the set of validators or leader election of a round cannot be updated as soon as the transaction time is reached, but only if the effect will be the same in all branches. For example, suppose that the length of a pay day (`rewardPeriodLength`) is set differently in two (unfinalized) branches, and that one branch already transitions to the next pay day with a new set of validators whilst the other starts a new epoch in the old pay day with the old set of validators. Then each branch could in theory be finalized due to the different sets of validators. More generally, since the stake distribution affects the list of validators and the leader election, any parameter which influences the stake—or when the stake is updated—falls in this category. When the parameters can be updated is explained further in Section 12.2.2.

12.2.1 Parameters

The initial values for all these parameters can be found in Chapter 21.

Non-updatable parameters. The leader election nonce and the validator and finalizer sets are computed algorithmically. Their initial values are set in the genesis block. The `epochDuration` is a non-updatable parameter, that is fixed in the genesis block.

`genesisData.nonce`$^{\text{LE}} \in \text{NONCE}_{\text{LE}}$
>   The initial *leader election* nonce.

`genesisData.validators` $\in \text{VALIDATORS}^*$
>   The *initial set of eligible validators.*

`genesisData.finCom` $\in \text{VALIDATORS}^*$
>   The *initial set of eligible finalizers.*

`genesisData.epochDuration` $\in \mathbb{N}$
>   Length of an epoch in seconds.

Updateable parameters. The following parameter can be set using Renovatio.

`rewardPeriodLength` $\in \mathbb{N}$
>   Number of epochs in a pay day.

12.2.2 Updating parameters

As explained at the beginning of Section 12.2, we have to be careful when updating parameters that influence the stake distribution, the set of validators or the leader election. It is crucial that the effects of changing such parameters in an unfinalized branch cannot result in different leaders or validator stake than an alternative unfinalized branch, i.e., the new parameter values must be in a finalized block before they have an effect. For example, the length of a pay day is determined by the value of `rewardPeriodLength` at the start of that pay day. Since a new epoch (and thus a new pay day) only starts after the trigger block has been finalized, even if there are two branches with different `rewardPeriodLength` values at the start of the pay day, the transition to the next pay day can only take place once one of the two branches has been finalized.

Similarly, if epoch e marks the beginning of a new pay day, the set of bakers and finalizers is computed from the transactions in blocks up to the end of epoch $e-2$ with transaction times and cooldown expiration before the nominal end of epoch $e-1$ (see Section 12.2.3).[1]

12.2.3 Interface

Validator and finalizer sets. The function Kontrol.getBakerSet returns the set of validators, including their keys and lottery power, for a specific epoch.

Function Kontrol.getBakerSet

Inputs
   `epoch` $\in \mathbb{N}$
>   The epoch for which we want the set of validators.
   `epochFinEntries` $\in \text{EPOCHFINALENTRIES}^*$
>   List of epoch finalization proofs for all previous epochs.

---

[1]The transactions have to be in blocks up to the end of epoch $e-2$ for the `nonce`$^{\text{LE}}$ to have high enough entropy, see Section 13.1.

Outputs

  validators ∈ VALIDATORS*
        The set of eligible validators in epoch epoch.

Promise

   Let e be the first epoch of the payday of epoch. The validators can be uniquely
   computed for epoch if $e = 1$ or epochFinEntries contains an entry for epoch $e - 1$.

Implementation The validator set is stable for an entire pay day. If the first epoch of the
pay day is epoch e, then the set is determined by all transactions in blocks up to the end of
epoch $e - 2$. For the first pay day, return genesisData.validators.

The *finalization committee* is determined similarly.

> ℹ The finalization committee is a subset of validators that have enough stake. The exact
> parameters for determining which validators are finalizers is part of the tokenomics and
> out of the scope of this bluepaper.

Function Kontrol.getFinalizationCommittee

Inputs

  epoch ∈ ℕ
        The epoch for which we want the finalization committee.
  epochFinEntries ∈ EPOCHFINALENTRIES*
        List of epoch finalization proofs for all previous epochs.

Outputs

  finCom ∈ VALIDATORS*
        The set of finalizers eligible to finalize any block in epoch epoch.

Promise

   Let e be the first epoch of the pay day of epoch. The finCom can be uniquely computed
   for epoch if $e = 1$ or epochFinEntries contains an entry for epoch $e - 1$.

Implementation The finCom is determined from the validator stake. The stake is stable for
an entire payday. If the first epoch of the pay day is epoch e, then the stake is determined
by all transactions in blocks up to the end of epoch $e - 2$. For the first pay day, return
genesisData.finCom.

> ℹ The two functions above require epochFinEntries to contain an entry for epoch $e - 1$.
> However, it is sufficient for any block in epoch $e - 1$ to be finalized to uniquely determine
> the validators and finalizers for the next payday. What is more, these sets can already be
> computed in the first block of epoch $e - 1$, even though different branches might have
> different sets, since only one branch will survive after a block in epoch $e - 1$ is finalized.

> </> In the current implementation, any validator pool which has at least 0.005% of the total stake is part of the finalization committee—as long as there are not more than 1000 such pools—which is roughly equivalent to the stake needed to open a new validator, currently set at CCD $500'000$.

Epochs and pay day lengths. The `epochDuration` is not an updatable parameter, so it is sufficient to provide a pointer to the chain to get the value.

Function Kontrol.getEpochDuration

Inputs

  $ptr \in \text{HASH}$
      A pointer to a block used as point of reference.

Outputs

  $epochDuration \in \mathbb{N}$
      The length of an epoch in seconds.

Promise

  ptr must point to a block in a valid chain.

Implementation

  1: **return** genesisData.epochDuration

Since `rewardPeriodLength` is updatable, the value returned by `getRewardPeriodLength` will be the one in the block provided as input.

Function Kontrol.getRewardPeriodLength

Inputs

  $ptr \in \text{HASH}$
      A pointer to the block for which the value should be returned.

Outputs

  $rewardPeriodLength \in \mathbb{N}$
      The number of epochs of the payday in which ptr is included.

Promise

  ptr must point to a block in a valid chain. If ptr gets finalized, then the value returned is that of the corresponding pay day.

Implementation Returns the value based on all transactions up to ptr. Since parameter updates are typically executed at a given time, the returned value will only take into account those executed by the time of the block.

12.3   Consensus Functions

This section contains the consensus parameter functions which allow a party to get among other things the threshold of signatures needed or the base timeout time.

12.3.1   Parameters

The initial values for all these parameters can be found in Chapter 21.

Non-Updatable Parameters   In a setting with gears[2], `sigThreshold`, the threshold of signatures needed for forming quorum certificates and timeout certificates, changes dynamically within the range $[2/3, 1]$. Without gears, it is fixed to the lower bound $2/3$, which provides safety and liveness up to $1/3$ corruption.

`genesisData.sigThreshold` $\in \mathbb{Q}$
     Weight of signatures needed for a QC or a TC.

Updateable parameters.   The following parameters can be set using Renovatio.

`timeoutBase`, `timeoutIncrease` and `timeoutDecrease` regulate the amount of time we wait for advancing the round before triggering a timeout. When a timeout is triggered we increase the timeout time, and when we finalize a block we decrease the timeout time (but never go below `timeoutBase`). `minBlockTime` and `maxBlockEnergy` determine the minimum time between blocks and the maximum block size, which directly impact the transactions per second.

`timeoutBase` $\in \mathbb{N}$
     The base value for triggering a timeout in milliseconds.

`timeoutIncrease` $\in \mathbb{Q}$
     Factor for increasing the timeout. Applied locally when a timeout occurs.

`timeoutDecrease` $\in \mathbb{Q}$
     Factor for decreasing the timeout. Applied when a block is finalized. The value `currentTimeout` never goes below `timeoutBase`.

`minBlockTime` $\in \mathbb{N}$
     Minimum time between blocks in milliseconds.

`maxBlockEnergy` $\in \mathbb{N}$
     Maximum energy for all the transactions in a block.

12.3.2   Interface

The following functions return the parameters introduced in Section 12.3.1.

The updatable parameters defined in Section 12.3.1 are all updatable "normally", i.e., the change is effective after the transaction time in any branch that has the transaction in an ancestor block.

Since `sigThreshold` is not updatable, it only requires a pointer to a chain to retrieve the value from the genesis.

---

[2]The mechanism by which the committee is renewed when a deadlock is detected.

Function Kontrol.getSigThreshold

Inputs

$\mathtt{ptr} \in$ HASH

A pointer to a block used as point of reference.

Outputs

$\mathtt{sigThreshold} \in \mathbb{Q}$

The signature threshold on the chain to which `ptr` points.

Promise

`ptr` must point to a block in a valid chain.

Implementation

1: **return genesisData.sigThreshold**

The following parameters are updatable, so the function will return the value in the block provided as input.

Function Kontrol.getTimeoutBase

Inputs

$\mathtt{ptr} \in$ HASH

A pointer to a block for which we want to know the parameter value.

Outputs

$\mathtt{timeoutBase} \in \mathbb{N}$

The base value for triggering a timeout in milliseconds as defined in `ptr`.

Promise

`ptr` must point to a block in a valid chain.

Implementation Returns the value based on all transactions up to `ptr`. Since parameter updates are typically executed at a given time, the returned value will only take into account those executed by the time of the block.

Function Kontrol.getTimeoutIncrease

Inputs

$\mathtt{ptr} \in$ HASH

A pointer to a block for which we want to know the parameter value.

Outputs

$\mathtt{timeoutIncrease} \in \mathbb{Q}$

The factor for increasing the timeout as defined in `ptr`.

Promise

`ptr` must point to a block in a valid chain.

Implementation Returns the value based on all transactions up to `ptr`. Since parameter updates are typically executed at a given time, the returned value will only take into account those executed by the time of the block.

Function Kontrol.getTimeoutDecrease

Inputs

`ptr` ∈ HASH
> A pointer to a block for which we want to know the parameter value.

Outputs

`timeoutDecrease` ∈ $\mathbb{Q}$
> The factor for decreasing the timeout as defined in `ptr`.

Promise
> `ptr` must point to a block in a valid chain.

Implementation Returns the value based on all transactions up to `ptr`. Since parameter updates are typically executed at a given time, the returned value will only take into account those executed by the time of the block.

Function Kontrol.getMinBlockTime

Inputs

`ptr` ∈ HASH
> A pointer to a block for which we want to know the parameter value.

Outputs

`minBlockTime` ∈ $\mathbb{N}$
> The minimum time between blocks in milliseconds as defined in `ptr`.

Promise
> `ptr` must point to a block in a valid chain.

Implementation Returns the value based on all transactions up to `ptr`. Since parameter updates are typically executed at a given time, the returned value will only take into account those executed by the time of the block.

Function Kontrol.getMaxBlockEnergy

Inputs

`ptr` ∈ HASH
> A pointer to a block for which we want to know the parameter value.

Outputs

`maxBlockEnergy` ∈ $\mathbb{N}$
> The maximum energy for all the transactions in a block as defined in `ptr`.

Promise

    `ptr` must point to a block in a valid chain.

Implementation Returns the value based on all transactions up to `ptr`. Since parameter updates are typically executed at a given time, the returned value will only take into account those executed by the time of the block.

Chapter 13

Proofs

13.1 Bakers, Finalizers, and Leader Election

In this section, we show that all parties agree on the current sets of bakers and finalizers and that dishonest parties cannot manipulate the leader election to make them win a majority of the elections for an extended period of time. For all the proofs here, we assume that safety was not violated in previous epochs, i.e., there are no finalized blocks on different forks.

> ❗ The safety proofs in Section 13.2.1 need to assume agreement on the set of finalizers and their relative stake. Hence, the overall security follows by induction over the epochs.

13.1.1 Unique Reward Period Lengths

We update the bakers and finalizers and their stakes every payday. Hence, to get agreement on these sets, it is a prerequisite to agree on the lengths of the paydays. Since these correspond to an updatable parameter controlled by getRewardPeriodLength, we first show that all honest parties obtain the same parameter in every epoch.

Lemma 71. *Consider some epoch* epoch *such that all finalized blocks in epochs up to and including* epoch -1 *are on a chain. Then, all honest nodes will obtain the same value from* Kontrol.getRewardPeriodLength(epoch, ·).

Proof. We prove this by induction. For the first epoch, the value is taken from the genesis block, so all nodes agree on it. Now assume all honest nodes agree on the reward period length for all epochs up to epoch -1. If epoch epoch is in the same payday as epoch -1, parties agree on that fact and since the reward period does not change during a payday, they also agree on the value for epoch. If epoch is the first epoch in a new payday, the payday length is computed from blocks up to the trigger block of epoch -1. This trigger block is finalized by the finalization entry of epoch -1. By our safety assumption, all honest parties agree on the finalized chain up to this trigger block, and thus also on the next reward period length. □

13.1.2 Unique Finalizer Sets and Leaders

We now show that in every epoch, all nodes agree on the set of bakers and finalizers and their relative stake.

Lemma 72. *Consider some epoch* epoch *such that all finalized blocks in epochs up to and including* epoch -1 *are on a chain. Then, all honest parties agree on the sets of bakers and*

finalizers and their relative stake for epoch epoch. *That is,* Kontrol.getBakerSet(epoch, ·) *and* Kontrol.getFinalizationCommittee(epoch, ·) *return the same values for all honest parties.*

Proof. It is sufficient to prove this for the first epoch of a payday, since later epochs in the same payday produce the same results. Note that by Lemma 71, all honest parties agree on which epoch is the first one in a payday. If epoch = 1, Kontrol.getBakerSet and Kontrol.getFinalizationCommittee return the unique values from the genesis block. Otherwise, epochFinEntries contains a valid finalization entry for epoch − 1. This contains a finalization entry for a block in epoch − 1 that implicitly finalizes all previous blocks, in particular all blocks up to the end of epoch − 2. By assumption, all honest parties agree on all those blocks, and since the two functions deterministically compute the results from the transactions in these blocks, the parties agree on the results of these functions. □

Lemma 73. *Consider some epoch* epoch *such that all finalized blocks in epochs up to and including* epoch − 1 *are on a chain. Then, all honest parties agree on the leader-election nonce for epoch* epoch. *That is, they all obtain the same value from* getLENonce(epoch, ·).

Proof. We prove this by induction. For epoch = 1, getLENonce returns a unique nonce from the genesis block. Now assume all leader-election nonces up to epoch − 1 are uniquely determined. getLENonce initializes nonce^{LE} with the unique value from epoch − 1. It then updates the nonce based on blocks in epoch − 1 up to the trigger block. This trigger block is finalized by the finalization entry of epoch − 1. Since we assume that all finalized blocks up to this epochs are on a chain, this trigger block and all predecessors are uniquely determined. Finally, getLENonce hashes the epoch number and the set of bakers determined by getBakerSet to the nonce. By Lemma 72, this set is also uniquely determined. Hence, all parties agree on the the leader-election nonce for epoch epoch. □

We next show that each round in every epoch has a unique leader, again assuming safety has not been violated previously.

Lemma 74. *Consider some epoch* epoch *such that all finalized blocks in epochs up to and including* epoch − 1 *are on a chain. Then, there is a unique leader for every round in epoch* epoch, *i.e., for all* round, getLeader(round, epoch, ·) *returns the same value to all honest parties.*

Proof. The leader of a round is deterministically computed from the current leader-election nonce and the baker set (including their stake distribution). Uniqueness of leaders thus follows from uniqueness of leader-election nonces (implied by Lemma 73) and uniqueness of baker sets (implied by Lemma 72). □

> ⚠ Lemma 74 does *not* imply that there is a unique leader for every round, only that the leaders are unique given a round *and* an epoch. That is, there can be two different leaders for the same round if different epochs are considered for that round. This is possible since there is no unique mapping from round numbers to epochs. In particular, this can happen if only some of the honest parties have seen the last epoch finalization proof, and the others remain in the previous epoch.

13.2 Safety and Liveness

In this section, we assume an adversary $\mathcal{A}$ controls bakers with at most a fraction $\alpha_B < 1$ of the bakers' stake and finalizers with at most a fraction $\alpha_F < 1/3$ (minus some margin for moving

stake between epochs) of the finalizers' stake. We further assume `sigThreshold` $\geq 2/3$. We will state the required conditions on these values in all statements explicitly.

> ⚠️ We assume for simplicity that corruptions are static. It is probably possible to generalize all statements to adaptive corruptions, but that would complicate all the proofs because one needs to keep track of who is corrupted when and different finalizers can be in different epochs at a given point in time etc.

13.2.1 Safety

> ⚠️ We here assume for simplicity that forging signatures is impossible. Hence, all results are conditioned on the adversary failing to forge signatures.

> ⚠️ It is here assumed that all finalizer sets and their stakes are uniquely determined without referring to the lemmata from Section 13.1. The statements here are thus not fully precise since the additional assumptions on not violating safety previously are missing at some points.

Definition 75. We denote by $\alpha_{\mathtt{fid}}^{\mathtt{epoch}}$ the relative stake of finalizer $\mathtt{fid}$ in epoch $\mathtt{epoch}$ with the convention $\alpha_{\mathtt{fid}}^{\mathtt{epoch}} = 0$ if $\mathtt{fid}$ is not a finalizer in epoch $\mathtt{epoch}$. For epochs $\mathtt{epoch}_1, \mathtt{epoch}_2$ let F be the set of finalizers that are finalizers in at least one of the two epochs, and define

$$\Delta_{\mathrm{finStake}}^{\mathtt{epoch}_1, \mathtt{epoch}_2} := \max_{S \subseteq F} \left| \sum_{\mathtt{fid} \in S} \alpha_{\mathtt{fid}}^{\mathtt{epoch}_2} - \sum_{\mathtt{fid} \in S} \alpha_{\mathtt{fid}}^{\mathtt{epoch}_1} \right|.$$

We first show a simple lemma about quorum intersections.

Lemma 76. *Let* $\mathtt{epoch}_1, \mathtt{epoch}_2$ *be two (possibly equal) epochs, let* Q_1 *and* Q_2 *be two sets of finalizers such that both sets contain finalizers with at least* `sigThreshold` *relative stake in epochs* $\mathtt{epoch}_1$ *and* $\mathtt{epoch}_2$, *respectively, and assume* $\alpha_F + \Delta_{\mathrm{finStake}}^{\mathtt{epoch}_1, \mathtt{epoch}_2} < 2 \cdot \mathtt{sigThreshold} - 1$. *Then, there is at least one honest finalizer in* $Q_1 \cap Q_2$.

Proof. Finalizers in Q_1 have at least `sigThreshold` relative stake in $\mathtt{epoch}_1$, and thus at least $\mathtt{sigThreshold} - \Delta_{\mathrm{finStake}}^{\mathtt{epoch}_1, \mathtt{epoch}_2}$ relative stake in $\mathtt{epoch}_2$. Hence, finalizers not in Q_1 have at most $1 - \mathtt{sigThreshold} + \Delta_{\mathrm{finStake}}^{\mathtt{epoch}_1, \mathtt{epoch}_2}$ relative stake in $\mathtt{epoch}_2$ and all of these can be in Q_2. Since in total finalizers with at least `sigThreshold` relative stake in $\mathtt{epoch}_2$ are in Q_2, the remaining at least $\mathtt{sigThreshold} - (1 - \mathtt{sigThreshold} + \Delta_{\mathrm{finStake}}^{\mathtt{epoch}_1, \mathtt{epoch}_2}) = 2 \cdot \mathtt{sigThreshold} - 1 - \Delta_{\mathrm{finStake}}^{\mathtt{epoch}_1, \mathtt{epoch}_2}$ relative stake must be in both Q_1 and Q_2. Since we assume $2 \cdot \mathtt{sigThreshold} - 1 > \alpha_F + \Delta_{\mathrm{finStake}}^{\mathtt{epoch}_1, \mathtt{epoch}_2}$, at least one honest finalizer is in $Q_1 \cap Q_2$. □

We now show that epoch numbers in quorum certificates monotonically increase with the round numbers of the quorum certificates.

Lemma 77. *Let* $\mathtt{qc}_1, \mathtt{qc}_2$ *be valid quorum certificates with* $\mathtt{qc}_1.\mathtt{round} \leq \mathtt{qc}_2.\mathtt{round}$ *and assume* $\alpha_F + \Delta_{\mathrm{finStake}}^{\mathtt{epoch}, \mathtt{epoch}-1} < 2 \cdot \mathtt{sigThreshold} - 1$ *for all* $\mathtt{epoch} \leq \max\{\mathtt{qc}_1.\mathtt{epoch}, \mathtt{qc}_2.\mathtt{epoch}\}$. *Then,* $\mathtt{qc}_1.\mathtt{epoch} \leq \mathtt{qc}_2.\mathtt{epoch}$.

Proof. We prove that if $\mathtt{qc}_1.\mathtt{epoch} > \mathtt{qc}_2.\mathtt{epoch}$, then $\mathtt{qc}_1.\mathtt{round} > \mathtt{qc}_2.\mathtt{round}$ by induction over $\delta := \mathtt{qc}_1.\mathtt{epoch} - \mathtt{qc}_2.\mathtt{epoch}$. First consider the case $\delta = 1$. Then, both quorum certificates are

signed by finalizers with relative stake at least `sigThreshold` in two consecutive epochs. Hence we can use our assumption on $\Delta_{\mathsf{finStake}}^{\mathsf{qc}_1.\mathsf{epoch},\mathsf{qc}_2.\mathsf{epoch}}$ and apply Lemma 76 to conclude that there exists an honest finalizer who contributed signatures to both quorum certificates. It is clear from the definitions that all honest finalizers never decrease their current epoch numbers, and they do not sign blocks from older epochs than their current epoch (cf. validateBlock in Section 11.4.2). Hence, we have $\mathsf{qc}_1.\mathsf{round} > \mathsf{qc}_2.\mathsf{round}$.

Now let $\delta > 1$ and assume the claim is true for all $\delta' < \delta$. Since $\mathsf{qc}_1.\mathsf{epoch} > \mathsf{qc}_2.\mathsf{epoch} + 2$, there exist valid epoch finalization entries for epochs $\mathsf{qc}_2.\mathsf{epoch}$ and $\mathsf{qc}_2.\mathsf{epoch} + 1$ in round $\mathsf{qc}_1.\mathsf{round}$. In particular, the one for epoch $\mathsf{qc}_2.\mathsf{epoch} + 1$ contains a quorum certificate qc' for a finalized block in epoch $\mathsf{qc}_2.\mathsf{epoch} + 1$. Hence, $\mathsf{qc}'.\mathsf{epoch} - \mathsf{qc}_2.\mathsf{epoch} = 1 < \delta$ and $\mathsf{qc}_1.\mathsf{epoch} - \mathsf{qc}'.\mathsf{epoch} = \delta - 1 < \delta$. We can thus apply the induction hypothesis twice to obtain $\mathsf{qc}_1.\mathsf{round} > \mathsf{qc}'.\mathsf{epoch} > \mathsf{qc}_2.\mathsf{round}$. □

The following lemma shows that if an honest finalizer signs a block leading to a quorum certificate finalizing a block `block` and in the same or a later round a timeout message, this timeout message contains a quorum certificate for a round at least as high as `block.round`. It will be useful to prove later lemmata.

Lemma 78. *Let* `finEntry` *be a valid finalization entry for a block* `block` *and let* `finalizer` *be an honest finalizer who contributed a signature to* `finEntry.qc`$_2$. *Further assume* `finalizer` *signed a timeout message* `timeout` *with* `timeout.round` $\geq$ `finEntry.qc`$_2$`.round`. *Then,* `timeout.qc.round` $\geq$ `block.round`.

Proof. The validity of `finEntry` implies that the block block_2 certified by `finEntry.qc`$_2$ extends `block`, which is certified by `finEntry.qc`$_1$. Hence, when `finalizer` received block_2, they set their `highestQC` to be at least `finEntry.qc`$_1$ (see line 149 in Protocol uponReceivingBlock). This finalizer has not signed a timeout message for round $\mathsf{block}_2.\mathsf{round}$ prior to receiving block_2, because otherwise, uponTimeoutEvent would have set `nextSignableRound` to $\mathsf{block}_2.\mathsf{round}+1$ and validateBlock would not sign block_2 anymore. Since `timeout.round` $\geq$ `finEntry.qc`$_2$`.round` $= \mathsf{block}_2.\mathsf{round}$, this implies that at the time `finalizer` signed `timeout`, this finalizer's `highestQC` was at least for round `block.round`. Hence, `timeout.qc.round` $\geq$ `block.round`. □

We next show that under certain assumptions, if a block from round `round` is finalized, then all timeout certificates for future rounds contain a quorum certificate for a round at least `round`.

Lemma 79. *Let* `block` *be a block for which a valid finalization entry* `finEntry` *exists and let* `tc` *be a valid timeout certificate with* `tc.round` $>$ `block.round` *in a certified block* `block`$'$, *and let* `block`$''$ *be the parent of* `block`$'$. *Assume that for all epochs* `epoch`$' \leq \max\{$`block.epoch`, `block`$'$`.epoch`, `block`$''$`.epoch`$\}$, *we have* $\alpha_F + \Delta_{\mathsf{finStake}}^{\mathsf{epoch}',\mathsf{epoch}'-1} < 2 \cdot \mathsf{sigThreshold} - 1$. *We then have* `block`$''$`.round` $\geq$ `block.round`.

Proof. If `block`$''$`.epoch` $>$ `block.epoch`, then Lemma 77 implies that `block`$''$`.round` $>$ `block.round` (note since both blocks are certified, we can apply the lemma to the corresponding quorum certificates).

From now on assume `block`$''$`.epoch` $\leq$ `block.epoch`. We have `block`$'$`.round` $- 1 =$ `tc.round` $>$ `block.round`. Hence, `block`$'$`.epoch` $\geq$ `block.epoch` by Lemma 77. Since `block`$''$ is the parent of `block`$'$, it can be at most one epoch behind (cf. Definition 64). Hence, `block`$''$`.epoch` $\geq$ `block.epoch` $- 1$. This implies $|$`block`$''$`.epoch` $-$ `block.epoch`$| \leq 1$.

Let Q_{qc_2} be the set of finalizers whose signatures contribute to `finEntry.qc`$_2$ and let Q_{tc} be the set of finalizers who signed timeout messages in `tc`. Then, Q_{qc_2} contains finalizers with relative stake at least `sigThreshold` in epoch `block.epoch`, and since the validity of `tc` is checked relative

to the epoch of $\texttt{block}''$, $Q_{\texttt{tc}}$ contains finalizers with relative stake at least $\texttt{sigThreshold}$ in epoch $\texttt{block}''.\texttt{epoch}$. We can thus apply Lemma 76 to conclude that there is at least one honest finalizer in $Q_{\texttt{qc}_2} \cap Q_{\texttt{tc}}$. Since $\texttt{finEntry.qc}_2.\texttt{round} = \texttt{finEntry.qc}_1.\texttt{round} + 1 = \texttt{block.round} + 1$ (cf. Definition 59) and $\texttt{tc.round} > \texttt{block.round}$, we have $\texttt{tc.round} \geq \texttt{finEntry.qc}_2.\texttt{round}$. Therefore, Lemma 78 implies that the honest finalizer signed a timeout message $\texttt{timeout}$ in $\texttt{tc}$ with $\texttt{timeout.qc.round} \geq \texttt{block.round}$. By Definition 64, we can conclude that

$$\texttt{block}''.\texttt{round} = \texttt{block}'.\texttt{qc.round} \geq \max\{\texttt{tc.qcrounds}\} \geq \texttt{timeout.qc.round} \geq \texttt{block.round}.$$

This concludes the proof. $\qquad\square$

We next prove that in every round, the current epochs of honest finalizers can differ by at most 1.

Lemma 80. *Let* $\texttt{round}$ *be a round and assume for all blocks* $\texttt{block}_1$, $\texttt{block}_2$ *with* $\texttt{block}_1.\texttt{round} \leq \texttt{block}_2.\texttt{round} < \texttt{round}$ *such that a valid finalization entry exists for* $\texttt{block}_1$ *and a valid quorum certificate exists for* $\texttt{block}_2$, *we have that the chain from the genesis block to* $\texttt{block}_2$ *contains* $\texttt{block}_1$. *Consider honest finalizers* $\texttt{finalizer}_1$ *and* $\texttt{finalizer}_2$ *that are in round* $\texttt{round}$ *in epochs* $\texttt{epoch}_1$ *and* $\texttt{epoch}_2$, *respectively. Further assume that for all epochs* $\texttt{epoch}' < \max\{\texttt{epoch}_1, \texttt{epoch}_2\}$, *we have* $\alpha_F + \Delta_{\text{finStake}}^{\texttt{epoch}',\texttt{epoch}'-1} < 2 \cdot \texttt{sigThreshold} - 1$. *We then have* $|\texttt{epoch}_2 - \texttt{epoch}_1| \leq 1$.

Proof. We prove this by induction over $\texttt{round}$. For $\texttt{round} = 1$, we have $\texttt{epoch}_1 = \texttt{epoch}_2 = 1$. Now assume up to round $\texttt{round} - 1$, all honest finalizers are in at most by one differing epochs. Let $\texttt{epoch}'_1 \leq \texttt{epoch}'_2$ be the epochs in which honest finalizers are in round $\texttt{round} - 1$. If no honest finalizers move to an epoch strictly greater than $\texttt{epoch}'_2$ in round $\texttt{round}$, then the claim holds for that round because honest finalizers never decrease their current epoch numbers. Now assume there is an honest finalizer $\texttt{finalizer}_2$ that moves to an epoch $\texttt{epoch}_2 > \texttt{epoch}'_2$ in round $\texttt{round}$. Since honest finalizers never decrease their current epoch numbers, no honest finalizer has been in an epoch strictly greater than $\texttt{epoch}'_2$ up to round $\texttt{round} - 1$. Hence, no valid epoch finalization entry can exist for epochs strictly greater than $\texttt{epoch}'_2$ in round $\texttt{round} - 1$, which implies that $\texttt{epoch}_2 = \texttt{epoch}'_2 + 1$. Since $\texttt{finalizer}_2$ moves to $\texttt{epoch}'_2 + 1$, there exists an epoch finalization entry for epoch $\texttt{epoch}'_2$ in round $\texttt{round}$, which contains quorum certificates $\texttt{qc}_1$, $\texttt{qc}_2$ with

$$\texttt{qc}_1.\texttt{round} < \texttt{qc}_2.\texttt{round} < \texttt{round} \quad \text{and} \quad \texttt{qc}_1.\texttt{epoch} = \texttt{qc}_2.\texttt{epoch} = \texttt{epoch}'_2$$

(see Definition 62 and Definition 59 and note that if $\texttt{qc}_2.\texttt{round} \geq \texttt{round}$, then $\texttt{finalizer}_2$ would move to $\texttt{qc}_2.\texttt{round} + 1 > \texttt{round}$ when receiving $\texttt{qc}_2$, cf. $\texttt{processQuorumSignature}$). We have to show that $\texttt{epoch}_1 \geq \texttt{epoch}'_2$. When $\texttt{finalizer}_1$ moves to round $\texttt{round}$, there is either a valid quorum certificate for a block in round $\texttt{round} - 1$, or $\texttt{finalizer}_1$ forms a valid timeout certificate for that round.

We distinguish the two cases. First assume there is a valid quorum certificate for a block $\texttt{block}'$ in round $\texttt{block}'.\texttt{round} = \texttt{round} - 1 > \texttt{qc}_1.\texttt{round}$. Since $\texttt{qc}_1$ points to a finalized block, the assumptions of the lemma imply that this block is on the chain to $\texttt{block}'$. Since epoch numbers cannot decrease within a chain, this implies $\texttt{block}'.\texttt{epoch} \geq \texttt{qc}_1.\texttt{epoch} = \texttt{epoch}'_2$. Upon receiving $\texttt{block}'$, $\texttt{finalizer}_1$ moves at least to epoch $\texttt{block}'.\texttt{epoch} \geq \texttt{epoch}'_2$ (cf. $\texttt{uponReceivingBlock}$ in Section 11.4.2), which implies $\texttt{epoch}_1 \geq \texttt{epoch}'_2$ in this case.

Now assume $\texttt{finalizer}_1$ moves to $\texttt{round}$ when receiving enough timeout messages to form a valid timeout certificate for round $\texttt{round} - 1$. Let Q_1 be the set of finalizers that signed such a timeout message and let $\texttt{epoch}'_1 \leq \texttt{epoch}'_2$ be the epoch in which $\texttt{finalizer}_1$ is when forming the valid timeout certificate in round $\texttt{round} - 1$. The finalizers in Q_1 have relative stake at least $\texttt{sigThreshold}$ in epoch $\texttt{epoch}'_1$. Let Q_2 be the set of finalizers that signed the quorum

certificate qc_2. Validity of the quorum certificate implies that the weight of finalizers in Q_2 is at least sigThreshold in epoch $\text{qc}_2.\text{epoch} = \text{epoch}_2'$. The induction hypothesis implies that $|\text{epoch}_1' - \text{epoch}_2'| \leq 1$. Hence, we can use the assumption on $\Delta_{\text{finStake}}^{\text{epoch}_1', \text{epoch}_2'}$ to apply Lemma 76 and obtain that there is at least one honest finalizer $\text{finalizer}^* \in Q_1 \cap Q_2$. Let timeout be the timeout message signed by finalizer^*. Since $\text{timeout.round} = \text{round} - 1 \geq \text{qc}_2.\text{round}$, we can apply Lemma 78 to conclude that $\text{timeout.qc.round} \geq \text{qc}_1.\text{round}$. Note that the epoch numbers of the quorum certificates timeout.qc and qc_1 are at most $\text{epoch}_2' = \text{epoch}_2 - 1 < \max\{\text{epoch}_1, \text{epoch}_2\}$. Hence, by Lemma 77, this implies that $\text{timeout.qc.epoch} \geq \text{qc}_1.\text{epoch} = \text{epoch}_2'$. When receiving this timeout message, finalizer_1 thus catches up to obtain the block for timeout.qc (cf. uponReceivingTimeout) and when receiving that block, moves to at least epoch_2'. This shows that $\text{epoch}_1 \geq \text{epoch}_2'$ in both cases and concludes the proof. $\qquad\square$

Using the above lemmata, we show that there can be at most one certified block in every round.

Lemma 81. *Let* block, block' *be certified blocks with* $\text{round} := \text{block.round} = \text{block}'.\text{round}$ *and assume for all blocks* block_1, block_2 *with* $\text{block}_1.\text{round} \leq \text{block}_2.\text{round} < \text{round}$ *such that a valid finalization entry exists for* block_1 *and a valid quorum certificate exists for* block_2, *we have that the chain from the genesis block to* block_2 *contains* block_1. *Further assume* $\alpha_F + \Delta_{\text{finStake}}^{\text{epoch},\text{epoch}-1} < 2 \cdot \text{sigThreshold} - 1$ *for all* $\text{epoch} \leq \max\{\text{block.epoch}, \text{block}'.\text{epoch}\}$. *We then have* $\text{block} = \text{block}'$.

Proof. By definition of certified, there exist valid quorum certificates qc and qc' with $\text{qc.ptr} = \text{H}(\text{block})$ and $\text{qc}'.\text{ptr} = \text{H}(\text{block}')$. Validity of the quorum certificates implies that both contain valid signatures of finalizers with total relative stake at least sigThreshold in epochs qc.epoch and $\text{qc}'.\text{epoch}$, respectively. By Lemma 80, there is an epoch epoch^* such that all honest finalizers are in either epoch^* or $\text{epoch}^* - 1$ in round round. We therefore have $\text{block.epoch}, \text{block}'.\text{epoch} \in \{\text{epoch}^*, \text{epoch}^* - 1\}$, since honest finalizers only sign blocks matching their current epoch (cf. validateBlock) and $\alpha_F < 2 \cdot \text{sigThreshold} - 1 \leq \text{sigThreshold}$. Hence, we can apply Lemma 76 to obtain that at least one honest finalizer finalizer^* signed both block and block'. This finalizer follows the protocol specification validateBlock, i.e., after signing one of the two blocks, nextSignableRound is increased, and no further blocks for this round are signed by finalizer^*. This implies that $\text{block} = \text{block}'$. $\qquad\square$

We finally show that all finalized blocks are on the chains from the genesis block to certified blocks. Since finalized blocks are certified, this implies that all finalized blocks are on a chain.

Theorem 82. *Let* block_1 *be a block for which a finalization entry exist, and let* block_2 *be a certified block with* $\text{block}_2.\text{round} \geq \text{block}_1.\text{round}$ *and assume* $\alpha_F + \Delta_{\text{finStake}}^{\text{epoch},\text{epoch}-1} < 2 \cdot \text{sigThreshold} - 1$ *for all* $\text{epoch} \leq \max\{\text{block}_1.\text{epoch}, \text{block}_2.\text{epoch}\}$. *Then, the chain from the genesis block to* block_2 *contains* block_1.

Proof. We prove the theorem by induction over $\text{block}_2.\text{round}$. For $\text{block}_2.\text{round} = 1$, we have $\text{block}_1.\text{round} = \text{block}_2.\text{round} = 1$, and we can apply Lemma 81 to obtain $\text{block}_1 = \text{block}_2$.

Now let $\text{block}_2.\text{round} > 1$ and assume the claim holds for all smaller round numbers. We prove the claim for $\text{block}_2.\text{round}$ again by induction, now over $i := \text{block}_2.\text{round} - \text{block}_1.\text{round}$. For $i = 0$, Lemma 81 implies that $\text{block}_1 = \text{block}_2$, so the claim holds. Now assume $i > 0$ and that the claim holds for all $i' < i$. Let block' be the parent of block_2, i.e., $\text{H}(\text{block}') = \text{block}_2.\text{qc.ptr}$. We distinguish two cases: If block_2 and block' are in consecutive rounds, we have $\text{block}_1.\text{round} \leq \text{block}'.\text{round} < \text{block}_2.\text{round}$. Hence, the induction hypothesis implies that block_1 is on the chain to block'. Since block_2 extends block', the claim follows for this case. Now consider the case where block_2 and block' are not in consecutive rounds. Then, block_2 contains a timeout certificate tc for round $\text{block}_2.\text{round} - 1$. Since

$\texttt{block}_2.\texttt{round} > \texttt{block}_1.\texttt{round}$, we have $\texttt{block}_2.\texttt{tc.round} \geq \texttt{block}_1.\texttt{round}$. It is not possible to have $\texttt{block}_2.\texttt{tc.round} = \texttt{block}_1.\texttt{round}$, since that would mean that $\texttt{block}_1$ and $\texttt{block}_2$ are in consecutive rounds and $\texttt{block}_2$ does not extend $\texttt{block}_1$; but since $\texttt{block}_1$ is finalized, there is a certified block in the round after $\texttt{block}_1.\texttt{round}$ extending $\texttt{block}_1$ and by Lemma 81, there can be at most one certified block in each round. Hence, we have $\texttt{block}_2.\texttt{tc.round} > \texttt{block}_1.\texttt{round}$. Hence, Lemma 79 implies that $\texttt{block}'.\texttt{round} \geq \texttt{block}_1.\texttt{round}$. We can therefore again conclude using the induction hypothesis that $\texttt{block}_1$ is on the chain to $\texttt{block}'$ and consequently also to $\texttt{block}_2$. □

13.3 Liveness

> ⚠ We here assume that a message multicast by an honest party is delivered to all honest parties within time $\Delta_{\texttt{net}}$, where $\Delta_{\texttt{net}}$ is not used in the protocol and can change over time. We assume there are *periods of synchrony* during which $\Delta_{\texttt{net}}$ is bounded by some value known to the parties.

> ⚠ We further assume that obtaining the requested information via catch up can be completed by an honest party within time $\texttt{CatchUpTime}$.

> ⚠ We here only consider the network delays and timeout values for deriving bounds on the time of certain events and ignore the time it takes parties to verify signatures, process blocks, etc. For concrete bounds, these values have to considered as well, but more meaningful results can be obtained through benchmarking the actual implementation.

> ⚠ We again assume in this section that all baker sets, finalizer sets, and their stakes are uniquely determined without referring to the lemmata from Section 13.1.

> ⚠ For simplicity, we here assume that "relevant" messages (e.g., ones that allow parties to form a quorum certificate and advance their round) from honest parties are not ignored by other honest parties. This is nontrivial since the pseudocode and the implementation contain a multitude of checks when receiving new messages to filter out irrelevant and maliciously crafted messages. This partially mitigates denial-of-service attacks and reduces memory consumption. Formally proving that all these checks cannot interfere with liveness is beyond the scope of this section, due to the complexity of the pseudocode and the fact that the actual implementation may include additional and slightly different checks.

We first show that honest bakers and finalizers keep increasing their round numbers. This is implied by Lemma 83 and Lemma 84, where the former shows that if all honest finalizers are in some round, at least one of them moves to the next round, and the latter shows that once one honest finalizer or baker moves to some round, all honest finalizers and bakers also move to that round.

Lemma 83. *Let t be some point in time such that all honest finalizers are in some round at least* $\texttt{round}$ *at that time and assume $\alpha_F \leq 1 - \texttt{sigThreshold}$. Then, at least one honest finalizer is in round* $\texttt{round} + 1$ *by time $t + \max_{\texttt{fid}}\{\texttt{currentTimeout}_{\texttt{fid}}\} + \Delta_{\texttt{net}}$, where* $\texttt{currentTimeout}_{\texttt{fid}}$

corresponds to the current timeout time of finalizer `fid`*.*

Proof. If some of the honest finalizers form a valid quorum certificate for a block in round `round`, they move to `round + 1`. If this is not the case for any honest finalizer by time $t + \max_{\mathtt{fid}}\{\mathtt{currentTimeout}_{\mathtt{fid}}\}$, they all send a timeout message for this round. At most $\Delta_{\mathtt{net}}$ time later, all honest finalizers have received all these timeout messages, so they will all receive timeout message of finalizers with relative stake at least $1 - \alpha_F$ (in the latest epoch). Since we assume $\alpha_F \leq 1 - \mathtt{sigThreshold}$, this is sufficient to form a timeout certificate and move to round `round + 1`. $\qquad\square$

Note that the proof above relies on the assumption that honest finalizers do not ignore timeout messages from other honest finalizers. This follows from our general assumption that honest parties do not ignore "relevant" messages. For instructional purposes, we nevertheless here sketch a proof that this is the case.

First note that honest finalizers do not produce obviously invalid timeout messages, i.e., the finalizer is actually part of the finalization committee and all signatures are valid etc. There are, however, three places where it is not obvious that a timeout message cannot accidentally be ignored. We consider these three places individually.

The first one is the check in uponReceivingTimeout, whether $\texttt{timeout.qc.round} < r$ or $\texttt{timeout.qc.epoch} < e$, where (r, e) is the round and epoch of the last directly finalized block with $r < \texttt{timeout.round} - 1$, in line 9. Note that when an honest finalizer moves to round $\texttt{timeout.round}$, there is either a quorum certificate for $\texttt{timeout.round} - 1$ or a timeout certificate for that round. In case of a quorum certificate, Theorem 82 implies that the last directly finalized block in round r is on the chain to the block certified by the quorum certificate. Hence, the honest finalizer sending the timeout message knows about that finalized block and thus $\texttt{timeout.qc.round} \geq r$ in this case. Using Lemma 77, this also implies $\texttt{timeout.qc.epoch} \geq e$. Now assume the honest finalizer moved to round $\texttt{timeout.round}$ based on a timeout certificate $\texttt{tc}$ for round $\texttt{timeout.round} - 1$ and let $\texttt{finEntry}$ be a finalization entry for the last directly finalized block from round $r < \texttt{timeout.round} - 1$. Note that $\texttt{finEntry.qc}_2\texttt{.round} = r + 1 \leq \texttt{tc.round}$. Using Lemma 76, we obtain that at least one honest finalizer contributed both to $\texttt{tc}$ as well as $\texttt{finEntry.qc}_2$. Furthermore, Lemma 78 implies that the quorum certificate in the timeout message contributing to $\texttt{tc}$ has round at least r. We thus also have $\texttt{timeout.qc.round} \geq r$. As above, this implies $\texttt{timeout.qc.epoch} \geq e$ as well.

The second place is in line 83 of that procedure, which is reached if the timeout contains a quorum certificate such that another quorum certificate exists for the same round but different epoch. Assuming this is the case, we can use Lemma 76 to conclude that at least one honest finalizer has produced a quorum signature for both of these quorum certificates. An honest finalizer, however, never produces two different quorum signatures for the same round, so this cannot happen.

The third and final possible place is in processTimeout, the check whether the epochs of all stored $\texttt{timeouts}$ span strictly more than 2 epochs. If this is the case, all timeouts with epochs smaller than the two highest ones are deleted. By Lemma 80, the epochs of honest finalizers differs by at most 1. Hence, the only way an honest timeout message could be ignored is if honest parties are in epochs $\texttt{epoch}$ and $\texttt{epoch} + 1$, and a dishonest timeout message for $\texttt{epoch} + 2$ or higher has been processed. Since no honest party is in $\texttt{epoch} + 2$, they all tried to catch up before processing the dishonest timeout message. So to reach this point, there must exist a valid epoch finalization proof for $\texttt{epoch} + 1$. If this was first provided to a finalizer in epoch $\texttt{epoch} + 1$, that finalizer would advance to epoch $\texttt{epoch} + 2$ and thus be two epochs further than another finalizer who has not seen this epoch finalization proof, yet. Hence, the existence of this proof contradicts Lemma 80 and we can conclude that this step in processTimeout does not discard timeout messages from honest finalizers.

Lemma 84. *Assume some honest finalizer $\texttt{fid}$ moves to round $\texttt{round}$ at time t. Then, all honest finalizers and bakers move to round at least $\texttt{round}$ by time $t + \Delta_{\texttt{net}} + \texttt{currentTimeout}_{\texttt{fid}} + \texttt{CatchUpTime}$. Furthermore, if an honest baker has produced a valid block for round $\texttt{round}$ by time $t' \geq t$, then all honest finalizers and bakers move to round at least $\texttt{round}$ by time $t' + \Delta_{\texttt{net}} + \texttt{CatchUpTime}$.*

Proof. If finalizer `fid` multicasts a block for round `round`, all other parties receive that block after at most Δ_{net} time. If they are in a lower round, they will advance their round, after potentially catching up to obtain missing information, which takes at most `CatchUpTime` time. If the finalizer instead sends a quorum signature for that round, other parties in lower rounds receiving that signature will start the catch-up procedure to again subsequently advance their rounds. This proves the second part of the claim and the first part for the case that the finalizer `fid` has obtained a valid block by time $t + \text{currentTimeout}_{\text{fid}}$.

If the finalizer `fid` has not produced or received a valid block by time $t + \text{currentTimeout}_{\text{fid}}$, `fid` will multicast a timeout message for round `round`. This will again trigger all other parties in lower rounds to catch up and let them advance their rounds at most $\Delta_{\text{net}} + \text{CatchUpTime}$ time later. $\qquad\square$

> ℹ The previous lemmata show that parties keep increasing their round numbers. This is, however, insufficient for liveness because all rounds could time out. Lemma 84 implies that if the honest leader of a round advances to that round early enough, then all finalizers will also move to that round before timing out. This does not exclude the scenario in which honest finalizers that are not the round leaders continuously advance their rounds first and time out before a block can be produced. We show below that this cannot happen if all honest finalizers are in the same epoch.

> ⚠ One could try to argue that all honest finalizers advance to the next round before timing out using that all messages get relayed by honest parties and thus all messages leading one finalizer to advance their round are also received by all other finalizers within Δ_{net} time, which will make them also advance their rounds. This argument, however, requires that no finalizer ignores any of these messages. This is not necessarily the case for maliciously crafted messages. E.g., if one finalizer is in epoch `epoch` and another one in epoch `epoch + 1`, a corrupted finalizer can send a timeout message for epoch `epoch − 1`, which may allow the former honest finalizer to advance to the next round, but gets ignored by the latter.

> 💡 One way to mitigate the above mentioned issue is to let all finalizers and bakers multicast quorum certificates and timeout certificates once they are formed in a way such that they are not ignored by other honest parties. This would increase the robustness and simplify the analysis at the expense of a higher network traffic.

> ⚠ For the following lemma, we use the refined assumption that if all honest finalizers are in the same epoch, then they do not ignore "different relevant messages". More specifically, we need that in this case, all quorum signatures or timeout messages are either ignored by all honest finalizers or by none.

Lemma 85. *Assume all honest finalizers are in the same epoch and some honest finalizer* `fid` *moves to round* `round` *at time* t. *Then, all honest finalizers and bakers move to round at least* `round` *by time* $t + \Delta_{\text{net}} + \text{CatchUpTime}$.

Proof. The messages that made finalizer `fid` advance to round `round` are relayed to all other honest finalizers and bakers within time Δ_{net}. Using our assumption, they do not ignore these

messages, and after potentially catching up in time at most `CatchUpTime`, they all advance to round `round`. □

We next show that honest finalizers will not remain in different epochs for long.

Lemma 86. *Assume some honest finalizer or baker is in epoch* `epoch` *at some time t. Then, by time* $t + \Delta_{\mathtt{net}} + $ `CatchUpTime`*, there either is an epoch finalization entry for epoch* `epoch`*, or all honest finalizers and bakers are in epoch* `epoch`*.*

Proof. It is clear that if no epoch finalization entry for epoch `epoch` exists by time $t + \Delta_{\mathtt{net}} + $ `CatchUpTime`, no honest party moves to an epoch higher than `epoch`. When some honest finalizer or baker moves to epoch `epoch`, it has received messages leading to an epoch finalization entry for `epoch` − 1. By our assumption on delivery of messages through retransmission, all these messages are also received by all other honest parties within time $\Delta_{\mathtt{net}}$. Hence, they will also move to epoch `epoch` after receiving these messages and potentially catching up in time at most `CatchUpTime`. □

The following lemma shows that honest finalizers sign blocks from honest bakers if they are received before timing out.

Lemma 87. *Assume an honest baker* `baker` *is the leader of some round* `round` *in epoch* `epoch` *and let* `finalizer` *be an honest finalizer. If* `finalizer` *receives the block* `block` *for round* `round` *from* `baker` *at time t and* `finalizer` *does not time out round* `round` *by time* $t + $ `CatchUpTime` *and is in epoch* `block.epoch` *at time t, then* `finalizer` *produces a quorum signature for the block* `block` *by time* $t + $ `CatchUpTime`*.*

Proof. When `finalizer` receives the block `block`, it potentially needs to catch up in time at most `CatchUpTime`. Our assumption on not ignoring relevant messages from other honest parties, `finalizer` does not ignore `block`. Since honest leaders produce valid blocks, `finalizer` consequently produces a quorum signature. □

We can finally use the previous results to show that during (sufficiently long) periods of synchrony, honest bakers continuously produce blocks that get finalized.

Theorem 88 (Informal)**.** *Assume a sufficiently high fraction of honest bakers are chosen as leaders during a sufficiently long period of synchrony and assume* $\alpha_F \leq 1 - $ `sigThreshold`*. Then, there will be some block produced by an honest baker and later finalized.*

Proof sketch. The previous results imply that during a sufficiently long period of synchrony, all honest bakers and finalizers end up in the same epoch and keep increasing their round numbers. Furthermore, when an honest baker is chosen to produce a block for the current round, all honest finalizers move to that round before timing out, and consequently produce a quorum certificate for that block. Hence, that block gets certified and whenever there are two consecutive honest bakers, they will both produce a certified block, which finalizes the first one. □

Note that by Theorem 88, we also have that parties eventually advance their current epoch after the nominal epoch end time t, since finalized blocks after t get produced.

13.4 Fairness of Leader Election

The goal of this section is to prove that during sufficiently long periods of synchrony, the fraction of blocks baked by corrupted parties cannot be significantly higher than the corrupted fraction of all bakers' stake.

In this section, we again assume an adversary $\mathcal{A}$ controls bakers with at most a fraction $\alpha_B < 1$ of the bakers' stake. To be able to use results from Section 13.2.1, we also need that finalizers with at most a fraction $\alpha_F < 1/3$ of the finalizers' stake are corrupted and that `sigThreshold` $\geq 2/3$ (but it is not directly used in the proofs in this section).

Furthermore, all proofs in this section are in the random-oracle model.

13.4.1 Bounding the Adversarial Influence on Leader-Election Nonces

We first show that even if the adversary fully controls several epochs, the adversarial influence on the leader-election nonces is limited.

Lemma 89. *Assume that $\mathcal{A}$ can compute at most C hashes prior to some epoch E. Then, there is a set $\mathcal{N}$ of uniformly distributed leader-election nonces such that $|\mathcal{N}| \leq C$ and the leader-election nonce of epoch $E + 1$ is an adversarially chosen element of $\mathcal{N}$.*

Proof. For a fixed stake distribution and fixed block nonces for blocks in epoch E and before, the leader-election nonce for epoch $E + 1$ is computed by hashing these block nonces and the stake distribution (see getLENonce in Section 11.3.4). Since we assume the hash function acts as a random oracle, the resulting leader-election nonce is uniformly distributed.

Every change to the stake distribution and each way to influence the leader-election nonce via timing out rounds or influencing the number of blocks in an epoch requires at least one hash computation for the adversary to learn the resulting leader-election nonce. Hence, our computational assumption implies that $\mathcal{A}$ can try at most C combinations to obtain different leader election nonces, which are all uniformly distributed, and choose one of them. □

ℹ By May 2023, the all time high hash rate of the whole Bitcoin network was around $500 \cdot 10^{18}$ hashes per second (source: https://www.coinwarz.com/mining/bitcoin/hashrate-chart). This corresponds to roughly 2^{100} hashes in 100 years. It is therefore safe to assume that $C \leq 2^{100}$.

13.4.2 Bounding the Probability of Complete Corruption

We here show that the probability that there is no honest leader in an epoch with at least $R_{\min}$ rounds is negligible in $R_{\min}$.

Lemma 90. *Assume that $\mathcal{A}$ can compute at most C hashes prior to some epoch E and assume there are at least $R_{\min}$ rounds in that epoch. Let T_E be the event that $\mathcal{A}$ controls all leaders in epoch E. Then,*

$$\Pr[T_E] \leq C \cdot \alpha_B^{R_{\min}}.$$

Proof. For a fixed stake distribution and uniformly random leader-election nonce, the probability that a dishonest leader is elected for a given round is at most α_B. Since leaders for different rounds are chosen independently by the random oracle, the probability that all $R_{\min}$ rounds are controlled by $\mathcal{A}$ is at most $\alpha_B^{R_{\min}}$. By Lemma 89, the leader-election nonce for epoch E is one out of at most C uniformly random values chosen by $\mathcal{A}$. Since changing the stake distribution also changes the leader-election nonce, the stake distribution is fixed for each choice of the leader-election nonce. Hence, the union bound implies that $\mathcal{A}$ can boost the probability to control all leader at most by a factor of C. Hence,

$$\Pr[T_E] \leq C \cdot \alpha_B^{R_{\min}}. \qquad \square$$

Corollary 91. *For $R_{\min} \geq \frac{\kappa + \log_2 C}{-\log_2 \alpha_B}$, the probability that $\mathcal{A}$ controls all leaders in an epoch with at least $R_{\min}$ rounds is negligible in κ.*

Proof. By Lemma 90, we have

$$\Pr[T_E] \leq C \cdot \alpha_B^{R_{\min}} \leq C \cdot \alpha_B^{\frac{\kappa + \log_2 C}{-\log_2 \alpha_B}} = C \cdot 2^{-\kappa - \log_2 C} = 2^{-\kappa}. \qquad \square$$

Example: Assume $C \leq 2^{100}$. For $\kappa = 60$ and $\alpha_B = 1/3$, Corollary 91 requires

$$R_{\min} \geq \frac{\kappa + \log_2 C}{-\log_2 \alpha_B} = \frac{160}{\log_2(3)} \approx 100.9.$$

13.4.3 Bounding the Number of Adversarial Leaders

We finally show that the adversary cannot control substantially more leaders than granted by the adversarial stake.

Theorem 92. *Assume that $\mathcal{A}$ can compute at most C hashes prior to some epoch E and consider any set of R rounds in epoch E. Further let $\delta > 0$, let L_R be the number of adversarial leaders chosen for these rounds, and assume $R \geq \frac{\kappa + \log_2(C)}{2\delta^2 \log_2(e)}$. Then,*

$$\Pr[L_R \geq (\alpha_B + \delta) \cdot R] \leq 2^{-\kappa}.$$

Proof. For a fixed stake distribution and uniformly random leader-election nonce, the probability that a dishonest leader is elected for a given round is at most α_B. Hence, the expected number of adversarial leaders in the R considered rounds is at most $\alpha_B R$. Still considering a fixed stake distribution and uniformly random leader-election nonce, we can apply Hoeffding's inequality to obtain that for any $t > 0$, the probability to have at least $\alpha_B R + t$ dishonest leaders is at most

$$e^{-\frac{2t^2}{R}}.$$

As in the proof of Lemma 90, the adversary can boost this probability by at most a factor C by trying different stake distributions and manipulating the leader-election nonce. Hence, we get

$$\Pr[L_R \geq \alpha_B R + t] \leq C \cdot e^{-\frac{2t^2}{R}}.$$

For $t := \delta R$ and using $R \geq \frac{\kappa + \log_2(C)}{2\delta^2 \log_2(e)}$, this implies

$$\Pr[L_R \geq (\alpha_B + \delta) \cdot R] \leq C \cdot e^{-2\delta^2 R} \leq C \cdot e^{-2\delta^2 \frac{\kappa + \log_2(C)}{2\delta^2 \log_2(e)}} = 2^{-\kappa}. \qquad \square$$

Example: For $\delta = 0.16$, $\kappa = 60$, and $C = 2^{100}$, Theorem 92 implies that for $R \geq 2167$, the adversary cannot win more than 16% of the rounds more than the expectation of honest parties with the same stake, except with negligible probability. Assuming $\alpha_B < 1/3$, this in particular means that the adversary cannot control a majority of the rounds in an epoch.
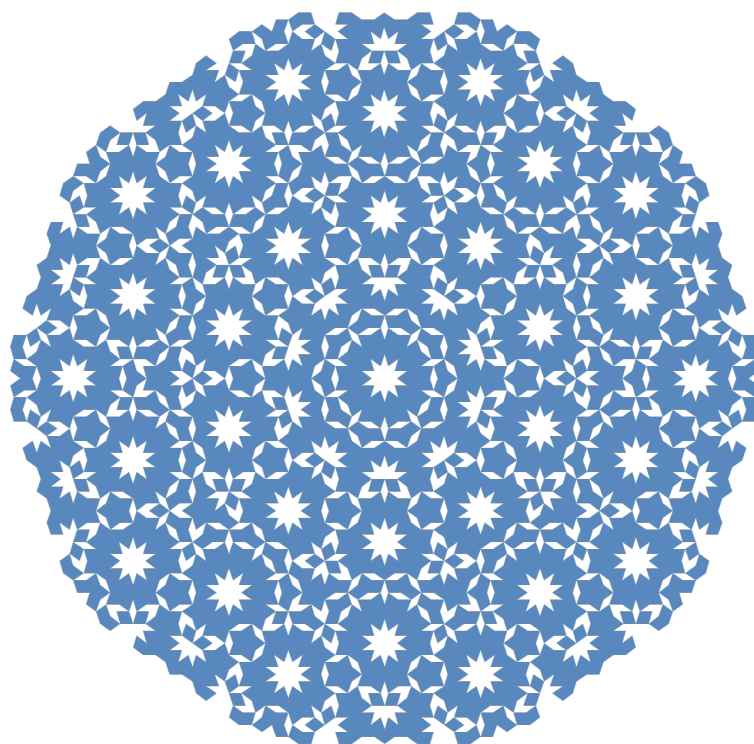
Part III

Personligt and Konto

Illustration: Penrose tiling, with tiling density 30 and a rotation angle of $0°$.

Chapter 14

Overview

This part describes the *identity layer* Personligt and its integration with the account based *execution layer* Konto.

Personligt, in Chapter 15, allows users to create a verifiable *identity credential* off-chain to ease compliance with relevant regulations, while also allowing that identity to be represented on-chain in a way that protects the user's privacy. The security of Personligt has been analyzed in [Dam+21].

Konto, in Chapter 16, mainly concerns with *transactions* between accounts. In combination with Personligt it provides pseudonymity for users, along with a mechanism that allows accountability to local regulations. This means that transactions are processed without exposing the identity of the sender or receiver (apart from the used account). But when a suspicious transaction or set of transactions have been detected, the real-world *identity* of the user can be obtained by qualified authorities with the help of privacy guardians. The same authorities can also help to trace all accounts and transactions of a given real-world identity.

The identity credentials of Personligt are a special case of *verifiable credentials*. In Chapter 17, we describe a *self-sovereign identity* infrastructure called Web3ID that comes with support for verifiable credentials. A verifiable credential system includes three roles: *issuer*, *holder*, and *verifier*. An issuer attests some facts about holders in the form of verifiable credentials. For example, a book club can attest memberships or an organization can attest the completion of a professional program by issuing a diploma.

Finally, Chapter 18 describes *deterministic key derivation* and key management using *seed phrases*.

14.1 Preliminaries

For group $\mathbb{G}_1^{\mathtt{BLS}}$ of the BLS12-381 pairing, we use the commitment key $(\bar{g}_{\mathtt{com}}, \bar{h}_{\mathtt{com}})$ as generators (cf. Section 8.1.5). In particular, we denote $g_1 := g := \bar{g}_{\mathtt{com}}$ and $h := \bar{h}_{\mathtt{com}}$.

Chapter 15

Personligt Identity Layer

This chapter describes the protocols of the identity layer Personligt. As part of the development, a research paper [Dam+21] was published, that provides the necessary security proofs for these protocols.

> **ⓘ** The notation in this chapter has been updated from previous versions to align better with external terminology. The source code of the implementation may contain references to the old terminology (see e.g. the Part) See Table 15.1 for an overview.

Table 15.1 Overview on updated terminology.

New Term	Old Term
public/secret holder identifier	public/secret identity credential.
identity disclosure	anonymity revocation
privacy guardian	anonymity revoker
identity credential	identity object
identity credential issuance	identity registration

15.1 Roles

There are three roles in the Personligt identity layer. Identity providers issue identity credentials to account holders which use the credentials to open accounts. Finally, privacy guardians help to identify account holders if need be.

15.1.1 Account Holder

An *account holder* AH is a private person or legal entity. They get issued an identity credential by an identity provider. With this credential they can open accounts on-chain.

When an account holder interacts with an identity provider, they must have a unique identifier. We use essentially a public key like identifier where the account holder knows the corresponding secret.

Definition 93. For a given *secret holder identifier* $\texttt{IDcred}_{\text{SEC}} \in \text{IDCRED}_{\text{SEC}} := \mathbb{F}_{\text{BLS}}$, the corresponding *public holder identifier* $\texttt{IDcred}_{\text{PUB}} \in \text{IDCRED}_{\text{PUB}} := \mathbb{G}_1^{\text{BLS}}$, is defined as $\texttt{IDcred}_{\text{PUB}} :=$

$g^{\texttt{IDcred}_{\texttt{SEC}}}$.

> ⚠ We require that one can efficiently compute $\texttt{IDcred}_{\texttt{PUB}}$ from $\texttt{IDcred}_{\texttt{SEC}}$ while the opposite should be computationally infeasible. This is the case for the above choice of $\mathrm{IDCRED}_{\texttt{SEC}} := \mathbb{F}_{\texttt{BLS}}$ and $\mathrm{IDCRED}_{\texttt{PUB}} := \mathbb{G}_1^{\texttt{BLS}}$.

> ℹ The above choice of $\mathrm{IDCRED}_{\texttt{SEC}} := \mathbb{F}_{\texttt{BLS}}$ and $\mathrm{IDCRED}_{\texttt{PUB}} := \mathbb{G}_1^{\texttt{BLS}}$ allows the identity provider to blindly sign $\texttt{IDcred}_{\texttt{SEC}}$ given $\texttt{IDcred}_{\texttt{PUB}}$ using the PS signature scheme (cf. Section 5.3.3).

15.1.2 Identity Provider

An identity provider IDP is a private person or legal entity that performs off-chain identification of account holders. After the identification, the IDP issues an identity credential to the account holder which allows opening accounts on the blockchain.

Definition 94. An *identity provider* IDP has the following values assigned.

Public Values

$\texttt{ID}_{\texttt{IDP}} \in \mathrm{IDs}_{\texttt{IDP}} := \mathbb{N}_0$
  The unique *identity provider identifier*.

$\texttt{vk}_{\texttt{IDP}}^{\texttt{ID}} \in \mathrm{VERIFICATIONKEYS}_{\texttt{ID}}$
  The *signature verification key* of the identity provider.

The following values are only known to the identity provider.

Private Values

$\texttt{sk}_{\texttt{IDP}}^{\texttt{ID}} \in \mathrm{SIGNKEYS}_{\texttt{ID}}$
  The *signing key* of the identity provider.

> ⚠ We assume that any party can establish an authenticated (and secure) connection to an identity provider IDP given the identifier $\texttt{ID}_{\texttt{IDP}}$. This can be done using a standard PKI, or simply because the set of such entities is fixed, and their public keys are on-chain. In particular, we assume that given IDP anyone can get the public verification key of the identity provider. For example, we will write $\texttt{ID}_{\texttt{IDP}}.\texttt{vk}_{\texttt{IDP}}^{\texttt{ID}}$ to denote the verification key of the identity provider IDP with identifier $\texttt{ID}_{\texttt{IDP}}$.

> ℹ In a previous version of the bluepaper, the account creation limit $\texttt{max}_{\texttt{AC}}$, the privacy guardian set and thresholds were part of the identity provider attributes and thus publicly known values. This slightly simplified the account opening proof. In the current version, the mentioned attributes are negotiated between account holder and identity provider during identity credential issuance.

> </> On the Concordium mainnet, the identity provider with $\texttt{ID}_{\texttt{IDP}} = 0$ is non-operational and was used to generate the genesis accounts. See Section 22.5 for more details.

15.1.3 Privacy Guardian

A privacy guardian PG is a private person or legal entity. They can help with finding the real-life identity of an account holder or identifying all accounts of a given real-life identity.

In the current design, the account holder and the identity provider agree on a set of n privacy guardians and a threshold $d \leq n$. This ensures that any $d + 1$ of the selected privacy guardians can disclose the account holder's identity, but any d of them have no information.

Definition 95. An *privacy guardian* PG has the following values assigned.

Public Values

$ID_{PG} \in IDs_{PG} := \mathbb{N}^+$
> The unique *privacy guardian identifier*.

$pk_{PG} \in PUBLICKEYS_{PG}$
> The public *encryption key* of the privacy guardian.

The following values are only known to the privacy guardian itself.

Private Values

$dk_{PG} \in SECRETKEYS_{PG}$
> The secret *decryption key* of the privacy guardian.

> ❗ We assume that any party can establish an authenticated (and secure) connection to a privacy guardian PG given the identifier ID_{PG}. This can be done using a standard PKI, or simply because the set of such entities is fixed, and their public keys are on-chain. In particular, we assume that given ID_{PG} anyone can get the public encryption key of the privacy guardian. For example, we will write $ID_{PG}.pk_{PG}$ to denote the public encryption key of the privacy guardian with identifier ID_{PG}.

> </> On the Concordium mainnet, the privacy guardian with $ID_{IDP} = 1$ is non-operational and was used to generate the genesis accounts. See Section 22.5 for more details.

15.2 Identity Credentials

The identity credential IDC, stored by the account holder AH, attests that AH has registered with the identity provider IDP. That is, the identity provider checked the attributes of the account holder. As part of this process, the holder also generate encrypted information for identity disclosure. A zero-knowledge proof is used to ensure that the holder generated the information correctly and that they know the secret holder identifier $IDcred_{SEC}$. See Section 15.2.1 below for more details.

Definition 96. An *identity credential* (IDC) consists of the following values.

$IDcred_{SEC} \in IDCRED_{SEC}$
> A *secret holder identifier*.

$IDcred_{PUB} \in IDCRED_{PUB}$
> A *public holder identifier* where $IDcred_{PUB} := g^{IDcred_{SEC}}$.

$\text{key}_{\text{PRF}} \in \text{PRFKEYS}$

> A *PRF key*.

$\text{m}_0 \in \mathbb{F}_{\text{BLS}}$

> The *signature blinding randomness* used during the identity issuance with the identity provider.

$\text{attrList} \in \mathbb{F}_{\text{BLS}}^{*}$

> A *list of attributes*, such as age, name, or citizenship. Formally, attrList is a vector over field $\mathbb{F}_{\text{BLS}}$, i.e., $\text{attrList} = (a_0, \ldots, a_\ell) \in \mathbb{F}_{\text{BLS}}^{\ell}$. This includes the *creation date* createdAt and the *expiry date* validTo.

$\text{ID}_{\text{IDP}} \in \text{IDS}_{\text{IDP}}$

> The identifier of an identity provider.

$\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_\text{n}} \in \text{IDS}_{\text{PG}}$

> The list of privacy guardians the account holder and the identity provider have agreed on.

$\text{threshold}_{\text{PG}} \in \mathbb{N}^{+}$

> The *identity disclosure threshold* the account holder and the identity provider have agreed on.

$\text{max}_{\text{AC}} \in \mathbb{N}^{+}$

> Denotes the *account credential limit* the account holder and the identity provider have agreed on.

$\sigma_{\text{IDP}} \in \text{SIGNATURES}_{\text{ID}}$

> A signature on $(\text{IDcred}_{\text{SEC}}, \text{key}_{\text{PRF}}, (\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_\text{n}}), \text{threshold}_{\text{PG}}, \text{max}_{\text{AC}}, \text{attrList})$ by ID_{IDP} under $\text{ID}_{\text{IDP}}.\text{sk}_{\text{IDP}}^{\text{ID}}$.

> </> It is in fact possible to create $\text{max}_{\text{AC}} + 1$ account credentials. This is because we used to have "initial accounts" created by the identity provider and the account holder could create max_{AC} "regular" accounts. For backwards compatibility, we still allow $\text{max}_{\text{AC}} + 1$ account credentials in total.

After the identity issuance for an account holder AH, the identity provider stores the account holder identity record. This record is crucial for the identity disclosure process.

Definition 97. The *account holder identity record* ($\text{AHIR} \in \text{AHIRS}$) relative to an identity credential IDC consists of the following values.

$\text{IDcred}_{\text{PUB}} \in \text{IDCRED}_{\text{PUB}}$

> The public holder identifier of the account holder.

$\text{attrList} \in \mathbb{F}_{\text{BLS}}^{*}$

> A list of attributes of the account holder where $\text{attrList} = \text{IDC.attrList}$.

$\sigma'_{\text{IDP}} \in \text{SIGNATURES}_{\text{ID}}$

> A blinded version of $\text{IDC}.\sigma_{\text{IDP}}$, i.e., it holds that $\sigma'_{\text{IDP}} = (a, ba^{\text{IDC.m}_0})$ where $\text{IDC}.\sigma_{\text{IDP}} = (a, b)$.

$\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_\text{n}} \in \text{IDS}_{\text{PG}}$

> A list of privacy guardians (names).

$\text{threshold}_{\text{PG}} \in \mathbb{N}$

> The identity disclosure threshold.

$\mathtt{max_{AC}} \in \mathbb{N}^+$

> The account credential limit.

$\mathtt{PRFkeyEnc_{ACC}} \in \text{CIPHERTEXTS}_{\mathtt{PG}}{}^{\mathtt{n}}$

> The shared encryption of $\mathtt{IDC.key_{PRF}}$ under the public keys $\mathtt{PK_{PG}}$ of the privacy guardians, i.e. $\mathtt{PRFkeyEnc_{ACC}} = \mathtt{Enc_{PG}^{n,d,t,s}}(\mathtt{PK_{PG}}, \mathtt{IDC.key_{PRF}})$ where $\mathtt{d} = \mathtt{threshold_{PG}}$ (cf. Section 7.1.5).

15.2.1 Identity Credential Issuance

During the *identity credential issuance* the identity provider IDP issues an identity credential to account holder AH. In preparation, the account holder needs to generate some values as described below.

Function prepareIDIssuance

Outputs

$\mathtt{IDcred_{PUB}} \in \text{IDCRED}_{\text{PUB}}$

> A public holder identifier.

$\mathtt{IDcred_{SEC}} \in \text{IDCRED}_{\text{SEC}}$

> A secret holder identifier.

$\mathtt{key_{PRF}} \in \text{PRFKEYS}$

> A PRF key.

$\mathtt{m_0} \in \mathbb{F}_{\text{BLS}}$

> Signature blinding randomness.

Implementation

1: Choose $\mathtt{IDcred_{SEC}} \leftarrow \mathbb{F}_{\text{BLS}}$ uniformly at random
2: $\mathtt{IDcred_{PUB}} := g^{\mathtt{IDcred_{SEC}}}$
3: Choose $\mathtt{key_{PRF}} \leftarrow \text{PRFKEYS}$ uniformly at random
4: Choose $\mathtt{m_0} \leftarrow \mathbb{F}_{\text{BLS}}$ uniformly at random
5: **return** $(\mathtt{IDcred_{PUB}}, \mathtt{IDcred_{SEC}}, \mathtt{key_{PRF}}, \mathtt{m_0})$

ℹ️ In Section 18.2.1, we provide a deterministic variant of this function that derives all random values from a seed phrase. This allows recovering identity credentials from the seed phrase if the identity provider permits (cf. Section 18.5.2).

The identity issuance process, a protocol between an account holder AH and an identity provider IDP, works as follows.

Protocol issueIdentity

This is a two party protocol between the account holder AH and identity provider IDP. The account holder knows the public verification key $\mathtt{IDP.vk_{IDP}^{ID}}$ of IDP.

Inputs

$\mathtt{identityInformation} \in \{0, 1\}^*$

> Documentation about the identity attributes held by the account holder, e.g. a passport.

$\mathtt{sk_{IDP}^{ID}} \in \text{SIGNKEYS}_{\text{ID}}$

> The signing key of the identity provider input by the identity provider IDP.

Outputs

IDC $\in$ IDCs

> Identity credential stored by the account holder AH.

AHIR $\in$ AHIRs

> Account holder identity record stored by the identity provider IDP.

Implementation

1. **Preparation:** The account holder AH computes $(\text{IDcred}_{\text{PUB}}, \text{IDcred}_{\text{SEC}}, \text{key}_{\text{PRF}}, \text{m}_0) :=$ prepareIDIssuance() locally.

2. **Identity Verification:** The account holder proves their identity attributes using `identityInformation`. This is done by non-cryptographic means, e.g. showing a passport and taking a selfie. The identity provider creates the attribute list `attrList` including the creation date `createdAt` and the expiry date `validTo`. The list is sent to the account holder.

3. **Privacy Guardian Set:** The account holder and the identity provider agree (via non-cryptographic means) on a set of privacy guardians $\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}$ with public-key set PK_{PG} and identity disclosure threshold $\text{threshold}_{\text{PG}}$.

4. **Account Credential Limit:** The account holder and the identity agree (via non-cryptographic means) on the account credential limit max_{AC}.

5. **PRF Key Encryption:** The account holder sends an encryption $\text{PRFkeyEnc}_{\text{ACC}} :=$ $\text{Enc}_{\text{PG}}^{\text{n,d,t,s}}(\text{PK}_{\text{PG}}, \text{key}_{\text{PRF}})$ of the PRF key key_{PRF} under the public-keys PK_{PG} of the privacy guardians to the identity provider where $\text{d} = \text{threshold}_{\text{PG}}$.

6. **Signing with Data Verification:** The account holder sends $\text{IDcred}_{\text{PUB}}$ to the identity provider. The identity provider then signs

$$\vec{m} := (\text{IDcred}_{\text{SEC}}, \text{key}_{\text{PRF}}, (\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}), \text{threshold}_{\text{PG}}, \text{max}_{\text{AC}}, \text{attrList})$$

blindly under $\text{IDP.sk}_{\text{IDP}}^{\text{ID}}$ using the protocol in Section 5.3.3, i.e.,

$$\sigma_{\text{IDP}} := \text{SignPartUnknownMessage}_{\text{ID}}((\text{m}_0, \vec{m}, \text{vk}_{\text{IDP}}^{\text{ID}}), \text{sk}_{\text{IDP}}^{\text{ID}}),$$

where m_0, $\text{IDcred}_{\text{SEC}}$, and key_{PRF} are not known to the identity provider. The zero-knowledge proof used in $\text{SignPartUnknownMessage}_{\text{ID}}$ is extended to additionally prove that

- $\text{IDcred}_{\text{PUB}} = g^{\text{IDcred}_{\text{SEC}}}$,
- and that $\text{PRFkeyEnc}_{\text{ACC}}$ is a proper encryption of key_{PRF} under PK_{PG}.

Details on the proof are found below in Section 15.2.1. If signing fails, e.g., if the zero-knowledge proof verification fails, the protocol is **aborted**. Additionally, the identity provider checks that this $\text{IDcred}_{\text{PUB}}$ is not yet recorded in its database, and **aborts** if it is. Otherwise, the account holder receives the signature σ_{IDP} and the identity provider the blinded signature σ'_{IDP}.

7. **Output:** The account holder stores

$$\text{IDC} = (\text{IDcred}_{\text{SEC}}, \text{IDcred}_{\text{PUB}}, \text{key}_{\text{PRF}}, \text{m}_0, \text{attrList}, \text{ID}_{\text{IDP}},$$
$$(\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}), \text{threshold}_{\text{PG}}, \text{max}_{\text{AC}}, \sigma_{\text{IDP}}),$$

and the identity provider stores

$$\text{AHIR} = (\text{IDcred}_{\text{PUB}}, \text{attrList}, \sigma'_{\text{IDP}}, (\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}),$$
$$\text{threshold}_{\text{PG}}, \text{max}_{\text{AC}}, \text{PRFkeyEnc}_{\text{ACC}}).$$

The values $\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}$, $\mathtt{d}$, and $\mathtt{max_{AC}}$ are custom to the identity credential. The account holder must thus prove that they are using the correct, i.e., signed, values when opening an account (cf. Section 15.3.3) or adding a credential (cf. Section 15.3.4).

At the moment, the account holder (resp. their wallet) proposes a set of $\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}$ privacy guardians and a threshold $\mathtt{max_{AC}}$. Account holders get information about $\mathtt{PGs}$ and threshold from Concordium service called "wallet proxy". The identity provider can then accept or reject the proposal. The Concordium $\mathtt{IDP}$ software accepts a proposal if it matches the configured $\mathtt{PG}$ set and the threshold is at least 1.

Identity Issuance Zero-Knowledge Proof

Let $\mathtt{IDP.vk_{IDP}^{ID}} = (\widetilde{X}, \left\{\widetilde{Y_i}, Y_i\right\}_{i=1,\ldots,\ell})$ be the signature verification key of the identity provider. The zero-knowledge protocol in the above protocol needs to show that

- the commitment $M' = g^{\mathtt{m_0}} Y_1^{\mathtt{IDcred_{SEC}}} Y_2^{\mathtt{key_{PRF}}}$ used for the blind signature can be opened by the account holder,

- and that the contained $\mathtt{IDcred_{SEC}}$ corresponds to $\mathtt{IDcred_{PUB}}$,

- and that $\mathtt{PRFkeyEnc_{ACC}}$ is a proper encryption of the contained $\mathtt{key_{PRF}}$ under $\mathtt{PK_{PG}}$.

Inputs and formal statement. The zero-knowledge proof has the following inputs and values.

Implicit public values consist of:

- The group $\mathbb{G}_1^{\mathtt{BLS}}$ with order q and generator g,

- the size $\mathtt{s}$ and number $\mathtt{t}$ of parts for ElGamal with data in the exponent using generators $g, \bar{h}$,

- the commitment key $(\bar{g}_{\mathtt{com}}, \bar{h}_{\mathtt{com}})$.

Public input consist of:

- privacy guardians public keys $\mathtt{PK_{PG}} = \mathtt{pk_{PG_1}}, \ldots, \mathtt{pk_{PG_n}}$

- the number of shares $\mathtt{n}$, threshold $\mathtt{d} := \mathtt{threshold_{PG}}$, and points $\alpha_1, \ldots, \alpha_\mathtt{n}$ used for sharing,

- parts Y_1, Y_2 of the PS verification key $\mathtt{vk_{IDP}^{ID}}$,

- M', $\mathtt{IDcred_{PUB}}$, $\mathtt{PRFkeyEnc_{ACC}} = \{c_{i,j}\}_{i\in\{1,\ldots,\mathtt{n}\}, j\in\{1,\ldots,\mathtt{t}\}}$, and $\{C_{i,j}\}_{i\in\{1,\ldots,\mathtt{n}\}, j\in\{1,\ldots,\mathtt{t}\}}$ (auxiliary commitments, see below).

Witness consists of:

- $\mathtt{m}_0$, $\mathtt{IDcred}_{\mathtt{SEC}}$, $\mathtt{key}_{\mathtt{PRF}}$,

- $\hat{a}_1, \ldots, \hat{a}_{\mathtt{d}}$ (sharing randomness),

- $\{k_{i,j}, r_{i,j}\}_{i \in \{1,\ldots,\mathtt{n}\}, j \in \{1,\ldots,\mathtt{t}\}}$ (content and randomness of each $c_{i,j}$),

- and $\{\hat{r}_{i,j}\}_{i \in \{1,\ldots,\mathtt{n}\}, j \in \{1,\ldots,\mathtt{t}\}}$ (randomness of auxiliary commitments).

The proven **statement** is

$$
\begin{aligned}
\mathrm{PK}\Big\{ &\big(\mathtt{m}_0, \mathtt{IDcred}_{\mathtt{SEC}}, \mathtt{key}_{\mathtt{PRF}}, \hat{a}_1, \ldots, \hat{a}_{\mathtt{d}}, r_0, \ldots, r_{\mathtt{d}}, \{k_{i,j}, r_{i,j}, \hat{r}_{i,j}\}_{i \in \{1,\ldots,\mathtt{n}\}, j \in \{1,\ldots,\mathtt{t}\}}\big) : \\
&M' = g^{\mathtt{m}_0} Y_1^{\mathtt{IDcred}_{\mathtt{SEC}}} Y_2^{\mathtt{key}_{\mathtt{PRF}}} \ \wedge\ \mathtt{IDcred}_{\mathtt{PUB}} = g^{\mathtt{IDcred}_{\mathtt{SEC}}} \\
&\wedge\ \forall i \in \{1, \ldots, \mathtt{n}\}\ \big(\mathtt{key}_{\mathtt{PRF}} + \sum_{1 \leq w \leq \mathtt{d}} \hat{a}_w \alpha_i^w\big) =: \mathtt{sh}(\mathtt{key}_{\mathtt{PRF}})_i = \sum_{1 \leq j \leq \mathtt{t}} 2^{\mathtt{s}(j-1)} k_{i,j} \\
&\wedge\ \forall i \in \{1, \ldots, \mathtt{n}\}, j \in \{1, \ldots, \mathtt{t}\}\ c_{i,j} = (g^{r_{i,j}}, \bar{h}^{k_{i,j}} \mathtt{pk}_{\mathtt{PG}_i}^{r_{i,j}}) \\
&\wedge\ \forall i \in \{1, \ldots, \mathtt{n}\}, j \in \{1, \ldots, \mathtt{t}\}\ C_{i,j} = \big(\bar{g}_{\mathtt{com}}^{k_{i,j}} \bar{h}_{\mathtt{com}}^{\hat{r}_{i,j}}\big) \\
&\wedge\ \forall i \in \{1, \ldots, \mathtt{n}\}, j \in \{1, \ldots, \mathtt{t}\}\ k_{i,j} \in [0, 2^{\mathtt{s}} - 1]\Big\}.
\end{aligned}
$$

Proof construction. The zero-knowledge proof is a composition of Σ-protocols and Bulletproofs. We first present the different parts used and then show how to compose them.

- To prove $M' = g^{\mathtt{m}_0} Y_1^{\mathtt{IDcred}_{\mathtt{SEC}}} Y_2^{\mathtt{key}_{\mathtt{PRF}}}$ we can use the Σ-protocol $\mathsf{agg\text{-}dlog}$ (see Section 9.2.2) which can be instantiated to prove

$$
\mathrm{PK}\left\{(x_1, x_2, x_3) : y_1 = g^{x_1} Y_1^{x_2} Y_2^{x_3}\right\}
$$

  where $y_1 := M'$ is the public input and x_1, x_2, x_3 are the witness. All other values form the shared public values $\mathtt{spub}_1$.

  Let $\varsigma_1 = \mathsf{genSigProofInfo}(\mathsf{agg\text{-}dlog}, \mathtt{spub}_1, M')$.

- To prove $\mathtt{IDcred}_{\mathtt{PUB}} = g^{\mathtt{IDcred}_{\mathtt{SEC}}}$ we can use the Σ-protocol dlog (see Section 9.2.1) which can be instantiated to prove

$$
\mathrm{PK}\left\{(x_4) : y_2 = g^{x_4}\right\}
$$

  where $y_2 := \mathtt{IDcred}_{\mathtt{PUB}}$ is the public input and x_4 is the witness. All other values form the shared public values $\mathtt{spub}_2$.

  Let $\varsigma_2 = \mathsf{genSigProofInfo}(\mathsf{dlog}, \mathtt{spub}_2, \mathtt{IDcred}_{\mathtt{PUB}})$.

- The statement $\forall i \in \{1, \ldots, \mathtt{n}\}\ (\mathtt{key}_{\mathtt{PRF}} + \sum_{1 \leq w \leq \mathtt{d}} \hat{a}_w \alpha_i^w) =: \mathtt{sh}(\mathtt{key}_{\mathtt{PRF}})_i = \sum_{1 \leq j \leq \mathtt{t}} 2^{\mathtt{s}(j-1)} k_{i,j}$ shows that the account holder encrypted a correct sharing of $\mathtt{key}_{\mathtt{PRF}}$. To prove this statement, we can use the homomorphic property of ciphertexts and compute for each i, $c_i = \mathsf{JoinEncExp}_{\mathsf{EG}}^{\mathtt{t},\mathtt{s}}(c_{i,1}, \ldots, c_{i,\mathtt{t}})$ which by definition contains $\sum_{1 \leq j \leq \mathtt{t}} 2^{\mathtt{s}(j-1)} k_{i,j}$. In the following we generate a Σ-protocol for the statement

$$
\begin{aligned}
\mathrm{PK}\Big\{ &\big(x_5, \hat{a}_1, \ldots, \hat{a}_{\mathtt{d}}, \{r_{i,j}\}_{1 \leq i \leq \mathtt{n}, 1 \leq j \leq \mathtt{t}}\big) : \\
&\forall i\ c_i = \left(g^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)} r_{i,j}}, \bar{h}^{x_5 + \sum_{w=1}^{\mathtt{d}} \hat{a}_w \alpha_i^w} \mathtt{pk}_{\mathtt{PG}_i}^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)} r_{i,j}}\right)\Big\}.
\end{aligned}
$$

  First, use a combination of dlog and $\mathsf{agg\text{-}dlog}$ to get Σ-protocol $\hat{\varsigma}_i$ for the statement

$$
\mathrm{PK}\left\{(r_i, t_i) : c_i = (g^{r_i}, \bar{h}^{t_i} \mathtt{pk}_{\mathtt{PG}_i}^{r_i})\right\}.
$$

Now apply $\mathsf{genLinRelComp}$ twice replacing r_i by $\sum_{1\leq j\leq \mathtt{t}} 2^{\mathtt{s}(j-1)}r_{i,j}$ and t_i by $x_{i,5}+\sum_{w=1}^{\mathtt{d}} \hat{a}_{i,w}\alpha_i^w$ resulting in Σ-protocol $\hat{\varsigma}_i'$ for the statement

$$\mathrm{PK}\Big\{(x_{i,5},\hat{a}_{i,1},\ldots,\hat{a}_{i,\mathtt{d}},\{r_{i,j}\}_{1\leq j\leq \mathtt{t}}):$$

$$c_i = \left(g^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)}r_{i,j}}, \bar{h}^{x_{i,5}+\sum_{w=1}^{\mathtt{d}}\hat{a}_{i,w}\alpha_i^w}\mathsf{pk}_{\mathsf{PG}_i}^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)}r_{i,j}}\right)\Big\}.$$

Next, apply $\mathsf{genAndComp}$ to get $\varsigma_3' = \mathsf{genAndComp}(\hat{\varsigma}_1',\ldots,\hat{\varsigma}_{\mathtt{n}}')$ for the statement

$$\mathrm{PK}\Big\{(\{x_{i,5},\hat{a}_{i,1},\ldots,\hat{a}_{i,\mathtt{d}},r_{i,j}\}_{1\leq i\leq \mathtt{n},1\leq j\leq \mathtt{t}}):$$

$$\forall i\ c_i = \left(g^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)}r_{i,j}}, \bar{h}^{x_{i,5}+\sum_{w=1}^{\mathtt{d}}\hat{a}_{i,w}\alpha_i^w}\mathsf{pk}_{\mathsf{PG}_i}^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)}r_{i,j}}\right)\Big\}.$$

Finally, apply $\mathsf{genEqComp}$ to replace $\hat{a}_{i,j}$ by $\hat{a}_j$ and $x_{i,5}$ by x_5 for all i resulting in ς_3 for statement

$$\mathrm{PK}\Big\{(x_5,\hat{a}_1,\ldots,\hat{a}_{\mathtt{d}},\{r_{i,j}\}_{1\leq i\leq \mathtt{n},1\leq j\leq \mathtt{t}}):$$

$$\forall i\ c_i = \left(g^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)}r_{i,j}}, \bar{h}^{x_5+\sum_{w=1}^{\mathtt{d}}\hat{a}_w\alpha_i^w}\mathsf{pk}_{\mathsf{PG}_i}^{\sum_{j=1}^{\mathtt{t}} 2^{\mathtt{s}(j-1)}r_{i,j}}\right)\Big\}.$$

> ⓘ The c_i can be computed directly from the $c_{i,j}$'s using $\mathsf{JoinEncExp}_{\mathsf{EG}}^{\mathtt{t},\mathtt{s}}$. As the $c_{i,j}$ are already part of the public input there is no need to store the c_i's explicitly.

- The statement $\forall i \in \{1,\ldots,\mathtt{n}\}, j \in \{1,\ldots,\mathtt{t}\}\ c_{i,j} = (g^{r_{i,j}}, \bar{h}^{k_{i,j}}\mathsf{pk}_{\mathsf{PG}_i}^{r_{i,j}}) \wedge k_{i,j} \in [0, 2^{\mathtt{s}} - 1]$ shows that the account holder did the splitting for the encryption with data in the exponent correctly, i.e., that each part is small enough.

  To prove the statement the account holder must generate auxiliary Pedersen commitments $\{C_{i,j}\}_{i\in\{1,\ldots,\mathtt{n}\},j\in\{1,\ldots,\mathtt{t}\}}$ which should contain the same values as the ciphertexts. The actual proof consists of a Σ-protocol showing that the commitments contain the same values as the ciphertexts and a bulletproof showing the range part given the commitments.

  For the Σ-protocol we start with $\mathsf{com\text{-}enc\text{-}eq}$ for proving that commitment $C_{i,j}$ contains the same value as $c_{i,j}$, i.e.,

$$\mathrm{PK}\left\{(r_{i,j}', k_{i,j}, \hat{r}_{i,j}) : c_{i,j} = \left(g^{r_{i,j}'}, \bar{h}^{k_{i,j}}\mathsf{pk}_{\mathsf{PG}_i}^{r_{i,j}'}\right) \wedge C_{i,j} = \left(\bar{g}_{\mathsf{com}}^{k_{i,j}}\bar{h}_{\mathsf{com}}^{\hat{r}_{i,j}}\right)\right\},$$

where $c_{i,j}$ and $C_{i,j}$ are public input and $r_{i,j}', k_{i,j}, \hat{r}_{i,j}$ are the witness. All other values form the shared public values $\mathtt{spub}_4$ (they are independent of i,j).

Denote by $\varsigma_4 = \mathsf{genAndComp}\left(\mathsf{genSigProofInfo}\left(\mathsf{com\text{-}enc\text{-}eq}, \mathtt{spub}_4, (c_{i,j}, C_{i,j})\right)_{\forall i,j}\right)$ the result of AND-combining all the $\mathsf{com\text{-}enc\text{-}eq}$ Σ-protocols. The proof ς_4 shows that

$$\mathrm{PK}\left\{\left(\{r_{i,j}', k_{i,j}, \hat{r}_{i,j}\}_{1\leq i\leq \mathtt{n},1\leq j\leq \mathtt{t}}\right) : \forall i \forall j\ c_{i,j} = \left(g^{r_{i,j}'}, \bar{h}^{k_{i,j}}\mathsf{pk}_{\mathsf{PG}_i}^{r_{i,j}'}\right) \wedge C_{i,j} = \left(\bar{g}_{\mathsf{com}}^{k_{i,j}}\bar{h}_{\mathsf{com}}^{\hat{r}_{i,j}}\right)\right\}.$$

We can then use an aggregate bulletproof to prove the statement:

$$\mathrm{PK}\left\{((k_{i,j}, \hat{r}_{i,j})_{\forall i,j}) : \forall i \in \{1,\ldots,\mathtt{n}\}\ \forall j \in \{1,\ldots,\mathtt{t}\}\quad C_{i,j} = \left(\bar{g}_{\mathsf{com}}^{k_{i,j}}\bar{h}_{\mathsf{com}}^{\hat{r}_{i,j}}\right) \wedge k_{i,j} \in [0, 2^{\mathtt{s}} - 1]\right\}.$$

Denote by $\mathtt{BPInfo} = \mathsf{aggregateBPInfo}\Big(\mathsf{genBPInfo}\big(\mathtt{s}, (\bar{g}_{\mathtt{com}}, \bar{h}_{\mathtt{com}}), (C_{i,j})\big)_{\forall i,j}\Big)$ the resulting bulletproof information.

> **ⓘ** One can save on $\mathtt{t}$ commitments (and parts of the ς_4 protocol) by using $(\bar{g}_{\mathtt{com}}, \bar{h}_{\mathtt{com}}) :=$
> $(\bar{h}, \mathtt{pk}_{\mathtt{PG}_1})$. In this case the ciphertexts $c_{1,j}$ already contain commitments to the $k_{1,j}$.

We can now combine the parts as follows.

1. Combining $(\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4)$ using the $\mathsf{genAndComp}$ operator results in ς' with witness

$$(x_1, x_2, x_3, x_4, x_5, \hat{a}_1, \dots, \hat{a}_{\mathtt{d}}, \left\{ r_{i,j}, r'_{i,j}, k_{i,j}, \hat{r}_{i,j} \right\}_{1 \le i \le \mathtt{n}, 1 \le j \le \mathtt{t}}).$$

2. Applying the $\mathsf{genEqComp}$ operator several times setting $x_2 =: x_4$, $x_3 =: x_5$, $r'_{i,j} =: r_{i,j}$, and then renaming $x_1 := \mathtt{m}_0$, $x_2 := \mathtt{IDcred}_{\mathtt{SEC}}$, $x_3 := \mathtt{key}_{\mathtt{PRF}}$ results in ς with witness

$$(\mathtt{m}_0, \mathtt{IDcred}_{\mathtt{SEC}}, \mathtt{key}_{\mathtt{PRF}}, \hat{a}_1, \dots, \hat{a}_{\mathtt{d}}, \{r_{i,j}, k_{i,j}, \hat{r}_{i,j}\}_{1 \le i \le \mathtt{n}, 1 \le j \le \mathtt{t}}).$$

The zero-knowledge proof is thus described by ς and $\mathtt{BPInfo}$. To generate the non-interactive proof the account holder uses $\mathsf{genSigmaANDBulletproof}$. As context string, we use

$$\mathtt{ctx} := \text{``regAccHolder''} \parallel \mathsf{H}(\text{genesis block}).$$

To verify the proof the identity provider uses $\mathsf{verifySigmaANDBulletproof}$.

> **ⓘ** The identity provider's verification key $\mathtt{vk}_{\mathtt{IDP}}^{\mathtt{ID}}$ is part of the public input and must be hashed into the proof challenge. We do not need to worry about replay attacks as the $\mathtt{IDP}$ can detect them (and only sign the first time $\mathtt{IDcred}_{\mathtt{PUB}}$ has been presented). Therefore, the choice of the context string $\mathtt{ctx}$ is not particularly important, also as the protocol is executed only between the user and the identity provider and does not relate to the blockchain directly.

> **</>** In the implementation the context string $\mathtt{ctx}$ does not contain $\mathsf{H}(\text{genesis block})$. Instead, the hash of the genesis cryptographic parameter and the genesis string are used.

15.3 Accounts and Account Credentials

Accounts are used to interact with the execution layer of the blockchain. For example, an account can hold native token or make a smart contract call. An account is controlled by a (non-empty) set of account holders which are represented by their account credentials.

15.3.1 Account Credential

An account credential represents an account holder and contains signature verification keys, information related to identity disclosure and the account holder's identity attributes. An account holder can create account credentials using their identity credential.

Definition 98. A (public) *account credential* (AC) associated with identity credential IDC consists of the following values.

$\text{ID}_{\text{AC}} \in \text{ACIDs} := \mathbb{G}_1^{\text{BLS}}$

The *account credential identifier* is the unique identifier of the account credential. It is generated as $\text{ID}_{\text{AC}} := \text{PRF}(\text{key}_{\text{PRF}}, x)$ where $\text{key}_{\text{PRF}} = \text{IDC.key}_{\text{PRF}}$ is the PRF-key of IDC. The account credential is the x'th account credential created by the account holder based on the given IDC.

$\text{vk}_{\text{ACC}1}, \ldots, \text{vk}_{\text{ACC}k} \in \text{VERIFICATIONKEYS}_{\text{ACC}}{}^k$

The *signature verification keys*.

$\text{threshold}_{\text{Cred}} \in \mathbb{N}^+$

The *signature threshold*, i.e., the number of keys required to produce a valid multi-signature in the name of this credential. It is required that $\text{threshold}_{\text{Cred}} \le k$ where k is the number of signature keys of the credential.

$C_0, \ldots, C_\ell \in (\mathbb{G}_1^{\text{BLS}})^\ell$

The *attribute commitments*, i.e., Pedersen commitments (cf. Section 8.1) to attributes in $\text{IDC.attrList} = (a_1, \ldots, a_\ell)$ where $C_i = \bar{g}_{\text{com}}^{r_i} \bar{h}_{\text{com}}^{a_i}$.

$\text{publicAttrPredicate} \in \{0,1\}^*$

The *public attribute predicate* the account holder wants to reveal.

$\text{ID}_{\text{IDP}} \in \text{IDS}_{\text{IDP}}$

The identity provider identifier of the identity provider who signed the IDC of the account holder.

$\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n} \in \text{IDS}_{\text{PG}}$

The list of privacy guardians as in IDC, i.e., $\text{ID}_{\text{PG}_i} := \text{IDC.ID}_{\text{PG}_i}$.

$\text{threshold}_{\text{PG}} \in \mathbb{N}$

The PG threshold as in IDC, i.e., $\text{threshold}_{\text{PG}} := \text{IDC.threshold}_{\text{PG}}$.

$\text{PGdata}_{\text{ACC}} \in \text{CIPHERTEXTS}_{\text{PG}}{}^n$

The *identity disclosure data* is the shared encryption of the account holders's public identity credential $\text{IDC.IDcred}_{\text{PUB}}$, i.e., $\text{PGdata}_{\text{ACC}} = \text{Enc}_{\text{PG}}^{\text{n,d}}(\text{PK}_{\text{PG}}, \text{IDC.IDcred}_{\text{PUB}})$ where PK_{PG} is the public-key set of the privacy guardians $\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}$ where $\text{d} = \text{IDC.threshold}_{\text{PG}}$.

⚠ An account credential can be used for at most one account. Re-using account credentials would allow to circumvent the account credential limit $\text{IDC.max}_{\text{AC}}$. The protocol *enforces* the uniqueness of account credential identifiers.

❗ If the account holder behave honestly, then ID_{AC} is unique for the account credential and $x \le \text{max}_{\text{AC}}$ where max_{AC}—the maximal account credential number—is part of the signed identity credential IDC. The uniqueness of ID_{AC} can be checked publicly while the $x \le \text{max}_{\text{AC}}$ condition is enforced by a zero-knowledge proof which must be part of the deployment transaction.

ℹ By design the attribute commitments are unconditionally hiding. This prevents "harvest now, break later" types of attacks as even an unbounded adversary cannot extract the attributes from the commitments.

> 💡 To safe space, one could use a single vector commitment, instead having a list of single-value commitments (cf. Section 8.1.4)

The information to control an account credential is stored in the private account credential. It consists of the account credential number, the signature keys, and the commitment randomness for identity attributes from `IDC`.

Definition 99. The *private account credential* ($\text{PAC} \in \text{PACs}$) relative to an `AC` and an `IDC` consists of

$x \in \mathbb{N}$
  The *account credential number*.

$\text{sk}_{\text{ACC}1}, \ldots, \text{sk}_{\text{ACC}k} \in \text{SIGNKEYS}_{\text{ACC}}{}^k$
  The *signature keys*.

$r_1, \ldots, r_\ell \in (\mathbb{F}_{\text{BLS}})^\ell$
  The *commitment randomness* for the attribute commitments.

> ℹ️ Note that account credentials are published on-chain as part of accounts. We therefore explicitly define the private account credential which contains the confidential information related to the account credential. In contrast, there is no "private" identity credential as identity credentials are assumed to be confidential.

For ease of notation we define the "signature" of an account credential as follows.

Definition 100. A message $m \in \{0,1\}^*$ is *signed* by account credential `AC` if there are signatures on m from at least $\text{AC.threshold}_{\text{Cred}}$ (different) keys in `AC`. We denote this set of signatures as *account credential signature* σ_{AC}.

An account holder can create an account credential `AC` and the corresponding private account credential `PAC` from an identity credential `IDC` as follows.

Function createCredential

Inputs
  $\text{IDC} \in \text{IDCs}$
    The *identity credential* of the account holder.
  $x \in \mathbb{N}$
    The *account credential number*. It should be unique among all account credentials created using `IDC`. It should also be less than or equal to $\text{IDC.max}_{\text{AC}}$.
  $k \in \mathbb{N}$
    The number of signature keys used in the account credential.
  $\text{threshold}_{\text{Cred}} \in \mathbb{N}$
    The *signature threshold*. It must hold $\text{threshold}_{\text{Cred}} \leq k$.
  $\text{publicAttrPredicate} \in \{0,1\}^*$
    A predicate on the attributes the account holder chooses to reveal.

Outputs
  $\text{AC} \in \text{ACs}$
    The account credential.

PAC ∈ PACs
> The private account credential.

Implementation
1. **Generate Account Credential ID:** Compute $\text{ID}_{\text{AC}} := \text{PRF}(\text{IDC.key}_{\text{PRF}}, x)$.
2. **Generate Credential Keys:** Generate k signature key pairs using $\text{KeyGen}_{\text{ACC}}$, i.e.,
   $(\text{sk}_{\text{ACC}i}, \text{vk}_{\text{ACC}i})_{i \in \{1,\ldots,k\}} = \text{KeyGen}_{\text{ACC}}$.
3. **Generate Attribute Commitments:** For each attribute a_i in IDC.attrList, choose
   $r_i \leftarrow \mathbb{F}_{\text{BLS}}$ uniformly at random and compute Pedersen commitments $C_i := \bar{g}_{\text{com}}^{r_i} \bar{h}_{\text{com}}^{a_i}$.
4. **Identity Disclosure Data:** Compute the identity disclosure data $\text{PGdata}_{\text{ACC}} := \text{Enc}_{\text{PG}}^{\text{n,d}}(\text{PK}_{\text{PG}}, \text{IDcred}_{\text{PUB}})$, where PK_{PG} is the public-key set of the privacy guardians
   $\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}$ from IDC, and $d = \text{IDC.threshold}_{\text{PG}}$.
5. **Output:** The account holder stores

$$\begin{aligned} \text{AC} := \big(&\text{ID}_{\text{AC}}, (\text{vk}_{\text{ACC}1}, \ldots, \text{vk}_{\text{ACC}k}), \text{threshold}_{\text{Cred}}, \\ &(C_0, \ldots, C_\ell), \text{publicAttrPredicate}, \\ &\text{IDC.ID}_{\text{IDP}}, \text{IDC.}(\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}), \text{IDC.threshold}_{\text{PG}}, \text{PGdata}_{\text{ACC}}\big) \end{aligned}$$

and

$$\text{PAC} := \big(x, (\text{sk}_{\text{ACC}1}, \ldots, \text{sk}_{\text{ACC}k}), (r_1, \ldots, r_\ell)\big).$$

> ⚠ When opening an account or adding a credential it must be checked that the attributes in IDC satisfy the predicate `publicAttrPredicate`.

> ℹ In Section 18.2.1, we provide a deterministic variant of this function where all random values are derived from a seed phrase. This allows to recover account credentials from a seed phrase and public information on the blockchain.

15.3.2 Account Information

The account information defines an on-chain account. The idea is that the account is controlled by a set of account holders where each of them is represented by an account credential. Thus, an account is essentially a set of account credentials with a threshold that indicates the number of credentials needed to create a valid account signature (e.g., on a transaction). We do not include information from the execution layer, e.g., account balances, in the account information (at least for the description of Personligt).

Definition 101. The *account information* (`AI`) consists of the following values.

$\text{RegID}_{\text{ACC}} \in \textsc{RegAccIDs} := \textsc{ACIDs}$
> The *account registration identifier* is the unique identifier of the account. It is the identifier of the first ever account credential of the account. See below for more details.

$\text{AC}_1, \ldots, \text{AC}_k \in \textsc{ACs}^*$
> The *account credentials* of the account.

$\text{threshold}_{\text{ACC}} \in \mathbb{N}$
> The *account signature threshold*. It must hold that $\text{threshold}_{\text{ACC}} \leq k$.

$\mathrm{pk_{ACC} \in PUBLICKEYS_{ACC}}$

> The *account public encryption key* used to receive shielded amounts.

The private account information, held jointly be the account holders, is the following.

Definition 102. The *private account information* ($\mathrm{PAI \in PAIs}$) consists of the following values:

$\mathrm{PAC_1, \ldots, PAC_k \in PACs^*}$

> The *private account credential* of the corresponding account credentials.

$\mathrm{dk_{ACC} \in SECRETKEYS_{ACC}}$

> The *account private decryption key* used to receive shielded amounts.

⚠️ The decryption key is currently linked to the PRF key of the identity credential of the first ever account credential ($\mathrm{dk_{ACC}} = \frac{1}{\mathrm{key_{PRF}} + x}$), Anyone knowing that key can thus link all account credentials of that account holder.

</> With Protocol 7, the account en-/decryption keys are considered a deprecated feature. The protocol will only allow to decrypt shielded balances, see Section 16.3 for more details.

In versions before, the encryption key had no functionality if the account contained multiple credentials. An account that used shielded amounts could not be converted into a multi-credential account.

❗ The protocol, as described below, uses $\mathrm{dk_{ACC}} = \frac{1}{\mathrm{key_{PRF}} + x}$ and thus $\mathrm{pk_{ACC}} = \mathrm{RegID_{ACC}}$ (cf. Section 6.1). This allows the privacy guardians to decrypt shielded transfers since they can recover $\mathrm{key_{PRF}}$ and guess x (as it has to be small). This also means, that $\mathrm{pk_{ACC}}$ does not need to be stored separately from $\mathrm{RegID_{ACC}}$. It is mentioned explicitly in AI as a future version may use encryption keys that are independent of $\mathrm{RegID_{ACC}}$. However, such a separation would require additional data to be encrypted as part of the identity disclosure data.

The account holders can jointly act by signing a transaction with their account credentials. The required number of account credential signatures is defined by the account signature threshold. Formally, an account signature is a set of sets of signatures, i.e., an element of $(\mathrm{SIGNATURES_{ACC}}^*)^*$.

Definition 103. A message m is *signed* by an account with account information AI if there is a set of valid signatures from (pairwise different) account credentials of AI of size at least $\mathrm{AI.threshold_{ACC}}$. We denote this set of signatures the *account signature* $\sigma_{\mathrm{account}} \coloneqq (\mathrm{SIGNATURES_{ACC}}^*)^*$.

The account information can be generated from the account credentials as follows.

Protocol generateAI

Inputs

$\mathrm{AC_1, \ldots, AC_k \in ACs^*}$

> The account credentials of the account. The first one is used to generate the account registration ID and the encryption key pair. They must be fresh, i.e., have not been deployed before.

$\texttt{PAC}_1, \ldots, \texttt{PAC}_k \in \text{PACs}^*$

> The private account credentials of the account. Here $\texttt{PAC}_i$ must be the private counterpart of $\texttt{AC}_i$.

$\texttt{IDC}_1 \in \text{IDCs}$

> The identity credential related to $\texttt{AC}_1$.

$\texttt{threshold}_{\texttt{ACC}} \in \mathbb{N}$

> The account threshold. It must hold that $\texttt{threshold}_{\texttt{ACC}} \leq k$.

Outputs

$\texttt{AI} \in \text{AIs}$

> Account information.

$\texttt{PAI} \in \text{PAIs}$

> Private account information.

Implementation

1. **Generate Account Registration ID:** Compute $\texttt{RegID}_{\texttt{ACC}} := \texttt{AC}_1.\texttt{ID}_{\texttt{AC}}$.
2. **Generate Account Decryption Key:** Set the account encryption key pair $(\texttt{dk}_{\texttt{ACC}}, \texttt{pk}_{\texttt{ACC}}) := \left(\frac{1}{\texttt{IDC}_1.\texttt{key}_{\texttt{PRF}} + \texttt{PAC}_1.x}, \texttt{RegID}_{\texttt{ACC}}\right)$ where $\texttt{IDC}_1$ is the identity credential corresponding to $\texttt{AC}_1$.
3. **Output:** The account information

$$\texttt{AI} := \big(\texttt{RegID}_{\texttt{ACC}}, (\texttt{AC}_1, \ldots, \texttt{AC}_k), \texttt{threshold}_{\texttt{ACC}}, \texttt{pk}_{\texttt{ACC}}\big)$$

and the private account information

$$\texttt{PAI} := (x, (\texttt{PAC}_1, \ldots, \texttt{PAC}_k), \texttt{dk}_{\texttt{ACC}}).$$

15.3.3 Open New Account

The following protocol allows an account holder $\texttt{AH}$ with identity credential $\texttt{IDC}$, account information $\texttt{AI}$, and private account information $\texttt{PAI}$ to open a new account on-chain.

> ❗ At the moment, account can only be opened with a single credential. Further credentials can be added after the account has been opened (cf. Section 15.3.4). In the future, we may consider supporting account openings with several credentials.

Protocol openAccount

Inputs

$\texttt{IDC} \in \text{IDCs}$

> The identity credential of the account holder.

$\texttt{AI} = (\texttt{RegID}_{\texttt{ACC}}, \texttt{AC}, \texttt{threshold}_{\texttt{ACC}} = 1, \texttt{pk}_{\texttt{ACC}} = \texttt{RegID}_{\texttt{ACC}}) \in \text{AIs}$

> The account information consisting of a single account credential $\texttt{AC}$.

$\texttt{PAI} = (\texttt{PAC}, \texttt{dk}_{\texttt{ACC}}) \in \text{PAIs}$

> The corresponding private account information consisting of a single private account credential.

The output defined below is the transaction submitted on-chain to open the account.

Outputs

$\mathtt{AI} \in \mathrm{AIs}$
    The input account information.

$\pi_{\mathtt{ACC}} \in \Sigma\text{-}\mathrm{BULLETPROOFS}$
    Non-interactive zero-knowledge proof of *account validity*.

$\mathtt{Auxdata} \in \{0,1\}^*$
    Auxiliary data for the zero-knowledge proof.

$\sigma_{\mathtt{AC}} \in \mathrm{SIGNATURES}_{\mathtt{ACC}}^*$
    Signature from account credential $\mathtt{AI.AC}$ on $(\mathtt{AI}, \pi_{\mathtt{ACC}}, \mathtt{Auxdata})$.

Implementation

1. **Proving Account Validity:** The account holder $\mathtt{AH}$ produces a non-interactive zero-knowledge proof $\pi_{\mathtt{ACC}}$ with auxiliary data $\mathtt{Auxdata}$ for the statement:

   (a) The account holder knows a valid signature, $\mathtt{IDC}.\sigma_{\mathtt{IDP}}$, under $\mathtt{AC.ID}_{\mathtt{IDP}}.\mathtt{vk}_{\mathtt{IDP}}^{\mathtt{ID}}$ for $\mathtt{IDC}$, i.e., on

   $$(\mathtt{IDC.IDcred}_{\mathtt{SEC}}, \mathtt{IDC.key}_{\mathtt{PRF}}, (\mathtt{IDC.ID}_{\mathtt{PG}_1}, \ldots, \mathtt{IDC.ID}_{\mathtt{PG}_n}),$$
   $$\mathtt{IDC.threshold}_{\mathtt{PG}}, \mathtt{IDC.max}_{\mathtt{AC}}, \mathtt{IDC.attrList}),$$

   (b) and $\mathtt{AC.ID}_{\mathtt{AC}} = \mathsf{PRF}(\mathtt{IDC.key}_{\mathtt{PRF}}, \mathtt{PAC}.x)$ where $\mathtt{PAC}.x \leq \mathtt{IDC.max}_{\mathtt{AC}}$,

   (c) and $(\mathtt{AC.ID}_{\mathtt{PG}_1}, \ldots, \mathtt{AC.ID}_{\mathtt{PG}_n}) = (\mathtt{IDC.ID}_{\mathtt{PG}_1}, \ldots, \mathtt{IDC.ID}_{\mathtt{PG}_n})$,

   (d) and $\mathtt{AC.threshold}_{\mathtt{PG}} = \mathtt{IDC.threshold}_{\mathtt{PG}}$,

   (e) and ciphertext $\mathtt{AC.PGdata}_{\mathtt{ACC}}$ is a valid encryption of $\mathtt{IDC.IDcred}_{\mathtt{PUB}}$, i.e., $\mathtt{AC.PGdata}_{\mathtt{ACC}} = \mathsf{Enc}_{\mathtt{PG}}^{\mathsf{n,d}}(\mathtt{PK}_{\mathtt{PG}}, \mathtt{IDC.IDcred}_{\mathtt{PUB}})$, where $\mathtt{PK}_{\mathtt{PG}}$ is the public-key set of the privacy guardians from $\mathtt{AC}$ and $\mathtt{d} = \mathtt{AC.threshold}_{\mathtt{PG}}$,

   (f) and that the attribute commitments in $\mathtt{AC}$ are well-formed, i.e., $\forall i\ \mathtt{AC}.C_i = \bar{g}_{\mathtt{com}}^{r_i} \bar{h}_{\mathtt{com}}^{a_i}$ where r_i is the i-th entry in $\mathtt{PAC}.(r_0, \ldots, r_\ell)$ and a_i is the i-th entry in $\mathtt{IDC.attrList}$,

   (g) and the identity credential attributes $\mathtt{IDC.attrList}$ satisfy the public predicate, i.e., $\mathtt{AC.publicAttrPredicate}(\mathtt{IDC.attrList}) = \mathtt{true}$.

   Details on the zero-knowledge proof are found in Section 15.3.3.

2. **Signing:** The account holder generates $\sigma_{\mathtt{AC}}$ by signing $(\mathtt{AI}, \pi_{\mathtt{ACC}}, \mathtt{Auxdata})$ using the signing key in the private account credential $\mathtt{PAC}$.

3. **Output:** To open the account on chain, $(\mathtt{AI}, \pi_{\mathtt{ACC}}, \mathtt{Auxdata}, \sigma_{\mathtt{AC}})$ is submitted in a special account-opening transaction. See Section 24.4.1 for details on the serialization.

🛈 There is no need to prove that account encryption key $\mathtt{pk}_{\mathtt{ACC}}$ is well-formed as we assume $\mathtt{pk}_{\mathtt{ACC}} = \mathtt{RegID}_{\mathtt{ACC}}$.

</> The current implementation only supports predicates that reveal attributes.

Verification of Account Opening Transaction

To verify an account opening transactions, validators use the following protocol.

Inputs

$\mathtt{AI} = (\mathtt{RegID_{ACC}}, \mathtt{AC}, \mathtt{threshold_{ACC}} = 1, \mathtt{pk_{ACC}}) \in \mathrm{AIs}$
  Account information of the new account.

$\pi_{\mathtt{ACC}} \in \Sigma\text{-}\mathrm{BULLETPROOFS}$
  Non-interactive zero-knowledge proof of account validity.

$\mathtt{Auxdata} \in \{0,1\}^*$
  Auxiliary data for the zero-knowledge proof.

$\sigma^{\mathtt{ACC}} \in \mathrm{SIGNATURES_{ACC}}$
  Signature from account holder on $(\mathtt{AI}, \pi_{\mathtt{ACC}}, \mathtt{Auxdata})$.

Outputs

  `true` or `false`.

Implementation

1. Check
   (a) that $\mathtt{AI}$ contains a single account credential $\mathtt{AI.AC}$,
   (b) and that the account identifier matches the account credential identifier, i.e., $\mathtt{AI.AC.ID_{AC}} = \mathtt{AI.RegID_{ACC}}$,
   (c) and that credential identifier $\mathtt{AI.AC.ID_{AC}}$ has not been used before on any account credential,
   (d) and that the signature threshold is $\mathtt{AI.threshold_{ACC}} = 1$,
   (e) and that $\mathtt{AI.pk_{ACC}} = \mathtt{AI.RegID_{ACC}}$,
2. Verify that $\sigma^{\mathtt{ACC}}$ is a valid $\mathtt{AI}$ signature on $(\mathtt{AI}, \pi_{\mathtt{ACC}}, \mathtt{Auxdata})$ (cf. Definition 103).
3. Verify the zero-knowledge proof $\pi_{\mathtt{ACC}}$. See below for details on the zero-knowledge proof.

Output `true` if all checks and verifications succeed, `false` otherwise.

❗ Most of the account validity checks, including the well-formedness of the identity disclosure data, is done inside the zero-knowledge proof.

ℹ The $\mathtt{AI.pk_{ACC}} = \mathtt{AI.RegID_{ACC}}$ is implicit if $\mathtt{AI}$ does not contain an explicit $\mathtt{pk_{ACC}}$.

Account Opening Zero-Knowledge Proof

In the account opening transaction $(\mathtt{AI}, \pi_{\mathtt{ACC}}, \mathtt{Auxdata}, \sigma_{\mathtt{AC}})$ the non-interactive zero-knowledge proof $\pi_{\mathtt{ACC}}$ with auxiliary $\mathtt{Auxdata}$ proves the following statement with respect to account credential $\mathtt{AC}$ in $\mathtt{AI}$ (with corresponding $(\mathtt{PAC})$) and identity credential $\mathtt{IDC}$:

1. The account holder knows a valid signature, $\mathtt{IDC}.\sigma_{\mathtt{IDP}}$, under $\mathtt{AC.ID_{IDP}}.\mathtt{vk_{IDP}^{ID}}$ for $\mathtt{IDC}$, i.e., on

$$(\mathtt{IDC.IDcred_{SEC}}, \mathtt{IDC.key_{PRF}}, (\mathtt{AC.ID_{PG_1}}, \ldots, \mathtt{AC.ID_{PG_n}}),$$
$$\mathtt{AC.threshold_{PG}}, \mathtt{IDC.max_{AC}}, \mathtt{IDC.attrList}),$$

2. and $\mathtt{AC.ID_{AC}} = \mathrm{PRF}(\mathtt{IDC.key_{PRF}}, \mathtt{PAC}.x)$ where $\mathtt{PAC}.x \leq \mathtt{IDC.max_{AC}}$,

3. and ciphertext $\mathtt{AC.PGdata_{ACC}}$ is a valid encryption of $\mathtt{IDC.IDcred_{PUB}}$, i.e., $\mathtt{AC.PGdata_{ACC}} = \mathrm{Enc}_{\mathtt{PG}}^{\mathtt{n,d}}(\mathtt{PK_{PG}}, \mathtt{IDC.IDcred_{PUB}})$, where $\mathtt{PK_{PG}}$ is the public-key set of the privacy guardians from $\mathtt{AC}$ and $\mathtt{d} = \mathtt{AC.threshold_{PG}}$,

4. and that the attribute commitments in $\mathtt{AC}$ are well-formed, i.e., $\forall i \ \mathtt{AC}.C_i = \bar{g}_{\mathsf{com}}^{r_i} \bar{h}_{\mathsf{com}}^{a_i}$ where r_i is the i-th entry in $\mathtt{PAC}.(r_0, \ldots, r_\ell)$ and a_i is the i-th entry in $\mathtt{IDC.attrList}$,

5. and the identity credential attributes $\mathtt{IDC.attrList}$ satisfy the public predicate, i.e.,

$$\mathtt{AC.publicAttrPredicate(IDC.attrList)} = \mathtt{true}.$$

> ❗ Note that the signature check ensures that the privacy guardian set of the account credential matches the signed set of the identity credential. This also holds for the identity disclosure threshold $\mathtt{threshold_{PG}}$.

Formal instance and inputs. In the following, we remove the $\mathtt{IDC}$, $\mathtt{AC}$, and $\mathtt{PAC}$ prefixes for the sake of readability (e.g., we write $\mathtt{key_{PRF}}$ instead of $\mathtt{IDC.key_{PRF}}$) whenever possible.

> ℹ️ We assume that the predicate $\mathtt{AC.publicAttrPredicate}$ *only contains* statements of the form $a_i = v$ where a_i is an attribute and v a public value. Such a predicate can be described by tuples of the form $(i, v) \in \mathbb{N} \times \mathbb{F}_{\mathsf{BLS}}$. See Section 15.4 for proving more complex statements about the attributes.

> ⚠️ The information in $\mathtt{AC}$, in particular the signature verification keys $\mathtt{vk_{ACC1}}, \ldots, \mathtt{vk_{ACC}}k$, *must be* part of the proof context, they are an implicit public input. Otherwise, the adversary could replace the signature keys in the opening transaction and essentially steal the identity. The same must hold for a reference to the genesis block. Otherwise, once could use an opening transaction from testnet on mainnet. See Section 9.3 for more details on proof context and how to hash it into the transcript.

The zero-knowledge proof has the following inputs and values.

Implicit public values consist of:

- The group $\mathbb{G}_1^{\mathsf{BLS}}$ with order q and generator g (used for the PRF and ElGamal encryption),
- the commitment keys $\bar{g}_{\mathsf{com}}, \bar{h}_{\mathsf{com}}$ used for Pedersen commitments,
- the account credential $\mathtt{AC}$

Public input consists of:

- The PS verification key $\mathtt{ID_{IDP}.vk_{IDP}^{ID}}$,
- the privacy guardians $(\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}) := (\mathtt{AC.ID_{PG_1}}, \ldots, \mathtt{AC.ID_{PG_n}})$,
- the public keys $\mathtt{PK_{PG}} = \mathtt{pk_{PG_1}}, \ldots, \mathtt{pk_{PG_n}}$ of the PGs in $\mathtt{AC}$,
- the threshold $\mathtt{d} := \mathtt{AC.threshold_{PG}}$,
- the number of shares $\mathtt{n} := |\mathtt{PK_{PG}}|$,
- the points $\alpha_1, \ldots, \alpha_{\mathtt{n}}$ used for sharing,

- the maximal number of accounts $\texttt{max}_{\texttt{AC}} := \texttt{IDC.max}_{\texttt{AC}}$,

- the account identifier $\texttt{ID}_{\texttt{AC}}$,

- the ElGamal ciphertext $\texttt{PGdata}_{\texttt{ACC}} = (c_1, \ldots, c_{\texttt{n}}) := \texttt{AC.PGdata}_{\texttt{ACC}}$ for the PG data,

- the Pedersen commitments $C_0, \ldots, C_\ell$ for attributes,

- the attribute predicate $\texttt{publicAttrPredicate}$ described by tuples (i_k, v_k),

- auxiliary information consisting of

  - PS signature $\widehat{\sigma}$, a randomized version of $\sigma_{\texttt{IDP}}$,

  - A Pedersen commitment C_x to x.

  - A Pedersen commitment $C_{\texttt{max}_{\texttt{AC}}}$ to $\texttt{max}_{\texttt{AC}}$.

  - Implicit default Pedersen commitments $C_{\texttt{ID}_{\texttt{PG}_1}}, \ldots, C_{\texttt{ID}_{\texttt{PG}_{\texttt{n}}}}$ to $\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_{\texttt{n}}}$ (cf. Definition 22).

  - Implicit default Pedersen commitment $C_{\texttt{d}}$ to $\texttt{d}$.

Witness consists of:

- The identity credential $\texttt{IDcred}_{\texttt{SEC}} := \texttt{IDC.IDcred}_{\texttt{SEC}}$,

- the PRF key $\texttt{key}_{\texttt{PRF}} := \texttt{IDC.key}_{\texttt{PRF}}$ and account credential number x,

- the identity attributes $a_0, \ldots, a_\ell$, and their commitment randomness $r_0, \ldots, r_\ell$,

- the blinding (r, r') for $\widehat{\sigma}$ (this allows to compute $\sigma_{\texttt{IDP}}$),

- the randomness $\tilde{r}_x$ for C_x,

- the randomness $\tilde{r}_{\texttt{max}_{\texttt{AC}}}$ for $C_{\texttt{max}_{\texttt{AC}}}$,

- the sharing randomness $\hat{a}_1, \ldots, \hat{a}_{\texttt{d}}$, and the encryption randomness $\hat{r}_1, \ldots, \hat{r}_{\texttt{n}}$.

- Implicit $\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_{\texttt{n}}}$, and $\texttt{d}$ (they are publicly known)

The proven **statement** is

$$
\text{PK}\Big\{ (\texttt{IDcred}_{\texttt{SEC}}, \texttt{key}_{\texttt{PRF}}, x, (a_0, \ldots, a_\ell), (r_0, \ldots, r_\ell), (r, r'), \tilde{r}_x, \tilde{r}_{\texttt{max}_{\texttt{AC}}}, (\hat{a}_1, \ldots, \hat{a}_{\texttt{d}}), (\hat{r}_1, \ldots, \hat{r}_{\texttt{n}})) :
$$

$$
\text{Verify}_{\text{PS}}(\texttt{ID}_{\texttt{IDP}}.\texttt{vk}_{\texttt{IDP}}^{\texttt{ID}}, (\texttt{IDcred}_{\texttt{SEC}}, \texttt{key}_{\texttt{PRF}}, (\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_{\texttt{n}}}), \texttt{d}, \texttt{max}_{\texttt{AC}}, a_0, \ldots, a_\ell), \sigma_{\texttt{IDP}}) = \texttt{true}
$$

$$
\wedge \; \texttt{ID}_{\texttt{AC}} = \text{PRF}(\texttt{IDC.key}_{\texttt{PRF}}, x) \; \wedge \; 0 \le x \le \texttt{max}_{\texttt{AC}}
$$

$$
\wedge \; \forall i \in \{1, \ldots, \texttt{n}\} \; \text{sh}(\texttt{IDcred}_{\texttt{SEC}})_i = (\texttt{IDcred}_{\texttt{SEC}} + \sum_{1 \le w \le \texttt{d}} \hat{a}_w \alpha_i^w)
$$

$$
\wedge \; \forall i \in \{1, \ldots, \texttt{n}\} \; c_i = (g^{\hat{r}_i}, \bar{h}^{\text{sh}(\texttt{IDcred}_{\texttt{SEC}})_i} \texttt{pk}_{\texttt{PG}_i}^{\hat{r}_i})
$$

$$
\wedge \; \forall i \in \{1, \ldots, \ell\} \; C_i = \bar{g}_{\texttt{com}}^{r_i} \bar{h}_{\texttt{com}}^{a_i}
$$

$$
\wedge \; \texttt{publicAttrPredicate}(a_0, \ldots, a_\ell) = \texttt{true} \Big\}
$$

> ⓘ The encryption of $\texttt{IDcred}_{\texttt{PUB}}$ is generated using $\texttt{IDcred}_{\texttt{SEC}}$ (cf. Section 7.1.5). It follows that $\texttt{IDcred}_{\texttt{PUB}}$ does not show up explicitly in the above proof statement.

Proof construction. Proofs for the individual statements described above can be constructed as follows:

- To prove knowledge of a Pointcheval-Sanders signature we can use the Σ-protocol `agg-dlog` (see Section 9.2.2 and Section 5.3.5) which can be instantiated to prove

$$\text{PK}\left\{(m_1, \ldots, m_w, r') : \nu_2 = \nu_3 \cdot \nu_1^{r'} \prod_{i=1}^{w} u_i^{m_i}\right\}$$

  where $\nu_1 = e(\hat{a}, g_{\text{BLS},2})$, $\nu_2 = e(\hat{b}, g_{\text{BLS},2})$, $\nu_3 = e(\hat{a}, \widetilde{X})$, and $\forall i\ u_i = e(\hat{a}, \widetilde{Y}_i)$ are the public input, and $m_1, \ldots, m_w, r'$ are the witness. The public input can be deterministically computed from the randomized signature $\hat{\sigma} = (\hat{a}, \hat{b}) := (a^r, (b \cdot a^{r'})^r)$ (for signature $\sigma_{\text{IDP}} = (a, b)$) and signature verification key $\text{ID}_{\text{IDP}}.\text{vk}_{\text{IDP}}^{\text{ID}} = (\widetilde{X}, \left\{Y_i, \widetilde{Y}_i\right\}_{i=1,\ldots,w})$. All other values form the shared public values spub_1.

  Note that for this proof part $\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}, \text{d}$, and the revealed attributes are technically part of the witness.

  Let $\varsigma_1 = \text{genSigProofInfo}(\text{agg-dlog}, \text{spub}_1, (\nu_1, \nu_2, \nu_3, u_1, \ldots, u_w))$.

- For the proof of $\text{ID}_{\text{AC}} = \text{PRF}(\text{key}_{\text{PRF}}, x) \wedge 0 \leq x \leq \text{max}_{\text{AC}}$, the prover needs to generate auxiliary commitments $C_x = \bar{g}_{\text{com}}^{\tilde{r}_x} \bar{h}_{\text{com}}^{x}$ and $C_{\text{max}_{\text{AC}}} = \bar{g}_{\text{com}}^{\tilde{r}_{\text{max}_{\text{AC}}}} \bar{h}_{\text{com}}^{\text{max}_{\text{AC}}}$. We first note that $\text{ID}_{\text{AC}} = g^{\frac{1}{x+\text{key}_{\text{PRF}}}}$ is equivalent to $\text{ID}_{\text{AC}}^{x+\text{key}_{\text{PRF}}} = g$. The proof overall consists of a Σ-protocol showing the consistency of ID_{AC} and C_x and a bulletproof to bound the value within C_x.

  For the Σ-protocol we start by combining a `dlog` and `com` using `genAndComp` to get Σ-protocol ς_2' for statement

$$\text{PK}\left\{(x_1', x_2', \tilde{r}_x) : g = \text{ID}_{\text{AC}}^{x_1'} \wedge C_x = \bar{g}_{\text{com}}^{\tilde{r}_x} \bar{h}_{\text{com}}^{x_2'}\right\}$$

  Now apply `genLinRelComp` replacing x_1' by $x_1 + x_3'$ to get ς_2'' for statement

$$\text{PK}\left\{(x_1, x_2', x_3', \tilde{r}_x) : g = \text{ID}_{\text{AC}}^{x_1+x_3'} \wedge C_x = \bar{g}_{\text{com}}^{\tilde{r}_x} \bar{h}_{\text{com}}^{x_2'}\right\}$$

  Finally, apply `genEqComp` replacing x_2', x_3' by x to get ς_2 for statement

$$\text{PK}\left\{(x_1, x, \tilde{r}_x) : g = \text{ID}_{\text{AC}}^{x_1+x} \wedge C_x = \bar{g}_{\text{com}}^{\tilde{r}_x} \bar{h}_{\text{com}}^{x}\right\}$$

  We can then use a bulletproof to prove the statement:

$$\text{PK}\left\{(x, \tilde{r}_x, \text{max}_{\text{AC}}, \tilde{r}_{\text{max}_{\text{AC}}}) : C_x = \bar{g}_{\text{com}}^{\tilde{r}_x} \bar{h}_{\text{com}}^{x} \wedge C_{\text{max}_{\text{AC}}} = \bar{g}_{\text{com}}^{\tilde{r}_{\text{max}_{\text{AC}}}} \bar{h}_{\text{com}}^{\text{max}_{\text{AC}}} \wedge x \in [0, \text{max}_{\text{AC}}]\right\}.$$

  Denote by `BPInfo` the resulting bulletproof information.

  > ❗ This uses the Bulletproof for an arbitrary range (cf. Section 9.5.3). Technically, we show that $\text{max}_{\text{AC}} - x \in [0, 2^s]$ and $\text{max}_{\text{AC}} \in [0, 2^s]$ where the commitments can store values of size at least 2^{s+1} (cf. Section 15.4).

- The statement $\text{PGdata}_{\text{ACC}i} = \text{Enc}_{\text{EG}}(\text{PK}_{\text{PG}i}, g^{\text{sh}(\text{IDcred}_{\text{SEC}})_i})$ (cf. Section 7.1.5) shows that $\text{IDcred}_{\text{PUB}}$ was encrypted correctly under PK_{PG}.

⚠️ It is important that the verifier computes $\texttt{PK}_{\texttt{PG}}$ from $\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_n}$.

We start by combining dlog and $\mathsf{agg\text{-}dlog}$ using $\mathsf{genAndComp}$ to get Σ-protocol $\hat{\varsigma}_i$ for the statement

$$\texttt{PK}\left\{(\hat{r}_i, t_i) : c_i = (g^{\hat{r}_i}, \bar{h}^{t_i}\texttt{pk}_{\texttt{PG}_i}^{\hat{r}_i})\right\}$$

for $i \in \{1, \ldots, n\}$.

Now apply $\mathsf{genLinRelComp}$ replacing t_i by $x_{i,2} + \sum_{1 \leq j \leq d} \hat{a}_{i,j}\alpha_i^j$ resulting in Σ-protocol $\hat{\varsigma}_i'$ for the statement

$$\texttt{PK}\left\{(x_{i,2}, \hat{a}_{i,1}, \ldots, \hat{a}_{i,d}, \hat{r}_i) : c_i = (g^{\hat{r}_i}, \bar{h}^{x_{i,2}+\sum_{1 \leq j \leq d} \hat{a}_{i,j}\alpha_i^j}\texttt{pk}_{\texttt{PG}_i}^{\hat{r}_i})\right\}.$$

Next, apply $\mathsf{genAndComp}$ to get $\varsigma_3' = \mathsf{genAndComp}(\hat{\varsigma}_1', \ldots, \hat{\varsigma}_2')$ for the statement

$$\texttt{PK}\left\{(\{x_{i,2}, \hat{a}_{i,1}, \ldots, \hat{a}_{i,d}, \hat{r}_i\}_{1 \leq i \leq n}) : \forall i\ c_i = (g^{\hat{r}_i}, \bar{h}^{x_{i,2}+\sum_{1 \leq j \leq d} \hat{a}_{i,j}\alpha_i^j}\texttt{pk}_{\texttt{PG}_i}^{\hat{r}_i})\right\}.$$

Finally, apply $\mathsf{genEqComp}$ to replace $\hat{a}_{i,j}$ by $\hat{a}_j$ and $x_{i,2}$ by x_2 for all i resulting in ς_3 for statement

$$\texttt{PK}\left\{(\{x_2, \hat{a}_1, \ldots, \hat{a}_d, \hat{r}_i\}_{1 \leq i \leq n}) : \forall i\ c_i = (g^{\hat{r}_i}, \bar{h}^{x_2+\sum_{1 \leq j \leq d} \hat{a}_j\alpha_i^j}\texttt{pk}_{\texttt{PG}_i}^{\hat{r}_i})\right\}.$$

- The statement $\forall i \in \{0, \ldots, \ell\}\ C_i = \bar{g}_{\texttt{com}}^{r_i}\bar{h}_{\texttt{com}}^{a_i}$ proves that the attribute commitments are well-formed.

  We get Σ-protocol σ_4 for statement

  $$\texttt{PK}\left\{(a_0, \ldots, a_\ell, r_0, \ldots, r_\ell) : \ \forall i \in \{0, \ldots, \ell\}\ C_i = \bar{g}_{\texttt{com}}^{r_i}\bar{h}_{\texttt{com}}^{a_i}\right\}$$

  by repeatedly combining com using $\mathsf{genAndComp}$.

- The well-formedness of the default commitments to $\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_n}$, and d.

  We get Σ-protocol σ_5 for statement

  $$\texttt{PK}\left\{(c_1, \ldots, c_\ell, d) : \ \forall i \in \{0, \ldots, n\}\ C_{\texttt{ID}_{\texttt{PG}_i}} = \bar{h}_{\texttt{com}}^{c_i} \ \wedge \ C_d = \bar{h}_{\texttt{com}}^{threshold}\right\}$$

  by repeatedly combining dlog using $\mathsf{genAndComp}$.

- Finally, we consider $\texttt{publicAttrPredicate}(a_0, \ldots, a_\ell) = \texttt{true}$ for the special case where the predicate only reveals some attributes fully.

  We get Σ-protocol σ_6 for statement

  $$\texttt{PK}\left\{(\{b_{i_k}\}) : \forall i_k\ b_{i_k} = v_k\right\}$$

  by repeatedly combining the Σ-protocol in Section 9.2.4 using $\mathsf{genAndComp}$.

We combine the parts as follows

1. Combining $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6)$ using genAndComp results in σ' with secret input

$$(m_1, \ldots, m_w, r', x_1, x, \tilde{r}_x, \tilde{r}_{\mathtt{max_{AC}}}, x_2, \hat{a}_1, \ldots, \hat{a}_{\mathtt{d}}, \{\hat{r}_i\}_{1 \leq i \leq \mathtt{n}},$$
$$a_1, \ldots, a_\ell, r_1, \ldots, r_\ell, c_1, \ldots, c_\ell, \mathtt{d}, b_{i_k})$$

2. Next, we apply the genEqComp operator several times setting

   - $b_{i_k} := a_{i_k}$

   - $m_1 = x_2$ and renaming it $\mathtt{IDcred_{SEC}}$

   - $m_2 = x_1$ and renaming it $\mathtt{key_{PRF}}$

   - $m_{3+i} = c_i$ and renaming it to $\mathtt{ID_{PG_i}}$

   - $m_{3+\mathtt{n}+1} := \mathtt{d}$

   - $m_{3+\mathtt{n}+2} := \mathtt{max_{AC}}$

   - $m_{3+\mathtt{n}+2+i} := a_i$ (note we have $m_1, \ldots, m_w$, and $a_0, \ldots, a_\ell$)

   This results in σ with witness

$$(\mathtt{IDcred_{SEC}}, \mathtt{key_{PRF}}, r', x, \tilde{r}_x, \tilde{r}_{\mathtt{max_{AC}}}, \mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}, \mathtt{d},$$
$$\hat{a}_1, \ldots, \hat{a}_{\mathtt{d}}, \{\hat{r}_i\}_{1 \leq i \leq \mathtt{n}}, a_1, \ldots, a_\ell, r_1, \ldots, r_\ell)$$

3. The ZK proof is thus described by ς and BPInfo. To generate the non-interactive proof the account holder uses genSigmaANDBulletproof. As context string, we use

$$\mathtt{ctx} := \text{``openAccount''} \parallel \mathsf{H}(\text{genesis block}).$$

   To verify the proof the identity provider uses verifySigmaANDBulletproof.

> ⚠ The above proof only shows knowledge of signed $\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}, \mathtt{d}$. To connect them to the public $\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}, \mathtt{d}$ in AC the verifier **must** compute the default commitments $\left\{ C_{\mathtt{ID_{PG_i}}} \right\}$ and $C_{\mathtt{d}}$ used in the verification from the AC data. In addition, the verifier must compute $\mathtt{PK_{PG}}$ from the public values as well and check that the parameters used in the shared encryption $\mathtt{AC.PGdata_{ACC}}$ match $\mathtt{d}$ and $\mathtt{n} = |\{\mathtt{ID_{PG_i}}\}|$.

> </> In the implementation the context string contains the hash of the cryptographic parameters and the genesis string. It does not contain the hash of the genesis block.

> </> In the implementation, the attribute predicate is a set of revealed values. To prove the correctness of the predicate, the implementation does not use the proposed proof. Instead, default commitments are used. The verifier checks that the revealed values match the default commitments as part of verification, i.e., they compute the default commitments from the revealed values.

> ❗ More complex attributes predicates could be implemented. However, for privacy reasons it should be considered to completely remove attributes predicates from accounts. It is better to make attributes proofs off-chain.

Verification of Legacy Initial Accounts

In previous version of this protocol, identity providers created so-called *initial account*s when issuing identity credentials. This feature has been removed from the current protocol. However, in order to catch-up from the genesis block validators still need to verify the opening of such accounts.

Initial accounts were opened on chain by publishing a transaction with $\mathtt{AI}$ and a signature $\sigma_{\mathtt{IDP}}$. The validity of this transaction can be checked by verifying the signature $\sigma_{\mathtt{IDP}}$ with the verification key $\mathtt{AI.AC.ID_{IDP}.vk_{IDP}^{ID}}$ and checking that $\mathtt{AI.AC.ID_{IDP}}$ is a trusted identity provider. Additionally, $\mathtt{AI.RegID_{ACC}}$ should not have been used before on any account.

> ❗ Zero knowledge proofs are not required for initial accounts since the identity provider already checked everything during the issuance of the identity credential to the account holder. We assume identity providers never sign wrong data, i.e., they are trusted.

15.3.4 Credential Management

Adding Credentials

The following (local) protocol can be used to create the information required to add a new credential to an existing account. This new credential can be added to the account using the transaction described below.

> ℹ️ The account holder in the following protocol is (usually) not the initial creator of the involved account.

Protocol createCredAddInfo

Inputs

$\mathtt{RegID_{ACC}} \in \mathrm{REGACCIDS}$
    Account registration ID of the account where the credential should be deployed.

$\mathtt{IDC} \in \mathrm{IDCs}$
    The identity credential of the to-be-added account holder.

$\mathtt{AC} \in \mathrm{ACs}$
    The account credential of the to-be-added account holder.

$\mathtt{PAC} \in \mathrm{PACs}$
    The private account credential of the to-be-added account holder.

Outputs

$\mathtt{AC} \in \mathrm{ACs}$
    The account credential from the input.

$\pi_{\mathtt{ACC}} \in \Sigma\text{-}\mathrm{BULLETPROOFS}$
    Non-interactive ZK proof of credential validity.

$\mathtt{Auxdata} \in \{0,1\}^*$
    Auxiliary data for the ZK proof.

$\sigma_{\mathtt{AC}} \in \mathrm{SIGNATURES_{ACC}}^*$
    Signature from account credential on $\mathtt{RegID_{ACC}}$, $\mathtt{AC}$, $\pi_{\mathtt{ACC}}$, and $\mathtt{Auxdata}$.

Implementation

1. **Proving Credential Validity:** The to-be-added account holder produces a non-interactive ZK proof $\pi_{\texttt{ACC}}$ with auxiliary data $\texttt{Auxdata}$ for the statement:

   (a) The to-be-added account holder knows a valid signature, $\texttt{IDC}.\sigma_{\texttt{IDP}}$, under $\texttt{AC.ID}_{\texttt{IDP}}.\texttt{vk}_{\texttt{IDP}}^{\texttt{ID}}$ for $\texttt{IDC}$, i.e., on

$$(\texttt{IDC.IDcred}_{\texttt{SEC}}, \texttt{IDC.key}_{\texttt{PRF}}, (\texttt{IDC.ID}_{\texttt{PG}_1}, \ldots, \texttt{IDC.ID}_{\texttt{PG}_n}),$$
$$\texttt{IDC.threshold}_{\texttt{PG}}, \texttt{IDC.max}_{\texttt{AC}}, \texttt{IDC.attrList}),$$

   (b) and $\texttt{AC.ID}_{\texttt{AC}} = \texttt{PRF}(\texttt{IDC.key}_{\texttt{PRF}}, \texttt{PAC}.x)$ where $\texttt{PAC}.x \leq \texttt{IDC.max}_{\texttt{AC}}$,

   (c) and $(\texttt{AC.ID}_{\texttt{PG}_1}, \ldots, \texttt{AC.ID}_{\texttt{PG}_n}) = (\texttt{IDC.ID}_{\texttt{PG}_1}, \ldots, \texttt{IDC.ID}_{\texttt{PG}_n})$,

   (d) and $\texttt{AC.threshold}_{\texttt{PG}} = \texttt{IDC.threshold}_{\texttt{PG}}$,

   (e) and ciphertext $\texttt{AC.PGdata}_{\texttt{ACC}}$ is a valid encryption of $\texttt{IDC.IDcred}_{\texttt{PUB}}$, i.e., $\texttt{AC.PGdata}_{\texttt{ACC}} = \texttt{Enc}_{\texttt{PG}}^{\texttt{n,d}}(\texttt{PK}_{\texttt{PG}}, \texttt{IDC.IDcred}_{\texttt{PUB}})$, where $\texttt{PK}_{\texttt{PG}}$ is the public-key set of the privacy guardians from $\texttt{AC}$ and $\texttt{d} = \texttt{AC.threshold}_{\texttt{PG}}$,

   (f) and that the attribute commitments in $\texttt{AC}$ are well-formed, i.e. $\forall i \; \texttt{AC}.C_i = \bar{g}_{\texttt{com}}^{r_i} \bar{h}_{\texttt{com}}^{a_i}$ where r_i is the i-th entry in $\texttt{PAC}.(r_0, \ldots, r_\ell)$ and a_i is the i-th entry in $\texttt{IDC.attrList}$,

   (g) and the credential attributes $\texttt{IDC.attrList}$ satisfy the public predicate, i.e. $\texttt{AC.publicAttrPredicate}(\texttt{IDC.attrList}) = \texttt{true}$.

   The zero-knowledge proof is essentially proving the same statements as used during account creation. The zero-knowledge proof *must* use $\texttt{RegID}_{\texttt{ACC}}$ and $\texttt{AC}$ as part of its context.

2. **Signing:** The to-be-added account holder generates signature $\sigma_{\texttt{AC}}$ by signing $(\texttt{RegID}_{\texttt{ACC}}, \texttt{AC}, \pi_{\texttt{ACC}}, \texttt{Auxdata})$ using singing key from the private account credential $\texttt{PAI.PAC}$.

3. **Output:** To add the new credential, $(\texttt{RegID}_{\texttt{ACC}}, \texttt{AC}, \pi_{\texttt{ACC}}, \texttt{Auxdata}, \sigma_{\texttt{AC}})$ is submitted in a special account-opening transaction by the existing account holders, see below.

The zero-knowledge proof can be constructed using the same components as in the one for an account opening (see Section 15.3.3).

> ⚠ To prevent others from adding the credential to a different account or adding the credential with a different signing key the ZK proof *must* use both $\texttt{RegID}_{\texttt{ACC}}$ and $\texttt{AC}$ as part of its context string, cf. Section 9.3.

Credential Management Transactions

The current holders of an account can update the credential structure of the account by publishing a *credential update* transaction on chain.

> ⚠ Since $\texttt{RegID}_{\texttt{ACC}}$ and the account encryption key $\texttt{pk}_{\texttt{ACC}}$ depend on (the initial) $\texttt{AC}_1$, this credential cannot be removed.

> </> In the current implementation, one cannot add new credential to accounts which used the shielded transaction feature (cf. Section 16.3).

Definition 104. A *credential update transaction* consists of the following:

- $\mathtt{RegID_{ACC}}$, the *account registration ID* of the account whose credentials are to be updated,

- $\mathtt{AC}_1, \ldots, \mathtt{AC}_k$, list of account credentials to be associated with the account; Any account credential that is currently associated with the account and is not part of the list will be removed,

- $\pi_{\mathtt{ACC}1}, \mathtt{Auxdata}_1, \sigma_{\mathtt{AC}1}, \ldots, \pi_{\mathtt{ACC}m}, \mathtt{Auxdata}_m, \sigma_{\mathtt{AC}m}$, list of zero-knowledge proofs, auxiliary data, and signatures created by $\mathsf{createCredAddInfo}$ for all $\mathtt{AC}_i$ that are not already associated with the account,

- $\mathtt{threshold_{ACC}}$, the new account signature threshold, determining how many credentials need to sign subsequent transactions from that account,

- $\sigma^{\mathtt{ACC}}$ signatures from the current account holders on the rest of the transaction.

We next define validity of a credential update transaction.

Definition 105. Let $\mathtt{tx}$ be a credential update transaction, i.e,

$$\mathtt{tx} = \big(\mathtt{RegID_{ACC}}, (\mathtt{AC}_1, \ldots, \mathtt{AC}_n), (\pi_{\mathtt{ACC}1}, \mathtt{Auxdata}_1, \sigma_{\mathtt{AC}1},$$
$$\ldots, \pi_{\mathtt{ACC}m}, \mathtt{Auxdata}_m, \sigma_{\mathtt{AC}m}), \mathtt{threshold_{ACC}}, \sigma^{\mathtt{ACC}}\big).$$

Let $\mathtt{threshold'_{ACC}}$ be the threshold that currently applies for the account $\mathtt{RegID_{ACC}}$, and let $\mathtt{AC}'_1, \ldots, \mathtt{AC}'_{n'}$ be the credentials currently associated with that account. This transaction is *valid* if all the following are satisfied:

- $\mathtt{AC}_1 = \mathtt{AC}'_1$, i.e., the initial credential is not changed,

- for all $\mathtt{AC}_i$ that are not already in $\{\mathtt{AC}'_1, \ldots, \mathtt{AC}'_{n'}\}$, $\mathtt{AC}_i.\mathtt{ID_{AC}}$ has never been used before,

- for all $\mathtt{AC}_i$ that are not already in $\{\mathtt{AC}'_1, \ldots, \mathtt{AC}'_{n'}\}$, there exists $j \in \{1, \ldots, m\}$ such that

  - $\sigma_{\mathtt{AC}j}$ is a valid signature of $(\mathtt{RegID_{ACC}}, \mathtt{AC}_i, \pi_{\mathtt{ACC}i}, \mathtt{Auxdata}_i)$ with respect to the signature verification keys $\mathtt{AC}_i.(\mathtt{vk_{ACC1}}, \ldots, \mathtt{vk_{ACC}k})$,

  - and $\pi_{\mathtt{ACC}j}$ constitutes a valid ZK proof with respect to the auxiliary data $\mathtt{Auxdata}_j$,

- $\sigma^{\mathtt{ACC}}$ contains valid signatures on the rest of the transaction from at least $\mathtt{threshold'_{ACC}}$ distinct credentials from $\mathtt{AC}'_1, \ldots, \mathtt{AC}'_{n'}$,

- and $\mathtt{threshold_{ACC}} \leq n$.

Effect of Credential Update Transaction. After a valid credential update transaction $\big(\mathtt{RegID_{ACC}}, (\mathtt{AC}_1, \ldots, \mathtt{AC}_k), \cdot, \mathtt{threshold_{ACC}}, \cdot\big)$ appears in a block, the account $\mathtt{RegID_{ACC}}$ from that point on is associated with the credentials $\mathtt{AC}_1, \ldots, \mathtt{AC}_k$ and the threshold $\mathtt{threshold_{ACC}}$. That means that all transactions from that account need to be signed by at least $\mathtt{threshold_{ACC}}$ credentials from $\mathtt{AC}_1, \ldots, \mathtt{AC}_k$.

Update Account Credential Keys

An account holder with account credential $\mathtt{AC}$ for some account $\mathtt{RegID_{ACC}}$ can update the credential keys using a credential key update transaction. This requires a signature from the current keys associated with $\mathtt{AC}$, to make sure that only the holder of that specific credential can change the keys, as well as an account signature, to be able to charge the account for the transaction fee.

Definition 106. A *credential key update transaction* consists of the following:

- $\mathtt{RegID_{ACC}}$, the account registration identifier of the account whose credentials are to be updated,

- $\mathtt{ID_{AC}}$ the identifier of the to-be-updated account credential $\mathtt{AC}$,

- $\mathtt{vk'_{ACC1}}, \ldots, \mathtt{vk'_{ACC}}_k$ the new account credential keys,

- $\mathtt{threshold'_{Cred}}$ the new account credential signature threshold,

- $\sigma_{\mathtt{AC}}$ an account credential signature of $(\mathtt{RegID_{ACC}}, \mathtt{ID_{AC}}, \mathtt{vk'_{ACC1}}, \ldots, \mathtt{vk'_{ACC}}_k, \mathtt{threshold'_{Cred}})$ under $\mathtt{AC}$,

- $\sigma^{\mathtt{ACC}}$ account signature (possibly including $\sigma_{\mathtt{AC}}$) from the current account holders on all the above values.

The validity of a credential key update transaction is defined as follows.

Definition 107. A credential key update transaction $(\mathtt{RegID_{ACC}}, \mathtt{ID_{AC}}, \mathtt{vk'_{ACC1}}, \ldots, \mathtt{vk'_{ACC}}_k, \mathtt{threshold'_{Cred}},$ $\sigma_{\mathtt{AC}}, \sigma^{\mathtt{ACC}})$ is *valid*

- If $\mathtt{ID_{AC}}$ denotes a current account credential $\mathtt{AC}$ of the account $\mathtt{RegID_{ACC}}$,

- $\mathtt{vk'_{ACC1}}, \ldots, \mathtt{vk'_{ACC}}_k$ are valid signature keys and the threshold $\mathtt{threshold'_{Cred}}$ is small enough,

- $\sigma_{\mathtt{AC}}$ is a valid credential signature of $(\mathtt{RegID_{ACC}}, \mathtt{ID_{AC}}, \mathtt{vk'_{ACC1}}, \ldots, \mathtt{vk'_{ACC}}_k, \mathtt{threshold'_{Cred}})$ under $\mathtt{AC}$ (with the current keys in $\mathtt{AC}.(\mathtt{vk_{ACC1}}, \ldots, \mathtt{vk_{ACC}}_k)$).

- $\sigma^{\mathtt{ACC}}$ is a valid account signature from the current account holders on all the above values.

> ♡ The implementation could allow that an arbitrary account could sign (and pay for) this type of transaction. This does not violate security as the account credential signature is still part of the transaction.

Effect. The effect of a valid credential key update transaction

$$(\mathtt{RegID_{ACC}}, \mathtt{ID_{AC}}, \mathtt{vk'_{ACC1}}, \ldots, \mathtt{vk'_{ACC}}_k, \mathtt{threshold'_{Cred}}, \sigma_{\mathtt{AC}}, \sigma^{\mathtt{ACC}})$$

is to replace in account credential $\mathtt{AC}$ with identifier $\mathtt{ID_{AC}}$

$$(\mathtt{vk_{ACC1}}, \ldots, \mathtt{vk_{ACC}}_k), \mathtt{threshold_{Cred}}$$

by

$$(\mathtt{vk'_{ACC1}}, \ldots, \mathtt{vk'_{ACC}}_k), \mathtt{threshold'_{Cred}}.$$

15.4 Identity Presentations using Zero-Knowledge Proofs

An identity presentation allows proving statements derived from the identity attributes. In this section, we provide two options for creating identity presentations, namely, from the commitments in an account credential or directly from an identity credential.

15.4.1 Account Based Presentations

Each account credential contains a list of commitments to the identity attributes of the corresponding account holder. This allows the account holder to prove statements about their attributes. For example, the proofs could show that the account holder is older than 18 and lives in an EU country. We call this object of statement and proof a *presentation*.

In general, the statement of a presentation can be described as a predicate that the attributes $\mathtt{IDC.attrList}$ must satisfy. This predicate can for example be represented as an arithmetic circuit.

Proving the satisfiability of such a general predicate would require the use of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs).

We therefore only consider the restricted case where the presented predicate can be described by statements of the form $a = v$, $a < v$, $a \in S$, or $a \notin S$ where $a \in \mathtt{IDC.attrList}$ is an attribute, v is a public value, S is a public set. Note that $a \in [v_1, v_2)$ can be reduced to $v_1 - 1 < a$ and $a < v$. An example of such a predicate is "The account holder is older than 18 and lives in Denmark."

Zero-knowledge protocols for simple predicates. The following simple building blocks can be used to create a predicate zero-knowledge proof from a given on-chain account.

> **ⓘ** These building blocks could also be used to augment the account opening proof. However, this would make the proven statement public forever as it cannot be deleted from the blockchain. See Section 15.3.3 for more details on the overall opening proof.

We assume that there exist valid Pedersen commitments to all involved attributes, e.g., as part of an account credential. The validity of these commitments must be checked by the application. In most cases the validity will be ensured by the fact that the commitments are part of an on-chain account; they have been verified as part of the validity check of the opening transaction. Furthermore, default commitments to public values (i.e., commitments with fixed randomness 0, cf. Definition 22) can be computed on the fly. The homomorphic property of Pedersen commitments allows anyone to compute commitments to any linear combination of attributes and public values.

We can use these commitments as public input to our attribute predicate ZK proof. The witness to the zero-knowledge proof are the involved attributes. We have the following basic templates.

Equality with Public Value: To show a statement of the form $a = v$ where a is a secret value and v a public value, one use the com-pub Σ-protocol. If the commitment randomness is not confidential, e.g. as it is a temporary commitment, one can simply open the commitment to v, i.e., publish the commitment randomness.

Equality with Secret Value: To show a statement of the form $a_1 = a_2$, we can use a Σ-protocol to show that the commitments to a_1 and a_2 contain the same value. Public inputs are commitments C_1 to a_1 and C_2 to a_2. Secret values are a_1, a_2 and the commitment randomness r_1, r_2. We start with the Σ-protocol agg-dlog showing

$$PK(a_i, r_i) : C_i = \bar{g}_{\mathtt{com}}^{r_i} \bar{h}_{\mathtt{com}}^{a_i}.$$

for $i = 1, 2$. We then combine the two copies of the proof using genAndComp and set $a := a_1 = a_2$ using genEqComp to obtain a Σ-protocol proof ς for the statement

$$PK(a, r_1, r_2) : C_1 = \bar{g}_{\mathtt{com}}^{r_1} \bar{h}_{\mathtt{com}}^{a} \wedge C_2 = \bar{g}_{\mathtt{com}}^{r_2} \bar{h}_{\mathtt{com}}^{a}.$$

Alternatively, one can use the homomorphic properties of the commitments and show that the commitment $C = C_1 C_2^{-1}$ is a commitment to zero using Σ-protocol com-pub.

Inequality of Values: To show a statement of the form $a \leq v$, Bulletproofs can be used. It does not matter whether a or v are public or private. We assume that $a, v \in [0, 2^k)$ where the commitments allow committing to values of size at least 2^{k+1}. To prove $a \leq v$, we use a commitment C to $v - a$ and show using a Bulletproof that $v - a \in [0, 2^k)$, i.e., $\mathtt{BPInfo} = \mathtt{genBPInfo}(k, (\bar{g}_{\mathtt{com}}, \bar{h}_{\mathtt{com}}), C)$. The commitment C can be obtained by a homomorphic operation on commitments to v and $-a$. Since commitments work for values

at least 2^{k+1}, we can conclude from $v - a \in [0, 2^k)$ that $v - a \geq 0$ despite possible under flows.

> ℹ️ If it is unclear that the (committed) values a, v are in the range $[0, 2^k)$, one can prove it with an additional Bulletproof. The proof can be aggregated with BPInfo using aggregateBPInfo.

Set-(Non)-Membership: To show *set-membership* or *set-non-membership*, i.e., $a \in S$ or $a \notin S$ for a public set S, a variant of Bulletproofs can be used. See Sections 9.5.4 and 9.5.5 for more details.

The above templates can be invoked with any linear combination of attributes or public values.

The Σ-protocol parts of the used templates can be combined using genAndComp and genEqComp. The Bulletproofs can be combined using aggregateBPInfo.

15.4.2 Identity Based Credentials

As for account credential based presentations, we assume that the statements are of the following form. For attribute $a \in$ IDC.attrList and public value v and public set S, the allowed predicates are $a = v$, $a < v$, $a \in S$, or $a \notin S$.

In addition, the identity based credentials contain, similar to account credentials, identity disclosure information.

Zero-knowledge protocol for simple predicates. The zero-knowledge proof construction is a streamlined version of the account opening proof (cf. Section 15.3.3). For simplicity, we assume that revealed attributes, with indices in J_{rev}, are not used in any other type of statement.

Public input consists of:

- The pairing e on $(\mathbb{G}_1^{\text{BLS}}, \mathbb{G}_2^{\text{BLS}}, \mathbb{G}_T^{\text{BLS}})$ and generators $g_{\text{BLS},1}, g_{\text{BLS},2}$,

- the commitment key $\text{ck} = (\bar{g}_{\text{com}}, \bar{h}_{\text{com}})$ used for Pedersen commitments,

- the PS verification key $\text{ID}_{\text{IDP}}.\text{vk}_{\text{IDP}}^{\text{ID}}$ of the identity provider,

- the statement information

  - revealed values: $(v_j)_{j \in J_{rev}}$ and index set J_{rev},

  - inequalities: $(v_j)_{j \in J_{ge}}$ and index set J_{ge},

  - set-membership: $(S_j)_{j \in J_{in}}$ and index set J_{in},

  - set-non-membership: $(S_j)_{j \in J_{out}}$ and index set J_{out},

- the identity disclosure information:

  - the ElGamal ciphertext $\text{PGdata}_{\text{ACC}} = (c_1, \ldots, c_n)$

  - the privacy guardians $(\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}) := (\text{AC.ID}_{\text{PG}_1}, \ldots, \text{AC.ID}_{\text{PG}_n})$,

  - the public keys $\text{PK}_{\text{PG}} = \text{pk}_{\text{PG}_1}, \ldots, \text{pk}_{\text{PG}_n}$ of the PGs in AC,

  - the threshold $\text{d} := \text{AC.threshold}_{\text{PG}}$,

  - the number of shares $\text{n} := |\text{PK}_{\text{PG}}|$,

  - the points $\alpha_1, \ldots, \alpha_n$ used for sharing,

- auxiliary information consisting of

  - PS signature $\widehat{\sigma}$, a randomized version of $\sigma_{\texttt{IDP}}$,

  - Implicit default Pedersen commitments $C_{\texttt{ID}_{\texttt{PG}_1}}, \ldots, C_{\texttt{ID}_{\texttt{PG}_n}}$ to $\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_n}$ (cf. Definition 22),

  - Implicit default Pedersen commitment $C_{\texttt{d}}$ to $\texttt{d}$,

  - Pedersen commitments C_j to attribute a_j for all $j \in J_{ge} \cup J_{in} \cup J_{out}$.

The Witness consists of:

- The signature $\sigma_{\texttt{IDP}}$ (implicitly), blinding (r, r') for $\widehat{\sigma}$,

- the signed values, i.e., $(\texttt{IDcred}_{\texttt{SEC}}, \texttt{key}_{\texttt{PRF}}, (\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_n}), \texttt{d}, \texttt{max}_{\texttt{AC}}, a_0, \ldots, a_\ell)$,

- the secret sharing randomness $\hat{a}_1, \ldots, \hat{a}_{\texttt{d}}$, and the encryption randomness $\hat{r}_1, \ldots, \hat{r}_{\texttt{n}}$,

- the commitment opening information r_j of C_j for all $j \in J_{ge} \cup J_{in} \cup J_{out}$,

- implicit $\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_n}$, and $\texttt{d}$ (they are publicly known)

The proven **statement** is

$$
\begin{aligned}
\mathsf{PK}\Big\{ & (\texttt{IDcred}_{\texttt{SEC}}, \texttt{key}_{\texttt{PRF}}, (a_0, \ldots, a_\ell), (r_0, \ldots, r_\ell), (r, r'), (\hat{a}_1, \ldots, \hat{a}_{\texttt{d}}), (\hat{r}_1, \ldots, \hat{r}_{\texttt{n}}), (r_j)) : \\
& \mathsf{Verify}_{\mathsf{PS}}(\texttt{ID}_{\texttt{IDP}}.\texttt{vk}_{\texttt{IDP}}^{\texttt{ID}}, (\texttt{IDcred}_{\texttt{SEC}}, \texttt{key}_{\texttt{PRF}}, (\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_n}), \texttt{d}, \texttt{max}_{\texttt{AC}}, a_0, \ldots, a_\ell), \sigma_{\texttt{IDP}}) = \texttt{true} \\
& \wedge\ \forall i \in \{1, \ldots, \texttt{n}\}\ \ \mathsf{sh}(\texttt{IDcred}_{\texttt{SEC}})_i = (\texttt{IDcred}_{\texttt{SEC}} + \sum_{1 \le w \le \texttt{d}} \hat{a}_w \alpha_i^w) \\
& \wedge\ \forall i \in \{1, \ldots, \texttt{n}\}\ \ c_i = (g^{\hat{r}_i}, \bar{h}^{\mathsf{sh}(\texttt{IDcred}_{\texttt{SEC}})_i} \texttt{pk}_{\texttt{PG}_i}^{\hat{r}_i}) \\
& \wedge\ \forall j \in J_{ge} \cup J_{in} \cup J_{out}\ \ C_j = \bar{g}_{\texttt{com}}^{r_j} \bar{h}_{\texttt{com}}^{a_j} \\
& \wedge\ \forall j \in J_{rev}\ a_j = v_j \wedge\ \forall j \in J_{ge}\ \text{value in } C_j \text{ is } < v_j \\
& \wedge\ \forall j \in J_{in}\ \text{value in } C_j \text{ is in } S_j \wedge\ \forall j \in J_{out}\ \text{value in } C_j \text{ is not in } S_j \Big\}
\end{aligned}
$$

The **proof** for the above statement can be constructed as a combination of a Sigma protocol and Bulletproofs. The details are as follows:

- To prove knowledge of a Pointcheval-Sanders signature on

$$
\vec{m} = (\texttt{IDcred}_{\texttt{SEC}}, \texttt{key}_{\texttt{PRF}}, (\texttt{ID}_{\texttt{PG}_1}, \ldots, \texttt{ID}_{\texttt{PG}_n}), \texttt{d}, \texttt{max}_{\texttt{AC}}, a_0, \ldots, a_\ell)
$$

  and reveal the values, i.e., show that $\forall j \in J_{rev}\ a_j = v_j$, we can use the Σ-protocol described in Section 5.3.5.

  Let J be the indices of revealed attributes in $\vec{m}$. Then the protocol is essentially an instance of the agg-dlog protocol from Section 9.2.2 to prove

$$
\mathsf{PK}\left\{ ((m_i)_{i \notin J}, r') : \nu_2 = \nu_3 \cdot \nu_1^{r'} \prod_{i \notin J} u_i^{m_i} \right\}
$$

  where $\nu_1 = e(\hat{a}, g_{\texttt{BLS},2})$, $\nu_2 = e(\hat{b}, g_{\texttt{BLS},2})$, $\nu_3 = e(\hat{a}, \widetilde{X} \cdot \prod_{j \in J} \widetilde{Y}_j^{m_j})$, and $\forall i\ u_i = e(\hat{a}, \widetilde{Y}_i)$ are the public input, and $(m_i)_{i \notin J}, r'$ are the witness.

  The public input can be deterministically computed from the randomized signature $\widehat{\sigma} = (\hat{a}, \hat{b}) \coloneqq (a^r, (b \cdot a^{r'})^r)$ (for signature $\sigma_{\texttt{IDP}} = (a, b)$), signature verification key $\texttt{ID}_{\texttt{IDP}}.\texttt{vk}_{\texttt{IDP}}^{\texttt{ID}} =$

$(\widetilde{X}, \left\{ Y_i, \widetilde{Y}_i \right\}_{i=1,\ldots,w})$, and the revealed attributes a_j $j \in J_{rev}$. All other values form the shared public values $\mathtt{spub}_1$.

Note that for this proof part $\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}, \mathtt{d}$, are technically part of the witness.

- To prove knowledge of opening commitments C_j for $j \in J_{ge} \cup J_{in} \cup J_{out}$ we can repeatedly combine com from Section 9.2.3 using genAndComp. This is similar to the account opening proof, except that here we have only commitments to a subset of the signed values.

- To prove the statement about the correct encryption of $\mathtt{IDcred_{PUB}}$, i.e.,

$$\forall i \in \{1, \ldots, \mathtt{n}\} \ \ \mathtt{sh}(\mathtt{IDcred_{SEC}})_i = \left(\mathtt{IDcred_{SEC}} + \sum_{1 \leq w \leq \mathtt{d}} \hat{a}_w \alpha_i^w \right)$$

$$\forall i \in \{1, \ldots, \mathtt{n}\} \ \ c_i = (g^{\hat{r}_i}, \bar{h}^{\mathtt{sh}(\mathtt{IDcred_{SEC}})_i} \mathtt{pk}_{\mathtt{PG}_i}^{\hat{r}_i})$$

we can use the same Σ-protocol as in the account opening (cf. Section 7.1.5).

- To prove inequalities or set-(non)-membership for values in C_j for $j \in J_{ge} \cup J_{in} \cup J_{out}$ we can use Bulletproofs described in Sections 9.5. The Bulletproofs are sequentially composed to the Σ-protocols.

15.5 Identity Disclosure

This section describes the protocols needed to identify an account holder or to link the accounts of a given account holder.

15.5.1 Account Identity Disclosure

Disclosing the identity of an account is done by disclosing the identity of the account credentials (and their holders).

The identity of an account credential $\mathtt{AC}$ can be disclosed with the help of at least $\mathtt{d} + 1$ of the assigned $\mathtt{n}$ privacy guardians together with the identity provider who issued the identity credential to the account holder.

Protocol dicloseIdentityOfAccountCredential

The $\mathtt{d} + 1$ privacy guardians and the identity provider take an account credential $\mathtt{AC}$ as global input.
1. The privacy guardians jointly decrypt $\mathtt{AC.PGdata_{ACC}}$ to learn $\mathtt{IDcred_{PUB}}$ (cf. Section 7.1) and send it to the identity provider.
2. The identity provider retrieves (from its local database) the corresponding account holder identity record $\mathtt{AHIR}$ where $\mathtt{AHIR.IDcred_{PUB}} = \mathtt{IDcred_{PUB}}$.
3. **Output:** The identity provider returns the account holder's attributes $\mathtt{AHIR.attrList}$.

</> In practice, the identity provider may have even more information on the account holder. For example, they may have an image of the passport data page.

226

15.5.2 Detection of Account or Account Credentials with Duplicate ID

> ℹ️ In the above, we assume that duplicates would be detected when validating account creation transactions or credential management transactions. If this is not feasible, e.g., due to efficiency reasons, the following passive method can be used instead.

In case two accounts with the same $\mathtt{RegID_{ACC}}$ or two account credentials with the same $\mathtt{ID_{AC}}$ are detected one can disclose the identity of the accounts or credentials as described in Section 15.5.1. This allows to apply sanctions to the account holder.

> 💡 For this to work, incentives should be created so that parties will search for duplicates and alert the relevant authorities.

> </> The current implementation prevents duplicates from occurring.

15.5.3 Account Identification of Account Holder

Given attributes of an account holder that got issued an identity credential by identity provider $\mathtt{IDP}$, the identity provider together with $\mathtt{d} + 1$ of the privacy guardians can find all accounts that have/had an account credential assigned to account holders with matching attributes. More generally, a predicate over attributes can be used to identify all accounts with attributes satisfying this predicate. For example, one can search for all account holders with a certain name and birth year, ignoring the precise birthdate and other attributes.

Protocol identifyAllAccounts

The $\mathtt{d} + 1$ privacy guardians and the identity provider take (a predicate of) account holder attributes as global input.
1. The identity provider retrieves (from its local storage) all account holder identity record $\mathtt{AHIR}$ with matching attributes.
2. The identity provider sends all corresponding $\mathtt{AHIR.PRFkeyEnc_{ACC}}$ to the privacy guardians.
3. The privacy guardians decrypt all $\mathtt{AHIR.PRFkeyEnc_{ACC}}$ to learn the corresponding $\mathtt{key_{PRF}}$ (cf. Section 7.1) and send them back to the identity provider.
4. The identity provider can compute all possible $\mathtt{ID_{AC}}$ using the obtained $\mathtt{key_{PRF}}$. This allows the identity provider to find on-chain all matching account credentials.
5. **Output:** The identity provider returns a list of all found account credentials (and accounts).

Chapter 16

Konto Execution Layer

This chapter describes the cryptographic aspects of the execution layer Konto. It does not cover other aspects, such as the smart contract language and its interpreter. We assume that there is a native token which can be used for transfers and transaction fees.

> </> In the implementation, the native token is the CCD.

> (i) The notation in this chapter has been updated from previous versions to align better with external terminology. See Table 16.1 for an overview.

Table 16.1 Overview on updated terminology.

New Term	Old Term
plain transfer	plaintext transfer
shielded transfer	encrypted transfer
shielded amount	secret amount

16.1 Accounts

Accounts are assigned to an account holder and contain a public balance and a set of shielded amounts.

Definition 108. An *account* $\mathtt{ACC} \in \mathrm{ACCOUNTS}$ consists of the following values.

$\mathtt{AI} \in \mathrm{AIs}$
> The account information which contains, among other things, the public encryption key required to receive shielded amounts.

$\mathtt{publicBalance} \in \mathbb{N}$
> The *public balance* of the native token.

$\mathtt{shieldedBalance} \in \mathrm{CIPHERTEXTS_{EG}}^*$
> A list of shielded amounts $\mathtt{shieldedBalance} = (S_1, \ldots, S_n)$ where ciphertext $S_i = \mathtt{Enc_{EG}}(\mathtt{AI.pk_{ACC}}, \bar{h}^{\alpha_i})$. The sum $\alpha = \sum_{i=1}^{n} \alpha_i$ is the *shielded balance* of the native token.

> ⚠ The α_i in the encrypted amounts are not necessarily small and thus cannot be efficiently decrypted. However, the amounts can be reconstructed from the transaction history if the account holder lost knowledge on them.

> </> Starting from Protocol Update 7, shielded transfers (cf. Section 16.3) are considered a deprecated feature. The protocol only allows to move funds from the shielded balance to the public one.

16.1.1 Signing Transactions

The execution layer supports different types of *transactions*, such as simple transfers or smart contract calls. All transactions must be signed by the account holder(s) from which the transaction originates. When a transaction is published, validators will check the signature and allow the transaction to take effect if the signature is valid and its payload is sound. The syntax and semantics of sound payloads depends on aspects of the execution layer that are (for most part) not covered here.

> ℹ See Section 24.4 for more details on the serialization of transactions.

16.1.2 Transaction Fees

For simplicity, we do not cover *transaction fees* in the description below. Transaction fees can be deducted from the public balance of the origin account. Transactions are only valid if there is enough balance to cover the transaction fee.

16.2 Plain Transfer

Plain transfers can be used by an account holder $\mathtt{AH_1}$ with an account $\mathtt{ACC_1}$ to send the native token to account $\mathtt{ACC_2}$ of account holder $\mathtt{AH_2}$.

Protocol PlainTransfer

Protocol to transfer public amount p from $\mathtt{ACC_1}$ to $\mathtt{ACC_2}$ with $\mathtt{AI_1}$ resp. $\mathtt{AI_2}$
1. An account holder $\mathtt{AH_1}$ signs transaction $(\mathtt{AI_1.RegID_{ACC}}, \mathtt{AI_2.RegID_{ACC}}, p)$ under $\mathtt{AI_1.vk_{ACC}}$ and sends it to the blockchain.
2. To validate the transaction $(\mathtt{AI_1.RegID_{ACC}}, \mathtt{AI_2.RegID_{ACC}}, p)$, validators check
    (a) The validity of the signature,
    (b) and that $\mathtt{ACC_1.publicBalance} > p$.
   If the checks pass the accounts are updated as follows
    (a) $\mathtt{ACC_1.publicBalance} := \mathtt{ACC_1.publicBalance} - p$
    (b) $\mathtt{ACC_2.publicBalance} := \mathtt{ACC_2.publicBalance} + p$

16.3 Shielded Transfers

> </> Starting from Protocol Update 7, this section describes a deprecated feature. The protocol will only allow to move shielded balances to the public ones.
> However, validators still need to be able to validate all (past) block for catching-up from genesis, hence we provide the description below.

Shielded transfers hide the amount sent by updating the shielded amounts of the involved accounts. The sender needs to prove in zero-knowledge that the sender's shielded balance was computed and encrypted correctly, based on the old balance and the amount to be sent (which are all encrypted).

16.3.1 Shielded Transfer

In this section, we describe how to execute a shielded transfer from account $\texttt{ACC}_1$ to account $\texttt{ACC}_2$. We assume that the involved shielded balance of account $\texttt{ACC}_1$ consists of a single ciphertext S containing amount α. Ciphertext S can be computed from a subset of $\texttt{ACC}_1.\texttt{shieldedBalance}$ using the homomorphic property of the encryption scheme.

Transaction Creation

We assume that $\texttt{ACC}_1$ has account information $\texttt{AI}_1$ and private account information $\texttt{PAI}_1$. Account $\texttt{ACC}_2$ has account information $\texttt{AI}_2$. To send amount a in shielded fashion one creates a transaction consisting of S, the new balance S' (a ciphertext containing $\alpha - a$), and the ciphertext A containing a. To ensure correctness of these ciphertexts a zero-knowledge proof is added as well.

Protocol genEncTrans

Inputs

$\texttt{AI}_1 \in \mathrm{AIs}$
    Account information of the sender's account $\texttt{ACC}_1$.

$\texttt{PAI}_1 \in \mathrm{PAIs}$
    Private account information of the sender's account.

$\texttt{AI}_2 \in \mathrm{AIs}$
    Account information of the receiver's account $\texttt{ACC}_2$.

$S = (S_1, S_2) \in \mathrm{CIPHERTEXTS_{EG}}$
    Encryption of current shielded balance of the sender's account $\texttt{ACC}_1$. Note that is only one ciphertext, possibly already aggregated from several S_i.

$\alpha \in \mathbb{N}$
    The amount encrypted in S.

$a \in \mathbb{N}$
    Amount to be sent to account $\texttt{ACC}_2$.

Outputs

$t \in \{0,1\}^* \times \mathrm{REGACCIDs}^2 \times \mathrm{CIPHERTEXTS_{EG}} \times \mathrm{CIPHERTEXTS_{ACC}}^2 \times \Sigma\text{-}\mathrm{BULLETPROOFS}$
    The shielded transfer transaction.

$\sigma^{\texttt{ACC}} \in \mathrm{SIGNATURES_{ACC}}$
    The signature of account $\texttt{ACC}_1$ on the transaction.

Promise

$a \leq \alpha.$

Implementation

1: $\alpha' := \alpha - a$

2: split α', and a into parts of size $\mathtt{s}$ each, $\alpha'_1, \ldots, \alpha'_{\mathtt{t}}, a_1, \ldots, a_{\mathtt{t}}$

3: $A := \mathsf{Enc}_{\mathsf{ACC}}^{\mathtt{t,s}}(\mathtt{AI}_2.\mathtt{pk}_{\mathsf{ACC}}, a)$ *// Let $r_1, \ldots, r_{\mathtt{t}}$ be randomness of encryption*

4: $S' := \mathsf{Enc}_{\mathsf{ACC}}^{\mathtt{t,s}}(\mathtt{AI}_1.\mathtt{pk}_{\mathsf{ACC}}, \alpha')$ *// Let $r'_1, \ldots, r'_{\mathtt{t}}$ be randomness of encryption*

5: */* Create non-interactive proof (see below for details)*   */

6: $(\varsigma, \mathsf{BPInfo}_1, \mathsf{BPInfo}_2) := \mathsf{genEncTransProofInfo}(\mathtt{AI}_1.\mathtt{pk}_{\mathsf{ACC}}, \mathtt{AI}_2.\mathtt{pk}_{\mathsf{ACC}}, S, A, S')$

7: */* Context String*   */

8: $\mathtt{ctx} := \text{"encTrans"} \parallel \mathtt{AI}_1.\mathtt{RegID}_{\mathsf{ACC}} \parallel \mathtt{AI}_2.\mathtt{RegID}_{\mathsf{ACC}} \parallel \mathsf{H}(\text{genesis block})$

9: */* Σ-Protocol witness*   */

10: $x_\Sigma := (\mathtt{PAI}_1.\mathtt{dk}_{\mathsf{ACC}}, (\mathtt{PAI}_1.\mathtt{dk}_{\mathsf{ACC}}, \alpha), (r_1, a_1, r_1), \ldots, (r_{\mathtt{t}}, a_{\mathtt{t}}, r_{\mathtt{t}}), (r'_1, \alpha'_1, r'_1), \ldots,$
    $(r'_{\mathtt{t}}, \alpha'_{\mathtt{t}}, r'_{\mathtt{t}}))$

11: */* Bulletproof witness*   */

12: $x_{\mathsf{BP}} := (a_1, \ldots, a_{\mathtt{t}}, \alpha'_1, \ldots, \alpha'_{\mathtt{t}}, r_1, \ldots, r_{\mathtt{t}}, r'_1, \ldots, r'_{\mathtt{t}})$

13: $\Sigma\text{-bp} := \mathsf{genSigmaANDBulletproof}(\mathtt{ctx}, \varsigma, (\mathsf{BPInfo}_1, \mathsf{BPInfo}_2), x_\Sigma, x_{\mathsf{BP}})$

14: */* transaction contains IDs of sender and receiver, encryptions of the sender's new*
    encrypted balance and the sent amount, and the proof   */

15: $t := (\text{"encTrans"}, \mathtt{AI}_1.\mathtt{RegID}_{\mathsf{ACC}}, \mathtt{AI}_2.\mathtt{RegID}_{\mathsf{ACC}}, S, A, S', \Sigma\text{-bp})$

16: $\sigma^{\mathsf{ACC}} := \mathsf{Sign}_{\mathsf{ACC}}(\mathtt{PAI}.\mathtt{sk}_{\mathsf{ACC}}, t)$

17: **return** $(t, \sigma^{\mathsf{ACC}})$

!   The amount α is given directly as input to $\mathsf{genEncTrans}$. Since S is an encryption in the exponent without splitting, α cannot be recovered from S efficiently. Note, however, that α can be recovered from the transaction history.

i   In contrast, both S' and A' are ElGamal "with data in the exponent", i.e., they are vectors of ElGamal ciphertexts, where the encrypted amount is split into parts (cf. Section 7.1.4). This ensures that both amounts can be decrypted efficiently which is crucial for the current protocol.

♡   A possible optimization would be to encrypt the new balance S' also without splitting. That would save the additional ciphertexts in the transaction. However, this would then require an additional bulletproof for the fact that α' is "non-negative". Since that would be a proof for a different range, it could not be aggregated with the other bulletproof for A. A further downside of that approach would be that the sender would not be able to recover α' efficiently, and thus further shielded transfers would not be possible efficiently.

The required information for the zero-knowledge correctness proof can be generated as follows.

Protocol genEncTransProofInfo

Inputs

$\text{pk}_{\text{ACC}_1} \in \text{PUBLICKEYS}_{\text{ACC}}$
    Public encryption key of the sender.

$\text{pk}_{\text{ACC}_2} \in \text{PUBLICKEYS}_{\text{ACC}}$
    Public encryption key of the receiver.

$S = (S_1, S_2) \in \text{CIPHERTEXTS}_{\text{EG}}$
    Encryption of current shielded balance α of the sender's account ACC_1. Note that is only one ciphertext, possibly already aggregated from several S_i.

$A \in \text{CIPHERTEXTS}_{\text{ACC}}$
    Encryption of the amount a to be sent to account ACC_2.

$S' \in \text{CIPHERTEXTS}_{\text{ACC}}$
    Encryption of new shielded balance α' of the sender's account ACC_1.

Outputs

$\varsigma \in \Sigma\text{-INFORMATIONS}$
    A Σ-protocol proof information.

$(\text{BPInfo}_1, \text{BPInfo}_2) \in \text{BP-INFORMATIONS}^2$
    Bulletproof information.

Implementation

1: /* generate $\text{PK}\left\{\text{dk}_{\text{ACC}} : \text{pk}_{\text{ACC}_1} = g^{\text{dk}_{\text{ACC}}}\right\}$ */

2: $\varsigma_1 := \text{genSigProofInfo}(\text{dlog}, (g, q), \text{pk}_{\text{ACC}_1})$

3: /* generate $\text{PK}\left\{(\text{dk}'_{\text{ACC}}, \alpha) : S_2 = \bar{h}^\alpha \cdot S_1^{\text{dk}'_{\text{ACC}}}\right\}$ */

4: $\varsigma_2 := \text{genSigProofInfo}(\text{elg-dec}, (\bar{h}, q, S_1), S_2)$

5: /* Show that A is correct encryption of a split into parts of size s */

6: $(\varsigma_3, \text{BPInfo}_1) := \text{genEncExpInfo}((g, \bar{h}), \text{pk}_{\text{ACC}_2}, q, \mathsf{s}, A)$

7: /* Show that S' is correct encryption of α' split into parts of size s */

8: $(\varsigma_4, \text{BPInfo}_2) := \text{genEncExpInfo}((g, \bar{h}), \text{pk}_{\text{ACC}_1}, q, \mathsf{s}, S')$

9: $\varsigma' := \text{genAndComp}(\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4)$

10: /* Show that dk_{ACC} in ς_1 and ς_2 are equal. */

11: let l_1 be position of dk_{ACC} from ς_1 in ς'

12: let l_2 be position of dk'_{ACC} from ς_2 in ς'

13: $\varsigma'' := \text{genEqComp}(\varsigma', \{l_1, l_2\})$

14: /* Show that $\alpha' = \alpha - a$. */

15: /* Since $a = \sum_{j=1}^{\mathsf{t}} 2^{\mathsf{s}\cdot(j-1)} a_j$ and $\alpha' = \sum_{j=1}^{\mathsf{t}} 2^{\mathsf{s}\cdot(j-1)} \alpha'_j$, we show */

16: /* $\alpha = \sum_{j=1}^{\mathsf{t}} 2^{\mathsf{s}\cdot(j-1)} a_j + \sum_{j=1}^{\mathsf{t}} 2^{\mathsf{s}\cdot(j-1)} \alpha'_j$. */

17: let i_0 be position of α in ς''

18: let l_{a_j} be position of a_j in ς''

19: let $l_{\alpha'_j}$ be position of α'_j in ς''

20: $v_{a_j} := 2^{\mathsf{s}\cdot(j-1)}$

21: $v_{\alpha'_j} := 2^{\mathsf{s}\cdot(j-1)}$

22: $\varsigma := \text{genLinRelCompEx}(\varsigma'', \left\{l_{a_1}, \ldots, l_{a_{\mathsf{t}}}, l_{\alpha'_1}, \ldots, l_{\alpha'_{\mathsf{t}}}\right\}, i_0, (v_{a_1}, \ldots, v_{a_{\mathsf{t}}}, v_{\alpha'_1}, \ldots, v_{\alpha'_{\mathsf{t}}}))$

23: **return** $(\varsigma, \text{BPInfo}_1, \text{BPInfo}_2)$

We also need that the values encrypted in A and S' are non-negative. Since the encrypted values live in the exponent of a group, being negative would mean something like being larger than half the group order. We do not have to generate an explicit proof for that, because `genEncExpInfo` already guarantees that the value was split correctly and thus lies in the correct range.

The bulletproofs $BPInfo_1$ and $BPInfo_2$ use different generators for the commitments (since they are encryptions for different public keys) and thus cannot be aggregated.

An alternative solution to providing two independent bulletproofs would be to create additional commitments (with the same generators), use Σ-protocols to prove that they are commitments to the same values as the encrypted ones, and perform the bulletproofs on the commitments, which could then be aggregated.

Details on the different zero-knowledge proof components are found in Sections 9.2 and 9.6.4.

Transaction Verification

Shielded transfer transactions are verified as follows.

Protocol verifyEncTrans

Inputs

$t \in \{0,1\}^* \times \text{REGACCIDS}^2 \times \text{CIPHERTEXTS}_{\text{EG}} \times \text{CIPHERTEXTS}_{\text{ACC}}^2 \times \Sigma\text{-BULLETPROOFS}$
   Transaction to verify.

$\sigma^{\text{ACC}} \in \text{SIGNATURES}_{\text{ACC}}$
   Signature on the transaction.

Outputs

$b \in \text{BOOL}$
   Returns true iff the transaction is valid.

Implementation

1: $(\text{“encTrans”}, \text{AI}_1.\text{RegID}_{\text{ACC}}, \text{AI}_2.\text{RegID}_{\text{ACC}}, S, A, S', \Sigma\text{-bp}) := t$
2: let AI_1 and AI_2 be the account information with IDs $\text{AI}_1.\text{RegID}_{\text{ACC}}$ and $\text{AI}_2.\text{RegID}_{\text{ACC}}$, respectively
3: /* Verify transaction signature. */
4: **if** $\text{Verify}_{\text{ACC}}(\text{AI}_1.\text{vk}_{\text{ACC}}, t, \sigma^{\text{ACC}}) = \texttt{false}$ **then**
5: $\quad$ **return false**
6: /* Verify consistency of S. */
7: **if** `shieldedBalance` of account $\text{AI}_1.\text{RegID}_{\text{ACC}}$ does not contain subset that can be aggregated (via $\text{HomMultEnc}_{\text{EG}}$) to S **then**
8: $\quad$ **return false**
9: /* Verify proof of correctness. */
10: $(\varsigma, \text{BPInfo}_1, \text{BPInfo}_2) := \text{genEncTransProofInfo}(\text{AI}_1.\text{pk}_{\text{ACC}}, \text{AI}_2.\text{pk}_{\text{ACC}}, S, A, S')$
11: $\text{ctx} := \text{“encTrans”} \parallel \text{AI}_1.\text{RegID}_{\text{ACC}} \parallel \text{AI}_2.\text{RegID}_{\text{ACC}} \parallel \text{H}(\text{genesis block})$
12: **return** $\text{verifySigmaANDBulletproof}(\text{ctx}, \varsigma, (\text{BPInfo}_1, \text{BPInfo}_2), \Sigma\text{-bp})$

> ⓘ Under typical conditions, S will be the aggregation of all values in `shieldedBalance`. However, if a user receives a shielded transfer after submitting one, `shieldedBalance` changes. We therefore allow S to correspond to some subset. Finding this subset can be made more efficient for the validators, e.g., by including a timestamp or a hash of the last block the user has seen in the transaction.

> </> In the current implementation, only prefixes are allowed as subsets, i.e., reordering of amounts is not considered valid. This can lead to transactions submitted by users to be deemed invalid so that the users need to submit them again.

> ⓠ When a shielded amount is sent, one can also add a proof that $a \in [\ell, u]$ for some interval $[\ell, u]$. This could be useful for compliance purposes where one needs to show that the transaction amount is not too large.

Transaction Effect

After verifying a transaction $t = (\text{``encTrans''}, \texttt{AI}_1.\texttt{RegID}_{\texttt{ACC}}, \texttt{AI}_2.\texttt{RegID}_{\texttt{ACC}}, S, A, S', \Sigma\text{-bp})$ with verifyEncTrans, validators update the shielded balances of the sender and receiver as follows:

Let $(S_1, \ldots, S_n)$ be the subset of the shielded amount `shieldedBalance` of account $\texttt{ACC}_1$ that gets aggregated to S, i.e., $S = \mathsf{HomMultEnc_{EG}}(S_1, \ldots, S_n)$. Then, set $\texttt{shieldedBalance}_1$ of the sender's account $\texttt{ACC}_1$ to

$$\texttt{shieldedBalance}_1 := \big(\texttt{shieldedBalance}_1 \setminus \{S_1, \ldots, S_n\}\big) \,\|\, \mathsf{JoinEncExp^{t,s}_{EG}}(S'),$$

and set $\texttt{shieldedBalance}_2$ of the receiver to

$$\texttt{shieldedBalance}_2 := \texttt{shieldedBalance}_2 \,\|\, \mathsf{JoinEncExp^{t,s}_{EG}}(A).$$

Here, $\mathsf{JoinEncExp^{t,s}_{EG}}$ converts the "ElGamal with data in the exponent" encryption into a single ElGamal ciphertext (cf. Section 7.1.4).

Incoming Transaction

An account holder receiving a shielded transfer, $t = (\text{``encTrans''}, \texttt{AI}_1.\texttt{RegID}_{\texttt{ACC}}, \texttt{AI}_2.\texttt{RegID}_{\texttt{ACC}}, S, A, S', \Sigma\text{-bp})$ needs to update their local view on the shielded balance of their account. To do so, they decrypt A (via $\mathsf{Dec^{t,s}_{ACC}}(\texttt{PAI.dk}_{\texttt{ACC}}, A)$), and update the local view of the current shielded balance and the corresponding encryptions. Since users only need to know the latest and maximally aggregated shielded balance, they can directly set

$$\texttt{shieldedBalance} := \mathsf{HomMultEnc_{EG}}\big(\texttt{shieldedBalance}, \mathsf{JoinEncExp^{t,s}_{EG}}(A)\big).$$

16.3.2 Transfers From Public to Shielded Balance

An account holder can transfer tokens from their account's public balance `publicBalance` to the account's shielded balance `shieldedBalance`. The necessary protocol is described below.

Transaction Creation

To *shield* an amount a, the account holder needs to encrypt a as described in Section 16.3 and prove that the encryption was done correctly. Since the amount is publicly known, the correctness proof does not need to be zero-knowledge. This allows us to improve efficiency with the following trick: By using a default randomness (e.g., $r = 1$) to encrypt, everyone can deterministically create the resulting ciphertext. Consequently, the ciphertext is not required to be part of the transaction and no proofs are required.

More formally, such a transaction can be created as follows.

Protocol genPubToSecTrans

Inputs

$\text{AI} \in \text{AIs}$
    Account information of the sender's account `ACC`.

$\text{PAI} \in \text{PAIs}$
    Private account information of the sender.

$a \in \mathbb{N}$
    Amount to transfer.

Outputs

$t \in \{0,1\}^* \times \text{REGACCIDs} \times \mathbb{N}$
    A transaction.

$\sigma^{\text{ACC}} \in \text{SIGNATURES}_{\text{ACC}}$
    Signature from `ACC` on the transaction

Promise

    `publicBalance` of the account corresponding to `AI` is at least a.

Implementation

1: $t := (\text{"pubToSecTrans"}, \text{AI.RegID}_{\text{ACC}}, a)$
2: $\sigma^{\text{ACC}} := \text{Sign}_{\text{ACC}}(\text{PAI.sk}_{\text{ACC}}, t)$
3: **return** (t, σ^{ACC})

Transaction Verification

Verification only needs to check the signature and that the public balance is sufficient for the transfer. It works as follows.

Protocol verifyPubToSecTrans

Inputs

$t \in \{0,1\}^* \times \text{REGACCIDs} \times \mathbb{N}$
    Transaction to verify.

$\sigma^{\texttt{ACC}} \in \text{SIGNATURES}_{\texttt{ACC}}$
      Signature on the transaction.

Outputs

$b \in \text{BOOL}$
      Returns true iff the transaction is valid.

Implementation

1: $(\text{``pubToSecTrans''}, \texttt{AI.RegID}_{\texttt{ACC}}, a) \coloneqq t$
2: let $\texttt{AI}$ be the account information with ID $\texttt{AI.RegID}_{\texttt{ACC}}$
3: **if** $\texttt{AI.publicBalance} < a$ **then**
4:     $\llcorner$ **return** false
5: **return** $\text{Verify}_{\texttt{ACC}}(\texttt{AI.vk}_{\texttt{ACC}}, t, \sigma^{\texttt{ACC}})$

Transaction Effect

After verifying a transaction $t = (\text{``pubToSecTrans''}, \texttt{AI.RegID}_{\texttt{ACC}}, a)$ with verifyPubToSecTrans, validators (and the sender) update the balances of the sender's account as follows:

Set $r \coloneqq 1$ and compute $A \coloneqq \text{Enc}_{\text{EG}}(\texttt{AI.pk}_{\texttt{ACC}}, \bar{h}^a; r)$, i.e., use $r = 1$ as the randomness to encrypt a in the exponent (without splitting). Then, set $\texttt{shieldedBalance}$ of the sender to

$$\texttt{shieldedBalance} \coloneqq \texttt{shieldedBalance} \parallel A,$$

and set $\texttt{publicBalance}$ to

$$\texttt{publicBalance} \coloneqq \texttt{publicBalance} - a.$$

16.3.3 Transfers From Shielded to Public Balance

A transfer of amount a from $\texttt{shieldedBalance}$ to $\texttt{publicBalance}$ of an account $\texttt{ACC}$ is similar to a shielded transfer between accounts as in Section 16.3, except that the transferred amount a is not encrypted. The user still needs to update the shielded balance by subtracting the transferred amount and prove correctness of the encryption.

Transaction Creation

We assume that the account has account information $\texttt{AI}$, private account information $\texttt{PAI}$, and S is the shielded balance with shielded amount α. To *unshield* an amount a one creates a transaction consisting of S, the new balance S' (a ciphertext containing $\alpha - a$), and a. To ensure correctness of these ciphertexts a zero-knowledge proof is added as well.

Protocol genSecToPubTrans

Inputs

$\texttt{AI} \in \text{AIs}$
      Account information of the sender's account $\texttt{ACC}$.
$\texttt{PAI} \in \text{PAIs}$
      Private account information of the sender.
$S = (S_1, S_2) \in \text{CIPHERTEXTS}_{\text{EG}}$
      Encryption of the shielded amount of $\texttt{ACC}$. Note that is only one ciphertext, possibly

already aggregated from several S_i.

$\alpha \in \mathbb{N}$

    The shielded amount encrypted in S.

$a \in \mathbb{N}$

    Amount to be unshielded.

Outputs

$t \in \{0,1\}^* \times \textsc{RegAccIDs} \times \textsc{ciphertexts}_{\text{EG}}{}^2 \times \mathbb{N} \times \Sigma\text{-}\textsc{Bulletproofs}$

    A transaction.

$\sigma^{\texttt{ACC}} \in \textsc{signatures}_{\texttt{ACC}}$

    Signature from $\texttt{ACC}$ on the transaction.

Promise

    $a \leq \alpha$.

Implementation

1: $\alpha' := \alpha - a$

2: compute $S' := \mathsf{Enc}_{\text{EG}}(\text{AI}.\mathrm{pk}_{\texttt{ACC}}, \bar{h}^{\alpha'})$ *// let r be randomness used for encryption*

3: $(\varsigma, \texttt{BPInfo}) := \mathsf{genSecToPubProofInfo}(\text{AI}.\mathrm{pk}_{\texttt{ACC}}, S, a, S')$

4: $\texttt{ctx} := \text{``secToPubTrans''} \parallel \text{AI}.\texttt{RegID}_{\texttt{ACC}} \parallel \mathsf{H}(\text{genesis block})$

5: $x_\Sigma := (\text{PAI}_1.\mathrm{dk}_{\texttt{ACC}}, (\text{PAI}_1.\mathrm{dk}_{\texttt{ACC}}, s), (r, \alpha', r), a)$

6: $x_{\text{BP}} := (\alpha', r)$

7: $\Sigma\text{-bp} := \mathsf{genSigmaANDBulletproof}(\texttt{ctx}, \varsigma, \texttt{BPInfo}, x_\Sigma, x_{\text{BP}})$

8: $t := (\text{``secToPubTrans''}, \text{AI}.\texttt{RegID}_{\texttt{ACC}}, S, S', a, \Sigma\text{-bp})$

9: $\sigma^{\texttt{ACC}} := \mathsf{Sign}_{\texttt{ACC}}(\text{PAI}.\mathrm{sk}_{\texttt{ACC}}, t)$

10: **return** $(t, \sigma^{\texttt{ACC}})$

The required information for the zero-knowledge correctness proof can be generated as follows.

Protocol genSecToPubProofInfo

Inputs

$\mathrm{pk}_{\texttt{ACC}} \in \textsc{publickeys}_{\texttt{ACC}}$

    Public encryption key of the sender (who is also receiver).

$S = (S_1, S_2) \in \textsc{ciphertexts}_{\text{EG}}$

    Encryption of shielded amount of the sender's account $\texttt{ACC}$.

$a \in \mathbb{N}$

    Amount to transfer.

$S' = (S'_1, S'_2) \in \textsc{ciphertexts}_{\text{EG}}$

    Encrypted amount corresponding to new balance of $\texttt{ACC}$.

Outputs

$\varsigma \in \Sigma\text{-}\textsc{Informations}$

    A Σ-protocol proof information.

$\texttt{BPInfo} \in \text{BP-}\textsc{Informations}$

    Bulletproof information.

Implementation

```
 1:  /* generate PK {dk_ACC : pk_ACC = g^{dk_ACC}}                                        */
 2:  ς₁ := genSigProofInfo(dlog, (g, q), pk_ACC)
 3:  /* generate PK {(dk'_ACC, α) : S₂ = h̄^α · S₁^{dk'_ACC}}                              */
 4:  ς₂ := genSigProofInfo(elg-dec, (h̄, q, S₁), S₂)
 5:  /* Show that S' is correct encryption of α'                                           */
 6:  ς₃ := genEncExpNoSplitInfo((g, h̄), pk_ACC, q, S')
 7:  /* Show knowledge of a                                                                */
 8:  ς₄ := genSigProofInfo(id, (ℤ_q, q), a)
 9:  ς' := genAndComp(ς₁, ς₂, ς₃, ς₄)
10:  /* Show that dk_ACC in ς₁ and ς₂ are equal.                                           */
11:  let l₁ be position of dk_ACC from ς₁ in ς'
12:  let l₂ be position of dk'_ACC from ς₂ in ς'
13:  ς'' := genEqComp(ς', {l₁, l₂})
14:  /* Next we want to show that α' = α − a.                                              */
15:  /* First introduce fresh t_α and t_a and show that α' = t_α − t_a                     */
16:  let i₀ be position of α' in ς''
17:  ς̂ := genLinRelComp(ς'', i₀, (1, −1))
18:  /* Now show that t_α = α and t_a = a                                                  */
19:  let l_α and l_{t_α} be positions of α and t_α in ς*, respectively
20:  ς* := genEqComp(ς̂, {l_α, l_{t_α}})
21:  let l_a and l_{t_a} be positions of a and t_a in ς̂, respectively
22:  ς := genEqComp(ς*, {l_a, l_{t_a}})
23:  /* Prove that α' ≥ 0. Since encryption is in exponent and thus modulo group order,
         being non-negative means being in the proper range (without underflow).          */
24:  /* Thus show that α' ∈ [0, 2^{maxCCDe} − 1], where maxCCDe is the bit length of CCD.   */
25:  BPInfo := genBPInfo(maxCCDe, (h̄, pk_ACC), S₂')
26:  return (ς, BPInfo)
```

Transaction Verification

Verification of the unshield transaction works as follows.

Protocol verifySecToPubTrans

Inputs

$t \in \{0, 1\}^* \times \text{REGACCIDS} \times \text{CIPHERTEXTS}_{\text{EG}}^2 \times \mathbb{N} \times \Sigma\text{-BULLETPROOFS}$
    Transaction to verify.

$\sigma^{\text{ACC}} \in \text{SIGNATURES}_{\text{ACC}}$
    Signature on the transaction.

Outputs

$b \in \text{BOOL}$
    Returns true iff the transaction is valid.

Implementation

```
1: ("secToPubTrans", AI.RegID_ACC, S, S', a, Σ-bp) := t
2: let AI be the account information with ID AI.RegID_ACC
3: /* Verify the account signature.                                                       */
```

```
 4: if Verify_ACC(AI.vk_ACC, t, σ^ACC) = false then
 5: ⌞  return false
 6: /* Verify consistency of S.                                              */
 7: if shieldedBalance of account AI.RegID_ACC does not contain subset that can be aggre-
    gated (via HomMultEnc_EG) to S then
 8: ⌞  return false
 9: /* Verify proof of correctness.                                          */
10: (ς, BPInfo) := genSecToPubProofInfo(AI.pk_ACC, S, a, S')
11: ctx := "secToPubTrans" ∥ AI.RegID_ACC ∥ H(genesis block)
12: return verifySigmaANDBulletproof(ctx, ς, BPInfo, Σ-bp)
```

Transaction Effect

After verifying a transaction $t = (\text{``secToPubTrans''}, \texttt{AI.RegID}_{ACC}, S, S', a, \Sigma\text{-bp})$ with verifySecToPubTrans, validators update the balances of the sender as follows:

Let $(S_1, \ldots, S_n)$ be the subset of $\texttt{shieldedBalance}$ that can be aggregated to S, that is $S = \mathsf{HomMultEnc}_{EG}(S_1, \ldots, S_n)$. Then, set $\texttt{shieldedBalance}$ of the sender to

$$\texttt{shieldedBalance} := (\texttt{shieldedBalance} \setminus \{S_1, \ldots, S_n\}) \parallel S',$$

and set $\texttt{publicBalance}$ of the sender to

$$\texttt{publicBalance} := \texttt{publicBalance} + a.$$

The account holder updates their local state accordingly.

16.4   Privacy Enhancing Extensions

This section provides ideas on how the functionality of the execution layer could be extended to provide more privacy to account holders.

> 💡 Shielded transfers only hide the amount being transferred, but not the sender or receiver. It is thus naturally to consider transfers where both the amount and the involved parties are not public. Such as system would have to be transparent for privacy guardians, so that it complies with regulations. This could be achieved using 'viewing keys', similar to the way shielded transfers are done.
>
> Among existing approaches for such a payment system, those based on accumulators appear to be most promising in terms of efficiency and privacy guarantees. For example, the recent work of Curve Trees [CHK23] constructs a Bulletproof-style accumulator, which could be used for this purpose.

Chapter 17

Web3 Verifiable Credentials

Web3 verifiable credentials are a *self-sovereign!identity* (SSI) system. In contrast to the ID layer Personligt which is limited to identity credentials, Web3 verifiable credentials cover various types of claims such as university degrees, chess club memberships, etc. These credentials are based on the Verifiable Credentials Data Model 2.0 [Spo+24], a standard set by the World Wide Web Consortium (W3C). See Figure 17.1 for an overview.

To issue a Web3 verifiable credential an issuer needs to deploy a registry smart contract. The contract acts as the issuer's identifier and contains information about the issued credentials. Specifically, each credential has an entry in the contract containing metadata about the credential and its status information, e.g., an expiration/activation date, revocation flag, etc. Decentralized Identifiers (DIDs) [SGS22] are used to identify credentials.

This chapter provides an overview of the credential system. More details are found in [Con23b; Con23a; Con23c].

17.1 Decentralized Identifiers

We use *decentralized identifiers* (DIDs) [SGS22] to refer to subjects in our *verifiable credential* model.
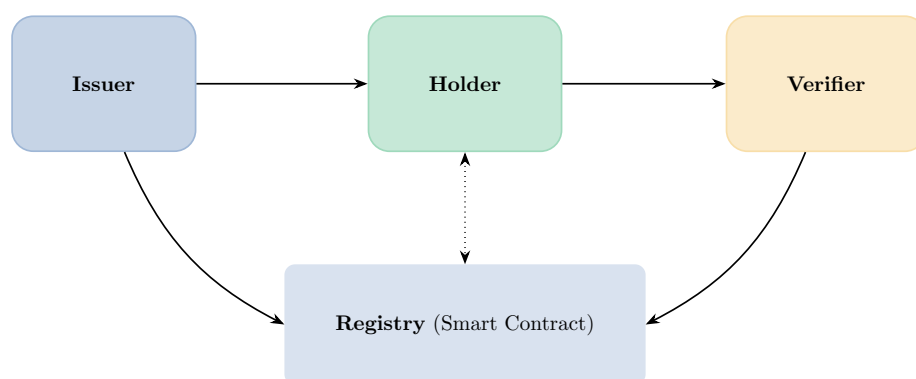
Figure 17.1 The issuer issues verifiable credentials to the holder. The registry contains status information on credentials. The holder can go to the verifier and show a presentation derived from the original credential.

17.1.1 Concordium DID

The *Concordium DID* is an extension of [SGS22] that allows referring to subjects represented on the Concordium blockchain, e.g., by means of an account. This section only provides an overview, see [Con23b] for the detailed specification.

All Concordium DIDs have the prefix `did:ccd:mainnet` for mainnet (resp. `did:ccd:testnet` for testnet).

Account DID. An *account DID* refers to an *account* on the blockchain. It has the form `did:ccd:NET:acc:ACCADDR` where `NET` refers to mainnet or testnet and `ACCADDR` is the account address.

Credential DID. A *credential DID* refers to an *account credential* on the blockchain. It has the form `did:ccd:NET:acc:CRED` where `NET` refers to mainnet or testnet and `CRED` is the account credential identifier.

Public Key DID. A *public key DID* refers to a public signature verification key. It has the form `did:ccd:NET:acc:PK` where `NET` refers to mainnet or testnet and `PK` is the public key. This is used to refer to the *holder* of a verifiable credential.

Smart contract DID. A *smart contract DID* refers to a *smart contract* on the blockchain. It has the form `did:ccd:NET:acc:SC(:SUB)` where `SC` refers to smart contract index and `SUB` refers to the optional subindex.

Identity provider DID. An *identity provider DID* refers to an *identity provider* on the blockchain. It has the form `did:ccd:NET:acc:IDP` where `NET` refers to mainnet or testnet and `IDP` is the identity provider index.

17.2 Roles

There are four roles in the Web3 verifiable credential system. Issuers issue verifiable credentials to holders. With a verifiable credential the issuer attests some statement about the credential's subject (often the holder). The holder can prove statements about their credentials to a verifier in a privacy preserving manner.

17.2.1 Issuer

An *issuer* issues verifiable credentials to a holder. An issuer can be identified by their registry smart contract (cf. Section 17.1.1). One can also see the identity providers from Personligt (cf. Section 15.1.2) as issuers.

17.2.2 Holder

A *holder* holds verifiable credentials, e.g., a university degree or a pet passport.

17.2.3 Subject

The credential *subject* is the entity a verifiable credential is about. Often, but not always, the subject is also the holder of a credential.

> **ⓘ** A pet passport, that is needed for traveling internationally with a dog, is an example
> where the subject (dog) is different from the holder (owner).

17.2.4 Verifier

A *verifier* wants to learn a statement about a credential from the holder. For example, the
holder could prove that they are older than 18 using an identity credential.

17.3 Registry

An issuer uses smart contracts as a *registry*. The registry contains information on the status of
issued verifiable credentials. In particular, the registry provides status information. For example,
it would indicate if a credential has been revoked. This section only provides an overview,
see [Con23a] for the detailed specification.

> **ⓘ** There is no (on-chain) registry for identity credentials issued by identity providers

17.4 Verifiable Credential

A *verifiable credential* is essentially a list of *attributes* of the credential subject signed by the issuer.
The credential *metadata* describes among other things the holder and the type of credential,
e.g., a membership. The Web3 credentials follow the standard defined in [Spo+24]. More details
are found in [Con23c].

17.4.1 Attribute Commitments

A Web3 ID credential stores a list of Pedersen *commitments* to the user's attributes, and a
signature of the issuer on the list of commitments. This allows generating presentations from
these credentials (cf. Section 17.5).

> **💡** In order to save space, one can optimize this by storing only a single vector Pedersen
> commitment to all attributes as described in Section 8.1.4. In this case, the Σ-protocol
> in Section 9.7.1 can be used to link the n-th value inside the vector commitment to the
> value inside another single commitment. This enables reusing existing protocols for ID
> layer for presentations. More technical details on how to adapt existing proofs for single
> commitments to vector Pedersen commitments can be found in Section 9.7.

17.5 Verifiable Presentation

A *verifiable presentation* is a special type of derived credential. It is generated by the holder from
existing credentials to be able to prove a statement to a verifier. For example, a presentation
from an identity credential could just contain the attribute "Is older than 18". As any other
credential a presentation comes with a proof for the attributes. In the simplest case, this is a
(randomized) signature from the issuer. However, to achieve fine-grained privacy the holder can
use zero-knowledge proofs. The Web3 presentations follow the standard defined in [Spo+24].
More details are found in [Con23c].

> **!** The trust level of proven statements depends on the trust level of the issuer.

> **!** To verify a presentation, the verifier checks the validity of the proof provided and the on-chain status of the original credential. Having the credential status on-chain ensures that the verifier does not need to contact the issuer directly (i.e., no phoning home).

Zero-Knowledge Proofs

Similar to presentation from identity credentials, a holder can use zero-knowledge proofs to make tailored presentations about their verifiable credentials. The proofs follow the template seen in Section 15.4. For more details see also Sections 9.2 and 9.5.

17.6   Key Management

Issuers and holders both need to securely manage cryptographic keys as part of handling credentials. The key management for issuers is not detailed here as it will depend on the issuer's individual setup. For credential holders one can use a seed phrase based approach as discussed in Section 18.4.

Chapter 18

Deterministic Key Derivation

The goal of (hardware) *deterministic wallets* is to reduce the wallet backup to a single secret, called the *deterministic key derivation!seed phrase*. All account keys can be generated from the seed phrase using a deterministic *key derivation function* (KDF). Often seed phrases are encoded as a vector of 24 words to make it human-readable. See for example *BIP-32* [Wui12] for more details.

18.1   Key Derivation Overview

We use the *SLIP-10* [HR16] key derivation scheme, a generalization of BIP-32. Here, keys are deterministically derived from the master secret using a path of the key derivation tree. More precisely, we use the variant of `Ed25519` SLIP-10 which allows us to derive `Ed25519` signature keys. To derive other keys, e.g., on `BLS12-381`, we derive the `Ed25519` private key for the corresponding path and use it as the randomness of the key generation function.

Seed phrase.   We follow the specification in SLIP-10 [HR16]. The backup of the binary seed should be encoded as a 24 word mnemonic seed phrase as defined in *BIP-39* [Pal+13].

Master key derivation.   We use the master key derivation function from SLIP-10 for `Ed25519`. This provides a private key $\texttt{masterkey} \in \text{SIGNKEYS}_{\texttt{EdDSA}} = \{0,1\}^{256}$ as the *master key*.

Child key derivation.   We use the child key derivation function from SLIP-10 for `Ed25519`. This directly provides a private key $\texttt{sk} \in \text{SIGNKEYS}_{\texttt{EdDSA}} = \{0,1\}^{256}$ as the *child key*. We always use a 'hardened' derivation (as required by SLIP-10 for `Ed25519`).

Extended key derivation.   Some use cases require keys that are not `Ed25519` signature keys. To derive such keys we use the following algorithm:

Algorithm deriveKey($\texttt{masterkey}, \texttt{path}, \texttt{type}$)

$\texttt{rnd} = \text{Ed25519SLIP10derivation}(\texttt{masterkey}, \texttt{path})$
return $\texttt{sk} = \text{keyGen}(\texttt{rnd}, \texttt{type})$

The keygen algorithms for different key `type` are described below.

EdDSA key The key generation function $\text{keyGen}(\cdot, \text{SIGNKEYS}_{\texttt{EdDSA}})$ for $\text{SIGNKEYS}_{\texttt{EdDSA}}$ is the identity.
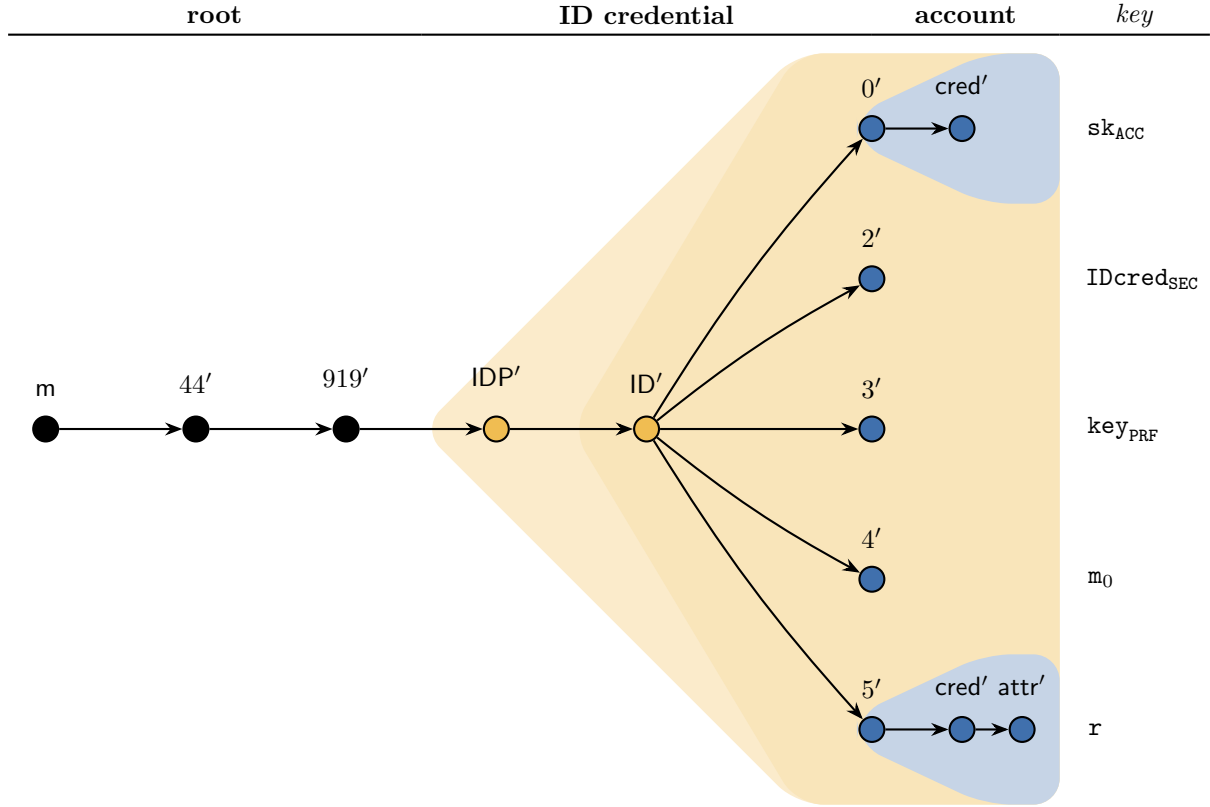
Figure 18.1 Key derivation tree for identity and account credentials.

BLS12-381 field element ($\mathbb{F}_{\text{BLS}}$) The key generation function $\mathsf{keyGen}(\cdot, \mathbb{F}_{\text{BLS}})$ is the KeyGen function from [Bon+20] where $\mathtt{rnd}$ acts as the IKM. This is used to generate randomness for Pedersen commitments (e.g., to attributes) over $\mathbb{G}_1^{\text{BLS}}$, $\mathtt{IDcred_{SEC}}$, $\mathtt{key_{PRF}}$, and the signature blinding $\mathtt{m_0}$. Naturally, this also allows us to generate actual BLS keys.

Commitment randomness For Pedersen commitments on $\mathbb{G}^{\text{Ed}}$ the key generation function is the identity.

AES-256 key The key generation function is the identity.

18.2 Key Derivation Tree for Identity and Account

The following *key derivation tree* structure is used by a wallet to generate the necessary keys and randomness for identity and account credentials. Its structure is similar to the one proposed in *BIP-44* (cf. [PR14]). We always use a 'hardened' derivation (as required by SLIP-10 for $\mathtt{Ed25519}$). Hardened derivations are denoted by apostrophes in the path (e.g., $\mathtt{m/purpose'/coin\_type'}$), and technically means that the highest-order bit of the integers is set to 1 such that they are $\geq 2^{31}$. That is, for some integer $\mathtt{x} < 2^{31}$, we define $\mathtt{x'} := \mathtt{x} + 2^{31}$. See Figure 18.1 for a depiction of the tree structure.

Root. The root of our tree is of the form

$$\mathtt{root} := \mathtt{m/purpose'/coin\_type'}$$

where

- the value $\mathtt{m}$ is a placeholder for the master secret key,

- the integer $\mathtt{purpose} \coloneqq 44$ acts as the ID of this derivation scheme (*BIP-44*),

- the integer $\mathtt{coin\_type} \coloneqq 919$ defines the derivation subtree for the Concordium mainnet (cf. [RP14]).

We use $\mathtt{coin\_type} = 1$ for testnet.

> ℹ️ Did you know that 919 is a palindromic prime and a centered hexagonal number?

Identity and account credential subtree. For the identity credential $\mathtt{IDC}$ with index $\mathtt{id}$ (starting at 0) registered at the identity provider with identifier $\mathtt{ID_{IDP}} = \mathtt{idp}$, the subtree

$$\mathtt{root/idp'/id'/*}$$

contains

- the *signature key* subtree $\mathtt{root/idp'/id'/0'/*}$ where $\mathtt{root/idp'/identity'/0'/cred'}$ is for the account credential signature key $\mathtt{AC.sk_{ACC}}$ for the account credential $\mathtt{AC}$ with index $\mathtt{cred}$ (starting at 0),

- the subtree $\mathtt{root/idp'/id'/1'/*}$ is reserved and not used,

- the *holder identifier* path $\mathtt{root/idp'/id'/2'}$ for $\mathtt{IDC.IDcred_{SEC}}$,

- the *PRF key* path $\mathtt{root/idp'/id'/3'}$ for $\mathtt{IDC.key_{PRF}}$,

- the *signature blinding randomness* path $\mathtt{root/idp'/id'/4'}$ for $\mathtt{IDC.m_0}$,

- the attribute *commitment randomness* subtree $\mathtt{root/idp'/id'/5'/*}$ where

$$\mathtt{root/idp'/id'/5'/cred'/attr'}$$

  is for the commitment randomness r for the attribute a with index $\mathtt{attr}$ (starting at 0) for the account credential $\mathtt{AC}$ with index $\mathtt{cred}$ (starting at 0).

Note that the account decryption key can be computed from the first credential and its identity. That is, from the PRF key at $\mathtt{root/idp'/id'/3'}$ and the credential index $\mathtt{cred}$ where the identity has index $\mathtt{id}$.

> ⚠️ The identity provider $\mathtt{idp}$ index corresponds to the identifier $\mathtt{ID_{IDP}}$ of the used identity provider. It can be found on the blockchain. For example, the first proper on mainnet identity provider has index 1. We *reserve* all indices $\mathtt{idp} \geq 2^{23}$ for future use, i.e., the highest 8 bits plus the bit indicating a hardened derivation. We also *reserve* index $\mathtt{idp} = 0$ for future use (this is the index of the non-functional genesis identity provider, cf. Section 22.4.1).

> ⓘ Our design extends the *BIP-44* standard to accommodate for our ID layer. In *BIP-44*, the subtree structure is `root/account′/change/address_index`. As mentioned above, we always use hardened derivations as in SLIP-10. Instead of the `account` field, we have two fields for the identity provider identifier and the index of the identity for that identity provider. Furthermore, our accounts correspond to the address indices in *BIP-44*. We finally (mis)use the "change" field in *BIP-44* for additional values not present in *BIP-44*, and skip the value $1′$, which *BIP-44* uses for so-called change addresses. This design allows users to have multiple identities from each identity provider and multiple accounts for each identity with a clean separation of the corresponding keys.

18.2.1 Deterministic ID and Credential Generation

This section provides deterministic seed-phrase based variants of the functions from Sections 15.2.1 and 15.3.1 required to generate all data to get an identity credential issued and to open accounts.

Function prepareIDIssuanceFromSeedPhrase

Inputs

  $\texttt{masterkey} \in \mathrm{SIGNKEYS_{EdDSA}}$
      Master secret key derived from seed phrase.

  $\texttt{idp} \in \mathrm{IDS_{IDP}}$
      Identity provider identifier.

  $\texttt{id} \in \mathbb{N}$
      Index of the to-be-created identity credential.

Outputs

  $\texttt{IDcred_{PUB}} \in \mathrm{IDCRED_{PUB}}$
      A public holder identifier.

  $\texttt{IDcred_{SEC}} \in \mathrm{IDCRED_{SEC}}$
      A secret holder identifier.

  $\texttt{key_{PRF}} \in \mathrm{PRFKEYS}$
      A PRF key.

  $\texttt{m}_0 \in \mathbb{F}_{\mathrm{BLS}}$
      Signature blinding randomness.

Implementation

1: $\texttt{IDcred_{SEC}} := \mathsf{deriveKey}(\texttt{masterkey}, \texttt{root}/\texttt{idp}′/\texttt{id}′/2′, \mathbb{F}_{\mathrm{BLS}})$
2: $\texttt{IDcred_{PUB}} := g^{\texttt{IDcred_{SEC}}}$
3: $\texttt{key_{PRF}} := \mathsf{deriveKey}(\texttt{masterkey}, \texttt{root}/\texttt{idp}′/\texttt{id}′/3′, \mathbb{F}_{\mathrm{BLS}})$
4: $\texttt{m}_0 := \mathsf{deriveKey}(\texttt{masterkey}, \texttt{root}/\texttt{idp}′/\texttt{id}′/4′, \mathbb{F}_{\mathrm{BLS}})$
5: **return** $(\texttt{IDcred_{PUB}}, \texttt{IDcred_{SEC}}, \texttt{key_{PRF}}, \texttt{m}_0)$

Function createCredentialFromSeedPhrase

Inputs

  $\texttt{masterkey} \in \mathrm{SIGNKEYS_{EdDSA}}$
      Master secret key derived from seed phrase.

IDC ∈ IDCs
> The identity credential of the account holder.

idp ∈ IDs_{IDP}
> Identity provider identifier.

id ∈ ℕ
> Index of the given identity credential

x ∈ ℕ
> The account credential number. It should be unique among all account credentials created using IDC. It must also be smaller than or equal to $\text{IDC.max}_{\text{AC}}$.

publicAttrPredicate ∈ $\{0,1\}^*$
> A predicate on the attributes the account holder chooses to reveal.

Outputs

AC ∈ ACs
> The account credential.

PAC ∈ PACs
> The private account credential.

Implementation

1: $\text{idp} := \text{IDC.ID}_{\text{IDP}}$
2: $\text{ID}_{\text{AC}} := \text{PRF}(\text{IDC.key}_{\text{PRF}}, x)$
3: $\text{sk}_{\text{ACC}} := \text{deriveKey}(\text{masterkey}, \text{root}/\text{idp}'/\text{id}'/0'/x', \text{SIGNKEYS}_{\text{EdDSA}})$.
4: Derive vk_{ACC} from sk_{ACC} as in [JL17]
5: **for** $a_i \in \text{IDC.attrList}$ $(i = 0, \ldots, \ell)$ **do**
6: $\quad r_i := \text{deriveKey}(\text{masterkey}, \text{root}/\text{idp}'/\text{id}'/5'/x'/i', \mathbb{F}_{\text{BLS}})$
7: $\quad C_i := \bar{g}_{\text{com}}^{r_i} \bar{h}_{\text{com}}^{a_i}$
8: $\text{PGdata}_{\text{ACC}} := \text{Enc}_{\text{PG}}^{\text{n,d}}(\text{PK}_{\text{PG}}, \text{IDC.IDcred}_{\text{PUB}})$, where PK_{PG} is the public-key set of the privacy guardians $\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}$ from IDC, and $\text{d} = \text{IDC.threshold}_{\text{PG}}$.
9: $\text{AC} := \big(\text{ID}_{\text{AC}}, \text{vk}_{\text{ACC}}, \text{threshold}_{\text{Cred}} := 1, (C_0, \ldots, C_\ell), \text{publicAttrPredicate}, \text{IDC.ID}_{\text{IDP}}, \text{IDC.}(\text{ID}_{\text{PG}_1}, \ldots, \text{ID}_{\text{PG}_n}), \text{IDC.threshold}_{\text{PG}}, \text{PGdata}_{\text{ACC}}\big)$
10: $\text{PAC} := \big(x, \text{sk}_{\text{ACC}}, (r_0, \ldots, r_\ell)\big)$

❗ Here, in contrast to the generic function createCredentialFromSeedPhrase from Section 15.3.1, we only allow creating credentials with a single signature key (and signature threshold 1). That is because it would not provide additional security to generate multiple keys from the same seed phrase. If a user wants to use multiple keys based on seed phrases, multiple credentials derived from different seed phrases must be used. This is primarily a restriction for wallets, the overall design would allow for multiple signature keys per credential (cf. Section 15.3.1).

18.3 Legacy Key Derivation Tree for Identity and Account

For completeness this section describes the older version of the key derivation for identity and account credentials. See Figure 18.2 for a depiction of the tree structure.

❗ This version of the key derivation should no longer be used.
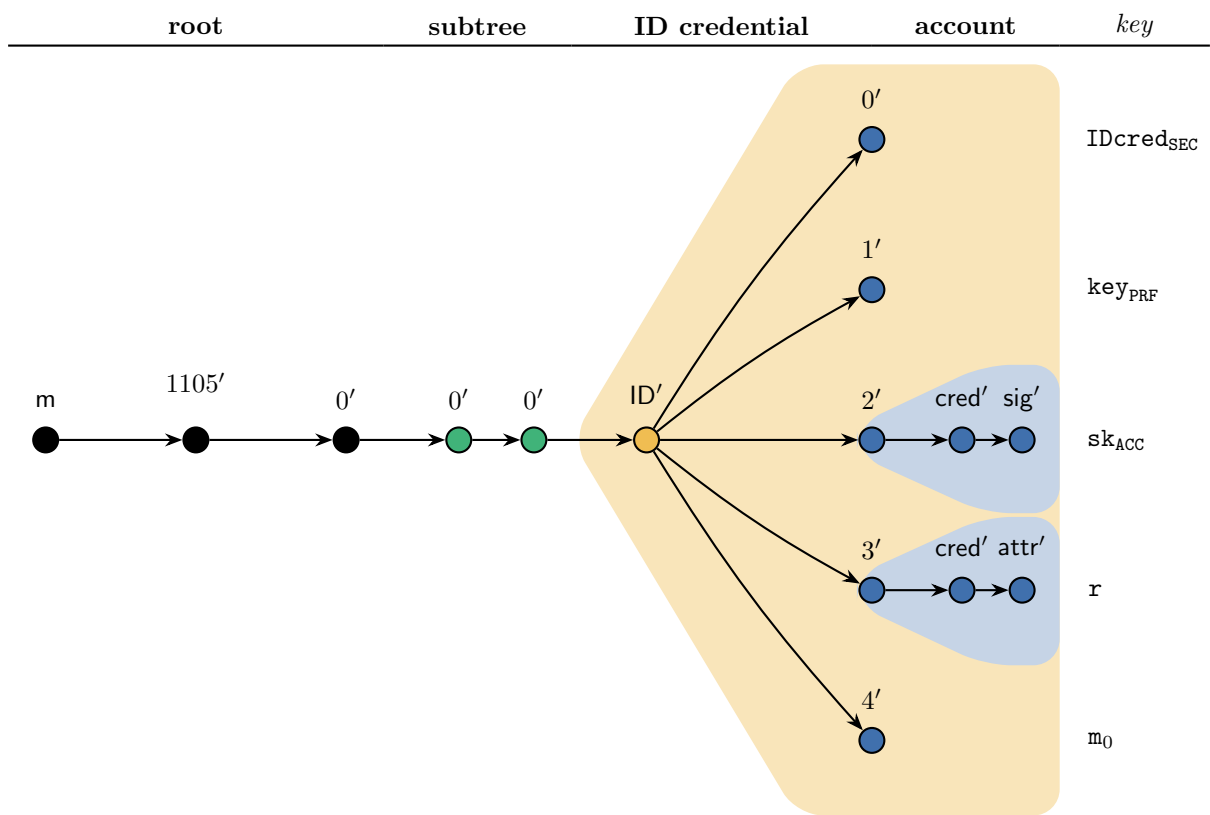
Figure 18.2 Legacy key derivation tree for identity and account credentials.

The legacy scheme uses the same hardened SLIP-10 derivation as the new one in Section 18.2, but differs in the paths.

Root.   The root of our tree is of the form

$$\texttt{root} := \texttt{m/purpose}'/\texttt{coin\_type}'/\texttt{acc}'/0'$$

where

- the value $\texttt{m}$ is a placeholder for the master secret key,
- the integer $\texttt{purpose} := 1105$ acts as the ID of this derivation scheme,
- the integer $\texttt{coin\_type} := 0$ defines the derivation subtree for the Concordium mainnet.
- the integer $\texttt{acc} := 0$ defines the account (and identity) subtree.

Identity and account credential subtree.   For identity credential $\texttt{IDC}$ with index $\texttt{id}$ (starting at 0) the subtree
$$\texttt{root/id}'/*$$
contains

- the holder identifier path $\texttt{root/id}'/0'$ for $\texttt{IDC.IDcred}_{\textsf{SEC}}$,
- the PRF key path $\texttt{root/id}'/1'$ for $\texttt{IDC.key}_{\textsf{PRF}}$,
- the signature key subtree $\texttt{root/id}'/2'/*$ where $\texttt{root/id}'/2'/\texttt{cred}'/\texttt{sig}'$ is for the credential signature key $\texttt{AC.sk}_{\textsf{ACC}}$ with index $\texttt{sig}$ (starting at 0) for the account credential $\texttt{AC}$ with index $\texttt{cred}$ (starting at 0),
- the attribute commitment randomness subtree $\texttt{root/id}'/3'/*$ where

$$\texttt{root/id}'/3'/\texttt{cred}'/\texttt{attr}'$$

  is for the commitment randomness r for the attribute a with index $\texttt{attr}$ (starting at 0) for the account credential $\texttt{AC}$ with index $\texttt{cred}$ (starting at 0),
- the signature blinding randomness path $\texttt{root/id}'/4'$ for $\texttt{IDC.m}_0$.

Note that the account decryption key can be computed from the first credential and its identity. That is, from the PRF key at $\texttt{root/id}'/1'$ and the credential index $\texttt{cred}$ where the identity has index $\texttt{id}$.

Reserved Subtrees.   The subtrees $\texttt{m/purpose}'/\texttt{coin\_type}'/\texttt{acc}'/1'/*'$, $\texttt{m/purpose}'/\texttt{coin\_type}'/1'/*'$, and $\texttt{m/purpose}'/\texttt{coin\_type}'/2'/*'$ are reserved but not used.

18.4   Key Derivation Tree Structure for Web3 Verifiable Credentials

For Web3 ID we extend our derivation tree from Section 18.2 with a new subtree at the purpose level. We again use 'hardened' derivations denoted by apostrophes in the path (e.g., $\texttt{m/purpose}'/\texttt{coin\_type}'$). See Figure 18.3 for a depiction of the tree structure.
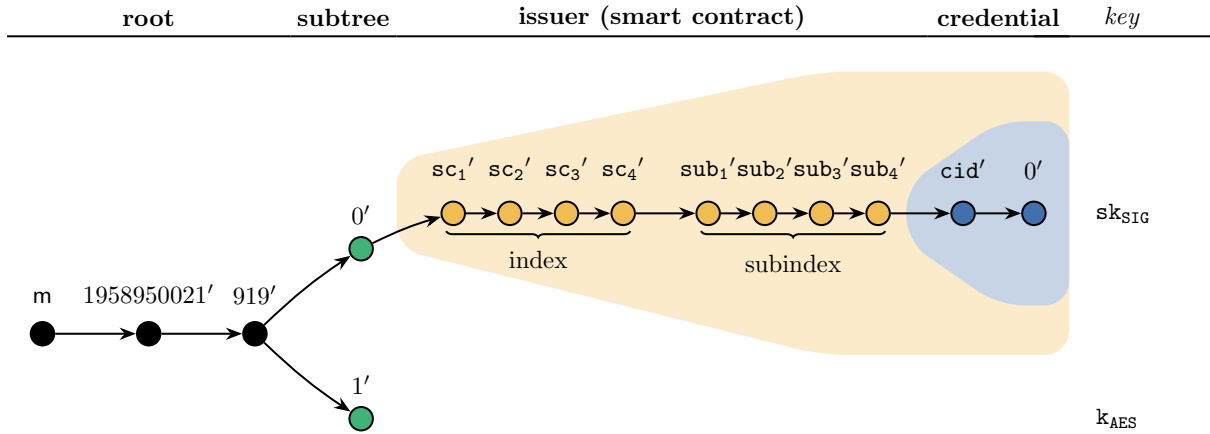
Figure 18.3 Key derivation tree for Web3 verifiable credentials.

Root. The root for our Web3 verifiable credentials' subtree is

$$\texttt{id3root} \coloneqq \texttt{m}/\texttt{purpose}'/\texttt{coin\_type}'$$

where

- the value $\texttt{m}$ is a placeholder for the master secret key,

- the integer $\texttt{purpose} \coloneqq 1958950021$ defines the purpose as Web3 ID to separate it from the account and identity subtree,

- the integer $\texttt{coin\_type} \coloneqq 919$ defines the derivation subtree for the Concordium mainnet.

We use $\texttt{coin\_type} = 1$ for testnet.

> **!** The number 1958950021 is $\texttt{web3id}$ in base36.

Verifiable credential subtree. Credentials signature keys are generated in subtree $\texttt{id3root}/0'/*$. For an issuer with a registry smart contract addressed by index $\texttt{sc}$ and subindex $\texttt{sub}$, we split the indices into four parts, where $\texttt{sc}_i$ corresponds to the ith 16 bits of $\texttt{sc}$ (in big-endian encoding), and analogously for $\texttt{sub}$. For a verifiable credential with index $\texttt{cid}$ (starting at 0) from an issuer with smart contract index $\texttt{sc}$ and sub-index $\texttt{sub}$ the signature key $\texttt{sk}_{\texttt{SIG}}$ of type $\mathrm{SIGNKEYS_{EdDSA}}$ is found at

$$\texttt{id3root}/0'/\texttt{sc}_1'/\texttt{sc}_2'/\texttt{sc}_3'/\texttt{sc}_4'/\texttt{sub}_1'/\texttt{sub}_2'/\texttt{sub}_3'/\texttt{sub}_4'/\texttt{cid}'/0'.$$

Backup encryption key subtree. The backup of the user's credentials are encrypted using an AES-256 key. That key $\texttt{k}_{\texttt{AES}}$ of type $\{0,1\}^{256}$ is derived from the seed phrase at position

$$\texttt{id3root}/1'.$$

> **!** We reserve subtrees $\texttt{id3root}/\texttt{i}'/*$ for $i > 1$ for future use.

18.5 Account and Identity Recovery from Seed Phrase

This Section assumes that keys have been generated according to the tree from Section 18.2.

18.5.1 Account Recovery Process

An account holder can recover the keys for all account credentials that have been deployed to the chain just from the seed phrase. The following algorithm allows an account holder to detect all deployed account credentials on chain, including all necessary indices to generate the keys to use them. The actual keys can be derived from the indices as described in Section 18.2.1.

Protocol AccountRecovery

Inputs

  masterkey $\in \mathrm{SIGNKEYS}_{\mathrm{EdDSA}}$
      Master secret key derived from seed phrase.

  idp_list $\in \mathbb{N}^*$
      List of all identity provider identifiers

Outputs

  $\mathfrak{A} \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N})^*$
      List of (identity provider identifier, identity index, credential index) triples for which account credentials were found.

Implementation

1: $\mathfrak{A} := \varnothing$, id_gap, cred_gap $:= 0$, identity $:= 0$
2: **for all** idp $\in$ idp_list **do**
3:     **repeat**
4:        $\mathrm{key}_{\mathrm{PRF}} := \mathsf{deriveKey}(\mathrm{masterkey}, \mathrm{root}/\mathrm{idp}'/\mathrm{identity}'/3', \mathbb{F}_{\mathrm{BLS}})$
5:        cred_found $:=$ false, cred_index $:= 0$
6:        **repeat**
7:           $\mathrm{ID}_{\mathrm{AC}} := \mathsf{PRF}(\mathrm{IDC.key}_{\mathrm{PRF}}, \mathrm{cred\_index})$
8:           **if** credential with ID $\mathrm{ID}_{\mathrm{AC}}$ exists on chain **then**
9:              append (idp, identity, cred_index) to $\mathfrak{A}$
10:            cred_found $:=$ true, cred_gap $:= 0$
11:           **else**
12:            cred_gap $:=$ cred_gap $+ 1$
13:           cred_index $:=$ cred_index $+ 1$
14:     **until** cred_gap ≥ 20
15:     **if** cred_found **then** id_gap $:= 0$
16:     **else** id_gap $:=$ id_gap $+ 1$
17:     identity $:=$ identity $+ 1$
18:   **until** id_gap ≥ 20
19: **return** $\mathfrak{A}$

> ⚠ The gap 20 is defined in *BIP-44* for accounts (corresponding to our identities in the derivation tree). We use the same gap both for accounts and identities.

18.5.2 Identity Recovery Process

The following protocol allows an account holder $\mathtt{AH}$ to recover the identity attributes $\mathtt{IDC.attrList}$, $\mathtt{IDC.max_{AC}}$, and the signature $\mathtt{IDC.}\sigma_{\mathtt{IDP}}$ from the identity provider. We assume that the IDP has an identity record $\mathtt{AHIR}$ which contains $\mathtt{IDcred_{PUB}, max_{AC}, attrList}$, and $\sigma'_{\mathtt{IDP}}$. We assume that the $\mathtt{AH}$ knows the identity provider index $\mathtt{ID_{IDP}}$, signature blinding randomness $\mathtt{m_0}$, and the secret identity credential $\mathtt{IDcred_{SEC}}$ (and thus $\mathtt{IDcred_{PUB}}$).

The protocol can be executed during the account recovery described above. It allows recovering identities for which an account credential has been found, as well as all other identities the user has created.

Protocol IdentityRecovery

Account Holder$(\mathtt{ID_{IDP}}, \mathtt{vk_{IDP}^{ID}}, \mathtt{m_0},$ $\mathtt{IDcred_{SEC}}, \mathtt{IDcred_{PUB}})$

Identity Provider$(\mathtt{ID_{IDP}}, \mathtt{vk_{IDP}^{ID}},$ $\mathtt{attrList}, \sigma'_{\mathtt{IDP}})$

Let t be the current Unix timestamp (in seconds) and $\mathtt{ctx} :=$ "recoverID" $\|\ t\ \|\ \mathtt{ID_{IDP}}\ \|\ \mathtt{vk_{IDP}^{ID}}\ \|\ \mathsf{H}(\text{genesis string})$. Compute NIZK proof π proving knowledge of $\mathtt{IDcred_{SEC}}$ for given $\mathtt{IDcred_{PUB}}$ using the context-string $\mathtt{ctx}$.

$\xrightarrow{\quad \mathtt{IDcred_{PUB}}, t, \pi \quad}$

If there is no identity record for $\mathtt{IDcred_{PUB}}$ or π does not validate or t is older or newer than the current time by more than Δ, abort. Otherwise, send $\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}$, $\mathtt{threshold_{PG}}, \mathtt{attrList}, \mathtt{max_{AC}}$, $\sigma'_{\mathtt{IDP}}$.

$\xleftarrow{\quad \mathtt{attrList}, \sigma'_{\mathtt{IDP}} = (a,b) \quad}$

Compute $\sigma_{\mathtt{IDP}} := (a, ba^{-\mathtt{m_0}})$. **Return** $(\mathtt{attrList}, \sigma_{\mathtt{IDP}})$.

</> In the current implementation, Δ is set to one minute.

> ⓘ We assume that the account holder and identity provider communicate over authenticated and secure channels such that the account holder can be sure to talk to the actual identity provider $\mathtt{ID_{IDP}}$.

> ⓘ In the above protocol the identity provider also sends the identity disclosure set $\mathtt{ID_{PG_1}}, \ldots, \mathtt{ID_{PG_n}}$ and threshold $\mathtt{threshold_{PG}}$. This ensures that an account holder can fully recover the identity credential even if they have not deployed an account credential on-chain.

> ⓘ We use a timestamp in the context string to prevent replay attacks. That is, an attacker who obtains an old proof π cannot use it to obtain data from the identity provider. The ID of the identity provider $\mathtt{ID_{IDP}}$ is part of the context string to bind the proof to this IDP and in particular prevent a malicious IDP to reuse the proof to obtain data from other IDPs. We further add $\mathtt{vk_{IDP}^{ID}}$ to the context string as an additional measure in case $\mathtt{ID_{IDP}}$ gets reused.

> ⓘ All information of the account holder can be reconstructed from the seed phrase. For $\mathtt{m_0}$ and $\mathtt{IDcred_{SEC}}$ reconstruction is direct via key derivation, and $\mathtt{IDcred_{PUB}}$ is derived from $\mathtt{IDcred_{SEC}}$.

The non-interactive zero-knowledge proof used in IdentityRecovery is a simple dlog sigma protocol (see Section 9.2.1).

Implicit public values consist of the group $\mathbb{G}_1^{\mathtt{BLS}}$ with order q and generator g.

Public input consists of the public identity credential $\mathtt{IDcred_{PUB}}$.

Secret input consists of the secret identity credential $\mathtt{IDcred_{SEC}}$.

The proven **statement** is

$$\mathtt{PK}\left\{(\mathtt{IDcred_{SEC}}) : g^{\mathtt{IDcred_{SEC}}} = \mathtt{IDcred_{PUB}}\right\}$$

> ⚠ The context string *must* contain the timestamp t to prevent replay attacks. The context string *must* contain $\mathtt{ID_{IDP}}$ or $\mathtt{vk_{IDP}^{ID}}$ to prevent man-in-the-middle attacks. The context string *should* contain $\mathsf{H}(\text{genesis string})$ to tie the proof to a specific network, e.g., mainnet.
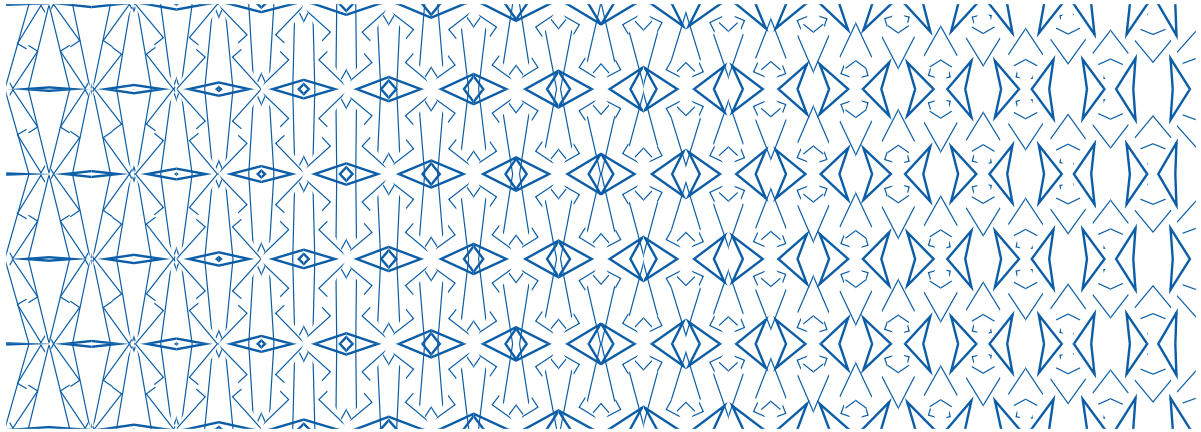
Part IV

Renovatio

Illustration: Morphing p2 Tessellation.

Chapter 19

Update Procedure

In this chapter, we describe how the decision for an *update*, such as a parameter or protocol update, is recorded on the chain and how the update is performed. The decision-making process is not within the scope of the bluepaper.

Since updates are very critical, we require an update instruction to be signed by more than one key. We use access structures to describe the necessary sets of keys to authorize updates. This allows for a fine-grained balance of security and usability. In the implementation, a (k, n)-threshold access structure is used where k out of n different signature keys must authorize updates.

There are four *categories* of updates. An *authorization update* is used for key management and allows changes to the access structure and signature keys. A *parameter update* allows updating blockchain parameters, one parameter for example is the minimal stake necessary to participate as a validator. A *protocol update* allows upgrading the overall blockchain system. For example, one can switch to a new consensus protocol. To ensure great flexibility when updating the protocol, the update procedure essentially starts a new chain.

All the above update categories require that the blockchain is producing finalized blocks. As a fallback, there are also *emergency updates* that can be used even if the chain stopped working and no blocks are produced. This update is a measure of last resort and will not be used in normal operation. We describe emergency updates separately in Section 20.

19.1 Data Structures

19.1.1 Data Types

For fine-grained access control, we provide multiple types of parameter and authorization updates. For example, one type of parameter update allows to add a new identity provider.

Definition 109. An *update type* $\mathtt{uT} \in \textsc{updateTypes} \coloneqq \mathbb{N}$ is an enumerated type indicating the nature of the update. Each update type is assigned to one of the four categories `auth`, `param`, `prot`, or `emerg`.

> ℹ️ Update types can be added or removed using a protocol update (cf. Section 19.7).

> `</>` In the implementation, each category has separate update types. See Section 23.4 and 24.4.4 for more details.

Updates are sent to chain in an update instruction.

Definition 110. An *update instruction* `upInst` $\in$ UPDATEINSTRUCTIONS consists of the following values.

`seqNumber` $\in \mathbb{N}^+$
> The *update sequence number*. Updates must be added in order defined by the sequence of number.

`updateTime` $\in \mathbb{N}_0$
> The *effective time* of the update is the point in time when the update is supposed to become effective in seconds (since 1.1.1970). We reserve the special value 0 for updates that take effect immediately after they appear in a block on the chain, i.e., starting from the following block.

`updateTimeout` $\in \mathbb{N}_0$
> The *update timeout* specifies the latest point in time the instruction can be added to a block in seconds (since 1.1.1970).

`uT` $\in$ UPDATETYPES
> The *update type*.

`payloadData` $\in \{0, 1\}^*$
> The *payload* data describing the update. The structure depends on the type of the update.

`updateSigs` $\in$ SIGNATURES$_{\text{UPD}}$*
> The *update signatures* is a list of signatures on the above entries of the update instruction.

> ❗ Instantaneous updates, i.e., updates with `updateTime` $= 0$, should be used carefully as they can take effect before the block containing the update transaction is finalized. Thus, `updateTime` $= 0$ should be avoided for consensus related updates. On the other hand, it may be acceptable to use such updates for values that are changed frequently, e.g., an exchange rate such as the value of the native token in EUR.

> `</>` In the implementation, the update type is encoded directly into the update payload. See Section 24.4.4 for an overview of update types and the serialization of their corresponding payloads.

19.2 Update Authorization

For each update type an *access structure* defines which sub-sets of keys from a global set `upVKs` can authorize such updates. More formally, we have:

`upVKs` $\in$ VERIFICATIONKEYS$_{\text{UPD}}$*
> The set of all *verification keys* eligible to sign update instructions.

$\mathfrak{A}_{\text{uT}} \in$ VERIFICATIONKEYS$_{\text{UPD}}$**
> *Access structure* over `upVKs` specifying the authorized sets of keys for updates of type `uT`.

The initial access structures for each update type and set of keys `upVKs` must be defined in the genesis block (cf. Section 21) or in the protocol update that introduces the update type. They can be changed using authorization updates (cf. Section 19.5).

`genesisData.upVKs` $\in$ VERIFICATIONKEYS_{UPD}*

The initial set of all verification keys eligible to sign update instructions.

`genesisData.`$\mathfrak{A}_{\mathtt{uT}}$ $\in$ VERIFICATIONKEYS_{UPD}**

Initial Access structure over `upVKs` specifying the authorized sets of keys for updates of type `uT`.

> **</>** In the current implementation, we use threshold access structures. Each such access structure is defined by a subset of `upVKs` and the minimal number k of keys necessary to authorize an update.

19.3 Validity of Update Instructions

An update instruction is considered *valid* at time `timestamp` with respect to a chain if its timeout is in the future, the timeout is before the effective time (or the effective time is 0), it contains valid signatures, and its payload is valid. More formally, we define following validity function.

Function verifyUpdateInstruction

Inputs

`upInst` $\in$ UPDATEINSTRUCTIONS $\setminus \{\mathtt{emerg}\}$
   The update instruction to verify.

`chain` $\in$ CHAINS
   Chain relative to which verification is done. Should contain all blocks up to the parent of the block containing `upInst`.

`timestamp` $\in \mathbb{N}_0$
   The time for which validity is checked.

Outputs

$b \in$ BOOL
   Indicates whether the update instruction is valid.

Implementation

1: /* Reject if the timeout is in the past   */
2: **if** `timestamp` $>$ `upInst.updateTimeout` **then**
3:     **return** false
4: /* Reject if the timeout after the effective time (ignore for instant updates)   */
5: **if** `upInst.updateTimeout` $\geq$ `upInst.updateTime` $\wedge$ `upInst.updateTime` $\neq 0$ **then**
6:     **return** false
7: /* Reject if the sequence number was already used or is too large   */
8: **if** `upInst.seqNumber` $\neq$ getCurrentSequenceNo(`upInst.uT`, `chain`, `timestamp`) **then return** false
9: /* Signature check   */
10: **if** `upInst.updateSigs` contains an invalid signature **then return** false
11: Let `upVKs`$'$ be the set of verification keys for the signatures in `upInst.updateSigs`.
12: $\mathfrak{A}_{\mathtt{uT}}$ = getCurrentAuthorization(`upInst.uT`, `chain`, `timestamp`).
13: **if** `upVKs`$' \notin \mathfrak{A}_{\mathtt{uT}}$ **then return** false
14: /* Verify the payload   */
15: **return** verifyUpdatePayload(`upInst.uT`, `upInst.payloadData`, `chain`, `timestamp`)

The function getCurrentSequenceNo returns the next sequence number given the list of update instruction in a chain. It is defined in Section 19.4. The function getCurrentAuthorization returns the updated verification keys and access structure for the given update type, and is defined in Section 19.5.2. The function verifyUpdatePayload checks that the update payload is valid at this moment on chain. For example, it checks that updated parameters are within a valid range (as defined by the protocol). More details are given in the following sections.

> ❗ The input `timestamp` corresponds to the time of the block that contains the update instruction. We assume that the consensus layer provides a function that maps blocks to (their creation) time.

> ℹ️ For protocol versions using the initial consensus protocol (cf. Part VII), the functions getCurrentSequenceNo, getCurrentAuthorization, and verifyUpdatePayload took the slot of the update block instead of `timestamp` as input.

19.3.1 Publication of Updates

Update instructions are published the same way as normal transactions by broadcasting them. The difference is that updates do not require an account or tokens to publish them. Validators will add broadcast update instructions to blocks if the instruction is valid.

19.4 Update Queue

On-chain, valid update instructions of the same type are ordered according to sequence numbers. From this list, we can extract the *update queue* which is the list of updates which took effect or will take effect. What makes the update queue special is the fact that we allow an update instruction to overrule previous updates of the same type. An update instruction is overruled if another update instruction of the same type with a larger sequence number and smaller or equal update time is added. The update queue can be computed as follows.

Function getUpdateQueue

Inputs

$uT \in \textsc{updateTypes} \setminus \{\texttt{emerg}\}$
    The *type* of the update for which the queue is returned.

$\texttt{chain} \in \textsc{chains}$
    A valid chain from the which the queue is built.

Outputs

$\texttt{queue} \in \textsc{updateInstructions}^*$
    A *queue of update* instructions.

Promise

    The function provides consistency in the sense that adding blocks to the chain can only change update instructions in the queue with an update time after the time of the last block in the chain. More formally: Let `chain` be a chain, let $\texttt{chain}'$ be a prefix of `chain`, and let $\texttt{timestamp}'$ be the creation time of the last block in $\texttt{chain}'$. Further, let for some uT, $\texttt{queue} := \texttt{getUpdateQueue}(uT, \texttt{chain})$ and $\texttt{queue}' := \texttt{getUpdateQueue}(uT, \texttt{chain}')$,

---

and let $\overline{\texttt{queue}}$, $\overline{\texttt{queue}}'$ be the queues with all update instructions that have `updateTime` larger than `timestamp`$'$ removed. Then, $\overline{\texttt{queue}} = \overline{\texttt{queue}}'$.

Implementation

1: Let `queue` be empty.
2: **for all** update instructions `upInst` in `chain` of type `uT` ordered by sequence number **do**
3:     Denote by `timestamp` the time of the block that contains `upInst`.
4:     /* *Instant updates are effective after timestamp* */
5:     **if** `upInst.updateTime` $= 0$ **then** Set `upInst.updateTime` to *timestamp*.
6:     /* *Remove updates that are overruled by* `upInst` */
7:     Remove all `upInst`$'$ from `queue` where `upInst`$'$`.updateTime` $\geq$ `upInst.updateTime`.
8:     Add `upInst` to end of `queue`.
9: **return** `queue`

Next, we define the getCurrentSequenceNo function which returns sequence number for the next valid update instruction of a given type.

Function getCurrentSequenceNo

Inputs

   `uT` $\in$ UPDATETYPES $\setminus \{\texttt{emerg}\}$
        An update instruction type.
   `chain` $\in$ CHAINS
        A valid chain.
   `timestamp` $\in \mathbb{N}_0$
        A timestamp.

Outputs

   `seqNumber` $\in \mathbb{N}^+$
        The *current sequence* number for update type `uT`.

Implementation

1: Let `chain`$'$ be the prefix of `chain` with all blocks that have been created before `timestamp`.
2: /* getUpdateQueue *is just used to filter according to update type.* */
3: `queue` := getUpdateQueue(`uT`, `chain`$'$)
4: **if** `queue` is empty **then return** 1.
5: /* *The last element in the queue has the maximum sequence number* */
6: Let `upInst` = last(`queue`).
7: **return** `upInst.seqNumber` $+ 1$

19.5 Authorization Updates

The payload `payloadData` of an *authorization update* essentially consists of the new set `upVKs`$'$ of verification keys and a list of updated access structures. The type of the authorization update define which keys and access structures the update may change.

> In the implementation, the set of verification keys is split into three parts. There are root keys that can update all keys and access structures. Level-1 keys can change all keys and access structures except the root keys and their access structure. Finally, level-2 keys are used to sign parameter and protocol updates.

> ⚠ Updating the authorization structure *must* be done carefully. There is the risk of locking oneself out.

19.5.1 Validity of Authorization Updates

The payload verification function verifyUpdatePayload checks that

- the updated authorization structure is sound,
- the update to the authorization structure fits the update type.

The second condition ensures for example that level-1 keys cannot update root keys.

> See Section 24.4.4 for more details on the serialization of authorization updates.

19.5.2 Applying Authorization Updates

Authorization updates take effect in the first block corresponding to a time at or after `updateTime` if the update instruction is in a block that was created before `updateTime`. The access structures and the key set in the update transaction replace the corresponding old ones.

19.5.3 Query Current Authorization

The function getCurrentAuthorization allows one to retrieve the verification keys and access structures for a given update type and a given chain.

Function getCurrentAuthorization

Inputs

$\text{uT} \in \text{UPDATETYPES}$
    The type of the update for which the authorization data is retrieved.

$\text{chain} \in \text{CHAINS}$
    The chain relative to which the data is retrieved, i.e., the returned authorization holds for blocks after `chain`.

$\text{timestamp} \in \mathbb{N}_0$
    The time for which the data is retrieved.

Outputs

$\text{upVKs} \in \text{VERIFICATIONKEYS}_{\text{UPD}}^*$
    The set of eligible verification keys for the given chain and time.

$\mathfrak{A}_{\text{uT}} \in \text{VERIFICATIONKEYS}_{\text{UPD}}^{**}$
    The access structure over `upVKs` for the given `uT`, `chain`, and `timestamp`.

Promise

> If `chain` contains blocks with at time `timestamp` or more, the output remains the same for all extensions `chain'` of `chain`.

We explicitly allow querying getCurrentAuthorization in the future, i.e., for timestamps that are larger than the one of the input chain's last block. This is needed for example to check the validity of an update instruction that is added to a fresh block.

> ❗ If getCurrentAuthorization is queried with time `timestamp` larger than the time `timestamp'` of the last block in `chain`, the output value is the correct value for any chain that extends `chain` and has no block between `timestamp` and `timestamp'`. This directly follows from the promise.

19.6   Parameter Updates

Each type of *parameter update* allow changing specific chain parameter. That is, the payload of each parameter update is a list where each entry consists of a parameter identifier and the new value.

> ❗ The admissible range of a parameter is defined by the current protocol.

The initial parameter values are found in the genesis data (cf. Section 21).

> ℹ️ Parameters will be added and removed as part of protocol updates. For example, a new the consensus protocol may introduce new parameters and deprecate existing ones.

> </> See Section 24.4.4 for more details on the current set of parameter updates and their serialization.

19.6.1   Validity of Parameter Updates

The payload verification function verifyUpdatePayload for parameter updates checks that the update contains all necessary parameters and that all values are within the admissible range.

> ⚠️ The function verifyUpdatePayload does not ensure that the new parameter values are safe to use. Parameter updates can break the chain.

> ❗ The admissible range of each parameter should prevent dangerous parameter choices (e.g., a minimum delay between blocks of 2 million years). This does not necessarily prevent negative impact on the chain by a combination of parameters.

19.6.2   Applying Parameter Updates

As authorization updates, parameter updates take effect in the first block corresponding to a time at or after `updateTime` if the update instruction is in a block that was created before `updateTime`. When in effect, the parameter update defines the new values of its parameters.

19.6.3 Query Current Parameter

The function getCurrentParameter allows one to retrieve the value of a given parameter for a given block. When creating a new block, one must query the parameters of the parent block to know what values to use for the new one.

> **❶** If a parameter update is effective at time t, then the state of the chain will be updated in the next block after this time. If the parameter affects block creation, e.g., maxBlockEnergy or minBlockTime, then the first block affected by the new parameter will be the second block after time t, since the first block after time t gets its parameters from the state in the parent block, which is before t. The same applies for "instant" updates that don't have a time, but are updated in the following block, therefore affecting the block after that.

19.7 Protocol Updates

In this section, we describe how *protocol updates* take effect and how the update is applied to the blockchain.

19.7.1 Update Description and Update Validity

We assume that parties can extract the full description of the protocol update from the update instruction. In practice, this could be achieved by including the hash of the update specification in the payload. This could be the hash of a PDF describing the update or the hash of the source code for a new client. We assume the required full specification is accessible publicly and its location is known. To ensure this, the payload could also include a URL pointing to the full update specification.

> **⚠** The mechanism in this section *DOES NOT* ensure that the update made is reasonable. We assume that the update has been properly tested beforehand. We therefore let the verifyUpdatePayload function return true for any protocol update.

> **</>** In practice, a protocol update comes with an update to the software components used to run the blockchain. The software contains the necessary logic to run the new protocol. The update transaction contains a URL to the update specifications given as a human-readable text file. See Section 24.4.4 for more details on the serialization of the update instruction.

Function verifyUpdatePayload(upInst.uT, upInst.payloadData)

Inputs

uT $\in$ UPDATETYPES
      The type of the update. Only described here for uT = prot.

payloadData $\in \{0, 1\}^*$
      The payload data to verify.

Outputs

19.7.2 Update Procedure

A protocol update instruction that has been added to the chain takes effect as soon as its effective time passed. The protocol update is applied as follows. First, the old protocol keeps running, i.e., blocks are produced and finalized. As soon as the first block that was created strictly after the effective time is finalized, the old protocol is stopped. The state of the chain up to that last finalized block is used to compute the genesis state of the updated protocol. Once the genesis state is computed, the new protocol starts.

Protocol UpdateProcedure

1. When the update instruction is added to the chain, parties start to prepare for the update. They make sure that they have the relevant tools and software to perform the update and run the updated protocol.
2. The old protocol continues normally, until the first block that was created strictly after the effective time is finalized.
3. At that point, the old protocol stops and the parties compute the new genesis state from the state defined by that finalized block.
4. Once they have computed the genesis state, they start the updated protocol.

❗ The update time has to be set far enough into the future such that all (honest) parties will be ready to run the updated protocol at that point in time. In practice, the value of "far enough" will depend on how the update is announced off-chain and how parties update their software.

Notes on the new genesis state. In Step 3 of the update procedure, the parties compute the *new genesis state*. The exact nature of this object depends on the new protocol. One can think of it as the information (e.g., user accounts and smart contracts) one wants to transfer from the old system. But the new genesis state can also contain new information, such as the initial values of added parameters.

Transactions that appeared in blocks after the last finalized block on the old chain need to be re-added in the new system.

If the new protocol is still a blockchain, then the new genesis block could simply be a block that contains the hash of the last finalized block and for example an added parameter.

It is recommended (but not required) that the computation of the new genesis state can be done locally and deterministically. We assume that instructions on how to compute the new genesis state are part of the protocol update description.

A note on the design choice. An advantage of such a radical update strategy where the old chain stops is the great flexibility it provides. In particular, we do not need to make assumptions on the new protocol. This allows us to radically change the system if need be.

Another advantage of this design is that the new chain can be designed and run in isolation. It is easier to run a new chain in the testnet until it is believed to be stable than it is to repeatedly testing some kind of dynamic update procedure (which depends on the update) on a running chain.

> The flexibility of the system was proven when we switch from the old consensus protocol (Nakamoto-style) to the new consensus (BFT-style).

Chapter 20

Emergency Updates

In this section we describe an update mechanism that will work even if the platform has suffered a catastrophic failure. For this we cannot make any assumption on the state of the platform. In particular, we cannot assume that finalization works or that blocks are produced. We only assume that there is some emergency channel to send messages to the involved parties, e.g. the validators.

> ℹ Realistically, the assumed emergency channel is the Concordium webpage or the Concordium twitter account. However, it could in principle also be a newspaper or messenger pigeons.

The protocol for an emergency update is simple

Protocol EmergencyUpdate

1. The Concordium foundation signs the emergency update instruction.
2. The emergency update instruction is broadcast on the emergency channel.
3. A node that receives the emergency update instruction extracts the current emergency authorization information from its best chain. It then checks that the update instruction was signed properly.
4. If the instruction is valid, the node immediately follows the instructions.

> ❗ The emergency update instruction can in principle contain arbitrary instructions. We assume that the entity behind the node will (manually) follow these instructions. The instructions could for example consist of a link (URL) to a webpage where further instructions wait.

> ⚠ This type of instruction CAN seriously damage the system as it allows to RESET the whole system. The singing keys for this type of update instruction have to be stored and handled with utmost care.
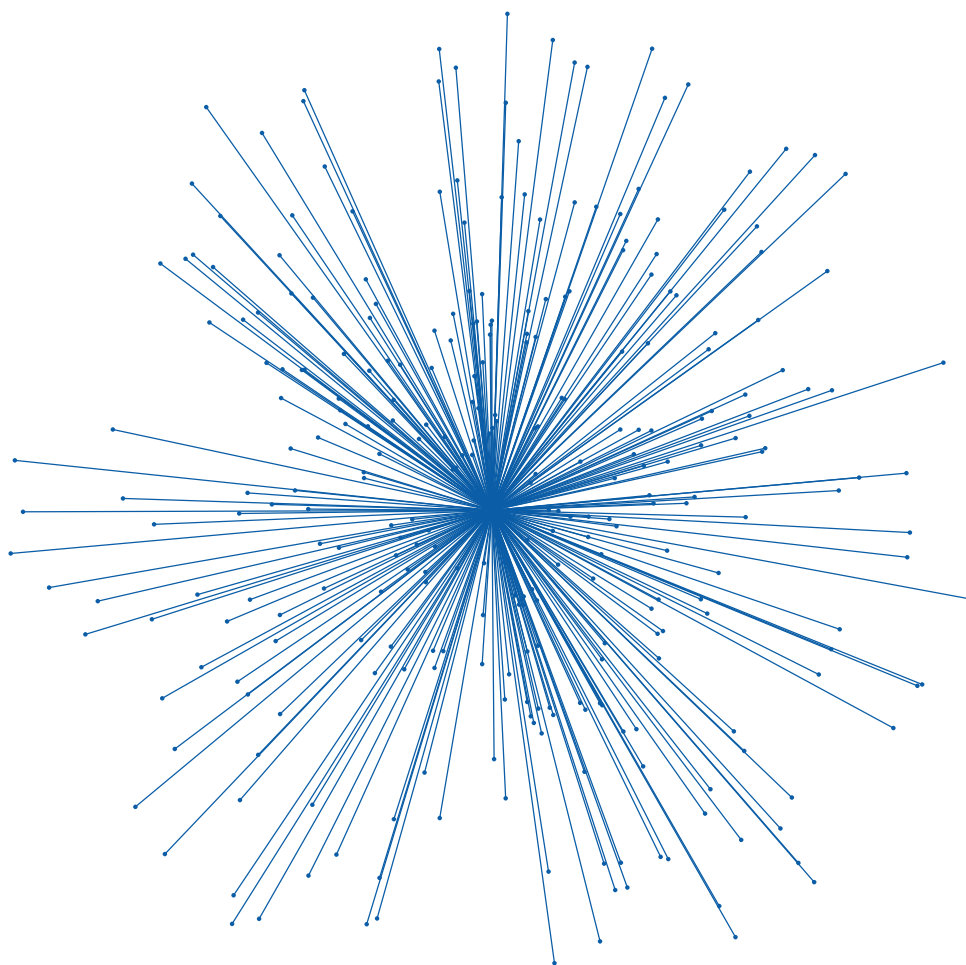
Part V

Urknall

Chapter 21

Genesis Data

This chapter provides an overview on the parameter, values, and data that need to be defined in order to start the blockchain system.

21.1 Cryptographic Primitives

$\mathtt{ck}_0, \dots, \mathtt{ck}_{256} \in$ COMKEYS

The *commitment keys* for Pedersen commitments, need to be generated safely (cf. Section 22.1.1).

21.2 Konsensus Layer

> ℹ️ In the implementation, we started using the initial consensus which uses different parameter than Concordium BFT. In this chapter, we will for completeness provide both parameter sets.

21.2.1 Parameters

$\mathtt{GenesisTime} \in \mathbb{N}$

The UNIX timestamp of the protocol start.

> </> In the implementation this was set to 2021-06-09T08:00:00+02:00.

ConcordiumBFT. These parameters are specific to Concordium BFT. More details are found in Section 12.

$\mathtt{sigThreshold} \in \mathbb{Q}$

Weight of signatures needed for a QC or a TC.

> </> The implementation uses $\frac{2}{3}$.

$\mathtt{timeoutBase} \in \mathbb{N}$

The base value for triggering a timeout in milliseconds.

> The implementation uses 10000 milliseconds.

`timeoutIncrease` $\in \mathbb{Q}$
> Factor for increasing the timeout. Applied locally when a timeout occurs.

> The implementation uses $\frac{6}{5}$.

`timeoutDecrease` $\in \mathbb{Q}$
> Factor for decreasing the timeout. Applied when a block is finalized.

> The implementation uses $\frac{3}{2}$.

$\mathtt{nonce}^{\mathrm{LE}} \in \mathrm{NONCE}_{\mathrm{LE}}$
> The initial leader election nonce. Has to be unpredictable until right before the protocol start. It should also be a nothing-up-my-sleeve value. More details are found in Section 22.2.1.

Initial consensus protocol. These parameters are specific to the initial consensus protocol. More details are found in Section E.

`SlotDuration` $\in \mathbb{N}$
> The duration of a slot in seconds.

$\mathtt{nonce}^{\mathrm{LE}} \in \mathrm{NONCE}_{\mathrm{LE}}$
> The initial leader election nonce. Has to be generated safely (cf. Section 22.2.1).

$\mathtt{diff}^{\mathrm{LE}} \in \mathrm{DIFFICULTY}$
> The initial leader election difficulty.

`epochLength` $\in \mathbb{N}$
> The length of epochs in number of slots.

`minSkip` $\in \mathbb{N}$
> The initial minimum skip for finalization.

`skipShrinkFactor` $\in \mathbb{Q}$
> The factor used to decrease the skip for the finalization depth.

`skipGrowFactor` $\in \mathbb{Q}$
> The factor used to increase the skip for the finalization depth.

`bDelShrinkFactor` $\in \mathbb{Q}$
> The factor used to decrease the block delay for finalization.

`bDelGrowFactor` $\in \mathbb{Q}$
> The factor used to increase the block delay for finalization.

`ABBADelayInc` $\in \mathbb{Q}$
> The time in seconds by which the waiting delay in ABBA is increased after each failed phase.

21.2.2 Initial Participants

> **!** In the implementation each of the participants needs an associated initial account.

ConcordiumBFT.

validators $\in$ VALIDATORS*
> The initial set of eligible validators.

finCom $\in$ VALIDATORS*
> The initial set of eligible finalizers.

> **</>** In the current consensus protocol (ConcordiumBFT), finalizers are a subset of validators (those with enough stake) determined by tokenomics parameters. But since the parameters are updatable, one cannot determine from the current state alone which validators are finalizers, since the current value of the parameters might not be those used to determine the set of finalizers at the beginning of the payday. For this reason, the set of finalizers is recorded separately. In the current implementation, any validator pool which has at least 0.005% of the total stake is part of the finalization committee—as long as there are not more than 1000 such pools—which is roughly equivalent to the stake needed to open a new validator, currently set at CCD $500'000$.

Initial consensus.

bakers $\in$ BAKERS*
> The initial set of eligible bakers.

finCom $\in$ FINALIZERS*
> The initial set of eligible finalizers.

> **</>** In the initial consensus design, the set of finalizers could be disjoint from the bakers, so it was also recorded separately. Though in the implentation, it was always a subset determined by stake.

21.3 Personligt and Konto Layer

The initial entities and accounts of the Personligt and Konto layer. All validators defined in Section 21.2.2 must have an account.

$\text{IDP}_1, \ldots, \text{IDP}_k \in \text{IDs}_{\text{IDP}}$*
> The list of initial identity provider.

$\text{PG}_1, \ldots, \text{PG}_k \in \text{IDs}_{\text{PG}}$*
> The list of initial privacy guardians.

$\text{ACC}_1, \ldots, \text{ACC}_n \in$ ACCOUNTS
> The list of *genesis accounts*.

> </> In the implementation, each of these entries has metadata. For example, the genesis block defines the initial balance of an initial account.

21.4 Governance

21.4.1 Chain parameters

Not all of the following parameters have been present since the launch of the blockchain. They have been added as needed throughout its development.

`maxBlockEnergy` $\in \mathbb{N}$
>   Maximum energy for all the transactions in a block.

> </> The implementation uses 3000000 NRG.

`minBlockTime` $\in \mathbb{N}$
>   Minimum time between blocks in milliseconds.

> </> The implementation uses 2000 milliseconds.

`epochDuration` $\in \mathbb{N}$
>   Length of an epoch in seconds.

> </> The implementation uses 3600 seconds.

`rewardPeriodLength` $\in \mathbb{N}$
>   Number of epochs in a pay day.

> </> The implementation uses 24 epochs.

> 🛈 Additionally, the chain has a lot of tokenomics parameters that govern how CCD is minted, how fees and rewards are distributed, the bounds on pools sizes, the criteria to be a validator or finalizer, etc. These are out of the scope of the bluepaper. An overview of all public tokenomics criteria can be found in [Con25].

21.4.2 Update Authorization

The keys and access structures for Renovatio.

`upVKs` $\in \textsc{verificationkeys}_{\textsf{UPD}}{}^{*}$
>   The set of initial verification keys eligible to sign update instructions.

$\mathfrak{A} \in \textsc{verificationkeys}_{\textsf{UPD}}{}^{***}$
>   The set of access structures over `upVKs` specifying the authorized sets of keys for updates of different types.

Chapter 22

Genesis Ceremony

This chapter describes how to securely generate the genesis data.

> ⚠ The parameter described in Section 22.2 **MUST** be generated in a trustworthy manner.

22.1 Setup for Cryptographic Primitives

See Section 2.2 and Part I for more details on the choice of groups and fields.

> ❗ At the moment our zero-knowledge protocols do not require any common reference string (CRS) or structured reference string (SRS). If a zkSNARK protocol is later added to the protocol stack, the CRS or SRS setup can be generated even after the launch of the chain.

22.1.1 Secure Generation of Commitment Keys

In the Personligt and Konto layer Pedersen commitments are in particular used (as auxiliary information) in zero-knowledge proofs (see e.g., the account opening procedure in Section 15.3.3). It is important that the used global commitment key ck was generated a 'nothing-up-my-sleeve' way (see [Wik20]). A Pedersen commitment key consists of two generators of $\mathbb{G}_1^{\mathsf{BLS}}$ and is of the form $\mathsf{ck} = (\bar{g}_{\mathsf{com}}, \bar{h}_{\mathsf{com}})$. Here the discrete logarithm of $\bar{h}_{\mathsf{com}}$ with respect to base $\bar{g}_{\mathsf{com}}$ must be kept secret, as it would allow a party to open a commitment to arbitrary values.

> ⚠ All commitment keys $\mathsf{ck} = (\bar{g}_{\mathsf{com}}, \bar{h}_{\mathsf{com}})$ **MUST** be generated such that no one knows the discrete logarithm of $\bar{h}_{\mathsf{com}}$ with respect to base $\bar{g}_{\mathsf{com}}$.

The following protocol generates 257 commitment keys. The first one $\mathsf{ck} := \mathsf{ck}_0$ is used as the main commitment key. The remaining ones $\mathsf{ck}_1, \ldots, \mathsf{ck}_{256}$ are used specifically for bulletproofs.

Let $\mathsf{H}_{\mathbb{G}_1^{\mathsf{BLS}}}^{\mathsf{BLS12\text{-}381}}$ be the hash function as defined in Section 3.2.1, which hashes bit strings to group elements in $\mathbb{G}_1^{\mathsf{BLS}}$.

> </> We assume in the following that $\mathsf{H}_{\mathbb{G}_1^{\mathsf{BLS}}}^{\mathsf{BLS12\text{-}381}}$ can hash group elements by encoding them first as bit strings. We do not make the serialization explicit here.

Protocol generateComKeys

Inputs

  $\pi \in \mathbb{R}$

      The circumference of a circle divided by its diameter.

Outputs

  $\mathtt{ck}_0, \ldots, \mathtt{ck}_{256} \in \text{COMKEYS}$

      Commitment keys.

Implementation

1: $\bar{g}_{\mathtt{com}_0} = \mathsf{H}_{\mathbb{G}_1^{\mathrm{BLS}}}^{\mathrm{BLS12\text{-}381}}(\text{first 999 digits of } \pi \text{ as ASCII characters, followed by } \mathtt{\backslash n})$

2: $\bar{h}_{\mathtt{com}_0} = \mathsf{H}_{\mathbb{G}_1^{\mathrm{BLS}}}^{\mathrm{BLS12\text{-}381}}(\bar{g}_{\mathtt{com}_0})$

3: $\mathtt{ck}[0] = (\bar{g}_{\mathtt{com}_0}, \bar{h}_{\mathtt{com}_0})$

4: **for** $i = 1, \ldots, 256$ **do**

5:     $\bar{g}_{\mathtt{com}_i} = \mathsf{H}_{\mathbb{G}_1^{\mathrm{BLS}}}^{\mathrm{BLS12\text{-}381}}(\bar{h}_{\mathtt{com}_{i-1}})$

6:     $\bar{h}_{\mathtt{com}_i} = \mathsf{H}_{\mathbb{G}_1^{\mathrm{BLS}}}^{\mathrm{BLS12\text{-}381}}(\bar{g}_{\mathtt{com}_{i-1}})$

7:     $\mathtt{ck}_i = (\bar{g}_{\mathtt{com}_i}, \bar{h}_{\mathtt{com}_i})$

8: **return** $\mathtt{ck}_0, \ldots, \mathtt{ck}_{256}$

> ⚠ The security relies on the assumption that $\mathsf{H}_{\mathbb{G}_1^{\mathrm{BLS}}}^{\mathrm{BLS12\text{-}381}}$ is a random oracle.

> ❗ For this construction, we need the group order to be prime. This guarantees that both $\bar{g}_{\mathtt{com}}$ and $\bar{h}_{\mathtt{com}}$ are generators, unless they are the neutral element. The latter happens only with negligible probability. It could additionally be checked that both elements are not the neutral element. If that check fails, we should use the first 1000 digits of π instead.

22.2 Trustworthy Generation of Critical Parameters

The following parameters must be generated in a trustworthy manner. They are prone to cryptographic backdoors or would give other advantages to an attacker if generated maliciously.

22.2.1 Initial Leader Election Nonce

The initial leader election nonce $\mathtt{nonce}^{\mathrm{LE}}$ influences the outcome of the block lottery in the first epoch (cf. Konsensus). An attacker could 'pre-mine' a nonce $\mathtt{nonce}^{\mathrm{LE}}$ such that a validator with a specific VRF key-pair gets a disproportionately high chance of winning the round-leader lottery in the first epoch. One could also mine VRF keys for a fixed nonce. Thus, it is not a solution to set the nonce to SHA256(first x digits of π).

> ⚠ The leader election nonce $\mathtt{nonce}^{\mathrm{LE}}$ **MUST NOT** be predictable until shortly before launch.

A simple solution is to set the nonce to the hash of the headline of some newspaper at the day of genesis. Alternatively, one could hash a Bitcoin block which is expected to be generated within

(a) NZZ headline.

(b) Berlingske headline.

Figure 22.1 Newspaper headlines used to generate the initial leader election nonce.

24h before launch.

Protocol generateLeaderElectionNonce

Implementation

1: Set $\mathtt{nonce}^{\mathrm{LE}} = \mathsf{H}_{\mathtt{SHA256}}(\text{Headline of NZZ}\,||\,\text{Headline of Berlingske})$.

Under the assumption that the headline is not predictable for more than 24 hours, an attacker would have less than 24 hours to generate a 'lucky' VRF-key pair. The validity of this nonce can be publicly verified.

For Concordium's mainnet the headlines

```
NZZ: Spionage-Vorwurf gegen Dänemark.  Zusammenarbeit mit US-Auslandgeheimdienst.\n
Berlingske: Regeringsjurister: Asylplan kan stride mod menneskerettigheder.
Selvom to danske ministre har rost Rwanda, der anses som en sandsynlig partner
i regeringens plan om at behandle asylansøgninger langt herfra, kan overførsel
af asylansøgere til det centralafrikanske land være i strid med Den Europæiske
Menneskerettighedskonvention.
```

were hashed with $\mathsf{H}_{\mathtt{SHA256}}$ to get the initial leader election nonce

```
60ab0feb036f5e3646f957085238f02fea83df5993db8e784e11500969af9420 .
```

22.3 Generation of Update Authorization Keys

The signing keys for update authorization are generated in a decentralized manner and stored on portable HSMs. Key custodians confirm that the right public-keys are added to the genesis block in a verification ceremony.

22.4 Generating Keys for Dummy Privacy Guardian and Identity Provider

The accounts in the genesis block are generated such that compatibility of normal accounts is ensured (see Section 22.5). To this end, we also need to add public keys for the identity providers and privacy guardians of these accounts. None of the keys will ever be used, and in particular, the corresponding secret keys are not required. To avoid the risk of the secret keys leaking, we generate the public keys without knowing the secret keys. This is done similarly to the generation of the commitment keys in Section 22.1.1.

22.4.1 Generating Keys for Dummy Privacy Guardian

The privacy guardian keys are ElGamal encryption keys in the group $\mathbb{G}_1^{\mathsf{BLS}}$. Hence, we only need to generate a single group element, which is done by hashing to that group.

Protocol generateDummyARKey

Inputs

$e \in \mathbb{R}$
    Euler constant.

Outputs

$\mathrm{pk}_{\mathsf{PG}} \in \mathrm{PUBLICKEYS}_{\mathsf{PG}}$
    Dummy privacy guardian public key.

Implementation

1: $\mathrm{pk}_{\mathsf{PG}} = \mathsf{H}_{\mathbb{G}_1^{\mathsf{BLS}}}^{\mathsf{BLS12\text{-}381}}(\text{first 999 digits of } e \text{ as ASCII characters, followed by } \backslash \mathtt{n})$
2: **return** $\mathrm{pk}_{\mathsf{PG}}$

</> The index of the dummy genesis privacy guardian is 1.

22.4.2 Generating Keys for Dummy Identity Provider

An identity provider verification key consists of several elements in the group $\mathbb{G}_2^{\mathsf{BLS}}$. We generate those via hashing to the group using hash function $\mathsf{H}_{\mathbb{G}_2^{\mathsf{BLS}}}^{\mathsf{BLS12\text{-}381}}$ as defined in Section 3.2.2.

Protocol generateDummyIDPKey

Inputs

$\mathrm{pk}_{\mathsf{PG}} \in \mathrm{PUBLICKEYS}_{\mathsf{PG}}$
    Dummy privacy guardian public key.

Outputs

$\mathrm{vk}_{\mathsf{IDP}}^{\mathtt{genesis}} \in \mathrm{VERIFICATIONKEYS}_{\mathsf{ID}}$
    The verification key for the dummy genesis identity provider.

Implementation

1: $Y_0 := X := \mathsf{H}_{\mathbb{G}_2^{\mathsf{BLS}}}^{\mathsf{BLS12\text{-}381}}(\mathrm{pk}_{\mathsf{PG}})$

```
2: for i = 1, ..., ℓ do
3:     Y_i := H_{G_2^{BLS}}^{BLS12-381}(Y_{i-1})
4: vk_IDP^{genesis} := (X, Y_1, ..., Y_ℓ)
5: return vk_IDP^{genesis}
```

</> The index of the dummy genesis identity provider is 0.

22.5 Generation of Genesis Accounts

For every account in the genesis block, we need to generate the corresponding account information (AI) object (cf. Section 15.3.2). These can be generated together with the private account information (PAI) as

$$(\texttt{AI}, \texttt{PAI}) = \texttt{generateAI}(\texttt{AC}_1, \ldots, \texttt{AC}_n, \texttt{PAC}_1, \ldots, \texttt{PAC}_n, \texttt{threshold}_\texttt{ACC}).$$

Hence, we need to generate the account credentials ($\texttt{AC}_i$) and the corresponding private account credentials ($\texttt{PAC}_i$). These are normally created using createCredential. This, however, requires an account holder certificate, which is signed by an identity provider. Since the genesis block is trusted, adding identity provider signatures can be skipped. The identity disclosure data is normally not needed as the holder of these initial accounts should be identifiable by the creator of the genesis block.

</> In the implementation, all initial accounts belong to the Concordium foundation.

To maintain compatibility with standard accounts, we provide IDs for dummy genesis identity providers and privacy guardians and include some well-formed (but dummy) identity disclosure data in the credentials. The following method can be used to generate genesis account credentials:

Protocol createGenesisCredential

Inputs

$\texttt{ID}_\texttt{IDP} \in \mathrm{IDS}_\texttt{IDP}$
    Identifier of the dummy genesis identity provider.

$\texttt{ID}_\texttt{PG} \in \mathrm{IDS}_\texttt{PG}$
    Identifier of the dummy genesis privacy guardian.

$\texttt{pk}_\texttt{PG} \in \mathrm{PUBLICKEYS}_\texttt{PG}$
    Dummy privacy guardian public encryption key to create valid looking identity disclosure data.

$\texttt{key}_\texttt{PRF} \in \mathrm{PRFKEYS}$
    The PRF key used to generate the credentials identifiers.

$x \in \mathbb{N}$
    The account credential number.

$n \in \mathbb{N}$
    The number of signature keys required.

$\texttt{threshold}_\texttt{Cred} \in \mathbb{N}$
    Threshold for signatures. It must hold $\texttt{threshold}_\texttt{Cred} \leq n$.

Outputs

AC ∈ ACs
> The account credential object.

PAC ∈ PACs
> The private account credential object.

Implementation

1. **Generate Account Credential ID:** Compute $\mathtt{ID_{AC}} := \mathsf{PRF}(\mathtt{key_{PRF}}, x)$.
2. **Generate Account Credential Keys:** Generate n signature key pairs as $(\mathtt{sk_{ACC}}_i, \mathtt{vk_{ACC}}_i)_{i \in \{1,\dots,n\}} = \mathsf{KeyGen_{ACC}}$.
3. **Identity Disclosure Data:** Compute dummy identity disclosure data $\mathtt{PGdata_{ACC}} := \mathsf{Enc_{EG}}(\mathtt{pk_{PG}}, g^1)$ and set $\mathtt{threshold_{PG}} := 1$.

> **ⓘ** Here, g corresponds to $\mathtt{IDcred_{PUB}}$ for $\mathtt{IDcred_{SEC}} = 1$. The value used is irrelevant. It could also be replaced by any other group element. Note that $\mathsf{Enc_{EG}}$ corresponds to shared encryption with a single key, i.e., $\mathsf{Enc_{PG}^{1,0}}$.

4. **Commitments and Attributes:** Set $\mathtt{publicAttrPredicate} := \bot$ and commitments $C_i = g$ for $1 \le i \le n$.
5. **Output:**

$$\mathtt{AC} := \big(\mathtt{ID_{AC}}, (\mathtt{vk_{ACC1}}, \dots, \mathtt{vk_{ACC}}n), \mathtt{threshold_{Cred}},$$
$$(C_0, \dots, C_\ell), \mathtt{publicAttrPredicate},$$
$$\mathtt{ID_{IDP}}, \mathtt{ID_{PG}}, \mathtt{threshold_{PG}}, \mathtt{PGdata_{ACC}}\big)$$

and

$$\mathtt{PAC} := \big(x, (\mathtt{sk_{ACC1}}, \dots, \mathtt{sk_{ACC}}n), \mathtt{AHC} = \bot\big)$$

```
0000 0172 eb59 566f 0000 0000 0000 0064 0000 0000 0000 0014 0000 0000 0000 0000
4268 0161 f3d3 8c70 6176 31ca 654c 5635 16d2 cf35 9fb3 3083 8c2e 2c0e 3a69 c60c
0b57 1050 e49f a67b a535 9437 8d7f 71cc 25bf 110a df61 03b4 627d a8ea 8bba a7d1
8db3 2e91 1e4f e2e5 d04d 99b0 6f46 c669 14d6 3c91 e93a 9a41 4d8c 9224 64ae 391d
a85c 0071 06cc 3dd2 8992 e345 58d0 0d12 0027 c7d8 8d59 8bc8 b62c f493 7806 4fad
b77b 2916 3478 35b9 3ffc 35cb 888a fb49 2914 6a2d 14fb 6135 c69d 7d9c 4670 62d1
0000 0008 2629 9e00 0ba3 6e61 5b04 9d7a a8c2 b8fd fe59 f803 e608 ca7a b500 8c8b
e974 b116 bf0c 0c0f 0000 0000 0000 0001 c64c 32cd f72a dfe7 3ec7 892f 6c1f f4b4
46d0 51f7 215f b95a bd7b 91bf 6355 0d6e ee94 fa68 59b1 b9e3 9439 9dbb 5fdd e4fe
58c6 88a5 d6fe 8d95 40ba dbc7 53ff e6d3 911d 1386 b56e 5f11 3c87 895e 7fe1 2ed4
f532 ce8e 4722 f204 5789 b1de 684d b219 006b 5e90 49b0 5528 271c 6f44 6b5a 5d70
117b 4519 2450 945f 20bb ecbf 19eb 4b15 bb0d c67b 3163 ab90 6307 9e1b 6b81 12cc
3afb 579b 3d5f 097e 68b2 bd24 32ba ec7e 0000 0008 2629 9e00 d013 e03b e02b 7d11
1e16 5b47 c4a5 aeb1 71d0 e98d d37c f0ce 2640 43c7 33c2 f403 0000 0000 0000 0002
d798 14a7 cc9c 24e3 71d9 f5bf fc5c e99a ece1 9caf 9021 3c29 2694 b929 8847 3989
```

Part VI

Byttes

Chapter 23

Preliminaries

23.1 Basic Representation Types

Bit	1 bit

A single *bit*, either 0 or 1. (Bits are always packed into bytes of 8 bits, with the first bit being the most significant bit.)

Word8	1 byte

A single 8-bit *byte*, representing a value in the range $[0, 255]$.

Word16	2 bytes

A 16-bit *unsigned integer*, big-endian encoding.

Word32	4 bytes

A 32-bit *unsigned integer*, big-endian encoding.

Word64	8 bytes

A 64-bit *unsigned integer*, big-endian encoding.

Int64	8 bytes

A 64-bit *signed integer*, big-endian encoding, 2's complement.

Double	8 bytes

An IEEE 754 double-precision binary *floating-point number* (binary64), big-endian encoding.

VLQ	variable length

An *unsigned integer* represented as a *variable-length quantity* [Wik25] in big-endian encoding. The high-order bit of each byte is set if further bytes follow; the remaining 7 bits are part of the data, most significant bits first. (This format allows smaller values to be represented more compactly.) We write VLQ_n to indicate that the value should be representable as an n-bit unsigned integer.

BYTES(n)	n bytes
An uninterpreted *sequence of bytes* of length n.	

SHORTBYTES	$2 +$ `length` bytes
A *sequence of bytes*, prefixed by its length represented as a WORD16.	
`length` : WORD16	
`data` : BYTES(`length`)	

LONGBYTES	$8 +$ `length` bytes
A *sequence of bytes*, prefixed by its length represented as a WORD64.	
`length` : WORD64	
`data` : BYTES(`length`)	

BOOL	1 byte
A *Boolean*. Either `true` or `false`.	
`value` : WORD8	
It must be the case that `value` $\in$ BOOL $:= \{0,1\}$. 0 represents `false`, and 1 represents `true`.	

23.2 Groups and Fields

Below we have the serialization of some groups and fields used in several cryptographic types (see also Section 2.2).

By $\mathbb{G}^{\mathtt{Ed}}$ we denote the group over the `edwards25519` curve. Details on the serialization are documented in RFC8032 [JL17].

EDG	32 bytes
A point in $\mathbb{G}^{\mathtt{Ed}}$.	
BYTES(32)	

By $\mathbb{G}_1^{\mathtt{BLS}}$ and $\mathbb{G}_2^{\mathtt{BLS}}$ we denote the subgroups of the `BLS12-381` curve as given by Definition 7. By $\mathbb{F}_{\mathtt{BLS}}$ we denote the finite field with $|\mathbb{G}_1^{\mathtt{BLS}}| = |\mathbb{G}_2^{\mathtt{BLS}}|$ elements. How exactly these are serialized are documented at [Gri25].

G1	48 bytes
A point in $\mathbb{G}_1^{\mathtt{BLS}}$.	
BYTES(48)	

<table>
<tr><td>G2</td><td>96 bytes</td></tr>
</table>

G2	96 bytes
A point in $\mathbb{G}_2^{\mathtt{BLS}}$.	
BYTES(96)	

F	32 bytes
A scalar in $\mathbb{F}_{\mathtt{BLS}}$.	
BYTES(32)	

23.3 Cryptographic Types

23.3.1 Hash

SHA256	32 bytes
A SHA-256 hash, i.e., HASH$_{\mathtt{SHA256}}$	
BYTES(32)	

SHA3	32 bytes
A SHA3-256 hash, i.e., HASH$_{\mathtt{SHA3}}$	
BYTES(32)	

23.3.2 Ed25519 Signature Scheme

ED25519SIGNPRIVKEY	32 bytes
A *signing key* $\in$ SIGNKEYS$_{\mathtt{EdDSA}}$ in the Ed25519 signature scheme.	
sk : BYTES(32)	

ED25519SIGNPUBKEY	32 bytes
A *verification key* $\in$ VERIFICATIONKEYS$_{\mathtt{EdDSA}}$ in the Ed25519 signature scheme.	
vk : EDG	

ED25519SIGNATURE	64 bytes
A *signature* $\in$ SIGNATURES$_{\mathtt{EdDSA}}$ in the Ed25519 signature scheme.	
R : EDG	
A $\mathbb{G}^{\mathtt{Ed}}$ element.	
S : BYTES(32)	
An integer encoded in 32 bytes.	

23.3.3 BLS Aggregate Signature Scheme

BLSAggSecretKey	32 bytes

A *signing key* $\in$ SIGNKEYS$_{\text{BLS}}$ in the BLS aggregate signature scheme.

`sk` : F

BLSAggPublicKey	96 bytes

A *verification key* $\in$ VERIFICATIONKEYS$_{\text{BLS}}$ in the BLS aggregate signature scheme.

`vk` : G2

BLSAggSignature	48 bytes

A *signature* $\in$ SIGNATURES$_{\text{BLS}}$ in the BLS aggregate signature scheme.

σ : G1

23.3.4 Pointcheval-Sanders Signature Scheme

In the implementation, we use the extended *Pointcheval-Sanders signature* scheme that allows signing partially unknown messages. We only provide the serialization for this variant of the scheme. See Section 5.3.3 for more details.

PSSecretKey	variable length

A *signing key* $\in$ SIGNKEYS$_{\text{PS}}$ in the extended Pointcheval-Sanders signature scheme.

g_1 : G1

A generator of $\mathbb{G}_1^{\text{BLS}}$.

g_2 : G2

A generator of $\mathbb{G}_2^{\text{BLS}}$.

`length` : Word32

Length of below list of $\mathbb{F}_{\text{BLS}}$ elements, i.e. `length` $= n$.

`yList` : `length` $\times$ F

$y_1, ..., y_n$

`x` : F

A $\mathbb{F}_{\text{BLS}}$ element.

PSPublicKey	variable length

A *verification key* $\in$ VERIFICATIONKEYS$_{\text{PS}}$ in the extended Pointcheval-Sanders signature scheme. In the description below it is assumed that the signing key is $(x, y_1, ..., y_n)$.

g_1 : G1

A generator of $\mathbb{G}_1^{\text{BLS}}$.

g_2 : G2

A generator of $\mathbb{G}_2^{\text{BLS}}$.

`length1` : WORD32	

Length of below list of $\mathbb{G}_1^{\mathtt{BLS}}$ elements, i.e. `length1` $= n$.

`g1List` : `length1` $\times$ G1	

$g_1^{y_1}, g_1^{y_2}, \ldots, g_1^{y_n}$

`length2` : WORD32	

Length of below list of $\mathbb{G}_2^{\mathtt{BLS}}$ elements, i.e. `length2` $= n$.

`g2List` : `length2` $\times$ G2	

$g_2^{y_1}, g_2^{y_2}, \ldots, g_2^{y_n}$

g_2^{x} : G2

A $\mathbb{G}_2^{\mathtt{BLS}}$ element.

PSSIGNATURE	96 bytes

A *signature* $\in$ SIGNATURES$_{\mathsf{PS}}$ in the Pointcheval-Sanders signature scheme.

$\sigma : 2 \times$ G1

A blinded signature used in zero-knowledge proofs. See Section 5.3.5 for more details.

BLINDEDSIGNATURE	96 bytes

A blinded signature.

PSSIGNATURE

23.3.5 Algebraic Pseudorandom Function

We use an algebraic *pseudorandom function* to generate account credential identifier. See Section 6.1 for more details.

CREDENTIALREGISTRATIONID	48 bytes

An *account credential identifier* $\in$ ACIDS.

G1

23.3.6 Verifiable Random Function

The following serialization is used for the elliptic curve VRF (EC-VRF) scheme presented in RFC9381 [Gol+23]. See Section 6.2 for more details.

VRFPRIVKEY	32 bytes

A *VRF private key* $\in$ PRIVATEKEY$_{\mathsf{VRF}}$ in the EC-VRF scheme.

`sk` : BYTES(32)

VRFPubKey	32 bytes

A *VRF public key* $\in$ PUBLICKEY$_{\mathsf{VRF}}$ in the EC-VRF scheme.

pk : EdG

VRFProof	80 bytes

A *VRF proof* $\in$ PROOF$_{\mathsf{VRF}}$ in the EC-VRF scheme.

Bytes(80)

23.3.7 ElGamal Encryption Scheme

The following serialization is for the *ElGamal encryption scheme* over group $\mathbb{G}_1^{\mathsf{BLS}}$ as used in our identity layer.

ElGamalSecretKey	80 bytes

A *decryption key* $\in$ PRIVATEKEYS$_{\mathsf{EG}}$ in the ElGamal encryption scheme.

g : G1

A generator of $\mathbb{G}_1^{\mathsf{BLS}}$.

dk : F

A $\mathbb{F}_{\mathsf{BLS}}$ element.

ElGamalPublicKey	96 bytes

An *encryption key* $\in$ PUBLICKEYS$_{\mathsf{EG}}$ in the ElGamal encryption scheme.

g : G1

A generator of $\mathbb{G}_1^{\mathsf{BLS}}$.

pk : G1

A $\mathbb{G}_1^{\mathsf{BLS}}$ element.

ElGamalCiphertext	96 bytes

A *ciphertext* CIPHERTEXTS$_{\mathsf{EG}}$ $\in$ in the ElGamal encryption scheme.

$2 \times$ G1

23.3.8 Pedersen Commitments

The following serialization is for the *Pedersen commitment scheme* over group $\mathbb{G}_1^{\mathsf{BLS}}$ as used in our identity layer.

CommitmentKey	96 bytes

A *commitment key* $\in$ COMKEYS in the Pedersen commitment scheme.

g : G1

A generator of $\mathbb{G}_1^{\mathsf{BLS}}$.

$h : \mathrm{G1}$
A generator of $\mathbb{G}_1^{\mathtt{BLS}}$.

VECCOMMITMENTKEY	variable length
A *commitment key* $\in$ COMKEYS in the Pedersen commitment scheme.	
`length` : WORD32	
Length of below list of $\mathbb{G}_1^{\mathtt{BLS}}$ generators, i.e. `length` $= n$.	
`gList` : `length` $\times$ G1	
Generators $g_1, \ldots, g_n$	
$h : \mathrm{G1}$	
A generator of $\mathbb{G}_1^{\mathtt{BLS}}$.	

COMMITMENT	48 bytes
A *commitment* in the Pedersen commitment scheme.	
G1	

COMMITMENTS	variable length
A list of *commitments* in the Pedersen commitment scheme.	
`length` : WORD64	
The number of Pedersen commitments.	
`commitments` : `length` $\times$ COMMITMENT	
The commitments.	

23.3.9 Sigma Protocols

The following is the serialization of a *discrete logarithm* proof over $\mathbb{G}^{\mathtt{Ed}}$ used to prove knowledge of a *signing key* $\in$ SIGNKEYS$_{\mathrm{EdDSA}}$ in the Ed25519 signature scheme. See Section 9.2.1 for details on the proof. Note that the serialized proof requires additional context, e.g. the verification key, to do proof verification.

DLOG25519PROOF	64 bytes
A dlog proof over $\mathbb{G}^{\mathtt{Ed}}$.	
`challenge` : BYTES(32)	
`response` : BYTES(32)	

More generally, we have a *discrete logarithm* proof over some group $\mathbb{G}$. See Section 9.2.1 for details on the proof. Note that the serialized proof requires additional context to do proof verification.

DLogProof	variable length
A dlog proof over a group $\mathbb{G}$.	
challenge : $\text{BYTES}(n)$	
response : $\text{BYTES}(m)$	

The following are the serializations of sub-protocols which are part of an AND-composed Σ protocol proof. That is, each serialization only contains the *response z* of the sub protocol. The challenge is stored in the serialization of the overall proof.

ComEncEqResponse	96 bytes
A response for the com-enc-eq Σ-protocol from Section 9.2.7.	
respnse : $3 \times \mathbb{F}$	

ComEqSigResponse	variable length
A response for the com-eq-sig Σ-protocol from Section 5.3.5.	
responseRho : $\mathbb{F}$	
length : WORD32	
Length of below list.	
responseCommit : $\text{length} \times (\mathbb{F}, \mathbb{F})$	

ComMultResponse	160 bytes
A response for the com-mult Σ-protocol from Section 9.2.10.	
$\texttt{s}_1, \texttt{s}_2 : \mathbb{F} \times \mathbb{F}$	
$\texttt{t}_1, \texttt{t}_2 : \mathbb{F} \times \mathbb{F}$	
$\texttt{t} : \mathbb{F}$	

ComEqResponse	64 bytes
A response for the com-dlog-eq Σ-protocol from Section 9.2.5.	
respnse : $2 \times \mathbb{F}$	

23.3.10 Range proofs

RangeProof	variable length
A Bulletproof *range proof* from Section 9.5.3.	
g1Elements : $4 \times \mathbb{G}1$	
Four $\mathbb{G}_1^{\text{BLS}}$ elements representing $\texttt{A}, \texttt{S}, \texttt{T1}, \texttt{T2}$.	
scalars1 : $3 \times \mathbb{F}$	
Three scalars from $\mathbb{F}_{\text{BLS}}$ representing $\texttt{tx}, \texttt{txTilde}, \texttt{eTilde}$.	

length : WORD32

Length of below list of pairs of G_1 elements.

groupElementList : length $\times$ (G1, G1)

A list of pairs of $\mathbb{G}_1^{\texttt{BLS}}$ elements, representing the LR vector of the inner product proof.

scalars2 : 2 $\times$ F

Two scalars from $\mathbb{F}_{\texttt{BLS}}$, representing a, b of the inner product proof.

23.3.11 Combined Proofs

The following are the serializations for the zero-knowledge proof used to open accounts or deploy account credentials. See Sections 15.3.3 and 15.3.4 for more details. The naming follows the implementation.

IDOWNERSHIPPROOFS	variable length

A proof showing that the credential information has been computed correctly from an identity credential.

sig : BLINDEDSIGNATURE

A blinded signature derived from the signature on the identity credential by the identity provider (IDP).

commitments : CREDENTIALDEPLOYMENTCOMMITMENTS

Auxiliary commitments of the proof.

challenge : BYTES(32)

The challenge of the Σ-protocol (an AND composition of sub-protocols)

proofIdCredPub : IDCREDPUBPROOF

Responses for the sub-protocol showing that the public holder identifier has been correctly encrypted under the privacy guardian keys.

proofIpSig : COMEQSIGRESPONSE

Response for the sub-protocol showing that the involved values have been signed by the IDP.

proofRegId : COMMULTRESPONSE

Proof that $\texttt{credId} = \mathsf{PRF}(\texttt{key}_{\texttt{PRF}}, x)$. Also establishes that the PRF key k has been signed by the IDP.

proofCredCounter : RANGEPROOF

Proof that the credential counter x is less than the maximal number of accounts.

CREDENTIALDEPLOYMENTCOMMITMENTS	variable length

Auxiliary commitments used in IDOWNERSHIPPROOFS.

prf : COMMITMENT

Commitment to the PRF key k of the identity credential.

credCounter : COMMITMENT	
Commitment to the credential counter x.	
maxAcoounts : COMMITMENT	
Commitment to the maximal number of accounts.	
attributes : ATTRIBUTECOMMITMENTS	
List of commitments to identity attributes that are not revealed.	
sharingCoeffs : COMMITMENTS	
List of commitments to the coefficients of the polynomial used to (secret) share the public holder identifier.	

ATTRIBUTECOMMITMENTS	variable length
A list of commitments to identity attributes that are not revealed.	
length : WORD16	
The number of attribute commitments.	
attributeCommitments : length × ATTRIBUTECOMMITMENT	
The list of attribute commitments.	

ATTRIBUTECOMMITMENT	49 bytes
An attribute tag and a Pedersen commitment to its value.	
attributeTag : WORD8	
Tag identifying which attribute is inside the commitment.	
commitment : COMMITMENT	
The Pedersen commitment to the attribute.	

IDCREDPUBPROOF	variable length
A proof part showing that the public holder identifier has been correctly shared and encrypted.	
length : WORD32	
Number of privacy guardians having encrypted shares of the public holder identifier.	
idCredPubProofEntry : length × IDCREDPUBPROOFENTRY	
A list containing a proof part for each share.	

IDCREDPUBPROOFENTRY	100 bytes
A proof part showing correct encryption of a privacy guardian's share of the public holder identifier. Technically, shows that the share ciphertext contains the same value as the (auxiliary) commitment to the share.	
arIdentity : ARIDENTITY	
The privacy guardian (PG) identity.	
response : COMENCEQRESPONSE	

The following serialization are for zero-knowledge proofs used in shielded transfers. See Section 16.3 for more details.

ENCTRANSWITNESS	variable length
A Σ-protocol response showing that the accounting is done correctly as part of a ENCRYPTEDAMOUNTTRANSFERPROOF or SECTOPUBAMOUNTTRANSFERPROOF.	
responseCommon : F	
length1 : WORD32	
Length of below list of responses.	
responseEncexp1 : length1 × COMEQRESPONSE	
length2 : WORD32	
Length of below list of responses.	
responseEncexp2 : length2 × COMEQRESPONSE	

ENCRYPTEDAMOUNTTRANSFERDATA	variable length
Data that will go onto a shielded amount transfer.	
remainingAmount : ENCRYPTEDAMOUNT	
Encryption of the remaining amount.	
transferAmount : ENCRYPTEDAMOUNT	
Encryption of the amount to be sent.	
index : WORD64	
The index such that the encrypted amount used in the transfer represents the aggregate of all encrypted amounts with indices < 'index' existing on the account at the time. New encrypted amounts can only add new indices.	
proofs : ENCRYPTEDAMOUNTTRANSFERPROOF	
A proof that the transfer has been crafted correctly.	

ENCRYPTEDAMOUNTTRANSFERPROOF	variable length
Proof that a shielded transfer data is well-formed.	
challenge : BYTES(32)	
The challenge of the Σ-protocol below.	
accounting : ENCTRANSWITNESS	
Response for the proof that accounting is done correctly, i.e., remaining + transfer is the original amount.	
transferAmount : RANGEPROOF	
Proof that the transferred amount is correctly encrypted, i.e., that the chunks are small enough.	
remainingAmount : RANGEPROOF	
Proof that the remaining amount is correctly encrypted, i.e., that the chunks are small	

enough.

SECTOPUBAMOUNTTRANSFERDATA	variable length

Data that will go onto a transfer from shielded to public amount.

remainingAmount : ENCRYPTEDAMOUNT

Encryption of the remaining amount.

transferAmount : AMOUNT

Amount to be made public.

index : WORD64

The index such that the encrypted amount used in the transfer represents the aggregate of all encrypted amounts with indices < 'index' existing on the account at the time. New encrypted amounts can only add new indices.

proofs : SECTOPUBAMOUNTTRANSFERPROOF

A proof that the transfer has been crafted correctly.

SECTOPUBAMOUNTTRANSFERPROOF	variable length

Proof that a shielded to public amount transfer data is well-formed

challenge : BYTES(32)

The challenge of the Σ-protocol below.

accounting : ENCTRANSWITNESS

Witness for the proof that accounting is done correctly, i.e., remaining + transfer is the original amount.

remainingAmount : RANGEPROOF

Proof that the remaining amount is correctly encrypted, i.e., chunks small enough.

23.4 Blockchain Types

ACCOUNTADDRESS	32 bytes

An account address.

BYTES(32)

ℹ An account address is normally displayed in base58 encoded form. The encoded information consists of a 1 byte version tag, currently set to 1, followed by the address (32 bytes), and a 4 byte checksum. The checksum and encoding is computed according to the Base58Check protocol as used for Bitcoin addresses.

SEQUENCENUMBER	8 bytes

The sequence number of a transaction on an account.

WORD64	

ENERGY	8 bytes

An amount of energy (NRG).

WORD64	

AMOUNT	8 bytes

An amount of CCD, expressed in micro-CCD.

WORD64	

AMOUNTFRACTION	4 bytes

A fraction of an amount represented as parts per hundred thousand.

fraction : WORD32	

$\texttt{fraction} \leq 100000$.

ENCRYPTEDAMOUNT	192 bytes

A shielded amount encrypted in two ElGamal parts (cf. Section 16.3).

$2 \times$ ELGAMALCIPHERTEXT	

MEMO	variable length

A memo attached to a transfer.

length : WORD16	

Length of the memo in bytes. $\texttt{length} \leq 256$.

memo : BYTES(length)	

The memo value.

PAYLOADSIZE	4 bytes

The size of a transaction payload in bytes.

WORD32	

TRANSACTIONEXPIRYTIME	8 bytes

Expiry time of a transaction in seconds since the UNIX epoch.

WORD64	

TRANSACTIONTIME	8 bytes

Transaction time in seconds since the UNIX epoch.

WORD64	

TIMESTAMP	8 bytes
Point in time in milliseconds since the UNIX epoch.	
WORD64	

DURATION	8 bytes
A duration in milliseconds.	
WORD64	

DURATIONSECONDS	8 bytes
A duration in seconds.	
WORD64	

23.4.1 Identity Provider and Privacy Guardian

DESCRIPTION	variable length
Name, URL, and description of either a privacy guardian or identity provider.	
nameLength : WORD32	
name : BYTES(nameLength)	
urlLength : WORD32	
url : BYTES(urlLength)	
descriptionLength : WORD32	
description : BYTES(descriptionLength)	

ARINFO	variable length
Information on a single privacy guardian held by the identity provider.	
arIdentity : ARIDENTITY	
description : DESCRIPTION	
publicKey : ARPUBLICKEY	

ARIDENTITY	4 bytes
A number (uniquely) identifying a privacy guardian. **Must be nonzero**.	
WORD32	

IPINFO	variable length
Public information about an identity provider.	
ipIdentity : IPIDENTITY	

`description` : Description	
`publicKeys` : IpPublicKeys	

IpIdentity	4 bytes
A number (uniquely) identifying an identity provider.	
Word32	

AnonymityRevocationThreshold	1 bytes
Number of privacy guardians required to disclose the identity of an account credential.	
Word8	

23.4.2 Chain Updates

UpdateSequenceNumber	8 bytes
The sequence number of a chain update.	
Word64	

RootUpdate	variable length
Root updates are the highest kind of updates. They can update every other set of keys, even themselves. They can only be performed by Root level keys. One of the following:	

RootUpdate_{RootKeys} → $\text{RootUpdate}_{\text{RootKeys}}$

$\text{RootUpdate}_{\text{RootKeys}}$

$\text{RootUpdate}_{\text{Level1Keys}}$

Protocol versions 1–3:

$\text{RootUpdate}_{\text{Level2KeysV0}}$

Protocol versions 4 onwards:

$\text{RootUpdate}_{\text{Level2KeysV1}}$

Protocol versions 9 onwards:

$\text{RootUpdate}_{\text{Level2KeysV2}}$

HigherLevelKeys	variable length
A list of higher level keys (either root or Level 1 keys).	
`length` : Word16	
Number of keys.	
`keys` : `length` × UpdateVerifyKey	
The keys.	

threshold : WORD16

The threshold. Must be nonzero.

ROOTUPDATE_{ROOTKEYS}   variable length

Updating the root keys.

type = 0 : WORD8

keys : HIGHERLEVELKEYS

ROOTUPDATE_{LEVEL1KEYS}   variable length

Updating the Level 1 keys.

type = 1 : WORD8

keys : HIGHERLEVELKEYS

ROOTUPDATE_{LEVEL2KEYSV0}   variable length

Updating the Level 2 keys, in protocol versions 1–3.

type = 2 : WORD8

authorization : AUTHORIZATIONSV0

ROOTUPDATE_{LEVEL2KEYSV1}   variable length

Updating the Level 2 keys, from protocol version 4 onwards.

type = 3 : WORD8

authorization : AUTHORIZATIONSV1

ROOTUPDATE_{LEVEL2KEYSV2}   variable length

Updating the Level 2 keys, from protocol version 9 onwards.

type = 4 : WORD8

authorization : AUTHORIZATIONSV2

LEVEL1UPDATE   variable length

Level 1 updates are the intermediate update kind. They can update themselves or level 2 keys. They can only be performed by Level 1 keys. One of the following:

> LEVEL1UPDATE_{LEVEL1KEYS}

Protocol versions 1–3:

> LEVEL1UPDATE_{LEVEL2KEYSV0}

Protocol version 4 onwards:

$$\boxed{\text{Level1Update}_{\text{Level2KeysV1}}}$$

Protocol version 9 onwards:

$$\boxed{\text{Level1Update}_{\text{Level2KeysV2}}}$$

Level1Update$_{\text{Level1Keys}}$	variable length

Updating the Level 1 keys.

type $= 0$: Word8

keys : HigherLevelKeys

Level1Update$_{\text{Level2KeysV0}}$	variable length

Updating the Level 2 keys, in protocol versions 1–3.

type $= 1$: Word8

authorization : AuthorizationsV0

Level1Update$_{\text{Level2KeysV1}}$	variable length

Updating the Level 2 keys, in protocol version 4 onwards.

type $= 2$: Word8

authorization : AuthorizationsV1

Level1Update$_{\text{Level2KeysV2}}$	variable length

Updating the Level 2 keys, in protocol version 9 onwards.

type $= 3$: Word8

authorization : AuthorizationsV2

AuthorizationsV0	variable length

The set of keys authorized for chain updates, together with access structures determining which keys are authorized for which update types. This is the payload of an update to authorization. This is used for protocol versions 0–3, after which it is superseded by AuthorizationsV1.

length : Word16

Number of keys.

keys : length $\times$ UpdateVerifyKey

All the keys that can do Level 2 updates.

emergency : AccessStructure

The subset of all the keys that can do emergency updates together with the threshold, i.e. how many of them needed to authorize the update.

protocol : AccessStructure

The subset of all the keys that can do protocol updates together with the threshold.

consensusParameters : AccessStructure

The subset of all the keys that can do election difficulty updates together with the threshold.

euroPerEnergy : AccessStructure

The subset of all the keys that can do euro-per-energy updates together with the threshold.

microCCDPerEuro : AccessStructure

The subset of all the keys that can do microCCD-per-euro updates together with the threshold.

foundationAccount : AccessStructure

The subset of all the keys that can do foundation account updates together with the threshold.

mintDistribution : AccessStructure

The subset of all the keys that can do mint distribution updates together with the threshold.

transactionFeeDistribution : AccessStructure

The subset of all the keys that can do transaction fee distribution updates together with the threshold.

GASRewards : AccessStructure

The subset of all the keys that can do gas rewards updates together with the threshold.

bakerStakeThreshold : AccessStructure

The subset of all the keys that can do baker stake threshold updates together with the threshold.

addAnonymityRevoker : AccessStructure

The subset of all the keys that can add privacy guardians together with the threshold.

addIdentityProvider : AccessStructure

The subset of all the keys that can add identity providers together with the threshold.

AuthorizationsV1   variable length

The set of keys authorized for chain updates, together with access structures determining which keys are authorized for which update types. This is the payload of an update to authorization. This is used for protocol versions 4–8, after which it is superseded by AuthorizationsV2.

length : Word16

Number of keys.

keys : length × UpdateVerifyKey

All the keys that can do Level 2 updates.

emergency : AccessStructure

The subset of all the keys that can do emergency updates together with the threshold, i.e. how many of them needed to authorize the update.

protocol : AccessStructure

The subset of all the keys that can do protocol updates together with the threshold.

consensusParameters : AccessStructure

The subset of all the keys that can do consensus parameter updates together with the threshold.

euroPerEnergy : AccessStructure

The subset of all the keys that can do euro-per-energy updates together with the threshold.

microCCDPerEuro : AccessStructure

The subset of all the keys that can do microCCD-per-euro updates together with the threshold.

foundationAccount : AccessStructure

The subset of all the keys that can do foundation account updates together with the threshold.

mintDistribution : AccessStructure

The subset of all the keys that can do mint distribution updates together with the threshold.

transactionFeeDistribution : AccessStructure

The subset of all the keys that can do transaction fee distribution updates together with the threshold.

GASRewards : AccessStructure

The subset of all the keys that can do gas rewards updates together with the threshold.

poolParameters : AccessStructure

The subset of all the keys that can do pool parameter updates together with the threshold. (This is a renaming of `bakerStakeThreshold` from AuthorizationsV0.)

addAnonymityRevoker : AccessStructure

The subset of all the keys that can add anonymity revokers together with the threshold.

addIdentityProvider : AccessStructure

The subset of all the keys that can add identity providers together with the threshold.

cooldownParameters : AccessStructure

The subset of all the keys that can update the cooldown parameters together with the threshold.

timeParameters : AccessStructure

The subset of all the keys that can update the length of a reward period and mint rate together with the threshold.

AuthorizationsV2   variable length

The set of keys authorized for chain updates, together with access structures determining which keys are authorized for which update types. This is the payload of an update to authorization. This is used from protocol version 9 onwards.

length : Word16

Number of keys.

keys : `length` × UPDATEVERIFYKEY

All the keys that can do Level 2 updates.

emergency : ACCESSSTRUCTURE

The subset of all the keys that can do emergency updates together with the threshold, i.e. how many of them needed to authorize the update.

protocol : ACCESSSTRUCTURE

The subset of all the keys that can do protocol updates together with the threshold.

consensusParameters : ACCESSSTRUCTURE

The subset of all the keys that can do consensus parameter updates together with the threshold.

euroPerEnergy : ACCESSSTRUCTURE

The subset of all the keys that can do euro-per-energy updates together with the threshold.

microCCDPerEuro : ACCESSSTRUCTURE

The subset of all the keys that can do microCCD-per-euro updates together with the threshold.

foundationAccount : ACCESSSTRUCTURE

The subset of all the keys that can do foundation account updates together with the threshold.

mintDistribution : ACCESSSTRUCTURE

The subset of all the keys that can do mint distribution updates together with the threshold.

transactionFeeDistribution : ACCESSSTRUCTURE

The subset of all the keys that can do transaction fee distribution updates together with the threshold.

GASRewards : ACCESSSTRUCTURE

The subset of all the keys that can do gas rewards updates together with the threshold.

poolParameters : ACCESSSTRUCTURE

The subset of all the keys that can do pool parameter updates together with the threshold. (This is a renaming of `bakerStakeThreshold` from AUTHORIZATIONSV0.)

addAnonymityRevoker : ACCESSSTRUCTURE

The subset of all the keys that can add anonymity revokers together with the threshold.

addIdentityProvider : ACCESSSTRUCTURE

The subset of all the keys that can add identity providers together with the threshold.

cooldownParameters : ACCESSSTRUCTURE

The subset of all the keys that can update the cooldown parameters together with the threshold.

timeParameters : ACCESSSTRUCTURE

The subset of all the keys that can update the length of a reward period and mint rate together with the threshold.

createPLT : ACCESSSTRUCTURE

The subset of all the keys that can perform an UPDATEPAYLOAD_CREATEPLT transaction together with the threshold.

ACCESSSTRUCTURE	variable length

Access structure for level 2 update authorization.

`length` : WORD8

Number of keys indices.

`keysIndices` : `length` × WORD16

Which keys that can authorize an update.

`threshold` : WORD16

The number of keys needed to authorize an update.

UPDATEVERIFYKEY	33 bytes

`scheme = 0` : WORD8

`key` : ED25519SIGNPUBKEY

23.4.3 Tokenomics

EXCHANGERATE	16 bytes

An exchange rate (e.g., microCCD/Euro or Euro/Energy). Infinity and zero are disallowed.

`numerator` : WORD64

`denominator` : WORD64

MINTRATE	5 bytes

A base-10 floating point number representation. The value is `mantissa` $* 10^{(-\texttt{exponent})}$.

`mantissa` : WORD32

`exponent` : WORD8

COMMISSIONRANGE	6 bytes

A permissible range for a particular commission rate.

`min` : AMOUNTFRACTION

Minimum commission rate.

`max` : AMOUNTFRACTION

Maximum commission rate.
`min ≤ max`.

COMMISSIONRANGES	18 bytes

A set of permissible commission rate ranges.

finalizationCommissionRange : COMMISSIONRANGE	

Permissible range for finalization commission.

bakingCommissionRange : COMMISSIONRANGE	

Permissible range for baking commission.

transactionCommissionRange : COMMISSIONRANGE	

Permissible range for transaction commission.

CAPITALBOUND	4 bytes

A bound on the relative share of the total staked capital that a validator may have as its stake.

capitalBound : AMOUNTFRACTION	

capitalBound.fraction > 0.

LEVERAGEFACTOR	16 bytes

The maximum ratio of baker's effective stake to its equity capital. This cannot be less than 1. This must be represented in normalized form.

numerator : WORD64	

The numerator.

denominator : WORD64	

The denominator.
$0 <$ denominator $\leq$ numerator.
$\gcd($numerator, denominator$) = 1$.

23.4.4 Smart Contracts

MODULE	variable length

Web assembly module in binary format.

version : WORD32	

length : WORD32	

Length of source.

source : BYTES(length)	

The source of a contract in binary wasm format.

MODULEREF	32 bytes

Unique module reference.

hash : SHA256	

INITNAME	variable length
Name of an init method inside a module.	
SHORTBYTES	

RECEIVENAME	variable length
Name of a receive method inside a module.	
SHORTBYTES	

PARAMETER	variable length
Parameter to either an init method or to a receive method. The parameter is limited to 1kB in size.	
SHORTBYTES	

CONTRACTADDRESS	16 bytes
A contract address consists of an index and a subindex.	
contractIndex : WORD64	
contractSubIndex : WORD64	

23.4.5 Consensus and Delegation

VALIDATORID	8 bytes
Identifier of a validator (cf. Definition 46).	
WORD64	

VALIDATORKEYSWITHPROOFS	352 bytes
The keys of a validator, together with proofs that the validator knows the corresponding secret keys (cf. Definition 46).	
electionVerifyKey : BAKERELECTIONVERIFYKEY	
Public key used for VRF proofs in the election process.	
proofElection : DLOG25519PROOF	
Proof that the validator knows the secret key corresponding to electionVerifyKey.	
signatureVerifyKey : BAKERSIGNVERIFYKEY	
Public key used for signing blocks.	
proofSig : DLOG25519PROOF	
Proof that the validator knows the secret key corresponding to signatureVerifyKey.	
aggregationVerifyKey : BAKERAGGREGATIONVERIFYKEY	
Public key used for aggregate signatures.	
proofAggregation : BAKERAGGREGATIONPROOF	
Proof that the validator knows the secret key corresponding to aggregationVerifyKey.	

ELECTIONDIFFICULTY	4 bytes

Election difficulty parameter in parts per hundred thousands (cf. Definition 120)

WORD32

ℹ️ The election difficulty is no longer used in the new consensus protocol. It is only relevant for protocol version 1-5.

COMMISSIONRATES	12 bytes

The commission rates charged by a pool owner (or for passive delegation).

finalizationCommission : AMOUNTFRACTION

Fraction of finalization rewards charged by the pool owner.

bakingCommission : AMOUNTFRACTION

Fraction of baking rewards charged by the pool owner.

transactionCommission : AMOUNTFRACTION

Fraction of transaction rewards charged by the pool owner.

OPENSTATUS	1 byte

The status of a validator's pool. One of the following:

OPENSTATUS_OPENFORALL

OPENSTATUS_CLOSEDFORNEW

OPENSTATUS_CLOSEDFORALL

OPENSTATUS_OPENFORALL	1 byte

The pool is open for all delegators.

$\text{status} = 0$: WORD8

OPENSTATUS_CLOSEDFORNEW	1 byte

The pool is closed to new delegators.

$\text{status} = 1$: WORD8

OPENSTATUS_CLOSEDFORALL	1 byte

The pool is closed for all delegators.

$\text{status} = 2$: WORD8

URLTEXT	variable length

A URL.

length : WORD16

Length of the URL in bytes. `length` $\leq$ 2048.

url : BYTES(**length**)

The URL, in UTF-8 encoding.

DELEGATIONTARGET	variable length

A target that an account may delegate stake to (Passive or a Validator). One of the following:

> DELEGATIONTARGET$_{\text{PASSIVE}}$

> DELEGATIONTARGET$_{\text{VALIDATOR}}$

DELEGATIONTARGET$_{\text{PASSIVE}}$

Delegate passively to all validators.

tag = 0 : WORD8

DELEGATIONTARGET$_{\text{VALIDATOR}}$

Delegate to a specific validator.

tag = 1 : WORD8

validator : VALIDATORID

The Validator ID of the validator pool to delegate to.

23.4.6 Protocol Level Token

TOKENID	variable length

The token id uniquely identifying a token.

length : WORD8

Length of the token id in bytes.

tokenId : BYTES(**length**)

The token id, only consisting of a-z, A-Z, 0-9, '.', '-' and '%'. Between 1 and 128 bytes.

Chapter 24

Network Serialization

24.1 Account Keys

An account public key consists of a scheme identifier followed by a public key in the corresponding scheme. Currently, only one scheme is supported: Ed25519.

CREDENTIALVERIFYKEY	variable length
A credential public key. One of the following:	
CREDENTIALVERIFYKEY$_{\text{Ed25519}}$	

CREDENTIALVERIFYKEY$_{\text{Ed25519}}$	33 bytes
$\texttt{scheme} = 0 :$ WORD8	
$\texttt{key} :$ ED25519SIGNPUBKEY	

ACCOUNTOWNERSHIPPROOF	variable length
A number of signatures by an account owner needed to proof ownership of an account. What exactly is signed depends on whether the credential is an initial or a normal one.	
$\texttt{count} :$ WORD8	
The number of signatures. *Must be nonzero.*	
$\texttt{proofs} : \texttt{count} \times$ ACCOUNTOWNERSHIPPROOFENTRY	
The signatures.	

ACCOUNTOWNERSHIPPROOFENTRY	65 bytes
A signature by one of the account owner's keys.	
$\texttt{index} :$ WORD8	
Index identifying which of the account owner's keys this signature is by.	
$\texttt{sig} :$ ED25519SIGNATURE	
The signature.	

SignatureThreshold	1 byte
threshold : Word8	
threshold > 0.	

AccountThreshold	1 byte
threshold : Word8	
threshold > 0.	

CredentialVerifyKeyEntry	variable length
index : Word8	
key : CredentialVerifyKey	

24.2 Privacy Guardian and Identity Provider Keys

ArPublicKey	96 bytes
Public key of a privacy guardian (cf. Definition 95).	
key : ElGamalPublicKey	
The key.	

IpPublicKeys	variable length
Public keys of an identity provider (cf. Definition 94).	
psKey : PSPublicKey	
The Pointcheval-Sanders public key.	
edKey : Ed25519SignPubKey	
The Ed25519 public key.	

24.3 Validator keys

BakerElectionVerifyKey	32 bytes
Public key for baker election.	
key : VRFPubKey	
The key.	

BakerSignVerifyKey	32 bytes
Public key for baker signatures.	
key : Ed25519SignPubKey	
The key.	

BAKERAGGREGATIONVERIFYKEY	96 bytes
Public key for aggregated signatures.	
key : BLSAGGPUBLICKEY	
The key.	

BAKERAGGREGATIONPROOF	64 bytes
Proof of knowledge of secret key	
proof : DLOGPROOF	
The proof.	

24.4 Transaction

24.4.1 Block Items

BLOCKITEM	variable length
One of the following:	

> BLOCKITEM_TRANSACTION
>
> BLOCKITEM_CREDENTIALDEPLOYMENT
>
> BLOCKITEM_CHAINUPDATE

BLOCKITEM_TRANSACTION	variable length
type = 0 : WORD8	
transaction : ACCOUNTTRANSACTION	

BLOCKITEM_CREDENTIALDEPLOYMENT	variable length
type = 1 : WORD8	
credential : ACCOUNTCREATION	

BLOCKITEM_CHAINUPDATE	variable length
type = 2 : WORD8	
update : UPDATEINSTRUCTION	

24.4.2 Transactions

ACCOUNTTRANSACTION	variable length

A transaction originating from a particular account.

signature : TRANSACTIONSIGNATURE

The signature on the transaction by the source account.

header : TRANSACTIONHEADER

Transaction header data.

encodedPayload : BYTES(`header.payloadSize`)

Transaction payload. Typically, this is a TRANSACTIONPAYLOAD, however, deserialization of the payload is a separate step from deserialization of the transaction.

For an ACCOUNTTRANSACTION, the message to be signed is the SHA256 hash of concatenation of the header and payload:

$$\text{SHA256}(\texttt{header}\|\texttt{encodedPayload})$$

TRANSACTIONSIGNATURE	variable length

A collection of cryptographic signatures on a transaction by the keys of credentials deployed to the source account.

count : WORD8

The number of account signatures, i.e. the number of owners of the source account signing the transaction.

accountSignatures : count × ACCOUNTSIGNATURE

Signatures, ordered by increasing key index, and with no duplicate keys.

⚠ The ACCOUNTSIGNATUREs in a TRANSACTIONSIGNATURE *must* be in ascending order, and have no duplicate keys. Otherwise the transaction will not be considered valid.

ACCOUNTSIGNATURE	variable length

credentialIndex : WORD8

Index identifying which of the account owners these signatures are by.

sigCount : WORD8

Number of signatures by this account owner.

signatures : sigCount × SIGNATUREENTRY

The signatures by this account owner.

SIGNATUREENTRY	

keyIndex : WORD8

Index identifying which of the account owner's keys this signature is by.

signature : SHORTBYTES	
The signature.	

TRANSACTIONHEADER	60 bytes
sender : ACCOUNTADDRESS	
The address of the account that is the source of the transaction.	
sequenceNumber : SEQUENCENUMBER	
The sequence number of the transaction. Transactions executed on an account must have sequential sequence numbers, starting from 1.	
energyAmount : ENERGY	
The amount of energy allocated for executing this transaction. (This is the maximum amount of energy that can be consumed by the transaction.)	
payloadSize : PAYLOADSIZE	
The size of the transaction payload.	
expiry : TRANSACTIONEXPIRYTIME	
The time at which the transaction expires. A transaction cannot be included in a block with a timestamp later than the transaction's expiry time.	

24.4.3 Credential Deployment

YEARMONTH	3 bytes
A year and month.	
year : WORD16	
$1000 \leq$ year ≤ 9999.	
month : WORD8	
$1 \leq$ month ≤ 12.	

ACCOUNTCREATION	variable length
A credential together with an expiry. It is a message that is included in a block, if valid, but it is not paid for directly by the sender.	
messageExpiry : TRANSACTIONEXPIRYTIME	
credential : ACCOUNTCREDENTIALWITHPROOFS	

ACCOUNTCREDENTIALWITHPROOFS	variable length
Different kinds of credentials that can go onto the chain. One of the following:	

> ACCOUNTCREDENTIALWITHPROOFS_{INITIAL}

> ACCOUNTCREDENTIALWITHPROOFS_{NORMAL}

AccountCredentialWithProofs_{Initial}	variable length

The initial credential deployment information consists of values deployed and a signature from the identity provider on said values.

type = 0 : Word8

0 indicates that this is the initial variant of AccountCredentialWithProofs.

values : InitialCredentialDeploymentValues

The data for the initial account creation. This is submitted by the identity provider on behalf of the account holder.

sig : Ed25519Signature

A signature under the public key of identity provider's key on the credential deployment values. This is the dual of 'proofs' for the normal credential deployment.

InitialCredentialDeploymentValues	variable length

publicKeys : CredentialPublicKeys

The account threshold together with the public keys of this credential, i.e. the keys used to check a signature made by the account owner holding this credential, and how many of the account owner's keys needed to sign.

credId : CredentialRegistrationID

Registration ID of this credential.

ipId : IPIdentity

Identity of the identity provider who signed the identity object from which this credential is derived.

policy : Policy

Policy.

Policy	variable length

validTo : YearMonth

Validity of credential.

createdAt : YearMonth

Creation of credential.

count : Word16

Number of attributes in the policy.

attributes : count × AttributeEntry

The attributes in this policy.

AttributeEntry	variable length

attributeTag : Word8

Tag identifying which attribute has the below value.

count : Word8

Length of the attribute value in bytes. `count` ≤ 31.

`attributeValue` : BYTES($count$)

Value of this attribute.

ACCOUNTCREDENTIALWITHPROOFS_{NORMAL}	variable length

`type = 1` : WORD8

1 indicates that this is the normal variant of ACCOUNTCREDENTIALWITHPROOFS.

`cdi` : CREDENTIALDEPLOYMENTINFORMATION

The credential deployment information consists of values deployed and the proofs about them.

CREDENTIALDEPLOYMENTINFORMATION	variable length

`values` : CREDENTIALDEPLOYMENTVALUES

`proofs` : CREDENTIALDEPLOYMENTPROOFS

CREDENTIALDEPLOYMENTPROOFS	variable length

`idProofs` : IDOWNERSHIPPROOFS

All proofs required to prove ownership of an identity, in a credential deployment.

`proofAccSk` : ACCOUNTOWNERSHIPPROOF

A signature by the credential holder on the credential deployment values, the IDOWNERSHIPPROOFS and either the account address, if the credential is being deployed to an existing account, or the transaction expiry, if the credential is being deployed to a new account.

CREDENTIALDEPLOYMENTVALUES	variable length

`publicKeys` : CREDENTIALPUBLICKEYS

The account threshold together with the public keys of this credential, i.e. the keys used to check a signature made by the account owner holding this credential, and how many of the account owner's keys needed to sign.

`credId` : CREDENTIALREGISTRATIONID

Registration ID of this credential.

`ipId` : IPIDENTITY

Identity of the identity provider who signed the identity object from which this credential is derived.

`revocationThreshold` : ANONYMITYREVOCATIONTHRESHOLD

Revocation threshold. Any set of this many privacy guardians can reveal IdCredPub.

`arData` : ARDATA

Identity disclosure data associated with this credential.

`policy` : POLICY

Policy.

CREDENTIALPUBLICKEYS

The public keys of a credential.

`count` : WORD8

Number of keys. `count` > 0.

`keys` : count × CREDENTIALVERIFYKEYENTRY

The keys.

`threshold` : SIGNATURETHRESHOLD

The signature threshold, i.e. how many of the concrete account owner's keys needed to sign.

ARDATA

Identity disclosure data associated with a credential.

`count` : WORD8

Number of privacy guardians. `count` > 0.

`data` : count × ARDATAENTRY

The data.

ARDATAENTRY   100 bytes

Identity disclosure data of one privacy guardian.

`arIdentity` : ARIDENTITY

The ID of the privacy guardian that can decrypt the below data.

`data` : CHAINARDATA

The data.

CHAINARDATA   96 bytes

Data needed on-chain to disclose the identity of the account holder.

`idCredPubShare` : ELGAMALCIPHERTEXT

Encrypted share of idCredPub.

24.4.4 Chain Updates

UPDATEINSTRUCTION   variable length

An update instruction.

`header` : UPDATEHEADER

`payload` : UPDATEPAYLOAD

`signatures` : UPDATEINSTRUCTIONSIGNATURES

UPDATEINSTRUCTIONSIGNATURES	variable length

Signatures on an update instruction.

length : WORD16

Number of signatures.

signatures : length × (WORD16, SHORTBYTES)

The indices of the keys used to sign together with the signatures.

UPDATEHEADER	28 bytes

The header for an update instruction, consisting of the sequence number, effective time, expiry time (timeout), and payload size. This structure is the same for all update payload types.

seqNumber : UPDATESEQUENCENUMBER

effectiveTime : TRANSACTIONTIME

timeout : TRANSACTIONEXPIRYTIME

payloadSize : PAYLOADSIZE

UPDATEPAYLOAD	variable length

The payload of an update instruction. One of the following:

> UPDATEPAYLOAD_PROTOCOL

> UPDATEPAYLOAD_EUROPERENERGY

> UPDATEPAYLOAD_MICROCCDPEREURO

> UPDATEPAYLOAD_FOUNDATIONACCOUNT

> UPDATEPAYLOAD_TRANSACTIONFEEDISTRIBUTION

> UPDATEPAYLOAD_ROOTUPDATE

> UPDATEPAYLOAD_LEVEL1

> UPDATEPAYLOAD_ADDANONYMITYREVOKER

> UPDATEPAYLOAD_ADDIDENTITYPROVIDER

The following are supported at protocol versions 1–3 only:

> UPDATEPAYLOAD_MINTDISTRIBUTIONV0

<div style="border:1px solid #bcd; padding:1em;">

$$\boxed{\text{UPDATEPAYLOAD}_{\text{BAKERSTAKETHRESHOLD}}}$$

The following are supported at protocol versions 1–5 only:

$$\boxed{\text{UPDATEPAYLOAD}_{\text{ELECTIONDIFFICULTY}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{GASREWARDSV0}}}$$

The following are supported from protocol version 4 onwards:

$$\boxed{\text{UPDATEPAYLOAD}_{\text{COOLDOWNPARAMETERS}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{POOLPARAMETERS}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{TIMEPARAMETERS}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{MINTDISTRIBUTIONV1}}}$$

The following are supported from protocol version 6 onwards:

$$\boxed{\text{UPDATEPAYLOAD}_{\text{TIMEOUTPARAMETERS}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{MINBLOCKTIME}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{BLOCKENERGYLIMIT}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{GASREWARDSV1}}}$$

$$\boxed{\text{UPDATEPAYLOAD}_{\text{FINALIZATIONCOMMITTEEPARAMETERS}}}$$

The following is supported from protocol version 8 onwards:

$$\boxed{\text{UPDATEPAYLOAD}_{\text{VALIDATORSCOREPARAMETERS}}}$$

The following is supported from protocol version 9 onwards:

$$\boxed{\text{UPDATEPAYLOAD}_{\text{CREATEPLT}}}$$

</div>

UPDATEPAYLOAD$_{\text{PROTOCOL}}$	variable length

A protocol update.

payloadType $= 1 :$ WORD8

length : WORD64

Length of the rest of the payload.

messageLength : WORD64

Length of message.

message : BYTES(messageLength)

A brief message about the update.

urlLength : WORD64

Length of URL.

url : BYTES(urlLength)

A URL of a document describing the update.

hash : SHA256

SHA256 hash of the specification document.

aux : BYTES(length − 8 − messageLength − 8 − urlLength − 32)

Auxiliary data whose interpretation is defined by the new specification.

UPDATEPAYLOAD$_{\text{ELECTIONDIFFICULTY}}$	5 bytes

An election difficulty update. Protocol versions 1–5.

payloadType = 2 : WORD8

electionDifficulty : ELECTIONDIFFICULTY

UPDATEPAYLOAD$_{\text{EUROPERENERGY}}$	17 bytes

A euro-per-energy parameter update.

payloadType = 3 : WORD8

euroPerEnergy : EXCHANGERATE

UPDATEPAYLOAD$_{\text{MICROCCDPEREURO}}$	17 bytes

A microCCD-per-euro parameter update.

payloadType = 4 : WORD8

microCCDPerEuro : EXCHANGERATE

UPDATEPAYLOAD$_{\text{FOUNDATIONACCOUNT}}$	33 bytes

An foundation account update.

payloadType = 5 : WORD8

account : ACCOUNTADDRESS

UPDATEPAYLOAD$_{\text{MINTDISTRIBUTIONV0}}$	14 bytes

A mint distribution update. Protocol versions 1–3. This is superseded by UPDATEPAYLOAD-MINTDISTRIBUTIONV1 from protocol version 4.

payloadType = 6 : WORD8

mintPerSlot : MINTRATE

bakingReward : AMOUNTFRACTION
finalizationReward : AMOUNTFRACTION

UPDATEPAYLOAD~TransactionFeeDistribution~	9 bytes

The distribution of block transaction fees among the block baker, the GAS account, and the foundation account.

payloadType = 7 : WORD8
baker : AMOUNTFRACTION
gas : AMOUNTFRACTION

UPDATEPAYLOAD~GASRewardsV0~	17 bytes

A GAS rewards update. Protocol versions 1–5. This is superseded by UPDATEPAYLOAD-GASREWARDSV1 from protocol version 6.

payloadType = 8 : WORD8
baker : AMOUNTFRACTION
finalizationProof : AMOUNTFRACTION
accountCreation : AMOUNTFRACTION
chainUpdate : AMOUNTFRACTION

UPDATEPAYLOAD~BakerStakeThreshold~	9 bytes

A baker stake threshold update. Protocol versions 1–3. This is superseded by UPDATEPAYLOAD-POOLPARAMETERS from protocol version 4.

payloadType = 9 : WORD8
amount : AMOUNT

UPDATEPAYLOAD~RootUpdate~	variable length

A root update.

payloadType = 10 : WORD8
update : ROOTUPDATE

UPDATEPAYLOAD~Level1~	variable length

A Level1 update.

payloadType = 11 : WORD8
update : LEVEL1UPDATE

UPDATEPAYLOAD_{AddAnonymityRevoker}	variable length

Add a privacy guardian.

`payloadType = 12 :` WORD8

`arInfo :` ARINFO

UPDATEPAYLOAD_{AddIdentityProvider}	variable length

Add an identity provider.

`payloadType = 13 :` WORD8

`ipInfo :` IPINFO

UPDATEPAYLOAD_{CooldownParameters}	17 bytes

An update to the parameters affecting cooldown periods for validators and delegators, from protocol version 4. From protocol version 7, the distinction in cooldown time between validators and delegators is removed and the lowest of the two values is used.

`payloadType = 14 :` WORD8

`poolOwnerCooldown :` DURATIONSECONDS

Number of seconds that pool owners (i.e. validators) must cooldown when reducing their equity capital or closing the pool.

`delegatorCooldown :` DURATIONSECONDS

Number of seconds that delegators must cooldown when reducing their delegated stake.

UPDATEPAYLOAD_{PoolParameters}	59 bytes

An update to the parameters affecting validator pools, from protocol version 4.

`payloadType = 15 :` WORD8

`passiveCommissions :` COMMISSIONRATES

Commission rates charged for passive delegation.

`commissionBounds :` COMMISSIONRANGES

Bounds on the commission rates that may be charged by bakers.

`minimumEquityCapital :` AMOUNT

Minimum equity capital required for a new baker.

`capitalBound :` CAPITALBOUND

Maximum fraction of the total staked capital of that a new baker can have.

`leverageBound :` LEVERAGEFACTOR

The maximum leverage that a baker can have as a ratio of total stake to equity capital.

UPDATEPAYLOAD$_{\text{TimeParameters}}$	14 bytes

An update to the parameters defining the reward period length and mint rate per payday, from protocol version 4.

payloadType = 16 : WORD8

rewardPeriod : WORD64

Number of epochs constituting a payday.
$0 <$ rewardPeriod.

mintPerPayday : MINTRATE

Mint rate per payday (as a proportion of the extant supply).

UPDATEPAYLOAD$_{\text{MintDistributionV1}}$	9 bytes

A mint distribution update, from protocol version 4.

payloadType = 17 : WORD8

bakingReward : AMOUNTFRACTION

finalizationReward : AMOUNTFRACTION

UPDATEPAYLOAD$_{\text{TimeoutParameters}}$	41 bytes

An update to the parameters controlling consensus timeouts, from protocol version 6.

payloadType = 18 : WORD8

timeoutBase : DURATION

The base timeout for consensus.

timeoutIncreaseNumerator : WORD64

timeoutIncreaseDenominator : WORD64

The factor by which the timeout is increased, expressed as a ratio.
$0 <$ timeoutIncreaseDenominator $<$ timeoutIncreaseNumerator.
$\gcd($timeoutIncreaseNumerator, timeoutIncreaseDenominator$) = 1$.

timeoutDecreaseNumerator : WORD64

timeoutDecreaseDenominator : WORD64

The factor by which the timeout is decreased, expressed as a ratio.
$0 <$ timeoutDecreaseNumerator $<$ timeoutDecreaseDenominator.
$\gcd($timeoutDecreaseNumerator, timeoutDecreaseDenominator$) = 1$.

UPDATEPAYLOAD$_{\text{MinBlockTime}}$	9 bytes

An update to the minimum time between blocks, from protocol version 6.

payloadType = 19 : WORD8

minBlockTime : DURATION

The minimum time between blocks.

UPDATEPAYLOAD_{BLOCKENERGYLIMIT} ... 9 bytes

An update to the maximum energy that can be consumed by a block, from protocol version 6.

payloadType = 20 : WORD8

blockEnergyLimit : ENERGY

The maximum energy that can be consumed by a block.

UPDATEPAYLOAD_{GASREWARDSV1} ... 13 bytes

A GAS rewards update, from protocol version 6.

payloadType = 21 : WORD8

baker : AMOUNTFRACTION

Fraction of the GAS account awarded for baking a block.

accountCreation : AMOUNTFRACTION

Fraction of the GAS account awarded for including a credential deployment block item.

chainUpdate : AMOUNTFRACTION

Fraction of the GAS account awarded for including a chain update block item.

UPDATEPAYLOAD_{FINALIZATIONCOMMITTEEPARAMETERS} ... 13 bytes

An update to the finalization committee parameters.

payloadType = 22 : WORD8

minFinalizers : WORD32

The minimum number of validators to include in the finalization committee before imposing the relative stake threshold.
minFinalizers > 0.

maxFinalizers : WORD32

The maximum number of validators to include in the finalization committee.
maxFinalizers $\geq$ minFinalizers.

relativeStakeThreshold : AMOUNTFRACTION

The fraction of the total stake required for a validator to be included in the finalization committee.

UPDATEPAYLOAD_{VALIDATORSCOREPARAMETERS} ... 9 bytes

An update to the validator score parameters.

payloadType = 23 : WORD8

maxMissedRounds : WORD64

The maximal number of rounds a validator can miss before it gets suspended.

UPDATEPAYLOAD$_{\textsc{CreatePlt}}$	variable length

Create a new protocol-level token.

payloadType $= 24 :$ WORD8

tokenId : TOKENID

The token id.

tokenModule : SHA256

A SHA256 hash that identifies the token module implementation.

decimals : WORD8

The number of decimal places used in the representation of amounts of this token. This determines the smallest representable fraction of the token.

parameterLength : WORD32

The length of the initialization parameters in bytes.

intializationParameters : BYTES(**parameterLength**)

The initialization parameters of the token, encoded in CBOR.

24.4.5   Transaction Payloads

TRANSACTIONPAYLOAD	variable length

One of the following:

> TRANSACTIONPAYLOAD$_{\textsc{DeployModule}}$ TRANSACTIONPAYLOAD$_{\textsc{InitContract}}$

> TRANSACTIONPAYLOAD$_{\textsc{Update}}$ TRANSACTIONPAYLOAD$_{\textsc{Transfer}}$

> TRANSACTIONPAYLOAD$_{\textsc{UpdateCredentialKeys}}$

> TRANSACTIONPAYLOAD$_{\textsc{TransferToPublic}}$

> TRANSACTIONPAYLOAD$_{\textsc{UpdateCredentials}}$ TRANSACTIONPAYLOAD$_{\textsc{RegisterData}}$

The following payload types are supported in protocol versions 1–3:

> TRANSACTIONPAYLOAD$_{\textsc{AddBaker}}$ TRANSACTIONPAYLOAD$_{\textsc{RemoveBaker}}$

> TRANSACTIONPAYLOAD$_{\textsc{UpdateBakerStake}}$

> TRANSACTIONPAYLOAD$_{\textsc{UpdateBakerRestakeEarnings}}$

> TRANSACTIONPAYLOAD$_{\textsc{UpdateBakerKeys}}$

The following payload types are supported in protocol versions 1–6:

> TRANSACTIONPAYLOAD$_{\textsc{EncryptedAmountTransfer}}$

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{TRANSFERTOENCRYPTED}}}$$

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{TRANSFERWITHSCHEDULE}}}$$

The following payload types are supported in protocol version 2 onwards:

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{TRANSFERWITHMEMO}}}$$

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{TRANSFERWITHSCHEDULEANDMEMO}}}$$

The following payload types are supported in protocol versions 2–6:

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{ENCRYPTEDAMOUNTTRANSFERWITHMEMO}}}$$

The following payload types are supported in protocol version 4 onwards:

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{CONFIGUREVALIDATOR}}}$$

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{CONFIGUREDELEGATION}}}$$

The following payload types are supported in protocol version 9 onwards:

$$\boxed{\text{TRANSACTIONPAYLOAD}_{\text{TOKENUPDATE}}}$$

TRANSACTIONPAYLOAD$_{\text{DEPLOYMODULE}}$

Deploys a code module.

$\texttt{payloadType} = 0 : \text{WORD8}$

$\texttt{module} : \text{MODULE}$

TRANSACTIONPAYLOAD$_{\text{INITCONTRACT}}$

Initializes a new smart contract instance.

$\texttt{payloadType} = 1 : \text{WORD8}$

$\texttt{amount} : \text{AMOUNT}$

$\texttt{moduleRef} : \text{MODULEREF}$

$\texttt{initName} : \text{INITNAME}$

$\texttt{parameter} : \text{PARAMETER}$

TRANSACTIONPAYLOAD$_{\text{UPDATE}}$

Invokes a smart contract instance.

$\texttt{payloadType} = 2 : \text{WORD8}$

$\texttt{amount} : \text{AMOUNT}$

$\texttt{address} : \text{CONTRACTADDRESS}$

receiveName : RECEIVENAME	
message : PARAMETER	

TRANSACTIONPAYLOAD$_{\text{TRANSFER}}$	41 bytes

A simple transfer from an account to an account.

payloadType = 3 : WORD8

to : ACCOUNTADDRESS

amount : AMOUNT

TRANSACTIONPAYLOAD$_{\text{ADDBAKER}}$	

Add a baker. (Only supported in protocol versions 1–3. From protocol version 4, this is superseded by TRANSACTIONPAYLOAD$_{\text{CONFIGUREVALIDATOR}}$.)

payloadType = 4 : WORD8

electionVerifyKey : BAKERELECTIONVERIFYKEY

signatureVerifyKey : BAKERSIGNVERIFYKEY

aggregationVerifyKey : BAKERAGGREGATIONVERIFYKEY

proofSig : DLOG25519PROOF

proofElection : DLOG25519PROOF

proofAggregation : BAKERAGGREGATIONPROOF

bakingStake : AMOUNT

restakeEarnings : BOOL

TRANSACTIONPAYLOAD$_{\text{REMOVEBAKER}}$	1 byte

Remove a baker. (Only supported in protocol versions 1–3. From protocol version 4, this is superseded by TRANSACTIONPAYLOAD$_{\text{CONFIGUREVALIDATOR}}$.)

payloadType = 5 : WORD8

TRANSACTIONPAYLOAD$_{\text{UPDATEBAKERSTAKE}}$	9 bytes

Change a baker's stake. (Only supported in protocol versions 1–3. From protocol version 4, this is superseded by TRANSACTIONPAYLOAD$_{\text{CONFIGUREVALIDATOR}}$.)

payloadType = 6 : WORD8

stake : AMOUNT

TRANSACTIONPAYLOAD$_{\text{UPDATEBAKERRESTAKEEARNINGS}}$	2 bytes

Change whether a baker's earnings are restaked. (Only supported in protocol versions 1–3. From protocol version 4, this is superseded by TRANSACTIONPAYLOAD$_{\text{CONFIGUREVALIDATOR}}$.)

payloadType $= 7 :$ WORD8

restakeEarnings $:$ BOOL

TRANSACTIONPAYLOAD$_{\text{UPDATEBAKERKEYS}}$

Update baker keys. (Only supported in protocol versions 1–3. From protocol version 4, this is superseded by TRANSACTIONPAYLOAD$_{\text{CONFIGUREVALIDATOR}}$.)

payloadType $= 8 :$ WORD8

electionVerifyKey $:$ BAKERELECTIONVERIFYKEY

signatureVerifyKey $:$ BAKERSIGNVERIFYKEY

aggregationVerifyKey $:$ BAKERAGGREGATIONVERIFYKEY

proofSig $:$ DLOG25519PROOF

proofElection $:$ DLOG25519PROOF

proofAggregation $:$ BAKERAGGREGATIONPROOF

TRANSACTIONPAYLOAD$_{\text{UPDATECREDENTIALKEYS}}$

New set of credential keys to be replaced with the existing ones, including updating the threshold.

payloadType $= 13 :$ WORD8

credId $:$ CREDENTIALREGISTRATIONID

keys $:$ CREDENTIALPUBLICKEYS

TRANSACTIONPAYLOAD$_{\text{ENCRYPTEDAMOUNTTRANSFER}}$

Send an encrypted amount to an account. (Only supported in protocol versions 1–6.)

payloadType $= 16 :$ WORD8

to $:$ ACCOUNTADDRESS

data $:$ ENCRYPTEDAMOUNTTRANSFERDATA

Encrypted amount and proof that it is done correctly.

TRANSACTIONPAYLOAD$_{\text{TRANSFERTOENCRYPTED}}$ 9 bytes

Transfer some amount from public to encrypted balance. (Only supported in protocol versions 1–6.)

payloadType $= 17 :$ WORD8

amount $:$ AMOUNT

TRANSACTIONPAYLOAD_{TRANSFERTOPUBLIC}

Decrypt a portion of the encrypted balance.

payloadType = 18 : WORD8

data : SECTOPUBAMOUNTTRANSFERDATA

How much to transfer and proof that remaining encrypted amount is correct.

TRANSACTIONPAYLOAD_{TRANSFERWITHSCHEDULE}   variable length

Send a transfer with an attached schedule

payloadType = 19 : WORD8

to : ACCOUNTADDRESS

length : WORD8

Number of scheduled transfers.

schedule : length $\times$ (TIMESTAMP, AMOUNT)

List of scheduled transfers.

TRANSACTIONPAYLOAD_{UPDATECREDENTIALS}   variable length

Update the account threshold and the credentials linked to an account by adding or removing credentials. The credential with index 0 can never be removed.

payloadType = 20 : WORD8

cdiLength : WORD8

Number of new credentials

newCredInfos : cdiLength $\times$ (WORD8, CREDENTIALDEPLOYMENTINFORMATION)

Indices and deployment information of the new credentials.

removeLength : WORD8

Number of credentials to be removed.

removeCredIds : removeLength $\times$ CREDENTIALREGISTRATIONID

The Credential IDs of the credentials to be removed.

newThreshold : ACCOUNTTHRESHOLD

TRANSACTIONPAYLOAD_{REGISTERDATA}   variable length

Register data on the chain.

payloadType = 21 : WORD8

data : SHORTBYTES

TRANSACTIONPAYLOAD_{TRANSFERWITHMEMO}   variable length

Send an amount to an account with a memo. (Only supported from protocol version 2 onwards.)

payloadType $= 22$: WORD8
to : ACCOUNTADDRESS
memo : MEMO
amount : AMOUNT

TRANSACTIONPAYLOAD$_{\text{ENCRYPTEDAMOUNTTRANSFERWITHMEMO}}$	variable length

Send an encrypted amount to an account with a memo. (Only supported in protocol versions 2–6.)

payloadType $= 23$: WORD8
to : ACCOUNTADDRESS
memo : MEMO
data : ENCRYPTEDAMOUNTTRANSFERDATA

Encrypted amount and proof that it is done correctly.

TRANSACTIONPAYLOAD$_{\text{TRANSFERWITHSCHEDULEANDMEMO}}$	variable length

Send an amount to an account with a schedule and a memo.

payloadType $= 24$: WORD8
to : ACCOUNTADDRESS
memo : MEMO
length : WORD8

Number of scheduled transfers.

schedule : length $\times$ (TIMESTAMP, AMOUNT)

List of scheduled transfers.

TRANSACTIONPAYLOAD$_{\text{CONFIGUREVALIDATOR}}$	variable length

Configure a validator.

payloadType $= 25$: WORD8
bitmap : CONFIGUREVALIDATORBITMAP

Which fields of the validator are being configured.

capital : bitmap.hasCapital $\times$ AMOUNT

The equity capital of the validator. If this is set to 0, the validator is removed.

restakeEarnings : bitmap.hasRestakeEarnings $\times$ BOOL

Whether the validator's earnings are restaked.

openForDelegation : bitmap.hasOpenForDelegation $\times$ OPENSTATUS

Whether the pool is open for delegators.

keysWithProofs : bitmap.hasKeysWithProofs $\times$ VALIDATORKEYSWITHPROOFS

The key/proof pairs to verify the validator.

metadataURL : bitmap.hasMetadataURL × URLTEXT

The URL referencing the validator's metadata.

transactionFeeCommission :
    bitmap.hasTransactionFeeCommission × AMOUNTFRACTION

The commission the pool owner takes on transaction fees.

bakingRewardCommission : bitmap.hasBakingRewardCommission × AMOUNTFRACTION

The commission the pool owner takes on baking rewards.

finalizationRewardCommission :
    bitmap.hasFinalizationRewardCommission × AMOUNTFRACTION

The commission the pool owner takes on finalization rewards.

suspended : bitmap.hasSuspended × BOOL

Whether the validator is suspended. This is only supported from protocol version 8 onwards.

CONFIGUREVALIDATORBITMAP	2 bytes

A bitmap indicating which fields of a validator are being configured.

$unused = (0, 0, 0, 0, 0, 0, 0) : 7 \times$ BIT

Unused. Reserved for future use.

hasSuspended : BIT

Supported from protocol version 8 onwards. Otherwise must be 0.

hasFinalizationRewardCommission : BIT

hasBakingRewardCommission : BIT

hasTransactionFeeCommission : BIT

hasMetadataURL : BIT

hasKeysWithProofs : BIT

hasOpenForDelegation : BIT

hasRestakeEarnings : BIT

hasCapital : BIT

TRANSACTIONPAYLOAD_{CONFIGUREDELEGATION}	variable length

Configure an account for delegation.

$payloadType = 26 :$ WORD8

bitmap : CONFIGUREDELEGATIONBITMAP

Which fields of the delegation are being configured.

capital : bitmap.hasCapital × AMOUNT

The staked capital to be delegated. If this is set to 0, the delegator is removed.

restakeEarnings : bitmap.hasRestakeEarnings × BOOL

Whether the delegator's earnings are restaked.

delegationTarget : bitmap.hasDelegationTarget × DELEGATIONTARGET

Whether the pool is open for delegators.

CONFIGUREDELEGATIONBITMAP   2 bytes

A bitmap indicating which fields of a delegator are being configured.

$unused = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) : 13 \times$ BIT

Unused. Reserved for future use.

hasDelegationTarget : BIT

hasRestakeEarnings : BIT

hasCapital : BIT

TRANSACTIONPAYLOAD$_{\text{TOKENUPDATE}}$ variable length

A token holder transaction.

payloadType $= 27$: WORD8

tokenId : TOKENID

Identifier of the token type to which the transaction refers.

operationsLength : WORD32

Length of the encoded operations in bytes.

operations : BYTES(operationsLength)

The CBOR-encoded operations to perform.
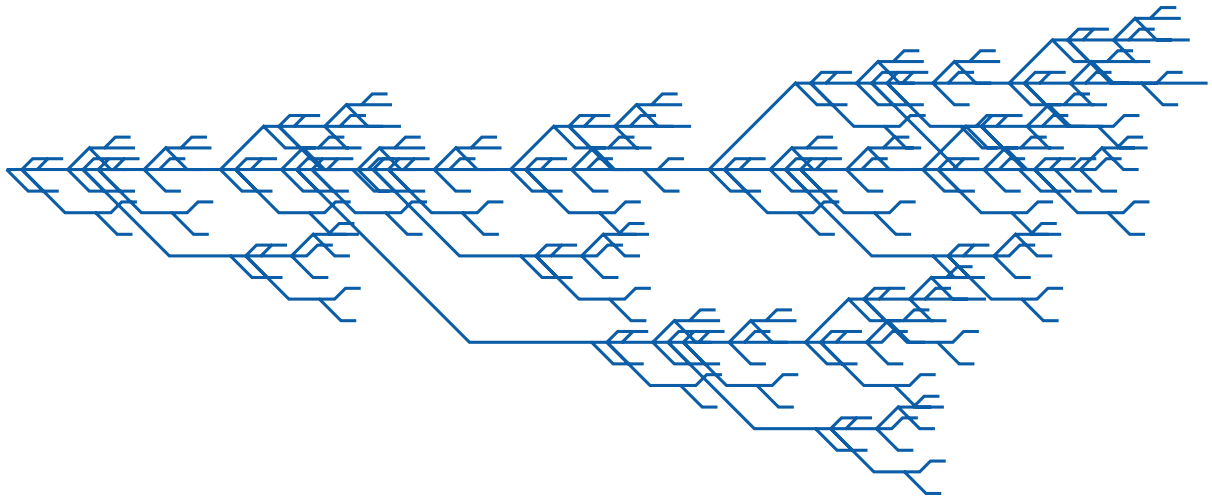
Part VII

Initial Konsensus

Appendix A

Konsensus Preliminaries

A.1 Participants

We work in the *permissionless* setting where the number of parties is arbitrary. Any parties can access the data stored in Konsensus. The consensus on the stored data is maintained by special parties, called bakers. Bakers also ensure that new data is properly added. More information on bakers is found in Section C.1.

Definition 111. A *party* is called *live* if it is connected to the network, up-to-date on the Konsensus data (in the form of a block tree), and awake (that is participates in the protocol). A party is called *honest* if it follows the protocol instructions. A party is called *parat* if it is live and honest.

A.2 Time and Synchrony

We assume that there is a notion of global physical time t. We assume that each party P_i has a local clock $\texttt{Clock}_i$. Likewise, we assume that there is a bound $\texttt{MaxDrift}$ on clock drift. Specifically we assume that $|t - \texttt{Clock}_i| < \frac{\texttt{MaxDrift}}{2}$. This implies that the clocks of any two parat parties P_i and P_j satisfy $|\texttt{Clock}_i - \texttt{Clock}_j| < \texttt{MaxDrift}$.

In Konsensus time is subdivided into *slots* (in other work also known as rounds or ticks). We assume that the local clocks of parties are accurate and synchronous enough that parties have a rough agreement on the current slot. The slot duration is therefore a tweakable parameter of the protocol and is to be determined by experimentation. For later it is planed that the control layer can dynamically adapt this parameter.

Definition 112. A *slot number* $\texttt{slot}$ is the index of a slot, i.e. $\texttt{slot} \in \textsc{slots} \coloneqq \mathbb{N}$.

> ⚠ The bound $\texttt{MaxDrift}$ is not known to honest parties and is *not* used in the protocol. However, the security guarantees of the consensus layer *do* depend on $\texttt{MaxDrift}$ (and its relation to $\texttt{Kontrol.SlotDuration}$).

A.3 Communication Network

We assume that parties have access to a *diffusion network* which allows them to multicast messages. We assume that parties can join or leave the network at any point in time. If a party is parat when a message is sent and stays parat long enough, then it is guaranteed to get that message. The ideal network functionality works as follows:

- Parties can register or deregister. Every registered party P_i has an interface which allows it to send or receive messages. The functionality maintains a message buffer $\mathtt{msgbuffer}_i$ for each registered party P_i.

- The network is parametrized by $\Delta_{\mathtt{net}}$ which is the bound on message delivery in seconds. Initially, $\Delta_{\mathtt{net}} = 10$. The environment has an interface where it once can set $\Delta_{\mathtt{net}}$ to a value > 0 as long as no messages have been sent so far.

- If a party P_i inputs a message m, the message is added to the message buffers of all currently registered parties.

- The environment has an interface where it can instruct the functionality to deliver any message $m \in \mathtt{msgbuffer}_j$ to party P_j. The delivered message is output at the interface of P_j and removed from $\mathtt{msgbuffer}_j$.

- **Bounded delay for honest messages:** Let m be a message from honest sender P_i. The functionality will automatically deliver m to any honest P_j which has m in its message buffer $\Delta_{\mathtt{net}}$ seconds after P_i has input m.

- **Bounded delay for dishonest messages:** Let m be a message from (dishonest) sender P_i which gets delivered to honest party P_j. The functionality will automatically deliver m to any honest party P_k which has m in its message buffer $\Delta_{\mathtt{net}}$ seconds after P_j got the message.

> **ⓘ** Right now we assume that all parties eventually receive all messages ever sent. This seems to indicate using TTL $= \infty$ in the network layer. This is not the intent. The assumption is merely a simplification which is made for conciseness of description. In an actual implementation it will be enough that if an up-to-date party acts on a message, then it makes sure that all up-to-date parties receive the message. For instance, on adding a new block to the tree, make sure to flood it, even if TTL just ran out.

> **⚠** The bound $\Delta_{\mathtt{net}}$ is not known to honest parties and is *not* used in the protocol (formally we are in the semi-synchronous model). However, the security guarantees of the consensus layer *do* depend on $\Delta_{\mathtt{net}}$.

The value of $\Delta_{\mathtt{net}}$ follows from how the network layer is implemented. It is not a tweak, but rather a measurable property. The initial design, however, will be targeted towards what this value is in practice. Our starting value so far is 10 seconds, but experimentation is needed.

> **ⓘ** In later versions, we will relax the assumptions on the network delay. For example, we could require that only 95% of all honest parties must receive a message within $\Delta_{\mathtt{net}}$ or that this bound holds during 95% of all slots. We would still assume that any honest parties *will* eventually receive the message. What exact requirements are needed will crystallize when we finish the full formal proofs of security.

In particular, the value $\Delta_{\mathtt{net}}$ has influence on the speed (i.e., number of blocks per second) of the consensus protocol. In future versions parties could measure the network delay empirically and Kontrol could update its parameters to fit the current $\Delta_{\mathtt{net}}$.

Appendix B

Skov Data-Structure

The Skov layer maintains the data structure for the Konsensus layer. In particular, it contains the block tree and the list of finalization entries.

B.1 Data Structure

In this section we describe the data structure of Skov.

B.1.1 Blockchain and Block Tree

The goal of Konsensus is to maintain a sequence of data blocks. These data blocks are encoded in the form of a blockchain which is a linked list of blocks. Each block contains a pointer to the previous block, information on its creation, and the block data.

Definition 113. A *block*, $\texttt{block} \in \text{BLOCKS}$, is a tuple consisting of

$\texttt{slot} \in \text{SLOTS} = \mathbb{N}$
> The *slot number* at which the block was created.

$\texttt{ptr} \in \text{HASH}$
> The *block pointer* is the hash of the previous block in the blockchain.

$\texttt{bid} \in \text{BAKERIDS}$
> A pointer to the *baker* which created the block, cf. Section C.1.

$\texttt{proof} \in \text{BLOCKPROOFS}$
> The *block proof* allows to check that the baker was allowed to create the block. Cf. Section C.3.

$\texttt{nonce} \in \text{BLOCKNONCES}$
> The *block nonce* is a random string $\texttt{val}$, with a proof $\texttt{prf}$ of correct generation, cf. Section C.3. The block nonce is used to generated randomness on the blockchain.

$\texttt{nextFinEntry} \in \text{FINALENTRIES} \cup \{\bot\}$
> If a *finalization entry* for the last finalized block (the baker saw when creating the block) is already contained in a predecessor of this block, then $\texttt{nextFinEntry} = \bot$. Otherwise, $\texttt{nextFinEntry}$ is the finalization entry with sequence number one more than the last finalization entry contained in a preceding block. See Section D for details on finalization.

$\texttt{data} \in \text{DATA}^*$
> The *block data* is the actual data stored in this block.

$\sigma^{\texttt{BC}} \in \text{SIGNATURES}$
    The signature is the signature of the baker `baker` on the block, i.e. on all the values above.

> **ℹ** In the implementation the block size is limited. It is part of engineering to find the right size of blocks. Intuitively, smaller blocks propagate faster over the network while bigger blocks allow adding more data per time unit.

> **</>** In the implementation the block size is limited by the amount of NRG (a measure for space and computational complexity). NRG is similar to Ethereum's gas.

> **!** As described here, a block can contain at most one finalization entry. More generally, it could be allowed to have several finalization entries in a block. That may help in situations where blocks are finalized very quickly and the entries in the chain are lagging behind (e.g., due to malicious bakers not including them).

The blockchain starts with the genesis block `genesis`. This unique block contains the initial system parameters of the Konsensus layer. In particular, it defines the initial stakeholders, i.e., the *bakers*, and *finalizers*.

Definition 114. The *genesis block*, `genesis` $\in$ BLOCKS, is a special block with the following values:

`slot := 0`
    The slot number is set to 0. All other blocks have larger slot numbers.

`ptr := ⊥`
    The genesis block has no parent.

`bid := ⊥`
    The genesis block was not created by a baker.

`proof := ⊥`
    The genesis block has no block proof.

`nonce := ⊥`
    The genesis block has no block nonce.

`nextFinEntry := genesisFE`
    The finalization entry for the genesis block itself, see Definition 127.

`data := genesisData`
    Details on the genesis data `genesisData` is given in Section E.

$\sigma^{\texttt{BC}} := \bot$
    The genesis block has no signature.

A blockchain is a linked list of blocks where the first block must be the genesis block `genesis`.

Definition 115. A *blockchain*, `chain` $\in$ CHAINS $:=$ BLOCKS*, is a sequence of blocks starting with the genesis block, i.e., $\texttt{chain} = \texttt{block}_0, \dots, \texttt{block}_n$, where the first block is the genesis block $\texttt{block}_0 = \texttt{genesis}$.

For a chain $\texttt{chain} = \texttt{block}_0, \dots, \texttt{block}_n$, we use the following notation and terminology:

- The *chain data* is the sequence of the block data, i.e., `chain.chainData` $= ($`block`$_0$`.data`, $\ldots,$ `block`$_n$`.data`$)$

- The *head* of the chain is the last block in the chain, i.e., `chain.head = block`$_n$.

- The *depth* of a block is its index in the blockchain, i.e., `depth(block`$_i$`)` $= i$. Note that the genesis block has depth 0.

> ℹ The chain is implicit in the definition of the depth of a block and assumed to be clear from context.

A block tree is a collection of blockchains which all start with the same genesis block.

Definition 116. A *block tree* `tree` $\in$ TREES is a tree of blocks with root `genesis` where every path from the root to a leaf forms a blockchain.

A block `block` can be added to `Skov.tree` if it points to a block in the tree and allows for a valid extension, i.e., if there exists `block`$'$ $\in$ `tree` with $\mathsf{H}($`block`$'$$) =$ `block.ptr` and $\mathsf{Kontrol.isValidChain}($`Skov.getChain`$($`tree`$,$ `block`$'$$)||$`block`$) =$ `true`.

B.1.2 Skov Data

In the following, we define the data structures maintained by Skov.

Definition 117. The Skov data structures are the following.

`Skov.tree` $\in$ TREES
  The *block tree* maintained by Skov. It initially contains the genesis block `genesis`.

`Skov.finEntries` $\in$ FINALENTRIES*
  The *finalization list* is a list of finalization entries. It initially contains the finalization entry for the genesis block `genesis`.

`Skov.blockPool` $\in$ BLOCKS*
  The *block pool* is a list of pending (valid) blocks which have not yet been added to `Skov.tree`. Initially `Skov.blockPool` is empty.

`Skov.finalentryPool` $\in$ FINALENTRIES*
  The *finalization pool* is a list of pending finalization entries which have not yet been added to `Skov.finEntries`. Initially `Skov.finalentryPool` is empty.

`Skov.inputPool` $\in$ INPUTDATA*
  The *input pool* is a list of pending input data which has not yet been added to blocks in `Skov.tree`.

> ℹ We explicitly distinguish between input data and (the processed) data which is in a block. For instance, the input data could be a transaction and the corresponding data (in a block) is the resulting update of the involved account (balances).

B.2 Interface and Guarantees

In this Section we describe the interfaces and guarantees provided by Skov. The interfaces of Skov allow to read the current data sequence or to provide new input in the form of blocks, finalization

entries, or input data. It is guaranteed that Skov incorporates new blocks or finalization entries periodically.

B.2.1 Data Output

The function Skov.getData returns the data sequence currently represented by Skov.tree.

Function Skov.getData

Inputs

Skov.tree ∈ TREES
> The block tree maintained by Skov.

Outputs

data ∈ DATA*
> The data is the *full* data sequence of the current best chain in Skov.tree where the best chain is selected using Kontrol.BestChain (cf. Section E.6).

---

❶ The above function returns the full data sequence of the best chain. Depending on the application, one needs to take the 'final' prefix of this sequence, i.e. the part of the sequence that will no longer change. This can be done, by going back to the latest final block (or a fixed number of blocks back if finalization is lagging behind). The exact pruning rule depends on the application.

B.2.2 Add Block

The function Skov.addBlock adds a block to Skov.blockPool, the pool of pending blocks. Then it adds all possible blocks from Skov.blockPool to Skov.tree.

Function Skov.addBlock

Inputs

block ∈ BLOCKS
> The block to be added to the block pool.

Skov.blockPool ∈ BLOCKS*
> The block pool maintained by Skov.

Skov.tree ∈ TREES
> The block tree maintained by Skov.

Outputs

Skov.blockPool ∈ BLOCKS*
> The updated block pool.

Skov.tree ∈ TREES
> The updated tree.

Promise

> The input block is either in the updated Skov.blockPool or the updated Skov.tree. The updated Skov.blockPool does not contain any block which can be added to the

updated `Skov.tree`. All blocks removed from `Skov.blockPool` have been added to `Skov.tree`.

This function must not add blocks with future slots to `Skov.tree`.

⚠️ Blocks with future slot numbers *MUST NOT* be added to the block tree. Otherwise, safety of Birk consensus might be compromised.

ℹ️ The promise of `addBlock` guarantees that a block stays in `Skov.blockPool` until it can be added to `Skov.tree`. However, the function could in principle also remove invalid blocks (including the input block) from `Skov.blockPool`, i.e., blocks which can never be added to `Skov.tree`. Note that the removal of invalid blocks is *not* required for security, but will increase the efficiency of the protocol and does *not* harm the security.

B.2.3 Add Finalization Entry

The function `Skov.addFinEntry` adds a finalization entry to `Skov.finalentryPool`, the pool of pending finalization entries. Then it adds all possible blocks from `Skov.finalentryPool` to `Skov.finEntries`. A finalization entry `finEntry` can be added to `Skov.finEntries` once the block `block` with `ptr = H(block)` is part of `Skov.tree`, and `finEntry` is a valid proof of finalization. The validity of `finEntry` is defined in Section D.2.1.

Function Skov.addFinEntry

Inputs

`finEntry` ∈ FINALENTRIES
 The finalization entry to be added to the finalization pool.

`Skov.finalentryPool` ∈ FINALENTRIES*
 The final pool maintained by Skov.

`Skov.finEntries` ∈ FINALENTRIES*
 The finalization entry list maintained by Skov.

Outputs

`Skov.finalentryPool` ∈ FINALENTRIES*
 The updated final pool.

`Skov.finEntries` ∈ FINALENTRIES*
 The updated finalization entry list.

Promise

 The input `finEntry` is either in the updated `Skov.finalentryPool` or the updated `Skov.finEntries`. The updated `Skov.finalentryPool` does not contain any entry which can be added to the updated `Skov.finEntries`. All entries removed from `Skov.finalentryPool` have been added to `Skov.finEntries`.

> ℹ The function `addFinEntry` could filter out entries which will never be added to Skov.`finalentryPool` (e.g., as they are invalid). This can include the input `finEntry`. Note that the filtering is *not* required for security, but will increase the efficiency of the protocol.

B.2.4 Add Input Data

The function Skov.`addInputs` adds input data to Skov.`inputPool`, the pool of input data.

Function Skov.addInputs

Inputs

  `inputData` ∈ INPUTDATA
      The input data to be added to the input pool.

  Skov.`inputPool` ∈ INPUTDATA*
      The input pool maintained by Skov.

Outputs

  Skov.`inputPool` ∈ INPUTDATA*
      The input pool with newly added `inputData`.

> ℹ The function `addInputs` could filter out entries which will under no circumstances be added to Skov.`inputPool` (e.g., as they are invalid). Note that the filtering is *not* required for security, but will increase the efficiency of the protocol.

B.2.5 Get Chain

The function Skov.`getChain` returns the chain from `genesis` up to the input block.

Function Skov.getChain

Inputs

  Skov.`tree` ∈ TREES
      The block tree maintained by Skov.

  `block` ∈ BLOCKS
      A block in Skov.`tree`.

Outputs

  `chain` ∈ CHAINS
      The chain from `genesis` up to `block`.

B.3 Required Interfaces

The Skov layer requires the following functions from the Kontrol layer (cf. Section E).

Chain Validation: The function Kontrol.isValidChain allows checking if a chain `chain` is valid with respect to tree Skov.tree and finalization entries Skov.finEntries. The function is specified in Section E.5.

Best Chain Rule: The function Kontrol.BestChain returns the best chain given tree Skov.tree and finalization entries Skov.finEntries. The function is specified in Section E.6.

B.4 Example Implementation

In this Section we provide pseudocode for the implementation of the interfaces defined in Section B.2.

> ❗ This code is not optimized for production. It only serves as an example.

Function Skov.getData(Skov.tree)

Implementation

1: **Return** Kontrol.BestChain(Skov.tree, Skov.finEntries).chainData

---

Function Skov.addBlock(block, Skov.blockPool, Skov.tree)

Implementation

1: Add `block` to Skov.blockPool and sort blocks in Skov.blockPool ascending according to slot numbers.
2: **for** $i = 0$ **to** $|\text{Skov.blockPool}| - 1$ **do**
3:     $\text{block}_i = \text{blockPool}[i]$
4:     **if** exists a `block` in Skov.tree with $\text{block}_i.\text{ptr} = \mathsf{H}(\text{block})$ **then**
5:        Let $\text{chain} = \text{Skov.getChain}(\text{tree}, \text{block}) \| \text{block}_i$.
6:        **if** Kontrol.isValidChain(Skov.tree, Skov.finEntries, chain) = true **then**
7:           Add block_i to Skov.tree and remove it from Skov.blockPool.
8: **Return** (Skov.tree, Skov.blockPool)

Function Skov.addFinEntry(finEntry, Skov.finalentryPool)

Implementation

1: Add `finEntry` to Skov.finalentryPool and sort entries in Skov.finalentryPool ascending according to counter.
2: **for** each `finEntry` in Skov.finalentryPool **do**
3:     **if** there exists $\text{block} \in \text{tree}$ with $\mathsf{H}(\text{block}) = \text{finEntry.ptr}$ **then**
4:        **if** Afgjort.verifyFinalProof(Skov.tree, Skov.finEntries, finEntry) = true **then**
5:           Add the finalization entry `finEntry` to Skov.finEntries and remove it from Skov.finalentryPool.
6: **Return** (Skov.finalentryPool, Skov.finEntries)

Function Skov.addInputs(`inputData`, Skov.`inputPool`)

Implementation

1: Add `inputData` to Skov.`inputPool`.
2: **Return** Skov.`inputPool`

B.5 Catchup

The synchronization process allows a fresh party to become parat. After the synchronization, the party should have (almost) the same tree as any other parat party.

The following is a rough idea of how synchronization could be implemented.

- A fresh party knows the genesis block `genesis` and some recent finalized block `block` (including a finalization proof). The idea is that the fresh party gets this information from a (parat) entity it trusts.

- The party checks that its network peers are on the same block tree (by asking for H(`genesis`)).

- The party starts Skov from `block` and collects new blocks and new finalization proofs.

- The party then asks peers with the same H(`genesis`) for the chain (or parts of it) up to `block` (including finalization proofs).

- The party also ask those peers for the chain part from `block` to their last finalized block (including finalization proofs). It repeats this process as long as it finds peers with a last finalized block at greater depth than the last finalized block in Skov.`finEntries`.

> </> In the implementation, a party periodically asks its peers for their newest, finalized block. If any peer report a newer block, the party will ask this peer for the necessary blocks to catch-up.

Appendix C

Birk Consensus

The Birk layer provides a resilient form of consensus. The protocol has similarities to Bitcoin consensus. But in contrast to the Bitcoin protocol, Birk consensus is based on proof-of-stake (instead of proof-of-work). The core idea in Birk consensus is that bakers participate in a lottery (with ticket count depending on stake) where the winner can propose a new block. The Birk layer uses the data stored in the Skov.

C.1 Bakers

The consensus of Birk is run by parties called *bakers*. A baker is identified by a unique ID and has a public-lottery power assigned.

Definition 118. A *baker*, $\mathtt{baker} \in \mathrm{BAKERS}$, is an entity and has the following values assigned. These values are publicly known and stored on the blockchain.

Public Values

$\mathtt{bid} \in \mathrm{BAKERIDS}$
> The unique *baker identifier*.

$\mathtt{lotteryPower} \in \mathrm{LOTTERYPOWER}$
> The *lottery power*.

$\mathtt{vk}^{\mathrm{LE}} \in \mathrm{VERIFICATIONKEY}_{\mathrm{LE}} = \mathrm{PUBLICKEY}_{\mathrm{VRF}}$
> The *leader election verification key*, a VRF verification key, cf. Section 6.2.

$\mathtt{vk}^{\mathrm{BC}} \in \mathrm{VERIFICATIONKEYS}_{\mathrm{BC}}$
> The *block signature verification key*, cf. Section 5.1.3.

These values are only known to the baker itself.

Private Values

$\mathtt{sk}^{\mathrm{LE}} \in \mathrm{PRIVATEKEY}_{\mathrm{LE}} = \mathrm{PRIVATEKEY}_{\mathrm{VRF}}$
> The *leader election private key*, a VRF secret key, cf. Section 6.2.

$\mathtt{sk}^{\mathrm{BC}} \in \mathrm{SIGNKEYS}_{\mathrm{BC}}$
> The *block signature signing key*, cf. Section 5.1.3.

The Konsensus layer only requires that baker-IDs are unique. Otherwise, the IDs can be arbitrary bit strings. In the implementation, a baker-id could be a public-key which is assigned with an identity or an account.

In the implementation, baker are assigned an integer as ID. Each baker is also associated with an account.

The set of bakers for a specific slot `slot` can be computed from Skov using the function Kontrol.getBakerSet from the Kontrol layer (see Section E.3).

C.2   Required Interfaces

C.2.1   Kontrol Interfaces

The Birk layer requires the following functions from Kontrol (cf. Section E).

Best Chain Rule: The function Kontrol.BestChain returns the best chain given tree `Skov.tree` and finalization entries `Skov.finEntries`. The function is specified in Section E.6.

Birk Parameters: The functions Kontrol.getLEDifficulty, Kontrol.getEpochLength, Kontrol.getBakerSet, and Kontrol.getLENonce take as input a slot `slot` and a chain `chain`. The functions return the parameters for Birk for that slot and that chain. They are specified in Section E.3.

C.2.2   Input Data Processing

We distinguish between input data (of type INPUTDATA) and block data (of type DATA). Bakers collect input data in `Skov.inputPool`. This input data is then processed to get the data for a new block. The exact selection of data for a new block depends on the application. We therefore assume that Birk has access to the function processInputs.

Function processInputs

Inputs

$\quad$ `chain` $\in$ CHAINS
$\qquad$ The chain which should be extended.
$\quad$ `slot` $\in$ SLOTS
$\qquad$ The slot of the to-be-created block.
$\quad$ `Skov.finEntries` $\in$ FINALENTRIES*
$\qquad$ The list of finalization entries maintained by Skov.
$\quad$ `Skov.inputPool` $\in$ INPUTDATA*
$\qquad$ The block input pool maintained by Skov.

Outputs

$\quad$ `data` $\in$ DATA*
$\qquad$ The block data for the to-be-created block.
$\quad$ `Skov.inputPool` $\in$ INPUTDATA*
$\qquad$ The updated input pool.

Promise

$\qquad$ The returned `data` allows to create a block for `slot` which can be validly added to

---

The Konsensus layer only requires that baker-IDs are unique. Otherwise, the IDs can be arbitrary bit strings. In the implementation, a baker-id could be a public-key which is assigned with an identity or an account.

In the implementation, baker are assigned an integer as ID. Each baker is also associated with an account.

The set of bakers for a specific slot `slot` can be computed from Skov using the function Kontrol.getBakerSet from the Kontrol layer (see Section E.3).

C.2   Required Interfaces

C.2.1   Kontrol Interfaces

The Birk layer requires the following functions from Kontrol (cf. Section E).

Best Chain Rule: The function Kontrol.BestChain returns the best chain given tree `Skov.tree` and finalization entries `Skov.finEntries`. The function is specified in Section E.6.

Birk Parameters: The functions Kontrol.getLEDifficulty, Kontrol.getEpochLength, Kontrol.getBakerSet, and Kontrol.getLENonce take as input a slot `slot` and a chain `chain`. The functions return the parameters for Birk for that slot and that chain. They are specified in Section E.3.

C.2.2   Input Data Processing

We distinguish between input data (of type INPUTDATA) and block data (of type DATA). Bakers collect input data in `Skov.inputPool`. This input data is then processed to get the data for a new block. The exact selection of data for a new block depends on the application. We therefore assume that Birk has access to the function processInputs.

Function processInputs

Inputs

$\quad$ `chain` $\in$ CHAINS
$\qquad$ The chain which should be extended.
$\quad$ `slot` $\in$ SLOTS
$\qquad$ The slot of the to-be-created block.
$\quad$ `Skov.finEntries` $\in$ FINALENTRIES*
$\qquad$ The list of finalization entries maintained by Skov.
$\quad$ `Skov.inputPool` $\in$ INPUTDATA*
$\qquad$ The block input pool maintained by Skov.

Outputs

$\quad$ `data` $\in$ DATA*
$\qquad$ The block data for the to-be-created block.
$\quad$ `Skov.inputPool` $\in$ INPUTDATA*
$\qquad$ The updated input pool.

Promise

$\qquad$ The returned `data` allows to create a block for `slot` which can be validly added to

---

> chain. All inputs removed from the updated `Skov.inputPool` have been used to create `data`.

> ⓘ This function abstracts the data selection for new blocks. The implementation of this algorithm highly depends on the actual data layer of the project. The bakers should be incentivized to employ a function `processInputs` which adds as much data as possible (within the limits of the maximum block size). Bitcoin uses transaction fees to achieve this incentive.

C.3 Leader Election

The *leader election* scheme is a stake based lottery where bakers can win the right to propose a new block in a specific slot.

The leader election scheme consists of two parts. First, the `leaderElection` function which allows a baker to (privately) check if they have won the lottery in a specific slot. If a baker has won, the function returns a block proof which the baker can add to their new block. The second part is the `verifyProof` function which allows any party to check the validity of block proofs.

Nonces are used to ensure a random and unpredictable leader election. Nonce are updated periodically using randomness generated from block nonces. The leader election scheme therefore also contains functions to generate and verify block nonces.

C.3.1 Lottery Power and Difficulty

The lottery power of a baker `baker` ∈ BAKERS corresponds to the percentage of lottery tickets a baker holds.

Definition 119. The *lottery power*, `lotteryPower` ∈ LOTTERYPOWER, is a real number between 0 and 1, i.e., LOTTERYPOWER $= [0, 1]$. A lottery power of 1 means that the baker controls all lottery tickets.

> ⓘ In an implementation the lottery power of a baker should roughly correspond to their current stake in the system. The actual translation of stake to lottery power is done at the application layer. Birk just assumes that the list of bakers with their lottery power can be somehow extracted from the blockchain (see Section E.3).

The leader-election difficulty diff^{LE} determines how likely it is to win the lottery.

Definition 120. The *leader-election difficulty*, $\text{diff}^{\text{LE}} \in$ DIFFICULTY, is a real number strictly between 0 and 1, i.e., DIFFICULTY $:= (0, 1)$. A difficulty close to 1 means it is easy to win.

The actual winning probability is computed using the leader-election difficulty function. The *leader-election difficulty function* φ_{LE} is defined as follows.

Function φ_{LE}

Inputs
  `lotteryPower` ∈ LOTTERYPOWER
        The lottery power of a baker.

> $\texttt{diff}^{\mathsf{LE}} \in \text{DIFFICULTY}$
>> The difficulty of the leader election.
>
> **Outputs**
>> $p \in [0, 1]$
>>> The probability of wining the lottery with lottery power $\texttt{lotteryPower}$.
>
> **Promise**
>> The output p is computed as
>>
>> $$p := 1 - (1 - \texttt{diff}^{\mathsf{LE}})^{\texttt{lotteryPower}}.$$

See Section E.3.1 for a discussion on the (initial) value of $\texttt{diff}^{\mathsf{LE}}$.

> ⚠ Note that a higher value of DIFFICULTY means a lower difficulty.

C.3.2 Block Proof, Block Nonce, and Leader Election Nonce

The block proof is used to validate that a block was created rightfully by its baker. It is essentially a VRF proof.

Definition 121. A *block proof*, $\texttt{proof} \in \text{BLOCKPROOFS}$, is a VRF proof, i.e, $\text{BLOCKPROOFS} = \text{PROOF}_{\mathsf{VRF}}$. See Section 6.2 for more details on VRF proofs.

The block nonce is required to generate the randomness which is used to update the leader election nonce.

Definition 122. A *block nonce*, $\texttt{nonce} \in \text{BLOCKNONCES}$, is a tuple consisting of:

$\texttt{val} \in \text{HASH}_{\mathsf{VRF}}$
> The actual block nonce value is a VRF hash, cf. Section 6.2.

$\texttt{prf} \in \text{PROOF}_{\mathsf{VRF}}$
> A proof for the correct generation of $\texttt{val}$ is a VRF proof, cf. Section 6.2.

The leader election nonce is used to make the leader election random and unpredictable.

Definition 123. A *leader election nonce*, $\texttt{nonce}^{\mathsf{LE}} \in \text{NONCE}_{\mathsf{LE}}$, is a bit string of length k, i.e., $\text{NONCE}_{\mathsf{LE}} := \{0, 1\}^k$.

> ℹ The leader election nonce for a specific slot is computed from block nonces using the Kontrol.getLENonce function (see Section E.3.2).

C.3.3 Leader Election Functions

The leader election scheme consists of the following algorithms.

The *leader election* function Birk.leaderElection allows a baker to check with its private key if they are slot leader for a specific slot. It is defined as follows

Function Birk.leaderElection

Inputs

  slot ∈ SLOTS
    The slot where the baker wants to be leader.
  $\mathtt{nonce}^{\mathrm{LE}} \in \mathrm{NONCE}_{\mathrm{LE}}$
    The leader election nonce for slot.
  $\mathtt{diff}^{\mathrm{LE}} \in \mathrm{DIFFICULTY}$
    The difficulty of the leader election for slot.
  lotteryPower ∈ LOTTERYPOWER
    The lottery power of the baker for slot.
  $\mathtt{sk}^{\mathrm{LE}} \in \mathrm{PRIVATEKEY}_{\mathrm{LE}}$
    The leader election secret key of the baker.

Outputs

  proof ∈ BLOCKPROOFS
    The block proof required to create a new block in slot.

Promise

  The function returns a valid block proof if and only if the baker won the right to create a block in slot slot.

Implementation The function is implemented using a VRF scheme, cf. Section 6.2.

1: Compute $h = \mathsf{VRFHash}(\mathtt{sk}^{\mathrm{LE}}, \text{``LE''} \| \mathtt{nonce}^{\mathrm{LE}} \| \mathtt{slot})$.
2: **if** $h < 2^{\ell_{\mathrm{VRF}}} \varphi_{\mathrm{LE}}(\mathtt{lotteryPower}, \mathtt{diff}^{\mathrm{LE}})$ **then**
3:     Return $\mathtt{proof} = \mathsf{VRFProve}(\mathtt{sk}^{\mathrm{LE}}, \text{``LE''} \| \mathtt{nonce}^{\mathrm{LE}} \| \mathtt{slot})$.
4: **else**
5:     Return false.

❗ To prevent attacks on future slot leaders (e.g., denial-of-service attacks), it should be infeasible to check whether a baker is slot leader without knowing a suitable block proof or the private key of the baker.

The *leader election verification* function Birk.verifyProof allows checking if a block proof is valid. It is defined as follows

---

Function Birk.verifyProof

Inputs

  slot ∈ SLOTS
    The slot where the baker wants to be leader.
  $\mathtt{nonce}^{\mathrm{LE}} \in \mathrm{NONCE}_{\mathrm{LE}}$
    The leader election nonce for slot.
  $\mathtt{diff}^{\mathrm{LE}} \in \mathrm{DIFFICULTY}$
    The difficulty of the leader election for slot.
  lotteryPower ∈ LOTTERYPOWER
    The lottery power of the baker for slot.

vk$^{\text{LE}} \in$ VERIFICATIONKEY$_{\text{LE}}$
> The leader election verification key of the baker.

proof $\in$ BLOCKPROOFS
> The leader election verification key of the baker.

Outputs

$b \in$ BOOL
> Indicates if the proof is valid

Promise

> Its guaranteed that verifyProof(slot, nonce$^{\text{LE}}$, diff$^{\text{LE}}$lotteryPower, vk$^{\text{LE}}$, proof) = true if and only if proof = leaderElection(slot, nonce$^{\text{LE}}$, diff$^{\text{LE}}$, lotteryPower, sk$^{\text{LE}}$).

Implementation The function is implemented using a VRF scheme, cf. Section 6.2.

1: **if** VRFVerifyKey(vk$^{\text{LE}}$) = false **then** return false.
2: **if** VRFVerify(vk$^{\text{LE}}$, "LE"||nonce$^{\text{LE}}$||slot, proof) = false **then** return false.
3: Compute $h = $ VRFproof2hash(proof).
4: **if** $h \geq 2^{\ell_{\text{VRF}}} \varphi_{\text{LE}}($lotteryPower, diff$^{\text{LE}})$ **then** return false.
5: **return** true.

C.3.4 Block-Nonce Functions

The *block-nonce generation* function Birk.computeBlockNonce allows computing block nonce for a specific slot and specific baker.

Function Birk.computeBlockNonce

Inputs

slot $\in$ SLOTS
> The slot for which the nonce should be computed.

nonce$^{\text{LE}} \in$ NONCE$_{\text{LE}}$
> The leader election nonce for slot.

sk$^{\text{LE}} \in$ PRIVATEKEY$_{\text{LE}}$
> The leader election secret key of the baker.

Outputs

nonce $\in$ BLOCKNONCES
> The block proof required to create a new block in slot.

Implementation The function is implemented using a VRF scheme, cf. Section 6.2.

1: Compute val $= $ VRFHash(sk$^{\text{LE}}$, "NONCE"||nonce$^{\text{LE}}$||slot).
2: Compute prf $= $ VRFProve(sk$^{\text{LE}}$, "NONCE"||nonce$^{\text{LE}}$||slot).
3: **Return** (val, prf).

The *block-nonce verification* function Birk.verifyBlockNonce allows to verify the block nonce of a block.

Function Birk.verifyBlockNonce

Inputs

$\text{slot} \in \text{SLOTS}$
> The slot for which the nonce should be computed.

$\text{nonce}^{\text{LE}} \in \text{NONCE}_{\text{LE}}$
> The leader election nonce for slot.

$\text{vk}^{\text{LE}} \in \text{VERIFICATIONKEY}_{\text{LE}}$
> The leader election verification key of the baker.

$\text{nonce} = (\text{val}, \text{prf}) \in \text{BLOCKNONCES}$
> The block nonce.

Outputs

$b \in \text{BOOL}$
> Indicates if the nonce is valid

Implementation The function is implemented using a VRF scheme, cf. Section 6.2.

1: **if** $\text{VRFVerifyKey}(\text{vk}^{\text{LE}}) = \text{false}$ **then** return false.
2: **if** $\text{VRFVerify}(\text{vk}^{\text{LE}}, \text{"NONCE"}\|\text{nonce}^{\text{LE}}\|\text{slot}, \text{prf}) = \text{false}$ **then** return false.
3: Return true.

> ❶ Block nonces are used to compute future leader election nonces. To prevent grinding-attacks, it must hold that for fixed vk^{LE} (or the corresponding private key sk^{LE}) the block nonce for slot slot is unique. To prevent parties from learning future leader election nonces too early, it must hold that the block nonce for slot slot and private key sk^{LE} from an honest party looks random if one does not know the private key sk^{LE}.

C.4 Consensus Protocol

The core part of Birk is the Nakamoto-style consensus protocol. Bakers check if they become slot leader. If this is the case the baker creates a new block using data from the input pool Skov.inputPool. The protocol is run locally by each baker baker.

Protocol BirkConsensus(Skov, baker)

Implementation

1: **for** each slot $\text{slot} > 0$ **do**
2:     */* If one of the following steps fails end the computation for the current slot. */*
3:     Let $\text{tree} = \text{Skov.tree}$ and $\text{finEntries} = \text{Skov.finEntries}$.
4:     */* Get the current best chain. */*
5:     $\text{chain} = \text{Kontrol.BestChain}(\text{tree}, \text{finEntries})$.
6:     */* Get the current Birk parameters. */*
7:     $\text{nonce}^{\text{LE}} = \text{Kontrol.getLENonce}(\text{slot}, \text{chain})$
8:     $\text{diff}^{\text{LE}} = \text{Kontrol.getLEDifficulty}(\text{slot}, \text{chain})$
9:     $\text{bakers} = \text{Kontrol.getBakerSet}(\text{slot}, \text{chain})$
10:     */* Get the current lotteryPower for baker. */*
11:     $\text{lotteryPower} = \text{bakers}[\text{baker}].\text{lotteryPower}$.

```
12:        /* Leader election                                                          */
13:        proof = Birk.leaderElection(slot, nonce^LE, diff^LE, baker.sk^LE, lotteryPower).
14:        /* If successful create a new block.                                        */
15:        /* Compute block nonce.                                                     */
16:        nonce = Birk.computeBlockNonce(slot, nonce^LE, baker.sk^LE).
17:        /* Check whether new finalization entry is available.                       */
18:        Let prevFinEntry be the deepest nextFinEntry ≠ ⊥ in chain
19:        if Skov.finEntries contains finEntry with finEntry.ctr = prevFinEntry.ctr + 1
           then
20:            nextFinEntry = finEntry with finEntry.ctr = prevFinEntry.ctr + 1
21:        else
22:            nextFinEntry = ⊥
23:        /* Compute the block data.                                                  */
24:        (data, Skov.inputPool) = processInputs(chain, slot, finEntries, Skov.inputPool).
25:        /* Sign the block.                                                          */
26:        blk = (slot, H(chain.head), baker.bid, proof, nonce, nextFinEntry, data).
27:        σ^BC = Sign_BC(baker.sk^BC, blk).
28:        block = (slot, H(chain.head), baker.bid, proof, nonce, nextFinEntry, data, σ^BC).
29:        /* Add block to local block tree and multicast it.                          */
30:        (Skov.blockPool, Skov.tree) = Skov.addBlock(block, Skov.blockPool, Skov.tree).
31:        Multicast block over the network.
```

⚠ Birk consensus roughly corresponds to the Ouroboros Praos consensus. See [Dav+17] for details on the security of the protocol.

Appendix D

Afgjort Finalization

The finalization layer Afgjort is used to detect when a block can no longer be subject to a roll-back. The finalization layer Afgjort is used to prevent long-term rollbacks in the tree-based consensus of Birk.

The idea is that a finalization committee determines at regular intervals finalized blocks. Those finalized blocks act as genesis blocks, in the sense that they (and any blocks behind them) cannot be rolled back.

In a proof-of-stake protocol as Birk it can be seen that if one uses the rule of looking back N blocks in the tree to prevent rollbacks, then even in fair weather condition a 50% attack can cause a roll back of length N with probability 2^{-N}. So to achieve a negligible probability of attack, like 2^{-80}, one would have to look back 80 blocks. And this rule would not even guarantee finalization under exceptional network conditions. The role of our finalization protocol is to guarantee finalization much faster than in 80 blocks and to do it even when the network is under attack and/or is partitioned for a period of time.

Our finality layer achieves the following properties: Finalized blocks form a chain (*chain-forming*), all parties agree on the finalized blocks (*agreement*), the last finalized block does not fall too far behind the last block in the underlying blockchain (*updated*), and all finalized blocks at some point have been on the chain adopted by honest parties with at least 1/3 of the total participating stake (1/3-*support*).

The Afgjort layer uses the data stored in the Skov (cf. Section B.1).

D.1   Finalizers

Finalization is run by the *finalization committee* which consists of finalizers. Finalizers are weighted according to their voting power.

Definition 124. The finalization *voting power*, votingPower $\in$ FINVOTINGPOWER, is a real number between 0 and 1, i.e., FINVOTINGPOWER $= [0, 1]$. The voting power of a finalizer is always relative to the committee. That is, the *total voting power* of the committee (that is the sum of all voting power of committee members) is defined as 1.

Each finalizer has a unique ID and has its assigned voting power.

Definition 125. A *finalizer*, finalizer $\in$ FINALIZERS, is an entity and has the following values assigned.

Public values

These values are publicly know and stored on the blockchain.

$\mathtt{fid} \in$ FINALIZERIDS
> The unique *finalizer identifier*.

$\mathtt{votingPower} \in$ FINVOTINGPOWER
> The *voting power*.

$\mathtt{vk}^{\text{FINVOTE}} \in$ PUBLICKEY$_{\mathtt{FinVote}} =$ PUBLICKEY$_{\mathtt{VRF}}$
> The *voting verification key*, a VRF verification key, cf. Section 6.2.

$\mathtt{vk}^{\text{Fin}} \in$ VERIFICATIONKEYS$_{\mathtt{Fin}}$
> The *signature verification key*, cf. Section 5.1.3.

Private values

These values are only known to the finalizer itself.

$\mathtt{sk}^{\text{FINVOTE}} \in$ PRIVATEKEY$_{\mathtt{FinVote}} =$ PRIVATEKEY$_{\mathtt{VRF}}$
> The *voting private key*, a VRF secret key, cf. Section 6.2.

$\mathtt{sk}^{\text{Fin}} \in$ SIGNKEYS$_{\mathtt{Fin}}$
> The *signature signing key*, cf. Section 5.1.3.

ⓘ The Konsensus layer only requires that finalizer IDs are unique within the finalization committee. Otherwise, the IDs can be arbitrary bit strings. Overall, a finalizer ID could be a public-key which is assigned with an identity or an account.

</> In the implementation, all finalizers are also baker. Thus, the implementation uses the baker-ID to identify finalizers.

ⓘ In the current version of the protocols the committee taking care of finalization is public. This potentially allows DDOS attacks on the finalization committee which in the worst case turn off finalization. However, the underlying Birk layer will still be running.

💡 Future versions of Afgjort should include a protocol to restart finalization in case the committee is dead (e.g., due to DDOS). The core idea of a restart is to select a fresh finalization committee. Thus, the problem is not primarily algorithmic. It involves intervention from the board or some other humans on which the power has been put to restart finalization.

D.2 Finalization Entries

Finalization entries act as proof for the finalization of a block. They are defined as follows.

Definition 126. A *finalization entry*, $\mathtt{finEntry} \in$ FINALENTRIES, is a tuple consisting of:

$\mathtt{ctr} \in \mathbb{N}$
> The *sequence number* of the entry.

ptr ∈ HASH

> The hash of the *finalized block*.

finBDelay ∈ ℕ

> The [finalization!block delay]finalization block delay used in the finalization. It means at least `finBDelay` blocks below the finalized block existed when it was finalized. See Section D for details.

prf ∈ SIGNATURES_{Fin}*

> The [finalization!proof]finalization proof is a tuple of signatures, cf. Section 5.1.3.

The genesis block is considered final by definition. For convenience, the corresponding finalization entry is defined as follows.

Definition 127. The *genesis finalization entry*, `genesisFE` ∈ FINALENTRIES, is a tuple consisting of:

ctr := 0

> The genesis finalization entry is the first finalization entry.

ptr := H(genesis)

> The genesis finalization entry points to the genesis block `genesis`.

finBDelay := 0

> The delay is for convenience in description initially set to 0.

prf := ⊥

> The genesis finalization entry has no proof.

A finalization entry for finalization `ctr` is considered valid with respect to `Skov.tree` and `Skov.finEntries` if it was properly signed by the finalization committee for finalization `ctr`. More formally, validity is defined as follows.

Definition 128. A finalization entry (`ctr`, `ptr`, `finBDelay`, `prf`) is *valid* with respect to `Skov.tree` and `Skov.finEntries` if the following holds:

- Let `chain` be the chain from genesis block to the block pointed at by `ptr`.

- Let `finEntries′` be the finalization entries from `Skov.finEntries` with index smaller than `ctr`.

- Let `finCom = Kontrol.getFinalizationCommittee(ctr, chain, finEntries′)`.

- The tuple `prf` contains valid signatures from $t + 1$ (in terms of weight) of the parties in `finCom` on the string $(H(genesis), ctr, finBDelay, \text{WEAREDONE}, ptr)$.

> ⚠ This definition assumes that the session identifier `sid` in the finalization protocol is set to `sid = H(genesis)` (see Section D.5).

A block is considered final if there exists a valid finalization entry for that block.

Definition 129. A block `block` is *final* if the block is part of `Skov.tree` and there exists a valid finalization entry in `Skov.finEntries` with `ptr = H(block)`.

> ❗ The definition of a final block does not prevent the existence of two final blocks for the same sequence number. It does not ensure that all final blocks are on one chain in `Skov.tree`.

D.2.1 Verification Algorithm

To (later) verify finalization entries, parties can use the function Afgjort.verifyFinalProof. It takes as input `Skov.tree` and `Skov.finEntries`, and a new finalization entry. It outputs true if and only if the new finalization entry is valid according to Definition 128.

D.3 Required Interfaces

D.3.1 Kontrol Interfaces

The Afgjort layer requires the following functions from Kontrol (cf. Section E).

Best Chain Rule The function Kontrol.BestChain returns the best chain given tree `Skov.tree` and finalization entries `Skov.finEntries`. The function is specified in Section E.6.

Finalization Committee Selection The function Kontrol.getFinalizationCommittee returns the finalization committee given a slot, tree `Skov.tree` and finalization entries `Skov.finEntries` (cf. Section E.4).

Finalization Minimum Skip The function Kontrol.getFinalizationMinimumSkip takes as input a sequence number `ctr`, a chain `chain` containing all finalized blocks up to finalization $ctr - 1$, and the corresponding set of finalization entries `finEntries`. It returns the minimum skip for the depth of finalization `ctr`.

D.4 Justifications

We introduce the concept of *justifications*. A justification J is a predicate which takes as input a value v and the local state of a party (in particular its tree). We say that the value v is J-justified for party P_i if the predicate evaluates to true for v and P_i's state.

> ❗ For the below definitions, the local state of a party consists of its tree plus the set of all messages ever received at the party. This is just a definition and does not mean the parties need to keep this local state. Messages can be deleted when they are no longer needed. Since many of the protocols interact in subtle ways, it is also subtle when a message can be deleted safely. For safety, the semantics of the protocols are defined as if no message was ever deleted. To be cautious about this, it is probably a good idea to use formal verification or model checking.

Definition 130. For a value v that can be sent or received, a *justification* is a predicate J which can be applied to v and the local state of a party. Justifications are monotone with respect to time, i.e, if J is true for a value v at party P at time t, then J is true (at that party) any time $\geq t$.

We write $J(v, P) = \top$ to mean that J is true when applied to the local state of party P. We write $J(v, P) = \bot$ to mean that J is false when applied to the state of party P. If the party is implicitly given, say by being the local party at which the code is running, then we write $J(v)$.

Definition 131. Let J be a justification. A *justified value* $v[J]$ is the tuple (v, J) where $J(v) = \top$.

Justified values can be passed in function calls or be returned from a function. If we say that $v[J]$ is sent on the network, then only the value v is sent. When a party received a justified value $v[J]$ from the network, then it means that (a) the party already knows J, (b) the party received value v, and (c) the party waited until $J(v) = \top$ (with respect to its local state). In particular, if a party received $v[J]$ it *cannot* be that $J(v) = \bot$.

Definition 132. A justification J is an *eventual justification* if for any value v and parties P_i and P_j the following holds. If v becomes justified for party P_i at time t and both P_i and P_j from that point in time are live and honest, then eventually v becomes justified for party P_j.

> **ⓘ** Some examples of eventual justifications for illustration.
> 1. A message m is justified if it was received on the peer-to-peer layer. This is an eventual justification by assumption on the network: when m was received by P_i it will also eventually be received by P_j.
> 2. A message m is justified if it was received on the peer-to-peer layer and validly signed by P_i. This is an eventual justification by assumption on the network and signatures being verifiable by all parties.
> 3. A message m is justified if it was received on the peer-to-peer layer and valid signatures on m were received by $n - t$ parties. This is an eventual justification by assumption on the network and signatures being verifiable by all parties: when m and the signatures were received by P_i they will also eventually be received by P_j who can also verify the signatures.

D.5   Finalization Protocol

The finalization algorithm Afgjort.AfgjortFinalization is used to periodically create finalized blocks in an infinite loop. In the loop the AgreeOnBlock sub-protocol is used to determine a finalized block (see Section D.5.2).

Algorithm AfgjortFinalization($\mathtt{sid}, \mathsf{Skov}, \mathtt{finalizer}$)

Each (potential) finalizer `finalizer` executes the following process:

1: **for** $\mathtt{ctr} = 1, 2, 3, \dots$ **do**
2:      **wait until** $\mathsf{Skov.finEntries}$ contains $(\mathtt{ctr}', \mathtt{ptr}', \mathtt{finBDelay}', \mathtt{prf}')$ with $\mathtt{ctr}' = \mathtt{ctr} - 1$, and $\mathtt{chain} := \mathsf{Kontrol.BestChain}(\mathsf{Skov.tree}, \mathsf{Skov.finEntries})$ contains block with hash $\mathtt{ptr}'$.
3:      $\mathtt{finCom} := \mathsf{Kontrol.getFinalizationCommittee}(\mathsf{Skov.finEntries}, \mathtt{chain})$
4:      **if** $\mathtt{finalizer} \in \mathtt{finCom}$ **then**
5:          $\mathtt{faid} := (\mathtt{sid}, \mathtt{ctr})$
6:          $\mathtt{depth} := \mathsf{getFinalizationDepth}(\mathtt{ptr}', \mathtt{chain})$
7:          $(\mathtt{bDelShrinkFactor}, \mathtt{bDelGrowFactor}) := \mathsf{getFinBDelFactors}(\mathtt{ctr}, \mathsf{Skov.finEntries}, \mathtt{chain})$
8:          $\mathtt{finBDelay} := \lfloor \mathtt{bDelShrinkFactor} \cdot \mathtt{finBDelay}' \rfloor$
9:          $\mathtt{ABBADelayInc} := \mathsf{Kontrol.getABBADelayInc}(\mathtt{ctr}, \mathsf{Skov.finEntries}, \mathtt{chain})$
10:         $(\mathtt{ptr}, \mathtt{finBDelay}'', \mathtt{prf}) := \mathsf{AgreeOnBlock}(\mathtt{faid}, J_{\mathrm{INTREE}}^{\mathtt{ctr}, \mathtt{depth}, \mathtt{finBDelay}}, \mathtt{ctr}, \mathtt{depth}, \mathtt{finBDelay}, \mathtt{ABBADelayInc})$
11:         multicast $\mathtt{finEntry} := (\mathtt{ctr}, \mathtt{ptr}, \mathtt{finBDelay}'', \mathtt{prf})$

> The above counting assumes that the genesis block is also counted as finalized with counter 0.

> The input `sid` given to AfgjortFinalization is a session identifier. When AfgjortFinalization is started initially, it is set to be $\mathsf{H}(\texttt{genesis})$, the hash of the genesis block. Using different session identifiers in different incarnations is done to avoid interference between different runs, like replay attacks of signed values.
> The definition of `sid` has influence on Definition 128 which defines the validity of finalization entries.

> In a future version, the `sid` could be of the form $(\mathsf{H}(\texttt{genesis}), c)$ where c is a counter c. Initially, $c = 1$. If finalization needs to be restarted, one can increase c.

> Within the AgreeOnBlock sub-protocol, `finBDelay` is increased each time the parties fail to achieve an agreement. Once finalization is achieved, the sub-protocol returns the (potentially increased) delay. To prevent the delay `finBDelay` from growing indefinitely, the value is decreased within the loop of AfgjortFinalization. This will empirically find a good value for `finBDelay` over time.

D.5.1 Computing Finalization Depth

To measure whether the finalization falls behind, we use the following approach. When a block `block` is finalized, let `block'` be the deepest block for which a finalization entry is in the chain up to and including `block`, and let `block''` be the deepest ancestor of `block` that has been finalized. If the chain does not grow too fast, we should have `block' = block''`. However, if finalization is falling behind the chain a lot, `block` has been added to the tree before `block''` was finalized, in which case we have `block' ≠ block''`. We use this observation to adjust the gap between finalized blocks: If `block' ≠ block''`, we increase it, otherwise we slightly decrease it.

Function Afgjort.getFinalizationDepth

Inputs
  `ctr` $\in \mathbb{N}$
       The index of the finalization for which the depth is to be determined.
  `finEntries` $\in$ FINALENTRIES*
       A set of final entries.
  `chain` $\in$ CHAINS
       A chain containing all blocks pointed to in `finEntries`.

Outputs
  `depth` $\in \mathbb{N}$
       The depth at which the next block should be finalized.

Promise
       `ctr` > 0. `finEntries` contains at least `ctr` entries with counter values $0, \ldots, \texttt{ctr} - 1$.
       `chain` contains all blocks pointed to in `finEntries`.

> depth is strictly larger than depth of block with hash ptr. Furthermore, (informally) the next depth should be chosen such that finalized blocks are close to the leaves of the block tree. The output of this function only depends on the first ctr entries in finEntries and the prefix of chain up to the block finalized in the ctrth entry in finEntries.
>
> **Implementation**
> 1: $\mathrm{minSkip} := \mathsf{Kontrol.getFinalizationMinimumSkip}(\mathtt{ctr}, \mathtt{finEntries}, \mathtt{chain})$
> 2: **if** $\mathtt{ctr} = 1$ **then**
> 3:   **return** minSkip   *// This is the depth of the first real finalized block.*
> 4: $(\mathrm{skipShrinkFactor}, \mathrm{skipGrowFactor}) := \mathsf{getFinSkipFactors}(\mathtt{ctr}, \mathtt{finEntries}, \mathtt{chain})$
> 5: Let $(\mathtt{ctr} - 1, \mathtt{ptr}_{\mathtt{ctr}-1}, \mathtt{finBDelay}_{\mathtt{ctr}-1}, \mathtt{prf}_{\mathtt{ctr}-1}) \in \mathtt{finEntries}$ be the entry for finalization $\mathtt{ctr} - 1$, and let $(\mathtt{ctr} - 2, \mathtt{ptr}_{\mathtt{ctr}-2}, \mathtt{finBDelay}_{\mathtt{ctr}-2}, \mathtt{prf}_{\mathtt{ctr}-2}) \in \mathtt{finEntries}$ be the one for $\mathtt{ctr} - 2$.
> 6: Let $\mathtt{block}_{\mathtt{ctr}-1}$ and $\mathtt{block}_{\mathtt{ctr}-2}$ be blocks on chain with hashes $\mathtt{ptr}_{\mathtt{ctr}-1}$ and $\mathtt{ptr}_{\mathtt{ctr}-2}$, respectively.
> 7: $\mathrm{prevSkip} := \mathrm{depth}(\mathtt{block}_{\mathtt{ctr}-1}) - \mathrm{depth}(\mathtt{block}_{\mathtt{ctr}-2})$
> 8: **if** chain up to and including $\mathtt{block}_{\mathtt{ctr}-1}$ contains finalization entry for $\mathtt{ptr}_{\mathtt{ctr}-2}$ **then**
> 9: $\mathrm{skip} := \lfloor \mathrm{skipShrinkFactor} \cdot \mathrm{prevSkip} \rfloor$
> 10: **else**
> 11: $\mathrm{skip} := \max(\lceil \mathrm{skipGrowFactor} \cdot \mathrm{prevSkip} \rceil, \mathrm{prevSkip} + 1)$ *// always grow*
> 12: **return** $\mathrm{depth}(\mathtt{block}_{\mathtt{ctr}-1}) + \max(\mathrm{skip}, \mathrm{minSkip})$

D.5.2 Block Agreement

The basic building block of the finalization scheme is the protocol Afgjort.AgreeOnBlock which is used by the finalization committee for index ctr to agree on the final block for ctr. The algorithm takes as (global) inputs a unique id faid, index ctr, a depth depth, and an integer delay finBDelay ≥ 0. It outputs a hash of the final block block for depth depth, the actual delay used in the algorithm, and a finalization proof.

The algorithm uses a *weak multivalued Byzantine agreement* (WMVBA), which is described in Section D.6.

We define the following eventual justification.

Definition 133. A block block is $J_{\mathrm{INTREE}}^{\mathtt{ctr}, \mathtt{depth}, \mathtt{finBDelay}}$-justified for finalizer $\mathtt{finalizer}_i$ if block is at depth depth and part of a path of length depth+finBDelay in the tree of $\mathtt{finalizer}_i$ that contains ctr finalized blocks.

> **❗** It is important that the path of length depth+finBDelay is part of the $J_{\mathrm{INTREE}}^{\mathtt{ctr}, \mathtt{depth}, \mathtt{finBDelay}}$-justification. If it were just the depth of a block, the finalization could deadlock.

> **❗** Requiring ctr finalized blocks on the path assumes the genesis block is counted as finalized and finalization starts with $\mathtt{ctr} = 1$.

Finalizer finalizer does the following:

1: **repeat**
2:      **wait until** chain := Kontrol.BestChain(Skov.tree, Skov.finEntries) has length $\geq$ depth + finBDelay and contains ctr finalized blocks.
3:      baid := (faid, finBDelay)
4:      Let $\text{block}_{\text{depth}}$ be the block on chain at depth depth.
5:      Run (block, prf) := WMVBA(baid, $J_{\text{INTREE}}^{\text{ctr,depth,finBDelay}}$) with input $\text{block}_{\text{depth}}$.
6:      **if** block = $\perp$ **then**   *// Increase finBDelay*
7:          (bDelShrinkFactor, bDelGrowFactor) := getFinBDelFactors(ctr, Skov.finEntries, chain)
8:          finBDelay := $\max(\lceil \text{bDelGrowFactor} \cdot \text{finBDelay} \rceil, \text{finBDelay} + 1)$
9: **until** block $\neq \perp$
10: **return** (H(block), finBDelay, prf)

D.6   Weak Multi-Valued Byzantine Agreement

At the core of the AgreeOnBlock algorithm from Section D.5.2, finalizers in the committee use a Byzantine agreement protocol, called WMVBA, relative to a justification J[1]. In the protocol finalizers are weighted according to their voting power. Each invocation of WMVBA is identified by a unique identifier baid. Each finalizer finalizer_i has an input $\text{p}_i \in \{0,1\}^*$ justified by eventual justification J which is called the proposal. Every honest finalizer gets (eventually) an output d_i where $\text{d}_i \in \{0,1\}^* \cup \perp$ which is called the decision. This decision is again justified, and the justification is used to create a finalization proof.

Let n denote the total voting power of the committee of finalizers and let t denote the voting power of the corrupted committee members. Given $n > 3t$ a WMVBA scheme relative to justification J must satisfy the following properties.

Consistency: If some honest finalizer_i and finalizer_j output decisions d_i respectively d_j, then $\text{d}_i = \text{d}_j$.

Termination: If all honest finalizer input some justified proposal, then eventually all honest finalizer output some decision.

Weak Validity: If during the protocol execution[2] there exists a decision d such that no other decision d' with $\text{d}' \neq \text{d}$ is J-justified for any honest party, then no honest finalizer_i outputs a decision d' with $\text{d}' \neq \text{d}$.

$n/3$-Support: If some honest finalizer_i outputs decision d with $\text{d} \neq \perp$, then at least $n/3$ of the honest finalizers (in terms of voting power) had J-justified input d.

Protocol idea. At the beginning of the WMVBA protocol all parties first run the Freeze subprotocol. In Freeze, parties send their proposals to all other parties and every party checks whether they received at least $n - t$ proposals for the same block. In that case, their output for Freeze is that block, otherwise it is $\perp$. Freeze thereby boils the decision for a finalized block down to the binary decision between $\perp$ and a unique block output by Freeze (if that exists). To

---

[1] It is the J_{INTREE} justification, but that is not important for this section. We only need that it is an eventual justification.

[2] That is until the first honest party gets an output. For the implementation with Freeze its even enough that it holds until the Freeze sub-protocol is done.

this end, a binary Byzantine agreement protocol ABBA is run after Freeze. We provide details about the sub-protocols and WMVBA in the following sections.

D.6.1 Afgjort Catch-Up

At the moment, the semantics of the protocols are defined relative to justifications being eventual. If implemented naively, this would require a network with a time to live (TTL) of ∞ and constant resending of justifications.

Thus, careful engineering must be applied to be able to clean up state without changing the semantics of the protocols. The intention is that justifications will be eventual under normal network conditions. Under strange conditions, like a partitioning, the parties can then start e.g., pushing and pulling for old justifications, like forwarding signatures and so on. This works for communication where all messages are anticipated, which holds for the BA protocols. The basic idea would be to use amble timeouts. And if then the asynchronous protocols did not progress within the timeout, then start pulling for old messages. This should be implemented as a general mechanism, not as a per-protocol ad-hoc mechanism.

> 💡 Instead of the pull and push methods, we could use the timeout and resend method. That is, send the message again and again (with a certain timeout between) until you get an ACK.

> </> In the implementation, parties periodically ask their peers for messages they might have missed.

D.6.2 Notation

To simplify notation in this section we denote by party P a finalizer the is part of the finalization committee which runs WMVBA as part of AgreeOnBlock. We denote by n the total voting power of the committee and let t denote the voting power of the corrupted committee members. In the protocols parties are weighted according to their voting power. For example, a statement of the form "at least x parties" is to be read as "parties with at least $\frac{x}{n}$ voting power".

D.6.3 Freeze Protocol

Each honest party P_i has a J-justified input p_i, called proposal. In our use case these proposals are blocks. Each honest party P_i (eventually) outputs a decision d_i which is either a block or $\bot$. The output decision d_i of P_i is justified by justification J_{dec} (see Definition 136). The Freeze protocol satisfies the following properties.

Weak Consistency: If honest parties P_i and P_j output decisions $\mathsf{d}_i \neq \bot$ and $\mathsf{d}_j \neq \bot$ respectively, then $\mathsf{d}_i = \mathsf{d}_j$.

Weak Validity: If during the protocol execution[3] there exists a J-justified proposal p such that no other proposal $\mathsf{p}' \neq \mathsf{p}$ is J-justified for any honest party, then no honest party P_j outputs p'.

$n-2t$-Support: If honest party P_i outputs decision $\mathsf{d}_i \neq \bot$, then at least $n-2t$ honest parties had d_i as input.

Termination: If all honest parties input some justified proposal, then eventually all honest parties output a decision.

---

[3]That is until the first honest party gets an output.

Next, we define the following justifications relative to the input justification J.

Definition 134. A proposal message $m = (\texttt{baid}, \text{PROPOSAL}, \texttt{p})$ from P_i is considered J_{prop}-justified for P_j if m is signed by P_i and $\texttt{p}$ is J-justified for P_j.

Definition 135. A vote message $m = (\texttt{baid}, \text{VOTE}, \texttt{v})$ from P_i is considered J_{vote}-justified for P_j if it is signed by P_i and either for $\texttt{v} \neq \perp$ P_j has collected J_{prop}-justified messages $(\texttt{baid}, \text{PROPOSAL}, \texttt{v})$ from at least $n - 2t$ parties or for $\texttt{v} = \perp$ P_j has collected J_{prop}-justified messages $(\texttt{baid}, \text{PROPOSAL}, \texttt{p})$ and $(\texttt{baid}, \text{PROPOSAL}, \texttt{p}')$ (from two different parties) where $\texttt{p}' \neq \texttt{p}$.

Definition 136. A decision message $(\texttt{baid}, \text{FROZEN}, \texttt{d})$ is J_{dec}-justified for P_j if P_j collected J_{vote}-justified messages $(\texttt{baid}, \text{VOTE}, \texttt{d})$ from at least $t + 1$ parties.

> ❗ Note that under the assumption that the underlying peer-to-peer layer has eventual delivery, all the above justifications are eventual too (see also Section D.6.1).

The protocol Freeze runs as follows.

Protocol Freeze($\texttt{baid}, J$)

Each party P_i has a proposal $\texttt{p}_i[J]$ as input.
Propose:
  1. Each party P broadcasts signed proposal message $(\texttt{baid}, \text{PROPOSAL}, \texttt{p})[J_{\text{prop}}]$.
Vote:
  2. Each party P collects proposal messages $(\texttt{baid}, \text{PROPOSAL}, \texttt{p}_i)[J_{\text{prop}}]$. After receiving proposal messages from at least $n - t$ parties do the below (but keep collecting to ensure eventual justification).
     (a) If the same proposal $\texttt{p}$ is received from at least $n - t$ parties, then broadcast signed vote message $(\texttt{baid}, \text{VOTE}, \texttt{p})[J_{\text{vote}}]$.
     (b) Otherwise, proposals $\texttt{p} \neq \texttt{p}'$ are received, then broadcast signed vote message $(\texttt{baid}, \text{VOTE}, \perp)[J_{\text{vote}}]$.
Freeze:
  3. Each party P collects $(\texttt{baid}, \text{VOTE}, \texttt{p}_i)[J_{\text{vote}}]$ messages. Once party P collected $(\texttt{baid}, \text{VOTE}, \texttt{p}_i)[J_{\text{vote}}]$ messages from $n - t$ parties:
     (a) If the same $(\texttt{baid}, \text{VOTE}, \texttt{p})[J_{\text{vote}}]$ is received from strictly more than t parties, output $(\texttt{baid}, \text{FROZEN}, \texttt{p})[J_{\text{dec}}]$.
     (b) Else if $(\texttt{baid}, \text{VOTE}, \perp)[J_{\text{vote}}]$ is received from strictly more than t parties, output $(\texttt{baid}, \text{FROZEN}, \perp)[J_{\text{dec}}]$.
     (c) Else, collapse universe and restart //This should not happen.
  4. Each party P keeps collecting vote messages until WMVBA is terminated (i.e., until P_i gets an output in WMVBA). Party P stores all decisions $(\texttt{baid}, \text{FROZEN}, \texttt{p})$ which become J_{dec}-justified.

> ⚠ It is crucial for the termination property of WMVBA that parties do not terminate Freeze after they get an output (see Step 4 of Freeze).

> ❗ Note that Freeze is an asynchronous protocol. In Step 2 once $n - t$ proposals have been collected, the party has to decide on its vote. The party must not wait (no further proposals might arrive). In particular, it could be that the party decides on $\bot$-vote as there are not enough proposals for a p-vote at the point it collected $n - t$ proposals. It does not matter that the party at a later point in time might have enough proposals for a p-vote. The same holds for the decision in Step 3.

> ❗ To prevent (simple) DoS attacks, parties should only relay one proposal message per protocol invocation and party.

> </> The implementation only accepts the first received vote/proposal per party. In case 3c the implementation does nothing.

Lemma 137. *For $n > 3t$ the protocol Freeze satisfies weak-agreement, pre-agreement validity and, termination. Furthermore, the output is justified with respect to J_{dec}.*

Lemma 138. *If an honest party P_i outputs (J_{dec}-justified) decision $\mathrm{d}_i \neq \bot$ in Freeze, then eventually all honest parties will accept d_i has J_{dec}-justified.*

All the proofs of the above lemmata are found in [Mag+19].

D.6.4 Another Binary Byzantine Agreement

We now use as a sub-protocol a binary Byzantine agreement protocol called ABBA. Parties use ABBA to decide whether they agreed on a non-$\bot$ decision in Freeze. The global inputs, i.e., the pre-agreed parameters, are an input justification J_{in} and ABBADelayInc determining by how long the waiting times in the protocol are increased after each failed attempt. Each party has a J_{in}-justified bit $\mathrm{b} \in \{\bot, \top\}$ as input. The output of honest parties in ABBA are J_{out}-justified bits (see Definition 146). The protocol has these properties:

Consistency: If for some baid some honest P_i and P_j output bits b_i respectively b_j, then $\mathrm{b}_i = \mathrm{b}_j$.

Validity: If for some baid all honest parties input the same J_{in}-justified bit b, then no honest P_j outputs a decision b'' with $\mathrm{b}'' \neq \mathrm{b}$.

Termination: If for some baid all honest voters input some J_{in}-justified bit, then eventually all honest voters output some bit.

Overview. The idea of the ABBA-protocol is to use randomized graded agreement protocol. The randomization comes in the form of a leader election where the elected leader helps parties to come to a decision. The ABBA-protocol uses two primitives, which we describe first. These are a *core set selection* protocol CSS, and a *lottery*.

Core Set Selection. For ABBA we need the *core set selection* protocol CSS. The global inputs, i.e., the pre-agreed parameters, are input justification J_{cssin} and a delay Δ_{CSS}. Each party inputs a J_{cssin}-justified bit where J_{cssin} is some (eventual) justification which is later defined by ABBA. Each honest party P_i (eventually) outputs a set Core_i which contains justified tuples (P, b).

The idea with the delay Δ_{CSS} is to give honest parties more time to submit their input to the core-set. This allows to counter the effect of desynchronization. In particular, assume that

honest parties start the protocol within Δ_{st} and that the network delay is at most $\Delta_{\mathtt{net}}$. Then honest parties are at most $\Delta_{st} + \Delta_{\mathtt{net}}$ desynchronized. By waiting $\Delta_{\mathtt{CSS}} > \Delta_{st} + \Delta_{\mathtt{net}}$ the inputs of all honest parties will be part of the core set. The protocol has the following properties.

Common Core: The output sets of honest parties have a common core $\mathtt{Core} \subseteq \bigcap_i \mathtt{Core}_i$ which contains tuples $(P, \mathtt{b})$ from at least $n - t$ different[4] parties.

Weak Validity: If during the protocol execution of CSS for some baid there exists a $J_{\mathtt{cssin}}$-justified $\mathtt{b}$ such that no other bit $\mathtt{b}'$ is $J_{\mathtt{cssin}}$-justified for any honest party, then all tuples in the output set $\mathtt{Core}_i$ of honest party P_i are of the form $(\cdot, \mathtt{b})$.

Unique Honest Tuple: The output set $\mathtt{Core}_i$ of honest party P_i contains for each honest party P_j at the tuple $(P_j, \mathtt{b}_j)$ where $\mathtt{b}_j$ is the input of P_j.

Termination: If all honest parties have $J_{\mathtt{cssin}}$-justified input, then all honest parties will eventually terminate.

$\Delta_{\mathtt{CSS}}$**-Waiting:** If $\Delta_{\mathtt{CSS}}$ is larger than the desynchronization of honest parties, then output set $\mathtt{Core}_i$ of honest party P_i contains tuples from all honest parties. Moreover, all honest outputs are fixed before the first honest party gives an output.

We define the following justifications relative to justification $J_{\mathtt{cssin}}$.

Definition 139. A tuple $(P_i, \mathtt{b}_i)$ is $J_{\mathtt{tpl}}$-justified for P_j if it is correctly signed by P_i and $\mathtt{b}_i$ is $J_{\mathtt{cssin}}$-justified for P_j.

Definition 140. A *seen message* $(\mathrm{SEEN}, P_k, (P_i, \mathtt{b}_i))$ is $J_{\mathtt{seen}}$-justified for P_j if it is correctly signed by P_k and $(P_i, \mathtt{b}_i)$ is $J_{\mathtt{tpl}}$-justified for P_j.

Definition 141. A *done-reporting message* $(\mathrm{DONEREPORTING}, P_k, \mathtt{iSaw}_k)$ is $J_{\mathtt{done}}$-justified for P_j if it is correctly signed by P_k and for each tuple $(P_i, \mathtt{b}_i) \in \mathtt{iSaw}_k$ P_j has a $J_{\mathtt{seen}}$-justified $(\mathrm{SEEN}, P_k, (P_i, \mathtt{b}_i))$.

The protocol runs as follows.

Protocol CSS(baid, $J_{\mathtt{cssin}}, \Delta_{\mathtt{CSS}}$)

The protocol is described from the view point of a party P_i which has $J_{\mathtt{cssin}}$-justified input bit $\mathtt{b}_i$.

Start:
- Party P_i sets flag $\mathtt{report}_i$ to $\top$. It initializes sets $\mathtt{iSaw}_i$ and $\mathtt{manySaw}_i$ to $\varnothing$. Then P_i sends its $J_{\mathtt{cssin}}$-justified input $\mathtt{b}_i$ signed to all parties.

Reporting Phase:
- Once P_i receives signed $\mathtt{b}_j$ from P_j such that $(P_j, \mathtt{b}_j)$ is $J_{\mathtt{tpl}}$-justified, P_i adds $(P_j, \mathtt{b}_j)$ to $\mathtt{iSaw}_i$ and sends signed $(\mathrm{SEEN}, P_i, (P_j, \mathtt{b}_j))$ to all parties. Party P_i does this for each party P_j at most once.
- Once P_i received $J_{\mathtt{seen}}$-justified $(\mathrm{SEEN}, P_k, (P_j, \mathtt{b}_j))$ from at least $n - t$ parties, party P_i adds $(P_j, \mathtt{b}_j)$ to $\mathtt{manySaw}_i$.
- Once $\mathtt{manySaw}_i$ contains tuples $(P_j, \cdot)$ for at least $n - t$ parties, P_i waits for $\Delta_{\mathtt{CSS}}$ seconds (while still collecting tuples) and then sets $\mathtt{report}_i$ to $\bot$.

Closing Down:
- Once P_i sets $\mathtt{report}_i$ to $\bot$, P_i sends to all parties signed $(\mathrm{DONEREPORTING}, P_i, \mathtt{iSaw}_i)$.
- Once P_i received $J_{\mathtt{done}}$-justified $(\mathrm{DONEREPORTING}, P_j, \mathtt{iSaw}_j)$ from at least $n - t$ parties, P_i sets $\mathtt{Core}_i$ to be the set of all currently $J_{\mathtt{tpl}}$-justified $(P_j, \mathtt{b}_j)$. It then waits

---

[4]Note that $\mathtt{Core}$ or any $\mathtt{Core}_i$ contain multiple tuples with the same (dishonest) party.

> for Δ_{CSS} seconds (and stops collecting messages), and afterwards outputs $\mathtt{Core}_i$.

> ❗ Instead of sending out $(\textsc{seen}, P_i, \mathtt{v}_i)$ each time a $(P_i, \mathtt{v}_i)$ is added to $\mathtt{iSaw}$ party P should send them out in batches (where parties are indicated in a bit vector). In particular, for the first batch P should wait a few seconds after it collected values from at least $n - t$ parties. After that occasionally send out new batches.

> ℹ The protocol can be augmented to report on malicious behavior. This is not useful for the protocol itself as it already achieves the best possible security guarantees. However, the reporting might be useful overall, e.g. to punish misbehaving parties (slashing).

> ℹ The protocol would also work for generic values, i.e. from some domain D and not just bits.

Lemma 142. *For $t < \frac{n}{3}$ the protocol CSS satisfies common core, weak validity, unique honest tuples, termination, and Δ_{CSS}-waiting.*

See [Mag+19] for a proof.

Lottery. The $\mathsf{A\mathcal{B}BA}$ protocol requires a lottery which ranks parties and where the expected rank correlates with the party's lottery power. As in Birk, we use a VRF scheme to implement the lottery (cf. Section C.3). We can, however, not use the same lottery as in Birk because there needs to be precisely one winner in $\mathsf{A\mathcal{B}BA}$, while it is possible in the lottery from Birk to have more than one winner, or no winner at all.

Analogous to Birk, we use the private (resp. public) keys of the VRF scheme as private (resp. verification) keys of the leader election. The algorithm $\mathsf{A\mathcal{B}BALottery}$ gets as input the unique invocation identifier $\mathtt{baid}$, the phase counter k, the secret key $\mathtt{sk}^{\mathsf{VRF}}$ of the VRF, and the lottery power $\mathtt{lotteryPower} \in (0, 1]$ of the party. It outputs a lottery ticket $\mathtt{ticket}$ consisting of a lottery number $n \in (0, 1]$ and a proof p. The latter can be used to verify the ticket using $\mathsf{verifyA\mathcal{B}BALotteryTicket}$. We assume the VRF hash is a value in $\left\{0, \ldots, 2^{\ell_{\mathsf{VRF}}} - 1\right\}$.

Algorithm $\mathsf{A\mathcal{B}BALottery}(\mathtt{baid}, k, \mathtt{sk}^{\mathsf{VRF}}, \mathtt{lotteryPower})$

1: $h \leftarrow \mathsf{VRFHash}(\mathtt{sk}^{\mathsf{VRF}}, \text{``AL''}||\mathtt{baid}||k)$
2: $p \leftarrow \mathsf{VRFProve}(\mathtt{sk}^{\mathsf{VRF}}, \text{``AL''}||\mathtt{baid}||k)$
3: $n \leftarrow \left(2^{-\ell_{\mathsf{VRF}}}(h + 1)\right)^{\frac{1}{\mathtt{lotteryPower}}}$
4: $\mathtt{ticket} \leftarrow (n, p)$
5: **return** $\mathtt{ticket}$

Algorithm $\mathsf{verifyA\mathcal{B}BALotteryTicket}(\mathtt{baid}, k, \mathtt{pk}^{\mathsf{VRF}}, \mathtt{lotteryPower}, \mathtt{ticket})$

1: Let $(n, p) \leftarrow \mathtt{ticket}$
2: **if** $\mathsf{VRFVerify}(\mathtt{pk}^{\mathsf{VRF}}, \text{``AL''}||\mathtt{baid}||k, p) = \mathtt{false}$ **then return** $\mathtt{false}$
3: $h \leftarrow \mathsf{VRFproof2hash}(p)$

The lottery has some desirable properties summarized in the following lemma. See Appendix F.1 for a proof.

Lemma 143. *Assuming the VRF behaves like a random oracle, we have the following properties:*

1. *Assume some party has lottery power $\sum_{i=1}^{k} \texttt{lotteryPower}_i$ and let p be the probability that this party has the highest lottery number. If this party is split into k sub-parties with lottery powers $\texttt{lotteryPower}_1, \ldots, \texttt{lotteryPower}_k$, respectively, the probability that any of these sub-parties has the highest lottery number is equal to $p + \epsilon$ for some negligible ϵ.*

2. *If party P_2 has twice the lottery power of P_1, then P_2 has a higher lottery number than P_1 with probability $\frac{2}{3} - \epsilon$ for some negligible ϵ.*

The binary Byzantine agreement protocol. In the AꝪBA protocol parties have as input a J_{in}-justified bit. The goal is to agree on a common J_{out}-justified output bit. The protocol is partially synchronous but proceeds in (locally determined) phases.

We define the following justifications relative to a justification J_{in}.

Definition 144. A bit b is $J_{\text{phase,1}}$-justified (*phase-1 justified*) for P_i if it is J_{in}-justified.

Definition 145. For $k > 1$ a bit b is $J_{\text{phase,k}}$-justified (*phase-k justified*) for P_i if P_i has $t+1$ signatures on $(\texttt{baid}, \text{JUSTIFIED}, \texttt{b}, k-1)$.

Definition 146. A bit b is J_{out}-justified (output) for P_i if P_i has $t+1$ signatures on $(\texttt{baid}, \text{WEAREDONE}, \texttt{b})$.

The protocol runs as follows.

Protocol AꝪBA($\texttt{baid}, J_{\text{in}}, \texttt{ABBADelayInc}$)

The protocol is described from the view point of a party P_i which has J_{in}-justified input $\texttt{b}_i$. The party starts both the "Graded Agreement" and the "Closing Down" part of the protocol.

Graded Agreement In each phase $k = 0, 1, 2, \ldots$ do the following:

1. The parties jointly run $\mathsf{CSS}(\texttt{baid}, J_{\text{phase,k}}, k \cdot \texttt{ABBADelayInc})$ where P_i inputs $\texttt{b}_i$. Denote by $\texttt{Core}_i$ the output of P_i.

2. P_i computes its lottery ticket $\texttt{ticket}_i$ and broadcasts signed $(\texttt{baid}, \text{JUSTIFIED}, \texttt{b}_i, k)$ along with $\texttt{ticket}_i$.

3. P_i waits for $k \cdot \texttt{ABBADelayInc}$ seconds and then does the following:
   - If all bits (of the tuples) in $\texttt{Core}_i$ are $\top$ let $\texttt{b}_i = \top$ and $\texttt{grade}_i = 2$.
   - Else if at least $n - t$ bits in $\texttt{Core}_i$ are $\top$ let $\texttt{b}_i = \top$ and $\texttt{grade}_i = 1$.
   - Else if all bits in $\texttt{Core}_i$ are $\bot$ let $\texttt{b}_i = \bot$ and $\texttt{grade}_i = 2$.
   - Else if at least $n - t$ bits in $\texttt{Core}_i$ are $\bot$ let $\texttt{b}_i = \bot$ and $\texttt{grade}_i = 1$.
   - Else, select a bit b which occurs $> t$ in $\texttt{Core}_i$. If this bit is not unique, verify all lottery tickets and select the bit b where $(\texttt{b}, P) \in \texttt{Core}_i$ and P has the highest valid lottery ticket for all parties in $\texttt{Core}_i$. Let $\texttt{b}_i = \texttt{b}$ and $\texttt{grade}_i = 0$.

4. Go to the next phase.

Closing Down Each party sends at most one $(\texttt{baid}, \text{WEAREDONE}, \cdot)$ message.

1. When P_i achieves grade 2 for the first time it sends signed $(\texttt{baid}, \text{WEAREDONE}, \texttt{b}_i)$

---

to all parties.

2. Once having receiving at least $t + 1$ signed $(\texttt{baid}, \textsc{WeAreDone}, \texttt{b})$ terminate the protocol and output $J_{\texttt{out}}$-justified $\texttt{b}$.

> **!** In ABBA, there are three places (including two within CSS) where parties wait. These waiting times are effective once they exceed the desynchronization of the parties (and are the reason our protocol is partially synchronous and not asynchronous): The first one in CSS ensures that all honest parties make it into the core-set, the second one in CSS ensures that all honest outputs of CSS are fixed before honest parties give their outputs (which in turn guarantees that these outputs do not depend on the lottery tickets in ABBA), and the last one in ABBA ensures that all honest lottery tickets arrive in time.

Lemma 147. *For $n > 3t$ the protocol* ABBA *satisfies agreement, validity and, termination.*

See [Mag+19] for a proof.

> **i** The used delay is increased by `ABBADelayInc` after each failed phase and the protocol works for any positive choice of `ABBADelayInc`. The delay is crucial in theory where the adversary can reorder and selectively delay messages on the network. Setting `ABBADelayInc` equal to the expected network delivery time should work well in this setting. We conjecture much smaller values of `ABBADelayInc` still work in practice, where an adversary has limited control over the delays.

D.6.5 WMVBA Protocol

We can now describe the actual WMVBA protocol. Each party inputs a J-justified proposal and gets a $J_{\texttt{fin}}$-justified output which is either a proposal or $\bot$.

The idea of is WMVBA to first call Freeze to boil down the choice to a unique proposal or $\bot$. Parties then use ABBA to agree on either $\bot$ or (if it exists) the unique proposal.

Note that it can happen that an honest party decides on $\bot$ at the end of Freeze while ABBA outputs $\top$. In this case, at least one honest party had a justified non-$\bot$ decision as output in Freeze. This decision is unique. So we must ensure that honest parties with $\bot$ output in Freeze can somehow get their hands on that decision. For that, parties do not terminate Freeze once they get their output, but instead continue to collect decisions and vote-messages. By Lemma 138, this ensures that all honest parties will eventually receive the unique non-$\bot$ decision.

We define the following justification. First, we look at the justification for inputs of ABBA. The idea is that parties input $\bot$ (resp. $\top$) to ABBA if their $J_{\texttt{dec}}$-justified output of Freeze was $(\texttt{baid}, \textsc{Frozen}, \bot)$ (resp. $(\texttt{baid}, \textsc{Frozen}, \texttt{d})$ for $\texttt{d} \neq \bot$).

Definition 148. A bit $\texttt{b}$ is $J_{\texttt{in}}$-justified (*input justified*) for party P_i if P_i has a $J_{\texttt{dec}}$-justified tuple $(\texttt{baid}, \textsc{Frozen}, \texttt{d})$ where $\texttt{d} \neq \bot$ if and only if $\texttt{b} \neq \bot$.

Finally, we define the justification for outputs of WMVBA.

Definition 149. A decision $\texttt{d}$ is considered justified with respect to *final justification $J_{\texttt{fin}}$* for P_i if P_i has $t + 1$ signatures on the message $(\texttt{baid}, \textsc{WeAreDone}, \texttt{d})$.

Note that a $\bot$ output from ABBA is already $J_{\texttt{fin}}$-justified. The protocol formally works as follows:

Protocol WMVBA($\mathtt{baid}, J, \mathtt{ABBADelayInc}$)

The protocol is described from the view point of party P_i which has J-justified input $\mathtt{p}_i$.

1. Run Freeze($\mathtt{baid}, J$) with input $\mathtt{p}_i$. Denote by $\mathtt{d}_i$ the $J_{\mathtt{dec}}$-justified output for P_i from Freeze.

2. Run ABBA($\mathtt{baid}, J_{\mathtt{in}}, \mathtt{ABBADelayInc}$) with input $\mathtt{b}_i$ where $\mathtt{b}_i = \bot$ if $\mathtt{d}_i = \bot$ and $\mathtt{b}_i = \top$ otherwise. Denote by $\mathtt{b}_i'$ the output of ABBA for P_i.

3. If $\mathtt{b}_i' = \bot$, then terminate and output $\mathtt{b}_i'$ (which is $J_{\mathtt{fin}}$-justified) together with $\mathtt{W} = \bot$, otherwise (if $\mathtt{b}_i' = \top$) do:

   • Once P_i has a $J_{\mathtt{dec}}$-justified decision message ($\mathtt{baid}, \text{FROZEN}, \mathtt{d}_i$) with $\mathtt{d}_i \neq \bot$ (from Freeze) it sends signed ($\mathtt{baid}, \text{WEAREDONE}, \mathtt{d}_i$) to all other parties.

   • Once $t + 1$ signed ($\mathtt{baid}, \text{WEAREDONE}, \mathtt{d}$), for some $\mathtt{d}$, have been received, terminate and output ($\mathtt{d}, \mathtt{W}$), where $\mathtt{W}$ contains $\mathtt{baid}$ and $t + 1$ of these signatures.

Theorem 150. *For $t < \frac{n}{3}$ the protocol* WMVBA *satisfies consistency, weak validity, $n/3$-support, and termination.*

See [Mag+19] for a proof.

Appendix E

Kontrol Layer

The control layer Kontrol makes tweaks and parameters available to the other layers. The core idea is that (almost) all data can be computed from Skov.tree (this includes Skov.finEntries). Kontrol provides the necessary function to do these computations. The actual management of tweakable parameters is done in Renovatio (see Section 12.1.1).

E.1 Required Interfaces

E.1.1 Renovatio Interface

The Kontrol layer requires the following functions from Renovatio.

Current Parameter Function The function Renovatio.getCurrentParameter returns the current value of a parameter given a slot `slot`, and (the best chain) `chain`.

This provides access to the parameter update system of Renovatio. The ranges and initial values for all updatable parameters are formally defined in the genesis block (cf. Section 21). For completeness, we provide them here as well.

> ❶ We assume that the initial values of parameters are part of the genesis data. For more details on the genesis data see Section 21.

E.2 Time Functions

This section contains functions that allow a party to access time-related information such as the current slot.

E.2.1 Parameters

The time functions depend on the following two parameters. We use seconds as time unit (SI standard).

`genesisData.GenesisTime` $\in \mathbb{N}$
  The *protocol start timestamp* in seconds (since 1.1.1970).

  - **Initial Value:** The start time of the protocol.

Updatable parameters. The slot duration can be updated using the Renovatio mechanism.

`SlotDuration ∈ ℚ`
> The *slot duration* in seconds. This value can be a fractional, e.g., 0.5.
>
>   - **Initial Value:** 0.25 seconds.

E.2.2 Interface

The function Kontrol.getSlotAtTime takes as input a chain and a time. It returns the current slot according to the given time.

Function Kontrol.getSlotAtTime

Inputs

  chain ∈ CHAINS
>   A blockchain within Skov.tree.

  t ∈ ℚ
>   Time in seconds.

Outputs

  slot ∈ SLOTS
>   The slot at time t relative to chain. If a new slot starts exactly at time t, already returns the new slot.

Promise

  $t \geq$ genesisData.GenesisTime

Implementation

```
1: slot′ := 0
2: t′ := genesisData.GenesisTime            // Invariant: t′ is beginning of slot′.
3: while t′ ≤ t do
4:     SlotDuration′ := Renovatio.getCurrentParameter("SlotDuration", chain, slot′)
5:     t′ := t′ + SlotDuration′
6:     slot′ := slot′ + 1
7: Return slot′ − 1.                         // slot′ is first slot to start after time t.
```

> ❗ We assume that the time "belonging" to a slot is the half-open interval [beginning, beginning + SlotDuration). For (an unrealistic) example, if the genesis time is 0 and the slot duration is 1 second, the first slot (slot = 0) encompasses the time interval $[0, 1)$, the second one $[1, 2)$, etc.

> </> The sample implementation above retrieves the duration of every single slot and adds them all up. In the actual implementation, this should be done more efficiently by keeping state and taking into account that the slot duration changes only rarely.

The function Kontrol.getTimeOfSlot takes as input a chain and a slot. It returns the timestamp of the start of the slot. It is essentially the inverse of the function getSlotAtTime, except that it returns the smallest time within the slot.

Function Kontrol.getTimeOfSlot

Inputs

chain $\in$ CHAINS
> A blockchain within Skov.tree.

slot $\in$ SLOTS
> The slot for which we want to compute the start time.

Outputs

t $\in \mathbb{Q}$
> The timestamp of the start of slot.

Implementation

1: t := genesisData.GenesisTime
2: **for all** slot' < slot **do**   *// Slots start at 0*
3: $\quad$ SlotDuration' := Renovatio.getCurrentParameter("SlotDuration", chain, slot')
4: $\quad$ t := t + SlotDuration'
5: **Return** t.

E.3 Birk Functions

This section contains the Birk parameter functions which allows a party to get among other things the baker set or the leader election difficulty for a specific slot.

E.3.1 Parameters

The leader election nonce and the baker set are computed algorithmically. The initial values set in the genesis block.

genesisData.nonce$^{\text{LE}} \in$ NONCE$_{\text{LE}}$
> The initial *leader election nonce*.

> - **Initial Value:** Has to be unpredictable until right before the protocol start. It should also be a nothing-up-my-sleeve value. More details are found in Section 22.2.1.

genesisData.bakers $\in$ BAKERS*
> The initial *set of eligible bakers*.

> - **Initial Value:** Set according to the stake distribution in the genesis block. More details are found in Section 21.3.

Updatable parameters.   The following parameters can be set using Renovatio.

epochLength $\in \mathbb{N}$
> The *epoch length* in slots.

> - **Initial Value:** 3600

> - **Range:** Should not be too short to avoid epochs with no blocks (see diff$^{\text{LE}}$). Should not be too long to avoid nothing at stake attacks.

> - **Effect Constraint:** If updated during an epoch the new value only takes effect for the next epoch.

$\mathtt{diff^{LE}} \in \textsc{difficulty}$

> The *leader election difficulty*.

- **Initial Value:** 0.04

- **Update Constraint:** A potential constraint would be that the new difficulty must be within 10% of the old value.

> ⓘ The idea is to set $\mathtt{diff^{LE}}$ such that there is one slot winner on average every $c \cdot \Delta_{\mathtt{net}}$ for some small constant c. Finding the right hardness requires some experimentation. In the future, with more empiric and theoretical understanding of how to set it, Kontrol (via Kontrol.getLEDifficulty) could set it optimally according to current network behavior.

E.3.2 Interface

The function Kontrol.getLEDifficulty returns the leader election difficulty for a specific slot and chain.

Function Kontrol.getLEDifficulty

Inputs

$\quad \mathtt{slot} \in \textsc{slots}$

> The slot where the baker wants to be leader.

$\quad \mathtt{chain} \in \textsc{chains}$

> The chain that the baker intends to extend.

Outputs

$\quad \mathtt{diff^{LE}} \in \textsc{difficulty}$

> The leader election difficulty for the given slot and chain.

Implementation

1: **Return** Renovatio.getCurrentParameter("$\mathtt{diff^{LE}}$", $\mathtt{chain}$, $\mathtt{slot}$).

> </> In the current implementation, the difficulty can be set using a parameter update transaction (cf. Renovatio).

The function Kontrol.getEpochLength returns the length of the epoch containing a specific slot with respect to a given chain.

Function Kontrol.getEpochLength

Inputs

$\quad \mathtt{slot} \in \textsc{slots}$

> A slot contained in the epoch in question.

$\quad \mathtt{chain} \in \textsc{chains}$

> The chain used as point of reference.

Outputs

---

epochLength ∈ SLOTS
> The length of the epoch containing `slot` with respect to `chain`.

Implementation

1: Let `slot′` be the last slot of the previous epoch. *// Epoch length should not change in the middle of an epoch. We thus take the parameter from the previous epoch.*
2: **Return** Renovatio.getCurrentParameter("epochLength", chain, slot′).

</> In the current implementation, the epoch length is fixed in the genesis block.

Relative to Kontrol.getEpochLength we can define the following functions. The function epochs = splitIntoEpochs(chain) splits a chain into epochs. For example, epochs[0] contains the blocks of the first epoch. The function e = getEpoch(chain, slot) returns the epoch of a slot given chain `chain`. The output might be undefined if the slot is too far into the future. However, if getEpoch(chain, slot) ≠ ⊥ then getEpoch(chain′, slot) ≠ ⊥ for any prefix `chain` of `chain′`.

The function Kontrol.getBakerSet returns the set of bakers, including their keys and lottery power, for a specific slot and chain.

Function Kontrol.getBakerSet

Inputs

slot ∈ SLOTS
> The slot.

chain ∈ CHAINS
> The chain used as point of reference.

Outputs

bakers ∈ BAKERS*
> The set of eligible bakers in slot `slot` with respect to `chain`.

Promise

> `chain` is a valid, nonempty chain.
> `bakers` is the same for all slots in an epoch.

Implementation

1: **if** slot = 0 **then return** genesisData.bakers.
2: Let e = getEpoch(chain, slot).
3: Let epochs = splitIntoEpochs(chain).
4: Let `chain′` be the prefix of `chain` up to and including epochs[e − 2].
5: **Return** the state of bakers defined by the block `chain′`.

❗ To prevent 'nothing-at-stake' attacks the baker set (and their lottery power) should roughly represent the stake distribution at that epoch. In [Dav+17] the baker set in epoch i is chosen according to the stake distribution at the end of epoch $i − 2$.

The function Kontrol.getLENonce returns the leader election nonce for a specific slot and chain.

Function Kontrol.getLENonce

Inputs

  slot $\in$ SLOTS
        The slot where the baker wants to be leader.

  chain $\in$ CHAINS
        The chain used as point of reference.

Outputs

  $\mathtt{nonce}^{\mathrm{LE}} \in \mathrm{NONCE}_{\mathrm{LE}}$
        The leader election nonce for `slot` with respect to `chain`.

Promise

    `chain` is a valid, nonempty chain.

Implementation

1: $\mathtt{nonce}^{\mathrm{LE}} = \mathtt{genesisData.nonce}^{\mathrm{LE}}$
2: Let $\mathtt{e} = \mathsf{getEpoch}(\mathtt{chain}, \mathtt{slot})$.
3: Let $\mathtt{epochs} = \mathsf{splitIntoEpochs}(\mathtt{chain})$.
4: **for** $i = 0, \ldots, \mathtt{e} - 1$ **do**
5:     **for** each `block` in the first $\frac{2}{3}$ of $\mathtt{epochs}[i]$ (sorted according to slots) **do**
6:         $\mathtt{nonce}^{\mathrm{LE}} = \mathsf{H}(\mathtt{nonce}^{\mathrm{LE}}, \mathtt{block.nonce}^{\mathrm{LE}})$
7:     $\mathtt{nonce}^{\mathrm{LE}} = \mathsf{H}(\mathtt{nonce}^{\mathrm{LE}}, i)$ *// update nonce even if epoch is empty*
8: **return** $\mathtt{nonce}^{\mathrm{LE}}$.

> ❗ The length of epochs should be chosen such that epochs can safely be assumed to have many honest blocks in the first $\frac{2}{3}$ slots. The step $\mathtt{nonce}^{\mathrm{LE}} = \mathsf{H}(\mathtt{nonce}^{\mathrm{LE}}, i)$ ensures that the nonce always gets updated, even if the epoch contains no blocks. This step is not strictly necessary for security, since it cannot provide real security in that scenario. That is because an attacker can manipulate the nonces if there are no honest blocks included in the nonce computation.

E.4 Afgjort Functions

This section contains the Afgjort parameter function which defines the finalization committee for a given finalization instance.

E.4.1 Parameters

The set of finalizers is computed algorithmically. The initial set is defined in the genesis block.

$\mathtt{genesisData.finCom} \in \mathrm{FINALIZERS}^*$
    The initial set of eligible finalizers.

        • **Initial Value:** Set according to the stake distribution in the genesis block. More details are found in Section 21.

Updatable parameters. The following parameters can be set using Renovatio.

`minSkip` $\in \mathbb{N}+$

    The initial minimum skip for finalization.

- **Initial Value:** 1

  Can be set to a higher value (e.g., 5) if one wants to save communication complexity by running finalization less frequently.

- **Range:** $[1, 50]$

  Note that too large values make finalization ineffective.

`ABBADelayInc` $\in \mathbb{Q}$

    The time in seconds by which the waiting delay in ABBA is increased after each failed phase.

- **Initial Value:** 0.1

- **Range:** $(0, 2]$

`skipShrinkFactor` $\in \mathbb{Q}$

    The factor used to decrease the skip for the finalization depth.

- **Initial Value:** 0.7

- **Range:** $(0, 1)$

`skipGrowFactor` $\in \mathbb{Q}$

    The factor used to increase the skip for the finalization depth.

- **Initial Value:** 2

- **Range:** $(1, 100]$

`bDelShrinkFactor` $\in \mathbb{Q}$

    The factor used to decrease the block delay for finalization.

- **Initial Value:** 0.7

- **Range:** $(0, 1)$

`bDelGrowFactor` $\in \mathbb{Q}$

    The factor used to increase the block delay for finalization.

- **Initial Value:** 2

- **Range:** $(1, 100]$

E.4.2 Interface

The function Kontrol.getFinalizationMinimumSkip takes as input a sequence number `ctr`, a chain `chain` containing all finalized blocks up to finalization `ctr − 1`, and the corresponding set of finalization entries `finEntries`. It returns the minimum skip for the depth of finalization `ctr`.

Function Kontrol.getFinalizationMinimumSkip

Inputs

  `ctr` $\in \mathbb{N}$

      The sequence number of the finalization.

> chain ∈ CHAINS
>> A chain containing all finalized blocks up to $\mathtt{ctr} - 1$.
>
> finEntries ∈ FINALENTRIES*
>> The set of finalization entries for all finalized blocks in chain.

Outputs

> minSkip ∈ ℕ
>> The minimum skip for the depth of finalization ctr.

Promise

> Returns the updated minimum skip parameter considering all updates up to the last finalized block. That is, parameter updates taking effect after the last finalized block are not considered for this finalization.

Implementation

1: Let chain′ be the prefix of chain up to the last block pointed to in finEntries.
2: Let slot be the slot number of the last block pointed to in finEntries.
3: **Return** Renovatio.getCurrentParameter("minSkip", chain′, slot).

---

❶ Ignoring parameter updates after the last finalized block ensures consistency and agreement about the parameter for each finalization.

The function Kontrol.getABBADelayInc returns the time by which the delay in AꓭBA is increased in each phase.

Function Kontrol.getABBADelayInc

Inputs

> ctr ∈ ℕ
>> The sequence number of the finalization.
>
> chain ∈ CHAINS
>> A chain containing all finalized blocks up to $\mathtt{ctr} - 1$.
>
> finEntries ∈ FINALENTRIES*
>> The set of finalization entries for all finalized blocks in chain.

Outputs

> ABBADelayInc ∈ ℚ
>> The time in seconds by which the delay in AꓭBA is increased in each phase for finalization ctr.

Promise

> Returns the updated parameter considering all updates up to the last finalized block. That is, parameter updates taking effect after the last finalized block are not considered for this finalization.

Implementation

1: Let chain′ be the prefix of chain up to the last block pointed to in finEntries.
2: Let slot be the slot number of the last block pointed to in finEntries.

The function Kontrol.getFinSkipFactors takes as input a sequence number `ctr`, a chain `chain` containing all finalized blocks up to finalization `ctr − 1`, and the corresponding set of finalization entries `finEntries`. It returns the factors used to decrease and increase the skip for the depth of finalization `ctr`.

Function Kontrol.getFinSkipFactors

Inputs

`ctr` $\in \mathbb{N}$
> The sequence number of the finalization.

`chain` $\in$ CHAINS
> A chain containing all finalized blocks up to `ctr − 1`.

`finEntries` $\in$ FINALENTRIES*
> The set of finalization entries for all finalized blocks in `chain`.

Outputs

`skipShrinkFactor` $\in \mathbb{Q}$
> The factor used to decrease the skip for the depth of finalization `ctr`.

`skipGrowFactor` $\in \mathbb{Q}$
> The factor used to increase the skip for the depth of finalization `ctr`.

Promise

> Returns the updated parameters considering all updates up to the last finalized block. That is, parameter updates taking effect after the last finalized block are not considered for this finalization.

Implementation

1: Let `chain'` be the prefix of `chain` up to the last block pointed to in `finEntries`.
2: Let `slot` be the slot number of the last block pointed to in `finEntries`.
3: `skipShrinkFactor` := Renovatio.getCurrentParameter("skipShrinkFactor", chain', slot).
4: `skipGrowFactor` := Renovatio.getCurrentParameter("skipGrowFactor", chain', slot).
5: **Return** (`skipShrinkFactor`, `skipGrowFactor`).

Similarly, the function Kontrol.getFinBDelFactors returns the factors used to decrease and increase the block delay for finalization.

Function Kontrol.getFinBDelFactors

Inputs

`ctr` $\in \mathbb{N}$
> The sequence number of the finalization.

`chain` $\in$ CHAINS
> A chain containing all finalized blocks up to `ctr − 1`.

`finEntries` $\in$ FINALENTRIES*
> The set of finalization entries for all finalized blocks in `chain`.

Outputs

bDelShrinkFactor $\in \mathbb{Q}$

> The factor used to decrease the block delay for finalization ctr.

bDelGrowFactor $\in \mathbb{Q}$

> The factor used to increase the block delay for finalization ctr.

Promise

> Returns the updated parameters considering all updates up to the last finalized block. That is, parameter updates taking effect after the last finalized block are not considered for this finalization.

Implementation

1: Let chain′ be the prefix of chain up to the last block pointed to in finEntries.
2: Let slot be the slot number of the last block pointed to in finEntries.
3: bDelShrinkFactor := Renovatio.getCurrentParameter("bDelShrinkFactor", chain′, slot).
4: bDelGrowFactor := Renovatio.getCurrentParameter("bDelGrowFactor", chain′, slot).
5: **Return** (bDelShrinkFactor, bDelGrowFactor).

In Afgjort blocks are finalized in regular intervals. For each finalization the *finalization committee* has to be selected. The function Kontrol.getFinalizationCommittee takes as input a sequence number ctr, a chain chain containing all finalized blocks up to finalization ctr − 1, and the corresponding set of finalization entries finEntries. It returns the finalization committee for finalization ctr.

Function Kontrol.getFinalizationCommittee

Inputs

ctr $\in \mathbb{N}$

> The sequence number of the finalization.

chain $\in$ CHAINS

> A chain containing all finalized blocks up to ctr − 1.

finEntries $\in$ FINALENTRIES*

> The set of finalization entries for all finalized blocks in chain.

Outputs

finCom $\in$ FINALIZERS*

> The set of finalizers eligible to finalize for finalization ctr.

Promise

> The function is monotone in the chain input. That is for fixed ctr, and finEntries and for two chains chain, chain′ that both contain all finalized blocks up to ctr − 1 the output is the same.

Implementation

1: Set c = 1.   *// See comments below.*
2: **if** ctr ≤ c **then**
3:     **return** genesisData.finCom   *// The genesis finalization committee.*
4: Let chain′ be the chain up to the ctr − c finalized block.
5: **return** The finalization committee defined by the state of chain′.

To prevent 'nothing-at-stake' attacks the finalization committee (and their voting power) should roughly represent the stake distribution at the depth of the `ctr` finalization. This can be ensured by using small values for `c` in the sample implementation.

Too small values of `c` in the sample implementation might not allow finalizers to prepare in advance.

In the implementation we set $c = 1$. Any party which has at least 0.1% of the total stake delegated in the state of `chain'` is part of the finalization committee.

E.5 Chain Validation

This section contains the Kontrol.isValidChain which defines the validity of blockchains.

E.5.1 Required Functions

The chain validation requires the function data validation function `checkdata`.

Function checkdata

Inputs

   chainData $\in$ DATA*

      A sequence of data blocks

Outputs

   b $\in$ BOOL

      The output b is true if `chainData` is a valid sequence of data.

This function abstracts the data validation. The implementation of this function highly depends on the actual data layer of the project. See also Section C.2.2.

E.5.2 Chain Validity Predicate

In this section we define the chain validity predicate.

Definition 151. A blockchain $\texttt{chain} = \texttt{block}_0, \ldots, \texttt{block}_n$ is *valid* with respect to block tree `tree` and finalization entries `finEntries` if

- The following chain properties hold:
    - The chain starts with the genesis block `genesis` of `tree`, i.e., $\texttt{block}_0 = \texttt{genesis}$.
    - The slot numbers are strictly increasing, i.e., $\texttt{block}_i.\texttt{slot} > \texttt{block}_j.\texttt{slot}$ for $i > j$.
    - The blocks are properly linked, i.e., $\texttt{block}_i.\texttt{ptr} = \mathsf{H}(\texttt{block}_{i-1})$ for all $i > 0$.
    - The finalization entries have consecutive counters, i.e., if $\texttt{block}_i.\texttt{nextFinEntry} \neq \bot$ and $\texttt{block}_j$ is the deepest ancestor of $\texttt{block}_i$ with $\texttt{block}_j.\texttt{nextFinEntry} \neq \bot$, then $\texttt{block}_i.\texttt{nextFinEntry}.\texttt{ctr} = \texttt{block}_j.\texttt{nextFinEntry}.\texttt{ctr} + 1$.

– The chain data is valid, i.e., $\mathsf{checkdata}(\mathtt{chain.chainData}) = \mathtt{true}$.

- For any block except the genesis block $\mathtt{genesis}$ the following holds.

  For $i > 0$ let $\mathtt{block}_i = (\mathtt{slot}, \mathtt{ptr}, \mathtt{bid}, \mathtt{proof}, \mathtt{nonce}, \mathtt{nextFinEntry}, \mathtt{data}, \sigma^{\mathrm{BC}})$. Consider the following values computed from $\mathtt{block}_i$:

    – Let $\mathtt{diff}^{\mathrm{LE}} = \mathsf{Kontrol.getLEDifficulty}(\mathtt{slot}, \mathtt{chain})$ the leader election difficulty.

    – Let $\mathtt{nonce}^{\mathrm{LE}} = \mathsf{Kontrol.getLENonce}(\mathtt{slot}, \mathtt{chain})$ be the leader election nonce.

    – Let $\mathtt{vk}^{\mathrm{LE}} = \mathsf{Kontrol.getBakerSet}(\mathtt{slot}, \mathtt{chain})[\mathtt{bid}].\mathtt{vk}^{\mathrm{LE}}$ be the VRF verification key of the baker.

    – Let $\mathtt{vk}^{\mathrm{BC}} = \mathsf{Kontrol.getBakerSet}(\mathtt{slot}, \mathtt{chain})[\mathtt{bid}].\mathtt{vk}^{\mathrm{BC}}$ be the signature verification key of the baker.

    – Let $\mathtt{lotteryPower} = \mathsf{Kontrol.getBakerSet}(\mathtt{slot}, \mathtt{chain})[\mathtt{bid}].\mathtt{lotteryPower}$ be the lottery power of the baker.

  The following must hold:

    – The block contains a valid block proof, i.e.,

    $$\mathsf{Birk.verifyProof}(\mathtt{slot}, \mathtt{nonce}^{\mathrm{LE}}, \mathtt{diff}^{\mathrm{LE}}, \mathtt{lotteryPower}, \mathtt{vk}^{\mathrm{LE}}, \mathtt{proof}) = \mathtt{true}.$$

    – The block contains a valid block nonce, i.e.,

    $$\mathsf{Birk.verifyBlockNonce}(\mathtt{slot}, \mathtt{nonce}^{\mathrm{LE}}, \mathtt{vk}^{\mathrm{LE}}, \mathtt{nonce}) = \mathtt{true}.$$

    – If $\mathtt{nextFinEntry} \neq \bot$, then $\mathtt{nextFinEntry}$ is a valid finalization entry according to Definition 128.

    – The block is properly signed, i.e.,

    $$\mathsf{Verify}_{\mathrm{BC}}(\mathtt{vk}^{\mathrm{BC}}, (\mathtt{slot}, \mathtt{ptr}, \mathtt{bid}, \mathtt{proof}, \mathtt{nonce}, \mathtt{nextFinEntry}, \mathtt{data}), \sigma^{\mathrm{BC}}) = \mathtt{true}.$$

> ❗ In practice, it makes sense to *limit* the size of a valid block for efficiency reasons. This might be combined with a gas/energy limit where transaction fees take the actual size of the transaction into account.

E.5.3 Chain Validation Function

The function $\mathsf{Kontrol.isValidChain}$ returns the validity of a chain with respect to Definition 151.

Function $\mathsf{Kontrol.isValidChain}$

Inputs

$\mathsf{Skov.tree} \in \textsc{trees}$
   The block tree maintained by Skov.

$\mathsf{Skov.finEntries} \in \textsc{finalentries}^*$
   The finalization entry list maintained by Skov.

$\mathtt{chain} \in \textsc{chains}$
   A blockchain.

Outputs

    b ∈ BOOL

        The output `b` is true if `chain` is a valid chain with respect to `Skov.tree` and `Skov.finEntries` (cf. Definition 151).

E.6   Best Chain Selection

This section contains the bests chain selection function Kontrol.BestChain.

E.6.1   Block Luck Function

As part of the tie-breaking strategy for BestChain, we use the concept of *block luck*. The luck of a block is a real value between 0 and 1 which describes how lucky the baker was in becoming slot leader. A value of 1 means that the baker won the election easily with large margin while a value of 0 means that the baker just barely won.

Function Kontrol.blockluck

Inputs

    Skov.tree ∈ TREES

        The block tree maintained by Skov.

    Skov.finEntries ∈ FINALENTRIES*

        The finalization entry list maintained by Skov.

    chain ∈ CHAINS

        A blockchain.

    block ∈ BLOCKS

        A block in chain `chain`.

Outputs

    $\ell \in [0, 1]$

        The *luck* of block `block`.

Implementation

1: **if** `block` is the genesis block `genesis` **then**
2:     **return** 1.
3: **else**
4:     **return** $1 - \frac{h}{t}$ where h is the leader election VRF output for `block` and `t` the threshold for winning the leader election (for the baker of `block`).

> **ⓘ**   Details on how to compute h and `t` are found in the implementation of Birk.verifyProof in Section C.3.3.

E.6.2   Lexicographic Ordering of Chains

To select the "best chain", we introduce a lexicographic order of chains with respect to `Skov.tree` and `Skov.finEntries` as described below.

For a chain $\texttt{chain} = \texttt{genesis}, \texttt{block}_1, \ldots, \texttt{block}_n$ consider tuple $(f, l, -s, b, h)$ where

- f is the number of finalized blocks on the chain (according to $\mathsf{Skov.finEntries}$),

- l is the length of the $\texttt{chain}$, i.e., $l = n + 1$,

- s is the slot number of the last block, i.e., $s = \texttt{block}_n.\texttt{slot}$,

- b is the luck of the last block, i.e., $b = \mathsf{Kontrol.blockluck}(\mathsf{Skov.tree}, \mathsf{Skov.finEntries}, \texttt{chain}, \texttt{block}_n)$,

- h is the hash of the last block, i.e., $h = \mathsf{H}(\texttt{block}_n)$.

Definition 152. A chain $\texttt{chain}_1$ with tuple $(f_1, l_1, -s_1, b_1, h_1)$ is *better* than $\texttt{chain}_2$ with tuple $(f_2, l_2, -s_2, b_2, h_2)$, denoted $\texttt{chain}_1 > \texttt{chain}_2$ if $(f_1, l_1, -s_1, b_1, h_1)$ is lexicographic greater than $(f_2, l_2, -s_2, b_2, h_2)$ where in each component the usual ordering is used.

> ❗ Note that the slot number has a negative sign, which means for the lexicographic ordering that smaller slot numbers are preferred.

> ℹ️ Preferring larger slot numbers instead of smaller ones has the following issue: When a malicious party wins a slot, they can "ignore" the last (honest) block in the chain and create a small fork. Since the new fork is equally long and the last block is newer (i.e., has a larger slot number) than the one containing the honest block, the malicious chain would be preferred. That is, an attacker could always eliminate the last block from the chain whenever they win a slot. This issue does not exist when preferring the oldest block.

E.6.3 Best Chain Algorithm

The function $\mathsf{Kontrol.BestChain}$ returns the best chain in $\mathsf{Skov.tree}$ according to Definition 152.

Function $\mathsf{Kontrol.BestChain}$

Inputs

  $\mathsf{Skov.tree} \in \text{TREES}$
    The block tree maintained by Skov.
  $\mathsf{Skov.finEntries} \in \text{FINALENTRIES}^*$
    The finalization entry list maintained by Skov.

Outputs

  $\texttt{chain} \in \text{CHAINS}$
    The best chain in $\mathsf{Skov.tree}$ according to the ordering from Definition 152.

> ❗ Note that this algorithm is deterministic. This ensures that two parties with the same local tree will select the same best chain. The Definition 152 makes it highly likely that there exists a unique best chain in $\mathsf{Skov.tree}$ (otherwise a collision in the hash function was found).

Appendix F

Proofs

F.1 ABBA Lottery

Lemma 153. *Assuming the VRF behaves like a random oracle, we have the following properties:*

1. *Assume some party has lottery power $\sum_{i=1}^{k} \texttt{lotteryPower}_i$ and let p be the probability that this party has the highest lottery number. If this party is split into k sub-parties with lottery powers $\texttt{lotteryPower}_1, \ldots, \texttt{lotteryPower}_k$, respectively, the probability that any of these sub-parties has the highest lottery number is equal to $p + \epsilon$ for some negligible ϵ.*

2. *If party P_2 has twice the lottery power of P_1, then P_2 has a higher lottery number than P_1 with probability $\frac{2}{3} - \epsilon$ for some negligible ϵ.*

Proof. We first analyze the probability that the lottery number is at most some given value. To this end, let $\texttt{lotteryPower} \in (0,1]$ and let h and n be the random variables corresponding to the VRF hash and lottery number generated by the algorithm ABBALottery, respectively. We then have for all $m \in (0,1]$,

$$\Pr[n \leq m] = \Pr\left[\left(2^{-\ell_{\mathrm{VRF}}}(h+1) \right)^{\frac{1}{\texttt{lotteryPower}}} \leq m \right] = \Pr\left[h + 1 \leq 2^{\ell_{\mathrm{VRF}}} \cdot m^{\texttt{lotteryPower}} \right]$$
$$= \frac{\left\lfloor 2^{\ell_{\mathrm{VRF}}} \cdot m^{\texttt{lotteryPower}} \right\rfloor}{2^{\ell_{\mathrm{VRF}}}}.$$

The last equality holds because $h + 1$ is a uniform value in $\left\{ 1, \ldots, 2^{\ell_{\mathrm{VRF}}} \right\}$ and, since $2^{\ell_{\mathrm{VRF}}} \cdot m^{\texttt{lotteryPower}} \leq 2^{\ell_{\mathrm{VRF}}}$, there are $\left\lfloor 2^{\ell_{\mathrm{VRF}}} \cdot m^{\texttt{lotteryPower}} \right\rfloor$ values less than or equal to $2^{\ell_{\mathrm{VRF}}} \cdot m^{\texttt{lotteryPower}}$ in that set. Hence, there is some $\epsilon \in [0, 2^{-\ell_{\mathrm{VRF}}})$ such that

$$\Pr[n \leq m] = m^{\texttt{lotteryPower}} - \epsilon. \tag{F.1}$$

We now prove the two claims of the lemma.

1. Let n_Σ be the random variable denoting the lottery number of the party with lottery power $\sum_{i=1}^{k} \texttt{lotteryPower}_i$ and let n_i denote the corresponding random variable for the ith sub-party. Further, let m be the highest lottery number among all other parties. The

probability that any of the sub-parties has the highest number is

$$\Pr\left[\bigvee_{i=1}^{k} n_i > m\right] = 1 - \Pr\left[\bigwedge_{i=1}^{k} n_i \leq m\right]$$

$$= 1 - \prod_{i=1}^{k} \Pr[n_i \leq m]$$

$$\stackrel{\text{(F.1)}}{=} 1 - \prod_{i=1}^{k} \left(m^{\texttt{lotteryPower}_i} - \epsilon_i\right)$$

$$= 1 - m^{\sum_{i=1}^{k} \texttt{lotteryPower}_i} + \hat{\epsilon}$$

$$\stackrel{\text{(F.1)}}{=} 1 - \Pr[n_\Sigma \leq m] + \epsilon'.$$

The last line equals $\Pr[n_\Sigma > m] + \epsilon'$, which concludes the proof of the first claim.

2. Let n_1 and n_2 denote the random variables corresponding to the lottery numbers of P_1 and P_2, respectively. Further, let n_2' and n_2'' be independent random variables with the same distribution as n_1. Because P_2 has twice the lottery power of P_1, we have, as shown in the first claim, that for all $m \in (0,1]$, $\Pr[n_2 > m] = \Pr[\max\{n_2', n_2''\} > m] - \epsilon'$. Hence,

$$\Pr[n_2 > n_1] = \sum_m \Pr[n_1 = m] \cdot \Pr[n_2 > m]$$

$$= \sum_m \Pr[n_1 = m] \cdot (\Pr[\max\{n_2', n_2''\} > m] - \epsilon')$$

$$= \sum_m \Pr[n_1 = m \wedge \max\{n_2', n_2''\} > n_1] - \epsilon' \cdot \sum_m \Pr[n_1 = m]$$

$$= \Pr[\max\{n_2', n_2''\} > n_1] - \epsilon'.$$

Now let D be the event that n_2', n_2'', and n_1 are distinct. We have

$$\Pr[\neg D] = \Pr[n_2' = n_2'' \vee n_2' = n_1 \vee n_2'' = n_1] \leq 3 \cdot \Pr[n_2' = n_2''] = 3 \cdot 2^{-\ell_{\text{VRF}}},$$

where the last equality follows from the fact that all values of n_2' are equally likely. Thus,

$$\Pr[\max\{n_2', n_2''\} > n_1] \geq \Pr[\max\{n_2', n_2''\} > n_1 \wedge D]$$

$$= \Pr[\max\{n_2', n_2''\} > n_1 \mid D] \cdot \Pr[D]$$

$$\geq \Pr[\max\{n_2', n_2''\} > n_1 \mid D] \cdot (1 - 3 \cdot 2^{-\ell_{\text{VRF}}})$$

$$\geq \Pr[\max\{n_2', n_2''\} > n_1 \mid D] - 3 \cdot 2^{-\ell_{\text{VRF}}}.$$

The random variables n_2', n_2'', and n_1 are independent and identically distributed. Conditioned on them being distinct, one can order them in $3! = 6$ different ways, each of which is equally likely. In 4 out of these 6 cases, $\max\{n_2', n_2''\} > n_1$. Hence, $\Pr[\max\{n_2', n_2''\} > n_1 \mid D] = \frac{2}{3}$. We can therefore conclude that

$$\Pr[n_2 > n_1] \geq \frac{2}{3} - \epsilon' - 3 \cdot 2^{-\ell_{\text{VRF}}}. \qquad \square$$

Bibliography

[AC20]   Thomas Attema and Ronald Cramer. *Compressed Σ-Protocol Theory and Practical Application to Plug & Play Secure Algorithmics.* Cryptology ePrint Archive, Report 2021/1377. 2020. URL: https://eprint.iacr.org/2020/152.

[AF21]   Thomas Attema and Serge Fehr. "Parallel Repetition of $(k_1, \ldots, k_\mu)$-Special-Sound Multi-Round Interactive Proofs". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1259. URL: https://eprint.iacr.org/2021/1259.

[AFK21]   Thomas Attema, Serge Fehr, and Michael Klooß. *Fiat-Shamir Transformation of Multi-Round Interactive Proofs.* Cryptology ePrint Archive, Report 2021/1377. 2021. URL: https://eprint.iacr.org/2021/1377.

[Ber+12]   Daniel J. Bernstein et al. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* 2.2 (Sept. 1, 2012), pp. 77–89. DOI: 10.1007/s13389-012-0027-1. URL: https://doi.org/10.1007/s13389-012-0027-1.

[Ber06]   Daniel Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography.* Vol. 3958. Apr. 2006, pp. 207–228. DOI: 10.1007/11745853_14.

[Bon+03]   Dan Boneh et al. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: *Proceedings of the 22nd International Conference on Theory and Applications of Cryptographic Techniques, EUROCRYPT'03.* EUROCRYPT'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 416–432.

[Bon+20]   Dan Boneh et al. *BLS Signatures.* Internet-Draft draft-irtf-cfrg-bls-signature-02. Work in Progress. Internet Engineering Task Force, Mar. 2020. 30 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-02.

[Bow17]   Sean Bowe. *BLS12-381: New zk-SNARK elliptic curve construction.* 2017. URL: https://electriccoin.co/blog/new-snark-curve/.

[Bün+18]   Benedikt Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy (SP).* 2018, pp. 315–334.

[CDS94]   Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. "Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols". In: *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings.* Vol. 839. Lecture Notes in Computer Science. Springer, 1994, pp. 174–187. DOI: 10.1007/3-540-48658-5_19.

[Cha+09]   Sanjit Chatterjee et al. *Comparing Two Pairing-Based Aggregate Signature Schemes.* Cryptology ePrint Archive, Report 2009/060. 2009. URL: https://eprint.iacr.org/2009/060.

[CHK23]   Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. "Curve Trees: Practical and Transparent Zero-Knowledge Accumulators". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, Aug. 2023, pp. 4391–4408. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/campanelli.

[CM09]   Sanjit Chatterjee and Alfred Menezes. *On Cryptographic Protocols Employing Asymmetric Pairings – The Role of Ψ Revisited*. Cryptology ePrint Archive, Report 2009/480. 2009. URL: https://eprint.iacr.org/2009/480.

[Con23a] Team Concordium. *CIS-4: Credential Registry Standard*. Concordium Standard. Concordium, Oct. 2023. URL: https://github.com/Concordium/concordium-update-proposals/blob/main/source/CIS/cis-4.rst.

[Con23b] Team Concordium. *Concordium DID Method Version 1.0*. Concordium Standard. Concordium, June 2023. URL: https://github.com/Concordium/concordium-update-proposals/blob/main/source/ID/concordium-did.rst.

[Con23c] Team Concordium. *concordium-web3id*. Concordium Standard. Concordium, Oct. 2023. URL: https://github.com/Concordium/concordium-web3id.

[Con24]   Concordium. *Concordium White Paper*. 2024. URL: https://developer.concordium.software/governance/whitepaper/Concordium%20White%20Paper.pdf.

[Con25]   Team Concordium. *Concordium Tokenomics System*. Concordium One Pager. Concordium, July 2025. URL: https://docs.concordium.com/governance/tokenomics/TokenomicsOnePager.pdf.

[CPS18]   T-H. Hubert Chan, Rafael Pass, and Elaine Shi. *PaLa: A Simple Partially Synchronous Blockchain*. Cryptology ePrint Archive, Paper 2018/981. 2018. URL: https://eprint.iacr.org/2018/981.

[Dam+21] Ivan Damgård et al. "Balancing Privacy and Accountability in Blockchain Identity Management". In: *Topics in Cryptology – CT-RSA 2021*. Ed. by Kenneth G. Paterson. Springer International Publishing, 2021, pp. 552–576. ISBN: 978-3-030-75539-3.

[Dam10]   Ivan Damgård. *On sigma-protocols*. Lecture Notes, Aarhus University, Department of Computer Science. 2010. URL: https://www.cs.au.dk/~ivan/Sigma.pdf.

[Dav+17] Bernardo David et al. *Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol*. Cryptology ePrint Archive, Report 2017/573. 2017. URL: https://eprint.iacr.org/2017/573.

[Div15]   NIST Computer Security Division. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS Publication 202. National Institute of Standards and Technology, U.S. Department of Commerce, Aug. 2015. URL: http://dx.doi.org/10.6028/NIST.FIPS.202.

[DY05]   Yevgeniy Dodis and Aleksandr Yampolskiy. "A Verifiable Random Function with Short Proofs and Keys". In: *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*. Ed. by Serge Vaudenay. Vol. 3386. Lecture Notes in Computer Science. Springer, Jan. 2005, pp. 416–431. DOI: 10.1007/978-3-540-30580-4_28. URL: https://doi.org/10.1007/978-3-540-30580-4%5C_28.

[Elg85]   T. Elgamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE Transactions on Information Theory* 31.4 (July 1985), pp. 469–472. ISSN: 1557-9654. DOI: 10.1109/TIT.1985.1057074.

[Faz+20]   Armando Faz-Hernandez et al. *Hashing to Elliptic Curves*. Internet-Draft draft-irtf-cfrg-hash-to-curve-09. Work in Progress. Internet Engineering Task Force, June 2020. 158 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-09.

[FS86]   Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194. DOI: 10.1007/3-540-47721-7_12.

[Gan+24]   Chaya Ganesh et al. "Fiat–Shamir Bulletproofs are Non-malleable (in the Random Oracle Model)". In: *Journal of Cryptology* 38.1 (Dec. 5, 2024), p. 11. DOI: 10.1007/s00145-024-09525-2. URL: https://doi.org/10.1007/s00145-024-09525-2.

[Gel+21]   Rati Gelashvili et al. "Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback". In: *CoRR* abs/2106.10362 (June 29, 2021). arXiv: 2106.10362. URL: https://arxiv.org/abs/2106.10362.

[GMR85]   Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)". In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*. Ed. by Robert Sedgewick. ACM, May 1985, pp. 291–304. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178.

[Gol+20]   Sharon Goldberg et al. *Verifiable Random Functions (VRFs)*. Internet-Draft draft-irtf-cfrg-vrf-07. Work in Progress. Internet Engineering Task Force, June 2020. 39 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-07.

[Gol+23]   Sharon Goldberg et al. *Verifiable Random Functions (VRFs)*. RFC 9381. Aug. 2023. DOI: 10.17487/RFC9381. Request for Comments: 9381. URL: https://www.rfc-editor.org/info/rfc9381.

[Gri25]   Jack Grigg. *pairing 0.15.0, BLS12-381 implementation*. 2025. eprint: https://docs.rs. URL: https://docs.rs/crate/pairing/0.15.0/source/src/bls12_381/README.md.

[HR16]   Jochen Hoenicke and Pavol Rusnak. *SLIP-10: Universal private key derivation from master private key*. 2016. URL: https://github.com/satoshilabs/slips/blob/master/slip-0010.md.

[JL17]   Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032. Jan. 2017. DOI: 10.17487/RFC8032. Request for Comments: 9381. URL: https://rfc-editor.org/rfc/rfc8032.txt.

[KL20]   J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2020. ISBN: 9781351133029. URL: https://www.cs.umd.edu/~jkatz/imc.html.

[Lam79]   Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. CSL-98. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. Stanford Research Institute, Oct. 1979. URL: https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[Mag+19]   Bernardo Magri et al. *Afgjort: A Partially Synchronous Finality Layer for Blockchains*. Cryptology ePrint Archive, Report 2019/504. 2019. URL: https://eprint.iacr.org/2019/504.

[Mau15]   Ueli Maurer. "Zero-knowledge proofs of knowledge for group homomorphisms". In: *Designs, Codes and Cryptography* 77.2-3 (2015), pp. 663–676. DOI: 10.1007/s10623-015-0103-5.

[NIS15]   NIST. *Secure Hash Standard (SHS)*. FIPS Publication 180-4. Aug. 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[NIS17]   NIST. *NIST Post-Quantum Cryptography*. 2017. URL: https://csrc.nist.gov/projects/post-quantum-cryptography.

[Ove09]   Raphael Overbeck. *A Step Towards QC Blind Signatures*. Cryptology ePrint Archive, Paper 2009/102. 2009. URL: https://eprint.iacr.org/2009/102.

[Pal+13]   Marek Palatinus et al. *BIP-39: Mnemonic code for generating deterministic keys*. 2013. URL: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

[Pap+17]   Dimitrios Papadopoulos et al. *Making NSEC5 Practical for DNSSEC*. Cryptology ePrint Archive, Report 2017/099. 2017. URL: https://eprint.iacr.org/2017/099.

[Ped25]   Torben Pryds Pedersen. *My biggest secret*. Private communication. Apr. 1, 2025.

[Ped92]   Torben Pryds Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140. ISBN: 978-3-540-46766-3.

[PR14]   Marek Palatinus and Pavol Rusnak. *BIP-44: Multi-Account Hierarchy for Deterministic Wallets*. 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

[PS15]   David Pointcheval and Olivier Sanders. *Short Randomizable Signatures*. Cryptology ePrint Archive, Report 2015/525. 2015. URL: https://eprint.iacr.org/2015/525.

[PSM17]   Albrecht Petzoldt, Alan Szepieniec, and Mohamed Saied Emam Mohamed. *A Practical Multivariate Blind Signature Scheme*. Cryptology ePrint Archive, Paper 2017/131. 2017. URL: https://eprint.iacr.org/2017/131.

[RP14]   Pavol Rusnak and Marek Palatinus. *SLIP-0044 : Registered coin types for BIP-0044*. 2014. URL: https://github.com/satoshilabs/slips/blob/master/slip-0044.md.

[Rüc08]   Markus Rückert. *Lattice-based Blind Signatures*. Cryptology ePrint Archive, Paper 2008/322. 2008. URL: https://eprint.iacr.org/2008/322.

[SGS22]   Manu Sporny, Amy Guy, and Markus Sabadello. *Decentralized Identifiers (DIDs) v1.0*. W3C Recommendation. W3C, July 2022. URL: https://www.w3.org/TR/did-core/.

[Sha79]   Adi Shamir. "How to Share a Secret". In: *Communications of the ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[Spo+24]   Manu Sporny et al. *Verifiable Credentials Data Model v2.0*. W3C Recommendation. W3C, July 2024. URL: https://www.w3.org/TR/vc-data-model-2.0/.

[Vos91]   Michael D. Vose. "A linear algorithm for generating random numbers with a given distribution". In: *IEEE Transactions on Software Engineering* 17.9 (1991), pp. 972–975. DOI: 10.1109/32.92917.

[Wal77]   Alastair J. Walker. "An Efficient Method for Generating Discrete Random Variables with General Distributions". In: *ACM Trans. Math. Softw.* 3.3 (Sept. 1977), pp. 253–256. ISSN: 0098-3500. DOI: 10.1145/355744.355749.

[WB19]   Riad S. Wahby and Dan Boneh. *Fast and simple constant-time hashing to the BLS12-381 elliptic curve*. Cryptology ePrint Archive, Report 2019/403. 2019. URL: https://eprint.iacr.org/2019/403.

[Wik20]   Wikipedia contributors. *Nothing-up-my-sleeve number*. Wikipedia, The Free Encyclopedia. 2020. URL: https://en.wikipedia.org/w/index.php?title=Nothing-up-my-sleeve_number&oldid=935008599.

[Wik25]   Wikipedia. *Variable-length quantity*. 2025. eprint: Wikipedia. URL: https://en.wikipedia.org/wiki/Variable-length_quantity.

[Wui12]   Pieter Wuille. *BIP-32: Hierarchical Deterministic Wallets*. 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

[Yin+19]   Maofan Yin et al. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 347–356. DOI: 10.1145/3293611.3331591.

[Zha+24]   Xinyu Zhang et al. *Loquat: A SNARK-Friendly Post-Quantum Signature based on the Legendre PRF with Applications in Ring and Aggregate Signatures*. Cryptology ePrint Archive, Paper 2024/868. 2024. URL: https://eprint.iacr.org/2024/868.

Index

384

387