Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 10 February 2026

M. Orr湛
CNRS
C. Yun
Apple, Inc.

9 August 2025

Interactive Sigma Proofs
draft-irtf-cfrg-sigma-protocols-00

Abstract

This document describes interactive sigma protocols, a class of secure, general-purpose zero-knowledge proofs of knowledge consisting of three moves: commitment, challenge, and response. Concretely, the protocol allows one to prove knowledge of a secret witness without revealing any information about it.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://mmaker.github.io/draft-irtf-cfrg-sigma-protocols/draft-irtf-cfrg-sigma-protocols.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-irtf-cfrg-sigma-protocols/.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (mailto:cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cfrg. Subscribe at https://www.ietf.org/mailman/listinfo/cfrg/.

Source for this draft and an issue tracker can be found at https://github.com/mmaker/draft-irtf-cfrg-sigma-protocols.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

Any sigma protocol must define three objects: a _commitment_ (computed by the prover), a _challenge_ (computed by the verifier), and a _response_ (computed by the prover).

1.1. Core interface

The public functions are obtained relying on an internal structure containing the definition of a sigma protocol.

```
class SigmaProtocol:
  def new(instance) -> SigmaProtocol
  def prover_commit(self, witness, rng) -> (commitment, prover_state)
  def prover_response(self, prover_state, challenge) -> response
  def verifier(self, commitment, challenge, response) -> bool
  def serialize_commitment(self, commitment) -> bytes
  def serialize_response(self, response) -> bytes
  def deserialize_commitment(self, data: bytes) -> commitment
  def deserialize_response(self, data: bytes) -> response
  # optional
  def simulate_response(self, rng) -> response
  # optional
  def simulate_commitment(self, response, challenge) -> commitment
```

Where:

* new(instance) -> SigmaProtocol, denoting the initialization function. This function takes as input an instance generated via the LinearRelation, the public information shared between prover and verifier.

* prover_commit(self, witness: Witness, rng) -> (commitment, prover_state), denoting the *commitment phase*, that is, the computation of the first message sent by the prover in a Sigma protocol. This method outputs a new commitment together with its associated prover state, depending on the witness known to the prover, the statement to be proven, and a random number generator rng. This step generally requires access to a high-quality entropy source to perform the commitment. Leakage of even just of a few bits of the commitment could allow for the complete recovery of the witness. The commitment is meant to be shared, while prover_state must be kept secret.

* prover_response(self, prover_state, challenge) -> response, 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 prover_state. The returned value response is meant to be shared.

* verifier(self, commitment, challenge, response) -> bool, denoting the *verifier algorithm*. This method checks that the protocol transcript is valid for the given statement. The verifier algorithm outputs true if verification succeeds, or false if verification fails.

* serialize_commitment(self, commitment) -> bytes, serializes the commitment into a canonical byte representation.

* serialize_response(self, response) -> bytes, serializes the response into a canonical byte representation.

* deserialize_commitment(self, data: bytes) -> commitment, deserializes a byte array into a commitment. This function can raise a DeserializeError if deserialization fails.

* deserialize_response(self, data: bytes) -> response, deserializes a byte array into a response. This function can raise a DeserializeError if deserialization fails.

The final two algorithms describe the *zero-knowledge simulator*. In particular, they may be used for proof composition (e.g. OR-composition). The function simulate_commitment is also used when verifying short proofs. We have:

* simulate_response(self, rng) -> response, denoting the first stage of the simulator. It is an algorithm drawing a random response given a specified cryptographically secure RNG that follows the same output distribution of the algorithm prover_response.

* simulate_commitment(self, response, challenge) -> commitment, returning a simulated commitment -- the second phase of the zero-knowledge simulator.

Together, these zero-knowledge simulators provide a transcript that should be computationally indistinguishable from the transcript generated by running the original sigma protocol.

The abstraction SigmaProtocol allows implementing different types of statements and combiners of those, such as OR statements, validity of t-out-of-n statements, and more.

2. Sigma protocols over prime-order groups

The following sub-section presents concrete instantiations of sigma protocols over prime-order elliptic curve groups. It relies on a prime-order elliptic-curve group as described in Section 2.1.

Valid choices of elliptic curves can be found in Section 2.3.

Traditionally, sigma protocols are defined in Camenisch-Stadler notation as (for example):

```
1. DLEQ(G, H, X, Y) = PoK{
2.   (x):        // Secret variables
3.   X = x * G, Y = x * H        // Predicates to satisfy
4. }
```

In the above, line 1 declares that the proof name is "DLEQ", the public information (the *instance*) consists of the group elements (G, X, H, Y) denoted in upper-case. Line 2 states that the private information (the *witness*) consists of the scalar x. Finally, line 3 states that the linear relation that need to be proven is x * G = X and x * H = Y.

2.1. Group abstraction

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.

We detail the functions that can be invoked on these objects. Example choices can be found in Section 2.3.

2.1.1. Group

* identity(), returns the neutral element in the group.
* generator(), returns the generator of the prime-order elliptic-curve subgroup used for cryptographic operations.
* order(): Outputs the order of the group p.
* random(): outputs a random element in the group.

* serialize(elements: [Group; N]), serializes a list of group elements and returns a canonical byte array buf of fixed length Ne * N.

* deserialize(buffer), attempts to map a byte array buffer of size Ne * N into [Group; N], and fails if the input is not the valid canonical byte representation of an element of the group. This function can raise a DeserializeError if deserialization fails.

* add(element: Group), implements elliptic curve addition for the two group elements.

* equal(element: Group), returns true if the two elements are the same and false otherwise.

* scalar_mul(scalar: Scalar), implements scalar multiplication for a group element by an element in its respective scalar field.

In this spec, instead of add we will use + with infix notation; instead of equal we will use ==, and instead of scalar_mul we will use *. A similar behavior can be achieved using operator overloading.

2.1.2. Scalar

* identity(): outputs the (additive) identity element in the scalar field.

* add(scalar: Scalar): implements field addition for the elements in the field.

* mul(scalar: Scalar), implements field multiplication.

* random(): outputs a random scalar field element.

* serialize(scalars: list[Scalar; N]): serializes a list of scalars and returns their canonical representation of fixed length Ns * N.

* deserialize(buffer), attempts to map a byte array buffer of size Ns * N into [Scalar; N], and fails if the input is not the valid canonical byte representation of an element of the group. This function can raise a DeserializeError if deserialization fails.

In this spec, instead of add we will use + with infix notation; instead of equal we will use ==, and instead of mul we will use *. A similar behavior can be achieved using operator overloading.

2.2. Proofs of preimage of a linear map

2.2.1. Core protocol

This defines the object SchnorrProof. The initialization function takes as input the statement, and pre-processes it.

2.2.2. Prover procedures

The prover of a sigma protocol is stateful and will send two messages, a "commitment" and a "response" message, described below.

2.2.2.1. Prover commitment

```
prover_commit(self, witness, rng)
```

Inputs:

- witness, an array of scalars
- rng, a random number generator

Outputs:

- A (private) prover state, holding the information of the interactive prover necessary for producing the protocol response
- A (public) commitment message, an element of the linear map image, that is, a vector of group elements.

Procedure:

```
1. nonces = [self.instance.Domain.random(rng) for _ in range(self.instance.linear_map.num_scalars)]
2. prover_state = self.ProverState(witness, nonces)
3. commitment = self.instance.linear_map(nonces)
4. return (prover_state, commitment)
```

2.2.2.2. Prover response

```
prover_response(self, prover_state, challenge)
```

Inputs:

- prover_state, the current state of the prover
- challenge, the verifier challenge scalar

Outputs:

- An array of scalar elements composing the response

Procedure:

```
1. witness, nonces = prover_state
2. return [nonces[i] + witness[i] * challenge for i in range(self.instance.linear_map.num_scalars)]
```

2.2.3. Verifier

verify(self, commitment, challenge, response)

Inputs:

- self, the current state of the SigmaProtocol
- commitment, the commitment generated by the prover
- challenge, the challenge generated by the verifier
- response, the response generated by the prover

Outputs:

- A boolean indicating whether the verification succeeded

Procedure:

```
1. assert len(commitment) == self.instance.linear_map.num_constraints and len(response) =
= self.instance.linear_map.num_scalars
2. expected = self.instance.linear_map(response)
3. got = [commitment[i] + self.instance.image[i] * challenge for i in range(self.instance
.linear_map.num_constraints)]
4. return got == expected
```

2.2.4. Witness representation

A witness is simply a list of num_scalars elements.

Witness = [Scalar; num_scalars]

2.2.5. Linear map

A LinearMap represents a function (a _linear map_ from the scalar field to the elliptic curve group) that, given as input an array of Scalar elements, outputs an array of Group elements. This can be represented as matrix-vector (scalar) product using group multi-scalar multiplication. However, since the matrix is oftentimes sparse, it is often more convenient to store the matrix in Yale sparse matrix format.

Here is an example:

```
class LinearCombination:
    scalar_indices: list[int]
    element_indices: list[int]
```

The linear map can then be presented as:

```
   class LinearMap:
       Group: groups.Group
       linear_combinations: list[LinearCombination]
       group_elements: list[Group]
       num_scalars: int
       num_elements: int

       def map(self, scalars: list[Group.ScalarField]) -> Group
```

2.2.5.1. Initialization

The linear map LinearMap is initialized with

```
linear_combinations = []
group_elements = []
num_scalars = 0
num_elements = 0
```

2.2.5.2. Linear map evaluation

A witness can be mapped to a group element via:

```
map(self, scalars: [Scalar; num_scalars])

Inputs:

- self, the current state of the constraint system
- witness,

1. image = []
2. for linear_combination in self.linear_combinations:
3.     coefficients = [scalars[i] for i in linear_combination.scalar_indices]
4.     elements = [self.group_elements[i] for i in linear_combination.element_indices]
5.     image.append(self.Group.msm(coefficients, elements))
6. return image
```

2.2.6. Statements for linear relations

The object LinearRelation has two attributes: a linear map linear_map, which will be defined in Section 2.2.5, and image, the linear map image of which the prover wants to show the pre-image of.

```
class LinearRelation: Domain = group.ScalarField Image = group.Group
```

```
       linear_map = LinearMap
       image = list[group.Group]

   def allocate_scalars(self, n: int) -> list[int]
   def allocate_elements(self, n: int) -> list[int]
   def append_equation(self, lhs: int, rhs: list[(int, int)]) -> None
   def set_elements(self, elements: list[(int, Group)]) -> None
```

2.2.6.1. Element and scalar variables allocation

Two functions allow to allocate the new scalars (the witness) and group elements (the instance).

```
allocate_scalars(self, n)

Inputs:
    - self, the current state of the LinearRelation
    - n, the number of scalars to allocate
Outputs:
    - indices, a list of integers each pointing to the new allocated scalars

Procedure:

1. indices = range(self.num_scalars, self.num_scalars + n)
2. self.num_scalars += n
3. return indices
```

and below the allocation of group elements

```
allocate_elements(self, n)

1. linear_combination = LinearMap.LinearCombination(scalar_indices=[x[0] for x in rhs], e
lement_indices=[x[1] for x in rhs])
2. self.linear_map.append(linear_combination)
3. self._image.append(lhs)
```

Group elements, being part of the instance, can later be set using the function set_elements

```
 set_elements(self, elements)

 Inputs:
     - self, the current state of the LinearRelation
     - elements, a list of pairs of indices and group elements to be set

 Procedure:

 1. for index, element in elements:
 2.   self.linear_map.group_elements[index] = element
```

2.2.6.2. Constraint enforcing

append_equation(self, lhs, rhs)

Inputs:

- self, the current state of the constraint system
- lhs, the left-hand side of the equation
- rhs, the right-hand side of the equation (a list of (ScalarIndex, GroupEltIndex) pairs)

Outputs:

- An Equation instance that enforces the desired relation

Procedure:

1. linear_combination = LinearMap.LinearCombination(scalar_indices=[x[0] for x in rhs], element_indices=[x[1] for x in rhs])
2. self.linear_map.append(linear_combination)
3. self._image.append(lhs)

2.2.7. Example: Schnorr proofs

The statement represented in Section 2 can be written as:

```
statement = LinearRelation(group)
[var_x] = statement.allocate_scalars(1)
[var_G, var_X] = statement.allocate_elements(2)
statement.append_equation(var_X, [(var_x, var_G)])
```

At which point it is possible to set var_G and var_X whenever the group elements are at disposal.

```
G = group.generator()
statement.set_elements([(var_G, G), (var_X, X)])
```

It is worth noting that in the above example, [X] == statement.linear_map.map([x]).

2.2.8. Example: DLEQ proofs

A DLEQ proof proves a statement:

```
DLEQ(G, H, X, Y) = PoK{(x): X = x * G, Y = x * H}
```

Given group elements G, H and X, Y such that x * G = X and x * H = Y, then the statement is generated as:

```
1. statement = LinearRelation()
2. [var_x] = statement.allocate_scalars(1)
3. statement.append_equation(X, [(var_x, G)])
4. statement.append_equation(Y, [(var_x, H)])
```

2.2.9. Example: Pedersen commitments

A representation proof proves a statement

```
REPR(G, H, C) = PoK{(x, r): C = x * G + r * H}
```

Given group elements G, H such that C = x * G + r * H, then the statement is generated as:

```
statement = LinearRelation()
var_x, var_r = statement.allocate_scalars(2)
statement.append_equation(C, [(var_x, G), (var_r, H)])
```

2.3. Ciphersuites

2.3.1. P-256 (secp256r1)

This ciphersuite uses P-256 [SP800] for the Group.

2.3.1.1. Elliptic curve group of P-256 (secp256r1) [SP800]

* order(): Return the integer 115792089210356248762697446949407573529996955224135760342422259061068512044369.

* serialize([A]): Implemented using the compressed Elliptic-Curve-Point-to-Octet-String method according to [SEC1]; Ne = 33.

* deserialize(buf): Implemented by attempting to read buf into chunks of 33-byte arrays and convert them using the compressed Octet-String-to-Elliptic-Curve-Point method according to [SEC1], and then performs partial public-key validation as defined in section 5.6.2.3.4 of [KEYAGREEMENT]. This includes checking that the coordinates of the resulting point are in the correct range, that the point is on the curve, and that the point is not the point at infinity.

2.3.1.2. Scalar Field of P-256

* serialize(s): Relies on the Field-Element-to-Octet-String conversion according to [SEC1]; Ns = 32.

* deserialize(buf): Reads the byte array buf in chunks of 32 bytes using Octet-String-to-Field-Element from [SEC1]. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1].

3. Security Considerations

Interactive sigma proofs are special sound and honest-verifier zero-knowledge. These proofs are deniable (without transferable message authenticity).

We focus on the security guarantees of the non-interactive Fiat-Shamir transformation, where they provide the following guarantees (in the random oracle model):

* *Knowledge soundness*: If the proof is valid, the prover must have knowledge of a secret witness satisfying the proof statement. This property ensures that valid proofs cannot be generated without possession of the corresponding witness.

* *Zero-knowledge*: The proof string produced by the prove function does not reveal any information beyond what can be directly inferred from the statement itself. This ensures that verifiers gain no knowledge about the witness.

While theoretical analysis demonstrates that both soundness and zero-knowledge properties are statistical in nature, practical security depends on the cryptographic strength of the underlying hash function, which is defined by the Fiat-Shamir transformation. It’s important to note that the soundness of a zero-knowledge proof provides no guarantees regarding the computational hardness of the relation being proven. An assessment of the specific hardness properties for relations proven using these protocols falls outside the scope of this document.

3.1. Privacy Considerations

Interactive sigma proofs are insecure against malicious verifiers and should not be used. The non-interactive Fiat-Shamir transformation leads to publicly verifiable (transferable) proofs that are statistically zero-knowledge.

4. Post-Quantum Security Considerations

The zero-knowledge proofs described in this document provide statistical zero-knowledge and statistical soundness properties when modeled in the random oracle model.

4.1. Privacy Considerations

These proofs offer zero-knowledge guarantees, meaning they do not leak any information about the prover's witness beyond what can be inferred from the proven statement itself. This property holds even against quantum adversaries with unbounded computational power.

Specifically, these proofs can be used to protect privacy against post-quantum adversaries, in applications demanding:

* Post-quantum anonymity
* Post-quantum unlinkability
* Post-quantum blindness
* Protection against "harvest now, decrypt later" attacks.

4.2. Soundness Considerations

While the proofs themselves offer privacy protections against quantum adversaries, the hardness of the relation being proven depends (at best) on the hardness of the discrete logarithm problem over the elliptic curves specified in Section 2.3. Since this problem is known to be efficiently solvable by quantum computers using Shor's algorithm, these proofs MUST NOT be relied upon for post-quantum soundness guarantees.

Implementations requiring post-quantum soundness SHOULD transition to alternative proof systems such as:

* MPC-in-the-Head approaches as described in [GiacomelliMO16]
* Lattice-based approaches as described in [AttemaCK21]
* Code-based approaches as described in [Stern93]

Implementations should consider the timeline for quantum computing advances when planning migration to post-quantum sound alternatives. Implementers MAY adopt a hybrid approach during migration to post-quantum security by using AND composition of proofs. This approach enables gradual migration while maintaining security against classical adversaries. This composition retains soundness if *both* problems remain hard. AND composition of proofs is NOT described in this specification, but examples may be found in the proof-of-concept implementation and in [BonehS23].

5. Generation of the protocol identifier

As of now, it is responsibility of the user to pick a unique protocol identifier that identifies the proof system. This will be expanded in future versions of this specification.

6. Generation of the instance identifier

As of now, it is responsibility of the user to pick a unique instance identifier that identifies the statement being proven.

7. References

7.1. Normative References

[KEYAGREEMENT]
Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", National Institute of Standards and Technology, DOI 10.6028/nist.sp.800-56ar3, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

7.2. Informative References

[AttemaCK21]
Attema, T., Cramer, R., and L. Kohl, "A Compressed Sigma-Protocol Theory for Lattices", <https://dl.acm.org/doi/10.1007/978-3-030-84245-1_19>.

[BonehS23] Boneh, D. and V. Schoup, "A Graduate Course in Applied Cryptography", n.d., <https://toc.cryptobook.us/>.

[fiat-shamir]
"draft-irtf-cfrg-fiat-shamir", <https://mmaker.github.io/spfs/draft-irtf-cfrg-fiat-shamir.html>.

[GiacomelliMO16]
Giacomelli, I., Madsen, J., and C. Orlandi, "ZKBoo: Faster Zero-Knowledge for Boolean Circuits", <https://eprint.iacr.org/2016/163.pdf>.

[SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", <https://www.secg.org/sec1-v2.pdf>.

[SP800] "Recommendations for Discrete Logarithm-based Cryptography", n.d., <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf>.

[Stern93] Stern, J., "A New Identification Scheme Based on Syndrome Decoding", 1993, <https://link.springer.com/chapter/10.1007/3-540-48329-2_2>.

Appendix A. Acknowledgments

The authors thank Jan Bobolz, Stephan Krenn, Mary Maller, Ivan Visconti, Yuwen Zhang for reviewing a previous edition of this specification.

Test Vectors

Test vectors will be made available in future versions of this specification. They are currently developed in the proof-of-concept implementation (https://github.com/mmaker/draft-zkproof-sigma-protocols/tree/main/poc/vectors).

Authors' Addresses

Michele Orr湛
CNRS
Email: m@orru.net

Cathie Yun
Apple, Inc.
Email: cathieyun@gmail.com