

1

Zeth Protocol Specification

2

Clearmatics Cryptography R&D

3

2021-01-27

Abstract

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

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

9 Contents

10	1 Preliminaries	6
11	1.1 Data structures and representation	6
12	1.1.1 Structured data	6
13	1.1.2 Representations	9
14	1.2 Ethereum	10
15	1.2.1 Ethereum account	11
16	1.2.2 Ethereum transaction	12
17	1.2.3 Ethereum events and Bloom filters	15
18	1.3 zk-SNARKs	16
19	1.3.1 Preliminary definitions	16
20	1.3.2 Computation representation – arithmetization	18
21	1.4 Decentralized Anonymous Payment schemes (DAP)	19
22	1.5 Definitions	21
23	1.5.1 Negligible function	21
24	1.5.2 Basic algebra notions	22
25	1.5.3 Security assumptions	22
26	1.5.4 Symmetric encryption	23
27	1.5.5 Asymmetric encryption	24
28	1.5.6 Block cipher-based compression functions	25
29	1.5.7 Hash functions	26
30	1.5.8 Pseudo Random Functions	28
31	1.5.9 Commitment scheme	28
32	1.5.10 Digital Signature	29
33	1.5.11 Message Authentication Code	30
34	2 Zeth protocol	32
35	2.1 Zeth Data Types	32
36	2.2 Zeth statement	35
37	2.3 Generating the inputs of the Mix function (Mix_{in})	36
38	2.4 Creating an Ethereum transaction tx_{Mix} to call Mixer	38
39	2.5 Processing tx_{Mix}	38
40	2.6 Receiving <i>ZethNotes</i>	40
41	2.7 Security requirements for the primitives	41

42	2.7.1	Additional notes	41
43	3	Instantiation of the cryptographic primitives	43
44	3.1	Instantiating the PRFs, ComSch and CRHs	44
45	3.1.1	Blake2 primitive	44
46	3.1.2	Commitment scheme	45
47	3.1.3	PRFs	45
48	3.1.4	Collision resistant hashes	47
49	3.2	Instantiating MKHASH	47
50	3.2.1	MIMC Encryption	47
51	3.2.2	MIMC-based compression function	48
52	3.2.3	An efficient instantiation of MIMC primitives	49
53	3.2.4	Security requirements satisfaction	50
54	3.3	Zeth statement after primitive instantiation	53
55	3.3.1	Instantiating the packing functions	54
56	3.4	Instantiate SigSch _{OT-SIG}	58
57	3.4.1	Security requirements satisfaction	59
58	3.4.2	Data types	59
59	3.5	Instantiate EncSch	60
60	3.5.1	DHAES encryption scheme	60
61	3.5.2	A DHAES instance	61
62	3.5.3	EncSch instantiation	64
63	3.5.4	Security requirements satisfaction	67
64	3.5.5	Final notes and observations	68
65	3.6	ZkSnarkSch instantiation	71
66	4	Implementation considerations and optimizations	74
67	4.1	Client security considerations	74
68	4.1.1	Syncing and waiting	75
69	4.1.2	Note management	76
70	4.1.3	Prepare arguments for Mix transaction	77
71	4.1.4	Wallet backup and recovery	77
72	4.2	Contract security considerations	78
73	4.3	Efficiency and scalability	78
74	4.3.1	Importance of performance	78
75	4.3.2	Cost centers	79
76	4.3.3	Client performance	79
77	4.3.4	Contract performance	80
78	4.3.5	Optimizing Blake2's circuit.	81
79	4.4	Encryption of the notes	88

80	A Transaction non malleability	90
81	A.1 Transaction malleability attack on Zeth	91
82	A.1.1 The attack	92
83	A.2 Solutions to address the transaction malleability attack	93
84	A.2.1 ZeroCash solution	93
85	A.2.2 Zcash’s solution	94
86	A.2.3 Solution on Ethereum	94
87	B Double spend attack on equivalent class	96
88	C Side-channel attacks and information leaks	97
89	C.1 Counterfeit data	97
90	C.2 Data leaked during synchronization	98
91	C.3 Queries on successful decryption	99
92	C.4 Invalid ciphertext	99
93	C.5 Using (and retrieving) nullifiers	100
94	C.6 Proof generation	101
95	C.7 Simple mixer calls	101
96	C.7.1 Small anonymity sets	102
97	D Security proofs of Blake2	104
98	D.1 Security model of Blake2	104
99	D.1.1 Weakly Ideal Cipher Model	104
100	D.2 Security proofs	106
101	D.2.1 Blake2 is a PRF	106
102	D.2.2 Proof of Blake2 collision resistance	107
103	D.2.3 Blake2 as a commitment scheme	108
104	D.2.4 Proof of commitment scheme security	109
105		

Change log

- 107 • **Version:** 0.0, **Date:** 04/12/2019, **Contributor:** Antoine Rondelet, **Descrip-**
108 **tion:** Creation of the document. Established initial table of content and started
109 to populate sections with bullet lists to develop in further versions of the document.
- 110 • **Version:** 0.1, **Date:** 20/12/2019, **Contributor:** Antoine Rondelet, **Descrip-**
111 **tion:** Refactored the structure of the document. Finalized the table of content,
112 wrote sections on notations and preliminaries, and introduced the content related
113 to the malleability fix.
- 114 • **Version:** 0.2, **Date:** 24/02/2020, **Contributor:** Clearmatics Cryptography R&D,
115 **Description:**
 - 116 – **Date:** 26/02/2020, **Contributor:** Duncan Tebbs, **Description:** Wrote sec-
117 tion on wallet implementation and side-channel attacks considerations.
 - 118 – **Date:** 02/03/2020, **Contributor:** Giuseppe Giffone, **Description:** Changed
119 Merkle tree hash function to MiMC compression function.
 - 120 – **Date:** 04/03/2020, **Contributor:** Duncan Tebbs, Michal Zajac, **Descrip-**
121 **tion:** Added background on Groth16 SNARK and SNARK scheme instanti-
122 ation in the protocol.
 - 123 – **Date:** 04/03/2020, **Contributor:** Raphael Toledo, **Description:** Wrote
124 section on the packing policy and corresponding attack.
 - 125 – **Date:** 04/03/2020, **Contributor:** Duncan Tebbs, **Description:** Refactored
126 the data structures preliminary section.
 - 127 – **Date:** 24/03/2020, **Contributor:** Raphael Toledo, **Description:** Changed
128 the PRF and commitment instantiation with Blake2s compression function.
 - 129 – **Date:** 17/04/2020, **Contributor:** Giuseppe Giffone, **Description:** Added
130 DHAES encryption scheme.
- 131 • **Version:** 0.3, **Date:** 09/06/2020, **Contributor:** Antoine Rondelet, **Descrip-**
132 **tion:** Fixed various inconsistencies throughout the document (notational mistakes
133 in document body and in proofs, latex macros, and typos).
- 134 • **Version:** 1.0, **Date:** 30/06/2020, **Contributor:** Antoine Rondelet, Duncan
135 Tebbs, Michal Zajac, **Description:** Global review of the document: fixed incon-
136 sistencies in definitions and notations, corrected grammatical mistakes and typos,

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

139 • **Version:** 1.1, **Date:** 06/11/2020, **Contributor:** Duncan Tebbs, **Description:**
140 Specification in terms of a generic curve, with constants provided for BN-254 and
141 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 $\text{STR} = \{a, b, \dots, y, z\}^*$ (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, \dots, f_{n-1} be n distinct elements of STR and let V_0, \dots, V_{n-1} be sets, for some $n \in \mathbb{N}$. We define the *structured data type* \mathbf{T} with fields $\{(f_i, V_i)\}_{i \in [n]}$ to be a set of values:

$$\mathbf{T} = V_0 \times \dots \times V_{n-1}$$

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

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

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 **Person** may be defined with fields:

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

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

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

$$\mathbf{a}.name = alice,$$

$$\mathbf{b}.age = 31,$$

$$\mathbf{b}.height = 1.74;$$

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.

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

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

Example 1.1.5. The values **a** and **b** in Example 1.1.3 might be written as follows:

$$\begin{aligned}\mathbf{a} &= \{name : \textit{alice}, age : 28, height : 1.65\} \\ \mathbf{b} &= \{name : \textit{bob}, age : 31, height : 1.74\}\end{aligned}$$

192 **Remark 1.1.6** (“dot” operators in assignment). The “dot” operators may be used in
193 algorithm descriptions to indicate *assignment to a specific component*. For example
194 $\mathbf{a}.age \leftarrow 29$ means that the value of the *age* field of **a** is replaced by the value 29.

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

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

is equivalent to:

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

195 We define one further operator and related assignment notation, convenient in cases
196 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 $[] : X^m \times [m] \rightarrow X$ as:

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

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

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

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 $\mathbf{x} = (x_0, \dots, x_{m-1})$ 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:

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

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

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

203 In this case, “dot” operators may be *chained*, so that $\mathbf{x}.f_j.f'_k$ refers to the k -th field
 204 v'_k of the j -th field v_j of $\mathbf{x} \in \mathbf{T}$.

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

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

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

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

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

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

208 1.1.2 Representations

209 The binary alphabet $\{0, 1\}$, denoted \mathbb{B} , is used to represent the presence or absence of an
 210 electrical signal in a computer. In fact, every piece of information in a computer is rep-
 211 resented as a string of bits. We assume the existence of an efficient binary representation
 212 for some set of primitive datatypes (such as the natural numbers \mathbb{N} , or alphanumeric

213 characters). Structured data types built up from primitive types (as described above)
 214 can then recursively be assigned similarly efficient representations. This is used to define
 215 the following functions to *encode* data to its bit-string representation, and to *decode* such
 216 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:

$$\begin{aligned}\text{encode}_X &: X \rightarrow \mathbb{B}^* \\ \text{decode}_X &: \mathbb{B}^* \rightarrow X \cup \perp\end{aligned}$$

satisfying

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

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

220 Without ambiguity, we overload the functions `encode` and `decode` to mean encode_X
 221 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,

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

222

223 1.2 Ethereum

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

234 To prevent unbounded state transitions in the state machine, each instruction exe-
 235 cuted by the EVM is associated with a cost in **Wei**, referred to as “the gas necessary to
 236 run the operation”. The “gas cost” of a transaction needs to be paid by the transaction
 237 originator (deduced from their account balance), and is awarded to the miner (added
 238 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 problem¹ 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
<i>nce</i>	The nonce of an account is a scalar value representing the number of transactions that have originated from the account, starting at 0.	$\mathbb{N}_{\text{ETHWORDLEN}}$
<i>bal</i>	The balance of an account is a scalar value representing the amount of Wei in the account.	$\mathbb{N}_{\text{ETHWORDLEN}}$
<i>sRoot</i>	The storage root is the Keccak256 hash representing the storage of the account.	$\mathbb{B}^{\text{KEK256DLEN}}$
<i>codeh</i>	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.	$\mathbb{B}^{\text{KEK256DLEN}}$

Table 1.1: Ethereum Account structure

¹https://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.

^aThe 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	$\mathbb{N}_{\text{ETHWORDLEN}}$
$gasP$	gasPrice	$\mathbb{N}_{\text{ETHWORDLEN}}$
$gasL$	gasLimit	$\mathbb{N}_{\text{ETHWORDLEN}}$
to	Recipient’s address	$\mathbb{B}^{\text{ADDRLEN}}$
val	Value of the transaction in Wei	$\mathbb{N}_{\text{ETHWORDLEN}}$
$init / data$	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	$\mathbb{B}^{\text{BYTELEN}}$
r	Field r of ECDSA signature [Por13]	$\mathbb{F}_{\text{PSECP}}$
s	Field s of ECDSA signature [Por13]	$\mathbb{F}_{\text{PSECP}}$

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).

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

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

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

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

```

TxGen( $sk_{\text{ECDSA}}, nce_{in}, gasP_{in}, gasL_{in}, to_{in}, val_{in}, init_{in}, data_{in}$ )
1 : if  $to_{in} = \emptyset$  do
2 :    $tx_{raw} \leftarrow \{nce : nce_{in}, gasP : gasP_{in}, gasL : gasL_{in}, to : to_{in}, val : val_{in}, init : init_{in}\};$ 
3 : else
4 :    $tx_{raw} \leftarrow \{nce : nce_{in}, gasP : gasP_{in}, gasL : gasL_{in}, to : to_{in}, val : val_{in}, data : data_{in}\};$ 
5 : endif
6 :  $\sigma_{\text{ECDSA}} \leftarrow \text{SigSch}_{\text{ECDSA}}.\text{Sig}(sk_{\text{ECDSA}}, \text{Keccak256}(tx_{raw}));$ 
7 :  $tx_{final} \leftarrow \{tx_{raw}, v : \sigma_{\text{ECDSA}}.v, r : \sigma_{\text{ECDSA}}.r, s : \sigma_{\text{ECDSA}}.s\};$ 
8 : 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 \emptyset 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 $\text{FS}(\cdot): \mathbb{B}^* \rightarrow \mathbb{B}^{4 \cdot \text{BYTELEN}}$ 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 `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 $tx.val + tx.gasP \cdot tx.gasL$.

Lifecycle 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 `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 executed²

²see <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 efficiently³ 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 $p(\text{length}(prim))$.

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 $\text{NIZK} = (\text{KGen}, P, V, \text{Sim})$. 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*

³Informally 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 deduced 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 $\text{prim} \in \mathbf{L}$ succeeds, i.e. gets his proof accepted by the verifier \mathbf{V} . Formally we require that for any λ , $\mathbf{R} \leftarrow \mathcal{R}(1^\lambda)$, $(\text{prim}, \text{aux}) \in \mathbf{R}$

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

Computational soundness which states that in case $\text{prim} \notin \mathbf{L}$ 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}

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

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

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

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 $\text{prim} \notin \mathbf{L}$) 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 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 $\text{Ext}_{\mathcal{A}}$, 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:

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

Second, zk-SNARKs are *succinct*, and so we require that proofs produced by 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}_{QAP} we provide some intuitions behind it. First, we observe that the circuit \mathbf{C} can be represented by three matrices $\vec{A}, \vec{B}, \vec{C}$ all in $\mathbb{F}_p^{\text{constNo} \times \text{inpNo} + 1}$ 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 $z \in \mathbb{F}_p^{\text{inpNo} + 1}$ the left input for the i -th gate is $\sum_{j=0}^{\text{inpNo}} A_{ij} z_j$ and the right input is $\sum_{j=0}^{\text{inpNo}} B_{ij} z_j$. 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 $j \in \{0, \dots, \text{inpNo}\}$, such that $u_j(\omega^i) = \vec{A}_{ij}$;
- $v_j(X)$, for $j \in \{0, \dots, \text{inpNo}\}$, such that $v_j(\omega^i) = \vec{B}_{ij}$;
- $w_j(X)$, for $j \in \{0, \dots, \text{inpNo}\}$, such that $w_j(\omega^i) = \vec{C}_{ij}$.

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

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

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.

411

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 $\text{DAP} = (\text{Setup}, \text{GenAddr}, \text{SendTx}, \text{VerifyTx}, \text{Receive})$ 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

418
419

420 party. The resulting public parameters pp are published and made available to all
 421 parties.

422 **Creating Zeth addresses** The algorithm `GenAddr` takes as input the public parame-
 423 ters 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 ad-
 424 dress” (Table 1.4), and $Addr_{sk}$ is an object referred to as the “private address”
 425 (Table 1.5) [ZCa19].
 426

427 **Transfer notes** The algorithm `SendTx` is used to transfer some public input vin as
 428 well as the value of a set of input (“old”) $Notes$ into a set of output (“new”) $Notes$
 429 as well as some public output value $vout$. The inputs $Notes$ are marked as
 430 “consumed” (alternatively, we say that the input $Notes$ are “spent”). `SendTx` takes
 431 as inputs the public parameters pp , the input value and the input (“old”) $Notes$
 432 to be transferred, as well as the Merkle root and the Merkle authentications paths
 433 of the commitments to the input $Notes$, the “spending keys” related to the input
 434 $Notes$, the output value to create and the “payment addresses” for the output
 435 (“new”) $Notes$. If the joinsplit equation is satisfied, the algorithm returns the new
 436 $Notes$ and the corresponding **Ethereum** transaction tx , else it returns \perp .

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

440 **Receiving notes** The algorithm `Receive` scans the ledger L and retrieves unspent $Notes$
 441 paid to a particular user address. It takes as input the recipient address key pair
 442 $\{pub : Addr_{pk}, priv : Addr_{sk}\}$ and the current ledger L and outputs the set of
 443 (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!*

444

Field	Description
apk	The <i>paying key</i>
$pkenc$	The <i>transmission key</i>

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

Field	Description
<i>ask</i>	The <i>spending key</i>
<i>skenc</i>	The <i>receiving key</i>

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

445 **Zeth** leverages zk-SNARKs (Section 1.3) and the possibility to deploy smart-contracts
 446 to specify privacy-preserving state transitions altering the **Ethereum** state ς (Section 1.2).
 447 As such, **Zeth** defines a smart-contract, $\widetilde{\mathbf{Mixer}}$, that keeps track of the set of *ZethNotes*
 448 (Section 2.1) in a committed form, stored in a Merkle tree; and which verifies the va-
 449 lidity of the state transitions generated by the **Zeth** users. As such a **Zeth** DAP is
 450 entirely determined by $\widetilde{\mathbf{Mixer}}$, the instance of the mixer smart-contract deployed on the
 451 **Ethereum** ledger. State transitions are executed on-chain by calling the **Mix** function of
 452 $\widetilde{\mathbf{Mixer}}$, which implements the algorithm **VerifyTx** of DAP, and which modifies ς iff the
 453 transaction is deemed valid.

Note

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

$$zdata = FS(\mathbf{Mix}) \parallel Mix_{in}$$

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

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

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

454

1.5 Definitions

455

1.5.1 Negligible function

456

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

460 1.5.2 Basic algebra notions

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

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

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

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

468 **Definition 1.5.3** (Finite Cyclic Group, adapted from [KL14, Sections 7.1.3, 7.3.2]). A
 469 finite cyclic group is given by a tuple $(q, \mathbb{G}, \mathbf{g}, \otimes)$, called the *group description*, where \mathbb{G}
 470 represents the set of group elements, \mathbf{g} is a generator and q is the order. The generator
 471 \mathbf{g} generates the group; namely, each $\mathbf{h} \in \mathbb{G}$ can be expressed by the generator as $\mathbf{h} =$
 472 $\mathbf{g} \otimes \dots \otimes \mathbf{g}$. Given a scalar x , we denote by $\llbracket x \rrbracket$ the *encoding* of x in \mathbb{G} : i.e. $\llbracket x \rrbracket = \mathbf{g} \otimes \dots \otimes \mathbf{g}$
 473 $(x \text{ times})$. As consequence, $\llbracket 1 \rrbracket = \mathbf{g}$.

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

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

481 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 $\log_{\mathbf{g}}(h)$ denote
 the discrete logarithm of h in the basis \mathbf{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

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

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

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

Definition 1.5.6 (One More Discrete Log Problem (om-DLog), cf. [PV05]). Let \mathbb{G} denote a group whose order p is prime and written over λ bits. We let $\log_{\mathbf{g}}(h)$ denote the discrete logarithm of h in the basis \mathbf{g} . A PPT adversary \mathcal{A} solving the om-DLog is given $q + 1$ random group elements as well as limited access to a discrete logarithm oracle $\text{O}^{\text{DLog}_{\mathbf{g}}}(q)$. \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 \mathbf{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

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

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

486 1.5.4 Symmetric encryption

487 **Definition 1.5.7** (Symmetric Encryption,[KL14, Definition 3.8]). A symmetric encryp-
 488 tion scheme Sym is given by a tuple of PPT algorithms $(\text{KGen}, \text{Enc}, \text{Dec})$ where:

- 489 • KGen , the key generation algorithm, takes a security parameter 1^λ and outputs a
 490 secret key ek ; we assume, without loss of generality, that $kLen(\lambda) = \text{length}(ek) \geq \lambda$.
 491 Note that $kLen(\lambda)$ is a polynomial function in λ .⁵
- 492 • Enc , the encryption algorithm, takes a key ek , a plaintext $m \in \{0, 1\}^*$ and returns
 493 a ciphertext ct .
- 494 • Dec , the decryption algorithm, takes a key ek and a ciphertext ct , and returns a
 495 message m . We assume, without loss of generality, that Dec is deterministic.

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

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

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

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

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

```

IND-CPA( $\lambda$ )
 $ek \leftarrow \text{KGen}(1^\lambda)$ 
 $(m_0, m_1, \text{state}) \leftarrow \mathcal{A}^{\text{O}^{\text{Enc}_{ek}}} \text{ with } \text{length}(m_0) = \text{length}(m_1)$ 
 $b \leftarrow \$\{0, 1\}$ 
 $ct \leftarrow \text{Enc}(ek, m_b)$ 
 $\tilde{b} \leftarrow \mathcal{A}^{\text{O}^{\text{Enc}_{ek}}}(ct, \text{state})$ 
return  $\tilde{b} = b$ 

```

Figure 1.2: IND-CPA game for Sym.

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

504 1.5.5 Asymmetric encryption

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

- 507 • KGen , the key generation algorithm, takes a security parameter 1^λ and returns a
508 pair of keys (sk, pk) . We refer to the first of these as the *private key* and the second
509 as the *public key*. We assume for convenience that pk and sk each have length at
510 least λ , and that λ can be determined from pk, sk ;
- 511 • Enc , the encryption algorithm, takes a public key pk , a plaintext m , from some
512 underlying plaintext space (that may depend on pk) and returns a ciphertext ct ;
- 513 • Dec , the decryption algorithm, takes a private key sk and a ciphertext ct , and
514 returns a message m or a special symbol \perp denoting decryption failure. We assume,
515 without loss of generality, that Dec is deterministic.

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

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

Definition 1.5.10 (IK-CCA). Let $\text{Asym} = (\text{KGen}, \text{Enc}, \text{Dec})$ 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 $\text{O}^{\text{Dec}_{sk_0}}$ or $\text{O}^{\text{Dec}_{sk_1}}$ on the challenge ciphertext

IK-CCA(λ)
 $(sk_0, pk_0), (sk_1, pk_1) \leftarrow \text{KGen}(1^\lambda)$
 $(m, state) \leftarrow \mathcal{A}^{\mathcal{O}^{\text{Dec}_{sk_0}}, \mathcal{O}^{\text{Dec}_{sk_1}}}(pk_0, pk_1)$
 $b \leftarrow \$\{0, 1\}$
 $ct \leftarrow \text{Enc}(pk_b, m)$
 $\tilde{b} \leftarrow \mathcal{A}^{\mathcal{O}^{\text{Dec}_{sk_0}}, \mathcal{O}^{\text{Dec}_{sk_1}}}(ct, state)$
return $\tilde{b} = b$

Figure 1.3: IK-CCA game.

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

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

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

524 1.5.6 Block cipher-based compression functions

525 **Definition 1.5.11.** Let $kl, il > 1$. A *block cipher* is a map $E: \{0, 1\}^{kl} \times \{0, 1\}^{il} \rightarrow \{0, 1\}^{il}$
 526 where, for each key $k \in \{0, 1\}^{kl}$, the function $E_k(\cdot) = E(k, \cdot)$ is a permutation on $\{0, 1\}^{il}$.
 527 If E is a block cipher then E^{-1} is its inverse, that on input (k, y) returns m such that
 528 $E_k(m) = y$.

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

535 **Definition 1.5.12** (Ideal Cipher Model [HKT11]). The Ideal Cipher Model (ICM),
 536 is a security model where all parties are granted access to an ideal cipher $E: \{0, 1\}^{kl} \times$
 537 $\{0, 1\}^{il} \rightarrow \{0, 1\}^{il}$, a random primitive such that $E(k, \cdot)$ for $k \in \{0, 1\}^{kl}$ are 2^{kl} independent
 538 random permutations.

539 For fixed kl and il , each party is given access to the oracles \mathcal{O}^E and $\mathcal{O}^{E^{-1}}$, simulating
 540 E and E^{-1} , which can be queried for encryption and decryption a polynomial number
 541 of times. The encryption oracle takes as input a key, $k \in \{0, 1\}^{kl}$, and a preimage,
 542 $m \in \{0, 1\}^{il}$, and returns a tuple comprising the image, $y \in \{0, 1\}^{il}$, along with the
 543 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

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

Figure 1.4: Oracles of an ideal block cipher, with Table_O being a table of tuples (key, preimage, image) of queries already answered by the oracle.

544 at random from $\{0, 1\}^{\text{il}}$ and added to the oracle's table. Otherwise, the oracle returns
545 y associated with query (k, m) in its table. The decryption oracle is defined similarly
546 with the image and key defined as inputs and the preimage chosen randomly, for details
547 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

$$f: \mathcal{BK}(\text{kl}, \text{il}) \times \{0, 1\}^a \times \{0, 1\}^b \rightarrow \{0, 1\}^c$$

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

550 **Remark 1.5.14.** We use f_E to denote a block cipher-based compression function f
551 restricted to a given block cipher E , i.e. $f_E: \{0, 1\}^a \times \{0, 1\}^b \rightarrow \{0, 1\}^c$ and $f_E = f(E, \cdot)$,
552 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

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

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

555 1.5.7 Hash functions

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

- 558 • **Setup** is a PPT algorithm which takes as input a security parameter 1^λ and outputs
559 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 $H(hk, x) = H_{hk}(x) \in \{0, 1\}^{hLen}$, 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 $\mathcal{H} = (\text{Setup}, H)$ is collision resistant if for all PPT adversaries \mathcal{A} there exists a negligible function $\text{negl}(\lambda)$ such that:

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

HDHI and HDHI2 assumptions

The Hash Diffie-Hellmann Independence (HDHI) assumption states that, given H in \mathcal{H} and a group description $(p, \mathbb{G}, \mathbf{g}, \otimes)$, for $\llbracket u \rrbracket$ and $\llbracket v \rrbracket$, with u, v sampled at random, it is hard for an attacker to distinguish $H(\llbracket u \rrbracket \parallel \llbracket uv \rrbracket)$ 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 $\mathcal{O}^{\text{HDHI}_v}$ that on input $\mathbf{r} \in \mathbb{G}$ returns $H(\mathbf{r} \parallel v \cdot \mathbf{r})$ (queries on $\llbracket u \rrbracket$ 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:

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

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:

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

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

⁸Note 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.

⁹In [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.

$\text{HDHI}(\lambda)$
 $hk \leftarrow \mathcal{H}.\text{Setup}(1^\lambda)$
 $(q, \mathbb{G}, \mathfrak{g}, \otimes) \leftarrow \text{SetupG}(1^\lambda)$
 $u, v \leftarrow \$[q]$
 $w_0 \leftarrow \mathcal{H}.H_{hk}(\llbracket u \rrbracket \parallel \llbracket uv \rrbracket)$
 $w_1 \leftarrow \$\{0, 1\}^{hLen}$
 $b \leftarrow \$\{0, 1\}$
 $\tilde{b} \leftarrow \mathcal{A}^{\text{O}^{\text{HDHI}_v}}(\llbracket u \rrbracket, \llbracket v \rrbracket, w_b)$
return $\tilde{b} = b$

Figure 1.5: HDHI game.

$\text{HDHI2}(\lambda)$
 $hk \leftarrow \mathcal{H}.\text{Setup}(1^\lambda)$
 $(q, \mathbb{G}, \mathfrak{g}, \otimes) \leftarrow \text{SetupG}(1^\lambda)$
 $u, v_0, v_1 \leftarrow \$[q]$
 $w_{0,0} \leftarrow \mathcal{H}.H_{hk}(\llbracket u \rrbracket \parallel \llbracket uv_0 \rrbracket), w_{0,1} \leftarrow \mathcal{H}.H_{hk}(\llbracket u \rrbracket \parallel \llbracket uv_1 \rrbracket)$
 $w_{1,0} \leftarrow \$\{0, 1\}^{hLen}, w_{1,1} \leftarrow \$\{0, 1\}^{hLen}$
 $b \leftarrow \$\{0, 1\}$
 $\tilde{b} \leftarrow \mathcal{A}^{\text{O}^{\text{HDHI}_{v_0}}, \text{O}^{\text{HDHI}_{v_1}}}(\llbracket u \rrbracket, \llbracket v_0 \rrbracket, \llbracket v_1 \rrbracket, w_{b,0}, w_{b,1})$
return $\tilde{b} = b$

Figure 1.6: HDHI2 game.

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

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

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 $H(\llbracket u \rrbracket \parallel \llbracket uv \rrbracket)$ and $\{H(\llbracket u \rrbracket \parallel \llbracket uv_0 \rrbracket), H(\llbracket u \rrbracket \parallel \llbracket uv_1 \rrbracket)\}$ instead of $H(\llbracket uv \rrbracket)$ and $\{H(\llbracket uv_0 \rrbracket), H(\llbracket uv_1 \rrbracket)\}$. By accordingly changing the instances of H in the games G_0, G_1, G_2 of [ABN10, Lemma 6.1] our lemma follows. \square

1.5.8 Pseudo Random Functions

Informally, a pseudorandom function family $\mathcal{PRF} = \{\text{PRF}_k : D \rightarrow C\}_{k \in \mathcal{K}}$ 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 $\mathcal{F} : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ 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:

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

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:

- 595 • **Setup**, is a PPT algorithm that takes a security parameter 1^λ and outputs public
596 parameters pp .
- 597 • **Com**, is a polynomial-time algorithm that takes a message $m \in \mathbb{B}^{\text{il}}$, a random coin
598 $r \in \mathbb{B}^{\text{nl}}$ and outputs a commitment $cm \in \mathbb{B}^{\text{ol}}$.

599 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:

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

600 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:

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

601 is at most negligible in λ .

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

604 1.5.10 Digital Signature

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

- 607 • $(sk, vk) \leftarrow \text{KGen}(1^\lambda)$. Key Generation randomized algorithm takes as input the
608 security parameter 1^λ and returns a signing key sk and verifying key vk .
- 609 • $\sigma \leftarrow \text{Sig}(sk, m)$. Given a signing key sk and a message m , the **Sig** algorithm
610 computes and outputs a signature σ .
- 611 • $\{0, 1\} \leftarrow \text{Vf}(vk, m, \sigma)$. Given a verification key vk , a message m and a signature
612 σ , the **Vf** algorithm returns 1 if σ is a valid signature else 0.

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

UF-CMA($1^\lambda, t, q$)

```

1 :   $(sk, vk) \leftarrow \text{KGen}(1^\lambda)$ 
2 :   $state \leftarrow \mathcal{A}^{\text{OSig}_{sk}}(vk, \cdot)$ 
3 :  //  $state = \{(m_i, \sigma_i)\}_{i \in [q]}$  where  $m_i$  denotes
4 :  // the  $i$ th query made to  $\text{OSig}_{sk}$  and
5 :  //  $\sigma_i$  denotes the  $i$ th oracle answers
6 :   $(m^*, \sigma^*) \leftarrow \mathcal{A}(state)$ 
7 :  return  $\text{Vf}(vk, m^*, \sigma^*) = 1$ 
8 :   $\wedge m^* \notin \{m_i\}_{i \in [q]}$ 

```

Figure 1.7: UF-CMA game

SUF-CMA($1^\lambda, t, q$)

```

1 :   $(sk, vk) \leftarrow \text{KGen}(1^\lambda)$ 
2 :   $state \leftarrow \mathcal{A}^{\text{OSig}_{sk}}(vk, \cdot)$ 
3 :  //  $state = \{(m_i, \sigma_i)\}_{i \in [q]}$  where  $m_i$  denotes
4 :  // the  $i$ th query made to  $\text{OSig}_{sk}$  and
5 :  //  $\sigma_i$  denotes the  $i$ th oracle answers
6 :   $(m^*, \sigma^*) \leftarrow \mathcal{A}(state)$ 
7 :  return  $\text{Vf}(vk, m^*, \sigma^*) = 1$ 
8 :   $\wedge (m^*, \sigma^*) \notin \{(m_i, \sigma_i)\}_{i \in [q]}$ 

```

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 $(\text{KGen}, \text{Tag}, \text{Vf})$ 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).

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

SUF-CMA (λ)
 $mk \leftarrow \text{KGen}(1^\lambda)$
 $(\bar{y}, \bar{\tau}) \leftarrow \mathcal{A}^{\text{O}^{\text{Tag}_{mk}}, \text{O}^{\text{Vf}_{mk}}}$
return $\text{Vf}(mk, \bar{y}, \bar{\tau}) = 1$

Figure 1.9: SUF-CMA game.

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

645 A security notion for MAC follows:

646 **Definition 1.5.29** (SUF-CMA, [ABR99, Section 3.2.3]). Let $\text{MAC} = (\text{KGen}, \text{Tag}, \text{Vf})$ be
 647 a message authentication scheme and let \mathcal{A} be an adversary. Consider the SUF-CMA
 648 game described in Figure 1.9, with the condition that $\text{Tag}(mk, \bar{y}) \neq \bar{\tau}$. We say that an
 649 adversary \mathcal{A} has *forged* a tag when it outputs a pair $(\bar{y}, \bar{\tau})$ such that $\text{Vf}_k(\bar{y}, \bar{\tau}) = 1$, where
 650 $(\bar{y}, \bar{\tau})$ was not previously obtained via a query to the tag oracle.

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

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

651 We say that MAC is SUF-CMA secure if for every PPT adversary \mathcal{A} the advantage
 652 $\text{Adv}_{\text{MAC}, \mathcal{A}}^{\text{suf-cma}}(\lambda)$ 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}_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	$\mathbb{B}^{\text{PRFADDRROUTLEN}}$
r	Note randomness	$\mathbb{B}^{\text{RTRAPLEN}}$
v	Note value	$\mathbb{B}^{\text{ZVALUELEN}}$
ρ	Note identifier	$\mathbb{B}^{\text{PRFRHOOUTLEN}}$

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 **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
<i>mkpath</i>	Merkle authentication path to the commitment corresponding to the <i>ZethNote</i> to spend	$(\mathbb{F}_r)^{\text{MKDEPTH}}$
<i>mkaddr</i>	Commitment address in the Merkle tree	$\mathbb{B}^{\text{MKDEPTH}}$
<i>znote</i>	Zeth note object	ZethNoteDType
<i>cm</i>	Zeth note commitment	\mathbb{F}_r
<i>ask</i>	Note owner's spending key	$\mathbb{B}^{\text{ASKLEN}}$
<i>nf</i>	Note nullifier	$\mathbb{B}^{\text{PRNFOUTLEN}}$

Table 2.2: JSInputDType data type

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

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

Table 2.3: PrimInputDType data type

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

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

Table 2.4: **AuxInputDType** data type

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

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

Table 2.5: **MixInputDType** data type

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

Field	Description	Data type
<i>mkroot</i>	New root of Merkle tree of commitments	\mathbb{F}_r
<i>nfs</i>	Nullifiers for input notes consumed	$(\mathbb{B}^{\text{PRNFOUTLEN}})^{\text{JSIN}}$
<i>cms</i>	Commitments to the output notes	$(\mathbb{F}_r)^{\text{JSOUT}}$
<i>ciphers</i>	Ciphertexts for the output notes	$(\mathbb{B}^{\text{ENCZETHNOTELEN}})^{\text{JSOUT}}$

Table 2.6: **MixEventDType** data type

690 2.2 Zeth statement

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

- 696 • For each $i \in [\text{JSIN}]$:
 - 697 1. $\text{aux.jsins}[i].\text{znote.apk} = \text{PRF}_{\text{aux.jsins}[i].\text{ask}}^{\text{addr}}(0)$
 - 698 2. $\text{aux.jsins}[i].\text{cm} = \text{ComSch.Com}(\text{aux.jsins}[i].\text{znote.apk}, \text{aux.jsins}[i].\text{znote}.\rho, \text{aux.jsins}[i].\text{znote}.v;$
 699 $\text{aux.jsins}[i].\text{znote}.r)$
 - 700 3. $\text{aux.jsins}[i].\text{nf} = \text{PRF}_{\text{aux.jsins}[i].\text{ask}}^{\text{nf}}(\text{aux.jsins}[i].\text{znote}.\rho)$
 - 701 4. $\text{aux.htags}[i] = \text{PRF}_{\text{aux.jsins}[i].\text{ask}}^{\text{pk}}(i, \text{aux.hsigs})$ (non-malleability, see Appendix A)
 - 702 5. $(\text{aux.jsins}[i].\text{znote}.v) \cdot (1 - e) = 0$ is satisfied for the boolean value e set such
 703 that if $\text{aux.jsins}[i].\text{znote}.v > 0$ then $e = 1$.
 - 704 6. The Merkle root mkroot' obtained after checking the Merkle authentica-
 705 tion path $\text{aux.jsins}[i].\text{mkpath}$ of commitment $\text{aux.jsins}[i].\text{cm}$, with MKHASH,
 706 equals to prim.mkroot if $e = 1$.
 - 707 7. $\text{prim.nfs}[i]$
 708 $= \{\text{Pack}_{\mathbb{F}_r}(\text{aux.jsins}[i].\text{nf}[k \cdot \text{FIELD CAP}:(k+1) \cdot \text{FIELD CAP}])\}_{k \in [\lfloor \text{PRNFOUTLEN}/\text{FIELD CAP} \rfloor]}$
 709 (see Section 3.3.1 for definition of Pack)
 - 710 8. $\text{prim.htags}[i]$
 711 $= \{\text{Pack}_{\mathbb{F}_r}(\text{aux.htags}[i][k \cdot \text{FIELD CAP}:(k+1) \cdot \text{FIELD CAP}])\}_{k \in [\lfloor \text{PRFPKOUTLEN}/\text{FIELD CAP} \rfloor]}$
 712 (see Section 3.3.1 for definition of Pack)
- 713 • For each $j \in [\text{JSOUT}]$:

- 714 1. $aux.znotes[j].\rho = \text{PRF}_{aux.\phi}^{\text{rho}}(j, aux.hsigs)$ (non-malleability, see Appendix A)
- 715 2. $prim.cms[j] = \text{ComSch.Com}(aux.znotes[j].apk, aux.znotes[j].\rho, aux.znotes[j].v;$
- 716 $aux.znotes[j].r)$
- 717 • $prim.hsigs = \{\text{Pack}_{\mathbb{F}_r}(aux.hsigs[k \cdot \text{FIELD CAP}:(k+1) \cdot \text{FIELD CAP}])\}_{k \in [\lfloor \text{CRHHSIGOUTLEN}/\text{FIELD CAP} \rfloor]}$
- 718 (see Section 3.3.1 for definition of Pack)
- 719 • $prim.rsd = \text{Pack}_{rsd}(\{aux.jsins[i].nf\}_{i \in [\text{JSIN}]}, aux.vin, aux.vout, aux.hsigs, \{aux.htags[i]\}_{i \in [\text{JSIN}]})$
- 720 (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

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

721 2.3 Generating the inputs of the Mix function (Mix_{in})

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

730 In order for \mathcal{U}_Z to transact via **Zeth**, \mathcal{U}_Z needs to create an object Mix_{in} of type
 731 **MixInputDType** to pass to the Mix function of **Mixer**:

- 732 1. Create an object $prim$ of type **PrimInputDType** to represent the primary input,
- 733 and an object aux of type **AuxInputDType** to represent the auxiliary input, where:
 - 734 (a) $prim.mkroot \in \text{Roots}$, where Roots is the set of *all* Merkle roots corresponding
 735 to one of the state of the Merkle tree on **Mixer** containing *all* the commit-
 736 ments to the input notes, in $aux.jsins$, in its set of leaves.
 - 737 (b) $aux.znotes[j].r \leftarrow \mathbb{B}^{\text{RTRAPLEN}}, \forall j \in [\text{JSOUT}]$, and $aux.\phi \leftarrow \mathbb{B}^{\text{PHILEN}}$
 - 738 (c) The public values $(aux.vin, aux.vout) \in (\mathbb{B}^{\text{ZVALUELEN}})^2$, $aux.znotes[j].v$ and
 739 $aux.znotes[j].apk \forall j \in [\text{JSOUT}]$ are all set by the sender, \mathcal{U}_Z , as desired as
 740 long as they satisfy the joinsplit equation.

¹where $\text{Pack}_{\mathbb{F}_r}(x)$ outputs the numerical value of x in \mathbb{F}_r . We rely on the fact that $\text{ZVALUELEN} < \text{FIELD CAP}$ 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 $\text{SigSch}_{\text{OT-SIG}}$ (see Section 3.4):

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

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

- 741 (e) $\text{Mix}_{in}.primIn \leftarrow prim$

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.

742

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

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

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

- 748 3. Encrypt all the *aux.znotes* using the recipient's *payment address*, using the en-
749 cryptation scheme EncSch (see Section 3.5).

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

$$ct_j \leftarrow \text{EncSch}.\text{Enc}(aux.znotes[j], \mathcal{U}_Z.\text{keystore}[recipient_j].pub.pk_{enc})$$

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

- 751 4. Generate a signature $\sigma_{\text{OT-SIG}}$ on the inputs of the **Mix** function, in order to prevent
752 any malleability attacks (c.f. Appendix A), using the desired signature scheme
753 $\text{SigSch}_{\text{OT-SIG}}$ (see Section 3.4):

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

$$\begin{aligned} dataToBeSigned &= \mathcal{S}_E.Addr \parallel \text{Mix}_{in}.primIn \parallel \text{Mix}_{in}.\pi \parallel \text{Mix}_{in}.ciphers \\ \sigma_{\text{OT-SIG}} &= \text{SigSch}_{\text{OT-SIG}}.\text{Sig}(sk_{\text{OT-SIG}}, \text{CRH}^{\text{ots}}(dataToBeSigned)) \end{aligned}$$

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

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

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

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

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

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

Then, an **Ethereum** user \mathcal{S}_E can ECDSA sign tx_{raw} , under $\mathcal{S}_E.sk$ 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}_E who sends the final transaction, and the **Zeth** user \mathcal{U}_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}_Z to create a **Zeth** transaction which is later signed by a distinct party \mathcal{S}_E . In particular, the only identifying information that appears in plaintext on the ledger will be that of \mathcal{S}_E .

2.5 Processing tx_{Mix}

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

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

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

Note

Even if **ZethVerifyTx**(tx_{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 **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’ ($\text{Mix}_{in}.primIn$) entries are elements of the scalar field over which the zk-proof is generated:

$$\text{Mix}_{in}.primIn \in \mathbb{F}_{\mathbf{r}}^*$$

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

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

- 778 3. Check the validity of the tx_{Mix} object (`ZethVerifyTx`):

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

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

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

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

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

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

779 where Nulls is the set of all nullifiers that are “declared” on $\widetilde{\text{Mixer}}$.

- Check that $\text{Mix}_{in}.otssig$ 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:

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

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

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

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

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

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

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

- 784 • Add the commitments $\text{Mix}_{in}.primIn.cms$ to the Merkle tree held on $\widetilde{\text{Mixer}}$.
- 785 • $Roots' \leftarrow Roots' \cup \{mkroot'\}$, where $mkroot'$ is the Merkle root of the Merkle
- 786 tree after insertion of the commitments $\text{Mix}_{in}.primIn.cms$ in the Merkle tree.
- 787 • $Nulls \leftarrow Nulls \cup \{nf_i\}_{i \in [JSIN]}$, i.e. the nullifiers nfs become “declared”.
- 788 • Modify the **Ethereum** balances according to the public values:
 - 789 – $\varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal = \varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal - vin$
 - 790 – $\varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal = \varsigma[\mathcal{S}_{\mathcal{E}}.Addr].bal + vout$
 - 791 – $\widetilde{\text{Mixer}}.bal = \widetilde{\text{Mixer}}.bal + vin$
 - 792 – $\widetilde{\text{Mixer}}.bal = \widetilde{\text{Mixer}}.bal - vout$
- 793 • Emit an event (Section 1.2.3) $evMixOut$ of type **MixEventDType**, contain-
- 794 ing the new root $mkroot'$ of the Merkle tree of commitments, the nullifiers
- 795 $\{nf_i\}_{i \in [JSIN]}$, commitments to the newly created *ZethNotes* $\text{Mix}_{in}.primIn.cms$,
- 796 and the corresponding ciphertexts $\text{Mix}_{in}.primIn.ciphers$.

797 2.6 Receiving *ZethNotes*

798 In order to confirm the reception of *ZethNotes*, \mathcal{R}_Z must listen to the events (Sec-
 799 tion 1.2.3) of type **MixEventDType** emitted by the processing of tx_{Mix} , and try to decrypt
 800 the ciphertexts using $\mathcal{R}_Z.priv.skenc$ to see if he is the recipient of a **Zeth** payment. If
 801 the decryption is successful (\mathcal{R}_Z is the recipient of a payment), \mathcal{R}_Z must verify that the
 802 *ZethNote* recovered is the opening of a commitment in the Merkle tree of $\widetilde{\text{Mixer}}$. If not,
 803 \mathcal{R}_Z rejects the (invalid) payment.

804 We describe below the steps that \mathcal{R}_Z needs to carry out for all events $evMixOut \in$
 805 **MixEventDType** emitted by $\widetilde{\text{Mixer}}$, in order to receive payments:

- 806 1. Compute the new root $mkroot'$ of the Merkle tree of commitments, after adding the
- 807 new values $evMixOut.cms$. If this value does not match the new root $evMixOut.mkroot$
- 808 emitted by $\widetilde{\text{Mixer}}$, abort.

2. Try to decrypt the ciphertexts:

$$zn_j = \text{EncSch.Dec}(\mathcal{R}_Z.priv.skenc, evMixOut.ciphers[j])$$

- 809 3. For each successful decryption, let j be the index of the decrypted ciphertext:
 - 810 (a) Check whether the recovered plaintext zn_j is a well-formed *ZethNote*. Abort
 - 811 if it is not well-formed.
 - (b) Check that the recovered *ZethNote* zn_j is the opening of the corresponding
 - commitment $evMixOut.cms[j]$:

$$evMixOut.cms[j] = \text{ComSch.Com}(zn_j.apk, zn_j.\rho, zn_j.v; zn_j.r)$$

812 Abort if the note is not a valid opening.

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

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

820 2.7 Security requirements for the primitives

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

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

832 2.7.1 Additional notes

833 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

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

834 As a private transaction is uniquely determined by its nullifiers $\text{nfs} = (\text{nf}_0, \dots, \text{nf}_{\text{JSIN}-1})$,
 835 and because of the collision resistance of CRH^{hsig} , a transaction is uniquely determined
 836 by *hsig* (with overwhelming probability). We did not use the *randomSeed* defined in
 837 **Zcash** however, since this is only necessary to achieve uniqueness of *hsig* for transactions
 838 *in transit* (i.e. not mined yet) [Hop16]. The uniqueness of *hsig* is a requirement to
 839 prevent the Fairy Gold attack.

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

840 **Security Requirement.**

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

844 **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,

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

845 **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

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

846

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}_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 it is 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 appropriate “outer” pairing. Support for **BLS12-377** in **Zeth** therefore admits several applications (not 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 CRH^{hsig} and CRH^{ots} are instantiated with SHA256 [oST15] which we assume to be collision resistant. Furthermore, ComSch , $\text{PRF}^{\text{pk}}(x)$, $\text{PRF}^{\text{rho}}(x)$, $\text{PRF}^{\text{addr}}(x)$, and $\text{PRF}^{\text{nf}}(x)$ 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 $\text{PRF}^{\text{pk}}(x)$, $\text{PRF}^{\text{rho}}(x)$, $\text{PRF}^{\text{addr}}(x)$, and $\text{PRF}^{\text{nf}}(x)$ 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:

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

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 security¹. 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 indistinguishability 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

¹The 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]).

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

Note

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

913

914 3.1.2 Commitment scheme

We define our commitment scheme as follows,

$$\begin{aligned} \text{ComSch.Setup} &: \{1^\lambda \text{ s.t. } \lambda \in \mathbb{N}\} \rightarrow \mathbb{B}^* \\ \text{ComSch.Com} &: (\mathbb{B}^{\text{PRFADDRROUTLEN}} \times \mathbb{B}^{\text{PRFRHOOUTLEN}} \times \mathbb{B}^{\text{ZVALUELEN}}) \times \mathbb{B}^{\text{RTRAPLEN}} \rightarrow \mathbb{F}_r \end{aligned}$$

We instantiate the commitment scheme with Blake2s as follows,

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

915 **Remark 3.1.1.** We set the commitment digest length in the parameter block PB [MJS15].

916 Security proof

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

920 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,

$$\begin{aligned} \text{PRF}^{\text{addr}} &: \mathbb{B}^{\text{ASKLEN}} \times \{0\} \rightarrow \mathbb{B}^{\text{PRFADDRROUTLEN}} \\ \text{PRF}^{\text{pk}} &: (\mathbb{B}^{\text{ASKLEN}} \times [\text{JSIN}]) \times \mathbb{B}^{\text{CRHHSIGOUTLEN}} \rightarrow \mathbb{B}^{\text{PRFPKOUTLEN}} \\ \text{PRF}^{\text{nf}} &: \mathbb{B}^{\text{ASKLEN}} \times \mathbb{B}^{\text{PRFRHOOUTLEN}} \rightarrow \mathbb{B}^{\text{PRFNFOUTLEN}} \\ \text{PRF}^{\text{rho}} &: (\mathbb{B}^{\text{PHILEN}} \times [\text{JSOUT}]) \times \mathbb{B}^{\text{CRHHSIGOUTLEN}} \rightarrow \mathbb{B}^{\text{PRFRHOOUTLEN}} \end{aligned}$$

As we instantiate the PRFs with Blake2s we have,

$$\text{PRFADDRROUTLEN, PRFNFOUTLEN, PRFPKOUTLEN, PRFRHOOUTLEN} = \text{BLAKE2sCLEN}$$

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,

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

The tagging functions are instantiated as follows,

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

921 where $\text{pad}_{\lceil \frac{\text{JSMAX}}{2} \rceil}(\text{encode}_{\mathbb{N}}(i))$ is the function that pads the binary representation of i by
 922 adding 0's before the most significant bit (e.g. assuming big endian encoding, $\text{pad}_2(\text{encode}_{\mathbb{N}}(1)) =$
 923 01).

We now present how the PRFs are instantiated,

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

924 **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 $\lambda = \text{BLAKE2sCLEN}/2 - \text{JSMAX}/4 - 3/2$.

3.1.4 Collision resistant hashes

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

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

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

957 exponent e (see Section 3.2.1). Note that *rounds* depends on the desired security level
 958 λ . We denote the encryption function by **MIMC-Encrypt** and illustrate it in Fig. 3.1.

```

MIMC-Encrypt( $k, m, c, e, rounds$ )
1: foreach  $i \in [rounds]$  :
2:    $m \leftarrow (k \text{ OP } c[i] \text{ OP } m)^e$ 
3: return ( $m \text{ OP } k$ )

```

Figure 3.1: MIMC Encryption function.

MIMC-Encrypt can be defined on both binary and prime fields. As such, the OP operation corresponds to either \oplus or $+$ (mod p) [AGR⁺16, GRR⁺16]. For general prime p , we denote by **MIMC_p** (resp. **MIMC_{2ⁿ}**) the MIMC-Encrypt function defined over \mathbb{F}_p (resp. \mathbb{F}_{2^n}). Since block ciphers are usually defined over the product space of keys and messages, we consider the variables c , *rounds* and e as fixed. The function signatures thus become,

$$\begin{aligned} \text{MIMC}_p &: \mathbb{F}_p \times \mathbb{F}_p \rightarrow \mathbb{F}_p \\ \text{MIMC}_{2^n} &: \mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n} \end{aligned}$$

959 From now on, we only consider MIMC defined over prime fields. In particular, the
 960 field \mathbb{F}_r with elements written over λ bits, over which ZkSnarkSch operates.

961 Security parameters and analysis

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

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

967 3.2.2 MIMC-based compression function

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

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

²We do not consider exponents of type $2^t + 1$ since the polynomial representing MIMC-Encrypt would be sparse and as such, the number of rounds becomes constant (see [AGR⁺16, Section 5.3]).

975 **Miyaguchi-Preneel compression construct**

976 Miyaguchi-Preneel (MP) [BRS02, f_3 function] is a general scheme for constructing com-
 977 pression functions from block ciphers (see Section 1.5.6). Given a block cipher E , the
 978 corresponding compression function by f_E^{MP} is given in Fig. 3.2. The original construc-
 979 tion is defined over binary fields, however **Zeth** operates over prime fields. Hence, in the
 980 general discussion here we replace the bitwise addition operator \oplus by modular addition
 981 in \mathbb{F}_r (see [Har19]).

982 We denote by **MIMC-MP** the compression function defined by the application of
 983 the Miyaguchi-Preneel construct over **MIMC**. Similarly, for general prime p we denote
 984 by **MIMC-MP_p** (see Fig. 3.3) the compression function defined by application of the
 985 Miyaguchi-Preneel construct over **MIMC_p**.

$f_E^{\text{MP}}(k, m)$
1 : $res \leftarrow E_k(m)$
2 : return $(res + m + k) \pmod{r}$

Figure 3.2: MP construct in \mathbb{F}_r .

MIMC-MP_r (k, m)
1 : $res \leftarrow \text{MIMC}_r(k, m)$
2 : return $(res + k + m) \pmod{r}$

Figure 3.3: **MIMC-MP_r** construction.

986 **3.2.3 An efficient instantiation of MIMC primitives**

987 To select appropriate instances of **MIMC_r** and **MIMC-MP_r**, we consider the cost (in terms
 988 of gas consumption and prover efficiency). For given e and $rounds$, the final definition
 989 of **MIMC-MP_r** is given in Fig. 3.4 and Fig. 3.5.

MIMC_r (k, m)	InitRoundConstants ()
1 : $c \leftarrow \text{InitRoundConstants}()$	$iv \leftarrow \text{Keccak256}(\text{"clearmatics_mt_seed"})$
2 : foreach $i \in [rounds]$:	$c[0] \leftarrow 0$
3 : $m \leftarrow (k + c[i] + m)^e \pmod{r}$	$c[1] \leftarrow \text{Keccak256}(iv)$
4 : return $(m + k) \pmod{r}$	foreach $i \in \{2, \dots, rounds\}$:
	$c[i] \leftarrow \text{Keccak256}(c[i-1])$
	return $c = (c[0], \dots, c[rounds-1])$

Figure 3.4: **MIMC_r** full construction

MIMC-MP_r (k, m)
return MIMC_r (k, m) + $m + k \pmod{r}$

Figure 3.5: **MIMC-MP_r** full construction

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

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

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

$$\text{MKHASH}(m_0, m_1) = \text{MIMC-MP}_r(m_0, m_1) \quad (3.1)$$

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

997 Note that small exponents e result in fewer constraints in the arithmetic circuit
 998 (see Section 2.2), while larger exponents can reduce the cost of Merkle tree operations
 999 on the contract (see Section 2.5). This is due to two factors, namely that exponentiation
 1000 is cheaper to execute on a contract than in an arithmetic circuit, and that the number of
 1001 rounds decreases with higher e . For instance, choosing $e = 7$ results in 365 constraints
 1002 and $\approx 20k$ gas while $e = 31$ corresponds to 417 constraints (+15%) and $\approx 17k$ (−10%)
 1003 in gas consumption. Repeating the same process for different exponents, we observe
 1004 roughly the same order of magnitude gain on the gas consumption and loss on the
 1005 number of constraints.

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

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

1006 where $\text{rounds} = \lceil \frac{\log_2 r}{\log_2 e} \rceil$, *mults* is the number of multiplications required for exponentia-
 1007 tion and the additional constraint (corresponding to +1 in the above formula) is a result
 1008 of the final message and key addition. Note that for $e = 2^t - 1$ we have $\text{mults} = 2 \cdot t - 2$
 1009 using the *square-and-multiply* algorithm [MVOV96].

1010 For several concrete values of e , the number of *rounds* required to attain the desired
 1011 security level, along with the number of constraints, are shown in Table 3.1.

1012 In conclusion, for the case of BN-254 we set $e = 7$ and *rounds* = 91, targetting a 254-
 1013 bit security level. Similarly, for BLS12-377 we set $e = 31$ and *rounds* = 51, targetting
 1014 a 253-bit security level. These values are chosen such that they satisfy the requirement
 1015 that $\gcd(e, r - 1) = 1$ and give a good balance between the number of constraints in the
 1016 arithmetic circuit and the gas cost of hashing on the contract.

1017 3.2.4 Security requirements satisfaction

1018 After presenting the state of the art of MiMC cryptanalysis, we present the security
 1019 proof of MIMC-MP collision resistance.

1020 Cryptanalysis of MIMC block cipher and primitives

1021 MIMC's security is increasingly being analysed since the primitive has gained traction
 1022 in zero-knowledge and cryptocurrency communities for its succinct algebraic constraint
 1023 representation. As of today, we do not know of any attacks breaking MIMC on prime
 1024 fields on full rounds.

e	BN-254		BLS12-377	
	<i>rounds</i>	<i>constraints</i>	<i>rounds</i>	<i>constraints</i>
7	91	365		
31	52	417	51	409
127	37	445	37	445
511	29	465		
2047	24	481	23	461
8191	20	481	20	481
32676	17	477		
131071	15	481	15	481
524287	14	505	14	505
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 $\gcd(e, r - 1) \neq 1$

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}_r by proving that, in the Ideal Cipher Model, the

collision resistance advantage of any adversary is bounded by $\frac{q(q+1)}{\mathbf{r}}$, 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 \mathbf{r} can be chosen to make the advantage small as needed and \mathbf{f}_E^{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 $\mathbb{F}_{\mathbf{r}}$ and the block cipher scheme, with key k , correspond to a family of \mathbf{r} independent random permutations $f_k : \mathbb{F}_{\mathbf{r}} \times \mathbb{F}_{\mathbf{r}} \rightarrow \mathbb{F}_{\mathbf{r}}$.

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 $\text{O}^E(k, m)$ and this returns y , then \mathcal{A} does not ask a subsequent query of $\text{O}^E(k, m)$ or $\text{O}^{E^{-1}}(k, y)$, 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 \mathbf{f}_E^{MP} be the MP compression function built on an ideal block-cipher \mathbf{E} on $\mathbb{F}_{\mathbf{r}}$, the probability for an adversary \mathcal{A} to find a collision is not greater than $q(q+1)/\mathbf{r}$ where q is a (positive) number of distinct oracle queries.*

The following proof has been adapted from [BRS02, Lemma 3.3]³.

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

In fact, for each $i \in [1, \dots, q]$, let C_i be the event that either $y_i + m_i + k_i = h_0$ or does exist $j \in [1, \dots, i-1]$ 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 $\mathbf{r} - (i-1)$ elements, so $\Pr[C_i] \leq i/(\mathbf{r} - i)$. This means that for the collision advantage of \mathcal{A} , $\text{Adv}_{\mathbf{f}_E^{\text{MP}}, \mathcal{A}}^{\text{coll}}$ it holds that $\text{Adv}_{\mathbf{f}_E^{\text{MP}}, \mathcal{A}}^{\text{coll}} \leq \Pr[C_1 \vee \dots \vee C_q] \leq \sum_{i=1}^q \Pr[C_i]$. For $q \leq \frac{\mathbf{r}}{2}$ this probability is bounded by $l \cdot \frac{q(q+1)}{\mathbf{r}}$. However, we allow only a polynomial number of queries, thus for $q = \text{poly}(\lambda)$ this probability becomes $\frac{\text{poly}(\lambda)}{\mathbf{r}}$, where $\mathbf{r} \approx 2^\lambda$. \square

³It states the collision resistance of a set of compression functions f_1, \dots, 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 .

Note

Lemma 3.2.2 is applicable to our case by the strong assumption of MIMCr 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.

1084

1085 **Remark 3.2.3.** Note that from Lemma 3.2.2 follows that the collision resistance security
1086 of the Zeth Merkle tree is $\log_2(r/2)$ (around 127 bits for $r = r_{BN}$ or r_{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.

1087

3.3 Zeth statement after primitive instantiation

1088

1089 After instantiating the various primitives and providing security proofs to justify that
1090 they comply with the security requirements listed in previous sections, \mathbf{R}^Z now becomes:

1091 • For each $i \in [\text{JSIN}]$:

- 1092 1. $aux.jsins[i].note.apk = \text{Blake2s}(tag_{ask}^{addr} \parallel \text{pad}_{\text{BLAKE2sCLEN}}(0))$
1093 with tag_{ask}^{addr} defined in Section 3.1.3
- 1094 2. $aux.jsins[i].nf = \text{Blake2s}(tag_{ask}^{nf} \parallel aux.jsins[i].note.\rho)$
1095 with tag_{ask}^{nf} defined in Section 3.1.3
- 1096 3. $aux.jsins[i].cm = \text{Blake2s}(aux.jsins[i].note.r \parallel m)$
1097 with $m = aux.jsins[i].note.apk \parallel aux.jsins[i].note.\rho \parallel aux.jsins[i].note.v$
- 1098 4. $aux.htags[i] = \text{Blake2s}(tag_{ask,i}^{pk} \parallel prim.hsig)$ (malleability fix, see Appendix A)
1099 with $tag_{ask,i}^{pk}$ defined in Section 3.1.3
- 1100 5. $(aux.jsins[i].note.v) \cdot (1 - e) = 0$ is satisfied for the boolean value e set such
1101 that if $aux.jsins[i].note.v > 0$ then $e = 1$.
- 1102 6. The Merkle root $mkroot'$ used to check the Merkle authentication path $aux.jsins[i].mkpath$
1103 of commitment $aux.jsins[i].cm$, with MIMC- MP_r , equals $prim.mkroot$ if $e = 1$.
- 1104 7. $prim.nfs[i]$
1105 $= \{\text{Pack}_{\mathbb{F}_r}(aux.jsins[i].nf[k \cdot \text{FIELD CAP}:(k+1) \cdot \text{FIELD CAP}])\}_{k \in [\lfloor \text{PRFNFOUTLEN}/\text{FIELD CAP} \rfloor]}$

- 1106 8. $\text{prim.htags}[i]$
 1107 $= \{\text{Pack}_{\mathbb{F}_r}(\text{aux.htags}[i][k \cdot \text{FIELD CAP}:(k+1) \cdot \text{FIELD CAP}])\}_{k \in [\lfloor \text{PRFPKOUTLEN}/\text{FIELD CAP} \rfloor]}$
- 1108 • For each $j \in [\text{JSOUT}]$:
- 1109 1. $\text{aux.znotes}[j].\rho = \text{Blake2s}(\text{tag}_{\text{ask},j}^\rho \parallel \text{prim.hsig})$ (malleability fix, see Appendix A)
 1110 with $\text{tag}_{\text{ask},j}^\rho$ defined in Section 3.1.3
- 1111 2. $\text{prim.cms}[j] = \text{Blake2s}(\text{aux.znotes}[j].r \parallel m)$
 1112 with $m = \text{aux.znotes}[j].\text{apk} \parallel \text{aux.znotes}[j].\rho \parallel \text{aux.znotes}[j].v$
- 1113 • $\text{prim.hsig} = \{\text{Pack}_{\mathbb{F}_r}(\text{aux.hsig}[k \cdot \text{FIELD CAP}:(k+1) \cdot \text{FIELD CAP}])\}_{k \in [\lfloor \text{CRHHSIGOUTLEN}/\text{FIELD CAP} \rfloor]}$
- 1114 • $\text{prim.rsd} = \text{Pack}_{\text{rsd}}(\{\text{aux.jsins}[i].\text{nf}\}_{i \in [\text{JSIN}]}, \text{aux.vin}, \text{aux.vout}, \text{aux.hsig}, \{\text{aux.htags}[i]\}_{i \in [\text{JSIN}]})$
- Check that the “joinsplit is balanced”, i.e. check that the joinsplit equation holds:

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

1115 **Remark 3.3.1.** For higher security, we could use **Blake2b** with 32-byte output instead
 1116 of **SHA256**. In fact, since a precompiled contract computing the **Blake2** compression
 1117 function [MJS15] has been added to the Istanbul release of **Ethereum** (EIP 152 [THH15]),
 1118 it could be possible to write a small wrapper on the smart contracts, in order to hash
 1119 with **Blake2b** with any parameter.

1120 3.3.1 Instantiating the packing functions

1121 As we consider SNARKs based on arithmetic circuits defined over a prime field, all
 1122 variables in the constraint system are interpreted as field elements. Nevertheless, as
 1123 illustrated in Section 2.2, part of the statement consists of functions whose co-domains
 1124 are sets of binary strings (which may be longer than the bit representation of elements of
 1125 the finite field). While a bit (i.e. $\{0, 1\}$) is an element of \mathbb{F}_p (p prime), it is important to
 1126 minimize the number of gates in the arithmetic circuit (for proof generation efficiency),
 1127 and to minimize the number of input wires (to improve verification time). This can be
 1128 done by representing fragments of binary strings as the base 2 decomposition of field
 1129 elements, thereby “packing” binary strings into multiple elements. Converting binary
 1130 strings into field elements requires the addition of some arithmetic gates (extending
 1131 the statement to be proven), but reduces the number of primary inputs (reducing the
 1132 complexity of the SNARK verification carried out on-chain). The cost of **Groth16** zk-
 1133 SNARK [Gro16] proof verification is linear in the number of primary inputs, since each
 1134 input acts as a scalar in a costly scalar multiplication of a curve point in \mathbb{G}_1 . Hence,
 1135 while packing slightly increases the prover cost – by adding constraints to the circuit –
 1136 it simplifies the verifier’s work.

1137 In this section, we detail the method by which we encode (resp. decode) a set of
 1138 binary strings to (resp. from) sets of field elements. In the rest of this section, the notion
 1139 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).

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

1140 The primary inputs that consist of binary strings are: the nullifiers *nfs*, the public
 1141 values *vin* and *vout*, the signature hash *hsig* and the authentication tags *htags*.

1142 For a binary string x , let $\alpha_x = \lceil \text{length}(x) / \text{FIELD CAP} \rceil$ be the number of field elements
 1143 required to completely encode x and let $\beta_x = \lfloor \text{length}(x) / \text{FIELD CAP} \rfloor$ be the number of
 1144 field elements whose capacity is fully used. Let $\gamma_x = \text{length}(x) \pmod{\text{FIELD CAP}}$ be the
 1145 number of “residual” bits remaining after fully using β_x field elements.

1146 **Example 3.3.2.** Consider binary strings $A \in \{0, 1\}^7$ of length 7, to be encoded over the
 1147 field \mathbb{F}_{41} . This field has a capacity of 5 bits, and therefore $\alpha_A = 2$, $\beta_A = 1$, and $\gamma_A = 2$.
 1148 That is, A can be represented as 2 field elements, or as 1 field element with 2 “residual”
 1149 bits.

1150 Consider $A = (1111011)$. Fig. 3.6 illustrates how A can be packed as field elements.
 1151 Note that the 2 residual bits are taken from the “beginning” of the bit string, that is,
 1152 the highest order bits.

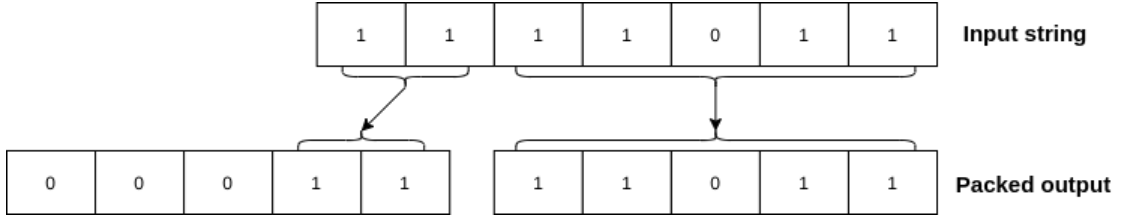


Figure 3.6: Packing of string A (see Example 3.3.2)

1153 We now consider strategies to pack all primary inputs that are binary strings. A naive
 1154 approach is to encode each binary string x as α_x field elements. In general, this results
 1155 in significant waste (and consequently more field elements than necessary), especially
 1156 when the number of residual bits is small compared to **FIELD CAP** (see Fig. 3.7). An
 1157 alternative strategy could be to concatenate all binary strings into a single string y and
 1158 pack this string into α_y field elements. While this approach minimizes the set of unused
 1159 bits, each unpack operation would require different shift and mask operations over 2 or
 1160 3 field elements. This significantly increases the complexity of the unpacking operation
 1161 that must be performed on-chain, resulting in a higher gas cost (due to extra logic) or
 1162 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** < **FIELD CAP**, and define the notation $\gamma_v = \text{ZVALUELEN}$ for the bit lengths of public values vin and $vout$. Thus **RSDBLEN** is given by

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

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

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

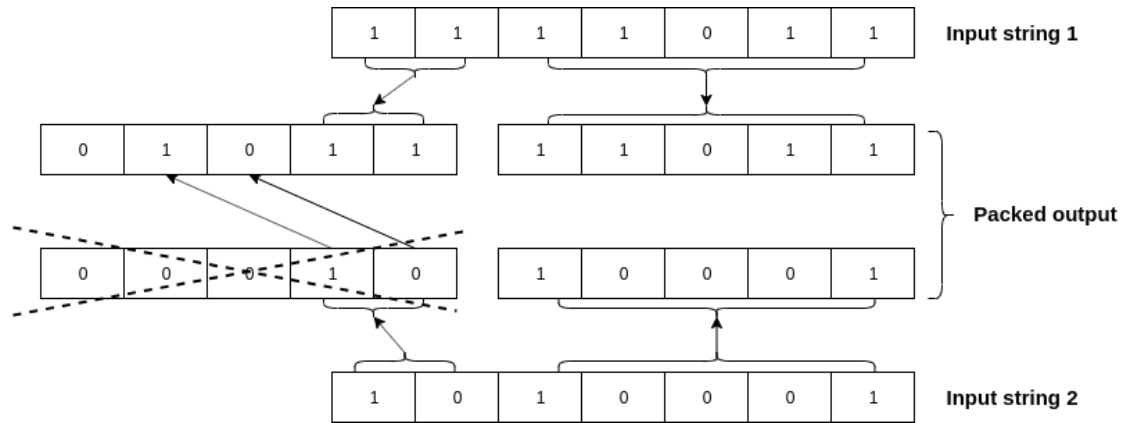


Figure 3.7: Packing of multiple strings. Observe that, by carefully arranging the bits of the input strings, it is possible to output fewer field elements

The residual bits rsd are formatted as follows:

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

1163 where \widetilde{hsig} , \widetilde{nfs} , \widetilde{htags} are, respectively, the γ_{hsig} , γ_{nf} , γ_h bits.

1164 Note that the public values are packed into the “last”, or lowest order, $2 \cdot \gamma_v$ bits of
 1165 the resulting field element(s). In this way, their unpack functions are independent of the
 1166 values **JSIN** and **JSOUT** and of the number of residual bits required for each bit string
 1167 (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 **FIELD CAP**, 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

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

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

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

Figure 3.9: Algorithm to pack residual bits.

packed field elements and the residual bits, the algorithm `Unpack` returns the variables reassembled as binary strings. In particular, we have that $\text{Unpack}_{nf}(\text{prim}.nfs, rsd) = \{aux.jsins[i].nf\}_{i \in [\text{JSIN}]}$.

$\text{Pack} : \mathbb{B}^{\leq \text{FIELD CAP}} \rightarrow \mathbb{F}_r$

$\text{Pack}_{rsd} : (\mathbb{B}^{\text{PRFNFOUTLEN}})^{\text{JSIN}} \times (\mathbb{B}^{\text{ZVALUELEN}})^2 \times \mathbb{B}^{\text{CRHHSIGOUTLEN}} \times (\mathbb{B}^{\text{PRFPKOUTLEN}})^{\text{JSIN}} \rightarrow (\mathbb{F}_r)^{\text{RSDFLEN}}$

$\text{Unpack} : \mathbb{F}_r^* \times (\mathbb{F}_r)^{\text{RSDFLEN}} \rightarrow \mathbb{B}^*$

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

$\text{Unpack}_{hsig} : (\mathbb{F}_r)^{\text{HSIGFLEN}} \times (\mathbb{F}_r)^{\text{RSDFLEN}} \rightarrow \mathbb{B}^{\text{CRHHSIGOUTLEN}}$

$\text{Unpack}_{nf} : (\mathbb{F}_r)^{\text{NFFLEN}} \times (\mathbb{F}_r)^{\text{RSDFLEN}} \rightarrow \mathbb{B}^{\text{PRFNFOUTLEN}}$

$\text{Unpack}_{vin} : \mathbb{F}_r^0 \times (\mathbb{F}_r)^{\text{RSDFLEN}} \rightarrow \mathbb{B}^{\text{ZVALUELEN}}$

$\text{Unpack}_{vout} : \mathbb{F}_r^0 \times (\mathbb{F}_r)^{\text{RSDFLEN}} \rightarrow \mathbb{B}^{\text{ZVALUELEN}}$

1168 Packing Policy Security

1169 **Proposition 3.3.1** (Packing security). *For a binary string x , it holds that $\text{Unpack}(\text{Pack}(x)) =$*
1170 *x and $\text{Unpack}(\text{Pack}_{rsd}(x)) = x$.*

1171 Packing Policy Example

In the case where $\text{JSIN} = \text{JSOUT} = 2$, the BN-254 is being used (in which field elements hold $\text{FIELD CAP}_{\text{BN}}$ bits) and all PRFs and CRH^{hsig} output bit-strings of length 256, the

unpacked primary inputs are 2167-bit long. The packing parameters are therefore:

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

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,

$$\underbrace{\text{padding}}_{113 \text{ bits}} \parallel \underbrace{\text{hsig}}_{3 \text{ bits}} \parallel \underbrace{\text{nf}_1}_{3 \text{ bits}} \parallel \underbrace{\text{nf}_0}_{3 \text{ bits}} \parallel \underbrace{h_1}_{3 \text{ bits}} \parallel \underbrace{h_0}_{3 \text{ bits}} \parallel \underbrace{\text{vin}}_{64 \text{ bits}} \parallel \underbrace{\text{vout}}_{64 \text{ bits}}$$

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

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

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

$$\underbrace{\text{padding}}_{108 \text{ bits}} \parallel \underbrace{\text{hsig}}_{4 \text{ bits}} \parallel \underbrace{\text{nf}_0}_{4 \text{ bits}} \parallel \underbrace{\text{nf}_1}_{4 \text{ bits}} \parallel \underbrace{h_0}_{4 \text{ bits}} \parallel \underbrace{h_1}_{4 \text{ bits}} \parallel \underbrace{\text{vin}}_{64 \text{ bits}} \parallel \underbrace{\text{vout}}_{64 \text{ bits}}$$

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

3.4 Instantiate SigSch_{OT-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 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 $(p, \mathbb{G}, \langle \mathbf{g} \rangle, \otimes)$. 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

$\text{KGen}(1^\lambda) :$ $hk \leftarrow \$ \mathbb{B}^{kl}$ $g \leftarrow \$ \mathbb{G}^*$ $x \leftarrow \$ \mathbb{F}_p$ $pk1 = (hk, g, \llbracket x \rrbracket)$ $sk1 = (hk, g, x)$ $y \leftarrow \$ \mathbb{F}_p$ $pk2 = \llbracket y \rrbracket$ $sk2 = (y, \llbracket y \rrbracket)$ $pk = (pk1, pk2)$ $sk = (sk1, sk2)$	$\text{Sig}(sk, m) :$ $hk, g, x = sk.sk1$ $y, \llbracket y \rrbracket = sk.sk2$ $c = \text{CRH}(hk, \llbracket y \rrbracket \parallel m)$ $\sigma = y \bmod p$ $\sigma += c \cdot x \bmod p$ return σ	$\text{Vf}(pk, m, \sigma) :$ $hk, g, \llbracket x \rrbracket = pk.pk1$ $\llbracket y \rrbracket = pk.pk2$ $c = \text{CRH}(hk, \llbracket y \rrbracket \parallel m)$ if $\sigma = \llbracket y \rrbracket \otimes c \cdot \llbracket x \rrbracket$ then return 1 else return 0 endif
---	---	---

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.

VK0tsDType Denotes the verification key associated with the one-time signature scheme.

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

Table 3.2: VK0tsDType data type

1205 **SK0tsDType** Denotes the signing key associated with the one-time signature scheme.

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

Table 3.3: SK0tsDType data type

1206 **Sig0tsDType** Denotes the signature data type associated with the one-time signature
1207 scheme. **Sig0tsDType** is an alias for $(\mathbb{F}_{\mathbf{r}_{\text{BN}}})^2$.

1208 3.5 Instantiate EncSch

1209 In this section we describe the instantiation of **EncSch** primitive introduced in Section 2.3.
1210 First, we present a general asymmetric encryption scheme called **DHAES** (Diffie-Hellman
1211 Asymmetric Encryption Scheme [ABR99]), which satisfies all the required security prop-
1212 erties for the in-band encryption scheme **EncSch** (see Section 1.5.3). Then, we give details
1213 of the concrete algorithms used for the implementation.

1214 3.5.1 DHAES encryption scheme

1215 Given a symmetric encryption scheme **Sym**, a group defined by **SetupG**, a family of hash
1216 function \mathcal{H}^4 and a message authentication scheme **MAC** as defined in Section 1.5, we
1217 define a **DHAES** scheme as the following public-key encryption scheme:

- 1218 • **Setup**, setup algorithm, takes as input a security parameter 1^λ . It runs $\mathcal{H}.\text{Setup}$,
1219 **SetupG** and returns public parameters $pp = (hk, (q, \mathbb{G}, \mathbf{g}, +))$.
- 1220 • **KGen**, key generation algorithm, takes as input public parameters pp . It samples
1221 at random $v \leftarrow_{\$} [q]$ and returns a keypair $(sk, pk) = (v, \llbracket v \rrbracket)$.

⁴Here, we only consider fixed-length hash functions with $h\text{InpLen}(\lambda) = 2g\text{Len}$ and $h\text{Len}(\lambda) = k\text{Len}(\lambda) + m\text{Len}(\lambda)$ (see Section 1.5).

- 1222 • **Enc**, encryption algorithm, takes as input public parameters pp , a message m and
1223 a public key pk . It runs **KGen** that returns an ephemeral keypair $(esk, epk) =$
1224 $(u, \llbracket u \rrbracket)$. Then, it computes a shared secret $ss = H_{hk}(epk \parallel esk \cdot pk) = H_{hk}(epk \parallel sk \cdot$
1225 $epk)$, parsed as $ek \parallel mk$ ⁵. It computes $ct_{\text{Sym}} = \text{Sym.Enc}(ek, m)$ and $\tau = \text{MAC.Tag}(mk, ct_{\text{Sym}})$
1226 and finally outputs the ciphertext $epk \parallel ct_{\text{Sym}} \parallel \tau$.
- 1227 • **Dec**, decryption algorithm, takes as input public parameters pp , a private key sk
1228 a ciphertext $epk \parallel ct_{\text{Sym}} \parallel \tau$. It computes $ss = H_{hk}(epk \parallel sk \cdot epk)$ and parses it, as
1229 above, as $ek \parallel mk$. If MAC verification passes, i.e. $\text{MAC.Vf}(mk, \tau) = 1$, the algorithm
1230 returns $\text{Sym.Dec}(ek, ct_{\text{Sym}})$ and \perp otherwise.

1231 The DHAES definition given above is an asymptotic adaptation of [ABR99, Section
1232 1.3].

1233 Inclusion of ephemeral key in hash input

1234 Given an ephemeral keypair $(u_0, \llbracket u_0 \rrbracket)$, If the group $\langle \mathbf{g} \rangle$, generated by **SetupG**, has com-
1235 posite order, then $\llbracket u_0 \rrbracket$ is required to be part of the hash input because $\llbracket u_0 v \rrbracket$ and $\llbracket v \rrbracket$
1236 together may not uniquely determine $\llbracket u_0 \rrbracket$. Equivalently, there may exist two values
1237 u_0 and u_1 such that $u_0 \neq u_1$ and $\llbracket u_0 v \rrbracket = \llbracket u_1 v \rrbracket$. As a result, both u_0 and u_1 can be
1238 used to produce two different *valid* ciphertexts of the same plaintext m , under different
1239 ephemeral keys $(\llbracket u_0 \rrbracket, \llbracket u_1 \rrbracket)$. It is easy to show this, for example, in the multiplicative
1240 group $\mathbb{Z}_p \setminus \{0\}$, where p is a prime (see [ABR99, Section 3.1]). A scheme having such
1241 malleability property clearly cannot be proven IND-CCA2 secure: an attacker could eas-
1242 ily win the related security game by altering the challenged ciphertext and query the
1243 decryption oracle that would not recognize that as a not allowed query. If the group
1244 has prime order this problem does not arise so only $\llbracket u_0 v \rrbracket$ is required as input of the **H**
1245 function [ABR01, Section 3].

1246 3.5.2 A DHAES instance

1247 Curve25519

1248 For a cyclic group we propose the use of a subgroup of **Curve25519** described in [Ber06]
1249 and in [LHT16]. **Curve25519** is a Montgomery elliptic curve [Mon87] defined by the
1250 equation $y^2 = x^3 + 486662x^2 + x$ and coordinates on \mathbb{F}_p , where p is the prime number
1251 $2^{255} - 19$. It has a prime order subgroup of order $2^{252} + 27742317777372353535851937$
1252 790883648493 and cofactor 8. **Curve25519** comes with an efficient scalar multiplication
1253 denoted as **X25519**⁶. In a Diffie-Hellman-based scheme it allows to have 32-byte long
1254 public and private keys (given a point $P = (x, y)$ only the x coordinate is actually used)
1255 and the 32-byte sequence representing 9 is specified as base point.

⁵Note that ek and mk must have the same length.

⁶**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]

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]⁷.

Interestingly, Curve25519 does not require *public key validation*⁸, 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-based⁹ 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 process around 2^{32} blocks, corresponding to roughly 256 GB of data. We propose to use the same parameters in Zeth.

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

⁷In 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.

⁸Informally, it is a set of security checks that a user performs before using a not trusted public key (e.g. see [BCK⁺18])

⁹Addition-Rotation-XOR

1285 Security of Chacha

1286 Recent cryptanalysis results for ChaCha are available in [AFK⁺08, Ish12, SZFW12,
1287 Mai16, CM16, CM17]: all of them make use of advanced cryptanalysis techniques able
1288 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].

1289

1290 Poly1305

1291 Poly1305 [Ber05] is a high-speed message authentication code, easy to implement and
1292 make side-channel attack resistant. It takes a 32-byte one-time key mk and a message m
1293 and produces a 16-byte tag τ that authenticates the message. mk must be unpredictable
1294 and it is represented as a couple (r, s) , where both components are given as a sequence
1295 of 16 bytes each. It can be generated by using pseudorandom algorithms: in [Ber05,
1296 Section 2], for example, AES and a nonce are used to generate s . The second part of
1297 the key, r , is expected to have a given form [Ber05, Section 2], and must be “clamped”
1298 as follows: top four bits of $r[3]$, $r[7]$, $r[11]$, $r[15]$ and bottom two bits of $r[4]$, $r[8]$, $r[12]$
1299 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.

1300

1301 We refer to [LN18, Section 2.5, Section 3] for Tag and Vf implementations of Poly1305.

$$\begin{aligned}\text{Poly1305.Tag} &: \mathbb{B}_Y^{32} \times \mathbb{B}_Y^* \rightarrow \mathbb{B}_Y^{16} \\ \text{Poly1305.Vf} &: \mathbb{B}_Y^{32} \times \mathbb{B}_Y^{16} \times \mathbb{B}_Y^* \rightarrow \mathbb{B}\end{aligned}$$

1302 Security of Poly1305

1303 Citing Poly1305 [LN18, Section 4], “the Poly1305 authenticator is designed to ensure that
1304 forged messages are rejected with a probability of $1 - (n/(2^{102}))$ for a $16n$ -byte message,
1305 even after sending 2^{64} legitimate messages, so it is **SUF-CMA** (strong unforgeability
1306 against chosen-message attacks)”.

1307 Blake2b-512

1308 Since we need a total of 64 bytes for the key material (32 for ChaCha20 and 32 for
 1309 Poly1305) Blake2b512 can be used. ZCash protocol [ZCa19, Section 5.4.3], instead, makes
 1310 use of Blake2b256 since a DHAES variant, denoted as ChaCha20-Poly1305, is adopted
 1311 (see [LN18, Section 2.8]).

$$\text{Blake2b512} : \mathbb{B}^* \rightarrow \mathbb{B}_{\mathbb{Y}}^{32}$$

1312 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:

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

1313 EncSch.KGen

1314 The keypair (sk, pk) generation is defined as:

- 1315 • Randomly sample a sequence of ESKBYTELEN bytes and assign to sk .
- Clamp sk as follows:

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

1316 where $|$ and $\&$ denotes, respectively, OR and AND binary operators between bit
 1317 strings of same the length.¹⁰

- 1318 • Compute $pk = \text{X25519}(sk, 0x09)$.
- 1319 • Return $(sk, pk) \in \mathbb{B}_{\mathbb{Y}}^{\text{ESKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\text{EPKBYTELEN}}$

¹⁰E.g Given two bytes 0x15 and 0x03 then $0x15|0x03 = 0x17$ and $0x15\&0x03 = 0x01$.

1320 EncSch.Enc

1321 The encryption, on inputs $(pk, m) \in \mathbb{B}_{\mathbb{Y}}^{\text{EPKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\text{NOTEBYTELEN}}$, is defined as follows:

1322 1. Generate an ephemeral Curve25519 keypair $(esk, epk) \in \mathbb{B}_{\mathbb{Y}}^{\text{ESKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\text{EPKBYTELEN}}$
 1323 (as above).

2. Compute the shared secret¹¹ $ss \in \mathbb{B}_{\mathbb{Y}}^{\text{EPKBYTELEN}}$:

$$ss = \text{X25519}(esk, pk) \in \mathbb{B}_{\mathbb{Y}}^{\text{EPKBYTELEN}}$$

3. Generate a session key:

$$\text{Blake2b512}(\text{encTag} \| epk \| ss) \in \mathbb{B}_{\mathbb{Y}}^{\text{KDFDIGESTBYTELEN}}$$

where $\text{encTag} = 0x5A \| 0x65 \| 0x74 \| 0x68 \| 0x45 \| 0x6E \| 0x63$, that is the UTF-8 encoding of “ZethEnc” string (used for domain separation purposes). The result, then, is parsed as follows:

$$ek = \text{Blake2b512}(\text{encTag} \| epk \| ss)[\text{SYMKEYBYTELEN} - 1]$$

$$mk = \text{Blake2b512}(\text{encTag} \| epk \| ss)[\text{SYMKEYBYTELEN} : \text{SYMKEYBYTELEN} + \text{MACKEYBYTELEN} - 1].$$

4. Encrypt the confidential data:

$$ct_{\text{sym}} = \text{ChaCha20}(ek, \text{CHACHABLOCKCOUNTERVALUE}, \text{CHACHANONCEVALUE}, m) \in \mathbb{B}^{\text{NOTEBYTELEN} * \text{BYTELEN}}$$

1324 **Remark 3.5.1.** Formally speaking we should have written $ct_{\text{sym}} \in \mathbb{B}^n$, where
 1325 n is the length of binary representation of the encrypted message m . In Zeth
 1326 however, the only data encrypted are the notes. As such, the size of the plaintexts
 1327 is $\text{NOTEBYTELEN} * \text{BYTELEN}$ bits.

1328 **Remark 3.5.2.** In the following, we omit the explicit conversion from \mathbb{B}^n to
 1329 $\mathbb{B}_{\mathbb{Y}}^{\lceil n/\text{BYTELEN} \rceil}$ when passing the output of ChaCha20 to the Poly1305 algorithms.

5. Randomly generate $(r, s) \in \mathbb{B}_{\mathbb{Y}}^{\text{MACKEYBYTELEN}/2} \times \mathbb{B}_{\mathbb{Y}}^{\text{MACKEYBYTELEN}/2}$ and clamp it:

$$\begin{aligned} r[3] &\leftarrow r[3] \ \& \ 0x0F \\ r[7] &\leftarrow r[7] \ \& \ 0x0F \\ r[11] &\leftarrow r[11] \ \& \ 0x0F \\ r[15] &\leftarrow r[15] \ \& \ 0x0F \\ r[4] &\leftarrow r[4] \ \& \ 0xFC \\ r[8] &\leftarrow r[8] \ \& \ 0xFC \\ r[12] &\leftarrow r[12] \ \& \ 0xFC \end{aligned}$$

¹¹We assume here that esk has been clamped as discussed in Section 3.5.2

6. Generate the related tag:

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

7. Create the asymmetric ciphertext as:

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

1330 8. Return ct . As consequence $\text{ENCZETHNOTELEN} = \text{CTBYTELEN} * \text{BYTELEN}$ bits.

1331 **EncSch.Dec**

1332 The decryption, on inputs $(sk, ct) \in \mathbb{B}_{\mathbb{Y}}^{\text{ESKBYTELEN}} \times \mathbb{B}_{\mathbb{Y}}^{\text{CTBYTELEN}}$, is defined as follows:

1. Parse the ciphertext ct as:

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

2. Recover the shared secret

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

3. Compute the $ek \| mk$

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

4. Verify that the ciphertext has not been forged:

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

5. (If the MAC verifies) decrypt:

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

1333 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])¹² 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 $(\mathcal{H}, \text{SetupG})$ and a SUF-CMA adversary \mathcal{C} of MAC such that*

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

The adversaries \mathcal{B} and \mathcal{C} have the same running time as \mathcal{A} ¹³.

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 $\bar{\mathcal{B}}$ attacker for the ODH2 game [ABN10, Figure 20] and then apply [ABN10, Lemma 6.1] to obtain an attacker \mathcal{B} ¹⁴ 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 $\bar{\mathcal{B}}$ be an attacker for the HDHI2 game described in Fig. 3.11. We show that,

$$\text{Adv}_{\mathcal{H}, \text{SetupG}, \bar{\mathcal{B}}}^{\text{hdhi2}}(\lambda) = |\Pr[\text{IK-CCA}^{\mathcal{A}}(\lambda) = 1] + \Pr[\text{G}_0^{\mathcal{A}}(\lambda) = 1] - 1|$$

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

¹²Specifically, if Sym is IND-CPA secure, it holds that H is HDHI secure and MAC is SUF-CMA secure.

¹³In order to give an asymptotic version of the theorem, the number of queries q has been substituted by the fact of considering PPT adversaries.

¹⁴Note 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 $\bar{\mathcal{B}}(\llbracket u \rrbracket, \llbracket v_0 \rrbracket, \llbracket v_1 \rrbracket, w_{b_2,0}, w_{b_2,1})$	$\bar{\mathcal{B}}$ simulation of $\mathcal{O}^{\text{Dec}_{sk_i}}(\tau \parallel ct_{\text{Sym}} \parallel \tau)$
$b \leftarrow \$\{0, 1\}$	if $\tau \neq \llbracket u \rrbracket$
$(m, state) \leftarrow \mathcal{A}^{\mathcal{O}^{\text{Dec}_{sk_0}}, \mathcal{O}^{\text{Dec}_{sk_1}}}(\llbracket v_0 \rrbracket, \llbracket v_1 \rrbracket)$	$ek \parallel mk \leftarrow \mathcal{O}^{\text{HDH}_{v_i}}(\tau)$
$ek \parallel mk \leftarrow w_{b_2,b}$	else
$\tau^* \leftarrow u$	$ek \parallel mk \leftarrow w_{b_2,i}$
$ct_{\text{Sym}}^* \leftarrow \text{Sym.Enc}(ek, m)$	fi
$\tau^* \leftarrow \text{MAC.Tag}(mk, ct_{\text{Sym}}^*)$	if $\text{MAC.Vf}(mk, ct_{\text{Sym}}, \tau) = 1$
$\tilde{b} \leftarrow \mathcal{A}^{\mathcal{O}^{\text{Dec}_{sk_0}}, \mathcal{O}^{\text{Dec}_{sk_1}}}(\tau^* \parallel ct_{\text{Sym}}^* \parallel \tau^*, state)$	return $\text{Sym.Dec}(ek, ct_{\text{Sym}})$
return $\tilde{b} = b$	else
	return \perp
	fi

Figure 3.11: Description of the adversary $\bar{\mathcal{B}}$ 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 \mathbf{G}_0 game described in Fig. 3.12. So we get,

$$\Pr[\text{HDHI2}^{\bar{\mathcal{B}}}(\lambda) = 1] = \frac{1}{2} \cdot \Pr[\text{IK-CCA}^{\mathcal{A}}(\lambda) = 1] + \frac{1}{2} \cdot \Pr[\mathbf{G}_0^{\mathcal{A}}(\lambda) = 1].$$

1360 And from the definition of HDHI2 advantage we have

$$\text{Adv}_{\mathcal{H}, \text{Setup}_{\mathbf{G}}, \bar{\mathcal{B}}}^{\text{hdhi2}}(\lambda) = |\Pr[\text{IK-CCA}^{\mathcal{A}}(\lambda) = 1] + \Pr[\mathbf{G}_0^{\mathcal{A}}(\lambda) = 1] - 1|.$$

1361 At this point, we can conclude as in [ABN10, Theorem 6.2], with the only difference
 1362 of applying Lemma 1.5.1 instead of [ABN10, Lemma 6.1] and by defining a game \mathbf{G}_1
 1363 that is *identical until bad*¹⁵ \mathbf{G}_0 defined in Fig. 3.12. \square

1364 3.5.5 Final notes and observations

1365 In this section we list some notes regarding the approach taken in **Zcash** (see [ZCa19,
 1366 Section 8.7]), and other observations:

- 1367 • *Key derivation parameters:* in DHAES construction, the only required input vari-
 1368 ables are the shared secret ss and epk . In the Sprout release of **Zcash**, additional
 1369 parameters were added (i.e. h_{sig} , pk_{enc} and a counter i) (see [ZCa19, 5.4.4.2]):
 1370 they state that h_{sig} was used in order to get a different randomness extractor for
 1371 each joinsplit transfer in order to limit the degradation of the security and weaken
 1372 assumption on the hash. The authors believed, about the use of long-standing
 1373 public key pk_{enc} , that it might be necessary for IND-CCA2 security and for post-
 1374 quantum privacy (in the case where the quantum attacker does not have the public

¹⁵Games \mathbf{G}_i and \mathbf{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*

$G_0(\lambda)$	Oracle $O^{\overline{\text{Dec}}_{sk_i}}(\tau \ ct_{\text{Sym}} \ \tau)$
$(q, \mathbb{G}, g, +) \leftarrow \text{SetupG}(1^\lambda)$	if $\tau = \tau^*$
$(sk_0, pk_0), (sk_1, pk_1) \leftarrow \$ \text{KGen}(1^\lambda)$	$m \leftarrow \perp$
$\tau^* \leftarrow \$ \mathbb{G}$	if $\text{MAC.Vf}(mk^*, ct_{\text{Sym}}, \tau) = 1$
$ek^* \leftarrow \$ \{0, 1\}^{kLen}$	$\text{bad} \leftarrow \text{true}$
$mk^* \leftarrow \$ \{0, 1\}^{mLen}$	$m \leftarrow \text{Sym.Dec}(ek^*, ct_{\text{Sym}})$
$(m, state) \leftarrow \mathcal{A}^{O^{\overline{\text{Dec}}_{sk_0}}, O^{\overline{\text{Dec}}_{sk_1}}}(pk_0, pk_1)$	fi
$b \leftarrow \$ \{0, 1\}$	else
$ct_{\text{Sym}}^* \leftarrow \text{Sym.Enc}(ek^*, m)$	$m \leftarrow \text{Dec}(sk_i, \tau \ ct_{\text{Sym}} \ \tau)$
$\tau^* \leftarrow \text{MAC.Tag}(mk^*, ct_{\text{Sym}}^*)$	fi
$\tilde{b} \leftarrow \mathcal{A}^{O^{\overline{\text{Dec}}_{sk_0}}, O^{\overline{\text{Dec}}_{sk_1}}}(\tau^* \ ct_{\text{Sym}}^* \ \tau^*, state)$	return m
return $\tilde{b} = b$	

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

1399 is successful but returns an incorrect plaintext. This prevents situations where
1400 a user, scanning the **Mixer** logs for incoming transactions, gets a false positive
1401 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 **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.

1402
1403 In [ABN10, Section 6], the authors prove that DHIES can be made strongly robust.
1404 The proof can be easily adapted to work with DHAES.

1405 • *No public key validation for X25519*: cryptographers have been discussing the ab-
1406 sence of any mandated public key validation or checks on the result of X25519.
1407 For example, in [LHT16, Section 6.1], an optional zero check is introduced in order
1408 to assure that the result of X25519 is not 0: this avoids a situation in which one
1409 of the two parties can force the result of the key-exchange by using a small order
1410 point as public key. This property is generally defined as *contributory behaviour*,
1411 that is, none of the parties is able to force the output of a key exchange. However,
1412 protocols do not have all the same security requirements and adding default checks
1413 in the Curve25519 specifications would be superfluous in most cases and would add
1414 complexities that Bernstein has deliberately chosen to avoid (*simple implementa-*
1415 *tion principle*). More importantly, Diffie-Hellman does not require *contributory*
1416 *behaviour* property [Per17]: modern view is that the only requirements are key
1417 indistinguishability and, in case of an active attacker, that the output of the key
1418 exchange should not produce a low-entropy function of the honest party’s private
1419 key (e.g. small-subgroup and invalid-curve attacks). Since these two properties are
1420 considered satisfied by Curve25519, there is no need to add extra checks to the
1421 Curve25519 specification. We conclude by observing that in the Sprout release, the
1422 Zcash protocol does not specify any point validation and makes use only of the
1423 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 (consisting of addition and multiplication gates over a finite field \mathbb{F}). 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 **R** be fixed. Without loss of generality we consider *constNo* to be an *upper bound* on the number of constraints in the **R** parameter, and assume that there exists some *constNo*-th root of unity $\omega \in \mathbb{F}_{\mathbf{r}}$. Define $\ell_i(X)$ to be the *i*-th Lagrange polynomial of degree (*constNo* − 1) over the set $\{\omega^i\}_{i \in [\text{constNo}]}$, and let $\ell(X)$ be the unique non-zero polynomial of degree *constNo* that satisfies $\ell(\omega^i) = 0$ for all $i \in [\text{constNo}]$.

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 **R** that the scheme can support. In the particular case of $\omega \in \mathbb{F}_{\mathbf{r}_{\text{BN}}}$, the restriction becomes $\text{constNo} \leq 2^{28}$. For $\mathbb{F}_{\mathbf{r}_{\text{BLS}}}$ this becomes $\text{constNo} \leq 2^{47}$.

Furthermore, we denote by $\text{inp} \in \mathbb{F}^{\text{inpNo}+1}$ the tuple of variables (i.e. “circuit wires”) in the algebraic representation of the relation **R**, such that:

- $\text{inp}_0 = 1_{\mathbb{F}}$ (the multiplicative identity in \mathbb{F}),
- $(\text{inp}_1, \dots, \text{inp}_{\text{inpNoPrim}})$ represent variables in the statement,
- $(\text{inp}_{\text{inpNoPrim}+1}, \dots, \text{inp}_{\text{inpNo}})$ represent variables in the witness (so-called “auxiliary input”).

KGen(**R**, 1^λ):

- i. Pick trapdoor $td = (\tau, \alpha, \beta, \delta) \leftarrow \$(\mathbb{Z}_p^* \setminus \{\omega^{i-1}\}_{i=1}^{\text{constNo}}) \times (\mathbb{Z}_p^*)^3$;
- ii. For $j \in \{0, \dots, \text{inpNo}\}$, let

$$u_j(\tau) = \sum_{i=1}^{\text{constNo}} U_{ij} \ell_i(\tau),$$

$$v_j(\tau) = \sum_{i=1}^{\text{constNo}} V_{ij} \ell_i(\tau),$$

$$w_j(\tau) = \sum_{i=1}^{\text{constNo}} W_{ij} \ell_i(\tau);$$

iii. Set

$$\begin{aligned} srs_P &\leftarrow \left(\llbracket \alpha \rrbracket_1, \llbracket \beta \rrbracket, \llbracket \delta \rrbracket, \{ \llbracket u_j(\tau) \rrbracket_1 \}_{j=0}^{inpNo}, \{ \llbracket v_j(\tau) \rrbracket \}_{j=0}^{inpNo}, \right. \\ &\quad \left. \{ \llbracket (u_j(\tau)\beta + v_j(\tau)\alpha + w_j(\tau))/\delta \rrbracket_1 \}_{j=inpNoPrim+1}^{inpNo}, \right. \\ &\quad \left. \{ \llbracket \tau^i \ell(\tau)/\delta \rrbracket_1 \}_{i=0}^{constNo-2} \right) \\ srs_V &\leftarrow \left(\llbracket \alpha \rrbracket_1, \llbracket \beta \rrbracket_2, \llbracket \delta \rrbracket_2, \{ \llbracket \beta u_j(\tau) + \alpha v_j(\tau) + w_j \rrbracket_1 \}_{j=0}^{inpNoPrim} \right) \\ srs &\leftarrow (srs_P, srs_V) \end{aligned}$$

1450 **return** srs, td

1451 $P(\mathbf{R}, srs_P, prim = (inp_j)_{j=1}^{inpNoPrim}, aux = (inp_j)_{j=inpNoPrim+1}^{inpNo})$:

i. Define

$$a^\dagger(X) = \sum_{j=0}^{inpNo} inp_j u_j(X), \quad b^\dagger(X) = \sum_{j=0}^{inpNo} inp_j v_j(X), \quad c^\dagger(X) = \sum_{j=0}^{inpNo} inp_j w_j(X);$$

1452 ii. Define the polynomial $h(X) = (a^\dagger(X)b^\dagger(X) - c^\dagger(X))/\ell(X)$ and compute the
1453 coefficients $\{h_i\}_{i=0}^{constNo-2}$ of h , such that $h(X) = \sum_{i=0}^{constNo-2} h_i X^i$.

1454 iii. $r_a \leftarrow \$\mathbb{Z}_p$;

1455 iv. $r_b \leftarrow \$\mathbb{Z}_p$;

v. Compute proof elements:

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

1456 **return** $\pi \leftarrow (\mathbf{a}, \mathbf{b}, \mathbf{c})$;

1457 $V(\mathbf{R}, srs_V, prim = (inp_j)_{j=1}^{inpNoPrim}, \pi)$:

i. Check that:

$$\begin{aligned} \mathbf{a} \bullet \mathbf{b} &= \mathbf{c} \bullet \llbracket \delta \rrbracket_2 \\ &\quad + \left(\sum_{j=0}^{inpNoPrim} inp_j \llbracket u_j(\tau)\beta + v_j(\tau)\alpha + w_j(\tau) \rrbracket_1 \right) \bullet \llbracket 1 \rrbracket_2 \\ &\quad + \llbracket \alpha \rrbracket_1 \bullet \llbracket \beta \rrbracket_2 \end{aligned}$$

1458 Note that $\llbracket \alpha \rrbracket_1$ and $\llbracket \beta \rrbracket_2$ are stored individually and used by the prover to re-
 1459 compute $\llbracket \alpha\beta \rrbracket_T$ seemingly redundantly. This is required in order to leverage the
 1460 pairing check functionality built in to **Ethereum**, which accepts a sequence of tuples
 1461 in $\mathbb{G}_1 \times \mathbb{G}_2$ and returns **true** if and only if the product of the resulting pairings
 1462 equals $\llbracket 1 \rrbracket_T$.

1463 **Sim**(**R**, *srs*, *td*, *prim*):

- 1464 i. Sample $a \leftarrow \$\mathbb{Z}_p$; $b \leftarrow \$\mathbb{Z}_p$;
 ii. Compute proof elements:

$$\begin{aligned} \mathbf{a} &\leftarrow \llbracket a \rrbracket_1 \\ \mathbf{b} &\leftarrow \llbracket b \rrbracket_2 \\ \mathbf{c} &\leftarrow \frac{1}{\delta} \cdot \left(ab - \alpha\beta - \sum_{j=0}^{inpNoPrim} inp_j(u_j(\tau)\beta + v_j(\tau)\alpha + w_j(\tau)) \right) \llbracket 1 \rrbracket_1 \end{aligned}$$

1465 **return** $\pi \leftarrow (\mathbf{a}, \mathbf{b}, \mathbf{c})$;

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]).

1486

1487 4.1.1 Syncing and waiting

1488 **Zeth** clients must periodically synchronize with the latest state of the blockchain. This
1489 is necessary to keep track of the data held by the **Mixer** contract, and to detect notes
1490 received by the user of the wallet, storing them for future transactions.

1491 Clients should synchronize with **Ethereum** nodes in such a way that information is
1492 not leaked. As such:

- 1493 1. Clients **MUST** use consensus evidence and block headers to verify all data they
1494 receive from **Ethereum** nodes.
- 1495 2. Clients **MUST** locally store all parts of the **Mixer** state they require in order to
1496 function.
- 1497 3. Clients **MUST** obtain all such information by “synchronizing” with the **Ethereum**
1498 blockchain and parsing relevant events emitted by **Mixer**. Clients **MUST NOT** query
1499 the **Mixer** state via RPC.
- 1500 4. Clients **SHOULD** take steps to avoid being identified while synchronizing (see Ap-
1501 pendix C.2. For example, clients **SHOULD** vary the set of **Ethereum** nodes that they
1502 connect to, and **SHOULD NOT** always sync from the block following the last one that
1503 they processed.
- 1504 5. Clients **SHOULD NOT** re-request blocks or transaction receipts that are of particular
1505 interest to them. They **SHOULD** process all events, emitted by **Mixer**, in the same
1506 way.
- 1507 6. Clients **SHOULD NOT** make any RPC calls or change their externally visible behaviour
1508 in response to blocks or transaction receipts that are of interest to them.

1509 Use of contract queries

1510 We suggest that clients **SHOULD NOT** directly query the contract state, for the reasons
1511 discussed in Appendix C.2 and Appendix C.3 (and consequently, Section 4.2 suggests
1512 that the **Mixer** contract should, as far as possible, not expose public methods). The

1513 Zeth protocol prohibits direct queries of the state of $\widetilde{\text{Mixer}}$ (via *public* smart-contract
1514 functions) because they introduce a risk that client implementations will leak information
1515 by using them.

1516 If implementers choose to add public methods to the $\widetilde{\text{Mixer}}$ contract (for application-
1517 specific reasons), they should consider carefully the security issues raised in Appendix C.
1518 This specification assumes that `Mix` is the only public method of the $\widetilde{\text{Mixer}}$ contract.

1519 4.1.2 Note management

1520 `Mix` calls on the $\widetilde{\text{Mixer}}$ contract emit log events containing new commitment values,
1521 nullifiers, the new Merkle root and the secret data for new notes (encrypted using a key
1522 derived from the recipients public key). As clients synchronize with the latest state of
1523 the blockchain, they **MUST** read these events and correctly process the data they contain.

- 1524 1. Clients **MUST** process the `MixEventDType` event for every `Mix` transaction, in the
1525 order in which they appear in the blockchain.
- 1526 2. Clients implementing spending functionality **MUST** use the commitment values in
1527 events to track the state of the Merkle tree. The Merkle tree state will be used
1528 to generate Merkle paths for future transactions, and **MUST** be made available to
1529 the client without the need to query the contract. (Note that not all commitments
1530 must necessarily be persisted – see Section 4.3).
- 1531 3. Clients that can receive notes **MUST** attempt to decrypt the ciphertexts for every
1532 transaction (see Item 2 in Section 2.6).
- 1533 4. Clients **MUST NOT** perform any network-related action, including closing the RPC
1534 connection, dependent on successful/unsuccessful decryption of ciphertexts (see
1535 Appendix C.3).
- 1536 5. Clients that can receive notes **MUST** attempt to parse any successfully decrypted
1537 plaintext (that is, ensure it is well-formed as in Item 3a in Section 2.6).
- 1538 6. Clients **MUST NOT** perform any network-related action, including closing the con-
1539 nection, dependent on successful / unsuccessful parsing (see Appendix C.4).
- 1540 7. Clients that can receive notes **MUST** verify that successfully parsed plaintext data
1541 is the opening of the corresponding commitment in the transaction (see Item 3b
1542 in Section 2.6).
- 1543 8. Clients **MUST NOT** perform any network-related action, including closing the con-
1544 nection, dependent on whether the parsed note data is the opening of the corre-
1545 sponding commitment (see Appendix C.4).
- 1546 9. Clients **MUST** confirm that, after adding the new commitments, the local repre-
1547 sentation of the Merkle tree of commitments has a root consistent with the event
1548 data.

- 1549 10. Clients **SHOULD** keep a *local* record of the data related to valid decrypted notes.
1550 This will be required in order to spend the notes in a future transaction.
- 1551 11. Clients implementing spending functionality **SHOULD** process all nullifiers in Mix
1552 transaction events, checking for any corresponding notes previously recorded. Any
1553 such note should be marked as spent in the client's local record.

1554 4.1.3 Prepare arguments for Mix transaction

1555 Clients **MUST NOT** query **Ethereum** nodes while generating any arguments to a Mix call.
1556 In particular, Merkle paths **MUST** be calculated using the client's local representation of
1557 the Merkle tree of commitments that was constructed by parsing events.

1558 Where the zero-knowledge proof is generated by some external process, clients **MUST**
1559 put in place sufficient security schemes to ensure that:

- 1560 • they are communicating with an authentic proof generation process (not a man-
1561 in-the-middle), and
- 1562 • data sent to and from the proving process cannot be observed in transit and tam-
1563 pered with by a third party, and
- 1564 • the proof has been generated for the correct instance-witness pair¹

1565 Without these safe-guards, the operation of the system and the secret data required
1566 to spend the input notes may be compromised. See Appendix C.6.

1567 4.1.4 Wallet backup and recovery

1568 Given the restrictions placed on clients and their interaction with the **Mixer** contract,
1569 it follows that clients must store all data required to spend notes owned by their users'
1570 addresses, and to verify the validity of incoming notes. If this local data is lost, it must
1571 be reconstructed before client operations can resume.

1572 **Zeth** private keys (see Table 1.5) can be used to fully restore client state. In this
1573 case, clients **MUST** retrieve all events from the beginning of the **Mixer** contract's his-
1574 tory, decrypting notes and tracking nullifiers, as described in the previous sections, to
1575 reconstruct the set of unspent notes that they own.

1576 Without a backup of the private keys it is not possible to restore wallet state. As
1577 such, private keys are the minimal set of data that must be securely stored and backed
1578 up, and clients **SHOULD** provide support for this mode of recovery. However, to avoid the
1579 need to scan all events emitted by **Mixer** (a very expensive operation) implementations
1580 **SHOULD** also support back ups of further state data (such as the representation of the
1581 Merkle tree of note commitments, the set of unspent notes, etc) to allow more efficient
1582 modes of recovery.

¹Although 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.

1583 4.2 Contract security considerations

1584 Section 4.1 mentions several considerations for client implementations, concerning how
1585 they interact with the contract. These must be taken into account when authoring the
1586 contract code, to ensure that clients can securely retrieve the information they need to
1587 operate without encouraging them to perform insecure operations.

- 1588 1. **Mixer** MUST validate inputs, the contract needs to ensure that the primary inputs
1589 are elements of the scalar field \mathbb{F}_r (that is, they are in the range $\{0, \dots, r - 1\}$).
- 1590 2. **Mixer** MUST output events for valid Mix calls, including:
 - 1591 (a) commitment for each new note;
 - 1592 (b) nullifier for each spent note;
 - 1593 (c) value of new Merkle root of commitments;
 - 1594 (d) ciphertexts for each new note;
 - 1595 (e) implementation-specific data (such as the one-time sender public key specified
1596 in Section 3.5, required to decrypt the ciphertexts).
- 1597 3. The Mix function MUST be *payable*², to support non-zero *vin*.
- 1598 4. **Mixer** MUST NOT expose any public methods except for Mix.

1599 4.3 Efficiency and scalability

1600 4.3.1 Importance of performance

1601 Poor performance and scalability has several impacts on the viability of the system.

1602 Efficiency and performance are arguably most important for the **Mixer** contract,
1603 where gas usage directly affects the monetary cost of using **Zeth** to transfer value. That
1604 is, high gas costs could make transactions very expensive, and therefore not practical for
1605 many use-cases, undermining the utility and viability.

1606 High storage or compute requirements on the client would severely restrict the set
1607 of devices on which **Zeth** client software can run, and long delays when sending or
1608 receiving transactions can adversely affect the user-experience, discouraging some users
1609 and undermining the privacy promises of the system.

1610 Although the proof-of-concept implementation of **Zeth** is not intended to be used in a
1611 production environment, one of its aims is to demonstrate the practicality of the protocol
1612 in terms of transaction costs. Therefore, some of the techniques described here have been
1613 included in the proof-of-concept implementation, while in some cases implementers of
1614 production software may wish to make different trade-offs.

²see <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^{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 $2^{\text{MKDEPTH}} - 1$ 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 $\mathcal{O}(\log_2(\text{JSOUT}))$ layers (requiring $\mathcal{O}(\text{JSOUT})$ hashes) we reach the common ancestor of all new leaves and can update the Merkle tree by proceeding along a single branch (of approximately `MKDEPTH` - $\log_2(\text{JSOUT})$ layers). Thus, the cost of updating the Merkle tree for a single transaction has a fixed bound which is linear in `JSOUT` and

1654 MKDEPTH. However, this doubles the storage cost of the tree since non-leaf nodes must
1655 also be persisted.

1656 In the case of the client, the Merkle tree will only be used to generate proofs for notes
1657 owned by the user of the client. Thereby **Zeth** clients need only store nodes of the Merkle
1658 tree that are required for this purpose, and may discard all others. In particular, any
1659 full sub-tree need only contain nodes that are part of Merkle paths associated with the
1660 user's notes. Implementations that discard unnecessary nodes can achieve vast savings
1661 in storage space.

1662 **Zero-knowledge proof generation**

1663 As well as keeping the number of constraints as low as possible, it is important to ensure
1664 that the prover implementation is optimal and thereby that proving times are as short as
1665 possible. Proof generation should also exploit any available parallelism, to help reduce
1666 the time taken. This may require specific programming languages or frameworks to be
1667 used, necessitating that proof generation be performed by some external process (as is
1668 the case in the proof-of-concept implementation).

1669 The proof generation process can also be very memory intensive (in part due to the
1670 FFT calculations required), and so ensuring that enough RAM is present in the system
1671 is important to avoid long proof times.

1672 See Appendix C.6 for a discussion of related security concerns.

1673 **4.3.4 Contract performance**

1674 For most components of the contract, the set of operations to be performed is strictly
1675 defined and the set of possible algorithmic optimizations that can be made is limited.
1676 In these cases, it is important to ensure that code is benchmarked and optimized to
1677 a reasonable degree, to minimize gas costs. We note that apart from the number and
1678 type of compute instructions executed, store and lookup operations have a significant
1679 impact on the gas used. In particular, storing new values is more expensive than over-
1680 writing existing values, and a gas rebate is made when contracts release stored values.
1681 See [Woo19, Appendix H.1] for further details.

1682 The primary component in which algorithmic optimizations can be made is the
1683 Merkle tree of note commitments. The **Mixer** contract must compute (and store) the
1684 new Merkle root after adding the **JSOUT** new commitments as leaves.

1685 As in Section 4.3.3, the simplest possible implementation which stores only the data
1686 items at the leaves of the tree, requires the full root to be recomputed, involving $2^{\text{MKDEPTH}} -$
1687 1 hash invocations. This quickly becomes impractical for non-trivial values of MKDEPTH.

1688 The first-pass optimization (also described in Section 4.3.3) can be used to ensure
1689 that the cost of updating the Merkle tree (number of hash computations, stores and
1690 loads) is bounded by a constant that is linear in the Merkle tree depth. This is the
1691 strategy used in the proof-of-concept implementation of **Mixer**.

1692 It may be possible to gain further improvements in gas costs by discarding nodes
1693 from the Merkle tree that are not required. Unlike clients, **Mixer** is only required to

1694 compute the new Merkle root, and does not need to create or validate Merkle proofs
 1695 (as these are checked as part of the zero-knowledge proof). Consequently, *all* nodes in a
 1696 sub-tree can be discarded when the sub-tree is full, and the optimization is much simpler
 1697 to implement than on the client.

1698 Another possible strategy for decreasing the gas costs associated with Merkle trees
 1699 is *Merkle Shrubs*, described in [Lab19, Section 2.2]. Under this scheme, the contract
 1700 maintains a “frontier” of roots of sub-trees and Merkle proofs provided by clients (as
 1701 auxiliary inputs to the \mathbf{R}^Z circuit) contain a path from the leaf to one of the nodes in the
 1702 frontier. The gas savings in this scheme are due to the fact that, for new commitments,
 1703 the contract need only recompute the value of nodes from the leaf to the “frontier” (not
 1704 all the way to the root of the tree). However this comes at the cost of complexity in the
 1705 arithmetic circuit, which must verify a Merkle path to one of several frontier nodes.

1706 When choosing cryptographic primitives to be used on the EVM (and considering
 1707 the trade-off with other platforms, described in Section 4.3.1) it may be valuable to note
 1708 that the EVM supports so-called “pre-compiled contracts”. These behave like built-
 1709 in functions providing very gas-efficient access to certain algorithms, such as Keccak.
 1710 However, pre-compiled contracts exist only for a limited set of algorithms. Others must
 1711 be implemented using EVM instructions.

1712 4.3.5 Optimizing Blake2’s circuit.

1713 After presenting Blake2s circuit and its components working on little endian variables,
 1714 we show a few optimizations.

1715 Helper circuits

1716 We first define the following helper circuits needed in the Blake2s routine, operating on
 1717 w -bit long words.

1718 **XOR circuits** The following XOR circuits on w -bit long variables have been imple-
 1719 mented, we assume the inputs are boolean (this is not checked in these circuits),

- 1720 • “Classic” XOR circuit, which xors 2 variables,
 1721 $a \oplus b = c$;
- 1722 • XOR with constant, which xors two variables and a constant,
 1723 $a \oplus b \oplus c = d$, with c constant;
- 1724 • XOR with rotation, which xors two variables and rotates the result.
 1725 $a \oplus b \ggg r = c$, with r constant, and \ggg the rightward rotation [MJS15, Section
 1726 2.3]; i.e. for and constant $r < w$ we have $a_i \oplus b_i = c_{i+r \pmod{w}}$, for $i = 0, \dots, w$,

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: $a + b = c \pmod{2^w}$. 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 booleanness 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 booleanness.

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:

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

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

Triple modular addition: $a + b + c = d \pmod{2^w}$. 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 booleanness 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 booleanness.

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:

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

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

³Making 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 RICS programs is to make sure that the statement is correctly represented, without redundancy, in order to keep the constraint system as small as possible.

$G(a, b, c, d; x, y) \mapsto (a_2, b_2, c_2, d_2)$	$\text{getSigma}()$
1 : $a_1 \leftarrow a + b + x \pmod{2^w}$	1 : $\Sigma \in (\mathbb{N}^{16})^{10}$
2 : $d_1 \leftarrow d \oplus a_1 \ggg r_1$	2 : $\Sigma[0] \leftarrow (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)$
3 : $c_1 \leftarrow c + d_1 \pmod{2^w}$	3 : $\Sigma[1] \leftarrow (14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3)$
4 : $b_1 \leftarrow b \oplus c_1 \ggg r_2$	4 : $\Sigma[2] \leftarrow (11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4)$
5 : $a_2 \leftarrow a_1 + b_1 + y \pmod{2^w}$	5 : $\Sigma[3] \leftarrow (7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8)$
6 : $d_2 \leftarrow d_1 \oplus a_2 \ggg r_3$	6 : $\Sigma[4] \leftarrow (9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13)$
7 : $c_2 \leftarrow c_1 + d_2 \pmod{2^w}$	7 : $\Sigma[5] \leftarrow (2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9)$
8 : $b_2 \leftarrow b_1 \oplus c_2 \ggg r_4$	8 : $\Sigma[6] \leftarrow (12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11)$
9 : return a_2, b_2, c_2, d_2	9 : $\Sigma[7] \leftarrow (13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10)$
	10 : $\Sigma[8] \leftarrow (6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5)$
	11 : $\Sigma[9] \leftarrow (10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0)$
	12 : return Σ

Figure 4.1: G primitive [MJS15, Section 3.1]

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

1749 Blake2s routine circuit

1750 We define in this section the circuit of the Blake2 routine (see [MJS15, Section 3.1]
 1751 and Fig. 4.1) known as “ G function” [ANWOW13, Section 2.4]. G is based on ChaCha
 1752 encryption [Ber08a]. It works on w -bit long words, and presents $8 \cdot w + 10$ constraints.
 1753 The function mixes a state $(a, b, c$ and $d)$ with the inputs $(x$ and $y)$ and returns the
 1754 updated state.

1755 This circuit does not check the booleanness of the inputs or state. However, given that
 1756 the state is boolean, the output is automatically boolean due to the use of the modular
 1757 addition circuits.

1758 For Blake2s, we have $w = 32$, $r_1 = 16$, $r_2 = 12$, $r_3 = 3$ and $r_4 = 7$.

1759 Blake2s compression function circuit

The compression function is defined as follows, for more details see Fig. 4.3,

$$\text{Blake2sC} : \mathbb{B}^n \times \mathbb{B}^{2n} \times \mathbb{B}^{n/4} \times \mathbb{B}^{n/4} \rightarrow \mathbb{B}^n.$$

1760 Blake2C takes as input a state $h \in \mathbb{B}^n$ which is used as chaining value when hashing,
 1761 a message to compress $x \in \mathbb{B}^{2n}$, a message length written in binary $t \in \mathbb{B}^{n/4}$ which is
 1762 incremented when hashing and a binary flag $f \in \mathbb{B}^{n/4}$ to tell whether the current block
 1763 is the last to be compressed to prevent length extension attacks.

1764 Blake2C uses the G function iteratively over **rounds** number of rounds on a state
 1765 and message. The constant initialization vector IV and the permutation table Σ are
 1766 hard-coded. Blake2sC works in little endian (see [MJS15, Section 2.4]) on n -bit long
 1767 variables ($n = 256$), w -bit long words ($w = 32$), and the rotation constants specified

1768 in Section 4.3.5 (see [MJS15, Section 2.1]). We have the following constants (see speci-
1769 fications [ANWOW13] and [MJS15, Section 2.2]),

- 1770 • **IV** is the $(8 \cdot w)$ -bit long initialization vector; it corresponds to the first w bits of the
1771 fractional parts of the square roots of the first eight prime numbers $(2, 3, 5, 7, \dots)$
1772 (see [MJS15, Section 2.6]);
- 1773 • Σ are the $10 \cdot 16$ permutation constants of Blake2 (see Fig. 4.2 and [MJS15, Section
1774 2.7]);
- 1775 • **rounds**, the number of rounds: 10 for Blake2sC, 12 for Blake2bC.

1776 We have the following variables (see specifications [ANWOW13] and [MJS15, Section
1777 2.2]),

- 1778 • **H** is the $(8 \cdot w)$ -bit long initial state while v is the $(16 \cdot w)$ -bit long final state;
- 1779 • $T[i]$ are two w -bit long counters encoding the block length;
- 1780 • $F[i]$ are two w -bit long finalization flags. We set the first one $F[0]$ to $2^w - 1$ to state
1781 when the input block is the last one to be hashed. The second, $F[1] = 0$ is only set
1782 for tree hashing mode (which is not our case) and is therefore unused.

1783 We introduce the following functions to write Blake2C (see specifications [ANWOW13]
1784 and [MJS15, Section 2.6]):

- 1785 • The function **prime** takes a positive integer i as input and outputs the i -th prime
1786 number;
- 1787 • The function **dec** takes a real number x as input outputs its positive decimal part.

1788 This circuit presents $((64 \cdot \mathbf{rounds} + 8) \cdot w + 8 \cdot \mathbf{rounds} + 10)$ constraints. For Blake2sC,
1789 as $w = 32$ and **rounds** = 10, we have 21536 constraints.

1790 We do not check the input block booleaness in this circuit. Given that the initial
1791 state is boolean, the output is automatically boolean. This can be proved iteratively by
1792 the booleaness of **G** primitive's output.

1793 **Security requirement.** The inputs to Blake2sC **MUST** be boolean.

1794 Blake2s hash function

The hash function is defined as follows, for more details see Fig. 4.3,

$$\mathbf{Blake2s} : \mathbb{B}^{\leq 2n} \times \mathbb{B}^* \rightarrow \mathbb{B}^n$$

1795 Blake2 takes as input a hash key $k \in \mathbb{B}^n$ and the message to hash $x \in \mathbb{B}^{2n}$. Blake2
1796 uses the Blake2C function iteratively over each $2n$ -bit long chunk of the padded message.
1797 If the key is non null, it is used as the first block to be hashed. The constant initialization
1798 vector **IV** and part of the parameter block **PB** are hard-coded. We have the following
1799 constants (see specifications [ANWOW13] and [MJS15, Section 2.2]),

Blake2C(h, m, t, f)

```

1 :   $\mathbf{T}, \mathbf{F}, \mathbf{H}, \mathbf{IV}, v \in (\mathbb{B}^w)^2 \times (\mathbb{B}^w)^2 \times (\mathbb{B}^w)^8 \times (\mathbb{B}^w)^8 \times (\mathbb{B}^w)^{16}$ 
2 :   $\{\mathbf{IV}[i]\}_{i \in [8]} \leftarrow \left\{ \left\lfloor 2^w \cdot \text{dec}(\sqrt{\text{prime}(i+1)}) \right\rfloor \right\}_{i \in [8]}$ 
3 :   $\Sigma \leftarrow \text{getSigma}()$ 
4 :   $\{\mathbf{H}[i]\}_{i \in [8]} \leftarrow \{h[i \cdot w : (i+1) \cdot w]\}_{i \in [8]}$ 
5 :   $\{m[i]\}_{i \in [8]} \leftarrow \{x[i \cdot w : (i+1) \cdot w]\}_{i \in [8]}$ 
6 :   $\mathbf{T}[0], \mathbf{T}[1] \leftarrow t[w:2w], t[0:w]$ 
7 :   $\mathbf{F}[0], \mathbf{F}[1] \leftarrow f[w:2w], f[0:w]$ 
8 :   $\{v[i]\}_{i \in [8]} \leftarrow \{\mathbf{H}[i]\}_{i \in [8]}$ 
9 :   $\{v[i+8]\}_{i \in [8]} \leftarrow \{\mathbf{IV}[i]\}_{i \in [8]}$ 
10 :  $v[12], v[13] \leftarrow v[12] \oplus \mathbf{T}[0], v[13] \oplus \mathbf{T}[1]$ 
11 :  $v[14], v[15] \leftarrow v[14] \oplus \mathbf{F}[0], v[15] \oplus \mathbf{F}[1]$ 
12 : foreach  $r \in [\text{rounds}]$  do
13 :    $\tau \leftarrow \Sigma[r \pmod{15}]$ 
14 :    $v[0], v[4], v[8], v[12] \leftarrow \mathbf{G}(v[0], v[4], v[8], v[12], m[\tau[0]], m[\tau[1]])$ 
15 :    $v[1], v[5], v[9], v[13] \leftarrow \mathbf{G}(v[1], v[5], v[9], v[13], m[\tau[2]], m[\tau[3]])$ 
16 :    $v[2], v[6], v[10], v[14] \leftarrow \mathbf{G}(v[2], v[6], v[10], v[14], m[\tau[4]], m[\tau[5]])$ 
17 :    $v[3], v[7], v[11], v[15] \leftarrow \mathbf{G}(v[3], v[7], v[11], v[15], m[\tau[6]], m[\tau[7]])$ 
18 :    $v[0], v[5], v[10], v[15] \leftarrow \mathbf{G}(v[0], v[5], v[10], v[15], m[\tau[8]], m[\tau[9]])$ 
19 :    $v[1], v[6], v[11], v[12] \leftarrow \mathbf{G}(v[1], v[6], v[11], v[12], m[\tau[10]], m[\tau[11]])$ 
20 :    $v[2], v[7], v[8], v[13] \leftarrow \mathbf{G}(v[2], v[7], v[8], v[13], m[\tau[12]], m[\tau[13]])$ 
21 :    $v[3], v[4], v[9], v[14] \leftarrow \mathbf{G}(v[3], v[4], v[9], v[14], m[\tau[14]], m[\tau[15]])$ 
22 : return  $\|_{i=0}^8 \mathbf{H}[i] \oplus v[i] \oplus v[i+8]$ 

```

Figure 4.3: Blake2 compression function [MJS15, Section 3.2]. Set n , w and \mathbf{G} 's constants to obtain Blake2sC.

- 1800 • \mathbf{IV} is the $(8 \cdot w)$ -bit long Initialization Vector; it corresponds to the first w bits of the
1801 fractional parts of the square roots of the first eight prime numbers $(2, 3, 5, 7, \dots)$
1802 (see [MJS15, Section 2.6]).
- 1803 We have the following variables (see specifications [ANWOW13] and [MJS15, Section
1804 2.2]),
- 1805 • \mathbf{PB} is the $(16 \cdot w)$ -bit long parameter block used to initialize the state (see [MJS15,
1806 Section 2.5]). In big endian encoding, the first byte corresponds to the digest
1807 length (fixed to 32 bytes), the second byte to the key length, the third and fourth
1808 bytes correspond to the use of the serial mode;
- 1809 • $\mathbf{H} \in \mathbb{B}^{\text{BLAKE2sCLEN}}$, the chaining value.

Blake2(k, x)

```

1 : H, IV, PB  $\in \mathbb{B}^{8w} \times \mathbb{B}^{8w} \times \mathbb{B}^{8w}$ 
2 : PB  $\leftarrow \text{pad}_{8 \cdot w}(\text{encode}_{\mathbb{N}}(0x0101)) \parallel \text{pad}_w(\text{encode}_{\mathbb{N}}(\lceil \text{length}(k)/\text{BYTELEN} \rceil)) \parallel \text{encode}_{\mathbb{N}}(0x20)$ 
3 : IV  $\leftarrow \parallel_{i=0}^8 \left[ 2^w \cdot \text{dec}(\sqrt{\text{prime}(i+1)}) \right]$ 
4 : H  $\leftarrow \text{PB} \oplus \text{IV}$ 
5 :  $y \leftarrow x$ 
6 : if  $\text{length}(k) \neq 0$  do
7 :    $y \leftarrow \text{pad}_{2n}(k) \parallel y$ 
8 :    $z \leftarrow \text{pad}_{2n \cdot \lceil \text{length}(y)/2n \rceil}(y)$ 
9 :   for  $i \in [\lceil \text{length}(z)/2n \rceil]$  do
10 :    if  $i = \lceil \text{length}(z)/2n \rceil - 1$  do
11 :      H  $\leftarrow \text{Blake2C}(H, z[i \cdot 2n:(i+1) \cdot 2n], \text{pad}_{2w}(\text{encode}_{\mathbb{N}}(\lceil \text{length}(y)/\text{BYTELEN} \rceil)), \text{pad}_{2w}(\text{encode}_{\mathbb{N}}(2^w - 1)))$ 
12 :    else
13 :      H  $\leftarrow \text{Blake2C}(H, z[i \cdot 2n:(i+1) \cdot 2n], \text{pad}_{2w}(\text{encode}_{\mathbb{N}}((i+1) \cdot 2n/\text{BYTELEN})), \text{pad}_{2w}(0))$ 
14 :   return H

```

Figure 4.4: Blake2 hash function [MJS15, Section 3.3]. Set $n = 16w$ and G 's constants accordingly to obtain Blake2s.

1810 We do not check the input block booleaness in this circuit. Given that the initial
1811 state is boolean, the output is automatically boolean. This can be proved iteratively by
1812 the booleaness of Blake2C primitive's output.

1813 **Security requirement** To ensure the correct use of Blake2s, Blake2s's inputs **MUST** be
1814 boolean.

1815 Optimizing the circuits

1816 The above helper circuits form the building blocks of the Blake2s compression function.
1817 We show here two exclusive methods to optimize these circuits.

1818 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:

$$\left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i \right) \cdot \left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i - 2^w \right) = 0 \quad (4.5)$$

$$\forall j \in \{0, \dots, w-1\}, (c_j - 0) \cdot (c_j - 1) = 0 \quad (4.6)$$

1819 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).

1820 These equations describe $w + 1$ constraints to prove the bit equality $a + b = c$ (note
1821 that an additional $2 \cdot w$ constraints would be required to prove the booleanness of input
1822 variables a and b). We now explain how we obtained them.

1823 *Proof.* The most straightforward way to prove that $a + b = c \pmod{2^w}$ and c booleanness
1824 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 booleanness. 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 $2^{2 \cdot w}$.

$$\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i = c_w \cdot 2^w \quad (4.7)$$

$$2^w \cdot (c_w - 0) \cdot 2^w \cdot (c_w - 1) = 0 \quad (4.8)$$

We finally replace $c_w \cdot 2^w$ in Eq. (4.8) by the value from Eq. (4.7).

$$\begin{aligned} 0 &= 2^w \cdot (c_w - 0) \cdot 2^w \cdot (c_w - 1) = 2^w \cdot c_w \cdot (2^w \cdot c_w - 2^w) \\ &= \left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i \right) \cdot \left(\left(\sum_{i=0}^{w-1} (a_i + b_i - c_i) \cdot 2^i \right) - 2^w \right) \end{aligned}$$

1825 This results in Eq. (4.5) and Eq. (4.6). All references to c_w have disappeared and, with
1826 a single multiplication by a variable, we still have bilinear gates. \square

1827 **Triple modular addition: $a + b + c = d \pmod{2^w}$.** To optimize, we use the
1828 above circuit twice. We define a temporary variable d' such that $a + b = d' \pmod{2^w}$.
1829 As such, we have $c + d' = d \pmod{2^w}$. As d' is the addition of two w -bit long variables,
1830 it is $(w + 1)$ -bit long. However as we evaluate the sum modulo 2^w , we discard the last bit
1831 of d' . We proceed similarly for d . To ensure that d is boolean, we check the booleanness
1832 of the $w + 1$ bits of d as well as the booleanness of the last bit of d' (to account for d 's
1833 $w + 2^{th}$ bit in the original expression $(a + b + c = d \pmod{2^w})$).

We thus obtain the following circuit with $w + 2$ constraints,

$$\begin{aligned} \left(\sum_{i=0}^{w-1} (a_i + b_i - d'_i) \cdot 2^i \right) \cdot \left(\sum_{i=0}^{w-1} (a_i + b_i - d'_i) \cdot 2^i - 2^w \right) &= 0 \\ \left(\sum_{i=0}^{w-1} (c_i + d'_i - d_i) \cdot 2^i \right) \cdot \left(\sum_{i=0}^{w-1} (c_i + d'_i - d_i) \cdot 2^i - 2^w \right) &= 0 \\ \forall j \in \{0, \dots, w - 1\}, (d_j - 0) \cdot (d_j - 1) &= 0 \end{aligned}$$

1834 These optimizations lead to a gain of 320 constraints ($= 4 \cdot 8 \cdot \text{rounds}$).

1835 **Optimizing Blake2s routine’s circuit** As seen in Fig. 4.1, our routine presents 2
1836 double and 2 triple modular additions. Each of these circuits comprises at least one
1837 modular constraint which pack several w -bit long variables. The circuit is however
1838 processed in \mathbb{F}_r , that is to say most integers can be written over **FIELD**CAP bits. We can
1839 thus batch the modular constraints. As the **G** primitive performs 2 double modular and
1840 2 triple modular, we have in total 6 modular checks per iteration. We can batch up to
1841 $\text{FIELD}CAP/w$ constraints together. For $w = 32$ and $\text{FIELD}CAP \geq 224$ (which holds for
1842 **BN-254** and **BLS12-377**), we can encode up to 7 words per field element, that is to say
1843 we can include all the modular constraints into a single one.

1844 This optimization leads to a gain of 274 constraints ($= 4 \cdot 8 \cdot 10 - \lceil \frac{4 \cdot 8 \cdot 10}{7} \rceil$).

1845 **Optimization conclusion** Using the more efficient optimization on the modular ad-
1846 ditions, the Blake2s compression function comprises 21216 constraints.

1847 **Increasing the PRF security with Blake**

1848 As Blake2 comprises a personalization tag in its parameter block **PB**, one could ensure the
1849 independence of the PRFs by writing different tags for each of them (we would be able to
1850 consider up to 2^{30} inputs and outputs). We did not choose to write this enhancement in
1851 the instantiation to keep a general tagging method in case of a change of hash function.

1852 **4.4 Encryption of the notes**

1853 In this section we give some remarks concerning the implementation of the **Zeth** en-
1854 cryptation scheme, described in Section 3.5. As noted, there are several details in the
1855 specification of the underlying primitives which can impact security if not carefully im-
1856 plemented. The following list is by no means exhaustive but includes several details
1857 noted during development of the proof-of-concept system.

- 1858 • Private keys for **Curve25519** **MUST** be randomly generated as 32 bytes where the
1859 first byte is a multiple of 8, and the last byte takes a value between 64 and 127
1860 (inclusive). Further details are given in [Ber06], including an example algorithm
1861 for generation. Implementations **MUST** take care to ensure that their code, or any
1862 external libraries they rely upon, correctly perform this step.
- 1863 • A similar observation holds for **Poly1305** in which the r component of the **MAC**
1864 key (r, s) **MUST** be *clamped* in a specific way (see Section 3.5.3). This step is also
1865 essential and **MUST** be performed.
- 1866 • In the implementation of the **ChaCha** stream cipher, correct use of the *key*, *counter*
1867 and *nonce* **MUST** be ensured in order to adhere to the standard and guarantee the
1868 appropriate security properties.

1869 During the proof-of-concept implementation it was not obvious that the cryptogra-
1870 phy library⁴ adhered to the specifications with respect to the above points. In particular,
1871 it was not clear whether key clamping was performed at generation time and/or when
1872 performing operations. Moreover, the interface to the **ChaCha** cipher accepted a differ-
1873 ent set of input parameters (namely *key* and *nonce* with no *counter*). This left some
1874 ambiguity about the responsibility for clamping, and whether the **ChaCha** block data
1875 would be updated as described in the specification. Details of how this was resolved are
1876 given in the proof-of-concept encryption code, which may prove a useful reference for
1877 implementers⁵.

⁴<https://cryptography.io/en/latest/>

⁵see <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 TR-NM involving a polynomial-time adversary \mathcal{A} as described below.

Definition A.0.1. Let DAP be a (candidate) Decentralized Anonymous Payment scheme.

$$\text{DAP} = (\text{Setup}, \text{GenAddr}, \text{SendTx}, \text{VerifyTx}, \text{Receive})$$

We say that DAP is TR-NM secure if, for every $\text{poly}(\lambda)$ -time adversary \mathcal{A}

$$\text{Adv}_{\text{DAP}, \mathcal{A}}^{\text{tr-nm}}(\lambda) < \text{negl}(\lambda),$$

where $\text{Adv}_{\text{DAP}, \mathcal{A}}^{\text{tr-nm}}(\lambda) = \Pr[\text{TR-NM}(\text{DAP}, \mathcal{A}, \lambda) = 1]$ is \mathcal{A} 's advantage in the TR-NM experiment.

Below, we adapt [BSCG⁺14, Appendix C.2] to our specific DAP—Zeth.

We start by describing the TR-NM experiment. Given a (candidate) Zeth DAP, adversary \mathcal{A} , and security parameter λ , the (probabilistic) game $\text{TR-NM}(\text{DAP}, \mathcal{A}, \lambda)$ 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 $pp \leftarrow \text{Setup}(\lambda)$ and sends pp to \mathcal{A} . Next, \mathcal{C} initializes a DAP oracle O^{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 **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 O^{DAP} in response to **SendTx** queries, there exists $tx_{\text{Mix}} \in T$ such that:

1. tx_{Mix} was not inserted in L by \mathcal{A} ;
2. $tx_{\text{Mix}}^*.data \neq tx_{\text{Mix}}.data$;
3. $\text{VerifyTx}(pp, tx_{\text{Mix}}^*, L') = 1$ 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*. ρ is chosen at random. Finally, the NP-relation used in Zeth, now denoted \mathbf{R}^{mal} , becomes the following:

- For each $i \in [\text{JSIN}]$:

1. $\text{aux.jsins}[i].\text{znote.apk} = \text{Blake2s}(\text{tag}_{\text{ask}}^{\text{addr}} \parallel \text{pad}_{\text{BLAKE2SCLEN}}(0))$
with $\text{tag}_{\text{ask}}^{\text{addr}}$ defined in Section 3.1.3
2. $\text{aux.jsins}[i].\text{nf} = \text{Blake2s}(\text{tag}_{\text{ask}}^{\text{nf}} \parallel \text{aux.jsins}[i].\text{znote}.\rho)$
with $\text{tag}_{\text{ask}}^{\text{nf}}$ defined in Section 3.1.3
3. $\text{aux.jsins}[i].\text{cm} = \text{Blake2s}(\text{aux.jsins}[i].\text{znote}.r \parallel m)$
with $m = \text{aux.jsins}[i].\text{znote.apk} \parallel \text{aux.jsins}[i].\text{znote}.\rho \parallel \text{aux.jsins}[i].\text{znote}.v$
4. $(\text{aux.jsins}[i].\text{znote}.v) \cdot (1 - e) = 0$ is satisfied for the boolean value e set such that if $\text{aux.jsins}[i].\text{znote}.v > 0$ then $e = 1$.
5. The Merkle root mkroot' obtained after checking the Merkle authentication path $\text{aux.jsins}[i].\text{mkpath}$ of commitment $\text{aux.jsins}[i].\text{cm}$, with MKHASH, is equal to prim.mkroot if $e = 1$.
6. $\text{prim.nfs}[i]$
 $= \{\text{Pack}_{\mathbb{F}_r}(\text{aux.jsins}[i].\text{nf}[k \cdot \text{FIELD CAP}:(k+1) \cdot \text{FIELD CAP}])\}_{k \in [\lfloor \text{PRNFOUTLEN}/\text{FIELD CAP} \rfloor]}$

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

1. $\text{prim.cms}[j] = \text{Blake2s}(\text{aux.znotes}[j].r \parallel m)$
with $m = \text{aux.znotes}[j].\text{apk} \parallel \text{aux.znotes}[j].\rho \parallel \text{aux.znotes}[j].v$

- $\text{prim.rsd} = \text{Pack}_{\text{rsd}}(\{\text{aux.jsins}[i].\text{nf}\}_{i \in [\text{JSIN}]}, \text{aux.vin}, \text{aux.vout})$
- Check that the “joinsplit is balanced”, i.e. check that the joinsplit equation holds:

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

1928 A.1.1 The attack

1929 In order to win the game TR-NM on the weak Zeth DAP above, an adversary \mathcal{A} inter-
 1930 cepts a target transaction tx_{Mix} by passively listening to the network (remember that
 1931 transactions are broadcasted to the **Ethereum** network in order to be mined, see Sec-
 1932 tion 1.2.2), extracts the zk-proof and primary inputs from $tx_{\text{Mix}}.data$ and uses these
 1933 extracted pieces of information in order to create a malicious transaction tx_{Mix}' , where
 1934 the ciphertexts are replaced by arbitrary data. The adversary can then broadcast tx_{Mix}'
 1935 to the network in order for it to be mined. If the malicious transaction gets mined before
 1936 the legitimate one, the input notes become spent and the ciphertexts are undecryptable
 1937 making the new notes unredeemable (by any Zeth user!), since all attempts to decrypt
 1938 the ciphertexts will fail (see Section 2.6).

```

TxMalGen( $sk'_{\text{ECDSA}}, nce_{in}, tx_{\text{Mix}}$ )
1 :  $p \leftarrow tx_{\text{Mix}}.gasP + 1$ 
2 :  $l \leftarrow tx_{\text{Mix}}.gasL + 1$ 
3 :  $zdata' \leftarrow tx_{\text{Mix}}.data$ 
4 :  $zdata'.ciphers \leftarrow \mathbb{B}^*$ 
5 :  $tx_{raw} \leftarrow \{nce : nce_{in}, gasP : p, gasL : l, to : tx_{\text{Mix}}.to, val : tx_{\text{Mix}}.val, data : zdata'\};$ 
6 :  $\sigma_{\text{ECDSA}} \leftarrow \text{SigSch}_{\text{ECDSA}}.\text{Sig}(sk'_{\text{ECDSA}}, \text{Keccak256}(tx_{raw}));$ 
7 :  $tx_{final} \leftarrow \{tx_{raw}, v : \sigma_{\text{ECDSA}}.v', r : \sigma_{\text{ECDSA}}.r', s : \sigma_{\text{ECDSA}}.s'\};$ 
8 : return  $tx_{final}$ ;

```

Figure A.1: Transaction malleability attack function TxMalGen

1939 As shown on Fig. A.1, during the attack, the adversary extracts the proof and pri-
 1940 mary inputs from the honest transaction, and replaces the ciphertexts by some arbitrary
 1941 information. The attacker then formats this data into a transaction that calls the Mix
 1942 function of **Mixer**, and submits it to the network. While the data fields ($tx_{\text{Mix}}.data$
 1943 and $tx_{\text{Mix}}'.data$) are different, the nullifiers revealed by both transactions are the same
 1944 (i.e. $tx_{\text{Mix}}.data.proof = tx_{\text{Mix}}'.data.proof$, and $tx_{\text{Mix}}.data.prim = tx_{\text{Mix}}'.data.prim$).
 1945 As a consequence, if the adversary makes sure that tx_{Mix}' satisfies all the checks of
 1946 **EthVerifyTx** (Section 1.2.2), he can ensure that **ZethVerifyTx**(tx_{Mix}') will return the same
 1947 value as **ZethVerifyTx**(tx_{Mix}). Furthermore, if $tx_{\text{Mix}}'.gasP > tx_{\text{Mix}}.gasP$, then the adver-
 1948 sary maximizes his chances of having his transaction mined first (Section 1.2.2), and so
 1949 maximizes the chances for the malleability attack to be successful; leading to lost funds
 1950 on **Mixer**.

1951 **Remark A.1.1.** Note that, although not directly contained within the *data* field of a
 1952 **Mixer** call transaction, the **Ethereum** address $\mathcal{S}_{\mathcal{E}}.Addr$ of the transaction sender is also
 1953 used by the **Mixer** call (this is either the calling contract's address, or the transaction
 1954 signer's address recovered as described in Remark 1.2.1). In particular, the balance
 1955 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 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 SUF-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 $hsig = CRH(vk)$, where CRH is a collision resistant hash function, see [BSCG⁺14], and derives a value $h_i = PRF_{ask_i}^{pk}(hsig)$, for each input note (i.e. $i \in [JSIN]$), 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 $\{h_i\}_{i \in [JSIN]}$ 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 UF-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

1992 this leads to the check verifying that $hsig$ is correctly computed to fail. If the adversary
1993 attempts to change $hsig$, the zk-proof verification fails as the NP-statement has changed.
1994 Hence, any attempt to carry out a malleability attack results in the violation of at least
1995 one check in the verification of the transaction on the DAP. The solution presented
1996 effectively solves the transaction-malleability attack initially described.

1997 **Remark A.2.1.** The one-timeness property of the signature scheme was required in
1998 **ZeroCash** to retain anonymity. It also makes analysing non-adaptive adversary sufficient.
1999 As **Ethereum** transaction senders need to pay the gas cost associated with their trans-
2000 actions, the senders are not anonymous. This said, making sure that **Zeth** is designed
2001 with anonymity in mind is worth the effort in order to minimize information leakages
2002 and be ready if/when **Ethereum** incorporates protocol changes that enable anonymous
2003 transactions.

2004 A.2.2 Zcash’s solution

2005 In addition to the changes aforementioned, **Zcash**’s solution [ZCa19] also consists of:

- Redefining the variable $hsig$ as,

$$hsig = \text{CRH}(\text{randomSeed}, \{nf_i\}_{i \in [\text{JSIN}]}, vk)$$

2006 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 ($j \in [\text{JSOUT}]$) and ensure their uniqueness (with overwhelming probability).

2010 These changes were made to prevent the Faerie Gold attack [ZCa19, Section 8.4], as well
2011 as to prevent linkability: if $hsig$ were repeated in two transactions, the circuit would
2012 leak, via $\{h_i\}_{i \in [\text{JSIN}]}$, the fact that the input notes in both transactions were spent with
2013 the same ask_i (if that were the case).

2014 More particularly, using the input notes’ nullifiers to derive $hsig$ ensures that $hsig$ is
2015 unique with overwhelming probability for all *accepted* transaction. Furthermore, us-
2016 ing randomSeed ensures the uniqueness of $hsig$ for transactions *in transit* (as before
2017 validation there may be several in transit transactions with the same set of nullifiers).

2018 A.2.3 Solution on Ethereum

2019 As described in the **Ethereum** yellow paper [Woo19, Appendix F], **Ethereum** transactions
2020 are ECDSA signed. Further, as described in Section 2.3, the one-time signature used to
2021 sign the Mix data also signs the **Ethereum** address used to sign the transaction. As such,
2022 any modification to the transaction object will result in a new transaction hash, and
2023 any attempt to sign the transaction with a different ECDSA key will be rejected by the
2024 **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:

$$hsig = \text{CRH}(\{nf_i\}_{i \in [\text{JSIN}]}, \mathcal{S}_{\mathcal{E}}.Addr)$$

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}_r and they can be written over FIELDLEN bits. Note that the projection of $\mathbb{B}^{\text{FIELDLEN}}$ onto \mathbb{F}_r formed by interpreting elements in $\mathbb{B}^{\text{FIELDLEN}}$ as FIELDLEN -bit numbers and reducing modulo r , is surjective.

When we pass the primary inputs to the **Mixer** contract, they are interpreted as elements of $\mathbb{B}^{\text{ETHWORDLEN}}$, and $\mathbb{B}^{\text{FIELDLEN}} \subset \mathbb{B}^{\text{ETHWORDLEN}}$. As previously noted, this means that there exist pairs of elements in $\mathbb{B}^{\text{ETHWORDLEN}}$ with the same projection in \mathbb{F}_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 $\mathbb{B}^{\text{ETHWORDLEN}}$) 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}_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 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}_r , that is to say that they are smaller than 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.

2076 Appendix C

2077 Side-channel attacks and 2078 information leaks

2079 The following subsections describe several side-channel attacks and possible weaknesses
2080 that implementers should be aware of and attempt to mitigate.

2081 We consider cases in which the attacker is able to observe the RPC communications
2082 between **Zeth** client software, and Ethereum P2P nodes. This situation may occur if an
2083 observer is able to monitor the network traffic between the Ethereum node and the **Zeth**
2084 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.

2085

2086 C.1 Counterfeit data

2087 Malicious Ethereum nodes or attackers able to compromise the network have the oppor-
2088 tunity to send invalid data to RPC clients. This could be used to inject invalid data
2089 into the client's record of state, which could prevent it from generating valid **Mix** calls
2090 or allow it to be identified in the future. In general, data from any remote host should
2091 be treated as malicious, unless accompanied by evidence that convinces the client of its
2092 authenticity.

2093 In the case of Ethereum event logs (the main source of data used to track the on-
2094 chain state – see Section 4.1.1 for details), clients **MUST** leverage the consensus evidence
2095 and block headers to verify that log data is genuine and has been committed to the
2096 blockchain. See Section 1.2.3 for further information about how such data is secured.

2097 C.2 Data leaked during synchronization

2098 In order to receive private payments and keep their local data up-to-date, **Zeth** client
2099 software **MUST** scan the blockchain and process *all* the event data emitted by **Mixer**
2100 during Mix calls (as described in Section 4.1.1). There are several issues to consider
2101 when determining exactly how and when this “synchronization” takes place.

2102 Client implementations that only connect to the RPC endpoint in response to user
2103 input, or in preparation for performing a Mix call, may leak information. Observers may
2104 deduce that such client are likely to be the recipient of a recent or upcoming transaction,
2105 or that they may be about to perform a Mix call.

2106 Similarly, payment provider software that only listens for events when awaiting a
2107 transaction, and remains disconnected otherwise, may reveal that it is the recipient of
2108 an upcoming transaction, and possibly *which* transaction or block it was paid by (based
2109 on when it stops listening).

2110 Further, consider wallet software that performs RPC operations to explicitly wait
2111 for the Ethereum transaction corresponding to a specific Mix call. This would most
2112 likely be for transactions emitted by the **Zeth** client, in order to inform the user and
2113 update the wallet state once the payment is complete (but could possibly happen on
2114 the receiver side, if he somehow knows the ID of the transaction of interest – e.g. via
2115 off-chain communication with the sender). If such a *wait* procedure is implemented by
2116 querying the status of a specific transaction by its ID, or by listening for blocks *until*
2117 the transaction of interest is received, the connected Ethereum node may infer that this
2118 client is interested in the transaction, and likely to be the sender or recipient.

2119 Consider a client which periodically connects to some Ethereum node and requests
2120 all relevant data from the last block it saw, up to the latest block available. Each client
2121 will have information up to some block n (where n varies per client), and n is known to
2122 the Ethereum node that served the client. The client could then potentially be identified
2123 by n (even if it hides its IP for each connection) since a client that connects and queries
2124 **Zeth** transactions from block $n + 1$ reveals that it is one of the clients who synced up to
2125 block n when it last connected.

2126 Note that, if the client always broadcasts the Mix transaction via this same Ethereum
2127 node, then the Ethereum node may already deduce that the client is the sender. However,
2128 implementations may wish to use techniques (such as sending transactions from other
2129 nodes or hiding their IP address in other way) to obfuscate any relationship between
2130 transactions and the clients that originated them.

C.3 Queries on successful decryption

The event data emitted by $\widetilde{\text{Mixer}}$ 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 $\widetilde{\text{Mixer}}$ 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 $\widetilde{\text{Mixer}}$ **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}^Z , nor by the contract (which is unable to decrypt it). Hence, $\widetilde{\text{Zeth}}$ allows such transactions with malicious ciphertexts to be accepted by the $\widetilde{\text{Mixer}}$ contract, and clients must handle this case with care.

2159 In the case described in [TBP20], there is no distinction between “client” or “wallet”
2160 software, and the underlying P2P nodes. Before a fix was applied (see [zcab]), nodes
2161 explicitly rejected transactions of the above form, proving to their peers that they were
2162 able to decrypt the ciphertext and were therefore the intended recipient.

2163 In **Zeth**, P2P nodes and wallet software are separated, so there will be no explicit
2164 rejection of the transaction. However, careless error handling (such as exceptions which
2165 causes the RPC connection to be closed) could potentially be detected by the connected
2166 Ethereum node. As in the “REJECT Attack”, this reveals that the connected RPC
2167 client is the intended recipient of a transaction, and the owner of the corresponding
2168 encryption key.

2169 C.5 Using (and retrieving) nullifiers

2170 Any non-trivial wallet implementation will need to track which of the user’s **Zeth** notes
2171 have been spent, and which are still available. Naturally, the wallet software could mark
2172 the notes as it broadcasts transactions that spend them. However, this approach is
2173 subject to several problems.

2174 Firstly, for each note spent, the client software must record the ID of the spending
2175 transaction, in order to track it and confirm that it is accepted into a block. Once each
2176 spending transaction is accepted the client can finally mark the appropriate **Zeth** notes
2177 as “spent”. This requires significant complexity in order to asynchronously mark the
2178 notes, and to deal with the issues described in Appendix C.2.

2179 Secondly, this approach does not support multiple wallets using the same key, or
2180 wallets being restored from **Zeth** addresses. A user that wishes to rebuild his wallet
2181 (see the discussion in Section 4.1.4), or check for any spending activity by other wallets,
2182 would not be able to do so by simply scanning the blockchain.

2183 By using the nullifiers passed to **Mix** calls, clients can determine the availability of
2184 notes in a more robust way. That is, to determine whether a note is spent or available,
2185 the client can compute the nullifier and check whether that nullifier has been seen by
2186 the **Mixer** contract.

2187 In a similar way to Appendix C.3, queries to **Mixer** for specific nullifiers reveals
2188 to observers that the client was the sender of any previous or future transaction that
2189 generates such a nullifier. To mitigate this, **Mixer** MUST include nullifier values in the
2190 event data it emits, and clients SHOULD use this to track which of their notes are spent.
2191 This MUST happen as part of the regular sync operation, so that no extra RPC traffic is
2192 generated and observers cannot distinguish between clients that do and do not recognize
2193 any given nullifier. Note that this approach also supports tracking spent notes from
2194 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 $\text{Mix}_{in}.primIn.vout = 0$ and $\text{Mix}_{in}.primIn.vin \neq 0$ may indicate a simple “deposit” of funds into the mixer. Similarly, if both $\text{Mix}_{in}.primIn.vout$ and $\text{Mix}_{in}.primIn.vin$ are zero, the transaction must be spending only notes already within **Mixer**, into new notes. Finally, if $\text{Mix}_{in}.primIn.vin = 0$ and $\text{Mix}_{in}.primIn.vout \neq 0$, 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

2228 makes it much more difficult for an observer to attribute a specific purpose to the `Mix`
2229 call.

2230 Clients can also perform `Mix` calls in which $vin = vout = 0$ and 0-valued notes are
2231 created from other 0-valued notes. Such “dummy” self-payments can further obfuscate
2232 the activity of a wallet, by adding “noise” to the system. Note, however, that the gas
2233 cost for such transactions must still be paid.

2234 Wallet implementations **SHOULD** encourage the use of these complex calls where pos-
2235 sible, either via the user interface or by automatically adding complexity to transactions,
2236 and **SHOULD** support features such as adding “noise”¹ if the user wishes to pay for extra
2237 protection of this kind.

2238 C.7.1 Small anonymity sets

2239 Until there is a large number of commitments and users of the mixer, it may be easy
2240 for an observer to infer some of the private data that is intended to be hidden by mixer
2241 calls.

2242 In the simple case, if there are very few commitments in the `Mixer`’s Merkle tree, an
2243 attacker has a small list of candidate commitments that are being spent by subsequent
2244 `Mix` calls. Similarly, if the number of distinct Ethereum addresses that have been used
2245 to call `Mixer` is very small, observers can trace the original source of funds subsequently
2246 withdrawn to a small set of original depositors.

2247 Client software may wish to track metrics about the `Mixer` state, and either prevent
2248 certain actions or design the user interface to discourage users² from creating trans-
2249 actions whose features can be identified with high probability. We provide below a
2250 non-exhaustive list of metrics of interest:

- 2251 • **Number of commitments.** If there is a low absolute number of commitments,
2252 clearly any non-zero output must spend one of these (although we note that only
2253 $vout$ can be publicly known to be non-zero).
- 2254 • **Number of unspent commitments.** If $\#Comms - \#Nulls$ is small and a new
2255 commitment is created and then spent, observers can deduce that there is a high
2256 chance that the spend operation targeted the new commitment.
- 2257 • **Number of Ethereum addresses.** While very few distinct addresses (or groups
2258 of addresses that are not associated) have used the contract, observers can de-
2259 duce that subsequent `Mix` calls are likely to spend commitments created by clients
2260 associated with one of this small set of addresses.

2261 The set of Ethereum addresses that have interacted with the contract can leak data
2262 in other ways. An Ethereum address that withdraws value from the contract, but has not
2263 previously been used to make a `Mix` call (or a `Mix` call that deposits value into `Mixer`),

¹By randomly scheduling dummy payments, for instance

²By, for example, displaying warning messages and/or asking the user for confirmation

2264 must have been the recipient of zeth notes created by a previous depositor. The details
2265 may not be directly available to an observer, but this is another example of information
2266 which could be combined with other leaked data to infer connections between entities
2267 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 indistinguishability 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 $\mathbb{B}^{2 \cdot \text{ol}}$ into weak and strong sets, where w is the word length ($16 \cdot w = 2 \cdot \text{ol}$):

$$\mathcal{W} = \left\{ \text{aaaabbbbccccdddd} \in \mathbb{B}^{2 \cdot \text{ol}} \mid a, b, c, d \in \mathbb{B}^w \right\}$$

$$\mathcal{S} = \mathbb{B}^{2 \cdot \text{ol}} \setminus \mathcal{W}$$

Let $\mathcal{BLC}(2 \cdot \text{ol}, 2 \cdot \text{ol})$ denote the set of all block ciphers $E : \mathbb{B}^{2 \cdot \text{ol}} \times \mathbb{B}^{2 \cdot \text{ol}} \rightarrow \mathbb{B}^{2 \cdot \text{ol}}$. Define $\mathcal{BLC}^*(2 \cdot \text{ol}, 2 \cdot \text{ol})$ as the set of all block ciphers $E \in \mathcal{BLC}(2 \cdot \text{ol}, 2 \cdot \text{ol})$ with the additional restriction that $E(k_w, \cdot)$ is \mathcal{W} - and \mathcal{S} -subspace invariant for all keys $k_w \in \mathcal{K}_{\text{weak}}$. That is, inputs in \mathcal{W} map to \mathcal{W} , and likewise for \mathcal{S} . Here, $\mathcal{K}_{\text{weak}}$ is the set of weak keys, defined as

$$\mathcal{K}_{\text{weak}} = \left\{ k = \text{kkkkkkkkkkkkkkkk} \in \mathbb{B}^{2 \cdot \text{ol}} \mid k \in \mathbb{B}^w \right\}.$$

2291 A random $E \leftarrow \$\mathcal{BLC}^*(2 \cdot \text{ol}, 2 \cdot \text{ol})$ can now be modeled as follows:

- 2292 • on input of $(k, x) \in \mathcal{K}_{\text{weak}} \times \mathcal{W}$, E generates its response y randomly from \mathcal{W} up
2293 to repetition;
- 2294 • on input of $(k, x) \in \mathcal{K}_{\text{weak}} \times \mathcal{S}$, E generates its response y randomly from \mathcal{S} up to
2295 repetition.

2296 For key values $k \in \mathbb{B}^{2 \cdot \text{ol}} \setminus \mathcal{K}_{\text{weak}}$, E behaves like an ideal cipher: it either outputs a
2297 new random value or if the key-message-image tuple has already been queried the tuple's
2298 image. The case of inverse queries is analogous.

Blake2C is defined over the following domains and codomain:

$$\text{Blake2C} : \mathcal{BLC}^*(2 \cdot \text{ol}, 2 \cdot \text{ol}) \times \mathbb{B}^{\text{ol}} \times \mathbb{B}^{2 \cdot \text{ol}} \times \mathbb{B}^{\text{ol}/4} \times \mathbb{B}^{\text{ol}/4} \rightarrow \mathbb{B}^{\text{ol}}$$

2299 We write $\text{Blake2C}_E(h, m, t, f)$ for the output of the Blake2 compression function, defined
2300 over encryption scheme E on inputs h , m , t and f . The compression function, in par-
2301 ticular, computes the state $x = (h \parallel \text{pad}_{\text{ol}/2}(0) \parallel t \parallel f) \oplus (\text{pad}_{\text{ol}}(0) \parallel \text{IV})$ for some IV . It then
2302 encrypts x under m (where m is treated as a key for the encryption) and splits $E(m, x)$
2303 in two same size variables, the left part l_E and right part r_E . It finally outputs $l_E \oplus r_E \oplus h$.
2304

2305 Zeth uses the Blake2 compression function with a fixed encryption scheme E^* based on
2306 ChaCha stream cipher [Ber08a]. Thus, we write $\text{Blake2C}(h, m, t, f) = \text{Blake2C}_{E^*}(h, m, t, f)$.

2307 **Indifferentiability.** One way to measure the extent to which a certain cryptographic
2308 function behaves like a random function is via the indistinguishability framework where
2309 a distinguisher is given oracle access to either the cryptographic function or the random
2310 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:

$$\text{Indiff}_{\mathcal{C}^{\mathcal{P}}, \text{Sim}}(\text{Dist}) = \left| \Pr \left[\text{Dist}^{\mathcal{C}^{\mathcal{P}}, \mathcal{P}} = 1 \right] - \Pr \left[\text{Dist}^{\mathcal{R}, \text{Sim}^{\mathcal{R}}} = 1 \right] \right|$$

2311 The distinguisher Dist can query both its left oracle (either \mathcal{C} or \mathcal{R}) and its right
 2312 oracle (either \mathcal{P} or Sim). We refer to $\mathcal{C}^{\mathcal{P}}$, \mathcal{P} as the real world, and to \mathcal{R} , $\text{Sim}^{\mathcal{R}}$ as the
 2313 simulated world; the distinguisher Dist converses with either of these worlds and its goal
 2314 is to tell both worlds apart.

Theorem D.1.1 (Indifferentiability of Blake2 [LMN16]). *Let an encryption scheme $E \leftarrow \$\mathcal{BLC}^*(2 \cdot \text{ol}, 2 \cdot \text{ol})$ be a weakly ideal cipher, and consider the hash function Blake2_E that internally uses E . There exists a simulator Sim such that for any distinguisher Dist with total complexity q , we have:*

$$\text{Indiff}_{\text{Blake2}_E, \text{Sim}}(\text{Dist}) \leq \frac{\binom{q}{2}}{2^{2\text{ol}}} + \frac{2\binom{q}{2}}{2^{\text{ol}}} + \frac{q}{2^{\text{ol}/2}}$$

2315 where Sim makes at most $O(q^3)$ queries to a random function \mathcal{R} .

2316 *Proof.* See [LMN16, Corollary 1]. □

2317 For asymptotic security, we assume the distinguisher to be PPT and that the number
 2318 of queries made is polynomial $q \leq \text{poly}(\text{ol})$.

2319 **Additional remarks.** Luykx et al. [LMN16] remark that, by resorting to the WICM,
 2320 they do not make stronger assumptions than those used in previous results (ICM), and,
 2321 despite the fact that they give distinguishers more power (by weakening the cipher),
 2322 they are able to get similar results.

2323 D.2 Security proofs

2324 D.2.1 Blake2 is a PRF

2325 Luykx et al. already prove the PRFness of Blake2 *keyed* hash function in the multi-key
 2326 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 $\mathcal{R}_1, \dots, \mathcal{R}_\mu$ be random functions with the same domains and ranges as $\mathcal{C}_{k_1}, \dots, \mathcal{C}_{k_\mu}$. Let D be a distinguisher. The PRF distinguishing advantage of D is defined as,

$$\text{PRF}_{\mathcal{C}^{\mathcal{P}}}(\text{D}) = \left| \Pr[\text{Dist}_{\mathcal{C}_{k_1}^{\mathcal{P}}, \dots, \mathcal{C}_{k_\mu}^{\mathcal{P}}, \mathcal{P}} = 1] - \Pr[\text{Dist}_{\mathcal{R}_1, \dots, \mathcal{R}_\mu, \mathcal{P}} = 1] \right|$$

Blake2 supports keyed hashing by simply prepending the key to the message:

$$\text{Blake2}_{E,k}(m) = \text{Blake2}_E(k \| 0^{2\text{ol}-\text{kl}} \| m)$$

2327 where $\text{kl} \leq 2\text{ol}$ denotes the key size. In other words, the key gets processed as other data
 2328 and the HAIFA counter and flags are designated to the key in a similar fashion as if they
 2329 were for normal data blocks.

Theorem D.2.1 (PRF-security of Blake2 keyed mode [LMN16]). *Let $\mu \geq 1$ and let $k \leftarrow \$ (\mathbb{B}^{\text{kl}})^\mu$. Let an encryption scheme $E \leftarrow \$ \mathcal{BLC}^*(2 \cdot \text{ol}, 2 \cdot \text{ol})$ be a weakly ideal cipher, and consider the keyed hash function $\text{Blake2}_{E,k}$ that internally uses Blake2C_E that internally uses E . For any distinguisher Dist with total complexity q :*

$$\text{PRF}_{\text{Blake2}_{E,k}}(\text{Dist}) \leq \frac{\binom{q}{2}}{2^{2\text{ol}}} + \frac{2\binom{q}{2}}{2^{\text{ol}}} + \frac{q}{2^{\text{ol}/2}} + \frac{\mu q}{2^{\text{kl}}} + \frac{\binom{\mu}{2}}{2^{\text{kl}}}$$

2330 *Proof.* See [LMN16, Corollary 3]. □

2331 **Remark D.2.2.** We can note that in the case of keyed hashing, the key is padded only
 2332 to be processed in a single block to differentiate the key from the message. The security
 2333 proof of Theorem D.2.1 does not rely on this padding and as such also works with no
 2334 padding.

Theorem D.2.2 (PRF-security of Blake2 with a single key [LMN16]). *Let $k \leftarrow \$ \mathbb{B}^{\text{kl}}$. Let an encryption scheme $E \leftarrow \$ \mathcal{BLC}^*(2 \cdot \text{ol}, 2 \cdot \text{ol})$ be a weakly ideal cipher, and consider the keyed hash function $\text{Blake2}_E(k, \cdot) = \text{Blake2}_E(k \| \cdot)$ that internally uses Blake2C_E that internally uses E . For any distinguisher Dist with total complexity q :*

$$\text{PRF}_{\text{Blake2}_E}(\text{Dist}) \leq \frac{\binom{q}{2}}{2^{2\text{ol}}} + \frac{2\binom{q}{2}}{2^{\text{ol}}} + \frac{q}{2^{\text{ol}/2}} + \frac{q}{2^{\text{kl}}}$$

2335 *Proof.* See Remark D.2.2 and Theorem D.2.1 with $\mu = 1$. □

2336 **Remark D.2.3.** Since we analyse the security of Blake2 asymptotically, we assume that
 2337 for a security parameter λ holds $\text{ol} = \mathcal{O}(\lambda)$, $\text{kl} = \mathcal{O}(\lambda)$, and $q = \text{poly}(\lambda)$.

2338 D.2.2 Proof of Blake2 collision resistance

2339 We want to prove here the collision resistance of Blake2. To do so, we are going to
 2340 prove by contradiction that if Blake2 is not collision resistant, it is not indifferentiable
 2341 according to Definition D.1.1.

2342 **Theorem D.2.3.** *Blake2 is collision resistant.*

2343 *Informal proof.* Let us assume that there exists a PPT adversary \mathcal{B} which breaks the
 2344 collision resistance of Blake2. We build an adversary \mathcal{A} that uses this adversary to
 2345 differentiate between the real and simulated worlds. More particularly, \mathcal{A} gets left and
 2346 right oracles (see [LMN16, Figure 3]), which are either an oracle for a hash function and
 2347 for a weakly ideal block cipher or a random oracle and an encryption simulator with
 2348 oracle access to the random oracle.

2349 On each \mathcal{B} 's query m_i , $i \in \{1, \dots, q\}$, \mathcal{A} passes them to his left oracle and returns
 2350 the answer h_i to \mathcal{B} . Eventually, if \mathcal{B} finds a collision, that is a pair (m_i, m_j) such that
 2351 $m_i \neq m_j$ 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 $q \leq \text{poly}(\lambda)^1$.
 On the other hand, \mathcal{B} finds a collision with non-negligible probability if the oracles were real. Hence, \mathcal{A} wins the indistinguishability game with non-negligible advantage, which is a contradiction. \square

D.2.3 Blake2 as a commitment scheme

We prove here that Blake2 is a commitment scheme, i.e. is binding and hiding. To do so we rely on the previous results that Blake2 is collision resistant and a PRF.

Theorem D.2.4. *Let $E \leftarrow \$\mathcal{BLC}(2\text{ol}, 2\text{ol})$ and for a message $x \in \mathbb{B}^*$ and randomness $r \in \mathbb{B}^l$ commitment to x using r be $\text{ComSch.Com}(x; r) = \text{Blake2}_E(r \| x)$. 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 Blake2 with a non-negligible advantage η . We build an adversary \mathcal{A} that uses \mathcal{B} to break the PRF property of Blake2 with advantage $\eta/2$.

First, the PRF game is initiated, that is, the challenger chooses a random encryption scheme E and key $k \in \mathbb{B}^l$ and instantiates two oracles $O^{\text{Blake2}_k} = \text{Blake2}_E(k, \cdot)$ 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, \dots, m_q to \mathcal{A} (adaptively) who extends them with random r_1, \dots, 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 $\text{Com}(x; r) = \text{Com}(x'; r')$. This is implied by collision resistance of Blake2. Thus if \mathcal{B} is an algorithm that breaks the binding property with advantage η , there is another algorithm \mathcal{A} that breaks Blake2 collision resistance with the same advantage. \square

¹The probability would be $\frac{q^2}{2^{\text{ol}}}$ 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 $(i - 1) \cdot \frac{1}{2^{\text{ol}}}$ (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.

2385 Assuming that Blake2s is as secure as Blake2, a commitment scheme based on a
 2386 Blake2s, i.e. $\text{Com}(x; r) = \text{Blake2s}_E(r \| x)$ is hiding and binding.

2387 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^* ,

$$\begin{aligned} \text{ComSch}^*.\text{Setup} : \{1^\lambda \text{ s.t. } \lambda \in \mathbb{N}\} &\rightarrow \mathbb{B}^* \\ \text{ComSch}^*.\text{Com} : \mathcal{B}\mathcal{L}\mathcal{K}^*(2 \cdot \text{BLAKE2sCLEN}, 2 \cdot \text{BLAKE2sCLEN}) &\times \mathbb{B}^{2 \cdot \text{BLAKE2sCLEN}} \\ &\times (\mathbb{B}^{\text{PRFADDRROUTLEN}} \times \mathbb{B}^{\text{PRFRHOOUTLEN}} \times \mathbb{B}^{\text{ZVALUELEN}}) \times \mathbb{B}^{\text{RTRAPLEN}} \rightarrow \mathbb{B}^{\text{BLAKE2sCLEN}} \end{aligned}$$

The commitment scheme is defined as follows,

$$\begin{aligned} \text{ComSch}^*.\text{Setup}(1^\lambda) &= pp^* = \epsilon \\ \text{ComSch}^*.\text{Com}(m = (apk, \rho, v); r) &= cm \\ &= \text{Blake2E}^*(r \| apk \| \rho \| v) \end{aligned}$$

Given a commitment scheme ComSch^* , the bijective function $\text{decode}_{\mathbb{N}}(\cdot)$ and $p_\lambda \in \mathbb{N}$, a prime which can be represented using λ bits, we define the commitment scheme ComSch' as follows:

$$\begin{aligned} \text{ComSch}'.\text{Setup}(1^\lambda) &= (\text{ComSch}^*.\text{Setup}(1^\lambda), p_\lambda) \\ \text{ComSch}'.\text{Com}(m; r) &= \text{decode}_{\mathbb{N}}(\text{ComSch}^*.\text{Com}(m; r)) \pmod{p_\lambda} \text{ for } m = (apk \| \rho \| v) \end{aligned}$$

2388 Note that ComSch (see Section 3.1.2) is a particular instantiation of ComSch' where E^*
 2389 is set as ChaCha encryption scheme [Ber08a], k^* is a random key, and p_λ is r .

2390 **Theorem D.2.5** (Hiding). *If ComSch^* is hiding then ComSch' is hiding.*

2391 *Proof.* We prove the theorem by contradiction i.e. we assume that there exists an adver-
 2392 sary \mathcal{B} that breaks ComSch' 's hiding property and construct an adversary \mathcal{A} that uses
 2393 \mathcal{B} to break ComSch^* 's hiding property with non-negligible probability.

2394 Let \mathcal{C} be a challenger that sets up the hiding game for ComSch^* and \mathcal{A} . The adversary
 2395 \mathcal{A} , given public parameters pp^* of ComSch^* and access to an oracle that runs the Com
 2396 algorithm of ComSch^* scheme, simulates a hiding game for ComSch' for \mathcal{B} . The adversary
 2397 \mathcal{A} starts by setting public parameters pp' for ComSch' using public parameters pp^*
 2398 given by \mathcal{C} . Parameters pp' are passed to \mathcal{B} who outputs a pair of messages m_0, m_1 .
 2399 The adversary \mathcal{A} forwards them to the challenger who samples a bit b at random and
 2400 generates $cm^* = \text{ComSch}^*.\text{Com}(m_b; r)$ for some randomness r . The result is returned
 2401 to \mathcal{A} (see Definition 1.5.21). Then \mathcal{A} passes $cm = \text{decode}_{\mathbb{N}}(cm^*) \pmod{p_\lambda}$ to \mathcal{B} who
 2402 returns his guess b' . The adversary \mathcal{A} returns the same b' to the challenger.

2403 By construction, it is clear that \mathcal{A} wins the hiding game with the same probability
 2404 that \mathcal{B} wins the simulated hiding game. Since \mathcal{B} 's advantage is non-negligible, this means
 2405 that \mathcal{A} wins the ComSch^* hiding game with non-negligible probability as well. \square

2406 **Theorem D.2.6** (Binding). *Let ComSch^* be a computationally binding commitment*
 2407 *scheme and $\text{ComSch}^*.\text{Com}$ indifferentiable from a random oracle. Then ComSch' is also*
 2408 *computationally binding if $l = \lceil 2^\lambda/p_\lambda \rceil$ is at most $\text{poly}(\lambda)$.*

2409 *Proof.* Assume that \mathcal{A} asks the ComSch' commit and open oracles a total of q_λ distinct
 2410 queries. Let us denote the result of the q_λ queries and output of the attacker (the
 2411 candidate collision) as $((m_1, r_1, y_1), \dots, (m_{q_\lambda}, r_{q_\lambda}, y_{q_\lambda}), \text{out})$. If \mathcal{A} is successful it means
 2412 that it outputs (m, r) , (m', r') such that $(m, r) \neq (m', r')$ and $\text{ComSch}'.\text{Com}(m; r) =$
 2413 $\text{ComSch}'.\text{Com}(m'; r')$.

By the definition of ComSch' , we have that,

$$\text{ComSch}'.\text{Com}(m; r) = \text{decode}_{\mathbb{N}}(\text{ComSch}^*.\text{Com}(m; r)) \pmod{p_\lambda}$$

Hence, we have a collision in ComSch' if there exists $k \in [l]$, l being the ratio of the
 codomains of $\text{ComSch}^*.\text{Com}$ and $\text{ComSch}'.\text{Com}$, such that,

$$|\text{decode}_{\mathbb{N}}(\text{ComSch}^*.\text{Com}(m; r)) - \text{decode}_{\mathbb{N}}(\text{ComSch}^*.\text{Com}(m'; r'))| = k \cdot p_\lambda.$$

2414 We show that this event is unlikely.

2415 In fact, for each $i \in [q_\lambda]$, let C_i be the event that the adversary wins at the i -th
 2416 query. That is, the last commitment y_i is a ComSch' collision with one of the previous
 2417 y_j . More precisely there exists $j \leq i$ and $k < l$ such that $y_i = y_j + k \cdot p_\lambda$.

2418 Since ComSch^* is a random oracle, y_i is randomly selected from a set of at least p_λ
 2419 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 $\Pr[C_1 \vee \dots \vee C_{q_\lambda}] \leq$
 $\sum_{i=1}^{q_\lambda} \Pr[C_i] = l/p_\lambda \cdot \sum_{i=1}^{q_\lambda} i$. This probability is bounded by $l \cdot \frac{q_\lambda(q_\lambda+1)}{2p_\lambda}$. However,
 we allow only polynomial number of queries. Thus for $q_\lambda = \text{poly}(\lambda)$ this probability
 becomes,

$$\frac{2^\lambda \cdot \text{poly}(\lambda)}{p_\lambda^2},$$

2420 what is negligible for $2^\lambda/p_\lambda \leq \text{poly}(\lambda)$. □

2421 **Remark D.2.4.** Note that in **Zeth**'s commitment scheme, we set $p_\lambda = \mathbf{r}$ and $2^\lambda =$
 2422 $2^{\text{BLAKE2sCLEN}}$. Thus, for **BN-254** and **BLS12-377** have $l = 6$ and $l = 14$, respectively.
 2423 Therefore, the probability of an attacker breaking the binding property due to reduction
 2424 modulo \mathbf{r} increases approximately by these factors. This is still negligible.

2425 **Corollary.** *Assume that Blake2 is indifferentiable from a random oracle and a PRF,*
 2426 *then ComSch^* is computationally binding and computationally hiding. Furthermore, the*
 2427 *reduction is tight. That is, the advantage of any PPT adversary against the binding*
 2428 *(resp. hiding) property is the same as the advantage of an adversary against collision*
 2429 *resistance and binding (resp. hiding).*

2430

Bibliography

- 2432 [AAM12] Imad Fakhri Alshaikhli, Mohammad A Alahmad, and Khanssaa Munthir.
2433 Comparison and analysis study of sha-3 finalists. In *2012 International*
2434 *Conference on Advanced Computer Science Applications and Technologies*
2435 *(ACSAT)*, pages 366–371. IEEE, 2012.
- 2436 [abi] Contract abi specification, section "function selector". [https://solidity.readthedocs.io/en/develop/abi-spec.html#](https://solidity.readthedocs.io/en/develop/abi-spec.html#function-selector)
2437 [function-selector](https://solidity.readthedocs.io/en/develop/abi-spec.html#function-selector).
2438
- 2439 [ABM⁺03] Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, and Scott
2440 Vanstone. Validation of elliptic curve public keys. In *International Work-*
2441 *shop on Public Key Cryptography*, pages 211–223. Springer, 2003.
- 2442 [ABN10] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption.
2443 In *Theory of Cryptography Conference*, pages 480–497. Springer, 2010.
2444 <https://eprint.iacr.org/2008/440.pdf>.
- 2445 [ABR99] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. Dhaes:
2446 An encryption scheme based on the diffie-hellman prob-
2447 lem. 1999. [https://pdfs.semanticscholar.org/95f4/](https://pdfs.semanticscholar.org/95f4/63d097086fba325086a4cf88706648dafd09.pdf)
2448 [63d097086fba325086a4cf88706648dafd09.pdf](https://pdfs.semanticscholar.org/95f4/63d097086fba325086a4cf88706648dafd09.pdf).
- 2449 [ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. Dhies: An en-
2450 cryptation scheme based on the diffie-hellman problem., 2001. [https://](https://web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf)
2451 web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf.
- 2452 [ACG⁺19] Martin R Albrecht, Carlos Cid, Lorenzo Grassi, Dmitry Khovratovich,
2453 Reinhard Lüftenegger, Christian Rechberger, and Markus Schofnegger.
2454 Algebraic cryptanalysis of stark-friendly designs: application to marvel-
2455 lous and mimc. In *International Conference on the Theory and Appli-*
2456 *cation of Cryptology and Information Security*, pages 371–397. Springer,
2457 2019.
- 2458 [AFK⁺08] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier,
2459 and Christian Rechberger. New features of latin dances: analysis of salsa,
2460 chacha, and rumba. In *International Workshop on Fast Software Encryp-*
2461 *tion*, pages 470–488. Springer, 2008.

- 2462 [AG18] Andreas M. Antonopoulos and Wood Gavin. *Mastering Ethereum*.
2463 O'Reilly Media, 2018.
- 2464 [AGM⁺09] Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, Yu Sasaki, and Lei
2465 Wang. Preimages for step-reduced sha-2. In *International Conference*
2466 *on the Theory and Application of Cryptology and Information Security*,
2467 pages 578–597. Springer, 2009. [https://link.springer.com/content/
2468 pdf/10.1007/978-3-642-10366-7_34.pdf](https://link.springer.com/content/pdf/10.1007/978-3-642-10366-7_34.pdf).
- 2469 [AGR⁺16] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and
2470 Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing
2471 with minimal multiplicative complexity. In *International Conference on*
2472 *the Theory and Application of Cryptology and Information Security*, pages
2473 191–219. Springer, 2016.
- 2474 [AHMP08] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W
2475 Phan. Sha-3 proposal blake. *Submission to NIST*, 229:230, 2008.
- 2476 [ALM12] Elena Andreeva, Atul Luykx, and Bart Mennink. Provable security of
2477 blake with non-ideal compression function. In *International Conference*
2478 *on Selected Areas in Cryptography*, pages 321–338. Springer, 2012.
- 2479 [AMP10] Elena Andreeva, Bart Mennink, and Bart Preneel. Security reductions
2480 of the second round sha-3 candidates. In *International Conference on*
2481 *Information Security*, pages 39–53. Springer, 2010.
- 2482 [AMPŠ12] Elena Andreeva, Bart Mennink, Bart Preneel, and Marjan Škrobot. Se-
2483 curity analysis and comparison of the sha-3 finalists blake, grøstl, jh,
2484 keccak, and skein. In *International Conference on Cryptology in Africa*,
2485 pages 287–305. Springer, 2012.
- 2486 [ANWOW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and
2487 Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *Interna-*
2488 *tional Conference on Applied Cryptography and Network Security*, pages
2489 119–135. Springer, 2013. <https://eprint.iacr.org/2013/322.pdf>.
- 2490 [AS09] Kazumaro Aoki and Yu Sasaki. Meet-in-the-middle preimage attacks
2491 against reduced sha-0 and sha-1. In *Annual International Cryptology Con-*
2492 *ference*, pages 70–89. Springer, 2009.
- 2493 [BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval.
2494 Key-privacy in public-key encryption. In *International Conference on the*
2495 *Theory and Application of Cryptology and Information Security*, pages
2496 566–582. Springer, 2001. [https://iacr.org/archive/asiacrypt2001/
2497 22480568.pdf](https://iacr.org/archive/asiacrypt2001/22480568.pdf).

- 2498 [BCC⁺15] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, Jens
2499 Groth, and Christophe Petit. Short accountable ring signatures based on
2500 ddh. In *European Symposium on Research in Computer Security*, pages
2501 243–265. Springer, 2015.
- 2502 [BCD⁺20] Tim Beyne, Anne Canteaut, Itai Dinur, Maria Eichlseder, Gregor Le-
2503 ander, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Yu Sasaki,
2504 Yosuke Todo, and Friedrich Wiemer. Out of oddity – new cryptana-
2505 lytic techniques against symmetric primitives optimized for integrity proof
2506 systems. Cryptology ePrint Archive, Report 2020/188, 2020. <https://eprint.iacr.org/2020/188>.
2507
- 2508 [BCG⁺20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush
2509 Mishra, and Howard Wu. ZEXE: enabling decentralized private com-
2510 putation. In *2020 IEEE Symposium on Security and Privacy, SP 2020,*
2511 *San Francisco, CA, USA, May 18-21, 2020*, pages 947–964. IEEE, 2020.
- 2512 [BCK⁺18] Elaine Barker, Lily Chen, Sharon Keller, Allen Roginsky, Apostol
2513 Vassilev, and Richard Davis. Recommendation for pair-wise key-
2514 establishment schemes using discrete logarithm cryptography. Technical
2515 report, National Institute of Standards and Technology, 2018. [Online;
2516 last accessed 10-January-2020].
- 2517 [BD07] Eli Biham and Orr Dunkelman. A framework for iterative hash
2518 functions—haifa. Technical report, Computer Science Department, Tech-
2519 nion, 2007. <https://eprint.iacr.org/2007/278.pdf>.
- 2520 [BDPVA07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche.
2521 Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer,
2522 2007.
- 2523 [Ber05] Daniel J Bernstein. The poly1305-aes message-authentication code.
2524 In *International Workshop on Fast Software Encryption*, pages 32–49.
2525 Springer, 2005. <https://cr.yp.to/mac/poly1305-20050329.pdf>.
- 2526 [Ber06] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In
2527 *International Workshop on Public Key Cryptography*, pages 207–228.
2528 Springer, 2006. <https://cr.yp.to/ecdh/curve25519-20060209.pdf>.
- 2529 [Ber08a] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record*
2530 *of SASC*, volume 8, pages 3–5, 2008. [https://cr.yp.to/chacha/](https://cr.yp.to/chacha/chacha-20080120.pdf)
2531 [chacha-20080120.pdf](https://cr.yp.to/chacha/chacha-20080120.pdf).
- 2532 [Ber08b] Daniel J. Bernstein. New stream cipher designs. chapter The Salsa20 Fam-
2533 ily of Stream Ciphers, pages 84–97. Springer-Verlag, Berlin, Heidelberg,
2534 2008.

- 2535 [BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party compu-
2536 tation for zk-snark parameters in the random beacon model. *Cryptology*
2537 *ePrint Archive*, Report 2017/1050, 2017. [https://eprint.iacr.org/](https://eprint.iacr.org/2017/1050)
2538 2017/1050.
- 2539 [BL] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves
2540 for elliptic-curve cryptography. <https://safecurves.cr.yp.to>. [Online;
2541 last accessed 09-December-2019].
- 2542 [Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable
2543 errors. *Commun. ACM*, 13(7):422–426, 1970.
- 2544 [Bon19] Xavier Bonnetain. Collisions on feistel-mimc and univariate gmimc. 2019.
- 2545 [Bou03] Nicolas Bourbaki. *Elements of mathematics: Algebra*. Springer, 2003.
- 2546 [Bra97] S. Bradner. Key words for use in rfcs to indicate requirement levels. RFC
2547 2119, RFC Editor, March 1997.
- 2548 [BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box
2549 analysis of the block-cipher-based hash-function constructions from
2550 pgv. In *Annual International Cryptology Conference*, pages 320–335.
2551 Springer, 2002. [https://link.springer.com/content/pdf/10.1007/](https://link.springer.com/content/pdf/10.1007/3-540-45708-9_21.pdf)
2552 3-540-45708-9_21.pdf.
- 2553 [BS07] Mihir Bellare and Sarah Shoup. Two-tier signatures, strongly unforge-
2554 able signatures, and fiat-shamir without random oracles. In *International*
2555 *Workshop on Public Key Cryptography*, pages 201–216. Springer, 2007.
- 2556 [BSCG⁺14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green,
2557 Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized
2558 anonymous payments from bitcoin. In *2014 IEEE Symposium on Security*
2559 *and Privacy*, pages 459–474. IEEE, 2014.
- 2560 [CHM⁺20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah
2561 Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with
2562 universal and updatable SRS. pages 738–768, 2020.
- 2563 [Cle19] Clearmatics. Zeth Sproken release. [https://github.com/clearmatics/](https://github.com/clearmatics/zeth/releases/tag/v0.2)
2564 [zeth/releases/tag/v0.2](https://github.com/clearmatics/zeth/releases/tag/v0.2), 2019. [Online; released 04-April-2019].
- 2565 [CM16] Arka Rai Choudhuri and Subhamoy Maitra. Differential cryptanalysis of
2566 salsa and chacha-an evaluation with a hybrid model. *IACR Cryptology*
2567 *ePrint Archive*, 2016:377, 2016. [https://eprint.iacr.org/2016/377.](https://eprint.iacr.org/2016/377.pdf)
2568 pdf.
- 2569 [CM17] Arka Rai Choudhuri and Subhamoy Maitra. Significantly improved multi-
2570 bit differentials for reduced round salsa and chacha. *IACR Transactions*
2571 *on Symmetric Cryptology*, 2016(2):261–287, Feb. 2017.

2572 [DG09] George Danezis and Ian Goldberg. Sphinx: A compact and provably
2573 secure mix format. In *30th IEEE Symposium on Security and Privacy*
2574 *(S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 269–282.
2575 IEEE Computer Society, 2009.

2576 [EFK15] Thomas Espitau, Pierre-Alain Fouque, and Pierre Karpman. Higher-order
2577 differential meet-in-the-middle preimage attacks on sha-1 and blake. In
2578 *Annual Cryptology Conference*, pages 683–701. Springer, 2015. <https://eprint.iacr.org/2015/515>.
2579

2580 [EGL⁺20] Maria Eichlseder, Lorenzo Grassi, Reinhard Lüftenegger, Morten Øygarden,
2581 Christian Rechberger, Markus Schofnegger, and Qingju Wang. An
2582 algebraic attack on ciphers with low-degree round functions: Application
2583 to full mimc. Cryptology ePrint Archive, Report 2020/182, 2020.
2584 <https://eprint.iacr.org/2020/182>.

2585 [est] Estream project. <https://en.wikipedia.org/wiki/ESTREAM>.

2586 [Gab19] Ariel Gabizon. AuroraLight: Improved prover efficiency and SRS size in
2587 a sonic-like system. Cryptology ePrint Archive, Report 2019/601, 2019.
2588 <https://eprint.iacr.org/2019/601>.

2589 [GFBR06] Decio Gazzoni Filho, Paulo SLM Barreto, and Vincent Rijmen. The
2590 maelstrom-0 hash function. In *Brazilian Symposium on Information and*
2591 *Computer System Security*. , 2006.

2592 [GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova.
2593 Quadratic span programs and succinct nizks without peps. In Thomas
2594 Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EU-*
2595 *ROCRYPT 2013, 32nd Annual International Conference on the Theory*
2596 *and Applications of Cryptographic Techniques, Athens, Greece, May 26-*
2597 *30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*,
2598 pages 626–645. Springer, 2013.

2599 [GJMG11] B Guido, D Joan, P Michaël, and VA Gilles. The keccak sha-3 submission.
2600 2011.

2601 [GKN⁺14] Jian Guo, Pierre Karpman, Ivica Nikolić, Lei Wang, and Shuang Wu.
2602 Analysis of blake2. In *Cryptographers’ Track at the RSA Conference*,
2603 pages 402–423. Springer, 2014. [https://eprint.iacr.org/2013/467](https://eprint.iacr.org/2013/467.pdf).
2604 [pdf](https://eprint.iacr.org/2013/467.pdf).

2605 [GLRW10] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang. Ad-
2606 vanced meet-in-the-middle preimage attacks: First results on full tiger,
2607 and improved results on md4 and sha-2. In *International Conference on*
2608 *the Theory and Application of Cryptology and Information Security*, pages
2609 56–75. Springer, 2010. <https://eprint.iacr.org/2010/016.pdf>.

- 2610 [GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures
2611 of knowledge from simulation-extractable snarks. In Jonathan Katz and
2612 Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th*
2613 *Annual International Cryptology Conference, Santa Barbara, CA, USA,*
2614 *August 20-24, 2017, Proceedings, Part II*, volume 10402 of *Lecture Notes*
2615 *in Computer Science*, pages 581–612. Springer, 2017.
- 2616 [Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1.
2617 Cambridge University Press, Cambridge, UK, 2001.
- 2618 [Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and
2619 constant size group signatures. pages 444–459, 2006.
- 2620 [Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In
2621 *Annual International Conference on the Theory and Applications of Crypt-*
2622 *ographic Techniques*, pages 305–326. Springer, 2016.
- 2623 [GRR⁺16] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and
2624 Nigel P Smart. Mpc-friendly symmetric key primitives. In *Proceedings of*
2625 *the 2016 ACM SIGSAC Conference on Computer and Communications*
2626 *Security*, pages 430–443. ACM, 2016. [https://eprint.iacr.org/2016/](https://eprint.iacr.org/2016/542)
2627 [542](https://eprint.iacr.org/2016/542).
- 2628 [GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK:
2629 Permutations over lagrange-bases for oecumenical noninteractive argu-
2630 ments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.
2631 <https://eprint.iacr.org/2019/953>.
- 2632 [Hao14] Yonglin Hao. The boomerang attacks on blake and blake2. In *Interna-*
2633 *tional Conference on Information Security and Cryptology*, pages 286–310.
2634 Springer, 2014. <https://eprint.iacr.org/2014/1012.pdf>.
- 2635 [Har19] HarryR. Conversation about Miyaguchi-Preneel security. [https://](https://github.com/HarryR/ethsnarks/issues/119)
2636 github.com/HarryR/ethsnarks/issues/119, 2019. Online; accessed
2637 June-2019.
- 2638 [HG20] Youssef El Housni and Aurore Guillevic. Optimized and secure pairing-
2639 friendly elliptic curves suitable for one layer proof composition. Cryptol-
2640 ogy ePrint Archive, Report 2020/351, 2020. [https://eprint.iacr.org/](https://eprint.iacr.org/2020/351)
2641 [2020/351](https://eprint.iacr.org/2020/351).
- 2642 [HKT11] Thomas Holenstein, Robin Künzler, and Stefano Tessaro. The equiva-
2643 lence of the random oracle model and the ideal cipher model, revisited.
2644 In *Proceedings of the forty-third annual ACM symposium on Theory of*
2645 *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.

2684 [LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. RFC 7748, <https://tools.ietf.org/pdf/rfc7748.pdf>, 2016.

2685

2686 [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.

2687

2688

2689

2690

2691 [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>.

2692

2693

2694 [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>.

2695

2696

2697

2698 [LN18] Adam Langley and Yoav Nir. Chacha20 and poly1305 for ietf protocols. *RFC 8439*, 2018. <https://tools.ietf.org/html/rfc8439>.

2699

2700 [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.

2701

2702

2703 [Mai16] Subhamoy Maitra. Chosen iv cryptanalysis on reduced round chacha and salsa. *Discrete Applied Mathematics*, 208:88–97, 2016.

2704

2705 [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.

2706

2707

2708 [MJS15] Ed. M-J. Saarinen. Blake Compression Function F. <https://tools.ietf.org/html/rfc7693#section-3.2>, 2015. [Online; accessed November-2019].

2709

2710

2711 [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>.

2712

2713

2714 [MNS11] Florian Mendel, Tomislav Nad, and Martin Schl  ffer. 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.

2715

2716

2717

2718

- 2719 [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.
- 2720
- 2721
- 2722
- 2723
- 2724 [Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48(177):243–264, 1987.
- 2725
- 2726 [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.
- 2727
- 2728
- 2729 <https://eprint.iacr.org/2015/909.pdf>.
- 2730 [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.
- 2731
- 2732
- 2733 [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.
- 2734
- 2735
- 2736 [NA19] Samuel Neves and Filipe Araujo. An observation on norx, blake2, and chacha. *Information Processing Letters*, 149:1–5, 2019.
- 2737
- 2738 [oST15] National Institute of Standards and Technology. Secure Hash Standard (SHS). <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>, 2015.
- 2739
- 2740
- 2741 [Per17] Trevor Perrin. X25519 and zero outputs. <https://moderncrypto.org/mail-archive/curves/2017/000896.html>, 2017. [Online; last accessed 08-January-2020].
- 2742
- 2743
- 2744 [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.
- 2745
- 2746
- 2747
- 2748
- 2749 [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.
- 2750
- 2751
- 2752 [Pro14] Gordon Procter. A security analysis of the composition of chacha20 and poly1305. *IACR Cryptology ePrint Archive*, 2014:613, 2014.
- 2753

2754 [PV05] Pascal Paillier and Damien Vergnaud. Discrete-log-based signatures may
2755 not be equivalent to discrete log. In *International Conference on the*
2756 *Theory and Application of Cryptology and Information Security*, pages
2757 1–20. Springer, 2005.

2758 [Qu99] Minghua Qu. Sec 2: Recommended elliptic curve domain parameters. *Cer-*
2759 *ticom Res., Mississauga, ON, Canada, Tech. Rep. SEC2-Ver-0.6*, 1999.

2760 [Rk19] Antoine Rondelet and @karalabe. Go-ethereum BN256 package.
2761 [https://github.com/ethereum/go-ethereum/blob/master/crypto/](https://github.com/ethereum/go-ethereum/blob/master/crypto/bn256/cloudflare/constants.go)
2762 [bn256/cloudflare/constants.go](https://github.com/ethereum/go-ethereum/blob/master/crypto/bn256/cloudflare/constants.go), 2019. [Online; released 28-May-2019].

2763 [Ron20] Antoine Rondelet. Zecale: Reconciling privacy and scalability on
2764 ethereum, 2020.

2765 [RZ19] Antoine Rondelet and Michal Zajac. ZETH: On Integrating Zerocash on
2766 Ethereum. [Online; released April-2019], 2019.

2767 [SS08] Somitra Kumar Sanadhya and Palash Sarkar. New collision attacks
2768 against up to 24-step sha-2. In *International conference on cryptology in*
2769 *India*, pages 91–103. Springer, 2008. [https://eprint.iacr.org/2008/](https://eprint.iacr.org/2008/270.pdf)
2770 [270.pdf](https://eprint.iacr.org/2008/270.pdf).

2771 [Ste15] Stevens, Marc. On collisions for md5. [https://www.win.tue.](https://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf)
2772 [nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.](https://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf)
2773 [%20Stevens.pdf](https://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf), 2015.

2774 [SZFW12] Zhenqing Shi, Bin Zhang, Dengguo Feng, and Wenling Wu. Improved
2775 key recovery attacks on reduced-round salsa20 and chacha. In *Interna-*
2776 *tional Conference on Information Security and Cryptology*, pages 337–351.
2777 Springer, 2012.

2778 [TBP20] Florian Tramèr, Dan Boneh, and Kenneth G. Paterson. Remote side-
2779 channel attacks on anonymous transactions. Cryptology ePrint Archive,
2780 Report 2020/220, 2020. <https://eprint.iacr.org/2020/220>.

2781 [THH15] Piotr Dyraga Tjaden Hess, Matt Luongo and James Hancock. EIP 152:
2782 Add BLAKE2 compression function ‘F’. [https://eips.ethereum.org/](https://eips.ethereum.org/EIPS/eip-152)
2783 [EIPS/eip-152](https://eips.ethereum.org/EIPS/eip-152), 2015. [Online; accessed November-2019].

2784 [VNP10] Janoš Vidali, Peter Nose, and Enes Pašalić. Collisions for variants of the
2785 blake hash function. *Information processing letters*, 110(14-15):585–590,
2786 2010.

2787 [W⁺] Gavin Wood et al. Ethereum: A secure decentralised generalised trans-
2788 action ledger.

2789 [wc] Ethereum wiki contributors. Patricia tree. [https://github.com/](https://github.com/ethereum/wiki/wiki/Patricia-Tree)
2790 [ethereum/wiki/wiki/Patricia-Tree](https://github.com/ethereum/wiki/wiki/Patricia-Tree).

2791 [wc19] Ethereum wiki contributors. RLP. [https://github.com/ethereum/](https://github.com/ethereum/wiki/wiki/RLP)
2792 [wiki/wiki/RLP](https://github.com/ethereum/wiki/wiki/RLP), 2019. [Online; accessed December-2019].

2793 [Woo19] Dr Gavin Wood. ETHEREUM: A Secure Decentralised Gener-
2794 alised Transaction Ledger Byzantium. [https://ethereum.github.io/](https://ethereum.github.io/yellowpaper/paper.pdf)
2795 [yellowpaper/paper.pdf](https://ethereum.github.io/yellowpaper/paper.pdf), 2019. [VERSION 7e819ec - 2019-10-20].

2796 [zcaa] On the security of sprout/sapling in-band en-
2797 crypton. [https://forum.zcashcommunity.com/t/](https://forum.zcashcommunity.com/t/on-the-security-of-sprout-sapling-in-band-encryption/34986)
2798 [on-the-security-of-sprout-sapling-in-band-encryption/34986](https://forum.zcashcommunity.com/t/on-the-security-of-sprout-sapling-in-band-encryption/34986).

2799 [zcab] "zcash alerts - security announcement 2019-09-24".
2800 [https://z.cash/support/security/announcements/](https://z.cash/support/security/announcements/security-announcement-2019-09-24/a)
2801 [security-announcement-2019-09-24/a](https://z.cash/support/security/announcements/security-announcement-2019-09-24/a).

2802 [ZCa19] ZCash. ZCash protocol specification. [https://github.com/zcash/zip](https://github.com/zcash/zip/blob/master/protocol/protocol.pdf)/
2803 [blob/master/protocol/protocol.pdf](https://github.com/zcash/zip/blob/master/protocol/protocol.pdf), 2019. [Online; initially released
2804 14-December-2015].