Zeth Protocol Specification

Clearmatics Cryptography R&D

November 23, 2020

Abstract

This document specifies the Zeth protocol with various security fixes and performance improvements from the initial design [RZ19].

Keywords— Ethereum, Zerocash, Zcash, financial-privacy, zero-knowledge proofs, Zeth, privacy-preserving state transitions

Contents

Notation

Basic mathematical notation

$\varnothing$ The empty set, i.e. $\varnothing = \{\}$

$\#S$ The number of elements in the finite set S (also referred to as "cardinality of the set S"). By convention, $\#\varnothing = 0$

$x \in S$ Represents that x is an element of S. If x is a variable such that $x \in S$, we will say that "x has type S", i.e. the unordered collection of objects S represents all the values that x can take

$S \setminus T$ Set difference of sets S and T, i.e. $S \setminus T = \{x \in S : x \notin T\}$ (voiced "the set of elements x in S such that x is not in T")

$S \subseteq T$ S is a subset of T, i.e. $x \in S \Rightarrow x \in T$

$S \subset T$ S is a *proper* (or "strict") subset of T, i.e. $x \in S \Rightarrow x \in T \wedge \exists y \in T, y \notin S$

$S = T$ $S \subseteq T \wedge T \subseteq S$

$S \cup T$ Union of set S and set T, i.e. $\{x : x \in S \vee x \in T\}$

$S \cap T$ Intersection of set S with set T, i.e. $\{x : x \in S \wedge x \in T\}$

$f : S \to T$ Function f that maps elements of the non-empty set S, the "domain", to the non-empty set T, the "codomain"

$\mathbb{N}$ Set of natural numbers. $\mathbb{N}^+$ represents $\mathbb{N} \setminus \{0\} = \{1, 2, \ldots\}$, where $\{n, \ldots\}$ represents the application of the successor operator $\mathsf{Succ}(n) = n + 1$, defined by the Peano axioms, infinitely many times

$\mathbb{Z}$ Set of integers, i.e. $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, where $\{\ldots, n\}$ represents the application of the predecessor operator $\mathsf{Pred}(n) = n - 1$ infinitely many times

$\mathbb{Q}, \mathbb{R}$ Set of rational, real numbers

$[n]$ Set $\{0, \ldots, n - 1\}$, where $n \in \mathbb{N}$

$\{a, \ldots, b\}$ Set of integers from a through b inclusive, where $a \leq b$

$(a_0, a_1, \ldots, a_{n-1})$ n-tuple, i.e. ordered collection of items of length n. If $n = 1$, we call it a "singleton", if $n = 2$, we call the tuple a "pair". Finally, if $n = 3$, we call it a "triple". We use the terms "tuples" and "lists" interchangeably.

$S \times T$ Cartesian product of sets S and T, i.e. set of all ordered pairs $\{(x, y) : x \in S \wedge y \in T\}$

S^n n-fold Cartesian product of S with itself, i.e. $S^n = \{(x_0, \ldots, x_{n-1}) : x_i \in S \, \forall i \in [n]\}$, where $n \in \mathbb{N}$

Λ General notation for an alphabet, i.e. a *non-empty finite set* such that every string (ordered collection of symbols, or letters, all in Λ) has a unique decomposition. The number of symbols in a string is denoted the "length" of the string

ε The empty string. ε is a string over any alphabet.

Λ^n Set of all strings, defined over alphabet Λ, containing n symbols (i.e. "of length n")

Λ^* The Kleene star of Λ represents the set of all strings of finite length, defined over alphabet Λ, including the empty string ε, i.e. $\Lambda^* = \bigcup_{n \in \mathbb{N}} \Lambda^n$

$\mathsf{length}(x)$ $\mathsf{length}\colon \Lambda^* \to \mathbb{N}$ computes the length of a string x defined over Λ, i.e. $\mathsf{length}(x)$ returns the number of symbols composing the string x. By convention, $\mathsf{length}(\varepsilon) = 0$

$x \| y$ Infix notation for the concatenation function, $\|\colon \Lambda^* \times \Lambda^* \to \Lambda^*$. If $\mathsf{length}(x) = n, \mathsf{length}(y) = m$ and $(n, m) \in \mathbb{N}^2$, then for $z = x \| y$ holds $\mathsf{length}(z) = n + m$

$\mathsf{trunc}_x(k)$ $\mathsf{trunc}\colon \Lambda^* \to \Lambda^k$ is the truncation function that returns the sequence formed from the first k elements of x, where $x \in \Lambda^*$. If $k > \mathsf{length}(x)$, then $\mathsf{trunc}_x(k) = x$

$x[a{:}b]$ $[:]\colon \Lambda^n \times \mathbb{N} \times \mathbb{N} \to \Lambda^{\leq b-a}$ is the slice function that, if $b \geq a$, returns the string starting at index $\min(n, a)$ of x and finishing at index $\min(n, b)$. The function additionally intereprets $x[{:}b]$ as $x[0{:}b]$ and $x[a{:}]$ as $x[a{:}n]$

$\mathsf{pad}_n(x)$ $\mathsf{pad}\colon \Lambda^{\leq n} \to \Lambda^n$ is the padding function which pads x by 0's to reach a size of n. The padding depends on the variable type and endianness.

$\mathsf{append}(l, x)$ $\mathsf{append}\colon D^n \times D \to D^{n+1}$ is the algorithm that appends x to the list of n element(s) l, if all x and l share the same data type D

$\mathbb{B}$ Alphabet of binary symbols, i.e. $\{0, 1\}$

$\langle \mathfrak{g}_1, \ldots, \mathfrak{g}_l \rangle$ Cyclic group generated by $\{\mathfrak{g}_1, \ldots, \mathfrak{g}_l\}$

$(q, \mathbb{G}, \mathfrak{g}, \otimes)$ Description of the cyclic group $\mathbb{G} = \langle \mathfrak{g} \rangle$ of order q, with operation $\otimes$

$\mathbb{Z}/r\mathbb{Z}$ Quotient group defined as the set of equivalence classes modulo r. $\mathbb{Z}/r\mathbb{Z}$, also written $\mathbb{Z}_r$, is an additive group. If $r = p$ a prime number, then $\mathbb{Z}_p = \{0, \ldots, p-1\} = \mathbb{Z}/p\mathbb{Z}$ is a finite field of elements modulo prime p, also denoted $\mathbb{F}_p$, where $0_{\mathbb{F}_p}$ and $1_{\mathbb{F}_p}$ respectively represent the additive and multiplicative identity

$\mathbb{F}_q$ Finite field of cardinality $q = p^m$, where p is prime, and $m \in \mathbb{N}$

$[\![x]\!]$ Represents the encoding of the scalar x in a group $\mathbb{G}$ described as $(p, \mathbb{G}, \langle \mathfrak{g} \rangle, \otimes)$, i.e. $[\![x]\!] = x \cdot [\![1]\!] = \mathfrak{g} \otimes \ldots \otimes \mathfrak{g}$ (x times). Thus, by convention, $[\![1]\!] = \mathfrak{g}$

$\bullet$ Represents an inline operator for bilinear pairing. That is for a bilinear pairing from $\mathbb{G}_1 \times \mathbb{G}_2$ to $\mathbb{G}_T$ and elements $[\![a]\!]_1, [\![b]\!]_2$ we write $[\![ab]\!]_t = [\![a]\!]_1 \bullet [\![b]\!]_2$

$\lceil x \rceil$ Round $x \in \mathbb{R}$ to the next integer

$\lfloor x \rfloor$ Round $x \in \mathbb{R}$ to the previous integer

$\log_b(x)$ Logarithm with respect to base b, i.e. $x = b^y, \log_b(x) = y$

Algorithmic notation

$x \leftarrow\!\!\$\ \mathcal{X}$ Element chosen uniformly at random from set $\mathcal{X}$

$x \leftarrow y$ The value y is assigned to the variable x (i.e. "x receives the value y")

PPT Probabilistic polynomial time. A polynomial time algorithm A is one for which there exists a polynomial f such that the running time of A on input $x \in \{0,1\}^*$ is $f(|x|)$. A probabilistic algorithm has the ability to "flip" random coins and use the result of these coin tosses in its computation

NUPPT Non-uniform probabilistic polynomial time

$\mathcal{O}(f)$ Big-O notation

$\mathsf{il}, \mathsf{kl}, \mathsf{nl}, \mathsf{rl}, \mathsf{ol}$ The input il, key kl, nonce nl, randomness rl and output ol length

Cryptography notation

$\mathsf{O}^{\mathsf{X}}(n)$ Public oracle for algorithm X which can be accessed at most n times; O^{X} is an unrestricted oracle for algorithm X

λ Security parameter ($\lambda \in \mathbb{N}$)

negl Negligible function. In this document, negligible will usually mean $\mathcal{O}\big(2^{-\lambda}\big)$

poly Polynomial function

$\mathcal{A}$ Adversary algorithm

$\mathsf{Adv}^{\mathsf{prop}}_{F,\mathcal{A}}(\lambda)$ Advantage of the adversary $\mathcal{A}$ with regard to the attack game prop on F (e.g. F can be a function, a family of functions or a group on which a given property represented by the game prop is supposed to hold)

$\mathsf{prop}^{\mathcal{A}}$ Adversary $\mathcal{A}$ running a security game prop

Zeth notation

π Output of the proving algorithm of a zk-SNARK scheme. π is also informally referred to as a "zk-SNARK proof", "zk-proof", or simply "proof"

$\mathcal{P}_{\mathcal{Z}}$ Standard notation for a `Zeth` user

$\widetilde{\textbf{Mixer}}$ The mixer smart-contract instance

EncSch In-band encryption scheme used to share `Zeth` notes

Ethereum notation

Account Standard notation for an `Ethereum` account object

$\widetilde{\textbf{Cntrct}}$ Standard notation for an `Ethereum` smart-contract instance

$\mathcal{P}_{\mathcal{E}}$ Standard notation for an `Ethereum` user

ς Mapping representing the `Ethereum` state (i.e. "World state")

$\varsigma[a]$ Account object stored at address a in ς if it exists, $\perp$ is returned otherwise

Constants

`ADDRLEN` The bit-length of an `Ethereum` address $160\,bits$

`BLAKE2sCLEN` Output size of $\mathsf{Blake2s}$ compression function [ANWOW13] $256\,bits$

$\texttt{FIELDCAP}_{\texttt{BLS}}$ Field capacity of $\mathbb{F}_{\texttt{r}_{\texttt{BLS}}}$. $\lfloor \log_2 \texttt{r}_{\texttt{BLS}} \rfloor = 252$ bits

$\texttt{FIELDLEN}_{\texttt{BLS}}$ Bit-length of a field element $x \in \mathbb{F}_{\texttt{r}_{\texttt{BLS}}}$ $\lceil \log_2 \texttt{r}_{\texttt{BLS}} \rceil = 253$ bits

$\texttt{FIELDCAP}_{\texttt{BN}}$ Field capacity of $\mathbb{F}_{\texttt{r}_{\texttt{BN}}}$. $\lfloor \log_2 \texttt{r}_{\texttt{BN}} \rfloor = 253$ bits

$\texttt{FIELDLEN}_{\texttt{BN}}$ Bit-length of a field element $x \in \mathbb{F}_{\texttt{r}_{\texttt{BN}}}$ $\lceil \log_2 \texttt{r}_{\texttt{BN}} \rceil = 254$ bits

`BYTELEN` Bit-length of a byte $8\,bits$

`ENCZETHNOTELEN` Size of an encrypted note (see Section 3.5.3) $\texttt{CTBYTELEN} * \texttt{BYTELEN}\,bits$

`ETHWORDLEN` Width of a storage cell on the Ethereum Virtual Machine stack, i.e. size of a word on the EVM $256\,bits$

$\mathtt{FIELDCAP}$ Field capacity of $\mathbb{F}_{\mathtt{r}}$, defined as the maximum bit length l such that all numbers x encoded on l bits are elements of $\mathbb{F}_{\mathtt{r}}$. In other words, $\mathtt{FIELDCAP} = \max_{x \in \mathbb{F}_{\mathtt{r}}} \{\lceil \log_2 x \rceil\}$ s.t. $\sum_{i \in [\mathtt{FIELDCAP}]} 2^i \in \mathbb{F}_{\mathtt{r}}$

$\mathtt{FIELDLEN}$ Bit-length of elements in field element $x \in \mathbb{F}_{\mathtt{r}}$ — $\lceil \log_2 \mathtt{r} \rceil$ bits

$\mathtt{JSIN}, \mathtt{JSOUT}, \mathtt{JSMAX}$ The number of inputs and outputs of a joinsplit and $\mathtt{JSMAX} = \max\{\mathtt{JSIN}, \mathtt{JSOUT}\}$

$\mathtt{KEK256DLEN}$ Message digest size of Keccak256 [GJMG11] — $256\, bits$

$\mathtt{MKDEPTH}$ The depth of the Merkle tree used to store commitments

$\mathtt{p_{SECP}}$ Prime defining the finite field of curve secp256k1 [wik]

$\mathtt{r_{BLS}}$ Characteristic of the scalar field of BLS12-377, $\mathtt{r_{BLS}} = 8444461749428370424248824938781546531375899335154063827935233455917409239041$ [BCG+20]

$\mathtt{r_{BN}}$ Characteristic of the scalar field of BN-254, $\mathtt{r_{BN}} = 21888242871839275222246405745257275088548364400416034343698204186575808495617$ [Rk19]

$\mathtt{r}$ Characteristic of the scalar field of some chosen curve Curve

$\mathtt{SECPFIELDLEN}$ Bit-length of a field element $x \in \mathbb{F}_{\mathtt{p_{SECP}}}$ — $\lceil \log_2 \mathtt{p_{SECP}} \rceil = 256\, bits$

$\mathtt{SHA256BLEN}$ Block size of SHA256 [oST15, Figure 1] — $512\, bits$

$\mathtt{SHA256DLEN}$ Message digest size of SHA256 [oST15, Figure 1] — $256\, bits$

$\mathtt{SHA256MLEN}$ Message size of SHA256 [oST15, Figure 1] — $< 2^{64}\, bits$

$\mathtt{DGAS}$ The default/intrinsic gas cost of an $\mathtt{Ethereum}$ transaction — 21000 Wei

$\mathtt{ZVALUELEN}$ Size in bits of the transferable maximal value — $64\, bits$

Change log

- **Version:** 0.0, **Date:** 04/12/2019, **Contributor:** Antoine Rondelet, **Description:** Creation of the document. Established initial table of content and started to populate sections with bullet lists to develop in further versions of the document.
- **Version:** 0.1, **Date:** 20/12/2019, **Contributor:** Antoine Rondelet, **Description:** Refactored the structure of the document. Finalized the table of content, wrote sections on notations and preliminaries, and introduced the content related to the malleability fix.
- **Version:** 0.2, **Date:** 24/02/2020, **Contributor:** Clearmatics Cryptography R&D, **Description:**
  - **Date:** 26/02/2020, **Contributor:** Duncan Tebbs, **Description:** Wrote section on wallet implementation and side-channel attacks considerations.
  - **Date:** 02/03/2020, **Contributor:** Giuseppe Giffone, **Description:** Changed Merkle tree hash function to MiMC compression function.
  - **Date:** 04/03/2020, **Contributor:** Duncan Tebbs, Michal Zajac, **Description:** Added background on Groth16 SNARK and SNARK scheme instantiation in the protocol.
  - **Date:** 04/03/2020, **Contributor:** Raphael Toledo, **Description:** Wrote section on the packing policy and corresponding attack.
  - **Date:** 04/03/2020, **Contributor:** Duncan Tebbs, **Description:** Refactored the data structures preliminary section.
  - **Date:** 24/03/2020, **Contributor:** Raphael Toledo, **Description:** Changed the PRF and commitment instantiation with Blake2s compression function.
  - **Date:** 17/04/2020, **Contributor:** Giuseppe Giffone, **Description:** Added DHAES encryption scheme.
- **Version:** 0.3, **Date:** 09/06/2020, **Contributor:** Antoine Rondelet, **Description:** Fixed various inconsistencies throughout the document (notational mistakes in document body and in proofs, latex macros, and typos).
- **Version:** 1.0, **Date:** 30/06/2020, **Contributor:** Antoine Rondelet, Duncan Tebbs, Michal Zajac, **Description:** Global review of the document: fixed inconsistencies in definitions and notations, corrected grammatical mistakes and typos,

added examples, figures and merged sections 5 and 6 of Chapter 1 for clarity, added missing references.

- **Version:** 1.1, **Date:** 06/11/2020, **Contributor:** Duncan Tebbs, **Description:** Specification in terms of a generic curve, with constants provided for BN-254 and BLS12-377. Some clarification and grammar fixes.

Chapter 1

Preliminaries

Zeth is a protocol which enables private transactions on Ethereum [Woo19]. It is a modification of the Decentralized Anonymous Payment (DAP) system ZeroCash [BSCG+14]. The design described in [RZ19] presents the mechanisms by which ZeroCash can be used on Ethereum, and argues that the information leakages of the solution are well defined and controlled. This document, however, serves as a specification of the protocol and provides security fixes and optimizations from the first proof of concept release of the protocol [Cle19].

This document assumes familiarity with blockchain and Ethereum in particular. It does not, in any way, aim at replacing the Ethereum yellow paper [Woo19]. The reader is strongly advised to read about Ethereum before delving into this specification document.

The key words MUST, MUST NOT, SHOULD, SHOULD NOT, MAY, and RECOMMENDED in this document are to be interpreted as described in [Bra97] when they appear in ALL CAPS. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

1.1 Data structures and representation

1.1.1 Structured data

When describing the operations to be performed and the data to be manipulated as part of the protocol, we commonly employ tuples of related data where each element of the tuple has some associated semantic meaning and which must often satisfy some conditions. In this section, we develop a framework to reason about such *structured* data, where a single datum may consist of one or more logical parts (called *fields*). The framework is built on top of simple mathematical concepts such as sets, and mappings between them, ensuring that we can always reason about structured data in a rigorous way. We also define notation to aid the specification of structured data, and to refer to specific components of a datum. This will be used extensively in the specification of the protocol.

As a simple motivating example, consider a protocol that processes data relating to individual people. This fictional system may send and receive data such as *name*, *age* and *address* for a single person, grouping this data into a logical unit. Further, each piece of data must satisfy specific conditions (*name* must be a series of characters from some alphabet, *age* must be a positive integer, etc.) We shall make use of this example several times during the formulation below.

In what follows, let $\mathsf{STR} = \{a, b, \ldots, y, z\}^*$ (the Kleene star of the *Roman alphabet*). In our formulation, field names f_i will be elements in this set.

Remark 1.1.1. Note that a similar formulation could be made using an arbitrary set, such as the same alphabet augmented with specific symbols, or the alphabet of a different language. Our choice of STR here is for simplicity.

We begin by defining a data type as a set of values called "fields", each with a "name" from STR. Abstract sets are used to constrain the values of each field.

Definition 1.1.2 (Structured Data Type)**.** Let $f_0, \ldots, f_{n-1}$ be n distinct elements of STR and let $V_0, \ldots, V_{n-1}$ be sets, for some $n \in \mathbb{N}$. We define *the structured data type* $\mathtt{T}$ *with fields* $\{(f_i, V_i)\}_{i\in[n]}$ to be a set of values:

$$\mathtt{T} = V_0 \times \cdots \times V_{n-1}$$

with associated post-fix "dot" operators $.f_i : \mathtt{T} \to V_i$ for $i = 0, \ldots, n-1$, acting on values $\mathbf{x} \in \mathtt{T}$ to extract the individual elements:

$$\mathbf{x}.f_i = v_i, \text{ where } \mathbf{x} = (v_0, \ldots, v_{n-1}) \in \mathtt{T}$$

Here, we say that the i-th field has *field name* f_i, with *value set* V_i. Each "dot" operator $.f_i$ *extracts* the i-th component, or the *value with field name* f_i.

Example 1.1.3. Consider our example protocol that processes information about people. A potentially useful structured data type $\mathtt{Person}$ may be defined with fields:

$$\big\{(name, \mathsf{STR}), (age, \mathbb{N}), (height, \mathbb{R}^+)\big\}$$

Values $\mathbf{p}$ in $\mathtt{Person}$ are simply tuples in $\mathsf{STR} \times \mathbb{N} \times \mathbb{R}^+$, with semantic meaning (name, age, height) assigned to each component of $\mathbf{p}$.

Examples of valid elements in $\mathtt{Person}$ include $\mathbf{a} = (alice, 28, 1.65)$ and $\mathbf{b} = (bob, 31, 1.74)$, where the following equalities hold:

$$\begin{aligned}\mathbf{a}.name &= alice,\\ \mathbf{b}.age &= 31,\\ \mathbf{b}.height &= 1.74;\end{aligned}$$

For clarity, structured data types may be specified using tables of names, descriptions and value sets, rather than sets of the form $\{(f_i, V_i)\}_{i\in[n]}$. Similarly, it is frequently convenient to include the *field names* alongside values when specifying structured data values.

Example 1.1.4. `Person` from Example 1.1.3 might be described in table-form as follows:

Field	Description	Data type
name	Name of the person	STR
age	Age in years	$\mathbb{N}$
height	Height in meters	$\mathbb{R}^+$

Example 1.1.5. The values $\mathbf{a}$ and $\mathbf{b}$ in Example 1.1.3 might be written as follows:

$$\mathbf{a} = \{name: \ alice, \ age: \ 28, \ height: \ 1.65\}$$
$$\mathbf{b} = \{name: \ bob, \ age: \ 31, \ height: \ 1.74\}$$

Remark 1.1.6 (“dot” operators in assignment)**.** The “dot” operators may be used in algorithm descriptions to indicate *assignment to a specific component.* For example $\mathbf{a}.age \leftarrow 29$ means that the value of the *age* field of $\mathbf{a}$ is replaced by the value 29.

Formally, for a structured data type `T` with fields $\{(f_i, V_i)\}_{i\in[n]}$ where $\mathbf{x} = (v_0, \ldots, v_{n-1}) \in$ `T` and $v'_i \in V_i$:

$$\mathbf{x}.f_i \leftarrow v'_i$$

is equivalent to:

$$\mathbf{x} \leftarrow (v_0, \ldots, v_{i-1}, v'_i, v_{i+1}, \ldots, v_{n-1})$$

We define one further operator and related assignment notation, convenient in cases where $V_i = X^m$ for sets X and $m \in \mathbb{N}$.

Definition 1.1.7 (Square bracket operator)**.** For $m \in \mathbb{N}$ and set X, define the operator $[\,] : X^m \times [m] \to X$ as:

$$\mathbf{x}[i] = x_i \text{ where } \mathbf{x} = (x_0, \ldots, x_m)$$

For the set X^*, the operator takes the form $[\,] : X^* \times \mathbb{N} \to X$, defined as:

$$\mathbf{x}[i] = \begin{cases} x_i & \text{if } \mathsf{length}(\mathbf{x}) > i \text{ where } \mathbf{x} = (x_0, \ldots) \\ \perp & \text{otherwise} \end{cases}$$

Remark 1.1.8 (Square bracket operators in assignment)**.** Similarly to Remark 1.1.6, we develop assignment notation for the square bracket operator $[\,]$. Let $\mathbf{x} = (x_0, \ldots, x_{m-1})$ be a member of X^m, and x'_i be some element in X. The statement:

$$\mathbf{x}[i] \leftarrow x'_i$$

is equivalent to:

$$\mathbf{x} \leftarrow (x_0, \ldots, x_{i-1}, x'_i, x_{x+1}, \ldots, x_{m-1})$$

Informally, this can be interpreted as replacing the i-th component of $\mathbf{x}$ with x'_i.

Remark 1.1.9 (Deep structures and chained "dot" operators)**.** Consider the case of structured data $\mathtt{T}$ with fields $\{(f_i, V_i)\}_{i\in[n]}$ for $n \in \mathbb{N}$. Let $\mathtt{T}'$ be another structured data type with fields $\{(f_i', V_i')\}_{i\in[n']}$ for $n' \in \mathbb{N}$, and assume that $V_j = \mathtt{T}'$ for some $j \in [n]$. Informally, the values of the j-th field of elements of $\mathtt{T}$ are themselves structured data of type $\mathtt{T}'$.

In this case, "dot" operators may be *chained*, so that $\mathbf{x}.f_j.f_k'$ refers to the k-th field v_k' of the j-th field v_j of $\mathbf{x} \in \mathtt{T}$.

Example 1.1.10. Define a structured data type `Address` with fields $(\mathit{country}, \mathsf{STR}), (\mathit{zipcode}, \mathsf{STR})$. We redefine the structured data type `Person` from Example 1.1.3, with an extra field *address* of type `Address`. That is, `Person` is the structured data type with fields:

Field	Description	Data type
name	Name of the person	STR
age	Age in years	$\mathbb{N}$
height	Height in meters	$\mathbb{R}^+$
address	Address of the person	`Address`

An example element $\mathbf{a}$ in `Person` is:

$$\begin{aligned}
\mathbf{a} = \{& \\
& \mathit{name}: \ \mathit{alice}, \\
& \mathit{age}: \ 28, \\
& \mathit{height}: \ 1.65, \\
& \mathit{address}: \ (\mathit{country}: \ \mathit{UK}, \mathit{zipcode}: \ \mathit{SW1A}) \\
\}&
\end{aligned}$$

In this case, the following equalities using the dot and square bracket operators all hold:

$$\begin{aligned}
&\mathbf{a}.\mathit{name} = \mathit{alice} \\
&\mathbf{a}.\mathit{height} = 1.65 \\
&\mathbf{a}.\mathit{address}.\mathit{country} = \mathit{UK} \\
&\mathbf{a}.\mathit{address}.\mathit{zipcode} = \mathit{SW1A} \\
&\mathbf{a}.\mathit{address}.\mathit{country}[1] = K
\end{aligned}$$

1.1.2 Representations

The binary alphabet $\{0, 1\}$, denoted $\mathbb{B}$, is used to represent the presence or absence of an electrical signal in a computer. In fact, every piece of information in a computer is represented as a string of bits. We assume the existence of an efficient binary representation for some set of primitive datatypes (such as the natural numbers $\mathbb{N}$, or alphanumeric characters). Structured data types built up from primitive types (as described above) can then recursively be assigned similarly efficient representations. This is used to define the following functions to *encode* data to its bit-string representation, and to *decode* such bit-strings back to elements of the original type.

Definition 1.1.11. For a set X of values which are to be represented as bit strings, we define functions:

$$\mathsf{encode}_X : X \to \mathbb{B}^*$$
$$\mathsf{decode}_X : \mathbb{B}^* \to X \cup \bot$$

satisfying

$$\mathsf{decode}_X(\mathsf{encode}_X(x)) = x \ \forall x \in X$$

to be the functions which encode (resp. decode) elements of X into (resp. from) the bit-string representations chosen above. We note that decode_X may return $\bot$ in the case that the input bit-string is malformed.

Without ambiguity, we overload the functions encode and decode to mean encode_X and decode_X where the set X is clear from context.

In the following sections, we assume that elements of $\mathbb{N}$ are encoded as big-endian binary numbers in the natural way. We denote by $\mathbb{N}_b$ the set of natural numbers that can be uniquely encoded in this way using b bits (possibly with padding). In other words,

$$\mathbb{N}_b = \left\{ x \in \mathbb{N} \text{ s.t. } \mathsf{encode}_{\mathbb{N}}(x) \in \mathbb{B}^b \right\}$$

1.2 Ethereum

In a nutshell, `Ethereum` is a distributed deterministic state machine, consisting of a globally accessible singleton state ("the World state") and a virtual machine that applies changes to that state [AG18]. State transitions in the state machine are represented by transactions on the system. As such, each transaction represents a change in the global state represented as a Merkle Patricia Tree [wc] whose nodes are objects called "accounts" (Section 1.2.1). The Ethereum Virtual Machine (EVM) allows state transitions to be specified by creating a type of accounts which are associated with a piece of code (smart-contracts). The code of such accounts, and so, the corresponding state transitions, can be executed to transition to another state in the automata, by creating a transaction that calls the given piece of code (Section 1.2.2).

To prevent unbounded state transitions in the state machine, each instruction executed by the EVM is associated with a cost in Wei, referred to as "the gas necessary to run the operation". The "gas cost" of a transaction needs to be paid by the transaction originator (deduced from their account balance), and is awarded to the miner (added to their account balance) who successfully mines the block containing the transaction. In addition to the cost of every instruction executed as part of a state transition, every transaction has an intrinsic "gas cost" of `DGAS` Wei [Woo19, Appendix G]. Bounding modifications to the Ethereum state by the amount of Wei held in the transaction originator's account allows the system to avoid the Halting problem[1] and protects against a range of Denial of Service (DOS) attacks.

1.2.1 Ethereum account

An Ethereum account [Woo19, Section 4.1] is an object containing 4 attributes, as represented Table 1.1. We distinguish two types of accounts:

- "Externally Owned Accounts" (EOA), that are created by derivation of an ECDSA secret key; and
- Smart-contract accounts, that are derived from EVM code specifying a state transition on the state machine.

Each account object is accessible in the Merkle Patricia Tree representing the "World state" by a unique ADDRLEN-bit long identifier called the address. In the context of EOA, the address is obtained by generating a new ECDSA [JMV01] key pair (sk, vk) over curve secp256k1 [Qu99] and taking the rightmost ADDRLEN bits of the Keccak256 hash of the verification key vk.

Field	Description	Data type
nce	The nonce of an account is a scalar value representing the number of transactions that have originated from the account, starting at 0.	$\mathbb{N}_{\texttt{ETHWORDLEN}}$
bal	The balance of an account is a scalar value representing the amount of Wei in the account.	$\mathbb{N}_{\texttt{ETHWORDLEN}}$
$sRoot$	The storage root is the Keccak256 hash representing the storage of the account.	$\mathbb{B}^{\texttt{KEK256DLEN}}$
$codeh$	The code hash is the hash of the EVM code governing the account. If this field is the Keccak256 hash of the empty string, then the account is said to be an "Externally owned Account" (EOA), and is controlled by the corresponding ECDSA private key. If, however, this field is not the Keccak256 hash of the empty string, the account represents a smart contract whose interactions are governed by its EVM code.	$\mathbb{B}^{\texttt{KEK256DLEN}}$

Table 1.1: Ethereum Account structure

[1] https://en.wikipedia.org/wiki/Halting_problem

Note

In the rest of this document, we will refer to an *Ethereum user* $\mathcal{U}_{\mathcal{E}}$ as a person, modeled as an object, holding *one*[a] secret key, sk (object attribute), associated with an existing EOA in the "World state". We denote by $\mathcal{U}_{\mathcal{E}}.Addr$ the `Ethereum` address of $\mathcal{U}_{\mathcal{E}}$ derived from $\mathcal{U}_{\mathcal{E}}.sk$, and which allows $\mathcal{U}_{\mathcal{E}}$ to access the state of their account $\varsigma[\mathcal{U}_{\mathcal{E}}.Addr]$.

We denote by **SmartC** a smart-contract instance/object (i.e. deployed smart-contract with an address, Section 1.2.2), and denote by **SmartC**.$Addr$ its address.

[a] The same physical person may correspond to multiple "`Ethereum` users" and thus control multiple accounts in the Merkle Patricia Tree.

1.2.2 Ethereum transaction

We now briefly mention what `Ethereum` transactions [Woo19, Section 4.2] are, and how they are created, signed and validated. Once more, the reader is highly encouraged to refer to [Woo19] for a detailed presentation. Informally, a transaction object (tx) is a signed message originating from an `Ethereum` user $\mathcal{U}_{\mathcal{E}}$ (the *transaction originator*, or simply *sender*) that represents a state transition on the distributed state machine (i.e. a change in the "World state" ς).

Raw transaction

In the following, we define a raw transaction as an unsigned transaction (Table 1.2).

Field	Description	Data type
nce	Transaction nonce	$\mathbb{N}_{\texttt{ETHWORDLEN}}$
$gasP$	gasPrice	$\mathbb{N}_{\texttt{ETHWORDLEN}}$
$gasL$	gasLimit	$\mathbb{N}_{\texttt{ETHWORDLEN}}$
to	Recipient's address	$\mathbb{B}^{\texttt{ADDRLEN}}$
val	Value of the transaction in Wei	$\mathbb{N}_{\texttt{ETHWORDLEN}}$
$init$ / $data$	Contract Creation data $init$ Message call data $data$	$\mathbb{B}^*$

Table 1.2: Structure of a *raw transaction data type* `TxRawDType`

Finalizing raw transactions

A raw transaction needs to be finalized to be accepted. In the context of this document, "finalizing a raw transaction" will be a synonym of "signing a raw transaction". The transaction structure is represented in Table 1.3.

Field	Description	Data type
tx_{raw}	Raw transaction object	`TxRawDType`
v	Field v of ECDSA signature used for public key recovery	$\mathbb{B}^{\texttt{BYTELEN}}$
r	Field r of ECDSA signature [Por13]	$\mathbb{F}_{\texttt{p}_{\texttt{SECP}}}$
s	Field s of ECDSA signature [Por13]	$\mathbb{F}_{\texttt{p}_{\texttt{SECP}}}$

Table 1.3: Structure of a (finalized) *transaction data type* `TxDType`

We define the transaction generation function, cf. Fig. 1.1, as the function taking the sender's ECDSA signing key and the components of a raw transaction as arguments, and returning a signed (or finalized) transaction (tx_{final} or tx for short).

$$
\begin{aligned}
&tx_{final} = \mathsf{TxGen}(sk_{\mathsf{ECDSA}}, nce_{in}, gasP_{in}, gasL_{in}, to_{in}, val_{in}, init_{in}, data_{in})\\
&tx_{final} = \{\\
&\qquad \left.\begin{aligned}
&nce &&: nce_{in},\\
&gasP &&: gasP_{in},\\
&gasL &&: gasL_{in},\\
&to &&: to_{in},\\
&val &&: val_{in},\\
&init/data &&: init_{in}/data_{in},
\end{aligned}\right\} tx_{raw}\\
&\qquad \left.\begin{aligned}
&v : \sigma_{\mathsf{ECDSA}}.v,\\
&r : \sigma_{\mathsf{ECDSA}}.r,\\
&s : \sigma_{\mathsf{ECDSA}}.s
\end{aligned}\right\} \sigma_{\mathsf{ECDSA}}\\
&\quad \}
\end{aligned}
$$

To sign a transaction, the sender first computes the hash of the raw transaction using $\mathsf{Keccak256}$, cf. Eq. (1.1), and then uses their ECDSA signing key, sk_{ECDSA}, to sign the obtained digest. cf. Eq. (1.2). The signature is then appended to the raw transaction to obtain a finalized transaction, cf. Fig. 1.1.

$$digest_{\mathsf{ECDSA}} = \mathsf{Keccak256}(nce_{in}, gasP_{in}, gasL_{in}, to_{in}, val_{in}, init_{in}/data_{in}) \tag{1.1}$$

$$\sigma_{\mathsf{ECDSA}} = \mathsf{SigSch}_{\mathsf{ECDSA}}.\mathsf{Sig}(sk_{\mathsf{ECDSA}}, digest_{\mathsf{ECDSA}})\ (= (v, r, s)) \tag{1.2}$$

$\mathsf{TxGen}(sk_{\mathsf{ECDSA}}, nce_{in}, gasP_{in}, gasL_{in}, to_{in}, val_{in}, init_{in}, data_{in})$

if $to_{in} = \varnothing$ **do**

  $tx_{raw} \leftarrow \{nce : nce_{in}, gasP : gasP_{in}, gasL : gasL_{in}, to : to_{in}, val : val_{in}, init : init_{in}\};$

else

  $tx_{raw} \leftarrow \{nce : nce_{in}, gasP : gasP_{in}, gasL : gasL_{in}, to : to_{in}, val : val_{in}, data : data_{in}\};$

endif

$\sigma_{\mathsf{ECDSA}} \leftarrow \mathsf{SigSch}_{\mathsf{ECDSA}}.\mathsf{Sig}(sk_{\mathsf{ECDSA}}, \mathsf{Keccak256}(tx_{raw}));$

$tx_{final} \leftarrow \{tx_{raw}, v : \sigma_{\mathsf{ECDSA}}.v, r : \sigma_{\mathsf{ECDSA}}.r, s : \sigma_{\mathsf{ECDSA}}.s\};$

return $tx_{final};$

Figure 1.1: Transaction generation function TxGen

Remark 1.2.1. As one can see, there is no "from" attribute in a transaction. The sender's `Ethereum` address can be recovered from the ECDSA signature. This method is defined in the `Ethereum` yellow paper as a "sender function" S [Woo19, Appendix F] which maps each transaction to its sender.

Types of transactions

While only two types of transactions are described in [Woo19, Section 4.2]; namely those which result in message calls and those which result in the creation of new accounts with associated code, we will instead differentiate the types of transactions based on their purpose. The reader is encouraged to read [Woo19] for a formal discussion.

Informally, a transaction can be used to achieve three things: transferring Wei from an EOA to another EOA, creating a new account with associated code (i.e. "deploying a smart-contract"), and calling a function of a smart-contract. We will detail here the differences between these usages.

Creating a contract The $tx.to$ address is set to $\varnothing$ in the transaction. The contract creation data ($tx.init$) includes the new contract's code. The contract address is computed as the rightmost `ADDRLEN` bits of the Keccak256 hash of the RLP encoding [wc19] of the transaction originator's address and account nonce [Woo19, Section 6].

Calling a contract function The $tx.to$ address is set to the address of the contract. The message call data byte array ($tx.data$) is set to the contract's function address (or *"Function Selector"* [abi]) which are the first 4 bytes of the Keccak256 hash of the function signature, and the function input arguments (`ETHWORDLEN` bits per input) [Woo19, Section 8].

Transferring Wei from an EOA to another EOA This corresponds to a "plain transaction" spending Wei from an address to send them to another. In that case the $tx.to$ address corresponds to the recipient's address while the transaction data is left empty.

Note

In order to keep notations simple, we assume, in the rest of the document, that smart-contract functions are uniquely determined by their name. As such, we denote by $\mathsf{FS}(\cdot) \colon \mathbb{B}^* \rightarrow \mathbb{B}^{4 \cdot \mathtt{BYTELEN}}$ the function that takes a function name as input and returns its function selector.

Transaction validity

Importantly, not all finalized transactions constitute valid state transitions on the state machine [Woo19, Section 6]. We denote by $\mathsf{EthVerifyTx}$ the function that takes an `Ethereum` transaction object tx as input and return true (resp. false) if tx is valid (resp. invalid). To be deemed valid, a transaction `MUST` satisfy *all* the following conditions:

1. The transaction is correctly RLP encoded, with no additional trailing bytes;
2. the transaction signature (v, r, s) is valid;
3. the transaction nonce $(tx.nce)$ is valid, i.e. it is equal to the account nonce of the transaction originator;
4. the gas limit is no smaller than the gas used by the transaction;
5. the transactor has enough funds on his account balance to cover at least the cost $tx.val + tx.gasP \cdot tx.gasL$.

Lifecyle of a transaction, and miners' incentives

After the creation of an `Ethereum` transaction tx by a user from an `Ethereum` client (machine running a piece of software that enables to be connected to the `Ethereum` network), the transaction is broadcasted to the network and received by a set of peers/nodes.

The transaction is then stored in each node's transaction pool, which is a data structure containing all transactions that should be validated (pending transactions) by the node and mined. To maximize miners' returns, the transaction pools are ordered according to the gas price of the transactions. As such, transactions with the highest $tx.gasP$ are subject to be validated and included into a block first. Once tx is selected from the transaction pool, it is validated (fed into $\mathsf{EthVerifyTx}$), executed, and included into a block (i.e. "mined"). The block is then broadcasted to all the nodes of the network and is used as the predecessor for the next block to be mined on the network (i.e. "it is added to the chain").

1.2.3 Ethereum events and Bloom filters

The EVM contains the set of "LOGX" instructions enabling smart-contract functions to "emit events" (i.e. log data) when they are executed[2]

[2]see `https://ethgastable.info/`

As such, when a block is generated by a miner or verified by the rest of the network, the address of any logging contract, and all the indexed fields from the logs generated by executing those transactions are added to a Bloom filter [Blo70], which is included in the block header [Woo19, Section 4.3]. Importantly, the actual logs *are not included in the block data* in order to save space. As such, when an application wants to find ("consume") all the log entries from a given contract, or with specific indexed fields (or both), the node can quickly scan over the header of each block, checking the Bloom filter to see if it may contain relevant logs. If it does, *the node re-executes the transactions from that block, regenerating the logs, and returning the relevant ones to the application* [Joh16].

Note

The ability for a smart-contract function to "emit" some pieces of data when executed, and for an application to "consume" such pieces of data, is used in `Zeth` in order to construct a *confidential receiver-anonymous channel* [KMO$^+$13].

1.3 zk-SNARKs

In this section we introduce notions necessary to understand zero-knowledge proofs, define properties crucial for them, and introduce zk-SNARKs. We refer the reader to Section 3.6 in which we describe the zk-SNARK scheme used in `Zeth`.

1.3.1 Preliminary definitions

NP class of languages. Since the considered proof systems are designed to work with languages in NP we begin with defining this class. Intuitively, a language $\mathbf{L}$ belongs to NP if for each element *prim* from the language there is a short witness *aux* that allows to efficiently[3] verify that in fact $prim \in \mathbf{L}$.

Definition 1.3.1 (NP class of languages, cf. [Gol01])**.** We say that a language $\mathbf{L}$ belongs to a class NP if there exist a polynomial p and a Turing machine M such that for every primary input $prim \in \{0,1\}^*$, $prim \in \mathbf{L}$ iff there exists an auxiliary input aux such that M accepts the pair $(prim, aux)$ in time at most $p(\mathsf{length}(prim))$.

The set of all pairs $(prim, aux)$ acceptable by M constitutes an NP *relation* $\mathbf{R}$ corresponding to the language $\mathbf{L}$.

Non-interactive zero knowledge. A non-interactive zero-knowledge proof system NIZK for an NP language $\mathbf{L}$ is a tuple of four algorithms $\mathsf{NIZK} = (\mathsf{KGen}, \mathsf{P}, \mathsf{V}, \mathsf{Sim})$. NIZK for a language $\mathbf{L}$ and instance $prim \in \mathbf{L}$ allows a party, called prover and denoted by P, to convince another party, called verifier and denoted by V, that $prim \in \mathbf{L}$ and nothing else.

Without loss of generality, we focus on zk-proof systems that are universal, that is, are able to work with any given NP relation $\mathbf{R}$. To that end, we define a *relation*

[3] Informally we say that an algorithm is efficient if it runs in time polynomial in the size of its inputs.

generator $\mathcal{R}$ that on input 1^λ (i.e. the security parameter represented in unary) outputs an NP relation $\mathbf{R}$. We assume that the security parameter λ can be easily deducted from $\mathbf{R}$.

We require from a NIZK to have three substantial properties, cf. [Gro06]:

Completeness that assures that an honest prover, who proves that $prim \in \mathbf{L}$ succeeds, i.e. gets his proof accepted by the verifier V. Formally we require that for any λ, $\mathbf{R} \leftarrow \mathcal{R}(1^\lambda)$, $(prim, aux) \in \mathbf{R}$

$$\Pr\left[\mathsf{V}(\mathbf{R}, crs, prim, \mathsf{P}(\mathbf{R}, crs, prim, aux)) \;\middle|\; \begin{array}{r} \mathbf{R} \leftarrow \mathcal{R}(1^\lambda); \\ (crs, td) \leftarrow \mathsf{KGen}(\mathbf{R}, 1^\lambda) \end{array}\right] = 1 \ .$$

Computational soundness which states that in case $prim \notin \mathbf{L}$ the verifier accepts the proof for $prim$ with negligible probability only. Formally we require that for any $\mathbf{R} \leftarrow \mathcal{R}(1^\lambda)$ and PPT adversary $\mathcal{A}$

$$\Pr\left[\mathsf{V}(\mathbf{R}, crs, prim, \pi) \;\middle|\; \begin{array}{r} \mathbf{R} \leftarrow \mathcal{R}(1^\lambda); \\ (crs, td) \leftarrow \mathsf{KGen}(\mathbf{R}, 1^\lambda); \\ (prim, \pi) \leftarrow \mathcal{A}(\mathbf{R}, crs); \\ prim \notin \mathbf{L} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Zero knowledge assures that the verifier learns from a proof nothing except the veracity of the proven statement. More precisely we require that there exist a PPT algorithm Sim and negligible function $\eta(\lambda)$ such that for every adversary $\mathcal{A}$ and security parameter λ

$$\left| \Pr\left[\mathcal{A}(\mathbf{R}, crs, \pi) = 1 \;\middle|\; \begin{array}{r} \mathbf{R} \leftarrow \mathcal{R}(1^\lambda); \\ (crs, td) \leftarrow \mathsf{KGen}(\mathbf{R}, 1^\lambda); \\ (prim, aux) \leftarrow \mathcal{A}(\mathbf{R}, crs); \\ (prim, aux) \in \mathbf{R}; \\ \pi \leftarrow \mathsf{Sim}(\mathbf{R}, crs, td, prim) \end{array}\right] - \Pr\left[\mathcal{A}(\mathbf{R}, crs, \pi) = 1 \;\middle|\; \begin{array}{r} \mathbf{R} \leftarrow \mathcal{R}(1^\lambda); \\ (crs, td) \leftarrow \mathsf{KGen}(\mathbf{R}, 1^\lambda); \\ (prim, aux) \leftarrow \mathcal{A}(\mathbf{R}, crs); \\ (prim, aux) \in \mathbf{R}; \\ \pi \leftarrow \mathsf{P}(\mathbf{R}, crs, prim, aux) \end{array}\right] \right| \leq \eta(\lambda).$$

We say that NIZK is *perfectly* zero-knowledge if $\eta = 0$.

We note that the existence of the simulator which by using the trapdoor is able to make a proof for a false statement (i.e. for $prim \notin \mathbf{L}$) makes the whole zk-proof system vulnerable to adversaries that also know the trapdoor. More precisely, an adversary who knows a trapdoor td can break the soundness property. This vulnerability comes with each CRS-based NIZK (for languages in NP). Thus in the real-life deployment of a CRS-based NIZK it has to be enforced that nobody learns the trapdoor.

A zk-SNARK scheme, denoted $\mathsf{ZkSnarkSch}$, is a special type of NIZK which is equipped with two more properties. First, zk-SNARKs are arguments *of knowledge*, as such they have to follow a stronger definition of soundness, called *knowledge soundness*.

Knowledge soundness assures that if a prover provided a proof π for a statement $prim$ acceptable to a verifier, then she knows the corresponding auxiliary input aux. More precisely, we require that for each $\mathbf{R} \leftarrow \mathcal{R}(1^\lambda)$, and malicious PPT prover $\mathcal{A}$ there exists a machine $\mathsf{Ext}_\mathcal{A}$, called extractor, that given access to randomness r used by $\mathcal{A}$ and its inputs, *extracts* the auxiliary input aux from $\mathcal{A}$; that is:

$$\Pr\left[\begin{array}{c|r} \neg(\mathbf{R}(prim, aux))\wedge & \mathbf{R} \leftarrow \mathcal{R}(1^\lambda); \\ \mathsf{V}(\mathbf{R}, crs, prim, \pi) & (crs, td) \leftarrow \mathsf{KGen}(\mathbf{R}, 1^\lambda); \\ & (prim, \pi) \leftarrow \mathcal{A}(\mathbf{R}, crs; r); \\ & aux \leftarrow \mathsf{Ext}_\mathcal{A}(\mathbf{R}, crs; r) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Second, zk-SNARKs are *succinct*, and so we require that proofs produced by $\mathsf{ZkSnarkSch.P}$ are short, i.e. sublinear to the size of the primary and auxiliary inputs. Importantly, in many modern zk-SNARKs, like [Gro16, MBKM19, Gab19, GWC19, CHM+20] the proof size is constant regardless the size of the input.

1.3.2 Computation representation – arithmetization

In `Zeth` the sender shows that the transaction is correct by arguing (in zero knowledge, i.e. hiding private inputs) about correctness of evaluation of some predefined predicate. This predicate ensures that the soundness of the blockchain system is not violated, i.e. the zk-proof is used to prove that a transaction follows the "rules of the system" without disclosing its attributes. The proof system that `Zeth` uses operates on an algebraic representation of the "predicate to prove". Informally, representing the computation as a set of algebraic constraints is called *arithmetization*. One of such representations is Quadratic Arithmetic Programs (QAP) [GGPR13], which, following [Gro16], is used in `Zeth`. This representation is considered one of the most efficient for general arithmetic circuits.

Remark 1.3.2. Preprocessing SNARKs such as [Gro16] rely on common reference strings with a specific structure. As such, we may use crs and srs (*structured reference string*) interchangeably in the rest of this document.

QAP (R1CS). Let C be an arithmetic circuit of fan-in 2 over $\mathbb{F}_p$. The number of multiplication gates in C is denoted by $constNo$. Likewise, the number of all wires in C is denoted by $inpNo$.

Before we formally introduce the QAP relation $\mathbf{R}_{\mathtt{QAP}}$ we provide some intuitions behind it. First, we observe that the circuit C can be be represented by three matrices $\vec{A}, \vec{B}, \vec{C}$ all in $\mathbb{F}_p^{constNo \times inpNo+1}$ such that the i-th row in matrix $\vec{A}$ (and $\vec{B}$) denotes left (and right) input to the i-th multiplication gate, which is also the k-th input to the circuit. That is for a circuit evaluation $z \in \mathbb{F}_p^{inpNo+1}$ the left input for the i-th gate is $\sum_{j=0}^{inpNo} A_{ij} z_j$ and the right input is $\sum_{j=0}^{inpNo} B_{ij} z_j$. Furthermore, entry $\vec{C_{ik}}$ contains the output of i-th multiplication gate that is k-th input to the circuit.

Second, for the sake of efficiency we represent each matrix as a sequence of polynomials. Each matrix's column is represented by a polynomial in $\mathbb{F}_p[X]$ such that the column's i-th input equals polynomial's evaluation at ω^i – the i-th primitive root of unity modulo p. More precisely, we define polynomials:

- $u_j(X)$, for $j \in \{0, \dots, inpNo\}$, such that $u_j(\omega^i) = \vec{A}_{ij}$;
- $v_j(X)$, for $j \in \{0, \dots, inpNo\}$, such that $v_j(\omega^i) = \vec{B}_{ij}$;
- $w_j(X)$, for $j \in \{0, \dots, inpNo\}$, such that $w_j(\omega^i) = \vec{C}_{ij}$.

We consider inputs from 1 to $inpNoPrim$ public (primary input), for some $inpNoPrim \leq inpNo$. The rest of the inputs is considered private (auxiliary input). The QAP relation $\mathbf{R}_{\mathtt{QAP}}$ is defined as follows:

$$\mathbf{R}_{\mathtt{QAP}} = \left\{ (prim, aux) \middle| \begin{array}{l} a_0 = 1; prim = (a_1, \dots, a_{inpNoPrim}) \in \mathbb{F}_p^{inpNoPrim}; \\ aux = (a_{inpNoPrim+1}, \dots, a_{inpNo}) \in \mathbb{F}_p^{inpNo-inpNoPrim}; \\ \sum_{j=0}^{inpNo} a_j u_j(X) \cdot \sum_{j=0}^{inpNo} a_j v_j(X) = \sum_{j=0}^{inpNo} a_j w_j(X) \end{array} \right\}.$$

Note

Importantly, we note that efficient computation on standard hardware may not necessarily lead to an efficient QAP representation. As such, a function can be very efficient to evaluate on a standard computer, but very slow to evaluate in QAP form.

1.4 Decentralized Anonymous Payment schemes (DAP)

`Zeth` [RZ19] is a Decentralized Anonymous Payment scheme (DAP) [BSCG+14, Section 3] defined on top of an `Ethereum` ledger L. A DAP is a tuple of polynomial-time algorithms $\mathsf{DAP} = (\mathsf{Setup}, \mathsf{GenAddr}, \mathsf{SendTx}, \mathsf{VerifyTx}, \mathsf{Receive})$ that manipulate (*create*, *spend*) data objects called *Notes*. These objects are bound to a given owner and have a value v attribute (see Section 2.1).

System Setup The algorithm Setup takes the security parameter λ as input and generates the public parameters pp. The algorithm Setup is executed by a trusted party. The resulting public parameters pp are published and made available to all parties.

Creating `Zeth` addresses The algorithm GenAddr takes as input the public parameters pp and generates a new DAP address object $Addr = \{pub : Addr_{pk}, priv : Addr_{sk}\}$. More precisely, $Addr_{pk}$ is an object referred to as the "payment address" (Table 1.4), and $Addr_{sk}$ is an object referred to as the "private address" (Table 1.5) [ZCa19].

Transfer notes The algorithm SendTx is used to transfer some public input vin as well as the value of a set of input ("old") $Notes$ into a set of output ("new") $Notes$ as well as some public output value $vout$. The inputs $Notes$ are marked as "consumed" (alternatively, we say that the input $Notes$ are "spent"). SendTx takes as inputs the public parameters pp, the input value and the input ("old") $Notes$ to be transferred, as well as the Merkle root and the Merkle authentications paths of the commitments to the input $Notes$, the "spending keys" related to the input $Notes$, the output value to create and the "payment addresses" for the output ("new") $Notes$. If the joinsplit equation is satisfied, the algorithm returns the new $Notes$ and the corresponding `Ethereum` transaction tx, else it returns $\perp$.

Verifying transactions The algorithm VerifyTx checks the validity of a transaction. It takes as inputs the public parameters pp, a transaction and the current ledger L and outputs a bit equal 1 iff the transaction is valid, 0 otherwise.

Receiving notes The algorithm Receive scans the ledger L and retrieves unspent $Notes$ paid to a particular user address. It takes as input the recipient address key pair $\{pub : Addr_{pk}, priv : Addr_{sk}\}$ and the current ledger L and outputs the set of (unspent) received $Notes$.

Note

In the rest of this document, we will refer to a *Zeth user* $\mathcal{U}_{\mathcal{Z}}$ as a person, modeled as an object, holding one `Zeth` address (object attribute), and thus holding a *private address*, $Addr_{sk}$. We denote by $\mathcal{U}_{\mathcal{Z}}.Addr$ the `Zeth` address of $\mathcal{U}_{\mathcal{Z}}$ derived from $Addr_{sk}$, and which allows $\mathcal{U}_{\mathcal{Z}}$ to be the recipient of payments via `Zeth`, and to send funds via `Zeth`. Importantly, *not all* ***`Ethereum`*** *users are* ***`Zeth`*** *users, and vice-versa!*

Field	Description
apk	The *paying key*
$pkenc$	The *transmission key*

Table 1.4: "Payment address", $Addr_{pk}$, of a DAP address

Field	Description
ask	The *spending key*
skenc	The *receiving key*

Table 1.5: "Private address", $Addr_{sk}$, of a DAP address

`Zeth` leverages zk-SNARKs (Section 1.3) and the possibility to deploy smart-contracts to specify privacy-preserving state transitions altering the `Ethereum` state ς (Section 1.2). As such, `Zeth` defines a smart-contract, $\widetilde{\textbf{Mixer}}$, that keeps track of the set of *ZethNotes* (Section 2.1) in a committed form, stored in a Merkle tree; and which verifies the validity of the state transitions generated by the `Zeth` users. As such a `Zeth` DAP is entirely determined by $\widetilde{\textbf{Mixer}}$, the instance of the mixer smart-contract deployed on the `Ethereum` ledger. State transitions are executed on-chain by calling the Mix function of $\widetilde{\textbf{Mixer}}$, which implements the algorithm VerifyTx of DAP, and which modifies ς iff the transaction is deemed valid.

Note

We denote by Mix_{in} the inputs taken by the Mix function defined on $\widetilde{\textbf{Mixer}}$. Let *zdata* be the value of the *data* field of an `Ethereum` transaction such that:

$$zdata = \mathsf{FS}(\mathsf{Mix}) || \mathsf{Mix}_{in}$$

Then, we define tx_{Mix} as being the `Ethereum` transaction object returned by SendTx such that:

$$tx_{\mathsf{Mix}}.to = \widetilde{\textbf{Mixer}}.Addr \wedge tx_{\mathsf{Mix}}.data = zdata$$

Importantly, when it is clear from context, we will omit the function selector from the definition of *zdata*, and only assume that $zdata = \mathsf{Mix}_{in}$.

1.5 Definitions

1.5.1 Negligible function

Definition 1.5.1 (Negligible function, [KL14, Definition 3.4])**.** A function f from $\mathbb{N}$ to $\mathbb{R}^+$ (positive real numbers) is negligible if for every positive polynomial p there exists N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.

1.5.2 Basic algebra notions

Definition 1.5.2 (Group, see [Bou03, Section I.4])**.** A group is given by a tuple $(\mathbb{G}, \otimes)$, where $\mathbb{G}$ is a set and $\otimes$ is a binary operation in $\mathbb{G}$, i.e. $\otimes : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, with the following properties:

- $(\mathfrak{g} \otimes \mathfrak{h}) \otimes \mathfrak{k} = \mathfrak{g} \otimes (\mathfrak{h} \otimes \mathfrak{k})$ (associativity)
- There exists an element $\epsilon \in \mathbb{G}$ s.t. for each $\mathfrak{g} \in \mathbb{G}$, $\mathfrak{g} \otimes \epsilon = \epsilon \otimes \mathfrak{g} = \mathfrak{g}$ (identity element).
- For each $\mathfrak{g} \in \mathbb{G}$ there exist $\mathfrak{h} \in \mathbb{G}$ s.t. $\mathfrak{g} \otimes \mathfrak{h} = \mathfrak{h} \otimes \mathfrak{g} = \epsilon$ (inverse element).

For simplicity, we may also use the additive notation for groups: $\otimes$ is denoted as $+$, the identity element as $\mathfrak{o}$ and the inverse element of $\mathfrak{g}$ as $-\mathfrak{g}$. Given $\mathfrak{g} \in \mathbb{G}$ and $x \in \mathbb{Z}$, we have that:

$$x \cdot \mathfrak{g} = \begin{cases} \mathfrak{o} & \text{if } x = 0 \\ \mathfrak{g} + \ldots + \mathfrak{g}, (x \text{ times}) & \text{if } x > 0 \\ -\mathfrak{g} + \ldots + (-\mathfrak{g}), (x \text{ times}) & \text{if } x < 0 \end{cases}.$$

Definition 1.5.3 (Finite Cyclic Group, adapted from [KL14, Sections 7.1.3, 7.3.2])**.** A finite cyclic group is given by a tuple $(q, \mathbb{G}, \mathfrak{g}, \otimes)$, called the *group description*, where $\mathbb{G}$ represents the set of group elements, $\mathfrak{g}$ is a generator and q is the order. The generator $\mathfrak{g}$ generates the group; namely, each $\mathfrak{h} \in \mathbb{G}$ can be expressed by the generator as $\mathfrak{h} = \mathfrak{g} \otimes \ldots \otimes \mathfrak{g}$. Given a scalar x, we denote by $\llbracket x \rrbracket$ the *encoding* of x in $\mathbb{G}$: i.e. $\llbracket x \rrbracket = \mathfrak{g} \otimes \ldots \otimes \mathfrak{g}$ (x times). As consequence, $\llbracket 1 \rrbracket = \mathfrak{g}$.

For theoretical purposes, we introduce the SetupG algorithm that for a given security parameter λ outputs a cyclic group, formally:

Definition 1.5.4 (Group Setup Algorithm, taken from [KL14, Sections 7.1.3, 7.3.2])**.** A group setup algorithm SetupG is a PPT algorithm which takes as input a security parameter 1^λ and outputs a group description $(q, \mathbb{G}, \mathfrak{g}, \otimes)$, where the binary representation of q is given by λ bits and each group element can be represented by $gLen(\lambda)$ bits. Note that $gLen$ is $\mathsf{poly}(\lambda)$.[4]

1.5.3 Security assumptions

Definition 1.5.5 (Discrete Log Problem(DLog), cf. [BS07])**.** Let $\mathbb{G}$ denote a group (Section 1.5.2) whose order p is prime and written over λ bits. We let $\log_{\mathfrak{g}}(h)$ denote the discrete logarithm of h in the basis $\mathfrak{g}$. We assume $\mathbb{G}$, p are fixed and known to all parties. We denote the advantage of a PPT adversary $\mathcal{A}$ in attacking the discrete logarithm problem as

$$\mathsf{Adv}^{\mathsf{dlog}}_{\mathbb{G},\mathcal{A}} = \Pr\big[\mathfrak{g} \leftarrow\$\, \mathbb{G}^*,\ x \leftarrow\$\, \mathbb{F}_p,\ x' \leftarrow \mathcal{A}(\llbracket 1 \rrbracket, \llbracket x \rrbracket) :\ \llbracket x' \rrbracket = \llbracket x \rrbracket\big]$$

[4] For simplicity, we may denote $gLen(\lambda)$ as $gLen$.

We say that the DLog is hard in $\mathbb{G}$ if and only if $\mathsf{Adv}^{\mathsf{dlog}}_{\mathbb{G},\mathcal{A}}(\lambda)$ is negligible for any PPT adversary $\mathcal{A}$.

Definition 1.5.6 (One More Discrete Log Problem ($\mathsf{om\text{-}DLog}$), cf. [PV05]). Let $\mathbb{G}$ denote a group whose order p is prime and written over λ bits. We let $\log_{\mathfrak{g}}(h)$ denote the discrete logarithm of h in the basis $\mathfrak{g}$. A PPT adversary $\mathcal{A}$ solving the $\mathsf{om\text{-}DLog}$ is given $q+1$ random group elements as well as limited access to a discrete logarithm oracle $\mathsf{O}^{\mathsf{DLog}_{\mathfrak{g}}}(q)$. $\mathcal{A}$ is allowed to query this oracle at most q times, thus obtaining the discrete logarithm of q group elements of his choice with respect to a fixed base $\mathfrak{g}$. Eventually, $\mathcal{A}$ must output the $q+1$ discrete logarithms. We denote the advantage of a PPT adversary $\mathcal{A}$ in attacking the one more discrete logarithm problem as

$$\mathsf{Adv}^{\mathsf{om\text{-}dlog}}_{\mathbb{G},\mathcal{A}}(\lambda) = \Pr\left[\begin{array}{c} \mathfrak{g} \leftarrow\!\$\, \mathbb{G}^*,\ \{[\![r_i]\!]\}_{i\in[q+1]} \leftarrow\!\$\, \mathbb{G}^{q+1}, \\ \{r'_i\}_{i\in[q+1]} \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{DLog}_{\mathfrak{g}}}(q)}([\![1]\!], \{[\![r_i]\!]\}_{i\in[q+1]}) : \\ \forall i \in [q+1],\ r'_i = \log_{\mathfrak{g}}([\![r_i]\!]) \end{array}\right]$$

We say that the $\mathsf{om\text{-}DLog}$ is hard in $\mathbb{G}$ if and only if $\mathsf{Adv}^{\mathsf{om\text{-}dlog}}_{\mathbb{G},\mathcal{A}}(\lambda)$ is negligible for any PPT adversary $\mathcal{A}$.

1.5.4 Symmetric encryption

Definition 1.5.7 (Symmetric Encryption,[KL14, Definition 3.8]). A symmetric encryption scheme Sym is given by a tuple of PPT algorithms $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ where:

- KGen, the key generation algorithm, takes a security parameter 1^λ and outputs a secret key ek; we assume, without loss of generality, that $kLen(\lambda) = \mathsf{length}(ek) \geq \lambda$. Note that $kLen(\lambda)$ is a polynomial function in λ.[5]

- Enc, the encryption algorithm, takes a key ek, a plaintext $m \in \{0,1\}^*$ and returns a ciphertext ct.

- Dec, the decryption algorithm, takes a key ek and a ciphertext ct, and returns a message m. We assume, without loss of generality, that Dec is deterministic.

For every security parameter λ, key ek output by $\mathsf{KGen}(1^\lambda)$, and message $m \in \{0,1\}^*$, it holds that $\mathsf{Dec}(ek, \mathsf{Enc}(ek, m)) = m$ (*correctness property*).

Let $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. If there exists a polynomial l such that, for all $\lambda > 0$ and key ek output by $\mathsf{KGen}(1^\lambda)$, $\mathsf{Enc}(ek, \cdot)$ is only defined for messages $m \in \{0,1\}^{l(\lambda)}$, then we say that $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is a *fixed-length symmetric encryption scheme* with *length parameter* $l(\lambda)$. A security notion for Sym follows:

Definition 1.5.8 ($\mathsf{IND\text{-}CPA}$). Let Sym be a symmetric encryption scheme and let $\mathcal{A}$ be an adversary. Consider the $\mathsf{IND\text{-}CPA}$ game described in Figure 1.2. We define the $\mathsf{IND\text{-}CPA}$ advantage of $\mathcal{A}$ as follows:

$$\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{Sym},\mathcal{A}}(\lambda) = |2 \cdot \Pr[\mathsf{IND\text{-}CPA}(\lambda) = 1] - 1|.$$

[5]For simplicity, we may denote $kLen(\lambda)$ as $kLen$.

$\underline{\mathsf{IND\text{-}CPA}(\lambda)}$

$ek \leftarrow \mathsf{KGen}(1^\lambda)$

$(m_0, m_1, state) \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Enc}_{ek}}}$ with $\mathsf{length}(m_0) = \mathsf{length}(m_1)$

$b \leftarrow\$ \{0, 1\}$

$ct \leftarrow \mathsf{Enc}(ek, m_b)$

$\widetilde{b} \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Enc}_{ek}}}(ct, state)$

return $\widetilde{b} = b$

Figure 1.2: IND-CPA game for Sym.

Sym is said to be IND-CPA secure if, for every PPT adversary $\mathcal{A}$, the advantage $\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{Sym},\mathcal{A}}(\lambda)$ is a negligible function.

1.5.5 Asymmetric encryption

Definition 1.5.9 (Asymmetric encryption, [KL14, Definition 10.1])**.** An *asymmetric encryption scheme* Asym is given by a tuple of PPT algorithms (KGen, Enc, Dec) where:

- KGen, the key generation algorithm, takes a security parameter 1^λ and returns a pair of keys (sk, pk). We refer to the first of these as the *private key* and the second as the *public key*. We assume for convenience that pk and sk each have length at least λ, and that λ can be determined from pk, sk;

- Enc, the encryption algorithm, takes a public key pk, a plaintext m, from some underlying plaintext space (that may depend on pk) and returns a ciphertext ct;

- Dec, the decryption algorithm, takes a private key sk and a ciphertext ct, and returns a message m or a special symbol $\perp$ denoting decryption failure. We assume, without loss of generality, that Dec is deterministic.

We require that for all (sk, pk) returned by KGen, and every message m in the appropriate underlying plaintext space, it holds that $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m$ (*correctness property*).

Secure communication usually requires ciphertext indistinguishability (e.g. IND-CCA2 [ABR99, Definition 8]). In `Zeth`, however, the key privacy property IK-CCA [BBDP01] is also required – it ensures indistinguishability of the key under which an encryption is performed.

Definition 1.5.10 (IK-CCA)**.** Let $\mathsf{Asym} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be an asymmetric encryption scheme and let $\mathcal{A}$ be an adversary. Given the IK-CCA game described in Figure 1.3, with the condition that $\mathcal{A}$ cannot query $\mathsf{O}^{\mathsf{Dec}_{sk_0}}$ or $\mathsf{O}^{\mathsf{Dec}_{sk_1}}$ on the challenge ciphertext

$\underline{\mathsf{IK\text{-}CCA}(\lambda)}$

$(sk_0, pk_0), (sk_1, pk_1) \leftarrow \mathsf{KGen}(1^\lambda)$

$(m, state) \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Dec}_{sk_0}}, \mathsf{O}^{\mathsf{Dec}_{sk_1}}}(pk_0, pk_1)$

$b \leftarrow\$ \{0, 1\}$

$ct \leftarrow \mathsf{Enc}(pk_b, m)$

$\widetilde{b} \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Dec}_{sk_0}}, \mathsf{O}^{\mathsf{Dec}_{sk_1}}}(ct, state)$

return $\widetilde{b} = b$

Figure 1.3: IK-CCA game.

ct[6], we define the IK-CCA advantage of $\mathcal{A}$ as follows:

$$\mathsf{Adv}^{\mathsf{ik\text{-}cca}}_{\mathsf{Asym},\mathcal{A}}(\lambda) = |2 \cdot \Pr[\mathsf{IK\text{-}CCA}(\lambda) = 1] - 1|$$

We say that Asym is IK-CCA secure if for every PPT adversary $\mathcal{A}$ the advantage $\mathsf{Adv}^{\mathsf{ik\text{-}cca}}_{\mathsf{Asym},\mathcal{A}}(\lambda)$ is a negligible function.

1.5.6 Block cipher-based compression functions

Definition 1.5.11. Let $\mathsf{kl}, \mathsf{il} > 1$. A *block cipher* is a map $\mathsf{E}\colon \{0,1\}^{\mathsf{kl}} \times \{0,1\}^{\mathsf{il}} \to \{0,1\}^{\mathsf{il}}$ where, for each key $k \in \{0,1\}^{\mathsf{kl}}$, the function $\mathsf{E}_k(\cdot) = \mathsf{E}(k, \cdot)$ is a permutation on $\{0,1\}^{\mathsf{il}}$. If E is a block cipher then E^{-1} is its inverse, that on input (k, y) returns m such that $\mathsf{E}_k(m) = y$.

Let $\mathcal{BLK}(\mathsf{kl}, \mathsf{il})$ be the set of all block ciphers $\mathsf{E}\colon \{0,1\}^{\mathsf{kl}} \times \{0,1\}^{\mathsf{il}} \to \{0,1\}^{\mathsf{il}}$. In order to analyse the security properties of block cipher-based cryptographic constructions it is common to use a security model denoted *the ideal cipher model (ICM).* Informally speaking, in ICM attackers are allowed to query an oracle simulating a random block cipher, but have no information about the oracle's internal structure. We formalize this notion in the following definition:

Definition 1.5.12 (Ideal Cipher Model [HKT11])**.** The Ideal Cipher Model (ICM), is a security model where all parties are granted access to an ideal cipher $\mathsf{E}\colon \{0,1\}^{\mathsf{kl}} \times \{0,1\}^{\mathsf{il}} \to \{0,1\}^{\mathsf{il}}$, a random primitive such that $\mathsf{E}(k, \cdot)$ for $k \in \{0,1\}^{\mathsf{kl}}$ are 2^{kl} independent random permutations.

For fixed kl and il, each party is given access to the oracles O^{E} and $\mathsf{O}^{\mathsf{E}^{-1}}$, simulating E and E^{-1}, which can be queried for encryption and decryption a polynomial number of times. The encryption oracle takes as input a key, $k \in \{0,1\}^{\mathsf{kl}}$, and a preimage, $m \in \{0,1\}^{\mathsf{il}}$, and returns a tuple comprising the image, $y \in \{0,1\}^{\mathsf{il}}$, along with the inputs, k and m. If (k, m) is queried for the first time, the image y is taken uniformly

[6] *state* is some state information that the adversary outputs after the choice of the message to encrypt. It can be some preprocessed information that can be helpful to win the game

$\mathsf{O}^{\mathsf{E}}(k,m)$	$\mathsf{O}^{\mathsf{E}^{-1}}(k,y)$
if $(k,m,\cdot) \notin \text{Table}_{\mathsf{O}}$	**if** $(k,\cdot,y) \notin \text{Table}_{\mathsf{O}}$
$\quad y \leftarrow\$ \{0,1\}^{\mathsf{il}}$	$\quad m \leftarrow\$ \{0,1\}^{\mathsf{il}}$
$\quad \text{Table}_{\mathsf{O}}.\text{append}(k,m,y)$	$\quad \text{Table}_{\mathsf{O}}.\text{append}(k,m,y)$
else $y \leftarrow \text{Table}_{\mathsf{O}}(k,m)$	**else** $m \leftarrow \text{Table}_{\mathsf{O}}(k,y)$
return (k,m,y)	**return** (k,m,y)

Figure 1.4: Oracles of an ideal block cipher, with $\text{Table}_{\mathsf{O}}$ being a table of tuples (key, preimage, image) of queries already answered by the oracle.

at random from $\{0,1\}^{\mathsf{il}}$ and added to the oracle's table. Otherwise, the oracle returns y associated with query (k,m) in its table. The decryption oracle is defined similarly with the image and key defined as inputs and the preimage chosen randomly, for details see Fig. 1.4.

Definition 1.5.13 (Block cipher-based compression function [BRS02])**.** A *block cipher-based compression function* is a map f such that

$$\mathsf{f}\colon \mathcal{BLK}(\mathsf{kl},\mathsf{il}) \times \{0,1\}^a \times \{0,1\}^b \to \{0,1\}^c$$

where kl, il, a, b, $c > 1$ and $a+b > c$. The function f, given $m \in \{0,1\}^a \times \{0,1\}^b$, computes $\mathsf{f}(\mathsf{E},m)$ using an E-oracle.

Remark 1.5.14. We use f_{E} to denote a block cipher-based compression function f restricted to a given block cipher E, i.e. $\mathsf{f}_{\mathsf{E}} : \{0,1\}^a \times \{0,1\}^b \to \{0,1\}^c$ and $\mathsf{f}_{\mathsf{E}} = \mathsf{f}(\mathsf{E},\cdot)$, for a,b,c as given in the definition above.

Let f be a compression function based on a block cipher. Fix a constant $h_0 \in \{0,1\}^c$ and an adversary $\mathcal{A}$. We define the advantage in finding a collision in f as

$$\mathsf{Adv}^{\mathsf{coll}}_{\mathsf{f},\mathcal{A}} = \Pr\left[\begin{matrix}\mathsf{E} \leftarrow\$ \mathcal{BLK}(\mathsf{kl},\mathsf{il}); ((k,m),(k',m')) \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{E}},\mathsf{O}^{\mathsf{E}^{-1}}}(\mathsf{f}_{\mathsf{E}},h_0) : \\ ((k,m) \neq (k',m') \wedge \mathsf{f}_{\mathsf{E}}(k,m) = \mathsf{f}_{\mathsf{E}}(k',m')) \vee \mathsf{f}_{\mathsf{E}}(k,m) = h_0\end{matrix}\right].$$

The previous definition gives credit for finding an (k,m) such that $\mathsf{f}_{\mathsf{E}}(k,m) = h_0$ for a fixed $h_0 \in \{0,1\}^c$.

1.5.7 Hash functions

Definition 1.5.15 (Hash function, [KL14, Definition 4.9])**.** A hash function $\mathcal{H}$ is a pair of algorithms $(\mathsf{Setup},\mathsf{H})$ fulfilling the following properties:

- Setup is a PPT algorithm which takes as input a security parameter 1^λ and outputs a key hk. We assume that 1^λ is included in hk.

- H is (deterministic) polynomial-time algorithm that takes as input a key hk and any string $x \in \{0,1\}^*$, and outputs a string $\mathsf{H}(hk, x) = \mathsf{H}_{hk}(x) \in \{0,1\}^{hLen}$, where $hLen$ is a polynomial in λ.[7]

If for every λ and hk, H_{hk} is defined only over inputs of length $hInpLen(\lambda)$ and $hInpLen(\lambda) > hLen(\lambda)$, then we say that $\mathcal{H}$ is a *fixed-length hash function* with length parameter $hInpLen$. Note that $hInpLen(\lambda)$ is a polynomial in λ.

Informally, for a given function f we say that (x, y) is a *collision* if $f(x) = f(y)$ and $x \neq y$. In the following, we formalize this notion for a hash function $\mathcal{H}$.

Definition 1.5.16 (Collision Resistance [KL14, Definitions 4.10])**.** A hash function $\mathcal{H} = (\mathsf{Setup}, \mathsf{H})$ is collision resistant if for all PPT adversaries $\mathcal{A}$ there exists a negligible function $\mathsf{negl}(\lambda)$ such that:

$$\mathsf{Adv}^{\mathsf{cr}}_{\mathcal{H},\mathcal{A}}(\lambda) = \Pr\Big[hk \leftarrow \mathsf{Setup}(1^\lambda), (x, y) \leftarrow \mathcal{A}(hk) :\ x \neq y \wedge \mathsf{H}_{hk}(x) = \mathsf{H}_{hk}(y)\Big] \leq \mathsf{negl}(\lambda)\,.$$

HDHI and HDHI2 assumptions

The Hash Diffie-Helmann Independence (HDHI) assumption states that, given H in $\mathcal{H}$ and a group description $(p, \mathbb{G}, \mathfrak{g}, \otimes)$, for $[\![u]\!]$ and $[\![v]\!]$, with u, v sampled at random, it is hard for an attacker to distinguish $\mathsf{H}([\![u]\!] \| [\![uv]\!])$ from a random string of the same size.[8] This is formalized in Definition 1.5.17, where an attacker can also access an oracle $\mathsf{O}^{\mathsf{HDHI}_v}$ that on input $\mathfrak{x} \in \mathbb{G}$ returns $\mathsf{H}(\mathfrak{x} \| v \cdot \mathfrak{x})$ (queries on $[\![u]\!]$ are forbidden).[9]. In other words, the HDHI assumption measures the sense in which H is "independent" of the underlying Diffie-Hellman problem.

Definition 1.5.17 (HDHI, [ABR99, Definition 7])**.** Let $\mathcal{H}$ be a hash function, SetupG be a group generation algorithm and $\mathcal{A}$ be an adversary. Consider the HDHI game described in Figure 1.5. We define the advantage of $\mathcal{A}$ in violating the HDHI assumption as:

$$\mathsf{Adv}^{\mathsf{hdhi}}_{\mathcal{H},\mathsf{SetupG},\mathcal{A}}(\lambda) = |2 \cdot \Pr[\mathsf{HDHI}(\lambda) = 1] - 1|.$$

Note that the above definition corresponds to [ABR99, Section 3.2.1, Definition 3]. In the following, we introduce a similar notion denoted as HDHI2 (this is an adaptation of the ODH2 notion in [ABN10, Section 6]) which will be useful in the IK-CCA proof Section 3.5.4.

Definition 1.5.18 (HDHI2)**.** Let $\mathcal{H}$ be a hash function, SetupG a group generation algorithm and let $\mathcal{A}$ be an adversary. Consider the HDHI2 game described in Figure 1.6. We define the advantage of $\mathcal{A}$ in violating the HDHI2 assumption as:

$$\mathsf{Adv}^{\mathsf{hdhi2}}_{\mathcal{H},\mathsf{SetupG},\mathcal{A}}(\lambda) = |2 \cdot \Pr[\mathsf{HDHI2}(\lambda) = 1] - 1|.$$

---

[7] For simplicity, we may denote $hLen(\lambda)$ as $hLen$.

[8] Note that H takes as inputs bit strings, so technically we should make use of an encoding function from $\mathbb{G}$ to $\{0,1\}^{gLen}$ but we may omit this step through the document to improve readability.

[9] In [ABR99, Section 3.2.1] this notion is denoted as adaptive HDH independence assumption. Since we only introduce the adaptive version we denote it as HDHI.

$\mathsf{HDHI}(\lambda)$

$hk \leftarrow \mathcal{H}.\mathsf{Setup}(1^\lambda)$
$(q, \mathbb{G}, \mathfrak{g}, \otimes) \leftarrow \mathsf{SetupG}(1^\lambda)$
$u, v \leftarrow\$ [q]$
$w_0 \leftarrow \mathcal{H}.\mathsf{H}_{hk}([\![u]\!] \| [\![uv]\!])$
$w_1 \leftarrow\$ \{0,1\}^{hLen}$
$b \leftarrow\$ \{0,1\}$
$\widetilde{b} \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{HDHI}_v}}([\![u]\!], [\![v]\!], w_b)$
return $\widetilde{b} = b$

Figure 1.5: HDHI game.

$\mathsf{HDHI2}(\lambda)$

$hk \leftarrow \mathcal{H}.\mathsf{Setup}(1^\lambda)$
$(q, \mathbb{G}, \mathfrak{g}, \otimes) \leftarrow \mathsf{SetupG}(1^\lambda)$
$u, v_0, v_1 \leftarrow\$ [q]$
$w_{0,0} \leftarrow \mathcal{H}.\mathsf{H}_{hk}([\![u]\!] \| [\![uv_0]\!]), w_{0,1} \leftarrow \mathcal{H}.\mathsf{H}_{hk}([\![u]\!] \| [\![uv_1]\!])$
$w_{1,0} \leftarrow\$ \{0,1\}^{hLen}, w_{1,1} \leftarrow\$ \{0,1\}^{hLen}$
$b \leftarrow\$ \{0,1\}$
$\widetilde{b} \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{HDHI}_{v_0}}, \mathsf{O}^{\mathsf{HDHI}_{v_1}}}([\![u]\!], [\![v_0]\!], [\![v_1]\!], w_{b,0}, w_{b,1})$
return $\widetilde{b} = b$

Figure 1.6: HDHI2 game.

Lemma 1.5.1. *Let $\mathcal{A}$ be an adversary with advantage $\mathsf{Adv}^{\mathsf{hdhi2}}_{\mathcal{H},\mathsf{SetupG},\mathcal{A}}$ in solving the* HDHI2 *problem. Then there exists an adversary $\mathcal{B}$ such that*

$$\mathsf{Adv}^{\mathsf{hdhi2}}_{\mathcal{H},\mathsf{SetupG},\mathcal{A}}(\lambda) \leq 2 \cdot \mathsf{Adv}^{\mathsf{hdhi}}_{\mathcal{H},\mathsf{SetupG},\mathcal{B}}(\lambda).$$

Proof. We reuse the proof described in [ABN10, Lemma 6.1] by applying minor modifications. In fact, HDHI and HDHI2 are, respectively, slightly different from ODH and ODH2 notions: in the related security games, if $b = 0$ the challenges are constructed as $\mathsf{H}([\![u]\!] \| [\![uv]\!])$ and $\{\mathsf{H}([\![u]\!] \| [\![uv_0]\!]), \mathsf{H}([\![u]\!] \| [\![uv_1]\!])\}$ instead of $\mathsf{H}([\![uv]\!])$ and $\{\mathsf{H}([\![uv_0]\!]), \mathsf{H}([\![uv_1]\!])\}$. By accordingly changing the instances of H in the games $\mathsf{G}_0, \mathsf{G}_1, \mathsf{G}_2$ of [ABN10, Lemma 6.1] our lemma follows. □

1.5.8 Pseudo Random Functions

Informally, a pseudorandom function family $\mathcal{PRF} = \{\mathsf{PRF}_k : D \to C\}_{k \in \mathcal{K}}$ is a collection of functions such that for a randomly chosen $k \in \mathcal{K}$, the function PRF_k is indistinguishable from a random function that maps D to C.

Definition 1.5.19 (PRF Family [KL14, Definition 3.23])**.** Let $\mathcal{F} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. We say $\mathcal{F}$ is a pseudo random function if for all probabilistic polynomial-time distinguishers Dist, there exists a negligible function negl such that:

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathcal{F},\mathsf{Dist}}(\lambda) = \left| \Pr\left[\mathsf{Dist}^{\mathcal{F}_k(\cdot)}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Dist}^{f_\lambda(\cdot)}(1^\lambda) = 1\right] \right| \leq \mathsf{negl}(\lambda)\,,$$

where $k \leftarrow\$ \mathcal{K} = \{0,1\}^\lambda$ is chosen uniformly at random and f_λ is chosen uniformly at random from the set of functions mapping λ-bit strings to λ-bit strings.

1.5.9 Commitment scheme

Definition 1.5.20 (Non-interactive commitment scheme [BCC+15, Section 2.1])**.** A non-interactive commitment scheme ComSch is defined by the following algorithms:

- Setup, is a PPT algorithm that takes a security parameter 1^λ and outputs public parameters pp.

- Com, is a polynomial-time algorithm that takes a message $m \in \mathbb{B}^{\mathsf{il}}$, a random coin $r \in \mathbb{B}^{\mathsf{nl}}$ and outputs a commitment $cm \in \mathbb{B}^{\mathsf{ol}}$.

We assume that pp is implicitly passed to Com.

Definition 1.5.21 (Computational hiding)**.** We say that a commitment scheme is computationally hiding if for all PPT adversary $\mathcal{A}$, the advantage:

$$\left| \Pr\left[\begin{matrix} pp \leftarrow \mathsf{Setup}(1^\lambda), (m_0, m_1) \leftarrow \mathcal{A}(pp), b \leftarrow\!\!\$\, \{0,1\}, \\ r \leftarrow\!\!\$\, \mathbb{B}^{\mathsf{nl}}, cm \leftarrow \mathsf{Com}(m_b; r), \widetilde{b} \leftarrow \mathcal{A}(cm), b = \widetilde{b} \end{matrix}\right] - \frac{1}{2} \right|$$

is at most negligible in λ.

Definition 1.5.22 (Computational binding)**.** We say that a commitment scheme is computationally binding if for all PPT adversary $\mathcal{A}$, the advantage:

$$\Pr\left[\begin{matrix} pp \leftarrow \mathsf{Setup}(1^\lambda), (m_0, r_0, m_1, r_1) \leftarrow \mathcal{A}(pp) \\ m_0 \neq m_1 \wedge \mathsf{Com}(m_0; r_0) = \mathsf{Com}(m_1; r_1) \end{matrix}\right]$$

is at most negligible in λ.

Note that the previous definitions can be made *statistical* if we consider unbounded attackers $\mathcal{A}$.

1.5.10 Digital Signature

Definition 1.5.23 (Digital signature [KL14, Definition 12.1])**.** A digital signature scheme SigSch is defined by the tuple of functions $\mathsf{SigSch} = (\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$,

- $(sk, vk) \leftarrow \mathsf{KGen}(1^\lambda)$. Key Generation randomized algorithm takes as input the security parameter 1^λ and returns a signing key sk and verifying key vk.

- $\sigma \leftarrow \mathsf{Sig}(sk, m)$. Given a signing key sk and a message m, the Sig algorithm computes and outputs a signature σ.

- $\{0,1\} \leftarrow \mathsf{Vf}(vk, m, \sigma)$. Given a verification key vk, a message m and a signature σ, the Vf algorithm returns 1 if σ is a valid signature else 0.

A signature scheme must satisfy the *correctness property* (i.e $\mathsf{Vf}(vk, m, \mathsf{Sig}(sk, m)) = \mathsf{true}$, where $(sk, vk) \leftarrow \mathsf{KGen}(1^\lambda)$) and be *unforgeable* (i.e. it is intractable to produce a signature, without knowing the signing key sk, on a message that has not been signed yet). In addition to these properties, certain digital signature schemes have an additional property called *one-timeness*, also defined below.

$\mathsf{UF\text{-}CMA}(1^\lambda, t, q)$

$$
\begin{array}{l}
(sk, vk) \leftarrow \mathsf{KGen}(1^\lambda) \\
state \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Sig}_{sk}}}(vk, \cdot) \\
\quad /\!/ \ state = \{(m_i, \sigma_i)\}_{i \in [q]} \text{ where } m_i \text{ denotes} \\
\quad /\!/ \text{ the ith query made to } \mathsf{O}^{\mathsf{Sig}_{sk}} \text{ and} \\
\quad /\!/ \ \sigma_i \text{ denotes the ith oracle answers} \\
(m^*, \sigma^*) \leftarrow \mathcal{A}(state) \\
\textbf{return } \mathsf{Vf}(vk, m^*, \sigma^*) = 1 \\
\quad \wedge\, m^* \notin \{m_i\}_{i \in [q]}
\end{array}
$$

Figure 1.7: UF-CMA game

$\mathsf{SUF\text{-}CMA}(1^\lambda, t, q)$

$$
\begin{array}{l}
(sk, vk) \leftarrow \mathsf{KGen}(1^\lambda) \\
state \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Sig}_{sk}}}(vk, \cdot) \\
\quad /\!/ \ state = \{(m_i, \sigma_i)\}_{i \in [q]} \text{ where } m_i \text{ denotes} \\
\quad /\!/ \text{ the ith query made to } \mathsf{O}^{\mathsf{Sig}_{sk}} \text{ and} \\
\quad /\!/ \ \sigma_i \text{ denotes the ith oracle answers} \\
(m^*, \sigma^*) \leftarrow \mathcal{A}(state) \\
\textbf{return } \mathsf{Vf}(vk, m^*, \sigma^*) = 1 \\
\quad \wedge\, (m^*, \sigma^*) \notin \{(m_i, \sigma_i)\}_{i \in [q]}
\end{array}
$$

Figure 1.8: SUF-CMA game

Definition 1.5.24 (Unforgeability (UF-CMA) [KL14, Definition 12.2])**.** A digital signature scheme SigSch is UF-CMA if for any PPT adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the UF-CMA game, depicted in Fig. 1.7, is negligible.

Definition 1.5.25 (Strong Unforgeability (SUF-CMA))**.** A digital signature scheme SigSch is SUF-CMA if the probability that any PPT adversary $\mathcal{A}$ wins the SUF-CMA game, depicted in Fig. 1.8, is negligible.

Definition 1.5.26 (One-Time (OT) Signature [KL14, Definition 12.6])**.** A *one-time* signature scheme is a digital signature scheme that uses each key-pair at most once.

Remark 1.5.27. It is worth noting that users may use one-time signing keys to sign multiple messages. In this case no security claims can be made.

1.5.11 Message Authentication Code

A message authentication code is a scheme that enables users to tag data for the purpose of authenticity and integrity. Formally:

Definition 1.5.28 (Message Authentication Code,[KL14, Definition 4.1])**.** A message authentication code MAC is given by a tuple of PPT algorithms $(\mathsf{KGen}, \mathsf{Tag}, \mathsf{Vf})$ where:

- KGen, the key generation algorithm, takes a security parameter 1^λ, and returns a key $mk \in \{0,1\}^{mLen(\lambda)}$.[10]
- Tag, the tag generation algorithm, takes a key mk and a message $y \in \{0,1\}^*$ and returns a string $\tau \in \{0,1\}^*$, called *tag*.
- Vf, the tag verification algorithm, takes a key mk, a message $y \in \{0,1\}^*$ and a tag $\tau \in \{0,1\}^*$. It returns a value in $\{0,1\}$ where: 0 denotes that the message was rejected (i.e. deemed unauthentic) and 1 denotes that the message was accepted (i.e. deemed authentic).

[10]For simplicity, we may denote $mLen(\lambda)$ as $mLen$.

$$
\begin{array}{l}
\underline{\mathsf{SUF\text{-}CMA}\ (\lambda)} \\
mk \leftarrow \mathsf{KGen}(1^{\lambda}) \\
(\overline{y}, \overline{\tau}) \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Tag}_{mk}}, \mathsf{O}^{\mathsf{Vf}_{mk}}} \\
\textbf{return } \mathsf{Vf}(mk, \overline{y}, \overline{\tau}) = 1
\end{array}
$$

Figure 1.9: SUF-CMA game.

We require that for all $mk \in \{0,1\}^{\lambda}$ and $y \in \{0,1\}^*$ we have $\mathsf{Vf}(mk, y, \mathsf{Tag}(mk, y)) = 1$. If $\mathsf{Tag}(mk, \cdot)$ is defined only over messages of length $l(\lambda)$ and $\mathsf{Vf}(mk, y, \tau)$ outputs 0 for every y that is not of length $l(\lambda)$, then we say that $(\mathsf{KGen}, \mathsf{Tag}, \mathsf{Vf})$ is a *fixed-length* MAC with length parameter $l(\lambda)$.

A security notion for MAC follows:

Definition 1.5.29 (SUF-CMA,[ABR99, Section 3.2.3])**.** Let $\mathsf{MAC} = (\mathsf{KGen}, \mathsf{Tag}, \mathsf{Vf})$ be a message authentication scheme and let $\mathcal{A}$ be an adversary. Consider the SUF-CMA game described in Figure 1.9, with the condition that $\mathsf{Tag}(mk, \overline{y}) \neq \overline{\tau}$. We say that an adversary $\mathcal{A}$ has *forged* a tag when it outputs a pair $(\overline{y}, \overline{\tau})$ such that $\mathsf{Vf}_k(\overline{y}, \overline{\tau}) = 1$, where $(\overline{y}, \overline{\tau})$ was not previously obtained via a query to the tag oracle.

We define the SUF-CMA advantage of $\mathcal{A}$ as follows:

$$\mathsf{Adv}^{\mathsf{suf\text{-}cma}}_{\mathsf{MAC},\mathcal{A}}(\lambda) = \Pr[\mathsf{SUF\text{-}CMA}(\lambda) = 1]$$

We say that MAC is SUF-CMA secure if for every PPT adversary $\mathcal{A}$ the advantage $\mathsf{Adv}^{\mathsf{suf\text{-}cma}}_{\mathsf{MAC},\mathcal{A}}(\lambda)$ is a negligible function.

Chapter 2

Zeth protocol

In this section, we detail the `Zeth` protocol and provide a set of requirements that need to be respected to guarantee the security of the protocol.

2.1 `Zeth` Data Types

We begin by describing, and giving intuition about, the data types (see Section 1.1) used in `Zeth`. We follow some design rationale from `ZeroCash` [BSCG+14], and `Zcash` [ZCa19] in order to prevent the transaction malleability attack, and the Faerie Gold attack[ZCa19, Section 8.4]. We refer the reader to Appendix A for more details.

In what follows Curve represents a curve with scalar field $\mathbb{F}_{\mathtt{r}}$, satisfying the requirements of Section 3.6. The specification is described in terms of this generic curve, with examples and notes relating to specific instances of interest (namely BN-254 and BLS12-377, see Chapter 3).

`ZethNoteDType` Represents a note in `Zeth`. This data type consists of the note owner's public address *apk*, identifier ρ, randomness r and value v.

Field	Description	Data type
apk	Note owner's paying key	$\mathbb{B}^{\mathtt{PRFADDROUTLEN}}$
r	Note randomness	$\mathbb{B}^{\mathtt{RTRAPLEN}}$
v	Note value	$\mathbb{B}^{\mathtt{ZVALUELEN}}$
ρ	Note identifier	$\mathbb{B}^{\mathtt{PRFRHOOUTLEN}}$

Table 2.1: `ZethNoteDType` data type

`JSInputDType` Denotes a joinsplit input. It comprises the opening of a commitment *cm* which is in the set of leaves in the Merkle tree of $\widetilde{\mathbf{Mixer}}$ (i.e. a *ZethNote*), its address *mkaddr* and authentication path *mkpath* on the contract's Merkle tree as well as the spending key *ask* of the note holder and the note nullifier *nf*.

Field	Description	Data type
mkpath	Merkle authentication path to the commitment corresponding to the *ZethNote* to spend	$(\mathbb{F}_{\mathtt{r}})^{\mathtt{MKDEPTH}}$
mkaddr	Commitment address in the Merkle tree	$\mathbb{B}^{\mathtt{MKDEPTH}}$
znote	`Zeth` note object	`ZethNoteDType`
cm	`Zeth` note commitment	$\mathbb{F}_{\mathtt{r}}$
ask	Note owner's spending key	$\mathbb{B}^{\mathtt{ASKLEN}}$
nf	Note nullifier	$\mathbb{B}^{\mathtt{PRFNFOUTLEN}}$

Table 2.2: `JSInputDType` data type

`PrimInputDType` Represents the primary inputs used to generate the zk-SNARK proof π. *prim* is a tuple defined as the current Merkle root *mkroot* of the Merkle tree maintained by **Mixer**, the input notes nullifiers $nfs = (nf_0, \ldots, nf_{\mathtt{JSIN}-1})$, the output notes commitments $cms = (cm_0, \ldots, cm_{\mathtt{JSOUT}-1})$, the signature hash *hsig*, the message authentication tags $htags = (h_0, \ldots, h_{\mathtt{JSIN}-1})$ and the residual bits field *rsd*, which aggregates the former's fields bits which could not be contained in a field element.

Field	Description	Data type
mkroot	Merkle root of the Merkle tree	$\mathbb{F}_{\mathtt{r}}$
nfs	Indexed set of nullifiers of the "old" notes to spend (see Section 3.3.1 for definition of `NFFLEN`)	$((\mathbb{F}_{\mathtt{r}})^{\mathtt{NFFLEN}})^{\mathtt{JSIN}}$
cms	Indexed set of commitments to the newly created notes	$(\mathbb{F}_{\mathtt{r}})^{\mathtt{JSOUT}}$
hsig	Signature hash (non-malleability, see Appendix A and Section 3.3.1 for definition of `HSIGFLEN`))	$(\mathbb{F}_{\mathtt{r}})^{\mathtt{HSIGFLEN}}$
htags	Indexed set of message authentication tags (non-malleability, see Appendix A and Section 3.3.1 for definition of `HFLEN`))	$((\mathbb{F}_{\mathtt{r}})^{\mathtt{HFLEN}})^{\mathtt{JSIN}}$
rsd	Residual bits corresponding to unpacked bits of former fields (see Section 3.3.1 for definition of `RSDFLEN`)	$(\mathbb{F}_{\mathtt{r}})^{\mathtt{RSDFLEN}}$

Table 2.3: `PrimInputDType` data type

`AuxInputDType` Represents the auxiliary inputs used to generate the zk-SNARK proof π. *aux* is a tuple defined as joinsplit inputs (i.e. "old outputs to be spent"), the new *ZethNotes*, the joinsplit's randomness ϕ as well the public values *vin* and *vout*, the signature hash *hsig* and the message authentication tags $htags = (h_0, \ldots, h_{\texttt{JSIN}-1})$.

Field	Description	Data type
jsins	Indexed set of `JSIN` joinsplit inputs	$\texttt{JSInputDType}^{\texttt{JSIN}}$
znotes	Indexed set of `JSOUT` newly created notes	$\texttt{ZethNoteDType}^{\texttt{JSOUT}}$
ϕ	The joinsplit randomness (non-malleability, see Appendix A)	$\mathbb{B}^{\texttt{PHILEN}}$
vin	Public input value to the joinsplit	$\mathbb{B}^{\texttt{ZVALUELEN}}$
vout	Public output value to the joinsplit	$\mathbb{B}^{\texttt{ZVALUELEN}}$
hsig	Signature hash (non-malleability, see Appendix A)	$\mathbb{B}^{\texttt{CRHHSIGOUTLEN}}$
htags	Indexed set of message authentication tags (non-malleability, see Appendix A)	$(\mathbb{B}^{\texttt{PRFPKOUTLEN}})^{\texttt{JSIN}}$

Table 2.4: `AuxInputDType` data type

`MixInputDType` Represents the set of inputs to the Mix function of $\widetilde{\textbf{Mixer}}$. The input of the Mix function is a tuple defined as the primary inputs *prim*, the zk-proof π, the ciphertexts of the newly created notes $ciphers = (ct_0, \ldots, ct_{\texttt{JSOUT}-1})$, a one-time signature σ and the associated verification key *vk*.

Field	Description	Data type
primIn	Primary input object associated with the zk-proof π	`PrimInputDType`
proof	The zk-SNARK associated to the `Zeth` statement (see Section 2.2)	`ZKPDType` (see Section 3.6)
otssig	The one-time signature used to prevent transaction malleability (see Appendix A)	`SigOtsDType` (see Section 3.4.2)
otsvk	The verification key associated with the signature *otssig* used to prevent transaction malleability (see Appendix A)	`VKOtsDType` (see Section 3.4.2)
ciphers	Indexed set of ciphertexts of the newly generated notes	$(\mathbb{B}^{\texttt{ENCZETHNOTELEN}})^{\texttt{JSOUT}}$ (see Section 3.5)

Table 2.5: `MixInputDType` data type

`MixEventDType` Represents the data emitted as an `Ethereum` event (Section 1.2.3) during a successful execution of the Mix function of **Mixer**. Clients are required to read this data and use it to update their representation of **Mixer**'s state.

Field	Description	Data type
mkroot	New root of Merkle tree of commitments	$\mathbb{F}_{\mathtt{r}}$
nfs	Nullifiers for input notes consumed	$(\mathbb{B}^{\mathtt{PRFNFOUTLEN}})^{\mathtt{JSIN}}$
cms	Commitments to the output notes	$(\mathbb{F}_{\mathtt{r}})^{\mathtt{JSOUT}}$
ciphers	Ciphertexts for the output notes	$(\mathbb{B}^{\mathtt{ENCZETHNOTELEN}})^{\mathtt{JSOUT}}$

Table 2.6: `MixEventDType` data type

2.2 `Zeth` statement

As explained in [RZ19], the Mix function of **Mixer** verifies the validity of π on the given primary inputs in order to determine whether the state transition is valid. As such, **Mixer** verifies whether for π, and primary input *prim*, there exists an auxiliary input *aux*, such that the tuple (*prim*, *aux*) satisfies the NP-relation $\mathbf{R}^{\mathsf{z}}$, consisting of the following constraints:

- For each $i \in [\mathtt{JSIN}]$:
  1. $aux.jsins[i].znote.apk = \mathsf{PRF}^{\mathsf{addr}}_{aux.jsins[i].ask}(0)$
  2. $aux.jsins[i].cm = \mathsf{ComSch.Com}(aux.jsins[i].znote.apk, aux.jsins[i].znote.\rho, aux.jsins[i].znote.v; aux.jsins[i].znote.r)$
  3. $aux.jsins[i].nf = \mathsf{PRF}^{\mathsf{nf}}_{aux.jsins[i].ask}(aux.jsins[i].znote.\rho)$
  4. $aux.htags[i] = \mathsf{PRF}^{\mathsf{pk}}_{aux.jsins[i].ask}(i, aux.hsig)$ (non-malleability, see Appendix A)
  5. $(aux.jsins[i].znote.v) \cdot (1 - e) = 0$ is satisfied for the boolean value e set such that if $aux.jsins[i].znote.v > 0$ then $e = 1$.
  6. The Merkle root $mkroot'$ obtained after checking the Merkle authentication path $aux.jsins[i].mkpath$ of commitment $aux.jsins[i].cm$, with MKHASH, equals to $prim.mkroot$ if $e = 1$.
  7. $prim.nfs[i]$
     $= \{\mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.jsins[i].nf[k \cdot \mathtt{FIELDCAP}{:}(k+1) \cdot \mathtt{FIELDCAP}])\}_{k \in [\lfloor \mathtt{PRFNFOUTLEN}/\mathtt{FIELDCAP} \rfloor]}$
     (see Section 3.3.1 for definition of Pack)
  8. $prim.htags[i]$
     $= \{\mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.htags[i][k \cdot \mathtt{FIELDCAP}{:}(k+1) \cdot \mathtt{FIELDCAP}])\}_{k \in [\lfloor \mathtt{PRFPKOUTLEN}/\mathtt{FIELDCAP} \rfloor]}$
     (see Section 3.3.1 for definition of Pack)
- For each $j \in [\mathtt{JSOUT}]$:

1. $aux.znotes[j].\rho = \mathsf{PRF}^{\mathsf{rho}}_{aux.\phi}(j, aux.hsig)$ (non-malleability, see Appendix A)
2. $prim.cms[j] = \mathsf{ComSch.Com}(aux.znotes[j].apk, aux.znotes[j].\rho, aux.znotes[j].v;$ $aux.znotes[j].r)$

- $prim.hsig = \{\mathsf{Pack}_{\mathbb{F}_r}(aux.hsig[k \cdot \texttt{FIELDCAP}{:}(k+1) \cdot \texttt{FIELDCAP}])\}_{k \in [\lfloor \texttt{CRHHSIGOUTLEN}/\texttt{FIELDCAP} \rfloor]}$ (see Section 3.3.1 for definition of Pack)
- $prim.rsd = \mathsf{Pack}_{rsd}(\{aux.jsins[i].nf\}_{i \in [\texttt{JSIN}]}, aux.vin, aux.vout, aux.hsig, \{aux.htags[i]\}_{i \in [\texttt{JSIN}]})$ (see Section 3.3.1 for definition of Pack_{rsd})
- Check that the "joinsplit is balanced", i.e. check that the joinsplit equation holds:[1]

$$\mathsf{Pack}_{\mathbb{F}_r}(aux.vin) + \sum_{i \in [\texttt{JSIN}]} \mathsf{Pack}_{\mathbb{F}_r}(aux.jsins[i].znote.v)$$
$$= \sum_{j \in [\texttt{JSOUT}]} \mathsf{Pack}_{\mathbb{F}_r}(aux.znotes[j].v) + \mathsf{Pack}_{\mathbb{F}_r}(aux.vout)$$

2.3 Generating the inputs of the Mix function (Mix_{in})

In the following section, we assume that the system is initialized. In other words, we assume that a ledger L is available (i.e. an `Ethereum` network is operated by a set of miners), the $\widetilde{\textbf{Mixer}}$ contract is deployed on L. Likewise, we assume that the public parameters $pp_{\mathsf{ZkSnarkSch}} \leftarrow \mathsf{ZkSnarkSch.KGen}(1^\lambda, \mathbf{R^z})$ are available to $\widetilde{\textbf{Mixer}}$ and to all parties willing to call the Mix function of $\widetilde{\textbf{Mixer}}$. Furthermore, we assume that there exists a set of `Ethereum` and `Zeth` users, and that the *payment address* of each `Zeth` user is easily discoverable. In the rest of this section, the set of *payment addresses* discovered by a zeth user $\mathcal{U}_{\mathcal{Z}}$ is represented as a list attribute $\mathcal{U}_{\mathcal{Z}}.keystore$ indexed by usernames.

In order for $\mathcal{U}_{\mathcal{Z}}$ to transact via `Zeth`, $\mathcal{U}_{\mathcal{Z}}$ needs to create an object Mix_{in} of type `MixInputDType` to pass to the Mix function of $\widetilde{\textbf{Mixer}}$:

1. Create an object *prim* of type `PrimInputDType` to represent the primary input, and an object *aux* of type `AuxInputDType` to represent the auxiliary input, where:
   (a) $prim.mkroot \in Roots$, where $Roots$ is the set of *all* Merkle roots corresponding to one of the state of the Merkle tree on $\widetilde{\textbf{Mixer}}$ containing *all* the commitments to the input notes, in $aux.jsins$, in its set of leaves.
   (b) $aux.znotes[j].r \leftarrow\$\, \mathbb{B}^{\texttt{RTRAPLEN}}, \forall j \in [\texttt{JSOUT}]$, and $aux.\phi \leftarrow\$\, \mathbb{B}^{\texttt{PHILEN}}$
   (c) The public values $(aux.vin, aux.vout) \in (\mathbb{B}^{\texttt{ZVALUELEN}})^2$, $aux.znotes[j].v$ and $aux.znotes[j].apk$ $\forall j \in [\texttt{JSOUT}]$ are all set by the sender, $\mathcal{U}_{\mathcal{Z}}$, as desired as long as they satisfy the joinsplit equation.

[1] where $\mathsf{Pack}_{\mathbb{F}_r}(x)$ outputs the numerical value of x in $\mathbb{F}_r$. We rely on the fact that $\texttt{ZVALUELEN} < \texttt{FIELDCAP}$ to perform this sum.

(d) All attributes of the *prim* and *aux* objects should be derived as specified in the statement (see Section 2.2), alongside a signature hash ($aux.hsig$) that is generated as the hash of the nullifiers and a one-time signing verification key (non-malleability, see Appendix A), using the desired signature scheme $\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}$ (see Section 3.4):

$$(sk_{\mathsf{OT\text{-}SIG}}, vk_{\mathsf{OT\text{-}SIG}}) = \mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}.\mathsf{KGen}(1^{\lambda}) \tag{2.1}$$

$$aux.hsig = \mathsf{CRH}^{\mathsf{hsig}}(\{aux.jsins[i].nf\}_{i \in [\mathtt{JSIN}]}, vk_{\mathsf{OT\text{-}SIG}}) \tag{2.2}$$

(e) $\mathsf{Mix}_{in}.primIn \leftarrow prim$

Note

If one of the attributes of *prim* and *aux* is not correctly generated, then the proof of computational integrity generated in the next step will be rejected on **Mixer**, and the state of **Mixer** will not be modified.

2. Generate a zk-SNARK proof π to prove, in zero-knowledge, that the relation $\mathbf{R}^{\mathsf{z}}$ (Section 2.2) holds on the primary and auxiliary inputs, using the desired zk-SNARK scheme $\mathsf{ZkSnarkSch}$ (see Section 3.6):

   (a) $\pi \leftarrow \mathsf{ZkSnarkSch.P}(pp_{\mathsf{ZkSnarkSch}}, prim, aux)$

   (b) $\mathsf{Mix}_{in}.proof \leftarrow \pi$

3. Encrypt all the $aux.znotes$ using the recipient's *payment address*, using the encryption scheme EncSch (see Section 3.5).

   (a) For all $j \in [\mathtt{JSOUT}]$, do:

   $$ct_j \leftarrow \mathsf{EncSch.Enc}(aux.znotes[j], \mathcal{U}_{\mathcal{Z}}.keystore[recipient_j].pub.pkenc)$$

   (b) $\mathsf{Mix}_{in}.ciphers \leftarrow \{ct_j\}_{j \in [\mathtt{JSOUT}]}$

4. Generate a signature $\sigma_{\mathsf{OT\text{-}SIG}}$ on the inputs of the Mix function, in order to prevent any malleability attacks (c.f. Appendix A), using the desired signature scheme $\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}$ (see Section 3.4):

   (a) Using the one-time signature keypair generated in Eq. (2.1), do:

   $$dataToBeSigned = \mathcal{S}_{\mathcal{E}}.Addr \| \mathsf{Mix}_{in}.primIn \| \mathsf{Mix}_{in}.\pi \| \mathsf{Mix}_{in}.ciphers$$
   $$\sigma_{\mathsf{OT\text{-}SIG}} = \mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}.\mathsf{Sig}(sk_{\mathsf{OT\text{-}SIG}}, \mathsf{CRH}^{\mathsf{ots}}(dataToBeSigned))$$

   (b) $\mathsf{Mix}_{in}.otssig \leftarrow \sigma_{\mathsf{OT\text{-}SIG}}$

   (c) $\mathsf{Mix}_{in}.otsvk \leftarrow vk_{\mathsf{OT\text{-}SIG}}$

   Here, $\mathcal{S}_{\mathcal{E}}.Addr$ represents the address of the `Ethereum` user $\mathcal{S}_{\mathcal{E}}$ who must sign the transaction (see Section 2.4). In general, this is likely to be owned by the holder $\mathcal{U}_{\mathcal{Z}}$ of the `Zeth` notes to be spent, but this is not a requirement.

2.4 Creating an `Ethereum` transaction tx_{Mix} to call $\widetilde{\textbf{Mixer}}$

After generating a Mix_{in} object, $\mathcal{U}_{\mathcal{Z}}$ can generate an object tx_{raw} of type `TxRawDType`, such that:

$$tx_{raw}.to = \widetilde{\textbf{Mixer}}.Addr \wedge tx_{raw}.data = zdata$$

Then, an `Ethereum` user $\mathcal{S}_{\mathcal{E}}$ can ECDSA sign tx_{raw}, under $\mathcal{S}_{\mathcal{E}}.sk$ in order to transform this object of type `TxRawDType` into an finalized transaction, i.e. an object tx_{Mix} of type `TxDType`.

Finally, the transaction tx_{Mix} is broadcasted on the `Ethereum` network and eventually gets mined.

Note

Here, the `Ethereum` user $\mathcal{S}_{\mathcal{E}}$ who sends the final transaction, and the `Zeth` user $\mathcal{U}_{\mathcal{Z}}$ may represent the same person or entity, but this is not necessarily the case. It is perfectly feasible (and in some cases may be desirable) for a `Zeth` user $\mathcal{U}_{\mathcal{Z}}$ to create a `Zeth` transaction which is later signed by a distinct party $\mathcal{S}_{\mathcal{E}}$. In particular, the only identifying information that appears in plaintext on the ledger will be that of $\mathcal{S}_{\mathcal{E}}$.

2.5 Processing tx_{Mix}

When a tx_{Mix} is mined (hence assuming that $\mathsf{EthVerifyTx}(tx_{\mathsf{Mix}})$ returns true), the state transition specified by the Mix function of $\widetilde{\textbf{Mixer}}$ is executed.

To preserve the soundness of `Zeth`, and make sure that no $\mathcal{U}_{\mathcal{Z}}$ is able to create value by double spending *ZethNotes*, various checks need to be satisfied. The function $\mathsf{ZethVerifyTx}$ is defined as the function that returns true if all the checks are satisfied, and false otherwise.

If $\mathsf{ZethVerifyTx}(tx_{\mathsf{Mix}})$ returns true, then Mix modifies the "World state" ς to account for the spent *ZethNotes* and the newly generated ones. However, if $\mathsf{ZethVerifyTx}(tx_{\mathsf{Mix}})$ returns false, then the state transition ends.

Note

Even if $\mathsf{ZethVerifyTx}(tx_{\mathsf{Mix}})$ returns false, ς is modified since the `Ethereum` balances of the transaction originator is decremented by the sum of `DGAS` and the gas consumed by the $\mathsf{ZethVerifyTx}$ function, and the balance of the `Ethereum` account of the miner gets incremented by the same amount.

Thus, Mix proceeds as follows:

1. Check that all the values of the primary inputs' ($\mathsf{Mix}_{in}.primIn$) entries are elements of the scalar field over which the zk-proof is generated:

$$\mathsf{Mix}_{in}.primIn \in \mathbb{F}_{\mathtt{r}}^{*}$$

2. Unpack the nullifiers, signature hash and public values (see Section 3.3.1 for the definitions of the Unpack functions):

$$\begin{aligned}
nf_i &= \mathsf{Unpack}_{nf}(\mathsf{Mix}_{in}.primIn.nfs[i], \mathsf{Mix}_{in}.primIn.rsd) \ \forall i \in [\mathtt{JSIN}] \\
vin &= \mathsf{decode}_{\mathbb{N}}(\mathsf{Unpack}_{vin}((), \mathsf{Mix}_{in}.primIn.rsd)) \\
vout &= \mathsf{decode}_{\mathbb{N}}(\mathsf{Unpack}_{vout}((), \mathsf{Mix}_{in}.primIn.rsd)) \\
hsig &= \mathsf{Unpack}_{hsig}(\mathsf{Mix}_{in}.primIn.hsig, \mathsf{Mix}_{in}.primIn.rsd)
\end{aligned}$$

3. Check the validity of the tx_{Mix} object ($\mathsf{ZethVerifyTx}$):

   - Check that $\mathsf{Mix}_{in}.primIn.hsig$ is correctly computed, i.e. check that the following equation holds (to prevent transaction malleability, see Appendix A):

   $$hsig = \mathsf{CRH}^{\mathsf{hsig}}(\mathsf{Mix}_{in}.primIn.nfs, \mathsf{Mix}_{in}.otsvk)$$

   - Check that π is a valid zk-SNARK proof for $\mathsf{Mix}_{in}.primIn$, i.e. check that:

   $$\mathsf{ZkSnarkSch.V}(pp_{\mathsf{ZkSnarkSch}}, \pi, \mathsf{Mix}_{in}.primIn) = \mathsf{true}$$

   - Check that none of the nullifiers in $\mathsf{Mix}_{in}.primIn.nfs$ have already been used, i.e. check that:

   $$nf_i \notin Nulls, \forall i \in [\mathtt{JSIN}]$$

   where $Nulls$ is the set of all nullifiers that are "declared" on $\widetilde{\textbf{Mixer}}$.

   - Check that $\mathsf{Mix}_{in}.otssig$ is a valid signature of the `Ethereum` sender's address $Addr$ (see Section 2.4) and the attributes of Mix_{in}, to prevent transaction malleability (see Appendix A), i.e. check that:

   $$\begin{aligned}
   &\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}.\mathsf{Vf}(\mathsf{Mix}_{in}.otsvk, m, \mathsf{Mix}_{in}.otssig) = \mathsf{true} \\
   &\text{where } m = \mathsf{CRH}^{\mathsf{ots}}(Addr \| \mathsf{Mix}_{in}.primIn \| \mathsf{Mix}_{in}.\pi \| \mathsf{Mix}_{in}.ciphers)
   \end{aligned}$$

   - Check that $\mathsf{Mix}_{in}.primIn.mkroot$ corresponds to a valid state of the Merkle tree held on $\widetilde{\textbf{Mixer}}$, i.e. check that:

   $$\mathsf{Mix}_{in}.primIn.mkroot \in Roots'$$

   where $Roots'$ is the set of all Merkle roots corresponding to one of the states of the Merkle tree.

   - Check that vin corresponds to the value val of the transaction object, i.e. check that:

   $$vin = tx_{\mathsf{Mix}}.val$$

4. If all checks above pass, i.e. if $\mathsf{ZethVerifyTx}(tx_{\mathsf{Mix}})$ returns true, then the following additional modifications are made in ς:

- Add the commitments $\mathsf{Mix}_{in}.primIn.cms$ to the Merkle tree held on $\widetilde{\textbf{Mixer}}$.
- $Roots' \leftarrow Roots' \cup \{mkroot'\}$, where $mkroot'$ is the Merkle root of the Merkle tree after insertion of the commitments $\mathsf{Mix}_{in}.primIn.cms$ in the Merkle tree.
- $Nulls \leftarrow Nulls \cup \{nf_i\}_{i \in [\texttt{JSIN}]}$, i.e. the nullifiers nfs become "declared".
- Modify the `Ethereum` balances according to the public values:
  - $\varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal = \varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal - vin$
  - $\varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal = \varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal + vout$
  - $\widetilde{\textbf{Mixer}}.bal = \widetilde{\textbf{Mixer}}.bal + vin$
  - $\widetilde{\textbf{Mixer}}.bal = \widetilde{\textbf{Mixer}}.bal - vout$
- Emit an event (Section 1.2.3) $evMixOut$ of type `MixEventDType`, containing the new root $mkroot'$ of the Merkle tree of commitments, the nullifiers $\{nf_i\}_{i \in [\texttt{JSIN}]}$, commitments to the newly created *ZethNotes* $\mathsf{Mix}_{in}.primIn.cms$, and the corresponding ciphertexts $\mathsf{Mix}_{in}.primIn.ciphers$.

2.6 Receiving *ZethNotes*

In order to confirm the reception of *ZethNotes*, $\mathcal{R}_{\mathcal{Z}}$ must listen to the events (Section 1.2.3) of type `MixEventDType` emitted by the processing of tx_{Mix}, and try to decrypt the ciphertexts using $\mathcal{R}_{\mathcal{Z}}.priv.skenc$ to see if he is the recipient of a `Zeth` payment. If the decryption is successful ($\mathcal{R}_{\mathcal{Z}}$ is the recipient of a payment), $\mathcal{R}_{\mathcal{Z}}$ must verify that the *ZethNote* recovered is the opening of a commitment in the Merkle tree of $\widetilde{\textbf{Mixer}}$. If not, $\mathcal{R}_{\mathcal{Z}}$ rejects the (invalid) payment.

We describe below the steps that $\mathcal{R}_{\mathcal{Z}}$ needs to carry out for all events $evMixOut \in$ `MixEventDType` emitted by $\widetilde{\textbf{Mixer}}$, in order to receive payments:

1. Compute the new root $mkroot'$ of the Merkle tree of commitments, after adding the new values $evMixOut.cms$. If this value does not match the new root $evMixOut.mkroot$ emitted by $\widetilde{\textbf{Mixer}}$, abort.
2. Try to decrypt the ciphertexts:
$$zn_j = \mathsf{EncSch.Dec}(\mathcal{R}_{\mathcal{Z}}.priv.skenc, evMixOut.ciphers[j])$$
3. For each successful decryption, let j be the index of the decrypted ciphertext:
   (a) Check whether the recovered plaintext zn_j is a well-formed *ZethNote*. Abort if it is not well-formed.
   (b) Check that the recovered *ZethNote* zn_j is the opening of the corresponding commitment $evMixOut.cms[j]$:
   $$evMixOut.cms[j] = \mathsf{ComSch.Com}(zn_j.apk, zn_j.\rho, zn_j.v; zn_j.r)$$
   Abort if the note is not a valid opening.

(c) Additionally, if sender $\mathcal{S}_{\mathcal{Z}}$, and recipient $\mathcal{R}_{\mathcal{Z}}$ had a contractual agreement, then $\mathcal{R}_{\mathcal{Z}}$ needs to check that the terms of this agreement are fulfilled by all the recovered *ZethNotes*, abort otherwise.

Note that Steps 1 and 3b are required to ensure that data decrypted by $\mathcal{R}_{\mathcal{Z}}$ exactly matches the data committed to in **Mixer**. In particular, Step 1 requires $\mathcal{R}_{\mathcal{Z}}$ to maintain or have access to some representation of the Merkle tree of commitments. See Section 4.1.2 for further details.

2.7 Security requirements for the primitives

We list below the security requirements to instantiate the primitives of the `Zeth` protocol.

- $\mathsf{CRH}^{\mathsf{hsig}}$ and $\mathsf{CRH}^{\mathsf{ots}}$ `MUST` be collision resistant functions (see Definition 1.5.16).
- $\mathsf{PRF}^{\mathsf{addr}}, \mathsf{PRF}^{\mathsf{nf}}, \mathsf{PRF}^{\mathsf{rho}}$ and $\mathsf{PRF}^{\mathsf{pk}}$ `MUST` be PRF when keyed by *ask* and ϕ, and be collision resistant (see Definition 1.5.16, and Section 1.5.8).
- $\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}$ `MUST` be $\mathsf{UF\text{-}CMA}$ (see Definition 1.5.24 and Appendix A.2.3).
- ComSch `MUST` be computationally hiding and binding (see Section 1.5.9).
- MKHASH `MUST` be collision resistant with $h_0 = 0_{\mathbb{F}_r}$ (see Section 1.5.6). [2]
- EncSch `MUST` be $\mathsf{IND\text{-}CCA2}$ and $\mathsf{IK\text{-}CCA}$ (see, respectively, [ABR99, Definition 8] and Definition 1.5.10).
- $\mathsf{Unpack}(\mathsf{Pack}(X)) = X$ and $\mathsf{Unpack}(\mathsf{Pack}_{rsd}(X)) = X$ `MUST` hold.
- $\mathsf{decode}(\mathsf{encode}(X)) = X$ `MUST` hold.

2.7.1 Additional notes

Defining *hsig*

The signature hash *hsig* is a variable used to bind the signature keys to the primary inputs. We use the same definition of *hsig* as `Zcash` to prevent the Faerie Gold attack and thus

$$hsig = \mathsf{CRH}^{\mathsf{hsig}}(nfs, vk)\,.$$

As a private transaction is uniquely determined by its nullifiers $nfs = (nf_0, \dots, nf_{\mathtt{JSIN}-1})$, and because of the collision resistance of $\mathsf{CRH}^{\mathsf{hsig}}$, a transaction is uniquely determined by *hsig* (with overwhelming probability). We did not use the *randomSeed* defined in `Zcash` however, since this is only necessary to achieve uniqueness of *hsig* for transactions *in transit* (i.e. not mined yet) [Hop16]. The uniqueness of *hsig* is a requirement to prevent the Fairy Gold attack.

[2] This security requirement is equivalent to the one in [ZCa19, Section 5.4.1.3] where finding a preimage of $0^{\mathtt{SHA256DLEN}}$ must be hard.

Security Requirement.

- The variable *hsig* `MUST` be derived from the nullifiers $\{nf_i\}_{i \in [\texttt{JSIN}]}$ and the signing key *vk* using a collision resistant function. Doing so, makes sure that *hsig* is unique for each tx_{Mix} with overwhelming probability.

Defining ρ

We define ρ like in `Zcash` in order to prevent the Faerie Gold attack. A malicious sender could reuse the same ρ for a given recipient, hence correctly generating a *ZethNote* which could become unspendable by the recipient. Making ρ the output of a collision resistant PRF with random variable ϕ as key and with tx_{Mix}'s *hsig* as input ensures, with overwhelming probability, the uniqueness of ρ and prevents this attack. Thus,

$$\rho_j = \mathsf{PRF}^{\mathsf{rho}}_{\phi}(j, hsig)\,.$$

Message authentication tags h_i

The message authentication tags are used to bind the signature hash to the input notes spending keys, to show ownership of the spent notes. Each tag derived from a note owner's spending key and the signature hash `MUST` be unique for each note with overwhelming probability. We define

$$h_i = \mathsf{PRF}^{\mathsf{pk}}_{ask_i}(i, hsig)\,.$$

Chapter 3

Instantiation of the cryptographic primitives

In this chapter, we start by instantiating the cryptographic building blocks used in previous sections to describe the Zeth DAP design. Finally, we proceed by providing security proofs justifying that our instantiation complies with the security requirements listed in previous sections.

Note that, in several cases, it is necessary to specify details in terms of concrete properties of the curve Curve and associated scalar field $\mathbb{F}_{\mathtt{r}}$. In these cases, we focus on two curves of interest: BN-254 and BLS12-377. We note, however, that other suitable curves could be used.

BN-254 [Rk19] has several properties that make it implementation-friendly. Elements of both the base field and scalar field can be represented in ETHWORDLEN bits (the native word size of the EVM), allowing efficient encoding and manipulation of such elements. Moreover, a subset of operations on BN-254 are supported by the EVM through precompiled contracts. These precompiled contracts enable verification of signatures (Section 3.4) and zero-knowledge proofs (Section 3.6), required by this protocol, with minimal gas overhead.

BLS12-377 [BCG+20], like BN-254, has the advantage that scalar field elements can be represented within ETHWORDLEN-bit words (although the same is not true of base field elements). However, the EVM provides no native support for BLS12-377, which increases the complexity of the **Mixer** implementation (see Section 2.5 for details of the operations to be performed). An advantage that BLS12-377 does provide, is that is it the "inner" curve of a one-layer chain (as described in [BCG+20, HG20]). Therefore zero-knowledge proofs using BLS12-377 can be efficiently verified by statements in other zero-knowledge proofs using an approporiate "outer" pairing. Support for BLS12-377 in Zeth therefore admits several applications (no explicitly covered by this document), such as aggregation of proofs over multiple Zeth transactions (e.g. [Ron20]).

Further details related to implementation and optimization are given in Chapter 4.

3.1 Instantiating the PRFs, ComSch and CRHs

The functions $\mathsf{CRH}^{\mathsf{hsig}}$ and $\mathsf{CRH}^{\mathsf{ots}}$ are instantiated with SHA256 [oST15] which we assume to be collision resistant. Furthermore, ComSch, $\mathsf{PRF}^{\mathsf{pk}}(x)$, $\mathsf{PRF}^{\mathsf{rho}}(x)$, $\mathsf{PRF}^{\mathsf{addr}}(x)$, and $\mathsf{PRF}^{\mathsf{nf}}(x)$ are all instantiated with Blake2's hash function optimized for 32-bit platforms, Blake2s, which we prove in the Weakly Ideal Cipher Model [LMN16] to be from a family of PRF and collision resistant functions. The Weakly Ideal Cipher model assumes that Blake2's underlying block cipher is ideal and has no structural weaknesses (see Appendix D.2). In addition to that, and to ensure that the functions $\mathsf{PRF}^{\mathsf{pk}}(x)$, $\mathsf{PRF}^{\mathsf{rho}}(x)$, $\mathsf{PRF}^{\mathsf{addr}}(x)$, and $\mathsf{PRF}^{\mathsf{nf}}(x)$ compute images lying in different domains, we use different message prefixes (or "domain separators") for the PRFs inputs. This approach ensures that the apk_i's, nf_i's, ρ_i's, and h_i's have independent distributions from a PPT adversary point of view.

Note

It is important to note that, for this approach to be secure, the hash function used needs to be secure against *chosen-prefix collision attacks* [Ste15].

Furthermore, we take:

- $\mathtt{RTRAPLEN}, \mathtt{ASKLEN}, \mathtt{PHILEN} = \mathtt{BLAKE2sCLEN}$

3.1.1 Blake2 primitive

Blake [AHMP08] is a hash family that was presented as a candidate at the SHA3 competition. Blake2 is the next iteration of the family which has been further optimized to achieve higher throughput thanks to some optimizations and by being less conservative on its security[1]. Blake and Blake2 are based on the ChaCha stream cipher [Ber08a] composed with the HAIFA framework [BD07]. ChaCha defined over 20 rounds, as used in Blake2, is deemed secure and a PRF based on today's cryptanalysis [Pro14, CM16]. Blake2 is specified in RFC-7693 [MJS15] and licensed under CC0. Blake2s is an instantiation of Blake2 optimized for 32-bit platforms. As such, to reason about the security of Blake2s we prove the security of Blake2.

Blake security Blake security has been heavily scrutinized through the SHA3 competition [VNP10, MQZ10, AMP10, AAM12, AMPŠ12, ALM12, HMRS12]. Blake2 has also been thoroughly cryptanalyzed independently [GKN$^+$14, Hao14, EFK15, NA19]. For n-bit long digests/outputs, the hash and compression functions present $n/2$-bit of collision resistance and n-bit of preimage resistance, immunity to length extension, and indifferentiability from a random oracle [ANWOW13]. They have furthermore been demonstrated secure in the Weakly Ideal Cipher Model [LMN16] (WICM, see Appendix D.1.1). More

[1] The authors increased the number of rounds of Blake for the SHA3 competition to be more conservative on security. They however showed afterwards that this change was not "meaningfully more secure" and thus reverted it for Blake2 (see [ANWOW13, Section 2.1]).

particularly, Luykx et al. show that $\mathsf{Blake2}$ is indifferentiable from a random oracle in this model and is a PRF. We use this result in Appendix D.2 to show the collision resistance of $\mathsf{Blake2}$. We also prove that, given that $\mathsf{Blake2}$ is collision resistant and a PRF, $\mathsf{Blake2}(r\|x)$ is a computationally binding and computationally hiding commitment scheme for input x and randomness r.

Note

We assume that the encryption scheme used in the $\mathsf{Blake2}$ underlying compression function – which is derived from $\mathsf{ChaCha20}$ – has no exploitable structural behaviour. More precisely, that this encryption scheme behaves like a weak ideal cipher. We provide proofs in this model.

3.1.2 Commitment scheme

We define our commitment scheme as follows,

$$
\begin{aligned}
&\mathsf{ComSch.Setup} &&: \left\{1^\lambda \ s.t\ \lambda \in \mathbb{N}\right\} \to \mathbb{B}^* \\
&\mathsf{ComSch.Com} &&: \left(\mathbb{B}^{\mathtt{PRFADDROUTLEN}} \times \mathbb{B}^{\mathtt{PRFRHOOUTLEN}} \times \mathbb{B}^{\mathtt{ZVALUELEN}}\right) \times \mathbb{B}^{\mathtt{RTRAPLEN}} \to \mathbb{F}_{\mathtt{r}}
\end{aligned}
$$

We instantiate the commitment scheme with $\mathsf{Blake2s}$ as follows,

$$
\begin{aligned}
pp &= \mathsf{ComSch.Setup}(1^\lambda) \text{ (corresponds to } \mathsf{Blake2s}\text{'s constant } \mathtt{PB} \text{ and } \mathtt{r}) \\
cm &= \mathsf{ComSch.Com}(m = (apk, \rho, v); r) \\
&= \mathsf{decode}_{\mathbb{N}}(\mathsf{Blake2s}(r\|apk\|\rho\|v)) \pmod{\mathtt{r}}
\end{aligned}
$$

Remark 3.1.1. We set the commitment digest length in the parameter block $\mathtt{PB}$ [MJS15].

Security proof

The commitment scheme defined above is computationally hiding and binding in the WICM, see Appendix D.2.4. However, because of the modulo $\mathtt{r}$ operation, the scheme is only $(\mathtt{FIELDLEN}/2)$-bit binding.

3.1.3 PRFs

We show in this section how we instantiate the PRFs with Blake primitives. As a reminder the PRFs are defined as follows,

$$
\begin{aligned}
\mathsf{PRF}^{\mathsf{addr}} &: \mathbb{B}^{\mathtt{ASKLEN}} \times \{0\} \to \mathbb{B}^{\mathtt{PRFADDROUTLEN}} \\
\mathsf{PRF}^{\mathsf{pk}} &: \left(\mathbb{B}^{\mathtt{ASKLEN}} \times [\mathtt{JSIN}]\right) \times \mathbb{B}^{\mathtt{CRHHSIGOUTLEN}} \to \mathbb{B}^{\mathtt{PRFPKOUTLEN}} \\
\mathsf{PRF}^{\mathsf{nf}} &: \mathbb{B}^{\mathtt{ASKLEN}} \times \mathbb{B}^{\mathtt{PRFRHOOUTLEN}} \to \mathbb{B}^{\mathtt{PRFNFOUTLEN}} \\
\mathsf{PRF}^{\mathsf{rho}} &: \left(\mathbb{B}^{\mathtt{PHILEN}} \times [\mathtt{JSOUT}]\right) \times \mathbb{B}^{\mathtt{CRHHSIGOUTLEN}} \to \mathbb{B}^{\mathtt{PRFRHOOUTLEN}}
\end{aligned}
$$

As we instantiate the PRFs with Blake2s we have,

$$\mathtt{PRFADDROUTLEN}, \mathtt{PRFNFOUTLEN}, \mathtt{PRFPKOUTLEN}, \mathtt{PRFRHOOUTLEN} = \mathtt{BLAKE2sCLEN}$$

To ensure that the PRFs have independent distributions, we first introduce tagging functions tag^x which truncate and prepend with a distinct tag the PRFs key. We have,

$$\begin{aligned}
&\mathsf{tag}^{\mathsf{addr}} : \mathbb{B}^{\mathtt{ASKLEN}} \to \mathbb{B}^{\mathtt{BLAKE2sCLEN}} \\
&\mathsf{tag}^{\mathsf{pk}} : \mathbb{B}^{\mathtt{ASKLEN}} \times [\mathtt{JSIN}] \to \mathbb{B}^{\mathtt{BLAKE2sCLEN}} \\
&\mathsf{tag}^{\mathsf{nf}} : \mathbb{B}^{\mathtt{ASKLEN}} \to \mathbb{B}^{\mathtt{BLAKE2sCLEN}} \\
&\mathsf{tag}^{\mathsf{rho}} : \mathbb{B}^{\mathtt{PHILEN}} \times [\mathtt{JSOUT}] \to \mathbb{B}^{\mathtt{BLAKE2sCLEN}}
\end{aligned}$$

The tagging functions are instantiated as follows,

$$\begin{aligned}
\mathsf{tag}^{\mathsf{addr}}(aux.jsins[i].ask) &= tag^{addr}_{ask} \\
&= (1)\|(1)^{\lceil \frac{\mathtt{JSMAX}}{2} \rceil}\|(0,0)\|\mathsf{trunc}_{\mathtt{BLAKE2sCLEN}-3-\lceil \frac{\mathtt{JSMAX}}{2} \rceil}(aux.jsins[i].ask) \\
\mathsf{tag}^{\mathsf{nf}}(aux.jsins[i].ask) &= tag^{nf}_{ask} \\
&= (1)\|(1)^{\lceil \frac{\mathtt{JSMAX}}{2} \rceil}\|(1,0)\|\mathsf{trunc}_{\mathtt{BLAKE2sCLEN}-3-\lceil \frac{\mathtt{JSMAX}}{2} \rceil}(aux.jsins[i].ask) \\
\mathsf{tag}^{\mathsf{pk}}(aux.jsins[i].ask, i) &= tag^{pk}_{ask,i} \\
&= (0)\|\mathsf{pad}_{\lceil \frac{\mathtt{JSMAX}}{2} \rceil}(\mathsf{encode}_{\mathbb{N}}(i))\|(0,0)\|\mathsf{trunc}_{\mathtt{BLAKE2sCLEN}-3-\lceil \frac{\mathtt{JSMAX}}{2} \rceil}(aux.jsins[i].ask) \\
\mathsf{tag}^{\mathsf{rho}}(aux.\phi, j) &= tag^{\rho}_{ask,j} \\
&= (0)\|\mathsf{pad}_{\lceil \frac{\mathtt{JSMAX}}{2} \rceil}(\mathsf{encode}_{\mathbb{N}}(j))\|(1,0)\|\mathsf{trunc}_{\mathtt{BLAKE2sCLEN}-3-\lceil \frac{\mathtt{JSMAX}}{2} \rceil}(aux.\phi)
\end{aligned}$$

where $\mathsf{pad}_{\lceil \frac{\mathtt{JSMAX}}{2} \rceil}(\mathsf{encode}_{\mathbb{N}}(i))$ is the function that pads the binary representation of i by adding 0's before the most significant bit (e.g. assuming big endian encoding, $\mathsf{pad}_2(\mathsf{encode}_{\mathbb{N}}(1)) = 01$).

We now present how the PRFs are instantiated,

$$\begin{aligned}
\mathsf{PRF}^{\mathsf{addr}}_{aux.jsins[i].ask}(0) &= aux.jsins[i].znote.apk \\
&= \mathsf{Blake2s}(\mathsf{tag}^{\mathsf{addr}}(aux.jsins[i].ask)\|\mathsf{pad}_{\mathtt{BLAKE2sCLEN}}(0)) \\
\mathsf{PRF}^{\mathsf{nf}}_{aux.jsins[i].ask}(aux.jsins[i].\rho) &= prim.nfs[i] \\
&= \mathsf{Blake2s}(\mathsf{tag}^{\mathsf{nf}}(aux.jsins[i].ask)\|aux.jsins[i].znote.\rho) \\
\mathsf{PRF}^{\mathsf{pk}}_{aux.jsins[i].ask}(i, prim.hsig) &= prim.htags[i] \\
&= \mathsf{Blake2s}(\mathsf{tag}^{\mathsf{pk}}(aux.jsins[i].ask, i)\|prim.hsig) \\
\mathsf{PRF}^{\mathsf{rho}}_{aux.\phi}(j, prim.hsig) &= aux.znotes[j].\rho \\
&= \mathsf{Blake2s}(\mathsf{tag}^{\mathsf{rho}}(aux.\phi, j)\|prim.hsig)
\end{aligned}$$

Remark 3.1.2. We set the PRFs' output length in the Blake2s's parameter block PB.

Security proof

The functions defined above are collision resistant and PRFs in the WICM, see Appendix D.2. Because of the tagging functions, the security parameter of the PRFs becomes $\lambda = \texttt{BLAKE2sCLEN}/2 - \texttt{JSMAX}/4 - 3/2$.

3.1.4 Collision resistant hashes

We instantiate in this section the collision resistant hash functions $\mathsf{CRH}^{\mathsf{hsig}}$ and $\mathsf{CRH}^{\mathsf{ots}}$ with SHA256. As a consequence, we have,

$$\texttt{CRHHSIGOUTLEN} = \texttt{CRHOTSOUTLEN} = \texttt{SHA256DLEN}$$

SHA256 **Security** SHA-256 (Secure Hash Algorithm 256) is a hash function designed by the National Security Agency (NSA) in 2001. It is based on the Merkle–Damgård structure, the Davies–Meyer compression function construct [BRS02, Function f_5 in Figure 3] and the classified SHACAL-2 block cipher.

Collision attacks have been thoroughly studied by the research community [SS08, MNS11]. The best attacks at this day, are second-order differential attack by Lamberger et al. [LM11] on the SHA-256 compression function reduced to 46 out of 64 rounds.

Many researchers [IS09, AGM+09] have also studied preimage attacks on SHA-256 with reduced rounds. Guo et al. [GLRW10] in particular were among the first to use the meet in the middle strategy [AS09] and achieved more efficient ones on 42-step SHA-256. Khovratovich et al. in 2012 [KRS12] have so far presented the best preimage attacks, on 45-round and 52-round SHA-256 as well as a 52-round attack on the SHA-256 compression function.

Li et al. have published in 2012 [LIS12] a noteworthy paper on converting meet in the middle preimage attack into pseudo collision attack. Using preimage attacks by bicliques, they found pseudo collisions attacks on 52 steps of SHA-256.

Claim 1. SHA256 is 128-bit collision resistant.

3.2 Instantiating MKHASH

In this section we describe the instantiation of MKHASH with a compression function based on MIMC [AGR+16]. We firstly show how the compression function is constructed, and prove that this instantiation complies with the security requirements mentioned in Section 2.7

3.2.1 MIMC Encryption

MIMC is a block cipher with a simple design, consisting of a number of rounds (denoted *rounds*). During the i-th round, the message m is mixed with the encryption key k and a randomly chosen constant $c[i]$, and a permutation function is applied to generate a new value of m. The permutation function consists of exponentiation with a carefully chosen exponent e (see Section 3.2.1). Note that *rounds* depends on the desired security level λ. We denote the encryption function by MIMC-Encrypt and illustrate it in Fig. 3.1.

$\underline{\mathsf{MIMC\text{-}Encrypt}(k, m, c, e, rounds)}$

foreach $i \in [rounds]$:

  $m \leftarrow (k \ \mathsf{OP} \ c[i] \ \mathsf{OP} \ m)^e$

return $(m \ \mathsf{OP} \ k)$

Figure 3.1: MIMC Encryption function.

MIMC-Encrypt can be defined on both binary and prime fields, and as such the OP operation corresponds to either $\oplus$ or $+ \pmod p$ [AGR+16, GRR+16]. For general prime p (resp. positive integer n), we denote by MIMCp (resp. MIMC$_{2^n}$) the MIMC-Encrypt function defined over $\mathbb{F}_p$ (resp. $\mathbb{F}_{2^n}$). In this document, we only consider MIMC defined over prime fields (in particular, the field $\mathbb{F}_{\mathtt{r}}$ with elements written over λ bits, over which ZkSnarkSch operates).

Since block ciphers are usually defined over the product space of keys and messages, we consider the variables c, *rounds* and e as fixed. We thereby consider an instantiation of MIMC with signature

$$\mathsf{MIMC}\mathtt{r} : \mathbb{F}_{\mathtt{r}} \times \mathbb{F}_{\mathtt{r}} \rightarrow \mathbb{F}_{\mathtt{r}}$$

Security parameters and analysis

To ensure that the exponentiation leads to a permutation in $\mathbb{F}_{\mathtt{r}}$, we consider e of the form $e = 2^t - 1$ and $e = 2^t + 1$ such that $\gcd(e, \mathtt{r} - 1) = 1$. To achieve a security of λ, we require that $rounds = \left\lceil \frac{\log_2 \mathtt{r}}{\log_2 e} \right\rceil$.

We refer to the MIMC paper [AGR+16, Section 4.2 and 5.1] for more details on the security analysis and attacks on the scheme. Note that MIMC$\mathtt{r}$ does not suffer from *inversion subfield attacks* as there are no proper subfields of $\mathbb{F}_{\mathtt{r}}$.

3.2.2 MIMC-based compression function

There exist two main techniques to construct a hash function from a block-cipher (or permutation): sponge functions [BDPVA07] and iterated compression functions [BRS02].

A Merkle tree is a binary tree of values of fixed size, where the values in each "layer" are generated by hashing pairs of values from the previous "layer". That is, we require a compression function MKHASH, which we construct via the Miyaguchi-Preneel scheme. (Miyaguchi-Preneel is more secure [BRS02, f_5 function] than the more flexible Davies-Meyer construct [GFBR06, Section 3], but this flexibility is not required in our case).

Miyaguchi-Preneel compression construct

Miyaguchi-Preneel (MP) [BRS02, f_3 function] is a general scheme for constructing compression functions from block ciphers (see Section 1.5.6). Given a block cipher E, the corresponding compression function by $\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}$ is given in Fig. 3.2. The original construction is defined over binary fields, however `Zeth` operates over prime fields. Hence, in the general discussion here we replace the bitwise addition operator $\oplus$ by modular addition in $\mathbb{F}_{\mathtt{r}}$ (see [Har19]).

We denote by MIMC-MP the compression function defined by the application of the Miyaguchi-Preneel construct over MIMC. Similarly, for general prime p we denote by $\mathsf{MIMC\text{-}MP}_p$ (see Fig. 3.3) the compression function defined by application of the Miyaguchi-Preneel construct over $\mathsf{MIMC}p$.

$\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}(k, m)$

$res \leftarrow \mathsf{E}_k(m)$

$\textbf{return } (res + m + k) \pmod{\mathtt{r}}$

Figure 3.2: MP construct in $\mathbb{F}_{\mathtt{r}}$.

$\mathsf{MIMC\text{-}MP}_{\mathtt{r}}(k, m)$

$res \leftarrow \mathsf{MIMC}\mathtt{r}(k, m)$

$\textbf{return } (res + k + m) \pmod{\mathtt{r}}$

Figure 3.3: $\mathsf{MIMC\text{-}MP}_{\mathtt{r}}$ construction.

3.2.3 An efficient instantiation of MIMC primitives

To select appropriate instances of $\mathsf{MIMC}\mathtt{r}$ and $\mathsf{MIMC\text{-}MP}_{\mathtt{r}}$, we consider the cost (in terms of gas consumption and prover efficiency). For given e and *rounds*, the final definition of $\mathsf{MIMC\text{-}MP}_{\mathtt{r}}$ is given in Fig. 3.4 and Fig. 3.5.

$\mathsf{MIMC}\mathtt{r}(k, m)$

$c \leftarrow \mathsf{InitRoundConstants}()$

$\textbf{foreach } i \in [rounds]:$

$\quad m \leftarrow (k + c[i] + m)^e \pmod{\mathtt{r}}$

$\textbf{return } (m + k) \pmod{\mathtt{r}}$

$\mathsf{InitRoundConstants}()$

$iv \leftarrow \mathsf{Keccak256}(\text{“clearmatics\_mt\_seed”})$

$c[0] \leftarrow 0$

$c[1] \leftarrow \mathsf{Keccak256}(iv)$

$\textbf{foreach } i \in \{2, \ldots, rounds\}:$

$\quad c[i] \leftarrow \mathsf{Keccak256}(c[i-1])$

$\textbf{return } c = (c[0], \ldots, c[rounds - 1])$

Figure 3.4: $\mathsf{MIMC}\mathtt{r}$ full construction

$\mathsf{MIMC\text{-}MP}_{\mathtt{r}}(k, m)$

$\textbf{return } \mathsf{MIMC}\mathtt{r}(k, m) + m + k \pmod{\mathtt{r}}$

Figure 3.5: $\mathsf{MIMC\text{-}MP}_{\mathtt{r}}$ full construction

Remark 3.2.1. Note that Keccak256 is the 256-bit digest instance of the Keccak family that won the NIST SHA-3 competition [GJMG11]. It is supported by the EVMvia an opcode (see [W$^+$, Appendix G]), making it convenient for use in smart contracts.

Remark 3.2.2. To increase the security of the MKHASH, different round constants for each level of the Merkle tree could be used.

We define MKHASH to be MIMC-MP over $\mathbb{F}_{\mathtt{r}}$. Thereby, for input values m_0 and m_1, MKHASH : $\mathbb{F}_{\mathtt{r}} \times \mathbb{F}_{\mathtt{r}} \to \mathbb{F}_{\mathtt{r}}$ is defined by

$$\mathsf{MKHASH}(m_0, m_1) = \mathsf{MIMC\text{-}MP}_{\mathtt{r}}(m_0, m_1) \tag{3.1}$$

For specific values of $\mathtt{r}$ (such as $\mathtt{r}_{\mathtt{BN}}$ for BN-254 or $\mathtt{r}_{\mathtt{BLS}}$ for BLS12-377), it remains to choose concrete values of e and *rounds*.

Note that small exponents e result in fewer constraints in the arithmetic circuit (see Section 2.2), while larger exponents can reduce the cost of Merkle tree operations on the contract (see Section 2.5). This is due to two factors, namely that exponentiation is cheaper to execute on a contract than in an arithmetic circuit, and that the number of rounds decreases with higher e. For instance, choosing $e = 7$ results in 365 constraints and $\approx 20k$ gas while $e = 31$ corresponds to 417 constraints (+15%) and $\approx 17k$ (-10%) in gas consumption. Repeating the same process for different exponents, we observe roughly the same order of magnitude gain on the gas consumption and loss on the number of constraints.

The number of constraints of MIMC-MP for several exponents e is given by the formula

$$\mathit{constraints} = \mathit{rounds} \cdot \mathit{mults} + 1$$

where $\mathit{rounds} = \lceil \frac{\log_2 \mathtt{r}}{\log_2 e} \rceil$, *mults* is the number of multiplications required for exponentiation and the additional constraint (corresponding to +1 in the above formula) is a result of the final message and key addition. Note that for $e = 2^t - 1$ we have $\mathit{mults} = 2 \cdot t - 2$, using the *square-and-multiply* algorithm [MVOV96], and for $e = 2^t + 1$ we have $\mathit{mults} = t + 1$.

For several concrete values of e, the number of *rounds* required to attain the desired security level, along with the number of constraints, are shown in Table 3.1.

For the case of BN-254 we set $e = 7$ and $\mathit{rounds} = 91$, targetting a 254-bit security level. For BLS12-377 we set $e = 31$ and $\mathit{rounds} = 51$, targetting a 253-bit security level. These values are chosen such that they satisfy the requirement that $\gcd(e, \mathtt{r} - 1) = 1$ and give a good balance between the number of constraints in the arithmetic circuit and the gas cost of hashing on the contract.

3.2.4 Security requirements satisfaction

After presenting the state of the art of MiMC cryptanalysis, we present the security proof of MIMC-MP collision resistance.

Cryptanalysis of MIMC block cipher and primitives

MIMC's security is increasingly being analysed since the primitive has gained traction in zero-knowledge and cryptocurrency communities for its succinct algebraic constraint representation. As of today, we do not know of any attacks breaking MIMC on prime fields on full rounds.

e	BN-254		BLS12-377	
	rounds	*constraints*	*rounds*	*constraints*
5	110	331		
7	91	365		
17	65	316	62	311
31	52	417	51	409
127	37	445	37	445
257	32	289	32 289	
511	29	465		
2047	24	481	23	461
8191	20	481	20	481
32676	17	477		
65537	16	273	16	273
131071	15	481	15	481
524287	14	505	14	505
1048577	13	274	13	274
2097151	13	521		

Table 3.1: Arithmetic constraints required to represent MIMC-MP as an R1CS program, for different exponents e and curves. Missing entries where $\gcd(e, \mathtt{r} - 1) \neq 1$

The first attack on MIMC was an interpolation attack [LP19] which targets a reduced-round version for a scenario in which the attacker has only limited memory. An attack on Feistel-based MIMC [Bon19] was discovered shortly after, by using generic properties of the used Feistel construction (instead of exploiting properties of the primitive itself). Additionally, [ACG+19] proposes an attack based on Gröbner basis. The authors state that by introducing a new intermediate variable in each round, the resulting multivariate system of equations is a Gröbner basis. As such, the first step of a Gröbner basis attack can be obtained for free. However, the following steps of the attack are so computationally demanding that the attack becomes infeasible in practice. A recent work [EGL+20] targets MIMC on binary fields, and achieves a full-round break of the scheme. While, the attack presented does not apply to prime fields, the authors note that it "can be generalized to include ciphers over $\mathbb{F}_p$", and that only the lack of efficient distinguishers over prime fields precludes this. Another attack from Beyne et al [BCD+20] uses a low complexity distinguisher against full MIMC permutation leading to a practical collision attack on reduced round sponge-based MIMC hash defined with security of 128 bits.

Security proof of MIMC-MP collision resistance

We now prove that this compression scheme satisfies all the security requirements listed in Section 2.7. To do so, we first assume that the round constants are pseudo-random, i.e. that Keccak256 is a PRF.

Lemma 3.2.1. Keccak256 *is a* PRF *with* $\lambda = 128$.

The security of MIMC-MP derives from a more general result, i.e. from modelling MIMC as an ideal cipher (see Definition 1.5.12). More specifically, we show a security result for the MP construction on $\mathbb{F}_{\mathtt{r}}$ by proving that, in the Ideal Cipher Model, the collision resistance advantage of any adversary is bounded by $\frac{q(q+1)}{\mathtt{r}}$, where q is the number of different queries that the attacker makes to the oracle. This means that, assuming a maximum q number of possible encryption/decryption queries, parameter $\mathtt{r}$ can be chosen to make the advantage small as needed and $\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}$ considered collision resistant. Similar result applies to the 2^n case.

The instance of MIMC we use is modelled as an ideal cipher defined on field elements, for this reason we consider a variant of the ICM model where the keys, inputs and outputs are field elements in $\mathbb{F}_{\mathtt{r}}$ and the block cipher scheme, with key k, correspond to a family of $\mathtt{r}$ independent random permutations $f_k : \mathbb{F}_{\mathtt{r}} \times \mathbb{F}_{\mathtt{r}} \to \mathbb{F}_{\mathtt{r}}$.

In the proof, without loss of generality, we assume the following conventions for an adversary $\mathcal{A}$:

- the adversary asks distinct queries: i.e. if $\mathcal{A}$ asks a query $\mathsf{O}^{\mathsf{E}}(k, m)$ and this returns y, then $\mathcal{A}$ does not ask a subsequent query of $\mathsf{O}^{\mathsf{E}}(k, m)$ or $\mathsf{O}^{\mathsf{E}^{-1}}(k, y)$, and inversely;

- the adversary necessarily obtained the candidate collision from the oracle. This property follows suite from modelling MIMC as an ideal cipher.

Lemma 3.2.2. *Let* $\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}$ *be the* MP *compression function built on an ideal block-cipher* E *on* $\mathbb{F}_{\mathtt{r}}$*, the probability for an adversary* $\mathcal{A}$ *to find a collision is not greater than* $q(q+1)/\mathtt{r}$ *where* q *is a (positive) number of distinct oracle queries.*

The following proof has been adapted from [BRS02, Lemma 3.3][2].

Proof. Fix $h_0 \in \mathbb{F}_{\mathtt{r}}$. Let $\mathcal{A}$ be an adversary attacking the compression function $\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}$. Assume that $\mathcal{A}$ asks the oracles O^{E} and $\mathsf{O}^{\mathsf{E}^{-1}}$ a total of *distinct* q queries. Let us denote the result of the q queries and output of the attacker (candidate collision) as $((k_1, m_1, y_1), \ldots, (k_q, m_q, y_q), \text{out})$. If $\mathcal{A}$ is successful it means that it outputs (k, m), (k', m') such that either $(k, m) \neq (k', m')$ and $\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}(k, m) = \mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}(k', m')$ or $\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}(k, m) = h_0$. By the definition of $\mathsf{f}_{\mathsf{E}}^{\mathsf{MP}}$, we have that $\mathsf{E}_k(m) + m + k = \mathsf{E}_{k'}(m') + m' + k'$ for the first case, or $\mathsf{E}_k(m) + m + k = h_0$ for the second. So either there are distinct $r, s \in [1, \ldots, q]$ such that $(k_r, m_r, y_r) = (k, m, \mathsf{E}_k(m))$ and $(k_s, m_s, y_s) = (k', m', \mathsf{E}_{k'}(m'))$ and $\mathsf{E}_{k_r}(m_r) + m_r + k_r = \mathsf{E}_{k_s}(m_s) + m_s + k_s$ or else there is an $r \in [1, \ldots, q]$ s.t. $(k_r, m_r, y_r) = (k, m, h_0)$ and $\mathsf{E}_{k_r}(m_r) + m_r + k_r = h_0$. We show that this event is unlikely.

In fact, for each $i \in [1, \ldots, q]$, let C_i be the event that either $y_i + m_i + k_i = h_0$ or does exist $j \in [1, \ldots, i-1]$ s.t. $y_i + m_i + k_i = y_j + m_j + k_j$. When carrying out the simulation y_i or m_i was randomly selected from a set of at least $\mathtt{r} - (i-1)$ elements, so

[2] It states the collision resistance of a set of compression functions $f_1, , \ldots, , f_{12}$, denoted as *group-1 compression functions* and showed in [BRS02, Figure 3]. As mentioned above, Miyaguchi-Preneel corresponds to f_3 of that group. Since the proof of [BRS02, Lemma 3.3] shows collision resistance of f_1, we slightly modified it to work for f_3.

$\Pr[C_i] \leq i/(\mathtt{r} - i)$. This means that for the collision advantage of $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{coll}}_{\mathsf{f}^{\mathsf{MP}}_{\mathsf{E}},\mathcal{A}}$ it holds that $\mathsf{Adv}^{\mathsf{coll}}_{\mathsf{f}^{\mathsf{MP}}_{\mathsf{E}},\mathcal{A}} \leq \Pr[C_1 \vee \cdots \vee C_q] \leq \sum_{i=1}^{q} \Pr[C_i]$. For $q \leq \frac{\mathtt{r}}{2}$ this probability is bounded by $l \cdot \frac{q(q+1)}{\mathtt{r}}$. However, we allow only a polynomial number of queries, thus for $q = \mathsf{poly}(\lambda)$ this probability becomes $\frac{\mathsf{poly}(\lambda)}{\mathtt{r}}$, where $\mathtt{r} \approx 2^{\lambda}$. □

Note

Lemma 3.2.2 is applicable to our case by the strong assumption of $\mathsf{MIMC}\mathtt{r}$ being an ideal cipher. In other words, the proof does not take into account any structural weakness or knowledge that an attacker is aware of. Any such additional information could make Lemma 3.2.2 invalid, and consequently could be used to break the collision resistance.

Remark 3.2.3. Note that from Lemma 3.2.2 follows that the collision resistance security of the `Zeth` Merkle tree is $\log_2(\mathtt{r}/2)$ (around 127 bits for $\mathtt{r} = \mathtt{r}_{\mathtt{BN}}$ or $\mathtt{r}_{\mathtt{BLS}}$).

Note

MIMC has *not* received as much cryptanalytic scrutiny as other "older" and more established hash functions. This is important to note since, for these type of primitives which are not provably secure, the amount of attacks received by a scheme is a great indicator of its security and robustness. A natural alternative to MIMC here consists in using Pedersen hash which is provably collision resistant under the discrete-logarithm assumption.

3.3 `Zeth` statement after primitive instantiation

After instantiating the various primitives and providing security proofs to justify that they comply with the security requirements listed in previous sections, $\mathbf{R}^{\mathsf{z}}$ now becomes:

- For each $i \in [\mathtt{JSIN}]$:
    1. $aux.jsins[i].znote.apk = \mathsf{Blake2s}(tag^{addr}_{ask} \| \mathsf{pad}_{\mathtt{BLAKE2sCLEN}}(0))$
       with tag^{addr}_{ask} defined in Section 3.1.3
    2. $aux.jsins[i].nf = \mathsf{Blake2s}(tag^{nf}_{ask} \| aux.jsins[i].znote.\rho)$
       with tag^{nf}_{ask} defined in Section 3.1.3
    3. $aux.jsins[i].cm = \mathsf{Blake2s}(aux.jsins[i].znote.r \| m)$
       with $m = aux.jsins[i].znote.apk \| aux.jsins[i].znote.\rho \| aux.jsins[i].znote.v$
    4. $aux.htags[i] = \mathsf{Blake2s}(tag^{pk}_{ask,i} \| prim.hsig)$ (malleability fix, see Appendix A)
       with $tag^{pk}_{ask,i}$ defined in Section 3.1.3
    5. $(aux.jsins[i].znote.v) \cdot (1 - e) = 0$ is satisfied for the boolean value e set such that if $aux.jsins[i].znote.v > 0$ then $e = 1$.

6. The Merkle root $mkroot'$ used to check the Merkle authentication path $aux.jsins[i].mkpath$ of commitment $aux.jsins[i].cm$, with $\mathsf{MIMC\text{-}MP}_{\mathtt{r}}$, equals $prim.mkroot$ if $e = 1$.
7. $prim.nfs[i]$
   $= \{\mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.jsins[i].nf[k \cdot \mathtt{FIELDCAP}{:}(k+1) \cdot \mathtt{FIELDCAP}])\}_{k \in [\lfloor \mathtt{PRFNFOUTLEN}/\mathtt{FIELDCAP} \rfloor]}$
8. $prim.htags[i]$
   $= \{\mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.htags[i][k \cdot \mathtt{FIELDCAP}{:}(k+1) \cdot \mathtt{FIELDCAP}])\}_{k \in [\lfloor \mathtt{PRFPKOUTLEN}/\mathtt{FIELDCAP} \rfloor]}$

- For each $j \in [\mathtt{JSOUT}]$:

  1. $aux.znotes[j].\rho = \mathsf{Blake2s}(tag^{\rho}_{ask,j} \| prim.hsig)$ (malleability fix, see Appendix A) with $tag^{\rho}_{ask,j}$ defined in Section 3.1.3
  2. $prim.cms[j] = \mathsf{Blake2s}(aux.znotes[j].r \| m)$ with $m = aux.znotes[j].apk \| aux.znotes[j].\rho \| aux.znotes[j].v$

- $prim.hsig = \{\mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.hsig[k \cdot \mathtt{FIELDCAP}{:}(k+1) \cdot \mathtt{FIELDCAP}])\}_{k \in [\lfloor \mathtt{CRHHSIGOUTLEN}/\mathtt{FIELDCAP} \rfloor]}$

- $prim.rsd = \mathsf{Pack}_{rsd}(\{aux.jsins[i].nf\}_{i \in [\mathtt{JSIN}]}, aux.vin, aux.vout, aux.hsig, \{aux.htags[i]\}_{i \in [\mathtt{JSIN}]})$

- Check that the "joinsplit is balanced", i.e. check that the joinsplit equation holds:

$$\begin{aligned} &\mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.vin) + \sum_{i \in [\mathtt{JSIN}]} \mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.jsins[i].znote.v) \\ &= \sum_{j \in [\mathtt{JSOUT}]} \mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.znotes[j].v) + \mathsf{Pack}_{\mathbb{F}_{\mathtt{r}}}(aux.vout) \end{aligned}$$

Remark 3.3.1. For higher security, we could use Blake2b with 32-byte output instead of SHA256. In fact, since a precompiled contract computing the Blake2 compression function [MJS15] has been added to the Istanbul release of `Ethereum` (EIP 152 [THH15]), it could be possible to write a small wrapper on the smart contracts, in order to hash with Blake2b with any parameter.

3.3.1 Instantiating the packing functions

As we consider SNARKs based on arithmetic circuits defined over a prime field, all variables in the constraint system are interpreted as field elements. Nevertheless, as illustrated in Section 2.2, part of the statement consists of functions whose co-domains are sets of binary strings (which may be longer than the bit representation of elements of the finite field). While a bit (i.e. $\{0, 1\}$) is an element of $\mathbb{F}_p$ (p prime), it is important to minimize the number of gates in the arithmetic circuit (for proof generation efficiency), and to minimize the number of input wires (to improve verification time). This can be done by representing fragments of binary strings as the base 2 decomposition of field elements, thereby "packing" binary strings into multiple elements. Converting binary strings into field elements requires the addition of some arithmetic gates (extending the statement to be proven), but reduces the number of primary inputs (reducing the complexity of the SNARK verification carried out on-chain). The cost of $\mathsf{Groth16}$ zk-SNARK [Gro16] proof verification is linear in the number of primary inputs, since each input acts as a scalar in a costly scalar multiplication of a curve point in $\mathbb{G}_1$. Hence, while packing slightly increases the prover cost – by adding constraints to the circuit – it simplifies the verifier's work.

In this section, we detail the method by which we encode (resp. decode) a set of binary strings to (resp. from) sets of field elements. In the rest of this section, the notion of *packing policy* refers to the set of *packing* and *unpacking* functions.

The set of primary inputs is composed of the input nullifiers, the output commitments, the public values (see [RZ19, Section 3.4.3]) along with the signature hash and the authentication tags for security (malleability fix, see Appendix A). The complete description of the public inputs is represented in Eq. (3.2).

$$(\{prim.nf_i\}_{i\in[\texttt{JSIN}]}, \{prim.cms[j]\}_{j\in[\texttt{JSOUT}]}, vin, vout, hsig, \{prim.htags[i]\}_{i\in[\texttt{JSIN}]}) \quad (3.2)$$

The primary inputs that consist of binary strings are: the nullifiers *nfs*, the public values *vin* and *vout*, the signature hash *hsig* and the authentication tags *htags*.

For a binary string x, let $\alpha_x = \lceil \mathsf{length}(x)/\texttt{FIELDCAP} \rceil$ be the number of field elements required to completely encode x and let $\beta_x = \lfloor \mathsf{length}(x)/\texttt{FIELDCAP} \rfloor$ be the number of field elements whose capacity is fully used. Let $\gamma_x = \mathsf{length}(x) \pmod{\texttt{FIELDCAP}}$ be the number of "residual" bits remaining after fully using β_x field elements.

Example 3.3.2. Consider binary strings $A \in \{0,1\}^7$ of length 7, to be encoded over the field $\mathbb{F}_{41}$. This field has a capacity of 5 bits, and therefore $\alpha_A = 2$, $\beta_A = 1$, and $\gamma_A = 2$. That is, A can be represented as 2 field elements, or as 1 field element with 2 "residual" bits.

Consider $A = (1111011)$. Fig. 3.6 illustrates how A can be packed as field elements. Note that the 2 residual bits are taken from the "beginning" of the bit string, that is, the highest order bits.

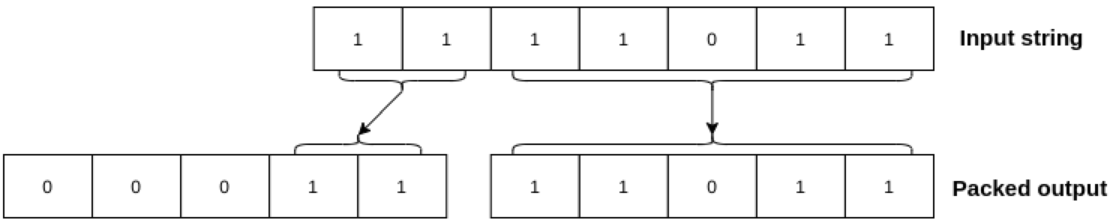

Figure 3.6: Packing of string A (see Example 3.3.2)

We now consider strategies to pack all primary inputs that are binary strings. A naive approach is to encode each binary string x as α_x field elements. In general, this results in significant waste (and consequently more field elements than necessary), especially when the number of residual bits is small compared to $\texttt{FIELDCAP}$ (see Fig. 3.7). An alternative strategy could be to concatenate all binary strings into a single string y and pack this string into α_y field elements. While this approach minimizes the set of unused bits, each unpack operation would require different shift and mask operations over 2 or 3 field elements. This significantly increases the complexity of the unpacking operation that must be performed on-chain, resulting in a higher gas cost (due to extra logic) or more contract code (if each unpack operation is hard-coded).

The `Zeth` protocol requires that each binary string variable x is packed into β_x field elements, and the residual bits from all binary strings, along with the public values *vin* and *vout*, are aggregated into a variable *rsd*. Let `RSDBLEN` be the total number of residual bits, and `RSDFLEN` be the number of field elements required to represent *rsd*. We assume that `ZVALUELEN` < `FIELDCAP`, and define the notation γ_v = `ZVALUELEN` for the bit lengths of public values *vin* and *vout*. Thus `RSDBLEN` is given by

$$\texttt{RSDBLEN} = \gamma_{hsig} + 2 \cdot \gamma_v + \texttt{JSIN} \cdot (\gamma_{nf} + \gamma_h)$$

and the lengths, in field elements, of each of the correponding public inputs are

$$\begin{aligned}
\texttt{NFFLEN} &= \beta_{nf} \\
\texttt{HSIGFLEN} &= \beta_{hsig} \\
\texttt{HFLEN} &= \beta_h \\
\texttt{RSDFLEN} &= \lceil \texttt{RSDBLEN}/\texttt{FIELDCAP} \rceil
\end{aligned}$$

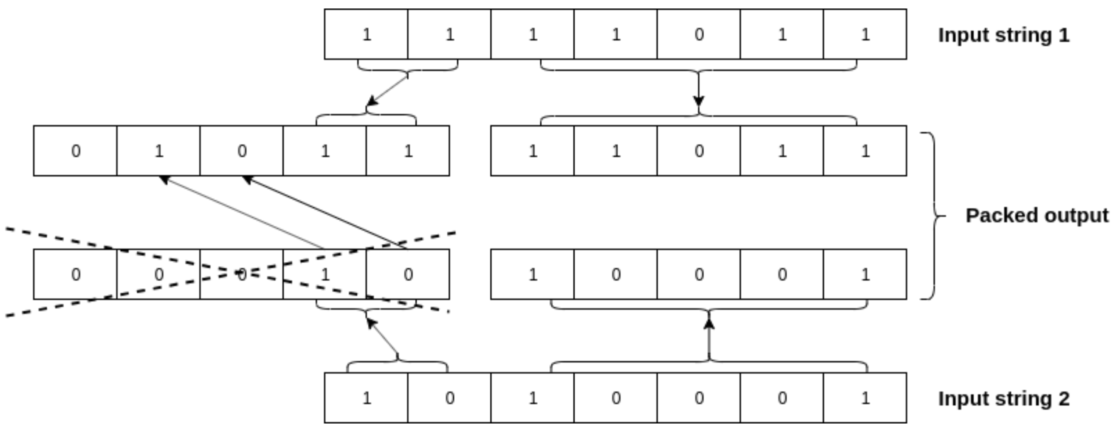

Figure 3.7: Packing of multiple strings. Observe that, by carefully arranging the bits of the input strings, it is possible to output fewer field elements

The residual bits *rsd* are formatted as follows:

$$\widetilde{hsig} \| \widetilde{nfs} \| \widetilde{htags} \| vin \| vout$$

where $\widetilde{hsig}, \widetilde{nfs}, \widetilde{htags}$ are, respectively, the $\gamma_{hsig}, \gamma_{nf}, \gamma_h$ bits.

Note that the public values are packed into the "last", or lowest order, $2 \cdot \gamma_v$ bits of the resulting field element(s). In this way, their unpack functions are independent of the

$\mathsf{Pack}_{\mathbb{F}_\mathtt{r}}(x)$

$out \leftarrow 0_{\mathbb{F}_\mathtt{r}}$;
for $i \in [\mathsf{length}(x)]$ **do** :
  if $x[i] = 1$ **do** :
    $out \leftarrow out +_{\mathbb{F}_\mathtt{r}} 2^{\mathsf{length}(x)-1-i}$
return out;

Figure 3.8: Algorithm to pack bits into a field element.

$\mathsf{Pack}_{rsd}(nfs, vin, vout, hsig, htags)$

$out \leftarrow []; r \leftarrow \epsilon$;
$r \leftarrow vout$;
$r \leftarrow vin \| r$;
for $i \in [\mathtt{JSIN}]$ **do** :
  $r \leftarrow htags[i][\beta_{htags[i]} \cdot \mathtt{FIELDCAP}{:}] \| r$;
for $i \in [\mathtt{JSIN}]$ **do** :
  $r \leftarrow nfs[i][\beta_{nfs[i]} \cdot \mathtt{FIELDCAP}{:}] \| r$;
$r \leftarrow hsig[\beta_{hsig} \cdot \mathtt{FIELDCAP}{:}] \| r$;
for $i \in [\lceil \mathsf{length}(r)/\mathtt{FIELDCAP} \rceil]$ **do** :
  $out[i] \leftarrow \mathsf{Pack}_{\mathbb{F}_{\mathtt{r}_\mathtt{BN}}}(r[i \cdot \mathtt{FIELDCAP}{:}(i+1) \cdot \mathtt{FIELDCAP}])$;
return out;

Figure 3.9: Algorithm to pack residual bits.

values JSIN and JSOUT and of the number of residual bits required for each bit string (and consequently, independent of the finite field used).

To format the unpacked primary inputs into field elements, we define the following functions. Given a bit string of length less than FIELDCAP, the algorithm Pack (see Fig. 3.8) returns a field element. Given the nullifiers, public values and authentication tags, the algorithm Pack_{rsd} (see Fig. 3.9) outputs the residual bits. Given a set of packed field elements and the residual bits, the algorithm Unpack returns the variables reassembled as binary strings. In particular, we have that $\mathsf{Unpack}_{nf}(prim.nfs, rsd) = \{aux.jsins[i].nf\}_{i \in [\mathtt{JSIN}]}$.

$\mathsf{Pack} : \mathbb{B}^{\leq \mathtt{FIELDCAP}} \rightarrow \mathbb{F}_\mathtt{r}$
$\mathsf{Pack}_{rsd} : (\mathbb{B}^{\mathtt{PRFNFOUTLEN}})^{\mathtt{JSIN}} \times (\mathbb{B}^{\mathtt{ZVALUELEN}})^2 \times \mathbb{B}^{\mathtt{CRHHSIGOUTLEN}} \times (\mathbb{B}^{\mathtt{PRFPKOUTLEN}})^{\mathtt{JSIN}} \rightarrow (\mathbb{F}_\mathtt{r})^{\mathtt{RSDFLEN}}$
$\mathsf{Unpack} : \mathbb{F}_\mathtt{r}^* \times (\mathbb{F}_\mathtt{r})^{\mathtt{RSDFLEN}} \rightarrow \mathbb{B}^*$

The Unpack functions for nullifiers, public values and signature hash are represented as follows.

$$\mathsf{Unpack}_{hsig} : (\mathbb{F}_\mathtt{r})^{\mathtt{HSIGFLEN}} \times (\mathbb{F}_\mathtt{r})^{\mathtt{RSDFLEN}} \rightarrow \mathbb{B}^{\mathtt{CRHHSIGOUTLEN}}$$
$$\mathsf{Unpack}_{nf} : (\mathbb{F}_\mathtt{r})^{\mathtt{NFFLEN}} \times (\mathbb{F}_\mathtt{r})^{\mathtt{RSDFLEN}} \rightarrow \mathbb{B}^{\mathtt{PRFNFOUTLEN}}$$
$$\mathsf{Unpack}_{vin} : \mathbb{F}_\mathtt{r}^0 \times (\mathbb{F}_\mathtt{r})^{\mathtt{RSDFLEN}} \rightarrow \mathbb{B}^{\mathtt{ZVALUELEN}}$$
$$\mathsf{Unpack}_{vout} : \mathbb{F}_\mathtt{r}^0 \times (\mathbb{F}_\mathtt{r})^{\mathtt{RSDFLEN}} \rightarrow \mathbb{B}^{\mathtt{ZVALUELEN}}$$

Packing Policy Security

Proposition 3.3.1 (Packing security). *For a binary string x, it holds that* $\mathsf{Unpack}(\mathsf{Pack}(x)) = x$ *and* $\mathsf{Unpack}(\mathsf{Pack}_{rsd}(x)) = x$.

Packing Policy Example

In the case where $\texttt{JSIN} = \texttt{JSOUT} = 2$, the $\mathsf{BN\text{-}254}$ is being used (in which field elements hold $\texttt{FIELDCAP}_{\texttt{BN}}$ bits) and all PRFs and $\mathsf{CRH}^{\mathsf{hsig}}$ output bit-strings of length 256, the unpacked primary inputs are 2167-bit long. The packing parameters are therefore:

$$\begin{aligned}\texttt{RSDBLEN} &= 5 \times 3 + 64 + 64 = 143\\ \texttt{NFFLEN} &= \texttt{HSIGFLEN} = \texttt{HFLEN} = \texttt{RSDFLEN} = 1\end{aligned}$$

The packed primary inputs are 2277 bits long, corresponding to a small space overhead of $\approx 5\%$ unused bits. Moreover, the 143-bit residual bits can be packed into a single field element. As such, the primary inputs are encoded as 9 field elements. Finally, the residual bits are formatted as follows,

$$\underbrace{padding}_{113\text{ bits}} \| \underbrace{hsig}_{3\ bits} \| \underbrace{nf_1}_{3\ bits} \| \underbrace{nf_0}_{3\ bits} \| \underbrace{h_1}_{3\ bits} \| \underbrace{h_0}_{3\ bits} \| \underbrace{vin}_{64\ bits} \| \underbrace{vout}_{64\ bits}$$

For the analogous case using $\mathsf{BLS12\text{-}377}$ (in which field elements hold $\texttt{FIELDCAP}_{\texttt{BLS}}$ bits), the packing parameters are:

$$\begin{aligned}\texttt{RSDBLEN} &= 5 \times 4 + 64 + 64 = 148\\ \texttt{NFFLEN} &= \texttt{HSIGFLEN} = \texttt{HFLEN} = \texttt{RSDFLEN} = 1\end{aligned}$$

The residual bits can be packed into a single field element of the form

$$\underbrace{padding}_{108\text{ bits}} \| \underbrace{hsig}_{4\ bits} \| \underbrace{nf_0}_{4\ bits} \| \underbrace{nf_1}_{4\ bits} \| \underbrace{h_0}_{4\ bits} \| \underbrace{h_1}_{4\ bits} \| \underbrace{vin}_{64\ bits} \| \underbrace{vout}_{64\ bits}$$

and the primary inputs are again encoded as 9 field elements.

3.4 Instantiate $\mathsf{SigSch}_{\mathsf{OT\text{-}SIG}}$

`Zeth` uses the one-time Schnorr-based signature scheme introduced by Bellare and Shoup [BS07] for its long proven security, simplicity, speed and size. Its security relies on the one-more discrete log problem (see Definition 1.5.6) and the collision resistance of the underlying hash function CRH (see Definition 1.5.16) that we instantiate with $\mathsf{SHA256}$.

Note that no signature operations or data are used in the arithmetic circuit describing the `Zeth` statement. Hence the curve used for the signature scheme can be chosen independently of Curve (the scalar field of which is used for the arithmetic circuit, and consequently for commitments and bit string encodings described in Section 3.1 and

Section 3.2). BN-254 is used since it is supported by the EVM, in the form of precompiled contracts. This allows a gas-efficient implementation in the **Mixer** contract.

This one-time signature scheme (see Definition 1.5.26) is defined by the two-tier signature scheme over a cyclic group $(p, \mathbb{G}, \langle \mathfrak{g} \rangle, \otimes)$. In the two-tier signature scheme, the hash function CRH only needs to be collision resistant (the random oracle model is not used). Similarly, the variable hk represents the key of the hash function (a particular instance).

To turn this two-tier signature scheme into a one-time signature scheme, one simply has to define the one-time signature key generation KGen as the combination of both primary and secondary key generations of the two-tier (see [BS07, Section 6]). The one-time signing key (respectively verification key) of the one time signature scheme is defined as both the primary and secondary signing key (respectively verification key) of the two-tier scheme, Fig. 3.10

$\underline{\mathsf{KGen}(1^\lambda):}$	$\underline{\mathsf{Sig}(sk, m):}$	$\underline{\mathsf{Vf}(pk, m, \sigma):}$
$hk \leftarrow\$ \mathbb{B}^{\mathsf{kl}}$	$hk, \mathfrak{g}, x = sk.sk1$	$hk, \mathfrak{g}, [\![x]\!] = pk.pk1$
$\mathfrak{g} \leftarrow\$ \mathbb{G}^*$	$y, [\![y]\!] = sk.sk2$	$[\![y]\!] = pk.pk2$
$x \leftarrow\$ \mathbb{F}_p$	$c = \mathsf{CRH}(hk, [\![y]\!] \Vert m)$	$c = \mathsf{CRH}(hk, [\![y]\!] \Vert m)$
$pk1 = (hk, \mathfrak{g}, [\![x]\!])$	$\sigma = y \bmod p$	**if** $\sigma = [\![y]\!] \otimes c \cdot [\![x]\!]$ **then**
$sk1 = (hk, \mathfrak{g}, x)$	$\sigma \mathrel{+}= c \cdot x \bmod p$	**return** 1
$y \leftarrow\$ \mathbb{F}_p$	**return** σ	**else**
$pk2 = [\![y]\!]$		**return** 0
$sk2 = (y, [\![y]\!])$		**endif**
$pk = (pk1, pk2)$		
$sk = (sk1, sk2)$		

Figure 3.10: One-time signature scheme from two tier Schnorr based signature scheme by Bellare and Shoup [BS07]

3.4.1 Security requirements satisfaction

We now prove that this signature scheme satisfies all the security requirements listed in Section 2.7.

Theorem 3.4.1. *The One-Time Schnorr signature is strongly unforgeable under chosen-message attacks* (SUF-CMA) *assuming that the* om-DLog *problem is hard in* $\mathbb{G}$ *and that the hash function* CRH *is collision resistant.*

Proof. See [BS07, Theorems 5.1, 5.2 and 6.1]. ☐

3.4.2 Data types

We now describe the data types and operations associated with this signature scheme.

`VKOtsDType` Denotes the verification key associated with the one-time signature scheme.

Field	Description	Data type
$pk1$	Encoding of the scalar x in the group	$(\mathbb{F}_{\mathtt{r_{BN}}})^2$
$pk2$	Encoding of the scalar y in the group	$(\mathbb{F}_{\mathtt{r_{BN}}})^2$

Table 3.2: `VKOtsDType` data type

`SKOtsDType` Denotes the signing key associated with the one-time signature scheme.

Field	Description	Data type
$sk1$	Scalar element x	$\mathbb{F}_{\mathtt{r_{BN}}}$
$sk21$	Scalar element y	$\mathbb{F}_{\mathtt{r_{BN}}}$
$sk22$	Encoding of the scalar y in the group	$(\mathbb{F}_{\mathtt{r_{BN}}})^2$

Table 3.3: `SKOtsDType` data type

`SigOtsDType` Denotes the signature data type associated with the one-time signature scheme. `SigOtsDType` is an alias for $(\mathbb{F}_{\mathtt{r_{BN}}})^2$.

3.5 Instantiate EncSch

In this section we describe the instantiation of EncSch primitive introduced in Section 2.3. First, we present a general asymmetric encryption scheme called DHAES (Diffie-Hellman Asymmetric Encryption Scheme [ABR99]), which satisfies all the required security properties for the in-band encryption scheme EncSch (see Section 1.5.3). Then, we give details of the concrete algorithms used for the implementation.

3.5.1 DHAES encryption scheme

Given a symmetric encryption scheme Sym, a group defined by SetupG, a family of hash function $\mathcal{H}$[3] and a message authentication scheme MAC as defined in Section 1.5, we define a DHAES scheme as the following public-key encryption scheme:

- Setup, setup algorithm, takes as input a security parameter 1^λ. It runs $\mathcal{H}$.Setup, SetupG and returns public parameters $pp = (hk, (q, \mathbb{G}, \mathfrak{g}, +))$.

[3] Here, we only consider fixed-length hash functions with $hInpLen(\lambda) = 2gLen$ and $hLen(\lambda) = kLen(\lambda) + mLen(\lambda)$ (see Section 1.5).

- KGen, key generation algorithm, takes as input public parameters pp. It samples at random $v \leftarrow\!\!\$\, [q]$ and returns a keypair $(sk, pk) = (v, [\![v]\!])$.

- Enc, encryption algorithm, takes as input public parameters pp, a message m and a public key pk. It runs KGen that returns an ephemeral keypair $(esk, epk) = (u, [\![u]\!])$. Then, it computes a shared secret $ss = \mathsf{H}_{hk}(epk \| esk \cdot pk) = \mathsf{H}_{hk}(epk \| sk \cdot epk)$, parsed as $ek \| mk$[4]. It computes $ct_{\mathsf{Sym}} = \mathsf{Sym.Enc}(ek, m)$ and $\tau = \mathsf{MAC.Tag}(mk, ct_{\mathsf{Sym}})$ and finally outputs the ciphertext $epk \| ct_{\mathsf{Sym}} \| \tau$.

- Dec, decryption algorithm, takes as input public parameters pp, a private key sk a ciphertext $epk \| ct_{\mathsf{Sym}} \| \tau$. It computes $ss = \mathsf{H}_{hk}(epk \| sk \cdot epk)$ and parses it, as above, as $ek \| mk$. If MAC verification passes, i.e. $\mathsf{MAC.Vf}(mk, \tau) = 1$, the algorithm returns $\mathsf{Sym.Dec}(ek, ct_{\mathsf{Sym}})$ and $\perp$ otherwise.

The DHAES definition given above is an asymptotic adaptation of [ABR99, Section 1.3].

Inclusion of ephemeral key in hash input

Given an ephemeral keypair $(u_0, [\![u_0]\!])$, If the group $\langle \mathfrak{g} \rangle$, generated by SetupG, has composite order, then $[\![u_0]\!]$ is required to be part of the hash input because $[\![u_0 v]\!]$ and $[\![v]\!]$ together may not uniquely determine $[\![u_0]\!]$. Equivalently, there may exist two values u_0 and u_1 such that $u_0 \neq u_1$ and $[\![u_0 v]\!] = [\![u_1 v]\!]$. As a result, both u_0 and u_1 can be used to produce two different *valid* ciphertexts of the same plaintext m, under different ephemeral keys $([\![u_0]\!], [\![u_1]\!])$. It is easy to show this, for example, in the multiplicative group $\mathbb{Z}_p \setminus \{0\}$, where p is a prime (see [ABR99, Section 3.1]). A scheme having such malleability property clearly cannot be proven IND-CCA2 secure: an attacker could easily win the related security game by altering the challenged ciphertext and query the decryption oracle that would not recognize that as a not allowed query. If the group has prime order this problem does not arise so only $[\![u_0 v]\!]$ is required as input of the H function [ABR01, Section 3].

3.5.2 A DHAES instance

Curve25519

For a cyclic group we propose the use of a subgroup of Curve25519 described in [Ber06] and in [LHT16]. Curve25519 is a Montgomery elliptic curve [Mon87] defined by the equation $y^2 = x^3 + 486662x^2 + x$ and coordinates on $\mathbb{F}_p$, where p is the prime number $2^{255} - 19$. It has a prime order subgroup of order $2^{252} + 27742317777372353535851937790883648493$ and cofactor 8. Curve25519 comes with an efficient scalar multiplication denoted as X25519[5]. In a Diffie-Hellman-based scheme it allows to have 32-byte long

[4] Note that ek and mk must have the same length.

[5] X25519 is actually introduced in [LHT16] in order to avoid notation issues due to the use Curve25519 to indicate both curve and scalar multiplication as done in [Ber06]

public and private keys (given a point $P = (x, y)$ only the x coordinate is actually used) and the 32-byte sequence representing 9 is specified as base point.

Efficiency and security of Curve25519

High-speed and timing-attack resistant implementations of X25519 are available and its security level is conjectured to be 128 bits [Ber06, Section 1]. However, combined attacks can lead to 124 bits of security (see [BL, Section "Twist Security"]). By design, Curve25519 is resistant to state-of-the-art attacks and satisfies all security criteria and principles listed in *Safecurves* [BL][6].

Interestingly, Curve25519 does not require *public key validation*[7], while we know that, on other curves, active attacks – consisting of sending malformed public keys – could be carried out by adversaries, to violate the confidentiality of private keys, e.g. [ABM+03]. However, Curve25519 specification mandates the *clamping* of private keys: that is, after the random sampling of 32 bytes, the user clears bits 0, 1 and 2 of the first byte, clears bit 7 and sets bit 6 of the last byte. The resulting 32 bytes are then used as private key. This particular structure for private keys prevents various types of attacks (see [Ber06, Section 3] for more details).

> **Note**
>
> Note that the *clamping* procedure is vital to ensure the security guarantees of the Curve25519 specification, and implementations `MUST` perform this exactly as described.

Chacha20

ChaCha20 is an ARX-based[8] stream cipher introduced in [Ber08a]. It is an improved version of Salsa20 [Ber08b] that won the *eSTREAM* challenge [est]. Compared with Salsa20, it has been designed to improve diffusion per round, conjecturally increasing resistance to cryptanalysis, while preserving time efficiency per round. It is considerably faster than AES in software-only implementations and can be easily implemented to be timing-attacks resistant. Several versions of the cipher can be used. The original paper presents ChaCha20 with a 128-bit key and 64-bit nonce/block count. However, the length of the key, nonce and block count – which indicates how many chunks can be processed by using the same key and nonce – can be modified depending on the application. In [LN18][Section 2.3], for instance, the key is a 256-bit string, the nonce is a string of 96 bits and the block count is encoded on a 32-bit word. This configuration allows to

[6] In this work, the authors take into account both Elliptic Curve Discrete Logarithm Problem (ECDLP) and Elliptic Curve Cryptosystems (ECC) security, that allows to have an overall evaluation of the security guarantees.

[7] Informally, it is a set of security checks that a user performs before using a not trusted public key (e.g. see [BCK+18])

[8] Addition-Rotation-XOR

process around 2^{32} blocks, corresponding to roughly 256 GB of data. We propose to use the same parameters in `Zeth`.

$$\mathsf{ChaCha20} : \mathbb{B}^{256} \times \mathbb{B}^{32} \times \mathbb{B}^{96} \times \mathbb{B}^{*} \rightarrow \mathbb{B}^{*}$$

Security of Chacha

Recent cryptanalysis results for ChaCha are available in [AFK+08, Ish12, SZFW12, Mai16, CM16, CM17]: all of them make use of advanced cryptanalysis techniques able to perform key-recovery attacks only on reduced versions (6 and 7 rounds) of ChaCha.

> **Note**
>
> Importantly, the security properties of ChaCha rely on the fact that, for a given key, all blocks are processed with distinct values in the state words 12 to 15 (storing the counter and the nonce) [LN18, Section 2.3].

Poly1305

Poly1305 [Ber05] is a high-speed message authentication code, easy to implement and make side-channel attack resistant. It takes a 32-byte one-time key mk and a message m and produces a 16-byte tag τ that authenticates the message. mk must be unpredictable and it is represented as a couple (r, s), where both components are given as a sequence of 16 bytes each. It can be generated by using pseudorandom algorithms: in [Ber05, Section 2], for example, AES and a nonce are used to generate s. The second part of the key, r, is expected to have a given form [Ber05, Section 2], and must be "clamped" as follows: top four bits of $r[3]$, $r[7]$, $r[11]$, $r[15]$ and bottom two bits of $r[4]$, $r[8]$, $r[12]$ are cleared (see also Section 3.5.3).

> **Note**
>
> Similarly to Curve25519, the *clamping* procedure here is essential to the security of the Poly1305 scheme. Implementations `MUST` ensure that this is performed correctly in order for all security guarantees to hold.

We refer to [LN18, Section 2.5, Section 3] for Tag and Vf implementations of Poly1305.

$$\mathsf{Poly1305.Tag} : \mathbb{B}_{\mathbb{Y}}^{32} \times \mathbb{B}_{\mathbb{Y}}^{*} \rightarrow \mathbb{B}_{\mathbb{Y}}^{16}$$
$$\mathsf{Poly1305.Vf} : \mathbb{B}_{\mathbb{Y}}^{32} \times \mathbb{B}_{\mathbb{Y}}^{16} \times \mathbb{B}_{\mathbb{Y}}^{*} \rightarrow \mathbb{B}$$

Security of Poly1305

Citing Poly1305 [LN18, Section 4],"the Poly1305 authenticator is designed to ensure that forged messages are rejected with a probability of $1 - (n/(2^{102}))$ for a $16n$-byte message, even after sending 2^{64} legitimate messages, so it is SUF-CMA (strong unforgeability against chosen-message attacks)".

Blake2b-512

Since we need a total of 64 bytes for the key material (32 for ChaCha20 and 32 for Poly1305) Blake2b512 can be used. ZCash protocol [ZCa19, Section 5.4.3], instead, makes use of Blake2b256 since a DHAES variant, denoted as ChaCha20-Poly1305, is adopted (see [LN18, Section 2.8]).

$$\mathsf{Blake2b512} : \mathbb{B}^* \to \mathbb{B}_{\mathbb{Y}}^{32}$$

3.5.3 EncSch instantiation

In the following we instantiate EncSch as a DHAES scheme, detailing the KGen, Enc and Dec components. First, we introduce some required constant values:

$$\begin{aligned}
\mathtt{ESKBYTELEN} &= 32\\
\mathtt{EPKBYTELEN} &= 32\\
\mathtt{NOTEBYTELEN} &= (\mathtt{PRFADDROUTLEN} + \mathtt{RTRAPLEN} + \mathtt{ZVALUELEN} + \mathtt{PRFRHOOUTLEN})/\mathtt{BYTELEN}\\
\mathtt{SYMKEYBYTELEN} &= 32\\
\mathtt{MACKEYBYTELEN} &= 32\\
\mathtt{KDFDIGESTBYTELEN} &= \mathtt{SYMKEYBYTELEN} + \mathtt{MACKEYBYTELEN}\\
\mathtt{CTBYTELEN} &= \mathtt{EPKBYTELEN} + \mathtt{NOTEBYTELEN} + \mathtt{TAGBYTELEN}\\
\mathtt{TAGBYTELEN} &= 16\\
\mathtt{CHACHANONCEVALUE} &= 0^{32}\\
\mathtt{CHACHABLOCKCOUNTERVALUE} &= 0^{96}
\end{aligned}$$

EncSch.KGen

The keypair (sk, pk) generation is defined as:

- Randomly sample a sequence of ESKBYTELEN bytes and assign to sk.
- Clamp sk as follows:

$$\begin{aligned}
sk[0] &\leftarrow sk[0] \ \& \ \mathtt{0xF8}\\
sk[31] &\leftarrow sk[31] \ \& \ \mathtt{0x7F}\\
sk[31] &\leftarrow sk[31] \mid \mathtt{0x40}
\end{aligned}$$

where | and & denotes, respectively, OR and AND binary operators between bit strings of same the length.[9]

- Compute $pk = \mathsf{X25519}(sk, \mathtt{0x09})$.
- Return $(sk, pk) \in \mathbb{B}_{\mathbb{Y}}^{\mathtt{ESKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\mathtt{EPKBYTELEN}}$

EncSch.Enc

The encryption, on inputs $(pk, m) \in \mathbb{B}_{\mathbb{Y}}^{\mathtt{EPKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\mathtt{NOTEBYTELEN}}$, is defined as follows:

1. Generate an ephemeral Curve25519 keypair $(esk, epk) \in \mathbb{B}_{\mathbb{Y}}^{\mathtt{ESKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\mathtt{EPKBYTELEN}}$ (as above).
2. Compute the shared secret[10] $ss \in \mathbb{B}_{\mathbb{Y}}^{\mathtt{EPKBYTELEN}}$:
$$ss = \mathsf{X25519}(esk, pk) \in \mathbb{B}_{\mathbb{Y}}^{\mathtt{EPKBYTELEN}}$$
3. Generate a session key:
$$\mathsf{Blake2b512}(\mathtt{encTag} \| epk \| ss) \in \mathbb{B}_{\mathbb{Y}}^{\mathtt{KDFDIGESTBYTELEN}}$$
where $\mathtt{encTag} = \mathtt{0x5A} \| \mathtt{0x65} \| \mathtt{0x74} \| \mathtt{0x68} \| \mathtt{0x45} \| \mathtt{0x6E} \| \mathtt{0x63}$, that is the UTF-8 encoding of "*ZethEnc*" string (used for domain separation purposes). The result, then, is parsed as follows:
$$\begin{aligned} ek &= \mathsf{Blake2b512}(\mathtt{encTag} \| epk \| ss)[: \mathtt{SYMKEYBYTELEN} - 1] \\ mk &= \mathsf{Blake2b512}(\mathtt{encTag} \| epk \| ss)[\mathtt{SYMKEYBYTELEN} : \mathtt{SYMKEYBYTELEN} + \mathtt{MACKEYBYTELEN} - 1]. \end{aligned}$$
4. Encrypt the confidential data:
$$ct_{\mathsf{Sym}} = \mathsf{ChaCha20}(ek, \mathtt{CHACHABLOCKCOUNTERVALUE}, \mathtt{CHACHANONCEVALUE}, m) \in \mathbb{B}^{\mathtt{NOTEBYTELEN}*\mathtt{BYTELEN}}$$

Remark 3.5.1. Formally speaking we should have written $ct_{\mathsf{Sym}} \in \mathbb{B}^n$, where n is the length of binary representation of the encrypted message m. In `Zeth` however, the only data encrypted are the notes. As such, the size of the plaintexts is $\mathtt{NOTEBYTELEN} * \mathtt{BYTELEN}$ bits.

Remark 3.5.2. In the following, we omit the explicit conversion from $\mathbb{B}^n$ to $\mathbb{B}_{\mathbb{Y}}^{\lceil n/\mathtt{BYTELEN} \rceil}$ when passing the output of ChaCha20 to the Poly1305 algorithms.

5. Randomly generate $(r, s) \in \mathbb{B}_{\mathbb{Y}}^{\mathtt{MACKEYBYTELEN}/2} \times \mathbb{B}_{\mathbb{Y}}^{\mathtt{MACKEYBYTELEN}/2}$ and clamp it:
$$r[3] \leftarrow r[3] \ \& \ \mathtt{0x0F}$$

[9] E.g Given two bytes `0x15` and `0x03` then $\mathtt{0x15}|\mathtt{0x03} = \mathtt{0x17}$ and $\mathtt{0x15}\&\mathtt{0x03} = \mathtt{0x01}$.
[10] We assume here that *esk* has been clamped as discussed in Section 3.5.2

$$r[7] \leftarrow r[7] \ \& \ \texttt{0x0F}$$
$$r[11] \leftarrow r[11] \ \& \ \texttt{0x0F}$$
$$r[15] \leftarrow r[15] \ \& \ \texttt{0x0F}$$
$$r[4] \leftarrow r[4] \ \& \ \texttt{0xFC}$$
$$r[8] \leftarrow r[8] \ \& \ \texttt{0xFC}$$
$$r[12] \leftarrow r[12] \ \& \ \texttt{0xFC}$$

6. Generate the related tag:

$$\tau = \mathsf{Poly1305.Tag}(mk, ct_{\mathsf{Sym}}) \in \mathbb{B}_{\mathbb{Y}}^{\texttt{TAGBYTELEN}}.$$

7. Create the asymmetric ciphertext as:

$$ct = epk \| ct_{\mathsf{Sym}} \| \tau \in \mathbb{B}_{\mathbb{Y}}^{\texttt{CTBYTELEN}}.$$

8. Return ct. As consequence $\texttt{ENCZETHNOTELEN} = \texttt{CTBYTELEN} * \texttt{BYTELEN}$ bits.

EncSch.Dec

The decryption, on inputs $(sk, ct) \in \mathbb{B}_{\mathbb{Y}}^{\texttt{ESKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\texttt{CTBYTELEN}}$, is defined as follows:

1. Parse the ciphertext ct as:

$$\begin{aligned} epk &\leftarrow ct[: \texttt{EPKBYTELEN} - 1] \\ ct_{\mathsf{Sym}} &\leftarrow ct[\texttt{EPKBYTELEN} : \texttt{EPKBYTELEN} + \texttt{NOTEBYTELEN} - 1] \\ \tau &\leftarrow ct[\texttt{EPKBYTELEN} + \texttt{NOTEBYTELEN} : \texttt{EPKBYTELEN} + \texttt{NOTEBYTELEN} + \texttt{TAGBYTELEN} - 1] \end{aligned}$$

2. Recover the shared secret

$$ss = \mathsf{X25519}(sk, epk).$$

3. Compute the $ek \| mk$

$$\begin{aligned} ek &= \mathsf{Blake2b512}(\texttt{encTag} \| epk \| ss)[: \texttt{SYMKEYBYTELEN} - 1] \\ mk &= \mathsf{Blake2b512}(\texttt{encTag} \| epk \| ss)[\texttt{SYMKEYBYTELEN} : \texttt{SYMKEYBYTELEN} + \texttt{MACKEYBYTELEN} - 1]. \end{aligned}$$

4. Verify that the ciphertext has not been forged:

$$\mathsf{Poly1305.Vf}(mk, \tau, ct_{\mathsf{Sym}})$$

5. (If the MAC verifies) decrypt:

$$m = \mathsf{ChaCha20.Dec}(ek, \texttt{CHACHABLOCKCOUNTERVALUE}, \texttt{CHACHANONCEVALUE}, ct_{\mathsf{Sym}})$$

6. Return m.

3.5.4 Security requirements satisfaction

DHAES has already been proved to be IND-CCA2 secure (see [ABR99, Section 3.5, Theorem 6])[11] and to the best of our knowledge there is no paper showing IK-CCA security. The only proof we have found is related to DHIES scheme [ABN10], that is a prime order group version of DHAES. In the following, we provide a proof for IK-CCA security of DHAES by adapting that proof to our case.

Theorem 3.5.1 (IK-CCA of DHAES)**.** *Let* DHAES *be the asymmetric encryption scheme as defined above. Let $\mathcal{A}$ be an adversary for the* IK-CCA *game, then there exists a* HDHI *adversary $\mathcal{B}$ of $(\mathcal{H}, \mathsf{SetupG})$ and a* SUF-CMA *adversary $\mathcal{C}$ of* MAC *such that*

$$\mathsf{Adv}^{\text{ik-cca}}_{\mathsf{DHAES},\mathcal{A}}(\lambda) \leq 2 \cdot \mathsf{Adv}^{\text{hdhi}}_{\mathcal{H},\mathsf{SetupG},\mathcal{B}}(\lambda) + \mathsf{Adv}^{\text{suf-cma}}_{\mathsf{MAC},\mathcal{C}}(\lambda).$$

The adversaries $\mathcal{B}$ and $\mathcal{C}$ have the same running time as $\mathcal{A}$[12]*.*

Informal proof. As already mentioned, DHAES is similar to DHIES scheme, except for the underlying group and the way the symmetric keys are constructed. As consequence, IK-CCA property for DHAES can be shown similarly to the approach in [ABN10, Theorem 6.2]. More precisely, they show that one can construct from an attacker $\mathcal{A}$ for the IK-CCA game two attackers $\mathcal{B}$ and $\mathcal{C}$ for the ODH and SUF-CMA games. Actually, they make use of a $\overline{\mathcal{B}}$ attacker for the ODH2 game [ABN10, Figure 20] and then apply [ABN10, Lemma 6.1] to obtain an attacker $\mathcal{B}$[13] in the ODH game. We adopt a similar strategy, working with HDHI, HDHI2 and Lemma 1.5.1.

Let $\mathcal{A}$ be an attacker for the IK-CCA game, and let $\overline{\mathcal{B}}$ be an attacker for the HDHI2 game described in Fig. 3.11. We show that,

$$\mathsf{Adv}^{\text{hdhi2}}_{\mathcal{H},\mathsf{SetupG},\overline{\mathcal{B}}}(\lambda) = \left|\Pr\left[\mathsf{IK\text{-}CCA}^{\mathcal{A}}(\lambda) = 1\right] + \Pr\left[\mathsf{G_0}^{\mathcal{A}}(\lambda) = 1\right] - 1\right|$$

where G_0 is the security game described in Fig. 3.12.

Given an HDHI2 challenge $(\llbracket u \rrbracket, \llbracket v_0 \rrbracket, \llbracket v_1 \rrbracket, w_{b_2,0}, w_{b_2,1})$, an adversary $\overline{\mathcal{B}}$ samples $b \leftarrow\$ \{0,1\}$ and runs $\mathcal{A}$ on $\llbracket v_0 \rrbracket$, $\llbracket v_1 \rrbracket$ (note that b_2 is the random bit chosen by the $\overline{\mathcal{B}}$ challenger in the HDHI2 game). $\overline{\mathcal{B}}$ constructs oracles $\mathsf{O}^{\mathsf{Dec}_{sk_i}}$ where the queries $(\mathfrak{r} \| ct_{\mathsf{Sym}} \| \tau)$ are processed as follows: if $\mathfrak{r} \neq \llbracket u \rrbracket$, then $\overline{\mathcal{B}}$ queries related HDHI2 oracle to obtain $ek \| mk \leftarrow \mathsf{O}^{\mathsf{HDHI}_{v_i}}(\mathfrak{r})$ (see Fig. 3.11). If $\mathfrak{r} = \llbracket u \rrbracket$, $w_{b_2,i}$ is parsed as $ek \| mk$. In both cases, it checks that $\mathsf{MAC.Vf}(mk, ct_{\mathsf{Sym}}, \tau) = 1$ and, if so, returns $m \leftarrow \mathsf{Sym.Dec}(ek, ct_{\mathsf{Sym}})$. We note that $\mathcal{A}$ cannot query the challenged ciphertext. $\overline{\mathcal{B}}$ returns 0 if and only if $b = \widetilde{b}$. It easy to see that if b_2 is equal to 0, then all symmetric encryption and MAC keys used for the challenge ciphertext $(\mathfrak{r}^* \| ct^*_{\mathsf{Sym}} \| \tau^*)$ and decryption responses are exactly as in a DHAES game.

[11] Specifically, if Sym is IND-CPA secure, it holds that H is HDHI secure and MAC is SUF-CMA secure.

[12] In order to give an asymptotic version of the theorem, the number of queries q has been substituted by the fact of considering PPT adversaries.

[13] Note that in [ABN10] the IK-CCA game is a particular case of the AI-CCA game that requires two input messages in the LR query. In order to reason only about the key-privacy, the two messages m_0 and m_1 are constrained to be equal.

Adversary $\overline{\mathcal{B}}(\llbracket u \rrbracket, \llbracket v_0 \rrbracket, \llbracket v_1 \rrbracket, w_{b_2,0}, w_{b_2,1})$

$b \leftarrow\$ \{0,1\}$
$(m, state) \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Dec}_{sk_0}}, \mathsf{O}^{\mathsf{Dec}_{sk_1}}}(\llbracket v_0 \rrbracket, \llbracket v_1 \rrbracket)$
$ek \| mk \leftarrow w_{b_2,b}$
$\mathfrak{r}^* \leftarrow u$
$ct^*_{\mathsf{Sym}} \leftarrow \mathsf{Sym.Enc}(ek, m)$
$\tau^* \leftarrow \mathsf{MAC.Tag}(mk, ct^*_{\mathsf{Sym}})$
$\widetilde{b} \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{Dec}_{sk_0}}, \mathsf{O}^{\mathsf{Dec}_{sk_1}}}(\mathfrak{r}^* \| ct^*_{\mathsf{Sym}} \| \tau^*, state)$
return $\widetilde{b} = b$

$\overline{\mathcal{B}}$ simulation of $\mathsf{O}^{\mathsf{Dec}_{sk_i}}(\mathfrak{r} \| ct_{\mathsf{Sym}} \| \tau)$

if $\mathfrak{r} \neq \llbracket u \rrbracket$
  $ek \| mk \leftarrow \mathsf{O}^{\mathsf{HDHI}_{v_i}}(\mathfrak{r})$
else
  $ek \| mk \leftarrow w_{b_2,i}$
fi
if $\mathsf{MAC.Vf}(mk, ct_{\mathsf{Sym}}, \tau) = 1$
  return $\mathsf{Sym.Dec}(ek, ct_{\mathsf{Sym}})$
else
  return $\perp$
fi

Figure 3.11: Description of the adversary $\overline{\mathcal{B}}$ for HDHI2, simulating DHAES game for $\mathcal{A}$.

If $b_2 = 1$, then $w_{1,0}$ and $w_{1,1}$ are random strings and the challenge ciphertext and decryption responses are given as in the G_0 game described in Fig. 3.12. So we get,

$$\Pr\left[\mathsf{HDHI2}^{\overline{\mathcal{B}}}(\lambda) = 1\right] = \frac{1}{2} \cdot \Pr\left[\mathsf{IK\text{-}CCA}^{\mathcal{A}}(\lambda) = 1\right] + \frac{1}{2} \cdot \Pr\left[\mathsf{G_0}^{\mathcal{A}}(\lambda) = 1\right].$$

And from the definition of HDHI2 advantage we have

$$\mathsf{Adv}^{\mathsf{hdhi2}}_{\mathcal{H},\mathsf{SetupG},\overline{\mathcal{B}}}(\lambda) = \left|\Pr\left[\mathsf{IK\text{-}CCA}^{\mathcal{A}}(\lambda) = 1\right] + \Pr\left[\mathsf{G_0}^{\mathcal{A}}(\lambda) = 1\right] - 1\right|.$$

At this point, we can conclude as in [ABN10, Theorem 6.2], with the only difference of applying Lemma 1.5.1 instead of [ABN10, Lemma 6.1] and by defining a game G_1 that is *identical until* `bad`[14] G_0 defined in Fig. 3.12. □

3.5.5 Final notes and observations

In this section we list some notes regarding the approach taken in `Zcash` (see [ZCa19, Section 8.7]), and other observations:

- *Key derivation parameters*: in DHAES construction, the only required input variables are the shared secret ss and epk. In the Sprout release of `Zcash`, additional parameters were added (i.e. h_{sig}, pk_{enc} and a counter i) (see [ZCa19, 5.4.4.2]): they state that h_{sig} was used in order to get a different randomness extractor for each joinsplit transfer in order to limit the degradation of the security and weaken assumption on the hash. The authors believed, about the use of long-standing public key pk_{enc}, that it might be necessary for IND-CCA2 security and for post-quantum privacy (in the case where the quantum attacker does not have the public

[14]Games $\mathsf{G_i}$ and $\mathsf{G_j}$ are said to be *identical until* `bad` if they differ only in statements that follow the setting of the `bad` variable to $True$. `bad` is initialized with $False$

$\underline{\mathsf{G}_0(\lambda)}$

$(q, \mathbb{G}, \mathfrak{g}, +) \leftarrow \mathsf{SetupG}(1^\lambda)$
$(sk_0, pk_0), (sk_1, pk_1) \leftarrow\$\ \mathsf{KGen}(1^\lambda)$
$\mathfrak{r}^* \leftarrow\$\ \mathbb{G}$
$ek^* \leftarrow\$\ \{0,1\}^{kLen}$
$mk^* \leftarrow\$\ \{0,1\}^{mLen}$
$(m, state) \leftarrow \mathcal{A}^{\mathsf{O}^{\overline{\mathsf{Dec}}_{sk_0}}, \mathsf{O}^{\overline{\mathsf{Dec}}_{sk_1}}}(pk_0, pk_1)$
$b \leftarrow\$\ \{0,1\}$
$ct^*_{\mathsf{Sym}} \leftarrow \mathsf{Sym.Enc}(ek^*, m)$
$\tau^* \leftarrow \mathsf{MAC.Tag}(mk^*, ct^*_{\mathsf{Sym}})$
$\widetilde{b} \leftarrow \mathcal{A}^{\mathsf{O}^{\overline{\mathsf{Dec}}_{sk_0}}, \mathsf{O}^{\overline{\mathsf{Dec}}_{sk_1}}}(\mathfrak{r}^* \| ct^*_{\mathsf{Sym}} \| \tau^*, state)$
return $\widetilde{b} = b$

$\underline{\text{Oracle } \mathsf{O}^{\overline{\mathsf{Dec}}_{sk_i}}(\mathfrak{r} \| ct_{\mathsf{Sym}} \| \tau)}$

if $\mathfrak{r} = \mathfrak{r}^*$
  $m \leftarrow \perp$
  if $\mathsf{MAC.Vf}(mk^*, ct_{\mathsf{Sym}}, \tau) = 1$
    $\mathsf{bad} \leftarrow$ **true**
    $m \leftarrow \mathsf{Sym.Dec}(ek^*, ct_{\mathsf{Sym}})$
  fi
else
  $m \leftarrow \mathsf{Dec}(sk_i, \mathfrak{r} \| ct_{\mathsf{Sym}} \| \tau)$
fi
return m

Figure 3.12: G_0 game and related decryption oracles for Theorem 3.5.1.

key) [zcaa]. None of these additional components are used any longer starting from the Sapling release (see [ZCa19, 5.4.4.4]). To the best of our knowledge there is no formal reason to use the note counter i as an input to the KDF: an explanation could be to avoid the same session key being reused for multiple notes, but this should not be a problem since a different nonce or block counter is used for the symmetric cipher (actually this is already mandated in the case where epk is reused, as described below).

- *Reuse of ephemeral keys epk*: `Zcash` reuses the same ephemeral keys epk (and different nonces) for two ciphertexts in a joinsplit description, claiming that this does not affect the security of the scheme as soon as the HDHI assumption of the DHAES security proof is adapted. Note that the proof they refer to is related to the IND-CCA2 notion.

- Note that in `Zcash` Sprout and Sapling, being able to break the Elliptic Curve Diffie-Hellman Problem on Curve25519 or Jubjub would not help to decrypt the transmitted notes ciphertext unless the receiver pk_{enc} is known or guessed. On the other hand, having pk_{enc} into the hash (as used in Sprout) may violate in principle the key-privacy of the encryption scheme. For these reasons, we underline that the protocol should enforce a mechanism that does not reveal users public keys to increase the security.

- In [ABN10], the concept of *robustness* for an asymmetric encryption scheme is introduced: it formalizes the infeasibility of producing a ciphertext valid under two different public encryption keys. We note that this is particularly useful for `Zeth` since only the intended receiver will be able to decrypt the encrypted note. In fact, the definition is more general since it also covers the case in which a decryption is successful but returns an incorrect plaintext. This prevents situations where a user, scanning the $\widetilde{\textbf{Mixer}}$ logs for incoming transactions, gets a false positive decryption and stores garbage notes.

> **Note**
>
> We note however, that the "false-positive" situation above can be prevented by relying on a weaker notion of robustness called *collision-freeness* [Moh10]. In fact, as described in Section 2.6, the procedure to receive a *ZethNote* requires to decrypt the ciphertext emitted by the $\widetilde{\textbf{Mixer}}$, and then to verify that the recovered plaintext is the opening of a commitment in the Merkle tree. As such, since the *collision-freeness* of the encryption ensures that plaintexts recovered under different keys are different (i.e. "do not produce a collision"), then we know that plaintexts recovered by parties who are not the intended recipient will fail the "commitment opening verification", leading the payment to be rejected, and solving the aforementioned false-positive issue.

In [ABN10, Section 6], the authors prove that DHIES can be made strongly robust. The proof can be easily adapted to work with DHAES.

- *No public key validation for* X25519: cryptographers have been discussing the absence of any mandated public key validation or checks on the result of X25519. For example, in [LHT16, Section 6.1], an optional zero check is introduced in order to assure that the result of X25519 is not 0: this avoids a situation in which one of the two parties can force the result of the key-exchange by using a small order point as public key. This property is generally defined as *contributory behaviour*, that is, none of the parties is able to force the output of a key exchange. However, protocols do not have all the same security requirements and adding default checks in the Curve25519 specifications would be superfluous in most cases and would add complexities that Bernstein has deliberately chosen to avoid (*simple implementation principle*). More importantly, Diffie-Hellman does not require *contributory behaviour* property [Per17]: modern view is that the only requirements are key indistinguishability and, in case of an active attacker, that the output of the key exchange should not produce a low-entropy function of the honest party's private key (e.g. small-subgroup and invalid-curve attacks). Since these two properties are considered satisfied by Curve25519, there is no need to add extra checks to the Curve25519 specification. We conclude by observing that in the Sprout release, the `Zcash` protocol does not specify any point validation and makes use only of the private key clamping to keep Diffie-Hellman key exchange secure.

3.6 ZkSnarkSch instantiation

Groth's proof system Groth16 [Gro16] is the most efficient known zk-SNARK (in terms of the proof size and proof and verification cost) for QAPs, and thus one of the most efficient NIZK for proving statements on arithmetic circuits. Below we present Groth16's key generation, prover, verifier, and simulator algorithms, adjusted as described in [BGM17] to further reduce the size of *srs* and proofs, and to make the KGen algorithm more amenable to implementation as a multi-party computation.

In what follows, let the number *constNo* of constraints in the relation $\mathbf{R}$ be fixed. Without loss of generality we consider *constNo* to be an *upper bound* on the number of constraints in the $\mathbf{R}$ parameter, and assume that there exists some *constNo*-th root of unity $\omega \in \mathbb{F}_{\mathtt{r}}$. Define $\ell_i(X)$ to be the i-th Lagrange polynomial of degree $(constNo - 1)$ over the set $\{\omega^i\}_{i\in[constNo]}$, and let $\ell(X)$ be the unique non-zero polynomial of degree *constNo* that satisfies $\ell(\omega^i) = 0$ for all $i \in [constNo]$.

We note that the requirement that there exists a *constNo*-th root of unity ω imposes a restriction on the maximum number of constraints in $\mathbf{R}$ that the scheme can support. In the particular case of $\omega \in \mathbb{F}_{\mathtt{r}_{\mathtt{BN}}}$, the restriction becomes $constNo \leq 2^{28}$. For $\mathbb{F}_{\mathtt{r}_{\mathtt{BLS}}}$ this becomes $constNo \leq 2^{47}$.

$\mathsf{KGen}(\mathbf{R}, 1^\lambda)$**:**

i. Pick trapdoor $td = (\tau, \alpha, \beta, \delta) \leftarrow\$ (\mathbb{Z}_p^* \setminus \{\omega^{i-1}\}_{i=1}^{constNo}) \times (\mathbb{Z}_p^*)^3$;

ii. For $j \in \{1, \ldots, inpNo\}$, let

$$u_j(\tau) = \sum_{i=1}^{constNo} U_{ij}\ell_i(\tau),$$
$$v_j(\tau) = \sum_{i=1}^{constNo} V_{ij}\ell_i(\tau),$$
$$w_j(\tau) = \sum_{i=1}^{constNo} W_{ij}\ell_i(\tau);$$

iii. Set

$$srs_{\mathsf{P}} \leftarrow \begin{pmatrix} [\![\alpha]\!]_1, [\![\beta]\!], [\![\delta]\!], \{[\![u_j(\tau)]\!]_1\}_{j=1}^{inpNo}, \{[\![v_j(\tau)]\!]\}_{j=0}^{inpNo}, \\ \{[\![(u_j(\tau)\beta + v_j(\tau)\alpha + w_j(\tau))/\delta]\!]_1\}_{j=inpNoPrim+1}^{inpNo}, \\ \{[\![\tau^i\ell(\tau)/\delta]\!]_1\}_{i=0}^{constNo-2} \end{pmatrix}$$
$$srs_{\mathsf{V}} \leftarrow \left([\![\alpha]\!]_1, [\![\beta]\!]_2, [\![\delta]\!]_2, \{[\![\beta u_j(\tau) + \alpha v_j(\tau) + w_j]\!]_1\}_{j=0}^{inpNoPrim} \right)$$
$$srs \leftarrow (srs_{\mathsf{P}}, srs_{\mathsf{V}})$$

return srs, td

$\mathsf{P}(\mathbf{R}, srs_{\mathsf{P}}, prim = (inp_j)_{j=1}^{inpNoPrim}, aux = (inp_j)_{j=inpNoPrim+1}^{inpNo})$**:**

i. Define

$$a^\dagger(X) = \sum_{j=1}^{inpNo} inp_j u_j(X), \quad b^\dagger(X) = \sum_{j=1}^{inpNo} inp_j v_j(X), \quad c^\dagger(X) = \sum_{j=1}^{inpNo} inp_j w_j(X);$$

ii. Define the polynomial $h(X) = (a^\dagger(X)b^\dagger(X) - c^\dagger(X))/\ell(X)$ and compute the coefficients $\{h_i\}_{i=0}^{constNo-2}$ of h, such that $h(X) = \sum_{i=0}^{constNo-2} h_i X^i$.

iii. $r_a \leftarrow\$ \mathbb{Z}_p$;

iv. $r_b \leftarrow\$ \mathbb{Z}_p$;

v. Compute proof elements:

$$\begin{aligned}
\mathfrak{a} \leftarrow & \sum_{j=1}^{inpNo} inp_j \llbracket u_j(\tau) \rrbracket_1 + \llbracket \alpha \rrbracket_1 + r_a \llbracket \delta \rrbracket_1 \\
\mathfrak{b} \leftarrow & \sum_{j=1}^{inpNo} inp_j \llbracket v_j(\tau) \rrbracket_2 + \llbracket \beta \rrbracket_2 + r_b \llbracket \delta \rrbracket_2 \\
\mathfrak{c} \leftarrow & r_b \mathfrak{a} + r_a \left(\sum_{j=1}^{inpNo} inp_j \llbracket v_j(\tau) \rrbracket_1 + \llbracket \beta \rrbracket_1 \right) + \\
& \sum_{j=inpNoPrim+1}^{inpNo} inp_j \left\llbracket \frac{u_j(\tau)\beta + v_j(\tau)\alpha + w_j(\tau)}{\delta} \right\rrbracket_1 + \\
& \sum_{i=0}^{constNo-2} h_i \left\llbracket \tau^i \ell(\tau)/\delta \right\rrbracket_1
\end{aligned}$$

return $\pi \leftarrow (\mathfrak{a}, \mathfrak{b}, \mathfrak{c})$;

$\mathsf{V}(\mathbf{R}, srs_{\mathsf{V}}, prim = (inp_j)_{j=1}^{inpNoPrim}, \pi)$**:**

i. Check that:

$$\begin{aligned}
\mathfrak{a} \bullet \mathfrak{b} = & \mathfrak{c} \bullet \llbracket \delta \rrbracket_2 \\
& + \left(\sum_{j=1}^{inpNoPrim} inp_j \llbracket u_j(\tau)\beta + v_j(\tau)\alpha + w_j(\tau) \rrbracket_1 \right) \bullet \llbracket 1 \rrbracket_2 \\
& + \llbracket \alpha \rrbracket_1 \bullet \llbracket \beta \rrbracket_2
\end{aligned}$$

Note that $\llbracket \alpha \rrbracket_1$ and $\llbracket \beta \rrbracket_2$ are stored individually and used by the prover to recompute $\llbracket \alpha\beta \rrbracket_T$ seemingly redundantly. This is required in order to leverage the pairing check functionality built in to `Ethereum`, which accepts a sequence of tuples in $\mathbb{G}_1 \times \mathbb{G}_2$ and returns `true` if and only if the product of the resulting pairings equals $\llbracket 1 \rrbracket_T$.

$\mathsf{Sim}(\mathbf{R}, srs, td, prim)$**:**

i. Sample $\mathfrak{a}^* \xleftarrow{\$} \mathbb{Z}_p$; $\mathfrak{b}^* \xleftarrow{\$} \mathbb{Z}_p$;

ii. Compute proof elements:

$$
\begin{aligned}
\mathfrak{a} &\leftarrow [\![\mathfrak{a}^*]\!]_1 + [\![\alpha]\!]_1 \\
\mathfrak{b} &\leftarrow [\![\mathfrak{b}^*]\!]_1 + [\![\beta]\!]_2 \\
\mathfrak{c} &\leftarrow \frac{1}{\delta} \cdot \Bigg[\mathfrak{a}^* \mathfrak{b}^* \, [\![1]\!]_1 + \mathfrak{a}^* \, [\![\beta]\!]_1 + \mathfrak{b}^* \, [\![\alpha]\!]_1 \\
&\qquad - \sum_{j=1}^{inpNoPrim} inp_j \, [\![u_j(\tau)\beta + v_j(\tau)\alpha + w_j(\tau)]\!]_1 \Bigg]
\end{aligned}
$$

return $\pi \leftarrow (\mathfrak{a}, \mathfrak{b}, \mathfrak{c})$;

Chapter 4

Implementation considerations and optimizations

4.1 Client security considerations

In this section we consider some details of client *wallet* software that manages user's private and public keys, `Zeth` notes, and interacts with the **Mixer** contract.

Due to the processing and storage requirements involved, we consider it impractical for all `Zeth` client implementations to assume that a dedicated `Ethereum` node (miner node or archive node) is run on the same host as the wallet. Therefore, in order to interact with the `Ethereum` network, wallet software must communicate with external `Ethereum` P2P nodes via their RPC channel, and must assume that these nodes are completely outside the wallet's control. *From a security standpoint, connected* `Ethereum` *nodes should therefore be considered untrusted, and in particular the details of all RPC calls and responses should be considered publicly visible.* Note that even if the connected `Ethereum` node itself is not malicious, 3rd parties able to see network traffic may also be able to gain an insight into the RPC communication of a specific `Zeth` client.

> **Note**
>
> Note that there are several possible models besides the fully untrusted `Ethereum` node. Organizations or individuals could host one or more "trusted" `Ethereum` nodes, which clients can securely connect to (if they trust the host). This centralization would represent a security trade-off. From the point of view of clients it would create a single point of trust, and for potential malicious observers or attackers it would represent a valuable target.

In what follows we focus on preventing data leaks through network traffic. We do not consider adversaries with physical access to the machine running the wallet (see Appendix C).

> **Note**
>
> Importantly, we focus here on information leakages intrinsic to network communication patterns of the `Zeth` protocol. However, in order to protect against sophisticated adversaries, it is necessary to use network-level anonymity solutions to protect the source of messages emitted on the network. While this is outside of the scope of the `Zeth` protocol, we highly encourage implementers to establish threat models and consider using technologies like *mixnets* to protect against network analysis (see e.g. [PHE+17, DG09]).

4.1.1 Syncing and waiting

`Zeth` clients must periodically synchronize with the latest state of the blockchain. This is necessary to keep track of the data held by the **Mixer** contract, and to detect notes received by the user of the wallet, storing them for future transactions.

Clients should synchronize with `Ethereum` nodes in such as way that information is not leaked. As such:

1. Clients `MUST` use consensus evidence and block headers to verify all data they receive from `Ethereum` nodes.

2. Clients `MUST` locally store all parts of the **Mixer** state they require in order to function.

3. Clients `MUST` obtain all such information by "synchronizing" with the `Ethereum` blockchain and parsing relevant events emitted by **Mixer**. Clients `MUST NOT` query the **Mixer** state via RPC.

4. Clients `SHOULD` take steps to avoid being identified while synchronizing (see Appendix C.2. For example, clients `SHOULD` vary the set of `Ethereum` nodes that they connect to, and `SHOULD NOT` always sync from the block following the last one that they processed.

5. Clients `SHOULD NOT` re-request blocks or transaction receipts that are of particular interest to them. They `SHOULD` process all events, emitted by **Mixer**, in the same way.

6. Clients `SHOULD NOT` make any RPC calls or change their externally visible behaviour in response to blocks or transaction receipts that are of interest to them.

Use of contract queries

We suggest that clients `SHOULD NOT` directly query the contract state, for the reasons discussed in Appendix C.2 and Appendix C.3 (and consequently, Section 4.2 suggests that the **Mixer** contract should, as far as possible, not expose public methods). The

`Zeth` protocol prohibits direct queries of the state of $\widetilde{\textbf{Mixer}}$ (via *public* smart-contract functions) because they introduce a risk that client implementations will leak information by using them.

If implementers choose to add public methods to the $\widetilde{\textbf{Mixer}}$ contract (for application-specific reasons), they should consider carefully the security issues raised in Appendix C. This specification assumes that Mix is the only public method of the $\widetilde{\textbf{Mixer}}$ contract.

4.1.2 Note management

Mix calls on the $\widetilde{\textbf{Mixer}}$ contract emit log events containing new commitment values, nullifiers, the new Merkle root and the secret data for new notes (encrypted using a key derived from the recipients public key). As clients synchronize with the latest state of the blockchain, they `MUST` read these events and correctly process the data they contain.

1. Clients `MUST` process the `MixEventDType` event for every Mix transaction, in the order in which they appear in the blockchain.

2. Clients implementing spending functionality `MUST` use the commitment values in events to track the state of the Merkle tree. The Merkle tree state will be used to generate Merkle paths for future transactions, and `MUST` be made available to the client without the need to query the contract. (Note that not all commitments must necessarily be persisted – see Section 4.3).

3. Clients that can receive notes `MUST` attempt to decrypt the ciphertexts for every transaction (see Item 2 in Section 2.6).

4. Clients `MUST NOT` perform any network-related action, including closing the RPC connection, dependent on successful/unsuccessful decryption of ciphertexts (see Appendix C.3).

5. Clients that can receive notes `MUST` attempt to parse any successfully decrypted plaintext (that is, ensure it is well-formed as in Item 3a in Section 2.6).

6. Clients `MUST NOT` perform any network-related action, including closing the connection, dependent on successful / unsuccessful parsing (see Appendix C.4).

7. Clients that can receive notes `MUST` verify that successfully parsed plaintext data is the opening of the corresponding commitment in the transaction (see Item 3b in Section 2.6).

8. Clients `MUST NOT` perform any network-related action, including closing the connection, dependent on whether the parsed note data is the opening of the corresponding commitment (see Appendix C.4).

9. Clients `MUST` confirm that, after adding the new commitments, the local representation of the Merkle tree of commitments has a root consistent with the event data.

10. Clients SHOULD keep a *local* record of the data related to valid decrypted notes. This will be required in order to spend the notes in a future transaction.

11. Clients implementing spending functionality SHOULD process all nullifiers in Mix transaction events, checking for any corresponding notes previously recorded. Any such note should be marked as spent in the client's local record.

4.1.3 Prepare arguments for Mix transaction

Clients MUST NOT query Ethereum nodes while generating any arguments to a Mix call. In particular, Merkle paths MUST be calculated using the client's local representation of the Merkle tree of commitments that was constructed by parsing events.

Where the zero-knowledge proof is generated by some external process, clients MUST put in place sufficient security schemes to ensure that:

- they are communicating with an authentic proof generation process (not a man-in-the-middle), and
- data sent to and from the proving process cannot be observed in transit and tampered with by a third party, and
- the proof has been generated for the correct instance–witness pair[1]

Without these safe-guards, the operation of the system and the secret data required to spend the input notes may be compromised. See Appendix C.6.

4.1.4 Wallet backup and recovery

Given the restrictions placed on clients and their interaction with the $\widetilde{\mathbf{Mixer}}$ contract, it follows that clients must store all data required to spend notes owned by their users' addresses, and to verify the validity of incoming notes. If this local data is lost, it must be reconstructed before client operations can resume.

Zeth private keys (see Table 1.5) can be used to fully restore client state. In this case, clients MUST retrieve all events from the beginning of the $\widetilde{\mathbf{Mixer}}$ contract's history, decrypting notes and tracking nullifiers, as described in the previous sections, to reconstruct the set of unspent notes that they own.

Without a backup of the private keys it is not possible to restore wallet state. As such, private keys are the minimal set of data that must be securely stored and backed up, and clients SHOULD provide support for this mode of recovery. However, to avoid the need to scan all events emitted by $\widetilde{\mathbf{Mixer}}$ (a very expensive operation) implementations SHOULD also support back ups of further state data (such as the representation of the Merkle tree of note commitments, the set of unspent notes, etc) to allow more efficient modes of recovery.

[1] Although given an acceptable zk-proof π for an instance *prim* it is infeasible to check which witness has been used – which comes directly from the zero-knowledge property – we need to assure security measures that prevents any third party from mauling and tampering with the proof generation process.

4.2 Contract security considerations

Section 4.1 mentions several considerations for client implementations, concerning how they interact with the contract. These must be taken into account when authoring the contract code, to ensure that clients can securely retrieve the information they need to operate without encouraging them to perform insecure operations.

1. $\widetilde{\textbf{Mixer}}$ `MUST` validate inputs, the contract needs to ensure that the primary inputs are elements of the scalar field $\mathbb{F}_{\mathtt{r}}$ (that is, they are in the range $\{0, \ldots, \mathtt{r} - 1\}$).
2. $\widetilde{\textbf{Mixer}}$ `MUST` output events for valid Mix calls, including:
   (a) commitment for each new note;
   (b) nullifier for each spent note;
   (c) value of new Merkle root of commitments;
   (d) ciphertexts for each new note;
   (e) implementation-specific data (such as the one-time sender public key specified in Section 3.5, required to decrypt the ciphertexts).
3. The Mix function `MUST` be *payable*[2], to support non-zero *vin*.
4. $\widetilde{\textbf{Mixer}}$ `MUST NOT` expose any public methods except for Mix.

4.3 Efficiency and scalability

4.3.1 Importance of performance

Poor performance and scalability has several impacts on the viability of the system.

Efficiency and performance are arguably most important for the $\widetilde{\textbf{Mixer}}$ contract, where gas usage directly affects the monetary cost of using `Zeth` to transfer value. That is, high gas costs could make transactions very expensive, and therefore not practical for many use-cases, undermining the utility and viability.

High storage or compute requirements on the client would severely restrict the set of devices on which `Zeth` client software can run, and long delays when sending or receiving transactions can adversely affect the user-experience, discouraging some users and undermining the privacy promises of the system.

Although the proof-of-concept implementation of `Zeth` is not intended to be used in a production environment, one of its aims is to demonstrate the practicality of the protocol in terms of transaction costs. Therefore, some of the techniques described here have been included in the proof-of-concept implementation, while in some cases implementers of production software may wish to make different trade-offs.

[2] see `https://solidity.readthedocs.io/en/v0.6.2/types.html?highlight=payable#function-types`

4.3.2 Cost centers

One important factor, primarily affecting client performance, is the cost of zero-knowledge proof generation. This is directly related to the number of constraints used to represent the statement in Section 2.2, which in turn depends on the specific cryptographic primitives used (see Chapter 3).

Note that cryptographic primitives which are "snark-friendly" (i.e. can be implemented using fewer gates in an arithmetic circuit) may not necessarily run efficiently on the EVM or on standard hardware. As such, trade-offs must be made between proof generation cost and the gas costs of state transitions. An example of this is the hash function used in the Merkle tree of commitments. This is not only used in the statement of Section 2.2 (to verify Merkle proofs, see Section 2.2), but also on the client (to create Merkle proofs, see Section 2.3) and in the **Mixer** contract (to compute the Merkle root, see Section 2.5).

Aside from the specific hash function used, implementers have some freedom in the data structures and algorithms used to maintain the Merkle tree and generate proofs. Because of this freedom, and the importance of the chosen algorithms on performance across all components of the system, the majority of this section focuses on the details of the Merkle tree.

As described in Chapter 2, `Zeth` notes are maintained and secured by the Merkle tree, whose depth `MKDEPTH` must be fixed when the contract is deployed. Therefore, `MKDEPTH` determines the maximum number of notes ($2^{\texttt{MKDEPTH}}$) that may be created over the lifetime of the deployment. To ensure the utility of `Zeth`, `MKDEPTH` must be sufficiently large, and therefore the following includes a discussion of *scalability* with respect to `MKDEPTH`.

Also, due to the fact that `MKDEPTH` is fixed, we assume that Merkle proofs are computed as `MKDEPTH`-tuples, no matter how many leaves have been populated. Unpopulated leaves are assumed to take some default value (usually a string of zero bits).

4.3.3 Client performance

Commitment Merkle tree

The simplest possible implementation, which stores only the data items at the leaves of the tree, requires $2^{\texttt{MKDEPTH}} - 1$ hash invocations to compute the Merkle root or to generate a Merkle proof. The cost of this is too high to be practical for non-trivial values of `MKDEPTH`.

An immediate improvement in compute costs can be achieved by simply storing all nodes (or all nodes whose value is not the default value) and updating only those necessary as new commitments are added. When adding `JSOUT` consecutive leaves to the tree, after $\mathcal{O}(\log_2(\texttt{JSOUT}))$ layers (requiring $\mathcal{O}(\texttt{JSOUT})$ hashes) we reach the common ancestor of all new leaves and can update the Merkle tree by proceeding along a single branch (of approximately $\texttt{MKDEPTH} - \log_2(\texttt{JSOUT})$ layers). Thus, the cost of updating the Merkle tree for a single transaction has a fixed bound which is linear in `JSOUT` and

MKDEPTH. However, this doubles the storage cost of the tree since non-leaf nodes must also be persisted.

In the case of the client, the Merkle tree will only be used to generate proofs for notes owned by the user of the client. Thereby Zeth clients need only store nodes of the Merkle tree that are required for this purpose, and may discard all others. In particular, any full sub-tree need only contain nodes that are part of Merkle paths associated with the user's notes. Implementations that discard unnecessary nodes can achieve vast savings in storage space.

Zero-knowledge proof generation

As well as keeping the number of constraints as low as possible, it is important to ensure that the prover implementation is optimal and thereby that proving times are as short as possible. Proof generation should also exploit any available parallelism, to help reduce the time taken. This may require specific programming languages or frameworks to be used, necessitating that proof generation be performed by some external process (as is the case in the proof-of-concept implementation).

The proof generation process can also be very memory intensive (in part due to the FFT calculations required), and so ensuring that enough RAM is present in the system is important to avoid long proof times.

See Appendix C.6 for a discussion of related security concerns.

4.3.4 Contract performance

For most components of the contract, the set of operations to be performed is strictly defined and the set of possible algorithmic optimizations that can be made is limited. In these cases, it is important to ensure that code is benchmarked and optimized to a reasonable degree, to minimize gas costs. We note that apart from the number and type of compute instructions executed, store and lookup operations have a significant impact on the gas used. In particular, storing new values is more expensive than overwriting existing values, and a gas rebate is made when contracts release stored values. See [Woo19, Appendix H.1] for further details.

The primary component in which algorithmic optimizations can be made is the Merkle tree of note commitments. The **Mixer** contract must compute (and store) the new Merkle root after adding the JSOUT new commitments as leaves.

As in Section 4.3.3, the simplest possible implementation which stores only the data items at the leaves of the tree, requires the full root to be recomputed, involving $2^{\texttt{MKDEPTH}}-1$ hash invocations. This quickly becomes impractical for non-trivial values of MKDEPTH.

The first-pass optimization (also described in Section 4.3.3) can be used to ensure that the cost of updating the Merkle tree (number of hash computations, stores and loads) is bounded by a constant that is linear in the Merkle tree depth. This is the strategy used in the proof-of-concept implementation of **Mixer**.

It may be possible to gain further improvements in gas costs by discarding nodes from the Merkle tree that are not required. Unlike clients, **Mixer** is only required to compute the new Merkle root, and does not need to create or validate Merkle proofs (as these are checked as part of the zero-knowledge proof). Consequently, *all* nodes in a sub-tree can be discarded when the sub-tree is full, and the optimization is much simpler to implement than on the client.

Another possible strategy for decreasing the gas costs associated with Merkle trees is *Merkle Shrubs*, described in [Lab19, Section 2.2]. Under this scheme, the contract maintains a "frontier" of roots of sub-trees and Merkle proofs provided by clients (as auxiliary inputs to the $\mathbf{R}^{\mathsf{z}}$ circuit) contain a path from the leaf to one of the nodes in the frontier. The gas savings in this scheme are due to the fact that, for new commitments, the contract need only recompute the value of nodes from the leaf to the "frontier" (not all the way to the root of the tree). However this comes at the cost of complexity in the arithmetic circuit, which must verify a Merkle path to one of several frontier nodes.

When choosing cryptographic primitives to be used on the EVM (and considering the trade-off with other platforms, described in Section 4.3.1) it may be valuable to note that the EVM supports so-called "pre-compiled contracts". These behave like built-in functions providing very gas-efficient access to certain algorithms, such as Keccak. However, pre-compiled contracts exist only for a limited set of algorithms. Others must be implemented using EVM instructions.

4.3.5 Optimizing Blake2's circuit.

After presenting Blake2s circuit and its components working on little endian variables, we show a few optimizations.

Helper circuits

We first define the following helper circuits needed in the Blake2s routine, operating on w-bit long words.

XOR circuits The following XOR circuits on w-bit long variables have been implemented, we assume the inputs are boolean (this is not checked in these circuits),

- "Classic" XOR circuit, which xors 2 variables,
  $a \oplus b = c$;
- XOR with constant, which xors two variables and a constant,
  $a \oplus b \oplus c = d$, with c constant;
- XOR with rotation, which xors two variables and rotates the result.
  $a \oplus b \ggg r = c$, with r constant, and $\ggg$ the rightward rotation [MJS15, Section 2.3]; i.e. for and constant $r < w$ we have $a_i \oplus b_i = c_{i+r \pmod w}$, for $i = 0, \ldots, w$,

Each of these circuits presents w constraints. Assuming that the inputs are boolean, the output is automatically boolean. To ascertain that both inputs are boolean (a and b), we would need $2 \cdot w$ more gates per circuit.[3]

Modular addition We present here two circuits to verify modular arithmetic.

Double modular addition: $\boldsymbol{a + b = c \pmod{2^w}}$. This circuit checks that the sum of two w-bit long variables in little endian format modulo 2^w is equal to a w-bit long variable. More precisely, it checks the equality of the modular addition of $a + b \pmod{2^w}$ and c and the booleaness of the later. We assume the inputs are boolean (this is not checked in this circuit).

As the addition of two w-bit long integers results in at most an $(w + 1)$-bit integer, we consider c to be $(w + 1)$-bit long. We do not care about the last bit value, c_w, but have to ensure its booleaness.

The circuit presents the following $w + 2$ constraints, for a and b of size w, where $w = 32$ in practice, and variable c of size $w + 1$, that:

$$\sum_{i=0}^{w-1} (a_i + b_i) \cdot 2^i = \sum_{j=0}^{w} c_j \cdot 2^j \tag{4.1}$$

$$\forall j \in \{0, \dots, w\},\ (c_j - 0) \cdot (c_j - 1) = 0 \tag{4.2}$$

Triple modular addition: $\boldsymbol{a + b + c = d \pmod{2^w}}$. This circuit checks the equality of a w-bit long variable d with the sum of three w-bit long variables in little endian format modulo 2^w. More precisely, it checks the equality of the modular addition of $a + b + c \pmod{2^w}$ and d and the booleaness of the latter. We assume the inputs are boolean (this is not checked in this circuit).

As the addition of three w-bit long integers results in at most an $(w + 2)$-bit integer, we consider d to be $(w + 2)$-bit long. We do not care about the values of the last two bits (d_w and d_{w+1}), but have to ensure their booleaness.

The circuit presents the following $w + 3$ constraints, for a, b and c of size w, where $w = 32$ in practice, and variable d of size $w + 2$, that:

$$\sum_{i=0}^{w-1} (a_i + b_i + c_i) \cdot 2^i = \sum_{j=0}^{w+1} d_j \cdot 2^j \tag{4.3}$$

$$\forall j \in \{0, \dots, w + 1\},\ (d_j - 0) \cdot (d_j - 1) = 0 \tag{4.4}$$

[3]Making sure that no gates are duplicated in the circuit is very important to keep the proving time as small as possible. One challenge of writing R1CS programs is to make sure that the statement is correctly represented, without redundancy, in order to keep the constraint system as small as possible.

$\mathsf{G}(a, b, c, d; x, y) \mapsto (a_2, b_2, c_2, d_2)$

$$
\begin{aligned}
& a_1 \leftarrow a + b + x \pmod{2^w} \\
& d_1 \leftarrow d \oplus a_1 \ggg r_1 \\
& c_1 \leftarrow c + d_1 \pmod{2^w} \\
& b_1 \leftarrow b \oplus c_1 \ggg r_2 \\
& a_2 \leftarrow a_1 + b_1 + y \pmod{2^w} \\
& d_2 \leftarrow d_1 \oplus a_2 \ggg r_3 \\
& c_2 \leftarrow c_1 + d_2 \pmod{2^w} \\
& b_2 \leftarrow b_1 \oplus c_2 \ggg r_4 \\
& \textbf{return } a_2, b_2, c_2, d_2
\end{aligned}
$$

Figure 4.1: G primitive [MJS15, Section 3.1]

$\mathsf{getSigma}()$

$$
\begin{aligned}
& \Sigma \in (\mathbb{N}^{16})^{10} \\
& \Sigma[0] \leftarrow (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) \\
& \Sigma[1] \leftarrow (14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3) \\
& \Sigma[2] \leftarrow (11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4) \\
& \Sigma[3] \leftarrow (7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8) \\
& \Sigma[4] \leftarrow (9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13) \\
& \Sigma[5] \leftarrow (2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9) \\
& \Sigma[6] \leftarrow (12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11) \\
& \Sigma[7] \leftarrow (13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10) \\
& \Sigma[8] \leftarrow (6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5) \\
& \Sigma[9] \leftarrow (10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0) \\
& \textbf{return } \Sigma
\end{aligned}
$$

Figure 4.2: Blake2 permutation table [MJS15, Section 2.7]

Blake2s **routine circuit**

We define in this section the circuit of the Blake2 routine (see [MJS15, Section 3.1] and Fig. 4.1) known as "G function" [ANWOW13, Section 2.4]. G is based on ChaCha encryption [Ber08a]. It works on w-bit long words, and presents $8 \cdot w + 10$ constraints. The function mixes a state (a, b, c and d) with the inputs (x and y) and returns the updated state.

This circuit does not check the booleaness of the inputs or state. However, given that the state is boolean, the output is automatically boolean due to the use of the modular addition circuits.

For Blake2s, we have $w = 32$, $r_1 = 16$, $r_2 = 12$, $r_3 = 3$ and $r_4 = 7$.

Blake2s **compression function circuit**

The compression function is defined as follows, for more details see Fig. 4.3,

$$\mathsf{Blake2sC} : \mathbb{B}^n \times \mathbb{B}^{2n} \times \mathbb{B}^{n/4} \times \mathbb{B}^{n/4} \to \mathbb{B}^n\,.$$

Blake2C takes as input a state $h \in \mathbb{B}^n$ which is used as chaining value when hashing, a message to compress $x \in \mathbb{B}^{2n}$, a message length written in binary $t \in \mathbb{B}^{n/4}$ which is incremented when hashing and a binary flag $f \in \mathbb{B}^{n/4}$ to tell whether the current block is the last to be compressed to prevent length extension attacks.

Blake2C uses the G function iteratively over `rounds` number of rounds on a state and message. The constant initialization vector `IV` and the permutation table `Σ` are hard-coded. Blake2sC works in little endian (see [MJS15, Section 2.4]) on n-bit long variables ($n = 256$), w-bit long words ($w = 32$), and the rotation constants specified in Section 4.3.5 (see [MJS15, Section 2.1]). We have the following constants (see specifications [ANWOW13] and [MJS15, Section 2.2]),

- `IV` is the $(8 \cdot w)$-bit long initialization vector; it corresponds to the first w bits of the fractional parts of the square roots of the first eight prime numbers $(2, 3, 5, 7, \ldots)$ (see [MJS15, Section 2.6]);
- $\mathtt{\Sigma}$ are the $10 \cdot 16$ permutation constants of Blake2 (see Fig. 4.2 and [MJS15, Section 2.7]);
- `rounds`, the number of rounds: 10 for Blake2sC, 12 for Blake2bC.

We have the following variables (see specifications [ANWOW13] and [MJS15, Section 2.2]),

- `H` is the $(8 \cdot w)$-bit long initial state while v is the $(16 \cdot w)$-bit long final state;
- $\mathtt{T}[i]$ are two w-bit long counters encoding the block length;
- $\mathtt{F}[i]$ are two w-bit long finalization flags. We set the first one $\mathtt{F}[0]$ to $2^w - 1$ to state when the input block is the last one to be hashed. The second, $\mathtt{F}[1] = 0$ is only set for tree hashing mode (which is not our case) and is therefore unused.

We introduce the following functions to write Blake2C (see specifications [ANWOW13] and [MJS15, Section 2.6]):

- The function prime takes a positive integer i as input and outputs the i-th prime number;
- The function dec takes a real number x as input outputs its positive decimal part.

This circuit presents $((64 \cdot \mathtt{rounds} + 8) \cdot w + 8 \cdot \mathtt{rounds} + 10)$ constraints. For Blake2sC, as $w = 32$ and $\mathtt{rounds} = 10$, we have 21536 constraints.

We do not check the input block booleaness in this circuit. Given that the initial state is boolean, the output is automatically boolean. This can be proved iteratively by the booleaness of G primitive's output.

Security requirement. The inputs to Blake2sC `MUST` be boolean.

Blake2s **hash function**

The hash function is defined as follows, for more details see Fig. 4.3,

$$\mathsf{Blake2s} : \mathbb{B}^{\leq 2n} \times \mathbb{B}^* \to \mathbb{B}^n$$

Blake2 takes as input a hash key $k \in \mathbb{B}^n$ and the message to hash $x \in \mathbb{B}^{2n}$. Blake2 uses the Blake2C function iteratively over each $2n$-bit long chunk of the padded message. If the key is non null, it is used as the first block to be hashed. The constant initialization vector `IV` and part of the parameter block `PB` are hard-coded. We have the following constants (see specifications [ANWOW13] and [MJS15, Section 2.2]),

$\mathsf{Blake2C}(h, m, t, f)$

$\mathtt{T}, \mathtt{F}, \mathtt{H}, \mathtt{IV}, v \in (\mathbb{B}^w)^2 \times (\mathbb{B}^w)^2 \times (\mathbb{B}^w)^8 \times (\mathbb{B}^w)^8 \times (\mathbb{B}^w)^{16}$

$\{\mathtt{IV}[i]\}_{i\in[8]} \leftarrow \left\{ \left\lfloor 2^w \cdot \mathsf{dec}(\sqrt{\mathsf{prime}(i+1)}) \right\rfloor \right\}_{i\in[8]}$

$\Sigma \leftarrow \mathsf{getSigma}()$

$\{\mathtt{H}[i]\}_{i\in[8]} \leftarrow \{h[i \cdot w{:}(i+1)\cdot w]\}_{i\in[8]}$

$\{m[i]\}_{i\in[8]} \leftarrow \{x[i \cdot w{:}(i+1)\cdot w]\}_{i\in[8]}$

$\mathtt{T}[0],\ \mathtt{T}[1] \leftarrow t[w{:}2w],\ t[0{:}w]$

$\mathtt{F}[0],\ \mathtt{F}[1] \leftarrow f[w{:}2w],\ f[0{:}w]$

$\{v[i]\}_{i\in[8]} \leftarrow \{\mathtt{H}[i]\}_{i\in[8]}$

$\{v[i+8]\}_{i\in[8]} \leftarrow \{\mathtt{IV}[i]\}_{i\in[8]}$

$v[12],\ v[13] \leftarrow v[12] \oplus \mathtt{T}[0],\ v[13] \oplus \mathtt{T}[1]$

$v[14],\ v[15] \leftarrow v[14] \oplus \mathtt{F}[0],\ v[15] \oplus \mathtt{F}[1]$

foreach $r \in [\mathtt{rounds}]$ **do**

  $\tau \leftarrow \Sigma[r \pmod{15}]$

  $v[0], v[4], v[8], v[12] \leftarrow \mathsf{G}(v[0], v[4], v[8], v[12], m[\tau[0]], m[\tau[1]])$

  $v[1], v[5], v[9], v[13] \leftarrow \mathsf{G}(v[1], v[5], v[9], v[13], m[\tau[2]], m[\tau[3]])$

  $v[2], v[6], v[10], v[14] \leftarrow \mathsf{G}(v[2], v[6], v[10], v[14], m[\tau[4]], m[\tau[5]])$

  $v[3], v[7], v[11], v[15] \leftarrow \mathsf{G}(v[3], v[7], v[11], v[15], m[\tau[6]], m[\tau[7]])$

  $v[0], v[5], v[10], v[15] \leftarrow \mathsf{G}(v[0], v[5], v[10], v[15], m[\tau[8]], m[\tau[9]])$

  $v[1], v[6], v[11], v[12] \leftarrow \mathsf{G}(v[1], v[6], v[11], v[12], m[\tau[10]], m[\tau[11]])$

  $v[2], v[7], v[8], v[13] \leftarrow \mathsf{G}(v[2], v[7], v[8], v[13], m[\tau[12]], m[\tau[13]])$

  $v[3], v[4], v[9], v[14] \leftarrow \mathsf{G}(v[3], v[4], v[9], v[14], m[\tau[14]], m[\tau[15]])$

return $\|_{i=0}^{8} \mathtt{H}[i] \oplus v[i] \oplus v[i+8]$

Figure 4.3: Blake2 compression function [MJS15, Section 3.2]. Set n, w and G's constants to obtain Blake2sC.

- IV is the $(8 \cdot w)$-bit long Initialization Vector; it corresponds to the first w bits of the fractional parts of the square roots of the first eight prime numbers $(2, 3, 5, 7, \ldots)$ (see [MJS15, Section 2.6]).

We have the following variables (see specifications [ANWOW13] and [MJS15, Section 2.2]),

- PB is the $(16 \cdot w)$-bit long parameter block used to initialize the state (see [MJS15, Section 2.5]). In big endian encoding, the first byte corresponds to the digest length (fixed to 32 bytes), the second byte to the key length, the third and fourth bytes correspond to the use of the serial mode;

- $\mathtt{H} \in \mathbb{B}^{\mathtt{BLAKE2sCLEN}}$, the chaining value.

$\mathsf{Blake2}(k, x)$

$\mathtt{H}, \mathtt{IV}, \mathtt{PB} \in \mathbb{B}^{8w} \times \mathbb{B}^{8w} \times \mathbb{B}^{8w}$

$\mathtt{PB} \leftarrow \mathsf{pad}_{8 \cdot w}(\mathsf{encode}_{\mathbb{N}}(0\text{x}0101) \| \mathsf{pad}_w(\mathsf{encode}_{\mathbb{N}}(\lceil \mathsf{length}(k)/\mathtt{BYTELEN} \rceil)) \| \mathsf{encode}_{\mathbb{N}}(0\text{x}20))$

$\mathtt{IV} \leftarrow \|_{i=0}^{8} \left\lfloor 2^w \cdot \mathsf{dec}(\sqrt{\mathsf{prime}(i+1)}) \right\rfloor$

$\mathtt{H} \leftarrow \mathtt{PB} \oplus \mathtt{IV}$

$y \leftarrow x$

if $\mathsf{length}(k) \neq 0$ **do**

    $y \leftarrow \mathsf{pad}_{2n}(k) \| y$

$z \leftarrow \mathsf{pad}_{2n \cdot \lceil \mathsf{length}(y)/2n \rceil}(y)$

for $i \in [\lceil \mathsf{length}(z)/2n \rceil]$ **do**

    if $i = \lceil \mathsf{length}(z)/2n \rceil - 1$ **do**

        $\mathtt{H} \leftarrow \mathsf{Blake2C}(\mathtt{H}, z[i \cdot 2n{:}(i+1) \cdot 2n], \mathsf{pad}_{2w}(\mathsf{encode}_{\mathbb{N}}(\lceil \mathsf{length}(y)/\mathtt{BYTELEN} \rceil)), \mathsf{pad}_{2w}(\mathsf{encode}_{\mathbb{N}}(2^w - 1)))$

    else

        $\mathtt{H} \leftarrow \mathsf{Blake2C}(\mathtt{H}, z[i \cdot 2n{:}(i+1) \cdot 2n], \mathsf{pad}_{2w}(\mathsf{encode}_{\mathbb{N}}((i+1) \cdot 2n/\mathtt{BYTELEN})), \mathsf{pad}_{2w}(0))$

return $\mathtt{H}$

Figure 4.4: Blake2 hash function [MJS15, Section 3.3]. Set $n = 16w$ and G's constants accordingly to obtain Blake2s.

We do not check the input block booleaness in this circuit. Given that the initial state is boolean, the output is automatically boolean. This can be proved iteratively by the booleaness of Blake2C primitive's output.

Security requirement To ensure the correct use of Blake2s, Blake2s's inputs `MUST` be boolean.

Optimizing the circuits

The above helper circuits form the building blocks of the Blake2s compression function. We show here two exclusive methods to optimize these circuits.

Optimizing the Modular additions

Double modular addition: $a + b = c \pmod{2^w}$. We present here an optimization on the circuit to save one constraint by merging the modular constraint with a boolean constraint. The optimized circuit presents the following constraints:

$$\left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i \right) \cdot \left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i - 2^w \right) = 0 \tag{4.5}$$

$$\forall j \in \{0, \ldots, w-1\},\ (c_j - 0) \cdot (c_j - 1) = 0 \tag{4.6}$$

with $\sum_{i=0}^{w-1} x_i \cdot 2^i$ a binary encoding of x (x_i is the i^{th} bit of x).

These equations describe $w+1$ constraints to prove the bit equality $a+b=c$ (note that an additional $2 \cdot w$ constraints would be required to prove the booleaness of input variables a and b). We now explain how we obtained them.

Proof. The most straightforward way to prove that $a+b=c \pmod{2^w}$ and c booleaness is with the set of constraints illustrated in Eq. (4.1) and in Eq. (4.2).

As we perform arithmetic modulo 2^w, we do not care about the value of c_w but would like to ensure its booleaness. As one may notice, the summing constraint Eq. (4.1) is an equality of two linear combinations with no multiplication by a variable. Hence, we can combine it with the boolean constraint of c_w to remove any reference to c_w and still have a bilinear gate. To do so, we first rewrite Eq. (4.1) as an equality check over $c_w \cdot 2^w$ and multiply Eq. (4.2) for $j=n$ by $2^{2 \cdot w}$.

$$\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i = c_w \cdot 2^w \tag{4.7}$$

$$2^w \cdot (c_w - 0) \cdot 2^w \cdot (c_w - 1) = 0 \tag{4.8}$$

We finally replace $c_w \cdot 2^w$ in Eq. (4.8) by the value from Eq. (4.7).

$$\begin{aligned} 0 = 2^w \cdot (c_w - 0) \cdot 2^w \cdot (c_w - 1) &= 2^w \cdot c_w \cdot (2^w \cdot c_w - 2^w) \\ &= \left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i \right) \cdot \left(\left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i \right) - 2^w \right) \end{aligned}$$

This results in Eq. (4.5) and Eq. (4.6). All references to c_w have disappeared and, with a single multiplication by a variable, we still have bilinear gates. □

Triple modular addition: $a+b+c=d \pmod{2^w}$. To optimize, we use the above circuit twice. We define a temporary variable d' such that $a+b=d' \pmod{2^w}$. As such, we have $c+d'=d \pmod{2^w}$. As d' is the addition of two w-bit long variables, it is $(w+1)$-bit long. However as we evaluate the sum modulo 2^w, we discard the last bit of d'. We proceed similarly for d. To ensure that d is boolean, we check the booleaness of the $w+1$ bits of d as well as the booleaness of the last bit of d' (to account for d's $w+2^{th}$ bit in the original expression ($a+b+c=d \pmod{2^w}$)).

We thus obtain the following circuit with $w+2$ constraints,

$$\begin{aligned} \left(\sum_{i=0}^{w-1} (a_i + b_i - d'_i) \cdot 2^i \right) \cdot \left(\sum_{i=0}^{w-1} (a_i + b_i - d'_i) \cdot 2^i - 2^w \right) &= 0 \\ \left(\sum_{i=0}^{w-1} (c_i + d'_i - d_i) \cdot 2^i \right) \cdot \left(\sum_{i=0}^{w-1} (c_i + d'_i - d_i) \cdot 2^i - 2^w \right) &= 0 \\ \forall j \in \{0, \ldots, w-1\},\ (d_j - 0) \cdot (d_j - 1) &= 0 \end{aligned}$$

These optimizations lead to a gain of 320 constraints ($= 4 \cdot 8 \cdot$ *rounds*).

Optimizing Blake2s routine's circuit As seen in Fig. 4.1, our routine presents 2 double and 2 triple modular additions. Each of these circuits comprises at least one modular constraint which pack several w-bit long variables. The circuit is however processed in $\mathbb{F}_{\mathtt{r}}$, that is to say most integers can be written over `FIELDCAP` bits. We can thus batch the modular constraints. As the G primitive performs 2 double modular and 2 triple modular, we have in total 6 modular checks per iteration. We can batch up to $\mathtt{FIELDCAP}/w$ constraints together. For $w = 32$ and $\mathtt{FIELDCAP} \geq 224$ (which holds for BN-254 and BLS12-377), we can encode up to 7 words per field element, that is to say we can include all the modular constraints into a single one.

This optimization leads to a gain of 274 constraints ($= 4 \cdot 8 \cdot 10 - \lceil \frac{4 \cdot 8 \cdot 10}{7} \rceil$).

Optimization conclusion Using the more efficient optimization on the modular additions, the Blake2s compression function comprises 21216 constraints.

Increasing the PRF security with Blake

As Blake2 comprises a personalization tag in its parameter block `PB`, one could ensure the independence of the PRFs by writing different tags for each of them (we would be able to consider up to 2^{30} inputs and outputs). We did not choose to write this enhancement in the instantiation to keep a general tagging method in case of a change of hash function.

4.4 Encryption of the notes

In this section we give some remarks concerning the implementation of the `Zeth` encryption scheme, described in Section 3.5. As noted, there are several details in the specification of the underlying primitives which can impact security if not carefully implemented. The following list is by no means exhaustive but includes several details noted during development of the proof-of-concept system.

- Private keys for Curve25519 `MUST` be randomly generated as 32 bytes where the first byte is a multiple of 8, and the last byte takes a value between 64 and 127 (inclusive). Further details are given in [Ber06], including an example algorithm for generation. Implementations `MUST` take care to ensure that their code, or any external libraries they rely upon, correctly perform this step.

- A similar observation holds for Poly1305 in which the r component of the MAC key (r, s) `MUST` be *clamped* in a specific way (see Section 3.5.3). This step is also essential and `MUST` be performed.

- In the implementation of the ChaCha stream cipher, correct use of the *key*, *counter* and *nonce* `MUST` be ensured in order to adhere to the standard and guarantee the appropriate security properties.

During the proof-of-concept implementation it was not obvious that the cryptography library[4] adhered to the specifications with respect to the above points. In particular, it was not clear whether key clamping was performed at generation time and/or when performing operations. Moreover, the interface to the ChaCha cipher accepted a different set of input parameters (namely *key* and *nonce* with no *counter*). This left some ambiguity about the responsibility for clamping, and whether the ChaCha block data would be updated as described in the specification. Details of how this was resolved are given in the proof-of-concept encryption code, which may prove a useful reference for implementers[5].

[4] `https://cryptography.io/en/latest/`

[5] see `https://github.com/clearmatics/zeth/blob/v0.4/client/zeth/encryption.py`

Appendix A

Transaction non malleability

The transaction malleability problem for a DAP (Section 1.4) is characterized by a game $\mathsf{TR\text{-}NM}$ involving a polynomial-time adversary $\mathcal{A}$ as described below.

Definition A.0.1. Let DAP be a (candidate) Decentralized Anonymous Payment scheme.

$$\mathsf{DAP} = (\mathsf{Setup},\ \mathsf{GenAddr},\ \mathsf{SendTx},\ \mathsf{VerifyTx},\ \mathsf{Receive})$$

We say that DAP is $\mathsf{TR\text{-}NM}$ secure if, for every $\mathsf{poly}(\lambda)$-time adversary $\mathcal{A}$

$$\mathsf{Adv}^{\mathsf{tr\text{-}nm}}_{\mathsf{DAP},\mathcal{A}}(\lambda) < \mathsf{negl}(\lambda),$$

where $\mathsf{Adv}^{\mathsf{tr\text{-}nm}}_{\mathsf{DAP},\mathcal{A}}(\lambda) = \Pr[\mathsf{TR\text{-}NM}(\mathsf{DAP}, \mathcal{A}, \lambda) = 1]$ is $\mathcal{A}$'s advantage in the $\mathsf{TR\text{-}NM}$ experiment.

Below, we adapt [BSCG+14, Appendix C.2] to our specific DAP—`Zeth`.

We start by describing the $\mathsf{TR\text{-}NM}$ experiment. Given a (candidate) `Zeth` DAP, adversary $\mathcal{A}$, and security parameter λ, the (probabilistic) game $\mathsf{TR\text{-}NM}(\mathsf{DAP}, \mathcal{A}, \lambda)$ consists of an interaction between $\mathcal{A}$ and a challenger $\mathcal{C}$, terminating with a binary output by $\mathcal{C}$.

At the beginning of the game, $\mathcal{C}$ samples $pp \leftarrow \mathsf{Setup}(\lambda)$ and sends pp to $\mathcal{A}$. Next, $\mathcal{C}$ initializes a DAP oracle $\mathsf{O}^{\mathsf{DAP}}$ with pp and allows $\mathcal{A}$ to issue queries to it [RZ19, Appendix B].

At the end of the experiment, $\mathcal{A}$ sends to $\mathcal{C}$ a $\widetilde{\textbf{Mixer}}$ contract call transaction tx^*_{Mix}, and $\mathcal{C}$ outputs 1 iff the following conditions hold. Letting T be the set of transactions generated by $\mathsf{O}^{\mathsf{DAP}}$ in response to SendTx queries, there exists $tx_{\mathsf{Mix}} \in T$ such that:

1. tx_{Mix} was not inserted in L by $\mathcal{A}$;
2. $tx^*_{\mathsf{Mix}}.data \neq tx_{\mathsf{Mix}}.data$;
3. $\mathsf{VerifyTx}(pp, tx^*_{\mathsf{Mix}}, L') = 1$ where L' is the portion of the ledger L preceding tx_{Mix};
4. a serial number revealed in tx^*_{Mix} is also revealed in tx_{Mix}.

A.1 Transaction malleability attack on Zeth

In this section we present the threat related to the transaction malleability attack on `Zeth` and expose the solutions by `ZeroCash` [BSCG+14] and `Zcash` [ZCa19] that we adapted.

First, we start by assuming that none of the checks related to transaction malleability attack have been added in the protocol Chapter 2. As such, we assume that *hsig* and *htags* are not attributes of `PrimInputDType`, ϕ is not an attribute of `AuxInputDType`, and *otssig* and *otsvk* are not attributes of the `MixInputDType` data type anymore. As a consequence, all checks related to these attributes are removed from the protocol. Moreover, if zn is an object of type `ZethNoteDType`, then $zn.\rho$ is chosen at random. Finally, the NP-relation used in `Zeth`, now denoted $\mathbf{R}^{\mathtt{mal}}$, becomes the following:

- For each $i \in [\mathtt{JSIN}]$:
  1. $aux.jsins[i].znote.apk = \mathsf{Blake2s}(tag^{addr}_{ask} \| \mathsf{pad}_{\mathtt{BLAKE2sCLEN}}(0))$
     with tag^{addr}_{ask} defined in Section 3.1.3
  2. $aux.jsins[i].nf = \mathsf{Blake2s}(tag^{nf}_{ask} \| aux.jsins[i].znote.\rho)$
     with tag^{nf}_{ask} defined in Section 3.1.3
  3. $aux.jsins[i].cm = \mathsf{Blake2s}(aux.jsins[i].znote.r \| m)$
     with $m = aux.jsins[i].znote.apk \| aux.jsins[i].znote.\rho \| aux.jsins[i].znote.v$
  4. $(aux.jsins[i].znote.v) \cdot (1 - e) = 0$ is satisfied for the boolean value e set such that if $aux.jsins[i].znote.v > 0$ then $e = 1$.
  5. The Merkle root $mkroot'$ obtained after checking the Merkle authentication path $aux.jsins[i].mkpath$ of commitment $aux.jsins[i].cm$, with MKHASH, is equal to $prim.mkroot$ if $e = 1$.
  6. $prim.nfs[i]$
     $= \{\mathsf{Pack}_{\mathbb{F}_r}(aux.jsins[i].nf[k \cdot \mathtt{FIELDCAP}{:}(k+1) \cdot \mathtt{FIELDCAP}])\}_{k \in [\lfloor \mathtt{PRFNFOUTLEN}/\mathtt{FIELDCAP} \rfloor]}$
- For each $j \in [\mathtt{JSOUT}]$:
  1. $prim.cms[j] = \mathsf{Blake2s}(aux.znotes[j].r \| m)$
     with $m = aux.znotes[j].apk \| aux.znotes[j].\rho \| aux.znotes[j].v$
- $prim.rsd = \mathsf{Pack}_{rsd}(\{aux.jsins[i].nf\}_{i \in [\mathtt{JSIN}]}, aux.vin, aux.vout)$
- Check that the "joinsplit is balanced", i.e. check that the joinsplit equation holds:

$$\mathsf{Pack}_{\mathbb{F}_r}(aux.vin) + \sum_{i \in [\mathtt{JSIN}]} \mathsf{Pack}_{\mathbb{F}_r}(aux.jsins[i].znote.v)$$
$$= \sum_{j \in [\mathtt{JSOUT}]} \mathsf{Pack}_{\mathbb{F}_r}(aux.znotes[j].v) + \mathsf{Pack}_{\mathbb{F}_r}(aux.vout)$$

A.1.1 The attack

In order to win the game TR-NM on the weak `Zeth` DAP above, an adversary $\mathcal{A}$ intercepts a target transaction tx_{Mix} by passively listening to the network (remember that transactions are broadcasted to the `Ethereum` network in order to be mined, see Section 1.2.2), extracts the zk-proof and primary inputs from $tx_{\mathsf{Mix}}.data$ and uses these extracted pieces of information in order to create a malicious transaction ${tx_{\mathsf{Mix}}}'$, where the ciphertexts are replaced by arbitrary data. The adversary can then broadcast ${tx_{\mathsf{Mix}}}'$ to the network in order for it to be mined. If the malicious transaction gets mined before the legitimate one, the input notes become spent and the ciphertexts are undecryptable making the new notes unredeemable (by any `Zeth` user!), since all attempts to decrypt the ciphertexts will fail (see Section 2.6).

$\mathsf{TxMalGen}(sk'_{\mathsf{ECDSA}}, nce_{in}, tx_{\mathsf{Mix}})$

$$p \leftarrow tx_{\mathsf{Mix}}.gasP + 1$$
$$l \leftarrow tx_{\mathsf{Mix}}.gasL + 1$$
$$zdata' \leftarrow tx_{\mathsf{Mix}}.data$$
$$zdata'.ciphers \leftarrow\!\$\, \mathbb{B}^*$$
$$tx_{raw} \leftarrow \{nce : nce_{in}, gasP : p, gasL : l, to : tx_{\mathsf{Mix}}.to, val : tx_{\mathsf{Mix}}.val, data : zdata'\};$$
$$\sigma_{\mathsf{ECDSA}} \leftarrow \mathsf{SigSch}_{\mathsf{ECDSA}}.\mathsf{Sig}(sk'_{\mathsf{ECDSA}}, \mathsf{Keccak256}(tx_{raw}));$$
$$tx_{final} \leftarrow \{tx_{raw}, v : \sigma_{\mathsf{ECDSA}}.v', r : \sigma_{\mathsf{ECDSA}}.r', s : \sigma_{\mathsf{ECDSA}}.s'\};$$
$$\textbf{return}\ \ tx_{final};$$

Figure A.1: Transaction malleability attack function TxMalGen

As shown on Fig. A.1, during the attack, the adversary extracts the proof and primary inputs from the honest transaction, and replaces the ciphertexts by some arbitrary information. The attacker then formats this data into a transaction that calls the Mix function of **Mixer**, and submits it to the network. While the data fields ($tx_{\mathsf{Mix}}.data$ and ${tx_{\mathsf{Mix}}}'.data$) are different, the nullifiers revealed by both transactions are the same (i.e. $tx_{\mathsf{Mix}}.data.proof = {tx_{\mathsf{Mix}}}'.data.proof$, and $tx_{\mathsf{Mix}}.data.prim = {tx_{\mathsf{Mix}}}'.data.prim$). As a consequence, if the adversary makes sure that ${tx_{\mathsf{Mix}}}'$ satisfies all the checks of EthVerifyTx (Section 1.2.2), he can ensure that $\mathsf{ZethVerifyTx}({tx_{\mathsf{Mix}}}')$ will return the same value as $\mathsf{ZethVerifyTx}(tx_{\mathsf{Mix}})$. Furthermore, if ${tx_{\mathsf{Mix}}}'.gasP > tx_{\mathsf{Mix}}.gasP$, then the adversary maximizes his chances of having his transaction mined first (Section 1.2.2), and so maximizes the chances for the malleability attack to be successful; leading to lost funds on **Mixer**.

Remark A.1.1. Note that, although not directly contained within the *data* field of a **Mixer** call transaction, the `Ethereum` address $\mathcal{S}_{\mathcal{E}}.Addr$ of the transaction sender is also used by the **Mixer** call (this is either the calling contract's address, or the transaction signer's address recovered as described in Remark 1.2.1). In particular, the balance of this `Ethereum` address is incremented by the value *vout* by successful Mix calls. If we again assume the absence of the malleability checks, an attacker could re-sign any **Mixer** call transaction with a key under his control, rebroadcast it as described above, and (with some reasonable probability) become the recipient of any public output value $vout$.

Remark A.1.2. We note that the attack described above cannot be prevented by merely substituting a malleable $\mathsf{Groth16}$ zk-SNARK by a simulation-extractable one like e.g. [GM17]. This comes since the attack does not utilise malleability of the proof system, but malleability of data that are broadcasted along with the zk-proof.

A.2 Solutions to address the transaction malleability attack

A.2.1 `ZeroCash` solution

The idea of the solution presented in [BSCG+14] is to use a one-time $\mathsf{SUF\text{-}CMA}$ digital signature and bind its verification key with the zk-proof primary inputs to prevent an adversary from corrupting part of a transaction's data.

Specifically, to transact via `Zeth`, the user first samples a key pair (sk, vk) for a one-time signature scheme. He then computes the hash $hsig = \mathsf{CRH}(vk)$, where CRH is a collision resistant hash function, see [BSCG+14], and derives a value $h_i = \mathsf{PRF}^{\mathsf{pk}}_{ask_i}(hsig)$, for each input note (i.e. $i \in [\mathtt{JSIN}]$), which acts as a MAC binding $hsig$ to the address spending key of a note (ask_i).

The user then generates the zk-proof with the additional statement that the values $\{h_i\}_{i\in[\mathtt{JSIN}]}$ are computed correctly. He finally uses sk to sign every value associated with the operation, thus obtaining a signature, which is included, along with the signature verification key vk, in the transaction. To verify a transaction on the DAP, it is necessary to verify that

- the primary inputs are correctly formatted,
- the Merkle root corresponds to one of the previous states of the Merkle tree,
- the nullifiers have not been declared in a previous transaction,
- the $hsig$ is correctly computed from vk, and
- both the zk-proof and the one-time signature verifications pass successfully.

Now, an adversary trying to carry out the aforementioned attack has to either change the ciphertexts or the encryption key. Nevertheless, doing so should lead to the one-time signature verification to fail or should yield an attack that breaks the $\mathsf{UF\text{-}CMA}$ property of the one-time signature (as this corresponds to creating a forgery on a different message, not changing the signature). Thereby, the adversary also has to modify the signature, however he does not know the one-time signing key used by the creator of the targeted transaction. As such, the adversary needs to use another signing key pair, however this leads to the check verifying that *hsig* is correctly computed to fail. If the adversary attempts to change *hsig*, the zk-proof verification fails as the NP-statement has changed. Hence, any attempt to carry out a malleability attack results in the violation of at least one check in the verification of the transaction on the DAP. The solution presented effectively solves the transaction-malleability attack initially described.

Remark A.2.1. The one-timeness property of the signature scheme was required in `ZeroCash` to retain anonymity. It also makes analysing non-adaptive adversary sufficient. As `Ethereum` transaction senders need to pay the gas cost associated with their transactions, the senders are not anonymous. This said, making sure that `Zeth` is designed with anonymity in mind is worth the effort in order to minimize information leakages and be ready if/when `Ethereum` incorporates protocol changes that enable anonymous transactions.

A.2.2 `Zcash`'s solution

In addition to the changes aforementioned, `Zcash`'s solution [ZCa19] also consists of:

- Redefining the variable *hsig* as,

  $$hsig = \mathsf{CRH}(randomSeed, \{nf_i\}_{i \in [\mathtt{JSIN}]}, vk)$$

  for some random seed *randomSeed*.

- Defining a new random variable ϕ and using it with *hsig*, as key and input of a PRF respectively, to compute the identifier of each output notes ρ_j ($j \in [\mathtt{JSOUT}]$) and ensure their uniqueness (with overwhelming probability).

These changes were made to prevent the Faerie Gold attack [ZCa19, Section 8.4], as well as to prevent linkability: if *hsig* were repeated in two transactions, the circuit would leak, via $\{h_i\}_{i \in [\mathtt{JSIN}]}$, the fact that the input notes in both transactions were spent with the same ask_i (if that were the case).

More particularly, using the input notes' nullifiers to derive *hsig* ensures that *hsig* is unique with overwhelming probability for all *accepted* transaction. Furthermore, using *randomSeed* ensures the uniqueness of *hsig* for transactions *in transit* (as before validation there may be several in transit transactions with the same set of nullifiers).

A.2.3 Solution on `Ethereum`

As described in the `Ethereum` yellow paper [Woo19, Appendix F], `Ethereum` transactions are ECDSA signed. Further, as described in Section 2.3, the one-time signature used to sign the Mix data also signs the `Ethereum` address used to sign the transaction. As such, any modification to the transaction object will result in a new transaction hash, and any attempt to sign the transaction with a different ECDSA key will be rejected by the **Mixer** contract (see Section 2.5). We thereby conclude that the one-time signature used to sign the transaction data does not need to be SUF-CMA, but *only needs to achieve* UF-CMA.

Specifically, carrying out any change on the one-time signature will change the `Ethereum` transaction data and result in a failure to verify the ECDSA signature of the `Ethereum` transaction. To obtain a new valid signature on this transaction, the adversary needs to break the UF-CMA property of the ECDSA signature scheme or use another ECDSA keypair to sign the transaction. In the last case, the one-time signature will no longer be valid.

Note that including the `Ethereum` transaction sender in the data to be signed by the one-time signature scheme also addresses the possible attack described in Remark A.1.1. An attacker trying to resign the same `Ethereum` transaction with a different key will cause **Mixer** to reject the transaction when the one-time signature is checked.

Remark A.2.2. We note that the transaction malleability issue can also be addressed in another way. In fact, one could use the ECDSA signatures on `Ethereum` transactions to fix all inputs and ciphertexts, and then tie the sender of the `Ethereum` transaction to the zk-snark by putting the sender address $\mathcal{S}_{\mathcal{E}}.Addr$ in *hsig*. In other words, it is also possible to define *hsig* as:

$$hsig = \mathsf{CRH}(\{nf_i\}_{i \in [\mathtt{JSIN}]}, \mathcal{S}_{\mathcal{E}}.Addr)$$

As such, if an attacker extracts the ciphertexts of a tx_{Mix} transaction in order to craft another malicious transaction tx_{Mix}', the key-pair used to sign tx_{Mix}' differs from the one used to sign tx_{Mix}, which changes the transaction sender address recovered on **Mixer**. As a consequence, the check on *hsig* would fail on the **Mixer**, invalidating the transaction, and preventing the attack.

While such a solution would avoid the need to generate one-time signing keys and could avoid a signature check in the **Mixer**, it would also require every `Zeth` user to have an `Ethereum` account. Doing so, would be a major hindrance toward the design of mechanisms aiming to provide anonymity to `Zeth` transactions initiators. In fact, the addressing scheme used in `Zeth` along with the solution to the malleability introduced in `Zcash` makes it possible to generate raw `Zeth` transactions without having an `Ethereum` account. These raw transactions could then be broadcasted – to a set of `Ethereum` user nodes – on an anonymous p2p network, before being finalized and submitted to the `Ethereum` network by `Ethereum` users who would be rewarded according to an incentive structure. While such a protocol is outside of the scope of this document, it shows that defining *hsig* using the senders address alters the flexibility of `Zeth`; hence this solution has not been favoured.

Appendix B

Double spend attack on equivalent class

The primary inputs of our zk-SNARK are elements of $\mathbb{F}_{\mathtt{r}}$ and they can be written over `FIELDLEN` bits. Note that the projection of $\mathbb{B}^{\mathtt{FIELDLEN}}$ onto $\mathbb{F}_{\mathtt{r}}$ formed by interpreting elements in $\mathbb{B}^{\mathtt{FIELDLEN}}$ as `FIELDLEN`-bit numbers and reducing modulo $\mathtt{r}$, is surjective.

When we pass the primary inputs to the **Mixer** contract, they are interpreted as elements of $\mathbb{B}^{\mathtt{ETHWORDLEN}}$, and $\mathbb{B}^{\mathtt{FIELDLEN}} \subset \mathbb{B}^{\mathtt{ETHWORDLEN}}$. As previously noted, this means that there exist pairs of elements in $\mathbb{B}^{\mathtt{ETHWORDLEN}}$ with the same projection in $\mathbb{F}_{\mathtt{r}}$. An adversary could make use of this to perform a double spend attack.

Indeed, to check that a note is not double spent, the contract stores the nullifiers of spent notes (as elements of $\mathbb{B}^{\mathtt{ETHWORDLEN}}$) and verifies that the nullifier of the note to be spent is not stored. The adversary could thus modify the nullifier to a different value with the same projection. As the SNARK verification operates in $\mathbb{F}_{\mathtt{r}}$, the proof would still be valid. However, the value stored for this nullifier would be different from the adversarial one. Hence, the nullifier would be validated, the transaction would succeed and the note would be double spent. In practice, the adversary can perform the attack by simply adding $\mathtt{r}$ to one of the elements representing the nullifier.

To prevent this attack, the contract checks that all primary inputs are elements of $\mathbb{F}_{\mathtt{r}}$, that is to say that they are smaller than $\mathtt{r}$. As one may see, the attack described above is not due to the packing of hash digests into field elements but to the contract storage of field elements as `Ethereum` words.

Appendix C

Side-channel attacks and information leaks

The following subsections describe several side-channel attacks and possible weaknesses that implementers should be aware of and attempt to mitigate.

We consider cases in which the attacker is able to observe the RPC communications between `Zeth` client software, and Ethereum P2P nodes. This situation may occur if an observer is able to monitor the network traffic between the Ethereum node and the `Zeth` client software, or if the Ethereum node itself is compromised.

Note

In this discussion, we do not consider adversaries with physical access to the machine running the client software. Such adversaries could make precise measurements of timing, power-consumption or other physical quantities that could reveal fine-grained details of the operations being carried out by the software, or the data it is operating on. Protecting against attacks of this kind often involves implementation techniques such as: avoiding branches based on private data, being careful with memory access patterns, and making all operations constant time, to only name a few. We leave consideration of these attacks and prevention methods for a future discussion.

C.1 Counterfeit data

Malicious Ethereum nodes or attackers able to compromise the network have the opportunity to send invalid data to RPC clients. This could be used to inject invalid data into the client's record of state, which could prevent it from generating valid Mix calls or allow it to be identified in the future. In general, data from any remote host should be treated as malicious, unless accompanied by evidence that convinces the client of its authenticity.

In the case of Ethereum event logs (the main source of data used to track the on-chain state – see Section 4.1.1 for details), clients `MUST` leverage the consensus evidence and block headers to verify that log data is genuine and has been committed to the blockchain. See Section 1.2.3 for further information about how such data is secured.

C.2 Data leaked during synchronization

In order to receive private payments and keep their local data up-to-date, `Zeth` client software `MUST` scan the blockchain and process *all* the event data emitted by $\widetilde{\mathbf{Mixer}}$ during Mix calls (as described in Section 4.1.1). There are several issues to consider when determining exactly how and when this "synchronization" takes place.

Client implementations that only connect to the RPC endpoint in response to user input, or in preparation for performing a Mix call, may leak information. Observers may deduce that such client are likely to be the recipient of a recent or upcoming transaction, or that they may be about to perform a Mix call.

Similarly, payment provider software that only listens for events when awaiting a transaction, and remains disconnected otherwise, may reveal that it is the recipient of an upcoming transaction, and possibly *which* transaction or block it was paid by (based on when it stops listening).

Further, consider wallet software that performs RPC operations to explicitly wait for the Ethereum transaction corresponding to a specific Mix call. This would most likely be for transactions emitted by the `Zeth` client, in order to inform the user and update the wallet state once the payment is complete (but could possibly happen on the receiver side, if he somehow knows the ID of the transaction of interest – e.g. via off-chain communication with the sender). If such a *wait* procedure is implemented by querying the status of a specific transaction by its ID, or by listening for blocks *until* the transaction of interest is received, the connected Ethereum node may infer that this client is interested in the transaction, and likely to be the sender or recipient.

Consider a client which periodically connects to some Ethereum node and requests all relevant data from the last block it saw, up to the latest block available. Each client will have information up to some block n (where n varies per client), and n is known to the Ethereum node that served the client. The client could then potentially be identified by n (even if it hides its IP for each connection) since a client that connects and queries `Zeth` transactions from block $n+1$ reveals that it is one of the clients who synced up to block n when it last connected.

Note that, if the client always broadcasts the Mix transaction via this same Ethereum node, then the Ethereum node may already deduce that the client is the sender. However, implementations may wish to use techniques (such as sending transactions from other nodes or hiding their IP address in other way) to obfuscate any relationship between transactions and the clients that originated them.

C.3 Queries on successful decryption

The event data emitted by **Mixer** contains the note data for new commitments, encrypted using a key derived from the recipients' public key. As described in Section 2.6, clients scan the blockchain for these events and attempt to decrypt the ciphertext using their secret decryption keys. If they are successful, they are the recipient of the note and can try to parse the plaintext to extract the secret note data.

When decryption is successful and the note data has been extracted from the plaintext (we discuss parsing failure in Appendix C.4), clients `MUST` check that this note data does indeed open the commitment for the note.

A naive implementation of this check could query the state of **Mixer** via RPC to check the relevant entry in the set of commitments. However, this would reveal to an observer that the client had successfully decrypted and parsed the corresponding ciphertext, and was therefore the recipient of that note.

For this reason, the protocol specifies that **Mixer** `MUST` emit events informing clients of new commitment values and locations in the Merkle tree. Clients `MUST` consume *all* such data to maintain their view of contract state (as described in Appendix C.2). Further, clients `MUST` attempt to decrypt *all* ciphertexts and, for successful decryptions, `MUST` verify that the plaintext opens the note's commitment. This avoids the need for any extra RPC queries that would reveal which ciphertexts were successfully decrypted.

> **Note**
>
> Emitting events containing all data necessary to carry out the local checks implemented in the wallet is a way to enforce that all wallets behave exactly the same to the eyes of network (passive) adversaries (regardless whether the user is the recipient of a note or not).

C.4 Invalid ciphertext

The attack described in [TBP20, Section 4.2.1] illustrates the importance of correctly handling invalid data in client software. A so-called "REJECT Attack" is described whereby an attacker creates a Mix call with specially crafted ciphertext. The ciphertext can be successfully decrypted by the correct recipient – that is, the plaintext note is encrypted with an encryption key derived from the recipients public key – but the corresponding plaintext is invalid and cannot be parsed correctly by the recipient.

> **Note**
>
> Note that the above is possible because the plaintext is neither verified by the circuit encoding $\mathbf{R}^{\mathsf{z}}$, nor by the contract (which is unable to decrypt it). Hence, `Zeth` allows such transactions with malicious ciphertexts to be accepted by the **Mixer** contract, and clients must handle this case with care.

In the case described in [TBP20], there is no distinction between "client" or "wallet" software, and the underlying P2P nodes. Before a fix was applied (see [zcab]), nodes explicitly rejected transactions of the above form, proving to their peers that they were able to decrypt the ciphertext and were therefore the intended recipient.

In `Zeth`, P2P nodes and wallet software are separated, so there will be no explicit rejection of the transaction. However, careless error handling (such as exceptions which causes the RPC connection to be closed) could potentially be detected by the connected Ethereum node. As in the "REJECT Attack", this reveals that the connected RPC client is the intended recipient of a transaction, and the owner of the corresponding encryption key.

C.5 Using (and retrieving) nullifiers

Any non-trivial wallet implementation will need to track which of the user's `Zeth` notes have been spent, and which are still available. Naturally, the wallet software could mark the notes as it broadcasts transactions that spend them. However, this approach is subject to several problems.

Firstly, for each note spent, the client software must record the ID of the spending transaction, in order to track it and confirm that it is accepted into a block. Once each spending transaction is accepted the client can finally mark the appropriate `Zeth` notes as "spent". This requires significant complexity in order to asynchronously mark the notes, and to deal with the issues described in Appendix C.2.

Secondly, this approach does not support multiple wallets using the same key, or wallets being restored from `Zeth` addresses. A user that wishes to rebuild his wallet (see the discussion in Section 4.1.4), or check for any spending activity by other wallets, would not be able to do so by simply scanning the blockchain.

By using the nullifiers passed to Mix calls, clients can determine the availability of notes in a more robust way. That is, to determine whether a note is spent or available, the client can compute the nullifier and check whether that nullifier has been seen by the **Mixer** contract.

In a similar way to Appendix C.3, queries to **Mixer** for specific nullifiers reveals to observers that the client was the sender of any previous or future transaction that generates such a nullifier. To mitigate this, **Mixer** `MUST` include nullifier values in the event data it emits, and clients `SHOULD` use this to track which of their notes are spent. This `MUST` happen as part of the regular sync operation, so that no extra RPC traffic is generated and observers cannot distinguish between clients that do and do not recognize any given nullifier. Note that this approach also supports tracking spent notes from multiple wallets, and rebuilding wallets by re-syncing the blockchain.

C.6 Proof generation

Generation of the zero-knowledge proofs, required for valid Mix calls, is a very computationally intensive process. The proof generation itself does not require any communication with external parties, and so may not directly leak information about the client, but implementers should consider some indirect ways in which information may be leaked.

Implementers may also wish to consider the possible indirect impact of proof generation on the RPC channel. For example, a client that "waits" for proof generation without servicing the RPC connection may fail to respond to (or take significantly longer to respond to) new log events. The connected Ethereum node might then deduce that its peer is generating a proof and therefore likely to be the sender of an upcoming transaction.

Note

As stated in the introduction to this chapter, this discussion does not consider general timing attacks. We mention this extreme case of a client that completely stalls during proof generation only to illustrate how a poor implementation may leak information to its RPC peer.

In the case where proof generation is carried out on some external host, or by an external process on the same host, there may be a risk of network traffic or other IPC traffic being observed. If an observer can detect that a given client is communicating with a prover process, it can reliably deduce that the client will be the sender of an upcoming transaction.

An observer able to see the content of the communication between the wallet and prover process will also gain knowledge of the auxiliary inputs to the proof (including the data required to spend the input notes and secret attributes of the output notes). It is therefore important to secure any such connection, protect any prover process from being maliciously modified or observed, and to ensure that wallets only communicate with trusted processes.

C.7 Simple mixer calls

The public parameters to a Mix call can reveal information about the nature of a transaction, even though they do not reveal recipient details or note amounts. For example, a Mix call in which $\mathsf{Mix}_{in}.primIn.vout = 0$ and $\mathsf{Mix}_{in}.primIn.vin \neq 0$ may indicate a simple "deposit" of funds into the mixer. Similarly, if both $\mathsf{Mix}_{in}.primIn.vout$ and $\mathsf{Mix}_{in}.primIn.vin$ are zero, the transaction must be spending only notes already within **Mixer**, into new notes. Finally, if $\mathsf{Mix}_{in}.primIn.vin = 0$ and $\mathsf{Mix}_{in}.primIn.vout \neq 0$, the sender may be performing a simple "withdrawal" of funds from some existing notes.

A Mix call can combine all of the above logical operations in a single transaction. That is, it can deposit value into the mixer, spend existing notes, create new notes, and withdraw value from **Mixer** *at the same time*. Combining logical operations in this way makes it much more difficult for an observer to attribute a specific purpose to the Mix call.

Clients can also perform Mix calls in which $vin = vout = 0$ and 0-valued notes are created from other 0-valued notes. Such "dummy" self-payments can further obfuscate the activity of a wallet, by adding "noise" to the system. Note, however, that the gas cost for such transactions must still be paid.

Wallet implementations `SHOULD` encourage the use of these complex calls where possible, either via the user interface or by automatically adding complexity to transactions, and `SHOULD` support features such as adding "noise"[1] if the user wishes to pay for extra protection of this kind.

C.7.1 Small anonymity sets

Until there is a large number of commitments and users of the mixer, it may be easy for an observer to infer some of the private data that is intended to be hidden by mixer calls.

In the simple case, if there are very few commitments in the $\widetilde{\textbf{Mixer}}$'s Merkle tree, an attacker has a small list of candidate commitments that are being spent by subsequent Mix calls. Similarly, if the number of distinct Ethereum addresses that have been used to call $\widetilde{\textbf{Mixer}}$ is very small, observers can trace the original source of funds subsequently withdrawn to a small set of original depositors.

Client software may wish to track metrics about the $\widetilde{\textbf{Mixer}}$ state, and either prevent certain actions or design the user interface to discourage users[2] from creating transactions whose features can be identified with high probability. We provide below a non-exhaustive list of metrics of interest:

- **Number of commitments.** If there is a low absolute number of commitments, clearly any non-zero output must spend one of these (although we note that only *vout* can be publicly known to be non-zero).
- **Number of unspent commitments.** If $\#Comms - \#Nulls$ is small and a new commitment is created and then spent, observers can deduce that there is a high chance that the spend operation targeted the new commitment.
- **Number of Ethereum addresses.** While very few distinct addresses (or groups of addresses that are not associated) have used the contract, observers can deduce that subsequent Mix calls are likely to spend commitments created by clients associated with one of this small set of addresses.

The set of Ethereum addresses that have interacted with the contract can leak data in other ways. An Ethereum address that withdraws value from the contract, but has not previously been used to make a Mix call (or a Mix call that deposits value into $\widetilde{\textbf{Mixer}}$),

[1] By randomly scheduling dummy payments, for instance

[2] By, for example, displaying warning messages and/or asking the user for confirmation

must have been the recipient of zeth notes created by a previous depositor. The details may not be directly available to an observer, but this is another example of information which could be combined with other leaked data to infer connections between entities and transactions.

Appendix D

Security proofs of Blake2

This appendix proves the collision resistance, PRF-ness, binding and hiding properties of the Blake2 hash function in the Weakly Ideal Cipher model (WICM, see [LMN16]). The proofs use definitions and results of Luykx et al. [LMN16], regarding the indifferentiability of Blake2 and a random oracle in the Weakly Ideal Cipher Model (WICM). In the following, we assume that the optimization of Blake2 for 8- to 32-bit platforms is as secure as Blake2 as described in [LMN16].

D.1 Security model of Blake2

The security analysis treats Blake2 as hash function built on top of a block-cipher-based compression function in the WICM (which derives from the Ideal Cipher Model). In this section, we present the WICM and prove that Blake2 is a collision resistant PRF, and thus a commitment scheme.

D.1.1 Weakly Ideal Cipher Model

The research community believes that Blake's underlying block cipher has no known weaknesses and could reasonably be modeled as an ideal cipher [LMN16, Section 2.1]. However, Blake2 admits weak keys with a specific structure [LMN16, Section 2.1]. Blake2 is therefore more appropriately analysed in the WICM, which is an extension of the Ideal Cipher Model that represents a block cipher as a set of independent random permutations [HKT11]. The WICM may also be viewed as a specialization for Blake2 of the Weak Cipher Model [MP15], which aims to be realistic by modeling particular characteristics, invariants or properties a block cipher may have.

A number of definitions in what follows are quoted directly from Luykx et al. [LMN16].

The Weakly Ideal Cipher Model. Let $\mathcal{W}$ and $\mathcal{S}$ be the following partition of $\mathbb{B}^{2\cdot\mathsf{ol}}$ into weak and strong sets, where w is the word length ($16 \cdot w = 2 \cdot \mathsf{ol}$):

$$\mathcal{W} = \left\{ aaaabbbbccccdddd \in \mathbb{B}^{2\cdot\mathsf{ol}} \mid a, b, c, d \in \mathbb{B}^{w} \right\}$$

$$\mathcal{S} = \mathbb{B}^{2\cdot\mathsf{ol}} \setminus \mathcal{W}$$

Let $\mathcal{BLK}(2\cdot\mathsf{ol}, 2\cdot\mathsf{ol})$ denote the set of all block ciphers $\mathsf{E} : \mathbb{B}^{2\cdot\mathsf{ol}} \times \mathbb{B}^{2\cdot\mathsf{ol}} \to \mathbb{B}^{2\cdot\mathsf{ol}}$. Define $\mathcal{BLK}^*(2\cdot\mathsf{ol}, 2\cdot\mathsf{ol})$ as the set of all block ciphers $\mathsf{E} \in \mathcal{BLK}(2\cdot\mathsf{ol}, 2\cdot\mathsf{ol})$ with the additional restriction that $\mathsf{E}(k_w, \cdot)$ is $\mathcal{W}$- and $\mathcal{S}$-subspace invariant for all keys $k_w \in \mathcal{K}_{weak}$. That is, inputs in $\mathcal{W}$ map to $\mathcal{W}$, and likewise for $\mathcal{S}$. Here, $\mathcal{K}_{weak}$ is the set of weak keys, defined as

$$\mathcal{K}_{weak} = \left\{ k = kkkkkkkkkkkkkkkk \in \mathbb{B}^{2\cdot\mathsf{ol}} \mid k \in \mathbb{B}^{w} \right\}.$$

A random $\mathsf{E} \leftarrow\!\!\$\, \mathcal{BLK}^*(2\cdot\mathsf{ol}, 2\cdot\mathsf{ol})$ can now be modeled as follows:

- on input of $(k, x) \in \mathcal{K}_{weak} \times \mathcal{W}$, E generates its response y randomly from $\mathcal{W}$ up to repetition;

- on input of $(k, x) \in \mathcal{K}_{weak} \times \mathcal{S}$, E generates its response y randomly from $\mathcal{S}$ up to repetition.

For key values $k \in \mathbb{B}^{2\cdot\mathsf{ol}} \setminus \mathcal{K}_{weak}$, E behaves like an ideal cipher: it either outputs a new random value or if the key-message-image tuple has already been queried the tuple's image. The case of inverse queries is analogous.

$\mathsf{Blake2C}$ is defined over the following domains and codomain:

$$\mathsf{Blake2C} : \mathcal{BLK}^*(2\cdot\mathsf{ol}, 2\cdot\mathsf{ol}) \times \mathbb{B}^{\mathsf{ol}} \times \mathbb{B}^{2\cdot\mathsf{ol}} \times \mathbb{B}^{\mathsf{ol}/4} \times \mathbb{B}^{\mathsf{ol}/4} \to \mathbb{B}^{\mathsf{ol}}$$

We write $\mathsf{Blake2C}_{\mathsf{E}}(h, m, t, f)$ for the output of the $\mathsf{Blake2}$ compression function, defined over encryption scheme E on inputs h, m, t and f. The compression function, in particular, computes the state $x = (h \| \mathsf{pad}_{\mathsf{ol}/2}(0) \| t \| f) \oplus (\mathsf{pad}_{\mathsf{ol}}(0) \| \mathtt{IV})$ for some $\mathtt{IV}$. It then encrypts x under m (where m is treated as a key for the encryption) and splits $\mathsf{E}(m, x)$ in two same size variables, the left part l_{E} and right part r_{E}. It finally outputs $l_{\mathsf{E}} \oplus r_{\mathsf{E}} \oplus h$.

`Zeth` uses the $\mathsf{Blake2}$ compression function with a fixed encryption scheme E^* based on ChaCha stream cipher [Ber08a]. Thus, we write $\mathsf{Blake2C}(h, m, t, f) = \mathsf{Blake2C}_{\mathsf{E}^*}(h, m, t, f)$.

Indifferentiability. One way to measure the extent to which a certain cryptographic function behaves like a random function is via the indistinguishability framework where a distinguisher is given oracle access to either the cryptographic function or the random function with the goal of determining which one it has access to.

Definition D.1.1. Let $\mathcal{C}$ be a construction with oracle access to an ideal primitive $\mathcal{P}$. Let $\mathcal{R}$ be an ideal primitive with the same domain and codomain as $\mathcal{C}$. Let Sim be a simulator with the same domain and codomain as $\mathcal{P}$ with oracle access to $\mathcal{R}$, and let Dist be a PPT distinguisher. The indifferentiability advantage of Dist is defined as:

$$\mathsf{Indiff}_{\mathcal{C}^{\mathcal{P}}, \mathsf{Sim}}(\mathsf{Dist}) = \left| \Pr\left[\mathsf{Dist}^{\mathcal{C}^{\mathcal{P}}, \mathcal{P}} = 1 \right] - \Pr\left[\mathsf{Dist}^{\mathcal{R}, \mathsf{Sim}^{\mathcal{R}}} = 1 \right] \right|$$

The distinguisher Dist can query both its left oracle (either $\mathcal{C}$ or $\mathcal{R}$) and its right oracle (either $\mathcal{P}$ or Sim). We refer to $\mathcal{C}^{\mathcal{P}}$, $\mathcal{P}$ as the real world, and to $\mathcal{R}$, $\mathsf{Sim}^{\mathcal{R}}$ as the simulated world; the distinguisher Dist converses with either of these worlds and its goal is to tell both worlds apart.

Theorem D.1.1 (Indifferentiability of $\mathsf{Blake2}$ [LMN16])**.** *Let an encryption scheme* $\mathsf{E} \leftarrow\$\, \mathcal{BLK}^*(2 \cdot \mathsf{ol}, 2 \cdot \mathsf{ol})$ *be a weakly ideal cipher, and consider the hash function* $\mathsf{Blake2}_{\mathsf{E}}$ *that internally uses* E*. There exists a simulator* Sim *such that for any distinguisher* Dist *with total complexity* q*, we have:*

$$\mathsf{Indiff}_{\mathsf{Blake2}_{\mathsf{E}},\mathsf{Sim}}(\mathsf{Dist}) \leq \frac{\binom{q}{2}}{2^{2\mathsf{ol}}} + \frac{2\binom{q}{2}}{2^{\mathsf{ol}}} + \frac{q}{2^{\mathsf{ol}/2}}$$

where Sim *makes at most* $O(q^3)$ *queries to a random function* $\mathcal{R}$*.*

Proof. See [LMN16, Corollary 1]. □

For asymptotic security, we assume the distinguisher to be PPT and that the number of queries made is polynomial $q \leq \mathsf{poly}(\mathsf{ol})$.

Additional remarks. Luykx et al. [LMN16] remark that, by resorting to the WICM, they do not make stronger assumptions than those used in previous results (ICM), and, despite the fact that they give distinguishers more power (by weakening the cipher), they are able to get similar results.

D.2 Security proofs

D.2.1 Blake2 is a PRF

Luykx et al. already prove the PRFness of $\mathsf{Blake2}$ *keyed* hash function in the multi-key setting.

Definition D.2.1 (PRF in multi-key setting [ML15])**.** Let $\mu \geq 1$ and $k \leftarrow\$\, \mathcal{K}^{\mu}$. Let $\mathcal{C}$ be a keyed construction with key space $\mathcal{K}$ and with oracle access to an ideal primitive $\mathcal{P}$. Let $\mathcal{R}_1, \ldots, \mathcal{R}_\mu$ be random functions with the same domains and ranges as $\mathcal{C}_{k_1}, \ldots, \mathcal{C}_{k_\mu}$. Let D be a distinguisher. The PRF distinguishing advantage of D is defined as,

$$\mathsf{PRF}_{\mathcal{C}^{\mathcal{P}}}(\mathsf{D}) = \left|\Pr\left[\mathsf{Dist}^{\mathcal{C}^{\mathcal{P}}_{k_1},\ldots,\mathcal{C}^{\mathcal{P}}_{k_\mu},\mathcal{P}} = 1\right] - \Pr\left[\mathsf{Dist}^{\mathcal{R}_1,\ldots,\mathcal{R}_\mu,\mathcal{P}} = 1\right]\right|$$

$\mathsf{Blake2}$ supports keyed hashing by simply prepending the key to the message:

$$\mathsf{Blake2}_{\mathsf{E},k}(m) = \mathsf{Blake2}_{\mathsf{E}}(k \| 0^{2\mathsf{ol}-\mathsf{kl}} \| m)$$

where $\mathsf{kl} \leq 2\mathsf{ol}$ denotes the key size. In other words, the key gets processed as other data and the HAIFA counter and flags are designated to the key in a similar fashion as if they were for normal data blocks.

Theorem D.2.1 (PRF-security of $\mathsf{Blake2}$ keyed mode [LMN16])**.** *Let $\mu \geq 1$ and let $k \leftarrow\$ \left(\mathbb{B}^{\mathsf{kl}}\right)^{\mu}$. Let an encryption scheme $\mathsf{E} \leftarrow\$ \mathcal{BLK}^*(2 \cdot \mathsf{ol}, 2 \cdot \mathsf{ol})$ be a weakly ideal cipher, and consider the keyed hash function $\mathsf{Blake2}_{\mathsf{E},k}$ that internally uses $\mathsf{Blake2C}_{\mathsf{E}}$ that internally uses E. For any distinguisher Dist with total complexity q:*

$$\mathsf{PRF}_{\mathsf{Blake2}_{\mathsf{E},k}}(\mathsf{Dist}) \leq \frac{\binom{q}{2}}{2^{2\mathsf{ol}}} + \frac{2\binom{q}{2}}{2^{\mathsf{ol}}} + \frac{q}{2^{\mathsf{ol}/2}} + \frac{\mu q}{2^{\mathsf{kl}}} + \frac{\binom{\mu}{2}}{2^{\mathsf{kl}}}$$

Proof. See [LMN16, Corollary 3]. ☐

Remark D.2.2. We can note that in the case of keyed hashing, the key is padded only to be processed in a single block to differentiate the key from the message. The security proof of Theorem D.2.1 does not rely on this padding and as such also works with no padding.

Theorem D.2.2 (PRF-security of $\mathsf{Blake2}$ with a single key [LMN16])**.** *Let $k \leftarrow\$ \mathbb{B}^{\mathsf{kl}}$. Let an encryption scheme $\mathsf{E} \leftarrow\$ \mathcal{BLK}^*(2 \cdot \mathsf{ol}, 2 \cdot \mathsf{ol})$ be a weakly ideal cipher, and consider the keyed hash function $\mathsf{Blake2}_{\mathsf{E}}(k, \cdot) = \mathsf{Blake2}_{\mathsf{E}}(k \| \cdot)$ that internally uses $\mathsf{Blake2C}_{\mathsf{E}}$ that internally uses E. For any distinguisher Dist with total complexity q:*

$$\mathsf{PRF}_{\mathsf{Blake2}_{\mathsf{E}}}(\mathsf{Dist}) \leq \frac{\binom{q}{2}}{2^{2\mathsf{ol}}} + \frac{2\binom{q}{2}}{2^{\mathsf{ol}}} + \frac{q}{2^{\mathsf{ol}/2}} + \frac{q}{2^{\mathsf{kl}}}$$

Proof. See Remark D.2.2 and Theorem D.2.1 with $\mu = 1$. ☐

Remark D.2.3. Since we analyse the security of $\mathsf{Blake2}$ asymptotically, we assume that for a security parameter λ holds $\mathsf{ol} = \mathcal{O}(\lambda)$, $\mathsf{kl} = \mathcal{O}(\lambda)$, and $q = \mathsf{poly}(\lambda)$.

D.2.2 Proof of Blake2 collision resistance

We want to prove here the collision resistance of $\mathsf{Blake2}$. To do so, we are going to prove by contradiction that if $\mathsf{Blake2}$ is not collision resistant, it is not indifferentiable according to Definition D.1.1.

Theorem D.2.3. $\mathsf{Blake2}$ *is collision resistant.*

Informal proof. Let us assume that there exists a PPT adversary $\mathcal{B}$ which breaks the collision resistance of $\mathsf{Blake2}$. We build an adversary $\mathcal{A}$ that uses this adversary to differentiate between the real and simulated worlds. More particularly, $\mathcal{A}$ gets left and right oracles (see [LMN16, Figure 3]), which are either an oracle for a hash function and for a weakly ideal block cipher or a random oracle and an encryption simulator with oracle access to the random oracle.

On each $\mathcal{B}$'s query m_i, $i \in \{1, \ldots, q\}$, $\mathcal{A}$ passes them to his left oracle and returns the answer h_i to $\mathcal{B}$. Eventually, if $\mathcal{B}$ finds a collision, that is a pair (m_i, m_j) such that $m_i \neq m_j$ and $h_i = h_j$, $\mathcal{A}$ guesses that his oracles were real; else $\mathcal{A}$ returns a random guess. Otherwise $\mathcal{A}$ guesses his oracles were simulated – if the left oracle was a random oracle, the probability of finding a collision would be negligible for $q \leq \mathsf{poly}(\lambda)$[1].

On the other hand, $\mathcal{B}$ finds a collision with non-negligible probability if the oracles were real. Hence, $\mathcal{A}$ wins the indifferentiability game with non-negligible advantage, which is a contradiction. $\square$

D.2.3 Blake2 as a commitment scheme

We prove here that $\mathsf{Blake2}$ is a commitment scheme, i.e. is binding and hiding. To do so we rely on the previous results that $\mathsf{Blake2}$ is collision resistant and a PRF.

Theorem D.2.4. *Let* $\mathsf{E} \leftarrow\!\$\, \mathcal{BLK}(\mathsf{2ol}, \mathsf{2ol})$ *and for a message* $x \in \mathbb{B}^*$ *and randomness* $r \in \mathbb{B}^{\mathsf{rl}}$ *commitment to* x *using* r *be* $\mathsf{ComSch.Com}(x; r) = \mathsf{Blake2}_{\mathsf{E}}(r \| x)$. *Then* ComSch *is hiding and binding.*

Informal proof. Hiding. A commitment scheme ComSch is computationally hiding if, knowing two potential openings, a PPT adversary cannot distinguish which was committed. Let us assume that there exists a PPT adversary $\mathcal{B}$ which breaks the hiding property of $\mathsf{Blake2}$ with a non-negligible advantage η. We build an adversary $\mathcal{A}$ that uses $\mathcal{B}$ to break the PRF property of $\mathsf{Blake2}$ with advantage $\eta/2$.

First, the PRF game is initiated, that is, the challenger chooses a random encryption scheme E and key $k \in \mathbb{B}^{\mathsf{rl}}$ and instantiates two oracles $\mathsf{O}^{\mathsf{Blake2}_k} = \mathsf{Blake2}_{\mathsf{E}}(k, \cdot)$ and O^R a random function. The challenger picks an oracle randomly and gives $\mathcal{A}$ access to it. $\mathcal{B}$ sends q oracle queries $m_1, \ldots, m_q$ to $\mathcal{A}$ (adaptively) who extends them with random $r_1, \ldots, r_q$ and sends $r_i \| m_i$ to his left oracle. Given the answer from the oracle, $\mathcal{A}$ returns them to $\mathcal{B}$. Eventually, $\mathcal{B}$ then outputs two challenge messages $(\tilde{m}_0, \tilde{m}_1)$ and sends them to $\mathcal{A}$ who randomly selects message $\tilde{m}_b$, extends it with r and sends $r \| \tilde{m}_b$ to his left oracle. The oracle answers with y_b which is also sent to $\mathcal{B}$. Finally, $\mathcal{B}$ returns the decision bit $\tilde{b}$ to $\mathcal{A}$. If $b = \tilde{b}$, $\mathcal{A}$ answers to the challenger that the oracle was instantiating the PRF. Otherwise, $\mathcal{A}$ answers with a random guess. The advantage of $\mathcal{A}$ equals advantage of $\mathcal{B}$ if it interacts with a real hash function. The advantage of $\mathcal{A}$ equals half the advantage of $\mathcal{B}$ when interacting with a random oracle and simulator.

Binding. A commitment scheme ComSch is said to be computationally binding if it is infeasible to find x, x' and r, r' such that $x \neq x'$ and $\mathsf{Com}(x; r) = \mathsf{Com}(x'; r')$. This is implied by collision resistance of $\mathsf{Blake2}$. Thus if $\mathcal{B}$ is an algorithm that breaks the biding property with advantage η, there is another algorithm $\mathcal{A}$ that breaks $\mathsf{Blake2}$ collision resistance with the same advantage. $\square$

[1] The probability would be $\frac{q^2}{2^{\mathsf{ol}}}$ which is negligible for a polynomial number of queries q. This is the sum of the probabilities of finding a collision when doing the i^{th} query. Indeed, let us suppose the adversary has done $i-1$, $i > 2$, queries without finding a collision, i.e. he knows $i-1$ distinct tuples of input-output. When receiving the i^{th} value, the adversary has thus $i-1$ chance to find a collision. The probability for the new output to be equal to any of the previous outputs is thus $(i-1) \cdot \frac{1}{2^{\mathsf{ol}}}$ (as we are in the random oracle model). Summing this probability over all queries, we find the probability of finding a collision after doing q queries.

Assuming that $\mathsf{Blake2s}$ is as secure as $\mathsf{Blake2}$, a commitment scheme based on a $\mathsf{Blake2s}$, i.e. $\mathsf{Com}(x; r) = \mathsf{Blake2s}_{\mathsf{E}}(r \| x)$ is hiding and binding.

D.2.4 Proof of commitment scheme security

To prove the binding and hiding property of ComSch (see Section 3.1.2), we introduce the following commitment scheme ComSch^*,

$$\mathsf{ComSch}^*.\mathsf{Setup} : \left\{1^\lambda \text{ s.t. } \lambda \in \mathbb{N}\right\} \to \mathbb{B}^*$$

$$\begin{aligned} \mathsf{ComSch}^*.\mathsf{Com} : \mathcal{BLK}^*(2 \cdot \mathtt{BLAKE2sCLEN}, 2 \cdot \mathtt{BLAKE2sCLEN}) \times \mathbb{B}^{2 \cdot \mathtt{BLAKE2sCLEN}} \\ \times \left(\mathbb{B}^{\mathtt{PRFADDROUTLEN}} \times \mathbb{B}^{\mathtt{PRFRHOOUTLEN}} \times \mathbb{B}^{\mathtt{ZVALUELEN}}\right) \times \mathbb{B}^{\mathtt{RTRAPLEN}} \to \mathbb{B}^{\mathtt{BLAKE2sCLEN}} \end{aligned}$$

The commitment scheme is defined as follows,

$$\begin{aligned} &\mathsf{ComSch}^*.\mathsf{Setup}(1^\lambda) = pp^* = \epsilon \\ &\mathsf{ComSch}^*.\mathsf{Com}(m = (apk, \rho, v); r) = cm \\ &= \mathsf{Blake2}_{\mathsf{E}^*}(r \| apk \| \rho \| v) \end{aligned}$$

Given a commitment scheme ComSch^*, the bijective function $\mathsf{decode}_{\mathbb{N}}(\cdot)$ and $p_\lambda \in \mathbb{N}$, a prime which can be represented using λ bits, we define the commitment scheme ComSch' as follows:

$$\begin{aligned} &\mathsf{ComSch}'.\mathsf{Setup}(1^\lambda) = (\mathsf{ComSch}^*.\mathsf{Setup}(1^\lambda), p_\lambda) \\ &\mathsf{ComSch}'.\mathsf{Com}(m; r) = \mathsf{decode}_{\mathbb{N}}(\mathsf{ComSch}^*.\mathsf{Com}(m; r)) \pmod{p_\lambda} \text{ for } m = (apk \| \rho \| v) \end{aligned}$$

Note that ComSch (see Section 3.1.2) is a particular instantiation of ComSch' where E^* is set as ChaCha encryption scheme [Ber08a], k^* is a random key, and p_λ is $\mathtt{r}$.

Theorem D.2.5 (Hiding)**.** *If* ComSch^* *is hiding then* ComSch' *is hiding.*

Proof. We prove the theorem by contradiction i.e. we assume that there exists an adversary $\mathcal{B}$ that breaks ComSch''s hiding property and construct an adversary $\mathcal{A}$ that uses $\mathcal{B}$ to break ComSch^*'s hiding property with non-negligible probability.

Let $\mathcal{C}$ be a challenger that sets up the hiding game for ComSch^* and $\mathcal{A}$. The adversary $\mathcal{A}$, given public parameters pp^* of ComSch^* and access to an oracle that runs the Com algorithm of ComSch^* scheme, simulates a hiding game for ComSch' for $\mathcal{B}$. The adversary $\mathcal{A}$ starts by setting public parameters pp' for ComSch' using public parameters pp^* given by $\mathcal{C}$. Parameters pp' are passed to $\mathcal{B}$ who outputs a pair of messages m_0, m_1. The adversary $\mathcal{A}$ forwards them to the challenger who samples a bit b at random and generates $cm^* = \mathsf{ComSch}^*.\mathsf{Com}(m_b; r)$ for some randomness r. The result is returned to $\mathcal{A}$ (see Definition 1.5.21). Then $\mathcal{A}$ passes $cm = \mathsf{decode}_{\mathbb{N}}(cm^*) \pmod{p_\lambda}$ to $\mathcal{B}$ who returns his guess b'. The adversary $\mathcal{A}$ returns the same b' to the challenger.

By construction, it is clear that $\mathcal{A}$ wins the hiding game with the same probability that $\mathcal{B}$ wins the simulated hiding game. Since $\mathcal{B}$'s advantage is non-negligible, this means that $\mathcal{A}$ wins the ComSch^* hiding game with non-negligible probability as well. $\square$

Theorem D.2.6 (Binding)**.** *Let* ComSch^* *be a computationally binding commitment scheme and* $\mathsf{ComSch}^*.\mathsf{Com}$ *indifferentiable from a random oracle. Then* ComSch' *is also computationally binding if* $l = \lceil 2^\lambda / p_\lambda \rceil$ *is at most* $\mathsf{poly}(\lambda)$*.*

Proof. Assume that $\mathcal{A}$ asks the ComSch' commit and open oracles a total of q_λ distinct queries. Let us denote the result of the q_λ queries and output of the attacker (the candidate collision) as $((m_1, r_1, y_1), \ldots, (m_{q_\lambda}, r_{q_\lambda}, y_{q_\lambda}), \mathrm{out})$. If $\mathcal{A}$ is successful it means that it outputs (m, r), (m', r') such that $(m, r) \neq (m', r')$ and $\mathsf{ComSch}'.\mathsf{Com}(m; r) = \mathsf{ComSch}'.\mathsf{Com}(m'; r')$.

By the definition of ComSch', we have that,

$$\mathsf{ComSch}'.\mathsf{Com}(m; r) = \mathsf{decode}_{\mathbb{N}}(\mathsf{ComSch}^*.\mathsf{Com}(m; r)) \pmod{p_\lambda}$$

Hence, we have a collision in ComSch' if there exists $k \in [l]$, l being the ratio of the codomains of $\mathsf{ComSch}^*.\mathsf{Com}$ and $\mathsf{ComSch}'.\mathsf{Com}$, such that,

$$\left|\mathsf{decode}_{\mathbb{N}}(\mathsf{ComSch}^*.\mathsf{Com}(m; r)) - \mathsf{decode}_{\mathbb{N}}(\mathsf{ComSch}^*.\mathsf{Com}(m'; r'))\right| = k \cdot p_\lambda.$$

We show that this event is unlikely.

In fact, for each $i \in [q_\lambda]$, let C_i be the event that the adversary wins at the i-th query. That is, the last commitment y_i is a ComSch' collision with one of the previous y_j. More precisely there exists $j \leq i$ and $k < l$ such that $y_i = y_j + k \cdot p_\lambda$.

Since ComSch^* is a random oracle, y_i is randomly selected from a set of at least p_λ elements. As such, we have $\Pr[C_i] \leq i \cdot l / p_\lambda$.

Thus the probability of finding a collision after q_λ queries is $\Pr[C_1 \vee \ldots \vee C_{q_\lambda}] \leq \sum_{i=1}^{q_\lambda} \Pr[C_i] = l/p_\lambda \cdot \sum_{i=1}^{q_\lambda} i$. This probability is bounded by $l \cdot \frac{q_\lambda(q_\lambda+1)}{p_\lambda}$. However, we allow only polynomial number of queries. Thus for $q_\lambda = \mathsf{poly}(\lambda)$ this probability becomes,

$$\frac{2^\lambda \cdot \mathsf{poly}(\lambda)}{p_\lambda^2},$$

what is negligible for $2^\lambda / p_\lambda \leq \mathsf{poly}(\lambda)$. □

Remark D.2.4. Note that in `Zeth`'s commitment scheme, we set $p_\lambda = \mathtt{r}$ and $2^\lambda = 2^{\mathtt{BLAKE2sCLEN}}$. Thus, for BN-254 and BLS12-377 have $l = 6$ and $l = 14$, respectively. Therefore, the probability of an attacker breaking the binding property due to reduction modulo `r` increases approximately by these factors. This is still negligible.

Corollary. *Assume that* Blake2 *is indifferentiable from a random oracle and a PRF, then* ComSch^* *is computationally binding and computationally hiding. Furthermore, the reduction is* tight*. That is, the advantage of any* PPT *adversary against the binding (resp. hiding) property is the same as the advantage of an adversary against collision resistance and binding (resp. hiding).*

Glossary

joinsplit Set of `JSIN` input *ZethNotes*, and `JSOUT` output *ZethNotes* as well as the public values *vin* and *vout* used in a tx_{Mix} transaction. 37, 39, 41, 59, 96, 115, 116

joinsplit equation Equation that checks that the sum of the values of the SendTx algorithm of DAP is equal to the sum of the values of its outputs. This equations checks that the joinsplit is "balanced" and thus, that no value is created while creating new *ZethNotes*. 25, 41, 59, 96, 115

Acronyms

DOS Denial of Service (Attack). 16, 115

EOA Externally Owned Account. 16, 17, 19, 115

EVM Ethereum Virtual Machine. 15, 16, 20, 48, 54, 64, 84, 86, 115

FFT Fast Fourier Transform. 85, 115

MAC Message Authentication Code. 98, 115

PoC Proof of Concept. 115

RAM Random-access Memory. 85, 115

RLP Recursive Length Prefix. 19, 20, 115

Bibliography

[AAM12] Imad Fakhri Alshaikhli, Mohammad A Alahmad, and Khanssaa Munthir. Comparison and analysis study of sha-3 finalists. In *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 366–371. IEEE, 2012.

[abi] Contract abi specification, section "function selector". `https://solidity.readthedocs.io/en/develop/abi-spec.html#function-selector`.

[ABM+03] Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, and Scott Vanstone. Validation of elliptic curve public keys. In *International Workshop on Public Key Cryptography*, pages 211–223. Springer, 2003.

[ABN10] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In *Theory of Cryptography Conference*, pages 480–497. Springer, 2010. `https://eprint.iacr.org/2008/440.pdf`.

[ABR99] Michel Abdalla, Mihir Bellare, and Phillipi Rogaway. Dhaes: An encryption scheme based on the diffie-hellman problem. 1999. `https://pdfs.semanticscholar.org/95f4/63d097086fba325086a4cf88706648dafd09.pdf`.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. Dhies: An encryption scheme based on the diffie-hellman problem., 2001. `https://web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf`.

[ACG+19] Martin R Albrecht, Carlos Cid, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, and Markus Schofnegger. Algebraic cryptanalysis of stark-friendly designs: application to marvellous and mimc. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 371–397. Springer, 2019.

[AFK+08] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of latin dances: analysis of salsa, chacha, and rumba. In *International Workshop on Fast Software Encryption*, pages 470–488. Springer, 2008.

[AG18] Andreas M. Antonopoulos and Wood Gavin. *Mastering Ethereum*. O'Reilly Media, 2018.

[AGM+09] Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, Yu Sasaki, and Lei Wang. Preimages for step-reduced sha-2. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 578–597. Springer, 2009. https://link.springer.com/content/pdf/10.1007/978-3-642-10366-7_34.pdf.

[AGR+16] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 191–219. Springer, 2016.

[AHMP08] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W Phan. Sha-3 proposal blake. *Submission to NIST*, 229:230, 2008.

[ALM12] Elena Andreeva, Atul Luykx, and Bart Mennink. Provable security of blake with non-ideal compression function. In *International Conference on Selected Areas in Cryptography*, pages 321–338. Springer, 2012.

[AMP10] Elena Andreeva, Bart Mennink, and Bart Preneel. Security reductions of the second round sha-3 candidates. In *International Conference on Information Security*, pages 39–53. Springer, 2010.

[AMPŠ12] Elena Andreeva, Bart Mennink, Bart Preneel, and Marjan Škrobot. Security analysis and comparison of the sha-3 finalists blake, grøstl, jh, keccak, and skein. In *International Conference on Cryptology in Africa*, pages 287–305. Springer, 2012.

[ANWOW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013. https://eprint.iacr.org/2013/322.pdf.

[AS09] Kazumaro Aoki and Yu Sasaki. Meet-in-the-middle preimage attacks against reduced sha-0 and sha-1. In *Annual International Cryptology Conference*, pages 70–89. Springer, 2009.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 566–582. Springer, 2001. https://iacr.org/archive/asiacrypt2001/22480568.pdf.

[BCC+15] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, Jens Groth, and Christophe Petit. Short accountable ring signatures based on ddh. In *European Symposium on Research in Computer Security*, pages 243–265. Springer, 2015.

[BCD+20] Tim Beyne, Anne Canteaut, Itai Dinur, Maria Eichlseder, Gregor Leander, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Yu Sasaki, Yosuke Todo, and Friedrich Wiemer. Out of oddity – new cryptanalytic techniques against symmetric primitives optimized for integrity proof systems. Cryptology ePrint Archive, Report 2020/188, 2020. `https://eprint.iacr.org/2020/188`.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 947–964. IEEE, 2020.

[BCK+18] Elaine Barker, Lily Chen, Sharon Keller, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. Technical report, National Institute of Standards and Technology, 2018. [Online; last accessed 10-January-2020].

[BD07] Eli Biham and Orr Dunkelman. A framework for iterative hash functions—haifa. Technical report, Computer Science Department, Technion, 2007. `https://eprint.iacr.org/2007/278.pdf`.

[BDPVA07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer, 2007.

[Ber05] Daniel J Bernstein. The poly1305-aes message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005. `https://cr.yp.to/mac/poly1305-20050329.pdf`.

[Ber06] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006. `https://cr.yp.to/ecdh/curve25519-20060209.pdf`.

[Ber08a] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5, 2008. `https://cr.yp.to/chacha/chacha-20080120.pdf`.

[Ber08b] Daniel J. Bernstein. New stream cipher designs. chapter The Salsa20 Family of Stream Ciphers, pages 84–97. Springer-Verlag, Berlin, Heidelberg, 2008.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. `https://eprint.iacr.org/2017/1050`.

[BL] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. `https://safecurves.cr.yp.to`. [Online; last accessed 09-December-2019].

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[Bon19] Xavier Bonnetain. Collisions on feistel-mimc and univariate gmimc. 2019.

[Bou03] Nicolas Bourbaki. *Elements of mathematics: Algebra*. Springer, 2003.

[Bra97] S. Bradner. Key words for use in rfcs to indicate requirement levels. RFC 2119, RFC Editor, March 1997.

[BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In *Annual International Cryptology Conference*, pages 320–335. Springer, 2002. `https://link.springer.com/content/pdf/10.1007/3-540-45708-9_21.pdf`.

[BS07] Mihir Bellare and Sarah Shoup. Two-tier signatures, strongly unforgeable signatures, and fiat-shamir without random oracles. In *International Workshop on Public Key Cryptography*, pages 201–216. Springer, 2007.

[BSCG+14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. pages 738–768, 2020.

[Cle19] Clearmatics. Zeth Sproken release. `https://github.com/clearmatics/zeth/releases/tag/v0.2`, 2019. [Online; released 04-April-2019].

[CM16] Arka Rai Choudhuri and Subhamoy Maitra. Differential cryptanalysis of salsa and chacha-an evaluation with a hybrid model. *IACR Cryptology ePrint Archive*, 2016:377, 2016. `https://eprint.iacr.org/2016/377.pdf`.

[CM17] Arka Rai Choudhuri and Subhamoy Maitra. Significantly improved multi-bit differentials for reduced round salsa and chacha. *IACR Transactions on Symmetric Cryptology*, 2016(2):261–287, Feb. 2017.

[DG09] George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 269–282. IEEE Computer Society, 2009.

[EFK15] Thomas Espitau, Pierre-Alain Fouque, and Pierre Karpman. Higher-order differential meet-in-the-middle preimage attacks on sha-1 and blake. In *Annual Cryptology Conference*, pages 683–701. Springer, 2015. `https://eprint.iacr.org/2015/515`.

[EGL+20] Maria Eichlseder, Lorenzo Grassi, Reinhard Lüftenegger, Morten Øygarden, Christian Rechberger, Markus Schofnegger, and Qingju Wang. An algebraic attack on ciphers with low-degree round functions: Application to full mimc. Cryptology ePrint Archive, Report 2020/182, 2020. `https://eprint.iacr.org/2020/182`.

[est] Estream project. `https://en.wikipedia.org/wiki/ESTREAM`.

[Gab19] Ariel Gabizon. AuroraLight: Improved prover efficiency and SRS size in a sonic-like system. Cryptology ePrint Archive, Report 2019/601, 2019. `https://eprint.iacr.org/2019/601`.

[GFBR06] Decio Gazzoni Filho, Paulo SLM Barreto, and Vincent Rijmen. The maelstrom-0 hash function. In *Brazilian Symposium on Information and Computer System Security. .*, 2006.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[GJMG11] B Guido, D Joan, P Michaël, and VA Gilles. The keccak sha-3 submission. 2011.

[GKN+14] Jian Guo, Pierre Karpman, Ivica Nikolić, Lei Wang, and Shuang Wu. Analysis of blake2. In *Cryptographers' Track at the RSA Conference*, pages 402–423. Springer, 2014. `https://eprint.iacr.org/2013/467.pdf`.

[GLRW10] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang. Advanced meet-in-the-middle preimage attacks: First results on full tiger, and improved results on md4 and sha-2. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 56–75. Springer, 2010. `https://eprint.iacr.org/2010/016.pdf`.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 581–612. Springer, 2017.

[Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. pages 444–459, 2006.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 305–326. Springer, 2016.

[GRR+16] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P Smart. Mpc-friendly symmetric key primitives. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 430–443. ACM, 2016. `https://eprint.iacr.org/2016/542`.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. `https://eprint.iacr.org/2019/953`.

[Hao14] Yonglin Hao. The boomerang attacks on blake and blake2. In *International Conference on Information Security and Cryptology*, pages 286–310. Springer, 2014. `https://eprint.iacr.org/2014/1012.pdf`.

[Har19] HarryR. Conversation about Miyaguchi-Preneel security. `https://github.com/HarryR/ethsnarks/issues/119`, 2019. Online; accessed June-2019.

[HG20] Youssef El Housni and Aurore Guillevic. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. Cryptology ePrint Archive, Report 2020/351, 2020. `https://eprint.iacr.org/2020/351`.

[HKT11] Thomas Holenstein, Robin Künzler, and Stefano Tessaro. The equivalence of the random oracle model and the ideal cipher model, revisited. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 89–98. ACM, 2011.

[HMRS12] Ekawat Homsirikamol, Paweł Morawiecki, Marcin Rogawski, and Marian Srebrny. Security margin evaluation of sha-3 contest finalists through sat-based attacks. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 56–67. Springer, 2012.

[Hop16] Daira Hopwood. Daira's comment on: "ensure spec retains distinctness assumption in hsig", 2016.

[IS09] Takanori Isobe and Kyoji Shibutani. Preimage attacks on reduced tiger and sha-2. In *International Workshop on Fast Software Encryption*, pages 139–155. Springer, 2009. `https://link.springer.com/content/pdf/10.1007/978-3-642-03317-9_9.pdf`.

[Ish12] Tsukasa Ishiguro. Modified version of" latin dances revisited: New analytic results of salsa20 and chacha". 2012. `https://eprint.iacr.org/2012/065.pdf`.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[Joh16] Nick Johnson. Response to "how does ethereum make use of bloom filters?". `https://ethereum.stackexchange.com/questions/3418/how-does-ethereum-make-use-of-bloom-filters`, 2016. [Online; last accessed 10-January-2020].

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* Chapman and Hall/CRC, 2014. `https://repo.zenk-security.com/Cryptographie%20.%20Algorithmes%20.%20Steganographie/Introduction%20to%20Modern%20Cryptography.pdf`.

[KMO+13] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. Anonymity-preserving public-key encryption: A constructive approach. In Emiliano De Cristofaro and Matthew K. Wright, editors, *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 19–39. Springer, 2013.

[KRS12] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for preimages: attacks on skein-512 and the sha-2 family. In *International Workshop on Fast Software Encryption*, pages 244–263. Springer, 2012. `https://link.springer.com/content/pdf/10.1007/978-3-642-34047-5_15.pdf`.

[Lab19] Matter Labs. Merkle shrubs. `https://github.com/matter-labs/MerkleShrubs`, 2019.

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. RFC 7748, `https://tools.ietf.org/pdf/rfc7748.pdf`, 2016.

[LIS12] Ji Li, Takanori Isobe, and Kyoji Shibutani. Converting meet-in-the-middle preimage attack into pseudo collision attack: Application to sha-2. In *International Workshop on Fast Software Encryption*, pages 264–286. Springer, 2012. `https://link.springer.com/content/pdf/10.1007/978-3-642-34047-5_16.pdf`.

[LM11] Mario Lamberger and Florian Mendel. Higher-order differential attack on reduced sha-256. *IACR Cryptology ePrint Archive*, 2011:37, 2011. `https://eprint.iacr.org/2011/037.pdf`.

[LMN16] Atul Luykx, Bart Mennink, and Samuel Neves. Security analysis of blake2's modes of operation. *IACR Transactions on Symmetric Cryptology*, pages 158–176, 2016. `https://www.esat.kuleuven.be/cosic/publications/article-2705.pdf`.

[LN18] Adam Langley and Yoav Nir. Chacha20 and poly1305 for ietf protocols. *RFC 8439*, 2018. `https://tools.ietf.org/html/rfc8439`.

[LP19] Chaoyun Li and Bart Preneel. Improved interpolation attacks on cryptographic primitives of low algebraic degree. In *International Conference on Selected Areas in Cryptography*, pages 171–193. Springer, 2019.

[Mai16] Subhamoy Maitra. Chosen iv cryptanalysis on reduced round chacha and salsa. *Discrete Applied Mathematics*, 208:88–97, 2016.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. pages 2111–2128, 2019.

[MJS15] Ed. M-J. Saarinen. Blake Compression Function F. `https://tools.ietf.org/html/rfc7693#section-3.2`, 2015. [Online; accessed November-2019].

[ML15] Nicky Mouha and Atul Luykx. Multi-key security: The even-mansour construction revisited. In *Annual Cryptology Conference*, pages 209–223. Springer, 2015. `https://hal.inria.fr/hal-01240988/document`.

[MNS11] Florian Mendel, Tomislav Nad, and Martin Schläffer. Finding sha-2 characteristics: searching through a minefield of contradictions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 288–307. Springer, 2011. `https://link.springer.com/content/pdf/10.1007/978-3-642-25385-0_16.pdf`.

[Moh10] Payman Mohassel. A closer look at anonymity and robustness in encryption schemes. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*, volume 6477 of *Lecture Notes in Computer Science*, pages 501–518. Springer, 2010.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48(177):243–264, 1987.

[MP15] Bart Mennink and Bart Preneel. On the impact of known-key attacks on hash functions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 59–84. Springer, 2015. `https://eprint.iacr.org/2015/909.pdf`.

[MQZ10] Mao Ming, He Qiang, and Shaokun Zeng. Security analysis of blake-32 based on differential properties. In *2010 International Conference on Computational and Information Sciences*, pages 783–786. IEEE, 2010.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. Handbook of applied cryptography. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.2838&rep=rep1&type=pdf`, 1996.

[NA19] Samuel Neves and Filipe Araujo. An observation on norx, blake2, and chacha. *Information Processing Letters*, 149:1–5, 2019.

[oST15] National Institute of Standards and Technology. Secure Hash Standard (SHS). `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`, 2015.

[Per17] Trevor Perrin. X25519 and zero outputs. `https://moderncrypto.org/mail-archive/curves/2017/000896.html`, 2017. [Online; last accessed 08-January-2020].

[PHE+17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1199–1216. USENIX Association, 2017.

[Por13] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). `https://tools.ietf.org/html/rfc6979`, 2013.

[Pro14] Gordon Procter. A security analysis of the composition of chacha20 and poly1305. *IACR Cryptology ePrint Archive*, 2014:613, 2014.

[PV05] Pascal Paillier and Damien Vergnaud. Discrete-log-based signatures may not be equivalent to discrete log. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2005.

[Qu99] Minghua Qu. Sec 2: Recommended elliptic curve domain parameters. *Certicom Res., Mississauga, ON, Canada, Tech. Rep. SEC2-Ver-0.6*, 1999.

[Rk19] Antoine Rondelet and @karalabe. Go-ethereum BN256 package. `https://github.com/ethereum/go-ethereum/blob/master/crypto/bn256/cloudflare/constants.go`, 2019. [Online; released 28-May-2019].

[Ron20] Antoine Rondelet. Zecale: Reconciling privacy and scalability on ethereum, 2020.

[RZ19] Antoine Rondelet and Michal Zajac. ZETH: On Integrating Zerocash on Ethereum. [Online; released April-2019], 2019.

[SS08] Somitra Kumar Sanadhya and Palash Sarkar. New collision attacks against up to 24-step sha-2. In *International conference on cryptology in India*, pages 91–103. Springer, 2008. `https://eprint.iacr.org/2008/270.pdf`.

[Ste15] Stevens, Marc. On collisions for md5. `https://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf`, 2015.

[SZFW12] Zhenqing Shi, Bin Zhang, Dengguo Feng, and Wenling Wu. Improved key recovery attacks on reduced-round salsa20 and chacha. In *International Conference on Information Security and Cryptology*, pages 337–351. Springer, 2012.

[TBP20] Florian Tramèr, Dan Boneh, and Kenneth G. Paterson. Remote side-channel attacks on anonymous transactions. Cryptology ePrint Archive, Report 2020/220, 2020. `https://eprint.iacr.org/2020/220`.

[THH15] Piotr Dyraga Tjaden Hess, Matt Luongo and James Hancock. EIP 152: Add BLAKE2 compression function ‘F‘. `https://eips.ethereum.org/EIPS/eip-152`, 2015. [Online; accessed November-2019].

[VNP10] Janoš Vidali, Peter Nose, and Enes Pašalić. Collisions for variants of the blake hash function. *Information processing letters*, 110(14-15):585–590, 2010.

[W$^+$] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger.

[wc] Ethereum wiki contributors. Patricia tree. https://github.com/ethereum/wiki/wiki/Patricia-Tree.

[wc19] Ethereum wiki contributors. RLP. https://github.com/ethereum/wiki/wiki/RLP, 2019. [Online; accessed December-2019].

[wik] Bitcoin wiki. Secp256k1. https://en.bitcoin.it/wiki/Secp256k1. [Online; last accessed 04-January–2020].

[Woo19] Dr Gavin Wood. ETHEREUM: A Secure Decentralised Generalised Transaction Ledger Byzantium. https://ethereum.github.io/yellowpaper/paper.pdf, 2019. [VERSION 7e819ec - 2019-10-20].

[zcaa] On the security of sprout/sapling in-band encryption. https://forum.zcashcommunity.com/t/on-the-security-of-sprout-sapling-in-band-encryption/34986.

[zcab] "zcash alerts - security announcement 2019-09-24". https://z.cash/support/security/announcements/security-announcement-2019-09-24/a.

[ZCa19] ZCash. ZCash protocol specification. https://github.com/zcash/zips/blob/master/protocol/protocol.pdf, 2019. [Online; initially released 14-December-2015].