A Spec for Σ-Protocols

Authors: Michele Orrù, Stephan Krenn

Contributors:

Jan Bobolz
Mary Maller
Ivan Visconti
Yuwen Zhang

Version: v 0.2

Date: November 18, 2022

Contents

1 Introduction

Zero-knowledge [GMR89] proofs of knowledge [BG93] allow a prover to convince a verifier that he knows a secret piece of information, without revealing anything except what the claim itself already reveals. Many practically relevant proof goals can be realized using so-called Σ-protocols. Introduced by Schnorr [Sch91] over 30 years ago, they play an essential component in the building of a number of cryptographic constructions, such as anonymous credentials [CMZ14], password-authenticated key exchange [HR11], signatures [Sch90], ring signatures [MP15], blind signatures [PS97], multi signatures [NRSW20], threshold signatures [KG20] and more. This spec provides guidelines for correctly implementing Σ-protocols.

1.1 Notation

For the purpose of this document, the following notation will be used:

λ	main security parameter
$\mathcal{C}$	challenge set. A challenge is a vector of 32 bytes (octets).
Y	the statement, i.e., the public information in a zero-knowledge proof
w	the witness, i.e., the secret information in a zero-knowledge proof
T	the first message, called commitment
c	the second message, called challenge
s	the third message, called response
$x \coloneqq 1$	assignment of the value 1 to x
$x \xleftarrow{\$} \mathcal{S}$	assignment of a uniformly random element in $\mathcal{S}$ to x
$x \leftarrow \texttt{procedure}(in)$	assign to x the output of the randomized procedure on input in
R	binary relation
$\lvert y \rvert$	bit-length of a string
$\oplus$	binary operator denoting the bitwise XOR of two strings of equal length

Additional notation will be introduced when describing specific algebraic objects.

1.2 Terminology

Given sets $\mathcal{Y}$ and $\mathcal{W}$, a *binary relation* $\mathsf{R} \subseteq \mathcal{Y} \times \mathcal{W}$ associates elements from $\mathcal{Y}$ with elements from $\mathcal{W}$. For $(Y, w) \in \mathsf{R}$, we refer to Y as a *statement*, and to w as a *witness* (for Y). The statement is public information shared between prover and verifier. The witness is secret information solely known by the prover. Note there may be multiple witnesses for a given Y.

Example 1: Discrete logarithm equality

Let $\mathbb{G}$ be a cyclic group of prime order p, and let G and H be generators of $\mathbb{G}$. Then the following relation R_{dleq} contains as statements all pairs of elements having the same discrete logarithm with respect to G and H, with the corresponding witness

being their discrete logarithm:

$$\mathsf{R}_{\text{dleq}} = \left\{((Y_1, Y_2), w) \in \mathbb{G}^2 \times \mathbb{F}_p \ : \ Y_1 = wG \ \wedge \ Y_2 = wH\right\}.$$

Example 2: Representation

Let $\mathbb{G}$ be a cyclic group of prime order p, and let G and H be generators of $\mathbb{G}$. Then the following relation R_{rep} contains as statements all valid Perdersen commitments, with the corresponding witnesses being their openings:

$$\mathsf{R}_{\text{rep}} = \left\{(Y, (w_1, w_2)) \in \mathbb{G} \times \mathbb{F}_p^2 \ : \ Y = w_1 G + w_2 H\right\}.$$

Note that in this case, each statement has p valid witnesses.

In a zero-knowledge proof, the *witness* is secret information, while the *statement* is public. A *proof* is a sequence of bytes[1] attesting that a witness is in some relation with the statement.

Proofs described in this spec are zero-knowledge and sound. Zero knowledge means that the protocol does not leak any information about the prover's witness beyond what the attacker may infer from the proven statement or from other sources [ZKP19, 1.6.4]. Soundness means that it is not possible to make the verifier accept for statements for which no valid witness exists [ZKP19, 1.6.2].

In particular we will focus on non-malleable extractable proofs, that is, proofs where the witness does not only exist but is also precisely known by the prover. These protocols are also known as proofs of knowledge [GMR85, FFS87, BG93] and are said to satisfy *knowledge soundness* [Dam04]. Non-malleability means that, in addition, the proof is secure against man-in-the middle attacks, and attackers cannot produce a new valid proof by tampering another valid proof.

1.3 Guarantees

Michele's note: Question for the community: Should we require implementations to be constant-time? It is not immediate to build OR composition in constant-time. It is possible to double the cost and run prover and simulator for each branch, but at the end we still need the ternary operator to select the correct transcript. This is the only place where we are incurring in non non-constant-time operations (assuming the algebraic operations are already solved). All other algorithms are constant-time.

Michele's note: Question for the community: Who is the recipient of this standard? Are they meant to know already what extraction, or simulation means? Should we employ terms like *simulation-extractability* or *non-malleability* (the latter being different from the former and more imprecise, yet also more self-explainatory). Side-note: OR-composition is not simulation-extractable.

[1] Throughout this work, a byte is defined as a vector of 8 bits.

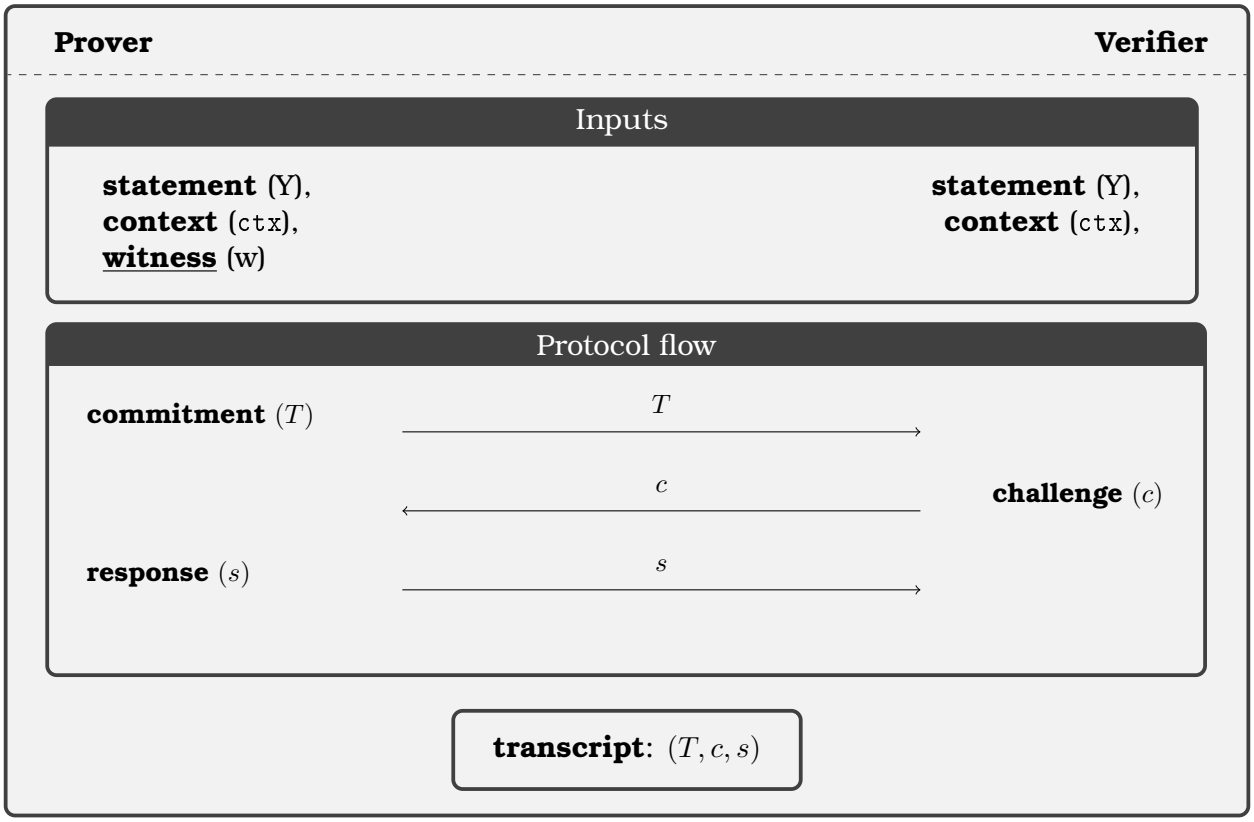

Figure 1: Overview of the steps composing a generic Σ-protocol. Underlined, the secrets in the protocol. In practice, the challenge is computed deterministically and the protocol is non-interactive. See Section 2.3 for more information.

1.4 Scope of this document

This document provides guidelines for secure implementations of Σ-protocols and is addressed to applied cryptographers and cryptographic engineers that are looking to implement a generic Σ-protocol or provide an ad-hoc instantiation as part of a larger protocol.

We consider the problem of having a high-quality entropy source well-suited for cryptographic purposes outside of the scope of this document. We will not talk about implementation of cryptographic primitives such as hash functions, or elliptic-curve algebra, but we will provide references for where to find them. We won't provide any guidance for securely storing secrets or producing constant-time code.

2 Generic Σ-Protocols

In the following, we describe a generic interface for Σ-protocols. Such protocols can be used to prove that, for some binary relation R and a public value Y, a witness w such that $(Y, w) \in \mathsf{R}$ is known. Basic statements include proofs of knowledge of a secret key, openings of commitments and, more generically, of representations. The type of these elements depends on the specific relation being implemented.

An important property of Σ-protocols is that they are composable: it is possible to prove conjunction and disjunctions of statements in zero-knowledge. Composition of Σ-protocols is dealt in Section 2.5; for an in-depth discussion of the underlying theory we refer to Cramer [Cra97].

2.1 Overview

Any Σ-protocol is structured as follows:

- the prover computes a fresh **commitment**, denoted T. This element is sometimes also called *nonce*.
- the prover computes, using the so-called Fiat-Shamir transform (cf. Section 2.3), a random **challenge**, denoted c.
- the prover computes a **response**, denoted s, that depends on the commitment and the challenge.

The final proof is constituted of the three-elements above (T, c, s), and is also referred to as the **transcript**.

2.2 An abstract class for generic Σ-protocols

We define a template class for Σ-protocols denoted `SigmaProtocol`. This is the basic interface that will be implemented in the remainder of this document. The methods composing `SigmaProtocol` should be considered *private* and *should not* be exposed externally. The public API is described later in Section 2.3. Instances are created via the `new` function, which is a class method, while all other functions act on a particular instance. A `SigmaProtocol` consists of the following methods:

- $\texttt{SigmaProtocol.new}(\texttt{ctx}, Y)$, denoting the initialization function. This function takes as input a label identifying local context information `ctx` (such as: session identifiers, to avoid replay attacks; protocol metadata, to avoid hijacking; optionally, a timestamp and some pre-shared randomness, to guarantee freshness of the proof) and a statement Y, the public information shared between prover and verifier. This function *should* pre-compute parts of the statement, or initialize the state of the hash function.
- $(T, \texttt{pstate}) \leftarrow \texttt{SigmaProtocol.prover\_commit}(w)$, denoting the *commitment phase*, that is, the computation of the first message sent by the prover in a Σ-protocol. This method outputs a new commitment together with its associated prover state, depending on the witness known to the prover and the statement to be proven. This step generally requires access to a high-quality entropy source. Leakage of even just of a few bits of the nonce could allow for the complete recovery of the witness [HGS01, Ble00, ANT+20]. The value T is meant to be shared, while `pstate` must be kept secret.

  In particular, we assume that there exists a function $\texttt{Serialize}(T)$ that serializes the commitment T and that its size is fixed and implicitly determined by the statement Y.

- $s \leftarrow$ `SigmaProtocol.prover_response`$(\texttt{pstate}, c)$, denoting the *response phase*, that is, the computation of the second message sent by the prover, depending on the witness, the statement, the challenge received from the verifier, and the internal state generated by `prover_commit`. The value s is meant to be shared.

- `yesno` $\leftarrow$ `SigmaProtocol.verifier`(T, c, s), denoting the *verifier algorithm*. This method checks that the *protocol transcript* is valid for the given statement. The verifier algorithm outputs nothing if verification succeeds, or an error if verification fails.

- `label` $\leftarrow$ `SigmaProtocol.label`$()$, returning a string of $32\,\mathrm{B}$ uniquely identifying the relation being proven. Implementing this function correctly is vital for security, and it must include all data available in the statement, as well as the parameters and the relation being proven. The label will be used for computing the challenge in the Fiat-Shamir transform (see Section 2.3). Precise indications on how to implement this function will be given in the following sections (see e.g. Sections 2.5 and 3.4).

  If the label is *not* tied to the relation, then it may be possible to produce another proof for a different relation without knowing its witness. Similarly, if the statement is not tied to the label, then it may be possible to produce proofs for another statement whose witness is related to the original proof.

The final two algorithms describe the *zero-knowledge simulator*. The simulator is primarily an efficient algorithm for proving zero-knowledge in a theoretical construction [BCF19], but it is also needed for verifying short proofs (cf. Section 2.3.6) and for or-composition (cf. Section 2.5), where a witness is not known and thus has to be simulated. We have:

- $s \leftarrow$ `SigmaProtocol.simulate_response`$()$, denoting the first stage of the *simulator*. It is an algorithm drawing a random response that follows the same output distribution of the algorithm `prover_response`.

- $T \leftarrow$ `SigmaProtocol.simulate_commitment`(c, s), denoting the second stage of the *simulator*, returning a commitment that follows the same output distribution of the algorithm `prover_commit`.

2.3 The Fiat-Shamir Transform

The interactive versions of the Σ-protocols presented in this document are not fit for practical applications, due to subtle yet impactful details in their security guarantees. In practice, public-coin protocols such as Σ-protocols can be converted into non-interactive ones through the Fiat and Shamir heuristic [FS87] and subsequent work, e.g., by Bernhard, Pereira and Warinschi [BPW12].

The underlying idea is to replace the verifier with a cryptographically secure hash function, hashing the context from the protocol and the previous message sent by the prover.

Warning 1: Interactive Σ-protocols

Σ-protocols illustrated in this spec **must not** be used interactively.

2.3.1 Syntax

When using a hash function we will employ comma-separated values to indicate values that are concatenated. To mitigate attacks and allow for state cloning, we also separate values with a vertical bar to indicate that they must appear in a subsequent block and that the previous is padded with zeros. For instance, $\mathsf{H}(a, b \mid c)$ denotes the hash of a concatenated to b, then padded with the null byte until reaching the closest multiple of `BLOCK_LEN`, and finally concatenating the resulting bit string to c.

Michele's note: Question for the community: how should padding be implemented? `10*1` used in sha3 would be straightforward but perhaps just filling with zeros is simpler and sufficient for our use-case?

Constants. Different hash functions may rely on different constants. We define some of the parameters associated to the hash function that will be used throughout this spec.

Const name	Notes
`DOMSEP`	Domain separator for this standard. Fixed to `'zkpstd/sigma/0.1'`.
`BLOCK_LEN`	Length of a hash block
`DIGEST_LEN`	Digest length

Michele's note: Question for the community: this spec assumes that the hash function takes as input a bit-string and returns a bit-string. This is however not the case for all hash functions that we would like to support. For instance, Poseidon [GKK+19]'s core function is a map $\mathbb{F}_p^* \to \mathbb{F}_p^\ell$ with ℓ fixed.

2.3.2 Hash Registry

This is the set of all supported hash functions. They take an arbitrary length sequence of bytes as input, and output $32\,\mathrm{B}$ of entropy.

Hash	Source	`BLOCK_LEN`	`DIGEST_LEN`
blake2b	[ANWW13]	128	64
sha3-256	[BDPA13]	136	32

Supported hash functions must all have `BLOCK_LEN` ≥ 32. We define labels and constant strings so that their length is always at most $32\,\mathrm{B}$. If `DIGEST_LEN` ≥ 32, we implicitly assume that the implementation considers only the least significant bytes and discards the remainder of the hash output when exactly $32\,\mathrm{B}$ bytes are needed.

2.3.3 Computing the challenge and seeding the commitment

Relying on a hash function allows us to both compute the challenge and generate the commitment securely. We define the following auxiliary variables that may be pre-computed during the call of `SigmaProtocol.new(ctx,`Y`)`. All variables will have fixed length `DIGEST_LEN` so to avoid canonicalization attacks.

$$\begin{aligned}\mathsf{hd} &\coloneqq \mathsf{H}(\texttt{DOMSEP})\\ \mathsf{ha} &\coloneqq \texttt{SigmaProtocol.label}()\\ \mathsf{hctx} &\coloneqq \mathsf{H}(\texttt{ctx})\end{aligned}$$

Seeding the commitment. The method `SigmaProtocol.prover_commit()` is a randomized function that generates a random element, unique per each execution. The commitment *should* be seeded as follows:

$$\begin{aligned}&\mathsf{z} \coloneqq \texttt{randombytes}(32)\\ &\textbf{return } \mathsf{H}(\mathsf{hd}, \mathsf{hctx}, \mathsf{ha} \mid \mathsf{z}, w).\end{aligned}$$

If the output length `DIGEST_LEN` of the hash function is not sufficient to provide enough entropy for the commitment, the seed may be expanded with a PRNG to provide the quantity of random bytes desired.

Michele's note: Question for the community: This must be more formal. What are we expecting the PRNG to be? It seems a waste not to use variable-length hash output from from supported hash functions. Should we opt for replacing sha3 with shake? How is the implementor supposed to know what is *enough entropy*? This could be misleading: there must be enough bytes to fill the domain of the morphism (which could be more than the security parameter!). Finally, the hash function used here may be different from the hash function to compute the challenge (this might make sense in the case of algebraic hashing). Should we support it?

Computing the challenge. The method `SigmaProtocol.challenge`(μ, T) is implemented as follows in order to produce a random challenge.

$$\begin{aligned}&\mathsf{hm} \coloneqq \mathsf{H}(\mu)\\ &\textbf{return } \mathsf{H}(\mathsf{hd}, \mathsf{hctx}, \mathsf{ha} \mid \mathsf{hm} \mid \texttt{Serialize}(T)).\end{aligned}$$

If no message is being set, i.e. if $\mu = \mathsf{None}$, then the implementation may decide to skip the computation of hm and consider it empty. This method is fixed for all implementations of `SigmaProtocol`. Note that the state of the hash function is partially shared between the commitment seed inputs and the challenge computation. Implementors may choose to store the partial hash state before generating the commitment, and reuse it when computing the challenge.

2.3.4 Non-interactive proofs

We define two *public methods* for generating proofs, meant to be exposed externally: `short_proof`, and `batchable_proof`. Since the challenge is computed deterministically from the commitment and the statement, it is not necessary to include the full transcript in a proof, as it can be deduced in the verification phase.

Short proofs are the most efficient if the protocol contains at least an *AND* composition (see Section 2.5 for a proper definition), and the gain in size is measured as $|\mathbf{T}| - \texttt{DIGEST\_LEN}$. (Note: the length of the commitment is the length of the statement.)

Batchable proofs are the canonical forms of proofs. Provers in the batchable form may raise an exception if the statement is not valid. Proofs are seen as fixed-length bit strings, whose exact length can be inferred from the statement during initialization of the Σ-protocol.

Remark 1: Witness validation

In the following we assume correctness of the witness w for the given statement Y. This can be ensured, e.g., by a higher-level application, or by running `SigmaProtocol.verifier`(T, c, s) before sending the resulting proof.

2.3.5 Batchable Proofs

Prover algorithm. The public function `SigmaProtocol.batchable_proof`(w, μ) computes:

1. $(T, \texttt{pstate}) \leftarrow$ `SigmaProtocol.prover_commit`(w).
2. $c \leftarrow$ `SigmaProtocol.challenge`(μ, T)
3. $s \leftarrow$ `SigmaProtocol.prover_response`$(\texttt{pstate}, c)$.
4. Return `Serialize`(T) concatenated to `Serialize`(s)

The challenge c is not provided within a batchable proof since it can be re-computed from the commitment.

Verifier algorithm. The verifier's algorithm `SigmaProtocol.batchable_verify`(π, μ) works as follows:

1. $(T, s) :=$ `Deserialize`(π)
2. $c \leftarrow$ `SigmaProtocol.challenge`(μ, T)
3. Return the result of `SigmaProtocol.verifier`(T, c, s).

Warning 2: Input validation

The case of batched verification must include an input validation sub-routine that asserts the statement and commitments are in question. In the case of elliptic curves described in Section 3.4, this boils down to point validation. Failure to properly check that a commitment is in the group could lead to subgroup attacks [vW96, LL97] or invalid curve attacks [BMM00, BBPV12].

2.3.6 Short Proofs

Prover algorithm. A new proof `SigmaProtocol.short_proof`(w, μ) is built as follows:

1. $(T, \texttt{pstate}) \leftarrow$ `SigmaProtocol.prover_commit`(w).
2. $c \leftarrow$ `SigmaProtocol.challenge`(μ, T)
3. $\mathrm{s} \leftarrow$ `SigmaProtocol.prover_response`$(\texttt{pstate}, c)$.
4. Return `Serialize`(c) concatenated to `Serialize`(s).

The commitment T is not provided within a short proof since it can be calculated again.

Verifier algorithm. The verifier's algorithm `SigmaProtocol.short_verify`(π, μ) works as follows:

1. $(c, s) \coloneqq$ `Deserialize`(π)
2. $T \leftarrow$ `SigmaProtocol.simulate_commitment`(c, s).
3. $c^* \leftarrow$ `SigmaProtocol.challenge`(μ, T)
4. Check whether $c = c^*$. Output **true** if this is the case, and **false** otherwise.

If input parsing fails, an exception should be raised. If verification fails, an exception should be raised. Otherwise, the verifier outputs **true**. Optionally, the implementation can choose to return the parsed statement.

Remark 2: Availability of the short form

While the short form as described here is applicable to all Σ-protocols currently covered by this document, it cannot be used for protocols where T is not uniquely determined by c and s, as is the case, e.g., for ZKBoo [GMO16], one-out-of-many proofs [GK15], or protocols, where a randomized signature is sent and proven correct subsequently, e.g., [PS16, CCs08].
A trade-off is presented, e.g., by Bobolz et al. [BEHF21], requiring an additional algorithm to shorten a full transcript to a compressed form which still allows for unique reconstruction of the transcript.

2.4 Input validation

Michele's note: TODO: validation of the statement, validation of the witness, validation of a short proof, and valddidation of batchable proof. can the challenge be zero; can the commitment be the point at infinity

2.5 Composition of Σ-Protocols

Σ-protocols can be composed to prove knowledge of multiple independent witnesses (*AND composition*), and for proving knowledge for one out of a set of witnesses (*OR composition*). An object `SigmaProtocol` can be seen as a recursive enumeration

```
enum SigmaProtocol {
  AndComposition {left: SigmaProtocol, right: SigmaProtocol},
  OrComposition  {left: SigmaProtocol, right: SigmaProtocol},
  [...]
}
```

whose instances expose the methods described above. The dots `[...]` denote (optional) Σ-protocols instantiations that will be covered in Section 3.2. Without loss of generality, the techniques presented in the following focus on the composition of two protocols. Composition of multiple protocols (e.g., proving knowledge of a witness for one out of many statements) can be achieved by recursively applying composition of two protocols.

2.5.1 AND Composition

In this section we show how to construct a Σ-protocol proving knowledge of multiple independent witnesses, e.g., knowledge of multiple secret keys, or openings to multiple commitments. That is, a Σ-protocol for the following relation:

$$\mathsf{R}_{\text{and}} = \{((Y_0, Y_1), (w_0, w_1) : (Y_0, w_0) \in \mathsf{R}_0 \ \wedge \ (Y_1, w_1) \in \mathsf{R}_1\}$$

For the rest of this section, the witness w for the Σ-protocol will now be a pair (w_0, w_1) of witnesses, and the associated statement Y will be a pair (Y_0, Y_1) of statements, where w_0 is the witness for the statement Y_0, and w_1 is the witness for Y_1.

Intuitively, the AND composition simply runs the instances of the different protocols to be composed in parallel, using the same challenge c for both instances. The verifier will then accept the protocol run if and only if all instances of the partial protocols output **true**.

The resulting Σ-protocol is specified by the following algorithms:

- `AndComposition.new(ctx, left, right)`: internally store `left` and `right`.
- $(\mathbf{T}, \texttt{pstate}) \leftarrow$ `AndComposition.prover_commit`$(\mathbf{w})$
  1. $(w_0, w_1) \coloneqq \mathbf{w}$
  2. $(T_0, \texttt{pstate}_0) \leftarrow$ `left.prover_commit`(w_0)
  3. $(T_1, \texttt{pstate}_1) \leftarrow$ `right.prover_commit`(w_1)
  4. Return $(\mathbf{T}, \texttt{pstate}) \coloneqq ((T_0, T_1), (\texttt{pstate}_0, \texttt{pstate}_1))$
- $\mathbf{s} \leftarrow$ `AndComposition.prover_response`$(\texttt{pstate}, c)$
  1. $(\texttt{pstate}_0, \texttt{pstate}_1) \coloneqq \texttt{pstate}$
  2. $s_0 \leftarrow$ `left.prover_response`$(\texttt{pstate}_0, c)$
  3. $s_1 \leftarrow$ `right.prover_response`$(\texttt{pstate}_1, c)$
  4. Return $\mathbf{s} \coloneqq (s_0, s_1)$
- `AndComposition.verifier`$(\mathbf{T}, c, \mathbf{s})$
  1. $(s_0, s_1) \coloneqq \mathbf{s}$.
  2. Return **true** if both `left.verifier`(T_0, c, s_0) and `right.verifier`(T_1, c, s_1) return **true**. Otherwise, return **false**.
- `AndComposition.label()` is computed as:

$$\mathsf{H}(\texttt{'and-composition'} \mid \texttt{left.label()}, \texttt{right.label()})$$

The supported hash functions are described in Section 2.3.2.

- $\mathbf{s} \leftarrow$ `AndComposition.simulate_response()` generates a simulated response as follows:
  1. $s_0 \leftarrow$ `left.simulate_response()`
  2. $s_1 \leftarrow$ `right.simulate_response()`
  3. Return $\mathbf{s} := (s_0, s_1)$.
- $\mathbf{T} \leftarrow$ `AndComposition.simulate_commitment`$(c, \mathbf{s})$ works as follows:
  1. $(s_0, s_1) := \mathbf{s}$.
  2. $T_0 \leftarrow$ `left.simulate_commitment`(c, s_0)
  3. $T_1 \leftarrow$ `right.simulate_commitment`(c, s_1)
  4. Return $\mathbf{T} := (T_0, T_1)$.

Warning 3: Witness equality

Note that the AND-composition defined in the following gives no guarantee about equality of the witnesses: if the same witness is used across different clauses of the AND-composition, the protocol will not guarantee that they are indeed the same. How to achieve such claims is discussed in Section 3.3.

2.5.2 OR Composition

In the following we explain how to construct a Σ-protocol proving knowledge of one out of a set of witnesses, for instance one of a set of secret keys (like ring signatures). That is, the algorithms specified below constitute a Σ-protocol for the following relation:

$$\mathsf{R}_{\text{or}} = \{((Y_0, Y_1), (w_0, w_1) : (Y_0, w_0) \in \mathsf{R}_0 \ \vee \ (Y_1, w_1) \in \mathsf{R}_1\}\,.$$

The statement Y is the pair (Y_0, Y_1) of the composing statements, and the witness w is the pair (w_0, w_1) of the witnesses for the respective relation. One of the witness can be set to None. In the following protocol specification, let j be such that w_j is known to the prover, whereas without loss of generality w_{1-j} is assumed to be unknown to the prover.

On a high level, the protocol works as follows. Using the simulator, the prover first simulates a transcript for the unknown witness (keeping the challenge and response of this transcript temporarily secret), and generates an honest commitment for the known witness. Having received the challenge, the prover then computes a challenge for the known witness, depending on the received challenge and the one from the simulated transcript. Having computed the response, the prover transfers the responses of both transcripts, as well as the partial challenges to the verifier, who accepts if and only if both instances of the partial protocols output **true** and the partial challenges correctly add up to the random challenge.

The main procedures of the resulting Σ-protocol are specified by the following algorithms:

- `OrComposition.new(ctx, left, right)`: internally store `left` and `right`.
- $(\mathbf{T}, \texttt{pstate}) \leftarrow$ `OrComposition.prover_commit`$(\mathbf{w})$:

1. `Prover` $= [$`left`$,$ `right`$]$
2. $(w_0, w_1) \coloneqq \mathbf{w}$, and let $j \in \{0, 1\}$ be the first index for which $w_j \neq$ None
3. $(T_j,$ `pstate`$_j) \leftarrow$ `Prover`$[j]$`.prover_commit`(Y_j, w_j)
4. $s_{1-j} \leftarrow$ `Prover`$[1-j]$`.simulate_response`$()$
5. Choose a random c_{1-j} in $\mathcal{C}$
6. $T_{1-j} \leftarrow$ `Prover`$[1-j]$`.simulate_commitment`(c_{1-j}, s_{1-j})
7. Return $(\mathbf{T},$ `pstate`$) \coloneqq ((T_0, T_1),$ (`pstate`$_j, c_{1-j}, s_{1-j}))$

- $\mathbf{s} \leftarrow$ `OrComposition.prover_response`$($`pstate`$, c)$:

  1. (`pstate`$_j, c_{1-j}, s_{1-j}) \coloneqq$ `pstate`
  2. $c_j \coloneqq c \oplus c_{1-j}$
  3. $s_j \leftarrow$ `Prover`$[j]$`.prover_response`$($`pstate`$_j, c_j)$
  4. Return $\mathbf{s} \coloneqq (s_0, s_1, c_0)$.

- `OrComposition.verifier`$(\mathbf{T}, c, \mathbf{s})$,

  1. $(s_0, s_1, c_0) \coloneqq \mathbf{s}$.
  2. $c_1 \coloneqq c \oplus c_0$.
  3. Return **true** if both `left.verifier`(T_0, c_0, s_0) and `right.verifier`(T_1, c_1, s_1) return **true**. Otherwise, return **false**.

- `OrComposition.label`$()$ is computed as:

  $$\mathsf{H}(\texttt{'or-composition'} \mid \texttt{left.label}(), \texttt{right.label}())$$

  The supported hash functions are described in Section 2.3.2.

- $\mathbf{s} \leftarrow$ `OrComposition.simulate_response`$()$

  1. $(Y_0, Y_1) \coloneqq \mathbf{Y}$.
  2. $s_0 \leftarrow$ `left.simulate_response`$()$
  3. $s_1 \leftarrow$ `right.simulate_response`$()$
  4. Choose a random c_0 in $\mathcal{C}$.
  5. Return $\mathbf{s} \coloneqq (s_0, s_1, c_0)$.

- $\mathbf{T} \leftarrow$ `OrComposition.simulate_commitment`$(c, \mathbf{s})$

  1. $(s_0, s_1, c_0) \coloneqq \mathbf{s}$.
  2. $c_1 \coloneqq c \oplus c_0$.
  3. $T_0 \leftarrow$ `left.simulate_commitment`(c_0, s_0)
  4. $T_1 \leftarrow$ `right.simulate_commitment`(c_1, s_1)
  5. Return $\mathbf{T} \coloneqq (T_0, T_1)$.

3 Σ-protocols on elliptic curves

The following section presents concrete instantiations of Σ-protocols over elliptic curves.

Remark 3: Protocols for residue classes

Because of their dominance, the presentation in the following focuses on proof goals over elliptic curves, therefore leveraging additive notation. For prime-order subgroups of residue classes, all notation needs to be changed to multiplicative, and references to elliptic curves (e.g., `curve`) need to be replaced by their respective counterparts over residue classes.

3.1 Ciphersuite Registry

We advise for the use of prime-order elliptic curves of size either 256 or 512 bits, depending on the desired security of the upper layers in the protocol[2].

Curve	Identifier	Security Level	Sources
P-521	`'-p-521'`	256	[NIS00]
P-256	`'-p-256'`	128	[NIS00]
secp256k1	`'-secp256k1'`	128	[SEC00]
Ristretto	`'-ristretto'`	128	[dVGT+20]
BLS12-381	`'-bls12-381'`	128	[Bow17]

We denote with $\mathbb{G}$ the prime-order group of the elliptic curve, with $\mathbb{F}_p$ the scalar field, and with G the generator of $\mathbb{G}$ chosen as per the curve parameters. We assume that all above curve parameters also provide the following group operations: check for equality, identity, addition, and scalar multiplication. Optionally, implementation might implement Pippenger's algorithm [Pip80] for multi-scalar multiplication. In addition, we consider:

- an identifier for the curve, chosen from the table above, and denoted `curve`;
- a deterministic sub-procedure $a := \texttt{FromBytes}(b)$, taking as input a bit string b of length $32\,\mathrm{B}$, and mapping it into an element $a \leftarrow\!\$\, \mathbb{F}_p$;
- a deterministic sub-procedure $s := \texttt{Serialize}(P)$, taking as input a group element $P \in \mathbb{G}$ and returning a fixed-length sequence of bits. For elliptic curve groups, `Serialize` must provide a compressed representation of the *affine* representation, such as the x-coordinate of P and one bit determining the sign of y.
- a deterministic sub-procedure $P := \texttt{Deserialize}(s)$, taking a (fixed-length and curve-dependent) sequence of bits and returning an elliptic curve point. This procedure may raise an exception or output `None` if the conversion fails.

[2]For instance, proving a DH relation with one fixed group element such as a public key, might expose the protocol to cryptanalytic attacks such as Brown-Gallant [BG04] and Cheon's attack [Che06], and some implementations might opt for larger curve sizes. We consider these attacks out of scope for this standardization effort, and believe this analysis should be deferred to the concrete security study of the larger protocol.

3.2 Basic Σ-Protocols in prime-order groups

We describe an abstract class for proving knowledge of a preimage under an arbitrary *group homomorphism*, which is a mapping between two groups respecting the structure of the groups. In particular, as will be discussed in Section 3.4, many statements related to discrete logarithms or representations in groups of prime order, can be expressed as statements over group homomorphisms. For an in-depth discussion of the underlying theory we refer to Cramer [Cra97].

Definition 1 *For two groups $\mathbb{G}_1, \mathbb{G}_2$, a function $\varphi : \mathbb{G}_1 \to \mathbb{G}_2 : x \mapsto \varphi(x)$ is a* (group) homomorphism, *if and only if for all $a, b \in \mathbb{G}_1$ it holds that $\varphi(a + b) = \varphi(a) + \varphi(b)$.*

Readers not familiar with the notation of group homomorphism may think of φ as a linear function from n elements into m elements.

Example 3: Discrete logarithm equality

Looking at the relation R_{dleq} from ex 1, the relevant homomorphism is given by:

$$\varphi_{\text{dleq}} : \mathbb{F}_p \to \mathbb{G}^2 : w \mapsto (wG, wH)\,.$$

If equality of discrete logarithms within *different* groups of the same prime order p is to be proven, the homomorphism to be considered would be:

$$\varphi'_{\text{dleq}} : \mathbb{F}_p \to \mathbb{G}_1 \times \mathbb{G}_2 : w \mapsto (wG, wH)\,,$$

where G and H would now be generators for $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. All the techniques discussed in the remainder of this spec equally apply to both cases.

Example 4: Representation

Looking at the relation R_{rep} from ex 2, the relevant homomorphism is given by:

$$\varphi_{\text{rep}} : \mathbb{F}_p^2 \to \mathbb{G} : (w_1, w_2) \mapsto w_1 G + w_2 H\,.$$

We provide a generic template for all Σ-protocols for statements of the following form over DLOG groups:

$$\mathsf{R}_{\text{dlog}} = \left\{((Y_1, \ldots, Y_m), (w_1, \ldots, w_n)) \in \mathbb{G}^m \times \mathbb{F}_p^n : (Y_1, \ldots, Y_m) = \varphi(w_1, \ldots, w_n)\right\}$$

where $\varphi : \mathbb{F}_p^n \to \mathbb{G}^m$ is a group homomorphism.

Remark 4: Selective disclosure of witnesses

Note that in the following descriptions, all witnesses are assumed to be kept secret, i.e., none of them is disclosed to the verifier. In case it is required to disclose w_j, as is the case, e.g., in the context of attribute-based credential systems, the relation

to be proven can be rewritten as follows:

$$\mathsf{R}_{\mathrm{dlog}}{}' = \left\{ \begin{array}{r} ((Y'_1,\ldots,Y'_m)),(w_1,\ldots,w_{j-1},w_{j+1},\ldots,w_n)) \in \mathbb{G}^m \times \mathbb{F}_p^{n-1} : \\ (Y'_1,\ldots,Y'_m) = \psi(w_1,\ldots,w_{j-1},w_{j+1},\ldots,w_n) \end{array} \right\}$$

where

$$(Y'_1,\ldots,Y'_m) := (Y_1,\ldots,Y_m) - \varphi(0,\ldots,0,w_j,0,\ldots,0) \text{ and}$$
$$\psi(w_1,\ldots,w_{j-1},w_{j+1},\ldots,w_n) := \varphi(w_1,\ldots,w_{j-1},0,w_{j+1},\ldots,w_n)\,.$$

However, the following *defines neither the morphism nor the label associated to the protocol*. These will be defined later in the specific protocols.

- `DlogTemplate.new(ctx,` $\mathbf{Y}$`)` internally stores $\mathbf{Y}$ and `ctx`.
- $(\mathbf{T}, \texttt{pstate}) \leftarrow$ `DlogTemplate.prover_commit(`$\mathbf{w}$`)` consists of the following steps:
  1. Sample random elements $r_1,\ldots,r_n \leftarrow\!\!\$\, \mathbb{F}_p$
  2. $\mathbf{T} := (T_1,\ldots,T_m) := \varphi(r_1,\ldots,r_n)$
  3. $\texttt{pstate} := (r_1,\ldots,r_n)$
  4. Return $(\mathbf{T}, \texttt{pstate})$
- $\mathbf{s} \leftarrow$ `DlogTemplate.prover_response(pstate,` c`)` proceeds as follows:
  1. $(r_1,\ldots,r_n) := \texttt{pstate}$
  2. $(w_1,\ldots,w_n) := \mathbf{w}$
  3. $e :=$ `FromBytes(`c`)`.
  4. For $i = 1,\ldots,n$: $s_i := r_i + ew_i$
  5. Return $\mathbf{s} := (s_1,\ldots,s_n)$.
- `DlogTemplate.label()` return `morphism_label()`.
- `DlogTemplate.verifier(`$\mathbf{T}, c, \mathbf{s}$`)` proceeds as follows:
  1. $(s_1,\ldots,s_n) := \mathbf{s}$
  2. $(T_1,\ldots,T_m) := \mathbf{T}$
  3. $e :=$ `FromBytes(`c`)`.
  4. For $i = 1,\ldots,n$: check $s_i \in \mathbb{F}_p$
  5. For $j = 1,\ldots,m$: check $T_j \in \mathbb{G}$
  6. Return **true** if $(T_1 + eY_1,\ldots,T_m + eY_m) = \varphi(s_1,\ldots,s_n)$; **false** otherwise
- $\mathbf{s} \leftarrow$ `DlogTemplate.simulate_response()`:
  1. Sample random elements $s_1,\ldots,s_n \leftarrow\!\!\$\, \mathbb{F}_p$
  2. Return $(s_1,\ldots,s_n)$
- $\mathbf{T} \leftarrow$ `DlogTemplate.simulate_commitment(`$c, \mathbf{s}$`)`:

1. $(Y_1, \ldots, Y_m) \coloneqq \mathbf{Y}$.
2. $(s_1, \ldots, s_n) \coloneqq \mathbf{s}$.
3. $e \coloneqq \texttt{FromBytes}(c)$.
4. $(T_1, \ldots, T_m) \coloneqq \varphi(s_1, \ldots, s_n) - e \cdot (Y_1, \ldots, Y_m)$.
5. Output $\mathbf{T} \coloneqq (s_1, \ldots, s_n)$.

3.3 Advanced: proving linear relations

While the above protocol allows one to efficiently prove knowledge of a pre-image under a homomorphism, many protocols found in the literature require one to prove relations among witnesses. Specifically, they require to prove relations like the following:

$$\mathsf{R}_{\text{lin}} = \left\{ ((Y_1, \ldots, Y_m), (w_1, \ldots, w_n)) : \begin{array}{c} (Y_1, \ldots, Y_m) = \varphi(w_1, \ldots, w_n) \\ A \cdot (w_1, \ldots, w_n)^\intercal = (b_1, \ldots, b_k)^\intercal \end{array} \right\},$$

where the matrix $A \in \mathbb{F}_p^{k \times n}$ and vector $(b_1, \ldots, b_k) \in \mathbb{F}_p^k$ specify the system of linear equations.

In the following, we sketch how such relations can be translated into relations of the form discussed in Section 3.2. We assume that A is of the following form:

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1k} & 1 & 0 & 0 & \ldots & 0 \\ a_{21} & \ldots & a_{2k} & 0 & 1 & 0 & \ldots & \vdots \\ \vdots & & \vdots & \vdots & & \vdots & & 0 \\ \vdots & & \vdots & \vdots & & 0 & 1 & 0 \\ a_{k1} & \ldots & a_{kk} & 0 & \ldots & \ldots & 0 & 1 \end{pmatrix}$$

Note that this is without loss of generality. If the system of linear equations has a different form, the above form can always be achieved using Gaussian elimination [Sho08, Sec. 14.4] and re-ordering of the witnesses. Note that we only need to consider the case where $k < n$, as otherwise the linear equations would uniquely determine the witnesses, which is not desirable in our context.

The following relation $\mathsf{R}_{\text{lin}}{}'$ is now equivalent to that specified by R_{lin}:

$$\mathsf{R}_{\text{lin}}{}' = \{((Y_1, \ldots, Y_m), (w_1, \ldots, w_{n-k})) : (Y_1, \ldots, Y_m) = \psi(w_1, \ldots, w_{n-k})\}$$

where

$$\psi(w_1, \ldots, w_{n-k}) \coloneqq \varphi(w_1, \ldots, w_{n-k}, -\sum_{i=1}^{n-k} a_{1i} w_i, \ldots, -\sum_{i=1}^{n-k} a_{ki} w_i) \text{ and}$$
$$(Y'_1, \ldots, Y'_m) \coloneqq (Y_1, \ldots, Y_m) - \varphi(0, \ldots, 0, b_1, \ldots, b_k)\,.$$

3.4 Instantiations

Let $\mathbb{G}$ be a group over an elliptic curve with prime order p. Denote with $G \in \mathbb{G}$ a generator of $\mathbb{G}$.

3.4.1 Schnorr signatures

Schnorr signatures, also known as Schnorr proofs or DLOG proofs, prove knowledge of the discrete logarithm $w \in \mathbb{F}_p$ of a point $Y = wG$ in base G.

- $\varphi : \mathbb{F}_p \to \mathbb{G} : w \mapsto wG$
- `Schnorr.morphism_label()`: return:

$$\mathsf{H}(\texttt{'schnorrsig'}, \texttt{curve} \mid \texttt{Serialize}(G), \texttt{Serialize}(Y))$$

For a description of this proof goal in the general case of residue classes, we also refer to [ZKP19, 1.4.1].

3.4.2 Discrete logarithm equality

So-called DLEQ proofs prove equality of the discrete logarithm, that is: $Y_1 = wG$ and $Y_2 = wH$.

- $\varphi : \mathbb{F}_p \to \mathbb{G}^2 : w \mapsto (wG, wH)$
- `Dleq.morphism_label()`: return

$$\mathsf{H}(\texttt{'dleq'}, \texttt{curve} \mid \texttt{Serialize}(G), \texttt{Serialize}(H), \texttt{Serialize}(Y_1), \texttt{Serialize}(Y_2))$$

3.4.3 Diffie-Hellman

Let $\mathbb{G}$ be a group over an elliptic curve with prime order p. Proving knowledge of the exponents of a valid Diffie-Hellman triple means proving knowledge of $w_1, w_2 \in \mathbb{F}_p$ such that $Y_1 = w_1 G$, $Y_2 = w_2 G$, and $Y_3 = w_1 w_2 G$. The mapping $\mathbb{F}_p^2 \to \mathbb{G}^3 : (w_1, w_2) \mapsto (w_1 G, w_2 G, w_1 w_2 G)$ is not a homomorphism, and consequently the basic protocol presented before cannot be deployed directly. However, the required multiplicative relation can be proven by observing that the proof goal is equivalent to $Y_1 = w_1 G$, $Y_2 = w_2 G$, and $Y_3 = w_2 Y_1$, leading the homomorphism

- $\varphi : \mathbb{F}_p^2 \to \mathbb{G}^3 : (w_1, w_2) \mapsto (w_1 G, w_2 G, w_2 Y_1)$
- `Dh.morphism_label()` return:

$$\mathsf{H}(\texttt{'diffie-hellman'}, \texttt{curve} \mid \texttt{Serialize}(G), \texttt{Serialize}(Y_1), \texttt{Serialize}(Y_2))$$

As shown in this example, and in contrast to linear relations, multiplicative relations among witnesses typically require a reformulation of the proof goal.

3.4.4 Representation

Let $\mathbb{G}$ be a group over an elliptic curve of prime order p, and let $G_1, \ldots, G_m$ be generators of $\mathbb{G}$. Proving knowledge of a valid opening of a Pedersen commitment means proving knowledge of $w_1, w_2, \ldots, w_m \in \mathbb{F}_p$ such that $Y = w_1 G_1 + w_2 G_2 + \cdots + w_m G_m$.

- $\varphi : \mathbb{F}_p^m \to \mathbb{G} : (w_1, w_2, \ldots, w_m) \mapsto \sum_i w_i G_i$
- `Rep.morphism_label()` returns

  $$\mathsf{H}(\texttt{'representation'}, \texttt{curve} \mid \texttt{Serialize}(G_1), \cdots, \texttt{Serialize}(G_m), \texttt{Serialize}(Y))$$

Example 5: Range proofs via bit decomposition

Let $\mathbb{G}$ be a cyclic group of prime order p, and let G and H be generators of $\mathbb{G}$, and let ℓ be a non-negative integer satisfying $\ell < \log_2 p$. Consider the following relation:

$$\mathsf{R}_{\text{range}} = \left\{ (Y, (w_1, w_2))) \;:\; Y = w_1 G + w_2 H \;\wedge\; w_1 \in [0, 2^\ell) \right\}.$$

Multiple techniques for proving that a secret witness lies within a certain range, cf. Morais et al. [MKvWK19] for a survey. We will use the so-called *bit decomposition* approach.

To do so, the prover computes $w_{1,i} \in \{0, 1\}$ for $i = 0, \ldots, \ell - 1$ such that $w_1 = \sum_{i=0}^{\ell-1} 2^i w_{1,i}$, and computes commitments to those individual bits, i.e., $Y_i = w_{1,i} G + w_{2,i} H$ for $w_{2,i} \leftarrow\!\!\$\, \mathbb{F}_p$. Furthermore, it sets $w^* = w_2 - \sum_{i=0}^{\ell-1} 2^i w_{2,i}$.

Assuming that the discrete logarithm problem is hard in $\mathbb{G}$, the above relation is now equivalent to the following relation:

$$\mathsf{R}_{\text{range}}{}' = \left\{ \left((Y, (Y_i)_{i=0}^{\ell-1}), (w_1, w_2, (w_{2,i})_{i=0}^{\ell-1}), w^*) \right) \;:\; Y = w_1 G + w_2 H \;\wedge \right. \tag{1}$$

$$Y - \sum_{i=0}^{\ell-1} 2^i Y_i = w^* H \;\wedge \tag{2}$$

$$\left. \bigwedge_{i=0}^{\ell-1} (Y_i = w_{2,i} H \;\vee\; Y_i - G = w_{2,i} H) \right\}. \tag{3}$$

It can now be seen that (1) ensures knowledge of the witnesses w_1 hidden within Y. Furthermore, (2) guarantees that the values hidden within Y_i correctly add up to w_1, i.e., that $w_1 - \sum_{i=0}^{\ell-1} 2^i w_{1,i} = 0$. Finally, the two clauses in (3) ensure that each Y_i is a commitment to either 0 or 1, thus implying the bound on w_1.

While (1) can be proven using the protocol from Section 3.4.4, (2) and (3) can be proven using those from Section 3.4.1. The different clauses can finally be composed using nested protocol compositions from Section 2.5.

3.5 Batch verification

The batchable form can take advantage of the following fact. Given ℓ verification equations of the form:

$$T_i + c_i Y_i = \sum_j s_j G_{i,j}$$

for $i = 1, \ldots, \ell$, the verifier can sample a random vector of coefficients $\mathbf{a} \in \mathbb{F}_p^\ell$ and instead test:

$$\left(\sum_{i=1}^{\ell} a_i T_i\right) + \left(\sum_{i=1}^{\ell} a_i \cdot c_i Y_i\right) = \left(\sum_{i=1}^{\ell} a_i \cdot \sum_j s_j G_{i,j}\right).$$

If the matrix $G_{i,j}$ of generators has identical rows, by linearity the right-hand side can be computed as a single scalar product. If the statements Y_i's have identical rows, by linearity the second term in the equation can be computed as a single scalar product.

In any case, the test can be efficiently implemented as a single multiscalar multiplication, minimizing the number of group operations needed:

$$\left(\sum_{i=1}^{\ell} a_i T_i\right) + \left(\sum_{i=1}^{\ell} (a_i \cdot c_i) Y_i\right) + \left(\sum_{i=1}^{\ell} \sum_j (-a_i \cdot s_j) G_{i,j}\right) = 0.$$

The random vector $\mathbf{a}$ can be *deterministically* generated by fixing $a_1 := 1$ and setting $(a_2, \ldots, a_\ell) := \mathsf{PRG}(\mathsf{H}(c, \mathbf{s}))$ [WNR18].

4 Encoding the statement

Statements in Σ-protocols take the form of a labeled binary tree: `Statement` is either:

- a label AND, or OR, and two children left and right of type `Statement`
- a `StatementLeaf` instance. Objects of this type depend on the specific algebraic setting used, and will be treated in Section 3.2.

Statements are serialized depth-first. There are many different options for serialization that could be considered:

- Concise Binary Object Representation (CBOR) RFC7049
- The zk proof Reference document provides a serialization document for `r1cs` [ZKP19, 3.4.2], but there is nothing in it.

Michele's note: Question for the community: what kind of serialization format should we pick? It seems that no clear choice has been taken already for `r1cs` and CBOR seems to be the most reasonable choice?

5 Additional proof types

Other protocols are not included here and are not part of the scope of the current version of this spec.

- mpc-in-the-head protocol such as ZKBoo [GMO16]
- one-out-of-many proofs such as [GK15]
- lwe-based sigma protocol [ACK21]
- syndrome decoding and LPN [Ste94, JKPT12]

References

[ACK21] Thomas Attema, Ronald Cramer, and Lisa Kohl. A compressed Σ-protocol theory for lattices. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 549–579, Virtual Event, August 2021. Springer, Heidelberg.

[ANT+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 225–242. ACM Press, November 2020.

[ANWW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 13*, volume 7954 of *LNCS*, pages 119–135. Springer, Heidelberg, June 2013.

[BBPV12] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 171–186. Springer, Heidelberg, February / March 2012.

[BCF19] Daniel Benarroch, Matteo Campanelli, and Dario Fiore. Community Standards Proposal for Commit-and-Prove Zero-Knowledge Proof Systems. ZKProof Community Standard Proposal, available at https://github.com/zkpstandard/zkreference/tree/master/standards-proposals, 2019. accessed on February 22, 2021.

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 313–314. Springer, Heidelberg, May 2013.

[BEHF21] Jan Bobolz, Fabian Eidens, Raphael Heitjohann, and Jeremy Fell. Cryptimeleon: A library for fast prototyping of privacy-preserving cryptographic schemes. Cryptology ePrint Archive, Report 2021/961, 2021. https://eprint.iacr.org/2021/961.

[BG93] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 390–420. Springer, Heidelberg, August 1993.

[BG04] Daniel R. L. Brown and Robert P. Gallant. The static Diffie-Hellman problem. Cryptology ePrint Archive, Report 2004/306, 2004. https://eprint.iacr.org/2004/306.

[Ble00] Daniel Bleichenbacher. On the generation of one-time keys in dl signature schemes. In *Presentation at IEEE P1363 working group meeting*, page 81, 2000.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146. Springer, Heidelberg, August 2000.

[Bow17] Sean Bowe. Bls12-381: New zk-snark elliptic curve construction, 2017. Available at: https://electriccoin.co/blog/new-snark-curve/.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 626–643. Springer, Heidelberg, December 2012.

[CCs08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 234–252. Springer, Heidelberg, December 2008.

[Che06] Jung Hee Cheon. Security analysis of the strong Diffie-Hellman problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 1–11. Springer, Heidelberg, May / June 2006.

[CMZ14] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic MACs and keyed-verification anonymous credentials. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1205–1216. ACM Press, November 2014.

[Cra97] Ronald Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, CWI Amsterdam, The Netherlands, 1997.

[Dam04] Ivan Damgård. On Σ-Protocols. Lecture on Crptologic Protocol Theory; Faculty of Science, University of Aarhus, 2004.

[dVGT+20] Henry de Valence, Jack Grigg, George Tankersley, Filippo Valsorda, isis lovecruft, and Mike Hamburg. The ristretto255 and decaf448 Groups. Internet-Draft draft-irtf-cfrg-ristretto255-decaf448-00, Internet Engineering Task Force, October 2020. Work in Progress.

[FFS87] Uriel Feige, Amos Fiat, and Adi Shamir. Zero knowledge proofs of identity. In Alfred Aho, editor, *19th ACM STOC*, pages 210–217. ACM Press, May 1987.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[GK15] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280. Springer, Heidelberg, April 2015.

[GKK+19] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and Poseidon: New hash functions for zero knowledge proof systems. Cryptology ePrint Archive, Report 2019/458, 2019. https://eprint.iacr.org/2019/458.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[HGS01] Nick A Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.

[HR11] Feng Hao and Peter Y. A. Ryan. Password authenticated key exchange by juggling. In Bruce Christianson, James A. Malcolm, Vashek Matyas, and Michael Roe, editors, *Security Protocols XVI*, pages 159–171, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[JKPT12] Abhishek Jain, Stephan Krenn, Krzysztof Pietrzak, and Aris Tentes. Commitments and efficient zero-knowledge proofs from learning parity with noise. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 663–680. Springer, Heidelberg, December 2012.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized schnorr threshold signatures. Cryptology ePrint Archive, Report 2020/852, 2020. https://eprint.iacr.org/2020/852.

[LL97] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 249–263. Springer, Heidelberg, August 1997.

[MKvWK19] Eduardo Morais, Tommy Koens, Cees van Wijk, and Aleksei Koren. A survey on zero knowledge range proofs and applications. *SN Applied Sciences*, 1(8):946, 2019.

[MP15] Gregory Maxwell and Andrew Poelstra. Borromean Signatures, 2015. Available at https://raw.githubusercontent.com/Blockstream/borromean_paper/master/borromean_draft_0.01_34241bb.pdfx.

[NIS00] NIST. Digital signature standard. FIPS 186-2, 2000. Available at: https://csrc.nist.gov/csrc/media/publications/fips/186/2/archive/2000-01-27/documents/fips186-2.pdf.

[NRSW20] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Schnorr multi-signatures with verifiably deterministic nonces. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1717–1731. ACM Press, November 2020.

[Pip80] Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980.

[PS97] David Pointcheval and Jacques Stern. New blind signatures equivalent to factorization (extended abstract). In Richard Graveman, Philippe A. Janson, Clifford Neuman, and Li Gong, editors, *ACM CCS 97*, pages 92–99. ACM Press, April 1997.

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 111–126. Springer, Heidelberg, February / March 2016.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.

[SEC00] Certicom research, standards for efficient cryptography group (SECG) — sec 1: Elliptic curve cryptography. http://www.secg.org/secg_docs.htm, September 20, 2000. Version 1.0.

[Sho08] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008. 2nd edition.

[Ste94] Jacques Stern. A new identification scheme based on syndrome decoding. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 13–21. Springer, Heidelberg, August 1994.

[vW96] Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 332–343. Springer, Heidelberg, May 1996.

[WNR18] Pieter Wuille, Jonasonas Nick, and Tim Ruffing. Bip 0340, 2018. Available at: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-July/016203.html.

[ZKP19] ZKProof. ZKProof Community Reference v0.2. Technical report, 2019. accessed on February 8, 2021.